

Loops & Arrays in C

COMP 2510 – Week 2

Hesam Alizadeh

Where We Are in the Course

Last Week (Week 1)

- C toolchain: **edit** → **compile** → **link** → **run**
- Basic syntax, types, `printf`/`scanf`
- **Decision making:** `if`, `else`, `switch`
- Debugging fundamentals

Today (Week 2)

- **Control Flow:** Loops (`while`, `for`, `do-while`)
- **Arrays:** Memory layout, initialization, iteration
- **Critical Differences** from Java
- Common pitfalls & debugging

Coming Up

- **Week 3:** Functions & Call Stack
- **Week 4:** Pointers

By the end of today you can...

- Use `switch` statements effectively and avoid fall-through bugs
- Safely write `while`, `for`, and `do-while` loops in C
- Declare, initialize, and iterate over **arrays**
- Explain **3 key differences** between C & Java arrays
- Manipulate Strings as character arrays
- Debug loop/array bugs using tracing & compiler warnings

Recap: `switch` Statements

Cleaner than many `else-if`s

Great for checking a single variable against specific **integer** or **char** constants.

```
char grade;
printf("Enter grade (A/B/C): ");
scanf(" %c", &grade);

switch (grade) {
    case 'A':
        printf("Excellent!\n");
        break;
    case 'B':
        printf("Good job.\n");
        break;
    case 'C':
        printf("You passed.\n");
        break;
    default:
        printf("Invalid grade.\n");
}
```

⚠️ Key Restrictions

1. Expression must be integer-compatible:

- `int`, `char`, `long` ✓
- `float`, `double`, Strings ✗

2. Cases must be Constants:

- `case 1:` ✓
- `case x:` ✗ (variable)
- `case x > 5:` ✗ (condition)

Note: Just like Java, `break` is crucial!

⚠ The `switch` Fall-Through

If you omit `break`, execution **falls through** to the next case.

```
int num = 1;
switch (num) {
    case 1:
        printf("One ");
        // No break here!
    case 2:
        printf("Two ");
        break;
    default:
        printf("Other");
}
```

Is this a bug or a feature?

Sometimes useful! stacking cases:

```
case 'a':
case 'e':
case 'i':
    printf("It is a vowel");
    break;
```

Loop Syntax: Java → C

Familiar Syntax

```
// Same structure as Java!  
while (condition) { /* body */ }  
  
for (init; condition; update) { /* body */ }  
  
do { /* body */ } while (condition);
```

Good news: The syntax you know works here.

Bad news: C won't save you from your mistakes.

C-Specific Gotchas

- Braces `{}` optional for single statements
- No `for-each` loop like Java's `for (int x : arr)`
- Must manually track loop termination

while Loop

Checks condition **BEFORE** each iteration. Runs **0+ times**.

```
int i = 0;
while (i < 3) {
    printf("%d ", i);
    i++;    // ⚠️ MUST update!
}
// Output: ?
```

Pitfall: Forgetting to update `i` → **infinite loop!**

Unlike Java: No IDE warning, no runtime exception. Your program just hangs.

Sentinel Value Pattern

```
int num;
printf("Enter numbers (-1 to stop):\n");
scanf("%d", &num);

while (num != -1) {
    printf("Got: %d\n", num);
    scanf("%d", &num);
}
```

Note: Must read BEFORE the loop AND inside it. Common pattern in C for input processing.



Activity: while Loop Prediction

Predict the output for each. Then trace through to verify.

Example 1:

```
int x = 5;
while (x > 0) {
    printf("%d ", x);
    x -= 2;
}
// Output: ?
```

Example 2:

```
int n = 1;
while (n < 50) {
    n = n * 2;
}
printf("%d", n);
// Output: ?
```

Example 3:

```
int count = 0;
int val = 100;
while (val > 1) {
    val = val / 2;
    count++;
}
printf("count = %d", count);
// Output: ?
```

Challenge: What's the relationship between the initial value and the count?

do-while Loop

Body runs **AT LEAST ONCE** before condition check.

```
int num;
do {
    printf("Enter 1-10: ");
    scanf("%d", &num);
} while (num < 1 || num > 10);
printf("You entered: %d\n", num);
```

Ideal for:

- Input validation
- Menu systems
- Games (play at least once)

Why not **while**?

```
int num; // UNINITIALIZED - garbage value!
while (num < 1 || num > 10) {
    printf("Enter 1-10: ");
    scanf("%d", &num);
}
```

In Java: Compiler error for uninitialized variable.

In C: Compiles fine. `num` contains garbage. Loop behavior is unpredictable!



Activity: Menu with do-while

Trace through this code. What happens if user enters: 2, 5, 3?

```
int choice;
do {
    printf("\n=== MENU ===\n");
    printf("1. Say Hello\n");
    printf("2. Say Goodbye\n");
    printf("3. Exit\n");
    printf("Choice: ");
    scanf("%d", &choice);

    if (choice == 1) {
        printf("Hello!\n");
    } else if (choice == 2) {
        printf("Goodbye!\n");
    } else if (choice != 3) {
        printf("Invalid choice!\n");
    }
} while (choice != 3);

printf("Program ended.\n");
```

What gets printed for inputs 2, 5, 3?

for Loop

```
for (int i = 0; i < 5; i++) {  
    printf("%d ", i);  
}  
// Output: 0 1 2 3 4
```

Structure:

- `init` → runs **once** at the start
- `condition` → checked **before** each iteration
- `update` → runs **after** each body execution

Scope (C99+): `i` only exists inside the loop.

Variations

```
// Count down  
for (int i = 10; i >= 1; i--) {  
    printf("%d ", i);  
}  
  
// Step by 2  
for (int i = 0; i < 10; i += 2) {  
    printf("%d ", i);  
}  
  
// Multiple variables  
for (int i = 0, j = 10; i < j; i++, j--) {  
    printf("(%d,%d) ", i, j);  
}
```



Activity: for Loop Patterns

Write for loops to produce each output:

1. Print: 10 8 6 4 2 0

```
for (int i = ?; ?; ?) {  
    printf("%d ", i);  
}
```

2. Print: 1 2 4 8 16 32 64

```
for (int i = ?; ?; ?) {  
    printf("%d ", i);  
}
```

3. Print: (0,5) (1,4) (2,3) (3,2) (4,1) (5,0)

```
for (int i = ?, j = ?; ?; ?, ?) {  
    printf("(%d,%d) ", i, j);  
}
```

⚠ The Semicolon of Death

This doesn't exist in Java!

```
for (int i = 0; i < 3; i++); // STRAY SEMICOLON!  
{  
    printf("Hello\n");  
}
```

What actually happens:

1. Loop runs 3 times with **empty body** (the `;` IS the body)
2. Block `{}` runs **once** after loop ends

In Java: IDE would likely warn you.

In C: Compiles silently. Your bug.

Same danger with `while`:

```
int x = 5;  
while (x > 0);  
{  
    x--;  
}
```

Result: Infinite loop!

Compiler Can Help

```
gcc -Wall -Wextra program.c  
# warning: for loop has empty body
```

🔑 **Always compile with `-Wall -Wextra`!**

Activity: Find the Bugs

Each snippet has a C-specific bug. Find and fix them.

Bug 1:

```
int sum = 0;
for (int i = 1; i <= 10; i++);
{
    sum += i;
}
printf("Sum: %d\n", sum);
```

Bug 2:

```
int x = 10;
while (x > 0)
    printf("%d ", x);
    x--;
```

Bug 3:

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

break and continue

break — Exit the loop immediately

```
for (int i = 0; i < 100; i++) {  
    if (i == 5) break;  
    printf("%d ", i);  
}  
// Output: 0 1 2 3 4
```

continue — Skip to next iteration

```
for (int i = 0; i < 5; i++) {  
    if (i == 2) continue;  
    printf("%d ", i);  
}  
// Output: 0 1 3 4
```

Nested Loops

break and **continue** only affect the **innermost** loop:

```
for (int i = 0; i < 3; i++) {  
    for (int j = 0; j < 3; j++) {  
        if (j == 1) break; // Only exits inner loop  
        printf("(%d,%d) ", i, j);  
    }  
    printf("\n");  
}  
// Output: ?
```



Activity: Trace `break` and `continue`

Trace through each. What is the final output?

Example 1:

```
int sum = 0;
for (int i = 1; i <= 10; i++) {
    if (i % 3 == 0) continue;
    sum += i;
}
printf("Sum: %d\n", sum);
// Which values are added? Final sum = ?
```

Example 2:

```
int product = 1;
for (int i = 1; i <= 10; i++) {
    product *= i;
    if (product > 100) break;
}
printf("i = ?, product = ?\n");
// What are final values of i and product?
```

Declaring & Initializing Arrays

```
// Uninitialized
int scores[5];

// Full initialization
int scores[5] = {90, 80, 70, 60, 50};

// Partial → rest become 0
int scores[5] = {90, 80};
// Result: {90, 80, 0, 0, 0}

// Let compiler count
int scores[] = {90, 80, 70}; // size = 3
```

⚠ Uninitialized = ?

```
int mystery[3];
printf("%d\n", mystery[0]);
// Could print ANYTHING!
```

In Java: `new int[3]` gives you zeros.

In C: Local arrays contain "random" data unless you initialize them.

Best Practice

```
int data[10] = {0}; // All zeros
```



Activity: Array Initialization

What values are in each array after initialization?

```
int b[] = {2, 4, 6, 8, 10};
```

```
// Size of b = ?
```

```
int c[3] = {0};
```

```
// c[0] = ? c[1] = ? c[2] = ?
```

```
int d[4] = {7};
```

```
// d[0] = ? d[1] = ? d[2] = ? d[3] = ?
```

Generating Random Numbers

Need test data for your arrays?

The `stdlib.h` & `time.h` way

1. **Seed the generator** (Do this ONCE at start of main)
2. **Call** `rand()` to get a number

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

int main() {
    // 1. Seed using current time
    srand(time(NULL));

    // 2. Generate numbers
    int r = rand(); // 0 to RAND_MAX

    // 3. Limit range (e.g., 0 to 9)
    int small = rand() % 10;
}
```

Common **Mistake**: Re-seeding

```
for (int i=0; i<5; i++) {
    srand(time(NULL)); // WRONG!
    printf("%d", rand());
}
```

Why? The loop runs so fast the time hasn't changed. You will get the **same number** 5 times!



Rule: Call `srand` exactly once at the beginning.



`rand()` is actually a pre-set list of numbers. `srand()` tells the computer where to start reading that list. If you don't seed, you start at the same spot every time!

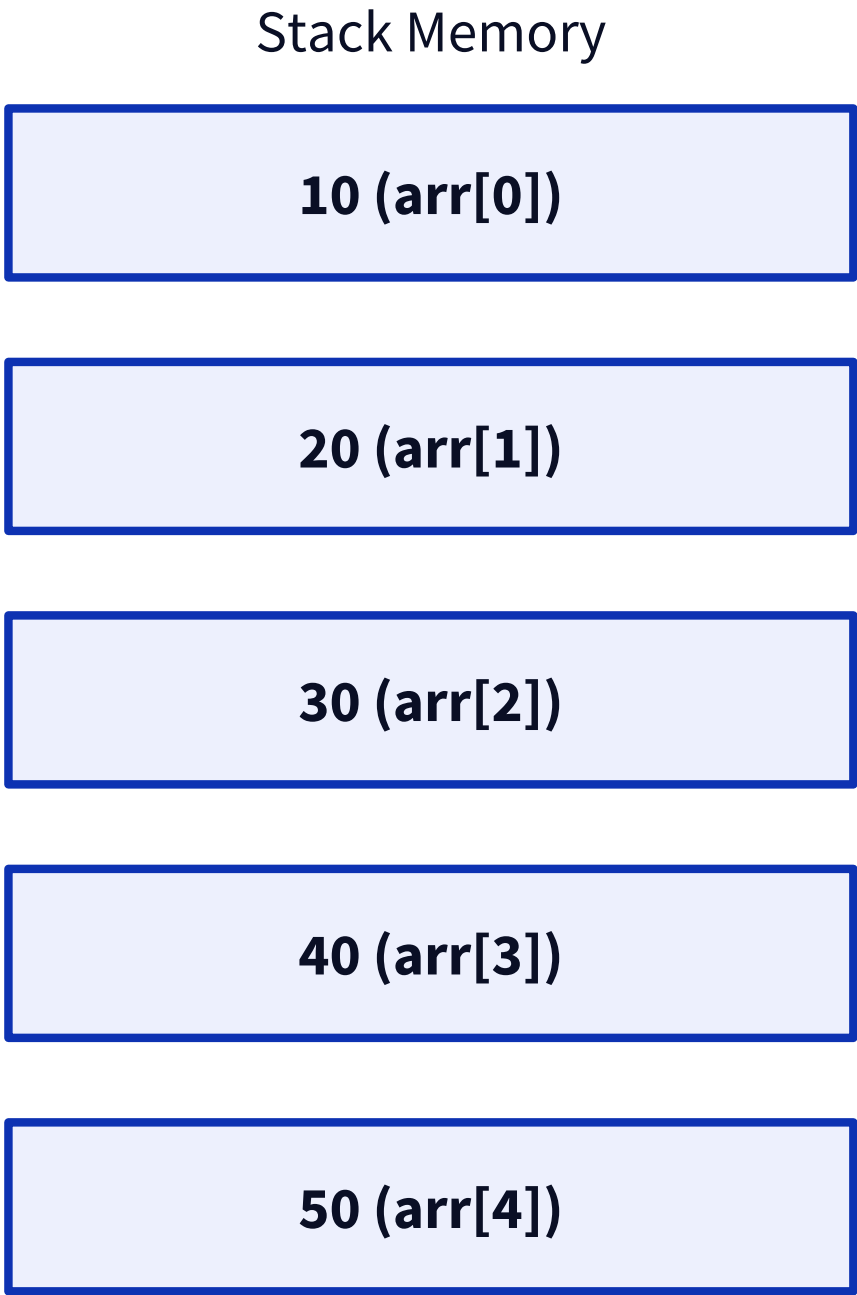
Array = Contiguous Memory

```
int arr[5] = {10, 20, 30, 40, 50};
```

In memory (assuming `int` = 4 bytes):

Index	Value	Address	Calculation
0	10	1000	<code>1000 + (0 * 4)</code>
1	20	1004	<code>1000 + (1 * 4)</code>
2	30	1008	<code>1000 + (2 * 4)</code>

Elements stored **consecutively** in memory. This is why C can calculate element positions quickly.



Iterating Over Arrays

ALWAYS use a named constant for size!

```
#define SIZE 5
int data[SIZE] = {10, 20, 30, 40, 50};

// Print all elements
for (int i = 0; i < SIZE; i++) {
    printf("data[%d] = %d\n", i, data[i]);
}
```

- Use `i < SIZE` (not `i <= SIZE`)
- Use `SIZE` constant everywhere
- Change once, works everywhere



Activity: Array Processing

Complete each task using the array below:

```
#define SIZE 6  
int values[SIZE] = {4, 8, 2, 9, 1, 7};
```

1. Calculate the sum of all elements

```
int sum = 0;  
// Your loop here  
printf("Sum = %d\n", sum); // Expected: 31
```

2. Find the minimum value

```
int min = values[0];  
// Your loop here  
printf("Min = %d\n", min); // Expected: 1
```

3. Count how many values are greater than 5

```
int count = 0;  
// Your loop here  
printf("Count = %d\n", count); // Expected: 3
```

⚠ Out-of-Bounds Access

C WILL NOT SAVE YOU

```
int a[3] = {10, 20, 30};

printf("%d\n", a[3]); // Reading garbage!
a[5] = 99;           // Corrupting memory!
a[-1] = 42;          // Also out of bounds!
```

Possible outcomes:

- Crash (segfault)
- Overwrite another variable
- Silent data corruption
- Security vulnerability (buffer overflow!)
- Works today, crashes tomorrow

Why No Safety?

Java: `ArrayIndexOutOfBoundsException`

C: Undefined Behavior

What does this code do?

```
int scores[3] = {85, 90, 88};
int secret = 12345;

scores[3] = 0;
printf("secret = %d\n", secret);
// Output: ?
```

💡 **Rule:** Always ensure `0 ≤ index < SIZE`

Common Array Patterns

Reverse Print (no modification)

```
#define SIZE 5
int arr[SIZE] = {1, 2, 3, 4, 5};

for (int i = SIZE - 1; i >= 0; i--) {
    printf("%d ", arr[i]);
}
// Output: 5 4 3 2 1
```

Copy Array

```
int source[SIZE] = {1, 2, 3, 4, 5};
int dest[SIZE];

for (int i = 0; i < SIZE; i++) {
    dest[i] = source[i];
}
```



Activity: Array Transformations

Start with:

```
#define SIZE 6  
int arr[SIZE] = {2, 5, 1, 8, 3, 9};
```

1. Replace each element with running sum

```
// After: arr = {2, 7, 8, 16, 19, 28}  
// (arr[i] = sum of all elements from 0 to i)
```

2. Shift all elements left by 1 (first element goes to end)

```
// After: arr = {5, 1, 8, 3, 9, 2}  
// Hint: Save the first element before shifting
```

3. Reverse the array

```
// After: arr = {5, 4, 3, 2, 1}  
// Hint: Swap the current index element with the (array length - index) element in a loop
```

Strings: The `char` Array

C does not have a `String` type.

C has **Arrays of Characters**.

```
char name[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

The Sentinel: `\0` (Null Terminator)

How does `printf` know when the name ends?

It looks for the special character `\0` (ASCII value 0).

Without `\0`, the string never ends.

String Literals

A shorthand for the array above:

```
// Compiler automatically adds '\0'  
char name[] = "Hello";
```

Size is 6 bytes (5 letters + 1 null).

Visualizing Memory:

```
['H']['e']['l']['l']['o']['\0']
```



Activity: Calculate string length

Java has `s.length()`. In C, we have to count.

Task: Write a loop that counts how many characters are in `text` before the `\0`.

```
char text[] = "Programming";
int length = 0;

// YOUR CODE HERE
// 1. Loop until you hit '\0'
// 2. Increment length
// 3. DO NOT count the '\0' itself

printf("Length is: %d\n", length); // Expected: 11
```

Hint: You can use a `while` loop checking `text[length] != ...`

Reading Strings

Using `scanf`

```
char name[20];  
printf("Enter name: ");  
  
scanf("%s", name);
```

Restriction: `scanf` stops at the first space character!

Input: `The Human`

Stored: `The`

What are some ways that this can be fixed?

Another Danger: Buffer Overflow

What if the user types 100 letters into `name[20]`?

C will crash or corrupt memory.

Solution:

```
scanf("%19s", name); // Read max 19 chars
```

Why 19?

2D Arrays (Matrices)

Declaration & Access

```
#define ROWS 3
#define COLS 4

// A 3x4 grid of integers
int grid[ROWS][COLS] = {
    {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11, 12}
};

// Accessing elements
int secondRowThirdColumn = grid[1][2] = 99;
```

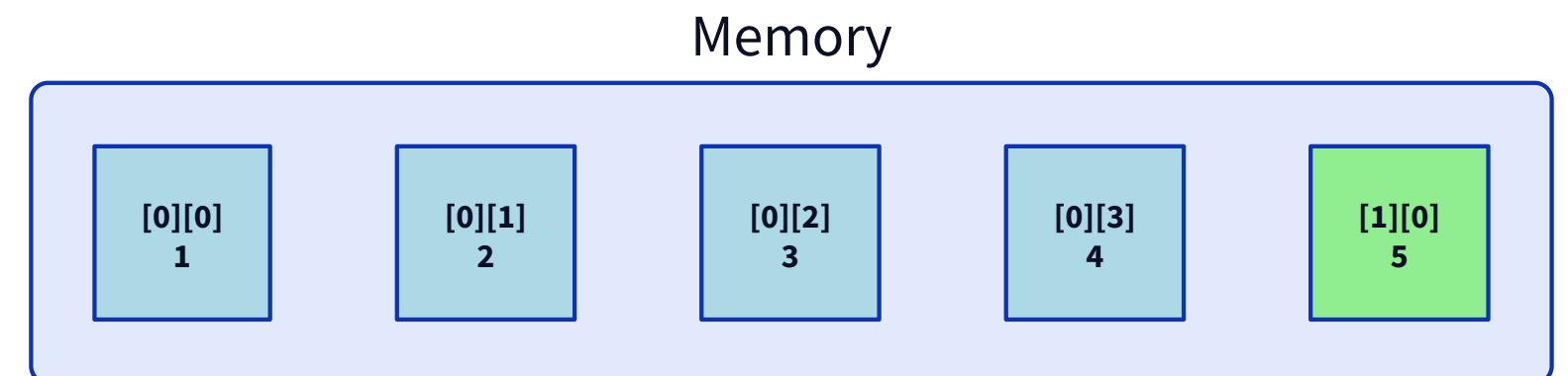
Memory Reality:

In Java, 2D arrays are "arrays of arrays" (references).
In C, it is **one solid block** of memory.

Nested Loop Iteration

```
for (int i = 0; i < ROWS; i++) {
    for (int j = 0; j < COLS; j++) {
        printf("%4d", grid[i][j]);
    }
    printf("\n");
}
```

Physical Memory (How it's actually stored)



The `sizeof` Trap

```
#define SIZE 5
int arr[SIZE] = {1, 2, 3, 4, 5};

printf("Array size: %zu bytes\n", sizeof(arr));
// Output: 20 (5 ints x 4 bytes)

printf("Element count: %zu\n", sizeof(arr) / sizeof(arr[0]));
// Output: 5
```

Useful for calculating array length:

```
int nums[] = {10, 20, 30, 40};
int count = sizeof(nums) / sizeof(nums[0]);
// count = 4
```

⚠ The Trap

This only works **where the array is declared!**

Next week we'll see that when you pass an array to a function, `sizeof` no longer gives you the full array size.

For now: Always track array size with a `#define` constant.

```
#define SIZE 10
int data[SIZE];
// Always use SIZE, never sizeof in loops
```

Debugging Loops & Arrays

Print Tracing

```
for (int i = 0; i < SIZE; i++) {  
    printf("DEBUG: i=%d, arr[i]=%d\n", i, arr[i]);  
    // ... rest of loop  
}
```

Compiler Warnings

```
gcc -Wall -Wextra program.c -o program
```

Catches:

- Empty loop bodies
- Uninitialized variables
- Some bounds issues

Common Bugs Checklist

- [] Off-by-one: `i <= SIZE` vs `i < SIZE`
- [] Semicolon after `for()`/`while()`
- [] Uninitialized array elements
- [] Out-of-bounds access
- [] Forgetting to update loop variable
- [] Using `=` instead of `==`
- [] Wrong array index in nested loops

Summary: The C Mindset

Three Key Concepts

1. **Arrays are Raw Memory:** No `.length`. No bounds checking. You are the safety net.
2. **Strings use Sentinels:** It's just a `char[]`. It **must** end with `\0`.
3. **Loops are Literal:** Computers do exactly what you say (even if you accidentally say "do nothing forever").

The "Save Your Grade" Checklist

- **[] Initialize:** `int arr[5] = {0};` (Kill the garbage values)
- **[] Bounds:** `0` to `SIZE - 1`. Never go higher.
- **[] Syntax:** No semicolon after `for(...)` or `while(...)`!
- **[] Space:** String arrays need size `N+1` to fit the `\0`.

Next week: Functions & The Call Stack.