Introduction to Computers and Programming

Lecture 4 –
Array and function call
Chap 8 & 9

Tien-Fu Chen

Dept. of Computer Science and Information Engineering

National Yang Ming Chiao Tung Univ.

Chap 6 & 7

Type Definitions

- The `#define` directive can be used to create a "Boolean type" macro:

```
#define BOOL int
```

- There's a better way using a feature known as a ***type definition:***

```
typedef int Bool;
```

- `Bool` can now be used in the same way as the built-in type names.
- Example:

```
Bool flag;    /* same as int flag; */
```

Advantages of Type Definitions

- Type definitions can make a program more understandable.
- If the variables `cash_in` and `cash_out` will be used to store dollar amounts, declaring `Dollars` as

  ```
  typedef float Dollars;
  ```

  and then writing

  ```
  Dollars cash_in, cash_out;
  ```

  is more informative than just writing

  ```
  float cash_in, cash_out;
  ```

Advantages of Type Definitions

- Type definitions can also make a program easier to modify.
- To redefine `Dollars` as `double`, only the type definition need be changed:

```
typedef double Dollars;
```

- Without the type definition, we would need to locate all `float` variables that store dollar amounts and change their declarations.

Type Definitions and Portability

- Type definitions are an important tool for writing portable programs.
- One of the problems with moving a program from one computer to another is that types may have different ranges on different machines.
- If `i` is an `int` variable, an assignment like

  ```
  i = 100000;
  ```

  is fine on a machine with 32-bit integers, but will fail on a machine with 16-bit integers.

Type Definitions and Portability

- Instead of using the `int` type to declare quantity variables, we can define our own "quantity" type:

  ```
  typedef int Quantity;
  ```

  and use this type to declare variables:

  ```
  Quantity q;
  ```

- When we transport the program to a machine with shorter integers, we'll change the type definition:

  ```
  typedef long Quantity;
  ```

- Note that changing the definition of `Quantity` may affect the way `Quantity` variables are used.

Type Definitions and Portability

- The C library itself uses `typedef` to create names for types that can vary from one C implementation to another; these types often have names that end with `_t`.
- Typical definitions of these types:

```
typedef long int ptrdiff_t;
typedef unsigned long int size_t;
typedef int wchar_t;
```

- In C99, the `<stdint.h>` header uses `typedef` to define names for integer types with a particular number of bits.

The `sizeof` Operator

- The value of the expression

  `sizeof (` *type-name* `)`

  is an unsigned integer representing the number of bytes required to store a value belonging to *type-name*.
- `sizeof(char)` is always 1, but the sizes of the other types may vary.
- On a 32-bit machine, `sizeof(int)` is normally 4.

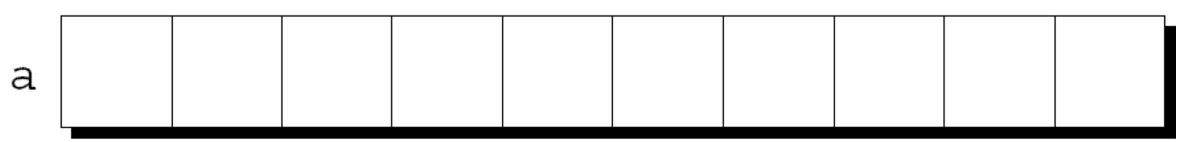
The `sizeof` Operator

- The `sizeof` operator can also be applied to constants, variables, and expressions in general.
  - If `i` and `j` are `int` variables, then `sizeof(i)` is 4 on a 32-bit machine, as is `sizeof(i + j)`.
- When applied to an expression—as opposed to a type—`sizeof` doesn't require parentheses.
  - We could write `sizeof i` instead of `sizeof(i)`.
- Parentheses may be needed anyway because of operator precedence.
  - The compiler interprets `sizeof i + j` as `(sizeof i) + j`, because `sizeof` takes precedence over binary `+`.

Array

One-Dimensional Arrays

- An ***array*** is a data structure containing a number of data values, all of which have the same type.
- These values, known as ***elements,*** can be individually selected by their position within the array.
- The simplest kind of array has just one dimension.
- The elements of a one-dimensional array `a` are conceptually arranged one after another in a single row (or column):



One-Dimensional Arrays

- To declare an array, we must specify the *type* of the array's elements and the *number* of elements:

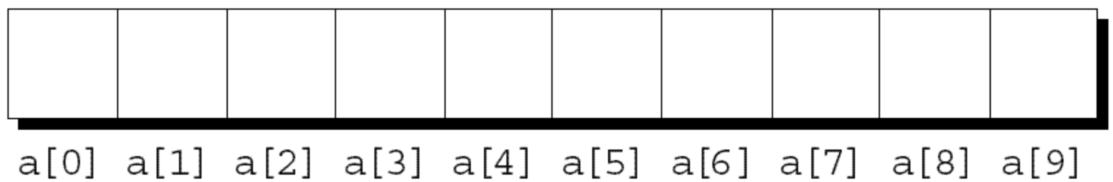
```
int a[10];
```

- The elements may be of any type; the length of the array can be any (integer) constant expression.
- Using a macro to define the length of an array is an excellent practice:

```
#define N 10
…
int a[N];
```

Array Subscripting

- This is referred to as ***subscripting*** or ***indexing*** the array.
- The elements of an array of length *n* are indexed from 0 to *n* – 1.



- Expressions of the form `a[i]` are lvalues, so they can be used in the same way as ordinary variables:

```
a[0] = 1;
printf("%d\n", a[5]);
++a[i];
```

Array Subscripting

- Many programs contain `for` loops whose job is to perform some operation on every element in an array.
- Examples of typical operations on an array `a` of length `N`:

```
for (i = 0; i < N; i++)
  a[i] = 0;                /* clears a */

for (i = 0; i < N; i++)
  scanf("%d", &a[i]);      /* reads data into a */

for (i = 0; i < N; i++)
  sum += a[i];             /* sums the elements of a */
```

Array Subscripting

- C doesn't require that subscript bounds be checked; if a subscript goes out of range, the program's behavior is undefined.
- A common mistake: forgetting that an array with *n* elements is indexed from 0 to *n* – 1, not 1 to *n*:

```
int a[10], i;

for (i = 1; i <= 10; i++)
  a[i] = 0;
```

  With some compilers, this innocent-looking `for` statement causes an infinite loop.

Array Subscripting with side effect

- Be careful when an array subscript has a side effect:

```
i = 0;
while (i < N)
  a[i] = b[i++];
```

- The expression `a[i] = b[i++]` accesses the value of `i` and also modifies `i`, causing undefined behavior.
- The problem can be avoided by removing the increment from the subscript:

```
for (i = 0; i < N; i++)
  a[i] = b[i];
```

Program: Reversing a Series of Numbers

- The `reverse.c` program prompts the user to enter a series of numbers, then writes the numbers in reverse order:

```
Enter 10 numbers: 34 82 49 102 7 94 23 11 50 31
In reverse order: 31 50 11 23 94 7 102 49 82 34
```

reverse.c

```
/* Reverses a series of numbers */

#include <stdio.h>

#define N 10

int main(void)
{
  int a[N], i;

  printf("Enter %d numbers: ", N);
  for (i = 0; i < N; i++)
    scanf("%d", &a[i]);

  printf("In reverse order:");
  for (i = N - 1; i >= 0; i--)
    printf(" %d", a[i]);
  printf("\n");
 }
```

Array Initialization

- If the initializer is shorter than the array, the remaining elements of the array are given the value 0:

```
int a[10] = {1, 2, 3, 4, 5, 6};
/* initial value of a is {1,2,3,4,5,6,0,0,0,0} */
```

- Using this feature, we can easily initialize an array to all zeros:

```
int a[10] = {0};
/* initial value of a is {0,0,0,0,0,0,0,0,0,0} */
```

There's a single 0 inside the braces because it's illegal for an initializer to be completely empty.

Array Initialization

- If an initializer is present, the length of the array may be omitted:

```
int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

- The compiler uses the length of the initializer to determine how long the array is.

Using the `sizeof` Operator with Arrays

- To avoid a warning, we can add a cast that converts `sizeof(a) / sizeof(a[0])` to a signed integer:

```
for (i = 0; i < (int) (sizeof(a) / sizeof(a[0])); i++)
  a[i] = 0;
```

- Defining a macro for the size calculation is often helpful:

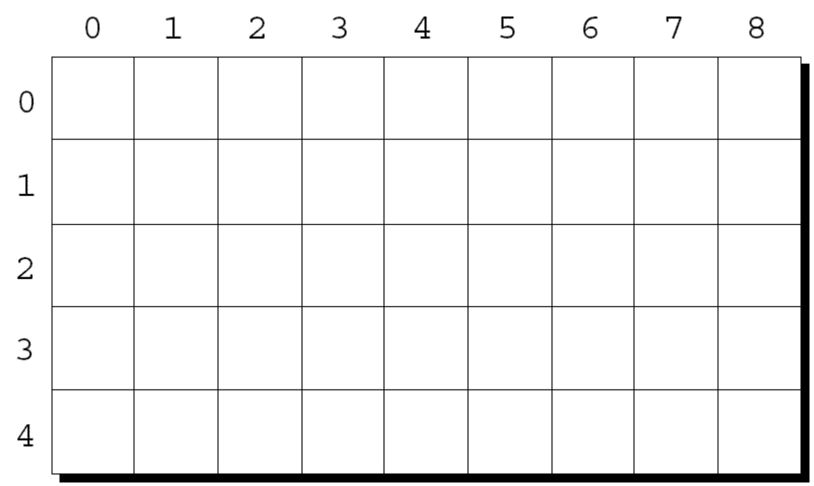
```
#define SIZE ((int) (sizeof(a) / sizeof(a[0])))

for (i = 0; i < SIZE; i++)
  a[i] = 0;
```

Multidimensional Arrays

- An array may have any number of dimensions.
- The following declaration creates a two-dimensional array (a *matrix*, in mathematical terminology):

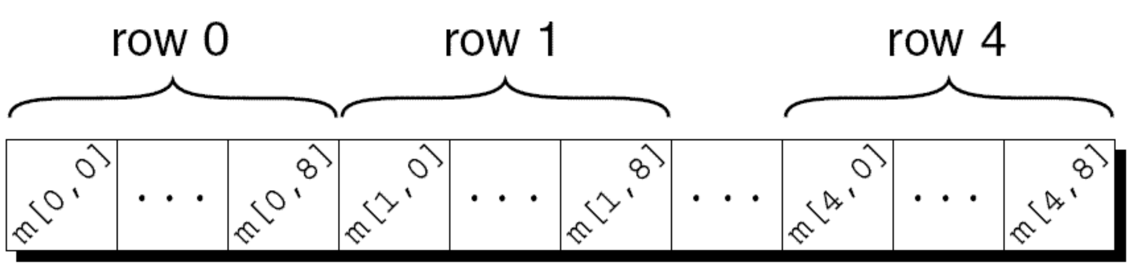
```
int m[5][9];
```

- `m` has 5 rows and 9 columns. Both rows and columns are indexed from 0:



Multidimensional Arrays

- Although we visualize two-dimensional arrays as tables, that's not the way they're actually stored in computer memory.
- C stores arrays in ***row-major order,*** with row 0 first, then row 1, and so forth.
- How the `m` array is stored:



Multidimensional Arrays

- Nested `for` loops are ideal for processing multidimensional arrays.
- Consider the problem of initializing an array for use as an identity matrix. A pair of nested `for` loops is perfect:

```
#define N 10

double ident[N][N];
int row, col;

for (row = 0; row < N; row++)
  for (col = 0; col < N; col++)
    if (row == col)
      ident[row][col] = 1.0;
    else
      ident[row][col] = 0.0;
```

Initializing a Multidimensional Array

- We can create an initializer for a two-dimensional array by nesting one-dimensional initializers:

```
int m[5][9] = {{1, 1, 1, 1, 1, 0, 1, 1, 1},
               {0, 1, 0, 1, 0, 1, 0, 1, 0},
               {0, 1, 0, 1, 1, 0, 0, 1, 0},
               {1, 1, 0, 1, 0, 0, 0, 1, 0},
               {1, 1, 0, 1, 0, 0, 1, 1, 1}};
```

- The following initializer fills only the first three rows of `m`; the last two rows will contain zeros:

```
int m[5][9] =
{{1, 1, 1, 1, 1, 0, 1, 1, 1},
 {0, 1, 0, 1, 0, 1, 0, 1, 0},
 {0, 1, 0, 1, 1, 0, 0, 1, 0}};
```

Constant Arrays

- An array can be made "constant" by starting its declaration with the word `const`:

```
const char hex_chars[] =
  {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
   'A', 'B', 'C', 'D', 'E', 'F'};
```

- An array that's been declared `const` should not be modified by the program.

Function calls

Introduction

- A function is a series of statements that have been grouped together and given a name.
- Each function is essentially a small program, with its own declarations and statements.
- Advantages of functions:
  - A program can be divided into small pieces that are easier to understand and modify.
  - We can avoid duplicating code that's used more than once.
  - A function that was originally part of one program can be reused in other programs.

Program: Computing Averages

- A function named `average` that computes the average of two `double` values:

```
double average(double a, double b)
{
  return (a + b) / 2;
}
```

- The word `double` at the beginning is the ***return type*** of `average`.
- The identifiers `a` and `b` (the function's ***parameters***) represent the numbers that will be supplied when `average` is called.

Program: Computing Averages

- Every function has an executable part, called the ***body,*** which is enclosed in braces.
- The body of `average` consists of a single `return` statement.
- Executing this statement causes the function to “return” to the place from which it was called; the value of `(a + b) / 2` will be the value returned by the function.

Program: Computing Averages

- A function call consists of a function name followed by a list of ***arguments.***
  - `average(x, y)` is a call of the `average` function.
- Arguments are used to supply information to a function.
  - The call `average(x, y)` causes the values of `x` and `y` to be copied into the parameters `a` and `b`.
- An argument doesn't have to be a variable; any expression of a compatible type will do.
  - `average(5.1, 8.9)` and `average(x/2, y/3)` are legal.

Program: Computing Averages

- The `average.c` program reads three numbers and uses the `average` function to compute their averages, one pair at a time:

```
Enter three numbers: 3.5 9.6 10.2
Average of 3.5 and 9.6: 6.55
Average of 9.6 and 10.2: 9.9
Average of 3.5 and 10.2: 6.85
```

average.c

```c
/* Computes pairwise averages of three numbers */

#include <stdio.h>

double average(double a, double b)
{
  return (a + b) / 2;
}

int main(void)
{
  double x, y, z;

  printf("Enter three numbers: ");
  scanf("%lf%lf%lf", &x, &y, &z);
  printf("Average of %g and %g: %g\n", x, y, average(x, y));
  printf("Average of %g and %g: %g\n", y, z, average(y, z));
  printf("Average of %g and %g: %g\n", x, z, average(x, z));
 }
```

Function Definitions

- General form of a ***function definition:***

return-type function-name (*parameters*)

{

declarations

statements

}

Function Definitions

- The body of a function may include both declarations and statements.
- An alternative version of the `average` function:

```
double average(double a, double b)
{
  double sum;          /* declaration */

  sum = a + b;         /* statement */
  return sum / 2;      /* statement */
}
```

Function Calls

- A function call consists of a function name followed by a list of arguments, enclosed in parentheses:

```
average(x, y)
print_count(i)
print_pun()
```

- If the parentheses are missing, the function won't be called:

```
print_pun;    /*** WRONG ***/
```

This statement is legal but has no effect.

Function Calls

- A call of a `void` function is always followed by a semicolon to turn it into a statement:

```
print_count(i);
print_pun();
```

- A call of a non-`void` function produces a value that can be stored in a variable, tested, printed, or used in some other way:

```
avg = average(x, y);
if (average(x, y) > 0)
  printf("Average is positive\n");
printf("The average is %g\n", average(x, y));
```

Program: Testing Whether a Number Is Prime

- The `prime.c` program tests whether a number is prime:

```
Enter a number: 34
Not prime
```

- The program uses a function named `is_prime` that returns `true` if its parameter is a prime number and `false` if it isn't.
- `is_prime` divides its parameter `n` by each of the numbers between 2 and the square root of `n`; if the remainder is ever 0, `n` isn't prime.

prime.c

```c
/* Tests whether a number is prime */

#include <stdbool.h>   /* C99 only */
#include <stdio.h>

bool is_prime(int n)
{
  int divisor;

  if (n <= 1)
    return false;
  for (divisor = 2; divisor * divisor <= n; divisor++)
    if (n % divisor == 0)
      return false;
  return true;
}
```

```
int main(void)
{
  int n;

  printf("Enter a number: ");
  scanf("%d", &n);
  if (is_prime(n))
    printf("Prime\n");
  else
    printf("Not prime\n");
  return 0;
}
```

Array Arguments

- Example: `/* no length specified */`

```
int sum_array(int a[], int n)
{
  int i, sum = 0;

  for (i = 0; i < n; i++)
    sum += a[i];

  return sum;
}
```

- Since `sum_array` needs to know the length of `a`, we must supply it as a second argument.

Array Arguments

- When `sum_array` is called, the first argument will be the name of an array, and the second will be its length:

```
#define LEN 100

int main(void)
{
  int b[LEN], total;
  …
  total = sum_array(b, LEN);
  …
}
```

- Notice that we don't put brackets after an array name when passing it to a function:

```
total = sum_array(b[], LEN);   /*** WRONG ***/
```

Array Arguments

- A function is allowed to change the elements of an array parameter, and the change is reflected in the corresponding argument.
- A function that modifies an array by storing zero into each of its elements:

```
void store_zeros(int a[], int n)
{
  int i;

  for (i = 0; i < n; i++)
    a[i] = 0;
}
```

- A call of `store_zeros`:

```
store_zeros(b, 100);
```

Array Arguments

- If a parameter is a multidimensional array, only the length of the first dimension may be omitted.
- If we revise `sum_array` so that `a` is a two-dimensional array, we must specify the number of columns in `a`:

```
#define LEN 10

int sum_two_dimensional_array(int a[][LEN], int n)
{
  int i, j, sum = 0;

  for (i = 0; i < n; i++)
    for (j = 0; j < LEN; j++)
      sum += a[i][j];

  return sum;
}
```

The `return` Statement

- `return` statements may appear in functions whose return type is `void`, provided that no expression is given:

```
return;  /* return in a void function */
```

- Example:

```
void print_int(int i)
{
  if (i < 0)
    return;
  printf("%d", i);
}
```

Program Termination

- Normally, the return type of `main` is `int`:

```
int main(void)
{
   …
}
```

- Older C programs often omit `main`'s return type, taking advantage of the fact that it traditionally defaults to `int`:

```
main()
{
   …
}
```

The `exit` Function

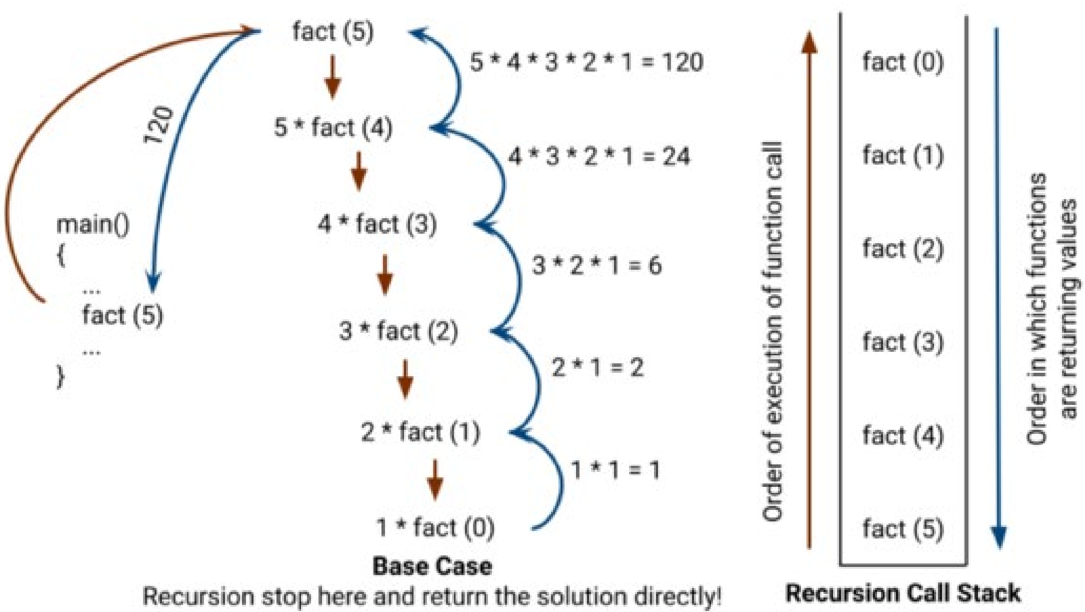
- Executing a `return` statement in `main` is one way to terminate a program.
- Another is calling the `exit` function, which belongs to `<stdlib.h>`.
- The argument passed to `exit` has the same meaning as `main`'s return value: both indicate the program's status at termination.
- To indicate normal termination, we'd pass 0:

```
exit(0);   /* normal termination */
```

Recursion

- A function is ***recursive*** if it calls itself.
- The following function computes n! recursively, using the formula $n! = n \times (n - 1)!$:

```
int fact(int n){
   if (n <= 1)
      return 1;
   else
      return n * fact(n - 1);
}
```



Recursion

- To see how recursion works, let's trace the execution of the statement

```
i = fact(3);
```

`fact(3)` finds that 3 is not less than or equal to 1, so it calls

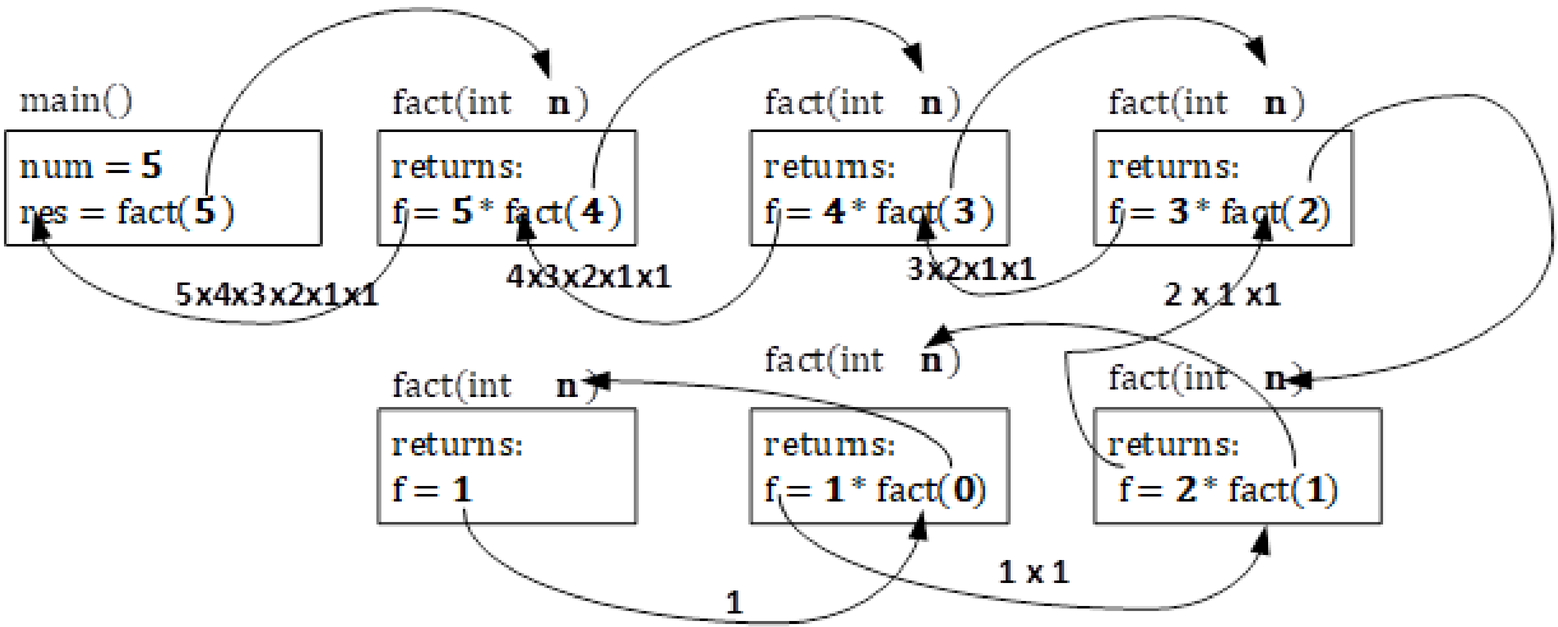
  `fact(2)`, which finds that 2 is not less than or equal to 1, so it calls

    `fact(1)`, which finds that 1 is less than or equal to 1, so it returns 1, causing

  `fact(2)` to return $2 \times 1 = 2$, causing

`fact(3)` to return $3 \times 2 = 6$.

Illustration of Recursive Function



Recursion

- The following recursive function computes x^n, using the formula $x^n = x \times x^{n-1}$.

```
int power(int x, int n)
{
  if (n == 0)
    return 1;
  else
    return x * power(x, n - 1);
}
```

Recursion

- We can condense the `power` function by putting a conditional expression in the `return` statement:

```
int power(int x, int n)
{
  return n == 0 ? 1 : x * power(x, n - 1);
}
```

- Both `fact` and `power` are careful to test a “termination condition” as soon as they’re called.
- All recursive functions need some kind of termination condition in order to prevent infinite recursion.

Recursion vs Iteration

- Any problem that can be solved recursively

  Can also be solved iteratively (nonrecursively)?
- Avoiding Stack Overflow errors using tail-recursive functions
- Avoid using recursion in performance situations
  - Recursive calls take time
  - And consume additional memory

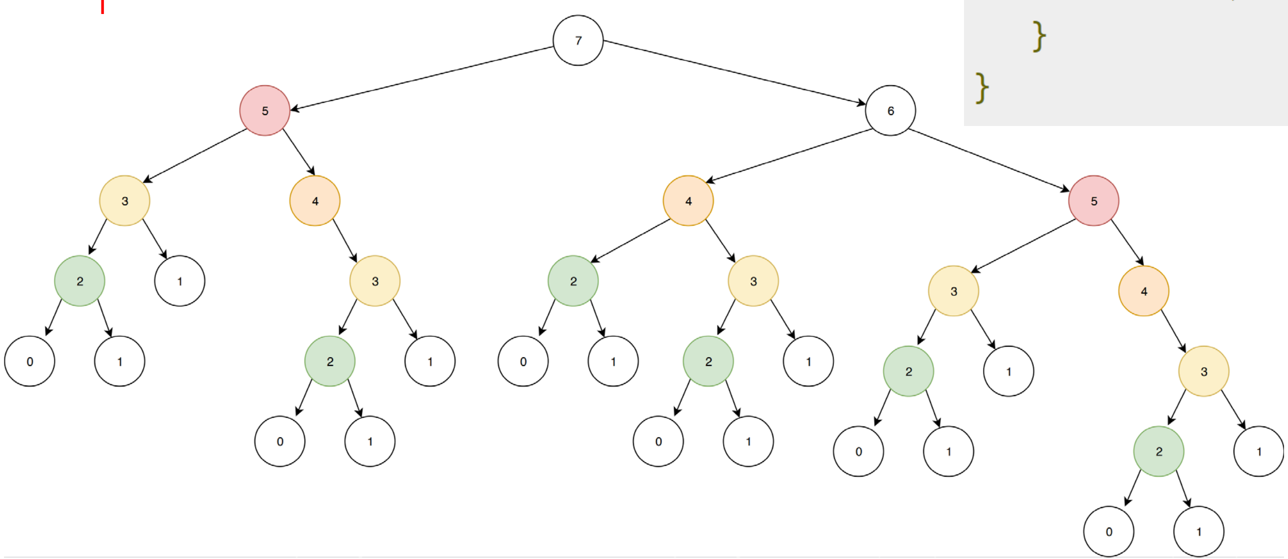
```
int fact(int n) {
   int i, prod=1;
   for (i=1; i<=n; i++)
      prod = prod * i;

   return prod;
}
```

```
int fact(int n) {
   if (n <= 1)
      return 1;
   Else
      return n *
            fact(n - 1);
}
```

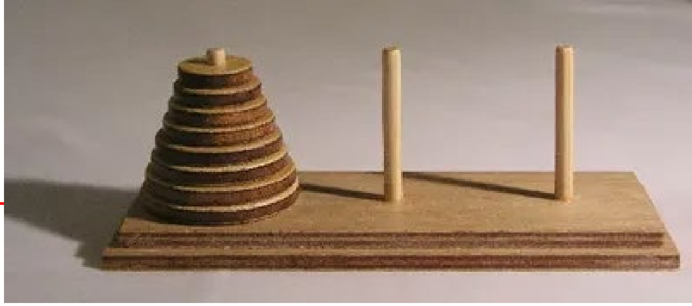
Fibonacci numbers

- The Fibonacci numbers are the integer sequence.
  0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...
- $F_n = F_{n-1} + F_{n-2}$ $F_0 = 0$ and $F_1 = 1$

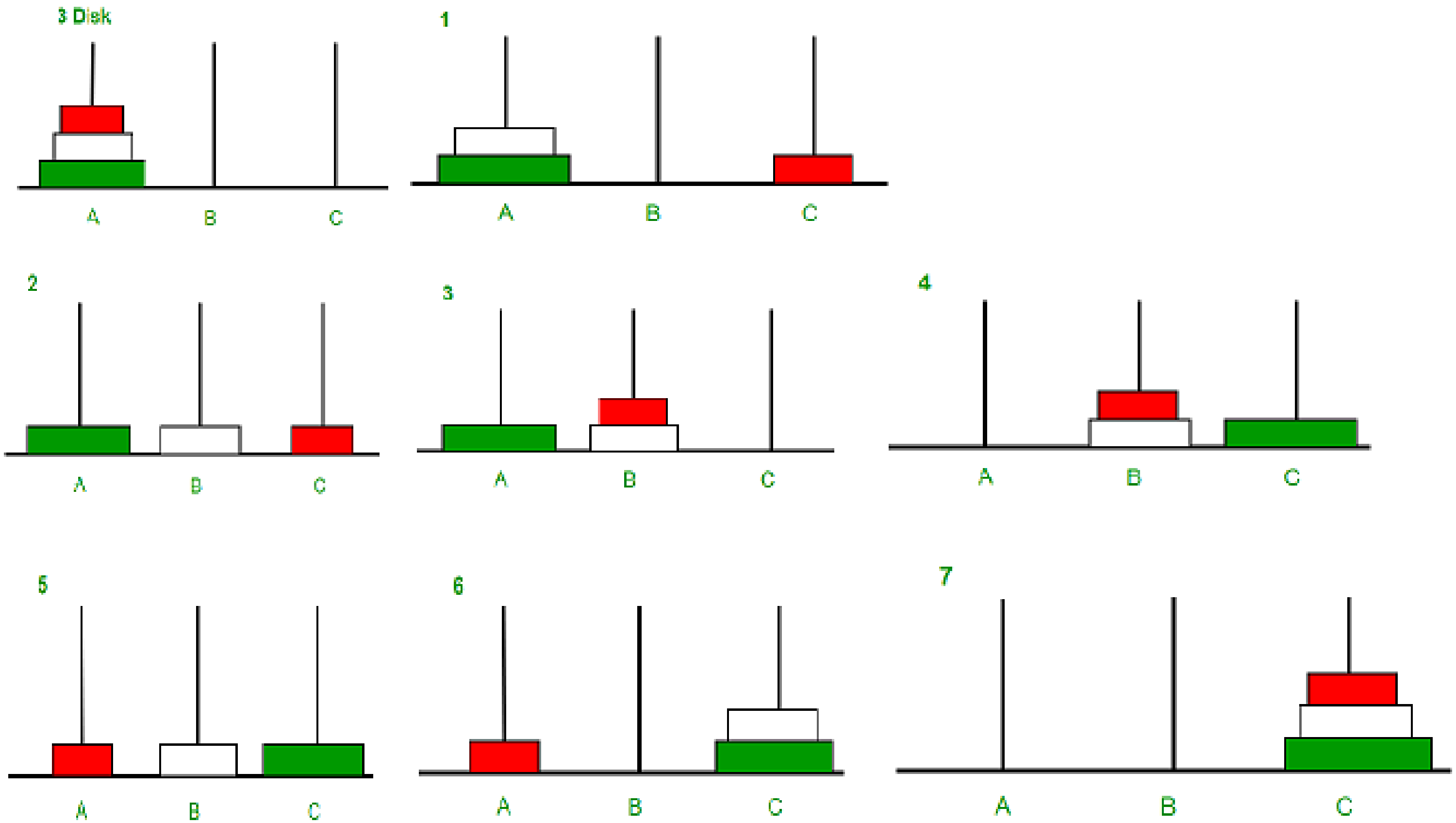
```
int fibbonacci(int n) {
   if(n == 0){
      return 0;
   } else if(n == 1) {
      return 1;
   } else {
      return (fibbonacci(n-1) + fibbonacci(n-2));
   }
}
```



Tower of Hanoi



- a mathematical puzzle where we have three rods and n disks.



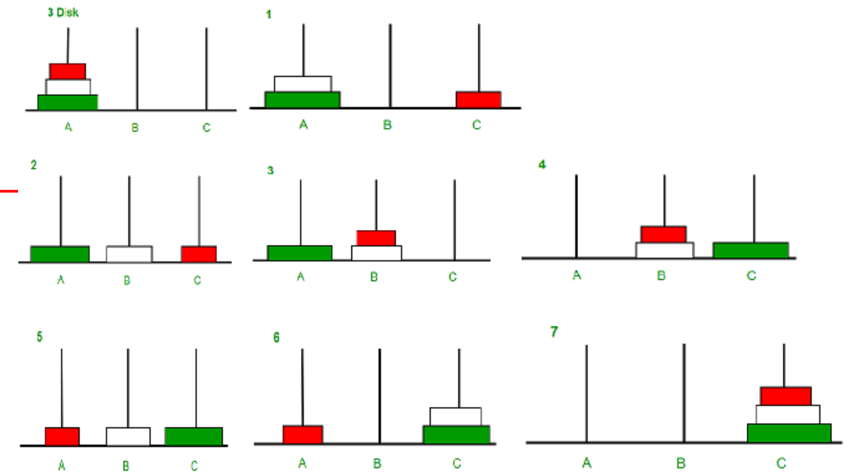
C recursive for hanoi tower

```
#include <stdio.h>
void Hanoi(int n, char from_rod, char to_rod, char aux_rod)
{
    if (n == 1)      {
        printf("\n Move disk 1 from rod %c to rod %c",
                from_rod, to_rod);
        return;
    }
    Hanoi(n-1, from_rod, aux_rod, to_rod);
    printf("\n Move disk %d from rod %c to rod %c",
        n, from_rod, to_rod);
    Hanoi(n-1, aux_rod, to_rod, from_rod);
}

int main()
{
    int n = 3; // Number of disks
    Hanoi(n, 'A', 'C', 'B’);
        // A, B and C are names of rods
}
```

Time Complexity: $O(2^n)$

Auxiliary Space: $O(n)$

Output:




```
Disk 1 moved from A to C

Disk 2 moved from A to B

Disk 1 moved from C to B

Disk 3 moved from A to C

Disk 1 moved from B to A

Disk 2 moved from B to C

Disk 1 moved from A to C
```

Memory segment in Linux system

1. Text segment (i.e. instructions)
2. Initialized data segment
3. Uninitialized data segment (bss)
4. Heap
5. Stack