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);
  – 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
    • \texttt{sqrt( 4 );}
  – Variables
    • \texttt{sqrt( x );}
  – Expressions
    • \texttt{sqrt( sqrt( x ) );}
    • \texttt{sqrt( 3 - 6 * x );}
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>ceil( $x$)</td>
<td>rounds $x$ to the smallest integer not less than $x$</td>
<td>ceil( 9.2 ) is 10.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ceil( -9.8 ) is -9.0</td>
</tr>
<tr>
<td>cos( $x$)</td>
<td>trigonometric cosine of $x$ ($x$ in radians)</td>
<td>cos( 0.0 ) is 1.0</td>
</tr>
<tr>
<td>exp( $x$)</td>
<td>exponential function $e^x$</td>
<td>exp( 1.0 ) is 2.71828</td>
</tr>
<tr>
<td></td>
<td></td>
<td>exp( 2.0 ) is 7.38906</td>
</tr>
<tr>
<td>fabs( $x$)</td>
<td>absolute value of $x$</td>
<td>fabs( 5.1 ) is 5.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>fabs( 0.0 ) is 0.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>fabs( -8.76 ) is 8.76</td>
</tr>
<tr>
<td>floor( $x$)</td>
<td>rounds $x$ to the largest integer not greater than $x$</td>
<td>floor( 9.2 ) is 9.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>floor( -9.8 ) is -10.0</td>
</tr>
<tr>
<td>fmod( $x$, $y$)</td>
<td>remainder of $x/y$ as a floating-point number</td>
<td>fmod( 13.657 , 2.333 ) is 1.992</td>
</tr>
<tr>
<td>log( $x$)</td>
<td>natural logarithm of $x$ (base $e$)</td>
<td>log( 2.718282 ) is 1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>log( 7.389056 ) is 2.0</td>
</tr>
<tr>
<td>log10( $x$)</td>
<td>logarithm of $x$ (base 10)</td>
<td>log10( 10.0 ) is 1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>log10( 100.0 ) is 2.0</td>
</tr>
<tr>
<td>pow( $x$, $y$)</td>
<td>$x$ raised to power $y$ ($x^y$)</td>
<td>pow( 2 , 7 ) is 128</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pow( 9 , .5 ) is 3</td>
</tr>
<tr>
<td>sin( $x$)</td>
<td>trigonometric sine of $x$ ($x$ in radians)</td>
<td>sin( 0.0 ) is 0</td>
</tr>
<tr>
<td>sqrt( $x$)</td>
<td>square root of $x$</td>
<td>sqrt( 900.0 ) is 30.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sqrt( 9.0 ) is 3.0</td>
</tr>
<tr>
<td>tan( $x$)</td>
<td>trigonometric tangent of $x$ ($x$ in radians)</td>
<td>tan( 0.0 ) is 0</td>
</tr>
</tbody>
</table>
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
// Fig. 3.3: fig03_03.cpp
// Creating and using a programmer-defined function.
#include <iostream>

using std::cout;
using std::endl;

int square( int ); // function prototype

int main()
{
    // loop 10 times and calculate and output
    // square of x each time
    for ( int x = 1; x <= 10; x++ )
        cout << square( x ) << " "; // function call

    cout << endl;

    return 0; // indicates successful termination
}

// square function definition returns square of an integer
int square( int y ) // y is a copy of argument to function
{
    return y * y; // returns square of y as an int
} // end function square

1  4  9  16  25  36  49  64  81  100
// Fig. 3.4: fig03_04.cpp
// Finding the maximum of three floating-point numbers.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

double maximum( double, double, double ); // function prototype

int main()
{
    double number1;
    double number2;
    double number3;

    cout << "Enter three floating-point numbers: ";
    cin >> number1 >> number2 >> number3;

    // number1, number2 and number3 are arguments to
    // the maximum function call
    cout << "Maximum is: "
         << maximum( number1, number2, number3 ) << endl;

    return 0; // indicates successful termination
}

Enter three floating-point numbers: 99.32 37.3 27.1928
Maximum is: 99.32
Comma separated list for multiple parameters.

double maximum(double x, double y, double z)
{
    double max = x;   // assume x is largest

    if ( y > max )  // if y is larger,
        max = y;       // assign y to max

    if ( z > max )    // if z is larger,
        max = z;       // assign z to max

    return max;       // max is largest value
}

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
  – \texttt{int square( int );}
    • Function takes an \texttt{int} and returns an \texttt{int}
  – Explained in more detail later

• Calling/invoking a function
  – \texttt{square(x);} 
  – Parentheses an operator used to call function
    • Pass argument \texttt{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 (\texttt{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
    \[
    \text{double maximum( double } x, \text{ double } y, \text{ double } z); \]
  – Another legal way to do a function prototype (but \textbf{NEVER DO THIS !!!!!})
    \[
    \text{double maximum( double } , \text{ double } , \text{ double } ); \]
  – Definition
    \[
    \text{double maximum( double } x, \text{ double } y, \text{ 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);`

• 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

<table>
<thead>
<tr>
<th>Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>long double</code></td>
</tr>
<tr>
<td><code>double</code></td>
</tr>
<tr>
<td><code>float</code></td>
</tr>
<tr>
<td><code>unsigned long int</code></td>
</tr>
<tr>
<td><code>long int</code></td>
</tr>
<tr>
<td><code>unsigned int</code></td>
</tr>
<tr>
<td><code>int</code></td>
</tr>
<tr>
<td><code>unsigned short int</code></td>
</tr>
<tr>
<td><code>short int</code></td>
</tr>
<tr>
<td><code>unsigned char</code></td>
</tr>
<tr>
<td><code>char</code></td>
</tr>
<tr>
<td><code>bool</code></td>
</tr>
</tbody>
</table>

- `bool` becomes 0 if `false`, and 1 if `true`.
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>**)
  - \( i = \text{rand}(); \)
  - 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
  - \( i = \text{rand}() \% 6 + 1; \)
    - "Rand() \% 6" generates a number between 0 and 5 (scaling)
    - "+ 1" makes the range 1 to 6 (shift)

- Next: program to roll dice
// Fig. 3.7: fig03_07.cpp
// Shifted, scaled integers produced by 1 + rand() % 6.
#include <iostream>

using std::cout;
using std::endl;

#include <iomanip>
using std::setw;

#include <cstdlib> // contains function prototype for rand

int main()
{
    // loop 20 times
    for ( int counter = 1; counter <= 20; counter++ ) {
        // pick random number from 1 to 6 and output it
        cout << setw( 10 ) << ( 1 + rand() % 6 );

        // if counter divisible by 5, begin new line of output
        if ( counter % 5 == 0 )
            cout << endl;
    } // end for structure
    return 0; // indicates successful termination
} // end main

Output of rand() scaled and shifted to be a number between 1 and 6.
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
// Fig. 3.8: fig03_08.cpp
// Roll a six-sided die 6000 times.
#include <iostream>

using std::cout;
using std::endl;

#include <iomanip>
using std::setw;

#include <cstdlib>   // contains function prototype for rand

int main()
{
    int frequency1 = 0;
    int frequency2 = 0;
    int frequency3 = 0;
    int frequency4 = 0;
    int frequency5 = 0;
    int frequency6 = 0;
    int face;    // represents one roll of the die
// loop 6000 times and summarize results
for ( int roll = 1; roll <= 6000; roll++ ) {
    face = 1 + rand() % 6;  // random number from 1 to 6

    // determine face value and increment appropriate counter
    switch ( face ) {
        case 1:          // rolled 1
            ++frequency1;
            break;

        case 2:          // rolled 2
            ++frequency2;
            break;

        case 3:         // rolled 3
            ++frequency3;
            break;

        case 4:          // rolled 4
            ++frequency4;
            break;

        case 5:         // rolled 5
            ++frequency5;
            break;
    }
```cpp
    case 6:         // rolled 6
        ++frequency6;
        break;

    default:         // invalid value
        cout << "Program should never get here!";

} // end switch

} // end for

// display results in tabular format
    cout << "Face" << setw( 13 ) << "Frequency" << endl;
    cout << "\n   1" << setw( 13 ) << frequency1 << endl;
    cout << "\n   2" << setw( 13 ) << frequency2 << endl;
    cout << "\n   3" << setw( 13 ) << frequency3 << endl;
    cout << "\n   4" << setw( 13 ) << frequency4 << endl;
    cout << "\n   5" << setw( 13 ) << frequency5 << endl;
    cout << "\n   6" << setw( 13 ) << frequency6 << endl;

return 0; // indicates successful termination
```

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

<table>
<thead>
<tr>
<th>Face</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1003</td>
</tr>
<tr>
<td>2</td>
<td>1017</td>
</tr>
<tr>
<td>3</td>
<td>983</td>
</tr>
<tr>
<td>4</td>
<td>994</td>
</tr>
<tr>
<td>5</td>
<td>1004</td>
</tr>
<tr>
<td>6</td>
<td>999</td>
</tr>
</tbody>
</table>
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
// Fig. 3.9: fig03_09.cpp
// Randomizing die-rolling program.
#include <iostream>

#include <iomanip>

#include <cstdlib>

int main()
{
    unsigned seed;

    cout << "Enter seed: ";
    cin >> seed;
    srand( seed );  // seed random number generator
}

Setting the seed with srand().
// loop 10 times
for ( int counter = 1; counter <= 10; counter++ ) {
    // pick random number from 1 to 6 and output it
    cout << setw( 10 ) << ( 1 + rand() % 6 );
    // if counter divisible by 5, begin new line of output
    if ( counter % 5 == 0 )
        cout << endl;
} // end for

return 0; // indicates successful termination
} // end main

rand() gives the same sequence if it has the same initial seed.
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 + 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
    ```java
    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
  ```java
  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
  ```java
class 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
// Fig. 3.10: fig03_10.cpp
// Craps.
#include <iostream>

using std::cout;
using std::endl;

// contains function prototypes for functions srand and rand
#include <cstdlib>

#include <ctime>   // contains prototype for function time

int rollDice( void );  // function prototype

int main()
{
    // enumeration constants represent game status
    enum Status { CONTINUE, WON, LOST };

    int sum;
    int myPoint;

    Status gameStatus;  // can contain CONTINUE, WON or LOST

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

    Enumeration to keep track of the current game.
// randomize random number generator using current time
srand( time( 0 ) );

sum = rollDice(); // first roll of dice

// determine game status and point based on sum of dice
switch ( sum ) {
    // win on first roll
    case 7:
    case 11:
        gameStatus = WON;
        break;
    
    // lose on first roll
    case 2:
    case 3:
    case 12:
        gameStatus = LOST;
        break;
}

switch statement determines outcome based on die roll.
// remember point
default:
    gameStatus = CONTINUE;
    myPoint = sum;
    cout << "Point is " << myPoint << endl;
    break;               // optional
}

// while game not complete ...
while ( gameStatus == CONTINUE ) {
    sum = rollDice();          // roll dice again

    // determine game status
    if ( sum == myPoint )       // win by making point
        gameStatus = WON;
    else
        if ( sum == 7 )          // lose by rolling 7
            gameStatus = LOST;

} // end while
// display won or lost message
if ( gameStatus == WON )
    cout << "Player wins" << endl;
else
    cout << "Player loses" << endl;

return 0; // indicates successful termination
}

// roll dice, calculate sum and display results
int rollDice( void )
{
    int die1;
    int die2;
    int workSum;

    die1 = 1 + rand() % 6; // pick random die1 value
    die2 = 1 + rand() % 6; // pick random die2 value
    workSum = die1 + die2; // sum die1 and die2

    // roll dice, calculate sum and display results
    int rollDice( void )
// display results of this roll
cout << "Player rolled " << die1 << " + " << die2
    << " = " << workSum << endl;

return workSum;  // return sum of dice

} // 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
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
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
// Fig. 3.12: fig03_12.cpp
// A scoping example.
#include <iostream>

using std::cout;
using std::endl;

void useLocal( void );        // function prototype
void useStaticLocal( void );  // function prototype
void useGlobal( void );      // function prototype

int x = 1;     // global variable

int main()
{
    int x = 5;   // local variable to main

    cout << "local x in main's outer scope is " << x << endl;

    { // start new scope
        int x = 7;
        cout << "local x in main's inner scope is " << x << endl;
    } // end new scope
cout << "local x in main's outer scope is " << x << endl;

useLocal();      // useLocal has local x
useStaticLocal(); // useStaticLocal has static local x
useGlobal();     // useGlobal uses global x
useLocal();      // useLocal reinitializes its local x
useStaticLocal(); // static local x retains its prior value
useGlobal();     // global x also retains its value

cout << "\nlocal x in main is " << x << endl;

return 0;       // indicates successful termination

} // end main
void useLocal( void )
{
    int x = 25;  // initialized each time useLocal is called
    cout << endl << "local x is " << x << " on entering useLocal" << endl;
    ++x;
    cout << "local x is " << x << " on exiting useLocal" << endl;
} // end function useLocal
useStaticLocal initializes static local variable x only the first time the function is called; value of x is saved between calls to this function.

```cpp
void useStaticLocal( void )
{
    // initialized only first time useStaticLocal is called
    static int x = 50;

    cout << endl << "local static x is " << x
        << " on entering useStaticLocal" << endl;
    ++x;
    cout << "local static x is " << x
        << " on exiting useStaticLocal" << endl;
}
```

Static local variable of function; it is initialized only once, and retains its value between function calls.
```
// useGlobal modifies global variable x during each call
void useGlobal( void )
{
  cout << endl << "global x is " << x
       << " on entering useGlobal" << endl;
  x *= 10;
  cout << "global x is " << x
       << " on exiting useGlobal" << endl;
} // end function useGlobal
```

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

---

**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**
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