Programming Technique

Lab 6 – Pointer

1 Expected outcomes

- Understand the core of a pointer
- Declare a pointer referencing a basic data type, struct and to another pointer.
- Access memory address through pointer
- Understand when to use *, & and ->
- Use dynamically allocated memory

2 Mandatory exercises

Exercise 1. Write a program according to the following steps:

- Declare two variable called n and d with data type int and double respectively.
- Declare a pointer ptr to point to an int: int*.
- Applying the following assignments:
  - Assign the address of n to ptr.
  - Assign the address of d to ptr.
  - Assign 0 to ptr.
  - Assign 1000 to ptr.
- Compile the program and discuss the result.

Exercise 2. Write a program to print the size (in byte) of basic data types:

- int, int*
- long double, long double*
- ...

Exercise 3. Write a program following these steps:

- Declare a variable n and initialize it with 100.
- Declare three pointers ptr1, ptr2 and ptr3 with data type int*.
- Applying the following assignments:
  - Assign the address of n to ptr1.
  - Assign the address of n to ptr2.

    - Assign ptr1 to ptr3.
- Print:
    - n, use “%d”.
    - *ptr1, use “%d”.
    - *ptr2, use “%d”.
    - *ptr3, use “%d”.
    - &n, use “%p”.
    - &ptr1, use “%p”.
    - &ptr2, use “%p”.
    - &ptr3, use “%p”.

Exercise 4. Write a program to:

- Declare an int array with a maximum of 10 elements, initialize the first 5 elements with 10, 20, 30, 40, 50. The name of the array is arr.
- Declare an int* pointer called ptr.
- Applying the assignments:
    - ptr = arr;
    - ptr = &arr[0];
    - ptr = &arr[1];
- Print the following values in different lines:
    - ptr[0]
    - ptr[1]
    - ptr[2]

Exercise 5. Write a program to:

- Declare an int* pointer called ptr.
- Assign ptr = 0.
- Print the following:
    - &ptr, use “%p”
    - *ptr, use “%d”
- Compile the codes and discuss the result.

3 Additional exercises

Exercise 1.

a) Re-implement the following exercises (listed in the array lab file) using pointer. To be precise, you still have to declare two static array storing N elements called vector_1 and vector_2.

b) Declare two double* pointers ptr1 and ptr2.
c) Apply these assignments:

```
ptr1 = vector_1;

ptr2 = vector_2;
```

d) Use ptr1 and ptr2 to access the arrays instead of vector_1 and vector_2.

Write a program to:

- Receive two vectors in N-dimensional space, N is defined as a constant by #define.
- Compute and print:
  - The length of each vector. For example: a = [a1 a2 a3 a4] in 4D has a length of sqrt(a1*a1 + a2*a2 + a3*a3 + a4*a4).
  - Dot product between two vectors. For example: a = [a1 a2 a3 a4] và b=[b1 b2 b3 b4]. The dot product is a1*b1 + a2*b2 + a3*b3 + a4*b4.

Guideline:

There are two ways to assign the address of vector_1 to ptr1:

- ptr1 = vector_1;
- ptr1 = &vector_1[0];

The same goes for ptr_2.

After assigning the address of an array to a pointer, we can access elements of the array through the pointer in two ways:

- Treating the pointer as if it is a normal array. Typing ptr1[0] to access the first element, ptr1[1] to access the second element, etc.
- Moving the pointer from the first element to the second, from the second to the third, etc. Each time, we can use the dereference (*) to get the value of the element. If we assign ptr1 = vector_1 then *ptr1 is the same as vector_1[0]. If we increase the pointer by one: ptr1++, the *ptr1 is the same as vector_1[1].
- Use pointer as an address, use the addition operator and index i to get the address of the i-th element of the array. *(ptr1+0) is vector_1[0], *(ptr1+1) is vector_1[1], *(ptr1+i) is vector_1[i];

After you are able to access elements of an array through pointers, you can start calculating the length and dot product using accumulation technique.

Exercise 2.

a) Declare a positive integer N. N indicates the length of a vector.
b) Input N.
c) Allocate two dynamic array using:

```
double *vector_1 = (double*) malloc(N*sizeof(double));
double *vector_2 = (double*) malloc(N*sizeof(double));
Or
double *vector_1 = new double[N];
double *vector_2 = new double[N];
```

d) Generate random values for the vectors ranging from -10 to 10.
e) Calculate the lengths of two vectors and their dot product:
f) Print the result.
g) Free the memory.

Guideline:

To generate random real values, you can use <random> in C++ 11. Here is an example on how to use it:

```
    std::random_device rd;

    std::mt19937 gen(rd)

    std::uniform_real_distribution<double> dis(-10, 10);

    std::cout << dis(gen) << ' '; //Each call to dis(gen) generates a new
random double
```

After dynamically allocating some memory, you must remember to free them later, using:

- free(vector_1); vector_1 = NULL; // If you used malloc
- delete[] vector_1; vector_1=NULL; // If you used new

Notice: after freeing the memory, always assign NULL to the corresponding pointer to avoid memory access violation.

Exercise 3.

Write a program to:

a) Declare a positive integer N. N indicates the length of a vector.
b) Input N.
c) Allocate a square matrix of N * N integers.

d) Generate random values for the matrix ranging from -40 to 50.
e) Print the matrix.
f) Print the matrix after flipping it over its diagonal.
g) Free the memory allocated in step c).

You must do this problem using two methods:

a) Method 1: still use static array to store the data. Declare a pointer pointing to the address of the static array. Access the elements of the array through the pointer.
b) Method 2: Do not use static array. Use a pointer and allocate memory dynamically.

Guideline:

- As you may have known, a normal pointer can only point to a 1D array. Therefore, to dynamically allocate a square matrix, we can use 2D pointers. A 2D pointer points to a normal pointer (1D pointer). Each 1D pointer will point to a row of the matrix.
- For example, to allocate a 2 x 3 matrix:
  - int **a = new int*[2];
  - a[0] = new int[3];
  - a[1] = new int[3];
- When freeing memory, we must free the memory allocated through the 1D pointers first then free the memory allocated by the 2D pointers after. The order of allocating and freeing are reversed:
  - delete[] a[0]; delete[] a[1]; delete a;
- Assign NULL to unused pointers.

Exercise 4.

Write a program that allows:

a) The user to enter a sequence of real-valued non-negative numbers and store them in a 1D array. The inputting stops when the users enter any negative number or the number of inputs exceeds MAX_SIZE (MAX_SIZE defined by #define).
b) Compute and print: the average value and the standard deviation of the sequence. Here is the equation to calculate the standard deviation of a sequence of N elements:

$$s = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})^2}.$$

c) Invert the sequence without using an additional array.

Guideline:

- Use the same techniques in Exercise 1 to solve a) and b).
- To solve c), review lab 5.

Note: you must implement this exercise using two methods, just like Exercise 3.

Exercise 5.

Write a program that allows:

a) Randomly generate pixels of a gray image having R rows and C columns. A gray image is a 2D array, each element has a value in the range 0...255. You should use a suitable datatype for this image.
b) Compute and print the histogram of all pixels in the image. A histogram is a 1D array, has 256 values. Each value at X in the histogram is the number of times X appears in the image. (For example, if pixel 30 appears 5 times in the image, histogram[30] will be 5).

Guideline:

a) Data:
   - Variables to store the numbers of rows and columns.
   - A variable to store the address of the array (2D pointer).
   - A variable to store the histogram (1D array).
b) Algorithm:
   - Input the size of the image.
   - Dynamically allocate the array.
   - Use srand, rand and time to randomly generate integers for each pixel of the array.
   - Compute the histogram.
   - Print the histogram.

Exercise 6.

Write a program to:

a) Input a square matrix with size N. N is defined by #define.
b) Input a vector of size N.
c) Compute the multiplication result between the matrix and the vector R = M x V. Here M is the matrix and V is the vector.
d) Print M, V and R.

Guidelines:

- You can reuse the results from Exercise 3. Allocate memory for the matrix M, the vector V and the result vector R.
- You must calculate the multiplication using two different methods: using array index and array pointer.

Exercise 7.

Let:

- M be a lower triangular matrix of size N.
- X and Y be two vectors in N dimensional space.

Write a program to:

a) Randomly generate numbers for the matrix ranging from -10 to 10. The numbers on the diagonal must not be 0.
b) Randomly generate values for Y.
c) Find X so that M * X = Y.

Guideline:

- Declare and dynamically allocate M using 2D pointer. Note: since we know that our matrix is a lower triangular matrix, hence the values above the diagonal are unused. Therefore, you shouldn't allocate them to save memory.
- When generate random values, remember to check if you are generating for the diagonal so that it does not have any 0.
- Use dynamic allocation to allocate memory for X and Y.
- Refer to **forward substitution** here to solve the M*X=Y equation for X.
- Free the memory.

Exercise 8.

Write a program allowing user to enter 2 matrices (using pointer). Multiply these two and print the result. Remember to free the memory before the end of the program.

Guidelines:

- Declare and allocate memory dynamically for the matrices A, B and the result matrix C. You should use 2D pointers just as the previous exercises.
- Applying matrix multiplication, similar to Lab 5.
- Free up memory before ending the program.