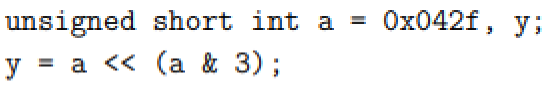
For each of the following, what is the **hexadecimal** value of y after the assignment has been executed? Remember that an unsigned short int is stored in 16 bits. Write down your answer in proper hexadecimal format: 0x????

(a) [**1 mark**]

```
unsigned short int a = 22, y;
y = a ^ a;
```



```
unsigned short int a = 0x042f, y;
y = a << (a & 3);
```

Implement the following function (using this exact function prototype):

```
unsigned int rotate_left(unsigned int i, int n);
```

rotate_left returns the result of shifting the bits in i to the left by n places, with the bits that were "shifted off" moved to the right end of i. For example, the call rotate_left(0x12345678, 4) must return 0x23456781 and the call rotate_left(0xc2345678, 2) must return 0x8d159e3. You **MUST** use bitwise operators to implement this function. You will not get any marks if you implement it in another way.

```
unsigned int rotate_left(unsigned int i, int n){
```

Consider a linked list that keeps the list of students enrolled in a course. The structure of each Student in the list is as given, and the head of the linked list is defined in a struct called `StudentList`. If the list is empty, then head is NULL.

Write three functions for this question: `createStudentList`, `insertStudent`, and `destroy`.

The details of each function are provided in the sub-question. You must dynamically allocate memory for each node. There is no maximum size for the length of each student name so student names must also be dynamically allocated. **You must handle all corner cases as applicable to each function: e.g., handling empty lists, lists with one node etc.** You can assume that any list passed to insertStudent or destroy has been created using createStudentList.

A main function has also been provided to illustrate how a client program would use these functions.

```
struct Student{                                struct StudentList{
    char *name;                                    struct Student *head;
    float grade;                             };
    struct Student *next;
};

int main(){
    struct StudentList *studentList = createStudentList();
    insertStudent(studentList, "Alice", 3.7); //list = Alice
    insertStudent(studentList, "Bob", 3.2); //list = Alice, Bob
    insertStudent(studentList, "Mark", 3.6); //list = Alice, Bob, Mark
    destroy(studentList);
    return 0;
}
```

```
//returns a new dynamically allocated empty student list
struct StudentList* createStudentList(){
```

```
//This inserts a new student with the given name and grade to the **END of the list**
void insertStudent(struct StudentList *studentList, char *name, float grade){
```

```
//Deletes all nodes from the list while ensuring that there are no memory leaks,
//and also frees the memory allocated for the list.
//After calling destroy, we expect that all memory has been freed and that
//the client will no longer access studentList
void destroy(struct StudentList *studentList){
```

Consider the following declarations inside prog.c for the next four sub-questions.

```
struct employee {
    char name[100];
    char phone[12];
};

struct employee list[200];
```

(a) **[3.5 marks]** Write a comparison function called cmp to be passed to qsort to sort list based on name in **DESCENDING** order. cmp will be defined in prog.c.

The following is an excerpt of the man page of qsort. The prototype of qsort also shows the prototype of the comparison function it accepts:

```
void qsort(void *base, size_t nmemb, size_t size,
                int (*compar) (const void *, const void *));

The qsort() function sorts an array with nmemb elements of size size.
The base argument points to the start of the array.

The contents of the array are sorted in ascending order according to a
comparison function pointed to by compar, which is called with two arguments
that point to the objects being compared.

The comparison function must return an integer less than, equal to, or greater than
zero if the first argument is considered to be respectively less than, equal to,
or greater than the second. If two members compare as equal, their order in
the sorted array is undefined.
```

(b) **[2 marks]** Using the cmp function you defined in the previous step, write a single call to qsort that sorts the first **100** employees in list, based on the name as the sort criterion. Assume that list is already correctly initialized with entries.

(c) **[1 mark]** You want to restrict the use of cmp only to the file prog.c. What is the new declaration of cmp in prog.c that would enforce this restriction?

The swap function below swaps two elements in a given array. It takes the array and the two ***indices*** of the elements to be swapped. Comments shown in the main function illustrate what the swap function does.

Re-write the code below such that the swap function takes the array `a` and two ***pointers to array elements*** rather than the integer indices currently used. The swap function must have exactly these 3 parameters, no more and no less. Make sure to change the function prototype, update the function code, and modify the calls to swap in main accordingly.

```
void swap(int a[], int position1, int position2);

int main(){
    int a[] = {1, 5, 6, 3, 2, 89, 34, 3, 9, 10};

    swap(a, 0, 9); //array a should now be {10, 5, 6, 3, 2, 89, 34, 3, 9, 1}
    swap(a, 1, 3); //array a should now be {10, 3, 6, 5, 2, 89, 34, 3, 9, 1}

    return 0;
}

void swap(int a[], int position1, int position2){
    int temp = a[position1];

    a[position1] = a[position2];
    a[position2] = temp;
}
```

The following program contains memory leaks. Fix the program so it contains no memory leaks. **Make sure to indicate where your changes will be added in the program**. You should not change any of the given statements during your fixes.

```
#include <stdlib.h>
#include <stdio.h>

int main(){

    int **numbers = malloc(5 * sizeof(int*));

    for(int i = 0; i < 5; i++){
        numbers[i] = malloc (sizeof(int));
        *(numbers[i]) = i;
    }

    printf("Array contains:\n");
    for(int i = 0; i < 5; i++){
        printf("%d\n", *(numbers[i]));
    }

    return 0;
}
```

Indicate whether each of the following statements is True or False.

(a) The `printf` statement will print 6

```
void addOne(int x){ x +=1; }
int main(){
    int x = 5;
    addOne(x);
    printf("%d", x);
}
```

(b) Memory allocated using `malloc` is allocated on the Stack

(c) Memory allocated using `malloc` exists for the lifetime of the program, unless explicitly freed

(d) An array can have elements of different types

(e) The leftmost bit of a negative number that is stored in a signed integer is 0

(f) Given a function `void findElement(int a[]);`, we can use `sizeof(a)/sizeof(a[0])` inside `findElement` to get the number of elements in the passed array `a`.

(g) Given the following array:

```
char *weekdays[7] = {"Sunday", "Monday", "Tuesday", "Wednesday",
                                    "Thursday", "Friday", "Saturday"};
```

If I were to print `weekdays + 2`, I would get the character `n`

You are given the main function below and are asked to implement **4 functions**. You must decide on the type of the parameters each function will take based on the following description and the `main` function code that shows how these functions will be used. **You MUST NOT change the code in main: your implemented functions MUST work correctly with the function calls in main**

1. **[4 marks]** `initArray` takes (1) **a POINTER to an array of character pointers** it is supposed to allocate memory for and (2) the number of elements it should allocate. After allocating memory, it sets each element to `NULL`. After calling `initArray` in the example below, `nameArray` would be the memory address of an array of 5 character pointers.

2. **[4 marks]** `addToArray` takes (1) an array of character pointers, (2) the number of elements in the array, and (3) a string. It finds the first available element in the array and makes it point to a **dynamically** allocated string that contains the value of the passed string. If there are no available elements, it prints an error message and returns.

3. **[2.5 marks]**`print` takes an array of character pointers and the number of elements it should print. It then prints a comma separated list of the strings stored in the array.

4. **[3 marks]** `swap`, which takes (1) an array of character pointers, (2) an index `i1`, and (3) an index `i2`. It then swaps the strings in indices `i1` and `i2`. You can assume that both indices are within the bounds of the array. **must** implement swap **WITHOUT** allocating any new memory using malloc/calloc/realloc or creating a new array.

```
int main(){
    char **nameArray;
    initArray(&nameArray, 5); //dynamically allocates memory for nameArray

    //first call adds "Bob" in the first available element in the array
    //(after dynamically allocating memory for Bob)
    addToArray(nameArray, 5, "Bob");
    addToArray(nameArray, 5, "Alice");
    addToArray(nameArray, 5, "Mary");
    addToArray(nameArray, 5, "Tom");
    addToArray(nameArray, 5, "Cindy");

    print(nameArray, 5); //prints Bob, Alice, Mary, Tom, Cindy
    swap(nameArray, 0, 1); //swaps elements 0 and 1
    print(nameArray, 5); //prints Alice, Bob, Mary, Tom, Cindy
    swap(nameArray, 1, 3); //swaps elements 1 and 3
    print(nameArray, 5); //prints Alice, Tom, Mary, Bob, Cindy
    swap(nameArray, 4, 0); //swaps elements 4 and 0
    print(nameArray, 5); //prints Cindy, Tom, Mary, Bob, Alice
}
```