

CS 137

# Pointers, Memory Allocation, Arrays, and Vectors

Fall 2025

Victoria Sakhnini

# Table of Contents

Pointers .....	2
Sections of memory .....	4
Pointer Arithmetic and Arrays .....	8
Dynamically Allocated Memory .....	11
Memory model .....	11
NULL Pointer .....	12
Pointers to Structures & Struct Hack .....	15
Vectors (Variable Size Array).....	18
Additional Examples.....	22
Extra Practice Problems .....	23

## Pointers

What if we want functions to change values inside memory that are outside the scope of a function?

Let us write a program that swaps the values of two variables by creating the function swap:

```
1. #include <stdio.h>
2.
3. void swap(int a, int b){
4.     printf("a=%d, b=%d\n", a, b);
5.     int tmp;
6.     tmp = a;
7.     a = b;
8.     b = tmp;
9.     printf("a=%d, b=%d\n", a, b);
10. }
11.
12. int main(void){
13.     int x = 10;
14.     int y = -15;
15.     printf("x=%d, y=%d\n", x, y);
16.     swap(x, y);
17.     printf("x=%d, y=%d\n", x, y);
18.     return 0;
19. }
```

What is printed?

x=10, y=-15

a=10, b=-15

a=-15, b=10

x=10, y=-15

Why?

Because a copy of the x and y values were assigned to the parameters a and b.

Any changes to a and b did not affect x and y. But how can we make the changes in the swap function affect the variables x and y in the main?

We saw this already when we changed values in an array. We can do this with other values by using pointers and references.

Let us have a look at the following example: *(please do pay attention to the comments)*

```

1. #include <stdio.h>
2.
3. int main(void)
4. {
5.     int i = 6;
6.     int *p;
7.     p = &i;          // we say "p has the address of i" or "pointing at i"
8.     *p = 10;         // p now points at i,
9.                     // the value that p is pointing at was changed to 10,
10.                    // thus i was changed to 10 as well
11.     printf("%d \n", i); // 10 is printed
12.     int *q;
13.     q = p;           // q is pointing where p is pointing at which is i
14.     *q = 17;         // q and p now point at i,
15.                     // the value that q is pointing at was changed to 17
16.                     // thus the value of i was changed to 17 as well
17.     printf("%d \n", i); // 17 is printed
18.
19.     return 0;
20. }
21.

```



int \*p is a pointer to an integer  
as double \*d is a pointer to double.

&i is the address of the variable i; this is different from the value stored in i.

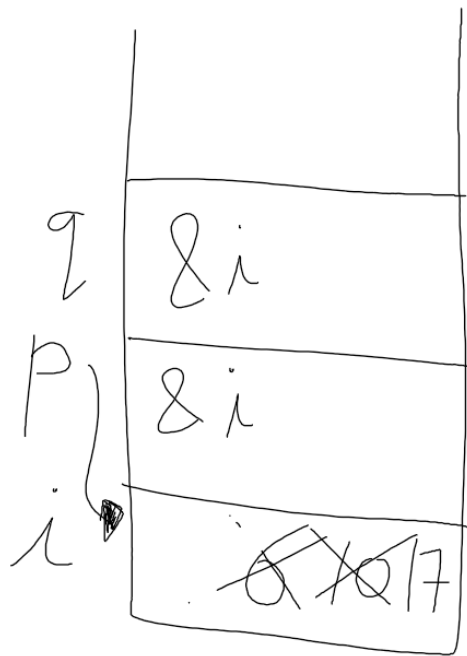
\*p is the dereferencing of p; it is the value stored where p points at, and can be used to modify that value.

## Sections of memory

In this course, we model five sections of memory:

1. **Code:** to store the program code/instructions in machine code (machine-readable), done during compiling
2. **Read-Only Data:** to store the global constants
3. **Global Data:** to store the global variables and make them available throughout the entire execution of the program.
4. **Heap:** used to allocate memory dynamically (we will talk about this later)
5. **Stack:** used to store local variables and return addresses<sup>1</sup> to manage function calls<sup>2</sup>, etc. Each function call creates a stack frame<sup>3</sup>. The stack grows toward lower addresses.

Let's go back to our last program and draw the stack section for it and how it looks before executing `return 0`



---

<sup>1</sup> When we encounter a return, we need to know: “what was the address we were at right before this function was called?” In other words, we need to “remember” the program location to “jump back to” when we return. This location is known as the return address. In this course, we use the name of the calling function and a line number (or an arrow) to represent the return address.

<sup>2</sup> As the program flow jumps from function to function, we need to “remember” the “history” of the return addresses. When we return from `h`, we jump back to the return address in `g`. The “last called” is the “first returned”. This “history” is known as the call stack. Each time a function is called, a new entry is pushed onto the stack. Whenever a return occurs, the entry is popped off of the stack.

<sup>3</sup> Each stack frame contains: 1) the argument values 2) all local variables (both mutable variables and constants) that appear within the function block (including any sub-blocks) 3) the return address

let's review one more example:

What is the output of the following program? Try to figure it out before looking at the solution provided later on.

```
1. #include <stdio.h>
2.
3. void swap(int *p1, int *p2)
4. {
5.     int tmp;
6.     tmp = *p1;
7.     *p1 = *p2;
8.     *p2 = tmp;
9. }
10.
11. int main(void)
12. {
13.     int x = 10;
14.     int y = -15;
15.     printf("x=%d, y=%d\n", x, y);
16.     swap(&x, &y); // Why did we pass &x and &y and not x,y?????
17.     printf("x=%d, y=%d\n", x, y);
18.     return 0;
19. }
```

The solution is on the next page. Don't look before you complete your trace and figure out the output!



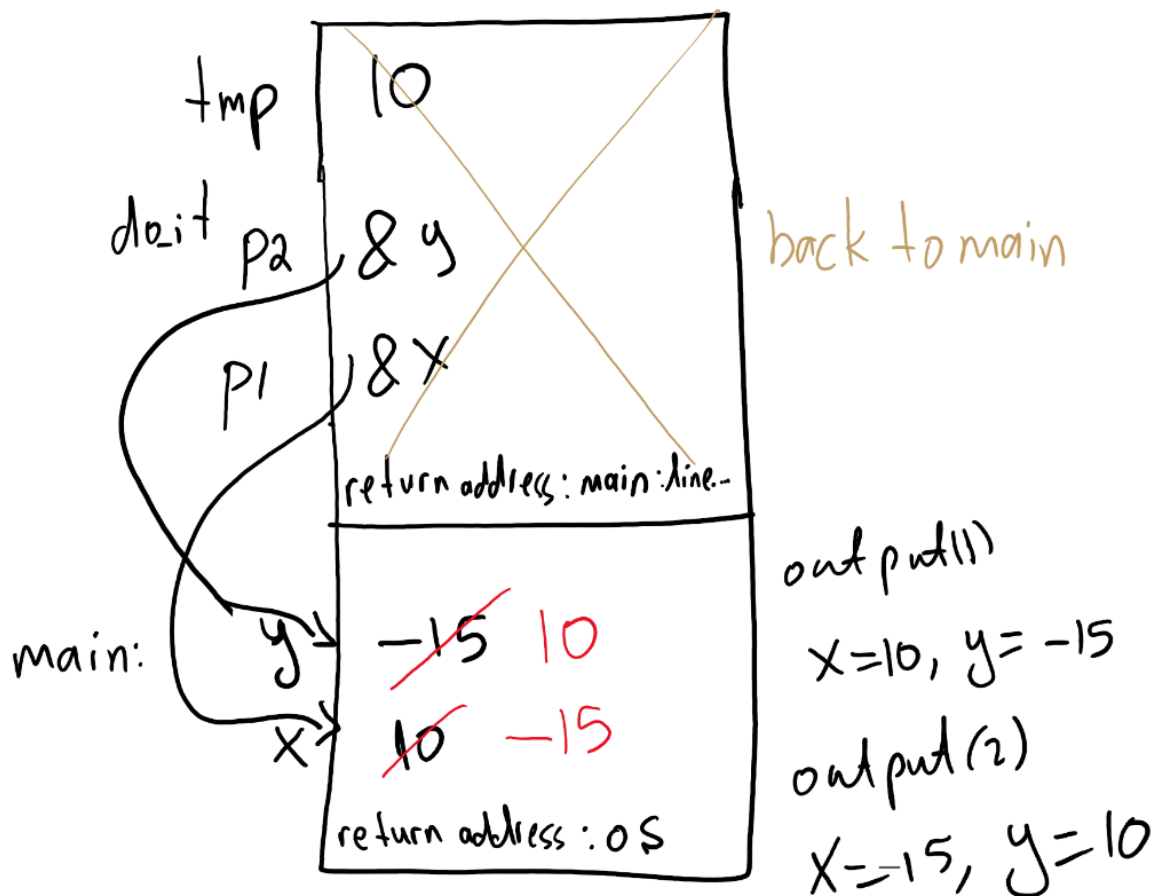
Why did we pass `&x` and `&y` as arguments when we called `do_it`?

Answer: Because `do_it` expects two memory addresses to be assigned to the two parameters of the type pointer.

Solution:

```
Console program output
x=10, y=-15
x=-15, y=10
Press any key to continue...
```

Why? Let's do a trace using a memory model:



**Just for Fun:** What is the output of the following program? Do a manual trace and figure it out before you execute the program. [Reminder: ^ is a bitwise operation]

```
1. #include <stdio.h>
2.
3. int main(void)
4. {
5.     int x = 10;
6.     int y = -15;
7.     printf("x=%d, y=%d\n", x, y);
8.     x ^= y;
9.     y ^= x;
10.    x ^= y;
11.    printf("x=%d, y=%d\n", x, y);
12.    return 0;
13. }
14.
```

One more exercise: What is the output of the following program? Do a manual trace and figure it out before you execute the program. What is returned by the function<sup>i</sup>?

```
1. #include <stdio.h>
2.
3. int calc_array(int a[], int n){
4.     int val = a[0];
5.     int *res = &a[0];
6.     for (int i = 1; i < n; i++)
7.         if (a[i] > val)
8.             {
9.                 val = a[i];
10.                res = &a[i];
11.            }
12.     return *res;
13. }
14.
15. int main(void){
16.     int a[10] = { 1, 10, 67, 876, -76, 0, -45, 8, 9, 1 };
17.     int ans;
18.     ans = calc_array(a, 10);
19.     printf("The result is %d\n", ans);
20.     return 0;
21. }
```



## Pointer Arithmetic and Arrays

C allows an integer to be added to a pointer. **The result depends on the type of pointer used.**

(p+1) adds the `sizeof` whatever p points at.

You can NOT add two pointers.

A pointer q can be subtracted from another pointer p if the pointers are the same type (point to the same type).

Pointers can be compared with the comparison operators: <, <=, ==, !=, >=, >

Example:

```
1. #include <stdio.h>
2.
3. int main(void){
4.     int a[8] = { 2, 3, 4, 5, 6, 7, 8, 9 };
5.     int *p, *q, i;
6.     p = &(a[2]); // p points to a[2]
7.     q = p + 3;   // q points to a[5]
8.     p += 4;      // p points to a[6]
9.     q = q - 2;   // q points to a[3]
10.    i = q - p;    // i = 3 - 6 = -3
11.    i = p - q;    // i = 6 - 3 = 3
12.    if (p <= q)
13.        printf(" less \n");
14.    else
15.        printf(" more \n"); // printed
16.    return 0;
17. }
```



Two-dimensional stack-allocated arrays are just glorified one-dimensional arrays. So, when doing pointer arithmetic with two-dimensional arrays, remember to treat it as a row-major array, and you will be fine.

```
1. #include <stdio.h>
2.
3. int main(void){
4.     int a[3][4] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
5.     int *p, *q, i;
6.     p = &(a[1][2]); // p points to a[1][2]
7.     q = p + 2;      // q points to a[2][0]
8.     p += 4;         // p points to a[2][2]
9.     q = q - 5;      // q points to a[0][3]
10.    i = q - p;       // -7
11.    i = p - q;       // 7
12.    if (p <= q)
13.        printf(" less \n");
14.    else
15.        printf(" more \n"); // printed
16.    return 0;
17. }
```

Let us revisit summing an array and finding the largest using pointer arithmetic.

```

1. #include <stdio.h>
2.
3. int sum(int a[], int n)
4. { // one way to do it
5.     int total = 0;
6.     for (int *p = a; p < a + n; p++) // p is a pointer that points at an
7.                                         // element in a
8.         total += *p;                // *p is the value that p is pointing at
9.     return total;
10. }
11.
12. int altsum(int a[], int n)
13. { //alternative summing
14.     int total = 0;
15.     for (int i = 0; i < n; i++) // i is a valid index in a
16.         total += *(a + i);    // (a+i) points at the element in index i in
17.                                // array a
18.                                // *(a+i) is the value in index i in array a
19.     return total;
20. }
21.
22. int *largest(int a[], int n) // note: the function returns a pointer
23. {
24.     int *m = a;              // m is pointing at the first value in array a
25.     for (int *p = a + 1; p < a + n; p++) // p is a pointer that points at an
26.                                         // element in a
27.     {
28.         if (*p > *m)
29.             m = p; // m points at the largest value in a (first occurrence)
30.     }
31.     return m; // return a pointer
32. }
33.
34. int main(void)
35. {
36.     int a[8] = { 9, 4, 5, 999, 2, 4, 3, 0 };
37.     int size = sizeof(a) / sizeof(a[0]);
38.     printf("%d\n", sum(a, size)); // 1026 is printed
39.     printf("%d\n", altsum(a, size)); // 1026 is printed
40.     printf("%d\n", *largest(a, size)); // 999 is printed
41.     return 0;
42. }

```

### **Attention:**

The \* operator and ++ operator can be combined:

- \*p++ is the same as \*(p++) (Use \*p first then increment pointer).
- (\*p)++ (Use \*p first, then increment \*p, increment the content that p is pointing at).
- ++\*p or \*(++p) (Increment p first then use \*p after increment).
- ++\*p or ++(\*p) (Increment \*p first then use \*p after increment).

More examples with pointers (two versions of the same task, calculation of the total of an array):

```
1. #include <stdio.h>
2.
3. int main(void)
4. {
5.     int a[4] = { 5, 2, 9, 4 };
6.     int sum = 0;
7.     for (int *p = a; p < a + 4; p++)
8.     {
9.         sum += *p;
10.    }
11.    printf("%d\n", sum);
12.    return 0;
13. }
```

```
1. #include <stdio.h>
2.
3. int main(void)
4. {
5.     int a[4] = { 5, 2, 9, 4 };
6.     int sum = 0;
7.     int *p = &a[0];
8.     while (p < &a[4])
9.     {
10.        sum += *p++;
11.    }
12.    printf("%d\n", sum);
13.    return 0;
14. }
```

## Dynamically Allocated Memory

Our memory usage has been on the stack up to this point. However, sometimes, we might want to allocate large chunks of memory or need some dynamically allocated memory.

Heap is great to

- resize arrays when they are full or to
- store global data that is available to the whole program.

**This is where the heap and memory allocation concepts will become necessary.**

### Memory model

Recall discussing the five sections of the memory model at the start of this chapter:

Code
Read-Only Data
Global Data
Heap
Stack

Now, we will focus on the Heap, also known as a "pool" of memory available to a program. The memory is dynamically borrowed/allocated from the Heap. When this allocated memory is no longer needed, we can deallocate it (it can be "returned" to OS and possibly reused). The following is a summary of Heap and Stack for review:

#### Stack

- Scratch space for a thread of execution.
- Each thread gets a stack.
- Elements are ordered (new elements are stacked on older elements).
- Faster since allocating/deallocating memory is very easy.

#### Heap

- Memory set aside for dynamic allocation.
- Typically only one heap for an entire application.
- Entries might be unordered and chaotic.
- Usually slower since need a lookup table for each element (ie. more bookkeeping).

We use `malloc` and `free` from `stdlib.h` library to allocate and deallocate memory from the Heap

#### Syntax:

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

```
// Allocates memory block of size number of bytes but doesn't initialize.
```

```
// Returns a pointer to it.
```

```
// Returns NULL, the null pointer, if insufficient memory or size==0.
```

```
void free(void *p)
```

```
// Frees a memory block that p is pointing at that was allocated by the user (say using malloc).
```

```
// Failure to free your allocated memory is called a memory leak.
```

#### NULL Pointer

Since pointers are memory addresses, we need to be able to distinguish between a pointer to something and a pointer to nothing.

We use the `NULL` pointer to do this. It can be called by

- `int *p = NULL;`
- `int *p = 0;`
- `int *p = (int *) 0;`
- `int *p = (void *) 0;`

The `(void *)` typecast will automatically get converted to the correct type. We will talk about it later on in this course.

The `NULL` pointer is in many libraries, including `<locale.h>`, `<stddef.h>`, `<stdio.h>`, `<stdlib.h>`, `<string.h>`, `<time.h>`, `<wchar.h>` and possibly others.

Example (Next page):

```

1. #include <assert.h>
2. #include <stdio.h>
3. #include <stdlib.h>
4.
5. int *numbers(int n);
6.
7. int main(void)
8. {
9.     int *q = numbers(100); // q is pointing at an array of length 100
10.                                // allocated on the HEAP
11.     printf("%d\n", q[50]); // 50 is printed
12.     free(q);               // Avoid memory leak
13.     return 0;
14. }
15.
16. int *numbers(int n)
17. {
18.     int *p = malloc(n * sizeof(int)); // allocate enough space for n integers
19.     assert(p);                        // Verify that malloc succeeded (p is not NULL)
20.     for (int i = 0; i < n; i++) // assign the values 0-99 to the array
21.         p[i] = i;
22.     return p;                      // returns a pointer to the beginning of the array
23. }

```



What is wrong with the code<sup>ii</sup> below?

```

1. int *my_array;
2. my_array = malloc(10*sizeof(int));
3. my_array = malloc(10*sizeof(int));

```

### More functions (allocators):

```
void* calloc(size_t nmemb, size_t size)
```

// Clear allocate.

// Allocates nmemb elements of size bytes, each initialized to 0

```
void* realloc (void *p, size_t size)
```

// Resizes a previously allocated block

// We may need to create a new block and copy over the old block contents.



Again, we need `<stdlib.h>` to use these.

Typically, `malloc` is used unless you have a good reason to do otherwise.

Demo about `realloc`:

```
1. #include <stdio.h>
2. #include <stdlib.h>
3.
4. int main()
5. {
6.     char *str;
7.     str = (char *) malloc(15);
8.     strcpy(str, "CS137 is FUN!!");
9.     printf("String = %s Address = %u\n", str, str);
10.
11.     str = (char *) realloc(str, 25);
12.     strcat(str, " Indead!!");
13.     printf("String = %s Address = %u\n", str, str);
14.
15.     str = (char *) realloc(str, 6);
16.     str[5] = '\0';
17.     printf("String = %s Address = %u\n", str, str);
18.
19.     free(str);
20.     return (0);
21. }
```

#### Console program output

```
String = CS137 is FUN!! Address = 19923848
String = CS137 is FUN!! Indead!! Address = 19923848
String = CS137 Address = 19923848
Press any key to continue...
```

Let's revisit our time-of-day structure example:

```
1. struct tod {
2.     int hours;
3.     int minutes;
4. };
```

To create a pointer to the structure, we can use:

```
struct tod *t = malloc(sizeof(struct tod));
```

Now, `t` points to the beginning of a struct where the integers `hours` and `minutes` are located.

We can modify these values by using `(*t).hours = 18;` or `t->hours = 18;`



Arrow operator can be overloaded (say in C++), whereas the dot cannot (You will learn about that in future courses). Brackets are necessary above because the dot has precedence. Arrow is left-associative (like addition, multiplication, etc.).

Do you recall the following example?

```
1. #include <stdio.h>
2.
3. struct tod {
4.     int hours;
5.     int minutes;
6. };
7.
8. void todPrint(struct tod when)
9. {
10.     printf(" %0.2d :%0.2d\n", when.hours, when.minutes);
11. }
12.
13. int main(void)
14. {
15.     struct tod now = { 16, 50 };
16.     struct tod later = {.hours = 18 };
17.     printf("now: ");
18.     todPrint(now);
19.     printf("later: ");
20.     todPrint(later);
21.     later.minutes = 1;
22.     printf("updated later: ");
23.     todPrint(later);
24.     return 0;
25. }
```

All values were stored in the STACK.



The following version uses the HEAP to allocate memory and free it later to avoid memory leaks.

```
1. #include <stdio.h>
2. #include <stdlib.h>
3.
4. struct tod {
5.     int hours;
6.     int minutes;
7. };
8.
9. void todPrint(struct tod *when){
10.     printf(" %0.2d :%0.2d\n", (*when).hours, (*when).minutes);
11. }
12.
13. int main(void){
14.     struct tod *now = malloc(sizeof(struct tod));
15.     (*now).hours = 16;
16.     (*now).minutes = 50;
17.     struct tod *later = malloc(sizeof(struct tod));
18.     (*later).hours = 18;
19.     printf("now: ");    todPrint(now);
20.     printf("later: "); todPrint(later);
21.     (*later).minutes = 1;
22.     printf("updated later: ");    todPrint(later);
23.     free(now);
24.     free(later);
25.     return 0;
26. }
```

In the time-of-day example, the sizes of all the elements were fixed. What happens if, for example, you want a struct with an array whose size is to be determined later? There are ways to handle this, but it must be done carefully. This is valid only in C99 and beyond. This technique is called the "struct hack".

Let us review the following program. (Pay close attention to the provided comments)

```

1. #include <assert.h>
2. #include <stdio.h>
3. #include <stdlib.h>
4.
5. struct flex_array {
6.     int length;
7.     int a[];          // Note : declared at end of struct definition
8. };
9.
10. int main(void) {
11.     printf("%zu\n", sizeof(struct flex_array)); // 4 is printed (size of an int.
12.                                                // a has size zero for now)
13.
14.     size_t array_size = 10;
15.     struct flex_array *fa = malloc(sizeof(struct flex_array) + array_size *
        sizeof(int));
16.     assert(fa);
17.     printf("%zu\n", sizeof(struct flex_array)); // 4 is printed
18.     printf("%zu\n", sizeof(fa)); // 8 is printed (size of a pointer)
19.     printf("%zu\n", sizeof(*fa)); // 4 is printed (size of the first integer that
        fa is pointing at)
20.     fa->length = array_size;
21.     for (int i = 0; i < fa->length; i++)
22.         fa->a[i] = i*i;
23.     printf("%d\n", fa->a[4]); // 16 is printed
24.     free(fa);
25.     return 0;
26. }

```



In `<stdlib.h>`, a data type `size_t` should be used when using `malloc`.

## Vectors (Variable Size Array)

Arrays have a fixed size. Can an array be created that expands as more terms are needed during the program's execution? There is a library in C++ that does this, the vector library, but not in C. We'll create a simplified instance of this to demonstrate how it works for a vector of integers.

**Idea:** Initialize contents to 0 and grow automatically by powers of 2.

Interface file: `vector.h`

```
1. #ifndef VECTOR_H
2. #define VECTOR_H
3.
4. struct vector ;
5.
6. //will create a new empty vector.
7. struct vector *vectorCreate(void);
8.
9. //deletes the vector *v. Returns NULL on success.
10. //(return NULL to allow for v=vectorDelete(v);)
11. struct vector *vectorDelete(struct vector *v);
12.
13. //adds val to the end of the vector. Allocates new space as necessary.
14. void vectorAdd(struct vector *v, int val);
15.
16. //sets value in index ind to be val.
17. void vectorSet(struct vector *v, int ind , int val);
18.
19. //returns element at index ind.
20. int vectorGet(struct vector *v, int ind);
21.
22. //returns the length of the vector *v.
23. int vectorLength(struct vector *v);
24.
25. #endif
```

```

1. #include "vector.h"
2. #include <assert.h>
3. #include <stdlib.h>
4.
5. struct vector {           // vector is a structure with three elements
6.     int *arr;             // pointer to array
7.     int size, length;     // size is the total storage.
8.     //length is the actual used storage
9. };
10.
11. struct vector *vectorCreate(void)
12. {
13.     // malloc returns NULL if it was not completed successfully
14.     struct vector *v = malloc(sizeof(struct vector));
15.     assert(v);           // to check that malloc was completed successfully
16.     v->size = 4;         // the first created vector is of size 4
17.     v->arr = malloc(4 * sizeof(int));
18.     assert(v->arr);
19.     v->length = 0;      // no values in the vector yet.
20.     return v;
21. }
22.
23. struct vector *vectorDelete(struct vector *v)
24. {
25.     if (v)
26.     {
27.         free(v->arr);    // free the inside array first
28.         free(v);        // then free the vector
29.     }
30.     return NULL;
31. }
32.
33. void vectorAdd(struct vector *v, int value)
34. {
35.     assert(v);
36.     if (v->length == v->size)
37.     {
38.         // if arr is full
39.         int newSize = v->size * 2;    //double the size
40.         // allocate the new size of array
41.         int *newArr = malloc(newSize * sizeof(int));
42.         for (int i = 0; i < v->size; ++i)
43.         {
44.             //copy the data from the old storage to the new storage
45.             newArr[i] = v->arr[i];
46.         }
47.         newArr[v->size] = value;
48.         free(v->arr);    // free the old array
49.         v->size = newSize;
50.         v->arr = newArr;    // make arr point to the new array
51.     }
52.     else
53.     {
54.         // if arr is not full
55.         v->arr[v->length] = value;    // add the new value
56.     }
57.     ++v->length;    // update length

```

```

58.
59. void vectorSet(struct vector *v, int ind, int value)
60. {
61.     assert(v && ind >= 0 && ind <= v->length);
62.     //update element in index ind with the new value
63.     v->arr[ind] = value;
64. }
65.
66. int vectorGet(struct vector *v, int ind)
67. {
68.     assert(v && ind >= 0 && ind < v->length);
69.     return v->arr[ind];    // get the value in index ind
70. }
71.
72. int vectorLength(struct vector *v)
73. {    //return the number of elements in vector
74.     assert(v);
75.     return v->length;
76. }
77.

```



- Notice how none of the implementation details were in our header file, only the declarations. This is a design principle known as information hiding. We do this to hide implementation details from the user yet keep the user interaction/interface the same. Thus, we can modify the internal code and not affect other people using our code externally.
- Notice that struct vector v is not possible with this header, whereas struct vector \*v is possible because the header doesn't know the struct size since it is implemented in the .c implementation file.

### Sample program:

```
1. #include <stdio.h>
2. #include "vector.h"
3.
4. int main(void)
5. {
6.     struct vector *v = vectorCreate();
7.     for (int i = 0; i < 20; ++i)
8.     {
9.         vectorAdd(v, i);
10.    }
11.    printf("%d\n\n", vectorLength(v));
12.    for (int i = 0; i < 20; ++i)
13.    {
14.        printf("v[%d]=%d  ", i, vectorGet(v, i));
15.    }
16.    printf("\n\n");
17.    for (int i = 0; i < 20; ++i)
18.    {
19.        vectorSet(v, i, i * i);
20.        printf("v[%d]=%d  ", i, vectorGet(v, i));
21.    }
22.    printf("\n\n");
23.    v = vectorDelete(v);
24.    if (v==NULL) printf("Success!\n");
25.    else printf("Freeing was not completed successfully!");
26.    return 0;
27. }
```

### Output:

20

```
v[0]=0 v[1]=1 v[2]=2 v[3]=3 v[4]=4 v[5]=5 v[6]=6 v[7]=7 v[8]=8 v[9]=9
v[10]=10 v[11]=11 v[12]=12 v[13]=13 v[14]=14 v[15]=15 v[16]=16 v[17]=17
v[18]=18 v[19]=19
```

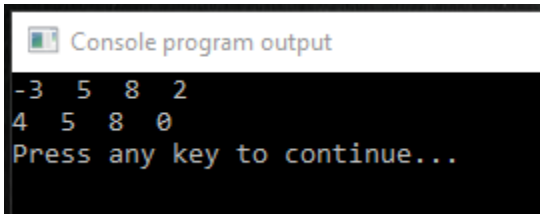
```
v[0]=0 v[1]=1 v[2]=4 v[3]=9 v[4]=16 v[5]=25 v[6]=36 v[7]=49 v[8]=64
v[9]=81 v[10]=100 v[11]=121 v[12]=144 v[13]=169 v[14]=196 v[15]=225
v[16]=256 v[17]=289 v[18]=324 v[19]=361
```

Success!

## Additional Examples

```
/* What is the output of the following program? */

#include <stdio.h>
void mysterious(int *a, int *b, int *c)
{
    *a = *c;
    *b = *b + *a;
    *c = *a - *b;
}
int main()
{
    int w = 5;
    int x = 1;
    int y = 3;
    int z = 2;
    mysterious(&x, &y, &w);
    printf("%d %d %d %d \n", w, x, y, z);
    mysterious(&w, &w, &z);
    printf("%d %d %d %d \n", w, x, y, z);
    return 0;
}
```



```
Console program output
-3 5 8 2
4 5 8 0
Press any key to continue...
```

## Extra Practice Problems

1) Write a function `int read_and_range(int *max, int *min, int *count_max, int *count_min)` that reads integers until a failure occurs and returns the number of integers successfully read. If no integers were successfully read in, the functions return zero and do not modify (mutate) the parameters' contents. Otherwise, the functions mutate the contents of their (pointer) parameters as follows:

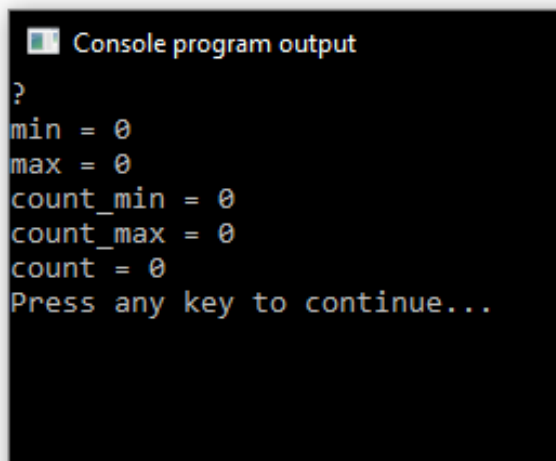
- `*max`: the largest number that was successfully read
- `*min`: the smallest number that was successfully read
- `*count_max`: the number of times the largest number was read
- `*count_min`: the number of times the smallest number was read

Note: You may not use arrays or structures for this question.

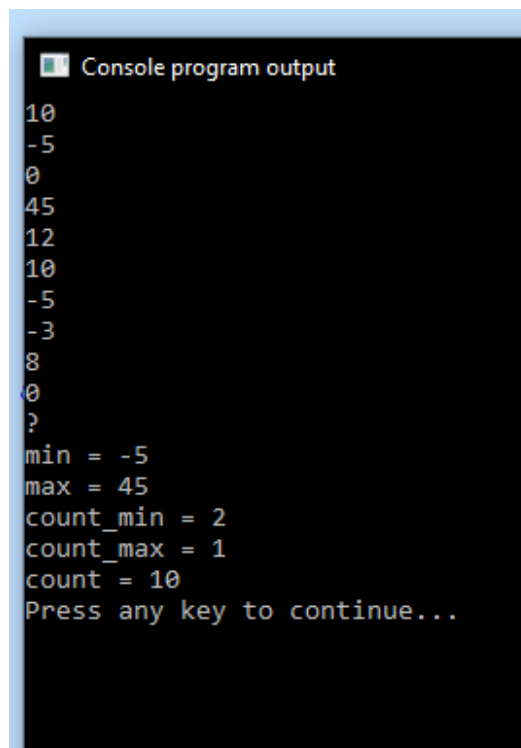
Advice: Don't look at my provided solution<sup>iii</sup> before you attempt solving this question on your own.

Here is a sample main function to test your solution

```
1. int main(void) {
2.     int min = 0;
3.     int max = 0;
4.     int count_min = 0;
5.     int count_max = 0;
6.     int count = read_and_range(&max, &min, &count_max, &count_min);
7.     printf("min = %d\n", min);
8.     printf("max = %d\n", max);
9.     printf("count_min = %d\n", count_min);
10.    printf("count_max = %d\n", count_max);
11.    printf("count = %d\n", count);
12. }
```



```
Console program output
?
min = 0
max = 0
count_min = 0
count_max = 0
count = 0
Press any key to continue...
```



```
Console program output
10
-5
0
45
12
10
-5
-3
8
0
?
min = -5
max = 45
count_min = 2
count_max = 1
count = 10
Press any key to continue...
```



2) What is the output of the following program?

Advice: Don't look at my provided solution<sup>iv</sup> before you attempt solving this question on your own.

```
1. #include <stdio.h>
2. void mysterious(int *a, int *b, int *c)
3. {
4.     *a = *c;
5.     *b = *b + *a;
6.     *c = *a - *b;
7. }
8. int main()
9. {
10.     int w = 5;
11.     int x = 1;
12.     int y = 3;
13.     int z = 2;
14.     mysterious(&x, &y, &w);
15.     printf("%d %d %d %d\n", w, x, y, z);
16.     mysterious(&w, &w, &z);
17.     printf("%d %d %d %d\n", w, x, y, z);
18.     return 0;
19. }
```

3) Define the function `int ** outerproduct(int a[], int m, int b[], int n)`

Where `a` is an array with `m` elements, `b` is an array with `n` elements. Return a heap-allocated `m×n` matrix that computes the outer product matrix for `a` and `b`. That is, if `i` is from `1, ..., m` and `j` is from `1, ..., n`, then the outer product matrix `c` satisfies `c[i][j]=a[i]*b[j]`.

Note: `int **` is a pointer to pointer of integer

Here is a sample `main` function to test your solution

```
1. #include <stdio.h>
2. #include <assert.h>
3. #include <stdlib.h>
4.
5. int main()
6. {
7.
8.     int m = 5, n = 3;
9.     int a[] = { 1, 2, 3, 4, 5 };
10.    int b[] = { 3, 2, 1 };
11.
12.    int **c = outerproduct(a, m, b, n);
13.
14.    assert(c);
15.    assert(c[0][0] == 3);
16.    assert(c[0][1] == 2);
17.    assert(c[2][2] == 3);
18.    assert(c[4][0] == 15);
19.
20.    int i;
21.    for (i = 0; i < m; i++)
22.    {
23.        free(c[i]);
24.    }
25.    free(c);
26.    return 0;
27. }
```

Advice: Don't look at my provided solution<sup>V</sup> before you attempt solving this question on your own.

4) We can use compression algorithms to save space when storing large amounts of data. Generally, we distinguish between lossless compression (e.g., gzip's DEFLATE) and lossy compression (e.g., JPG's DCT). We want to store a series of integers in this question so that a lossless compression algorithm might be our best choice.

A run-length encoding is a well-suited algorithm when data (i.e., single number) are repeated frequently. Instead of storing each number separately, run-length encoding stores a tuple of a single digit and a single number: the digit indicates how many times the following number is printed. For example, the series numbers 1 1 1 3 2 1 3 2 1 1 would be encoded as 3 1 1 3 1 2 1 1 3 1 2 2 1. Shorter examples would be

3 -> 1 3,

2 2 2 -> 3 2,

and 2 9 9 2 -> 1 2 2 9 1 2.

Given a run-length encoded numbers, it is possible to recreate the original sequence of numbers through decoding: 3 1 1 3 1 2 1 1 3 1 2 2 1 would turn into 1 1 1 3 2 1 3 2 1 1, i.e., the original numbers. (Generally, `decode(encode(data)) -> data` for all lossless compression algorithms.)

Write a function `int *unsay(const int *src, int src_len, int *dst_len)`, which returns a **decoding** array containing the sequence specified by `src`, modifying `*dst_len` to be the length of the new sequence.

Requires: `src` must be a valid array of length `src_len`;

`src_len > 1`

Sample input: 3 1

Sample output: 1 1 1

Explanation: The sample input contains 3 ones, so we transfer it into 1 1 1;

```
int main() {
    int src[] = {3, 1, 1, 3, 1, 2, 1, 1, 1, 3, 1, 2, 2, 1};
    int length = 14;
    int dst_len;
    int *ans = unsay(src, length, &dst_len);
    for (int i = 0; i < dst_len; i++) {
        printf("%d ", ans[i]);
        if (i == dst_len - 1) printf("\n");
    }
    free(ans);
}
```

5) Develop a C program, `market_simulation.c`, to simulate an economic market scenario where producers and consumers adjust their behavior based on market price. This simulation will use structs and pointers to represent entities and their interactions.

You are tasked with creating a simulation of a fundamental economic market. In this market, some producers produce goods, and consumers use them. The laws of supply and demand influence the market's price: when demand is higher than supply, prices go up, and when supply exceeds demand, prices go down. Both producers and consumers adjust their production and consumption levels based on the current market price.

Structs to be Defined:

Market: Holds the current market price.

- Fields: `double price`.

Producer: Represents a producer in the market.

- Fields: `int base_production`, `double current_production`, `Market *market`.

Consumer: Represents a consumer in the market.

- Fields: `int base_consumption`, `double current_consumption`, `Market *market`.

Functions to Implement:

- `adjustProducer`: Updates a producer's current production based on the market price.  
Signature: `void adjustProducer(Producer *producer)`
- `adjustConsumer`: Updates a consumer's current consumption based on the market price.  
Signature: `void adjustConsumer(Consumer *consumer)`
- `adjustMarketPrice`: Adjusts the market price based on the total supply and demand from all producers and consumers.  
Signature: `void adjustMarketPrice(Market *market, Producer producers[], Consumer consumers[])`

Simulation Logic:

The market starts with an initial price.

Each producer and consumer begins with a base production and consumption level.

In each simulation round, producers and consumers adjust their production and consumption according to the current market price.

After each round, the market price is adjusted based on the total supply (sum of all producers' production) and total demand (sum of all consumers' consumption).

Equations:

- Producer's Production Adjustment:  $\text{current\_production} = \text{base\_production} * (\text{market\_price} / \text{base\_price})$
- Consumer's Consumption Adjustment:  $\text{current\_consumption} = \text{base\_consumption} * (\text{base\_price} / \text{market\_price})$
- Market Price Adjustment:  $\text{new\_price} = \text{current\_price} + (\text{total\_demand} - \text{total\_supply}) * \text{price\_adjustment\_factor}$

Constraints:

- Use a fixed number of producers and consumers (e.g., 3 each).
- The `price_adjustment_factor` is a constant to moderate the rate of price changes (e.g., 0.05).
- The simulation should run for a fixed number of rounds (e.g., 5 rounds).

```
int main() {
    Market market = {30}; // Initial market price is $30
    Producer producers[NUM_AGENTS];
    Consumer consumers[NUM_AGENTS];

    // Initialize producers and consumers
    for (int i = 0; i < NUM_AGENTS; i++) {
        producers[i].base_production = 100; // Base production of 100 units
    }
}
```

```

    producers[i].market = &market;

    consumers[i].base_consumption = 10; // Base consumption of 10 units
    consumers[i].market = &market;
}

// Simulation loop for 5 rounds
for (int round = 1; round <= 5; round++){
    printf("Round %d:\n", round);
    printf("Market Price: $%.2f\n", market.price);
    for (int i = 0; i < NUM_AGENTS; i++){
        adjustProducer(&producers[i]);
        adjustConsumer(&consumers[i]);
    }
    adjustMarketPrice(&market, producers, consumers);
    printf("Total Production: %.2f units\n", producers[0].current_production *
NUM_AGENTS);
    printf("Total Consumption: %.2f units\n\n", consumers[0].current_consumption *
NUM_AGENTS);
}

    return 0;
}

```

Output:

Round 1:

Market Price: \$30.00

Total Production: 90.00 units

Total Consumption: 10.00 units

Round 2:

Market Price: \$26.00

Total Production: 78.00 units

Total Consumption: 11.54 units

Round 3:

Market Price: \$22.68

Total Production: 68.03 units

Total Consumption: 13.23 units

Round 4:

Market Price: \$19.94

Total Production: 59.81 units

Total Consumption: 15.05 units

Round 5:

Market Price: \$17.70

Total Production: 53.10 units

Total Consumption: 16.95 units

6) Write a C program, `particle_movement.c`, to simulate the movement of particles in a two-dimensional space, allowing the user to input the velocity for each particle. The program will use pointers for dynamic memory management of particle data.

#### Problem Description:

Develop a simulation program where each particle's position is updated based on its velocity in a two-dimensional space. Users should input the initial position and velocity for each particle. The program tracks and displays the position of these particles over time.

#### Structs to be Defined:

Particle: Represents a particle in space.

- Fields: `float x_position, float y_position, float x_velocity, float y_velocity`.

#### Functions to Implement:

- `createParticle`: Dynamically allocates and initializes a particle.  
Signature: `Particle* createParticle(float x_pos, float y_pos, float x_vel, float y_vel)`
- `updateParticle`: Updates the position of a particle based on its velocity.  
Signature: `void updateParticle(Particle *particle)`
- `displayParticle`: Prints the current position of a particle.  
Signature: `void displayParticle(const Particle *particle)`
- `destroyParticle`: Frees the memory allocated for a particle.  
Signature: `void destroyParticle(Particle *particle)`

#### User Input:

The user inputs the number of particles.

The user inputs the initial position (`x_position, y_position`) and velocity (`x_velocity, y_velocity`) for each particle.

#### Simulation Logic:

Initialize the particles based on user input.

For a predefined number of time steps, update and display the position of each particle.

At the end of the simulation, the memory allocated for the particles is freed.

#### Constraints:

Use dynamic memory allocation for creating particles.

The simulation runs for a fixed number of time steps (e.g., 10 time steps).

```
int main() {
    int numParticles, steps = 10;
    printf("Enter the number of particles: ");
    scanf("%d", &numParticles);

    Particle **particles = (Particle **)malloc(numParticles * sizeof(Particle *));
    if (particles == NULL) {
        fprintf(stderr, "Error allocating memory\n");
        return 1;
    }
    for (int i = 0; i < numParticles; i++) {
        float x_pos, y_pos, x_vel, y_vel;
        printf("Enter x_position, y_position, x_velocity, y_velocity for particle %d: ", i
+ 1);
        scanf("%f %f %f %f", &x_pos, &y_pos, &x_vel, &y_vel);
```

```

        particles[i] = createParticle(x_pos, y_pos, x_vel, y_vel);
    }
    for (int t = 0; t < steps; t++) {
        printf("Time step %d:\n", t + 1);
        for (int i = 0; i < numParticles; i++){
            updateParticle(particles[i]);
            displayParticle(particles[i]);
        }
        printf("\n");
    }
    for (int i = 0; i < numParticles; i++){
        destroyParticle(particles[i]);
    }
    free(particles);

    return 0;
}

```

Enter the number of particles: 1

Enter x\_position, y\_position, x\_velocity, y\_velocity for particle 1: 1 1 10 0

Time step 1:

Particle Position: (11.00, 1.00)

Time step 2:

Particle Position: (21.00, 1.00)

Time step 3:

Particle Position: (31.00, 1.00)

Time step 4:

Particle Position: (41.00, 1.00)

Time step 5:

Particle Position: (51.00, 1.00)

Time step 6:

Particle Position: (61.00, 1.00)

Time step 7:

Particle Position: (71.00, 1.00)

Time step 8:

Particle Position: (81.00, 1.00)

Time step 9:

Particle Position: (91.00, 1.00)

Time step 10:

Particle Position: (101.00, 1.00)

## Answers

---

i a pointer to the first occurrence of the largest element in a given array

ii The last line will cause memory leak because the pointer `my_array` is now pointing to a new block of memory, we lost access to the first block of memory (that we allocated via the second line of code) and we can't free it (the first allocated block of memory) anymore => memory leak!

iii

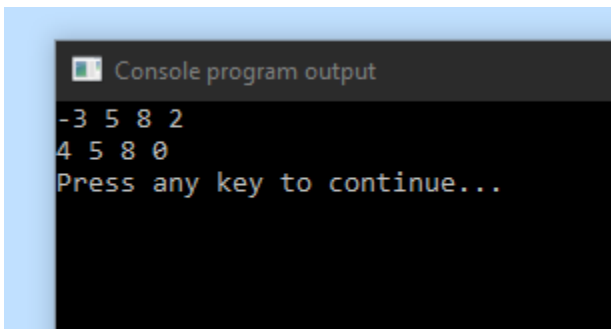
```
#include <stdio.h>

int read_and_range(int *max, int *min, int *count_max, int *count_min)
{
    int n;
    if (!scanf("%d", &n))
    {
        return 0;
    }
    int count = 1;
    *max = n;
    *min = n;
    *count_max = 1;
    *count_min = 1;

    while (1)
    {
        if (!scanf("%d", &n))
        {
            break;
        }
        ++count;
        if (n == *max)
        {
            ++(*count_max);
        }
        if (n == *min)
        {
            ++(*count_min);
        }
        if (n > *max)
        {
            *max = n;
            *count_max = 1;
        }
        if (n < *min)
        {
            *min = n;
            *count_min = 1;
        }
    }
    return count;
}
```



iv



```
Console program output
-3 5 8 2
4 5 8 0
Press any key to continue...
```

v

```
int **outerproduct(int a[], int b[], int m, int n)
{
    int i, j;
    int **result = malloc(sizeof(int *) * m);

    for (i = 0; i < m; i++)
    {
        result[i] = malloc(sizeof(int) * n);
    }

    for (i = 0; i < m; i++)
    {
        for (j = 0; j < n; j++)
        {
            result[i][j] = a[i] * b[j];
        }
    }
    return result;
}
```