Problem Solving and Program Design - Chapter 3

Cory L. Strope

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.
- Following the software development method generates important documentation 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 documentation can be used as a starting point in coding your program.
  - To develop the executable statements in the `main` function, first use the initial algorithm and its refinement as program comments.

Library Functions

- Predefined Functions and Code Reuse
- Use of Color to Highlight New Constructs
- C Library Functions
- A Look at Where We Are Heading

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.
- 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.

Use of Color to Highlight New Constructs

The book will use color for the purpose to illustrate new constructs.

C Library Functions I

Function	#include	Description
`abs(x)`	<stdlib.h>	integer absolute value abs(-5)=5
`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 it radians, outputs *trig_func_val*, in angles.
`tan(x)`		
`exp(x)`	<math.h>	Returns e^x
`log(x)`	<math.h>	Natural log, x > 0
`log10(x)`	<math.h>	Log base 10, x> 0
`pow(x,y)`	<math.h>	Returns x^y
`sqrt(x)`	<math.h>	Returns square root.

When using the CSE math.h library, compile your code using:

```
cc username_<problem #>.c -lm
```

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* and output the angle.
  - For example, `sin(1.57) = 1`, `cos(1.57) = 0`, and `tan(1.57) =` ∞.
- 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.

A Look at Where We Are Heading

C also allows us to write our own functions. Let's assume that we have already written functions `find_area` and `find_circum`

- Function `find_area(r)` returns the area of a circle with radius `r`
- Function `find_circum(r)` returns the circumference of a circle with radius `r`

Functions without Arguments

- Function Prototypes
- Function Definitions
- Placement of Functions in a Program

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, p. 112, Sec. 3.3

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

Function Prototypes

- As with other identifiers in C, a function must be declared before it can be referenced.
- 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`
  - Ex: `int main` {$\cdots$ `return 0;` }

Function Definitions

- The function prototype (i.e. Declaration) does not specify the function operation.
  - 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).

```
/* Draws a circle */
void draw_circle(void) {
    printf("   *  \n");
    printf(" *   *\n");
    printf("  * * \n");
}
```

- The function **heading** is similar to the function prototype, but *not* ended by the symbol ';'.
- The function **body** (enclosed in braces) is three calls to function `printf` that cause the computer to display a circular shape.
- The `return` statement because `draw_circle` does not return a result.

Example

```
#include <stdio.h>

/* Function prototypes */
void draw_circle(void);
void draw_triangle(void);

int main(void) {

   draw_triangle();
   draw_circle();
   return 0;
}

/* Function Definitions below */
```

Function Definition

- Each function body may contain declarations for its own variables.
- These variables are considered *local* to the function; in other words, they can be referenced only within the function.

Placement of Functions in a Program

The next slide shows a complete program with function subprograms.

- The subprogram prototype appear between the `main` function any `#include` or `#define` directives.
- 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.

Example

```
/* Function Hello, World */
#include <stdio.h>

/*Function Prototypes */
void Hello_World(void);

int main(void) {
   Hello_World();
   return 0;
}

/* Function Definitions */
/* Prints Hello, World */
void Hello_World() {
    printf("Hello, World\n");
}
```

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 display multiple lines of program output, such as 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

- In the last section, we used `void` functions like `draw_circle` to display several lines of program output.
- 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.)

```
/* Displays a real number in a box. */

void print_rboxed(double rnum) {

   printf("***********\n");
   printf("*           *\n");
   printf("*  %7.2f *\n", rnum);
   printf("*           *\n");
   printf("***********\n");

}
```

Functions with Input Argument and a Single Result

- The most common (pre-defined) function definition returns one result:
  - `sqrt(x)`, `abs(x)`, `pow(x,y)` $\cdots$
- Consider the problem of finding the area and circumference of a circle using functions with just one argument.

```
/* Compute the circumference of a circle with radius r */
double find_circum(double r) {
   return (2.0 * PI * r);
}

/* Compute the area of a circle with radius r */
double find_area(double r) {
   return (PI * pow(r,2));
}
```

- Each function heading begins with the reserved word `double`, indicating that the function result is a "real" number.
- Both function bodies consist of a single `return` statement.
  - When either function executes, the expression in its `return` statement is evaluated and returned as the function result.

Functions with Multiple Argument

Functions `find_area` and `find_circum` each have a single argument. We can also define functions with multiple arguments.

```
/* Multiplies its first argument by 10 raised to a power
 * i.e. x * 10^y, where x is the first argument and y
 * is the second argument
 * Pre.: double x and int y and math.h
 * Post: called value
 */

double scale(double x, int y) {

   double scale_factor;
   scale_factor = pow(10, y);

   return (x * scale_factor);
}
```

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.

Finally, if the function is to return meaningful results, assignment of each argument to the corresponding formal parameter (i.e. parameter passed into the function) must not cause any loss of information (Such as passing 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 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 area 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

Program Style

Each function begins with a comment that describes its purpose.

If the function subprograms were more complex, we would include comments on each major algorithm step just as we do in function `main`.

Block comments and heading that begins each function in contain all the information required in order to use the function.

- Block comments begin with a statement of what the function does
- The next lines show what values the function has as Input Arguments (**preconditions)**.
- The last line shows what value the function returns **(postcondition)**.

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 (`*.c`) preceding the main function; place the actual function definitions after the main function.
- The acronym **not** summerizes 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.