MECH215
Lecture Notes 5

Recursion

Recursion is a very powerful concept in programming. It is a coding technique which has a function calls itself. The concept is to solve a portion of the problem and then request that another version of the function solve the rest. This process is allows for the stepwise solution to a problem. Here is a very trivial but simple example which illustrates the main concepts involved in recursive function calls. It is taken from a very nice book on C++ written by Stephen Prata. It has been modified for style.

```
// Author:  Stephen Prata and John Peach
// countDown.cpp
// example program illustrating use of a contrived example of
// function recursion taken from the book C++ Primer Plus,
// 3rd Edition by Stephen Prata, Sams Publishing, 1998, p. 281

#include <iostream>
#include <string>

using namespace std;

void countDown(int n);

int main() {

    countDown(5);

    return 0;
}

void countDown(int n) {

    cout << "Counting down ... " << n << endl;
    if (n > 0) {
        countDown(n-1); // make a recursive call
    }
    cout << "Welcome back to version " << n << " of countDown. " <<
endl;
}
```

The program output is:

```
Counting down ... 5
Counting down ... 4
Counting down ... 3
Counting down ... 2
Counting down ... 1
Counting down ... 0
Welcome back to version 0 of countDown.
Welcome back to version 1 of countDown.
Welcome back to version 2 of countDown.
Welcome back to version 3 of countDown.
Welcome back to version 4 of countDown.
Welcome back to version 5 of countDown.
```

Let us trace the sequence of calls to function countDown starting from the initial call from main(). Every call to function countDown() causes a new stack frame to be allocated to the function. The functions value arguments and any local arguments are copied into this stack frame. When a function either executes an explicit return statement or when it reaches its closing }, the function returns to whoever called it. Program control then resumes with the caller continuing from where the caller left off.

Here is a pictorial representation of what the stack frame would look like when main calls countDown(5):

frame allocated for invocation of countDown(5)

```
---------------
n = 5               stack frame allocated for countDown(5)
---------------
```

when countDown(5) begins execution it prints the "Counting down ... " << n << endl; message and then makes a recursive call to countDown passing value argument of 4. A new stack frame is allocated and the function countDown begins execution from its first line of count. The stack frame looks like:

```
---------------
n = 4               stack frame allocated for countDown(4)
---------------
n = 5               stack frame allocated for countDown(5)
---------------
```

countDown(4) prints its "counting down... 4" message and executes its conditional if statement, which results in another recursive call to countDown with parameter n=3. A new stack frame is allocated:

```
---------------
n = 3              stack frame allocated for countDown(3)
---------------
n = 4              stack frame allocated for countDown(4)
---------------
n = 5              stack frame allocated for countDown(5)
---------------
```

Again, countDown(3) begins execution and prints its starting message and then makes a call to countDown(2), a new stack frame is allocated and countDown(2) begins execution:

```
---------------
n = 2              stack frame allocated for countDown(2)
---------------
n = 3              stack frame allocated for countDown(3)
---------------
n = 4              stack frame allocated for countDown(4)
---------------
n = 5              stack frame allocated for countDown(5)
---------------
```

countDown(2) prints its first message and then makes yet another call to countDown with value n= 1. A new stack frame is allocated and the function begins execution:

```
---------------
n = 1              stack frame allocated for countDown(1)
---------------
n = 2              stack frame allocated for countDown(2)
---------------
n = 3              stack frame allocated for countDown(3)
---------------
n = 4              stack frame allocated for countDown(4)
---------------
n = 5              stack frame allocated for countDown(5)
---------------
```

countDown(1) prints its counting down ... 1 message and then calls countDown with n = 0:

```
----------------
n = 0              stack frame allocated for countDown(0)
----------------
n = 1              stack frame allocated for countDown(1)
----------------
n = 2              stack frame allocated for countDown(2)
----------------
n = 3              stack frame allocated for countDown(3)
----------------
n = 4              stack frame allocated for countDown(4)
----------------
n = 5              stack frame allocated for countDown(5)
----------------
```

countDown(0) prints its Counting down ... 0 message, since it's local value of n (0) is not greater than 0, the if statement is false and countDown(0) prints its Welcome back to version 0 message and then reaches its closing } which causes a return (down the stack frame) to countDown(1). When a function return, its stack frame is DEALLOCATED. The stack frame after countDown(0) returns is:

```
----------------
n = 1              stack frame allocated for countDown(1)
----------------
n = 2              stack frame allocated for countDown(2)
----------------
n = 3              stack frame allocated for countDown(3)
----------------
n = 4              stack frame allocated for countDown(4)
----------------
n = 5              stack frame allocated for countDown(5)
----------------
```

Program execution now resumes with countDown(1) continuing from after the point where it made a recursive call. It prints its "Welcome back to version 1" and then returns. It's stack frame is deallocated:

```
----------------
n = 2              stack frame allocated for countDown(2)
----------------
n = 3              stack frame allocated for countDown(3)
----------------
n = 4              stack frame allocated for countDown(4)
----------------
n = 5              stack frame allocated for countDown(5)
----------------
```

Control returns to function countDown(2). It uses its local value of n in the current top of stack frames when it returns to the "Welcome back to version " << n portion of code, then it returns... and its stack frame is deallocated:

```
----------------
n = 3               stack frame allocated for countDown(3)
----------------
n = 4               stack frame allocated for countDown(4)
----------------
n = 5               stack frame allocated for countDown(5)
----------------
```

Eventually, control returns to countDown(5), it prints its welcome back to version 5 and control then returns to main() (since main intially called countDown with n = 5). At this point, the program stops.

Problems which have recursive solutions generally have the two following properties:

1. There exists a 'base case' (or terminating case) which does not make a recursive call.
2. In the other cases, it is possible to consider the problem which is similar to the original problem but with simpler (smaller) arguments and a recursive call is made.

In general, we can express recursive solutions along the form of:

```
if (some terminating condition is satisfied)
  return;
else
  make a recursive call with "simpler" arguments;
```

We will now solve the problem of calculating a factorial using a recursive function:

```
// Author: Ted Obuchowicz and John Peach
// factorial.cpp
// example program illustrating use of recursion to
// compute factorial
// n ! = 1 if n = 0
//     = n * (n-1)! if n >= 1

#include <iostream>
#include <string>

using namespace std;

int factorial(int n) {
    if (n == 0) {
        // base case
        return 1;
```

```
    } else {
        // calculate the factorial as n * (n-1)!
        return n * factorial(n-1);
    }
}

int main() {

    int n;

    // Get a positive number
    do {
        cout << "Enter a positive integer: ";
        cin >> n;
    } while(n<0);

    // Print the factorial
    cout << n << " ! = " << factorial(n) << endl;

    return 0;
}
```

The following was taken from:
http://www.cs.princeton.edu/~lworthin/126/precepts/recursion.html

You can trace through a recursive function by repeatedly 'pasting' in its code whenever it calls itself. Example (factorial function):

```
int f (int n) {
   if (n == 0) return 1;
   return n * f(n - 1);
}
```

Let's trace through the function with the pasting technique for n=4:

```
f(4) {
    if (4 == 0) return 1;   // n is 4 here
    return 4 * f(3) {
        if (3 == 0) return 1;  // we just 'pasted' the code here
        return 3 * f(2) {
            if (2 == 0) return 1;  //n is 2 here
            return 2 * f(1) {
                if (1 == 0) return 1;
                return 1 * f(0) {
                    if (0 == 0) return 1;
```

Now what? Well, we have finally reached the base case. We do not need to 'paste' again, because we do not get to the line that has a recursive call. We simply return 1. You can see why base cases are so important - without them, we would never finish 'pasting.' We have not finished yet, though...

f(0) just finished its execution. There is no mystery about what happens next. Whenever any function finishes (returns), its return value (if any) is returned to the calling function and execution resumes there. Make sure you understand this as it is critical.

So, the value 1 is returned. In the function f(1), we were executing the line "return 1 * f(0)" when f(0) was called. So, we continue: f(1) will return 1 * 1 to its current function: f(2). Then, f(2) will return 2 * f(1), which is 2 * 1. Next, this value, 2, is returned to f(3), so f(3) finishes by executing the line: 3 * f(2). The number returned by f(2) is 2. Thus, 3 * 2 is returned by f(3), and we're done - with the correct result, 3.

Our next example uses recursion to calculate the n^{th} Fibonacci number in the famed sequence:

```
// Author: Ted Obuchowicz and John Peach
// fibonacci.cpp
// Use of recursion to find the n'th Fibonacci number

#include <iostream>
#include <string>
using namespace std;

int fibonacci(int n) {
    if ( n <= 2) {
        return 1;
    } else {
        return fibonacci(n-1) + fibonacci(n-2);
    }
}

int main() {

    int number;

    do {
        cout << "Enter the number of the sequence you want: ";
        cin >> number;
    } while(number < 1);

    cout << "The " << number << " th in the Fibonacci sequence is  "
<< fibonacci(number) << endl;

    return 0;
}
```

The Tower Of Hanoi

The following description is modified from Wikipedia:

The Tower of Hanoi is a mathematical puzzle. It consists of three rods and a number of disks of different sizes which can slide onto any of three rods. The puzzle starts with the discs in a neat stack in ascending order of size on one rod, the smallest at the top, thus making a conical shape.

The objective of the puzzle is to move the entire stack to another rod, obeying the following rules:

1. Only one disc may be moved at a time.
2. Each move consists of taking the upper disc from one of the rods and sliding it onto another rod, on top of the other discs that may already be present on that rod.
3. No disk may be placed on top of a smaller disc.

Recursive solution:

A key to solving this puzzle is to recognize that it can be solved by breaking the problem down into a collection of smaller problems and further breaking those problems down into even smaller problems until a solution is reached. The following procedure demonstrates this approach.

1. Label the pegs A, B, C — these labels may move at different steps
2. Let n be the total number of discs
3. Number the discs from 1 (smallest, topmost) to n (largest, bottom most)

To move n discs from peg A to peg C:

1. Move n−1 discs from A to B. This leaves disc n alone on peg A
2. Move disc n from A to C
3. Move n−1 discs from B to C so they sit on disc n

The above is a recursive algorithm: to carry out steps 1 and 3, apply the same algorithm again for n − 1. The entire procedure is a finite number of steps, since at some point the algorithm will be required for n = 1. This step, moving a single disc from peg A to peg B, is trivial.

For 3 discs, the step-by-step solution looks like:

```
   *
  ***
*****


  ***
*****          *
From peg 1 to peg 3
```

```
*****  ***    *
From peg 1 to peg 2


        *
*****  ***
From peg 3 to peg 2


        *
       ***  *****
From peg 1 to peg 3



  *    ***  *****
From peg 2 to peg 1


             ***
  *         *****
From peg 2 to peg 3

              *
             ***
            *****
From peg 1 to peg 3
```

Here is the source code for the Tower of Hanoi.

```
// Author: Ted Obuchowicz and John Peach
// hanoiTower.cpp
// example program illustrating use of recursion to
// solve the Towers of Hanoi problem

#include <iostream>
#include <string>

using namespace std;

unsigned long int counter = 0;   // how many times was the function
called
```

```
void hanoiTower(int numberOfDisk, int from, int to, int temp) {

    if ( numberOfDisk > 0) {
        counter++;
        hanoiTower(numberOfDisk - 1, from, temp, to);
        cout << "From peg " << from << " to peg " << to <<  endl;
        hanoiTower(numberOfDisk - 1, temp, to, from);
    }
    // Notice that the base case does not do anything
}

int main() {
    int numberOfDisk;           // number of disk

    // Get the number of disks
    do {
        cout << "Enter number of disks: ";
        cin >> numberOfDisk;
    } while(numberOfDisk < 1);

    hanoiTower(numberOfDisk, 1,3,2);
    cout << "it took " << counter << " moves " << endl;

    return 0;
}
```

The output is:

```
Enter number of disks: 3
From peg 1 to peg 3
From peg 1 to peg 2
From peg 3 to peg 2
From peg 1 to peg 3
From peg 2 to peg 1
From peg 2 to peg 3
From peg 1 to peg 3
It took 7 moves
```

Determining If A Matrix Is Symmetric Using Recursion

A symmetric matrix is a square matrix where the transpose of the matrix is equal to the matrix. That is $A = A^T$ or at the element level $a_{ij} = a_{ji}$;

```
// Author: Ted Obuchowicz and John Peach
// squareMatrix.cpp
// example program illustrating use of recursion to
// determine if a matrix is symmetric or not developed
// by C. Taillefer on whiteboard

#include <iostream>
#include <string>

using namespace std;

int symmetric3x3(int a[3][3], int size) {
    int i = size-1;
    int j = size-1;

    if ( size < 1) {
        return true;
    } else {
        for(int k = 0 ; k < size-1 ; k++) {
            // print out elements being compared
            cout << "Comparing elements a[" << i << "][" << j-k-1
             << "]" << " and [" << i-k-1 << "][" << j << "]" << endl;
            if ( a[i][j-k-1] != a[i-k-1][j] ) {
                return false;
            }
        }
    }

    return (symmetric3x3(a, size-1));
}

int symmetric5x5(int a[5][5], int size) {
    int i = size-1;
    int j = size-1;

    if ( size < 1) {
        return true;
    } else {
        for(int k = 0 ; k < size-1 ; k++) {
            // print out elements being compared
            // clean up in final version
```

```
            cout << "Comparing elements a[" << i << "][" << j-k-1
             << "]" << " and [" << i-k-1 << "][" << j << "]" << endl;
            if ( a[i][j-k-1] != a[i-k-1][j] ) {
                return false;
            }
        }
    }

    return symmetric5x5(a, size-1);
}

int main() {

    int matrix3x3[3][3] = { {0,1,3},{1,2,5},{3,5,6} };
    int matrix5x5[5][5] = { {0, 1, 2,  3,  4},
                            {1, 0, 6,  7,  8},
                            {2, 6, 0,  9,  10},
                            {3, 7, 9,  0,  11},
                            {4, 8, 10, 11, 0} };

    // Test the 3 x 3 matrix
    if (symmetric3x3(matrix3x3, 3)){
        cout << "symmetric" << endl;
    } else {
        cout << "asymmetric" << endl;
    }

    // Test the 5 x 5 matrix
    if (symmetric5x5(matrix5x5, 5)){
        cout << "symmetric" << endl;
    } else {
        cout << "asymmetric" << endl;
    }

    return 0;
}
```

This is the output:

```
Comparing elements a[2][1] and [1][2]
Comparing elements a[2][0] and [0][2]
Comparing elements a[1][0] and [0][1]
symmetric
Comparing elements a[4][3] and [3][4]
Comparing elements a[4][2] and [2][4]
Comparing elements a[4][1] and [1][4]
Comparing elements a[4][0] and [0][4]
```

```
Comparing elements a[3][2] and [2][3]
Comparing elements a[3][1] and [1][3]
Comparing elements a[3][0] and [0][3]
Comparing elements a[2][1] and [1][2]
Comparing elements a[2][0] and [0][2]
Comparing elements a[1][0] and [0][1]
symmetric
```