



Chapter 4: Subprograms

Functions for Problem Solving

Mr. Dave Clausen
La Cañada High School

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).

User Defined Functions

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

[Page130.cpp](#)

[Page130.txt](#)

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

Constant Declaration & Type Declaration

Function Declarations

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

Function Implementations

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.

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
```

Function Declaration

```
double Cube (double x);
```

```
double Cube (double x)
```

```
{
```

```
    return x*x*x;
```

```
}
```

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

Roots1.txt

Roots2.cpp

Roots2.txt

Roots.cpp

Roots.txt

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.h>
```

```
int main()
```

```
{
```

```
    int data;
```

```
    cout << "Enter an integer: ";
```

```
    cin >> data;
```

```
    return 0;
```

```
}
```

//Start with a simple program

Driver Example 2

// Program file: driver.cpp driver.txt

//P.137-138

```
#include <iostream.h>
```

```
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 driver2.txt

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;  
}
```

Formal and Actual Parameters

■ Formal parameters

[Area.cpp](#)

- 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

[Area.txt](#)

■ 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

Area.txt

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](#)

[P147Swap.txt](#)

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

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

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



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](#)

[Pizza.txt](#)

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!!!

Scope of Identifiers

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



Programmer Defined Libraries

- To create your own libraries:
 - Declare all functions including comments and save as #####lib.h or mylib.h
 - Implement all functions and save as #####lib.cpp or mylib.cpp

Library Header File Example

- Use `#ifndef <file name>`
 - asks if the file identifier has been defined
- if it has been defined, it skips to `#endif` statement (prevents multiple redefining)
- if it hasn't been defined, the preprocessor will see the `#define` directive after preprocessing the file

myinput.h

myinputh.txt

Library Implementation File Example

- Contains all function implementations complete with local variables and formal parameters
- #include “myinput.h” to include the header file with the function declarations
- Need a program to call my library
- Build a project to include all parts

myinput.cpp myinput.txt

Program Main Driver for my library

- #include “myinput.h”
 - use quotes “ “ , not angle brackets < >
- Call the functions from the Main Driver
- Don’t forget to make and build a project to link all the parts.
- In this case, don’t forget:
 - #include “apstring.h”
 - apstring.cpp

[testmyin.cpp](#)

[testmyin.ide](#)

[testmyin.txt](#)

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 =  $\frac{1}{2}$  * Base * Height
```

```
double Triangle_Area (double side1, double  
side2, double side3);
```

```
//Hero's Formula using three sides for the area
```