COMP2401—Tutorial 6
Arrays of pointers, command line and valgrind tool

Learning Objectives
After this tutorial, you will be able to:
- Manipulate arrays of pointers
- Manage dynamic memory (allocate and free)
- Check if your program has a memory leak using the valgrind tool
- Implement command line arguments

Submit your tutorial in a tar file (t6.tar)

Tutorial

Download the tar file t6.tar and extract the files.

1 Allocating and Searching in an array

Purpose: a. passing structures as pointers, b. accessing structure fields using "->" operator c. using pointer arithmetic d. taking advantage of "call by value"

To do:

1. The file find_struct.c contains a declaration of a struct emp, a main() function and a function that populates the emp struct. Review the code and make sure that you understand it.

2. The `main()` function creates an array – `empArr` – of size `MAX_EMPLOYEES` where each element in the array is a pointer to an `emp struct`. Add code to initialize each of the elements in the structure to NULL. Recall that each element in the array holds an address to an `emp struct` (each element is a pointer). Use a "for" loop to (e.g., for (i = 0; i < MAX_EMPLOYEE; i++) set the pointer NULL).

3. For each record in the array empArr

   a. Allocate new memory for a struct emp by calling malloc() and store the result in the corresponding record in empArr. Do not forget to check if the allocation was successful.

   b. Check if the allocation was successful by comparing the return value to NULL. If the value is NULL then print an error message and return(2);

   c. Once memory for a new struct emp is allocated initialize (populate) the record with employee data by calling the provided function `populateEmployee()`. Pass to the function the address of the newly allocated memory. Review the code of `populateEmplyee()`.

4. Code a function that compares a single employee record against a given key (in this case it is a family name). The function specifications are:

   Prototype:

   ```
   int cmpEmployee(struct emp *p, char *familyName)
   ```

   input:
   familyName - family name of employee to be searched
   p - a pointer to an employee record

   Output:
   None

   Return:
   0 - if family name of employee in the provided record does not match the familyName
   1 - if family name of employee in the provided record matches the familyName

   Note 1: function prototype is already in the C file
   Note 2: use the operator -> to access the fields inside struct emp

5. Code a function that searches the array emp for an employee by family name.

   The function specifications are:

   Prototype:

   ```
   struct emp * findEmployee(struct emp **arr, int arraySize, char
   *familyName);
   ```

   input:
   arr – an array of pointers to employees
   arraySize – the number of elements in the array
   familyName – familyName to be used as a key

   Output:
   None

   Return:
   NULL – if no matching record was found
   a pointer to a struct in the array that matches the family name

Pseudo Code

```
// iteratively traverse the array using pointer arithmetic.  Namely by augmenting the value of
// arr by one  at every iteration.  Note that here we can take advantage of the fact that the pointer is
// a call by value and we can use pointer arithmetic.
// Also note that you will have to take care of the precedence order between the "*" and the "->"
operators

    // compare the family name of the record with the key that was given.

    // if a record with a matching name is found then print the record (see below)
```

Record printing

firstName   familyName
 salary=   years of service =

E.g.,
Dina Door
salary= 28500.00 years of service = 9.00

6. Call the function from `main()`.

2 Two dimensional array

Purpose: To create a two dimensional array by allocating memory from the dynamic memory. Creating a two dimensional array is somewhat different than declaring a two dimensional array (e.g., int a[5][6]). Here you will allocate a 2D array of integers. To do so you will first allocate memory for a 1D array of integer pointers and then for each one you will allocate an array of integers. Namely, the second dimension is an array of integers. Figure 1 shows an example memory layout of a 2D array arr. Here arr is declared as int **arr. The first dimension is of size three and is an array of pointers to ints. The second dimension is a one dimensional array of size 2 of int. Note that the pointers arr[0], arr[1] and arr[2] point to different memory locations. Each memory location has two integers. This is similar to the layout of command line variable (argv).

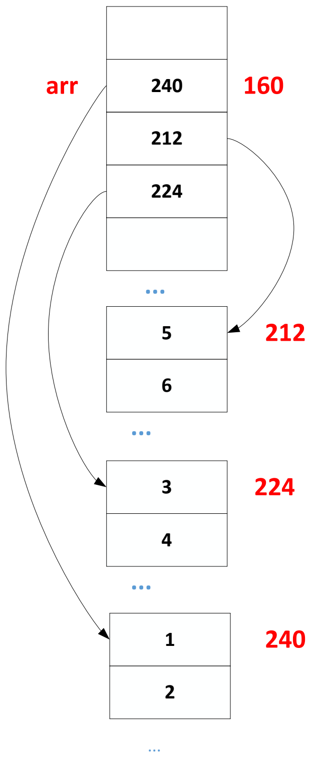

Figure 1: a 2D array of integers arr. Note that this layout is different than int a[3][2]

2.1.1 Create a 2D array Pseudo Code

Pseudo code overview

Open file main2D.c (in t6.tar) and complete the main function described below.

Input:

Here we will use "defined" values as the dimensions of the array where DIM_1 is the size of the first dimension and DIM_2 is the size of the second dimension

Output:

0 – if the operation was successful

1 – if no memory was allocated

Pseudo code + code

```
#define DIM_1   10
#define DIM_2     5

int main(int argc, char **argv)

{
        int **arr = NULL;  // declare the arr variable as a two dimensional array

        int size;           // will hold the size of the memory to be allocated

        int i;


        // allocate memory for the array of pointers to an int.  Namely, each element in the array will be
of type int *

        // compute the value of size.  Here you need to have an array that can hold DIM_1 addresses.
The size of each element is sizeof(int *), which gives the number of bytes to a pointer to an int
        size =  ???? ;
        arr = (int **) malloc(size);        // note the type casting matches the arr declaration
        // add code to check if arr was properly allocated by checking whether it is NULL

        // set each of the pointers in the array to NULL  for (i = 0 ; i < DIM_1; i++) set arr[i] to NULL)

        // allocate memory for each of the elements of arr
        for (i = 0 ; i < DIM_1; i++) {
                // allocate memory for each element of arr[i]
                // here the allocated memory is a one dimension of integers
                // add code
        }

        // using two nested for loop initialize each value of the two dimension array from 1 to
DIM_1xDim_2

        // using two nested for loop print one row at a time the array
        // pseudo code
        // set count to 1
        // for i starting at 0 until DIM1
        //      for j starting at 0 until DIM2
        //              set arr[i][j] to count;
        //              increment count by 1;
        //      }
        //}

        // print the array as a 2d matrix
        return(0);
}
```

Complete the main function and test it. Using the valgrind tool (see pp116-117 of Ch 3 of your notes), ensure that when the program terminates 11 blocks of memory were allocated. 10 blocks of memory of size 20 bytes and one block of memory of size 40 bytes (in a 32 bit architecture) or 80 bytes (in a 64 bit architecture).

2.1.2 Free the memory

In order to free the memory one needs to ensure that all allocated memory was freed. This is accomplished by freeing the memory in reverse order to the memory allocation order. Namely first releasing the memory of each of the elements in arr[i]. Once this is done the array arr can be released.

Releasing the memory of arr

```
// free memory for each of the elements of arr
for (i = 0; i < DIM_1; i++) {
        // free the memory for each element of arr[i]
}

// free the memory of arr
```

2.1.3 Ensuring that memory was released

Use valgrind to ensure that all memory was released

3 Main2D.c with command line arguments

Make your main2d.c accept the two dimensions as command line arguments. Copy main2d.c into main2d-args.c and make the additions in that new file. To review command-line arguments see Chapter 3 section 2 of your course notes (pp102-103).

Submit your tutorial work (3 files) in a tar file t6.tar!