# Chapter 5 - Fruitful functions

## Return values

[Return](#return), [statement!return](#statement!return), [function!fruitful](#function!fruitful), [fruitful function](#fruitful_function), [return value](#return_value), [void](#void), [function!void](#function!void)

Some of the built-in functions we have used, like the math functions, have produced results. That is, the effect of calling the function is to generate a new value, which we usually assign to a variable or use as part of an expression. For example:

double e = exp (1.0);

double height = radius \* sin (angle);

But so far all the functions we have written have been **void** functions; that is, functions that return no value. When you call a void function, it is typically on a line by itself, with no assignment -- there is nothing to store as no result was produced:

printLines (3);

countdown (n-1);

In this chapter, we will create functions that produce results, or “fruit,” as opposed to our previous void functions, which produced nothing. I will refer to as **fruitful** functions because they yield results.

The first example is area, which takes a double as a parameter, and returns the area of a circle with the given radius:

double area (double radius)

{

double pi = acos (-1.0);

double area = pi \* radius \* radius;

return area;

}

The first thing you should notice is that the beginning of the function definition is different. Instead of void, which indicates a void function (that will not produce a fruit), we see double, which indicates that the return value (the fruit) from this function will have type double.

Also, notice that the last line is an alternate form of the return statement that includes a return value. This statement means, “return immediately from this function and use the following expression as a return value.” The type of the expression in the return statement must match the return type of the function. In other words, when you declare that the return type is double, you are making a promise that this function will eventually produce a double. If you try to return with no expression, or an expression with the wrong type, the compiler will take you to task. The purpose of our function prototypes is to let the compiler know the type of the parameters and return value of our function.

The prototype for this function would be:

double area (double ) ;

When we define a fruitful function we can only return one value. The return expression you provide can be arbitrarily complicated, but it must yield only one value. We could have written this function more concisely, but ultimately we only return one value:

double area (double radius)

{

return acos(-1.0) \* radius \* radius;

}

[Temporary](#temporary_variable), or [local](#variable!temporary), variables like area and pi often make debugging easier and help to break up concepts into smaller more manageable parts.

There are two main schools of thought when it comes to returning from functions.

One idea is that functions should have only one return statement, a single exit point. Others favor multiple returns. This book takes no stance on this issue -- other than we strive to write readable and maintainable code. Sometimes this means we have single exit point, while other times using multiple return statements to return early from a function. We will note that single exit functions can be easier to debug because they uses local variables to store return results. We use multiple return statements in this absolute value example:

double absoluteValue (double x)

{

if (x < 0)

{

return -x;

}

else

{

return x;

}

}

Since these returns statements are in an alternative conditional, only one will be executed. Having more than one return statement in a function, means as soon as a return is executed, the function terminates without executing any subsequent statements. This can be used to exit a function when we know there is no point in executing the remaining code:

/\*

this function returns 0 if either

x or y are negative otherwise it returns

their product

\*/

double earlyReturnExample (double x)

{

if (x < 0) //gaurd from x being negative

{

return 0;

}

if (y < 0) //gaurd from y being negative

{

return 0;

}

return x \* y;

}

Code that appears after a return statement, or any place else where it can never be executed, is called **dead code**. Some compilers warn you if part of your code is dead.

If you put return statements inside a conditional, then you have to guarantee that every possible path through the program hits a return statement. For example:

double AbsoluteValue (double x)

{

if (x < 0)

{

return -x;

}

else if (x > 0)

{

return x;

} /\* WRONG!! \*/

}

This program is not correct because if x happens to be 0, then neither condition will be true and the function will end without hitting a return statement. Unfortunately, many C compilers do not catch this error. As a result, the program may compile and run, but the return value when x==0 could be anything, and will probably be different in different environments.

By now you are probably sick of seeing compiler errors, but as you gain more experience, you will realize that the only thing worse than getting a compiler error is not getting a compiler error when your program is wrong.

Here's the kind of thing that's likely to happen: you test absoluteValue() with several values of x and it seems to work correctly. Then you give your program to someone else and they run it in another environment. It fails in some mysterious way, and it takes days of debugging to discover that the problem is an incorrect implementation of absoluteValue(). If only the compiler had warned you!

From now on, if the compiler points out an error in your program, you should not blame the compiler. Rather, you should thank the compiler for finding your error and sparing you days of debugging. Some compilers have an option that tells them to be extra strict and report all the errors they can find. You should turn this option on all the time.

As an aside, you should know that there is a function in the math library called fabs() that calculates the absolute value of a double -- correctly.

# Program development

At this point you should be able to look at complete C functions and tell what they do. But it may not be clear yet how to go about writing them. I am going to suggest one technique that I call **incremental development**.

As an example, imagine you want to find the distance between two points, given by the coordinates $(x\_1, y\_1)$ and $(x\_2, y\_2)$. By the usual definition,

distance = sqrt((x2 - x1)^2 + (y2 - y1)^2)

The first step is to consider what a Distance function should look like in C. In other words, what are the inputs (parameters) and what is the output (return value).

In this case, the two points are the parameters, and it is natural to represent them using four doubles. The return value is the distance, which will have type double.

## Program development

Already we can write an outline of the function:

double distance (double x1, double y1, double x2, double y2)

{

return 0.0;

}

The return statement is a placekeeper so that the function will compile and return something, even though it is not the right answer. At this stage the function doesn't do anything useful, but it is worthwhile to try compiling it so we can identify any syntax errors before we make it more complicated.

In order to test the new function, we have to call it with sample values. Somewhere in main() I would add:

double dist = distance (1.0, 2.0, 4.0, 6.0);

printf ("%f\n" dist);

I chose these values so that the horizontal distance is 3 and the vertical distance is 4; that way, the result will be 5 (the hypotenuse of a 3-4-5 triangle). When you are testing a function, it is useful to know the right answer. Before you test any code you should state what you expect the output to be.

Once we have checked the syntax of the function definition, we can start adding lines of code one at a time. After each incremental change, we recompile and run the program. That way, at any point we know exactly where the error must be---in the last line we added.

The next step in the computation is to find the differences x2 - x1 and y2 - y1. I will store those values in temporary variables named dx and dy.

double distance (double x1, double y1, double x2, double y2)

{

double dx = x2 - x1;

double dy = y2 - y1;

printf ("dx is %f\n", dx);

printf ("dy is %f\n", dy;

return 0.0;

}

I added output statements that will let me check the intermediate values before proceeding. As I mentioned, I already know that they should be 3.0 and 4.0.

When the function is finished I will remove the output statements. Code like that is called **scaffolding**, because it is helpful for building the program, but it is not part of the final product. Sometimes it is a good idea to keep the scaffolding around, but comment it out, just in case you need it later.

The next step in the development is to square dx and dy. We could use the pow() function, but it is simpler and faster to just multiply each term by itself.

double Distance (double x1, double y1, double x2, double y2)

{

double dx = x2 - x1;

double dy = y2 - y1;

double dsquared = dx\*dx + dy\*dy;

printf ("d\_squared is %f\n", dsquared);

return 0.0;

}

Again, I would compile and run the program at this stage and check the intermediate value (which should be 25.0).

Finally, we can use the sqrt() function to compute and return the result.

double distance (double x1, double y1, double x2, double y2)

{

double dx = x2 - x1;

double dy = y2 - y1;

double dsquared = dx\*dx + dy\*dy;

double result = sqrt (dsquared);

return result;

}

Then in main(), we should output and check the value of the result.

As you gain more experience programming, you might find yourself writing and debugging more than one line at a time. Nevertheless, this incremental development process can save you a lot of debugging time.

The key aspects of the process are:

* Start with a working program and make small, incremental changes. At any point, if there is an error, you will know exactly where it is.
* Use temporary variables to hold intermediate values so you can output and check them.
* Once the program is working, you might want to remove some of the scaffolding or consolidate multiple statements into compound expressions, but only if it does not make the program difficult to read. We call this refactoring the code.

## Composition

As you should expect by now, once you define a new function, you can use it as part of an expression, and you can build new functions using existing functions. For example, what if someone gave you two points, the center of the circle and a point on the perimeter, and asked for the area of the circle?

Let's say the center point is stored in the variables xc and yc, and the perimeter point is in xp and yp. The first step is to find the radius of the circle, which is the distance between the two points. Fortunately, we have a function, Distance(), that does that.

double radius = distance (xc, yc, xp, yp);

The second step is to find the area of a circle with that radius, and return it.

double result = area (radius);

return result;

Wrapping that all up in a function, we get:

double areaFromPoints (double xc, double yc, double xp, double yp)

{

double radius = distance (xc, yc, xp, yp);

double result = area (radius);

return result;

}

The temporary variables radius and area are useful for development and debugging, but once the program is working we can make it more concise by composing the function calls, but we should always favor readability over conciseness:

double areaFromPoints (double xc, double yc, double xp, double yp)

{

return area (distance (xc, yc, xp, yp));

}

## Boolean values

The types we have seen so far can hold very large values. There are a lot of integers in the world, and even more floating-point numbers. By comparison, the set of characters is pretty small. Well, many computing languages implement an even more fundamental type that is even smaller. It is called **\_Bool**, and the only values in it are true and false.

Unfortunately, earlier versions of the C standard did not implement boolean as a separate type, but instead used the integer values 0 and 1 to represent truth values. By convention 0 represents false and 1 represents true. Strictly speaking C interprets any integer value different from 0 as true. This can be a source of error if you are testing a value to be true by comparing it with 1.

Without thinking about it, we have been using boolean values in the last of chapter. The condition inside an if statement is a boolean expression. Also, the result of a comparison operator is a boolean value. For example:

if (x == 5)

{

/\* do something\*/

}

The operator == compares two integers and produces a boolean value.

Pre C99 has no keywords for the expression of true or false. A lot of programs instead are using C preprocessor definitions anywhere a boolean expression is called for. For example,

#define FALSE 0

#define TRUE 1

...

if (TRUE)

{

/\* will be always executed \*/

}

is a standard idiom for a loop that should run forever (or until it reaches a return or break statement).

## Boolean variables

Since boolean values are not supported directly in C, we can not declare variables of the type boolean. Instead, programmers typically use the short datatype in combination with preprocessor definitions to store truth values.

#define FALSE 0

#define TRUE 1

...

short fred;

fred = TRUE;

short testResult = FALSE;

The first line is a simple variable declaration; the second line is an assignment, and the third line is a combination of a declaration and as assignment, called an initialization.

* **initialization**
* **statement!initialization**

As I mentioned, the result of a comparison operator is a boolean, so you can store it in a variable

short evenFlag = (n%2 == 0); /\* true if n is even \*/

short positiveFlag = (x > 0); /\* true if x is positive \*/

and then use it as part of a conditional statement later

if (evenFlag)

{

printf("n was even when I checked it");

}

A variable used in this way is called a **flag**, since it flags the presence or absence of some condition.

## Logical operators

There are three **logical operators** in C: AND, OR and NOT, which are denoted by the symbols &&, || and !. The semantics (meaning) of these operators is similar to their meaning in English. For example x > 0 && x < 10 is true only if x is greater than zero AND less than 10.

* **semantics**

evenFlag || n\%3 == 0 is true if either of the conditions is true, that is, if evenFlag is true OR the number is divisible by 3.

Finally, the NOT operator has the effect of negating or inverting a bool expression, so !evenFlag is true if evenFlag is false; that is, if the number is odd.

Logical operators often provide a way to simplify nested conditional statements. For example, how would you write the following code using a single conditional?

if (x > 0)

{

if (x < 10)

{

printf ("x is a positive single digit.\n");

}

}

## Bool functions

It is sometimes appropriate for functions to return boolean values just like any other return type. This is is especially convenient for hiding complicated tests inside functions. For example:

int IsSingleDigit (int x)

{

if (x >= 0 && x < 10)

{

return TRUE;

}

else

{

return FALSE;

}

}

The name of this function is isSingleDigit(). It is common to give such test functions names that sound like yes/no questions. The return type is int, which means that again we need to follow the agreement that 0 represents false and 1 represents true. Every return statement has to follow this convention, again, we are using preprocessor definitions.

The code itself is straightforward, although it is a bit longer than it needs to be. Remember that the expression x >= 0 && x < 10 is evaluated to a boolean value, so there is nothing wrong with returning it directly, and avoiding the if statement altogether:

int iSingleDigit (int x)

{

return (x >= 0 && x < 10);

}

In main() you can call this function in the usual ways:

printf("%i\n", isSingleDigit (2));

short bigFlag = !isSingleDigit (17);

The first line outputs the value true because 2 is a single-digit number. Unfortunately, when C outputs boolean values, it does not display the words TRUE and FALSE, but rather the integers 1 and 0.

The second line assigns the value true to bigFlag only if 17 is not a positive single-digit number.

The most common use of boolean functions is inside conditional statements:

//very readable with the bool funciton - if is single digit

if (isSingleDigit (x))

{

printf("x is little\n");

}

else

{

printf("x is big\n");

}

## Returning from main()

Now that we know functions that return values, we can look more closely at the return value of the main() function. It's supposed to return an integer:

int main (void)

The usual return value from main() is 0, which indicates that the program succeeded at whatever it was supposed to to. If something goes wrong, it is common to return -1, or some other value that indicates what kind of error occurred.

The C standard library <stdlib.h> provides two predefined constants EXIT\_SUCCESS and EXIT\_FAILURE. We can use these to return a descriptive result from our return statement.

#include <stdlib.h>

int main (void)

{

return EXIT\_SUCCESS; /\*program terminated successfully\*/

}

Of course, you might wonder who this value gets returned to, since we never call main() ourselves. It turns out that when the operating system executes a program, it starts by calling main() in pretty much the same way it calls all the other functions. When the program terminates it passes a value back that tells if the execution was successful or not. The operating system can use this value to create error reports or even pass this value on to other programs.

There are even some parameters that can be passed to main() by the system, but we are not going to deal with them for a little while, so we define main() as having no parameters: int main (void).

## Glossary

**return type:**

The type of value a function returns.

**return value:**

The value provided as the result of a function call.

**local variable:**

Also called a temporary variable, is a variable declared in a function and is only accessible from within the function in which it is declared

**dead code:**

Part of a program that can never be executed, often because it appears after a return statement.

**scaffolding:**

Code that is used during program development but is not part of the final version.

**void:**

A special return type that indicates a void function; that is, one that does not return a value.

**boolean:**

A value or variable that can take on one of two states, often called true and false. In C, boolean values are mainly stored in variables of type short and preprocessor statements are used to define the states.

**flag:**

A variable that records a condition or status information.

**comparison operator:**

An operator that compares two values and produces a boolean that indicates the relationship between the operands.

**logical operator:**

An operator that combines boolean values in order to test compound conditions.

**preprocessor statement:**

A statement that is executed by the preprocessor, which runs before the compiler. Preprocessor statements usually begin with a # symbol, and can be used to include libraries, define constants, and more.

Permission is granted to copy, distribute, transmit and adapt this work un- der the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 Inter- national License: https://creativecommons.org/licenses/by-nc/4.0/. If you are interested in distributing a commercial version of this work, please contact the author(s). The LATEX source and code for this book is available from: https://github.com/mkpenta/ThinkC