CS 1023 Intro to Engineering Computing 3: Computer Programming
LM4 – Programmer Defined Functions

Mark Kerstetter
Department of Computer Science
Western Michigan University

© Fall 2015
Learning about Functions

What do you already know?

- Built-in functions & How to Use Them
  - $h = \sqrt{a^2 + b^2}$;
  - `printf(“sin(%f) = %fn”, theta, sin(theta));`
  - `scanf(“%lf %lf”, &a, &b);`

You already know how to **use** functions!

Three things you need to learn about writing your own functions

- **Programmer-defined functions**
- Declare a function **before** you use it
- How to use a function (Already doing this!)
- Define what a function does by writing code for it

Now you will learn how to write your own functions.

You need to do **five** things.

The parameters that are actually used when a function is called or used are known as the **actual parameters**.

The parameters that are named and used within both a function declaration and definition are known as the **formal parameters**.

These **formal parameters** act like placeholders and are used to receive the values or addresses of the **actual parameters** when the function is called (i.e., actually used).
Programmer-Defined Functions

1. Understand the Function & Sketch Solution – What is it supposed to do?

2. Name the Function

3. Decide What Kind of Data the Function Returns – Return Type
   - Function performs activity, but does not return a single value
     - E.g., printf and openVatDoor (perform activities)
     - Return type void
   - Function returns a single value – What type?
     - E.g., sqrt, sin, max, firstChar (return int, double, char, etc.)
     - Return type int, double, or char
   - Function returns multiple values
     - E.g., scanf, sort, swap, reverse
     - Return type void (changes returned through parameters)

4. Decide Parameters
   - Number – How many parameters are needed?
   - Order – Which parameter is 1st, 2nd, 3rd, etc.?
   - Communication – How are parameters used to communicate data?
     - pass-by-value (or call-by-value) – one-way communication into the function
     - pass-by-reference (or call-by-reference) – two-way communication into and out of the function
   - Type of each parameter – What data type is associated with each parameter?
Parameter Communication in Functions

- Two Kinds of Parameter Communication between the Calling Program and a Function (Called Program)
  - One-way Communication into a Function
  - Two-way Communication into and/or out of a Function

- One-way Communication into a Function
  - *call-by-value* –or– *pass-by-value*
  - Copies the value of the actual parameter into the dummy/formal parameter in the function. Changes to the dummy/formal parameter can **not** be sent back using the *call-by-value* parameter!
  - Protects variables and data in the calling program
  - Examples of using functions with *call-by-value* parameters
    - `h = sqrt ( a*a + b*b ) ;`
    - `printf ( "sin(%f) = %f\n", theta, sin( theta ) ) ;`
Parameter Communication in Functions

- Two Kinds of Parameter Communication between the Calling Program and a Function (Called Program)
  - One-way Communication into a Function
  - Two-way Communication into and/or out of a Function

- Two-way Communication into and/or out of a Function
  - call-by-reference –or– call-by-address
    –or– pass-by-reference –or– pass-by-address
  - Both the actual parameter and the dummy/formal parameter share the same memory (storage) address. This means changing the value when a dummy/formal parameter is used within a function also changes the value of the actual parameter in the calling program!

- Examples of using functions with call-by-reference parameters
  \( \texttt{scanf ( "\%lf \%lf", \&a, \&b );} \)
Programmer-Defined Functions

- **Step 1 - Declare** the function using a *function prototype/signature/heading* followed by a semicolon
  - returnType functionName(type₁ formalPar₁, … , typeₙ formalParₙ);

- **Step 2 - Use** the function within your main program or within other functions
  - *Actual parameters* must conform to (agree with) *formal parameters*
  - Result type must conform to (agree with) actual use

- **Step 3 - Define** the function code *(see example)*
  - Follows pattern of writing a main program
  - Function definitions follow one another and follow (or precede) main program
    NEVER nest function definitions within each other or within the main program
  - return-statement(s)
  - local declarations – names and values created within the function
    stay within the function
Write a Function - Example

■ Write a function to raise a number to a power, e.g., \( x^n \)

■ What should we call the function?
  o Someone tells you, e.g., client, supervisor, etc.
  o Free to select the name yourself
  o Name the function `power`

■ What should be the type of result?
  o one number with the most precision, e.g., `double`

■ Parameters?
  o two parameters the `base` and the `exponent`
  o `base` is 1\(^{st}\) parameter, `exponent` 2\(^{nd}\) parameter
    Why? Naturally occur in that order. Easy to remember and understand.
  o both parameters need not be changed, so we’ll use `pass-by-value`
    communication; we’ll return the single answer using the `function return`
  o `base` type should be `double`
    `exponent` type will be `integer`
Programmer-Defined Functions

Power Function

How used? Means?
or1 = power(x, 3); \quad x^3
or2 = power(y, n); \quad y^n
or3 = power(3*z, m+2); \quad (3z)^{m+2}

Function Definition

Only works for exponents that are \( \geq 0 \)!

features: function return value; pass-by-value (one-way) parameters;
local identifiers (e.g., k and result)
Programmer-Defined Functions

Power Function

### Function Definition

**power**

```c
double power(double base, int expo)
{
    int k ; double result ;    /* local temporaries */
    result = 1 ;
    if ( expo < 0 ) { base = 1.0/base; expo = -expo; }  
    for ( k=1 ; k <= expo ; k++ )
    {
        result = result * base ;
    }
    return result ;
}
```

### Features:
- **Function return value**;
- **pass-by-value** (one-way) parameters;
- **local identifiers** (e.g., `k` and `result`)

### Use:
- `r1 = power(x, -3);`
- `r2 = power(y, n);`
- `r3 = power(3*z, -(m+2));`

### Means:
- `x^{-3}`
- `y^n`
- `(3z)^{-(m+2)}`

### Make it work for all integer exponents!

if ( expo < 0 ) { base = 1.0/base; expo = -expo; }
Program Skeleton vs. Function Skeleton

```
int main ( )
{
  declaration-part ;
  . .
  statement-part ;
}
```

```
double fnName (double n1, n2)
{
  declaration-part ;
  . .
  statement-part ;
}
```

This is an entire function definition statement!
The function definition is more than just the first line. The function includes the first line known as the function signature or function heading or function prototype and the body of the function which is enclosed in braces { and }.

- first line of both look similar, except for parameters
- body, i.e. part between { and } has same structure in both
What C programming non-executable statements do you know?

- programmer comments
  
  /* This part of the program does … */

- main program structure
  
  int main () { … }

- preprocessor commands
  
  #include <standard library headers>
  
  #define SYMBOLIC_CONSTANT value
  
  #define FILE_NM_CONST "filename.extension"

- declarations – variables & files
  
  int listOfVariablesOfIntegerType ;
  
  double listOfVariablesOfDoubleType ;
  
  FILE *fileVariable ;
  
  double swap (double *x, double *y) ;

- statement grouping symbols
  
  { and }

Comments tell humans, other programmers, what’s happening.

EVERY C program builds on this template.

Preprocessor commands begin with a # and don’t end with a ;
Preprocessor commands tell the compiler about predefined symbols, variables, and functions.

Program variables and programmer-defined functions MUST be declared before being used!
Tools Review – Executable Statement

- What C programming **executable** statements do you know?
  - **assignment statement**
    - `variable = expression ;`
    - What is an **expression**? Anything that produces a value, e.g.,
      - A constant, a variable, several variables and/or constants and/or functions (either built-in or programmer-defined) legally hooked together with appropriate operators (operators could be arithmetic, relational, or logical).
  - **selection statements**
    - `if (condition) … ;`
    - `if (condition) … ; else … ;`
  - **loop statements**
    - `while (condition) { … } ;`
    - `do { … } while (condition) ;`
    - `for (startValue; stopCondition; incrementExpression) { … } ;`
  - **functions** (either built-in or programmer-defined **void**)
  - **return statement**
    - `return expression ;`
    - `return ;`

This is the **entire** if-statement!
This is the entire else-statement! When it is used, it ALWAYS immediately follows an if-statement.
This is the **entire** while-loop-statement!
This is the entire do-while-loop-statement!
This is the entire for-loop-statement!
Passing Parameters to/from Functions

*Pass-by-Value*
Copy actual to formal, but not back.

*Pass-by-Reference*
Share same memory address, but only temporarily.
User-Defined Functions

**User-Defined Functions – Why?**

- Give single name to a logical task.
  - sort, search, findArea, swap, computeGrade, etc.
- Make programs more readable
  - getMasterList( ); getChanges( ); mergeLists( ); displayMasterList( )
- Make it easier to interchange parts
  - E.g., replace a function by a more efficient one
- Design and Build Program using Component Parts
  - Subdivide and distribute work to different programmers

**Illustrate Programmer-Defined Function – swap(&x, &y)**

- Function Declaration – Prototype/Signature Declaration
  - type of return value or no return value
  - function name
  - formal/dummy parameters – number, order, types, and use
- Calling/Using a Programmer-Defined Function
  - Just like using built-in functions
- Defining a Programmer-Defined Function
  - prototype/signature declaration (copy used as function declaration)
  - local variables
  - executable statements
Pass-by-Reference Functions
(Two-way Parameters)

- **Programmer-Defined Swap Function**
  - Using the swap-function
    
    ```
    swap (&x, &y) ;
    ```
  - Declaration of for the swap-function – function prototype/signature
    
    ```
    void swap (double *par1, double *par2) ;
    ```
Programmer-Defined Functions

How used?
swap(&x,&y);

How not used?
swap(x+5, y);

void swap(double *x, double *y)
{
    double temp;
    temp = *x;
    *x = *y;
    *y = temp;
    return;
}

Features: function does not return a value (i.e., void);
pass-by-reference (two-way) parameters; pointer to variables (i.e., *name);
local identifiers (e.g., temp)
Function Creation Exercise

- Create a function that computes the area and perimeter of a rectangle given the lengths of the two sides.
  - What should we name it?
  - How many parameters should it have?
  - What type, if any, should return?
  - What are the parameters? What types of parameters? How should parameters communicate between caller and the function?

```c
void CalcRectAreaPerim( double L, double W, double *area, double *perimeter )
{
    *perimeter = 2*L + 2*W;
    *area = L * W;
    return;
}
```