

# Chapter 3 - Functions

## Outline

- 3.1 Introduction
- 3.2 Program Components in C++
- 3.3 Math Library Functions
- 3.4 Functions
- 3.5 Function Definitions
- 3.6 Function Prototypes
- 3.7 Header Files
- 3.8 Random Number Generation
- 3.9 Example: A Game of Chance and Introducing `enum`
- 3.10 Storage Classes
- 3.11 Scope Rules



## 3.1 Introduction

- Divide and conquer
  - Construct a program from smaller pieces or components
  - Each piece more manageable than the original program



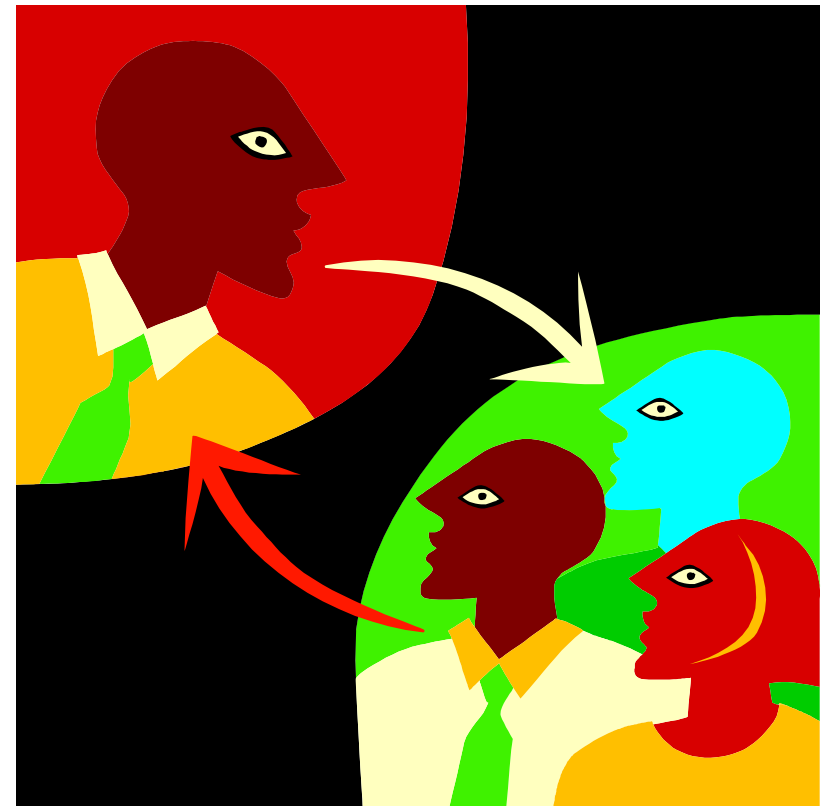
## 3.2 Program Components in C++

- Modules: functions and classes
- Programs use new and “prepackaged” modules
  - New: programmer-defined functions, classes
  - Prepackaged: from the standard library
- Functions invoked by function call
  - Function name and information (arguments) it needs
- Function definitions
  - Only written once
  - May be called many times
  - Internal details are hidden from other functions



## 3.2 Boss to worker analogy

- A boss (the calling function or caller) asks a
- worker (the called function) to
- perform a task (execute the function body) and
- report back (i.e. return) with the results when the task is done.



## 3.3 Math Library Functions

- Perform common mathematical calculations
  - Include the header file `<cmath>`
- Functions called by writing
  - `functionName (argument);`
  - or
  - `functionName(argument1, argument2, ...);`
- Example
  - `cout << sqrt( 900.0 );`
  - `sqrt` (square root) function  
(the preceding statement would print “30”)
  - All functions in math library return a **double**



## Example of using `<cmath>` functions



## Example of using <cmath> functions

```
// sqrtDemo.cc    P. Conrad, Spring 2004
// CISC181, 2/23/2004

#include <iostream>
using std::cout;
using std::endl;

#include <cmath>

int main(void)
{
    // write out square root of 36
    cout << sqrt(36.0) << endl;
    return 0;
}
```

## 3.3 Math Library Functions

- Function arguments can be
  - Constants
    - `sqrt( 4 );`
  - Variables
    - `sqrt( x );`
  - Expressions
    - `sqrt( sqrt( x ) );`
    - `sqrt( 3 - 6 * x );`



Method	Description	Example
<b>ceil( x )</b>	rounds $x$ to the smallest integer not less than $x$	ceil( 9.2 ) is 10.0 ceil( -9.8 ) is -9.0
<b>cos( x )</b>	trigonometric cosine of $x$ ( $x$ in radians)	cos( 0.0 ) is 1.0
<b>exp( x )</b>	exponential function $e^x$	exp( 1.0 ) is 2.71828 exp( 2.0 ) is 7.38906
<b>fabs( x )</b>	absolute value of $x$	fabs( 5.1 ) is 5.1 fabs( 0.0 ) is 0.0 fabs( -8.76 ) is 8.76
<b>floor( x )</b>	rounds $x$ to the largest integer not greater than $x$	floor( 9.2 ) is 9.0 floor( -9.8 ) is -10.0
<b>fmod( x, y )</b>	remainder of $x/y$ as a floating-point number	fmod( 13.657 , 2.333 ) is 1.992
<b>log( x )</b>	natural logarithm of $x$ (base $e$ )	log( 2.718282 ) is 1.0 log( 7.389056 ) is 2.0
<b>log10( x )</b>	logarithm of $x$ (base 10)	log10( 10.0 ) is 1.0 log10( 100.0 ) is 2.0
<b>pow( x, y )</b>	$x$ raised to power $y$ ( $x^y$ )	pow( 2, 7 ) is 128 pow( 9, .5 ) is 3
<b>sin( x )</b>	trigonometric sine of $x$ ( $x$ in radians)	sin( 0.0 ) is 0
<b>sqrt( x )</b>	square root of $x$	sqrt( 900.0 ) is 30.0 sqrt( 9.0 ) is 3.0
<b>tan( x )</b>	trigonometric tangent of $x$ ( $x$ in radians)	tan( 0.0 ) is 0



## Compiling with the math library

- When writing code, use `#include <cmath>`
- When compiling with some compilers, you may need to include `-lm` on command line
  - That's the letter "l" as in "library" and the letter "m" as in "math".
  - Examples:

```
CC myprog.cpp -lm
g++ myprog.cpp -lm
```
  - However, it seems that on strauss, g++ and CC both find the math library without any trouble, so maybe this isn't strictly necessary in our environment  
(at least on 2/23/2004 when I made this slide, that was true...)



## 3.4 User defined functions

- Functions
  - Help to break up a program into manageable pieces
    - “Modularize” a program
  - Software reusability
    - Call function multiple times
- Next: some program examples



fig03\_03.cpp  
 (1 of 1)

```

1 // Fig. 3.3: fig03_03.cpp
2 // Creating and using a programmer-defined function.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 int square( int ); // function prototype
9
10 int main()
11 {
12     // loop 10 times and calculate and output
13     // square of x each time
14     for ( int x = 1; x <= 10; x++ )
15         cout << square( x ) << " "; // function call
16
17     cout << endl;
18
19     return 0; // indicates successful termination
20
21 } // end main
22
23 // square function definition returns square of an integer
24 int square( int y ) // y is a copy of argument to function
25 {
26     return y * y; // returns square of y
27
28 } // end function square

```

Function prototype: specifies data types of arguments and return values. **square** expects and **int**, and returns an **int**.

Parentheses ( ) cause function to be called. When function is done, it returns the result, and the result is passed to the “cout <<” operation.

Definition of **square**. **y** is a copy of the argument passed. Returns **y \* y**, or **y** squared.

1 4 9 16 25 36 49 64 81 100

Outline

**fig03\_04.cpp**  
(1 of 2)

```
1 // Fig. 3.4: fig03_04.cpp
2 // Finding the maximum of three floating-point numbers.
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 double maximum( double, double, double ); // function prototype
10
11 int main()
12 {
13     double number1;
14     double number2;
15     double number3;
16
17     cout << "Enter three floating-point numbers: ";
18     cin >> number1 >> number2 >> number3;
19
20     // number1, number2 and number3 are arguments to
21     // the maximum function call
22     cout << "Maximum is: "
23         << maximum( number1, number2, number3 ) << endl;
24
25     return 0; // indicates successful termination
```



Function **maximum** takes 3 arguments (all **double**) and returns a **double**.

```
Enter three floating-point numbers: 99.32 37.3 27.1928
Maximum is: 99.32
```



## Outline

**fig03\_04.cpp**  
(2 of 2)

**fig03\_04.cpp**  
output (1 of 1)

Comma separated list for  
multiple parameters.

```
26
27 } // end main
28
29 // function maximum definition;
30 // x, y and z are parameters
31 double maximum( double x, double y, double z )
32 {
33     double max = x;    // assume x is largest
34
35     if ( y > max )    // if y is larger,
36         max = y;    // assign y to max
37
38     if ( z > max )    // if z is larger,
39         max = z;    // assign z to max
40
41     return max;    // max is largest value
42
43 } // end function maximum
```

Enter three floating-point numbers: 99.32 37.3 27.1928

Maximum is: 99.32

Enter three floating-point numbers: 1.1 3.333 2.22

Maximum is: 3.333

Enter three floating-point numbers: 27.9 14.31 88.99

Maximum is: 88.99

## 3.5 Function Definitions

- Format for function definition

```
return-value-type function-name( parameter-list )  
{  
    declarations and statements  
}
```

- Parameter list
  - Comma separated list of arguments
    - Data type needed for each argument
  - If no arguments, use **void** or leave blank
- Return-value-type
  - Data type of result returned (use **void** if nothing returned)



## 3.5 Function Definitions

- Function prototype
  - Tells compiler argument type and return type of function
  - **int square( int );**
    - Function takes an **int** and returns an **int**
  - Explained in more detail later
- Calling/invoking a function
  - **square(x);**
  - Parentheses an operator used to call function
    - Pass argument x
    - Function gets its own copy of arguments
  - After finished, passes back result



## 3.5 Facts about functions

- Local variables
  - Known only in the function in which they are defined
  - All variables declared in function definitions are local variables
  - All local variables (except static local variables) are located in memory in a place called the “stack”
    - Just memorize this fact for now; we’ll explain what it means later
- Parameters
  - local variables that are initialized with values passed from the caller (some other function that “calls” or “invokes” our user-defined function)
  - A way of getting “input” into the function without the user for it directly inside the function
- In C, C++: no function nesting
  - functions cannot be defined inside other functions
  - this is not true of some other languages (e.g. Pascal)



## 3.6 Function Prototypes

- Function prototype contains
  - Function name
  - Parameters (number and data type)
  - Return type (**void** if returns nothing)
  - Only needed if function call is before function definition (the prototype helps the compiler check the function call to make sure the number and type of parameters are correct)
- Prototype must match function definition
  - Function prototype
 

```
double maximum( double x, double y, double z );
```
  - Another legal way to do a function prototype (but **NEVER DO THIS !!!!!**)
 

```
double maximum( double , double , double );
```
  - Definition
 

```
double maximum( double x, double y, double z )
{
  ...
}
```

Syntax is legal, but style is horrible. Variable names help to document purpose of each variable.



## 3.6 Function Prototypes

- Function signature
  - Part of prototype with name and parameters
    - `double maximum( double x, double y, double z );`

Function signature
- Argument Coercion
  - Force arguments to be of proper type
    - Converting `int` (4) to `double` (4.0)
 

```
cout << sqrt(4)
```
  - Conversion rules
    - Arguments usually converted automatically
    - Changing from `double` to `int` can truncate data
      - 3.4 to 3
  - Mixed type goes to highest type (promotion)
 

```
int * double becomes a double
```



## 3.6 type promotion hierarchy

Data types	
long double	
double	
float	
unsigned long int	(synonymous with unsigned long)
long int	(synonymous with long)
unsigned int	(synonymous with unsigned)
int	
unsigned short int	(synonymous with unsigned short)
short int	(synonymous with short)
unsigned char	
char	
bool	(false becomes 0, true becomes 1)



## 3.7 Header Files

- Header files contain
  - Function prototypes
  - Definitions of data types and constants
- Header files ending with .h
  - Programmer-defined header files  
`#include "myheader.h"`
- Library header files  
`#include <cmath>`



## Spring 2004

### Conrad's "notes to self"

- 1:25pm lecture already covered random number generation in some detail, so skip to slide 24/25 to show "default case" issue, then to slide 30 to cover enumerated types.
- 8:00am and 9:05am lecture... didn't cover random number generation yet, so include slides 20-29.



## 3.8 Random Number Generation

- **rand** function (`<cstdlib>`)
  - `i = rand();`
  - Generates unsigned integer between 0 and `RAND_MAX` (usually 32767)
- **Scaling and shifting**
  - Modulus (remainder) operator: `%`
    - `10 % 3` is `1`
    - `x % y` is between `0` and `y - 1`
  - Example
    - `i = rand() % 6 + 1;`
      - `"Rand() % 6"` generates a number between `0` and `5` (scaling)
      - `" + 1"` makes the range `1` to `6` (shift)
  - Next: program to roll dice



**fig03\_07.cpp**  
 (1 of 1)

```

1 // Fig. 3.7: fig03_07.cpp
2 // Shifted, scaled integers produced by 1 + rand() % 6.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <iomanip>
9
10 using std::setw;
11
12 #include <cstdlib> // contains function prototype for rand
13
14 int main()
15 {
16     // loop 20 times
17     for ( int counter = 1; counter <= 20; counter++ ) {
18
19         // pick random number from 1 to 6 and output it
20         cout << setw( 10 ) << ( 1 + rand() % 6 );
21
22         // if counter divisible by 5, begin new line of output
23         if ( counter % 5 == 0 )
24             cout << endl;
25
26     } // end for structure
27
28     return 0; // indicates successful termination
29
30 } // end main

```

Output of `rand()` scaled and shifted to be a number between 1 and 6.

6	6	5	5	6
5	1	1	5	3
6	6	2	4	2
6	2	3	4	1

## 3.8 Random Number Generation

- Next
  - Program to show distribution of **rand( )**
  - Simulate 6000 rolls of a die
  - Print number of 1's, 2's, 3's, etc. rolled
  - Should be roughly 1000 of each





## Outline

**fig03\_08.cpp**  
(1 of 3)

```
1 // Fig. 3.8: fig03_08.cpp
2 // Roll a six-sided die 6000 times.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <iomanip>
9
10 using std::setw;
11
12 #include <cstdlib> // contains function prototype for rand
13
14 int main()
15 {
16     int frequency1 = 0;
17     int frequency2 = 0;
18     int frequency3 = 0;
19     int frequency4 = 0;
20     int frequency5 = 0;
21     int frequency6 = 0;
22     int face; // represents one roll of the die
23
```



## Outline

**fig03\_08.cpp**  
(2 of 3)

```
24 // loop 6000 times and summarize results
25 for ( int roll = 1; roll <= 6000; roll++ ) {
26     face = 1 + rand() % 6; // random number from 1 to 6
27
28     // determine face value and increment appropriate counter
29     switch ( face ) {
30
31         case 1: // rolled 1
32             ++frequency1;
33             break;
34
35         case 2: // rolled 2
36             ++frequency2;
37             break;
38
39         case 3: // rolled 3
40             ++frequency3;
41             break;
42
43         case 4: // rolled 4
44             ++frequency4;
45             break;
46
47         case 5: // rolled 5
48             ++frequency5;
49             break;
```



fig03\_08.cpp  
(3 of 3)

```

50
51     case 6:           // rolled 6
52         ++frequency6;
53         break;
54
55     default:         // invalid value
56         cout << "Program should never get here!";
57
58     } // end switch
59
60 } // end for
61
62 // display results in tabular
63 cout << "Face" << setw( 13 )
64     << "\n  1" << setw( 13 ) << frequency1
65     << "\n  2" << setw( 13 ) << frequency2
66     << "\n  3" << setw( 13 ) << frequency3
67     << "\n  4" << setw( 13 ) << frequency4
68     << "\n  5" << setw( 13 ) << frequency5
69     << "\n  6" << setw( 13 ) << frequency6 << endl;
70
71     return 0; // indicates successful termination
72
73 } // end main

```

Default case included even though it should never be reached. This is a matter of good coding style

Face	Frequency
1	1003
2	1017
3	983
4	994
5	1004
6	999

## 3.8 Random Number Generation

- Calling `rand()` repeatedly
  - Gives the same sequence of numbers
- Pseudorandom numbers
  - Preset sequence of "random" numbers
  - Same sequence generated whenever program run
- To get different random sequences
  - Provide a seed value
    - Like a random starting point in the sequence
    - The same seed will give the same sequence
  - **`srand(seed);`**
    - `<cstdlib>`
    - Used before **`rand()`** to set the seed





## Outline

**fig03\_09.cpp**  
(1 of 2)

```
1 // Fig. 3.9: fig03_09.cpp
2 // Randomizing die-rolling program.
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 #include <iomanip>
10
11 using std::setw;
12
13 // contains prototypes for functions srand and rand
14 #include <cstdlib>
15
16 // main function begins program execution
17 int main()
18 {
19     unsigned seed;
20
21     cout << "Enter seed: ";
22     cin >> seed;
23     srand( seed ); // seed random number generator
24
```

Setting the seed with  
**srand( )**.



## Outline

fig03\_09.cpp  
(2 of 2)

fig03\_09.cpp  
output (1 of 1)

```

25 // loop 10 times
26 for ( int counter = 1; counter <= 10; counter++ ) {
27
28 // pick random number from 1 to 6 and output it
29 cout << setw( 10 ) << ( 1 + rand() % 6 );
30
31 // if counter divisible by 5, begin new line of output
32 if ( counter % 5 == 0 )
33     cout << endl;
34
35 } // end for
36
37 return 0; // indicates s
38
39 } // end main

```

rand() gives the same sequence if it has the same initial seed.

Enter seed: 67

6	1	4	6	2
1	6	1	6	4

Enter seed: 432

4	6	3	1	6
3	1	5	4	2

Enter seed: 67

6	1	4	6	2
1	6	1	6	4

## 3.8 Random Number Generation

- Can use the current time to set the seed
  - No need to explicitly set seed every time
  - `srand( time( 0 ) );`
  - `time( 0 );`
    - `<ctime>`
    - Returns current time in seconds
- General shifting and scaling
  - $Number = shiftingValue + \mathbf{rand}() \% scalingFactor$
  - `shiftingValue` = first number in desired range
  - `scalingFactor` = width of desired range



## 3.9 Example: Game of Chance and Introducing enum

- Enumeration

- Set of integers with identifiers

```
enum typeName {constant1, constant2...};
```

- Constants start at 0 (default), incremented by 1

- Constants need unique names

- Cannot assign integer to enumeration variable

- Must use a previously defined enumeration type

- Example

```
enum Status {CONTINUE, WON, LOST};
```

```
Status enumVar;
```

```
enumVar = WON; // cannot do enumVar = 1
```



## 3.9 Example: Game of Chance and Introducing enum

- Enumeration constants can have preset values

```
enum Months { JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC};
```

  - Starts at 1, increments by 1
- Next: craps simulator
  - Roll two dice
  - 7 or 11 on first throw: player wins
  - 2, 3, or 12 on first throw: player loses
  - 4, 5, 6, 8, 9, 10
    - Value becomes player's "point"
    - Player must roll his point before rolling 7 to win



**fig03\_10.cpp**  
(1 of 5)

```
1 // Fig. 3.10: fig03_10.cpp
2 // Craps.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 // contains function prototypes for functions srand and rand
9 #include <cstdlib>
10
11 #include <ctime> // contains prototype for rand
12
13 int rollDice( void ); // function prototype
14
15 int main()
16 {
17     // enumeration constants represent game status
18     enum Status { CONTINUE, WON, LOST };
19
20     int sum;
21     int myPoint;
22
23     Status gameStatus; // can contain CONTINUE, WON or LOST
24
```

Function to roll 2 dice and return the result as an **int**.

Enumeration to keep track of the current game.



## Outline

**fig03\_10.cpp**  
**(2 of 5)**

```
25 // randomize random number generator using current time
26 srand( time( 0 ) );
27
28 sum = rollDice(); // first
29
30 // determine game status and
31 switch ( sum ) {
32
33     // win on first roll
34     case 7:
35     case 11:
36         gameStatus = WON;
37         break;
38
39     // lose on first roll
40     case 2:
41     case 3:
42     case 12:
43         gameStatus = LOST;
44         break;
45
```

**switch** statement  
determines outcome based on  
die roll.



## Outline

fig03\_10.cpp  
(3 of 5)

```
46     // remember point
47     default:
48         gameStatus = CONTINUE;
49         myPoint = sum;
50         cout << "Point is " << myPoint << endl;
51         break;           // optional
52
53 } // end switch
54
55 // while game not complete ...
56 while ( gameStatus == CONTINUE ) {
57     sum = rollDice();           // roll dice again
58
59     // determine game status
60     if ( sum == myPoint )       // win by making point
61         gameStatus = WON;
62     else
63         if ( sum == 7 )         // lose by rolling 7
64             gameStatus = LOST;
65
66 } // end while
67
```

**fig03\_10.cpp**  
(4 of 5)

```
68 // display won or lost message
69 if ( gameStatus == WON )
70     cout << "Player wins" << endl;
71 else
72     cout << "Player loses" << endl;
73
74 return 0; // indicates successful termination
75
76 } // end main
77
78 // roll dice, calculate sum and c
79 int rollDice( void )
80 {
81     int die1;
82     int die2;
83     int workSum;
84
85     die1 = 1 + rand() % 6; // pick random die1 value
86     die2 = 1 + rand() % 6; // pick random die2 value
87     workSum = die1 + die2; // sum die1 and die2
88
```

Function **rollDice** takes no arguments, so has **void** in the parameter list.



## Outline

```
89 // display results of this roll
90 cout << "Player rolled " << die1 << " + " << die2
91     << " = " << workSum << endl;
92
93 return workSum; // return sum of dice
94
95 } // end function rollDice
```

```
Player rolled 2 + 5 = 7
Player wins

Player rolled 6 + 6 = 12
Player loses

Player rolled 3 + 3 = 6
Point is 6
Player rolled 5 + 3 = 8
Player rolled 4 + 5 = 9
Player rolled 2 + 1 = 3
Player rolled 1 + 5 = 6
Player wins
```

**fig03\_10.cpp**  
**(5 of 5)**

**fig03\_10.cpp**  
**output (1 of 2)**

```
Player rolled 1 + 3 = 4
Point is 4
Player rolled 4 + 6 = 10
Player rolled 2 + 4 = 6
Player rolled 6 + 4 = 10
Player rolled 2 + 3 = 5
Player rolled 2 + 4 = 6
Player rolled 1 + 1 = 2
Player rolled 4 + 4 = 8
Player rolled 4 + 3 = 7
Player loses
```



## Outline



**fig03\_10.cpp**  
**output (2 of 2)**

## 3.10 Storage Classes

- Variables have attributes
  - Have seen name, type, size, value
  - Storage class
    - How long variable exists in memory
  - Scope
    - Where variable can be referenced in program
  - Linkage
    - For multiple-file program (see Ch. 6), which files can use it



## 3.10 Storage Classes

- Automatic storage class
  - Variable created when program enters its block
  - Variable destroyed when program leaves block
  - Only local variables of functions can be automatic
    - Automatic by default
    - keyword **auto** explicitly declares automatic
  - **register** keyword
    - Hint to place variable in high-speed register
    - Good for often-used items (loop counters)
    - Often unnecessary, compiler optimizes
  - Specify either **register** or **auto**, not both
    - **register int counter = 1;**



## 3.10 Storage Classes

- **Static storage class**
  - Variables exist for entire program
    - For functions, name exists for entire program
  - May not be accessible, scope rules still apply (more later)
- **static** keyword
  - Local variables in function
  - Keeps value between function calls
  - Only known in own function
- **extern** keyword
  - Default for global variables/functions
    - Globals: defined outside of a function block
  - Known in any function that comes after it



## 3.11 Scope Rules

- Scope
  - Portion of program where identifier can be used
- File scope
  - Defined outside a function, known in all functions
  - Global variables, function definitions and prototypes
- Function scope
  - Can only be referenced inside defining function
  - Only labels, e.g., identifiers with a colon (**case:**)



## 3.11 Scope Rules

- Block scope
  - Begins at declaration, ends at right brace }
    - Can only be referenced in this range
  - Local variables, function parameters
  - **static** variables still have block scope
    - Storage class separate from scope
- Function-prototype scope
  - Parameter list of prototype
  - Deitel says: Names in prototype optional
    - Compiler ignores
    - But CONRAD SAYS: leave 'em out, lose points (for style)!!!!
  - In a single prototype, name can be used only once




**fig03\_12.cpp**  
 (1 of 5)

```

1  // Fig. 3.12: fig03_12.cpp
2  // A scoping example.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  void useLocal( void );
9  void useStaticLocal( void );
10 void useGlobal( void );
11
12 int x = 1;           // global variable
13
14 int main()
15 {
16     int x = 5;      // local variable to main
17
18     cout << "local x in main's outer scope is " << x << endl;
19
20     { // start new scope
21
22         int x = 7;
23
24         cout << "local x in main's inner scope is " << x << endl;
25
26     } // end new scope

```

Declared outside of function;  
global variable with file  
scope.

Local variable with function  
scope.

Create a new block, giving **x**  
block scope. When the block  
ends, this **x** is destroyed.



## Outline

**fig03\_12.cpp**  
**(2 of 5)**

```
27
28     cout << "local x in main's outer scope is " << x << endl;
29
30     useLocal();           // useLocal has local x
31     useStaticLocal();    // useStaticLocal has static local x
32     useGlobal();         // useGlobal uses global x
33     useLocal();           // useLocal reinitializes its local x
34     useStaticLocal();    // static local x retains its prior value
35     useGlobal();         // global x also retains its value
36
37     cout << "\nlocal x in main is " << x << endl;
38
39     return 0;           // indicates successful termination
40
41 } // end main
42
```



## Outline



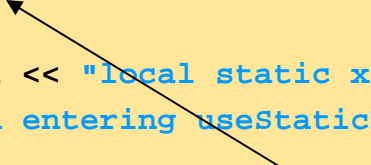
**fig03\_12.cpp**  
**(3 of 5)**

```
43 // useLocal reinitializes local variable x during each call
44 void useLocal( void )
45 {
46     int x = 25; // initialized each time useLocal is called
47
48     cout << endl << "local x is " << x << endl << " on entering useLocal
49         << " on exiting useLocal
50     ++x;
51     cout << "local x is " << x << endl << " on exiting useLocal
52         << " on exiting useLocal
53
54 } // end function useLocal
55
```

Automatic variable (local variable of function). This is destroyed when the function exits, and reinitialized when the function begins.

**fig03\_12.cpp**  
(4 of 5)

```
56 // useStaticLocal initializes static local variable x only the
57 // first time the function is called; value of x is saved
58 // between calls to this function
59 void useStaticLocal( void )
60 {
61     // initialized only first time useStaticLocal is called
62     static int x = 50;
63
64     cout << endl << "local static x is " << x
65         << " on entering useStaticLocal" << endl;
66     ++x;
67     cout << "local static x is " <<
68         << " on exiting useStaticLocal" << endl;
69
70 } // end function useStaticLocal
71
```



Static local variable of function; it is initialized only once, and retains its value between function calls.

```

72 // useGlobal modifies global variable x during each call
73 void useGlobal( void )
74 {
75     cout << endl << "global x is " << x
76         << " on entering useGlobal" << endl;
77     x *= 10;
78     cout << "global x is " << x
79         << " on exiting useGlobal" << endl;
80
81 } // end function useGlobal

```

This function does not declare any variables. It uses the global **x** declared in the beginning of the program.

03\_12.cpp  
of 5)

03\_12.cpp  
output (1 of 2)

```

local x in main's outer scope is 5
local x in main's inner scope is 7
local x in main's outer scope is 5

```

```

local x is 25 on entering useLocal
local x is 26 on exiting useLocal

```

```

local static x is 50 on entering useStaticLocal
local static x is 51 on exiting useStaticLocal

```

```

global x is 1 on entering useGlobal
global x is 10 on exiting useGlobal

```



## Outline

**fig03\_12.cpp**  
**output (2 of 2)**

```
local x is 25 on entering useLocal
```

```
local x is 26 on exiting useLocal
```

```
local static x is 51 on entering useStaticLocal
```

```
local static x is 52 on exiting useStaticLocal
```

```
global x is 10 on entering useGlobal
```

```
global x is 100 on exiting useGlobal
```

```
local x in main is 5
```