# MTH 4300/4299: Programming and Computer Science II

## Lecture 08: Recursion

Jaime Abbariao

## ▒▒▒ Recursion

A recursive function is a function that calls itself. Instead of solving a problem all at once with a loop, you break it into a smaller version of the same problem and let the function call itself on the smaller piece.

You've probably seen recursion before. Today we'll refresh the basics, then build up to more interesting patterns.

Every recursive function needs two things:

1. Base case: when to stop (without this, you get infinite recursion and a stack overflow)
2. Recursive case: how to break the problem into a smaller version of itself

## ░░░░ Why recursion?

Some problems are naturally recursive -- they have structure that "nests" or "repeats at a smaller scale."

- A folder contains files and other folders (which contain files and other folders...)
- A linked list is either empty, or a node followed by a linked list
- Many mathematical definitions are recursive: factorial, Fibonacci, etc.

Recursion can also make code shorter and more expressive than the equivalent loop, especially for tree/graph traversals and divide-and-conquer algorithms.

When not to use recursion: if a simple loop does the job just as clearly, prefer the loop. Recursion has overhead (each call adds a frame to the call stack), and deep recursion can cause stack overflows.

## The call stack

When a function calls itself, each call gets its own stack frame with its own local variables. The frames stack up until a base case is reached, then they "unwind" as each call returns.

```
factorial(4)
  -> factorial(3)
       -> factorial(2)
            -> factorial(1)
                 returns 1        // base case
            returns 2 * 1 = 2
       returns 3 * 2 = 6
  returns 4 * 6 = 24
```

If you forget the base case, the stack grows forever and your program crashes with a stack overflow.

## Single-parameter recursion

### Exercise: Recursive GCD

The greatest common divisor (GCD) of two integers is the largest number that divides both. Euclid's algorithm gives a naturally recursive definition:

- `gcd(a, 0) = a` (base case)
- `gcd(a, b) = gcd(b, a % b)` (recursive case)

Write a recursive function `int gcd(int a, int b)`.

Test with `gcd(48, 18)` (should print `6`) and `gcd(100, 75)` (should print `25`).

# Exercise: Recursive GCD (Solution)

```
#include <iostream>

int gcd(int a, int b) {
    if (b == 0) {
        return a;
    }
    return gcd(b, a % b);
}

int main() {
    std::cout << gcd(48, 18) << std::endl;   // 6
    std::cout << gcd(100, 75) << std::endl;  // 25
    return 0;
}
```

## Exercise: Count evens in a range

Write a recursive function `int count_evens(int low, int high)` that counts how many even numbers exist from `low` to `high` inclusive. Do not use any loops.

- Base case: if `low > high`, return `0`
- Recursive case: add `1` if `low` is even, then recurse with `low + 1`

Test with `count_evens(1, 10)` (should print `5`) and `count_evens(3, 7)` (should print `2`).

```
#include <iostream>

int count_evens(int low, int high) {
    if (low > high) {
        return 0;
    }
    return (low % 2 == 0 ? 1 : 0) + count_evens(low + 1, high);
}

int main() {
    std::cout << count_evens(1, 10) << std::endl;  // 5
    std::cout << count_evens(3, 7) << std::endl;   // 2
    return 0;
}
```

## ▒▒▒▒ Exercise: Is power of two

Write a recursive function `bool is_power_of_two(int n)` that returns `true` if `n` is a power of 2.

- Base case: if `n == 1`, return `true`
- Base case: if `n <= 0` or `n % 2 != 0`, return `false`
- Recursive case: recurse on `n / 2`

Test with `is_power_of_two(16)` (true), `is_power_of_two(18)` (false), and `is_power_of_two(1)` (true).

```cpp
#include <iostream>

bool is_power_of_two(int n) {
    if (n == 1) {
        return true;
    }
    if (n <= 0 || n % 2 != 0) {
        return false;
    }
    return is_power_of_two(n / 2);
}

int main() {
    std::cout << std::boolalpha;
    std::cout << is_power_of_two(16) << std::endl;  // true
    std::cout << is_power_of_two(18) << std::endl;  // false
    std::cout << is_power_of_two(1) << std::endl;   // true
    return 0;
}
```

## ░░░░ Exercise: Count down

Write a recursive function `void count_down(int n)` that prints the numbers from `n` down to `1`, each on its own line. Do not use any loops.

Expected output for `count_down(5)`:

```
5
4
3
2
1
```

Hint: Print `n`, then recursively count down from `n - 1`. What's the base case?

## Exercise: Count down (Solution)

```
#include <iostream>

void count_down(int n) {
    if (n <= 0) {
        return;
    }
    std::cout << n << std::endl;
    count_down(n - 1);
}

int main() {
    count_down(5);
    return 0;
}
```

What would happen if we swapped the two lines in the recursive case -- calling `count_down(n - 1)` before printing `n`? Try it mentally. You'd get `1 2 3 4 5` instead -- counting up. The placement of work relative to the recursive call matters.

## ░░░ Exercise: Fibonacci

The Fibonacci sequence is defined as:

- `fib(0) = 0`
- `fib(1) = 1`
- `fib(n) = fib(n - 1) + fib(n - 2)` for `n >= 2`

Write a recursive function `int fib(int n)` and print `fib(0)` through `fib(10)`.

Expected output:

```
0 1 1 2 3 5 8 13 21 34 55
```

Note: This recursive version is very slow for large `n` because it recomputes the same values many times. We'll see better approaches later in the course.

```cpp
#include <iostream>

int fib(int n) {
    if (n <= 0) {
        return 0;
    }
    if (n == 1) {
        return 1;
    }
    return fib(n - 1) + fib(n - 2);
}

int main() {
    for (int i = 0; i <= 10; i++) {
        std::cout << fib(i);
        if (i < 10) {
            std::cout << " ";
        }
    }
    std::cout << std::endl;
    return 0;
}
```

## ▒▒▒▒ Exercise: Recursive power

Write a recursive function `int power(int base, int exp)` that computes `base` raised to the power `exp` (assume `exp >= 0`).

- `power(x, 0) = 1` (base case)
- `power(x, n) = x * power(x, n - 1)` (recursive case)

Test with `power(2, 10)` (should print `1024`) and `power(3, 4)` (should print `81`).

## Exercise: Recursive power (Solution)

```cpp
#include <iostream>

int power(int base, int exp) {
    if (exp == 0) {
        return 1;
    }
    return base * power(base, exp - 1);
}

int main() {
    std::cout << power(2, 10) << std::endl;  // 1024
    std::cout << power(3, 4) << std::endl;   // 81
    return 0;
}
```

## ▒▒▒ Multi-parameter recursion

So far, every recursive function has had one parameter that "shrinks" toward the base case. But sometimes you need multiple parameters that change together.

This is common when you need to:

- Track a position or index alongside the data
- Carry an accumulator that builds up the result
- Work with two ends of a range (like binary search)

### Example: Print array recursively

Print every element of an array using recursion. We need two parameters beyond the array: the current index and the size.

```cpp
#include <iostream>

void print_array(int arr[], int index, int size) {
    if (index >= size) {
        return;
    }
    std::cout << arr[index] << " ";
    print_array(arr, index + 1, size);
}

int main() {
    int arr[] = {10, 20, 30, 40, 50};
    print_array(arr, 0, 5);
    std::cout << std::endl;
    return 0;
}
```

```
10 20 30 40 50
```

Here `index` starts at `0` and increases by `1` on each call. The base case is when `index` reaches `size`.

## Example: Recursive binary search

Binary search is a naturally recursive algorithm. We maintain low and high indices that narrow the search range.

```cpp
#include <iostream>

int binary_search(int arr[], int low, int high, int target) {
    if (low > high) {
        return -1;
    }
    int mid = low + (high - low) / 2;
    if (arr[mid] == target) {
        return mid;
    }
    if (target < arr[mid]) {
        return binary_search(arr, low, mid - 1, target);
    }
    return binary_search(arr, mid + 1, high, target);
}

int main() {
    int arr[] = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91};
    int size = 10;

    int idx = binary_search(arr, 0, size - 1, 23);
    std::cout << "Found 23 at index: " << idx << std::endl;   // 5

    idx = binary_search(arr, 0, size - 1, 50);
    std::cout << "Found 50 at index: " << idx << std::endl;   // -1

    return 0;
}
```

Two parameters (low and high) shrink the range on each call until we find the target or the range is empty.

## Exercise: Recursive sum of array

Write a recursive function `int array_sum(int arr[], int index, int size)` that returns the sum of all elements in the array.

- Base case: if `index >= size`, return `0`
- Recursive case: `arr[index] + array_sum(arr, index + 1, size)`

Test with `{3, 7, 1, 9, 2}`. Expected output: `22`

## Exercise: Recursive sum of array (Solution)

```cpp
#include <iostream>

int array_sum(int arr[], int index, int size) {
    if (index >= size) {
        return 0;
    }
    return arr[index] + array_sum(arr, index + 1, size);
}

int main() {
    int arr[] = {3, 7, 1, 9, 2};
    std::cout << array_sum(arr, 0, 5) << std::endl;  // 22
    return 0;
}
```

## Exercise: Recursive palindrome check

Write a function `bool is_palindrome(std::string s, int left, int right)` that checks whether a string is a palindrome by comparing characters from both ends moving inward.

- Base case: if `left >= right`, return `true`
- Recursive case: if `s[left] == s[right]`, recurse with `left + 1` and `right - 1`; otherwise return `false`

Test with `"racecar"` (true), `"hello"` (false), and `"abba"` (true).

## Exercise: Recursive palindrome check (Solution)

```cpp
#include <iostream>
#include <string>

bool is_palindrome(std::string s, int left, int right) {
    if (left >= right) {
        return true;
    }
    if (s[left] != s[right]) {
        return false;
    }
    return is_palindrome(s, left + 1, right - 1);
}

int main() {
    std::cout << std::boolalpha;
    std::string s1 = "racecar";
    std::cout << s1 << ": " << is_palindrome(s1, 0, s1.size() - 1) << std::endl;

    std::string s2 = "hello";
    std::cout << s2 << ": " << is_palindrome(s2, 0, s2.size() - 1) << std::endl;

    std::string s3 = "abba";
    std::cout << s3 << ": " << is_palindrome(s3, 0, s3.size() - 1) << std::endl;

    return 0;
}
```

```
racecar: true
hello: false
abba: true
```

## Recursion with references (tail recursion)

In the recursive functions we've written so far, the result is built up as the calls return. For example, in factorial, the multiplication happens after the recursive call comes back:

```
return n * factorial(n - 1);  // must wait for factorial(n-1) to return
```

An alternative pattern is tail recursion: instead of waiting for the recursive call to return and then doing more work, you pass the accumulated result forward as a parameter (by reference or by value). The recursive call is the very last thing the function does.

## ░░░ Tail recursion with reference parameters

We can use a reference parameter to accumulate the result as we recurse. This avoids building up a chain of pending operations on the call stack.

```
#include <iostream>

void factorial_helper(int n, int& result) {
    if (n <= 0) {
        return;
    }
    result *= n;
    factorial_helper(n - 1, result);
}

int main() {
    int result = 1;
    factorial_helper(5, result);
    std::cout << result << std::endl;  // 120
    return 0;
}
```

Here `result` is passed by reference. Each call multiplies into it directly -- no waiting for return values. The recursive call `factorial_helper(n - 1, result)` is the last thing the function does (this is what makes it "tail recursive").

## Comparing the two approaches

Standard recursion (builds result on the way back up):

```
factorial(4)
  -> 4 * factorial(3)
       -> 3 * factorial(2)
            -> 2 * factorial(1)
                 -> 1 * factorial(0)
                      returns 1
                 returns 1
            returns 2
       returns 6
  returns 24
```

Tail recursion with reference (builds result on the way down):

```
factorial_helper(4, result=1)
  result = 1 * 4 = 4
  -> factorial_helper(3, result=4)
       result = 4 * 3 = 12
       -> factorial_helper(2, result=12)
            result = 12 * 2 = 24
            -> factorial_helper(1, result=24)
                 result = 24 * 1 = 24
                 -> factorial_helper(0, result=24)
                      returns (base case)
```

The answer is ready as soon as we hit the base case -- no unwinding needed.

### Exercise: Tail-recursive sum of array

Rewrite array_sum using a reference parameter to accumulate the result.

```
void array_sum(int arr[], int index, int size, int& total);
```

Start with total = 0, then call array_sum(arr, 0, 5, total).

Test with {3, 7, 1, 9, 2}. Expected output: 22

## Exercise: Tail-recursive sum of array (Solution)

```cpp
#include <iostream>

void array_sum(int arr[], int index, int size, int& total) {
    if (index >= size) {
        return;
    }
    total += arr[index];
    array_sum(arr, index + 1, size, total);
}

int main() {
    int arr[] = {3, 7, 1, 9, 2};
    int total = 0;
    array_sum(arr, 0, 5, total);
    std::cout << total << std::endl;  // 22
    return 0;
}
```

## Exercise: Tail-recursive reverse string

Write a function that reverses a string in place using recursion and references.

```
void reverse_string(std::string& s, int left, int right);
```

- Base case: if `left >= right`, return
- Recursive case: swap `s[left]` and `s[right]`, then recurse with `left + 1` and `right - 1`

Test with `"hello"`. Expected output: `olleh`

## Exercise: Tail-recursive reverse string (Solution)

```cpp
#include <iostream>
#include <string>

void reverse_string(std::string& s, int left, int right) {
    if (left >= right) {
        return;
    }
    char temp = s[left];
    s[left] = s[right];
    s[right] = temp;
    reverse_string(s, left + 1, right - 1);
}

int main() {
    std::string s = "hello";
    reverse_string(s, 0, s.size() - 1);
    std::cout << s << std::endl;  // olleh
    return 0;
}
```

Notice how the reference parameter `std::string& s` lets each recursive call modify the same string. Without the reference, each call would get its own copy and the swaps would be lost.

### Exercise: Tail-recursive count occurrences

Write a function that counts how many times a value appears in an array, using a reference parameter for the count.

```
void count_occurrences(int arr[], int index, int size, int target, int& count);
```

Test with `{1, 3, 2, 3, 5, 3, 7}` and `target = 3`. Expected output: `3`

## Exercise: Tail-recursive count occurrences (Solution)

```cpp
#include <iostream>

void count_occurrences(int arr[], int index, int size, int target, int& count) {
    if (index >= size) {
        return;
    }
    if (arr[index] == target) {
        count++;
    }
    count_occurrences(arr, index + 1, size, target, count);
}

int main() {
    int arr[] = {1, 3, 2, 3, 5, 3, 7};
    int count = 0;
    count_occurrences(arr, 0, 7, 3, count);
    std::cout << count << std::endl;  // 3
    return 0;
}
```

## Summary

- Recursion = a function that calls itself with a smaller problem
- Every recursive function needs a base case and a recursive case
- Single-parameter recursion: one parameter shrinks toward the base case (factorial, fibonacci, countdown)
- Multi-parameter recursion: multiple parameters change together (binary search, palindrome check, array traversal)
- Tail recursion with references: pass the result forward as a reference parameter instead of building it up on the way back
  - The recursive call is the last thing the function does
  - The reference lets every call modify the same variable

When in doubt, ask yourself:

1. What is the smallest version of this problem? (base case)
2. How do I make the problem one step smaller? (recursive case)
3. Do I need to carry extra information forward? (additional parameters / references)

# Lab

## Linked List Manager with Recursive Operations

In this lab you will build a small linked list program that combines what we learned about linked list manipulation and recursion. Write everything in a single file `lab7.cpp`.

Use this node definition:

```
struct Node {
    int data;
    Node* next;
};
```

## Part 1: Build the list

Write a function `void append(Node*& head, int val)` that inserts a new node at the end of the list.

In `main`, use `append` to build a list from user input. The user will first enter the number of elements, then the elements themselves.

```
Enter number of elements: 6
Enter 6 values: 4 2 7 2 9 2
```

After building the list, print it in the format `4 -> 2 -> 7 -> 2 -> 9 -> 2 -> nullptr`.

## ░░░░ Part 2: Recursive count

Write a recursive function:

```
int count_value(Node* head, int target);
```

This function should return how many times `target` appears in the list. Do not use any loops.

After building the list, prompt the user for a target value and print the count:

```
Enter target value: 2
Count of 2: 3
```

### ▒▒▒▒ Part 3: Delete all occurrences

Write a function `void delete_all(Node*& head, int target)` that removes all nodes with the given `target` value from the list.

After counting, delete all occurrences of the target and print the resulting list:

```
After deleting all 2s: 4 -> 7 -> 9 -> nullptr
```

### Part 4: Recursive reverse print

Write a recursive function:

```
void print_reverse(Node* head);
```

This function should print the list in reverse order without modifying the list. Do not use any loops. Think about what happens if you print after the recursive call instead of before.

```
Reversed: 9 7 4
```

## ░░░ Part 5: Clean up

Write a recursive function:

```
void free_list(Node*& head);
```

This function should delete every node in the list and set head to nullptr. Do not use any loops.

Call free_list at the end of main before returning.

## Full sample run

```
Enter number of elements: 6
Enter 6 values: 4 2 7 2 9 2
List: 4 -> 2 -> 7 -> 2 -> 9 -> 2 -> nullptr
Enter target value: 2
Count of 2: 3
After deleting all 2s: 4 -> 7 -> 9 -> nullptr
Reversed: 9 7 4
```