# Module 3: Building Reusable Blocks of Code (Days 10-13)

## 1. Philosophy Focus: Interleaved Instruction

You could try to master functions in one go, but it's more effective to learn a bit, practice, and then revisit the topic from a new angle. In this module, we'll introduce the basic idea of a function, then interleave that with the concept of "scope," and finally circle back to how functions can return values. This layered approach improves retention.

### **DAY 10: Creating and Calling Simple Functions**

**Goal:** Understand how to package code into a reusable block called a function to avoid repetition.

1. The Problem: Repetitive Code

Imagine you need to print a fancy header for your program in multiple places.

#include <stdio.h>  
  
int main(void) {  
 printf("====================\n");  
 printf(" MY AWESOME PROGRAM\n");  
 printf("====================\n");  
  
 // ... some code ...  
  
 printf("====================\n");  
 printf(" MY AWESOME PROGRAM\n");  
 printf("====================\n");  
   
 // ... more code ...  
 return 0;  
}

This is messy and hard to maintain. If you want to change the header, you have to change it everywhere.

2. The Solution: Functions

A function is a named block of code that you can "call" whenever you need it.

**Inductive Example:**

#include <stdio.h>  
  
// This is a function DEFINITION.  
// It defines what the function does.  
void print\_header(void) {  
 printf("====================\n");  
 printf(" MY AWESOME PROGRAM\n");  
 printf("====================\n");  
}  
  
int main(void) {  
 print\_header(); // This is a function CALL.  
  
 // ... some code ...  
 printf("\nDoing the first task...\n\n");  
  
 print\_header(); // We can call it again!  
   
 // ... more code ...  
 return 0;  
}

**3. Discovering the Pattern**

* **Definition:** return\_type function\_name(parameters) { ... code ... }
  + void return type means this function doesn't send any data back.
  + void in the parentheses means this function doesn't accept any data.
* **Call:** function\_name(); When the program sees this, it jumps to the function's definition, runs the code inside it, and then jumps back to where it left off.
* **Placement:** In C, you must **declare** or **define** a function *before* you use it. That's why we put print\_header above main.

**4. Day 10 Practice**

1. Create a function called print\_menu that prints the menu for your tip calculator or another small program. Call it from main.
2. Write a function say\_goodbye that prints a friendly sign-off message. Call it at the end of main.

### **DAY 11: Passing Data to Functions (Parameters)**

**Goal:** Make functions more flexible by allowing them to accept input data.

**1. Inductive Example: A Flexible Adder**

#include <stdio.h>  
  
// This function accepts two integers as input.  
// 'a' and 'b' are PARAMETERS.  
void add\_and\_print(int a, int b) {  
 int sum = a + b;  
 printf("%d + %d = %d\n", a, b, sum);  
}  
  
int main(void) {  
 add\_and\_print(5, 3); // 5 and 3 are ARGUMENTS.  
   
 int x = 10;  
 int y = 20;  
 add\_and\_print(x, y); // Variables can be arguments too.  
  
 return 0;  
}

**2. Discovering the Pattern**

* **Parameters:** These are the variables declared inside the function's parentheses in its definition (int a, int b). They act as local variables inside the function.
* **Arguments:** These are the actual values passed to the function when you call it (5, 3, x, y). The arguments are copied into the parameters.
* This allows the function to operate on different data each time it's called, making it much more powerful.

**3. Day 11 Practice**

1. **Times Table Function:** Convert your "Times Table" practice from Day 8 into a function void print\_times\_table(int number). The main function should ask the user for a number and then pass that number as an argument to your new function.
2. **Greeting Function:** Write a function void greet\_user(int user\_id) that prints a message like "Hello, User #123!". Call it from main with a few different user IDs.

### **DAY 12: Getting Data Back from Functions (Return Values)**

**Goal:** Learn how a function can compute a result and send it back to the code that called it.

**1. Inductive Example: A Calculator Function**

#include <stdio.h>  
  
// This function's return type is 'int'.  
// It will send an integer result back.  
int calculate\_sum(int a, int b) {  
 int result = a + b;  
 return result; // The 'return' keyword sends the value back.  
}  
  
int main(void) {  
 // The return value of the function is assigned to the 'total' variable.  
 int total = calculate\_sum(10, 15);  
 printf("The sum is: %d\n", total);  
  
 // You can also use the return value directly.  
 printf("Another sum is: %d\n", calculate\_sum(100, 200));  
  
 return 0;  
}

**2. Discovering the Pattern**

* **Return Type:** The keyword before the function name (e.g., int) specifies the type of data the function will return. If it's void, it returns nothing.
* **The return keyword:** When this line is executed, the function immediately stops and sends the specified value back to the caller.
* The calling code can then capture this returned value in a variable or use it directly in an expression.

**3. Day 12 Practice**

1. **Area Function:** Write a function double calculate\_rectangle\_area(double width, double height) that takes width and height, calculates the area, and returns the result as a double. In main, get the width and height from the user, call your function, and print the returned result.
2. **Conversion Function:** Write a function double fahrenheit\_to\_celsius(double fahrenheit) that converts a temperature. The formula is (F - 32) \* 5.0/9.0. In main, call this function with a value like 68 and print the result.

### **DAY 13: Spaced Repetition - Review Project**

**Goal:** Combine everything you've learned about functions to build a modular, clean program.

1. Done-for-you Training Plan: Modular Calculator

Refactor the FizzBuzz program or the Tip Calculator from a previous module to be entirely function-driven.

**Project Requirements:**

1. Create a separate function for each major piece of logic.
2. The main function should be very simple and clean, mostly just calling other functions.

**Example Structure for a Modular Tip Calculator:**

* double get\_bill\_amount(void): This function should prompt the user for the bill, get the input using scanf, and return the value as a double.
* int get\_tip\_percentage(void): This function should prompt for the tip percentage and return it as an int.
* void calculate\_and\_print\_results(double bill, int tip\_percent): This function takes the bill and tip percent as parameters. It performs all the calculations and prints the final formatted output. It has a void return type because its job is just to print.

**main function would then look like this:**

int main(void) {  
 double bill = get\_bill\_amount();  
 int percent = get\_tip\_percentage();  
 calculate\_and\_print\_results(bill, percent);  
 return 0;  
}

Look how readable that is! This is the power of functions. They allow you to hide complexity and organize your code into logical, understandable blocks.