



PROGRAMMING FUNDAMENTALS

WEEK 11: FUNCTIONS IN C

TABLE OF CONTENTS

- Question: Use of brackets in C
- What is a function?
- 4 Types of Functions
- "Passing by Reference" (using pointers)
 - Recursive Functions
 - Assignment

BRACKETS IN C

SYMBOL	NAME	PURPOSE IN C	EXAMPLE
()	Parentheses	<ul style="list-style-type: none">- Function calls- Function parameters- Grouping expressions	printf("Hi"); int add(int a, int b) x = (a + b) * 2;
{ }	Curly Braces	<ul style="list-style-type: none">- Define blocks of code (functions, loops, conditionals)- Group statements	if (x>0) { printf("Pos"); } int main() { ... }
[]	Square Brackets	<ul style="list-style-type: none">- Declare and access arrays- Specify size or index	int arr[5]; arr[0] = 10;

WHAT IS A FUNCTION

Definition:

- A function is a reusable block of code that performs a specific task.

Why use functions?

- Avoid repetition
- Make code modular & readable
- Easier to debug and test

WHAT IS A FUNCTION

Basic Syntax:

```
return_type function_name(parameter_list) {  
    // body  
    return value; // if return_type is not void  
}
```

WHAT IS A FUNCTION

Example::

```
void greet() {  
    printf("Hello!\n");  
}
```

TYPE 1 – NO ARGUMENTS, NO RETURN

When to use: When a task needs no input and gives no output (e.g., print a menu).

Example:

```
void printWelcome() {  
    printf("Welcome to C Programming!\n");  
}  
  
int main() {  
    printWelcome(); // Call the function  
    return 0;  
}
```

TYPE 1 – NO ARGUMENTS, NO RETURN

Key Points:

- Return type = void
- Parameter list = empty ()
- Called with `function_name();`

TYPE 2 – NO ARGUMENTS, WITH RETURN

When to use: When you generate a value without input (e.g., get current year, roll dice).

Example:

```
int getLuckyNumber() {  
    return 42;  
}  
  
int main() {  
    int num = getLuckyNumber();  
    printf("Your lucky number is %d\n", num);  
    return 0;  
}
```

TYPE 2 – NO ARGUMENTS, WITH RETURN

Key Points:

- Return type ≠ void (e.g., int, float)
- Still no parameters
- Must return a value of matching type

TYPE 3 – WITH ARGUMENTS, NO RETURN

When to use: When you process input but don't need to send back a result (e.g., print sum).

Example:

```
void printSum(int a, int b) {  
    int s = a + b;  
    printf("Sum = %d\n", s);  
}  
  
int main() {  
    printSum(10, 20); // Output: Sum = 30  
    return 0;  
}
```

TYPE 3 – WITH ARGUMENTS, NO RETURN

Key Points - Pass-by-Value:

- Copies of a and b are passed
- Changes inside function do not affect original variables

TYPE 4 – WITH ARGUMENTS, WITH RETURN

Most common type!

Example:

```
int add(int x, int y) {  
    return x + y;  
}  
  
int main() {  
    int result = add(5, 7);  
    printf("Result = %d\n", result); // 12  
    return 0;  
}
```

TYPE 4 – WITH ARGUMENTS, WITH RETURN

Why better than Type 3?

- Reusable: you can store, print, or use the result later.
- More flexible

UNDERSTANDING POINTERS IN C

What is a Pointer?

- A pointer is a variable that stores the memory address of another variable.

```
int num = 20;
```

```
int *ptr = &num; // ptr holds the address of num
```

- `&num` → "address of num"
- `*ptr` → "value at the address stored in ptr" (i.e., 20)

WHY ARE POINTERS IMPORTANT?

- Modify original variables inside functions (simulate "pass-by-reference")
- Work with arrays, strings, and dynamic memory efficiently
- Build advanced data structures (linked lists, trees, etc.)
- Improve performance (avoid copying large data)
- Without pointers, C would be much less powerful!

"PASSING BY REFERENCE" IN C (USING POINTERS)

C has ONLY pass-by-value. But we can simulate reference using pointers.

- Goal: Modify original variable inside a function.
- How? Pass the address (&) and use pointer (*) in function.

"PASSING BY REFERENCE" IN C (USING POINTERS)

Example:

```
void increment(int *p) {      // p is a pointer
    (*p)++;
}

int main() {
    int num = 10;
    printf("Before: %d\n", num);    // 10
    increment(&num);            // Pass address of num
    printf("After: %d\n", num);   // 11 → changed!
    return 0;
}
```

Visual:

num = 10 → address: 1000
increment(&num) → p = 1000
(*p)++ → value at 1000 becomes 11

RECURSIVE FUNCTIONS

Definition:

- A function that calls itself to solve a smaller version of the same problem.

Must have:

- Base case (to stop recursion)
- Recursive case (calls itself with simpler input)

RECURSIVE FUNCTIONS

Example:

- int factorial(int n) {
 if (n == 0) // Base case
 return 1;
 else
 return n * factorial(n - 1); // Recursive call
}
- int main() {
 printf("5! = %d\n", factorial(5)); // 120
 return 0;
}

ASSIGNMENT

1. Write a function called `printCourseInfo()` that takes no arguments and returns nothing. Inside the function, print:
"Course: Programming Fundamentals – Learn, Code, Grow!"
Call this function from `main()`.
2. Write a function `cube(int x)` that takes one integer argument and returns its cube ($x * x * x$). In `main()`, ask the user for a number, call `cube()`, and print the result.

Instructions:
Submit as .c files

ASSIGNMENT

3. Write a function `swap(int *a, int *b)` that swaps the values of two integers using pointers. In `main()`, declare two integers (e.g., `x = 10, y = 20`), print them, call `swap(&x, &y)`, then print again to show they've been swapped.
4. Write a recursive function `sumNatural(int n)` that returns the sum of the first n natural numbers:
$$1 + 2 + 3 + \dots + n$$
Include a proper base case. In `main()`, test it with $n = 5$ (expected output: 15).

Instructions:
Submit as .c files