BIRZEIT UNIVERSITY

Top-Down Design with Functions

PROBLEM SOLVING AND PROGRAM DESIGN In C
7th EDITION
Jeri R. Hanly, Elliot B. Koffman

By: Mamoun Nawahdah (PhD)
2013/2014

Overview

- **C** Functions.
- Types of Functions.
  - **void** Functions without Arguments.
  - **void** Functions with Arguments.
  - Functions with arguments.
- Advantages of Using Function Subprograms.
  - Procedural Abstraction.
  - Reuse of Functions.

C Functions

- A **C** program is a set of functions.
- Every **C** program must contain one function called **main**. This is where the program starts.
- Function **main** calls other functions.
- So far we wrote **C** programs with **main** function only.
- We also have seen predefined library functions being used in **main** function.
- Now we are going to create our **own** functions.

Example

```
#include <stdio.h>
#define PI 3.14159
double find_area(double r);
int main(void){
        double radius, area;
        printf("Enter radius> ");
        scanf("%lf", &radius);
        area = find_area(radius);
        printf("The area is %.2f\n",area );
        return(0);
}
double find_area(double r){
        double a = PI * r * r;
        return (a);
}
```

Return Type

Function arguments

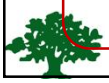
Function Declaration/Prototype

Function call

Function Header

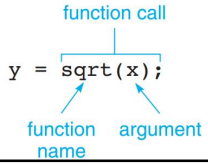
Function Body

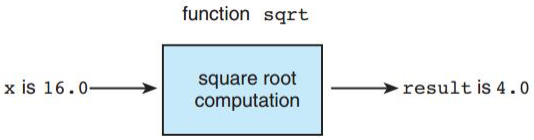
Function Definition



Arguments

- We can use function arguments to communicate with the function:
  - **Input arguments:** ones that used to pass information from the caller to the function.
  - **Output arguments:** ones that return results to the caller from the function.





C Library Functions

Function	Standard Header File	Purpose: Example	Argument(s)	Result
`abs(x)`	`<stdlib.h>`	Returns the absolute value of its integer argument: if `x is -5`, `abs(x)` is 5	`int`	`int`
`ceil(x)`	`<math.h>`	Returns the smallest integral value that is not less than `x`: if `x` is `45.23`, `ceil(x)` is `46.0`	`double`	`double`
`cos(x)`	`<math.h>`	Returns the cosine of angle `x`: if `x` is `0.0`, `cos(x)` is `1.0`	`double` (radians)	`double`
`exp(x)`	`<math.h>`	Returns e^x where e = `2.71828...`: if `x` is `1.0`, `exp(x)` is `2.71828`	`double`	`double`
`fabs(x)`	`<math.h>`	Returns the absolute value of its type `double` argument: if `x` is `-8.432`, `fabs(x)` is `8.432`	`double`	`double`
`floor(x)`	`<math.h>`	Returns the largest integral value that is not greater than `x`: if `x` is `45.23`, `floor(x)` is `45.0`	`double`	`double`
`log(x)`	`<math.h>`	Returns the natural logarithm of `x` for `x > 0.0`: if `x` is `2.71828`, `log(x)` is `1.0`	`double`	`double`
`log10(x)`	`<math.h>`	Returns the base-10 logarithm of `x` for `x > 0.0`: if `x` is `100.0`, `log10(x)` is `2.0`	`double`	`double`

C Library Functions

Function	Standard Header File	Purpose: Example	Argument(s)	Result
`pow(x, y)`	`<math.h>`	Returns x^y. If `x` is negative, `y` must be integral: if `x` is `0.16` and `y` is `0.5`, `pow(x,y)` is `0.4`	`double, double`	`double`
`sin(x)`	`<math.h>`	Returns the sine of angle `x`: if `x` is `1.5708`, `sin(x)` is `1.0`	`double` (radians)	`double`
`sqrt(x)`	`<math.h>`	Returns the nonnegative square root of `x` ($\sqrt{x}$) for `x` $\geq$ `0.0`: if `x` is `2.25`, `sqrt(x)` is `1.5`	`double`	`double`
`tan(x)`	`<math.h>`	Returns the tangent of angle `x`: if `x` is `0.0`, `tan(x)` is `0.0`	`double` (radians)	`double`

Types of Functions

- No input arguments, no value returned – **void functions without arguments.**
- Input argument(s), no value returned - **void functions with arguments**.
- Input argument(s), single value returned.
- Input argument(s), multiple value returned.

1- Void Functions without Arguments

- The function just does something without communicating anything back to its caller.
  - Output is normally placed in some place else (e.g. screen).

Function Prototype

```
/* This program draws a circle in the screen */
#include <stdio.h>
/* Function prototypes */
void draw_circle(void);
int main(void){
      draw_circle();
      return (0);
}
/* Draws a circle */
void draw_circle(void) {
      printf("   *   *    \n");
      printf(" *       *  \n");
      printf("   *   *    \n");
}
```

1st void means no value returned, 2nd void means no input arguments.

Function Prototype

- Like other identifiers in **C**, a function **must** be declared before it can be referenced.
- To do this, you can add a function prototype before **main** to tell the compiler what functions you are planning to use.

Function Prototype

- A function prototype tells the **C** compiler:

1. The data type the function will **return**. For example, the **sqrt** function returns a type of **double**.
2. The function **name**.
3. Information about the **arguments** that the function expects.

- The **sqrt** function expects a double argument.
- So the function prototype for **sqrt** would be:

```
double  sqrt ( double );
```

Function Definition

```
/* This program draws a circle in the screen */
#include <stdio.h>
/* Function prototypes */
void draw_circle(void);
int main(void){
      draw_circle();
      return (0);
}
/* Draws a circle */
void draw_circle(void) {
      printf("   *  *    \n");
      printf(" *      *   \n");
      printf("   *  *    \n");
}
```

❖ The prototype tells the compiler what arguments the function takes and what it returns, but not what it does.

Function Definition

❖ Function definition consists of:

- **Function Header**: The same as the prototype, except it is not ended by the symbol ➔ **;**
- **Function Body**: A code block enclosed by **{ }**, containing variable declarations and executable statements.

❖ In the function body, we define what actually the function does.

Placement of Functions

- In general, we declare all of our function prototypes at the beginning (after **#include** or **#define**).
- This is followed by the **main** function. After that, we define all of our functions.
- However, this is just a convention.
- As long as a function's prototype appears before it is used, it doesn't matter where in the file it is defined.
- The order we define them in does not have any impact on how they are executed

Execution Order of Functions

- Execution order of functions is determined by the order of execution of the function call statements.
- Because the prototypes for the function subprograms appear before the **main** function, the compiler processes the function prototypes before it translates the **main** function.

Execution Order of Functions

- The information in each prototype enables the compiler to correctly translate a call to that function.
- After compiling the **main** function, the compiler translates each function subprogram.
- At the end of a function, control always returns to the point where it was called.

Flow of Control Between the Main Function and a Function Subprogram

```
int main(void){
    draw_circle();

    return (0);
}
```

```
/* Draws a circle */
void draw_circle(void) {
    printf(" * *\n");
    printf("*    *\n");
    printf(" * *\n");
}
```

A Good Use of Void Functions – A Separate Function to Display Instructions for the User.

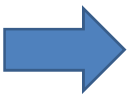
FIGURE 3.16 Function instruct and the Output Produced by a Call

```
1.  /*
2.   * Displays instructions to a user of program to compute
3.   * the area and circumference of a circle.
4.   */
5.  void
6.  instruct(void)
7.  {
8.        printf("This program computes the area\n");
9.        printf("and circumference of a circle.\n\n");
10.       printf("To use this program, enter the radius of\n");
11.       printf("the circle after the prompt: Enter radius>\n");
12. }
```

2- Void Functions with Input Arguments

- A void function does not **return** a result, but we can still pass it arguments.
- For example, we could have a function display its argument value in a more attractive way.
- The effect of the function call **print_rboxed(135.68);**

```
1.  /*
2.   * Displays a real number in a box.
3.   */
4.
5.  void
6.  print_rboxed(double rnum)
7.  {
8.       printf("***********\n");
9.       printf("*         *\n");
10.      printf("* %7.2f *\n", rnum);
11.      printf("*         *\n");
12.      printf("***********\n");
13. }
```



```
***********
*         *
*  135.68 *
*         *
***********
```

Actual Arguments & Formal Parameters

- **Actual argument**: an expression used inside the parentheses of a **function call**.
- **Formal parameter**: An identifier that represents a corresponding **actual** argument in a **function definition**.

```
print_rboxed(135.68);
```

Call `print_rboxed` with `rnum = 135.68`

```
void
print_rboxed(double rnum)
{
        printf("***********\n");
        printf("*         *\n");
        printf("* %7.2f *\n", rnum);
        printf("*         *\n");
        printf("***********\n");
}
```

Arguments make functions more versatile because they enable a function to manipulate different data each time it is called.

3. Functions with Input Arguments and a Single Result

- We can call these functions in expressions just like library functions.
- Let's consider the problem of finding the area and circumference of a circle using functions with just one argument.

```
#define PI 3.14159
/* Compute the circumference of a circle with radius r */
double find_circum( double r)  {
        return (2.0 * PI * r);
}
/* Compute the area of a circle with radius r */
double find_area( double r ) {
        return (PI * pow(r,2));
}
```

Functions with Input Arguments and a Single Result

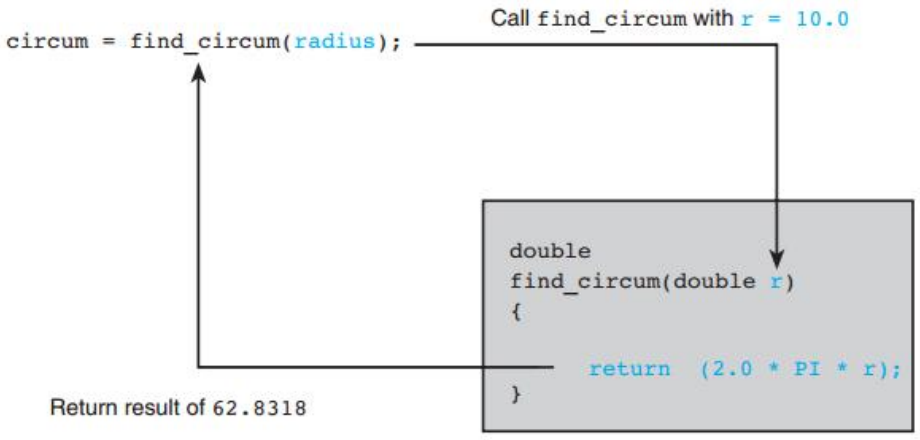
- Each function heading begins with the word **double**, indicating that the function result is a real number.
- Both function bodies consist of a single **return** statement.
- When either function executes, the expression in its **return** statement is evaluated and returned as the function result.
- If we call the function like:

double area = find_area(5.0);

- The value we returned from the function will be assigned to **area** variable.

Effect of Executing

```
circum = find_circum(radius);
```



More on Functions

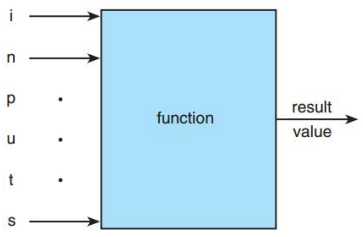
- Make sure that you understand the difference in function calls between **void** functions and functions that returns a single value.

```
draw_circle();
print_rboxed(135.68);
area = find_area(5.0);
```

- A function call that returns a result must do something with the result, otherwise, the value returned will be lost.

Functions with Multiple Arguments

- We can also define functions with multiple arguments.




```
double
scale(double x, int n)
{
        double scale_factor;

        scale_factor = pow(10, n);

        return (x * scale_factor);
}
```

- Function call **scale(2.5, 2)** returns the value **250.0**

Argument List Correspondence

- When using multiple-argument functions, the **number** of **actual argument** used in a function call must be the same as the number of **formal parameters** listed in the function prototype.
- The **order** or the **actual arguments** used in the function call must correspond to the order of the parameters listed in the function prototype.
- Each **actual argument** must be of a data type that can be assigned to the corresponding **formal parameter** with no unexpected loss of information.

```
#include <stdio.h>            /* printf, scanf definitions */
#include <math.h>             /* pow definition */

/* Function prototype */
double scale(double x, int n);

int
main(void)
{
      double num_1;
      int num_2;

      /* Get values for num_1 and num_2 */
      printf("Enter a real number> ");
      scanf("%lf", &num_1);
      printf("Enter an integer> ");
      scanf("%d", &num_2);

      /* Call scale and display result. */
      printf("Result of call to function scale is %f\n",
             scale(num_1, num_2));

      return (0);
}
```

The Function Data Area

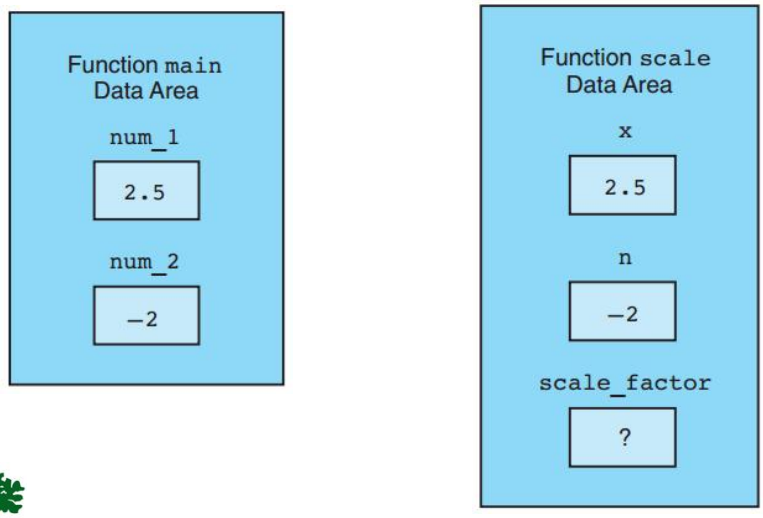
- Each time a function call is executed, an area of memory is allocated for storage of that function's data.
- Included in the function data area, storage cells for its **formal parameters** and any **local variables** that may be declared in the function.
- **Local Variables**: variable declarations within a function body.
  - Can only be used from within the function they are declared in – no other function can see them.
  - These variables are created only when the function has been activated and become undefined after the call.

The Function Data Area

- The function data area is always **lost** when the function terminates.
- It is recreated empty when the function is called again.
- So if you set a local variable value, that value will not still be set next time the function is called.

Data Areas After Call

```
scale(num_1,num_2);
```



Testing Functions Using Drivers

- A function is an independent program module.
- As such, it can be tested separately from the program that uses it.
- To run such a test, you should write a short piece of code called **driver** that
  - Defines the function arguments,
  - Calls the functions, and
  - Displays the value returned.

Why do we Use Functions?

❖ There are two major reasons:

1. A large problem can be solved **easily** by breaking it up into several small problems and giving the responsibility of a set of functions to a specific programmer.
   - It is easer to write two 10 line functions than one 20 line one and two smaller functions will be easier to read than one long one.
2. They can simplify programming tasks because existing functions can be **reused** as the building blocks for new programs.
   - Really useful functions can be bundled into libraries.

Procedural Abstraction

❖ **Procedural Abstraction:** A programming technique in which a main function consists of a sequence of function calls and each function is implemented separately.

❖ All of the details of the implementation to a particular sub-problem is placed in a separate function.

Procedural Abstraction

- The main functions becomes a more abstract outline of what the program does.
  - When you begin writing your program, just write out your algorithm in your main function.
  - Take each step of the algorithm and write a function that performs it for you.
- Focusing on one function at a time is much easier than trying to write the complete program at once.

Reuse of Function Subprograms

- Functions can be executed more than once in a program.
- Reduces the overall length of the program and the chance of error.
- Once you have written and tested a function, you can use it in other programs or functions.

Common Programming Errors

- Remember to use a **#include** preprocessor directives for every standard library from which you are using functions.
- Place prototypes for your own function subprogram in the source file preceding the **main** function; place the actual function definitions after the **main** function.

Common Programming Errors

- Providing the required **Number** of arguments.
- Making sure the **Order** of arguments is correct.
- Making sure each argument is the **correct Type** or that conversion to the correct type will lose no information.