Computer Science & Engineering 155E
Computer Science I: Systems Engineering Focus

Lecture 03 - Introduction To Functions

Christopher M. Bourke
cbourke@cse.unl.edu

Chapter 3

Existing Information

- Programmers seldom start off write completely original programs.
- Often the solution can be developed from information that already exists or from the solution to another problem.
- No point in "reinventing the wheel"
- Designing pseudocode generates important information before you even begin to code a program.
  - A description of a problem's data requirements,
  - A description of a problem's solution algorithm,
- This provides a starting point in coding your program.
  - What portions of this program can be taken care of by standard library functions?
  - What portions of this code can be grouped into a stand-alone function?

Library Functions

- C has several standard libraries that have been developed over the last several decades
  - Input/Output libraries
  - Math libraries
  - Helpful standard libraries
  - etc.
- Libraries are provided for the purpose of *code reuse*
- Highly optimized and thoroughly tested

Predefined Functions and Code Reuse

- A primary goal of software engineering is to write error-free code.
  - *Code reuse*, reusing program fragments that have already been written and tested
- C promotes reuse by providing many predefined functions that can be used to perform mathematical computations.

Standard Math Library

- Functions such as `sqrt` are found in the *standard math library* to perform the square root computation.
  - The function call in the assignment statement `y = sqrt(x);` activates the code for function `sqrt`, passing the argument `x` to the function.
  - After execution, the result of the function is substituted for the function call.
  - If `x` is 16.0, the assignment statement above is evaluated as follows: $\sqrt{16.0}$ is evaluated to 4.0, the call `sqrt(x)` is replaced with 4.0, and then y takes the value 4.0.
  - To include, use: `#include<math.h>`
  - Pitfall: when compiling with `gcc` using the standard math library, you must use the flag, `-lm`:
    `prompt:>gcc -lm myMathProgram.c`

C Library Functions

Examples

Function	#include	Description
abs(x)	stdlib.h	integer absolute value $|x|$
fabs(x)	math.h	double absolute value
ceil(x)	math.h	Returns ceiling value, ceil(46.3)=47.0
floor(x)	math.h	Returns floor value, floor(46.3)=46.0
cos(x)		
sin(x)	math.h	Input in radians
tan(x)		
exp(x)	math.h	Returns e^x
log(x)	math.h	Natural log, $ln(x), x > 0$
log10(x)	math.h	Log base 10, $log_{10}(x), x > 0$
pow(x,y)	math.h	Returns x^y
sqrt(x)	math.h	Returns square root.

Function Specifics

- ► abs(x) is the only function listed with an int value argument and result.
- ► All others have both double as the argument and double as the result.
- ► tan(x), cos(x) and sin(x) take as their input the *radians*
- ► If one of the functions in the next frame is called with an argument that is not arguments data type, the argument value is converted to the required data type before it is used.
  - ► Conversion of type int to type double cause no problems, but a conversion of type double to type int leads to the loss of any fractional part.
- ► The value for sqrt, log and log10 must be positive.
- ► Invalid inputs may result in NaN, inf, -inf etc.

Functions Without Arguments

In C, functions have three important parts:

- ► Function Prototypes - contains the name, return type and arguments of a function
- ► Function Definitions - the implementation of the function
- ► Placement of Functions in a Program - how do we use functions?

Functions without Arguments

Top-Down Design: Problem-Solving method in which you break a large problem into smaller, simpler, subproblems.

- ► Programmers implement top-down design in their programs is by defining their own functions.
  - ► Write one function (subprogram) for each subproblem
  - ► Case Study, Section 3.3

  To begin, we focus on simple functions that have no arguments and no return value (void functions).

Function Prototypes

- ► As with other identifiers in C, a function must be declared before it can be referenced (used).
- ► One way to declare a function is to insert a *function prototype* before the main function.
- ► A function prototype tells C compiler the **data type** of the function, the function **name**, and information (number, data type) about the **arguments** that the function expects.
  - ► **Data Type** of the function is the type of value returned by the function.
  - ► Functions that return no value are of type void

Function Definitions

- ► The function prototype (i.e. Declaration) does not specify the function operation (what it does).
  - ► The variable declaration: int c; does not tell you how c will be used.
- ► To do this, you need to provide a definition for each function subprogram (similar to the definition of the main function).
- ► The function **heading** is similar to the function prototype, but *not* ended by the symbol ';'.
- ► The function **body** (enclosed in braces) contains the implementation of the function (specifies what it does)
- ► The return statement is optional for void functions

Function Prototypes & Definition

Example

```
1
2    /* function prototype */
3    void printProgramInfo();
4
5    int main(void)
6    {
7      ...
8      return 0;
9    }
10   ...
11
12   /* function definition */
13   void printProgramInfo()
14   {
15     printf("Program Example for CSCE 150A\n");
16     printf("  copyright(c) C. Bourke 2009\n");
17     return;
18   }
```

Function Definition

Scope

- Each function body may contain declarations for its own variables.
- These variables are considered *local* to the function
- They can be referenced only within the function.
- No other function has access to their values and they are destroyed after the return statement.
- This is known as a variable's *scope*

Placement of Functions in a Program

- The placement of function prototypes and definitions is important.
- The compiler is dumb: it must at least be told a function exists before it can use it.
- Function prototypes should appear between the after the #include or #define directives but before the main function.
- The subprogram definition follows the end of the main function.
- The relative order of the function *definitions* does not affect their order of execution; that is determined by the order of execution of the function call statements.

Full Example

```
1    /* Function Hello, World */
2    #include <stdio.h>
3
4    /*Function Prototypes */
5    void Hello_World(void);
6
7    int main(void) {
8        Hello_World();
9        return 0;
10   }
11
12   /* Function Definitions */
13   /* Prints Hello, World */
14   void Hello_World() {
15       printf("Hello, World\n");
16   }
```

Displaying User Instructions

- Simple functions have limited capability.
- Without the ability to pass information into or out of a function, we can use functions only to do *local* computation
- Example: display multiple lines of program output, instructions to a program user or a title page or a special message that precedes a program's result.

Functions with Input Arguments

- void Functions with Input Arguments
- Functions with Input Arguments and a Single Result
- Functions with Multiple Arguments
- Argument List Correspondence
- The Function Data Area
- Testing Functions Using Drivers

Functions with Input Arguments

- ► Arguments of a function are used to carry information into the function subprogram from the `main` function (or from another function subprogram) or to return multiple results computed by a function subprogram.
  - ► Arguments that carry information into the function are called **input arguments**;
  - ► Arguments that return results are called **output arguments**.
- ► We can also return a single result from a function by executing a `return` statement in the function body.

void Functions with Input Arguments

- ► Functions without arguments are too limited.
- ► We can use a `void` function with an argument to "dress up" our program output by having the function display its argument value in a more attractive way.
- ► (Recall that a `void` function does not return a result.)

void Functions with one Input Argument

```
1  /* Displays a real number in a box. */
2
3  void print_rboxed(double rnum)
4  {
5      printf("+---------+\n");
6      printf("|         |\n");
7      printf("|  %7.2f  |\n", rnum);
8      printf("|         |\n");
9      printf("+---------+\n");
10 }
```

Functions with Input Argument and a Single Result

- ► C functions can only ever return *one* value
- ► `sqrt(x)`, `abs(x)`, `pow(x,y)` etc. return a single double value
- ► May return any built-in type or user defined type

Problem

Design two functions to compute the area and circumference of a circle using one input argument to each (the radius).

Answer

```
1  double find_circum(double r)
2  {
3      return (2.0 * PI * r);
4  }
5
6  double find_area(double r)
7  {
8      return (PI * pow(r,2));
9  }
```

Answer Continued

- ► Each function heading begins with the reserved word `double`
- ► Indicates both return a double-type number
- ► Both function bodies consist of a single `return` statement.
- ► Its assumed that `PI` is defined via a *global* preprocessor directive.
- ► Utilizes the standard math library!
- ► We would call this function just like with math library functions:
  `areaOfCircle = find_area(3.5);`

Additional Considerations

- What happens if we pass a negative value to `find_area`?
- Can we make it more efficient?
- Can we make it more readable?

Better Area Function

```
1  /*
2   * Compute the area of a circle
3   * Input: double radius
4   * Return Value: area
5   */
6  double find_area(double radius)
7  {
8      double area;
9      if(radius < 0)
10         area = 0.0;
11     else
12         area = 3.14159265 * radius * radius;
13     return area;
14 }
```

Functions with Multiple Argument

- Functions `find_area` and `find_circum` each have a single argument.
- We can also define functions with multiple arguments.
- We can have as many arguments (inputs) as we want, but the number must be fixed.

```
1  /*
2   * Multiplies its first argument by 10 raised
3   * its second power, i.e.
4   * x * 10^y,
5   * where x is the first argument and y
6   * is the second argument
7   */
8  double scale(double x, int y)
9  {
10     double scale_factor;
11     scale_factor = pow(10, y);
12     return (x * scale_factor);
13 }
```

Argument List Correspondence

- When using multiple-argument functions, be careful to include the correct number of arguments in the function call.
- The order or the actual arguments used in the function call *must* correspond to the order of the formal parameters listed in the function prototype.
- The *types* of each argument must match when calling the function: do not pass a `double` into a function where the formal parameter is data type `int`

The Function Data Area

- Each time a function call is executed, an area of memory is allocated (system stack) for storage of that function's data.
- Included in the function data area are storage cells for its formal parameters and any local variables that may be declared in the function.
- The function data are is always lost when the function terminates; it is recreated empty when the function is called again

Testing Functions Using Drivers

- A function is an independent program module, meaning it can be tested separately from the program that uses it.
- To run such a test, you should write a short **driver** function.
- A driver function defines the function arguments, calls the functions, and displays the value returned.

Wrap-Up

- Program Style
- Order of Execution of Function Subprograms and Main Function
- Advantages of Using Function Subprograms
- Displaying User Instructions

Order or Execution

- Prototypes for the function subprograms appear before the `main` function so that the compiler can process the function prototypes before it translates the `main` function.
    - The information in each prototype enables the compiler to correctly translate a call to that function.
- After compiling the `main` function, the compiler translates each function subprogram.
- During translation, when the compiler reaches the end of a function body, it inserts a machine language statement that causes a transfer of control back from the function to the calling statement.

Advantages of Using Function Subprograms

There are many advantages to using function subprograms.

- General programming
- Procedural Abstraction
- Reuse of Function Subprograms

General Programming

- Their availability changes the way in which an individual programmer organizes the solution to a programming problem
- For a team of programmers working together on a large problems, each member can focus on solving a set of subproblems.
- Simplify programming tasks by providing building blocks for new programs.

Procedural Abstraction

- Function subprograms allow us to remove from the `main` function the code that provides the detailed solution to a subproblem.
    - Because these details are provided in the function subprograms and not in the `main` function, we can write the `main` function as a sequence of function call statements as soon as we have specified the initial algorithm and before we refine any of the steps.
    - We should delay writing the function for an algorithm step until we have finished refining the previous step.
- With this approach to program design, called **procedural abstraction**, we defer implementation details until we are ready to write an individual function subprogram.
- Focusing on one function at a time is much easier than trying to write the complete program at once.

Reuse of Function Subprograms

Another advantage of using function subprograms is that functions can be executed more than once in a program.

Finally, once you have written and tested a function, you can use it in other programs or functions.

Common Programming Errors

- Remember to use a `#include` preprocessor directives for every standard library from which you are using functions.
- Use the `-lm` option when compiling code using the `math.h` standard library.
- Place prototypes for your own function subprogram in the source file preceding the main function; place the actual function definitions after the main function.
- The acronym **not** summarizes the requirements for argument list correspondence:
  - Provide the required **n**umber of arguments,
  - Make sure the **o**rder of arguments is correct, and
  - Each function argument is the correct **t**ype or that conversion to the correct type will lose no information.
- Also be careful in using functions that are undefined on some range of values.

Questions?

Questions?

Exercise

Problem

Design a program that takes prompts for inputs, a, b, c and uses two functions `quadraticRootOne`, `quadraticRootTwo` which return the real-valued roots of the quadratic equation,
$$ax^2 + bx + c$$

Hint: recall the quadratic equation:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Be sure to format your output (you may assume that the input doesn't result in any complex roots, that is, $b^2 \geq 4ac$).