

Computer Science & Engineering 150A

Problem Solving Using Computers

Lecture 03 - Functions

Stephen Scott
(Adapted from Christopher M. Bourke)

Fall 2009

3.1 Building Programs from Existing Information

3.2 Library Functions

3.4 Functions without Arguments

3.5 Functions with Arguments

3.6 Common Programming Errors

- Programmers seldom start off writing 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?

- 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

- A primary goal of software engineering is to write error-free code.
 - This is facilitated by *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.

- 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 `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>`
 - **Note:** Generally, when using the math library, you must use `-lm`:
`prompt:>gcc -lm myMathProgram.c`
(though cse's `gcc` doesn't require it)

C Library Functions

Examples

CSCE150A

Introduction

Library
Functions

Functions

Wrap-Up

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

- `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 a function is called with an argument that is not the argument's data type, the argument 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 arguments for `sqrt`, `log` and `log10` must be positive.
- Invalid inputs may result in `NaN`, `inf`, `-inf`, etc.

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?

Top-Down Design: Problem-solving method in which one breaks a large problem into smaller, simpler, subproblems.

- Programmers utilize top-down design in their programs 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).

- 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.
 - Standard library functions (`printf`, `scanf`, `sqrt`, etc.) are prototyped in header (`.h`) files
- 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`

- The function prototype (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

CSCE150A

Introduction

Library
Functions

Functions

No Arguments

With Arguments
Multiple Args

Wrap-Up

```
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(void)
14 {
15     printf("Program Example for CSCE 150A\n");
16     printf("    copyright(c) C. Bourke 2009\n");
17     return;
18 }
```

Function Definition

Scope

CSCE150A

Introduction

Library
Functions

Functions

No Arguments
With Arguments
Multiple Args

Wrap-Up

- 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*

Function Definition

Scope Example

CSCE150A

Introduction

Library
Functions

Functions

No Arguments

With Arguments

Multiple Args

Wrap-Up

```

1      /* function prototype */
2      void hereIsAFunction();
3
4      int main(void)
5      {
6          int x;
7          ...
8      }
9
10     /* function definition */
11     void hereIsAFunction(void)
12     {
13         int y;
14         ...
15     }
    
```

`main` cannot see `y` and `hereIsAFunction` cannot see `x`

Placement of Functions in a Program

CSCE150A

Introduction

Library
Functions

Functions

No Arguments

With Arguments
Multiple Args

Wrap-Up

- The placement of function prototypes and definitions is important.
- The compiler must be told that a function exists before it can be used
- Function prototypes should appear after the `#include` and `#define` preprocessor directives and before the `main` function.
- The function definitions follow 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 the function call statements.

Full Example

CSCE150A

Introduction

Library
Functions

Functions

No Arguments

With Arguments

Multiple Args

Wrap-Up

```

1  /* Program 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(void) {
15     printf("Hello, World\n");
16 }
    
```

- 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

CSCE150A

Introduction

Library
Functions

Functions

No Arguments

With Arguments

Multiple Args

Wrap-Up

- `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

CSCE150A

Introduction

Library
Functions

Functions

No Arguments

With Arguments

Multiple Args

Wrap-Up

- 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** and are denoted by the ampersand `&` (e.g. used in `scanf`)
- We can also return a single result from a function by executing a **return** statement in the function body.
 - This is preferred over output arguments unless the latter are absolutely necessary

void Functions with Input Arguments

CSCE150A

Introduction

Library
Functions

Functions

No Arguments

With Arguments

Multiple Args

Wrap-Up

- 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

CSCE150A

Introduction

Library
Functions

Functions

No Arguments

With Arguments

Multiple Args

Wrap-Up

```

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

CSCE150A

Introduction

Library
Functions

Functions

No Arguments

With Arguments

Multiple Args

Wrap-Up

- C functions can only ever return *one* value
- `sqrt(x)`, `abs(x)`, `pow(x,y)`, etc. return one value of type `double`
- 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).

```

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)

CSCE150A

Introduction

Library
Functions

Functions

No Arguments

With Arguments

Multiple Args

Wrap-Up

- 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

CSCE150A

Introduction

Library
Functions

Functions

No Arguments

With Arguments

Multiple Args

Wrap-Up

- 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

CSCE150A

Introduction

Library
Functions

Functions

No Arguments

With Arguments

Multiple Args

Wrap-Up

```

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 Arguments

CSCE150A

Introduction

Library
Functions

Functions

No Arguments
With Arguments

Multiple Args

Wrap-Up

- 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 is fixed.
 - Can have exceptions to this (e.g. `printf`, `scanf`), but this is not common

```
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

CSCE150A

Introduction

Library
Functions

Functions

No Arguments
With Arguments
Multiple Args

Wrap-Up

- 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 *type* 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

CSCE150A

Introduction

Library
Functions

Functions

No Arguments
With Arguments

Multiple Args

Wrap-Up

- 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 always lost when the function terminates; it is recreated empty when the function is called again

Testing Functions Using Drivers

CSCE150A

Introduction

Library
Functions

Functions

No Arguments
With Arguments

Multiple Args

Wrap-Up

- 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.

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

- 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

CSCE150A

Introduction

Library
Functions

Functions

Wrap-Up

There are many advantages to using function subprograms.

- General programming
- Procedural Abstraction
- Reuse of Function Subprograms

- 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 problem, each member can focus on solving a set of subproblems.
- Simplify programming tasks by providing building blocks for new programs.

- 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.
- E.g. program `circle` in textbook

Reuse of Function Subprograms

CSCE150A

Introduction

Library
Functions

Functions

Wrap-Up

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

CSCE150A

Introduction

Library
Functions

Functions

Wrap-Up

- Remember to use a `#include` preprocessor directive for every standard library from which you are using functions.
- Use appropriate options for the compiler to link in the libraries you need
- 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**rders 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?

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, $b^2 \geq 4ac$).