Finishing up with Functions

This Class

- Nested function calls
- Predefined functions
- Macros
- Argument passing
- Recursion

Scope of a Name

- Two variables can have the same name only if they are declared in separate scopes
- A variable can not be used outside its scope
- C program has
  - function/**block** scope
  - **file** scope
  - **global**/external scope

Local Variable

- Always declared within a function
- Also called *automatic variable*

  auto int a;
- The assigned value is lost when the function is exited
- Could also be declared in a single compound statement

Global Variable

- Variable declared outside every function definition
- Can be accessed by all functions in the program that follow the declaration
- Also called *External* variable
- What if a variable is declared inside a function that has the same name as a global variable?
  - The global variable is "shadowed" inside that particular function only

Global Variables

```
#include<stdio.h>
int g=10, h=20;

int add(){
  return g+h;
}

void fun1(){
  int g=200;
  printf("%d\n",g);
}

int main(){
  fun1();
  printf("%d %d %d\n",
          g, h, add());
  return 0;
}
```

```
200
10 20 30
```

2-18

1. The variable g and h have been defined as global variables.
2. The use of global variables is normally discouraged. Use local variables of functions as much as possible.
3. Global variables are useful for defining constants that are used by different functions in the program.

Global Variables: Example

```
const double PI = 3.14159;
double circum_of_circle(double r) {
   return 2 * PI * r; }
double area_of_circle (double r) {
   return PI * r * r;
}
```

defines PI to be of type double with value 3.14159. Qualified by const, which means that PI is a constant. The value inside the box associated with PI cannot be changed anywhere.

Constants via #define

```
#define PI  3.14159
double circum_of_circle(double r) {
   return 2 * PI * r; }
double area_of_circle (double r) {
   return PI * r * r;
}
```

replaces PI by the constant 3.14159 everywhere

#define *name replacement-text*

Static Variables

- We have seen two kinds of variables: local variables and global variables.
- There are static variables too.

```
int f () {
   static int ncalls = 0;
   ncalls  = ncalls + 1;
/* track the number of
times f() is called  */
   … body of f() …
}
```

- Use a local variable?
  - gets destroyed every time f returns
- Use a global variable?
  - other functions can change it! (dangerous)

GOAL: count number of calls to f()

SOLUTION: define ncalls as a static variable inside f().

It is created as an integer box the first time f() is called.

Once created, it **never** gets destroyed, and **retains its value across invocations of f().**

It is like a global variable, but visible only within f().

Static variables are not allocated on stack. So they are not destroyed when f() returns.

Macros

- Everything you write outside functions is a command to the C pre-processor
- All directives to pre-processor begin with #
- All other pre-processor statements involve using #define
  - To define macros and constants
- A C macro is a rule that maps an input sequence to an output sequence
  - Before the program has compiled

Macros

- Object-like macros
  - #define BUFFER_SIZE 1024
  - Pre-defined macros in C, e.g. _DATE_ , _TIME_ etc.
    - printf("%s\n", __DATE__);
    - printf("%s\n", __TIME__);
- Function-like macros
  - #define min(X, Y) ((X) < (Y) ? (X) : (Y))
    - x = min(a,b) will be expanded to

      x = ((a) < (b) ? (a) : (b))

    Before compilation

Macro Pitfalls

- Macro is a simple copy-paste
- Without parentheses, can make operator precedence betray your code’s logic

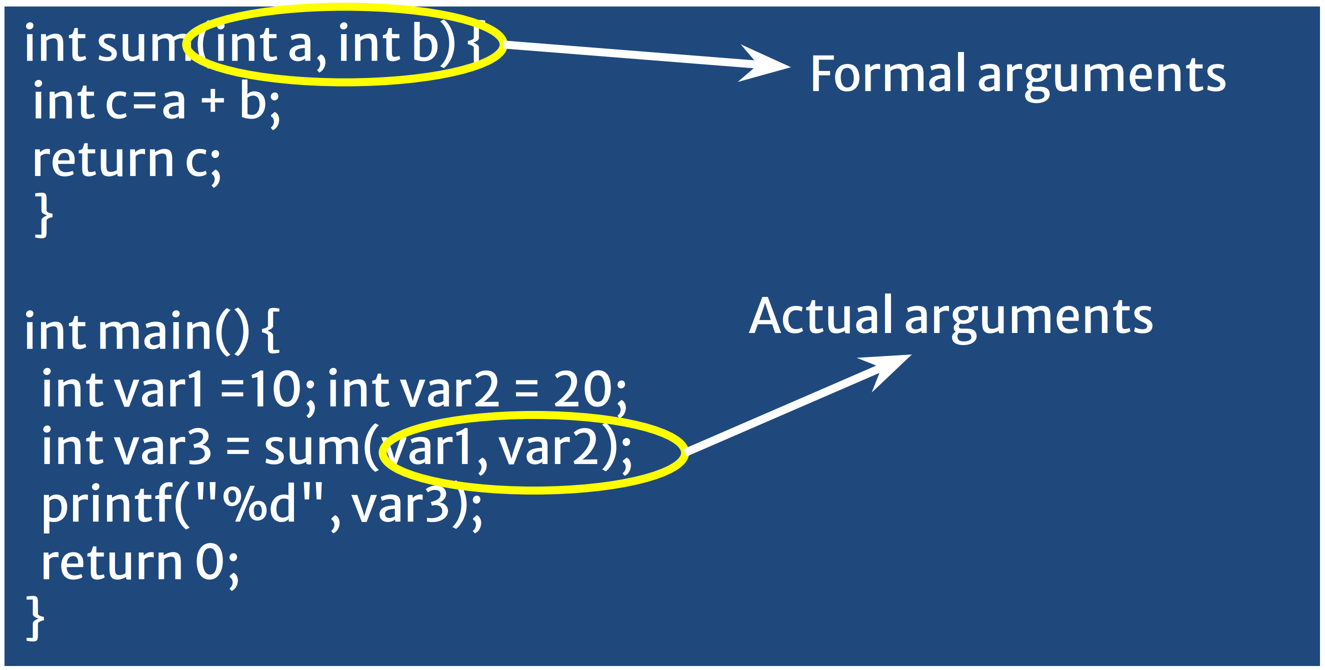
```
#define SQR(x) (x*x)

int main() {
 int a, b=3;
 a = SQR(b+5); // (b+5*b+5)
 printf("%d\n", a);
 return 0;
}
```

```
23
```

Argument Passing: Some Details

- Formal argument
  - What you put in the function definition
- Actual argument
  - What you put in the function call



```
int sum(int a, int b) {
 int c=a + b;
 return c;
 }

int main() {
  int var1 =10; int var2 = 20;
  int var3 = sum(var1, var2);
  printf("%d", var3);
  return 0;
}
```

Passing Parameters by Value

- Copy of variable is created in the function’s local scope
  - All changes are made to this local variable
  - Global variable retains its original value

```
int increment(int var) {
  var = var+1;
  return var;
}

int main() {
  int num1=20;
  int num2 = increment(num1);
  printf("num1 value is: %d", num1);
  printf("num2 value is: %d", num2);
  return 0;
}
```

```
20
21
```

Another Example

- What will this program output?

```
int main( ) {
  int num1 = 35, num2 = 45 ;
  printf("Before swapping: num1 = %d \t num2 = %d", num1, num2);
  swapnum ( num1, num2 ); //calling swap function
  printf("After swapping: num1 = %d  and num2 = %d”, num1, num2);
}

void swapnum ( int var1, int var2 ) {
  int tempnum ;
  tempnum = var1 ;
  var1 = var2 ;
  var2 = tempnum ;
}
```

```
Before swapping: num1 = 35 and num2 = 45
After swapping: num1 = 35 and num2 = 45
```

Passing Parameters by Address

- *Value* of actual argument not passed to formal argument
- *Address* of actual argument passed to formal argument
- Updates value of global variable

An Example

- Remember: & means ‘address of’
- New concept: * means ‘value from’

```
int increment(int *var) {
  *var = *var+1;
   return *var;
}

int main() {
  int num1=20; int num2 = increment(&num1);
  printf("num1 value is: %d", num1);
  printf("num2 value is: %d", num2);
  return 0;
}
```

```
21
21
```

Value from what?

Passing Arguments by Address

- Use pointers as formal arguments
  - Telling compiler you will be passing a memory address, not a value
  - Pass address using *address of* operator (&) during function call

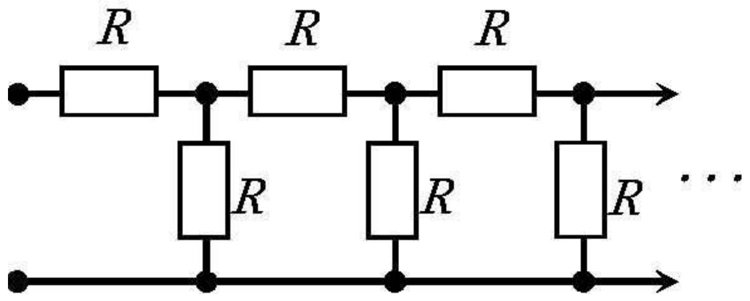
Swap Program Corrected

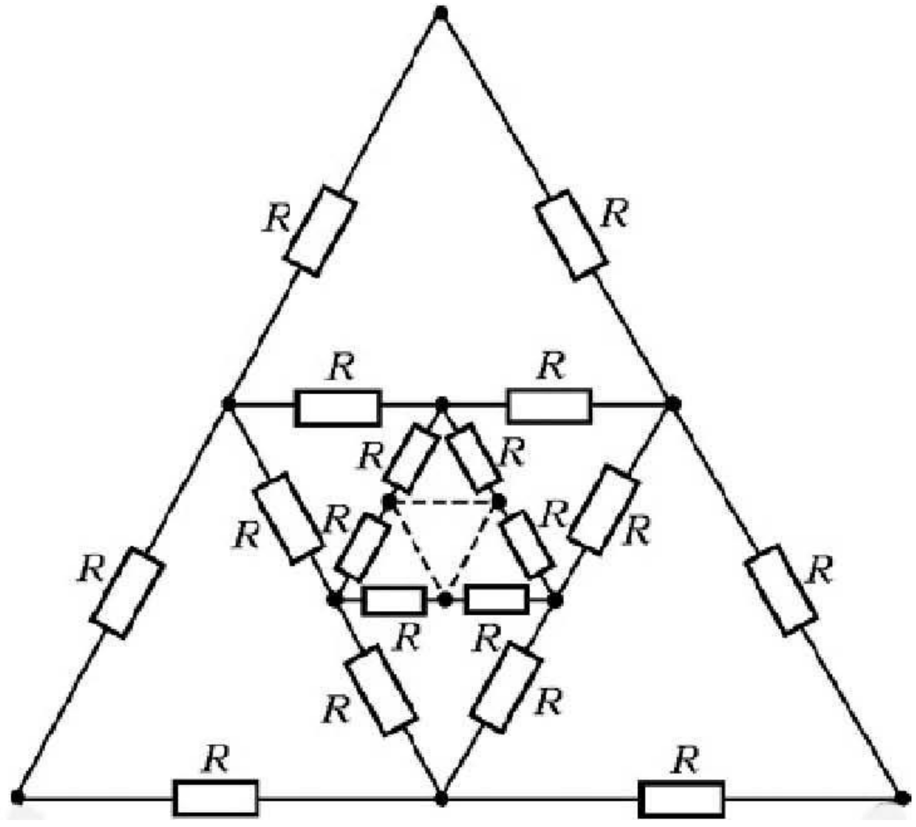
```
int main( ) {
  int num1 = 35, num2 = 45 ;
  printf("Before swapping: num1 = %d and num2 = %d", num1, num2);
  swapnum ( &num1, &num2 ); /*calling swap function*/
  printf("After swapping: num1 = %d and num2 = %d", num1, num2);
 }

void swapnum ( int *var1, int *var2 ) {
  int tempnum ;
 tempnum = *var1 ;
 *var1 = *var2 ;
 *var2 = tempnum ;
 }
```

```
Before swapping: num1 =35 and num2 = 45
After swapping: num1 =45 and num2 = 35
```

Recursion





```
1   #include <stdio.h>
2   int max(int a, int b) {
3      if (a > b)
4          return a;
5     else
6          return b;
7   }

8   int main () {
9         int x;     10a
10        x =         max(6, 4);
11        printf(“%d”,x);
12        return 0;
13  }
```

- Steps when a function is called: max(6,4) in step 10a.
- Allocate space for (i) return value, (ii) store return address and (iii) pass parameters.

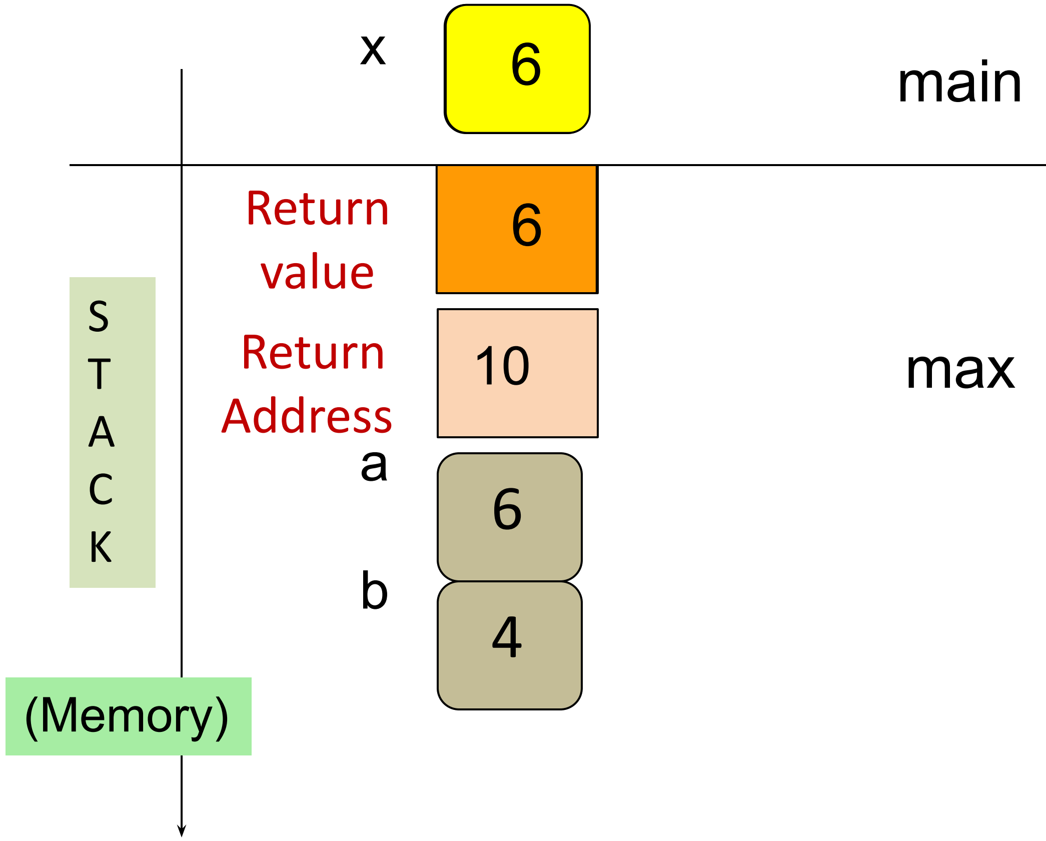
1. Create a box informally called ``Return value’’ of same type as the return type of function.
2. Create a box and store the location of the next instruction in the calling function (main)—return address. Here it is 10 (why not 11?). Execution resumes from here once function terminates.
3. Parameter Passing- Create boxes for each formal parameter: a, b here. Initialize them using actual parameters, 6 and 4.

```
1   #include <stdio.h>
2   int max(int a, int b) {
3      if (a > b)
4         return a;
5      else
6         return b;
7   }

8   int main() {
9         int x = -1;
10        x = max(6, 4);
11        printf("%d",x);
12        return 0;
13  }
```

Calling max(6,4):

1. Allocate space for return value.
2. Store return address (10).
3. Pass parameters.



After completing max(), execution in main() will re-start from address 10.

Recursion

- A function defined by calling itself, *directly* or *indirectly,* is called a *recursive function.*
  - The phenomenon itself is called recursion
- Examples:
  - Factorial:

```
0! = 1
n! = n * (n-1)!
```

  - Even and Odd:

```
Even(n) = (n == 0) || Odd(n-1)
Odd(n)  = (n != 0) && Even(n-1)
```

Recursive Functions: Properties

- The arguments change between the recursive calls

5! = 5 * 4! = 5 * 4 * 3! = …

- Change is towards a case for which solution is known (base case)
- There must be one or more base cases

0! is 1

Odd(0) is false

Even(0) is true

Example: Factorial

- Write a program to print the factorial of an integer *n*
- Can do it using loops
  - To calculate the factorial of n, you calculate all the factorials in between

```
long int factorial(int n){
     int i;
     long int fact = 1;
     for (i=1; i<=n; i++){
        fact = fact * i;
    }
    return fact;
}
```

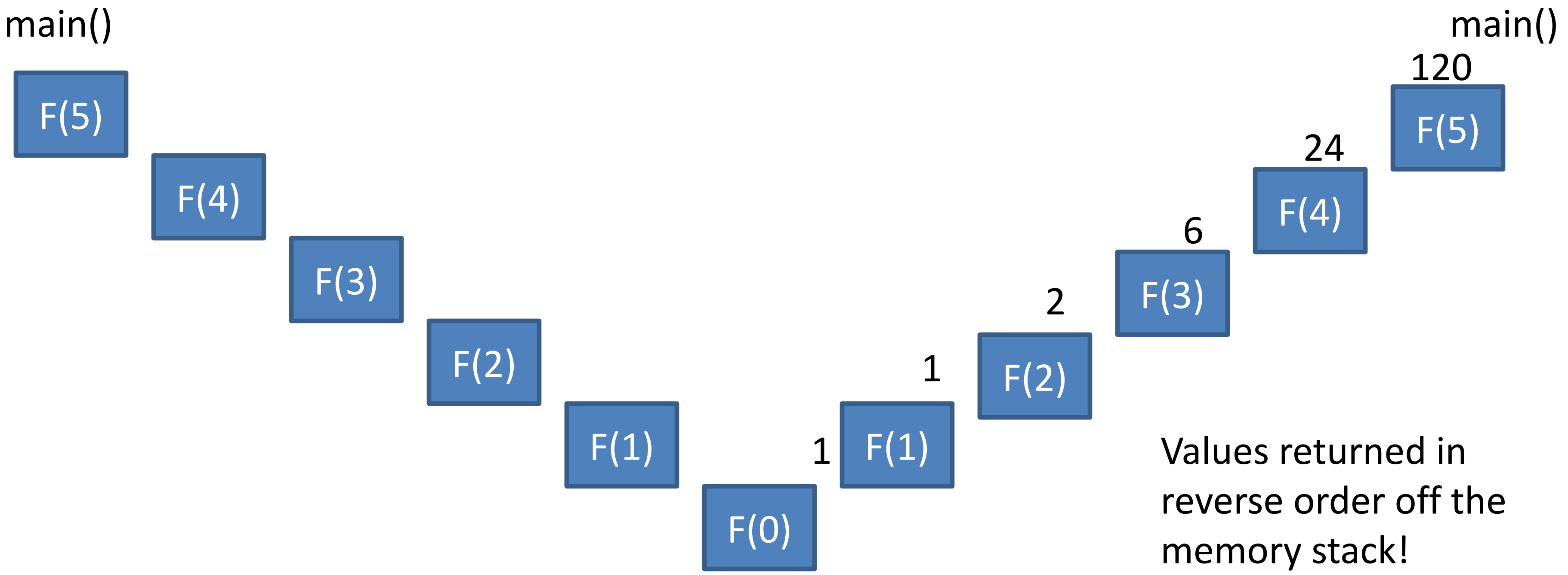
Example: Factorial

- Write a recursive program to print the factorial of an integer *n*
- Process
  - N! = N x (N-1)! *(recursion)*
  - 0! = 1 (*base case*)

```
long int factorial(int n){
    if(n==0)
        return 1;
    else
        return(n*factorial(n-1));
}
```

Tracing the flow of control

```
long int factorial(int n){
    if(n==0)
        return 1;
    else
        return(n*factorial(n-1));
}
```



Values returned in reverse order off the memory stack!

Recursion and Induction

When programming recursively, think inductively

- Mathematical induction for the natural numbers
- Structural induction for other recursively-defined types (to be covered later!)

Recursion and Induction

When writing a recursive function

- Write down a clear, concise *specification* of its behavior
- Give an *inductive proof* that your code satisfies the specification

Fibonacci Numbers

Fibonacci relationship

$F_1 = 1$

$F_2 = 1$

$F_3 = 1 + 1 = 2$

$F_4 = 2 + 1 = 3$

$F_5 = 3 + 2 = 5$

In general :

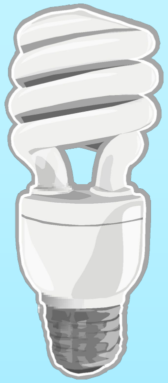
$$F_n = F_{n-1} + F_{n-2}$$

or

$$F_{n+1} = F_n + F_{n-1}$$

Recursion vs Iteration

```
int fib(int n)
{
   int first = 0, second = 1;
   int next, c;
   if (n <= 1)
       return n;
   for ( c = 1; c < n ; c++ ) {
       next = first + second;
       first = second;
       second = next;
    }
    return next;
}
```

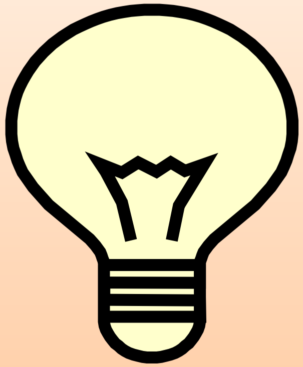


2-18

The recursive program is closer to the definition and easier to read.

But very very inefficient

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



Recursive fib

fib(5)

fib(4) + fib(3)

fib(3) + fib(2)   fib(2) + fib(1)

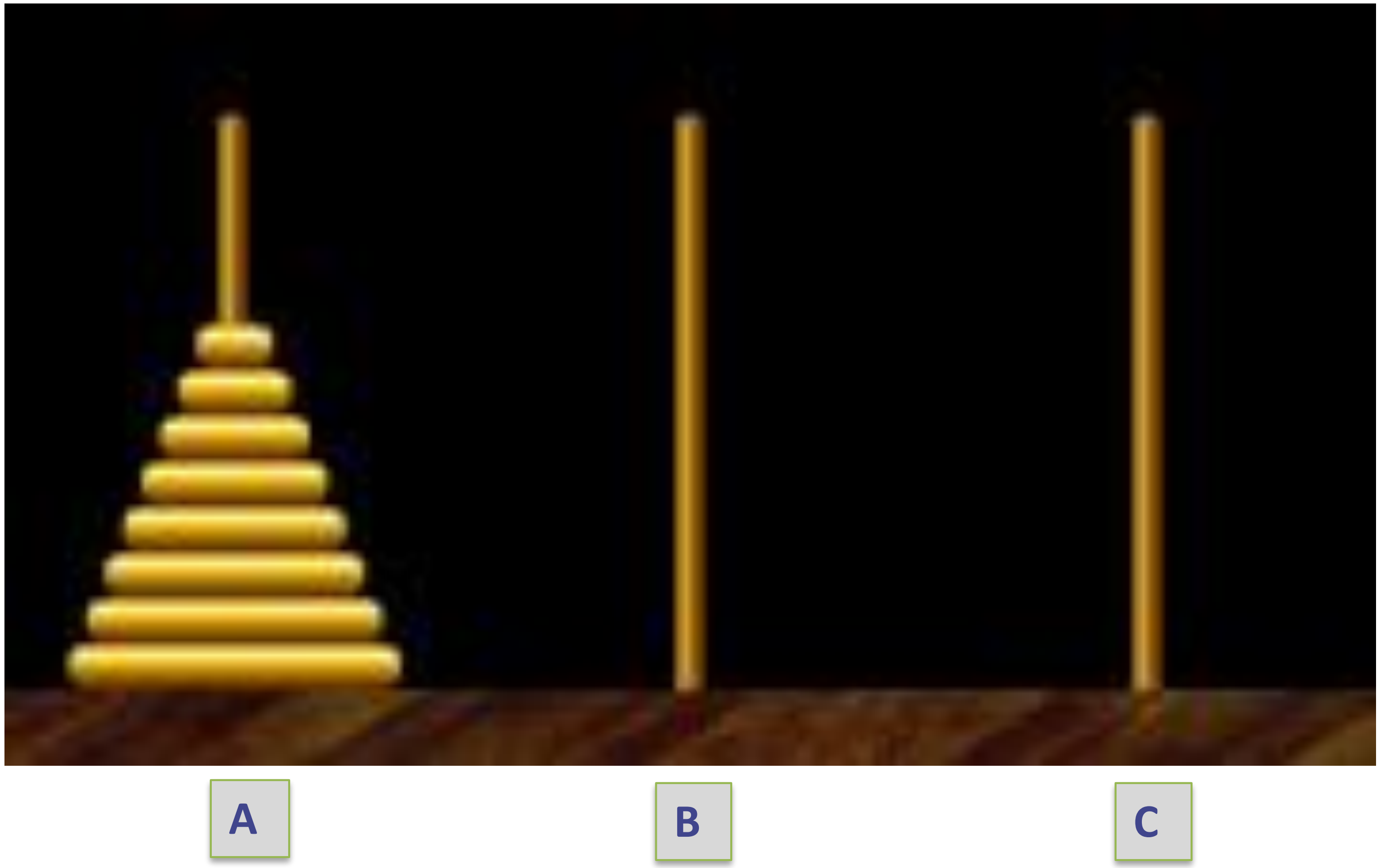
fib(2) + fib(1)   fib(1) + fib(0)   fib(1) + fib(0)

fib(1) + fib(0)

Recursion: Pros and Cons

- Pros
  - Very intuitive
  - Very compact in terms of code length
  - Very cool to look at
- Cons
  - Slower usually
  - Stack overflow
  - Harder to debug

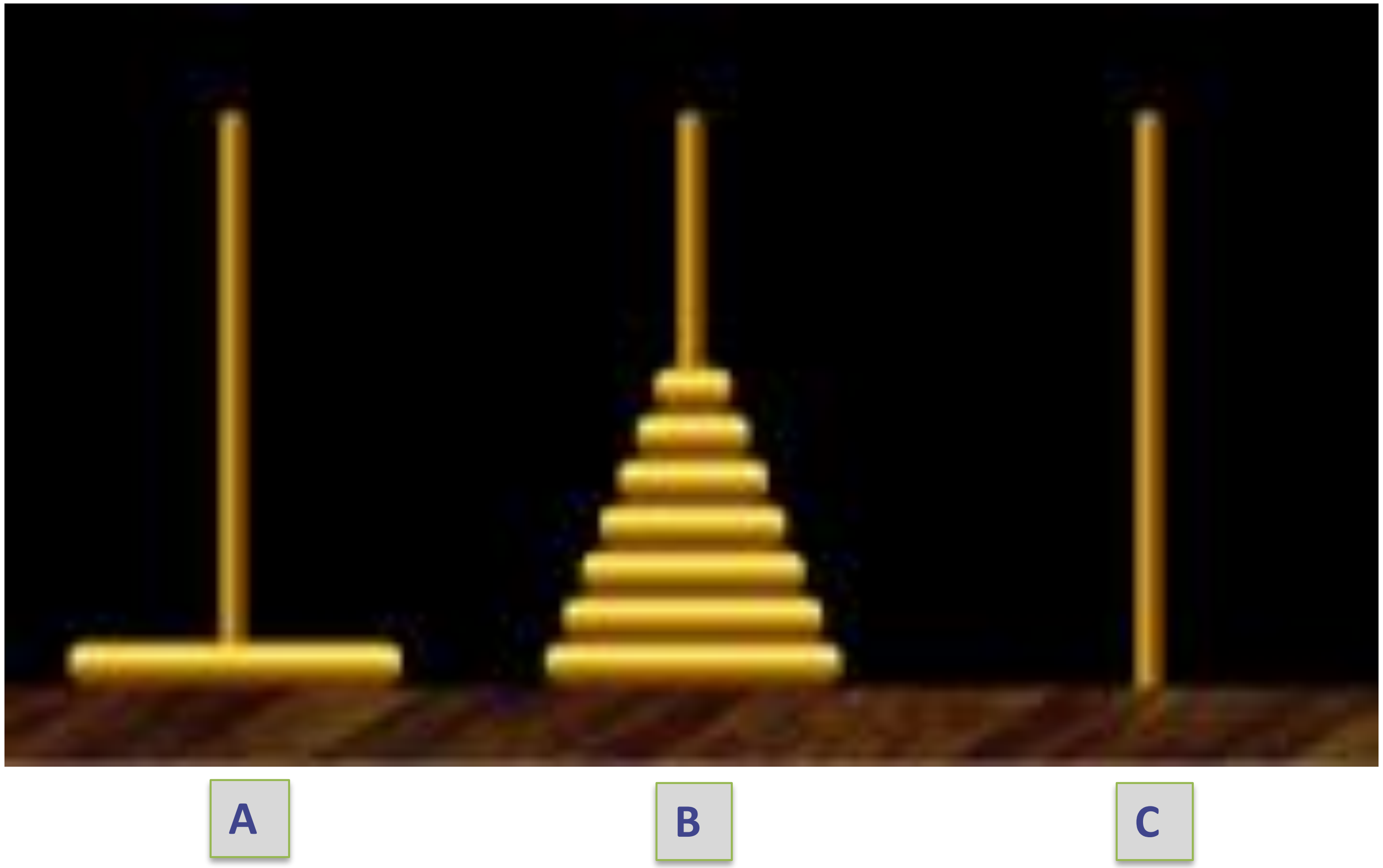
Recursion : Tower of Hanoi



No disk may be placed on top of a smaller disk.

Image Source: http://www.comscigate.com/cs/IntroSedgewick/20elements/27recursion/index.html

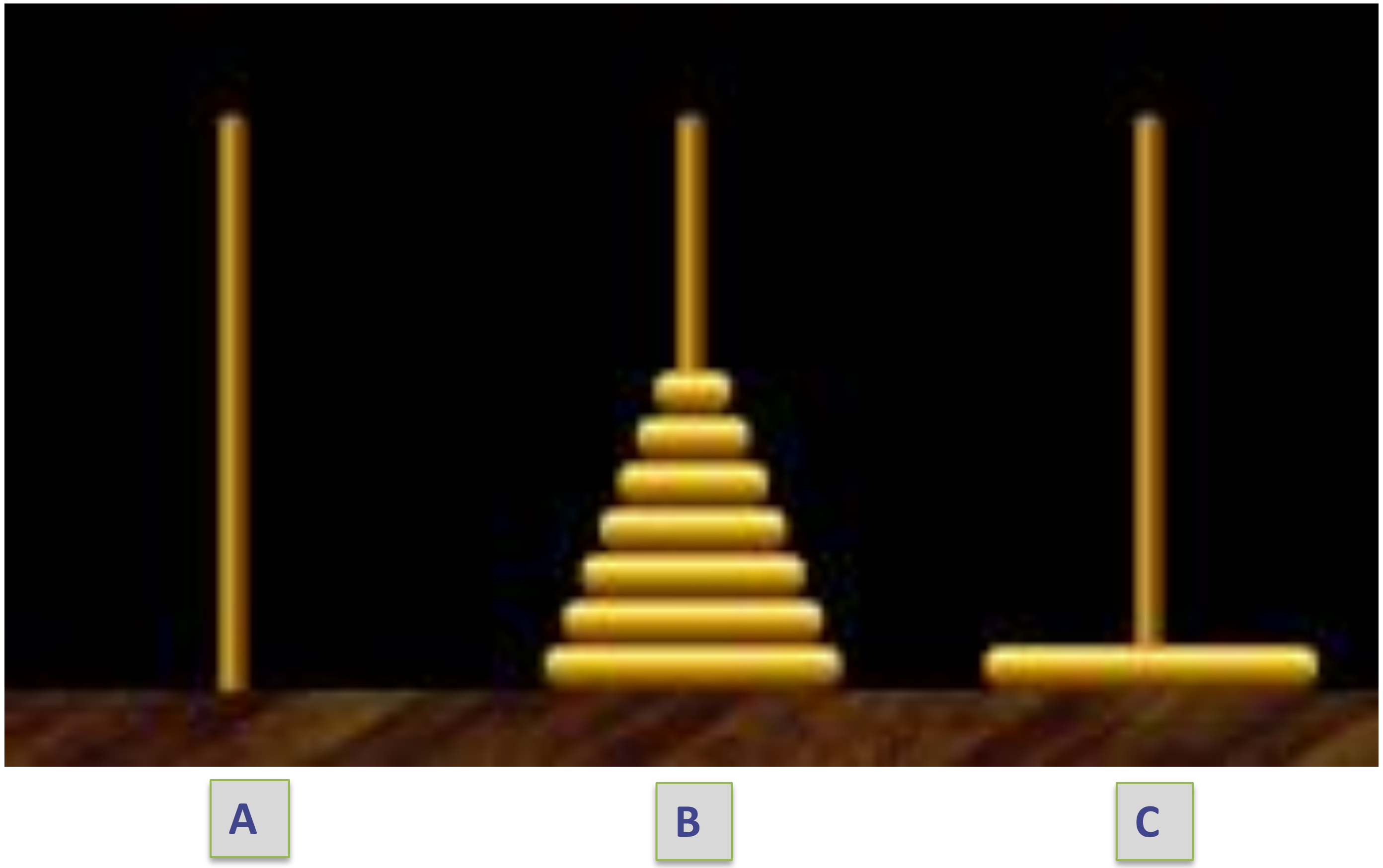
Recursion : Tower of Hanoi ..2



No disk may be placed on top of a smaller disk.

Image Source: http://www.comscigate.com/cs/IntroSedgewick/20elements/27recursion/index.html

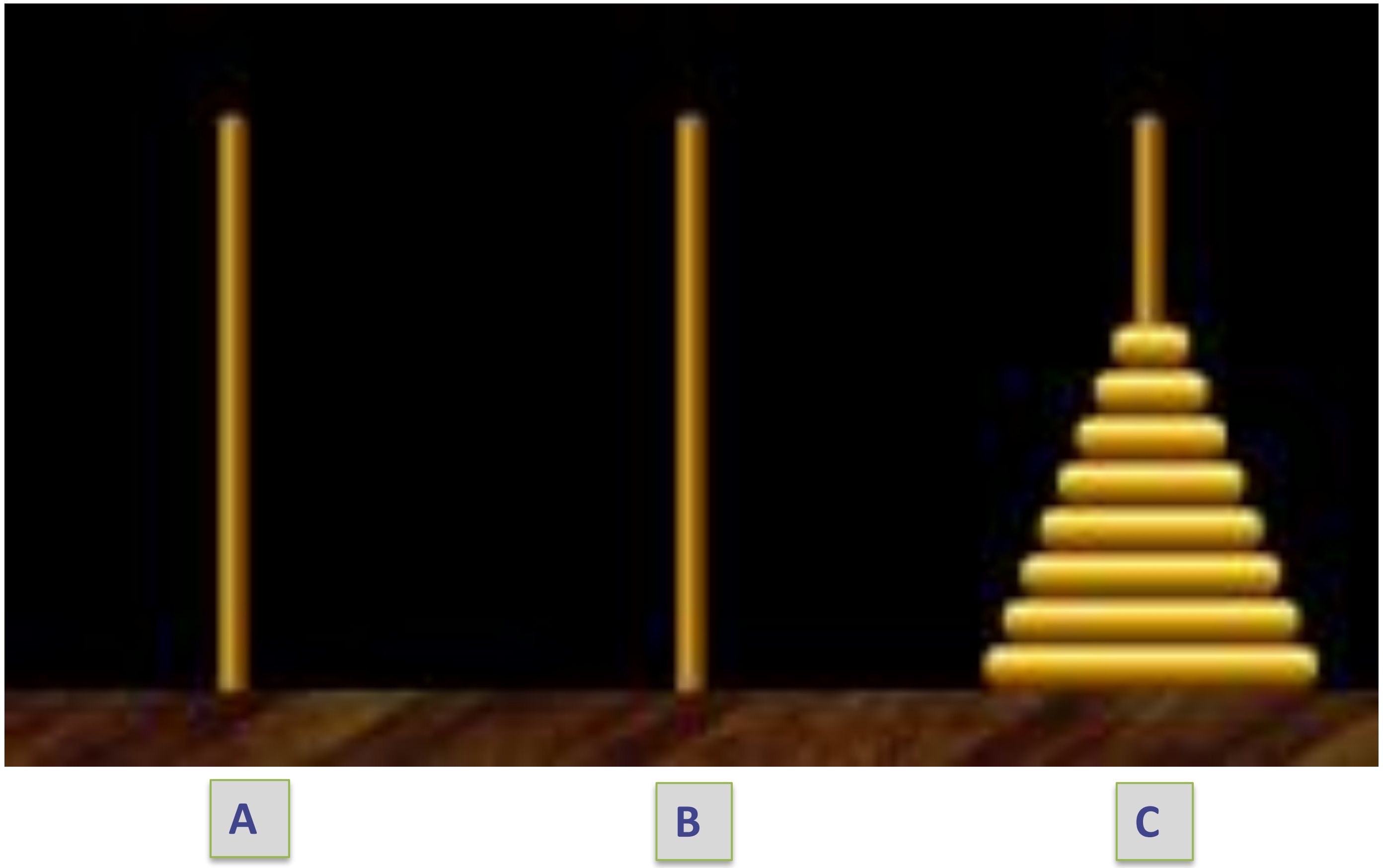
Recursion : Tower of Hanoi ..3



No disk may be placed on top of a smaller disk.

Image Source: http://www.comscigate.com/cs/IntroSedgewick/20elements/27recursion/index.html

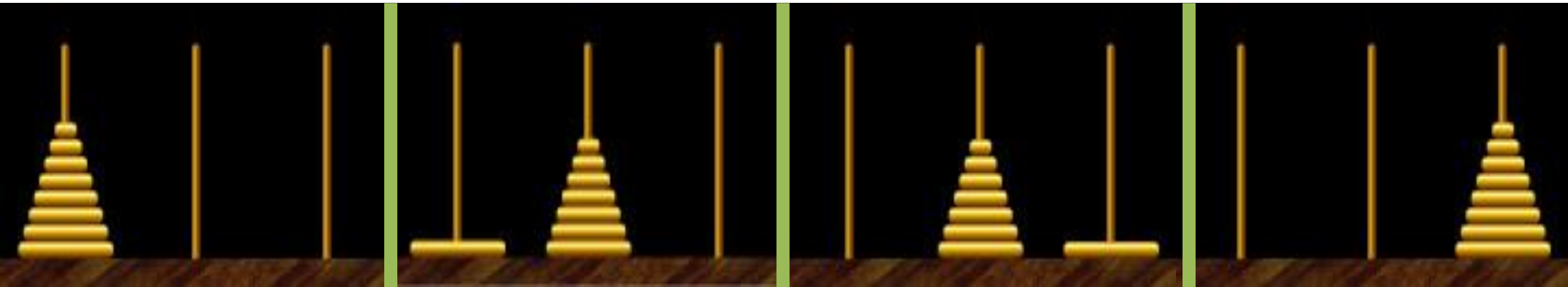
Recursion : Tower of Hanoi ..4



No disk may be placed on top of a smaller disk.

Image Source: http://www.comscigate.com/cs/IntroSedgewick/20elements/27recursion/index.html

```
// move n disks From A to C using B as aux
void hanoi(int n, char A, char C, char B) {
  if (n==0) { return; } // nothing to move!!
   // recursively move n-1 disks
   // from A to B using C as aux
   hanoi(n-1, A, B, C);
   // atomic move of a single disk
   printf("Move 1 disk from %c to %c\n", A, C);
   // recursively move n-1 disks
   // from B to C using A as aux
   hanoi(n-1, B, C, A);
}
```



OUTPUT for hanoi(4, ‘A’, ‘C’, ‘B’)

```
Move 1 disk from A to B
Move 1 disk from A to C
Move 1 disk from B to C
Move 1 disk from A to B
Move 1 disk from C to A
Move 1 disk from C to B
Move 1 disk from A to B
Move 1 disk from A to C
Move 1 disk from B to C
Move 1 disk from B to A
Move 1 disk from C to A
Move 1 disk from B to C
Move 1 disk from A to B
Move 1 disk from A to C
Move 1 disk from B to C
```

Image Source: http://upload.wikimedia.org/wikipedia/commons/6/60/Tower_of_Hanoi_4.gif

The puzzle was invented by the French mathematician **Édouard Lucas** in 1883.

There used to be a story about a temple in Kashi Vishwanath, which contains a large room with three posts surrounded by 64 golden disks. Brahmin priests have been moving these disks, in accordance with the immutable rules of the Brahma. The puzzle is therefore also known as the **Tower of Brahma** puzzle.

According to the legend, when the last move of the puzzle will be completed, the world will end.

If the legend were true, and if the priests were able to move disks at a rate of one per second, using the smallest number of moves, it would take them $2^{64}-1$ seconds or roughly 585 billion years or about 127 times the current age of the sun.

Source: https://en.wikipedia.org/wiki/Tower_of_Hanoi

Next Class

- Introduction to arrays
  - Syntax
  - Basic I/O
  - Simple usage