

Lecture 8: Functions

Sarah Nadi

nadi@ualberta.ca

Department of Computing Science
University of Alberta

CMPUT 201 - Practical Programming Methodology
Winter 2018

[With material/slides from Guohui Lin, Davood Rafei, and Michael Buro. Most examples taken from K.N. King's book]



Agenda

- Function declarations
- Defining and calling functions
- Arguments
- return statement
- Program termination

Readings

- Textbook Chapter 9

What is a Function?

- Essentially a series of statements, grouped together and given a name
- Functions are building blocks, each is like a mini program with its own declarations and statements
- Functions do not necessarily have parameters, nor necessarily compute something
- Functions are used to (1) divide your program such that it is easier to understand and modify, and (2) to avoid duplication and enable code re-use

Functions we Already Saw

- We have already seen the `main` function before
- The `main` function is special because it is required in any C program since it is the starting point of the program
- We already saw two variations of `main`:

```
int main (void) {  
    ...  
}
```

```
int main (int argc, char* argv[ ]){  
    ...  
}
```

Functions we Already Saw

- We have already seen the `main` function before
- The `main` function is special because it is required in any C program since it is the starting point of the program
- We already saw two variations of `main`:

```
int main (void) {  
    ...  
}
```

**example of a function that
has no parameters and
returns an integer**

```
int main (int argc, char* argv[ ]) {  
    ...  
}
```

Functions we Already Saw

- We have already seen the `main` function before
- The `main` function is special because it is required in any C program since it is the starting point of the program
- We already saw two variations of `main`:

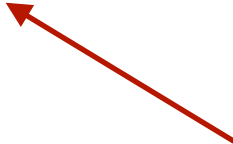
```
int main (void) {  
    ...  
}
```

**example of a function that
has no parameters and
returns an integer**



```
int main (int argc, char* argv[ ]) {  
    ...  
}
```

**example of a function that
has two parameters
and returns an integer**



Calculating Average

- Assume we want to calculate the average of two numbers `a` and `b` in various places in our program
- Instead of writing the same code multiple times in these places, we can simply create a single function called `average` that does that computation for us:

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

- ▶ Each time `average` is called, it returns a value of type `double`
- ▶ Values for parameters `a` and `b` need to be supplied when `average` is called, and they must have the type `double` (or a type that can be implicitly converted into a `double`)

Example of Using average From Previous Slide

```
int main( ){  
  
    double x, y, avg;  
  
    printf("Enter two numbers:");  
    scanf("%lf%lf", &x, &y);  
  
    avg = average(x,y);  
  
    printf("The average of %lf and %lf is %lf\n", x, y, avg);  
  
    printf("The average of 4.57 and 8.29 is %lf\n", x, y, average(4.57,8.29));  
  
    return 0;  
}
```


Example of Using average From Previous Slide

```
int main( ){
```

```
    double x, y, avg;
```

```
    printf("Enter two numbers:");  
    scanf("%lf%lf", &x, &y);
```

```
    avg = average(x,y);
```

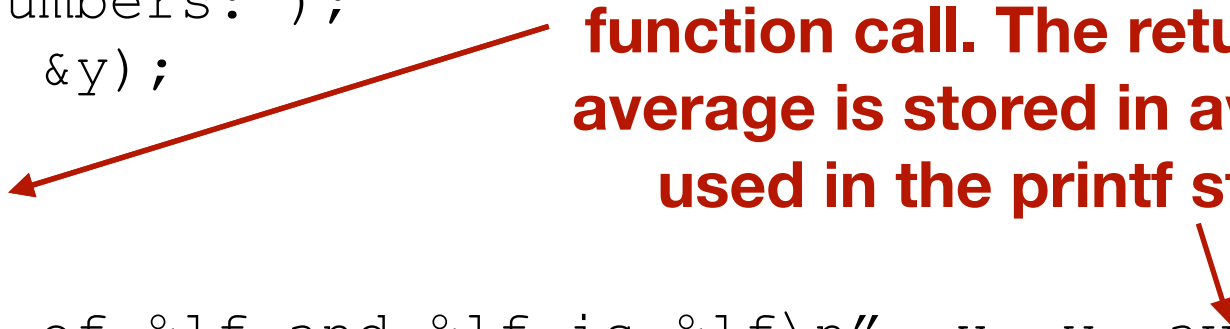
```
    printf("The average of %lf and %lf is %lf\n", x, y, avg);
```

```
    printf("The average of 4.57 and 8.29 is %lf\n", x, y, average(4.57,8.29));
```

```
    return 0;
```

```
}
```

This is a function call to average. *x* and *y* are called the *arguments* of the function call. The return value of average is stored in *avg* and then used in the printf statement



Example of Using average From Previous Slide

```
int main( ){
```

```
    double x, y, avg;
```

```
    printf("Enter two numbers:");  
    scanf("%lf%lf", &x, &y);
```

```
    avg = average(x,y);
```

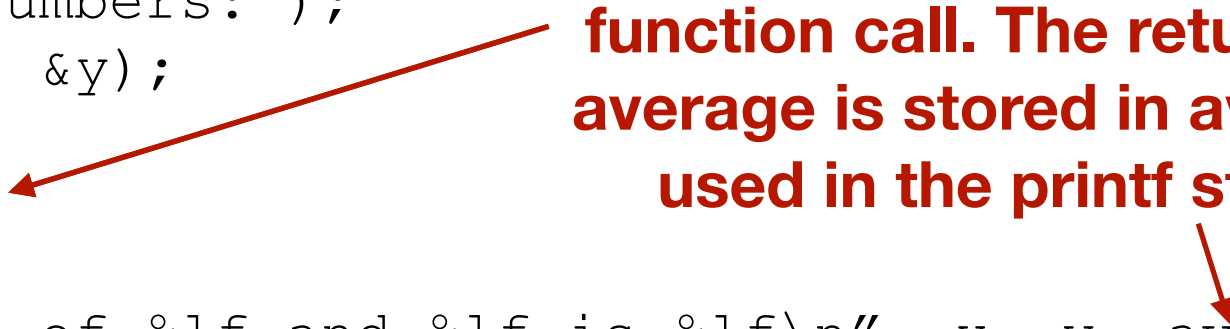
```
    printf("The average of %lf and %lf is %lf\n", x, y, avg);
```

```
    printf("The average of 4.57 and 8.29 is %lf\n", x, y, average(4.57,8.29));
```

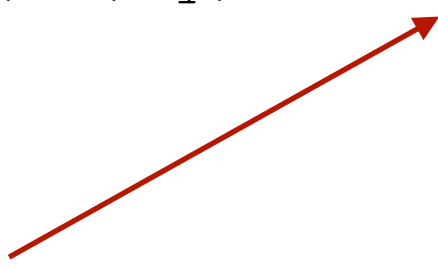
```
    return 0;
```

```
}
```

This is a function call to average. *x* and *y* are called the *arguments* of the function call. The return value of average is stored in *avg* and then used in the printf statement



This is another function call to average. Note how the return value of average is directly passed as an argument to printf without “saving” it in another variable first. Here, 4.57 and 8.29 are the arguments passed to average.



Function Definitions

```
return-type function-name ( parameters ) {  
    declarations  
    statements  
}
```

- A function returns a value of type `return-type`. **This return-type cannot be an array**
- A function does not need to return a value. In this case, its return-type is `void`
- Parameters are separated by commas and must have a type. The parameter list needs to be enclosed by `(...)`. If no parameters, use empty list `()` or `(void)`
- Variables declared inside a function belong exclusively to the function
- The function body must be enclosed by `{ ... }` even if it is empty

How Does the Compiler Know that a Function Exists?

- To be able to call a function, the compiler needs to know that it exists.
- This means that it should have seen the function's definition before

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

```
int main() {  
    ...  
    average(3, 9);  
}
```

Function Declarations

- Too many function definitions might make main difficult to find.
- In practice, the compiler does not need to see the full function definition to accept a call to the function. It only needs to have the necessary information about the function.
- Such information can be provided through *function declarations* (a.k.a *function prototypes*):

```
return-type function-name (parameters);
```

- ▶ Note the use of ; that indicates this is a function declaration.
- ▶ The detailed definition of the function can then be in another place

Function Declaration

Example

```
double average(double u, double v);
```

```
int main() {
```

```
    ...
```

```
    average(3, 9);
```

```
}
```

```
double average(double a, double b) {
```

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

```
}
```

Function Declaration

Example

```
double average(double u, double v);
```

```
int main() {
```

```
    ...
```

```
    average(3, 9);
```


```
}
```

```
double average(double a, double b) {
```

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

```
}
```

Note how the parameter names are different between the declaration and definition? This is because the declaration mainly provides the necessary type information about the function, while the parameter names are not important for the declaration.



What Happens When a Function is Called?

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

```
avg = average(x, y);
```


What Happens When a Function is Called?

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

```
avg = average(x, y);
```

In this call, the value of `x` is **copied**** into `a` and the value of `y` is ****copied**** into `b`.**

The function body is then executed and the return value is **copied**** into `avg`.**

Arguments vs. Parameters

- *Parameters* appear in function definitions
- *Arguments* are expressions that appear in the function calls
 - ▶ Arguments are passed *by value*. Each argument is evaluated and assigned to the corresponding parameter.
 - ▶ Example:

```
int addOne (int a) {  
    return a + 1;  
}
```

```
int main() {  
    int x = 4;  
    addOne(x);  
    x = addOne(x);  
}
```

Arguments vs. Parameters

- *Parameters* appear in function definitions
- *Arguments* are expressions that appear in the function calls
 - ▶ Arguments are passed *by value*. Each argument is evaluated and assigned to the corresponding parameter.
 - ▶ Example:



```
int addOne (int a) {  
    return a + 1;  
}
```

```
int main() {  
    int x = 4;  
    addOne(x) ;  after this statement, x is still 4  
    x = addOne(x) ;  
}
```

Arguments vs. Parameters

- *Parameters* appear in function definitions
- *Arguments* are expressions that appear in the function calls
 - ▶ Arguments are passed *by value*. Each argument is evaluated and assigned to the corresponding parameter.
 - ▶ Example:

```
int addOne (int a) {  
    return a + 1;  
}
```

```
int main() {  
    int x = 4;  
    addOne(x);  after this statement, x is still 4  
    x = addOne(x);  after this statement, x will be 5  
because we have assigned the return  
value of addOne to it  
}
```

Type Conversion in Function Calls

- If the type of an argument does not match the type of the corresponding parameter, the compiler performs an implicit type conversion
- Obviously, this only works if a type conversion can happen
- Note, however, that the type conversion might lead to unexpected behavior

1-d Array Arguments

```
int sum_array1(int a[10]);  
int sum_array2(int a[], int n);
```

- `sum_array1` has a single parameter, which is a 1-dimensional array of length 10. Note that:
 - ▶ `a` is of fixed length, and cannot be changed during calls to the function
 - ▶ if the length of the passed array is ≥ 10 , only the first 10 elements are used
- `sum_array2`'s first parameter is a 1-dimensional array of unknown length. The length is normally specified as a second parameter, `n` in this case.
 - ▶ without knowing `n`, it is difficult to determine the length
 - ▶ when called, the length argument must be \leq the actual array length (otherwise we will go over the bounds of the array)

`sizeof` does NOT Work with Array Parameters

```
int f(int a[]) {  
    int len = sizeof(a) / sizeof(a[0]);  
}
```

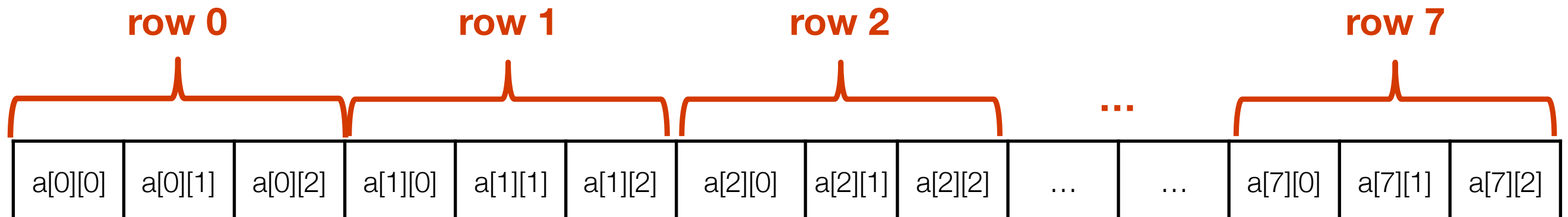
- The above code will not correctly give the number of elements in `a`
- We will understand this more when we cover pointers and their relationship to arrays

Multi-dimensional Array Arguments

```
int sum_array3(int a[][100], int n);
```

- only the length of the first dimension of the array can be unknown. This because how multi-dimensional arrays are laid out in memory (see below)
- We will learn later how to overcome this limitation

```
int a[8][3];
```



More on Array Arguments

```
int sum_array2(int a[], int n);  
int sum_array2(int n, int a[n]);  
int sum_array2(int n, int a[]);  
int sum_array2(int n, int a[*]);
```

```
int sum_twod(int a[][50], int n);  
int sum_twod(int n, int m, int a[n][m]);  
int sum_twod(int n, int m, int a[*][*]);  
int sum_twod(int n, int m, int a[][m]);  
int sum_twod(int n, int m, int a[][*]);
```

- those in blue can only be used in C99 (variable-length arrays)
- the `[*]` provides a clue that the length of the array is related to parameters that come earlier in the list
- In variable-length arrays, order is important. Any parameters used in the array dimensions must appear before the array parameter.

IMPORTANT: Functions can change the elements of array parameters

```
void multiplyByIndex (int a[], int n) {  
    for (int i = 0; i < n; i++) {  
        a[i] *= i;  
    }  
}
```

```
int main () {  
    int myArray[10] = {1,1,1,1,1,1,1,1,1,1};  
    multiplyByIndex(myArray, 10);  
    for (int i = 0; i < 10; i++)  
        printf("%d ", myArray[i]);  
  
    printf("\n");  
    return 0;  
}
```

What is the output of this program?

Dividing up your program into functions

Demo: `find_repeated.c`

The return statement

- `return expression;`
- e.g.,
 - ▶ `return;`
 - ▶ `return 0;`
 - ▶ `return false;`
 - ▶ `return -1;`
 - ▶ `return n >= 0 ? n : 0;`
- type of the expression must match the return type of the function (otherwise, implicit conversions may occur)
- `return;` is used for a void function
- Recall that for the main function, the return value is a status code, with 0 indicating normal termination

The `exit` Function

- We have already seen it before. It is a function from `<stdlib.h>`
- Remember that `exit` terminates the whole program, regardless of which function it is called in