

Programming Fundamentals for Engineers - 0702113

7. Functions

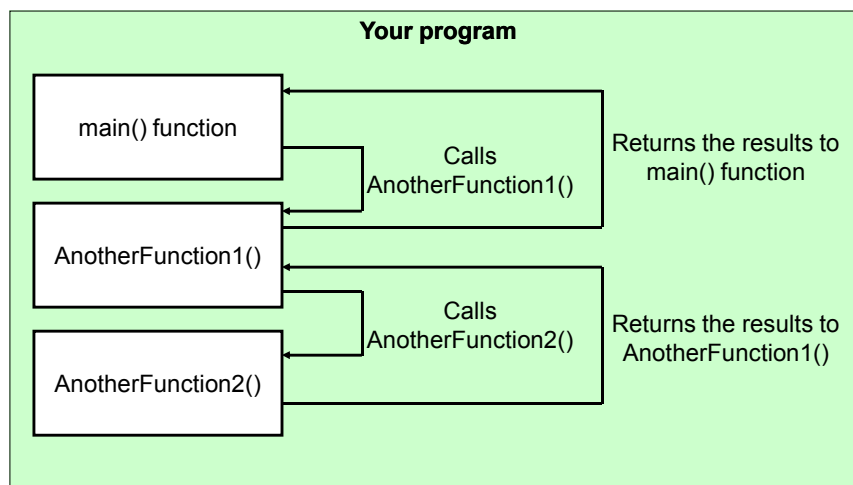
MUNTASER ABULAFI

YACOUB SABATIN

DR. LABIB ARAFEH

1

Modular programming



2

Functions - C's Building Blocks

- **Divide and conquer:**
 - Construct a program from **smaller** pieces / components;
 - These smaller pieces are called **modules**.
 - Each piece is **more manageable** than the original program.
- A **function** (Modules in C) is simply a chunk of C code (statements) that you have grouped together and given a **name**
- The value of doing this is that you can use that "chunk" of code **repeatedly** simply by writing its name;

3

- Programs combine *user-defined functions* with *library functions*;
- C standard library has a wide variety of functions
- C functions are the equivalent of what in other languages would be called *subroutines* or *procedures*;
- Functions should always be declared **prior** to its use to allow compiler to perform type checking on the arguments used in its call;

4

Benefits of functions

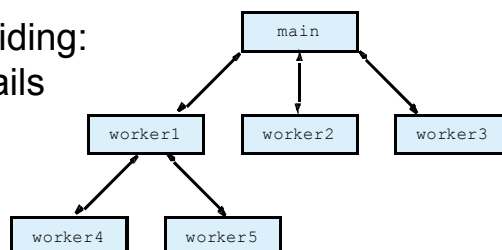
- **Divide and conquer:** Manageable program development (easier to understand and maintain);
- **Software reusability:**
 - Use existing functions as building blocks for new programs;
 - Abstraction - hide internal details (library functions)
- **Avoid code repetition.**
- Different programmers working on **one large project** can divide the workload by writing different functions.

5

• **Function calls**

- Invoking functions
 - Provide function name and arguments (data)
 - Function performs operations or manipulations
 - Function returns results
- Function call analogy:
 - Boss asks worker to complete task
 - Worker gets information, does task, returns result
 - Information hiding:

boss does not know details
(encapsulation)



6

Defining Functions

- Functions have a basic structure:
*return_value_type function_name (arguments,
arguments/Parameter_list)*

```
{
    function_body → declarations & statements
}
```
- Functions must be declared and defined so that they can be called.
- Return-value-type: data type of the result (default **int**)
 - **void** → Indicates that the **function returns nothing**.
- Function-name: any valid **identifier**

7

Summarizing Functions' main features:

1. **RETURN TYPE** is the data type of the **RETURN VALUE** of the function;
2. **RETURN VALUE** is the value that is passed back to the main program; after which Functions exit.
3. **FUNCTION NAME** is the identifier to the function, so that the computer knows which function is called;
4. Functions can take **ARGUMENTS / Parameter List** - a function might need extra information for it to work; Arguments are optional;
5. **FUNCTION BODY** is surrounded by curly brackets & contains statements of the function;

8

- **Math library functions:**
 - Perform common mathematical calculations;
 - `#include <math.h>`
- **Format for calling functions**
 - `FunctionName(argument)`, If multiple arguments, use comma-separated list
 - **`printf("%.2f", sqrt(900.0));`**
 - Calls function **`sqrt`**, which **returns the square root of its argument** (900.0 in this case).
 - All math functions return data type **double**
 - Arguments may be **constants, variables, or expressions.**

9

Function	Description	Example
<code>sqrt(x)</code>	square root of x	<code>sqrt(900.0)</code> is 30.0 <code>sqrt(9.0)</code> is 3.0
<code>exp(x)</code>	exponential function e^x	<code>exp(1.0)</code> is 2.718282 <code>exp(2.0)</code> is 7.389056
<code>log(x)</code>	natural logarithm of x (base e)	<code>log(2.718282)</code> is 1.0 <code>log(7.389056)</code> is 2.0
<code>log10(x)</code>	logarithm of x (base 10)	<code>log10(1.0)</code> is 0.0 <code>log10(10.0)</code> is 1.0 <code>log10(100.0)</code> is 2.0
<code>fabs(x)</code>	absolute value of x	<code>fabs(5.0)</code> is 5.0 <code>fabs(0.0)</code> is 0.0 <code>fabs(-5.0)</code> is 5.0
<code>ceil(x)</code>	rounds x to the smallest integer not less than x	<code>ceil(9.2)</code> is 10.0 <code>ceil(-9.8)</code> is -9.0
<code>floor(x)</code>	rounds x to the largest integer not greater than x	<code>floor(9.2)</code> is 9.0 <code>floor(-9.8)</code> is -10.0
<code>pow(x, y)</code>	x raised to power y (x^y)	<code>pow(2, 7)</code> is 128.0 <code>pow(9, .5)</code> is 3.0
<code>fmod(x, y)</code>	remainder of x/y as a floating point number	<code>fmod(13.657, 2.333)</code> is 1.992
<code>sin(x)</code>	trigonometric sine of x (x in radians)	<code>sin(0.0)</code> is 0.0
<code>cos(x)</code>	trigonometric cosine of x (x in radians)	<code>cos(0.0)</code> is 1.0
<code>tan(x)</code>	trigonometric tangent of x (x in radians)	<code>tan(0.0)</code> is 0.0
10	Commonly used math library functions.	

Example for a user-defined function:

```
int squareNumber(int a)
{
  int b = a*a;
  return b;
}
```

- *squareNumber* → the **name** of this function;
- Because an integer is returned, the *int* keyword must be placed **before** the function name

11

- If the function does not return a value, we put the **void** keyword before the function name
- This function has one argument, which is of the type **int**. If we have arguments, we must put variable declarations in the round brackets ()
- The function body consists of 2 statements:
 - The first, sees an **int** variable **b** declared and assigned **a*a**, i.e. **a** squared.
 - The second statement uses the return keyword to **pass** the value of **b** back into the main program, hence **exiting** the function

12

- Within the program, one might write:
`x = squareNumber(5);`
- This would assign 25 to **x**. We say that **5** is passed as an argument to the function **squareNumber**
- The variables within the **squareNumber** function are **LOCAL VARIABLES** - when the function exits, variables **a** and **b** are deleted from memory
- **int squareNumber(int a)** is also known as the **FUNCTION HEADER**

13

- ***return type***: type of data returned each time function is called (*int, float, char, etc.....*);
- By default, *return type* is ***int*** if it was omitted;
- Can be ***void*** if function is **NOT** meant to return a value (*example: function to print an output*)

14

Another example:

```
float average(float a, float b)
```

```
{
    return (a + b) / 2;
}
```

A separate type must be specified for each parameter EVEN if all parameters have same type

(float a, b) is WRONG!

- *average* → the **name** of this function;
- Because a float is returned, the *float* keyword must be placed **before** the function name.
- *a* and *b* are input parameters.
- The expression *(a + b) / 2* is **evaluated** then **returned** by this function to the function that called it.

15

```
/* Prints "T minus n and counting" 10 times using for
loop*/
#include <stdio.h>
void print_count(int n) // function doesn't return value
{
    printf("T minus %d and counting\n", n);
}
main()
{
    int i;
    for (i = 10; i > 0; --i)
        print_count(i); /* call print_count, it is a statement,
                           i occupied into n and takes its
                           value */

    return 0;
}
```

16

Output

T minus 10 and counting
 T minus 9 and counting
 T minus 8 and counting
 T minus 7 and counting
 T minus 6 and counting
 T minus 5 and counting
 T minus 4 and counting
 T minus 3 and counting
 T minus 2 and counting
 T minus 1 and counting

```

T minus 10 and counting
T minus 9 and counting
T minus 8 and counting
T minus 7 and counting
T minus 6 and counting
T minus 5 and counting
T minus 4 and counting
T minus 3 and counting
T minus 2 and counting
T minus 1 and counting
Press any key to continue_
  
```

- Note that this function **does not** return a value, it just prints something.
- In this case we specify its return type as **void**.

17

Example:

//A function that tests whether a number is prime

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

```
#define TRUE 1
```

```
#define FALSE 0
```

```
typedef int Bool;
```

```
Bool is_prime(int n)
```

```
{
```

```
  int divisor;
```

```
  if (n <= 1) return FALSE; //Not prime
```

```
  for (divisor = 2; divisor * divisor <= n; divisor++)
```

```
    if (n % divisor == 0) /* If n over divisor equals 0 – means divisor is a factor - Not prime */
```

```
      return FALSE;
```

```
  return TRUE; //Else
```

```
}
```

```

//main function
main()
{
    int x;
    printf("Enter a number: ");
    scanf("%d", &x);
    if (is_prime(x)) //If TRUE, then prime
        printf("Prime\n");
    else //FALSE
        printf("Not prime\n");
    return 0;
}

```

19

Function Declarations

- Until now, we have put function definitions before their calls.
This is not sometimes possible.....!
- *When compiler sees a function call prior to it's definition...compiler is forced to assume:*
 1. *return_type* of function is *int*
 2. call is supplying correct parameters number
 3. argument types will match those of parameters
- Such assumptions may be *wrong!*
- Can avoid such problem by declaring functions (called *prototypes*).

20

- This consists of the 1st line of the function definition.
return_type function_name (parameters);
- This declaration will inform compiler about function return type and arguments.
- Parameters and arguments refer to similar things, but:
 - **Parameters** → Appear in function **definition**.
 - **Arguments** → Expressions appear in function **calls**.

21

```

/* Try the Code that creates & uses a user-defined
   Function */
#include <stdio.h>
int square( int y ); /* function prototype */
main()
{
    int x; // Calculate & output square of x from 1 to 10.
    for ( x = 1; x <= 10; x++ )
    {
        printf( "%d ", square( x ) ); /* function call */
    } /* end for */
    printf( "\n" );
    return 0;
}

```

```
// square function returns square of an integer
int square( int y ) // y is a copy of argument to function
{
    return y * y; // returns square of y as an int
} /* end function square */
```

```
1 4 9 16 25 36 49 64 81 100
```

23

```
// Try this code that finds the Max. of 3 numbers
#include <stdio.h>
int maximum( int x, int y, int z ); //function prototype
int main()
{
    int num1; /* first integer */
    int num2; /* second integer */
    int num3; /* third integer */
    printf( "Enter three integers: " );
    scanf( "%d,%d,%d", &num1, &num2, &num3 );
    printf( "Maximum is: %d\n", maximum( num1,
                                         num2, num3 ) );

    return 0;
}
```

24

```

int maximum( int x, int y, int z )
{
    int max = x; // assume x is largest
    if ( y > max ) // if y > max, assign y to max
        max = y;
    if ( z > max ) // if z > max, assign z to max
        max = z;
    return max; // max is largest value
}

```

```

Enter three integers: 22 85 17
Maximum is: 85

```

```

Enter three integers: 85 22 17
Maximum is: 85

```

```

Enter three integers: 22 17 85
Maximum is: 85

```

25

- **Function definition format: (recalled)**
return-value-type function-name(parameter-list)
{ declarations and statements }
- **Definitions and statements:** function body (block)
 - Variables can be defined **inside** blocks → Can be nested!
 - A functions **can call** another function.
 - Functions **can not be defined inside** other functions
- Returning control to the calling function:
 - If nothing returned → at *return;*
OR
→ until reaches right brace '}'
 - If something returned → at *return expression;*

26

- **Function prototype**

- Return type;
- Function name;
- Parameters;
- Used to validate functions;
- Prototype only needed if function definition comes after use in program (ex. after main)
- The function with the prototype

int maximum(int x, int y, int z);

Or **int maximum(int, int, int);**

Takes in **3 int's** & Returns an **int**.

- Promotion rules and conversions

- z7 Converting to lower types can lead to errors

Function Prototypes

Data types	printf conversion specifications	scanf conversion specifications
long double	%Lf	%Lf
double	%f	%lf
float	%f	%f
unsigned long int	%lu	%lu
long int	%ld	%ld
unsigned int	%u	%u
int	%d	%d
short	%hd	%hd
char	%c	%c
Promotion hierarchy for data types.		

- **Header files**
 - Contain function prototypes for library functions
 - `<stdlib.h>`, `<math.h>` , etc
 - Load with `#include <filename>` →
`#include <math.h>`
- **Custom header files`**
 - Create file with functions
 - Save as `filename.h`
 - Load in other files with `#include "filename.h"`
 - Reuse functions / reusability
 - Modular programming.

29

Standard library header	Explanation
<code><assert.h></code>	Contains macros and information for adding diagnostics that aid program debugging.
<code><ctype.h></code>	Contains function prototypes for functions that test characters for certain properties, and function prototypes for functions that can be used to convert lowercase letters to uppercase letters and vice versa.
<code><errno.h></code>	Defines macros that are useful for reporting error conditions.
<code><float.h></code>	Contains the floating point size limits of the system.
<code><limits.h></code>	Contains the integral size limits of the system.
<code><locale.h></code>	Contains function prototypes and other information that enables a program to be modified for the current locale on which it is running. The notion of locale enables the computer system to handle different conventions for expressing data like dates, times, dollar amounts and large numbers throughout the world.
<code><math.h></code>	Contains function prototypes for math library functions.
<code><setjmp.h></code>	Contains function prototypes for functions that allow bypassing of the usual function call and return sequence.
<code><signal.h></code>	Contains function prototypes and macros to handle various conditions that may arise during program execution.
<code><stdarg.h></code>	Defines macros for dealing with a list of arguments to a function whose number and types are unknown.
<code><stddef.h></code>	Contains common definitions of types used by C for performing certain calculations.
<code><stdio.h></code>	Contains function prototypes for the standard input/output library functions, and information used by them.
<code><stdlib.h></code>	Contains function prototypes for conversions of numbers to text and text to numbers, memory allocation, random numbers, and other utility functions.
<code><string.h></code>	Contains function prototypes for string processing functions.
<code><time.h></code>	Contains function prototypes and types for manipulating the time and date.
Some of the standard library header.	
30	

Example

Problem: Output a multiplication table for numbers in an arbitrary range?

Reasoning: The problem calls for the output of a **table** whose values equal the **product** of the rows and column labels. For instance, for the values from 4 to 7 the corresponding output might look like the following:

	4	5	6	7
4	16	20	24	28
5	20	25	30	35
6	24	30	36	42
7	28	35	42	49

31

Analysis:

Input: Low_range, High_range

Output: Multip_Table

Process: Multip_Tab[Row][Column] = Row * Column;

Algorithm:

1. Start;

2. Input Low_range, High_range;

3. LOOP Row: Low_range TO High_range

 1. LOOP Column: Low_range TO High_range;

 1. Output Row * Column;

4. End.

32

```

//Main parts of the C Code:
#include <stdio.h>
//Prototypes:
void print_heading (int Low_range, int High_range);
void print_table (int Low_range, int High_range);
//Main function:
Main ()
{
    int Low_range, High_range;
    printf("\n Enter Low end & High end values : ");
    scanf("%d %d", &Low_range, &High_range);
    print_heading (Low_range, High_range) // Calling
    print_table (Low_range, High_range); // Calling
    return 0;
}

```

```

void print_heading (int Low_range, int High_range);
{
    int Col; // Column index

    for (Col = Low_range; Col <= High_range; Col++)
        printf("%3d ", Col);

    printf("\n");

    for (Col = Low_range; Col <= High_range; Col++)
        printf("  ");

    printf("\n");
}

```

34

```

void print_table (int Low_range, int High_range);
{
    int Row, Column;
    for (Row = Low_range; Row <= High_range; Row++)
    {
        printf("%3d|", Row); //For the
        for ( Column = Low_range; Column <=
            High_range; Column++ )
            printf("%3d", Row * Column);

        printf("\n");
    }
}

```

35

Examples

1. Develop a function that returns a '0' if parameter2 is a multiple of parameter1. Otherwise return a '1'.

```

int find_value( int parameter1, int parameter2)
{
    if ( (parameter2 % parameter1) == 0 )
        return 0;
    else
        return 1;
}

```

36

2. Develop a function that displays whether a character is a vowel or a consonant?

```
void test_char (char ch)
{
    if (ch=='A' || ch=='E' || ch=='I' || ch=='O' || ch=='U')
        printf("\n It's a Vowel.");
    else
        printf("\n It's not a Vowel.");
}
```

37

3. Develop a function to compute and return n factorial defined as:

$$n! = n * (n-1) * (n-2) * (n-3) * \dots * 2 * 1$$

```
int factorial (int n)
{
    int i, product = 1;
    for (i = 1; i <=n; i++)
        product *=i;
    return (product);
}
```

38

4. Develop a prototype function to compute the net pay based on wage rate and hours worked and take deductions for federal tax (28%) & social security (5.5%)?

Input parameters: wage, hours

Return value: net_pay

Processes:

gross_pay = wage * hours

fed_tax = Fed_Tax_Rate * gross_pay

soc_security = Soc_Sec_Rate * gross_pay

net_pay = gross_pay – (fed_tax + soc_security)

39

```
float cal_net_pay (float wage, int hours)
{
    float gross_pay, fed_tax, soc_security, net_pay;
    const float Fed_Tax_Rate = 0.28;
    const float Soc_Sec_Rate = 0.055;
    gross_pay = wage * hours;
    fed_tax = Fed_Tax_Rate * gross_pay;
    soc_security = Soc_Sec_Rate * gross_pay;
    net_pay = gross_pay – (fed_tax + soc_security);
    return (net_pay);
    //OR: return ((wage*hours)(1-0.28-0.055)); in one line!
}
```

40

```

// a C code might look like:
#include <stdio.h>
float cal_net_pay (float, int);
main()
{
    float wage, net_pay;
    int hours;

    scanf("%f %d", &wage, &hours);
    net_pay = cal_net_pay (wage, hours);
    printf("\n$%8.2f", net_pay);

    return 0;
}

```

41

5. Develop a C function to calculate an employee's gross pay based on the wage rate, and number of hours worked. Allow for overtime wages of 1.5 times the normal rate for all overtime hours worked?

- **Analysis:**

Input: wage, hours

Output: gross

Processes:

gross = wage * hours + overtime (if any)

42

Algorithm:**Step 1: Input wage, hours;****Step 2: If hours <= regular_hours****gross = wage * hours****Else****a. overtime_hours =
 hours – regular_hours;****b. overtime_pay =
 overtime_hours * overtime_rate****c. gross =
 wage * regular_hours + overtime_pay****Step 3: Output gross****Step 4: Stop.****Step 2 → Uses a function to compute the gross pay.**

43

```

float calc_net_pay (int wage, int hours)
{
    const int Regular_Hours = 40;
    const float Over_Time_Rate = 1.5;
    float gross_pay, overtime_hours, overtime_pay;

    if (hours <= Regular_Hours)
        gross_pay = wage * hours;
    else
    {
        overtime_hours = hours – Regular_Hours;
        overtime_pay = overtime_hours * Over_Time_Rate;
        gross_pay = wage * Regular_Hours + overtime_pay;
    }
    return (gross_pay);
}

```

44

6. Develop a function to compute and return n factorial defined as:

$$n! = n * (n-1) * (n-2) * (n-3) * \dots * 2 * 1$$

Using recursion.

```
int fact (int n)
{
    if (n<=1)
        return 1;
    else
        return n * fact(n-1);
}
```

45

- Trace the execution of above example with this statement: $i = \text{fact}(4)$;
- A function is recursive if it calls itself.
- Unfinished calls of the recursive function “*pile up*” until it returns a normal value.
- Be careful to test a “*termination condition*” to prevent *infinite recursion*.
- Another example:


```
int power(int x, int n)
{
    return n==0 ? 1 : x*power(x,n-1);
}
```
- This function computes x^n using the formula:

$$x^n = x * x^{n-1}.$$