Functions in C (part 2 of 3)

CMSC 104 Section 02
April 15, 2024

Administrative Notes

Quiz 4 next Wednesday (4/24)

- Same as always: in class, Blackboard;
- Will cover everything up through this Wednesday's (4/17) lecture
    - Functions part 3
    - Next Monday's lecture, on creating your own header files, is NOT on quiz 4
- Lots of classwork & homework
    - HW6 due tonight
    - CW7 due Wednesday
    - HW7 will be assigned at the end of today's lecture; due 4/22
    - More to come in the next few weeks

So what should you have learned about functions last week?

If you understand these things you're in great shape:

1. Functions in C are blocks of code, or modules, that are part of the program.
2. They have ***names***; they take zero or more inputs as ***parameters***, and they ***return*** zero or more values
3. Every C program has to have a function called main. In this class, main always returns one integer value, and it always takes zero parameters. That's why we always see `int main() {    }`
4. Every other function has a ***function prototype*** in the ***main*** function. It warns the C compiler that there will be function that looks like this defined later in the program.

What you should know so far, part 2

5. In this class, the ***function definition*** comes after the full ***main*** function. It starts with a type identifying what type of values are returned, followed by the function name, followed by a list of ***formal parameters*** including the type and name of each parameter; and then the code that makes up the function.

6. A function that has no return value is of type `void`. If there are no parameters to the function, you can either write (`void`) as the parameter list or you can leave an empty parameter list ( )

7. The ***function call*** occurs in the main program, or in another function*. The function call has the name of the function, followed by a list of ***actual parameters*** whose values are passed to the function.

*We haven't gotten to one function calling another yet.

So that’s where we are now. If you grok this, you’re in good shape.

Now on to new material

Today we’ll focus on:

- Parameter passing and matching
- Return values
- Local variables and variable scope

The basic program we’ll use

```
#include <stdio.h>

int main() {
  int height;  // the height of a right triangle
  int base;  // the base of a right triangle
  float a;   // the area of a right triangle with height and base

  float calc_area (int height, int base);  // the function prototype

  printf ("Please enter the height and base of a right triangle, separated by a tab. \n");
  printf ("We will calculate the area of that triangle \n");
  scanf("%d%d", &height, &base);
  a = calc_area(height, base);
  printf("The area of your triangle is %1.4f \n", a);

  return 0;

}

    float calc_area (int height, int base) {
      float area;

      area = 0.5 * height * base;
      return area;
    }
```

Parameter passing

The actual parameters in the function call and the formal parameters in the function definition are matched in order, NOT BY NAME

- There must generally be the same number of parameters, and they must be of the same type
    - In this class we’re not going to worry about the exceptions to that

C always passes the values from the actual parameters to the function using “call by value”

Call-by-value

C always uses call-by-value for parameter passing - what does that mean?

Each actual parameter has a value

- If the actual parameter is a literal, that’s the value
- If the actual parameter is a variable, the program goes to the symbol table, finds the location in memory, gets the value and sends a copy of that value to the function.

The function has its own symbol table, using a different part of memory. The passed value is stored there

- When the function ends, that function’s symbol table is deleted.

So what does this mean?

Among other things, that operations in a function cannot unintentionally modify variables and memory locations belonging to the main program

- This is mostly true; for this class we’ll operate as if it were a true statemen

Variable Scope

About that local symbol table for the function:

- If you need variables in the function code, you declare them in the function definition itself
    - These are called ***local variables***
- They're entered into the function's symbol table, not the main program's symbol table
    - It's a different part of memory
- You use them in the function
- When the function ends, that symbol table goes away and you lose access to the locations in memory

Return values

So the way to impact the main program is to return a value

- Using a return statement
- This returns one value, which is associated with the function name in the calling statement
- You can print that value out in the main program; you can assign the value to a variable;...

Can you return more than one value from a function?

- Yes; we’ll get to that later in the semester

If there's time: header files

- *Otherwise we'll talk about this on Monday*
- Anything that ends in ".h" is a header file
- Header files contain function prototypes for all of the functions found in the specified library.
  - They also contain definitions of constants and data types used in that library.
- By reusing code, especially from the standard C library, you reduce the chance for errors while making the code easier to maintain (instead of re-implementing everything yourself)

Header File	Contains Function Prototypes for:
stdio.h	standard input/output library functions, file input/output
math.h	math functions
ctype.h	functions for testing characters and data types
dirent.h	working with directories
errno.h	error handling
limits.h	sizes of basic types
locale.h	translating programs into different human languages
stddef.h	working with different specific-sized data types
string.h	functions for working with strings
time.h	Getting time and date information
unistd.h	Standard cross-Unix & Unix-like operating systems functions