Lecture 7.1

Recursion

What is Recursion?

- Recursion is defined as a function calling itself.
- It is in some ways similar to a loop because it repeats the same code
- It requires passing in the looping variable from one function call to another.
- Problem solving by reducing the problem to a smaller versions of itself.

Factorial Example

- Problem:
  - Mathematically, the factorial of an integer is defined as:-
  - 0! = 1
  - n! = n * (n- 1)!, if n>0

1! = 1
2! = 2 * 1
3! = 3 * 2 * 1
4! = 4 * 3 * 2 * 1

Factorial Example

- So

  4 ! = 4 * (4- 1)!
  3 ! = 3 * (3- 1)!
  2! = 2 * (2- 1)!
  1! = 1 * (1- 1)!
  0! = 1
- This gives us a recursive definition (smaller versions of itself)

Recursive definition

- Recursive functions must have
  - **Base case** (0! = 1)
  - **General case** (3!, 2!..etc i.e. n! = (n- 1)!)
  - Recursive function must have one or more base case and one general case.
  - Base case stops the recursion

Recursive definition

- Recursive algorithm
  - Algorithm that finds the solution to a given problem by reducing the problem to a smaller version of itself.
- Recursive function
  - A function that calls itself

Recursive function

```
int factorial ( int n ) {
   if (n > 1)
        return n * factorial ( n – 1 ) ;
   else
        return 1;
}
```

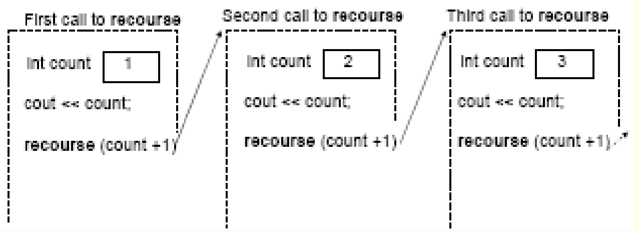
How Recursion Looks Like

```
void recurse( )
{
   recurse ( ); //Function calls itself
}

int main ( )
{
   recurse ( ); //Sets off the recursion
   return 0; //Rather pitiful, it will never be reached
}
```

Recursion Example

```
void recurse ( int count ) //The count variable is initialized
{ // by each function call
    cout<<count ;
    recurse(count+1); // It is not necessary to increment count
}                                   // each function's variables
                                    // are separate (so each count will be
                                    // initialized one greater)


int main ( )
{
    recurse(1); //First function call, so it starts at one
    return 0;
}
```

What’s happening?



- Will go on until the computer stack gets full,
- Or until we control it with the help of a variable

Base Case of the function

- The condition where the function will not call itself.
- It is an if-statement that checks some variable for a condition (such as a number being less than zero, or greater than some other number)
- If that condition is true, it will not allow the function to call itself again.

Example of base case

```
void doll ( int size )
{
    if (size == 0 ) // No doll can be smaller than 0
            return; // Return does not have to return
                        // something, it can be used to exit a
                        // function
    cout<<size<<endl;
    doll (size-1); // Decrements the size variable so the next
                        // doll will be smaller.
}

int main ( )
{
    doll (10); //Starts off with a large doll
    return 0; //Finally, it will be used }
```

Important Info!

- If our base case is not properly set up, it is possible to have a base case that is always true (or always false).
- Once a function has called itself, it will be ready to go to the next line after the call.
- It can still perform operations.

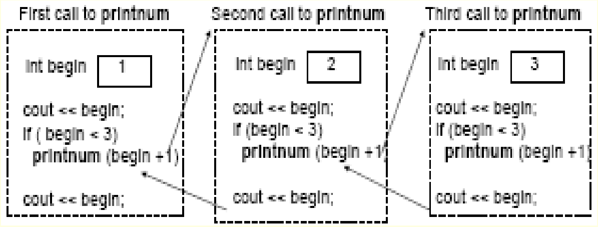
Performing operations **after** the call

- Problem:
- Write a function to print out the numbers
  123456789987654321.
- How can you use recursion to
  write a function to do this?

Solution

```
void printnum ( int begin)
{
   cout<<begin;
   if ( begin < 9 )
        printnum ( begin + 1);
                // The base case is when begin is greater
                // than 9, it will not recurse after the if
                // statement
   cout<<begin; } // Outputs the second begin, after the
                        // program has gone through and
                        //output the numbers from begin to 9.
```

What’s happening?

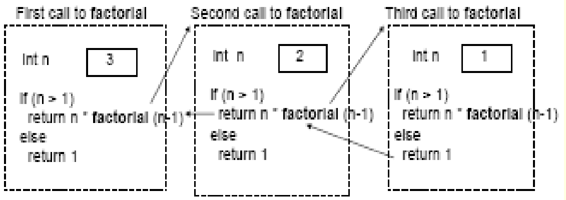


```
int begin 1

cout << begin;
if ( begin < 3)
  printnum (begin +1)

cout << begin;
```

```
int begin 2

cout << begin;
if (begin < 3)
  printnum (begin +1)

cout << begin;
```

```
int begin 3

cout << begin;
if (begin < 3)
  printnum (begin +1)

cout << begin;
```

What's happening?



First call to factorial

```
int n   3

if (n > 1)
  return n * factorial (n-1)
else
  return 1
```

Second call to factorial

```
int n   2

if (n > 1)
  return n * factorial (n-1)
else
  return 1
```

Third call to factorial

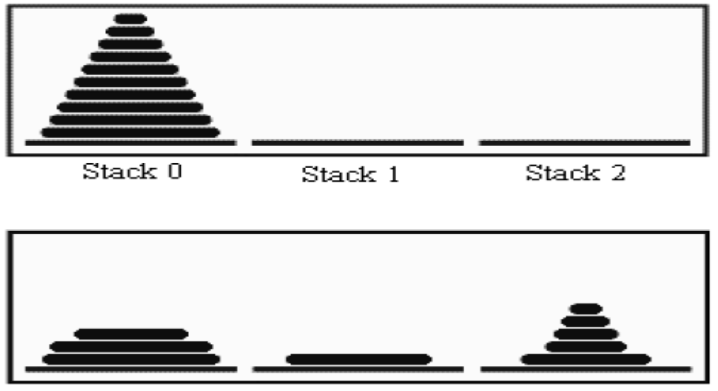
```
int n   1

if (n > 1)
  return n * factorial (n-1)
else
  return 1
```

- Third call will return 1
- Second call will return 2 * 1 = 2
- First call will return 3 * 2 = 6

Towers of Hanoi

- This problem involves a stack of various sized disks, piled up on a base in order of decreasing size.
- The object is to move the stack from one base to another, subject to two rules:
  - Only one disk can be moved at a time
  - No disk can ever be placed on top of a smaller disk.
- There is a third base that can be used as a "spare".

Towers of Hanoi representation



The stacks after a number of moves.

Towers of Hanoi Solution

- The base case is when there is only one disk to be moved.
- The solution in this case is trivial: Just move the disk in one step.

Towers of Hanoi Solution

- The base case is when there is only one disk to be moved.
- The solution in this case is trivial: Just move the disk in one step.

Towers of Hanoi Implementation

```
void TowersOfHanoi ( int disks, int from,
                                int to, int spare )
{
   if (disks == 1) // base case
        cout << "Move a disk from stack number "
                << from << " to stack number " << to
                <<endl;
```

Towers of Hanoi Implementation (cont)

```
else {
   // Move all but one disk to the spare
   //stack, then move the bottom disk, then
   //put all the other disks on top of it.
   TowersOfHanoi ( disks- 1, from , spare , to);
   cout << "Move a disk from stack number "
         << from << " to stack number " << to <<endl;
   TowersOfHanoi(disks- 1, spare, to, from);
}
```