Chapter 4: Subprograms Functions for Problem Solving

Mr. Dave Clausen

La Cañada High School

Objectives

- To understand the concepts of modularity and bottom up testing.
- To be aware of the use of structured programming.
- To understand the need for user defined functions.
- To be able to use correct syntax when declaring and implementing a function.
- To be able to use stubs and drivers to test functions.
- To be able to understand the need for and appropriate use of parameters in functions.
- To be able to understand the difference between value and reference parameters.
- To understand how functions can be used to design programs.
- To be able to use functions to get data for programs.
- To be able to use functions to perform required tasks in programs.
- To be able to use functions for output.

Objectives

- To understand what is meant by global identifiers.
- To understand what is meant by local identifiers.
- To understand the scope of an identifier.
- To recognize the appropriate use of global and local identifiers.
- To be able to understand and control side effects in programs.
- To be able to use appropriate names for identifiers.
- To be able to define libraries.
- To distinguish between library header files and library implementation files.

Today's Lesson "The How"

- Reading a "Technical Text"
- •Determine the central ideas of the text, summarizing the complex concepts, processes, and/or information presented in the text by paraphrasing them in simpler but still accurate terms.
- •Determine the meaning of symbols, key terms, and C++ commands.

Today's Lesson "The What"

- User Defined Functions
- •Read Section 4.1 (Pages 127 –128) AND Section 4.2 (Pages 129 –141)

Today's Lesson "The How Part 2"

- Start with private thinking time.
- –We will use "Listen & Compare" Structured Discussion with your partner.
- –Groups will share (Explain to your partner)
- •What you learned including:
  - Structured Programming
  - Function Declarations
  - Function Implementations

Program Design

- Modular programming
  - Stepwise refinement of main tasks into subtasks.
  - Modules or subprograms that contain all definitions and declarations necessary to solve the subtask.
  - The lowest level of a C++ program is a function.
  - Abstract Data Type: higher level of design with a class of objects, defined properties and set of operations for processing the objects.

Program Design 2

- Modularity
  - The organization of a program into simple tasks that work as independent units
- Bottom Up Testing
  - Testing each module independently (or even writing your program one function at a time).
- Structured Programming
  - independent modules with flow of control & data

Program Design 3

- Structured Design
  - The process of designing the modules and specifying the flow of data between them.
  - Information is shared from parameter lists: formal and actual parameters (function arguments).
  - Organization of modules and flow of parameters is done through the main program (main function).

Advantages of Using Functions

1. To help make the program more understandable
2. To modularize the tasks of the program
   - building blocks of the program
3. Write a module once
   - those lines of source code are called multiple times in the program

Advantages of Using Functions 2

4. While working on one function, you can focus on just that part of the program
   - **construct it,**
   - **debug it,**
   - **perfect it.**
5. Different people can work on different functions simultaneously.
6. If a function is needed in more than one place in a program, or in different programs, you can write it once and use it many times

User Defined Functions

- User Defined Function
  - A function designed and defined by the programmer.

Page130.cpp

How to define a C++ Function?

- Generally speaking, we define a C++ function in two steps
  - Step #1 – declare the ***function declaration*** before the main function of the program
  - Step #2 – Implement the function after the main function

Function Declarations

- Function Declaration
  - Function name, number and type of arguments it expects and type of value it returns (if any).
  - General Form:
    - <return type> <Function_Name> (<arguments>);
    - function name is descriptive valid identifier
    - return type, declares the one data type returned by the function
    - Style: include comments regarding input & output.

Function Declaration Example

```
//Function: Sqrt
//Compute the square root of a double precision floating
    point number
//
//Input: a nonnegative double precision floating point
    number
//Output: a nonnegative double precision floating point
    number that represents the square root of the number
    received

double Sqrt (double x);
```

Function Order Within a Program

Comments

Preprocessor Directives

using namespace std;

Constant Declaration & Type Declaration

Function Declarations

```
int main( )
{
    variable declarations
    main program: function calls
}
```

Function Implementations

Structure of a C++ Function

- A C++ function consists of two parts
  - The function header, and
  - The function body
- The function header has the following syntax

  <return value> <name> (<parameter list>)
- The function body is simply a C++ code enclosed between { }

Function Implementations

- Function Implementation
  - a complete description of the function

```
double Sqr (double x)          //no semicolon here
{
   return x*x;
}
```

Function Headings

- Function heading
  - the first line of the function implementation containing the function's name, parameter declarations, and return type
  - looks like the function declaration minus the semicolon
  - function heading must match the function declaration in type and parameter list types and order.

Function Headings 2

- Formal Parameters
  - The arguments in the function heading.
- Actual Parameters
  - The arguments in the function call.
- The number of formal and actual parameters must match and should match in order and type.

Function Heading General Form

- General Form
  - <return type> <Function_Name> (<formal parameter list>)
  - function name is a valid identifier
    - use descriptive names
    - a value may need to be returned if a return type is declared
  - return type declares the data type of the function
  - the formal parameters are pairs of data types and identifier names separated by commas.

Void Functions

- Some functions need data to do their work, but return no values.
- Other functions need NO data AND return no values. These are called void functions (like a Procedure in Pascal).
- The function call appears on a line by itself and not as a part of a statement.
- Style: Use function names that include action verbs and use a capital letter to begin each word.

Void Function Example

```
void Display_Results (int result1, int result2, int result3)

//This is used to send your output to the screen
// The output stream.
{
   cout<<“The first result is: “<<result1<<endl;
   cout<<“The second result is: “<<result2<<endl;
   cout<<“The third result is: “<<result3<<endl;
}
```

Void Function without Parameters

```
void Display_Header( )
   //Takes no parameters and returns no values
   //The sole purpose is to display text on the output
     screen.
  //Perhaps a header, a logon greeting, or whose output.
{
   cout<<"Grades for Computer Science 110" << endl;
   cout<<endl;
   cout<<"Student Name                              Grade"<< endl;
   cout<<"----------------                          -------"<<endl;
}
```

Value Returning Functions

- Declare function type of value to be returned
  - double, int, char, etc.
- Only ONE value is returned
  - use return command as last line of function
  - don't use pass by reference parameters
  - don't use cout statements
  - like mathematical function, calculates 1 answer
  - like a function in Pascal
  - Style: Use Noun names that start w/ a capital letter

Function Example

```
//Function:  Cube
// Computes the cube of an integer
//
//Input: a number
//Output: a number representing the cube of the input number

double Cube (double x);

double Cube (double x)
{
   return x*x*x;
}
```

Function Declaration

Function Implementation

Function Example 2

```
// Function: compute price per square inch
// Compute the price per square inch of pizza
//
// Input:  cost and size of pizza
// Output: price per square inch
double Price_Per_Square_Inch(double cost, int size);

double Price_Per_Square_Inch(double cost, int size)
{
        double radius, area;

        radius = size / 2;
        area = PI * sqr(radius);
        return cost / area;
}
```

Function Declaration

Function Implementation

Stub Programming

- Stub Programming
  - the use of incomplete functions to test data transmission between them.
  - Contains declarations and rough implementations of each function.
  - Tests your program logic and values being passed to and from subprograms.

Roots1.cpp Roots2.cpp Roots.cpp

Main Driver

- Main Driver
  - another name for the main function when subprograms are used.
  - The driver can be modified to test functions in a sequential fashion.
  - The main driver is a “function factory”
    - add a function declaration, then a rough implementation
    - test the function stub by calling the function from the main driver

Driver Example

```
// Program file: driver.cpp P.137
#include <iostream>
using namespace std;

int main()
{
    int data;

    cout << "Enter an integer: ";
    cin >> data;
    return 0;
}
```

//Start with a simple program

Driver Example 2

```
// Program file: driver.cpp
//P.137-138
#include <iostream>
using namespace std;

int main()
{
        int data;

        cout << "Enter an integer: ";
        cin >> data;
        cout << "The square is " << Sqr(data) << endl;
        return 0;

}
```

Add a function declaration, rough implementation, and a function call to test one function.

Driver: Test and Fill in Details

- Once you have tested the function declaration, call, and implementation by sending and returning the same value, fill in the details of the function.
- In the driver.cpp example, this means replace:
  - return x with
  - return x*x
  - driver2.cpp

Formal and Actual Parameters

Area.cpp

- Formal parameters
  - the parameters listed in the function declaration and implementation
  - they indicate the "form" that the parameters will take, a data type and identifier paired together
- Actual parameters
  - the parameters in the function call which must match the formal parameter list in order & type
  - choose synonyms for names of actual & formal or same name

Value Parameters

- Value Parameters (Passed by Value)
  - values passed only from caller to a function
  - think of this as a one way trip
  - a copy of the actual parameters are made and sent to the function where they are known locally.
  - any changes made to the formal parameters will leave the actual parameters unchanged
  - at the end of the function, memory is deallocated for the formal parameters.

Area.cpp

Calling a function by value

- The function receives a copy of the actual parameter values.
- The function *cannot* change the values of the actual parameters.

Reference Parameters

- Reference Parameters (Pass by Reference)
  - when you want a function to return more than one value
    - Use a void function with reference parameters. This will be similar to a Pascal procedure.
  - Like a two way trip for more than one parameter
  - The value of the parameter can be changed by the subprogram.
  - No copy of the parameters are made

Reference Parameters 2

- Only the “address” of the actual parameter is sent to the subprogram.
- Format for formal reference parameters
  - <type name> **&**<formal parameter name>
  - void Get_Data(int **&**length, int **&**width);
- Alias: two or more identifiers in a program that refer to the same memory location
- Remember:
  - avoid reference parameters in value returning functions
  - don’t send constants to a function with reference parameters

Page147Swap.cpp

Calling a function by reference

- Very useful when we need a function which "returns more than one value".
- The formal parameter becomes an alias for the actual parameter.
- The function *can* change the values of the actual parameters.

Constant Reference Parameters

- Constant Reference
  - declare the formal parameter so that the actual parameter is passed by reference, but cannot change within the function
  - Format for Constant Reference Parameters
    - **const** <type name> **&**<parameter name>

Constant Reference Parameter Example

```
string Get_String(const string &prompt)
{
   string data;

   cout<<prompt;
   cin>>data;
   return data;
}
```

The "const" modifier

- C++ provides us with protection against accidentally changing the values of variables passed by reference with the ***const*** operator

  function declaration:
  *int FindMax(**const** int&number1, **const** int&number2);*

  function header:
  *int FindMax(**const** int& number1, **const** int& number2);*

Choosing a Parameter Method

- When the actual parameter must be changed, or you need a "two way" trip use a reference parameter **//Get_Data or Input Function &**
- When the value should NOT be changed (a "one way trip") and the data size is small, declare as a value parameter. **//Calculations**
- When the value should NOT be changed and the data size is large, use a constant reference parameter. **//Output Functions (const double & area)**

Putting it all together

- Sample Program
  - Comparing the cost of pizzas by square inch

Pizza.cpp

Local and Global Identifiers

- Global Identifiers
  - those that are declared before the main program
  - Good Style limits global identifiers to:
    - constants
    - type declarations
    - function declarations

Please use global constants!

- Local Identifiers
  - declared between { and } of main or subprogram

Don't use global variables!!!

Local Variables

- Local variables are declared inside the function body and exist as long as the function is running and destroyed when the function exit
- You have to initialize the local variable before using it
- If a function defines a local variable and there was a global variable with the same name, the function uses its local variable instead of using the global variable

Scope of Identifiers

- Scope of an identifier
  - the largest block of code where the identifier is available.

Function Overloading

- Function Overloading
  - When two or more functions have the same name.
  - Function Overloading can only be allowed when the function declarations differ
    - by the number of parameters they use,
    - by using different data types for at least one parameter,
    - or belong to different **classes** (more on this later)

Function Overloading Examples

- Function Declarations

```
double Triangle_Area (double base, double
   height);
//Area = ½ * Base * Height

double Triangle_Area (double side1, double
   side2,   double side3);
//Hero's Formula using three sides for the area
```