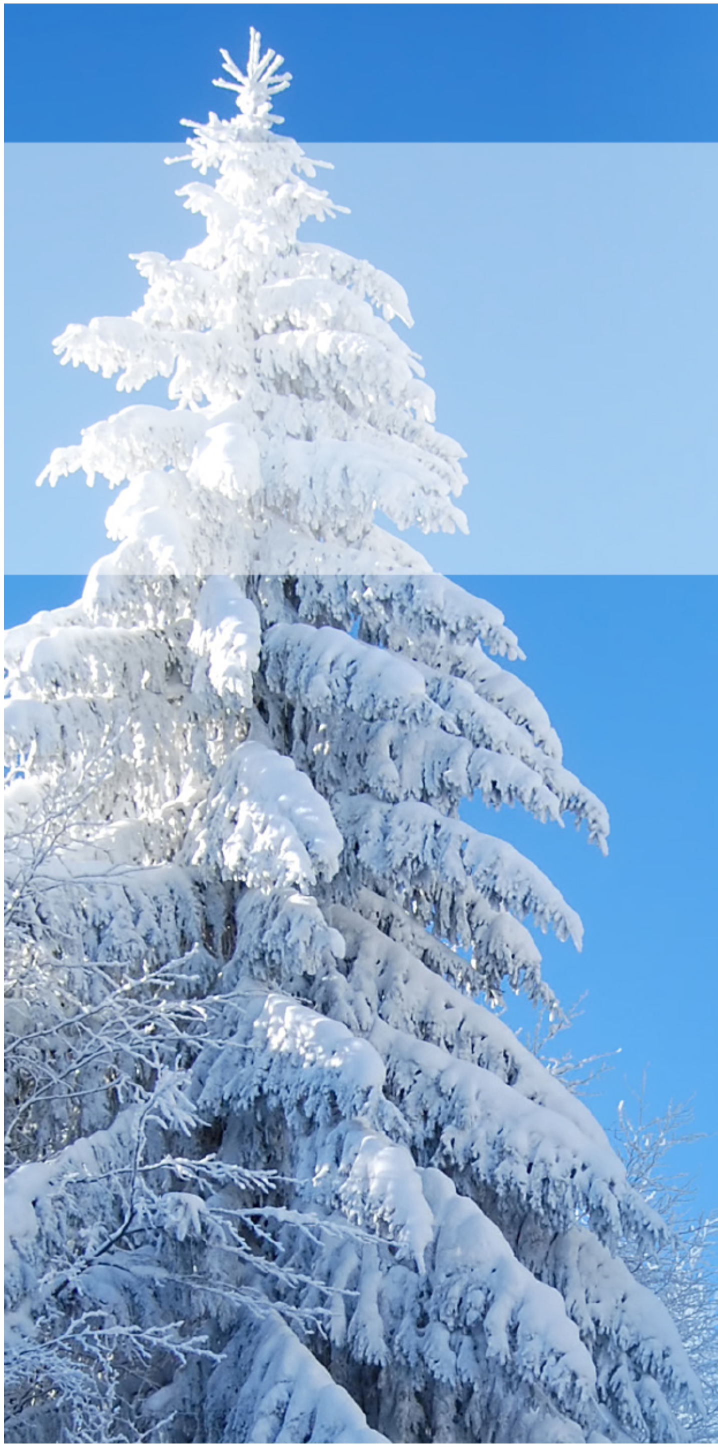
COMPREHENSIVE EDITION



# PROGRAMMING AND PROBLEM SOLVING WITH C++

SIXTH EDITION

Nell Dale and Chip Weems

# Chapter 9

## Scope, Lifetime, and More on Functions


www.jblearning.com

# Chapter 8 Topics

- Local Scope vs. Global Scope of an Identifier
- Detailed Scope Rules to Determine which Variables are Accessible in a Block
- Determining the Lifetime of a Variable
- Writing a Value-Returning Function for a Task
- Some Value-Returning Functions with Prototypes in Header Files `cctype` and `cmath`
- Creating and Using a Module Structure Chart
- Stub Testing a Program

# Scope of Identifier

The **scope** of an identifier (or named constant) is the region of program code in which it is legal to use that identifier for any purpose

# Local Scope vs. Global Scope

- The scope of an identifier that is declared *inside* a block (this includes function parameters) extends from the point of declaration to the end of the block

- The scope of an identifier that is declared *outside* of all namespaces, functions, and classes extends from point of declaration to the end of the entire file containing the program code

```
const  float   TAX_RATE = 0.05;  // Global constant
float  tipRate;                  // Global variable
void   handle (int,  float);     // Function prototype

using  namespace  std;

int  main ()
{
    int      age;    // age and bill local to this block
    float   bill;
    .                // a, b, and tax cannot be used
here
    .                // TAX_RATE and tipRate can be used
    handle (age, bill);

    return 0;
}

void  handle (int a,  float b)
{
     float  tax;     // a, b, and tax local to this
block
     .               // age and bill cannot be used here
     .               // TAX_RATE and tipRate can be used
}
```

# Detailed Scope Rules

1 **Function names have global scope**

2 **A function parameter's scope is identical to the scope of a local variable declared in the outermost block of the function body**

3 **A global variable's (or constant's) scope extends from its declaration to the end of the file, except as noted in rule 5**

4 **A local variable's (or constant's) scope extends from its declaration to the end of the block in which it is declared, including any nested blocks, except as noted in rule 5**

5 **An identifier's scope does not include any nested block that contains a locally declared identifier with the same name (local identifiers have name precedence)**

# Name Precedence Implemented by Compiler Determines Scope

- When an expression refers to an identifier,
  - The compiler first checks the local declarations
  - If the identifier isn't local, the compiler works outward through each level of nesting until it finds an identifier with same name where it stops
- Any identifier with the same name declared at a level further out is never reached
- If compiler reaches global declarations and still can't find the identifier, an error message results

# Namespace Scope

- The scope of an identifier declared in a namespace definition extends from the point of declaration to the end of the namespace body, and its scope includes the scope of a using directive specifying that namespace

# 3 Ways to Use Namespace Identifiers

- Use a qualified name consisting of the namespace, the scope resolution operator :: and the desired the identifier

```
alpha = std::abs(beta);
```

- Write a using declaration

```
using std::abs;
alpha = abs(beta);
```

- Write a using directive locally or globally

```
using namespace std;
alpha = abs(beta);
```

# Name Precedence (or Name Hiding)

- **When a function declares a local identifier with the same name as a global identifier, the local identifier takes precedence within that function**

# Memory Allocation

```
int  someInt;         // For the global variable

int  Square (int n) // For instructions in body
{
    int  result;      // For the local variable
    result  =  n * n;
    return  result;
}
```

# No Memory Allocation

```
int  Square (int n);
// Function prototype

extern  int  someInt;
// someInt is defined in another file
// and is thus global to everything in
// this file
```

# Lifetime of a Variable

- The lifetime of a variable is the time during program execution in which an identifier actually has memory allocated to it

# Lifetime of Local Automatic Variables

- Their storage is created (allocated) when control enters the function
- Local variables are "alive" while function is executing
- Their storage is destroyed (deallocated) when function exits

# Lifetime of Global Variables

- Their lifetime is the lifetime of the entire program
- Their memory is allocated when program begins execution
- Their memory is deallocated when the entire program terminates

# Automatic vs. Static Variable

- Storage for automatic variable is allocated at block entry and deallocated at block exit

- Storage for static variable remains allocated throughout execution of the entire program

# Default Allocation

- Local variables are automatic
- To obtain a static local variable, you must use the reserved word `static` in its declaration

# Static and Automatic Local Variables

```
int  popularSquare(int  n)
{
    static int timesCalled = 0;
    // Initialized only once
    int result =  n * n;
    // Initialized each time

    timesCalled  =  timesCalled + 1;
    cout  << "Call # "  << timesCalled  << endl;
    return  result;
}
```

# Data Flow Determines Passing-Mechanism

| Parameter Data Flow | Passing-Mechanism |
|---|---|
| Incoming /* in */ | Pass-by-value |
| Outgoing /* out */ | Pass-by-reference |
| Incoming/outgoing /* inout */ | Pass-by-reference |

# Prototype for `float` Function

AmountDue() is a function with 2 parameters

The first is type char, the other is type int

```
float  AmountDue (char,  int);
```

This function calculates and returns the amount due for local phone calls

The char parameter contains either a 'U' or an 'L' indicating Unlimited or Limited service; the int variable contains the number of calls made

Assume Unlimited service is $40.50 per month and limited service is $19.38 for up to 30 calls, and $.09 per additional call

```
float AmountDue (char kind, int calls)
// Two parameters
{
    float  result;   // One local variable

    const  float  UNLIM_RATE = 40.50,
                  LIM_RATE = 19.38,
                  EXTRA = .09;
    if (kind ==‘U’)
        result = UNLIM_RATE;

    else if ((kind == ‘L’) && (calls <= 30))
        result = LIM_RATE;

    else
        result = LIM_RATE + (calls - 30) * EXTRA;

    return result;
}
```

```
#include  <iostream>
#include  <fstream>
float AmountDue (char,  int);     // Prototype
using  namespace  std;

void  main ()
{
    ifstream  myInfile;
    ofstream  myOutfile;
    int       areaCode, phoneNumber, calls;
    int       count  = 0;
    float     bill;
    char      service;
        . . . . . .                 // Open files
    while (count < 100)
    {
        myInfile >> service >> phoneNumber >> calls;
        bill = AmountDue (service, calls); // Function call
        myOutfile << phoneNumber << bill << endl;
        count++;
     }
       . . . . . .                  // Close files
}
```

# To handle the call

# `AmountDue(service, calls)`

**MAIN PROGRAM MEMORY**

| Locations: | 4000 | 4002 | 4006 |
|---|---|---|---|
| | 200 | ? | 'U' |
| | calls | bill | service |

**TEMPORARY MEMORY for function to use**

| Locations: | 7000 | 7002 | 7006 |
|---|---|---|---|
| | calls | result | kind |

# Handling Function Call

```
bill  = AmountDue(service, calls);
```

- Begins by evaluating each argument
- A copy of the value of each is sent to temporary memory which is created and waiting for it
- The function body determines result
- Result is returned and assigned to bill

```
int   Power (/* in */    int    x,  // Base number
              /* in */    int    n)  // Power

// This function computes x to the n power
// Precondition:
//    x is assigned && n >= 0 && (x to the n) <= INT_MAX
// Postcondition:
//    Return value  ==  x to the n power

{
    int  result;      // Holds intermediate powers of x
    result  =  1;
    while  (n > 0)
    {
        result  =  result  *  x;
        n--;
    }
    return  result;
}
```

# Syntax Template for Function Definition

```
DataType  FunctionName  ( Parameter List )
{
    Statement
        .
        .
        .
}
```

# Using `bool` Type with a Loop

```
  . . .
bool  dataOK;  // Declare Boolean variable
float temperature;
     .   .   .
dataOK = true; // Initialize the Boolean variable
while (dataOK)
{
    .   .   .
    if (temperature  >  5000)
        dataOK = false;
}
```

# A Boolean Function

```
bool IsTriangle ( /* in */   float  angle1,
                  /* in */   float  angle2,
                  /* in */   float  angle3)
// Function checks if 3 incoming values add up to
//     180 degrees, forming a valid triangle
// Precondition:
//     angle1, angle2, angle3 are assigned
// Postcondition:
//     Return == true, f sum is within 0.000001 of
//                180.0 degrees
//            == false, otherwise
{
    return (fabs(angle1 + angle2 + angle3 - 180.0)
      <  0.000001);

}
```

# Some Prototypes in Header File < cctype >

```
int    isalpha (char  ch);
//  If ch is an alphabet character,
//       Return value == nonzero
//                    == zero, otherwise
```

```
int    isdigit (char  ch);
//  If ch is a digit ('0' - '9'),
//       Return value == nonzero
//                    == zero, otherwise
```

```
int   islower (char  ch);
//  If ch is a lowercase letter ('a' - 'z'),
//       Return value == nonzero
//                    == zero, otherwise
```

```
int   isupper (char ch);
//  If ch is an uppercase letter ('A' - 'Z'),
//       Return value == nonzero
//                    == zero, otherwise
```

# Some Prototypes in Header File < cmath >

```
double   cos (double  x);
// Return value == trigonometric cosine of angle x
//      in radians
```

```
double   exp (double  x);
// Return value == the value e (2.718 . . .) raised to
//      the power x
```

```
double   log (double x);
// Return value == natural (base e) logarithm of x
```

```
double   log10 (double x);
// Return value == common (base 10) logarithm of x
```

```
double  pow (double  x, double y);
// Return value == x raised to the power y
```

# *What will the function do with your argument(s)?*

**The answer to this question determines whether your function parameter should be value or reference as follows . . .**

# Value vs Reference

| If the function-- | Function parameter should be-- |
|---|---|
| only uses its value | */* in */* value parameter |
| assigns it a value | */* out */* reference parameter using & |
| changes its value | */* inout */* reference parameter using & |

***NOTE: I/O stream variables and arrays are exceptions***

# *Use Void or Value-Returning Functions?*

1. **If it must return more than one value or modify any of the caller's arguments, do not use a value-returning function**

2. **If it must perform I/O, do not use a value-returning function**

3. **If there is only one value returned, and it is Boolean, a value-returning function is appropriate**

4. **If there is only one value returned, and that value will be used immediately in an expression, a value-returning function is appropriate**

5. **When in doubt, use a void function; you can recode any value-returning function as a void function by adding an extra outgoing parameter**

6. **If both void and value-returning are acceptable, use the one you prefer**

# Use Stubs in Testing a Program

A **stub** is a dummy function with a very simple body, often just an output statement that this function was reached, and a return value (if any is required) of the correct type

Its name and parameter list is the same as the function that will actually be called by the program being tested