# CS 137

Programming Principles

Chapter 9

# Pointers, Memory Allocation, Arrays, and Vectors

*Victoria Sakhnini*

## Table of Contents

## 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:

```
#include <stdio.h>

void swap(int a, int b) {
    printf("a=%d, b=%d\n", a, b);
    int tmp;
    tmp = a;
    a = b;
    b = tmp;
    printf("a=%d, b=%d\n", a, b);
}

int main(void) {
    int x = 10;
    int y = -15;
    printf("x=%d, y=%d\n", x, y);
    swap(x, y);
    printf("x=%d, y=%d\n", x, y);
    return 0;
}
```

**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. How can we make the changes in swap affect x and y in main?

We can do this with **pointers and references**. Consider the following example carefully, paying attention to the comments:

```
#include <stdio.h>

int main(void)
{
    int i = 6;
    int *p;
    p = &i;    // "p has the address of i" or "p is pointing at i"
    *p = 10;   // p points at i; the value p is pointing at is changed to 10,
               // thus i is changed to 10 as well
    printf("%d \n", i);  // 10 is printed
    int *q;
    q = p;     // q is pointing where p is pointing at, which is i
    *q = 17;  // q and p both point at i;
              // the value q is pointing at is changed to 17,
              // thus i is changed to 17 as well
    printf("%d \n", i);  // 17 is printed
    return 0;
}
```

int *p is a pointer to an integer; similarly double *d is a pointer to a double. &i is the address of variable i — different from the value stored in i. *p is the dereferencing of p: it is the value stored where p points, and can be used to modify that value.

## Sections of Memory

In this course, we model five sections of memory:

- **Code** — stores program instructions in machine code, populated during compilation.
- **Read-Only Data** — stores global constants.
- **Global Data** — stores global variables, available throughout the entire execution of the program.
- **Heap** — used to allocate memory dynamically (discussed later).
- **Stack** — stores local variables and return addresses to manage function calls. Each function call creates a stack frame. The stack grows toward lower addresses.

Each stack frame contains: 1) the argument values 2) all local variables (both mutable variables and constants) within the function block 3) the return address — the location to jump back to when the function returns. As program flow jumps from function to function, the "history" of return addresses is the call stack. Each function call pushes a new entry; each return pops one off. The "last called" is the "first returned".

### Corrected swap — Using Pointers

What is the output of the following program? Figure it out before reading on.

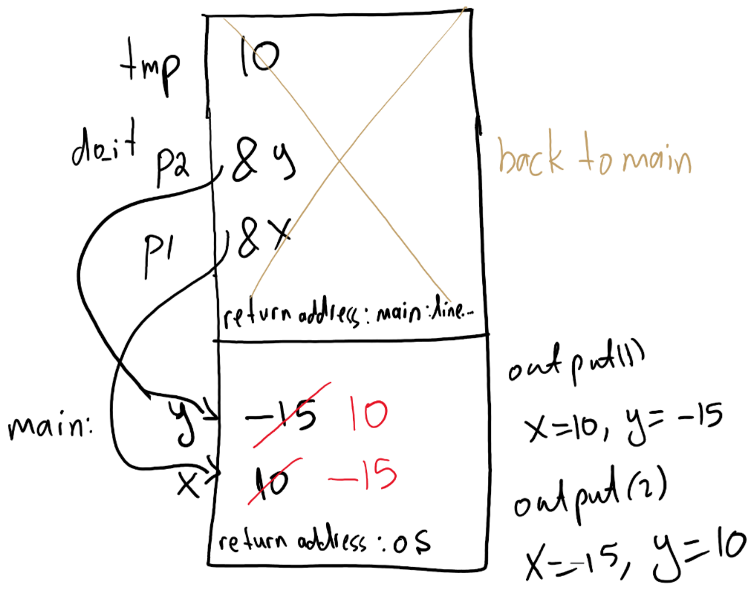
```
#include <stdio.h>

void swap(int *p1, int *p2)
{
    int tmp;
    tmp  = *p1;
    *p1  = *p2;
    *p2  = tmp;
}

int main(void)
{
    int x = 10;
    int y = -15;
    printf("x=%d, y=%d\n", x, y);
    swap(&x, &y);    // Why &x and &y instead of x, y?
    printf("x=%d, y=%d\n", x, y);
    return 0;
}
```

❓ Why did we pass &x and &y as arguments when we called swap?

Because swap expects two memory addresses to be assigned to its pointer parameters. Passing &x and &y gives swap the addresses of x and y so it can modify their values directly.



## Bitwise XOR Swap — Just for Fun

What is the output of the following program? Trace manually before running. Reminder: ^ is a bitwise XOR operation.

```
#include <stdio.h>

int main(void)
{
    int x = 10;
    int y = -15;
    printf("x=%d, y=%d\n", x, y);
    x ^= y;
    y ^= x;
    x ^= y;
    printf("x=%d, y=%d\n", x, y);
    return 0;
}
```

### Exercise — calc_array

What is the output of the following program? What does the function return?

```c
#include <stdio.h>

int calc_array(int a[], int n) {
    int val  = a[0];
    int *res = &a[0];
    for (int i = 1; i < n; i++)
        if (a[i] > val)
        {
            val = a[i];
            res = &a[i];
        }
    return *res;
}

int main(void) {
    int a[10] = { 1, 10, 67, 876, -76, 0, -45, 8, 9, 1 };
    int ans;
    ans = calc_array(a, 10);
    printf("The result is %d\n", ans);
    return 0;
}
```

The function returns the value of the largest element in the array (876). More specifically, it returns *res, which is the value at the address of the first occurrence of the maximum element.

## Pointer Arithmetic and Arrays

C allows an integer to be added to a pointer. The result depends on the pointer's type: `(p + 1)` advances the pointer by `sizeof` whatever p points at. You **cannot** add two pointers.

A pointer q can be **subtracted** from another pointer p if both point to the same type. Pointers can be compared with <, <=, ==, !=, >=, >.

### Example — 1D Pointer Arithmetic

```c
#include <stdio.h>

int main(void) {
    int a[8] = { 2, 3, 4, 5, 6, 7, 8, 9 };
    int *p, *q, i;
    p = &(a[2]);  // p points to a[2]
    q = p + 3;    // q points to a[5]
    p += 4;       // p points to a[6]
    q = q - 2;    // q points to a[3]
    i = q - p;    // i = 3 - 6 = -3
    i = p - q;   // i = 6 - 3 = 3
    if (p <= q)
        printf(" less \n");
    else
        printf(" more \n");  // printed
    return 0;
}
```

Two-dimensional stack-allocated arrays are just glorified one-dimensional arrays. When doing pointer arithmetic with 2D arrays, treat it as a row-major array and you will be fine.

### Example — 2D Pointer Arithmetic

```
#include <stdio.h>

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

### Summing and Finding Largest with Pointer Arithmetic

```
#include <stdio.h>

int sum(int a[], int n)
{   // one way
    int total = 0;
    for (int *p = a; p < a + n; p++)  // p points at elements of a
        total += *p;                   // *p is the value p is pointing at
    return total;
}

int altsum(int a[], int n)
{   // alternative
    int total = 0;
    for (int i = 0; i < n; i++)
        total += *(a + i);  // (a+i) points at index i; *(a+i) is its value
    return total;
}

int *largest(int a[], int n)  // returns a pointer
{
    int *m = a;  // m points at the first element
    for (int *p = a + 1; p < a + n; p++)
    {
        if (*p > *m)
            m = p;  // m points at the largest value (first occurrence)
    }
    return m;
}

int main(void)
{
    int a[8] = { 9, 4, 5, 999, 2, 4, 3, 0 };
    int size = sizeof(a) / sizeof(a[0]);
    printf("%d\n", sum(a, size));      // 1026
    printf("%d\n", altsum(a, size));   // 1026
    printf("%d\n", *largest(a, size)); // 999
    return 0;
}
```

The * and ++ operators can be combined in several ways: *p++ → same as *(p++): use *p first, then increment the pointer. (*p)++ → use *p first, then increment the value p points at. *++p → same as *(++p): increment p first, then use *p. ++*p → same as ++(*p): increment the value p points at first, then use it.

### Two Versions of Array Sum Using *p++

```
#include <stdio.h>

int main(void)
{
    int a[4] = { 5, 2, 9, 4 };
    int sum = 0;
    for (int *p = a; p < a + 4; p++)
    {
        sum += *p;
    }
    printf("%d\n", sum);
    return 0;
}
```

```
#include <stdio.h>

int main(void)
{
    int a[4] = { 5, 2, 9, 4 };
    int sum = 0;
    int *p = &a[0];
    while (p < &a[4])
    {
        sum += *p++;  // use *p then increment p
    }
    printf("%d\n", sum);
    return 0;
}
```

## Dynamically Allocated Memory

Our memory usage so far has been on the stack. Sometimes we need large chunks of memory or dynamically allocated memory.

**The Heap is useful for:**

- Resizing arrays when they are full.
- Storing global data available to the whole program.

### Memory Model

Recall the five sections of memory. We now focus on the **Heap** — a pool of memory available to the program. Memory is dynamically borrowed from the Heap and can be deallocated (returned to the OS and possibly reused) when no longer needed.

| | Stack | Heap |
|---|---|---|
| Size | Fixed (determined at compile time) | Dynamic (grows at runtime) |
| Management | Automatic (by compiler) | Manual (programmer calls malloc/free) |
| Lifetime | Until function returns | Until explicitly freed |
| Speed | Fast | Slightly slower |
| Risk | Stack overflow | Memory leaks, dangling pointers |

We use `malloc` and `free` from `<stdlib.h>` to allocate and deallocate heap memory.

```
void *malloc(size_t size);
// Allocates a block of 'size' bytes; does NOT initialize the memory.
// Returns a pointer to the block.
// Returns NULL if insufficient memory or size == 0.

void free(void *p);
// Frees a block allocated by malloc (or calloc/realloc).
// Failure to free allocated memory is called a memory leak.
```

## NULL Pointer

Since pointers are memory addresses, we need to distinguish between a pointer to something and a pointer to nothing. We use the **NULL pointer** for this.

```
int *p = NULL;
int *p = 0;
int *p = (int *) 0;
int *p = (void *) 0;  // void* is automatically converted to the correct type
```

NULL is defined in many standard headers, including `<stdio.h>`, `<stdlib.h>`, `<string.h>`, and others.

### Example — malloc and free

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

int *numbers(int n);

int main(void)
{
    int *q = numbers(100);  // q points at a heap-allocated array of length
100
    printf("%d\n", q[50]); // 50 is printed
    free(q);                // avoid memory leak
    return 0;
}

int *numbers(int n)
{
    int *p = malloc(n * sizeof(int));  // allocate space for n integers
    assert(p);                          // verify malloc succeeded (p != NULL)
    for (int i = 0; i < n; i++)        // assign values 0-(n-1)
        p[i] = i;
    return p;  // return pointer to the beginning of the array
}
```

**Tricky:** What is wrong with the following code?

```
int *my_array;
my_array = malloc(10 * sizeof(int));
my_array = malloc(10 * sizeof(int));  // PROBLEM!
```

The second malloc overwrites my_array, so the pointer to the first allocated block is lost. We can never free that first block — this is a memory leak.

### More Allocators

```
void *calloc(size_t nmemb, size_t size);
// Clear-allocate: allocates nmemb elements of size bytes each,
// all initialized to 0.

void *realloc(void *p, size_t size);
// Resizes a previously allocated block.
// May create a new block and copy over old contents.
```

Typically, malloc is used unless there is a specific reason to use calloc or realloc. All three require <stdlib.h>.

### Demo — realloc

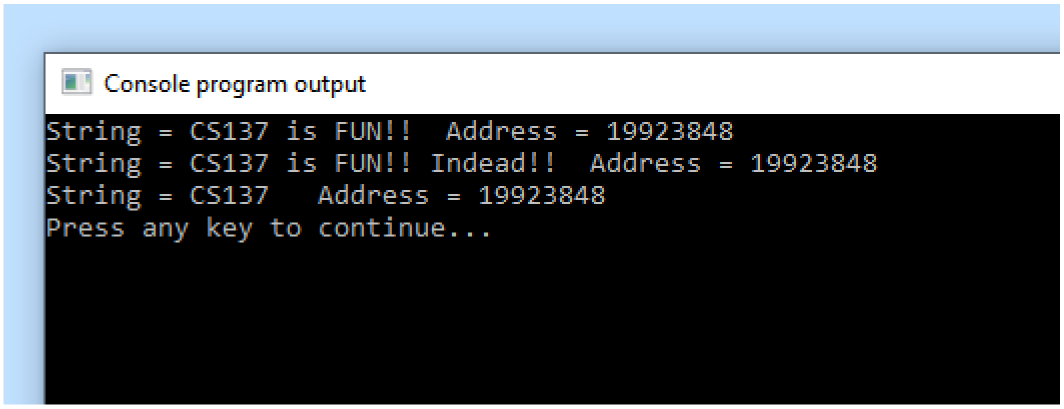
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
    char *str;
    str = (char *) malloc(15);
    strcpy(str, "CS137 is FUN!!");
    printf("String = %s  Address = %u\n", str, str);

    str = (char *) realloc(str, 25);
    strcat(str, " Indeed!!");
    printf("String = %s  Address = %u\n", str, str);

    str = (char *) realloc(str, 6);
    str[5] = '\0';
    printf("String = %s  Address = %u\n", str, str);

    free(str);
    return (0);
}
```


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...
```

## Pointers to Structures & Struct Hack

Revisiting our time-of-day structure:

```
struct tod {
    int hours;
    int minutes;
};
```

To create a pointer to a structure on the heap:

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

Now `t` points to the beginning of a struct. We can modify members using either notation:

```
(*t).hours = 18;    // dereference then access member
t->hours = 18;      // arrow operator — equivalent and preferred
```

The -> (arrow) operator is left-associative (like addition). Parentheses are necessary with (*t).hours because the dot operator has higher precedence than *. The arrow operator avoids this issue entirely.

## tod Example — Stack Version (from Chapter 8)

```
#include <stdio.h>

struct tod {
    int hours;
    int minutes;
};

void todPrint(struct tod when)
{
    printf(" %0.2d:%0.2d\n", when.hours, when.minutes);
}

int main(void)
{
    struct tod now   = { 16, 50 };
    struct tod later = { .hours = 18 };
    printf("now: ");            todPrint(now);
    printf("later: ");          todPrint(later);
    later.minutes = 1;
    printf("updated later: "); todPrint(later);
    return 0;
}
// All values stored on the STACK
```

## tod Example — Heap Version

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

struct tod {
    int hours;
    int minutes;
};
```

```
void todPrint(struct tod *when) {
    printf(" %0.2d:%0.2d\n", (*when).hours, (*when).minutes);
}

int main(void) {
    struct tod *now   = malloc(sizeof(struct tod));
    (*now).hours   = 16;
    (*now).minutes = 50;
    struct tod *later = malloc(sizeof(struct tod));
    (*later).hours = 18;
    printf("now: ");           todPrint(now);
    printf("later: ");         todPrint(later);
    (*later).minutes = 1;
    printf("updated later: "); todPrint(later);
    free(now);
    free(later);
    return 0;
}
```

### The Struct Hack — Flexible Array Member

What if you want a struct with an array whose size is determined at runtime? This technique — valid in C99 and beyond — is called the **struct hack** (flexible array member).

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

struct flex_array {
    int length;
    int a[];   // declared at end of struct; size is zero until malloc
};

int main(void) {
    printf("%zu\n", sizeof(struct flex_array)); // 4 (size of int; a has size
0)

    size_t array_size = 10;
    struct flex_array *fa = malloc(sizeof(struct flex_array)
                                   + array_size * sizeof(int));
    assert(fa);
    printf("%zu\n", sizeof(struct flex_array)); // 4
    printf("%zu\n", sizeof(fa));                // 8 (size of a pointer)
    printf("%zu\n", sizeof(*fa));               // 4 (size of the int fa
points at)

    fa->length = array_size;
    for (int i = 0; i < fa->length; i++)
        fa->a[i] = i * i;
    printf("%d\n", fa->a[4]);  // 16
    free(fa);
    return 0;
}
```

In <stdlib.h>, the type size_t should be used with malloc for portability.

## Vectors (Variable Size Array)

Arrays have a fixed size. Can we create an array that expands as more elements are needed at runtime? C++ has a `vector` library for this, but not C. We will build a simplified version for integers to demonstrate how it works.

**Idea:** Initialize contents to 0 and grow automatically by powers of 2 when the array is full.

### Interface File — vector.h

```
#ifndef VECTOR_H
#define VECTOR_H

struct vector;

// Creates and returns a new empty vector.
struct vector *vectorCreate(void);

// Deletes vector *v and returns NULL on success.
// (returns NULL to allow: v = vectorDelete(v);)
struct vector *vectorDelete(struct vector *v);

// Adds val to the end of the vector; allocates new space as necessary.
void vectorAdd(struct vector *v, int val);

// Sets the value at index ind to val.
void vectorSet(struct vector *v, int ind, int val);

// Returns the element at index ind.
int vectorGet(struct vector *v, int ind);

// Returns the length (number of elements) of the vector.
int vectorLength(struct vector *v);

#endif
```

### Implementation File — vector.c

```
#include "vector.h"
#include <assert.h>
#include <stdlib.h>

struct vector {         // hidden from the user
    int *arr;           // pointer to the underlying array
    int size;           // total allocated storage
    int length;         // number of elements currently in use
};

struct vector *vectorCreate(void)
{
    struct vector *v = malloc(sizeof(struct vector));
    assert(v);
    v->size   = 4;                     // initial capacity
    v->arr    = malloc(4 * sizeof(int));
    assert(v->arr);
    v->length = 0;                     // no elements yet
    return v;
}

struct vector *vectorDelete(struct vector *v)
{
    if (v)
    {
        free(v->arr);   // free inner array first
        free(v);        // then free the struct
    }
    return NULL;
}

void vectorAdd(struct vector *v, int value)
{
    assert(v);
    if (v->length == v->size)          // array is full — grow it
    {
        int newSize = v->size * 2;
        int *newArr = malloc(newSize * sizeof(int));
        for (int i = 0; i < v->size; ++i)
            newArr[i] = v->arr[i];     // copy old data
        newArr[v->size] = value;
        free(v->arr);                  // free old array
        v->size = newSize;
        v->arr  = newArr;
    }
    else
    {
        v->arr[v->length] = value;     // add to end
    }
    ++v->length;
}

void vectorSet(struct vector *v, int ind, int value)
{
    assert(v && ind >= 0 && ind <= v->length);
    v->arr[ind] = value;
}

int vectorGet(struct vector *v, int ind)
{
    assert(v && ind >= 0 && ind < v->length);
    return v->arr[ind];
}

int vectorLength(struct vector *v)
{
    assert(v);
    return v->length;
}
```

Notice how no implementation details appear in the header file — only declarations. This is the design principle of information hiding: implementation details are hidden from the user while the interface stays the same. We can change the internal code without affecting external users.  Also note: struct vector v (value) is not possible using this header, because the header does not know the struct's size (it is defined in the .c file). However, struct vector *v (pointer) is possible.

## Sample Program

```
#include <stdio.h>
#include "vector.h"

int main(void)
{
    struct vector *v = vectorCreate();
    for (int i = 0; i < 20; ++i)
        vectorAdd(v, i);

    printf("%d\n\n", vectorLength(v));

    for (int i = 0; i < 20; ++i)
        printf("v[%d]=%d ", i, vectorGet(v, i));
    printf("\n\n\n");

    for (int i = 0; i < 20; ++i)
    {
        vectorSet(v, i, i * i);
        printf("v[%d]=%d ", i, vectorGet(v, i));
    }
    printf("\n\n");

    v = vectorDelete(v);
    if (v == NULL) printf("Success!\n");
    else           printf("Freeing was not completed successfully!");
    return 0;
}
```

## 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!
```

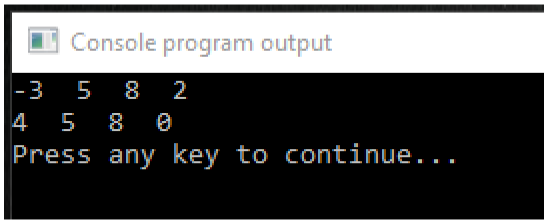
## Additional Examples

What is the output of the following program? Trace manually before running.

```c
#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;
}
```



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

## Extra Practice Problems

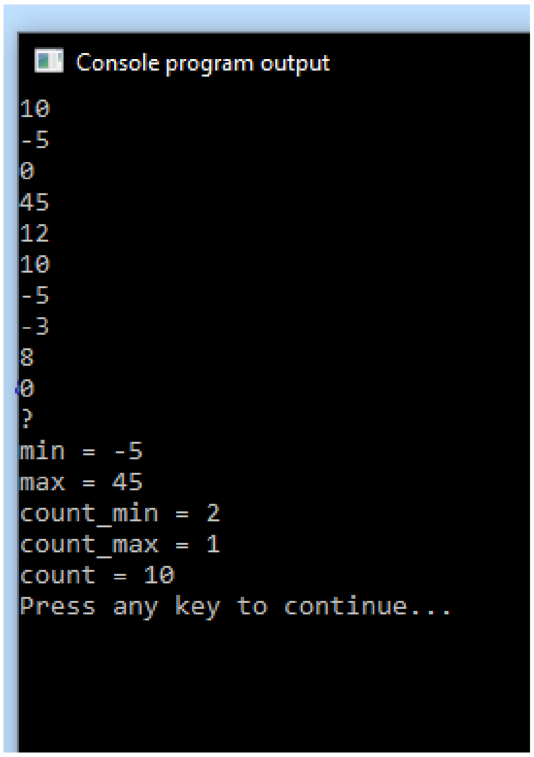
**1.**

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 total number of integers successfully read.

If no integers are successfully read, return 0 and do not modify any parameters.
Otherwise, set:
 *max — the largest number read
 *min — the smallest number read
 *count_max — number of times the largest value appeared
 *count_min — number of times the smallest value appeared

Note: You may not use arrays or structures.
Advice: Attempt this yourself before looking at the provided solution.

```
int main(void) {
    int min = 0, max = 0, count_min = 0, count_max = 0;
    int count = read_and_range(&max, &min, &count_max, &count_min);
    printf("min = %d\n",       min);
    printf("max = %d\n",       max);
    printf("count_min = %d\n", count_min);
    printf("count_max = %d\n", count_max);
    printf("count = %d\n",     count);
}
```


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? Trace manually before running.
Advice: Attempt this yourself before looking at the provided solution.

```
#include <stdio.h>

void mysterious(int *a, int *b, int *c)
{
    *a = *c;
    *b = *b + *a;
    *c = *a - *b;
}

int main()
{
    int w = 5, x = 1, y = 3, 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;
}
```

**3.**

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

Where a has m elements and b has n elements. Return a heap-allocated m×n matrix where c[i][j] = a[i] * b[j].
Note: int ** is a pointer to a pointer to an integer.
Advice: Attempt this yourself before looking at the provided solution.

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

int main()
{
    int m = 5, n = 3;
    int a[] = { 1, 2, 3, 4, 5 };
    int b[] = { 3, 2, 1 };
    int **c = outerproduct(a, m, b, n);
    assert(c);
    assert(c[0][0] == 3);
    assert(c[0][1] == 2);
    assert(c[2][2] == 3);
    assert(c[4][0] == 15);
    for (int i = 0; i < m; i++)
        free(c[i]);
    free(c);
    return 0;
}
```

**4.**

A run-length encoding stores repeated numbers as a (count, value) pair.
Example: 1 1 1 3 2 1 3 2 1 1 → encoded as 3 1 1 3 1 2 1 1 1 3 1 2 2 1
Shorter examples: 3 → 1 3 | 2 2 2 → 3 2 | 2 9 9 2 → 1 2 2 9 1 2

Write: int *unsay(const int *src, int src_len, int *dst_len)
Returns a heap-allocated decoded array. Sets *dst_len to its length.
Requires: src is valid with length src_len; src_len > 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.**

Simulate an economic market where producers and consumers adjust based on price.

Structs:
Market { double price }
Producer { int base_production; double current_production; Market *market }
Consumer { int base_consumption; double current_consumption; Market *market }

Implement:
void adjustProducer(Producer *producer)
void adjustConsumer(Consumer *consumer)
void adjustMarketPrice(Market *market, Producer producers[], Consumer consumers[])

Equations:
current_production = base_production × (market_price / base_price)
current_consumption = base_consumption × (base_price / market_price)
new_price = current_price + (total_demand - total_supply) × price_adjustment_factor

Constraints: 3 producers, 3 consumers; price_adjustment_factor = 0.05; 5 simulation rounds.

Expected output excerpt:
```
Round 1: Market Price $30.00 | Production 90.00 | Consumption 10.00
Round 2: Market Price $26.00 | Production 78.00 | Consumption 11.54
Round 3: Market Price $22.68 | Production 68.03 | Consumption 13.23
Round 4: Market Price $19.94 | Production 59.81 | Consumption 15.05
Round 5: Market Price $17.70 | Production 53.10 | Consumption 16.95
```

```
int main() {
    Market market = {30};   // initial price $30
    Producer producers[NUM_AGENTS];
    Consumer consumers[NUM_AGENTS];
    for (int i = 0; i < NUM_AGENTS; i++) {
        producers[i].base_production  = 100;
        producers[i].market           = &market;
        consumers[i].base_consumption = 10;
        consumers[i].market           = &market;
    }
    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;
}
```

**6.**

Simulate particles in 2D space over 10 time steps using dynamic memory allocation.

Struct:
  Particle { float x_position; float y_position; float x_velocity; float y_velocity }

Implement:
  Particle *createParticle(float x_pos, float y_pos, float x_vel, float y_vel)
  void updateParticle(Particle *particle)   — update position by adding velocity
  void displayParticle(const Particle *particle)
  void destroyParticle(Particle *particle)

The user inputs the number of particles and their initial positions and velocities.
Run the simulation for 10 time steps, updating and displaying each particle.
Free all allocated memory at the end.

Sample (1 particle at (1,1) with velocity (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)
```

```
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 ");
        printf("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;
}
```

Problem 3 solution:

```
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;
}
```