Data Structures

Lecture 3 - Recursion

What Is Recursion?

- Recursion - defining the solution of a problem in terms of itself except for one or more primitive cases.

Is Factorial Recursive?

- The factorial function is defined as:

  n! = n · (n-1) · (n-2) …· 1

  or

  $$n! = \prod_{i=1}^{n} i$$

- The recursive definition is:

  n! = n (n-1)! for n > 0

  n! = 1 for n = 0

Factorial function

- We can write a factorial function:

```
float factorial (int n)
{
  float   prod;
  int     n;

  x = n;
  prod = 1;
  while (x != 0)
     prod *= x--;
  return(prod);
}
```

Factorial Function (continued)

- This is ***iterative***; it performs a calculation until a certain condition is met.
- By recursion:

(1) 5 ! = 5 • 4!	(5') 1! = 1 • 0! = 1 • 1 = 1
(2) 4! = 4 • 3!	(4') 2! = 2 • 1! = 2 • 1 = 2
(3) 3! = 3 • 2!	(3') 3! = 3 • 2! = 3 • 2 = 6
(4) 2! = 2 • 1!	(2') 4! = 4 • 3! = 4 • 4 = 24
(5) 1! = 1 • 0!	(1') 5! = 5 • 4! = 5 • 24 = 120
(6) 0 ≅ 1	

Other Examples of Recursion

- **<u>Multiplication</u>** - a • b

  a • b = a • (b-1) + a   if b > 1

  a • 1 = a   if b = 1

- **<u>Fibonacci Numbers</u>** - 0, 1, 1, 2, 3, 5, 8, 13, 21 , …

  Fib(n) = Fib(n-1) + Fib(n-2)   if n > 1

  Fib(n) = n   if n = 1, n = 0

  This is doubly recursive = 2 recursive call in the definition

Binary Search

- A binary search is a fairly quick way to search for data in a ***presorted*** data, sorted by ***key***.
- Algorithm

```
Initialize low and high
If low > high  THEN binsrch = 0
ELSE BEGIN
    mid = (low + high) / 2
    IF x = a[mid]   THEN binsrch = mid
            ELSE IF x < a[mid]
                    THEN search from low to mid-1
                    ELSE search from mid+1 to high
END
```

Binary Search (continued)

- Given numbers stored in an array sorted in ***ascending*** order, search for 25:

i	x[i]
1	2
2	4
3	5
4	6
5	10
6	15
7	17
8	20
9	25
10	32

Each pass:

Pass #	Low	High	Mid
1	1	10	5
2	6	10	8
3	9	10	9

Found it!

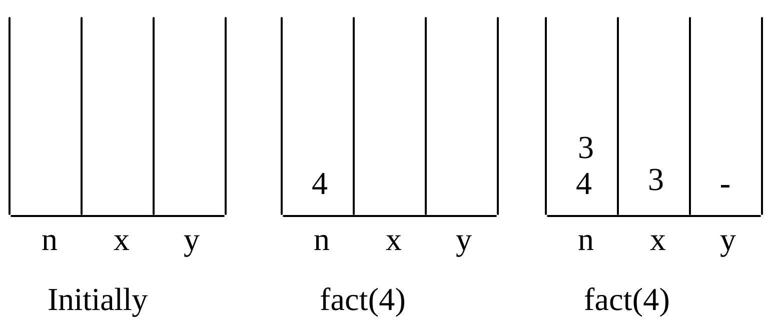
Tracing the Recursive Factorial

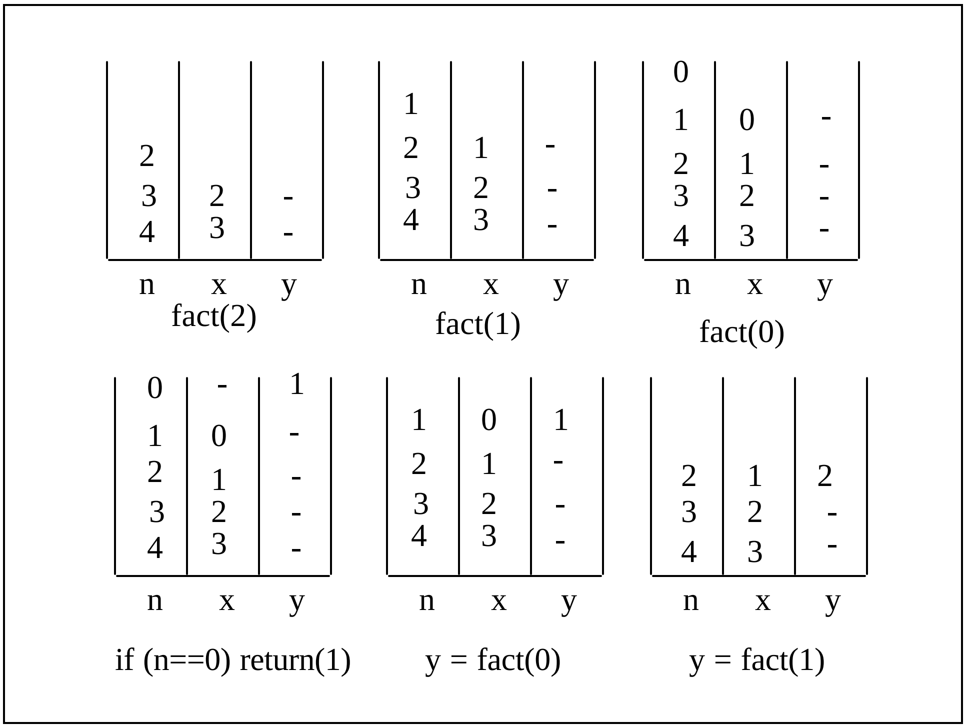
- Writing a recursive factorial function:

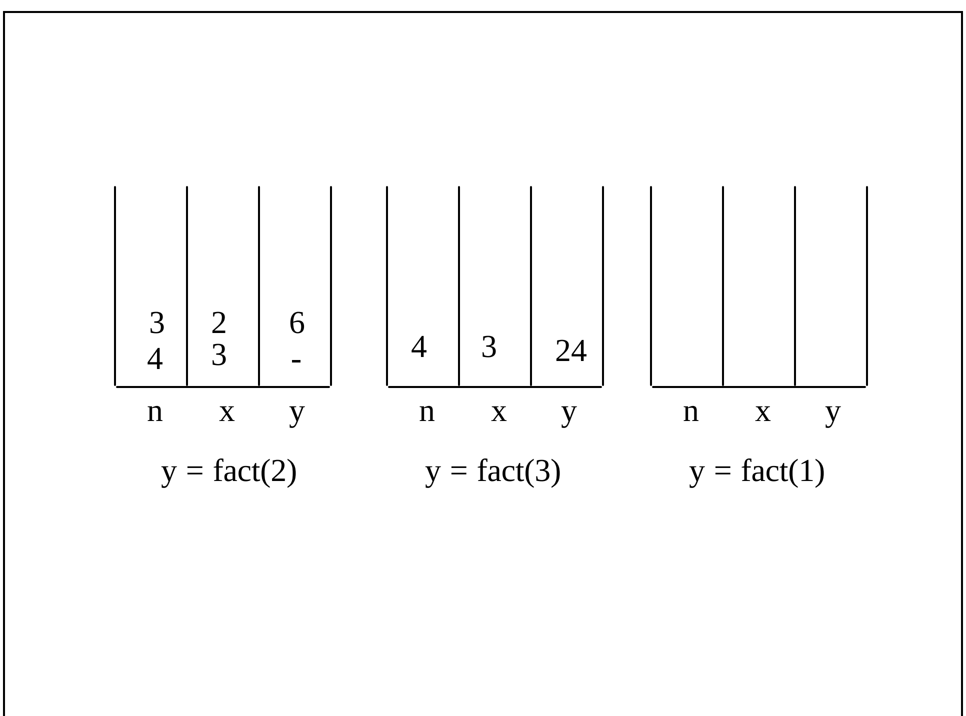
```
float   fact(int n)
{
  int   x;
  float y;
  if (n == 0)
   return(1);
  x = n – 1;
  y = fact(x);
  return(n*y);
}
```

Tracing Recursive Factorial (continued)

Let’s trace **`cout << fact(4);`**




n x y
fact(2)
n x y
fact(1)
n x y
fact(0)
n x y
if (n==0) return(1)
n x y
y = fact(0)
n x y
y = fact(1)


n x y
y = fact(2)
n x y
y = fact(3)
n x y
y = fact(1)

Recursive Multiplication

- We can also write a recursive multiplication function:

```
int      mult(int a, int b)
{
   if (b == 1)
      return(a);
   return(mult(a, b-1) + a);
}
```

or

```
int      mult(int a, int b)
{
  return(b == 1? a : mult(a, b-1) + a);
}
```

Rewriting **`fact`**

- We can rewrite fact as

```
float     fact(int n)
{
   return(n == 0? 1 : n * fact(n-1));
}
```

- What about fact(-1)? We want to catch the error and our current function does not.

Rewriting **`fact`** (continued)

```
float fact(int  n)
{
     int  x;
     if (n < 0){
          cerr << “Negative parameter in”
               << “ factorial function\n”;
          exit(1);
     }
     return ((n == 0)? 1 : n * fact(n-1));
}
```

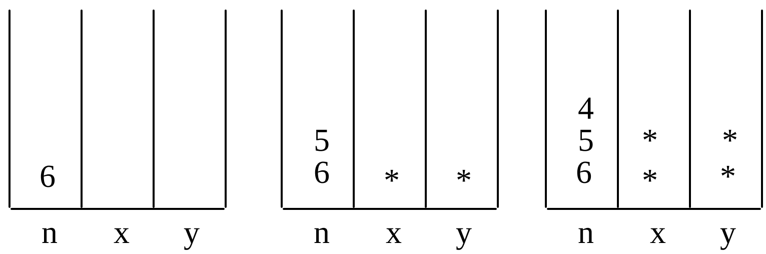
Writing the Fibonacci Function

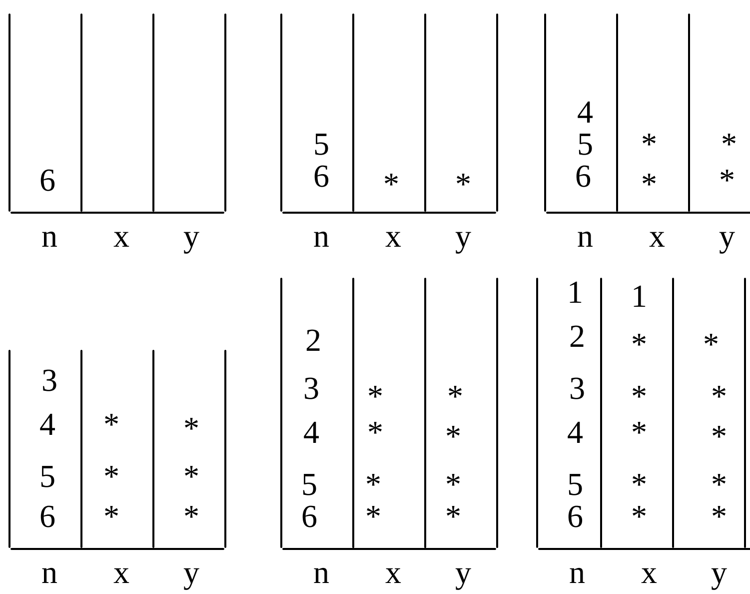
- $F_n = F_{n-1} + F_{n-2}$ for $n > 1$
- $F_0 = 0$; $F_1 = 1$

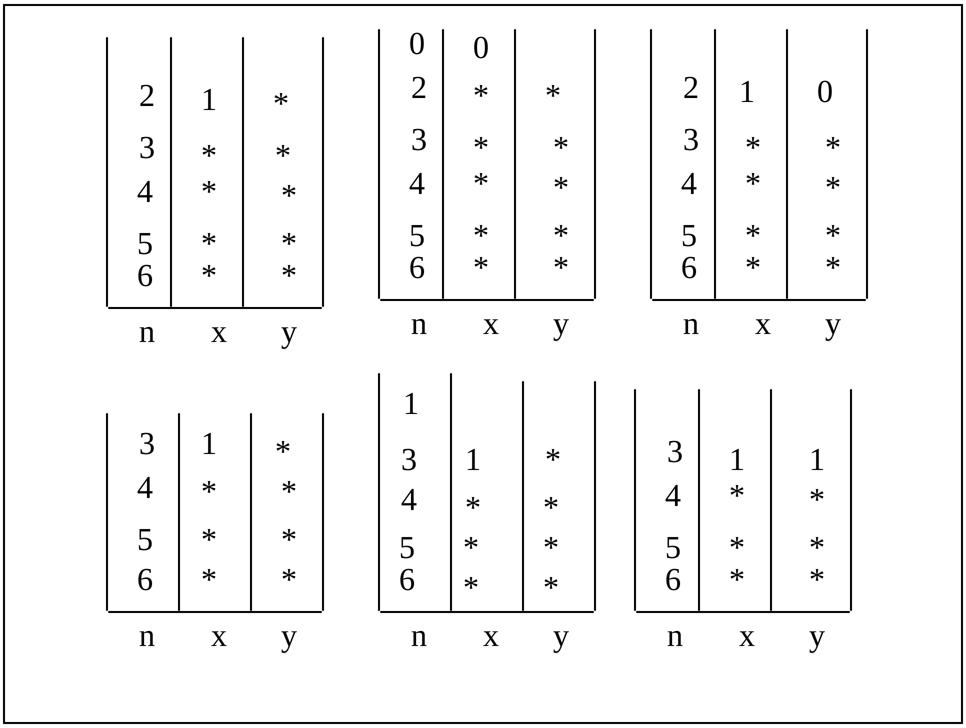
```
int   fib(int  n)
{
      int   x, y;
      if (n >= 1)
            return(n);
      x = fib(n-1);
      y = fib(n-2)
      return(x + y);
}
```

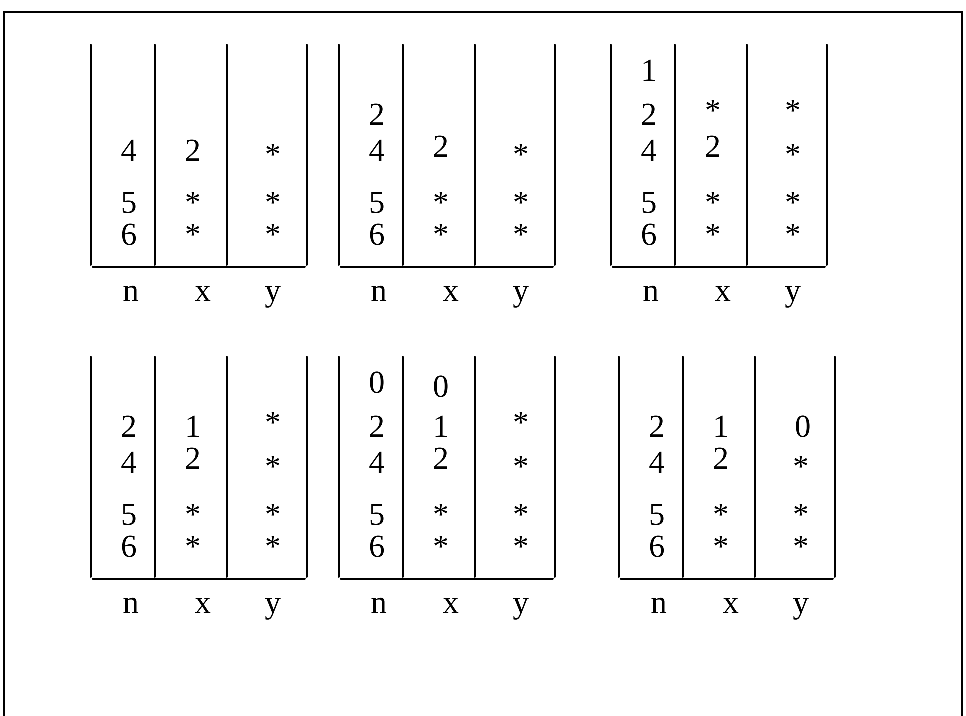
Tracing Recursive Fibonacci (continued)

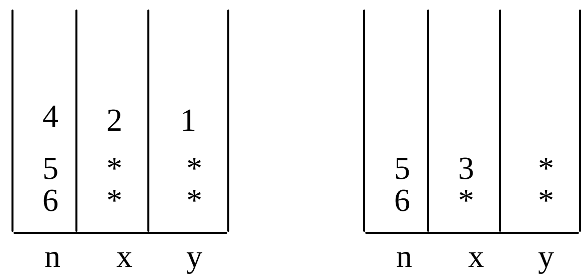
Let’s trace `cout << fib(6);`






2 3 4 5 6
1 * * * *
* * * * *
n x y
0 2 3 4 5 6
0 * * * * *
* * * * *
n x y
2 3 4 5 6
1 * * * *
0 * * * *
n x y
3 4 5 6
1 * * *
* * * *
n x y
1 3 4 5 6
1 * * *
* * * *
n x y
3 4 5 6
1 * * *
1 * * *
n x y


4 5 6
2 * *
* * *
n x y
2 4 5 6
2 * *
* * *
n x y
1 2 4 5 6
* 2 * *
* * * *
n x y
2 4 5 6
1 2 * *
* * * *
n x y
0 2 4 5 6
0 1 2 * *
* * * *
n x y
2 4 5 6
1 2 * *
0 * * *
n x y



Writing the Binary Search

- We invoke the binary search by writing:

  `i = binsrch(a, x);`

  It will check the array a for an integer x.

```
// binsrch() -        The classic binary search algorithm
//                    written recursively.  It requires
//                    that the array, search key and bounds
//                    of the subarray being searched be
//                    passed as parameters.
int    binsrch(int a[], int x, int low, int high)
{
       int    mid;
```

```
        if (low > high)
               return(-1);   // Not in the array
        mid = (low + high)/2;
        return((x == a[mid])? mid :
                      (x < a[mid])?
                             //Check the lower half
                             binsrch(a, x, low, mid-1):
                             //Check the upper half
                             binsrch(a, x, mid+1, high));
}
```

Revising the Binary Search

- Passing a and x should not be necessary since they do change from one recursive call to the next. Let's make them global:

```
int   a[ArraySize];
int   x;
```

and it's called by

```
i = binsrch(0, n-1);
```

Revised Binary Search

```
int    binsrch(int low, int high)
{
       int    mid;

       if (low > high)
              return(-1);   // Not in the array
       mid = (low + high)/2;
       return((x == a[mid])? mid :
                     (x < a[mid])?
                            //Check the lower half
                            binsrch(low, mid-1):
                            //Check the upper half
                            binsrch(mid+1, high));
}
```

Recursive Chains

- A recursive function does not have to call itself directly; it can call another function which in turn called the first function:
- ```
  a(…)                    b(…)
   {                      {
    ….                        ….
    b(…);                 a(…);
   }                      }
  ```
- This is called a ***recursive chain***.
  ```

Recursive Definition of Algebraic Expression

- An example of a recursive chain might be an algebraic expression where:
  - An expression is a term followed by a plus sign followed by term or a single term.
  - A term is a factor followed by a asterisk followed by factor or a single factor.
  - A factor is either a letter or an expression enclosed in parentheses.

The `expr` Program

```
#include      <iostream.h>
#include      <string.h>
#include      <ctype.h>

enum boolean  {false, true};
const int     MaxStringSize = 100;

int    getsymb(char str[], int length, int &pos);
void   readstr(char *instring, int &inlength);
int    expr(char str[], int length, int &pos);
int    term(char str[], int length, int &pos);
int    factor(char str[], int length, int &pos);
```

```
// main() -   This program allows a user to test whether
//            an expression is valid or invalid.  All
//            variables and constants are restricted to
//            one character.
int    main(void)
{
       char   str[MaxStringSize];
       int    length, pos;

       readstr(str, length);
       pos = 0;
```

```
       if (expr(str, length, pos) && pos >= length)
              cout << "Valid expression" << endl;
       else
              cout << "Invalid expression" << endl;
       // The condition can fail for one (or both) of two
       // reason.  If expr(str, length, pos) == false
       // then there is no valid expression beginning at
       // pos.  If pos < length there may be a valid
       // expression starting at pos but it does occupy
       // the entire string.
        return(0);
}
```

```
// expr() -   Returns true if str is a valid expression
//            Returns false is str is not.
int expr(char str[], int length, int &pos)
{
       // Look for a term
       if (term(str, length, pos) == false)
              return(false);

       // We have found a term – now look at the
       //     next symbol
```

```
       if (getsymb(str, length, pos) != '+')   {
              // We have found the longest expression
              // (a single term).  Reposition pos so it
              // refers to the last position of the
              // expression
              --pos;
              return(true);
       }
       // At this point, we have found a term and a
       // plus sign.  We must looj for another term
        return(term(str, length, pos));
}
```

```
// term() -   Returns true if str is a valid term
//            Returns false is str is not.
int    term(char str[], int length, int &pos)
{
       if (factor(str, length, pos) == false)
              return(false);

       if (getsymb(str, length, pos) != '*')    {
              --pos;
                return(true);
       }
       return(factor(str, length, pos));
}
```

```
// factor() - Returns true if str is a valid factor
//            Returns false is str is not.
int    factor(char str[], int length, int &pos)
{
        int   c;

       if ((c = getsymb(str, length, pos)) != '(')
              //The factor is not inside parentheses
              return(isalpha(c));

        // Examine parenthetic terms
       return(expr(str, length, pos)
                     && getsymb(str, length, pos) == ')');
}
```

```
//getsymb() - Returns the next character in the string str
int    getsymb(char str[], int length, int &pos)
{
       char   c;
       if (pos < length)
              c = str[pos];
       else
              // Beyond the end of the line of text
              c = ' ';
       pos++;
        return(c);
}
```

```
//readstr() - Reads a line of text that is assumed to be
//             an expression
void   readstr(char *instring, int &inlength)
{
       cin >> instring;
       inlength = strlen(instring);
}
```

Towers of Hanoi

The Tower of Hanoi gives us an example of a problem that can only be solved by recursion:

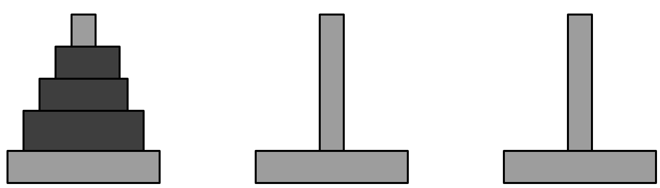
1. Three pegs with n disks - the smallest disk is on top and the largest is on the bottom.
2. A disk cannot beplaced on top of a smaller disk
3. Only one disk can be moved at a time.

<u>The solution is:</u>

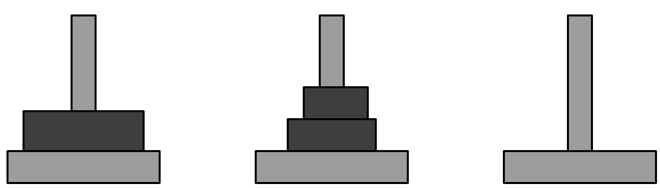
1. Assume n-1 disks are moved onto the auxiliary disk
2. Move bottom disk to destination peg.
3. Move n-1 disks to destination peg.
4. If n=1, move the only disk to the destination peg.

<u>Example</u> - Towers of Hanoi For 3 Disks

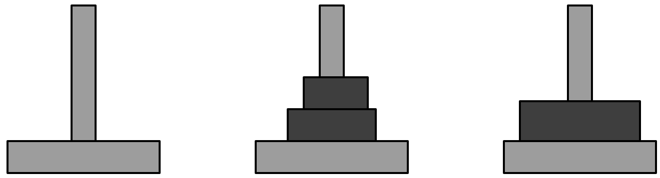
1.



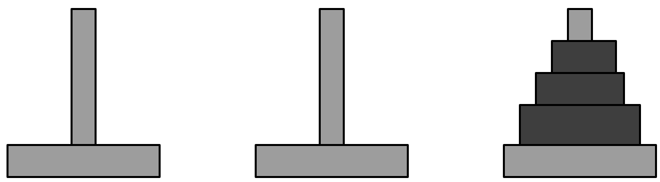
2.



3.



4.



Towers of Hanoi Program

```
#include    <iostream.h>

void  towers (int n, char frompeg, char topeg, char
auxpeg);

// main() - A driver for the towers function
int   main(void)
{
      int   n;

      cout << "How many disks on the Towers of”
            “ Hanoi  ?";
      cin >> n;
      towers(n, 'A', 'C', 'B');
      return(0);
}
```

```
// towers() - A recursive solution to the Tower of
//                Hanoi Problem
void  towers (int n, char frompeg, char topeg,
                        char auxpeg)
{
      // If only one disk, make the move and return
      if (n==1)   {
            cout << "Move disk 1 from peg ”
                  << frompeg  << " to peg ”
                  << topeg << endl;
            return;
      }
```

```
      // Move top n-1 disks  from A to B using C as
      //    auxiliary
      towers(n-1, frompeg, auxpeg, topeg);

      // Move remaining disk from A to C
      cout << "Move disk " << n << " from peg ”
            << frompeg << " to peg " << topeg
            << endl;

      // Move n-1 disks  from B to CB using A as
      //    auxiliary
      towers(n-1, auxpeg, topeg, frompeg);

}
```

Simulating Recursion

- Being able to simulate recursion is important because:
  - Many programming languages do not implement it, .e.g., FORTRAN, COBOL, assembler, etc.
  - It teaches us the implementation of recursions and its pitfalls.
  - Recursion is often more expensive computationally than we find we can afford.
- In order to simulate recursion, we must understand how function calls and function returns work.

Calling a function

Calling a function consists of 3 actions:

- Passing arguments (or parameters)

  A copy of the parameter is made locally within the function and any changes to the parameter are made to the local copy.
- Allocating and initializing local variables

  These local variables include those declared directly in the functions and any temporary variables that must be created during execution e.,g., if we add `x + y + z`, we need a place to store `x + y` temporarily.
- Transferring control to the function

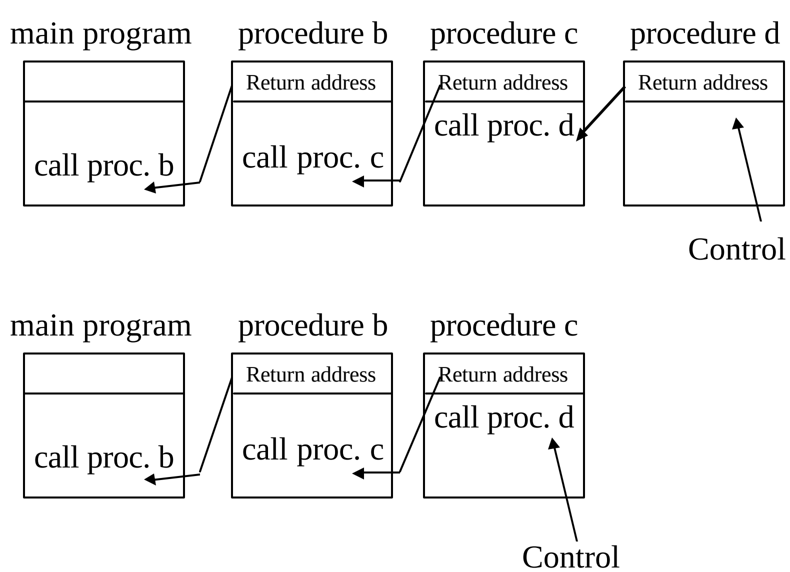
  Save the return address and transfer control to the function

Returning From a Function Call

Returning from a function call consists of the following steps:

- The return address is retrieved and saved in a safe place (i.e., outside the function's data area).
- The function's data area is freed.
- The function returns control by branching to the return address.

Returning From a Function Call



Implementing Recursive Functions

```
typedef struct {
      int   param;
      int   x;
      long  y;
      short retaddr;
} dataarea;
```

```
class  stack  {
public:
      boolean       empty(void);
      dataarea      pop(void);
      void          push(dataarea x);
      dataarea      stacktop(void);
      boolean       popandtest(dataarea &x);   //Tests
before popping
      boolean pushandtest(dataarea x);  stack(void);
      //Default constructor
      stack(dataarea x);  //Init.  constructor
private:
      int           top;
      dataarea      item[StackSize];
};
```

Simfact function

```
int   simfact(int n)
{
      dataarea    currarea;
      stack       s;
      short       i;
      long        result;

      // Initialize a dummy data area
      currarea.param = 0;
      currarea.x = 0;
      currarea.y = 0;
      currarea.retaddr = 0;
```

```
      // Push the data data area onto the stack
      s.push(currarea);

      // Set the parametetr and the return address
      // of the current data area to their proper
      // values
      currarea.param = n;
      currarea.retaddr = 1;
```

```
//This is the beginning of the simulated
//    factorial routine
start:
if (currarea.param == 0)      {
// Simulation of return(1)
      result = 1;
      i = currarea.retaddr;
      currarea = s.pop();
      switch(i)   {
        case 1:   goto label1;
        case 2:   goto label2;
      }
}
```

```
currarea.x = currarea.param - 1;
//Simulation of recursive call to fact
s.push(currarea);
currarea.param = currarea.x;
currarea.retaddr = 2;
goto start;

// This is the point to which we return from
//    the recursive call.
// Set currarea.y to the returned value
```

```
label2:
currarea.y = result;
// Simulation of return(n*y);
result = currarea.param * currarea.y;
i = currarea.retaddr;
currarea = s.pop();
switch(i)   {
  case 1:   goto label1;
    case 2: goto label2;
}
```

```
//At this point we return to the main routine
  label1:
  return(result);
}
```

Improving **`simfact`**

- Do we need to stack all the local variables in this routine?
  - n changes and is used again after returning from a recursive call.
  - x is never used again after the recursive call
  - y is not used until after we return.
  - We can avoid saving the return address if we use stack underflow as a criterion for exiting the routine.

`simfact` with a limited stack

```
int   simfact(int n)
{
      stack s;     //s stacks only the current
                   // parameter
      short und;
      long  result, y;
      int   currparam, x;


      // Set the parameter and the return address
      // of the current data area to their proper
      // values
      currparam = n;
```

```
//This is the beginning of the simulated
//    factorial routine
start:
if (currparam == 0)     {
// Simulation of return(1)
      result = 1;
      und = s.popandtest(currparam);
      switch(und) {
        case true:      goto label1;
            case false: goto label2;
      }
}

// currparam != 0
x = currparam - 1;
//Simulation of recursive call to fact
s.push(currparam);
currparam = x;
goto start;
```

```
        // This is the point to which we return from
        //    the recursive call.
        // Set y to the returned value
        label2:
        y = result;
        // Simulation of return(n*y);
        result = currparam * y;
        und = s.popandtest(currparam);
        switch(und) {
          case true:      goto label1;
            case false:   goto label2;
        }

        //At this point we return to the main
routine
          label1:
          return(result);
}
```

Eliminating **goto**s

- Goto is a bad programming form because it obscures the meaning and intent of the algorithm.
- We will combine the two references to `popandtest` and `switch(und)` into one.

simfact with one **switch**

```
int   simfact(int n)
{
      stack s;
      short und;
      long  y;
      int   x;

      x = n;
      //This is the beginning of the simulated
      //    factorial routine
      start:
      if (x == 0)
            y = 1;
      else  {
            s.push(x--);
                goto start;
      }
```

```
        label1:
        und=s.popandtest(x);
        if (und == true)
              return(y);

        label2:
        y *= x;
        goto label1;

}
```

Eliminating the `goto`s

- We recognize that there are really two loops:
  - one loop which generates additional function call (simulated by pushing the parameter on the stack)
  - another loop where we return from recursive calls (simulated by popping the parameter off the stack).

simfact without **goto**s

```
int   simfact(int n)
{
      stack s;
      short und;
      long  y;
      int   x;


      x = n;

      //This is the beginning of the simulated
      //    factorial routine
      start:
      while (x != 0)
            s.push(x--);
      y = 1;
      und=s.popandtest(x);
```

```
      label1:
      while (und == false)    {
            y *= x;
                und = s.popandtest(x);
      }

        return(y);
}
```

Finally…after eliminating the unnecessary pushes and pops...

```
int  simfact(int n)
{
     long y;
     int  x;

     for  (y = x= 1;  x <= n;  x++)
          y *= x;

     return(y);
}
```