# Dynamic Memory Allocation

## Dynamic Memory Allocation

- Can allocate storage for a variable while program is running
- Uses `new` operator to allocate memory

```
double *dptr;
dptr = new double;
```

  - `new` returns address of memory location
- Can also use `new` to allocate array

```
arrayPtr = new double[25];
```

  - Program often terminates if there is not sufficient memory
- Can then use `[]` or pointer arithmetic to access array

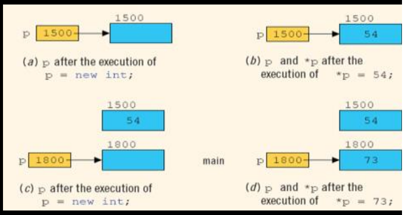
## Dynamic Memory Allocation

- Suppose you have the following declaration:

```
int *p;
```

  - This statement declares p to be a pointer variable of type int.

```
p = new int;
*p = 54;
p = new int;
*p = 73
```



(a) p after the execution of `p = new int;`

(b) p and *p after the execution of `*p = 54;`

(c) p after the execution of `p = new int;`

(d) p and *p after the execution of `*p = 73;`

## Releasing Dynamic Memory

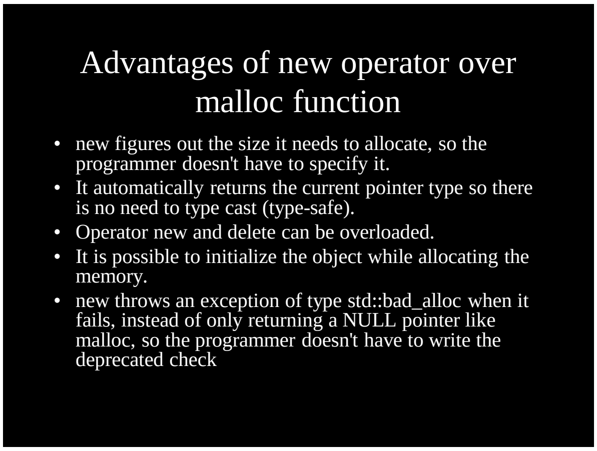
- Use **`delete`** to free dynamic memory

```
delete dptr;
```

- Use **`delete []`** to free dynamic array memory

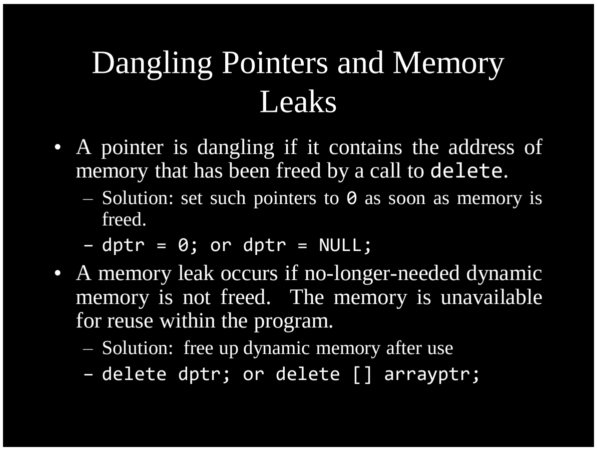
```
delete [] arrayptr;
```

- Only use **`delete`** with dynamic memory!



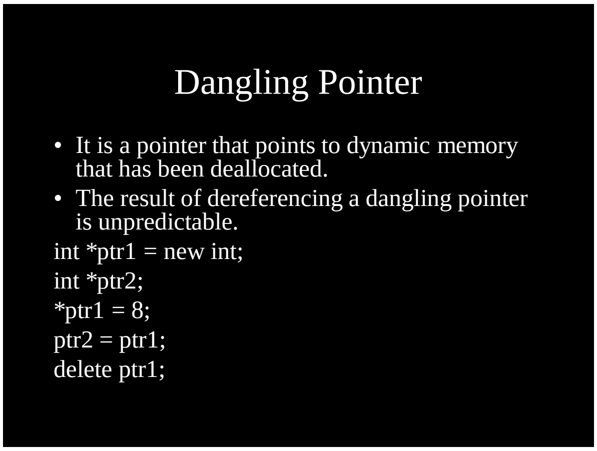
## Advantages of new operator over malloc function

- new figures out the size it needs to allocate, so the programmer doesn't have to specify it.
- It automatically returns the current pointer type so there is no need to type cast (type-safe).
- Operator new and delete can be overloaded.
- It is possible to initialize the object while allocating the memory.
- new throws an exception of type std::bad_alloc when it fails, instead of only returning a NULL pointer like malloc, so the programmer doesn't have to write the deprecated check



## Dangling Pointers and Memory Leaks

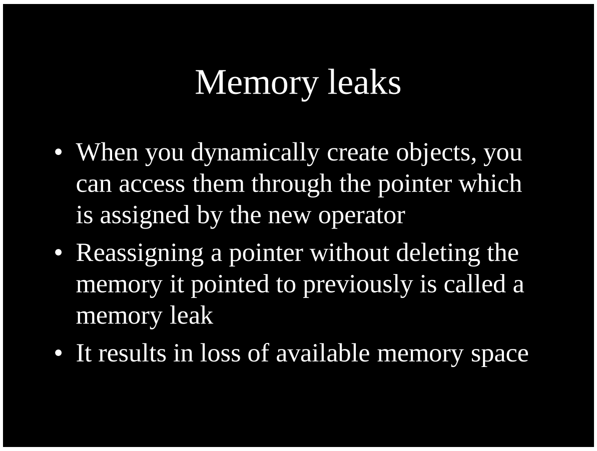
- A pointer is dangling if it contains the address of memory that has been freed by a call to `delete`.
  - Solution: set such pointers to `0` as soon as memory is freed.
  - `dptr = 0; or dptr = NULL;`
- A memory leak occurs if no-longer-needed dynamic memory is not freed. The memory is unavailable for reuse within the program.
  - Solution: free up dynamic memory after use
  - `delete dptr; or delete [] arrayptr;`



## Dangling Pointer

- It is a pointer that points to dynamic memory that has been deallocated.
- The result of dereferencing a dangling pointer is unpredictable.

```
int *ptr1 = new int;
int *ptr2;
*ptr1 = 8;
ptr2 = ptr1;
delete ptr1;
```



## Memory leaks

- When you dynamically create objects, you can access them through the pointer which is assigned by the new operator
- Reassigning a pointer without deleting the memory it pointed to previously is called a memory leak
- It results in loss of available memory space

## Memory leak example

```
int *ptr1 = new int;
int *ptr2 = new int;
*ptr1 = 8;
*ptr2 = 5;
ptr2 = ptr1;
```

- Inaccessible object

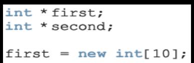
An inaccessible object is an unnamed object that was created by operator new and which a programmer has left without a pointer to it.

It is a logical error and causes memory leaks.

## When to use Dynamic memory allocation

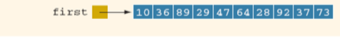
- Dynamic allocation is useful when
  - arrays need to be created whose extent is not known until run time
  - complex structures of unknown size and/or shape need to be constructed as the program runs
  - objects need to be created and the constructor arguments are not known until run time

## Shallow versus Deep Copy and Pointers



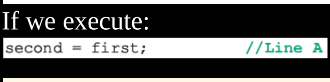
```
int *first;
int *second;

first = new int[10];
```

- Assume some data is stored in the array:

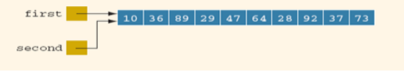


Pointer first and its array

- If we execute:



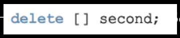
```
second = first;                //Line A
```



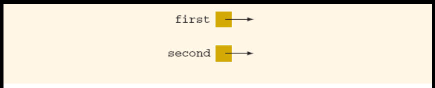
first and second after the statement second = first; executes

## Shallow versus Deep Copy and Pointers (cont'd.)

- Shallow Copy: two or more pointers of the same type point to the same memory
  - ame data



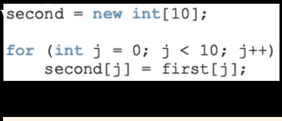
```
delete [] second;
```



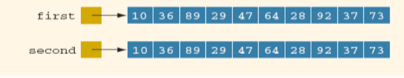
first and second after the statement delete [] second; executes

## Shallow versus Deep Copy and Pointers (cont'd.)

- Deep Copy: two or more pointers have their o



```
second = new int[10];

for (int j = 0; j < 10; j++)
    second[j] = first[j];
```



first and second both pointing to their own data

## Reading Assignment Examples

```
#include <iostream>
 using namespace std;
void X(int A, int &B)
{
A = A + B;
B = A - B;
A = A - B;
}
 int main()
{
 int a = 4, b = 18;
X(a,b);
cout << a << ", " << b;
return 0; }
```

- 
- Write the output of the following program :

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

void X(int &A, int &B)
{
   A = A + B;
   B = A - B;
   A = A - B;
}

int main()
{
   int a = 4, b = 18;
   X(a,b);
   cout << a << ", " << b;

   return 0;
}
```

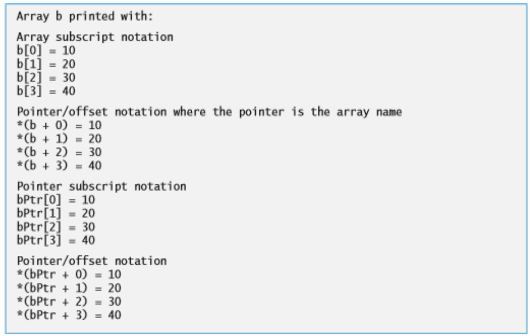
## Pointers to arrays

- A pointer variable can be used to access the elements of an array of the same type.
  - int gradeList[8] = {92,85,75,88,79,54,34,96};
  - int *myGrades = gradeList;
  - cout << gradeList[1];
  - cout << *myGrades;
  - cout << *(myGrades + 2);
  - cout << myGrades[3];
- Note that the array name gradeList acts like the pointer variable myGrades.

## Relationship between Arrays and Pointers

- int b[ 5 ]; // create 5-element int array b
- int *bPtr; // create int pointer bPtr
- bPtr =b; // assign address of array b to bPtr
- bPtr =&b[ 0 ]; // also assigns address of array b to bPtr
- *( bPtr + 3 ) //pointer-offset notation
- &b[ 3 ] //address
- bPtr + 3 //address
- *( b+3) //value at index 3
- bPtr[ 1 ] //pointer-subscript notation
- An array name is a constant pointer. It always points to the beginning of the array.
- b+=3;

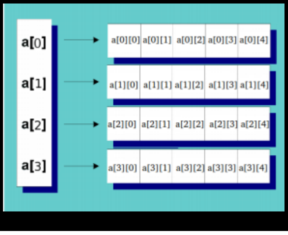
## Relationship between Arrays and



```
Array b printed with:

Array subscript notation
b[0] = 10
b[1] = 20
b[2] = 30
b[3] = 40

Pointer/offset notation where the pointer is the array name
*(b + 0) = 10
*(b + 1) = 20
*(b + 2) = 30
*(b + 3) = 40

Pointer subscript notation
bPtr[0] = 10
bPtr[1] = 20
bPtr[2] = 30
bPtr[3] = 40

Pointer/offset notation
*(bPtr + 0) = 10
*(bPtr + 1) = 20
*(bPtr + 2) = 30
*(bPtr + 3) = 40
```

## Dynamic allocation of 2D arrays

- A two dimensional array is really an array of arrays (rows).
- To dynamically declare a two dimensional array of int type, you need to declare a pointer to a pointer as:



a[0] → a[0][0] a[0][1] a[0][2] a[0][3] a[0][4]

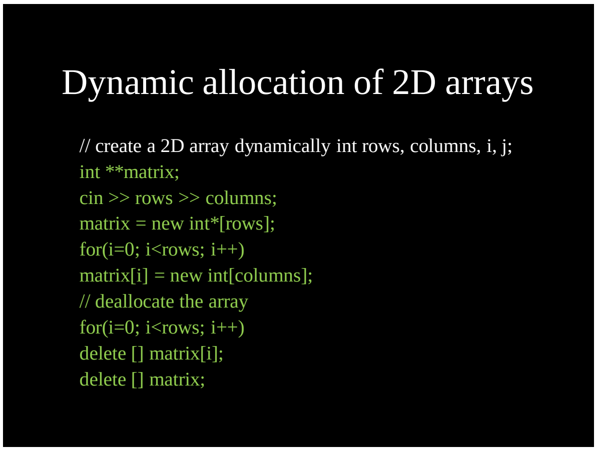
a[1] → a[1][0] a[1][1] a[1][2] a[1][3] a[1][4]

a[2] → a[2][0] a[2][1] a[2][2] a[2][3] a[2][4]

a[3] → a[3][0] a[3][1] a[3][2] a[3][3] a[3][4]

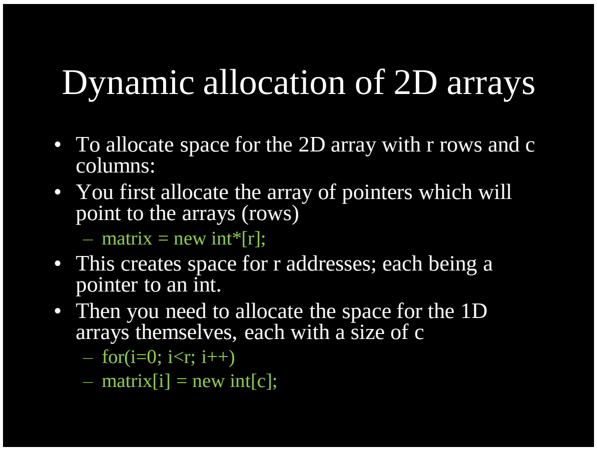
## Dynamic allocation of 2D arrays

- A two dimensional array is an array of arrays (rows).
- To dynamically declare a two dimensional array of int type, you need to declare a pointer to a pointer as:
  - int **matrix;
- The elements of the array matrix now can be accessed by the matrix[i][j] notation or (*(*matrix+i)+j) or pointer-offset notation.
- Keep in mind, the entire array is not in contiguous space (unlike a static 2D array)
- The elements of each row are in contiguous space, but the rows themselves are not.
- matrix[i][j+1] is after matrix[i][j] in memory, but matrix[i][0] may be before or after matrix[i+1][0] in memory



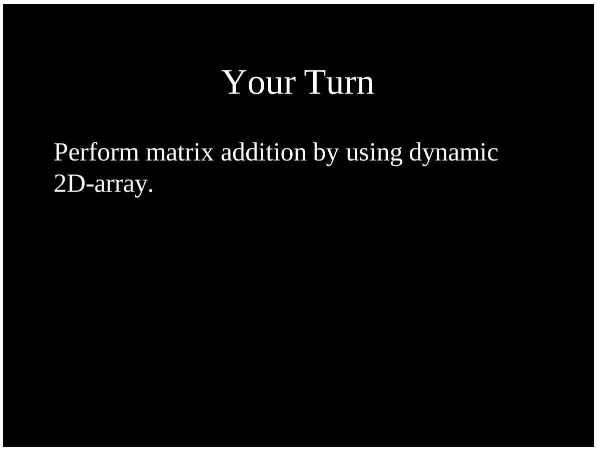
## Dynamic allocation of 2D arrays

```
// create a 2D array dynamically int rows, columns, i, j;
int **matrix;
cin >> rows >> columns;
matrix = new int*[rows];
for(i=0; i<rows; i++)
matrix[i] = new int[columns];
// deallocate the array
for(i=0; i<rows; i++)
delete [] matrix[i];
delete [] matrix;
```



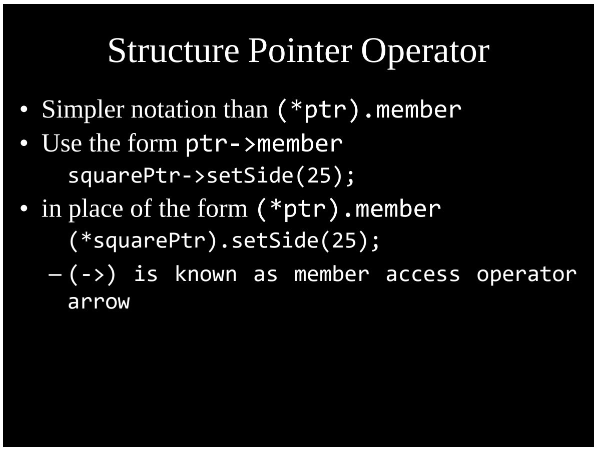
## Dynamic allocation of 2D arrays

- To allocate space for the 2D array with r rows and c columns:
- You first allocate the array of pointers which will point to the arrays (rows)
  - matrix = new int*[r];
- This creates space for r addresses; each being a pointer to an int.
- Then you need to allocate the space for the 1D arrays themselves, each with a size of c
  - for(i=0; i<r; i++)
  - matrix[i] = new int[c];



## Your Turn

Perform matrix addition by using dynamic 2D-array.



## Structure Pointer Operator

- Simpler notation than `(*ptr).member`
- Use the form `ptr->member`

```
squarePtr->setSide(25);
```

- in place of the form `(*ptr).member`

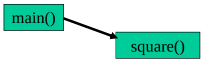
```
(*squarePtr).setSide(25);
```

  - (->) is known as member access operator arrow

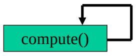
## Recursion

## Introduction to Recursion

- So far, we have seen functions that call other functions.



  - For example, the `main()` function calls the square`()` function.
- Recursive Function:
  - A recursive function is a function that calls itself.



## Recursion

- The best way to solve a problem is by solving a **smaller version** of the exact same problem first
- Recursion is a technique that solves a problem by solving a **smaller problem** of the same type

- Base case
- Recursive case

## Recursion vs. iteration

- Iteration can be used in place of recursion
  - An iterative algorithm uses a *looping construct*
  - A recursive algorithm uses a *branching structure*
- Recursive solutions are often less efficient, in terms of both *time* and *space*, than iterative solutions
- Recursion can simplify the solution of a problem, often resulting in shorter, more easily understood source code

# Factorials

- Computing factorials are a classic problem for examining recursion.
- A factorial is defined as follows:

  n! = n * (n-1) * (n-2) …. * 1;
- For example:

  1! = 1 (Base Case)

  2! = 2 * 1 = 2

  3! = 3 * 2 * 1 = 6

  4! = 4 * 3 * 2 * 1 = 24

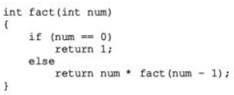
  5! = 5 * 4 * 3 * 2 * 1 = 120

# Iterative Approach

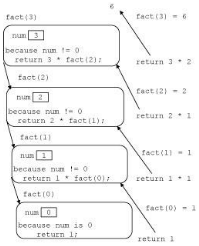
```
int findFactorial (int n)
{
      int i, factorial = n;
      for (i = n - 1; i >= 1; i--)
            factorial = factorial * i;
      return factorial;
}
```

# Recursive Factorial

- Recursive function implementing the factorial function

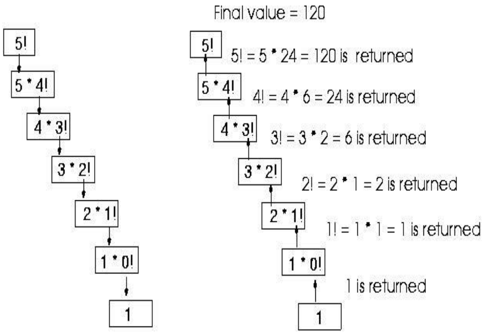


```
int fact(int num)
{
    if (num == 0)
        return 1;
    else
        return num * fact(num - 1);
}
```

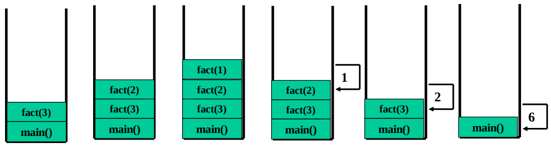


Execution of fact(4)

# Execution of fact(5)


Final value = 120
5!
5 * 4!
4 * 3!
3 * 2!
2 * 1!
1 * 0!
1
5!
5! = 5 * 24 = 120 is returned
5 * 4!
4! = 4 * 6 = 24 is returned
4 * 3!
3! = 3 * 2 = 6 is returned
3 * 2!
2! = 2 * 1 = 2 is returned
2 * 1!
1! = 1 * 1 = 1 is returned
1 * 0!
1 is returned
1

## Finding the factorial of 3



**Time 2:**
**Push:  fact(3)**

**Time 3:**
**Push: fact(2)**

**Time 4:**
**Push: fact(1)**

**Time 5:**
**Pop: fact(1)**
**returns 1.**

**Time 6:**
**Pop: fact(2)**
**returns 2.**

**Time 7:**
**Pop: fact(3)**
**returns 6.**

**Inside findFactorial(3):**
if (number <= 1) return 1;
**else return (3 * factorial (2));**

**Inside findFactorial(2):**
if (number <= 1) return 1;
**else return (2 * factorial (1));**

**Inside findFactorial(1):**
**if (number <= 1) return 1;**
else return (1 * factorial (0));

## Your Turn

- **The Collatz conjecture is:** *The process will eventually reach the number 1, regardless of which positive integer is chosen initially.*
- It is always possible to get back to 1 if you follow these steps. (applies to positive integers only
- ***int Collatz(int n)***
- If n=1, stop
- If n=even, repeat process n/2
- If n=odd, repeat process with 3n+1

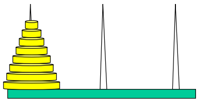
## Home Task

- Recursive SumOfN
- Fibbonacci
  - Each number in the series is sum of two previous numbers
    - e.g., 0, 1, 1, 2, 3, 5, 8, 13, 21…

      fibonacci(0) = 0
      fibonacci(1) = 1
      fibonacci(n) = fibonacci(n - 1) + fibonacci( n – 2 )
    - fibonacci(0) and fibonacci(1) are base cases

## Tower of Hanoi

- There are three towers
- 64 gold disks, with decreasing sizes, placed on the first tower
- You need to move all of the disks from the first tower to the last tower
- Larger disks can not be placed on top of smaller disks
- The third tower can be used to temporarily hold disks

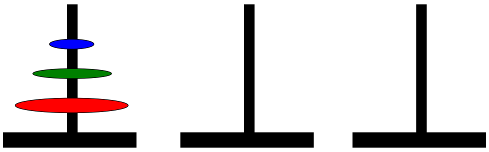


## Tower of Hanoi (cont'd.)

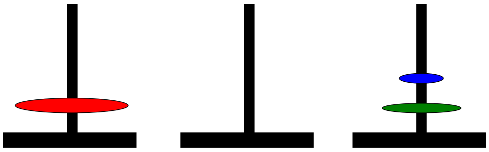
- Generalize problem to the case of 64 disks
  - Recursive algorithm in pseudocode
- Take (n-1) disc from source to destination
- Put the only 1 left over disc from source to destination
- Take (n-1) from Temporary to Destination

43

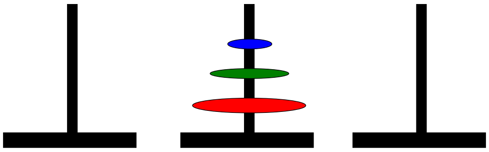
## Recursive Solution



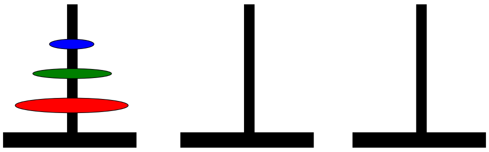
## Recursive Solution



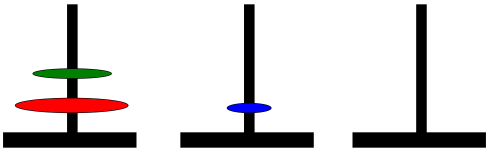
## Recursive Solution



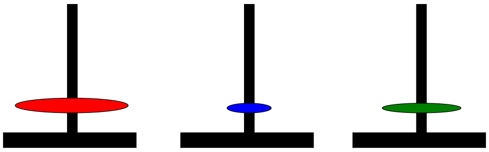
## Tower of Hanoi (Step by Step)



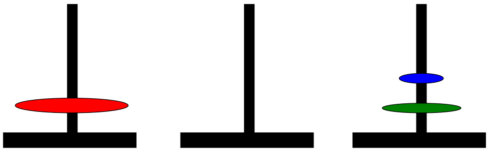
## Tower of Hanoi



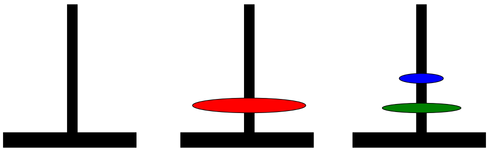
## Tower of Hanoi



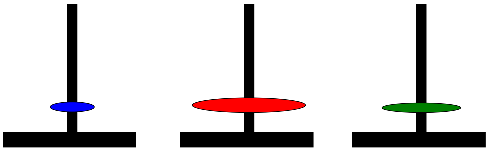
## Tower of Hanoi



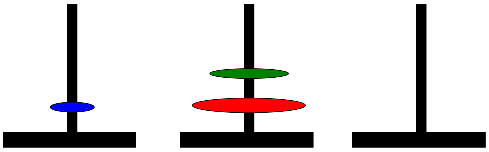
## Tower of Hanoi



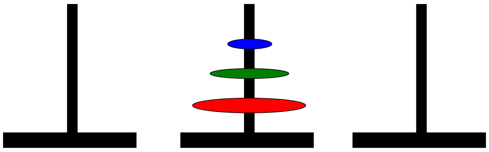
## Tower of Hanoi



## Tower of Hanoi



## Tower of Hanoi



## Recursive Algorithm

```
void Hanoi(int n, string a, string b, string c)
  {
    if (n == 1)  /* base case */
      Move(a,b);
    else { /* recursion */
      Hanoi(n-1,a,c,b);
      Move(a,b);
      Hanoi(n-1,c,b,a);
    }
  }
```

## Summary

Recursion is a valuable tool that allows some problems to be solved in an elegant and efficient manner.

Functions can sometimes require more than one recursive call in order to accomplish their task.

There are problems for which we can design a solution, but the nature of the problem makes solving it *effectively uncomputable*.