In this assignment, you will get used to working with the C language as well as working within the Linux/VirtualBox environment. The program will require you to use constants, variables, math expressions, random numbers, user I/O, IF statements, FOR loops and arrays.

(1) Chocolate Bar Orders

Assume that you have a corner store and need to order some chocolate bars. Write a program called **chocolateBars.c** that displays the following menu:

```
1. Oh Henry          $0.87 each
2. Coffee Crisp      $0.88 each
3. Aero              $0.50 each
4. Smarties          $1.29 each
5. Crunchie          $0.98 each
```

Your program should then ask the user for the # of boxes of each type of chocolate bar to be ordered (assume less than 1000 each time … you do not need to check for valid numbers ranges nor valid characters):

```
How many boxes of Oh Henry bars would you like (24 bars per box)?  5
How many boxes of Coffee Crisp bars would you like (36 bars per box)? 10
How many boxes of Aero bars would you like (48 bars per box)? 50
How many boxes of Smarties would you like (24 bars per box)? 100
How many boxes of Crunchie bars would you like (36 bars per box)? 500
```

Your program should then display the total cost for each type of chocolate bar and the sub total indicating the final cost before taxes. It should also then display the HST (13%) amount followed by the total amount due as follows:

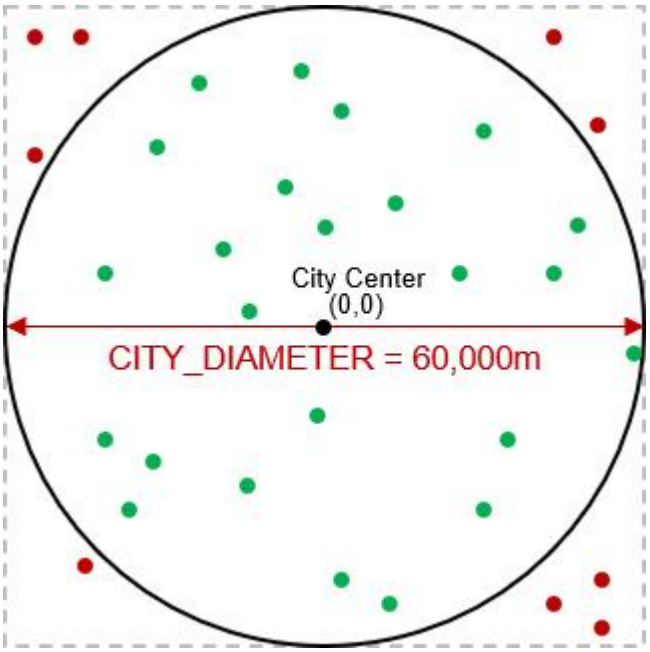
```
  5 boxes of Oh Henry      (24 x $0.87 per box) = $   104.40
 10 boxes of Coffee Crisp  (36 x $0.88 per box) = $   316.80
 50 boxes of Aero          (48 x $0.50 per box) = $  1200.00
100 boxes of Smarties      (24 x $1.29 per box) = $  3096.00
500 boxes of Crunchie      (36 x $0.98 per box) = $ 17640.00
-------------------------------------------------------------
Sub Total                                       = $ 22357.20
HST                                             = $  2906.44
=============================================================
Amount Due                                      = $ 25263.63
```

Notice how things are nicely aligned and formatted. Ensure that your code is also properly lined up. YOU MUST define the HST as well as all individual bar prices separately and bars per box separately as constants at the top of your code and use those throughout the program. You should not have any of these hard-coded in your calculations nor in your **printf()** statements. Test your program with each of the following test cases as well as the above test case:

Oh Henry	Coffee Crisp	Aero	Smarties	Crunchie	SubTotal	HST	Amount Due
1	1	1	1	1	$142.80	$ 18.56	$161.36
1	2	3	4	5	$456.48	$ 59.34	$515.82
0	0	0	0	0	$ 0.00	$ 0.00	$ 0.00

(2) Amazon Delivery

Amazon hires many delivery drivers to deliver their thousands of packages each day within a city. You will write a program to assign various package-delivery-locations to various drivers within the city's boundaries. The delivery area for the city will be defined by a circle with diameter of 60,000 meters centered at the city's center. Locations within the city's delivery area (i.e., green circles shown) will be allowed, but any locations outside of the city's delivery area (i.e., red circles shown) will not be considered.



The city center has location (**0**,**0**). Therefore, all valid delivery locations will have **x** and **y** coordinates each within the range of (**-30.0** km to **+30.0** km) … within the square bounding box. In addition, the delivery locations MUST also be within the city's radius. In the image here, the red circles have coordinates within the (**-30.0** km to **+30.0** km) but they are not within the city's radius, and are therefore invalid.

Write a program called **amazon1.c** that does the following:

a) The program should ask the user for the number of drivers, which must be a number from **2** to **200**, inclusively. If an invalid number is entered, it should display an error message and then ask the user again. This should continue until a valid number is entered. You may assume that only digit characters are entered by the user. Then, in the same manner, the program should ask the user for the number of packages to be delivered, which must be a number from **10** to **50000**, inclusively. Once both these numbers are valid, the program will continue.

b) The program should then compute a randomly-chosen (x, y) coordinate for each **driver** and store them in either one or two arrays (your choice). Each driver must have a coordinate that is within the city's circular area (e.g., like the green circles shown). The coordinates MUST be stored in Kilometers, so you will need them to be **floats**.

c) The program should then compute a randomly chosen (x, y) coordinate for each **package** and store them in either one or two arrays (your choice). Each package must have a coordinate that is within the city's circular area (e.g., like the green circles shown). The coordinates MUST be stored in Kilometers, so you will need them to be **floats**.

d) The program should then go through all the packages and determine the driver that is closest to it. You will need to use this formula to compute the distance between two points:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

For each package, you will need to remember which driver was closest by storing the index of the driver (you can decide how to do that). If, for example, you have **5** drivers, then the indices are **0**, **1**, **2**, **3** and **4**.

e) The program should then display the number of drivers, the coordinates of each driver, the number of packages and the coordinates of each package along with the driver index for that package. The coordinates must all be displayed aligned nicely with the decimals aligned. Here is an example of the output for the program (the red indicates what the user entered):

```
Enter number of delivery drivers: 1
 *** Number of delivery drivers must be between 2 and 200 ***
Enter number of delivery drivers: 6734
 *** Number of delivery drivers must be between 2 and 200 ***
Enter number of delivery drivers: 3
Enter number of packages to be delivered: 4
 *** Number of packages must be between 10 and 50000 ***
Enter number of packages to be delivered: 789234
 *** Number of packages must be between 10 and 50000 ***
Enter number of packages to be delivered: 10

Drivers:  3
 15.76   0.87
 18.97  16.14
-13.68  19.94

Packages: 10
 21.37  -18.43  0
  9.94  -14.75  0
 -5.67    4.20  2
-10.13   12.12  2
  6.60   13.68  1
 12.90    4.32  0
  9.83  -15.38  0
 24.23   -1.21  0
 15.70    4.24  0
 28.06    3.36  0
```

Make sure to test your program a few times. The values should be as random as possible, so make sure that you do not use the same random number sequence each time.

(3) A Visual Display

Now copy your code into a file called **amazon2.c**. Adjust the code so that it just waits for the values from the user without display the prompts with **printf()**. Also, instead of printing error messages when invalid numbers are entered, the program should simply exit. Now adjust the program so that it prints out exactly: the number of drivers (without a newline character), followed by a space character, followed by the number of packages and then followed by a newline character. It should then print out all the packages one at a time as follows: print the **x** value, print a **space** character, print the **y** value, print a **space** character, print the **driver index** then print a **space** character. You MUST NOT print any newline characters at all when you print out the package information.

Run the program. Assuming that you enter **3** drivers and **10** packages, the output would look like this (the red was typed in by the user … and t he numbers will differ):

```
3
10
3 10
12.65 -19.03 2 -11.82 22.99 0 -11.72 -15.07 2 -15.28 14.30 0 -22.73 -5.30 2 -13.80 -
18.51 2 -12.80 11.97 0 16.44 -23.42 2 16.00 -2.92 2 14.71 14.86 1
```

Notice how it does not ask the user for the number of drivers and packages … it simply allows the user to enter them one at a time. It then outputs these two values and a newline character and then all the coordinates and driver indices for the packages.

The reason for these changes is that you will now run your data through a program that will display the data. The program is called **display.c**. Download it and compile it as follows:

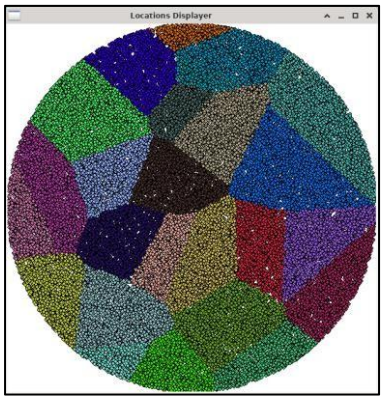
```
gcc -o display display.c -lX11.
```

The last part of that line is a "minus elle ex eleven" which is the standard C library that is used for windows and graphics. Do NOT alter the code in **display.c**.

To test your program, you will "pipe" the output (i.e., send the output) from the **amazon2** program into the **display** program. Just do the following in the shell window:

```
./amazon2 | ./display
```

The character in-between is a vertical bar character. On my keyboard it is above the ENTER key … but the location on your keyboard will likely differ.



When you run it, a blank window will appear. Click back onto the shell window and type in **25** followed by **50000**. You will need to grab the window and drag it to update it.

After a bit of a delay … you should see something that looks like this … although the colors and shapes will be different each time →

Try your program with different a different number of drivers and packages.

IMPORTANT SUBMISSION INSTRUCTIONS:

Submit all of your **.c source** files and all other files needed for testing/running your programs. Make sure that your name and student number is in each source file at the top as a comment.

The code **MUST** compile and run on the course VM.

- If your internet connection at home is down or does not work, we will not accept this as a reason for handing in an assignment late ... so make sure to submit the assignment WELL BEFORE it is due !

- You WILL lose marks on this assignment if any of your files are missing. So, make sure that you hand in the correct files and version of your assignment. You will also lose marks if your code is not **written neatly with proper indentation and containing a reasonable number of comments.** See course notes for examples of what is proper indentation, writing style and reasonable commenting).

COMP2401 - Assignment #2

(Due: **Wednesday, February 7th, 2024 @ 11pm**)

In this assignment, you will work with arrays of strings and pack data into bytes based on bits. Then you will unpack them.

Data compression is an important topic in computer science because as time goes on, more and more data becomes available and there is always a need to reduce the amount of storage required to save the data. In this assignment, you will implement a VERY SIMPLE strategy for compressing some basic text from 8 bits to 6 bits.

As you know, each byte (i.e., **unsigned char**) in C is represented by an 8-bit binary value associated with it. So, when we read in characters from the user, they are represented using ASCII code values. You will create a quick encoder that will reduce text by trimming 8-bit bytes down to a 6-bit representation. Since **6** bits only allows **64** values … we will encode only the letters, digits and a few other characters and all other characters will be represented by a single common character. Here is how we will map the characters to a 6-bit value:

CHARACTER	8-BIT ASCII VALUE	6-BIT VALUE
NULL	0	0
A-I	65-73	1-9
a-i	97-105	1-9
ENTER (i.e., CR)	10	10
J-Z	74-90	11-27
j-z	106-122	11-27
[	91	28
]	93	29
@	64	30
various symbols	32-63	32-63
all other characters	64-255	31

(1) Download the six files provided with this assignment (i.e., **convertTo6bit.c**, **usefulTools.c**, **usefulToos.h**, **compress.c**, **expand.c** and **convertToASCII.c**)

The program called **convertTo6bit.c** has been started for you. It reads the command-line arguments to the program (more on this later in the course) to decide if the debugging information should be shown. It then asks the user for a string of at most 255 characters. **You will insert code to convert the 8-bit ASCII characters from that string into a 6-bit character string by using the mapping shown in the above table**. The program then displays debugging information (if enabled) and finally sends the 6-bit encoded sentence to the console output. The code will make use of some functions provided in the **usefulTools.c** file … so you compile the program as follows:

```
gcc -o convertTo6bit convertTo6bit.c usefulTools.c -lm
```

Then run the program as follows:

```
./convertTo6bit -d
```

The program should display the 6-bit codes for each entered character and also display the string result (which will contain some characters that cannot be displayed). Here is some sample output. Make sure that these cases all work fine (also make sure that a **0** is always at the end of the output string):

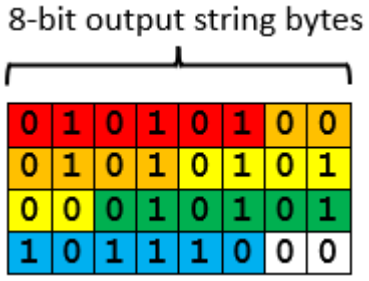
User Input	Displayed Codes	Visible Part of Converted String
`Aa 19 @.`	`01 01 32 49 57 32 30 46 00`	` 19 .`
`A0A0A0A0`	`01 48 01 48 01 48 01 48 00`	`0000`
`I Love This Assignment!`	`09 32 13 16 23 05 32 21 08 09 20 32 01 20 20 09 07 15 14 05 15 21 33 00`	` !`
`[abyz] ABYZ.@0189%~+=;`	`28 01 02 26 27 29 32 01 02 26 27 46 30 48 49 56 57 37 31 43 61 59 00`	``189%+=;`
`.`	`46 00`	`.`

Run the program without the `-d` option (i.e., `./convertTo6bit`). You should not see the byte codes but just the final converted string.

(2) The program called **compress.c** has also been started for you. After reading the command-line arguments it then reads in a string of converted characters from the **convertTo6bit** program. **You will insert code to pack the 6-bit character codes from the input string into 8-bit bytes (stored as an output string) so that the storage space required is reduced**. The program then displays debugging information (if enabled) and finally sends the compressed bytes to the console output as a string.

Consider the input string `"Test."`. The conversion of his string results in the 6-bit codes shown here on the right → along with their binary representation as stored in each of the 5 bytes.

To obtain the compressed output string bytes, you will need to take the 6 bits from each of the 5 input string bytes and pack them into the 8-bit bytes of the output string. To do this, just copy the 6 bits from the first character into the first byte of the output string. There will be two bits remaining in the output string. Copy the first two bits from the next input string character into the output string to complete the first output string byte. Then begin the second output string byte by copying in the remaining 4 bits from the second input string byte. Keep copying bits in this manner … filling up each output string byte and then moving on to the next one. As a result, the output string will have just 4 bytes with all the bits used except for the last two bits, which should be set to 0. Here is the result on the right →
This is an 80% compression ratio (i.e., the output string is 80% of the size of the original string).



You will need to keep track of how many bytes you are writing to the output string. You will not be able to use **strlen()** to determine the size of the output string because multiple bits could pack into a byte as all **0**'s, which will appear as a NULL-terminator to **strlen()** … causing it to return a smaller value that the string's size. Here is an example that shows how this can happen when a simple string such as `"A0A"` is entered:

You will compile the program as follows:

```
gcc -o compress compress.c usefulTools.c -lm
```

Then run the two programs together as follows: `./convertTo6bit | ./compress -d`

This will run the **convertTo6bit** program first and the **|** character tells the shell window to use the output from that program as input to the **compress** program. That will allow us to type in a string of characters and see the resulting bits that get packed. Here is some sample output. Make sure that these cases all work fine:

User Input	Displayed Output		Visible Part of Converted String
Aa 19 @.	Compression ratio = 75.0% Before compression: 000001 000001 100000 110001 111001 100000 011110 101110	 After compression: 00000100 00011000 00110001 11100110 00000111 10101110	1 pp &P\X E =X@
A0A0A0A0	Compression ratio = 75.0% Before compression: 000001 110000 000001 110000 000001 110000 000001 110000	 After compression: 00000111 00000000 01110000 00000111 00000000 01110000	
I Love This Assignment!	Compression ratio = 78.3% Before compression: 001001 100000 001101 010000 010111 000101 100000 010101 001000 001001 010100 100000 000001 010100 010100 001001 000111 001111 001110 000101 001111 010101 100001	 After compression: 00100110 00000011 01010000 01011100 01011000 00010101 00100000 10010101 00100000 00000101 01000101 00001001 00011100 11110011 10000101 00111101 01011000 01000000	

[abyz] ABYZ.@0189%~+=;	Compression ratio = 77.3% Before compression: 011100 000001 000010 011010 011011 011101 100000 000001 000010 011010 011011 101110 011110 110000 110001 111000 111001 100101 011111 101011 111101 111011	 After compression: 01110000 00010000 10011010 01101101 11011000 00000001 00001001 10100110 11101110 01111011 00001100 01111000 11100110 01010111 11101011 11110111 10110000	pm { xW
.	Compression ratio = 100.0% Before compression: 101110	 After compression: 10111000	

(3) The program called **expand.c** has also been started for you. It reads in an 8-bit compressed string from the **compress** program. You will insert code to expand the compressed bytes back into the 6-bit character codes that were originally output from the **convertTo6bit** program (i.e., you will do the reverse of the **compress** program). The program then displays debugging information (if enabled) and finally sends the expanded bytes to the console output as a string. This time, you will be able to se the **strlen()** function to determine the number of bytes in the output string. You will compile the program as follows:

```
gcc -o expand expand.c usefulTools.c -lm
```

Then run the three programs together as follows:

```
./convertTo6bit | ./compress | ./expand -d
```

Here is some sample output. Make sure that these cases all work fine:

User Input	Displayed Output		Visible Part of Converted String
Aa 19 @.	Expansion ratio = 133.3% Before expansion: 00000100 00011000 00110001 11100110 00000111 10101110	 After expansion: 000001 000001 100000 110001 111001 100000 011110 101110	19 .
A0A0A0A0	Expansion ratio = 133.3% Before expansion: 00000111 00000000 01110000 00000111 00000000	 After expansion: 000001 110000 000001 110000 000001	0000

	01110000	110000 000001 110000	
I Love This Assignment!	Expansion ratio = 127.8% Before expansion: 00100110 00000011 01010000 01011100 01011000 00010101 00100000 10010101 00100000 00000101 01000101 00001001 00011100 11110011 10000101 00111101 01011000 01000000	 After expansion: 001001 100000 001101 010000 010111 000101 100000 010101 001000 001001 010100 100000 000001 010100 010100 001001 000111 001111 001110 000101 001111 010101 100001	!
[abyz] ABYZ.@0189%~+=;	Expansion ratio = 129.4% Before expansion: 01110000 00010000 10011010 01101101 11011000 00000001 00001001 10100110 11101110 01111011 00001100 01111000 11100110 01010111 11101011 11110111 10110000	 After expansion: 011100 000001 000010 011010 011011 011101 100000 000001 000010 011010 011011 101110 011110 110000 110001 111000 111001 100101 011111 101011 111101 111011	��189%+=;
.	Expansion ratio = 100.0% Before expansion: 10111000	 After expansion: 101110	.

(4) Finally, the program called **convertToASCII.c** has also been started for you. It reads in the 6-bit code expanded string from the **expand** program. You will insert code to convert these 6-bit character codes back into ASCII codes (i.e., reverse of **convertTo6bit** program). The output string size will have the same as the input string. All 6-bit values of 31 represent all ASCII characters that were not able to be represented in 6-bits and when converted back to ASCII, they should be converted to the '_' (i.e., underscore) character. You will compile the program as follows:

```
gcc -o convertToASCII convertToASCII.c usefulTools.c -lm
```

Then run the four programs together as follows:

```
./convertTo6bit | ./compress | ./expand | ./convertToASCII -d
```

Here is some sample output. Make sure that these cases all work fine (also make sure that a **0** is always at the end of the output string):

User Input	Displayed Codes	Visible Part of Converted String
`Aa 19 @.`	`65 65 32 49 57 32 64 46 00`	`AA 19 @.`
`A0A0A0A0`	`65 48 65 48 65 48 65 48 00`	`A0A0A0A0`
`I Love This Assignment!`	`73 32 76 79 86 69 32 84 72 73 83 32 65 83 83 73 71 78 77 69 78 84 33 00`	`I LOVE THIS ASSIGNMENT!`
`[abyz] ABYZ.@0189%~+=;`	`91 65 66 89 90 93 32 65 66 89 90 46 64 48 49 56 57 37 95 43 61 59 00`	`[ABYZ] ABYZ.@0189%_+=;`
`.`	`46 00`	`.`

Run the program without the `-d` option. You should not see the byte codes but just the final converted string.

Now that all the programs are working … here are 5 user strings that you can try to run through all 4 programs. You can cut/paste each one:

```
Q: Did you hear about the racing snail who got rid of his shell? A: He thought it would make him faster, but it just made him sluggish.

Q: How does a mathematician induce good behavior in her children? A: `I've told you n times, I've told you n+1 times...'

Have you heard of that new band "1023 Megabytes"? They're pretty good, but they don't have a gig just yet.

There are only 10 kinds of people in this world: those who know binary and those who don't.

"Knock, knock."  "Who's there?"   very long pause….  "Java."
```

IMPORTANT SUBMISSION INSTRUCTIONS:

Submit all of your **.c source** files and all other files needed for testing/running your programs. DO NOT TAR your files. Make sure that your name and student number is in each source file at the top as a comment.

The code **MUST** compile and run on the course VM.

- If your internet connection at home is down or does not work, we will not accept this as a reason for handing in an assignment late ... so make sure to submit the assignment WELL BEFORE it is due !

- You WILL lose marks on this assignment if any of your files are missing. So, make sure that you hand in the correct files and version of your assignment. You will also lose marks if your code is not **written neatly with proper indentation and containing a reasonable number of comments.** See course notes for examples of what is proper indentation, writing style and reasonable commenting).

COMP2401 - Assignment #3

(Due: **Wednesday, February 28th @ 11pm**)

In this assignment, you will gain experience in using **structs** and **pointers**. Make sure to put comments in your code … especially ones to describe your functions, definitions and important parts of your code. You must also write your code as cleanly & efficiently as possible.

Assume that a private cell phone company wanted you to write a program to keep track of various calls made between its customers. You will use various structures to represent the type of phone plan, the customer, the calls made and the phone network itself. A header file called **phoneNetwork.h** has been provided with the necessary **struct** definitions that you will need as shown below … with type name definitions shown in red:

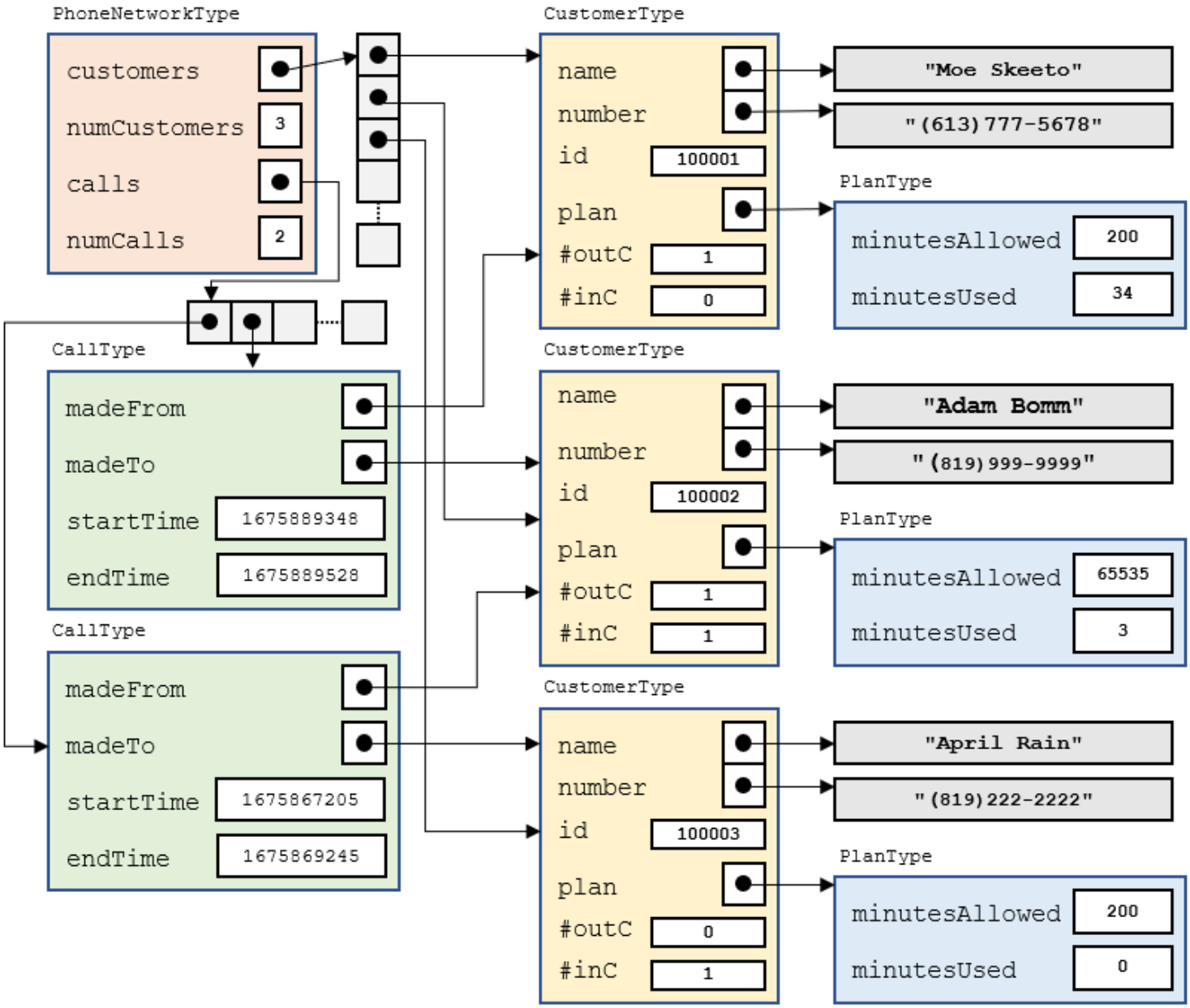
```
// Structure that stores a phone plan that a customer has
typedef struct {
  unsigned short int   minutesAllowed;  // # mins/month allowed in phone plan
  unsigned short int   minutesUsed;     // # mins used this month so far
} PlanType;


// Structure that stores a customer
typedef struct {
  char           *name;         // name of customer
  unsigned int    id;           // customer ID
  char           *number;       // phone number of customer
  PlanType        plan;         // phone plan of customer
  unsigned short  numOutCalls;  // number of calls made this month
  unsigned short  numInCalls;   // number of calls received this month
} CustomerType;


// Structure that stores a phone call that was made
typedef struct {
  CustomerType   madeFrom;    // customer who made call
  CustomerType   madeTo;      // customer who was called
  time_t         startTime;   // time that call was started
  time_t         endTime;     // time that call ended
} CallType;

// Structure that stores the phone network's customer and call data
typedef struct {
  CustomerType customers[MAX_CUSTOMERS];  // data for customers registered with network
  unsigned int numCustomers;              // number of customers registered
  CallType     calls[MAX_CALLS];          // data for all calls that took place this month
  unsigned int numCalls;                  // number of calls that took place this month
} PhoneNetworkType;
```

The next page shows an example of a phone network with **3** customers in which **2** phone calls were made. Make sure that you understand how the image matches up with the structures in the header file before you continue the assignment.



Download the **phoneSimulation.c** file. It contains a **main()** function with comments that indicate the steps you should follow. It also contains sample names, numbers and phone plan values that you will use. You will be writing a bunch of functions and procedures.

You must create a file called **phoneNetwork.c** that will contain ALL the functions and procedures that you will be asked to write.DO NOT write them in the **phoneSimulation.c** file.

You will need to include that additional file at the end of your **gcc** shell command so that it gets compiled along with the **phoneSimulation.c** file.

(1) Write a **registerCustomer()** procedure that will register a customer in the phone network defined by the **phoneNetwork** variable that is provided. The procedure must have the following signature which takes a pointer to a phone network, a name, a phone number and a number of allowed phone plan minutes/month:

```
registerCustomer(PhoneNetworkType *, char *, char *, int)
```

For each customer you should have some kind of global counter so that each time you register a customer they get assigned a unique id (starting with 100001). It must return **1** if the customer was able to be added (i.e., not past capacity) and **0** otherwise.
Call this procedure from the **main()** function so that it creates exactly **20** customers … with the names, phone numbers and the allowed number of phone plan minutes provided in the Sample arrays.

(2) Write a **displayCharges()** procedure that takes a pointer to a **PhoneNetworkType** and then displays all customers in the phone network provided … showing their id, phone number, name and phone plan minutes allowed. It should show "UN" for unlimited-minute phone plans and "PPM" for phone plans that require paying per minute of usage. Call this procedure at the end of the **main()** function as shown by the comment in the code. You will add more functionality to this procedure later. Make sure to test your code … the output should be aligned nicely and approximately as follows:

```
ID     Phone Number     Name                Plan
-------------------------------------------------
100001 (613)111-1111    Rob Banks             UN
100002 (819)222-2222    April Rain           200
100003 (613)333-3333    Rita Book            PPM
100004 (343)444-4444    Sue Permann           UN
100005 (819)555-5555    Tim Bur              PPM
100006 (613)666-6666    Paddy O'Lantern       UN
100007 (343)777-7777    Sam Pull             200
100008 (613)888-8888    Sandy Beach           UN
100009 (819)999-9999    Adam Bomm             UN
100010 (613)555-1234    Hugo First            UN
100011 (613)555-5678    Dan Druff            200
100012 (613)666-1234    Mabel Syrup           UN
100013 (613)666-5678    Mike Rohsopht         UN
100014 (343)777-1234    Adam Sapple          100
100015 (613)777-5678    Moe Skeeto           200
100016 (819)888-1234    Anita Bath            UN
100017 (343)888-5678    Rusty Chain          PPM
100018 (613)999-1234    Stu Pitt             200
100019 (613)999-5678    Val Crow              UN
100020 (613)444-1234    Neil Down             UN
```

(3) Write a **resetMonth()** procedure that simulates the resetting of phone call data for a new month of calls. It must take a pointer to a **PhoneNetworkType**. The procedure should reset everything as if there have been no calls made in the network so that things begin from scratch. Call this procedure in your **main()** function right after creating the customers.

(4) Write a **recordCall()** function that simulates a call that has been made in the network. It must have the following signature, taking a pointer to the network as its 1st parameter:

```
char recordCall(PhoneNetworkType *, char *, char *, time_t, time_t);
```

The 2nd parameter is the phone number that the call was made from, the 3rd parameter is the phone number that the call was made to, the 4th parameter is the time that the call was started and the last parameter is the time that the call ended. The time is designed as a **time_t** type. This is a large integer number that represents the number of seconds that has elapsed since 00:00:00 UTC, January 1, 1970. It is the typical standard that is used to record time-of-day in computer science.

The function should return **0** if the call could not be recorded (this could happen if the number of calls that were made in the phone network has reached its capacity or if either of the phone numbers are invalid). Otherwise, the function should return 1. To record the call, you will need to do a few things. You will need to:

(a) Determine how many minutes the call lasted (rounded up to the nearest minute)

(b) Find the customer that made the call and record that number of minutes used as well as the fact that this customer made this outgoing phone call.

(c) Find the customer that received the call and record the fact that this customer received this incoming phone call. If it was a PAY_PER_MINUTE customer that received the call, the number of minutes used should be reduced as a result of this phone call, but other types of phone plans do not need to have the minutes reduced for incoming calls.

(d) Update the appropriate from/to/start/end information in the network's **calls** array accordingly.

(5) To test the **recordCall()** function, after resetting the month in the **main()** function, you should write code to make **10** calls (we will increase this to **100** later). To do this, choose a random customer to make the call and a random one to receive the call (make sure that it is not the same customer that makes and receives the call at the same time as this is invalid). You will need to choose a random start time for the call and a random end time. To do this, we will need to generate realistic **time_t** values. There is a **`struct`** `tm` structure defined in `<time.h>` that allows us to specify an exact year, month, day, hour, minute and second time. Here is what the structure looks like:

```
struct tm {
  int tm_sec;         // seconds 0-59
  int tm_min;         // minutes 0-59
  int tm_hour;        // hours 0-23
  int tm_mday;        // day of the month 1-31
  int tm_mon;         // month 0-11
  int tm_year;        // year since 1900
  int tm_wday;        // day of the week 0-6
  int tm_yday;        // day in the year 0-365
  int tm_isdst;       // daylight saving time (we will use -1 to disable)
};
```

You should generate a start time for the phone call by making a variable of type **struct tm** and then setting its values so that the call was made in **Feb 2024**. The day of the month should be chosen randomly as a number from **1-29** (since 2024 is a leap year). The hour, minute and second of the day should all be chosen randomly.

To get your **struct tm** variable into a **time_t** format that you need for the **recordCall()** function, you simply call the **mktime()** function from `<time.h>` with the address of your struct tm variable. That function will return the value you need for your call's start time. As for the call's end time, you can take that start time value and simply add a random number of seconds so that the call lasts at most two hours (i.e., at most **120** mins *** 60** seconds). At this point, you will not know if your code works properly until you write the next procedure.

(6) Write a **displayCallLog()** procedure that displays the phone call data. It must take a <u>pointer</u> to a **PhoneNetworkType**. For each call made, the procedure should display the phone number that the call was made from, the number it was made to, the start date/time of the call, the end date/time of the call and the length of time (in minutes with one decimal place) that the call lasted. The output should be formatted neatly and look roughly as follows:

```
Call From #     Call To #       Start Time:           End Time              Call Length
-------------------------------------------------------------------------------------
(613)555-1234   (613)333-3333   2024-02-07 12:57:50   2024-02-07 13:19:32    21.7 minutes
(613)999-1234   (613)333-3333   2024-02-25 05:34:07   2024-02-25 07:26:46   112.7 minutes
(613)444-1234   (819)999-9999   2024-02-15 08:20:22   2024-02-15 09:59:54    99.5 minutes
(613)555-1234   (343)444-4444   2024-02-16 09:12:20   2024-02-16 09:52:32    40.2 minutes
(613)555-1234   (819)222-2222   2024-02-16 03:13:00   2024-02-16 03:24:41    11.7 minutes
(343)888-5678   (613)999-5678   2024-02-12 18:55:09   2024-02-12 20:35:15   100.1 minutes
(343)777-7777   (613)111-1111   2024-02-21 02:23:53   2024-02-21 03:40:39    76.8 minutes
(343)777-7777   (613)999-5678   2024-02-28 13:33:57   2024-02-28 14:51:59    78.0 minutes
(343)777-1234   (613)555-1234   2024-02-12 18:23:27   2024-02-12 18:54:04    30.6 minutes
(343)888-5678   (613)444-1234   2024-02-06 12:49:37   2024-02-06 12:50:00     0.4 minutes
```

Note: To extract the date/time information from the **time_t** values, you will need to get the **time_t** value into a struct tm variable so that you can extract what you need. To convert a **time_t** value called **timeValue** into a `struct tm` variable called **daytime**, you can call the **localtime()** function as follows: `struct tm daytime = *localtime(&timeValue);`

Don't forget to adjust the year and month values accordingly after the conversion.

(7) Finally, we will adjust the **displayCharges()** procedure so that it computes and displays the appropriate monthly charges for each customer. Here is how the charges work:

(a) For customers with UNLIMTED minute phone plans, the base cost is **$34.99** per month (before taxes).

(b) For customers with phone plans that have a specified number of minutes, the base cost is **$9.99** per month. But there is an extra **15** cents charge for every minute that the customer went over the plan's allowed minutes. So, a customer that used up **126** minutes on a 100-minute phone plan would pay **$9.99** + (**$0.15** * **26**) = **$13.89** before taxes.

(c) For customers with a PAY_PER_MINUTE phone plan, the customer simply pays **$0.15** per minute for the first **100** minutes and then they pay **$0.35** for every minute over the **100**.

Adjust your **displayCharges()** procedure so that it also displays (alongside the current customer information) the number of outgoing calls made, the number of incoming calls received, the number of minutes used, the number of minutes gone over, the base cost, the extra cos (for going over), the HST amount and the final total. The information must be aligned and displayed neatly. Here is an example of the output:

Phone Number	Name	Plan	Out	In	Used	Over	Base	Extra	HST	Total
(613)111-1111	Rob Banks	UN	3	5	234	0	34.99	0.00	4.55	$ 39.54
(819)222-2222	April Rain	200	6	3	303	103	9.99	15.45	3.31	$ 28.75
(613)333-3333	Rita Book	PPM	2	7	563	463	15.00	162.05	23.02	$ 200.07
(343)444-4444	Sue Permann	UN	6	9	382	0	34.99	0.00	4.55	$ 39.54
(819)555-5555	Tim Bur	PPM	4	4	367	267	15.00	93.45	14.10	$ 122.55
(613)666-6666	Paddy O'Lantern	UN	8	5	729	0	34.99	0.00	4.55	$ 39.54
(343)777-7777	Sam Pull	200	4	6	246	46	9.99	6.90	2.20	$ 19.09
(613)888-8888	Sandy Beach	UN	6	7	329	0	34.99	0.00	4.55	$ 39.54
(819)999-9999	Adam Bomm	UN	4	3	280	0	34.99	0.00	4.55	$ 39.54
(613)555-1234	Hugo First	UN	4	3	295	0	34.99	0.00	4.55	$ 39.54
(613)555-5678	Dan Druff	200	4	6	170	0	9.99	0.00	1.30	$ 11.29
(613)666-1234	Mabel Syrup	UN	3	10	165	0	34.99	0.00	4.55	$ 39.54
(613)666-5678	Mike Rohsopht	UN	5	2	343	0	34.99	0.00	4.55	$ 39.54
(343)777-1234	Adam Sapple	100	5	2	327	227	9.99	34.05	5.73	$ 49.77
(613)777-5678	Moe Skeeto	200	5	6	341	141	9.99	21.15	4.05	$ 35.19
(819)888-1234	Anita Bath	UN	10	4	752	0	34.99	0.00	4.55	$ 39.54
(343)888-5678	Rusty Chain	PPM	5	7	757	657	15.00	229.95	31.84	$ 276.79
(613)999-1234	Stu Pitt	200	4	5	285	85	9.99	12.75	2.96	$ 25.70
(613)999-5678	Val Crow	UN	5	2	209	0	34.99	0.00	4.55	$ 39.54
(613)444-1234	Neil Down	UN	7	4	488	0	34.99	0.00	4.55	$ 39.54

You MUST ensure that you are not using "magic numbers" in your code. You should add appropriate definitions to **phoneNetwork.h** for all cost amounts minute-limit values and tax amounts.

Make sure that your code compiles, runs and produces the correct output. Hand this test code in with your assignment.

IMPORTANT SUBMISSION INSTRUCTIONS:

Submit all your **.c** and **.h** source code files as individual files (i.e., DO NOT **TAR/RAR/ZIP/COMPRESS** anything):

1. A **Readme** text file containing
   - your name and studentNumber
   - a list of source files submitted

- any specific instructions for compiling and/or running your code

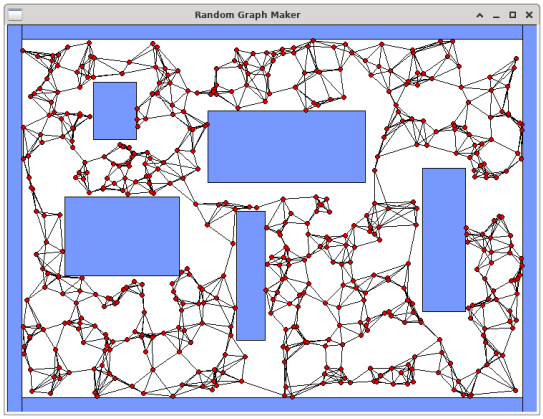
2. All of your **.c source** files and all other files needed for testing/running your programs.

The code **MUST** compile and run on the course VM. You WILL lose marks on this assignment if any of your files are missing. So, make sure that you hand in the correct files and version of your assignment. You will also lose marks if your code is not **written neatly with proper indentation and containing a reasonable number of comments.**

COMP2401 - Assignment #4

(Due: **Wednesday, March 13th, 2024 @ 11pm**)

In this assignment, you will gain additional practice using structs, pointers and dynamic memory allocation while creating a graph that makes use of Linked-Lists.



In the world of robotics, it is ALWAYS a good idea to program a robot to move around in its environment without hitting things. If a map of the environment is given, a robot can plan paths around the obstacles. There are many ways to perform path planning and some of the solutions involve creating a graph that can be traversed. In this assignment, we will create a random graph with vertices connected to their **k**-nearest neighbours, for some value of **k**. We will run our code on five fixed environments that are composed of rectangular obstacles. These environments will be given to you. The bottom left of the environments is always coordinate (0, 0).

To begin the assignment, you should download the following files:

- **display.c** and **display.h** – code for displaying the environment and graph.
- **obstacle.h** – structures required for this assignment.
- **graphMaker.c** – your code will be mostly written here (but there is no **main** function).
- **graphTester.c** – this is the test program that you will run.

You MUST not alter the **display.c**, **display.h** nor **obstacle.h** files. You are also NOT allowed to alter the **structs** defined in the **obstacle.h** file, NOR are you allowed to create any additional structs on this assignment. You MUST NOT CREATE ANY STATIC ARRAYS on this assignment.

***** ALERT ***** You must write your code in a modular/logical way by making helper functions/procedures for various parts of the code so that your **createGraph()** procedure (see parts 2 to 4) is **not more than 50 lines long** including comments and blank lines.

(1) You MUST first create a proper **makefile** that defines the proper dependencies, compiles the files and creates an executable called **graphTester**. You will need to include the **-lX11** library (that's "minus L X eleven") when creating the executable file. The **make clean** command should also work properly. The program must be run by supplying 3 command-line parameters as follows:

```
./graphTester <V> <K> <E>
```

Here, **<V>** is a number from 20 to 2000 (inclusively) which represents the number of vertices to add to the graph. **<K>** is a number between 1 and 25 (inclusively) that indicates the number of nearest-neighbours that each vertex will attempt to be connected to in the graph. **<E>** is a number from 1 to 5 (inclusively) representing the environment number to use. If your make file works properly, you should be able to run the program by supplying any numbers at this time, as long as they are within the proper range. The program should display the environment that you chose in a graphical window … and then wait for you to close the window.

(2) Your task will be to write all your code in the **graphMaker.c** file. Currently, there are just two blank procedures there and there is also a helper function called **linesIntersect()** that you will use. Your first task will be to write code in the **createGraph()** procedure so that it creates the vertices of the graph.

Look at the **Environment** typedef in the **obstacles.h** file. It contains a **numVertices** attribute which has been set from the command line and indicates how many vertices you need to create. The **vertices** attribute will point to a dynamically-allocated array of vertex pointers. You will need to allocate the array and set that **vertices** attribute to point to it. Then you need to create the **numVertices** vertices by allocating each vertex dynamically as well.

Look at the **Vertex** typedef in the **obstacles.h** file. Each vertex has an **x** and **y** coordinate as well as a pointer to a list of **neighbours**.

For each vertex, you must choose a random **x** value within the range of **0** to **maximumX** and a **y** value between **0** and **maximumY**. These maximums are attributes of the Environment structure. You must also ensure that the chosen (**x**, **y**) values are not inside (nor on the boundary of) any of the environment's obstacles. Notice that the environment has an **obstacles** array attribute. If you look at the **graphTester.c** file, you will notice that this array has already been created for you according to the environment selected via command-line arguments. Each **Obstacle** is assumed to be a rectangle with its top-left corner being an (**x**, **y**) point and having width **w** and height **h**.

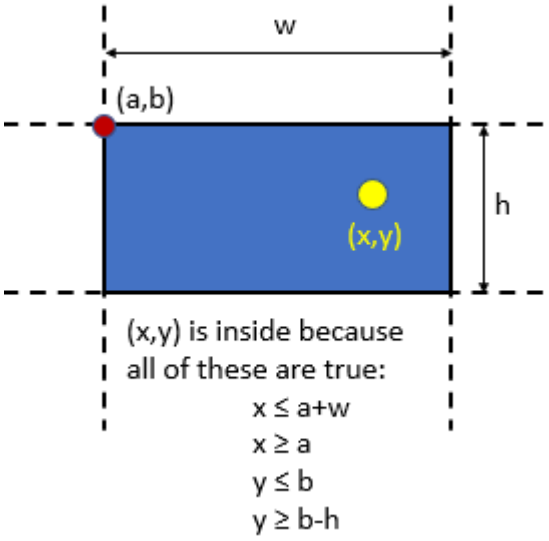
The diagram here shows how to tell whether or not a point **(x, y)** is inside the rectangle defined with **(a, b)** being the top-left corner.

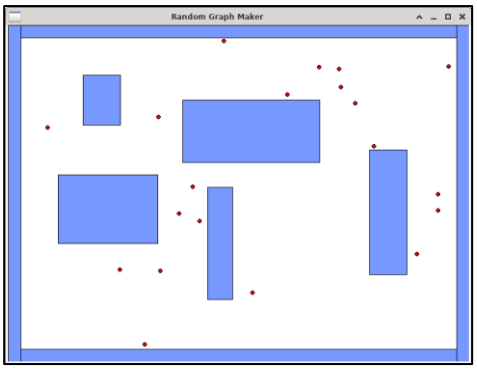
Once you have this completed, run your program as follows: `./graphTester 20 1 1` You should see something similar to what is shown here, except that your vertices will be in different locations. You should be able to count **20** red circles every time and no circle centers should be inside of a blue obstacle. It is possible, however, that a portion of the circle will be inside of an obstacle, but the very center pixel of the circle should NEVER be inside of an obstacle. Run again using `./graphTester 2000 1 1` and ensure that no circle centers are inside any obstacles. You may have to click on the window and move it a bit to ensure that the window displays all the vertices.

```
typedef struct {
  unsigned short  k;
  unsigned short  maximumX;
  unsigned short  maximumY;
  Obstacle        *obstacles;
  unsigned short  numObstacles;
  Vertex          **vertices;
  unsigned short  numVertices;
} Environment;


typedef struct vert {
  short           x;
  short           y;
  struct neigh  *neighbours;
} Vertex;


typedef struct obst {
  short       x;
  short       y;
  short       w;
  short       h;
} Obstacle;
```

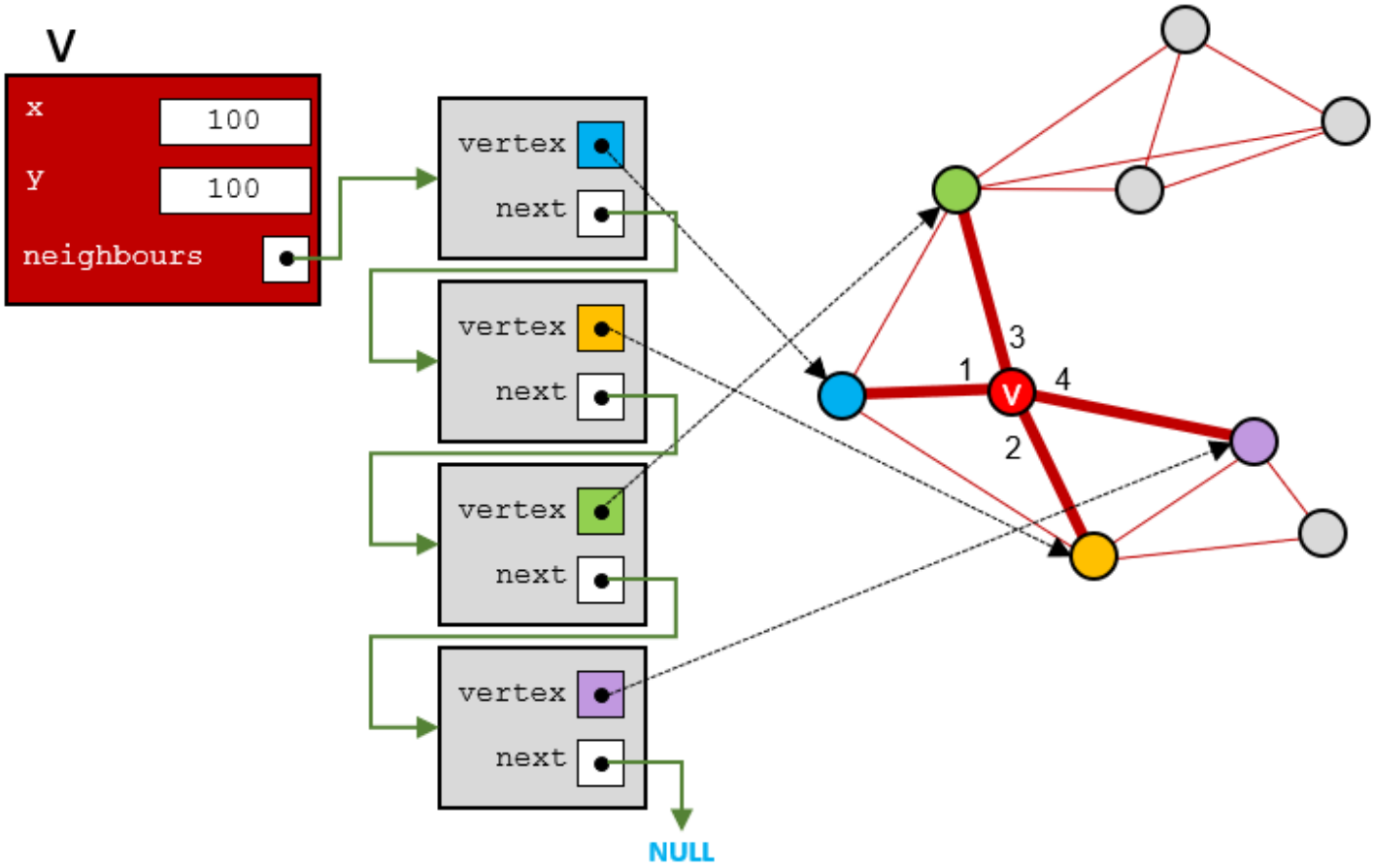




(3) Now it is time to create the graph edges. There will be no array to store the edges. Instead, each vertex will store a linked-list of the vertices that it is connected to (i.e., that vertices *neighbours*). Each vertex has an attribute called **neighbours** that should point to the head of a singly-linked-list of **Neighbour** types. The **Neighbour** type keeps a pointer to the **vertex** it represents as well as a pointer to the **next** neighbour in the list.

```
typedef struct neigh {
  Vertex        *vertex;
  struct neigh  *next;
} Neighbour;
```

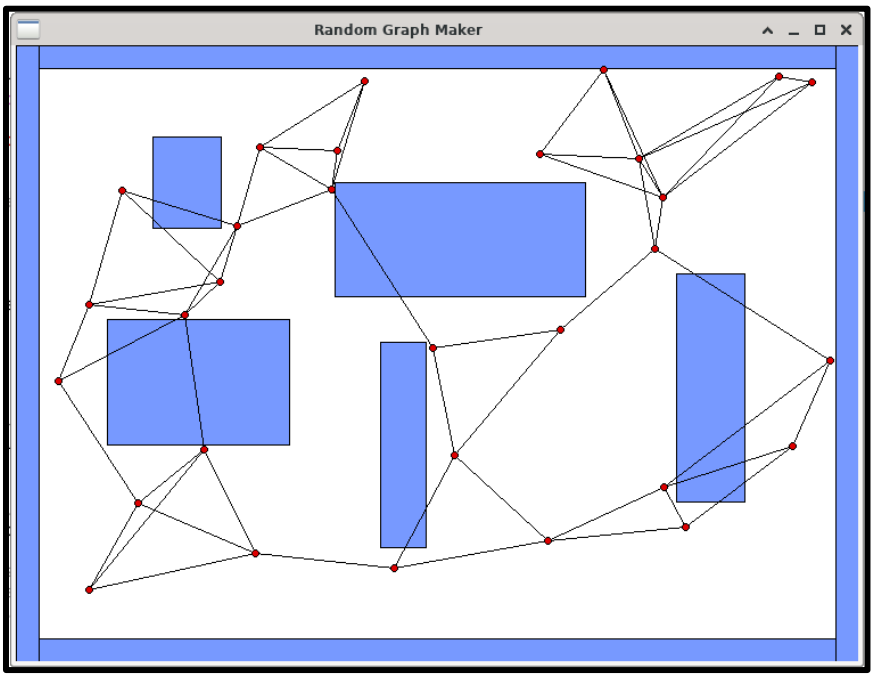
For each vertex **v**, you will go through all other vertices and find the **k**-closest vertices to **v**. The **k** value is chosen from the command-line-arguments and has been stored in the Environment struct. Consider vertex **v** in the graph below on the right. On the left, it shows the **Vertex** struct that represents it. The neighbours attribute points to the head of the linked-list that stores the **k**-nearest neighbours of **v**, where **k** = **4**. Although the order of the neighbours does not matter, it is shown such that the closest one to **v** (i.e., the blue one) is the head of the list.



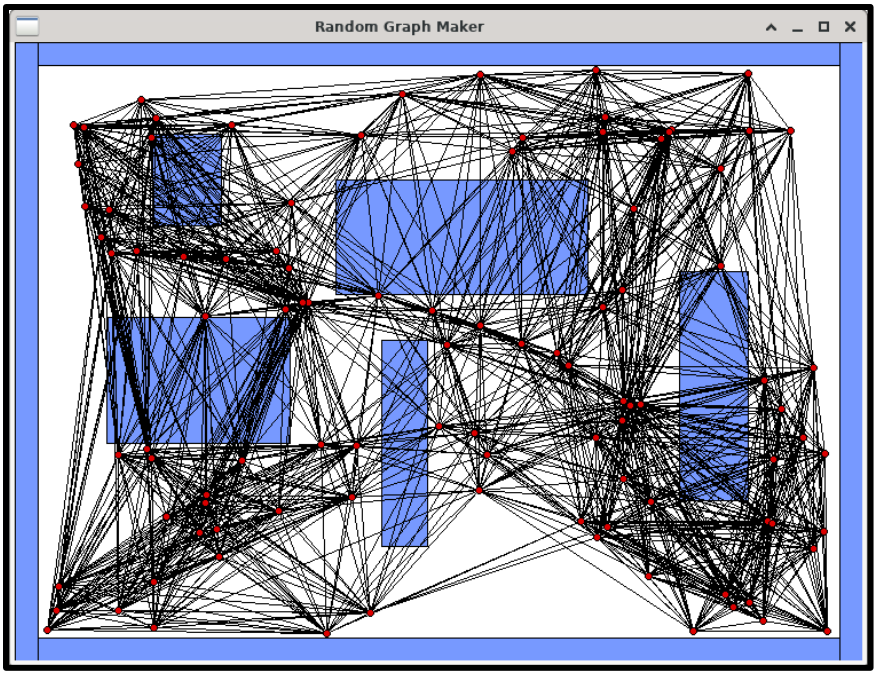
When checking for the nearest neighbours you will need to compute the distance between two vertices. You need not compute the exact distance because we don't need that value. We just need to find the closest neighbours. So, it is enough to compute the square of the distance between the points. This is faster since we do not have to compute the square root operation. The square of the distance between two points (x_1,y_1) and (x_2,y_2) is $(x_2-x_1)(x_2-x_1) + (y_2-y_1)(y_2-y_1)$. You can use whatever algorithm you want to determine the k-nearest neighbours, but simplest is best.

Once you have this completed, run your program as follows: `./graphTester 30 3 1` You should see something similar to what is shown here, except that your vertices will be in different locations. You may have to move the window once to make sure that all the vertices are

displayed. Each vertex should have **at least 3** edges connected to it. Some vertices have more than three edges connected to them. That is because in addition to that vertex's **3**-nearest neighbours, other vertices have that vertex as their neighbour, so they connect to it as well.



Run your program again as follows: `./graphTester 100 25 1` You should see something "similar to" what is shown below. Each vertex will have exactly **25** nodes in its **neighbours** list.

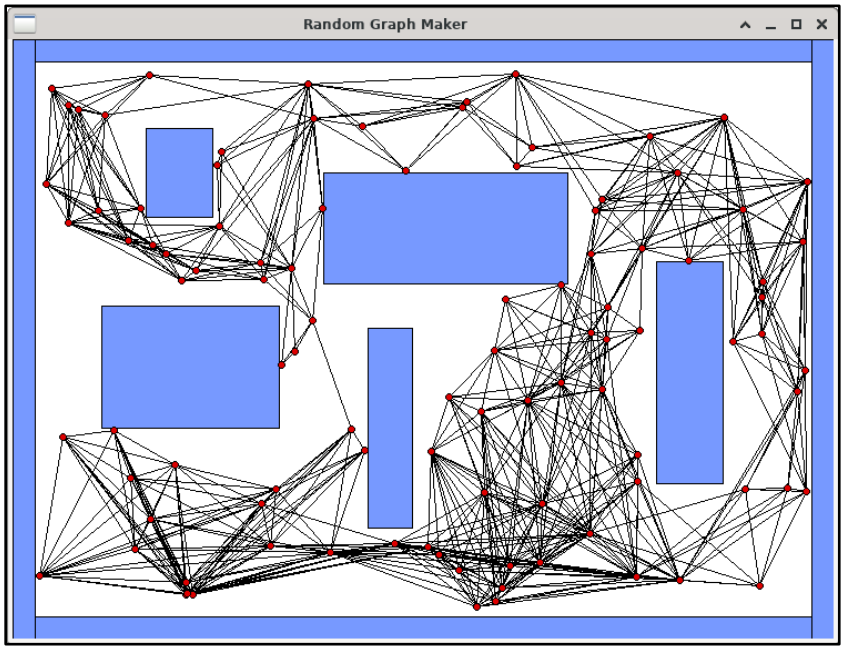


(4) Now we need to make sure that we DO NOT add any edges to the graph that cross obstacles. In your code, before you decide to add a neighbour to a vertex, you will need to first make sure that the edge that would join them does not cross any obstacles. If it does, then you will not add the neighbour. That means, it is possible that a vertex will end up with zero neighbours. If, for example, we find the **25**-nearest neighbours for a vertex **v** and **18** of those neighbours would

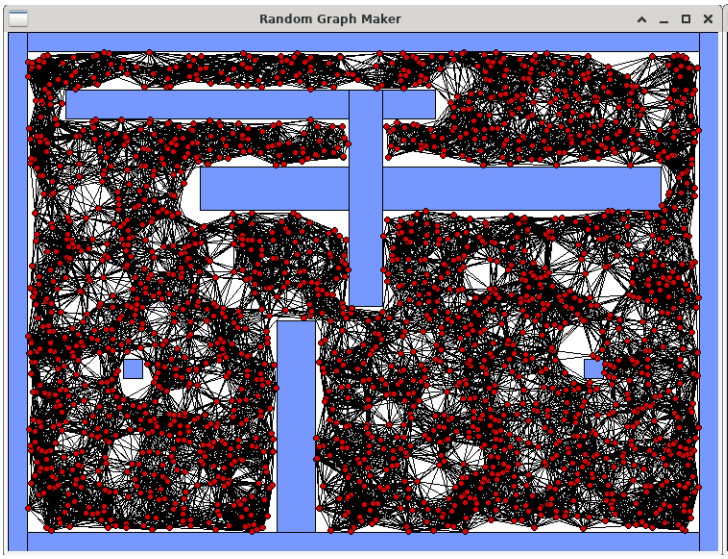
result in an edge that would cross an obstacle, then we only add the **7** neighbours to **v** that do not intersect anything, resulting in **v** having a **neighbours** list with only **7** nodes in it.

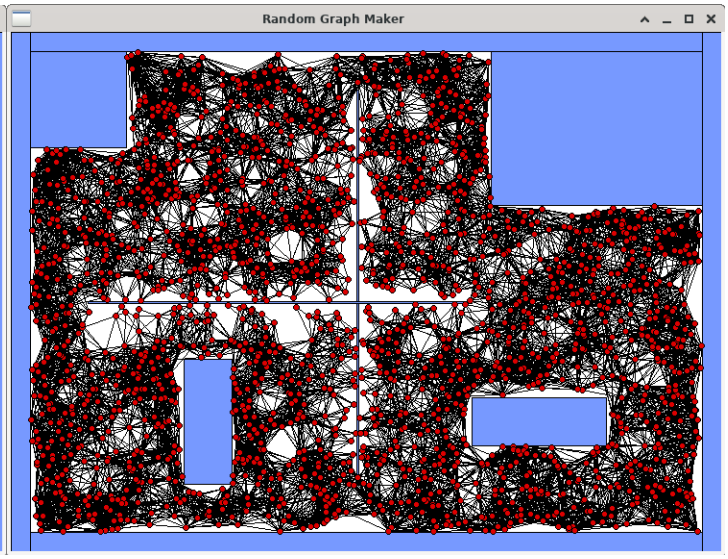
To do this, you should make use of the **linesIntersect()** function provided. Before adding neighbouring-vertex **w** to vertex **v**'s **neighbours** list, you will need to ensure that the line segment from **v** to **w** does not intersect any of sides of any rectangular obstacle in the environment's **obstacles** list.

Once you get your code written, run your program again as before: `./graphTester 100 25 1` You should see something "similar to" what is shown below. None of the edges should intersect any blue obstacle. Run it many times.

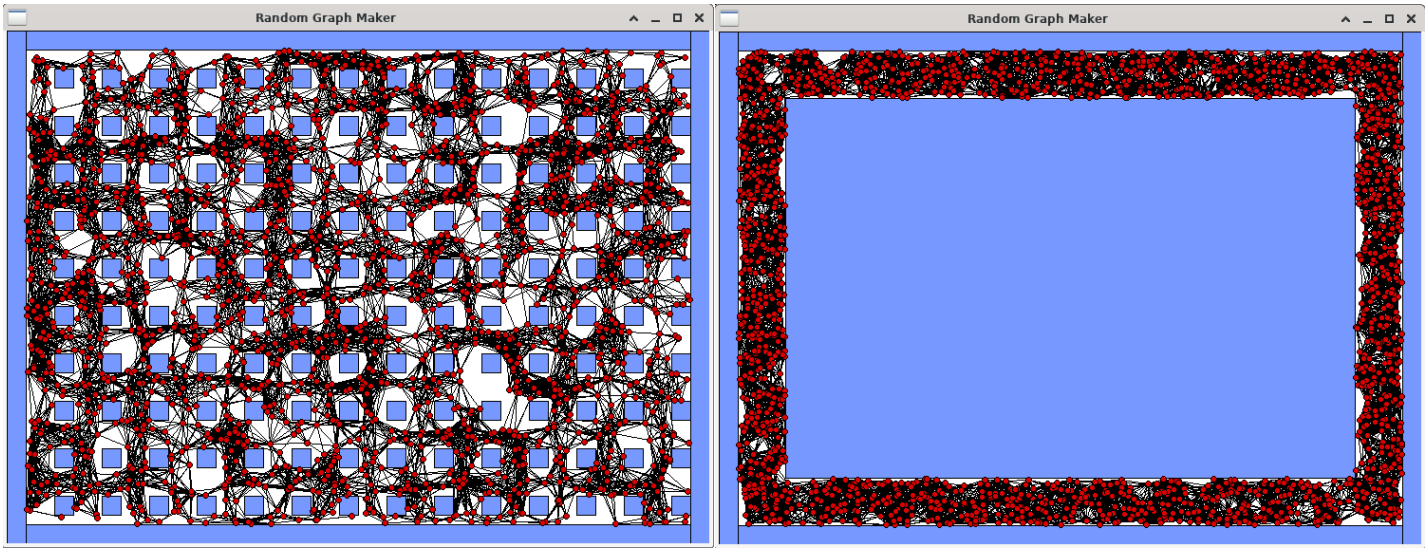


Try out a some of the other environments. Here are environments **2** & **3** with **2000** vertices and **k**=**25**:





Here are environments **4** & **5** with **2000** vertices and **k**=**25**:



(5) Finally, write code in the **cleanupEverything()** procedure so that your program has no memory leaks **nor errors**. Use `valgrind –leak-check=yes ./graphTester 2000 25 1` to do this … although you should try various combinations of grid sizes and environments as well.

IMPORTANT SUBMISSION INSTRUCTIONS:

Submit:

1. A **Readme** text file containing
   - your name and studentNumber
   - a list of source files submitted
   - any specific instructions for compiling and/or running your code

2. Your **makefile** and ALL the **.c** and **.h** files needed to compile and run.

The code **MUST** compile and run on the course VM.

- If your internet connection at home is down or does not work, we will not accept this as a reason for handing in an assignment late ... so make sure to submit the assignment WELL BEFORE it is due !

- You WILL lose marks on this assignment if any of your files are missing. So, make sure that you hand in the correct files and version of your assignment. You will also lose marks if your code is not **written neatly with proper indentation and containing a reasonable number of comments.** See course notes for examples of what is proper indentation, writing style and reasonable commenting).
