Functions

CSC 100 / Introduction to programming in C/C++ - Spring 2025

Marcus Birkenkrahe

April 25, 2025

README

- This script introduces C functions as a way to write reusable code.
- This section, including some sample code, is based on chapter 9 in King (2008).

Overview

- C functions do not always resemble math functions `f(x)`
- C functions don't need to have arguments, e.g. `main(void)` or `main()`
- C functions need not compute a value, e.g. `void hello(void)` or `void hello()`
- Each function is a small program with its own declarations and statements. The `main` function is the only mandatory function.
- Functions allow us to
  1. **reuse** functions in other programs
  2. **recall** functions instead of duplicating code
  3. **modularize**, and easier understand and modify programs

Example: `hello_world` function: `void` arguments

- Can you guess what the output of this code block will be?

```
// function definition
void hi(void) // return type + name + parameters
{ // function body begins
  printf("Hello, world!\n"); // what the function does
} // function body ends
// function call
hi();
hi();
hi();
```

```
Hello, world!
Hello, world!
Hello, world!
```

- This function is doubly `void`: no `return` value, no argument parameters = `()`. Compare with `int main(void)`.
- The function can be reused elsewhere: I can call `hi()` from anywhere without having to write the whole `printf` statement.
- Remember that the C compiler really sees this source file:
  1. `#include` header file for Input/output
  2. `main` function definition {...}
  3. `hello_world` function definition {...}
  4. three `hello_world` function calls

hello_world.c

```c
#include <stdio.h>

int main() {
// reusable function definition
void hello_world(void)
{
  printf("Hello world\n"); // what the function does
}
// reusable function call
hello_world();
hello_world();
hello_world();
return 0;
}
```

1 -\--- hello_world.c All (3,0) (C/*l ivy Abbrev

- It's even better for the compiler if you announce the function, and separate it from the `main` program like this:

```c
#include <stdio.h>

// function declaration (before main)
void hi(void);

/* main program */
int main(void) // BEGIN PGM
{
  // function call (inside main)
  hi();
  hi();
  hi();
  return 0;
```

```
} // END PGM

// function definition (after main)
void hi(void)
{
  printf("Hello, world!\n"); // what the function does
}
```

```
Hello, world!
Hello, world!
Hello, world!
```

Practice: `hello_world` function: `string` argument

- What if we want to get the name of the user for a "Hello, [name]!" greeting? That seems just polite!
- Let's define a function `hi2` that does this in C:
  1. The function should accept a name as an argument when called: `hi("Marcus")`.
  2. The function should print the greeting with the name when run: `Hello, Marcus`.
- Pseudo code: see onecompiler.com/c/43enmg7a5 to code along.

```
// declare function
// void hi2 (string)

/* main program */
// BEGIN main
// call function hi2 on "Marcus"
// END main

// define function
// void hi2 (string)
// BEGIN hi2
   // PRINT 'Hello, " + name
// END hi2
```

- Source code:

```
#include <stdio.h>

// declare function
// void hi2 (string)
void hi2(char name[]); // a string is an array

/* main program */
int main (void)
{ // BEGIN main
  // call function hi2 on "Marcus"
  hi2("Marcus");
  return 0;
} // END main

// define function
// void hi2 (string)
void hi2(char name[])
{ // BEGIN hi2
  // PRINT 'Hello, " + name
  printf("Hello, %s\n", name);
} // END hi2
```

```
Hello, Marcus
```

- How about getting the name from the keyboard?

```
#include <stdio.h>

// declare function
// void hi2 (string)
void hi3(char name[]); // a string is an array

/* main program */
int main (void)
{ // BEGIN main
  // declare string array
  char name[50];
```

```
  // GET string from user
  printf("Enter your name: ");
  scanf("%s", name); // address-of name[0] = 'M'
  printf("%s\n", name);
  // call function hi2 on "Marcus"
  hi3(name);
  return 0;
} // END main

// define function
// void hi2 (string)
void hi3(char name[])
{ // BEGIN hi2
  // PRINT 'Hello, " + name
  printf("Hello, %s\n", name);
} // END hi2
```

```
Enter your name: Marcus
Hello, Marcus
```

- Input file:

```
echo 'Marcus' > input
cat input
```

```
Marcus
```

Functions are everywhere in C!

- How many functions do you see in the following code block and what do you think will be its output (guess before running it)?

```
#include <stdio.h> // file for input / output
#include <math.h> // file math constants and functions

int main(void)  // function 1 - defined
{ // syntax to mark the beginning of the function body
  const double E = 2.7182818; // ‘const‘, ‘double‘ are keywords
```

```
  printf("%g\n", log(E));  // function 2 + 3 - called
  return 0; // ‘return‘ is not a function but a keyword
} // syntax to mark the end of the function body
```

```
1
```

Answer:

FUNCTION	DEFINITION	INPUT	OUTPUT
`main()`	main function	None (`void`)	`return 0`
`printf()`	printing function	Arithmetic	Formatted
`log()`	logarithmic function	Constant	Log of e

Practice: Function count

- How many functions do you see? How many function uses?

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

double square_root(double x);

int main(void)
{
  double number = 16.0;
  double result = square_root(number);

  printf("The square root of %g is %g\n", number, result);

  return 0;
}

double square_root(double x)
{
  return sqrt(x);
}
```

- Solution:

  There are **four** functions here!

  1. `square_root` - defined by you
  2. `main` - defined by you
  3. `sqrt` - defined in `<math.h>`
  4. `printf` - defined in `<stdio.h>`

```
#include <stdio.h> // function definition of ‘printf‘
#include <math.h>  // function definition of ‘sqrt‘

double square_root(double x);  // function prototype for ‘square_root‘

int main(void) // function definition for ‘main‘
{
  double number = 16.0;
  double result = square_root(number); // function call

  printf("The square root of %g is %g\n", number, result); // function
                                                           // call
  return 0;
}

double square_root(double x)  // function definition for ‘square_root‘
{
  return sqrt(x); // function call
}
```

Example: computing averages

- We want to compute the average of two `double` values `a` and `b`, we can define a function to do it.
- The `average` of `a` and `b` is defined as `(a + b) / 2`.
- This time we're computing the value inside the function and `return` it to the `main` program for printing.

Function definition

- Pseudocode for the function definition

```
// return the average of two ‘double‘ values
double average(double a, double b)
{ // BEGIN average
  // return (a + b)/2
  return (a + b) / 2;
} // END average
```

Putting the function into a program

- Let's put it into a program & code along: onecompiler.com/c/43ennnadq

```
#include <stdio.h>
/* function prototype */
// return the average of two ‘double‘ values
double average(double, double);

/* main program */
int main(void)
{ // BEGIN PGM
  // SET double values x and y
  double x = 5.1, y = 8.9;
  // SET result
  double result;
  // CALL average on x and y
  result = average(x,y);
  // PRINT "The average of x and y is" + result
  printf("The average of %g and %g is %g\n", x, y, result);

  return 0;
} // END PGM

/* function definition */
double average ( double a, double b)
{ // BEGIN average
  // declare result
  double result;
```

```
  // compute result
  result = (a + b) / 2.;
  // return result
  return result;
} // END average
```

```
The average of 5.1 and 8.9 is 7
```

- Here, `double` is *return type* and *argument data type*.
- `a` and `b` are *function parameters* or *arguments* - their values are supplied when the function is called.
- The *function body* is the executable part, enclosed in `{...}`
- What's being executed by the body of the function `average`?

    1. computing the average of two `double` numbers
    2. returning the result as a `double` number

Function calls

- To call a function, write the *function name* followed by a list of *function arguments*, e.g.

  ```
  average(x,y)
  ```

- All parts of this statement must be defined: the function and both arguments; and the data types need to be as defined:

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

- The arguments are assigned to the function parameters.

  ```
  x -> a
  y -> b
  ```

- The function is executed.
- The argument and the `return` value can be any *expression*: We could also have written

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

- Let's look at a few ways of calling this function.

Simple call with numbers

- You can call the function with numbers.

```
// function definition (one line version)
double average(double a,double b){return (a + b) / 2.;}

// function call - result assigned to variable avg
double avg = average(5.1, 8.9); // compute average of two numbers

// function call inside function
printf("Average of %g and %g: %g\n", 5.1, 8.9, avg);
```

```
Average of 5.1 and 8.9: 7
```

Call with expressions

- Functions can have expressions as arguments.

```
// function definition (one line version)
double average(double a,double b){return (a + b) / 2.;}

// declarations
double x=5.1, y=8.9, avg2;

// function call with expression
avg2 = average(x/2., y/2.);

// function call inside function
printf("Average of %g/2 and %g/2: %g\n", x, y, avg2);
```

```
Average of 5.1/2 and 8.9/2: 3.5
```

Call by other functions

- Functions can be called by other functions.

```
// function definition (one line version)
double average(double a,double b){return (a + b) / 2.;}

// declarations
double x=5.1, y=8.9;

// function call inside function
printf("Average of %g and %g: %g\n", x, y, average(x,y));
```

```
Average of 5.1 and 8.9: 7
```

- What's happening in the last line exactly? Describe it!

  1. The `average` function is called with `x` and `y` as arguments.
  2. `average` executes its `return` statement, returning `(a+b)/2`.
  3. `printf` prints the value that `average` returns.
  4. The `return` value of `average` becomes an argument of `printf`.
  5. The value of `average(x,y)` is lost once it's printed.

Practice: Multiple function calls in a program

- The program below reads three numbers and computes their averages, one pair at a time.

  Sample input: `3.5`, `9.6`, `10.2`

  Sample output:

```
: Average of 3.5 and 9.6: 6.55
: Average of 9.6 and 10.2: 9.9
: Average of 3.5 and 10.2: 6.85
```

- Pseudocode: onecompiler.com/c/43enqjaku - Complete the code!

```
// INCLUDE input/output functions

// function declaration
// return the average of two double values


/* main program */

// BEGIN main
  // SET three float values x,y,z

  // print average of x and y

  // print average of y and z

  // print average of x and z

// END main

// function definition
// return the average of two double values
```

- Solution:

```
#include <stdio.h>
// function declaration
// return the average of two double values
double average(double ,double );

/* main program */
int main (void)
{ // BEGIN main
  // SET three float values x,y,z
  float x=3.5, y=9.6, z=10.2;

  // print average of x and y
  printf("Average of %g and %g: %g\n", x, y, average(x,y));
  // print average of y and z
  printf("Average of %g and %g: %g\n", y, z, average(y,z));
  // print average of x and z
```

```
  printf("Average of %g and %g: %g\n", x, z, average(x,z));

  return 0;
} // END main

// function definition
// return the average of two double values
double average(double a,double b) {return (a+b)/2.;}
```

```
Average of 3.5 and 9.6: 6.55
Average of 9.6 and 10.2: 9.9
Average of 3.5 and 10.2: 6.85
```

- What happens if the function is not declared or defined before `main`?

  > Compilation aborts with an error: When the function is called inside `main`, it cannot be found because the compiler has not seen it yet.

Exercise 1: Add three integers

- Write a function `add` that takes three integer arguments, adds them and returns the result to the `main` function where it was called.
- Sample output for input values 1,2,3:

  ```
  Result = 6
  ```

- Write the function prototype for `add` first.
- Then write the `main` function with the call to `add`. Call `add` simply with three numbers, and print the result in `main`.
- Lastly, write the `add` function definition. It must `return` the sum of the three values passed to the function.
- Here is some starter code (onecompiler.com/c/43enrrj7p)

  ```
  // INCLUDE input/output functions
  // DECLARE function prototype
  ```

```
// add three integer numbers and return the result

/* main program */
// BEGIN main
   // SET result
   // ADD result to add(1,2,3)
   // PRINT result
// END main

// DEFINE function
// add three integer numbers and return the result
// BEGIN add
   // RETURN result
// END add
```

- Sample solution:

```
// INCLUDE input/output functions
#include <stdio.h>
// DECLARE function prototype
// add three integer numbers and return the result
int add(int, int, int);

/* main program */
int main (void)
{ // BEGIN main
   // SET result
   int result;
   // ADD result to add(1,2,3)
   result = add(1,2,3);
   // PRINT "Result = " + result
   printf("Result = %d\n", result);
   return 0;
} // END main

// DEFINE function
// add three integer numbers and return the result
int add(int a, int b, int c)
{ // BEGIN add
  // RETURN result
```

```
  return (a + b + c);
} // END add


Result = 6
```

Exercise 2: Return the larger of two integers

- **Goal:** Write a function `max2` that accepts two `int` arguments and returns the larger of the two.
- **Sample call:** `max2(5, 9)`
- **Expected output:**

  ```
  The larger value is 9
  ```

- **Steps:**
  1. Write the function prototype for `max2`.
  2. In `main`, call `max2` with two numbers and store the result.
  3. Print the result in `main`.
  4. Define the `max2` function using an `if` statement.
- **Tip:** Use a simple conditional expression (e.g. `if (a > b)`).
- **Starter code**: onecompiler.com/c/43ent6a2m

  ```
  // INCLUDE input/output functions

  // DECLARE function prototype
  // return the larger of two integers

  /* main program */
  // BEGIN main
     // SET result
     // CALL max2 on two numbers
     // PRINT result
  // END main

  // DEFINE function max2
  // RETURN the larger of two inputs
  ```

- **Sample solution**:

```
#include <stdio.h>

// function prototype
int max2(int, int);

int main(void)
{
  int result = max2(5, 9);
  printf("The larger value is %d\n", result);
  return 0;
}

int max2(int a, int b)
{
  if (a > b)
    return a;
  else
    return b;
}
```

```
The larger value is 9
```

References

- Kernighan/Ritchie (1978). The C Programming Language (1st). Prentice Hall.
- King (2008). C Programming - A modern approach (2e). W A Norton.