# Computer Programming

" Make everything as simple as possible, but not simpler."

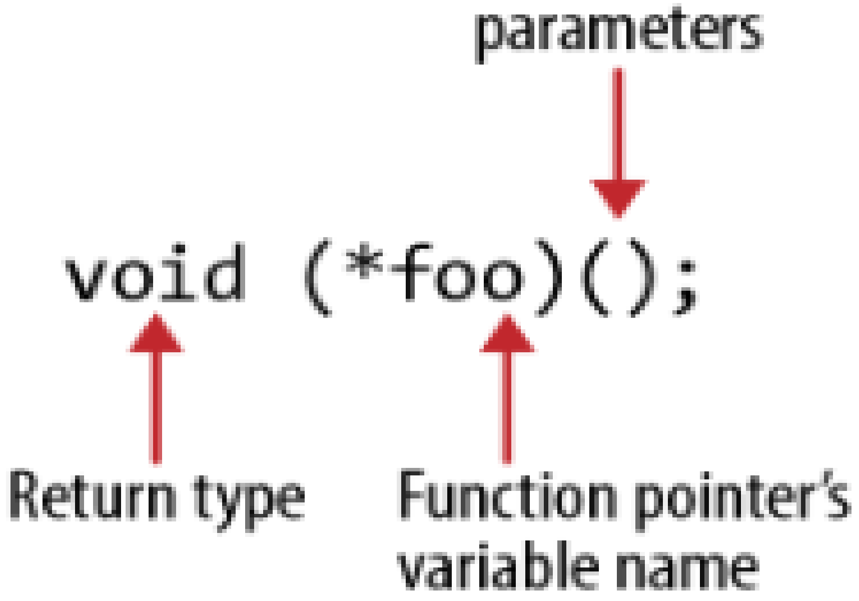
Albert Einstein

## Outline

- Pointers to functions
  - Declaring , using, comparing and casting pointers to functions
  - Examples
- Recursion
  - Mechanism
  - Examples:
    - String reverse
    - Smallest of *n* integers
    - Fibonacci numbers
    - Towers of Hanoi

## Declaring pointers to functions

- Example:



- **Important note:**
- You must be careful to ensure it is used properly because C does not check to see whether the correct parameters are passed

## Declaring pointers to functions

- More examples

```
int (*f1)(double); // Passed a double and
// returns an int
void (*f2)(char*); // Passed a pointer to char and
// returns void
double* (*f3)(int, int); // Passed two integers and
// returns a pointer to a double
```

- Suggested naming convention for function pointers:
  - Always begin their name with the prefix: **fptr**

## Pitfalls

- Do not confuse functions that return a pointer with function pointers
  - `int *f4();`
  - `int (*f5)();`
  - `int* (*f6)();`
- The whitespace within these expressions can be rearranged:
  - `int* f4();`
  - `int (*f5)();`

## Using a Function Pointer

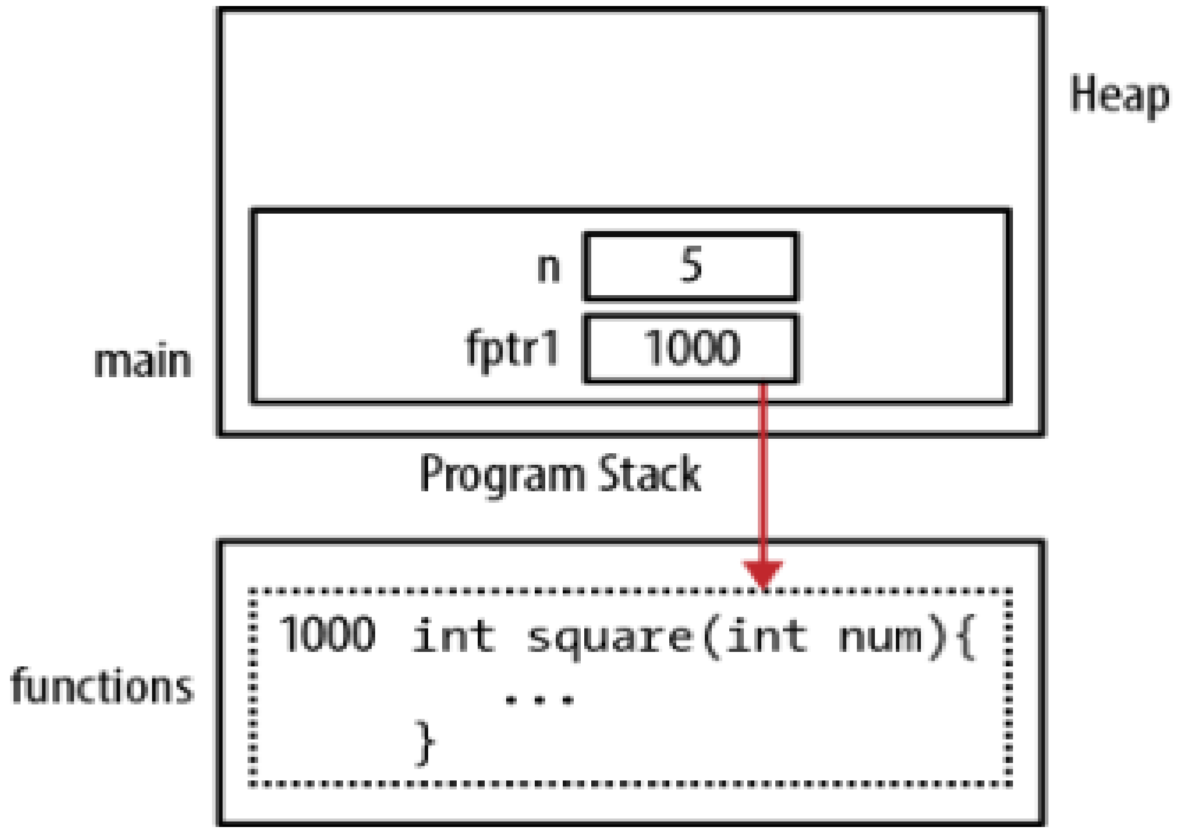
- Example:
  - Declarations

```
int (*fptr1)(int);
int square(int num) {
return num*num;
}
```

  - Usage

```
int n = 5;
fptr1 = square;
printf("%d squared is %d\n",n, fptr1(n));
```

## Location of functions



## Demo

- Ptr2Func Codeblocks project

## Returning Function Pointers

- Demo: RetFuncPtr codeclocks project

## Using an Array of Function Pointers

- Demo: ArrFuncPtr codeclocks project

## Other operations with function pointers

- Comparing Function Pointers

```
fptrOperation fptr1 = add;
if(fptr1 == add) {
   printf("fptr1 points to add function\n");
} else {
  printf("fptr1 does not point to add function\n");
}
```

- Casting Function Pointers

```
typedef int (*fptrToSingleInt)(int);
typedef int (*fptrToTwoInts)(int,int);
int add(int, int);
fptrToTwoInts fptrFirst = add;
fptrToSingleInt fptrSecond =
(fptrToSingleInt)fptrFirst;
fptrFirst = (fptrToTwoInts)fptrSecond;
printf("%d\n",fptrFirst(5,6));
```

## Notes

- The use of void* is not guaranteed to work with function pointers. Do NOT do as shown below:

```
void* pv = add;
```

- However it is common to see a “base” function pointer type as declared below.

```
typedef void (*fptrBase)();
```

- Demo of the use of this base pointer, which duplicates the previous example:

```
fptrBase basePointer;
fptrFirst = add;
basePointer = (fptrToSingleInt)fptrFirst;
fptrFirst = (fptrToTwoInts)basePointer;
printf("%d\n",fptrFirst(5,6));
```

- A base pointer is used as a *placeholder* to exchange function pointer values

## Pointers to functions

- A function *f*

```
f_ret_type f(f_formal_param_list)
```

- can be passed as an effective parameter to a function *g* defined as

```
g_ret_type(...,
f_ret_type(*p)(f_formal_param_list), ...)
```

- by

```
g(..., f, ...);
```

## Pointers to Functions

- Pointer to function
  - Contains address of function
  - Similar to how array name is address of first element
  - Function name is starting address of code that defines function
- Function pointers can be
  - Passed to functions
  - Stored in arrays
  - Assigned to other function pointers

## Pointers to Functions

- Example: **`bubblesort`**
  - Function bubble takes a function pointer
    - bubble calls this helper function
    - this determines ascending or descending sorting
  - The argument in bubblesort for the function pointer:

    ```
    int ( *compare )( int, int )
    ```

    - tells bubblesort to expect a pointer to a function that takes two ints and returns a int (used as a Boolean).
  - If the parentheses were left out:

    ```
    int *compare( int, int )
    ```

    - Declares a function that receives two integers and returns a pointer to an **int**

# Pointers to functions. Example

```
void bubble( int work[], const int size,
             int (*compare)( int, int ) )
{
   int pass, count;
   void swap( int *, int * );

   for ( pass = 1; pass < size; pass++ )
         for ( count = 0; count < size - 1; count++ )
                 if ( (*compare)( work[ count ], work[ count + 1 ] ) )
                         swap( &work[ count ], &work[ count + 1 ] );
}
int ascending(int a, int b)
{
   return (b < a); /* swap if b is less than a */
}
int descending(int a, int b)
{
   return (a < b); /* swap if b is greater than a */
}
void swap( int *element1Ptr, int *element2Ptr )
{
   int temp;
   temp = *element1Ptr;
   *element1Ptr = *element2Ptr;
   *element2Ptr = temp;
}
```

**ascending** and **descending** return **true** or **false**. **bubble** calls **swap** if the function call returns **true**.

Notice how function pointers are called using the dereferencing operator. The * is not required, but emphasizes that **compare** is a function pointer and not a function.

## Pointers to functions. Example

```
#define SIZE 10
int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
```

- Prototypes for functions:

```
void bubble( int [], const int, int (*)( int, int ));
int ascending( int, int );
int descending( int, int );
```

- Invocation of bubble
  - `bubble( a, SIZE, ascending );`
  - `bubble( a, SIZE, descending );`

## Pointers to functions. Example: integrating a function using the trapezoid method

- Trapezoid method formula for integral approximation:

$$\int_a^b f(x)\mathrm{d}x = h\left(\frac{f(a)+f(b)}{2}\sum_{1}^{n-1} f(a+i\times h)\right)$$

where $\boldsymbol{n}$ = number of subintervals for interval $[\boldsymbol{a}, \boldsymbol{b}]$; $\boldsymbol{h}$ = size of a subinterval (aka step)

## Pointers to functions. Example: integrating a function using the trapezoid method

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
double f(double x)
{
    return (3*x*x +1); /* the function to integrate; can be any function f(x) */
}
/* Computes an integral using the trapezoid approximation method */
double integral(double a, double b, int n, double(*p)(double x))
{
    int i;
    double h,s;
    h=(b-a)/n;
    s=((*p)(a)+(*p)(b))/2.0;
    for(i=1;i<n;i++) s+=(*p)(a+i*h);
    s=s*h;
    return s;
}
void main()
{
    double a,b;
    int n;
    char ch;
    printf("\na=");scanf("%lf",&a);
    printf("\nb=");scanf("%lf",&b);
    ch='y';
    while (ch=='Y' || ch=='y')
    {
          printf("\nn="); scanf("%d",&n); /* n is the number of sub-intervals */
          printf("\nFor n=%d the value of the integral is %lf", n, integral(a, b, n, f));
system("PAUSE");
          printf("\nNew value for n? [Yes=Y/y NO=any other character] ");
          ch=getch();
    }
}
```

## Recursion

```
procedure ODSN
begin One dark and stormy night,
      Three men sat in a cave,
      And one said to another,
      Dick, tell us a tale,
      And this is how the tale began:
      ODSN    -- again!
end
```

## Recursion

- *Recursion*: a powerful, general-purpose programming technique, where one function calls itself.
- The idea of recursion is at the heart of *mathematical induction*, a powerful technique for proving mathematical statements
- Programmers use recursion to decompose a large problem into one or more smaller ones, using solutions to the subproblems to solve the original problem.
- A recursive function is a function that calls itself, either
  - *directly* (contains a call to itself) or
  - *indirectly* (contains a call to another function which, in turn. calls the recursive function)
- With each call, the parameters and automatic variables are allocated on the stack in a new frame
- Static and global variables are allocated on the heap and preserve their locations

## Recursion. Stopping it

- If a function simply called itself as a part of its execution, then the called execution would itself call a further execution, which would itself call itself and there would be *no end* to the number of calls made. E.g.

  **Recursion** /ree-ker'-zhon/: See *Recursion*.

  - This situation is known as *infinite recursion* and is similar to the fault in indefinite loops where a faulty loop control condition causes the loop to iterate infinitely.
  - To avoid infinite recursion, the recursive function has to be carefully constructed to ensure that at some stage the function *terminates* without calling itself.
- In practice recursion *depth* must be kept shallow because each recursive call places on the stack:
  - function parameters
  - automatic local variables
  - return address

# Recursion example. Factorial

- Factorial definition:

$$fact(n) = \begin{cases} 1 & \text{if} \quad n = 0 \\ n \times fact(n-1) & \text{if} \quad n > 0 \end{cases}$$

```
#include <stdio.h>
#include <stdlib.h>
long long Factorial(long long n){
   long long y;
   printf("\nUpon function entry n=%I64d", n);
    if (n<=0)
      return 1;
   else
   {
      y = Factorial(n-1) * n; /* &2=return address after Factorial(n-1) */
      printf("\nUpon recursive function exit n=%I64d", n);
      return y;
   }
}
int main() {
   long long n;
   printf("\nPlease input n: ");  scanf("%I64d", &n);
   printf("\nFactorial of %I64d is %I64d\n", n, Factorial(n));
   /* &1=return address after evaluation of Factorial(n) */
   return 0;
}
```

**Works for n <=20;**
**Why?**

## Recursion example. Factorial

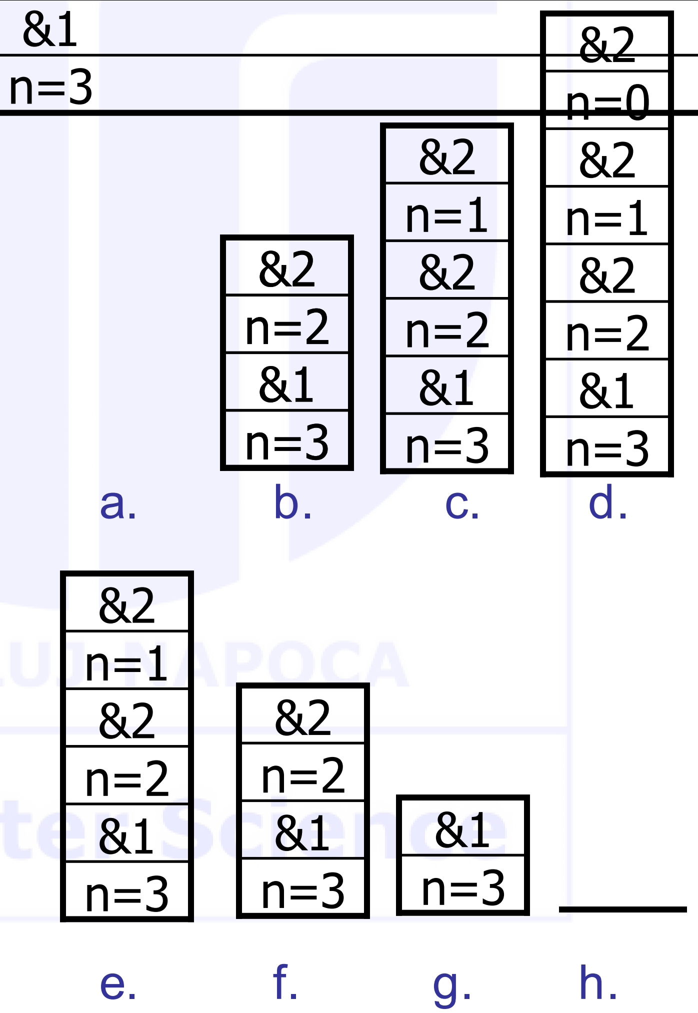
- Program output:

```
Please input n:  3

Upon function entry n=3
Upon function entry n=2
Upon function entry n=1
Upon function entry n=0
Upon recursive function exit n=1
Upon recursive function exit n=2
Upon recursive function exit n=3
Factorial of 3 is 6
```

# Recursion example. Factorial

- Stack evolution
  a. immediately after call to Factorial(3)
  b. after first call to Factorial(2) from itself
  c. after second call to Factorial(1) from itself
  d. after third call to Factorial(0) from itself
  e. after first exit from Factorial (for n= 0)
  f. after second exit from Factorial (for n= 1)
  g. after third exit from Factorial (for n= 2)
  h. after fourth exit from Factorial (for n= 3)



## Recursion. Linear recursion

- The simplest form of recursion is linear recursion.
- It occurs where an action has a simple repetitive structure consisting of some basic step followed by the action again.
- Can be turned into iteration:
  - The recursive step is placed within a loop and
  - The terminating condition is used to exit from the loop.

# Recursion example. String reverse

```
#include <stdio.h>
#include <stdlib.h>
/* Reads n words on one
   line, separated by
   space, and prints them
   reverted */
void reverse()
{
   char c;

   scanf("%c", &c);
   if (c!='\40')
   { /*space needed after
   each word */
       reverse();
       printf("%c", c);
   }
}
```

```
int main(int argc, char
   *argv[])
{
   unsigned int i, n;

   printf("Number of
   words="); scanf("%u",
   &n);
   for (i = 1; i <= n; i++)
   {
       reverse();
       puts("");
   }
   system("PAUSE");
   return 0;
}
```

## Recursion example. Smallest of *n* integers

```
#include <stdio.h>
#include <stdlib.h>

#define MAXN 100
#define MAXIMUM 32767

int string[MAXN];

int minimum(int x, int y)
{
   return ( x <= y)? x: y;
}

int minimal_element(int
   string_size)
{
   if (string_size >= 0)
      return
   minimum(string[string_size],
   minimal_element(string_size -
   1));
   else
      return MAXIMUM;
}
```

```
int main(int argc, char *argv[])
{
   unsigned int i, n;
   n = MAXIMUM;
   do
   {
      printf("\nPlease input the number
   of string elements, n=");
     scanf("%u", &n);
   }
   while (n > MAXN);
   printf("Please input the elements of
   the string\n");
   for (i = 0; i < n; i++)
   {
      printf("element_%u=", i+1);
      scanf("%d", &string[i]);
   }
   printf("\nYour string is:\n");
   for (i = 0; i < n; i++)
   {
      printf("\n%6d", string[i]);
      if ((i + 1) % 10 == 0) puts("");
   }
   printf("\nThe smallest element is
   %d\n", minimal_element(n - 1));
return 0;
}
```

## Recursion example. Fibonacci numbers

```c
#include <stdio.h>
/* Recursive version */
long fibR(int n)
{
   if (n == 0) return 0;
   if (n == 1) return 1;
   else return fibR(n-1) + fibR(n-2);
}
/* Iterative version */
long fibNR(int n)
{
   int i, x, y, z;
   if (n == 0) return 0;
   if (n == 1) return 1;
      x = 1; y = 0;
   for (i = 2; i <= n; i++)
   {
      z=x;  x += y; y = z;
   }
   return x;
}
```

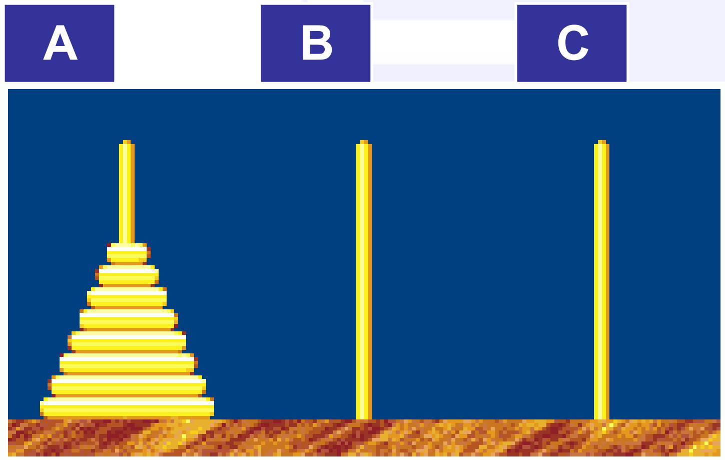
```c
int main()
{
  int n;
   char ch;

   for (ch = 'Y'; ch== 'Y' || ch ==
   'y'; ch = getchar())
   {
     printf("\nPlease input n=");
     scanf("%d%*c", &n);
     printf("\nRecursive computation
   fib(%d)=%ld\n", n, fibR(n));
     printf("\nIterative computation
   fib(%d)=%ld\n", n, fibNR(n));
     printf("Continue [Y/N]? ");
   }
   return 0;
}
```

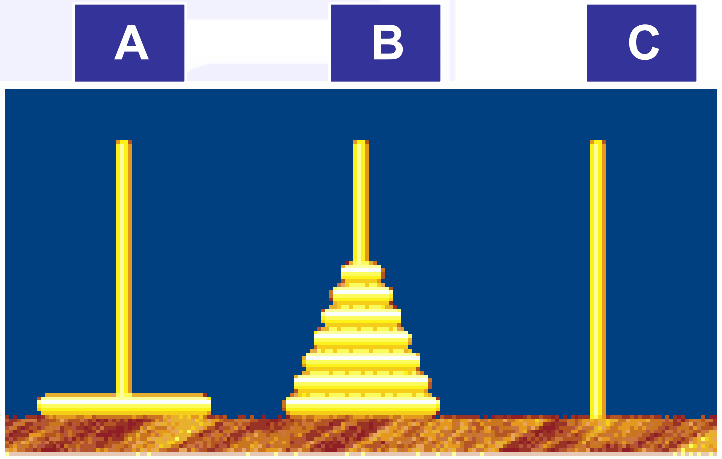
## Recursion example. Towers of Hanoi.

- Given a stack of disks arranged from largest on the bottom to smallest on top placed on a rod, together with two empty rods, the towers of Hanoi puzzle asks for the minimum number of moves required to move the stack from one rod to another, where moves are allowed only if they place smaller disks on top of larger disks.
  - The tower of Hanoi (commonly also known as the "towers of Hanoi"), is a puzzle invented by E. Lucas in 1883.
  - The puzzle with *n*=4 pegs and *n* disks is sometimes known as Reve's puzzle.

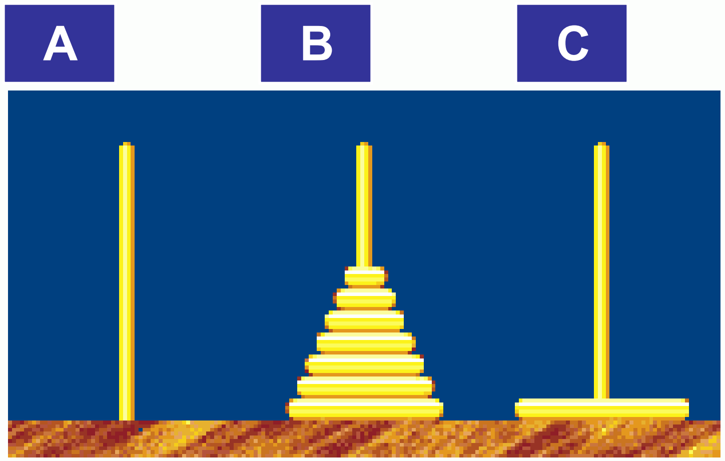
# Towers of Hanoi solution.

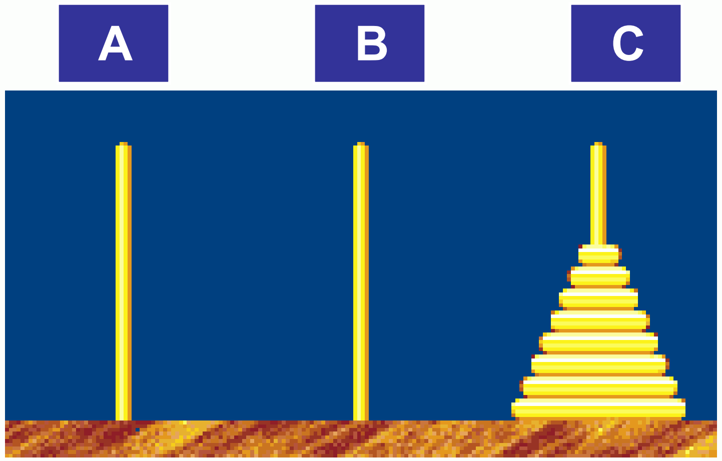


Move n-1 smallest disks to peg B



Move largest disk to peg C





Move n-1 smallest disks to peg C.

# Recursion example. Towers of Hanoi.

```c
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
enum {FROM, TO, USING};
void TH(int N, int from, int to,
   int using)
{
  if (N > 0)
   {
     TH(N-1, from, using, to);
     printf ("move %c --> %c\n",
    (char)('A'+from),
    (char)('A'+to));
     TH(N-1, using, to, from);
   }
}
```

```c
int main (int argc, char **argv)
{
  long int N;

  if (argc != 2)
   {
    fprintf(stderr, "usage: %s N\n",
   argv[0]);
    exit(1);
   }
  N = strtol(argv[1], (char **)NULL,
   10);
  if (N == LONG_MIN || N == LONG_MAX
   || N <= 0)
   {
    fprintf(stderr, "illegal value
   for number of disks\n");
    exit(2);
   }
  TH(N, FROM, TO, USING);
   system("Pause");
   return(0);
}
```

# Towers of Hanoi trace for 5 disks

move A --> B

move A --> C

move B --> C

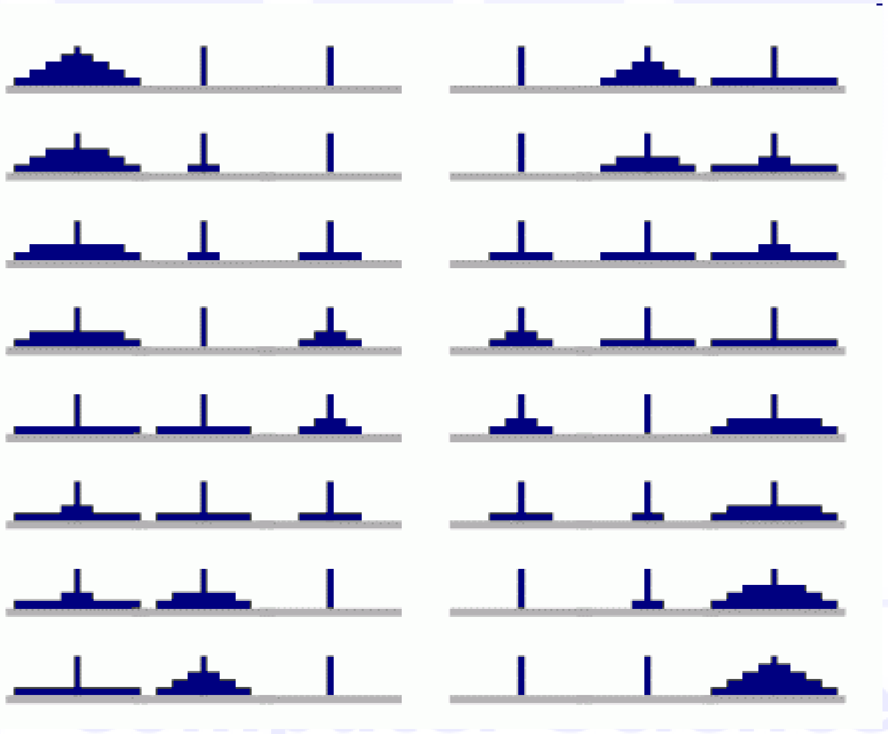
move A --> B

move C --> A

move C --> B

move A --> B

move A --> C



move C --> B

move A --> B

move A --> C

move B --> C

move A --> B

move C --> A

move C --> B

move A --> B

## Recursion pitfalls

What's wrong with the following code?

```
void print_number(int n)
{
   print_number(n / 10);
   printf("%d\n", (n % 10));
}

void print_again(int n)
{
    if (n < 10)
       printf("%d\n", n);
    else
    {
       print_again(n);
       printf("%d\n", (n % 10));
    }
}
```

## Recursion advice

- Use a base case to avoid an *infinite loop*.
- Functions can call other functions including themselves.
- It's easy to write spectacularly *inefficient* recursive programs if you're not careful.
- Recursion is an alternate to **for** and **while** loops for performing repetitive computations.
- Recursive functions are especially useful when either the underlying data structure (binary trees, Unix directories) or the underlying algorithmic paradigm (divide-and-conquer) are naturally self-referential.

## Recursive functions checklist

- Need to make sure your recursive function has three properties:

  1) No infinite recursion. Identify stopping case, ensure that chain of function calls will eventually lead to stopping case.

  2) Each stopping case either:

     - returns the correct value for that case (e.g., n^0 = 1) OR
     - performs correct action (void function)

  3) For cases that involve recursion, if recursive call performs correctly (or returns correct value), then final calculation/action is correct.

## Reading

- Deitel: 7.12 & 5.14..5.16
- Prata: chapter 17 (esp. section Functions and Pointers) chapter 9 (esp. section Recursion)
- King: 17.7 & 9.6

## Summary

- Pointers to functions
  - Declaring , using, comparing and casting pointers to functions
  - Examples
- Recursion
  - Mechanism
  - Examples:
    - String reverse
    - Smallest of *n* integers
    - Fibonacci numbers
    - Towers of Hanoi