



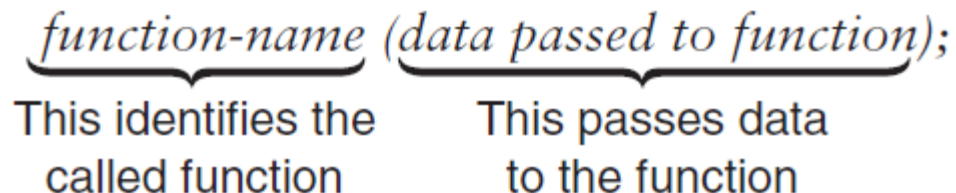
Chapter 6: Modularity Using Functions

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 to a function correctly when its called
 - Returning values from a function when it ceases operation
- A function is called by giving the function's name and passing arguments in the parentheses following the function name



function-name (*data passed to function*);

This identifies the
called function

This passes data
to the function

The diagram illustrates the syntax of a function call. It shows the text *function-name* (*data passed to function*); with two horizontal curly braces underneath. The first brace is under *function-name* and has the text 'This identifies the called function' below it. The second brace is under *data passed to function* and has the text 'This passes data to the function' below it.

Figure 6.1 Calling and passing data to a function

Function and Parameter Declarations

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

void findMax(int, int); // function declaration (prototype)

int main()
{
    int firstnum, secnum;

    cout << "\nEnter a number: ";
    cin  >> firstnum;
    cout << "Great! Please enter a second number: ";
    cin  >> secnum;

    findMax(firstnum, secnum); // function is called here

    return 0;
}
```

Refer to page 307 for
more explanations and
examples

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 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

Function and Parameter Declarations (continued)

The general form of function prototype statements is as follows:

```
returnDataType functionName(list of argument data types);
```

The *returnDataType* refers to the type of value the function returns. Here are some examples of function prototypes:

```
int fmax(int, int);  
double swap(int, char, char, double);  
void display(double, double);
```

Calling a Function

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

Calling a Function (continued)

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

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

This identifies the `findMax()` function

This causes two values to be passed to `findMax()`

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

Calling a Function (continued)

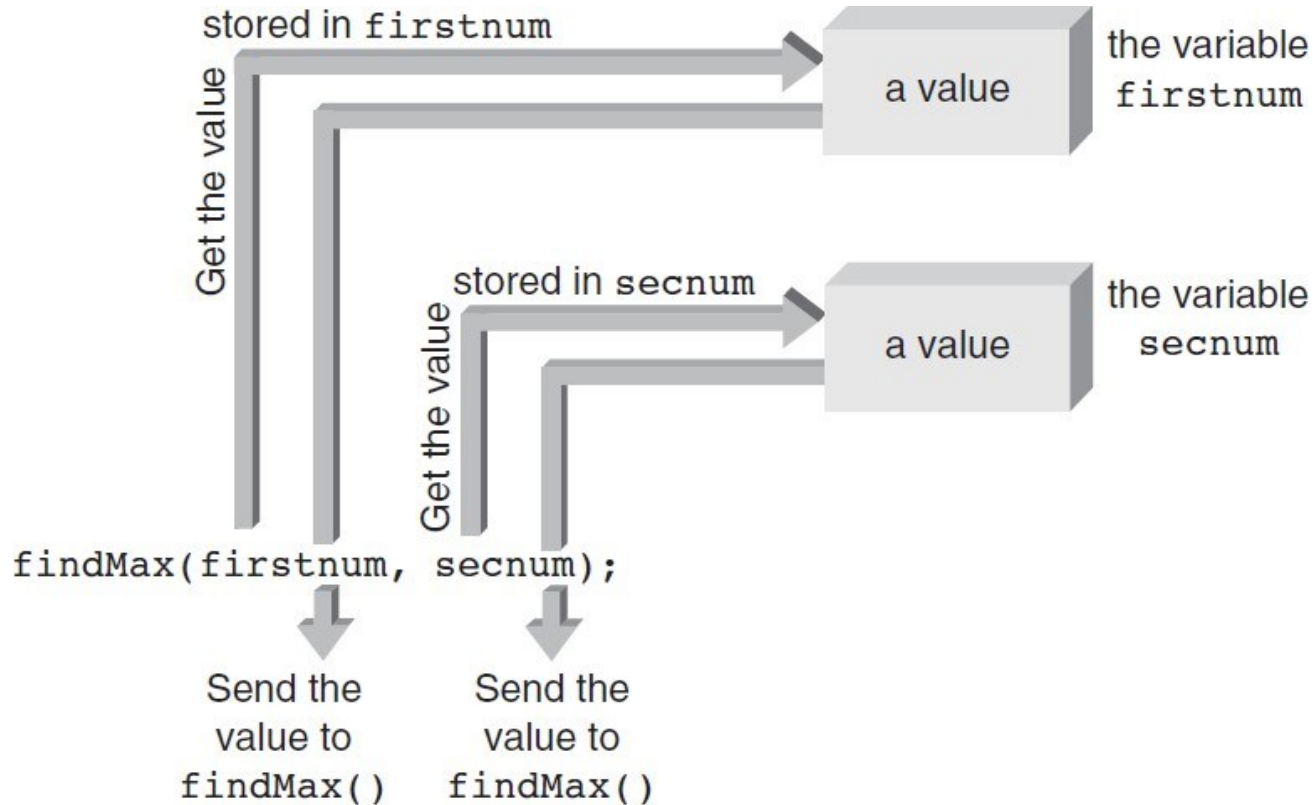


Figure 6.3 The `findMax()` function receives actual values

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

Defining a Function

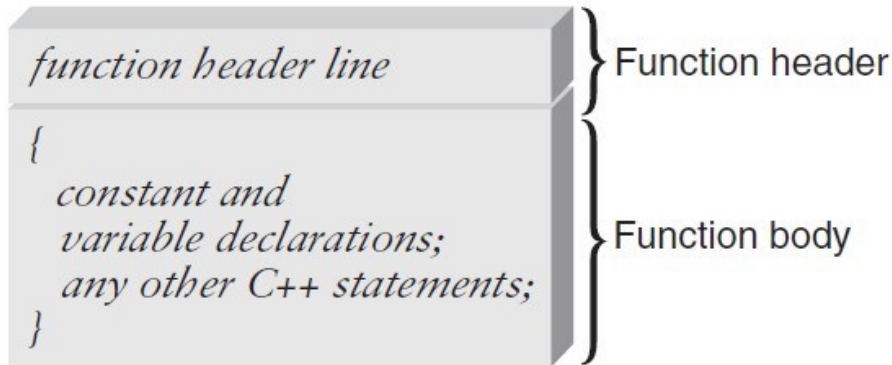


Figure 6.4 The general format of a function

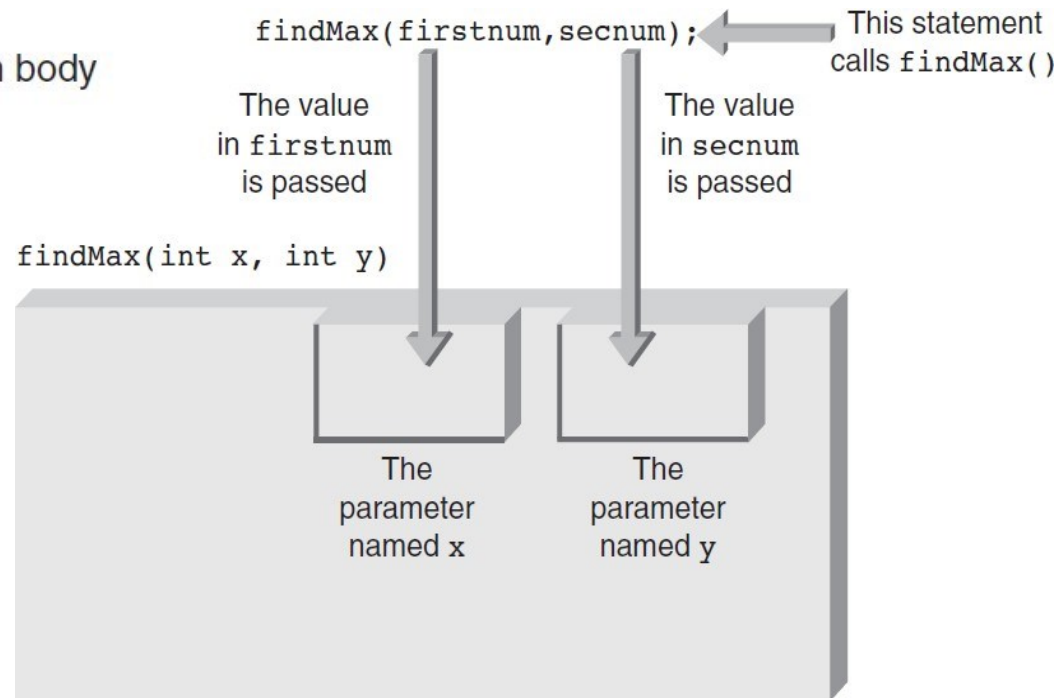


Figure 6.5 Storing values in parameters

Defining a Function

The following is a commonly used syntax for a function definition:

```
returnDataType functionName(parameter list)
{
    // constant declarations in here
    // variable declarations in here

    // other C++ statements in here

    return value;
}
```

Refer to page 313 for more explanations and examples

A function prototype declares a function. The syntax for a function prototype, which provides the function's return data type, the function's name, and the function's parameter list, is as follows:

```
returnDataType functionName(list of parameter data types);
```

Placement of Statements

- General rule for placing statements in a C++ program:
 - All preprocessor directives, named constants, variables, and functions must be declared or defined before they can be used
 - Although this rule permits placing both preprocessor directives and declaration statements throughout the program, doing so results in poor program structure

Placement of Statements

As a matter of good programming form, the following statement ordering should form the basic structure around which all C++ programs are constructed:

```
// preprocessor directives
// function prototypes

int main()
{
    // symbolic constants
    // variable declarations

    // other executable statements

    // return statement
}

// function definitions
```

Refer to page 315 for
more explanations and
examples

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);
```

Default Arguments

- C++ provides **default arguments** in a function call for added flexibility
 - Primary use: to extend parameter list of existing functions without requiring any change in calling parameter lists already used in a program
 - Listed in the function prototype and transmitted automatically to the called function when the corresponding arguments are omitted from the function call
 - Example: Function prototype with default arguments

```
void example(int, int = 5, double = 6.78)
```


Default Arguments (continued)

provides default values for the last two arguments. If any of these arguments are omitted when the function is actually called, the C++ compiler supplies the default values.

Therefore, all the following function calls are valid:

```
example(7, 2, 9.3)    // no defaults used  
example(7, 2)         // same as example(7, 2, 6.78)  
example(7)            // same as example(7, 5, 6.78)
```

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

Reusing Function Names (Overloading)

```
void cdabs(int x)  // compute and display absolute value of an integer
{
    if (x < 0)
        x = -x;
    cout << "The absolute value of the integer is " << x << endl;
}
```

```
void cdabs(float x)  // compute and display absolute value of a float
{
    if (x < 0)
        x = -x;
    cout << "The absolute value of the float is " << x << endl;
}
```

```
void cdabs(double x)  // compute and display absolute value of a double
{
    if (x < 0)
        x = -x;
    cout << "The absolute value of the double is " << x << endl;
}
```

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

Refer to pages 319-323
for more explanations
and examples

Returning a Single Value

- Function receiving an argument passed by value cannot inadvertently alter value stored in the variable used for the argument
- 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

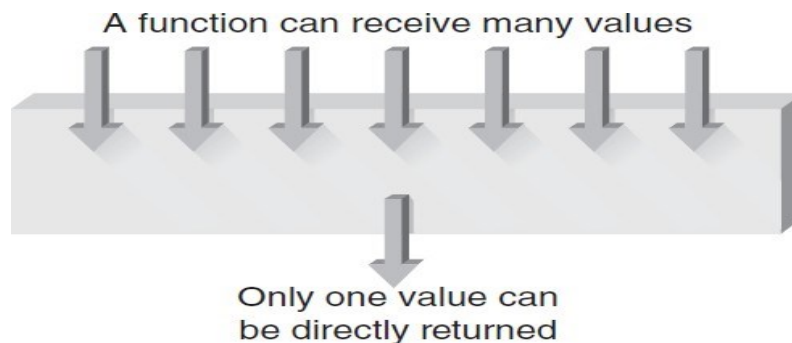


Figure 6.10 A function directly returns at most one value

Returning a Single Value

For example, the findMax() function written previously determines the maximum value of two numbers passed to it. For convenience, the findMax() code is listed again:

```
void findMax(int x, int y)
{
    // start of function body
    int maxnum;    // variable declaration

    if (x >= y)    // find the maximum number
        maxnum = x;
    else
        maxnum = y;

    cout << "\nThe maximum of the two numbers is "
         << maxnum << endl;

    return;
} // end of function body and end of function
```

Returning a Single Value

In this function header, x and y are the names chosen for the function's parameters:

```
void findMax(int x, int y)
```

If findMax() is to return a value, its header line must be amended to include the data type of the value being returned. For example, if an integer value is to be returned, this is the correct function header:

```
int findMax(int x, int y)
```

Similarly, if the function is to receive two single-precision parameters and return a single-precision value, this is the correct function header:

```
float findMax(float x, float y)
```

Returning a Single Value

Having declared the data type that findMax() will return, all that remains is including a statement in the function to cause the return of the correct value. To return a value, a function must use a return statement, which has this form:

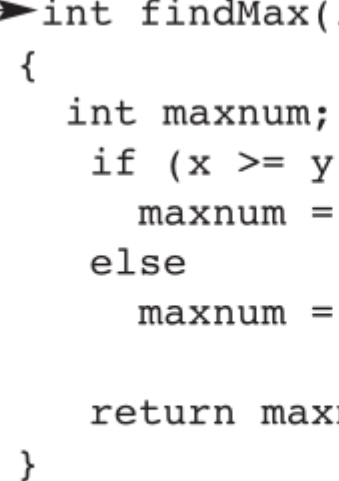
```
return expression;
```


Returning a Single Value

These should
be the same
data type

```
int findMax(int x, int y) // function header
{                          // start of function body
    int maxnum;           // variable declaration
    if (x >= y)
        maxnum = x;
    else
        maxnum = y;

    return maxnum;        // return statement
}
```



Refer to page 331 for
more explanations and
examples

Inline Functions

- Calling a function places a certain amount of overhead on a computer
 - Placing argument values in a reserved memory region (called the **stack**) that the function has access to
 - Passing control to the function
 - Providing a reserved memory location for any returned value (again, using the stack for this purpose)
 - Returning to the correct point in the calling program

Inline Functions (continued)

- Paying overhead associated with calling a function is justified when a function is called many times
 - Can reduce a program's size substantially
- For small functions that are not called many times, overhead of passing and returning values might not be warranted
- **Inline functions:**
 - Group repeating lines of code together under a common function name
 - Have the compiler place this code in the program wherever the function is called

Refer to page 334 for more explanations and examples

Inline Functions (continued)

- Advantage: Increase in execution speed •优势:提高执行速度
 - Because the inline function is expanded and included in every expression or statement calling it, no execution time is lost because of the call and return overhead a non-inline function requires
- Each time an inline function is referenced the complete code is reproduced and stored as an integral part of the program
- A non-inline function is stored in memory only once
- Inline functions should be used only for small functions that aren't called extensively in a program

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 that's accepted as a reference parameter is the same as calling a function and passing a value
- Whether a value or an address is actually passed depends on the parameter types declared

Passing and Using Reference Parameters

Program 6.8

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

void newval(double&, double&); // prototype with two reference parameters

int main()
{
    double firstnum, secnum;

    cout << "Enter two numbers: ";
    cin >> firstnum >> secnum;
    cout << "\nThe value in firstnum is: " << firstnum << endl;
    cout << "The value in secnum is: " << secnum << "\n\n";
    newval(firstnum, secnum); // call the function
    cout << "The value in firstnum is now: " << firstnum << endl;
    cout << "The value in secnum is now: " << secnum << endl;

    return 0;
}
```

Passing and Using Reference Parameters

Program 6.8

```
void newval(double& xnum, double& ynum)
{
    cout << "The value in xnum is: " << xnum << endl;
    cout << "The value in ynum is: " << ynum << "\n\n";
    xnum = 89.5;
    ynum = 99.5;

    return;
}
```


Passing and Using Reference Parameters (continued)

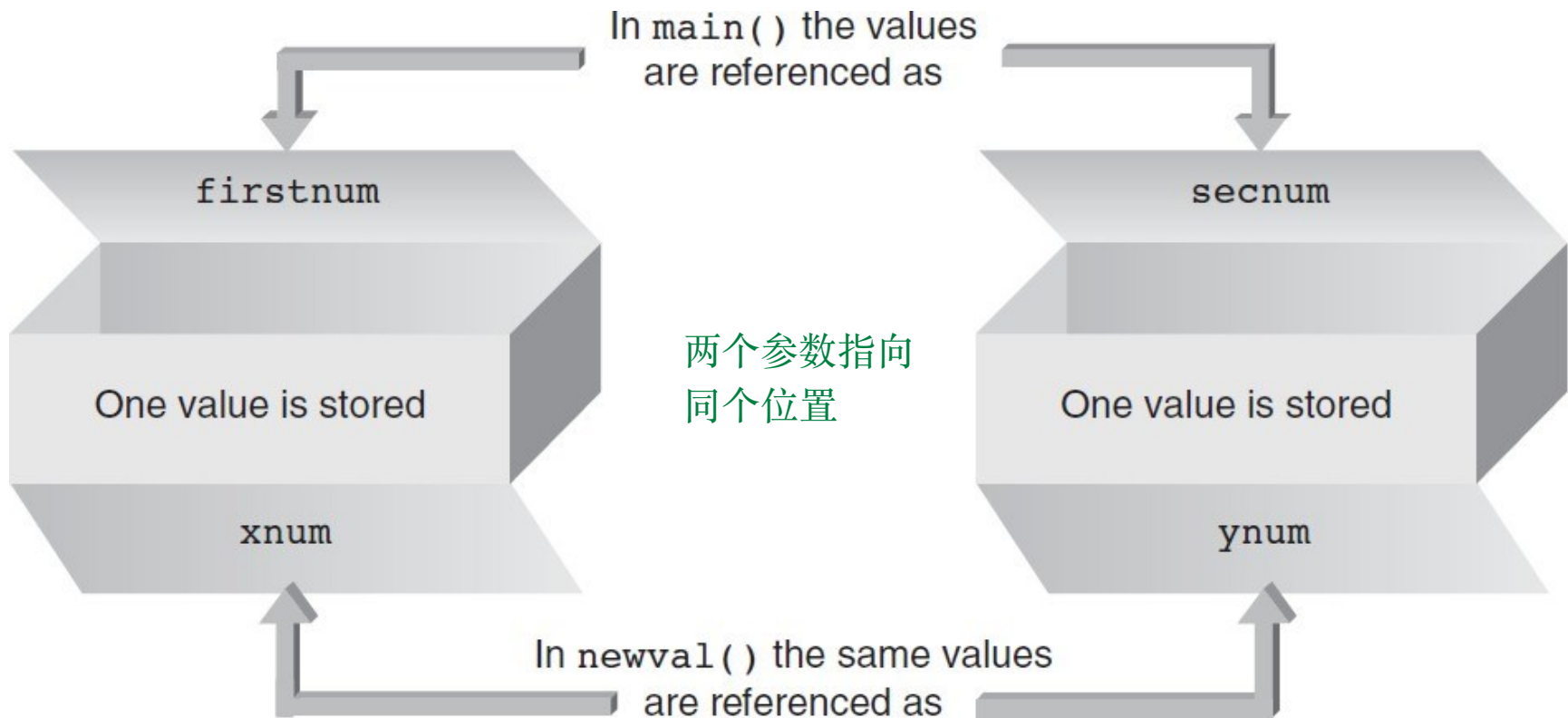


Figure 6.11 The equivalence of arguments and parameters in Program 6.8

Passing and Using Reference Parameters

The following is a sample run of Program 6.8:

```
Enter two numbers: 22.5 33.0
```

```
The value in firstnum is: 22.5
```

```
The value in secnum is: 33
```

```
The value in xnum is: 22.5
```

```
The value in ynum is: 33
```

```
The value in firstnum is now: 89.5
```

```
The value in secnum is now: 99.5
```

不用写星号指针*符号



Passing and Using Reference Parameters

Program 6.9

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

void calc(double, double, double, double&, double&); // prototype

int main()
{
    double firstnum, secnum, thirdnum, sum, product;

    cout << "Enter three numbers: ";
    cin >> firstnum >> secnum >> thirdnum;
    calc(firstnum, secnum, thirdnum, sum, product); // function call
    cout << "\nThe sum of the numbers is: " << sum << endl;
    cout << "The product of the numbers is: " << product << endl;

    return 0;
}
```

Passing and Using Reference Parameters

Program 6.9

```
void calc(double num1, double num2, double num3, double& total, double& product)
{
    total = num1 + num2 + num3;
    product = num1 * num2 * num3;
    return;
}
```

Passing and Using Reference Parameters (continued)

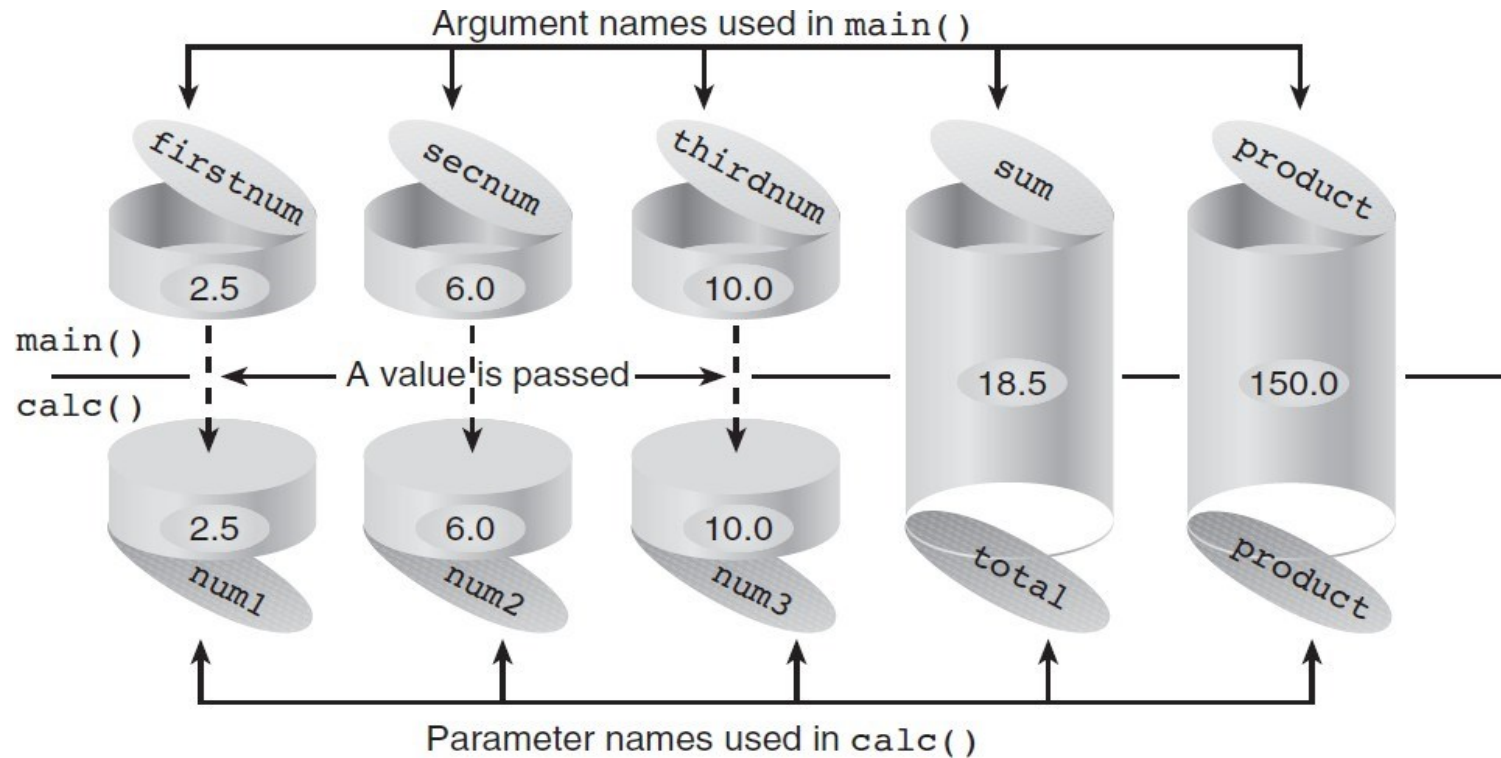


Figure 6.12 The relationship between argument and parameter names

Passing and Using Reference Parameters

After `calc()` is called, it uses its first three parameters to calculate values for total and product and then returns control to `main()`. Because of the order of its actual calling arguments, `main()` knows the values calculated by `calc()` as sum and product, which are then displayed. Following is a sample run of Program 6.9:

```
Enter three numbers: 2.5 6.0 10.0
```

```
The sum of the entered numbers is: 18.5
```

```
The product of the entered numbers is: 150
```

Refer to pages 344,346
for more explanations
and examples

A Case Study: Rectangular to Polar Coordinate Conversion

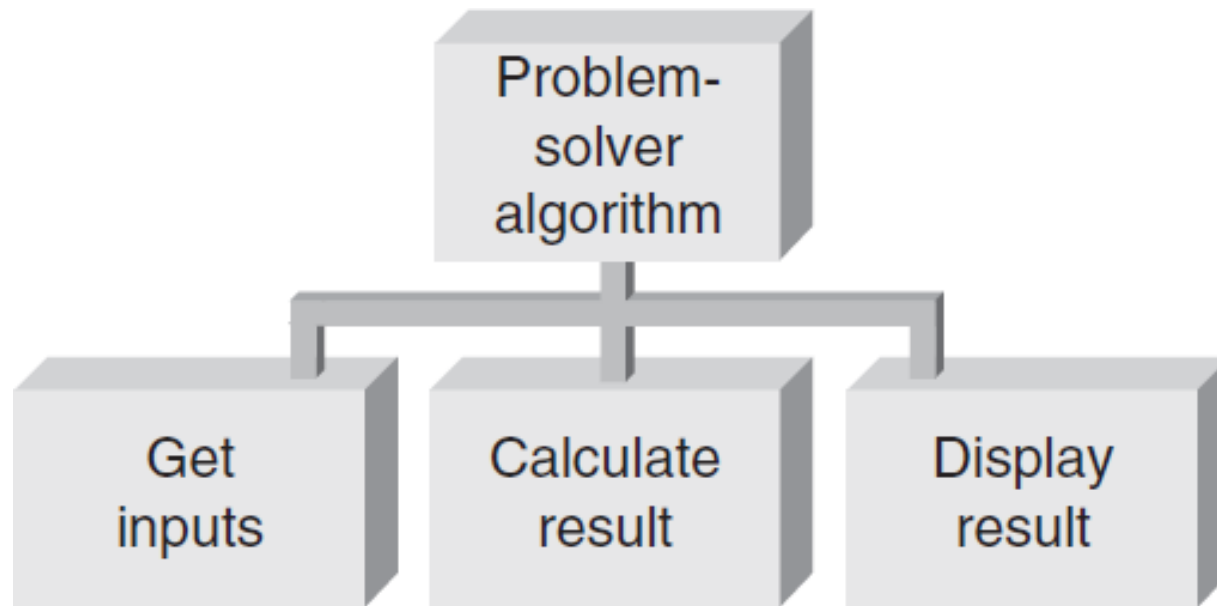
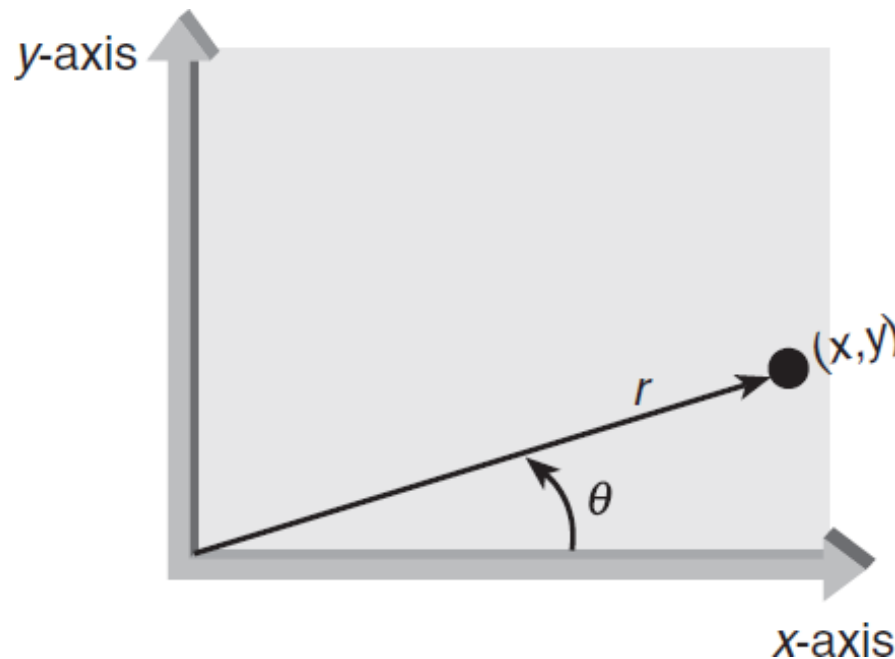


Figure 6.14 The problem-solver algorithm

A Case Study: Rectangular to Polar Coordinate Conversion (cont'd)



Refer to page 349 for more explanations and examples

Figure 6.15 The correspondence between polar (distance and angle) and Cartesian (x and y) coordinates

A Case Study: Rectangular to Polar Coordinate Conversion (cont'd)

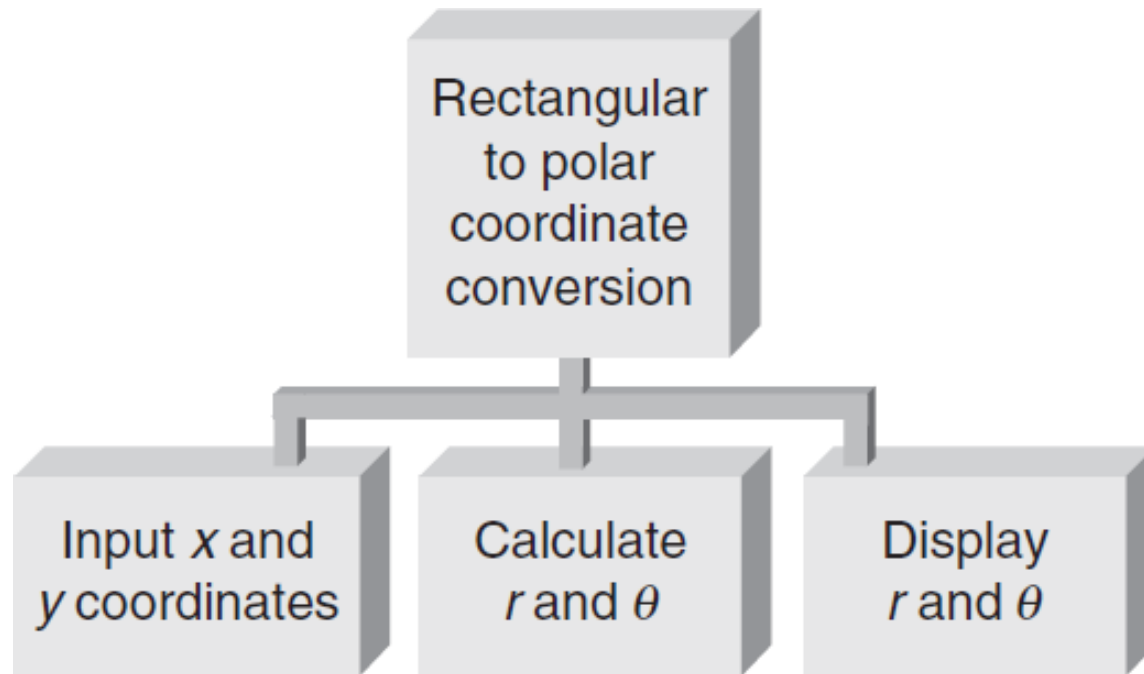


Figure 6.16 A top level structure diagram

A Case Study: Rectangular to Polar Coordinate Conversion (cont'd)

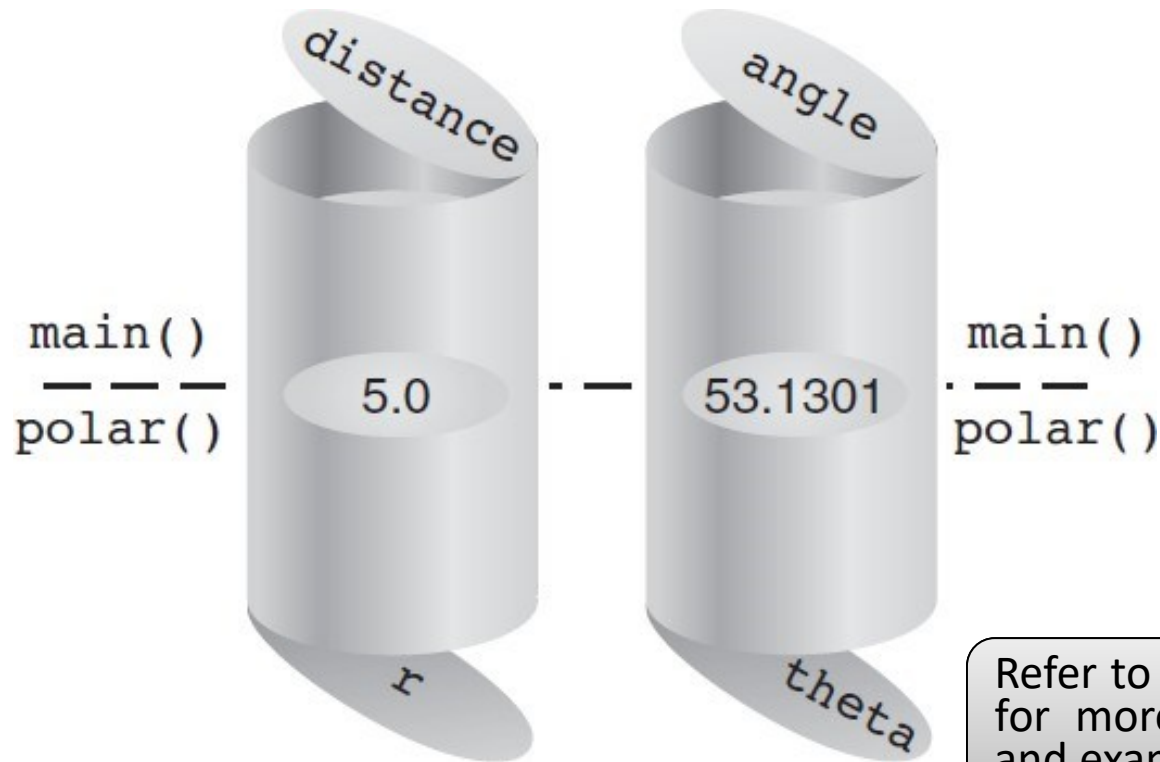
```
void polar(double x, double y, double& r, double& theta)
{
    const double TODGREES = 180.0/3.141593;
    r = sqrt(x * x + y * y);
    theta = atan(y/x) * TODGREES;

    return;
}
```

&用来返回数值 r && theta

形式上没有返回，实际上有&的

A Case Study: Rectangular to Polar Coordinate Conversion (cont'd)



Refer to pages 352-356
for more explanations
and examples

Figure 6.17 Parameter values when `polar()` is called

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

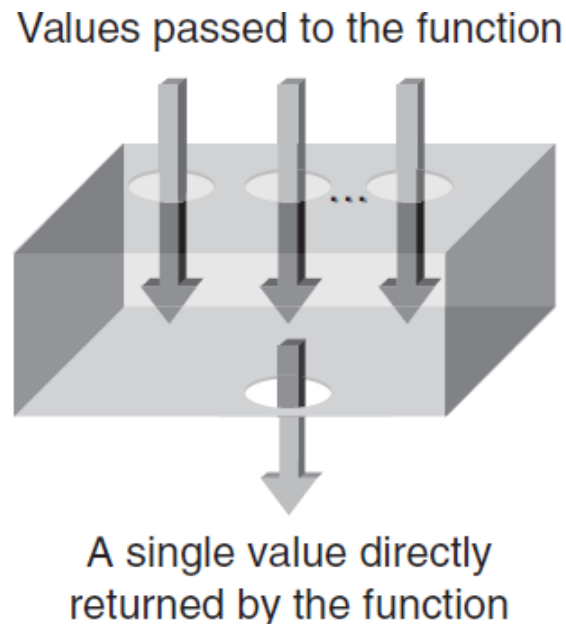


Figure 6.19 A function can be considered a closed box

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)

```
int firstnum;      // create a global variable named firstnum

void valfun();     // function prototype (declaration)

int main()
{
    int secnum;     // create a local variable named secnum

    firstnum = 10;   // store a value in the global variable
    secnum = 20;     // store a value in the local variable
}
```

Program 6.15

Variable Scope (continued)

```
valfun();           // call the function valfun

cout << "\nFrom main() again: firstnum = " << firstnum << endl;
cout << "From main() again: secnum = " << secnum << endl;
return 0;
}
```

Program 6.15

Variable Scope (continued)

```
void valfun()    // no values are passed to this function
{
    int secnum;    // create a second local variable named secnum

    secnum = 30;    // affects only this local variable's value

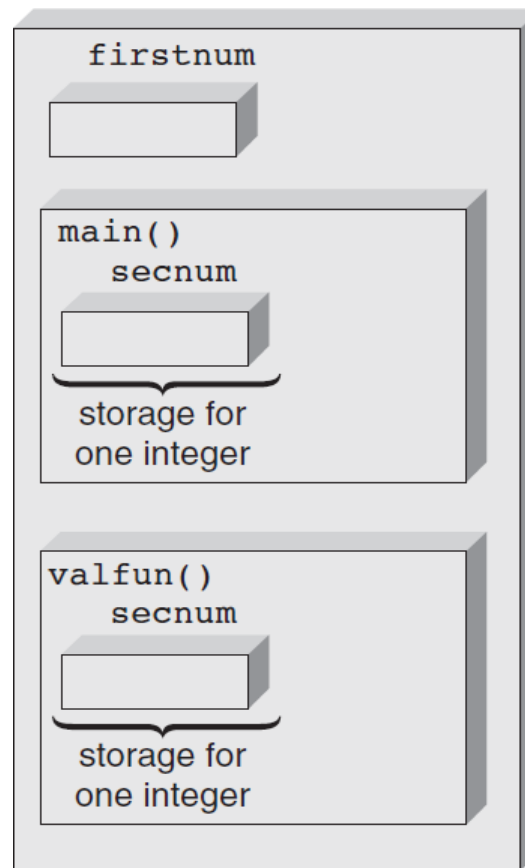
    cout << "\nFrom valfun(): firstnum = " << firstnum << endl;
    cout << "From valfun(): secnum = " << secnum << endl;

    firstnum = 40;    // changes firstnum for both functions

    return;
}
```

Program 6.15

Variable Scope (continued)



Refer to page 384 for more explanations and examples

Figure 6.20 The three storage areas reserved by Program 6.15

Variable Scope (continued)

```
From main(): firstnum = 10
```

```
From main(): secnum = 20
```

```
From valfun(): firstnum = 10
```

```
From valfun(): secnum = 30
```

```
From main() again: firstnum = 40
```

```
From main() again: secnum = 20
```

Variable Scope (continued)

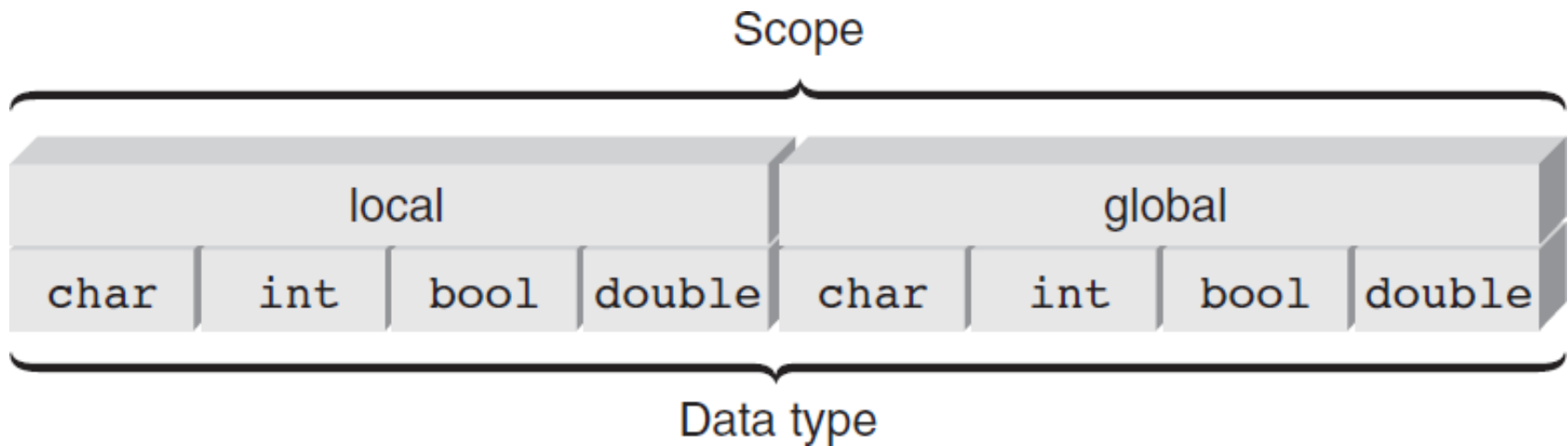


Figure 6.21 Relating the scope and type of a variable

Refer to page 363 for more explanations and examples

Scope Resolution Operator

- When a local variable has the same name as a global variable, all references to the variable name made within the local variable's scope refer to the local variable
- See Program 6.16

Scope Resolution Operator (continued)

- To reference a global variable when a local variable of the same name is in scope, use C++'s scope resolution operator, which is ::
- See Program 6.16.a

Misuse of Globals

- Global variables allow programmers to “jump around” the normal safeguards provided by functions
- Instead of passing variables to a function, it is possible to make all variables global: *do not do this*
 - Indiscriminate use of global variables destroys the safeguards C++ provides to make functions independent and insulated from each other
 - Using only global variables can be especially disastrous in large programs with many user-created functions

Variable Storage Categories

- A variable's scope can be thought of as the space in the program where the variable is valid
- In addition to space dimension represented by scope, variables have a time dimension that refers to the length of time storage locations are reserved for a variable
- This time, dimension is referred to as the variable's **lifetime**
- When and how long a variable's storage locations are kept before they are released can be determined by the variable's **storage category**

Variable Storage Categories (continued)

- The four available storage categories are:
 - auto
 - static
 - extern
 - register

```
auto int num;          // auto storage category and int data type
static int miles;      // static storage category and int data type
register int dist;     // register storage category and int data type
extern int volts;      // extern storage category and int data type
auto float coupon;     // auto storage category and float data type
static double yrs;     // static storage category and double data type
extern float yld;      // extern storage category and float data type
auto char inKey;       // auto storage category and char variable type
```


Local Variable Storage Categories

- Local variables can be members only of the `auto`, `static`, or `register` storage categories
- Storage for automatic local variables is reserved or created automatically each time a function
 - As long as the function hasn't returned control to its calling function, all automatic variables local to the function are “alive”
- A local `static` variable isn't created and destroyed each time the function declaring it is called
 - Local `static` variables remain in existence for the program's lifetime

Refer to pages 370,371
for more explanations
and examples

Local Variable Storage Categories (continued)

- Most computers have a few high-speed storage areas, called **registers**, located in the CPU that can also be used for variable storage
 - Because registers are located in the CPU, they can be accessed faster than normal memory storage areas located in the computer's memory unit

Global Variable Storage Categories

- Global variables are created by definition statements external to a function
- By their nature, global variables do not come and go with the calling of a function
- After a global variable is created, it exists until the program in which it's declared has finished executing
- Global variables can be declared with the `static` or `extern` storage category, but not both

Refer to pages 372-374
for more explanations
and examples

Common Programming Errors

- Passing incorrect data types
- Errors that occur when the same variable is declared locally in both the calling and the called functions
- 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
- Every variable has a storage category, which determines how long the variable's value is retained

Homework

1. P232, exercise 14
2. P303, exercise 10
3. P381, exercise 10