Functions: Part 1 of 3

CMSC 104, Fall 2012

John Y. Park

Functions, Part 1 of 3

Topics

- Using Predefined Functions
- Programmer-Defined Functions
- Using Input Parameters
- Function Header Comments

Reading

Review of Structured Programming

- Structured programming is a problem solving strategy and a programming methodology that includes the following guidelines:
  - The program uses only the sequence, selection, and repetition control structures.
  - The flow of control in the program should be as simple as possible.
  - The construction of a program embodies top-down design.

Review of Top-Down Design

- Involves repeatedly **decomposing** a problem into smaller problems
- Eventually leads to a collection of small problems or tasks each of which can be easily coded
- The **function** construct in C is used to write code for these small, simple problems.

Functions

- A C program is made up of one or more functions, one of which is main( ).
- Execution always begins with main( ), no matter where it is placed in the program. By convention, main( ) is located before all other functions.
- When program control encounters a function name, the function is **called** (**invoked**).
  - Program control passes to the function.
  - The function is executed.
  - Control is passed back to the calling function.

Sample Function Call

```
#include <stdio.h>

int main ( )
{
     printf (“Hello World!\n”) ;
     return 0 ;
}
```

printf is the name of a **predefined function** in the stdio library

this statement is is known as a **function call**

this is a string we are **passing** as an **argument** (**parameter**) to the printf function

Functions (con't)

- We have used three predefined functions so far:
  - printf
  - scanf
  - getchar
- Programmers can write their own functions.
- Typically, each module in a program's design hierarchy chart is implemented as a function.

Sample Programmer-Defined Function

```
#include <stdio.h>

void PrintMessage ( void ) ;

int main ( )
{
    PrintMessage ( ) ;
    return 0 ;
}

void PrintMessage ( void )
{
    printf ("A message for you:\n\n") ;
    printf ("Have a nice day!\n") ;
}
```

Examining printMessage

```
#include <stdio.h>

void PrintMessage ( void ) ;

int main ( )
{
    PrintMessage ( ) ;
    return 0 ;
}

void PrintMessage ( void )
{
    printf ("A message for you:\n\n") ;
    printf ("Have a nice day!\n") ;
}
```

function **prototype**

function call

function **header**

function **body**

function **definition**

The Function Prototype

- Even though this comes first, we'll describe this last…

The Function Call

- Passes program control to the function
- Must match the prototype in name, number of arguments, and types of arguments

```
void PrintMessage (void) ;

int main ( ) same name   no arguments
{
        PrintMessage ( ) ;
        return 0 ;
}
```

The Function Definition

- Control is passed to the function by the function call. The statements within the function body will then be executed.

```
void PrintMessage ( void )
{
      printf ("A message for you:\n\n") ;
      printf ("Have a nice day!\n") ;
}
```

- After the statements in the function have completed, control is passed back to the **calling function**, in this case main( ) .
  Note that the calling function does not have to be main( ) .

The Function Prototype

- (Now, we're ready for this) It informs the compiler that there will be a function defined later that:

returns this type

has this name

takes these arguments

```
void printMessage (void) ;
```

- Needed because the function call is made before the definition -- the compiler uses it to see if the call is made properly

General Function Definition Syntax

type functionName (*parameter*$_1$, . . . , *parameter*$_n$)
{
    variable declaration(s)
    statement(s)
}

- If there are no parameters, either

  functionName( )   OR   functionName(void)

  is acceptable.
- There may be no variable declarations.
- If the **function type** (**return type**) is void, a return statement is not required, but the following are permitted:

  return ;   OR   return( ) ;

Using Input Parameters

```
void PrintMessage (int counter) ;
int main ( )
{
    int num;
    printf ("Enter an integer: ") ;
    scanf ("%d", &num) ;
    PrintMessage (num) ;
    return 0 ;
}

void PrintMessage (int counter)
{
    int i ;
    for ( i = 0; i < counter; i++ )
    {
        printf ("Have a nice day!\n") ;
    }
}
```

one argument of type int

matches the one **formal parameter** of type int

Final “Clean” C Code

```
#include <stdio.h>

void PrintMessage (int counter) ;

int main ( )
{
    int num ;      /* number of times to print message */

    printf (“Enter an integer: “) ;
    scanf (“%d”, &num) ;
    PrintMessage (num) ;

     return 0 ;
}
```

Final “Clean” C Code (con’t)

```
/*************************************************************************
** PrintMessage - prints a message a specified number of times
** Inputs:  counter - the number of times the message will be
**                          printed
** Outputs:  None
*************************************************************************/
void PrintMessage ( int counter )
{
    int i ;   /* loop counter */

    for ( i = 0; i < counter; i++ )
    {
        printf (“Have a nice day!\n”) ;
    }
}
```

Good Programming Practice

- Notice the **function header comment** before the definition of function PrintMessage.
- This is a good practice and is required by the 104 C Coding Standards.
- Your header comments should be neatly formatted and contain the following information:
  - function name
  - function description (what it does)
  - a list of any input parameters and their meanings
  - a list of any output parameters and their meanings
  - a description of any special conditions