CS2130: Data Structures and Algorithms Lab (Jul – Nov 2018)

Lab 1

1. **Compiling and running a C program**

gcc <input_filename> -o <output_filename>
./filename

2. **Data types in C**

char: stores a single character and requires a single byte of memory in almost all compilers.
int: stores an integer.
float: stores decimal numbers (numbers with floating point value) with single precision.
double: stores decimal numbers (numbers with floating point value) with double precision.

The C99 standard for C language supports bool variables. Unlike C++, where no header file is needed to use bool, a header file “stdbool.h” must be included to use bool in C.

For a more comprehensive list, do visit https://en.wikipedia.org/wiki/C_data_types

3. **sizeof operator in C**
When *sizeof()* is used with the data types such as int, float, char… etc it simply returns the amount of memory allocated to that data type.

Uses:

a. Calculate number elements in the array

```
int arr[] = {1, 2, 3, 4, 7, 98, 0, 12, 35, 99, 14};
printf("Number of elements :%d",
                                    sizeof(arr)/sizeof(arr[0]));
```

b. To allocate memory dynamically

4. **Pointers in C and C++**

Based on the material at https://www.geeksforgeeks.org/pointers-in-c-and-c-set-1-introduction-arithmetic-and-array/

They store address of variables or a memory location

```
//General syntax
datatype *var_name;

int *ptr;

//Try the following example

int x = 10;
printf(“%p”, &x);

//Assigning address and accessing the value
```

```
int x = 10;
int *ptr;
ptr = &x;
printf(“%p  %d”, ptr, *ptr);


//Pointer arithmetic
int  v[]= {10, 20, 30};
int *ptr;
ptr = v;
for (int i = 0; i < 3; i++)
{
  printf("Value of *ptr = %d\n", *ptr);
  printf("Value of ptr = %p\n\n", ptr);

  // Increment pointer ptr by 1
  ptr++;
}
```

Change **int *ptr** to **char *ptr** and run the code. Note down and explain the output.
How can the above code be changed to print the array of strings {‘H’,’e’,’l’,’l’,’o’}?

```
//Pointer as array names
int  v[]= {10, 20, 30};
int *ptr;
ptr = v;
for (int i = 0; i < 3; i++)
{
  printf("Value of *ptr = %d\n", ptr[i]);
  printf("Value of ptr = %p\n\n", ptr);

  // Increment pointer ptr by 1
  ptr++;
}
```

```
//Pointer and multi-dimensional arrays
```

Consider the following 2-d array

```
int nums[2][3]  =  { {16, 18, 20}, {25, 26, 27} };
```

In general nums[i][j] = *(*(nums + i) + j)

5. **Pointer Typecasting**

Void pointer:

- A pointer that has no associated data type with it
- Can hold address of any type and can be typcasted to any type.

```
int main ()
{
```

```
    int a = 10;
    char b = 'x';

    void *p;

    p = (void*) &a; // void pointer holds address of int 'a'
    printf("%d\n", *(int *)p);

    p = (void*)  &b; // void pointer holds address of char 'b'
    printf("%c", *(char *)p);

    return 0;
}
```

- Pointer arithmetic with void pointers

```
int main()
{
   int a[2] = {1, 2};
   char c[2] = {'h','e'};
   void *ptr = &a;
   ptr = ptr + sizeof(int);
   printf("%d\n", *(int *)ptr);

   ptr = &c;
   ptr = ptr + sizeof(char);
   printf("%c", *(char*) ptr);
   return 0;
}
```

Note: May not work in compilers other than gcc.

6. **Allocating and freeing pointers**

malloc():

- Allocates requested size of bytes and returns a void pointer to the first byte of allocated space
- Doesn't initialize the allocated memory

```
void* malloc( size_t size );
```

calloc();

- Allocates space for an array elements, initializes to zero and then returns a void pointer to the first byte of allocated space

```
void* calloc( size_t num, size_t size );
```

- Parameters:

1) Number of blocks to be allocated.
2) Size of each block.

```
int main()
{
    int *arr;
    arr = (int *)malloc(5 * sizeof(int));

    for(int i = 0; i < 5; i++)
    {
        printf("%d\n", *(arr+i));
        *(arr + i) = i * i;
    }

    for(int i = 0; i < 5; i++)
        printf("%d\n", *(arr+i));

    return(0);
}
```

Try the above example with calloc()
Note: ***It would be better to use malloc over calloc, unless we want the zero-initialization because malloc is faster than calloc.***

realloc():

- Change the size of dynamically allocated memory

  ```
  void *realloc(void *ptr, size_t size);
  ```

(Deallocates the old object pointed to by ptr and returns a pointer to a new object that has the size specified by size)

free():

- To deallocate memory that was allocated with malloc.

```
int main()
{
    int *ptr = (int *)malloc(2*sizeof(int));
    int i;
    int *ptr_new;

    *ptr = 10;
    *(ptr + 1) = 20;

    ptr_new = (int *)realloc(ptr,3*sizeof(int));
    *(ptr_new + 2) = 30;
    for(i = 0; i < 3; i++)
        printf("%d ", *(ptr_new + i));

    free(ptr_new);
```

```
    return 0;
}
```

Familiarizing the tool: valgrind

7. **Structures**
   - Structure is a user defined data type
   - Used to group items of possibly different types into a single type
   - A structure variable, ‘p’ can either be declared either with the structure declaration or as a separate declaration like basic types:

Declaration:

```
struct Point
{
        int x, y;
}p;
```

OR

```
struct Point
{
        int x, y;
};

int main()
{
        struct Point p;
}
```

Initialization:

```
struct Point
{
  int x, y, z;
};

int main()
{
  struct Point p1 = {.y = 0, .z = 1, .x = 2};
  struct Point p2 = {.x = 20};

  printf ("x = %d, y = %d, z = %d\n", p1.x, p1.y, p1.z);
  printf ("x = %d, y = %d, z = %d\n", p2.x, p2.y, p2.z);
  return 0;
}
```

The output is as follows:
x = 2, y = 0, z = 1
x = 20, y = 0, z = 0

- Array of structures:

```
struct Point
{
        int x, y;
};
int main()
{
        struct Point arr[10];
        arr[0].x = 10;
        arr[0].y = 20;
        printf("%d %d", arr[0].x, arr[0].y);
        return 0;
}
```

Write a C program to determine if the given three points in 2D are collinear. Points are to represented using structure.

8. **Pointers for Structures**

```
struct Point
{
  int x, y;
};

int main()
{
  struct Point p1 = {1, 2};
  struct Point *p2 = &p1;
  printf("%d %d", (*p2).x, p2→y);   //two ways of accessing members
                                       of structure using pointer
  return 0;
}
```

//Type casting structure pointers to primitive datatype pointers

```
struct X
{
   int id;
   int val;
};

struct Y
{
   int id;
   int val[2];
};
```

```
int main()
{
    struct X x;
    struct Y y;
    void *ptr;
    x.id = 0;
    x.val = 10;

    y.id = 1;
    y.val[0] = 23;
    y.val[1] = 34;


    ptr = (void*) &y;
    //ptr = (void*) &x;
    printf("%d\n", *((int*) ptr));

    return 0;
}
```

Exercise

Write a C program to implement the following. Add comments to the code describing the code itself. You are expected to follow good programming practices. Refer to the book by Dietel and Dietel for the same.

1. Read two positive integers p and q and allocate memory for p $*$ q integers.
2. Read p $*$ q integers and interpret them as the elements of a p × q matrix M entered row-wise.
3. Print M as a 2D array by accessing each of its elements using pointers.
4. Compute the transpose of M and print the result as a 2D array.
5. Read two positive integers x and y and increase the size of M by x rows and y columns using realloc().
6. Read the additional elements of M row-wise and print M as a 2D array.
7. Print thse sum of the elements of M .