

CSE 1062 **Fundamentals of Programming**

Lecture #5

Spring 2016

Computer Science & Engineering Program
The School of EE & Computing
Adama Science & Technology University



I/O statements and functions

- Using Library Functions
 - Formatting Numbers for Program Output
 - Mathematical Library Functions
- Input using the **cin** object
- Symbolic Constants
- Writing Functions
 - Function and parameter declarations
 - Returning a value
 - Pass by Value and Pass by Reference
- Variable Scope and Lifetime

Case Study: (Chemistry) Acid Rain

(General Math) Length Conversion

(General Math) Rectangular to Polar Coordinate Conversion

Reading assignment

- Chapter 3 of the textbook
- Chapter 6 of the textbook
- Read about
 - Function Overloading
 - Default Arguments

- Proper output formatting contributes to ease of use and user satisfaction
- **cout** with stream manipulators can control output formatting

Manipulator	Action
setw (n)	Set the field width to n
scientific	Set the output to display real numbers in scientific notation
endl	Output a newline character and display all characters in the buffer
fixed	Always show a decimal point and use a default of six digits after the decimal point. Fill with trailing zeros, if necessary.
setprecision(n)	Set the floating-point precision to n places.

Formatting Numbers for Program Output



ASTU

```
1  #include <iostream>
2  #include <iomanip>
3  using namespace std;
4  int main()
5  {
6      cout << setw(3) << 6 << endl
7          << setw(3) << 18 << endl
8          << setw(3) << 124 << endl
9          << "---\n"
10         << (6+18+124) << endl;
11     return 0;
12 }
13
```



- The field width manipulator must be included for each value in the data stream sent to **cout**
- Other manipulators remain in effect until they are changed
- **iomanip** header file must be included to use manipulators requiring arguments

- Formatting floating-point numbers requires three field-width manipulators to:
 - Set the total width of the display
 - Force a decimal place
 - Set the number of significant digits after the decimal point

- Example:

```
cout << "|" << setw(10) << fixed  
      << setprecision(3) << 25.67 << "|";
```

produces this output: | 25.670|

- C++ has preprogrammed mathematical functions that can be included in a program
- You must include the **cmath** header file:
#include <cmath>
 - Math functions require one or more arguments as input, but will return only one value
 - All functions are **overloaded**, and can be used with integer and real arguments

Using Mathematical Library Functions



ASTU

Function Name	Description	Returned Value
<code>abs(a)</code>	Absolute value	Same data type as argument
<code>pow(a1,a2)</code>	a1 raised to the a2 power	Same data type as argument a1
<code>sqrt(a)</code>	Square root of a real number	Double-precision
<code>sin(a)</code>	Sine of a (a in radians)	Double
<code>cos(a)</code>	Cosine of a (a in radians)	Double
<code>tan(a)</code>	Tangent of a (a in radians)	Double
<code>log(a)</code>	Natural logarithm of a	Double
<code>log10(a)</code>	Common log (base 10) of a	Double
<code>exp(a)</code>	e raised to the a power	Double

- To use a math function, give its name and pass the input arguments within parentheses
- Expressions that can be evaluated to a value can be passed as arguments

function-name (data passed to the function);
This identifies the called function This passes data to the function

Using Mathematical Library Functions



ASTU

```
1  #include <iostream> // this line can be placed second instead of first
2  #include <cmath>    // this line can be placed first instead of second
3  using namespace std;
4  int main()
5  {
6      int height;
7      double time;
8      height = 800;
9      time = sqrt(2 * height / 32.2);
10     cout << "It will take " << time << " seconds to fall "
11          << height << " feet.\n";
12     return 0;
13 }
14
```

- **cin Object:** Allows data entry to a running program
- Use of the **cin** object causes the program to wait for input from the keyboard
- When keyboard entry is complete, the program resumes execution, using the entered data
- An output statement preceding the **cin** object statement provides a **prompt** to the user



Program Input Using cin

```
#include <iostream>
using namespace std;
int main()
{
    double num1, num2, product;
    cout << "Please type in a number: ";
    cin >> num1;
    cout << "Please type in another number: ";
    cin >> num2;
    product = num1 * num2;
    cout << num1 << " times " << num2 << " is " << product << endl;
    return 0;
}
```

- **cin** can accept multiple input values to be stored in different variables
- Multiple numeric input values must be separated by spaces

Example:

```
cin >> num1 >> num2
```

with keyboard entry: **0.052 245.79**

Program Input Using cin



```
#include <iostream>
using namespace std;
int main()
{
    int num1, num2, num3;
    double average;
    cout << "Enter three integer numbers: ";
    cin >> num1 >> num2 >> num3;
    average = (num1 + num2 + num3) / 3.0;
    cout << "The average of the numbers is " << average << endl;
    return 0;
}
```

- **User-input validation:** The process of ensuring that data entered by the user matches the expected data type
- **Robust program:** One that detects and handles incorrect user entry

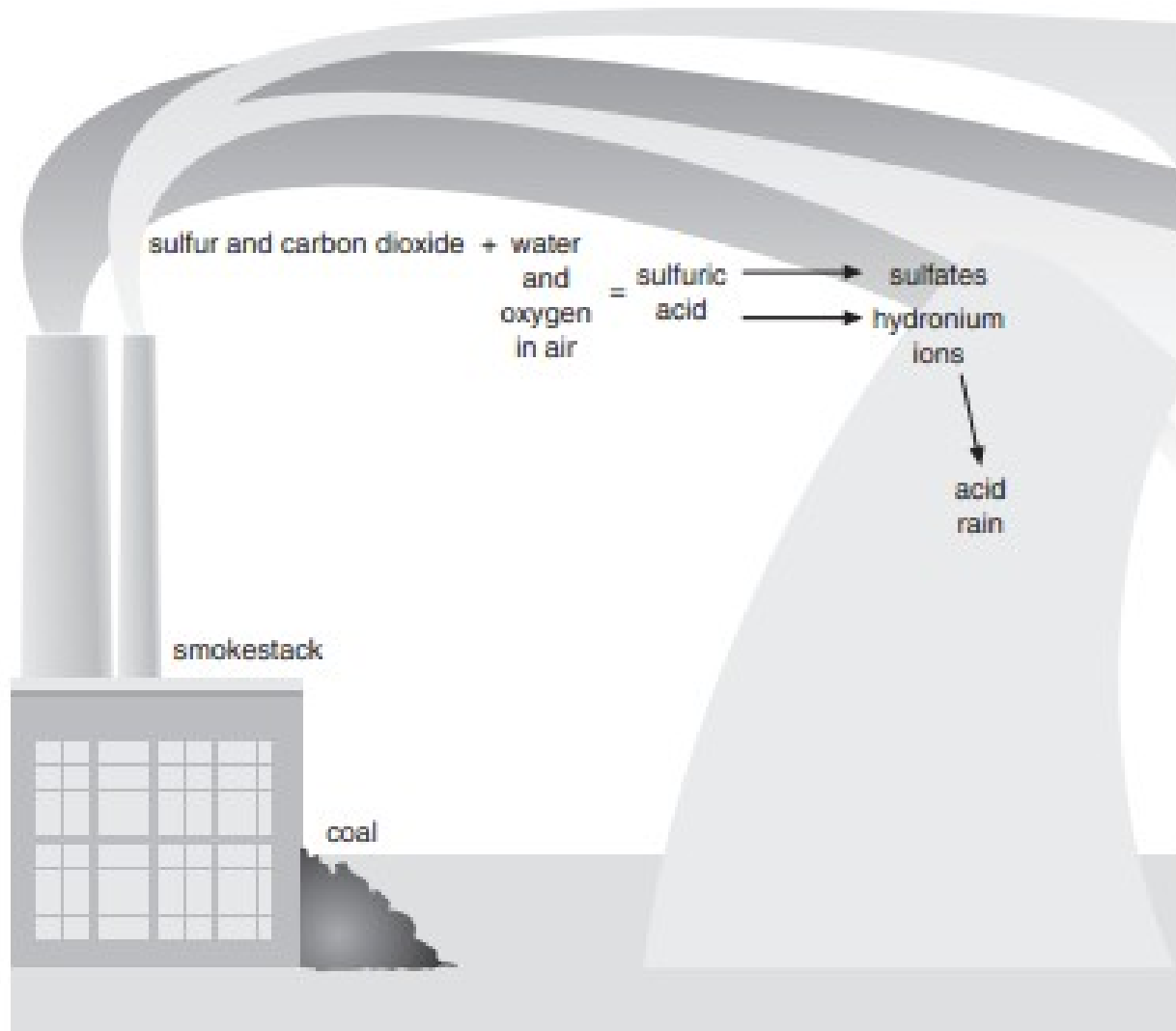
- Currently, coal is one of the principal sources of electrical power generation in many industrialized countries.
 - it is known that the oxygen used in the burning process combines with the carbon and sulfur in coal to produce carbon dioxide and sulfur dioxide.

- When these gases are released into the atmosphere, sulfur dioxide combines with water and oxygen in the air to form sulfuric acid
 - Then it is transformed into separate hydronium ions and sulfates see figure.
- The hydronium ions in the atmosphere that fall to Earth as components of rain are what change the acidity levels of lakes and forests

Case Study: Acid Rain



ASTU



- The acid level of rain and lakes is measured on a pH scale by using this formula:

$$pH = -\log_{10}(\text{concentration of hydronium ions})$$

- The concentration of hydronium ions is measured in units of moles/liter.
- pH value 7 $\Rightarrow$ neutral value
- pH below 7 $\Rightarrow$ acid,
- pH above 7 $\Rightarrow$ alkaline
- Marine life usually can't survive in water with a pH level below 4.

- Using the formula for pH, we will write a C++ program, using the software development procedure described in Lecture 1,

- **Acid Rain:** Develop a program to calculate the pH level of a substance based on user input of the concentration of hydronium ions
 - Step 1: Analyze the Problem
 - Step 2: Develop a Solution
 - Step 3: Code the Solution
 - Step 4: Test and Correct the Program

Step 1: Analyze the Problem

- Although the problem statement provides technical information on the composition of acid rain, from a programming viewpoint, this problem is rather simple.
- You have only one required output (a pH level) and one input (the concentration of hydronium ions).

Step 2: Develop a Solution

- Display a prompt to enter an ion concentration level
- Read a value for the concentration level
- Calculate a pH level, using the given formula
- Display the calculated value

Step 3: Code the Solution

```
1  #include <iostream>
2  #include <cmath>
3  using namespace std;
4  int main()
5  {
6      double hydron, pHlevel;
7      cout << "Enter the hydronium ion concentration: ";
8      cin >> hydron;
9      pHlevel = -log10(hydron);
10     cout << "The pH level is " << pHlevel << endl;
11     return 0;
12 }
13
```

Step 4: Test and Correct the Program



ASTU

- First test

A screenshot of a Windows command prompt window titled "C:\Lecture 5\AcidRain\bin\Debug\AcidRain.exe". The window shows the following text: "Enter the hydronium ion concentration: 0.0003", "The pH level is 3.52288", "Process returned 0 (0x0) execution time : 3.721 s", and "Press any key to continue.".

- Second test

- Think about validating the user input so that the pH level becomes between 0 and 14
 - We need selection statements

A screenshot of a Windows command prompt window titled "C:\Lecture 5\AcidRain\bin\Debug\AcidRain...". The window shows the following text: "Enter the hydronium ion concentration: 2", "The pH level is -0.30103", "Process returned 0 (0x0) execution time : 4.188", and "Press any key to continue.".

- **Symbolic constant:** Constant value that is declared with an identifier using the **const** keyword
- A constant's value may not be changed

Example:

```
const int MAXNUM = 100;
```

- Good programming places statements in appropriate order

- **Length Conversion:** Write a program that takes as input given lengths expressed in feet and inches. The program should then convert and output the lengths in centimeters. Assume that the given lengths in feet and inches are integers.
 - Step 1: Analyze the Problem
 - Step 2: Develop a Solution
 - Step 3: Code the Solution
 - Step 4: Test and Correct the Program

Step 1: Analyze the Problem

- The lengths are given in feet and inches, and we need to find the equivalent length in centimeters.
- **Input:** Length in feet and inches.
 - **Output:** Equivalent length in centimeters.

Step 2: Develop a Solution

1. Get the length in feet and inches.
2. Convert the length into total inches.
 - Convert the feet into inches
 - Multiply the number of feet by 12
 - Add the given inches
3. Convert total inches into centimeters.
 - Multiply the total inches by 2.54
4. Output centimeters.

Step 3: Coding



```
1  #include <iostream>
2  using namespace std;
3  //Named constants
4  const double CENTIMETERS_PER_INCH = 2.54;
5  const int INCHES_PER_FOOT = 12;
6  int main ()
7  {
8  //Declare variables
9      int feet, inches;
10     int totalInches;
11     double centimeter;
12     cout << "Enter two integers, one for feet and "
13           << "one for inches: ";
14     cin >> feet >> inches;
15     cout << endl;
16     cout << "The numbers you entered are " << feet
17           << " for feet and " << inches
18           << " for inches. " << endl;
19     totalInches = INCHES_PER_FOOT * feet + inches;
20     cout << "The total number of inches = "
21           << totalInches << endl;
22     centimeter = CENTIMETERS_PER_INCH * totalInches;
23     cout << "The number of centimeters = "
24           << centimeter << endl;
25     return 0;
26 }
27
28
```

- 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

- 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

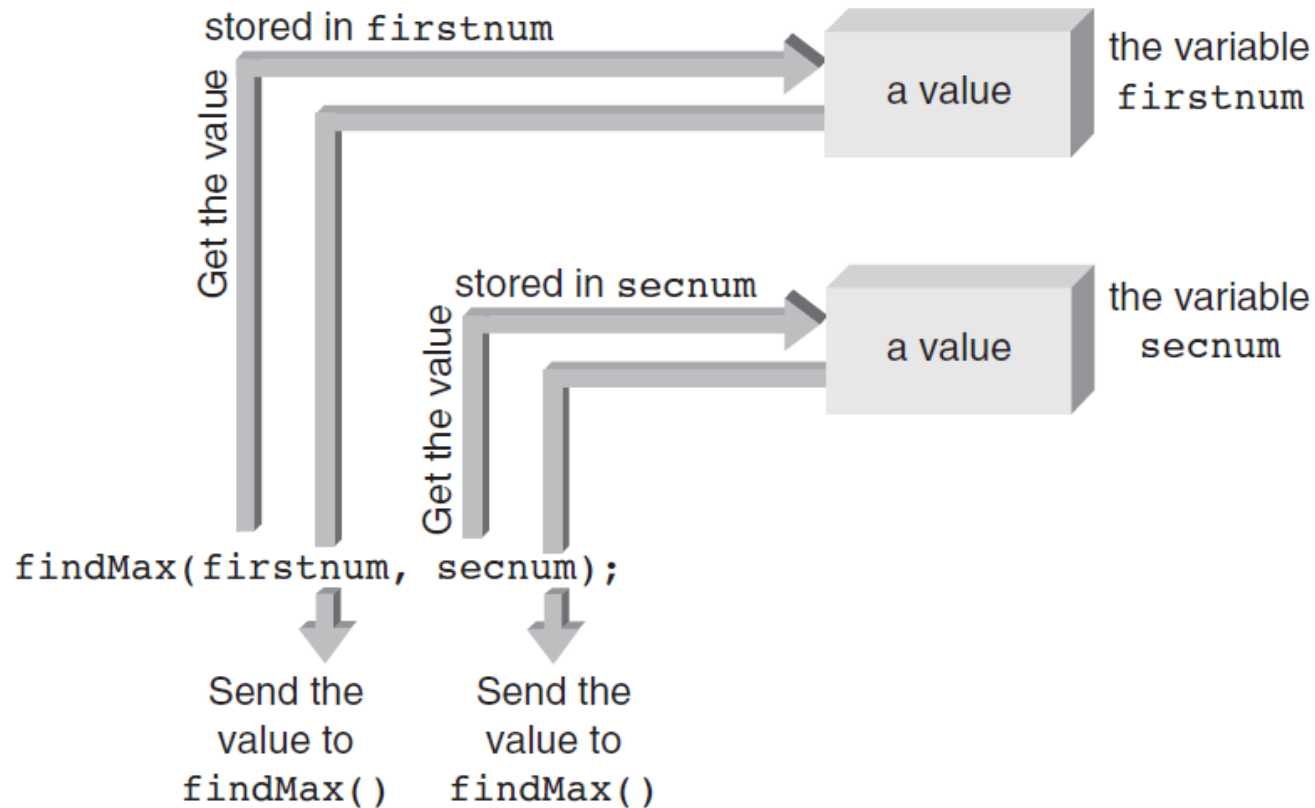
- Requirements when calling a function include:
 - Using the name of the function
 - Enclosing any data passed to the function in the parentheses following the function name, using the same order and type declared in the function prototype

- 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()`

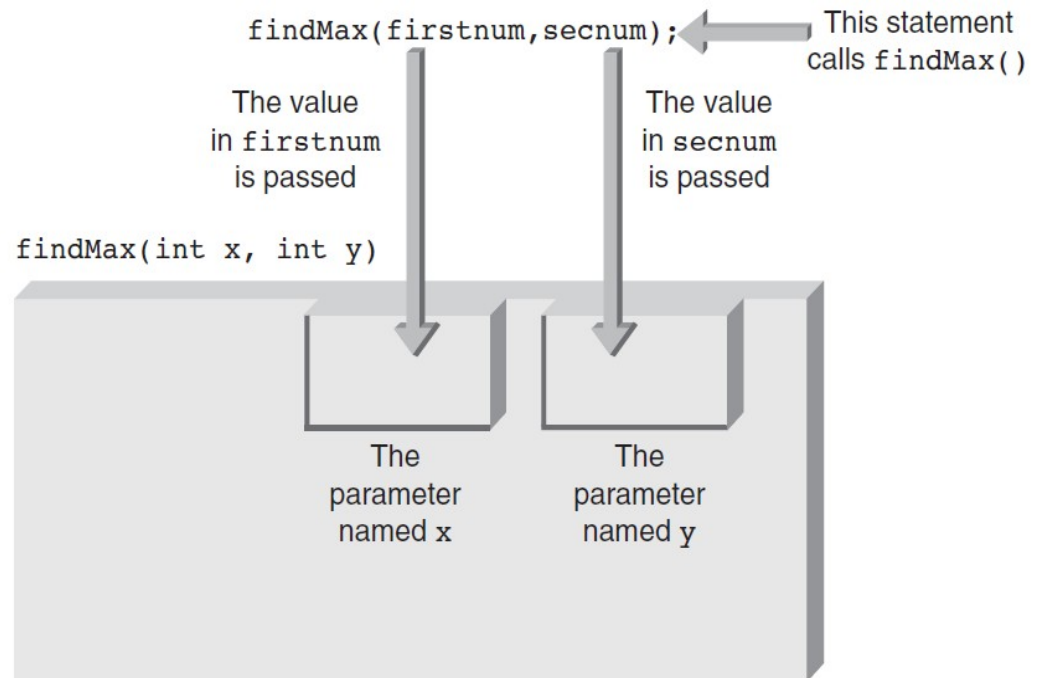
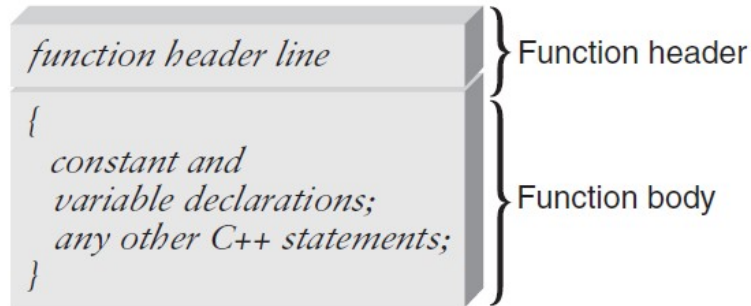
Calling a Function



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 and Calling Functions

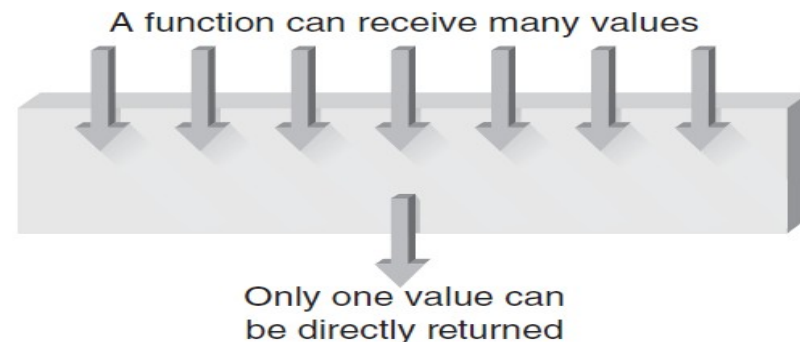


- General rule for placing statements in a C++ program:
 - All preprocessor directives, named constants, variables, and functions must be declared or defined before they can be used
 - Although this rule permits placing both preprocessor directives and declaration statements throughout the program, doing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

Empty Parameter List

- 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);`

- 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





Pass By Value and Pass By Reference

- 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

Pass By Value and Pass By Reference



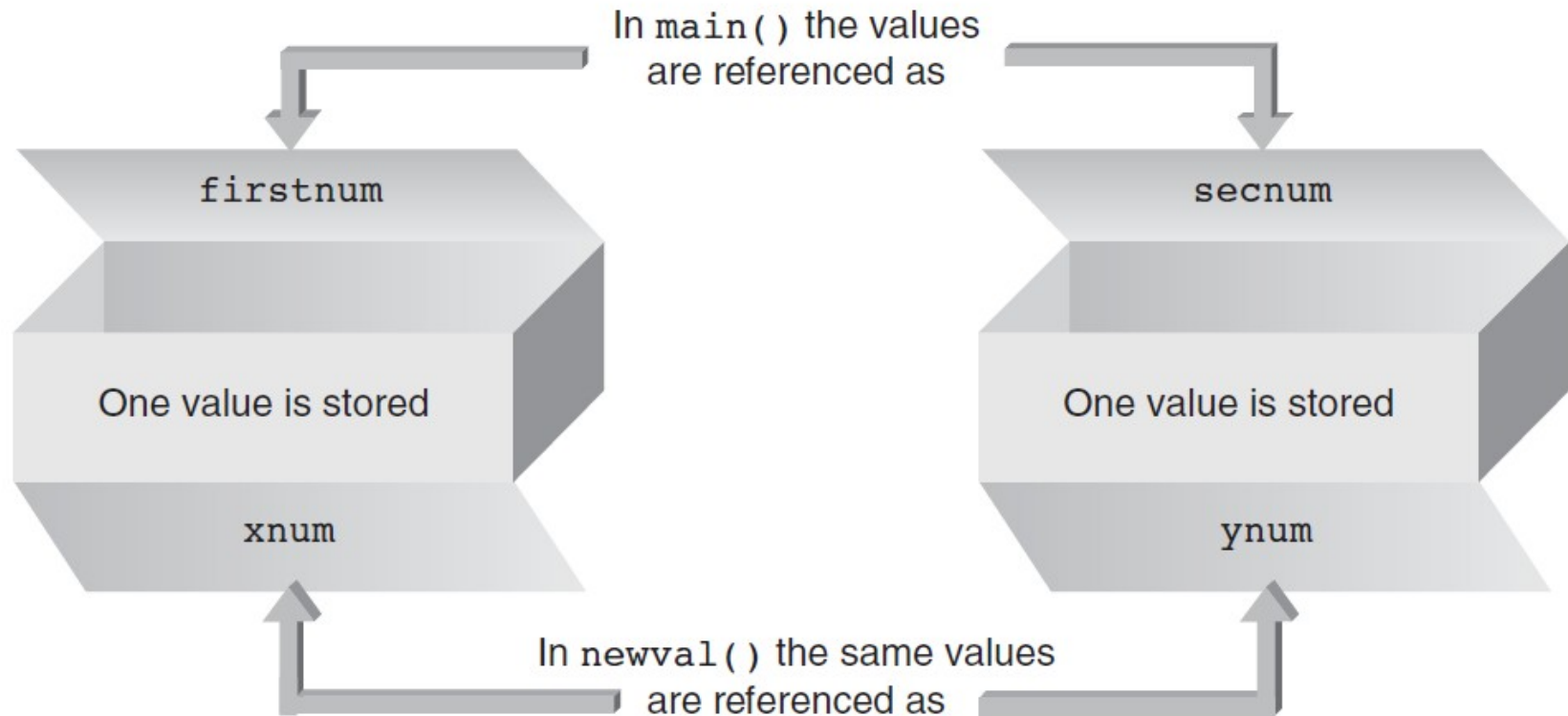
```
1  #include <iostream>
2  using namespace std;
3  void newval(double&, double&); // prototype with two reference parameters
4  int main()
5  {
6      double firstnum, secnum;
7      cout << "Enter two numbers: ";
8      cin >> firstnum >> secnum;
9      cout << "\nThe value in firstnum is: " << firstnum << endl;
10     cout << "The value in secnum is: " << secnum << "\n\n";
11     newval(firstnum, secnum); // call the function
12     cout << "The value in firstnum is now: " << firstnum << endl;
13     cout << "The value in secnum is now: " << secnum << endl;
14     return 0;
15 }
16 void newval(double& xnum, double& ynum)
17 {
18     cout << "The value in xnum is: " << xnum << endl;
19     cout << "The value in ynum is: " << ynum << "\n\n";
20     xnum = 89.5;
21     ynum = 99.5;
22     return;
23 }
```

- From the sending side(caller)
 - 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
- Address operator (&) provides the variable's address

Pass By Value and Pass By Reference



ASTU

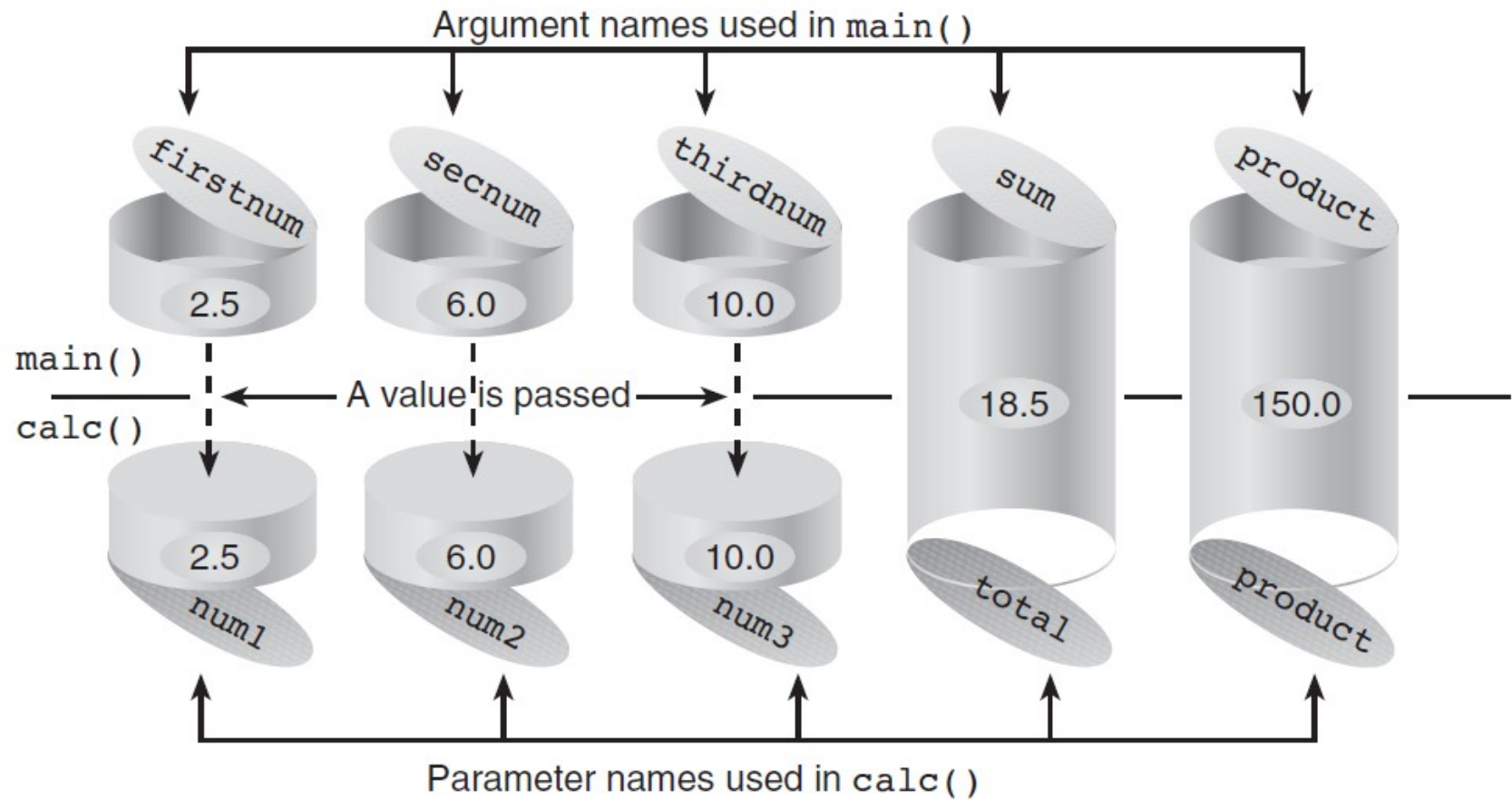




Pass By Value and Pass By Reference

```
1  #include <iostream>
2  using namespace std;
3  void calc(double, double, double, double&, double&); // prototype
4  int main()
5  {
6      double firstnum, secnum, thirdnum, sum, product;
7      cout << "Enter three numbers: ";
8      cin >> firstnum >> secnum >> thirdnum;
9      calc(firstnum, secnum, thirdnum, sum, product); // function call
10     cout << "\nThe sum of the numbers is: " << sum << endl;
11     cout << "The product of the numbers is: " << product << endl;
12     return 0;
13 }
14 void calc(double num1, double num2, double num3, double& total, double& product)
15 {
16     total = num1 + num2 + num3;
17     product = num1 * num2 * num3;
18     return;
19 }
20
```

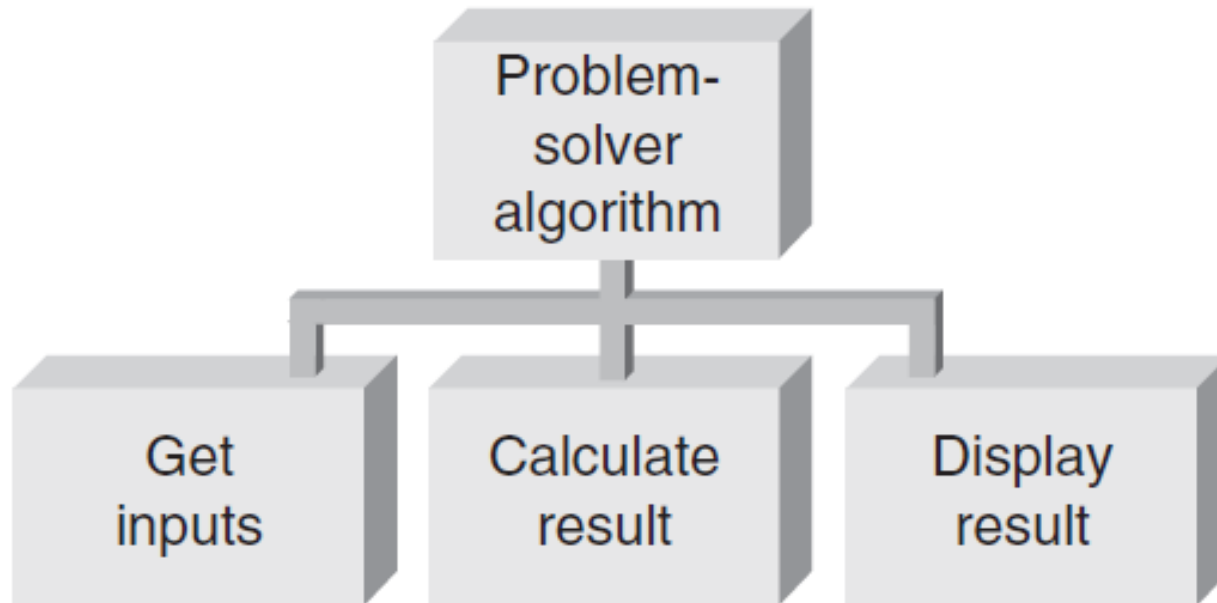
Pass By Value and Pass By Reference



- We follow Top Down Design and Step-wise refinement
 - This initial outline begins the process of defining a more complicated problem as a set of smaller, more manageable tasks.
 - Each of these tasks can be further subdivided or refined into even smaller tasks, if required.
 - After the tasks are well defined, the actual work of coding can begin, starting with any task in any order.

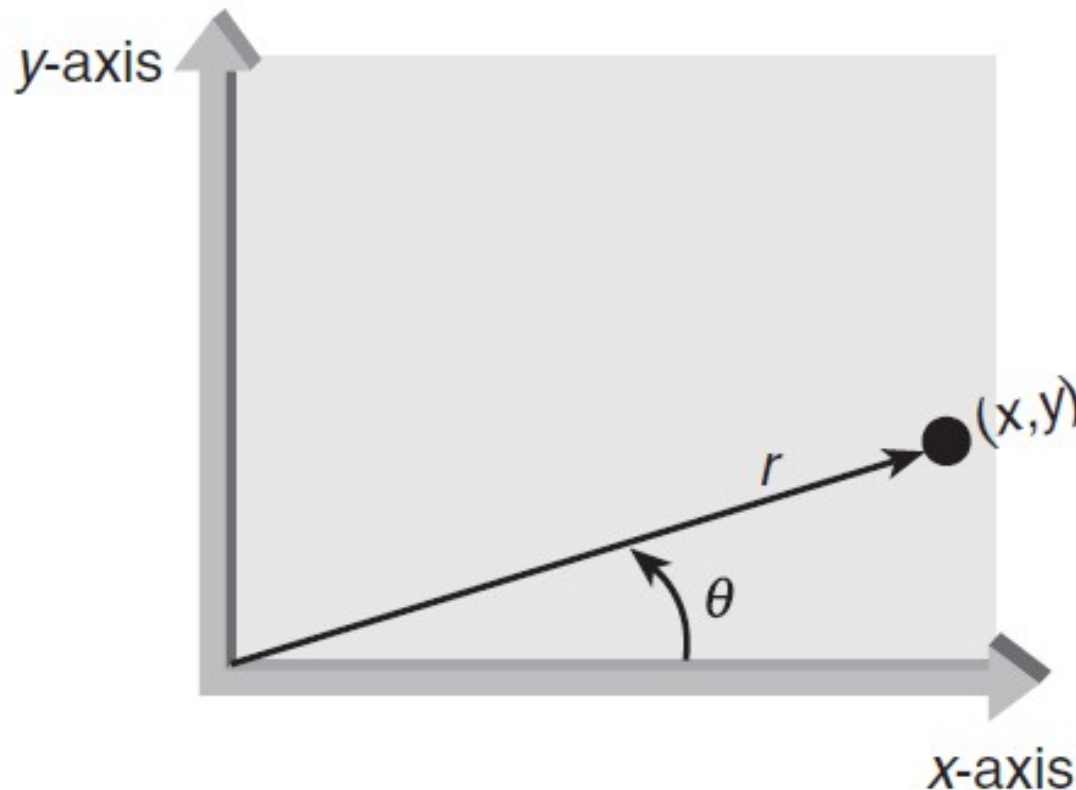


- Top Level Algorithm



Step 1: Define a Problem

- Correspondence between polar(distance and angle) and Cartesian(x and y) coordinate



Step 1: Define a Problem

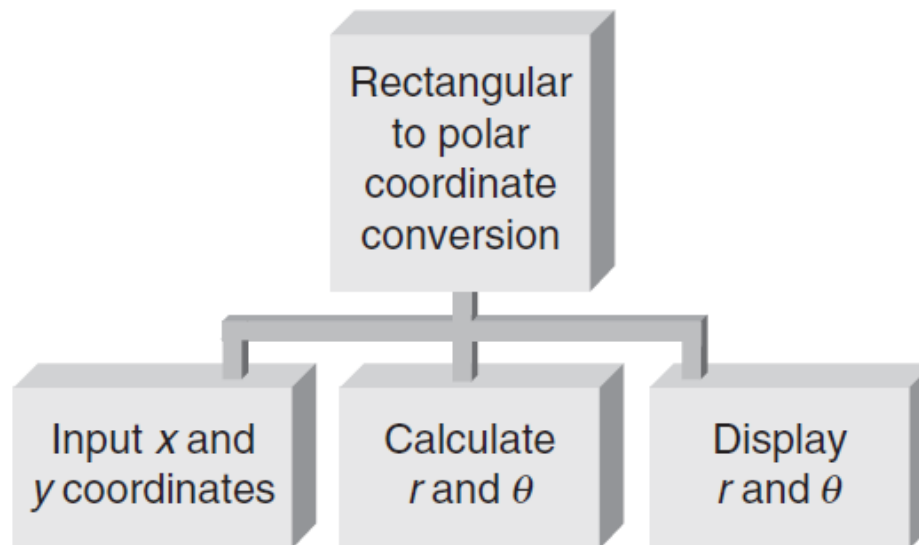
- When the x- and y-coordinates of a point are known, the equivalent r and θ coordinates can be calculated by using these formulas

$$r = \sqrt{x^2 + y^2}$$

$$\theta = \tan^{-1}(y/x); x \neq 0$$

Step 1: Define a Problem

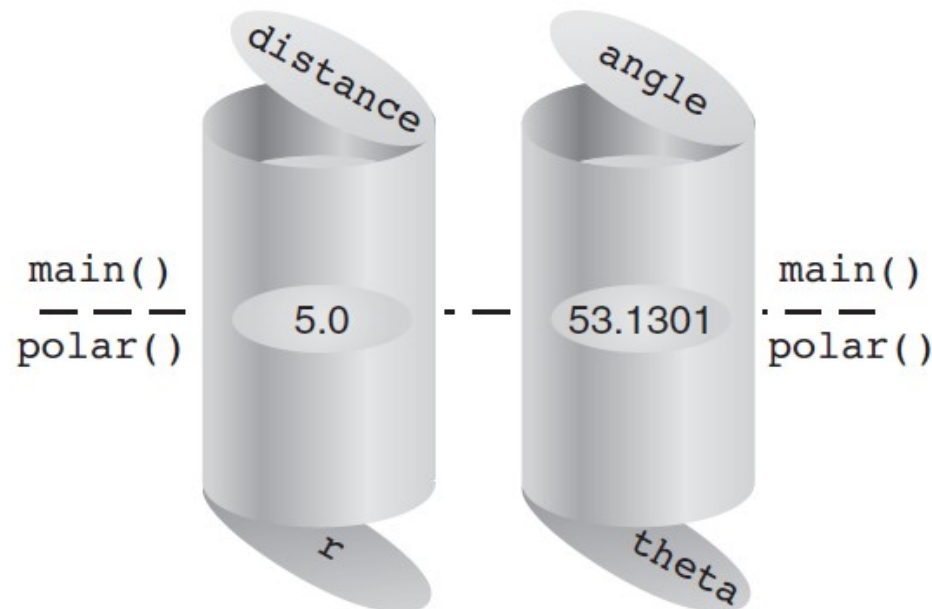
- Top Level Structure(Initial Pseudo Code)
 - Get the x- and y-coordinate values
 - Calculate the polar (r and θ) coordinate values
 - Display the polar coordinate values



Step 1: Define a Problem

- Initial Coding

```
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;
}
```



Step 2: Test the Function

- After polar() is written, it can be tested independently of any other function.

```
1  #include <iostream>
2  #include <cmath>
3  using namespace std;
4  void polar(double, double, double&, double&); // function prototype
5  int main()
6  {
7      double distance, angle;
8      polar(3.0, 4.0, distance, angle);
9      cout << "r = " << distance << endl;
10     cout << "angle = " << angle << endl;
11     return 0;
12 }
13 void polar(double x, double y, double& r, double& theta)
14 {
15     const double TODGREES = 180.0/3.141593;
16     r = sqrt(x * x + y * y);
17     theta = atan(y/x) * TODGREES;
18     return;
19 }
20
```

r=5

angle=53.1301

- The program also requires
 - writing functions for accepting two rectangular coordinates
 - The following function, **getrec()** , can be used to accept the input data
 - Test it
 - displaying the calculated polar coordinates
 - The function **showit()** for displaying polar coordinates is constructed in a similar manner.
 - Test it

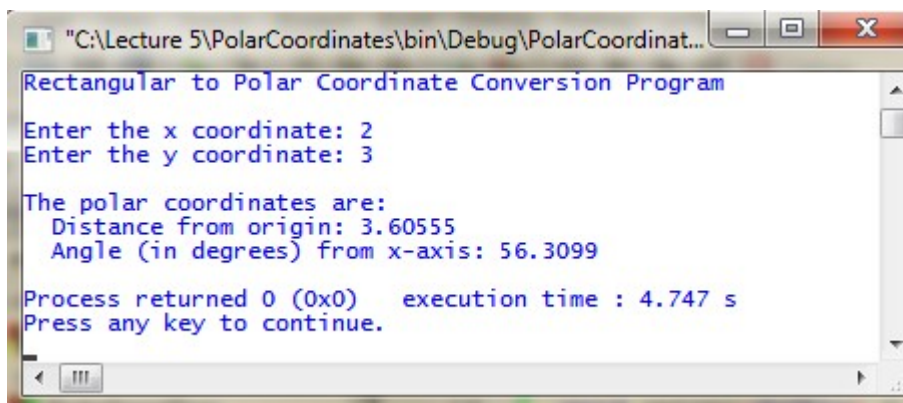
The Complete Program



```
1  /// This program converts rectangular coordinates to polar coordinates
2  // Functions used: getrec() - obtain the rectangular coordinates
3  // : polar() - calculate the polar coordinates
4  // : showit() - display the polar coordinates
5  #include <iostream>
6  #include <cmath>
7  using namespace std;
8  void getrec(double&, double&);           // function prototype
9  void polar(double, double, double&, double&); // function prototype
10 void showit(double, double);           // function prototype
11 int main()
12 {
13     double x, y, distance, angle;
14     getrec(x, y);
15     polar(x, y, distance, angle);
16     showit(distance, angle);
17     return 0;
18 }
19 void getrec(double& x, double& y)
20 {
21     cout << "Rectangular to Polar Coordinate"
22           << " Conversion Program\n" << endl;
23     cout << "Enter the x coordinate: ";
24     cin >> x;
25     cout << "Enter the y coordinate: ";
26     cin >> y;
27     return;
28 }
```

The Complete Program

```
29 void polar(double x, double y, double& r, double& theta)
30 {
31     const double TODEGREES = 180.0/3.141593;
32     r = sqrt(x * x + y * y);
33     theta = atan(y/x) * TODEGREES;
34     return;
35 }
36 void showit(double radius, double angle)
37 {
38     cout << "\nThe polar coordinates are: " << endl;
39     cout << "    Distance from origin: " << radius << endl;
40     cout << "    Angle (in degrees) from x-axis: " << angle << endl;
41     return;
42 }
43
```



```
"C:\Lecture 5\PolarCoordinates\bin\Debug\PolarCoordinat...
Rectangular to Polar Coordinate Conversion Program
Enter the x coordinate: 2
Enter the y coordinate: 3

The polar coordinates are:
Distance from origin: 3.60555
Angle (in degrees) from x-axis: 56.3099

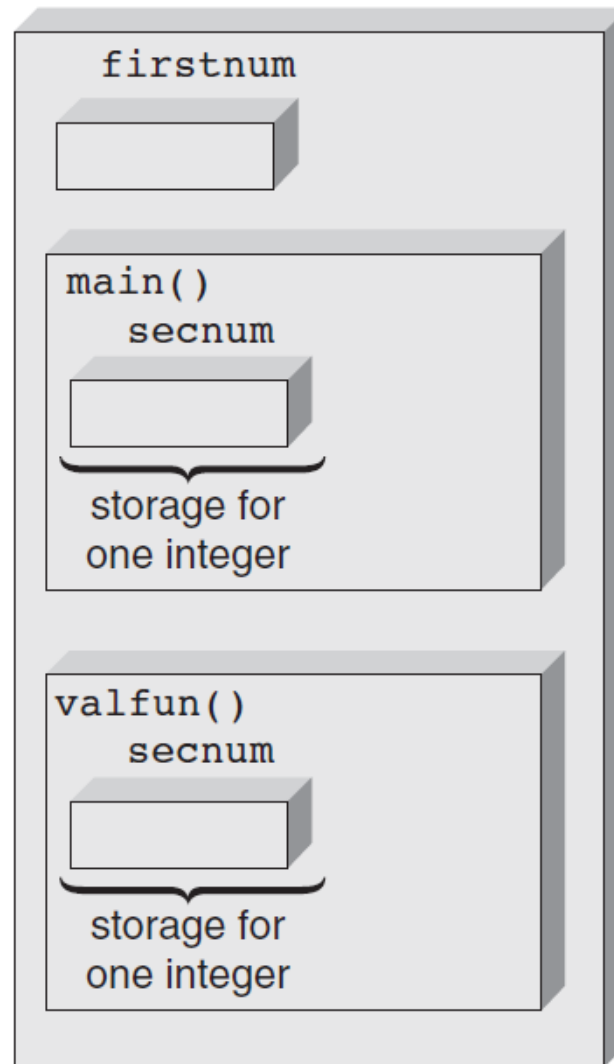
Process returned 0 (0x0)   execution time : 4.747 s
Press any key to continue.
```

- **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



ASTU



- 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

```
1  #include <iostream>
2  using namespace std;
3  double number = 42.8;           // a global variable named number
4  int main()
5  {
6      double number = 26.4;       // a local variable named number
7      cout << "The value of number is " << number << endl;
8      return 0;
9  }
```

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

```
1  #include <iostream>
2  using namespace std;
3  double number = 42.5;           // a global variable named number
4  int main()
5  {
6      double number = 26.4;       // a local variable named number
7      cout << "The value of number is " << ::number << endl;
8      return 0;
9  }
```

- 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

Lifetime of Variables

- 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

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

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 is called
 - 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

Local Automatic Variable



```
1  #include <iostream>
2  using namespace std;
3  void testauto();    // function prototype
4  int main()
5  {
6      int count;      // count is a local auto variable
7      testauto();
8      testauto();
9      testauto();
10     return 0;
11 }
12 void testauto()
13 {
14     int num = 0;      // num is a local auto variable
15                     // initialized to 0
16     cout << "The value of the automatic variable num is "
17          << num << endl;
18     num++;
19     return;
20 }
```

Local Static Variable

```
1  #include <iostream>
2  using namespace std;
3  void teststat();    // function prototype
4  int main()
5  {
6      int count;      // count is a local auto variable
7      teststat();
8      teststat();
9      teststat();
10     return 0;
11 }
12 void teststat()
13 {
14     static int num = 0;    // num is a local static variable
15     cout << "The value of the static variable num is now "
16          << num << endl;
17     num++;
18     return;
19 }
```

- 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