**LaGuardia Community College – Last Update**

08

**Fall**

Part 7: Local Variables and Recursive Functions

Dr. Andi Toce

Lecture Notes for MAC 101 (Introduction to Computer Science)

Last updated / viewed: April 21, 15

Table of Contents

1. Local and Global Variables 2

2. Recursive Functions 3

3. Pre-Defined Functions 6

# Local and Global Variables

A variable is local if it is declared inside a function. A global variable is declared outside of all functions including main. Global variables are typically declared at the very beginning of the program. Local variables are better choices unless there is a specific need for a global variable shared by multiple functions.

**Example**:

|  |  |
| --- | --- |
| LocalGlobalVariables.cpp | Output |
| #include <iostream>  using namespace std;  // Testing local and global variables  int i = 1;  void firstFunction();  void secondFunction();  int main() {  int i = 2;  cout << "Inside main(): i=" << i << endl;  firstFunction();  secondFunction();  return 0;  }  void firstFunction(){  int i = 3;  cout << "Inside firstFunction(): i=" << i << endl;  }  void secondFunction(){  cout << "Inside secondFunction(): i=" << i << endl;  } | Inside main(): i=2  Inside firstFunction(): i=3  Inside secondFunction(): i=1 |

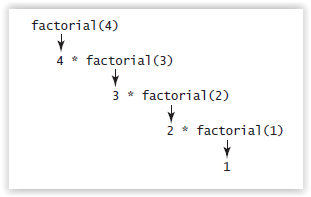
# Recursive Functions

Recursion is a technique when a function calls itself. To avoid infinite loops each recursive function has a mechanism of ending the recursion. Each recursive function has a base case.

**Example:** Recallthe program that defines and tests a factorial function. We change the factorial function to a recursive function.

|  |  |
| --- | --- |
| FactorialFunction.cpp | RecursiveFactorial.cpp |
| #include <iostream>  using namespace std;  int factorial(int x);  int main() {  int number;  cout << "Enter an integer for the desired factorial: ";  cin >> number;  cout << number << "! = " << factorial(number) << endl;  return 0;  } // end main() function  int factorial(int x) {  int f=1;  for(int i=2; i<=x; i++){  f \*= i;  }  return f;  } | **#include** <iostream>  **using** **namespace** std;  **int** **factorial**(**int** x);  **int** **main**() {  **int** number;  cout << "Enter an integer for the desired factorial: ";  cin >> number;  cout << number << "! = " << factorial(number) << endl;  **return** 0;  } // end main() function  **int** **factorial**(**int** x) {  **if** (x <= 1)  **return** 1;  **else**  **return** x \* factorial(x-1);  } |

During recursion a special area of memory called the stack is utilized to store the information as the multiple function calls are executed. Values that are stored in the stack follow the last in – first out (LIFO) order. The following picture illustrates the factorial recursive process.



**Question:** Which of the two approaches (iterative or recursive) is better in this case?

**Try now:** Write function *sum(int)* that takes as an argument a positive integer n and returns the sum of the first n positive integers. Use a while loop. Then write a recursive function that performs the same task.

Sample output:

Enter a Positive Integer: 5

The sum of the first 5 integers is: 15

**Questions:** Try using n=100,000 for each of the programs. What do you notice? Which version is a better choice for this example? Can we further optimize the code?

**Additional Example:** Fibonacci Series

|  |  |
| --- | --- |
| FibonacciIterative.cpp | FibonacciRecursive.cpp |
| #include <iostream>  using namespace std;  int fibonacci(int x);  int main() {  int number;  cout << "Enter the Fibonacci Term: ";  cin >> number;  cout << "The " << number << "-th Fibonacci Number is: " << fibonacci(number) << endl;  return 0;  } // end main() function  int fibonacci(int x) {  int i = 0, j = 1, k = 1, sum=0;  while(i < x-1){  cout << j << endl;  sum=j+k;  j=k;  k=sum;  i++;  }  return j;  } | **#include** <iostream>  **using** **namespace** std;  **int** **fibonacci**(**int** x);  **int** **main**() {  **int** number;  cout << "Enter the Fibonacci Term: ";  cin >> number;  cout << "The " << number << "-th Fibonacci Number is: " << fibonacci(number) << **endl**;  **return** 0;  } // end main() function  **int** **fibonacci**(**int** x) {  **if** (x==1 || x==2)  **return** 1;  **else**  **return** fibonacci(x-1) +fibonacci(x-2);  } |

**Try now:** Write a recursive function ***power(base,exponent)*** which when invoked returns base^exponent. Use only multiplication for this function.

Sample output

Enter the Base: 5

Enter the Exponent: 3

5^3 = 125

**Example:** Print the prime factorization of a given number.

|  |  |
| --- | --- |
| PrimeFactorizationRecursive.cpp | Output |
| #include <iostream>  #include <cmath>  using namespace std;  void get\_divisors(int n);  int main() {  int n = 0;  cout << "Enter a number and press ENTER: ";  cin >> n;  get\_divisors(n);  cout << endl;  return 0;  }// end main  // Get divisors function  // This function prints all the divisors of n,  // by finding the lowest divisor, i, and then  // rerunning itself on n/i, the remaining quotient.  void get\_divisors(int n) {  int i;  double sqrt\_of\_n = sqrt(n);  for (i = 2; i <= sqrt\_of\_n; i++)  if (n % i == 0) { // If i divides n evenly,  cout<<i<<",";  get\_divisors(n / i);  return;  }  // If no divisor is found, then n is prime;  // Print n and make no further calls.  cout << n;  }// end get\_divisors() | Enter a number and press ENTER: 140  2,2,5,7 |

**Homework:** (Exercise 4.3.3 in your book) Modify the program above so that it uses a *non-recursive* solution. You will end up having to write more code. (Hint: To make the job easier, write two functions: *get\_all\_divisors* and *get\_lowest\_divisor*. The **main** function should call *get\_all\_divisors*, which in turn has a loop: *get\_all\_divisors* calls *get\_lowest\_divisor* repeatedly, each time replacing n with n/i, where i is the divisor that was found. If n itself is returned, then the number is prime, and the loop should stop.

# Pre-Defined Functions

A number of pre-defined functions are already available for the programmer to use. These functions are grouped in collections called libraries. In order for the user to have access to these functions the program should include the given library. <cmath> is an example of a library. The following program illustrates some of its functions.

|  |  |
| --- | --- |
| CmathLibrary.cpp | Output |
| #include <iostream>  #include <cmath>  #include <iomanip>  using namespace std;  int main() {  cout << fixed << setprecision(2);  cout << "sqrt(12.0) = " << sqrt(12.0) << endl;  cout << "exp(1.0) = " << setprecision(3) << exp(1.0) << endl;  cout << "log(8.0) = " << log(8.32) << endl;  cout << "log(exp(8.23)) = " << log(exp(8.23)) << endl;  cout << "pow(2,3) = " << pow(2,3) << endl;  cout << "cos(90) = " << setprecision(15) << cos(90) << endl;  return 0;  } | sqrt(12.0) = 3.46  exp(1.0) = 2.718  log(8.0) = 2.119  log(exp(8.23)) = 8.230  pow(2,3) = 8.000  cos(90) = -0.448073616129170 |

Random number generator.

|  |  |
| --- | --- |
| RandomDiceSimulator.cpp | Output |
| #include <iostream>  #include <cmath>  #include <cstdlib>  #include <ctime>  using namespace std;  int main() {  int n, i;  int r;  srand(time(NULL)); // Set seed for random numbers.  cout << "Enter number of dice to roll: ";  cin >> n;  for (i = 1; i <= n; i++) {  r = rand() % 6 + 1; // Get a number 1 to 6  cout << r << " ";  }  return 0;  } | Enter number of dice to roll: 10  3 5 1 2 3 5 5 2 2 3 |

**Practice:** Modify RandomDiceSimulator.cpp to display a frequency distribution of the values and graph the results. Sample output.

Enter number of dice to roll: 20

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

Value Freq Graph

1 1 \*

2 4 \*\*\*\*

3 7 \*\*\*\*\*\*\*

4 5 \*\*\*\*\*

5 2 \*

6 1 \*