FCPA 2022  
  
Writing Functions

Student Workbook 05

January 13, 2022

Author

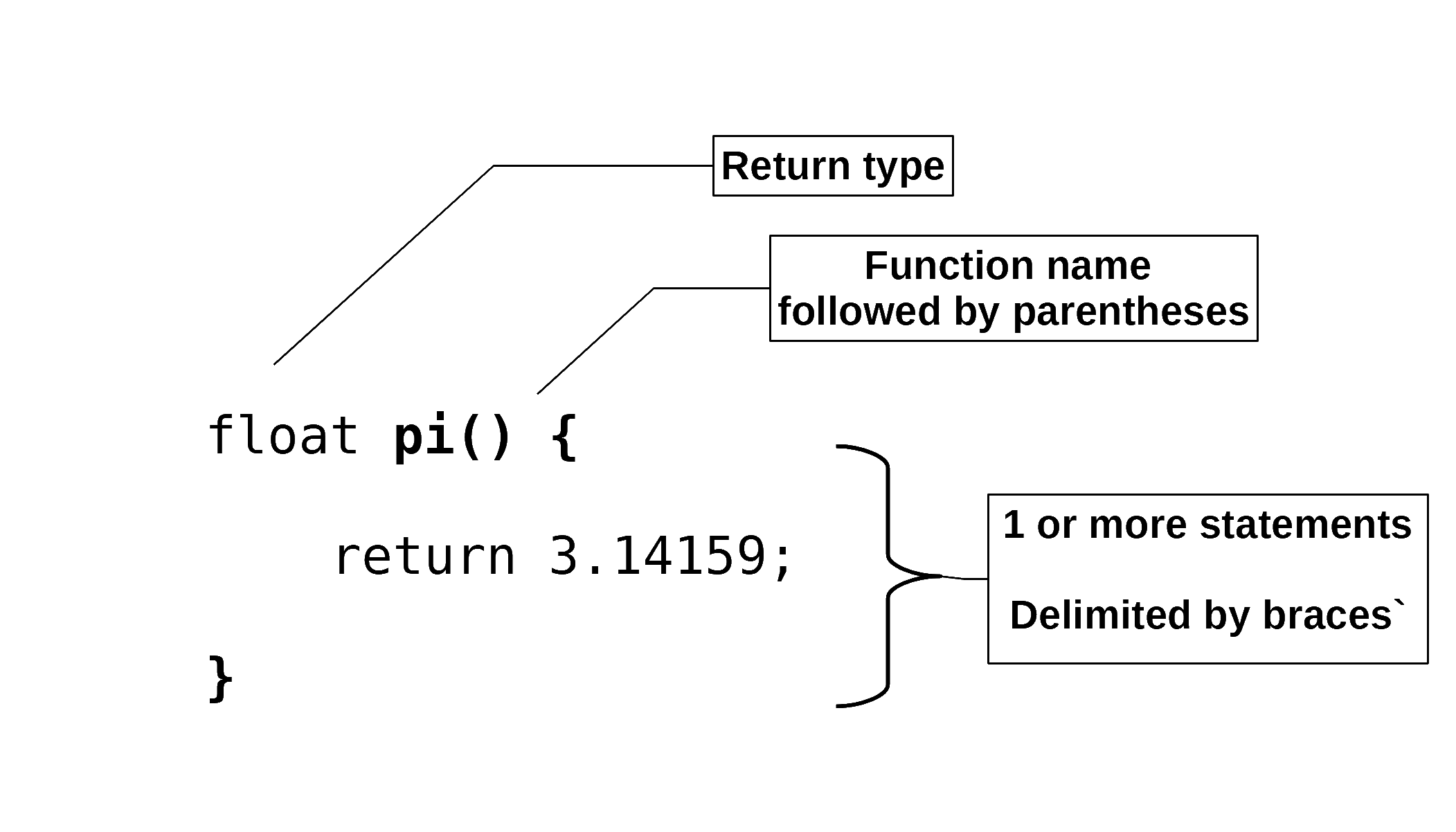
Paul Kimball  
Interface Associates

1. Recap
2. Writing Functions

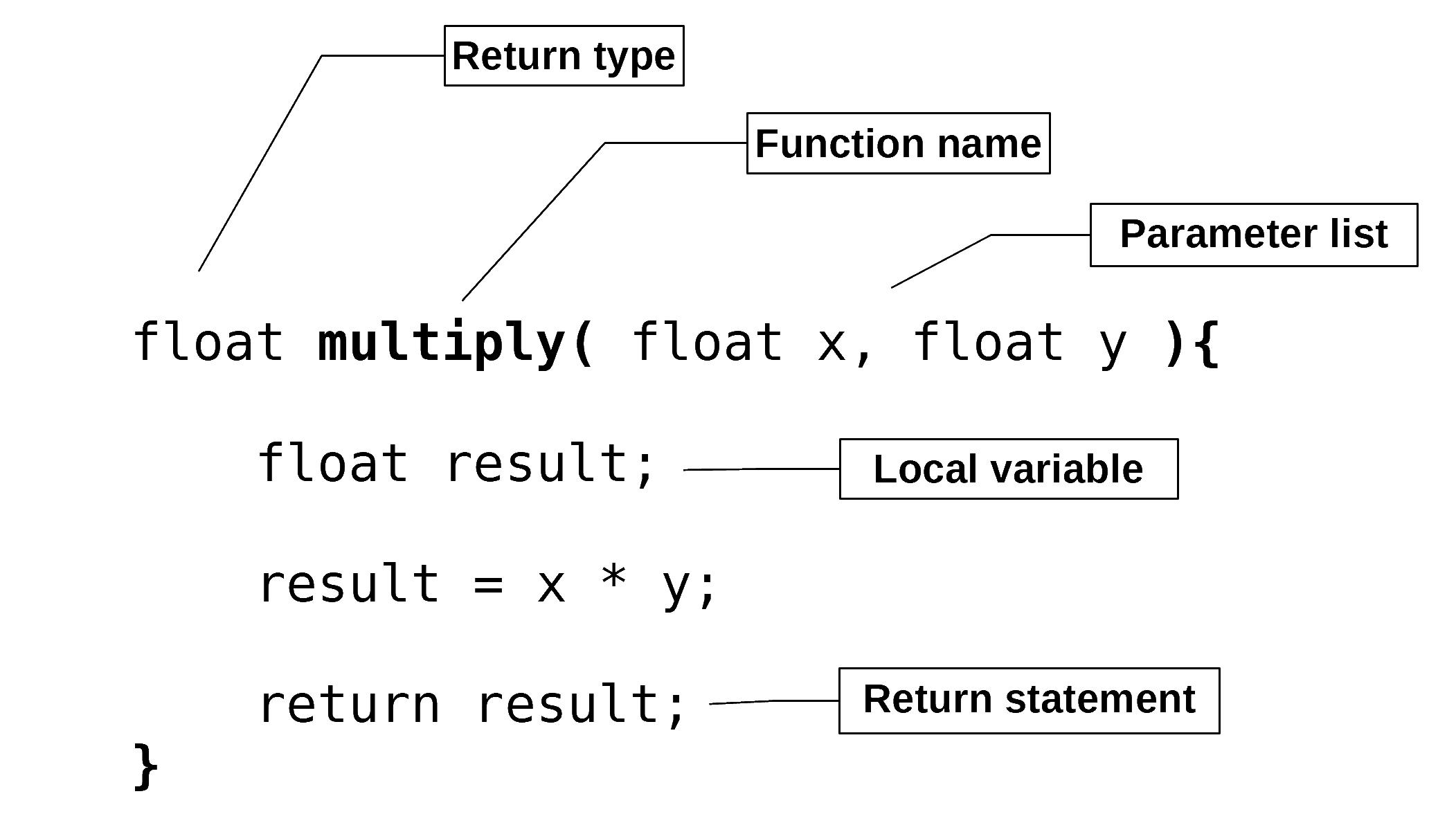
Why to Write a Function

* There's no existing library function that does the same thing
  + Always check first!
* Readability
  + Makes code clearer in intent by giving a name to a workflow
  + Makes your functions smaller
* Reusability
  + You see an opportunity to reuse an algorithm
* Modularity
  + Divide logic into subsystems that have distinct purposes
  + Break out complicated conditional logic that would otherwise involve a lot of nested selection statements

Simple Function Definition



Parameters and Return values



* Parameters represent *values* supplied by the caller of a function
  + Declared in the parameter list with required data type and name
  + Parameters are treated as local variables with block scope
* A single *value* may be returned from a function, and is accessible to the caller immediately after the function returns
  + The caller may use it or ignore it
* Local variables go out of scope and are not accessible after a function returns
  + They are not shared with or seen by the caller

Function call

* The caller of a a function is always another function
  + main, for example
* The caller passes values called *arguments*
  + Can be variables, constants, or results of expressions
  + Data types and order must match the function parameter list
* The caller receives the return value after the function returns
  + The caller may assign this to its own variable or use it in a surrounding expression

#include <stdio.h>

int main(void){

float how\_many = 42.F;

float millions = 1000000.F;

float price;

**price = multiply(how\_many, millions);**

printf("That will be %f dollars, please\n",

price);

**price = multiply(1., millions);**

printf("Fine, %f dollars, then\n",

price);

}

Pass by Value

* By default, objects are passed between functions by *value*
  + Caller's values are *copied* into parameters when function starts
  + Return value is *copied* back to the caller when function returns

#include <stdio.h>

**double** pi() {

**return 3.141592653**;

}

**double** multiply(double x, double y) {

**return x \* y**;

}

int main(void) {

double circle\_area = 0.;

double r = 5.;

circle\_area = pi() \* multiply(r, r);

printf("If radius is %lf, area is %lf",

r, circle\_area);

}

* The *values* of the caller's arguments are *copied* into the function's parameters
  + The caller's variables and the function's variables are *separate* objects
* The return *value* is *copied* back to the caller
  + Returned value replaces the function call in any surrounding expression
  + The caller may save the value by assigning it to a variable
  + The caller may ignore the returned value
* By default, a function cannot directly access its caller's variables

Pass by Reference

* A *reference* is the *address* of an object

#include <stdio.h>

**void swap(int \*left, int \*right) {**

int temporary;

**temporary = \*left;**

**\*left = \*right;**

**\*right = temporary;**

}

void print\_values(x1, x2) {

printf("Values are %d and %d\n", x1, x2);

}

int main(void) {

int value\_01 = 5;

int value\_02 = 99;

print\_values(value\_01, value\_02);

**swap(&value\_01, &value\_02);**

print\_values(value\_01, value\_02);

}

* Requires a special agreement between the function and its caller
  + Takes a little bit of work on both sides
  + The caller uses the ampersand to take the address of an object
  + The function declares its parameters as pointer types
  + The function uses the *dereference* operator (asterisk) to access the object
* Now, a function *can* change the values of its callers' variables !!
  + This is sometimes called changing "by side-effect"
  + You were warned - The asterisk and ampersand tell you this can happen

Function Declaration

* You can *declare* a function without *defining* it
  + Leave off the function body and add a semicolon

/\* Declare function \*/

**int multiply(int x, int y);**

/\* Use the identifier \*/

int main(void){

**int answer = multiply(7, 6);**

}

/\* Later, define function \*/

int multiply(int x, int y){

return x \* y;

}

* Lets you reorder your function definitions without harm
  + main definition can go before function definitions if you want
  + Called a "forward declaration" or "function prototype"
* Handy if you want to describe your function to a caller, without giving them the source code
  + A header file can declare the functions in the library
  + The definitions can be compiled ahead of time and saved in an object library
* The "Linker" resolves the linkage of identifiers *after* the compiler is done translating your source code
  + The compiler only cares that the source code is syntactically correct in both the caller and the function