C FUNCTIONS

Deitel 8th Edition, Chapter 5

TOPICS

Functions = Reusable Code

C Standard Libraries

Programmer-Defined Functions

Writing & Using Functions

- Function Definitions
- Function Prototypes
- Function Calls

Scope

Communicating with Functions

Common Errors & Other Observations

FUNCTIONS = REUSABLE CODE

Writing Code as a Collection of Reusable Pieces

Programs can be large and complex

Problems not usually unique

Other programs may have existing sections of code that can be reused as building blocks in a new program

Even sections of the same program can be reused, if properly designed

Reusable Blocks of Code → Fewer Errors

Construct a program from smaller blocks, each of which is more manageable than original program

Code reuse can promote goal of writing **error-free code**

Caution!

Only reuse code that has been written by you!

- Or your co-workers on a specific project
- Or that is part of the language

Often source code is protected by copyright

And remember plagiarism…

FUNCTION = REUSABLE BLOCKS OF C CODE

In C, these reusable blocks of code are called **functions**

Three main sources of functions:

1. C Standard Libraries
2. Programmer-defined (you!)
3. Libraries from other sources, or other libraries you create

C Standard Libraries

C Standard Libraries are Part of the C Language

Standard code included as part of the C language

Standard I/O library for input and output – stdio.h

Math library for math-related functions – math.h

String & character manipulations

Many other useful and common tasks

Math Library Functions

Common math calculations, such as using exponents, square roots, etc.

Example – to calculate and print the square root of 900.0 you might write:

```
printf( "%.2f", sqrt( 900.0 ) );
```

Fig. 5.2 Commonly used math library functions

Math Library Example: sqrt Function

1. It needs **input data to act on** (the number to calculate the square root of)

2. It does some **processing** on this input data (calculates square root)

3. It **provides a result** (the square root of the input data)

To use this function, we don’t need to know HOW it processes the input data. We just need to provide it the right input, and then handle the output.

Math Library Example: sqrt of 16

If we want to know the square root of 16, we provide that value in the parentheses to sqrt:

```
sqrt ( 16 );
```

The sqrt function will return a result: **4**

Math Library Example: sqrt of 16 & Saving the Result

If we want to save the result, put this into an assignment statement:

```
result = sqrt ( 16 );
```

The variable **result** will be assigned the value **4**.

MATH LIBRARY EXAMPLE: SQRT OF ANY NUMBER

We can do the same thing with any integer variable, and save the result:

```
x =  25;

result = sqrt ( x );
```

The variable **result** will be assigned the value **5.**

Math Library Example: pow

The function **pow** needs two inputs. It uses the first one as the base, and the second as the exponent.

If we wanted to compute 2^3, we could write:

```
result = pow ( 2, 3 );
```

The variable **result** will be assigned the value **8.**

Precedence of Function Calls

Order	Operator Type	Operator
1	Parentheses	( )
2	**Function calls**	
3	Unary operators	! + — &
4	Binary operators (MDR)	* / %
5	Binary operators (AS)	+ —
6	Relational operators	< <= > >=
7	Equality operators	== !=
8	Logical operators	&& ||
9	Assignment operator	=

Math Library Example: sqrt and pow in a Longer Expression 1

Usually, the result of a function is assigned to a variable

But not always

What is the value of **result** after this statement executes?

```
result = 3 + pow( 6, 2 ) / sqrt( 16 );
```

What are all the parts of this statement?

Math Library Example: sqrt and pow in a Longer Expression 2

result = 3 + pow(6, 2) / sqrt(16);

There are 3 parts:

1. Constant **3**

Math Library Example: sqrt and pow in a Longer Expression 3

result = 3 + pow(6, 2) / sqrt(16);

There are 3 parts:

1. Constant 3
2. Expression **pow(6, 2)**

Math Library Example: sqrt and pow in a Longer Expression 4

```
result = 3 + pow( 6, 2 ) / sqrt( 16 );
```

There are 3 parts:

1. Constant 3
2. Expression pow(6, 2)
3. Expression **sqrt(16)**

Each part must be solved separately before we can combine them to get the result.

The expressions are **function calls**.

Math Library Example: sqrt and pow in a Longer Expression 5

result = 3 + pow(6, 2) / sqrt(16);

Result of each part:

1. Constant **3 = 3**

Math Library Example: sqrt and pow in a Longer Expression 6

result = 3 + pow(6, 2) / sqrt(16);

Result of each part:

1. Constant 3 = 3
2. **pow(6, 2) = 36**

pow() is a function, so it is called with input data 6 and 2.

pow(6, 2) means 6 raised to the power of 2:

6^2 = **36**

Math Library Example: sqrt and pow in a Longer Expression 7

```
result = 3 + pow( 6, 2 ) / sqrt( 16 );
```

Result of each part:

1. Constant 3 = 3
2. pow(6, 2) = 36
3. **sqrt(16) = 4**

sqrt() is a function, so it is called with input data 16.

sqrt(16) means the square root of 16:

$\sqrt{16} = 4$

Math Library Example: sqrt and pow in a Longer Expression 8

Original expression:
result = 3 + pow(6, 2) / sqrt(16);

Function calls replaced w/results:
result = 3 +   36   /   4   ;

Adjusted spacing:
result = 3 + 36 / 4;

Replacing the function calls with their results gives an expression that can be evaluated using the normal order of operations.

Math Library Example: sqrt and pow in a Longer Expression 9

result = 3 + 36 / 4;

1. 36 / 4 = 9

result = 3 + 9;

2. 3 + 9 = 12

result = 12

3. result = 12

Using Math Library Functions in C Program

To use sqrt, pow and other math functions, reference C math library using a preprocessor directive:

```
#include <math.h>
```

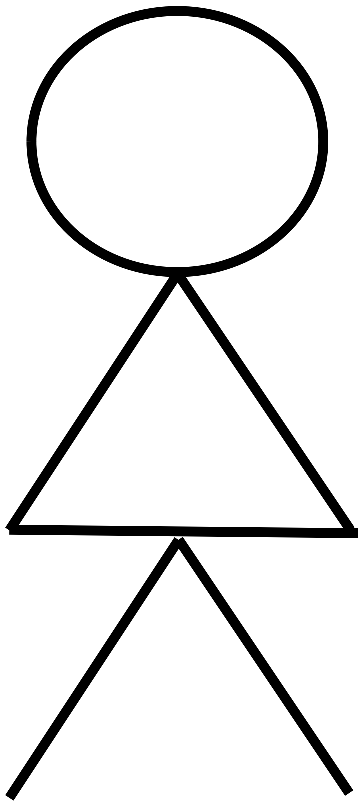
Programmer-Defined Functions

Programmers Can Define Our Own Functions Too

Example: Write a program to draw a stick figure

What are the "subproblems" of drawing a stick figure?

- i.e. what is a stick figure made up of?



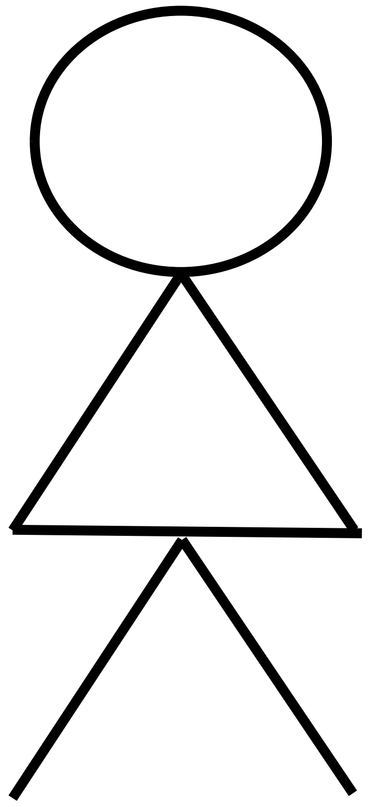
Steps in Drawing Stick Figure 1

Step 1: draw circle for the head

Step 2: draw triangle for the body

Step 3: draw intersecting lines for legs

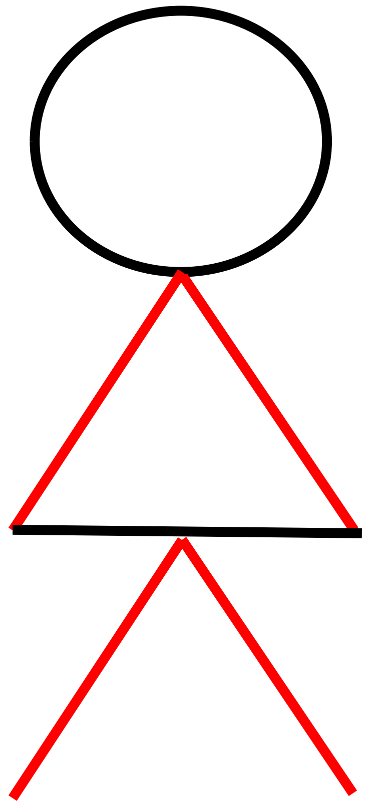
Is there any part of the figure that is used in more than one place?



Steps in Drawing Stick Figure 2

Part of the triangle duplicates the lines for the legs.

It makes sense to write that code once, and then reuse it where necessary



WRITING & USING FUNCTIONS

Three Steps to Using a Function

1. Write the function **definition**
2. Write the function's **prototype**
3. Write the code to **call** the function

Three Steps to Using a Function – The Order of These Steps

1. Write the function **definition**
2. Write the function’s **prototype**
3. Write the code to **call** the function

You can write the code for Steps 1 & 2 in any order, but the code for Step 3 is written last

FUNCTION DEFINITIONS

Step 1 – Write the Function Definition

Statements that define the function’s operation(s)

Function definition includes four parts:

1. Header (a.k.a. signature)
2. Opening curly brace
3. Body of function (C control structures that make up the function’s work)
4. Closing curly brace

The above should be familiar…

- Same as the parts of “main”

Header of a Function Has 3 Parts

1. **Return value** – data type of the result
2. **Name** of function
3. **Parameter list** – inside parentheses; input data

$$\overset{1}{\textbf{return value}}$$

$$\underset{2}{\textbf{name of function}}\ \underset{3}{\textbf{(parameter(s))}}$$

HEADER PART 1 – RETURN VALUE

In order to use a function, you need to know **what kind of results** it may provide.

The return value tells the **data type** of the result

Examples of return values:

- double, float, int, char, etc.

There can only be ONE return value indicated.

DO NOT include the name of a variable, only the data type.

Header Part 1 – Return Value Examples

double functionName1

- This function sends back a result that is the double data type

int functionName2

- This function sends back a result that is the int data type

char functionName3

- This function sends back a result that is the char data type

Header Part 1 – Some Functions Do Not Return a Result

Functions may be used to perform a task, but no result is created

Examples: printing some data, saving data in a file

To show that a function does not return a result, use the **void keyword**:

void functionName4

The code that uses this function then knows that no result will be sent back.

Header Part 2 – Function Name

Each function should be limited to performing a single, well-defined task

The function name should communicate that task using the **verbNoun** format:

- calculateAverage
- printReport

If you can’t choose a short, clear name that describes what the function does, maybe the function is trying to execute too many separate or unrelated tasks

HEADER PART 3 – PARAMETER(S)

If a function needs data to act on, that data is represented by a **parameter**

A parameter has **2 parts**:

1. **Data type** – int, float, double, char, etc.
2. **Variable name** – identifier

Header Part 3 – Parameter List

If a function needs more than one parameter, create a **parameter list** within the parentheses, separated by commas:

```
( int x, char c, float f )
```

Each parameter must have its own data type and name!

Incorrect syntax:

```
( int x, c, float f )
```

The Rest of the Function Definition

Remember a function definition includes four parts:

1. Header (a.k.a. signature) – We just talked about this.
2. Opening curly brace
3. Body of function (C control structures that make up the function’s work)
4. Closing curly brace

Structure of a Function Definition 1

return value

name of function (parameter(s))

{

C control structure(s) that perform the task of this function

}

1. These two lines make up the **header**.

Structure of a Function Definition 2

return value

name of function (parameter(s))

{

    C control structure(s) that perform the task of this function

}

1. These two lines make up the header.
2. **opening curly brace**

Structure of a Function Definition 3

return value

name of function (parameter(s))

{

C control structure(s) that perform the task of this function

}

1. These two lines make up the header.

2. opening curly brace

3. **function body**

Structure of a Function Definition 4

return value

name of function (parameter(s))

{

C control structure(s) that perform the task of this function

}

1. These two lines make up the header.
2. opening curly brace
3. function body
4. **closing curly brace**

Locating Function Definitions

Programmer-defined function definitions go **outside of main**, in any order

Best practice – put them after main

Function Prototypes

STEP 2 – WRITE THE FUNCTION'S PROTOTYPE

The next step in working with functions is to write the function's **prototype**.

Functions must be **declared** before use

- Just like variables must be declared before use

Function's **prototype** tells compiler about the function

- So, the **prototype is the function declaration**

Prototypes **always listed before main**

- Just like preprocessor directives give the compiler info, so do prototypes

PROTOTYPE HAS 4 PARTS

1. **Return value** – data type of the result
2. **Name** of function
3. **Parameter list** – inside parentheses
4. **Semicolon**

return value   name of function   (parameter(s)) ;

1   2   3   4

Prototype and Header are Similar

1.	**Return value**	exactly the same in both
2.	**Name** of function	exactly the same in both
3.	**Parameter List**	exactly the same in both
4.	**Semicolon**	prototype only

Prototype – same as function’s header, plus a semicolon

Header – same as function’s prototype, without a semicolon

Function Calls

Step 3 – Call the Function

To use a function in a program, there must be code that **calls** it

Function call

- A statement that causes a function to begin executing
- Causes the function's instructions and data to be loaded into memory

A function call may:

- Provide input data, if the function requires it
- Be part of an assignment statement to handle a result

Can be located anywhere in your code where you need it

To "**call**" a function = to "**invoke**" a function

Function Call Has Four Parts

1. **Name** of the function
2. **Open parenthesis (**
3. **Variable name(s) or literal value(s)** inside the parentheses, if the function requires it
4. **Closing parenthesis)**

Programmer-Defined Function “square”

fig05_03_simple.c

Computes the square of a number entered by the user

TRANSFER OF CONTROL

When **function call** is encountered, control is transferred to that function

- In 5.3_simple.c, line 15 contains a **call** to the function **square**
- There's also a function call in line 17 to what function?

When the function completes, control is transferred back to the statement that called it

Once this transfer of control is complete, memory associated with the function is released.

SCOPE

Scope of an Identifier

Where an identifier is “visible”

Identifier – variable or function name

The scope of an identifier is the section of code where it can be used

4 Types of Scope

1. Block scope
2. Function scope
3. File scope
4. Function-prototype scope

Scope Type 1: Block Scope

Identifier declared inside any set of curly braces (a block)

We saw an example with the for loop:

for (int x = 0; x < 5; x++) { … }

The variable x is only in scope within this for loop

A block can also be part of selection, or other types of loops, or functions

Scope Type 1: Block Scope – Local Variables

Remember, a function has starting and ending curly braces, so the function body is a block

Identifier declared within the curly braces of a function, usually at the top, is called a **local variable**

In scope from point of declaration to the last curly brace of its block

Function parameters

- Declared in the function header
- Also visible within that function

local.c

Scope Type 1: Block Scope – Nested Blocks

If within a nested block, only in scope within that block

blockExample1.c

If a variable in an outer block has the same name, it is hidden until the inner block terminates

blockExample2.c

Scope Type 2: Function Scope

Only labels (like the case labels in switch statements)

We're not going to see this type.

Scope Type 3: File Scope

Also called **global scope**

Identifier declared outside of any function

Visible to all functions from point of declaration until end of the file

Global variables, function definitions, function prototypes

We will not use global variables in this class…

global.c

Scope Type 4: Function-Prototype Scope

Applies only to the **identifiers in a prototype's parameter list**

Variable names not required here

- Often omitted

Scoping Issues Example

fig05_16.c

COMMUNICATING WITH FUNCTIONS

Functions are "Black Boxes"

In science, computing, and engineering, a ***black box*** *is a device, system or object which can be viewed in terms of its inputs and outputs (or transfer characteristics), without any knowledge of its internal workings.* https://en.wikipedia.org/wiki/Black_box

In other words, functions may need input data and may provide some result, but we **don't need to know how the function performs its tasks** in order to use the function

A FUNCTION IS A “BLACK BOX”

Variables declared **outside** of a function are not in scope **within** the function

And any variables declared **within** a function are not in scope **outside** the function

So How to Communicate with a Function?

How can a function receive values to work on?

How can a function provide results to the code that called it?

Remember the Function Header

1. **Return value** – data type
2. **Name** of function
3. **Parameter list** – inside parentheses

The **return value** and **parameter list** provide ways to communicate with function

FUNCTIONS CAN BE DEFINED:

With a return value

Without a return value

With parameters

Without parameters

And with any combination of return value and parameters

Functions with a Return Value

Functions with a **return value** provide a result to the calling code

Data type of return value indicates data type of this result

If a return value is used, the function MUST include a **return** statement

A call to a function with a return value can be used in an **expression**

Look at **fig05_03_simple.c** again

FUNCTIONS WITHOUT A RETURN VALUE

Must use the keyword **void** in place of the return value

Example: Function "**print_report**" doesn't need to send any info back to the code that called it.

Its prototype: **void print_report (int month_num);**

Its header: **void**
print_report (int month_num)

Void functions do not need a return statement print_report.c

Functions with Parameters

Some functions need information to act on from calling code

Some functions send lots of information back to calling code

- More than one result

This sending of information is done using **parameters** and **arguments**

Parameters allow function to manipulate different data each time it is called

- More versatile than using constant data within a function

Parameters vs. Arguments

Parameter

- Information about the data the function needs in order to perform its work
- Consists of a data type, and a variable name

Argument

- The **A**ctual data provided to the function when the function is called

It's common to just use the term argument for both, but technically they are two different things

Types of Parameters

Two types of parameters to a function:

1. **Input parameters** carry information into the function
2. **Output parameters** return multiple results out of the function (chapter 7)

Example: Function with One Input Parameter

void

print_report (int month)

print_report.c

Functions with Multiple Input Parameters

Arguments passed when function is called must match parameters in the function definition:

1. **N**umber of arguments
2. **O**rder of the arguments
3. **T**ype of the arguments

multiple_parameters.c

Passing Arguments By Value and By Reference

By value

- A **copy of the argument's value** is made and passed to the called function

By reference

- the caller allows the called function to modify the original variable's value

In C, all arguments are passed by value

Functions without Parameters

Some functions don't need information to act on from calling code

Leave the parentheses empty in the prototype and header

Do not send any data in the function call

print_column_headings.c

COMMON ERRORS & OTHER OBSERVATIONS

Common Errors When Using Functions

1. Parameters without a data type
2. Semicolon in the header
3. Redefining a parameter as a local variable
4. Defining a function within a function

Parameter without a Data Type

Every parameter in the prototype and the header must have a data type!

Correct:	(int x, int y)
Incorrect:	**(int x, y)**

This will result in a syntax error.

Semicolon in the Header

There is no semicolon after the right parenthesis in the **header**, which is part of the function definition

Correct: `void theFunction (int x, int y)`

Incorrect: `void theFunction (int x, y);`

This will result in a syntax error.

Redefining a Parameter as a Local Variable

Don’t reuse the name of a parameter within the function

Correct:

```
void theFunction (int x, int y) {
    int a;
    char b;
}
```

Incorrect:

```
void theFunction (int x, int y) {
    int x;
    char y;
}
```

This will result in a syntax error.

DEFINING A FUNCTION INSIDE A FUNCTION

Defining a function inside another function is a syntax error:

```
void theFunction (int x, int y) {
    int someOtherFunction() {


    }
}
```

This will result in a syntax error.

Main's Return Type

main has an int return type

Used to indicate whether the program executed correctly:

```
return (0);
```

Later standards remove need for this statement

ADVANTAGES OF USING FUNCTIONS

Procedural abstraction

- Allows details to be shielded from main
- main can be written as sequence of function calls

Reuse of function’s code

- Functions can be executed more than once in a program without having to rewrite code each time

Multiple programmers can be working on different parts of a system at the same time

Can be easier to debug in sections

Disadvantage of Using Functions

Too many function calls can be slow

In many C applications and systems, such as game programming, real-time systems, operating systems and embedded systems, performance is crucial