

ELEG 1043

Computer Applications in Engineering





Chapter 6: Modularity Using Functions

C++ FOR ENGINEERS
AND SCIENTISTS

Acknowledgement

- Some of the slides or images are from various sources. The copyright of those materials belongs to their original owners.

Objectives

- In this chapter, you will learn about:
 - Function and parameter declarations
 - Returning a single value
 - Returning multiple values
 - Rectangular to polar coordinate conversion
 - Variable scope
 - Variable storage categories
 - Common programming errors

Function and Parameter Declarations

- Interaction with a function includes:
 - **Passing data (Arguments)** to a function correctly when its called
 - **Returning values** from a function when it ceases operation
- A function is called by (1) **giving the function's name** and (2) **passing arguments in the parentheses** following the **function name**

function-name (*data passed to function*);
└──────────┘ └────────────────────────┘
This identifies the This passes data
called function to the function

Figure 6.1 Calling and passing data to a function

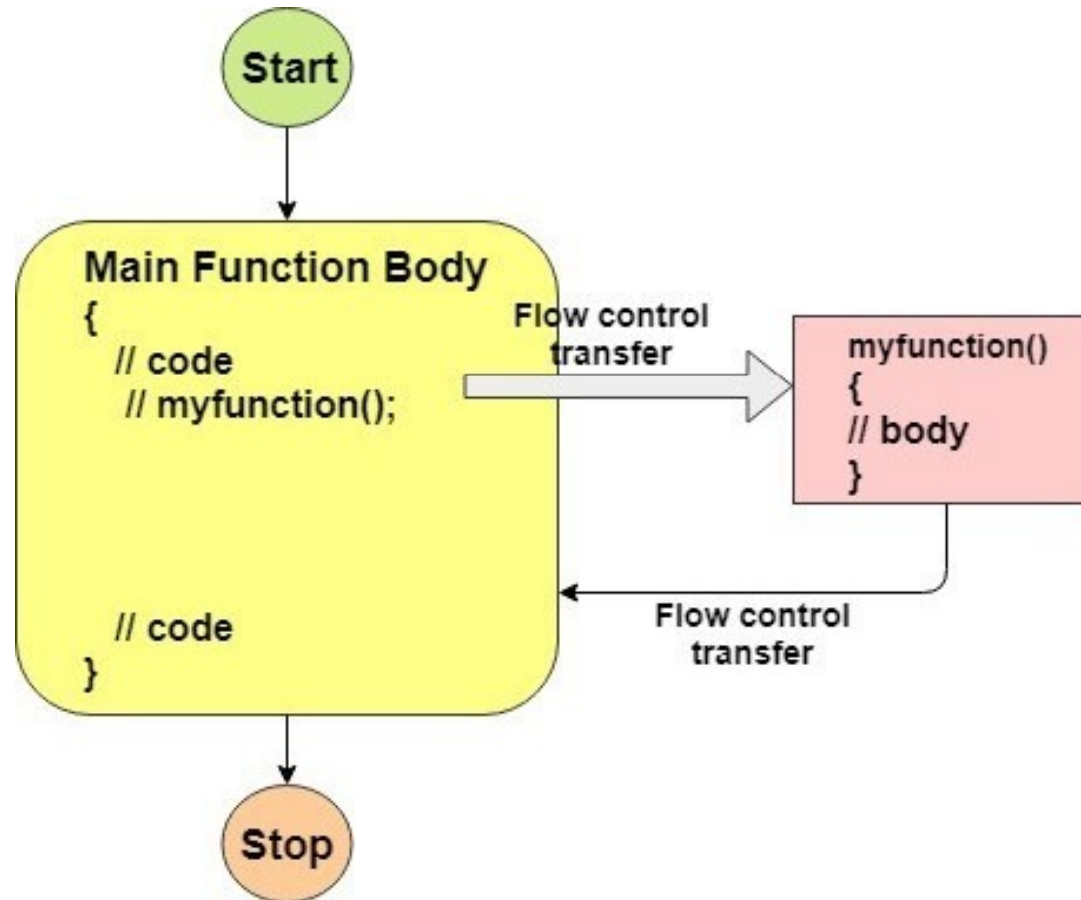
Function and Parameter Declarations (continued)

- Before a function is called, it must be declared to function that will do calling
- Declaration statement for a function is referred to as function prototype
- Function prototype tells calling function:
 - Type of value that will be formally returned, if any
 - Data type and order of the values (arguments) the calling function should transmit to the called function
- Function prototypes can be placed with the variable declaration statements above the calling function name or in a separate header file

Calling a Function

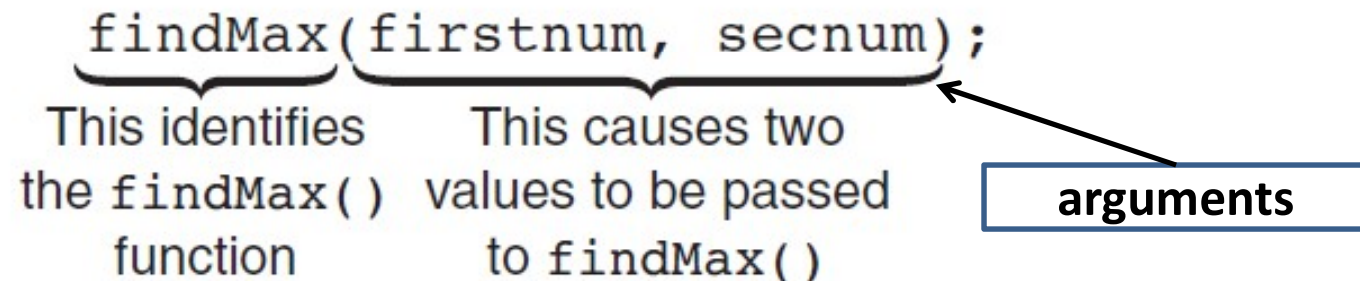
- Requirements when calling a function include:
 - Using the **name** of the function
 - Enclosing **any data passed (Arguments)** to the function in the parentheses following the function name, using the **same order and type** declared in the function prototype

Calling a Function



Calling a Function (continued)

- The items enclosed in the parentheses are called **arguments** of the called function



The diagram shows the function call `findMax(firstnum, secnum);`. A bracket under `findMax` is annotated with "This identifies the `findMax()` function". A bracket under `(firstnum, secnum)` is annotated with "This causes two values to be passed to `findMax()`". An arrow points from a box labeled "arguments" to the opening parenthesis of the argument list.

```
findMax(firstnum, secnum);
```

This identifies the `findMax()` function This causes two values to be passed to `findMax()`

arguments

Figure 6.2 Calling and passing two values to `findMax()`

Defining a Function

- Every C++ function consists of two parts:
 - **Function header**
 - **Function body**
- Function header's purpose:
 - **Identify data type** of value function returns, **provide** function with **name**, and **specify** number, order, and type of **arguments** function expects
- Function body's purpose:
 - To operate on passed data and return, **at most**, one value directly back to the calling function

Functions with Empty Parameter Lists

- Although useful functions having an empty parameter list are extremely limited, **they can occur**
- Function prototype for such a function requires writing the keyword **void** or **nothing** at all between the parentheses following the function's name
- Examples:

```
int display();  
int display(void);
```

Reusing Function Names (Overloading)

- C++ provides the capability of using the **same function name for more than one function**
 - Referred to as **function overloading**
- **Only requirement** for creating more than one function with same name:
 - Compiler must be able to determine which function to use based on **the parameters' data types** (not the data type of the return value, if any)
 - Which of the functions is called **depends on the argument type** supplied at the time of the call

Function Templates

- Function template: Single complete function that serves **as a model for a family of functions**
 - Function from the family that is actually created depends on the specific function call
- Generalize the writing of functions that perform essentially **the same operation**, but on **different parameter data types**
- Make it possible to write a general function that handles all cases but where the compiler can set parameters, variables, and even return type based on the actual function call

Returning a Single Value

- Function receiving passed by value arguments can process the values sent to it in any fashion and return one, and **only one**, “legitimate” value directly to the calling function

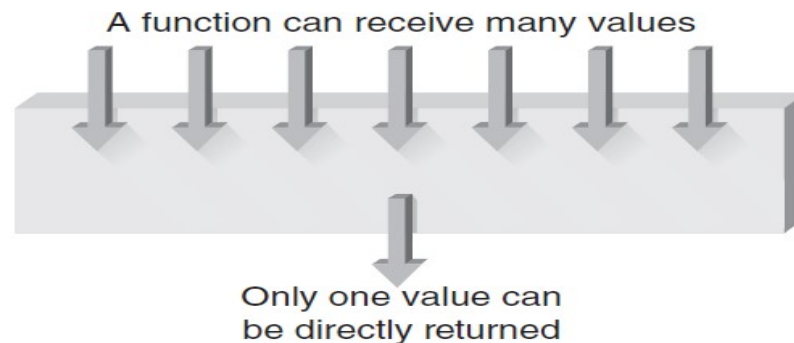


Figure 6.10 A function directly returns at most one value

Returning a Single Value

```
int sum(int num1, int num2, int num3)
{
    int sum = num1 + num2 + num3;
    return sum;
}
```

Returning Multiple Values

- In a typical function invocation, called function receives values from its calling function, stores and manipulates the passed values, and directly returns at most one value
 - **Pass by value:** When data is passed in this manner
 - **Pass by reference:** Giving a called function direct access to its calling function's variables is referred to as
 - The called function can **reference, or access**, the variable **whose address has been passed** as a pass by reference argument

Passing and Using Reference Parameters

- From the sending side, calling a function and **passing an address as an argument** is the same as calling a function and passing a value
- Whether a value or an address is actually passed **depends on** the problem

Passing and Using Reference Parameters (Example)

```
#include <iostream>
using namespace std;

void swapnum(int &i, int &j) {
    int temp = i;
    i = j;
    j = temp;
}

int main(void) {
    int a = 10;
    int b = 20;
    cout<<"A is "<<a<<" and B is "<<b<<endl;

    swapnum(a, b);
    cout<<"After swapping two numbers"<<endl;
    cout<<"A is "<<a<<" and B is "<<b<<endl;
    return 0;
}
```

A Case Study: Calculate the Circle Area

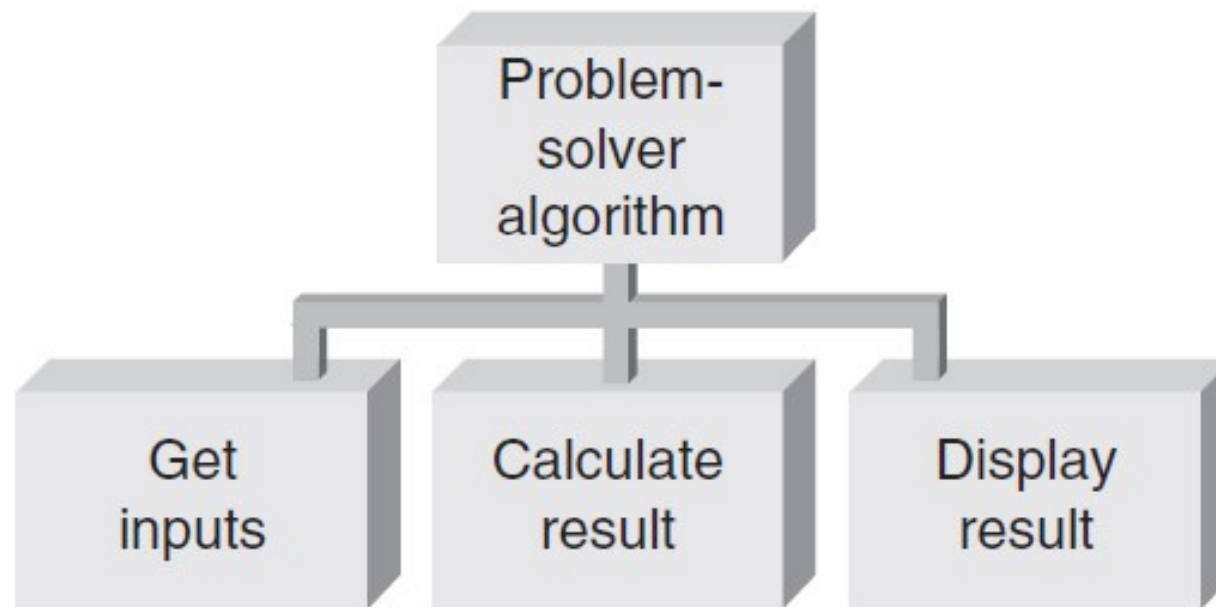
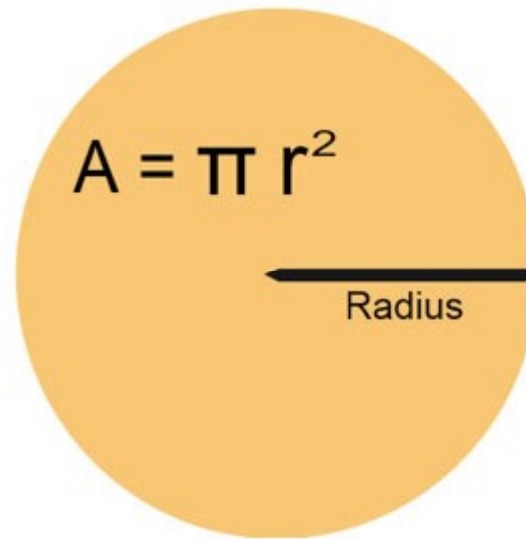


Figure 6.14 The problem-solver algorithm

A Case Study: Calculate the Circle Area (cont'd)



Variable Scope

- A function can be thought of as a closed box, with slots **at the top** to receive values and a single slot at the bottom to return a value

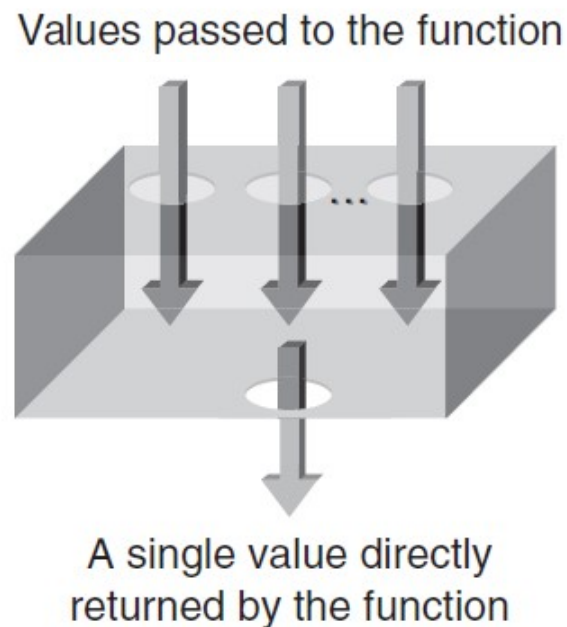


Figure 6.19 A function can be considered a closed box

Variable Scope (continued)

- **Local variables:** Variables created in a function that are conventionally **available only to the function**
- **Scope:** Section of the program **where the identifier is valid or “known”**
- A variable with **local scope** is simply one with storage locations set aside for it by a declaration statement inside the function that declared them
- A variable with **global scope** has storage created for it by a declaration statement located outside any function

Variable Scope (continued)

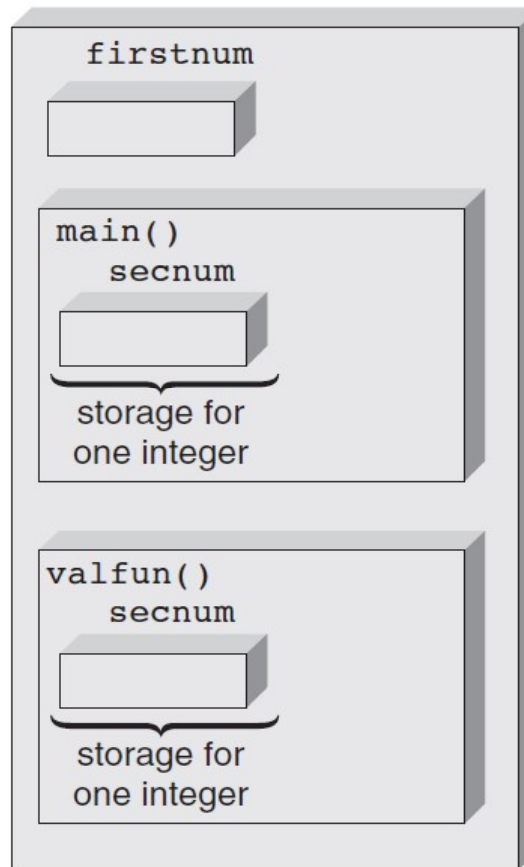


Figure 6.20 The three storage areas reserved by Program 6.15

Example: Variable Scope



Program 6.16

```
#include <iostream>
using namespace std;

double number = 42.8;          // a global variable named number

int main()
{
    double number = 26.4;      // a local variable named number

    cout << "The value of number is " << number << endl;

    return 0;
}
```


Common Programming Errors

- Passing **incorrect data types**
- **Omitting the called function's prototype** before or within the calling function
- Terminating a function header with a **semicolon**
- **Forgetting to include the data type** of a function's parameters in the function header

Summary

- A function is **called by giving its name and passing any data** to it in the parentheses following the name
- A function's return type is the data type of the value the function returns
- Arguments passed to a function when it is called must conform to the parameters specified by the function header
- Functions can be declared to all calling functions by means of **a function prototype**