Functions

Function

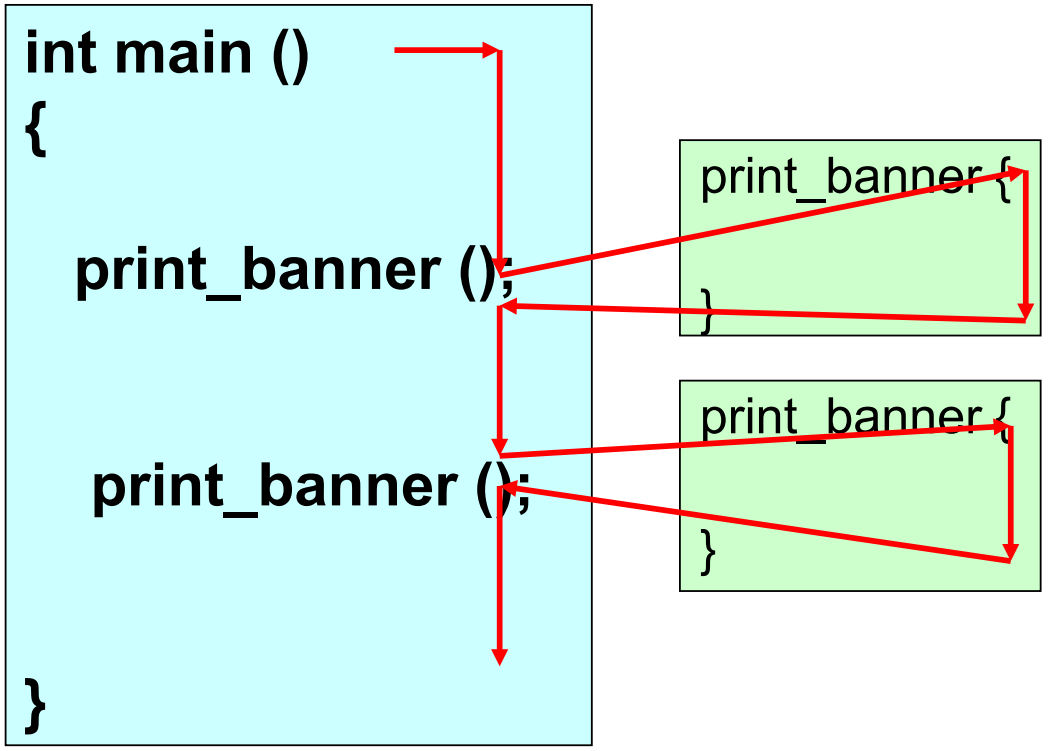
- A program segment that carries out some specific, well-defined task
- Example
  - A function to add two numbers
  - A function to find the largest of n numbers
- A function will carry out its intended task whenever it is called or invoked
  - Can be called multiple times

- Every C program consists of one or more functions
- One of these functions must be called main
- Execution of the program always begins by carrying out the instructions in main
- Functions call other functions as instructions

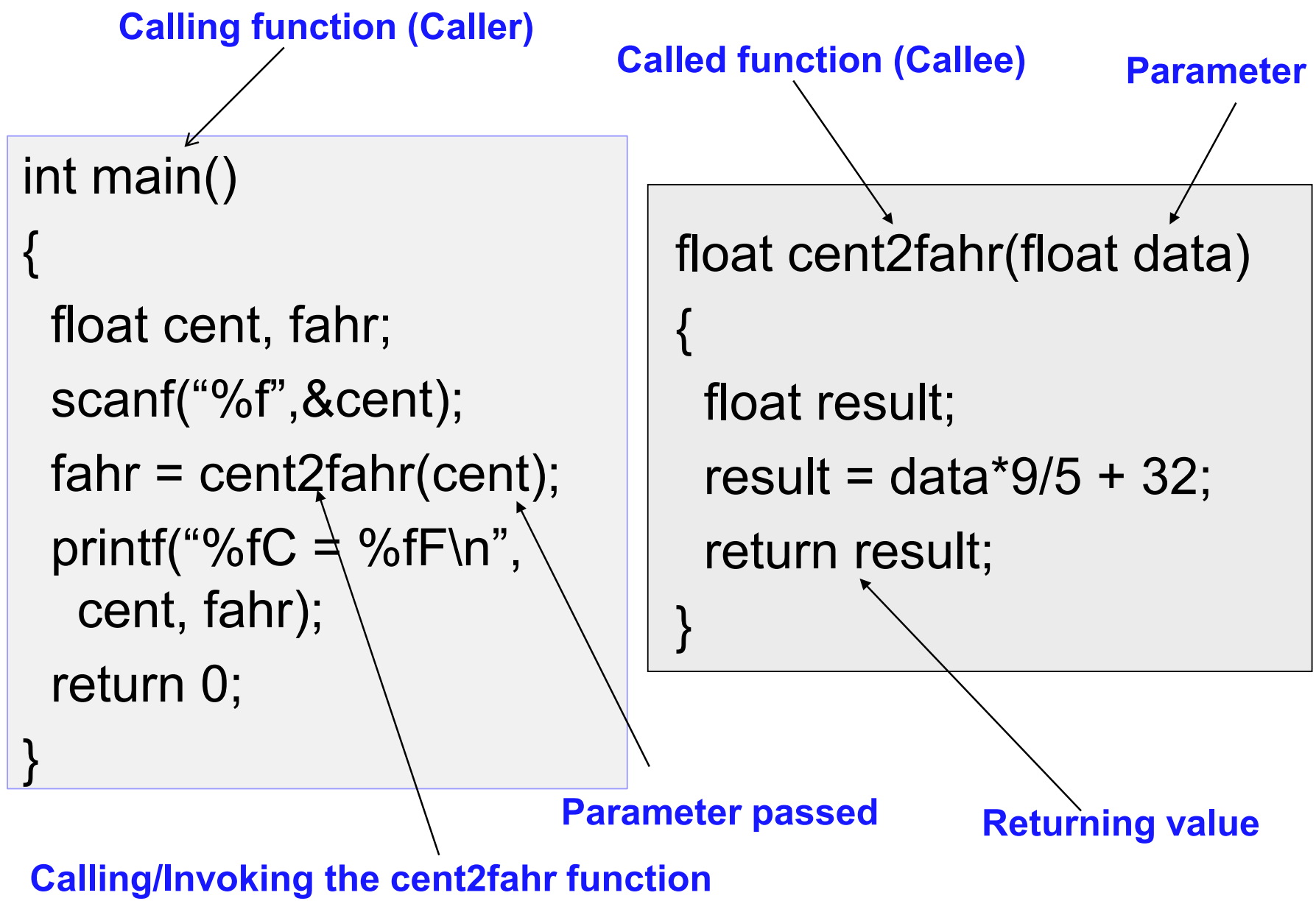
Function Control Flow

```
void print_banner ()
{
    printf("**********\n");
}
```

```
int main ()
{
    . . .
    print_banner ();
    . . .
    print_banner ();

}
```



- Calling function (caller) may pass information to the called function (callee) as parameters/arguments
  - For example, the numbers to add
- The callee may return a single value to the caller
  - Some functions may not return anything


Calling function (Caller)
Called function (Callee)
Parameter
int main()
{
float cent, fahr;
scanf("%f",&cent);
fahr = cent2fahr(cent);
printf("%fC = %fF\n",
cent, fahr);
return 0;
}
float cent2fahr(float data)
{
float result;
result = data*9/5 + 32;
return result;
}
Parameter passed
Returning value
Calling/Invoking the cent2fahr function

How it runs

```
float cent2fahr(float data)
{
  float result;
  printf("data = %f\n", data);
  result = data*9/5 + 32;
  return result;
   printf("result = %f\n", result);
}
int main()
{ float cent, fahr;
  scanf("%f",&cent);
  printf("Input is %f\n", cent);
  fahr = cent2fahr(cent);
  printf("%fC = %fF\n", cent, fahr);
  return 0;
}
```

Outputs

```
32
Input is 32.000000
data = 32.000000
32.000000C = 89.599998F
```

```
-45.6
Input is -45.599998
data = -45.599998
-45.599998C = -50.079998F
```

Another Example

```
int  factorial (int m)
{
    int i, temp=1;
    for (i=1; i<=m; i++)
          temp = temp * i;
    return (temp);
}
```

```
int main()
{
    int  n;
    for  (n=1; n<=5; n++)
          printf ("%d! = %d \n",
                n, factorial (n) );
    return 0;
}
```

Output

```
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
```

Why Functions?

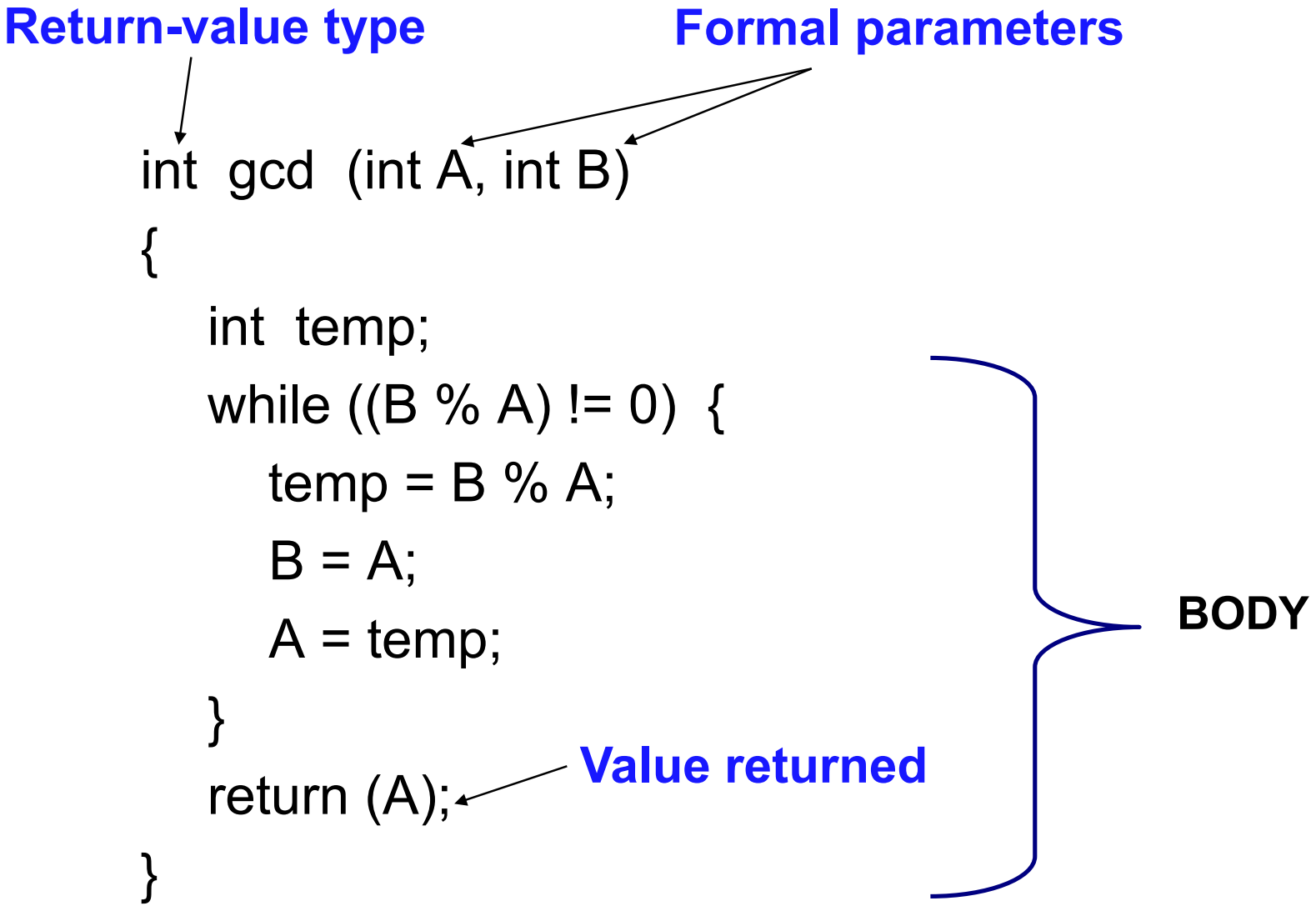
- Allows one to develop a program in a modular fashion
  - Divide-and-conquer approach
  - Construct a program from small pieces or components
- Use existing functions as building blocks for new programs
- Abstraction: hide internal details (library functions)

Defining a Function

- A function definition has two parts:
  - The first line, called header
  - The body of the function

```
return-value-type  function-name  ( parameter-list )
{
      declarations and statements
}
```

- The first line contains the return-value-type, the function name, and optionally a set of comma-separated arguments enclosed in parentheses
  - Each argument has an associated type declaration
  - The arguments are called formal arguments or formal parameters
- The body of the function is actually a block of statement that defines the action to be taken by the function


Return-value type
Formal parameters
int gcd (int A, int B)
{
int temp;
while ((B % A) != 0) {
temp = B % A;
B = A;
A = temp;
}
return (A);
}
Value returned
BODY

Return value

- A function can return a value
  - Using **return** statement
- Like all values in C, a function return value has a type
- The return value can be assigned to a variable in the caller

```
int x, y, z;
scanf("%d%d", &x, &y);
z = gcd(x,y);
printf("GCD of %d and %d is %d\n", x, y, z);
```

Function Not Returning Any Value

- Example: A function which prints if a number is divisible by 7 or not

```
void  div7 (int n)
{
   if  ((n % 7) == 0)
        printf ("%d is divisible by 7", n);
   else
        printf ("%d is not divisible by 7", n);
   return;
}
```

Return type is void

Optional

return statement

- In a value-returning function (result type is not void), return does two distinct things
  - specify the value returned by the execution of the function
  - terminate that execution of the callee and transfer control back to the caller
- A function can only return one value
  - The value can be any expression matching the return type
  - but it might contain more than one return statement.
- In a void function
  - return is optional at the end of the function body.
  - return may also be used to terminate execution of the function explicitly.
  - No return value should appear following return.

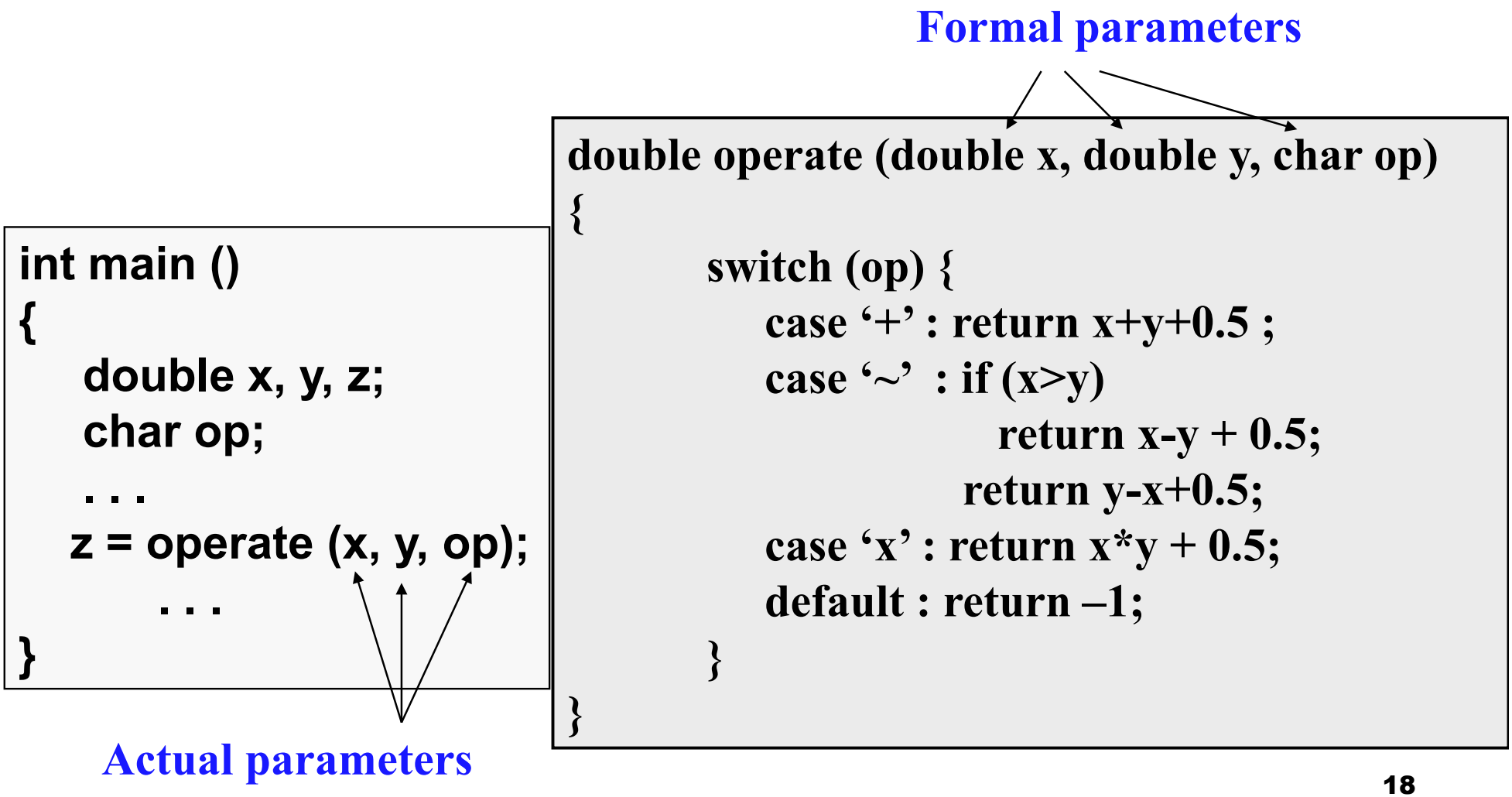
```
void compute_and_print_itax ()
{
    float income;
    scanf ("%f", &income);
    if (income < 50000)  {
          printf ("Income tax = Nil\n");
          return;
    }
    if (income < 60000)  {
          printf ("Income tax = %f\n", 0.1*(income-50000));
          return;
    }
    if (income < 150000)          {
           printf ("Income tax = %f\n", 0.2*(income-60000)+1000);
          return ;
    }
    printf ("Income tax = %f\n", 0.3*(income-150000)+19000);
}
```

Terminate function execution before reaching the end

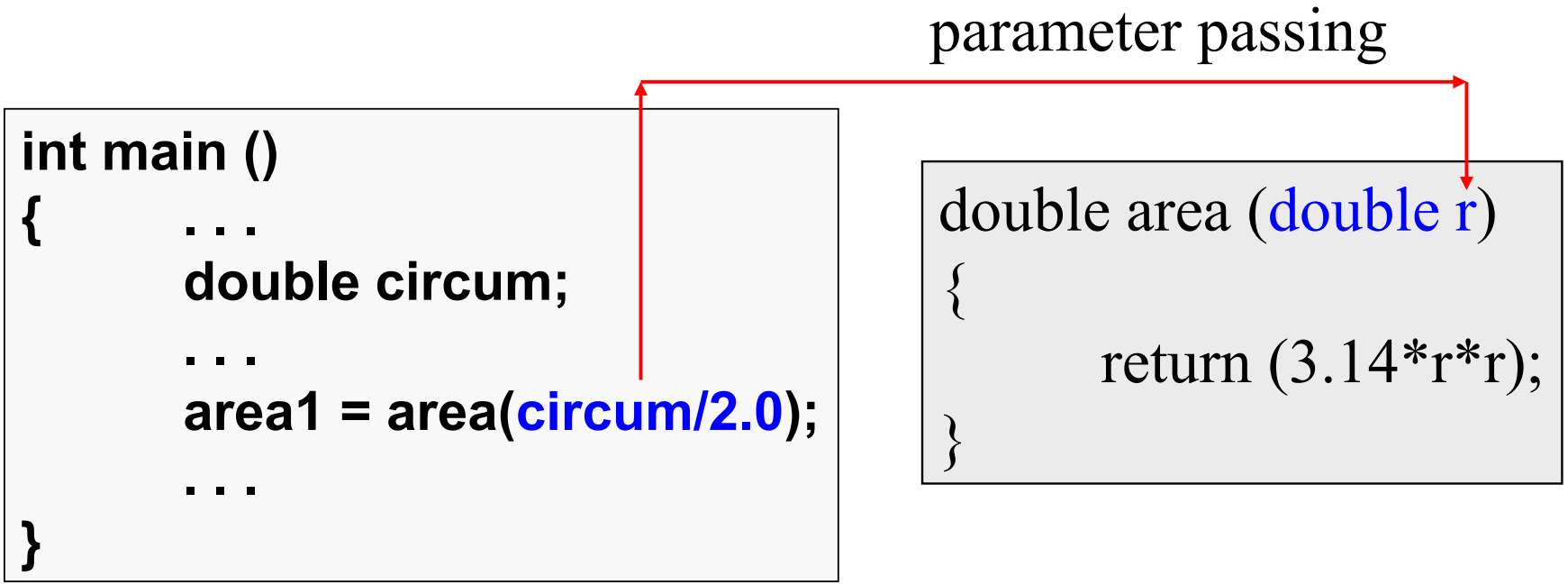
Calling a function

- Called by specifying the function name and parameters in an instruction in the calling function
- When a function is called from some other function, the corresponding arguments in the function call are called actual arguments or actual parameters
  - The function call must include a matching actual parameter for each formal parameter
  - Position of an actual parameters in the parameter list in the call must match the position of the corresponding formal parameter in the function definition
  - The formal and actual arguments must match in their data types

Example


Formal parameters
double operate (double x, double y, char op)
{
switch (op) {
case '+' : return x+y+0.5 ;
case '~' : if (x>y)
return x-y + 0.5;
return y-x+0.5;
case 'x' : return x*y + 0.5;
default : return –1;
}
}
int main ()
{
double x, y, z;
char op;
. . .
z = operate (x, y, op);
. . .
}
Actual parameters

- When the function is executed, the value of the actual parameter is copied to the formal parameter



Another Example

```
/* Compute the GCD of four numbers */
int main()
{
   int  n1, n2, n3, n4, result;
   scanf ("%d %d %d %d", &n1, &n2, &n3, &n4);
   result  =  gcd ( gcd (n1, n2), gcd (n3, n4) );
   printf ("The GCD of %d, %d, %d and %d is %d \n",
   n1, n2, n3, n4, result);
   return 0;
}
```

Another Example

```
int main()
{
  int numb, flag, j=3;
  scanf("%d",&numb);
   while (j <=numb)
   {
       flag = prime(j);
       if (flag==0)
          printf("%d is prime\n",j);
       j++;
   }
   return 0;
 }
```

```
int prime(int x)
{
   int i, test;
   i=2, test =0;
   while ((i <= sqrt(x)) && (test
    ==0))
   {
       if (x%i==0) test = 1;
       i++;
     }
    return test;
}
```

Tracking the flow of control

```
int main()
{
  int numb, flag, j=3;
  scanf("%d",&numb);
  printf("numb = %d \n",numb);
  while (j <= numb)
  {  printf("Main, j = %d\n",j);
     flag = prime(j);
     printf("Main, flag = %d\n",flag);
     if (flag == 0)
        printf("%d is prime\n",j);
     j++;
  }
  return 0;
 }
```

```
int prime(int x)
{
   int i, test;
   i = 2;  test = 0;
   printf("In function, x = %d \n",x);
   while ((i <= sqrt(x)) && (test == 0))
   {
      if (x%i == 0) test = 1;
      i++;
   }
   printf("Returning, test = %d \n",test);
   return test;
}
```

The output

```
5
numb = 5
Main, j = 3
In function, x = 3
Returning, test = 0
Main, flag = 0
3 is prime
Main, j = 4
In function, x = 4
Returning, test = 1
Main, flag = 1
Main, j = 5
In function, x = 5
Returning, test = 0
Main, flag = 0
5 is prime
```

Points to note

- The identifiers used as formal parameters are “local”.
  - Not recognized outside the function
  - Names of formal and actual arguments may differ
- A value-returning function is called by including it in an expression
  - A function with return type T (≠ void) can be used anywhere an expression of type T can be used

- Returning control back to the caller
  - If nothing returned
    - return;
    - or, until reaches the last right brace ending the function body
  - If something returned
    - return expression;

Function Prototypes

- Usually, a function is defined before it is called
  - main() is the last function in the program written
  - Easy for the compiler to identify function definitions in a single scan through the file

- However, many programmers prefer a top-down approach, where the functions are written after main()
  - Must be some way to tell the compiler
  - Function prototypes are used for this purpose
    - Only needed if function definition comes after use

- Function prototypes are usually written at the beginning of a program, ahead of any functions (including main())
- Prototypes can specify parameter names or just types (more common)
- Examples:

```
int  gcd (int , int );
void div7 (int number);
```

  - Note the semicolon at the end of the line.
  - The parameter name, if specifed, can be anything; but it is a good practice to use the same names as in the function definition

Example:

```
#include <stdio.h>
int sum(int, int);
int main()
{
        int x, y;
        scanf("%d%d", &x, &y);
        printf("Sum = %d\n", sum(x, y));
}
int sum (int a, int b)
{
        return(a + b);
}
```

Some more points

- A function cannot be defined within another function
  - All function definitions must be disjoint
- Nested function calls are allowed
  - A calls B, B calls C, C calls D, etc.
  - The function called last will be the first to return
- A function can also call itself, either directly or in a cycle
  - A calls B, B calls C, C calls back A.
  - Called recursive call or recursion

Example: main calls ncr, ncr calls fact

```
int ncr (int n, int r);
int fact (int n);

int main()
{
    int i, m, n, sum=0;
    scanf ("%d %d", &m, &n);
    for (i=1; i<=m; i+=2)
        sum = sum + ncr (n, i);
    printf ("Result: %d \n",
    sum);
     return 0;
}
```

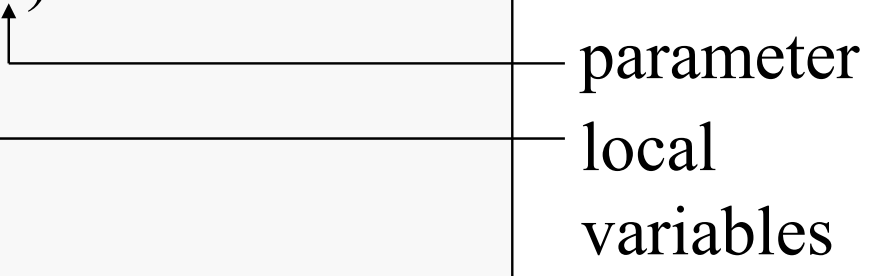
```
int  ncr (int n, int r)
{
    return (fact(n) / fact(r) /
    fact(n-r));
}

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

Local variables

- A function can define its own local variables
- The locals have meaning only within the function
  - Each execution of the function uses a new set of locals
  - Local variables cease to exist when the function returns
- Parameters are also local

Local variables

```
/* Find the area of a circle with diameter d */
double circle_area (double d)
{
        double radius, area;
        radius = d/2.0;
        area = 3.14*radius*radius;
        return (area);
}
```



Revisiting nCr

```
int fact(int x)
{ int i,fact=1;
  for(i=2; i<=x; ++i) fact=fact*i;
  return fact;
}
```

```
int main()
{
    int n, r;
    scanf("%d%d",&n,&r);
    printf("n=%d, r=%d,
    nCr=%d\n",n, r, ncr(n,r));
    return 0;
}
```

```
int ncr(int x,int y)
{
   int p,q,r;
   p=fact(x);
   q=fact (y);
   r = fact(x-y);
  return p/(q*r);
}
```

The variable x in function fact and x in function ncr are different.

The values computed from the arguments at the point of call are copied on to the corresponding parameters of the called function before it starts execution.

Scope of a variable

- Part of the program from which the value of the variable can be used (seen)
- Scope of a variable - Within the block in which the variable is defined
  - Block = group of statements enclosed within { }
- Local variable – scope is usually the function in which it is defined
  - So two local variables of two functions can have the same name, but they are different variables
- Global variables – declared outside all functions (even main)
  - scope is entire program by default, but can be hidden in a block if local variable of same name defined

Variable Scope

```
#include  <stdio.h>
int A = 1;
int main()
{
   myProc();
   printf ( "A = %d\n", A);
}

void myProc()
{    int A = 2;
     if ( A==2 )
     {
       int A = 3;
       printf ( "A = %d\n", A);
     }
     printf ( "A = %d\n", A);
}
```

Global variable

Hides the global A

Output:

A = 3

A = 2

A = 1

Parameter Passing: by Value and by Reference

- Used when invoking functions
- Call by value
  - Passes the value of the argument to the function
  - Execution of the function does not change the actual parameters
    - All changes to a parameter done inside the function are done on a copy of the actual parameter
    - The copy is removed when the function returns to the caller
    - The value of the actual parameter in the caller is not affected
  - Avoids accidental changes

- Call by reference
  - Passes the address to the original argument.
  - Execution of the function may affect the original
  - Not directly supported in C except for arrays

Parameter passing & return: 1

```
int main()
{
   int  a=10, b;
   printf  ("Initially a = %d\n", a);
   b = change (a);
   printf  ("a = %d, b = %d\n", a, b);
   return 0;
}
int  change  (int x)
{
  printf  ("Before x = %d\n",x);
   x = x / 2;
  printf  ("After x = %d\n", x);
   return (x);
}
```

Output

```
Initially a = 10
Before x = 10
After x = 5
a = 10, b = 5
```

Parameter passing & return: 2

```
int main()
{
    int  x=10, b;
    printf  ("M: Initially x = %d\n", x);
    b = change (x);
    printf  ("M: x = %d, b = %d\n", x, b);
    return 0;
}
int  change  (int x)
{
   printf  ("F: Before x = %d\n",x);
    x = x / 2;
   printf  ("F: After x = %d\n", x);
    return (x);
}
```

Output

```
M: Initially x = 10
F: Before x = 10
F: After x = 5
M: x = 10, b = 5
```

Parameter passing & return: 3

```
int main()
{
   int  x=10, b;
   printf  (“M: Initially x = %d\n”, x);
   x = change (x);
   printf  (“M: x = %d, b = %d\n”, x, x);
   return 0;
}
int  change  (int x)
{
  printf  (“F: Before x = %d\n”,x);
   x = x / 2;
  printf  (“F: After x = %d\n”, x);
   return (x);
}
```

Output

```
M: Initially x = 10
F: Before x = 10
F: After x = 5
M: x = 5, b = 5
```

Parameter passing & return: 4

```
int main()
{
   int  x=10, y=5;
   printf  ("M1:  x = %d, y = %d\n", x, y);
   interchange (x, y);
   printf  ("M2:  x = %d, y = %d\n", x, y);
   return 0;
}

void interchange  (int x, int y)
{ int temp;
  printf  ("F1:  x = %d, y = %d\n", x, y);
  temp= x; x = y; y = temp;
  printf  ("F2:  x = %d, y = %d\n", x, y);
}
```

Output

```
M1:  x = 10, y = 5
F1:  x = 10, y = 5
F2:  x = 5, y = 10
M2:  x = 10, y = 5
```

How do we write an interchange function? (will see later)

Passing Arrays to Function

- Array element can be passed to functions as ordinary arguments
  - IsFactor (x[i], x[0])
  - sin (x[5])

Passing Entire Array to a Function

- An array name can be used as an argument to a function
  - Permits the entire array to be passed to the function
  - The way it is passed differs from that for ordinary variables
- Rules:
  - The array name must appear by itself as argument, without brackets or subscripts
  - The corresponding formal argument is written in the same manner
    - Declared by writing the array name with a pair of empty brackets

Whole Array as Parameters

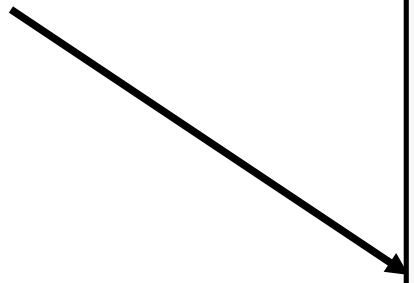
```
const int ASIZE = 5;
float average (int B[ ])
{
    int i, total=0;
    for (i=0; i<ASIZE; i++)
          total = total + B[i];
    return ((float) total / (float) ASIZE);
}

int main ( )  {
    int x[ASIZE] ; float x_avg;
     x = {10, 20, 30, 40, 50};
    x_avg = average (x) ;
     return 0;
}
```

Only Array Name/address passed. [ ] mentioned to indicate that is an array.

Called only with actual array name

Contd.

We don't need to write the array size. It works with arrays of any size.



```
int main()
{
   int  n;
   float   list[100], avg;
   :
   avg  =  average (n, list);
   :
}

float  average  (int a, float x[])
{
   :
   sum = sum + x[i];
}
```

Arrays used as Output Parameters

```
void VectorSum (int a[ ], int b[ ], int vsum[ ], int length) {
     int i;
     for (i=0; i<length; i=i+1)
              vsum[i] = a[i] + b[i] ;
}
void PrintVector (int a[ ], int length)  {
     int i;
     for (i=0; i<length; i++) printf ("%d ", a[i]);
}

int main ()  {
     int x[3] = {1,2,3}, y[3] = {4,5,6}, z[3];
     VectorSum (x, y, z, 3) ;
     PrintVector (z, 3) ;
     return 0;
}
```

The Actual Mechanism

- When an array is passed to a function, the values of the array elements are not passed to the function
  - The array name is interpreted as the address of the first array element
  - The formal argument therefore becomes a pointer to the first array element
  - When an array element is accessed inside the function, the address is calculated using the formula stated before
  - Changes made inside the function are thus also reflected in the calling program

Contd.

- Passing parameters in this way is called
  call-by-reference
- Normally parameters are passed in C using
  call-by-value
- Basically what it means?
  - If a function changes the values of array elements, then these changes will be made to the original array that is passed to the function
  - This does not apply when an individual element is passed on as argument

Library Functions

Library Functions

- Set of functions already written for you, and bundled in a “library”
- Example: printf, scanf, getchar,
- C library provides a large number of functions for many things
- Already seen math library functions earlier
- Will look at string library functions

String Library Functions

- String library functions
  - perform common operations on null terminated strings
  - Must include a special header file

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

- Example
  - printf ("%f", strlen(C));
    - C is a null-terminated string
    - Calls function strlen, which returns the number of characters in C (not counting the '\0' character)

Common string library functions

- strlen – returns the length of a string
- strcmp – compares two strings (lexicographic)
  - Returns 0 if the two strings are equal, < 0 if first string is less than the second string, > 0 if the first string is greater than the second string
  - Commonly used for sorting strings
- strcat – concatenates two strings
- strcpy – copy one string to another
  - we will need some basic knowledge of pointers to understand how to use strcat and strcpy
- Many others, but these are the ones you will know in this course

Example

```
int main()
{
    char A[20], B[20];
    int n, m, val;
    scanf(“%s%s”, A, B);
    n = strlen(A);
    m = strlen(B);
    printf(“The lengths of the strings are %d and %d\n”, n, m);
    val = strcmp(A, b);
    if (val == 0)
            printf(“The strings are the same\n”);
    else if (val < 0)
            printf(“%s is smaller than %s\n”, A, B);
    else
            printf(“%s is smaller than %s\n”, A, B);
}
```

Outputs

```
program program
The lengths of the strings are 7 and 7
The strings are the same
```

```
arobinda   abhijit
The lengths of the strings are 8 and 7
arobinda is larger than abhijit
```

```
iit-kgp   iit-mandi
The lengths of the strings are 7 and 9
iit-kgp is smaller than iit-mandi
```

```
arobinda   Arobinda
The lengths of the strings are 8 and 8
arobinda is larger than Arobinda
```


Recursion

Recursion

- A process by which a function calls itself repeatedly
  - Either directly.
    - X calls X
  - Or cyclically in a chain.
    - X calls Y, and Y calls X

- Used for repetitive computations in which each action is stated in terms of a previous result

  fact(n) = n * fact (n-1)

- For a problem to be written in recursive form, two conditions are to be satisfied:
  - It should be possible to express the problem in recursive form
    - Solution of the problem in terms of solution of the same problem on smaller sized data
  - The problem statement must include a stopping/terminating condition
    - The direct solution of the problem for a small enough size

fact(n) = 1, if n = 0 ← **Stopping/Terminating condition**

= n * fact(n-1), if n > 0 ← **Recursive definition**

- Examples:

  - Factorial:

    fact(0) = 1

    fact(n) = n * fact(n-1), if n > 0

  - GCD:

    gcd (m, m) = m

    gcd (m, n) = gcd (m%n, n), if m > n

    gcd (m, n) = gcd (n, n%m), if m < n

  - Fibonacci series (1,1,2,3,5,8,13,21,….)

    fib (0) = 1

    fib (1) = 1

    fib (n) = fib (n-1) + fib (n-2), if n > 1

Factorial

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

Factorial Execution

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

Factorial Execution

fact(4)

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

Factorial Execution

fact(4)

```
if   (4 = = 1) return (1);
else return  (4 * fact(3));
```

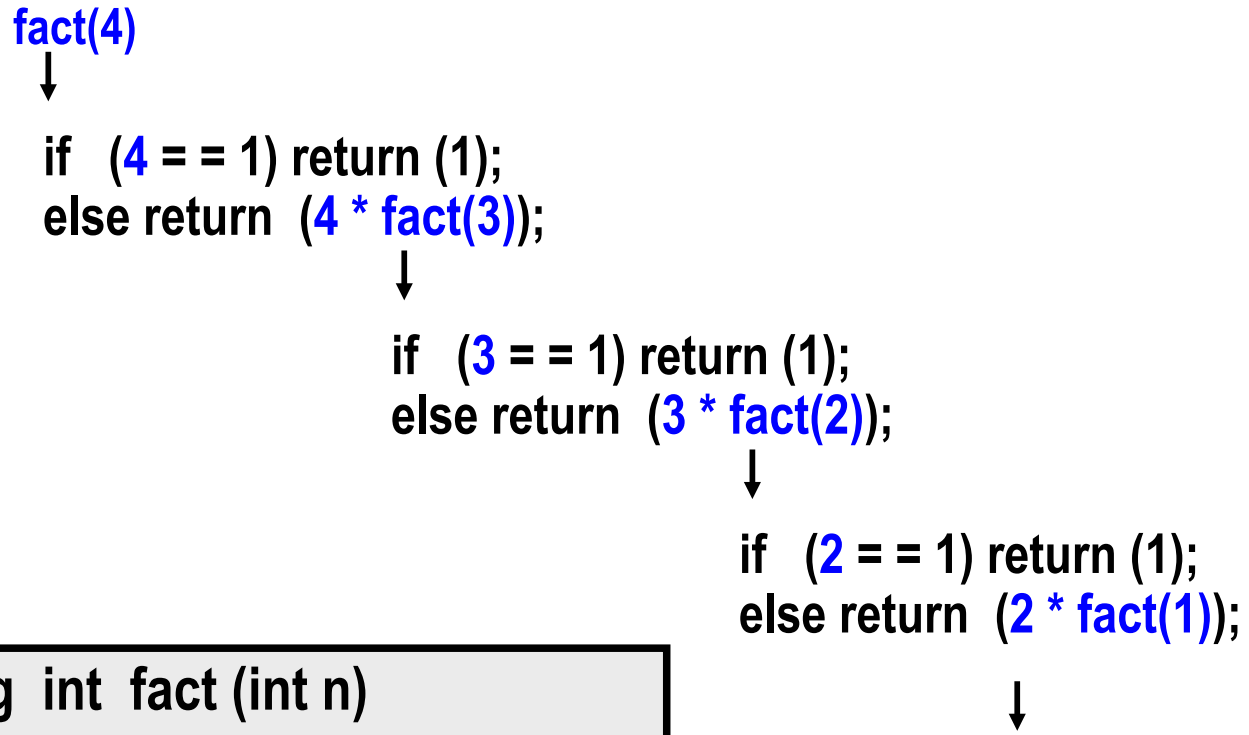
```
long  int  fact (int n)
{
   if   (n = = 1) return (1);
   else return  (n * fact(n-1));
}
```

Factorial Execution

```
fact(4)
  ↓
  if   (4 = = 1) return (1);
  else return  (4 * fact(3));
                    ↓
                    if   (3 = = 1) return (1);
                    else return  (3 * fact(2));
                                      ↓
```

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

Factorial Execution



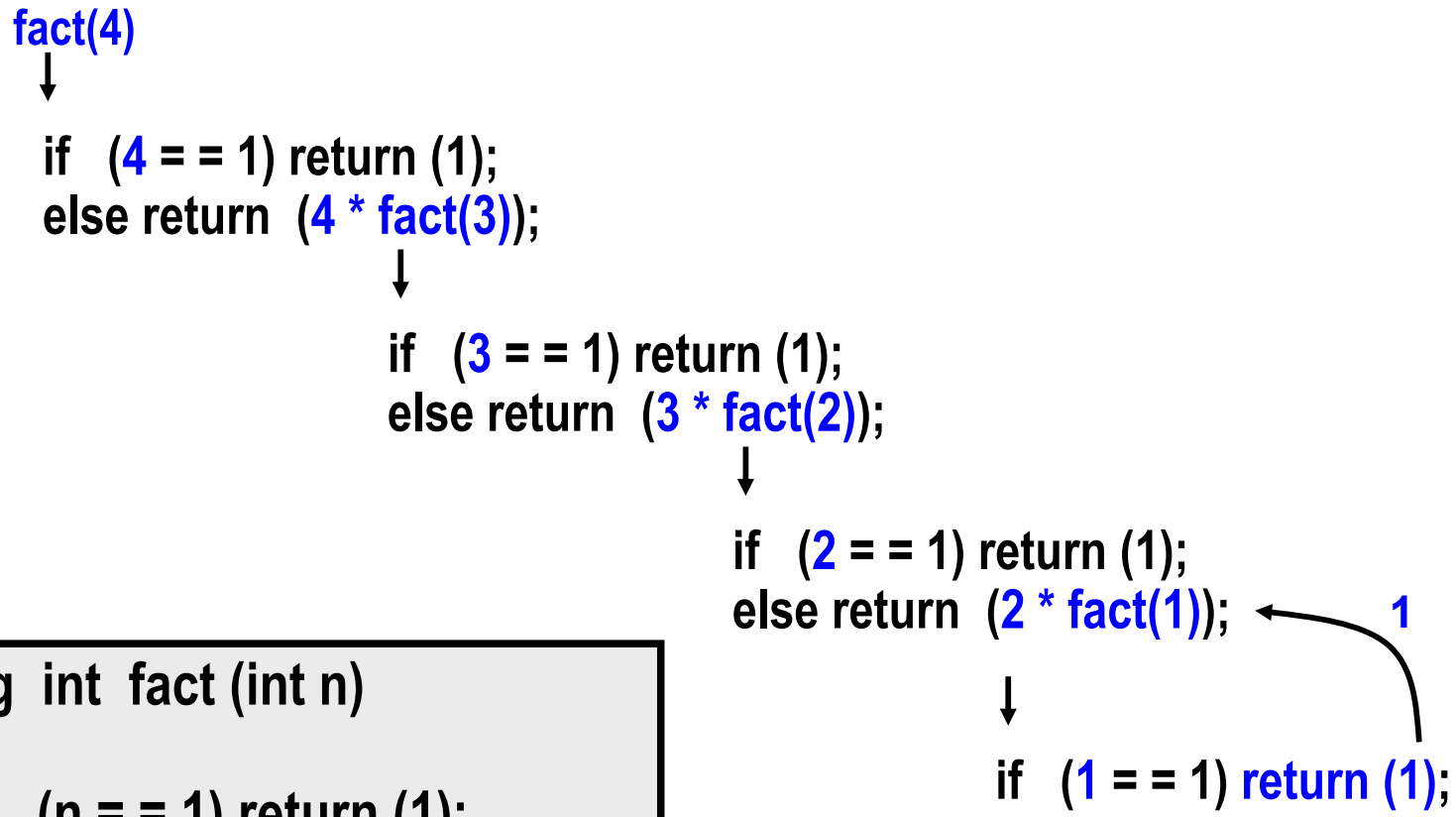

```
long  int  fact (int n)
{
   if  (n = = 1) return (1);
   else return  (n * fact(n-1));
}
```

Factorial Execution

```
fact(4)
  ↓
  if   (4 = = 1) return (1);
  else return  (4 * fact(3));
                    ↓
                    if   (3 = = 1) return (1);
                    else return  (3 * fact(2));
                                      ↓
                                      if   (2 = = 1) return (1);
                                      else return  (2 * fact(1));
                                                        ↓
                                                        if   (1 = = 1) return (1);
```

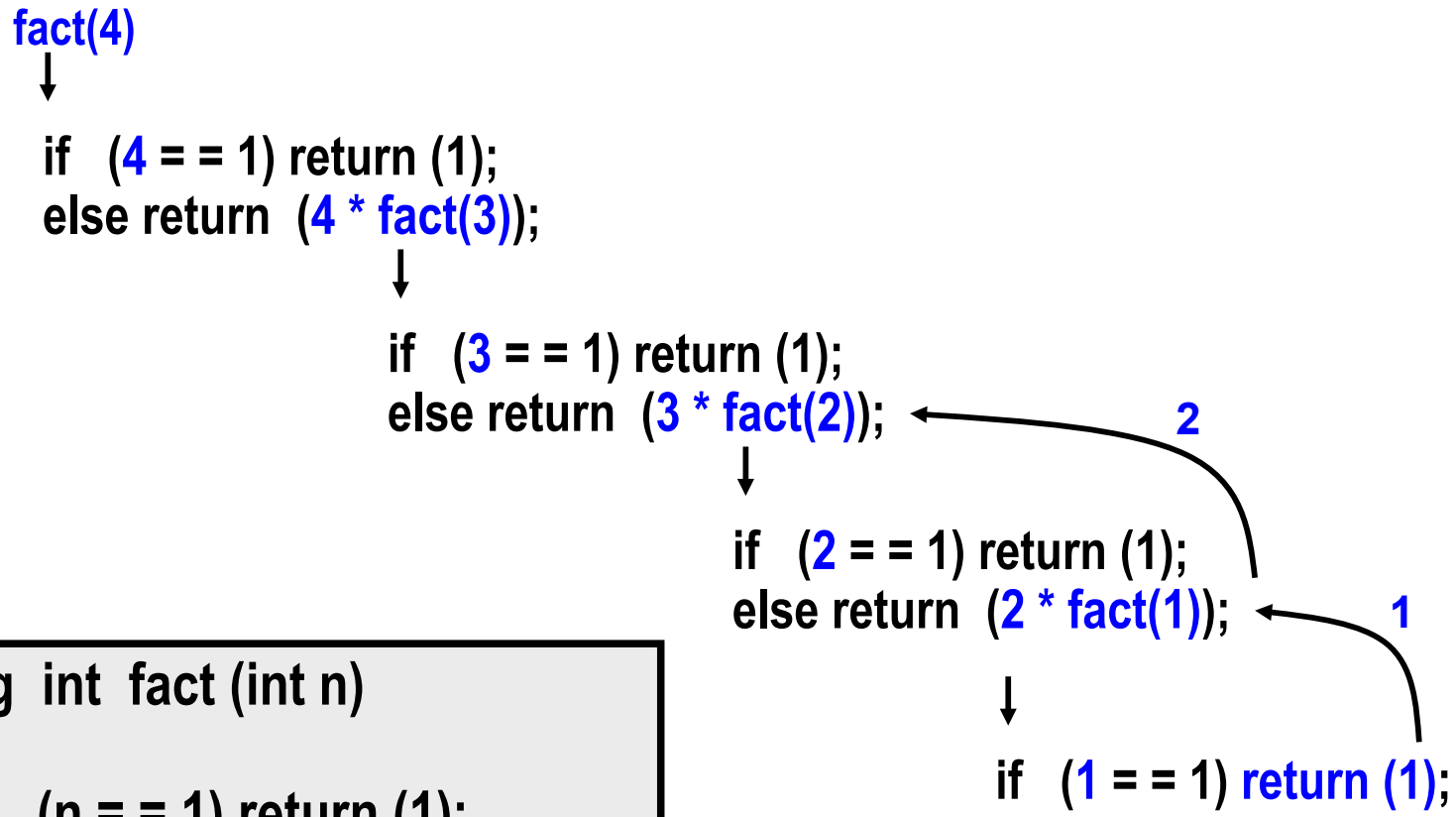
```
long  int  fact (int n)
{
   if   (n = = 1) return (1);
   else return  (n * fact(n-1));
}
```

Factorial Execution



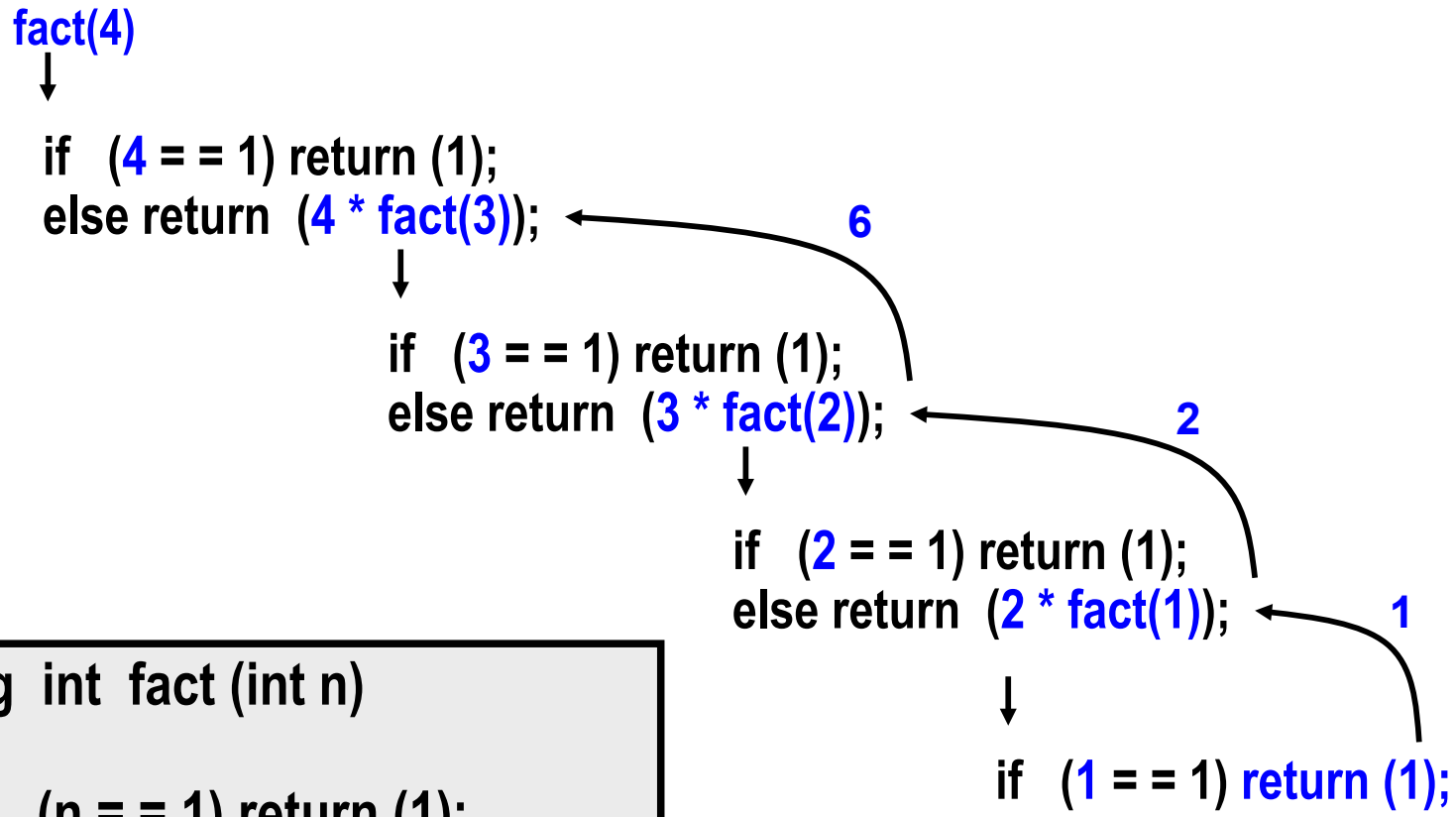

```
long  int  fact (int n)
{
   if  (n = = 1) return (1);
   else return  (n * fact(n-1));
}
```

Factorial Execution



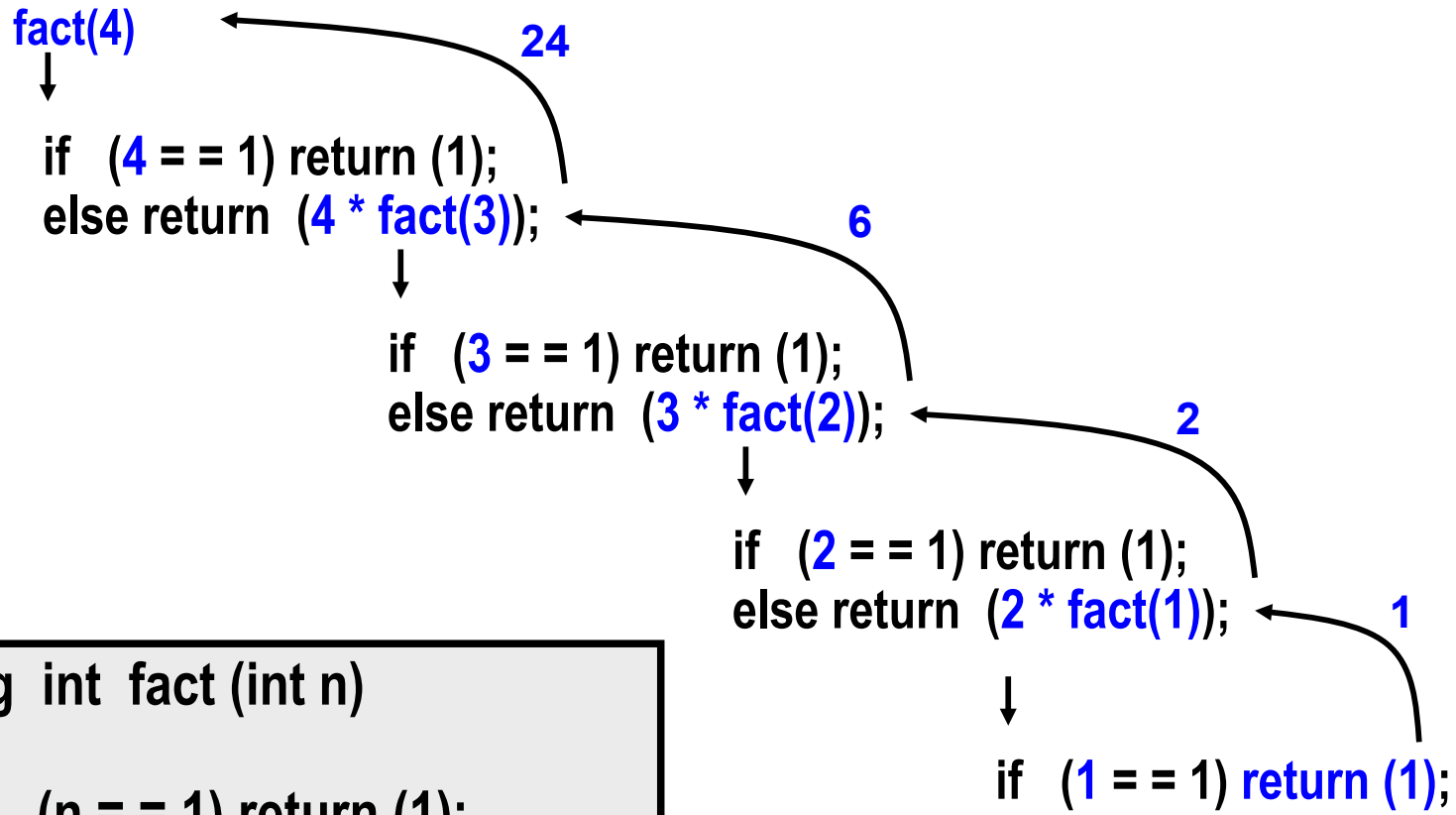

```
long  int  fact (int n)
{
   if  (n = = 1) return (1);
   else return  (n * fact(n-1));
}
```

Factorial Execution




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

Factorial Execution




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

Example: Finding max in an array

```
int findMax(int A[ ], int n)
{
        int temp;
        if (n==1)
        {
                return A[0];
        }
        temp = findMax(A, n-1);
        if (A[n-1] > temp)
                return A[n-1];
        else return temp;
}
```

Terminating condition. Small size problem that you know how to solve directly without calling any functions

Recursive call. Find the max in the first n-1 elements (exact same problem, just solved on a smaller array).

Important things to remember

- Think how the whole problem (finding max of n elements in A) can be solved if you can solve the exact same problem on a smaller problem (finding max of first n-1 elements of the array). But then, do NOT think how the smaller problem will be solved, just call the function recursively and assume it will be solved.
- When you write a recursive function
  - First write the terminating/base condition
  - Then write the rest of the function
  - Always double-check that you have both

Back to Factorial: Look at the variable addresses (a slightly different program) !

```
int main()
{
    int  x,y;
    scanf("%d",&x);
    y = fact(x);
    printf  ("M: x= %d, y = %d\n", x,y);
    return 0;
}
int fact(int data)
{ int val = 1;
  printf("F: data = %d, &data = %u \n
    &val = %u\n", data, &data, &val);
  if (data>1) val = data*fact(data-1);
    return val;
}
```

Output

```
4
F: data = 4, &data = 3221224528
 &val = 3221224516
F: data = 3, &data = 3221224480
 &val = 3221224468
F: data = 2, &data = 3221224432
 &val = 3221224420
F: data = 1, &data = 3221224384
 &val = 3221224372
M: x= 4, y = 24
```

- The memory addresses for the variable data are different in different calls!
- They are not the same variable.
- Each function call will have its own set of variables, even if the name of the variable is the same as it is the same function being called
- Change made to one will not be seen by the calling function on return

```
int main()
{
    int  x,y;
    scanf("%d",&x);
    y = fact(x);
    printf  ("M: x= %d, y = %d\n", x,y);
    return 0;
}
int fact(int data)
{
   int val = 1, count = 0;
   if (data>1) val = data*fact(data-1);
   count++;
   printf("count = %d, data = %d\n",
count, data);
   return val;
}
```

Output

```
4
count = 1, data = 1
count = 1, data = 2
count = 1, data = 3
count = 1, data = 4
M: x= 4, y = 24
```

- Count did not change even though ++ done!
- Each call does it on its own copy, lost on return

Fibonacci Numbers

Fibonacci recurrence:

fib(n) = 1 if n = 0 or 1;

= fib(n – 2) + fib(n – 1)

otherwise;

```
int fib (int n){
    if (n == 0 or n == 1)
        return 1;    [Base]
    return fib(n-2) + fib(n-1) ;
                          [Recursive]
 }
```

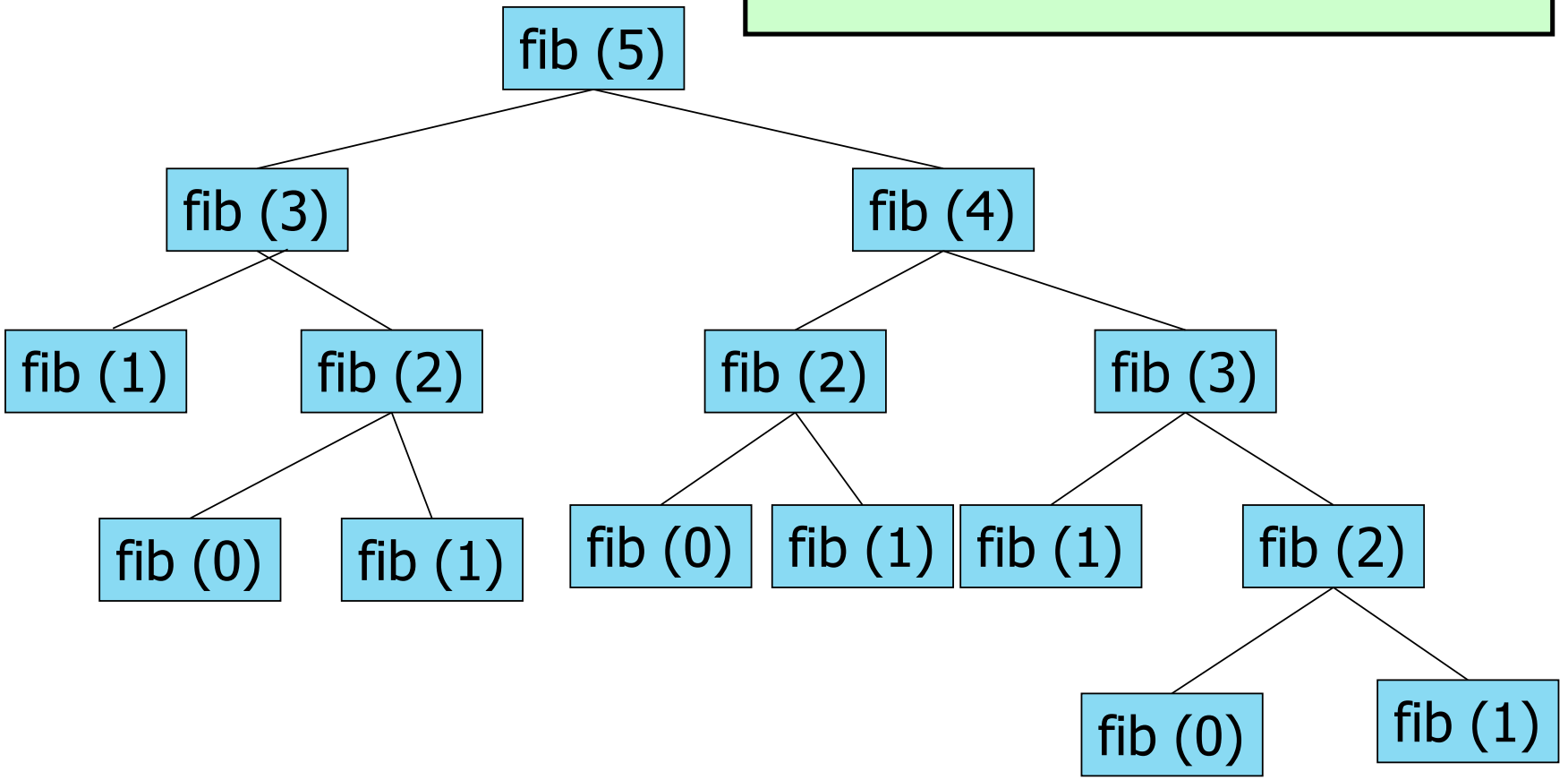
```
int fib (int n)    {
   if (n == 0 || n == 1)
         return 1;
   return fib(n-2) + fib(n-1) ;
}
```

Fibonacci recurrence:

fib(n) = 1 if n = 0 or 1;

= fib(n – 2) + fib(n – 1)

otherwise;



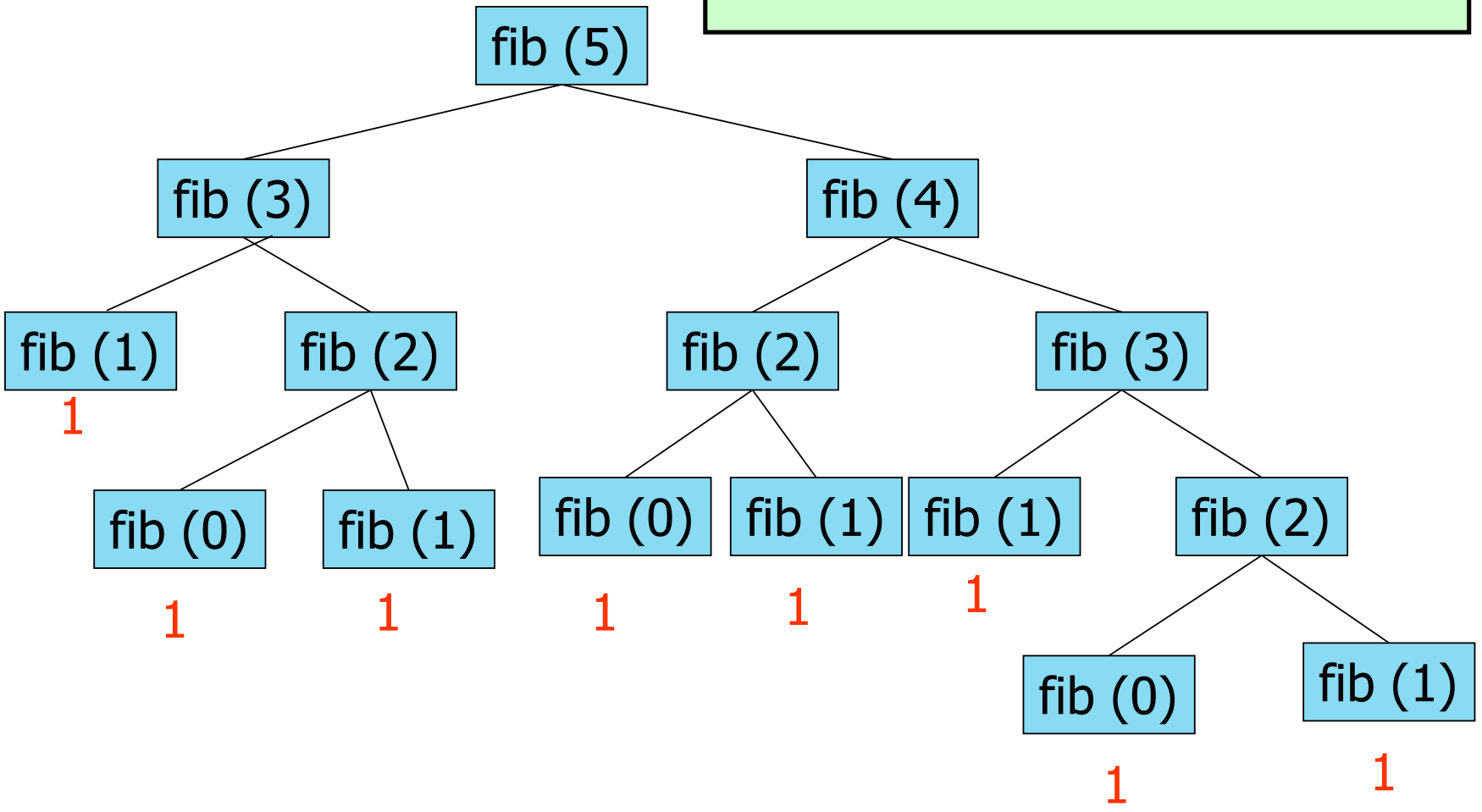
```
int fib (int n)    {
   if (n == 0 || n == 1)
          return 1;
   return fib(n-2) + fib(n-1) ;
}
```

Fibonacci recurrence:

fib(n) = 1 if n = 0 or 1;

= fib(n – 2) + fib(n – 1)

otherwise;



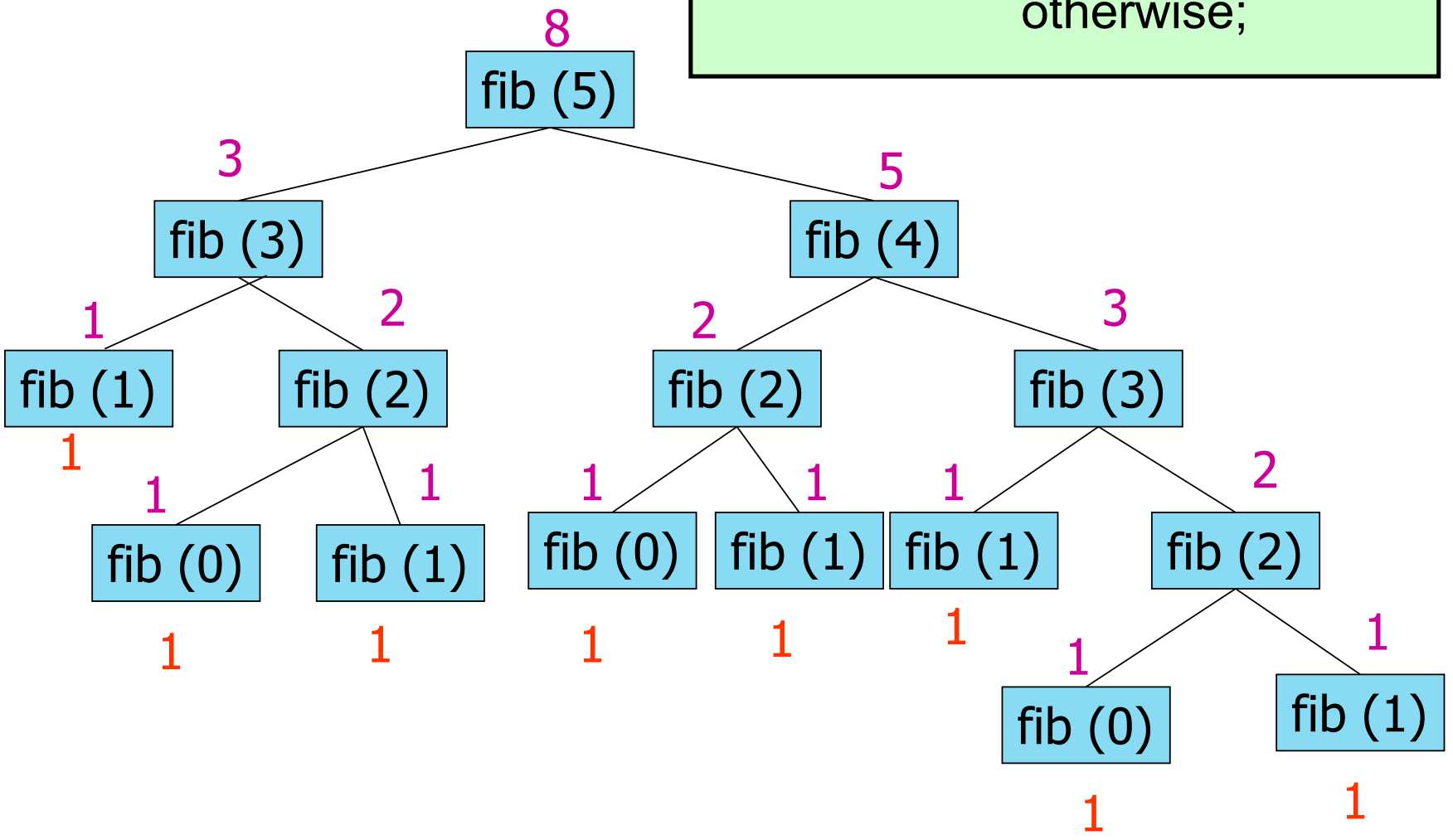
```
int fib (int n)    {
   if (n==0 || n==1)
          return 1;
   return fib(n-2) + fib(n-1) ;
}
```

Fibonacci recurrence:

fib(n) = 1 if n = 0 or 1;

= fib(n – 2) + fib(n – 1)

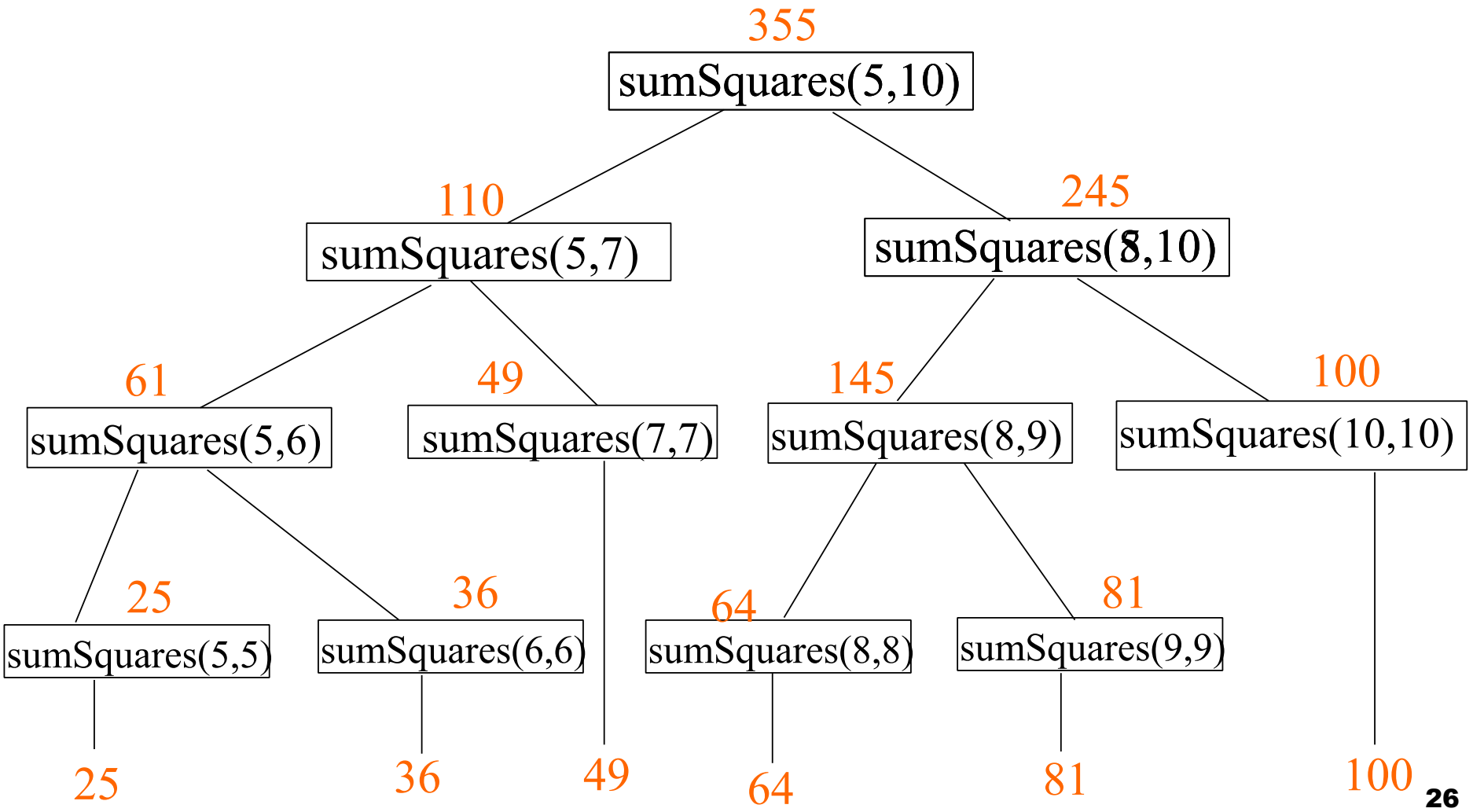
otherwise;



Example: Sum of Squares

```
int sumSquares (int m, int n)
{
    int middle ;
    if (m == n) return(m*m);
    else
    {
        middle = (m+n)/2;
        return (sumSquares(m,middle)
                    + sumSquares(middle+1,n));
     }
}
```

Annotated Call Tree



Example: Printing the digits of an Integer in Reverse

- Print the last digit, then print the remaining number in reverse
  - Ex: If integer is 743, then reversed is print 3 first, then print the reverse of 74

```
void printReversed(int i)
{
    if (i < 10)   {
       printf("%d\n", i); return;
    }
    else {
        printf("%d", i%10);
        printReversed(i/10);
    }
}
```

Counting Zeros in a Positive Integer

- Check last digit from right
  - If it is 0, number of zeros = 1 + number of zeroes in remaining part of the number
  - If it is non-0, number of zeros = number of zeroes in remaining part of the number

```
int zeros(int number)
{
    if(number<10) return 0;
    if (number%10 == 0)
            return(1+zeros(number/10));
    else
       return(zeros(number/10));
}
```

Example: Binary Search

- Searching for an element k in a sorted array A with n elements
- Idea:
  - Choose the middle element A[n/2]
  - If k == A[n/2], we are done
  - If k < A[n/2], search for k between A[0] and A[n/2 -1]
  - If k > A[n/2], search for k between A[n/2 + 1] and A[n-1]
  - Repeat until either k is found, or no more elements to search
- Requires less number of comparisons than linear search in the worst case ($\log_2 n$ instead of n)

```
int binsearch(int A[ ], int low, int high, int k)
{
     int mid;
     printf("low = %d, high = %d\n", low, high);
     if (low > high)
          return 0;
    mid = (low + high)/2;
    printf("mid = %d, A[%d] = %d\n\n", mid, mid, A[mid]);
    if (A[mid] == k)
          return 1;
    else {
          if (A[mid] > k)
                return (binsearch(A, low, mid-1, k));
          else
                return(binsearch(A, mid+1, high, k));
    }
}
```

```
int main()
{
    int A[25], n, k, i, found;

    scanf("%d", &n);
    for (i=0; i<n; i++) scanf("%d", &A[i]);
    scanf("%d", &k);
    found = binsearch(A, 0, n-1, k);
    if (found == 1)
        printf("%d is present in the array\n", k);
    else
        printf("%d is not present in the array\n", k);
}
```

Output

```
8
9 11 14 17 19 20 23 27
21
low = 0, high = 7
mid = 3, A[3] = 17

low = 4, high = 7
mid = 5, A[5] = 20

low = 6, high = 7
mid = 6, A[6] = 23

low = 6, high = 5
21 is not present in the array
```

```
8
9 11 14 17 19 20 23 27
14
low = 0, high = 7
mid = 3, A[3] = 17

low = 0, high = 2
mid = 1, A[1] = 11

low = 2, high = 2
mid = 2, A[2] = 14

14 is present in the array
```

Static Variables

- Declared using the static keyword
- Static variables stay in existence rather than coming and going everytime the function is called
- Example: Counting number of 0's in an array

```
int main() {

    int A[] = {1, 0, 0, 4, 5, 0};
    cntZero(A, 10, 1);
    return 0;
}
```

```
void cntZero(int A[], int n, int print)
{
   int i;
   static int count = 0;
   printf("Called with: n=%d,  count = %d\n", n, count);
   printf("&n=%p,  &count = %p\n\n", &n, &count);
   if (n == 1) {
      if (A[0] == 0) count++;
      return;
   }
   if (A[n-1] == 0) count++;
   cntZero(A, n-1, 0);
   if (print == 1)
      printf("n=%d, No. of zeros = %d\n", n, count);
   return;
}
```

Output

```
Called with: n=6, count= 0
&n=0x7ffe074392c4, &count= 0x40401c
Called with: n=5, count= 1
&n=0x7ffe074392a4, &count= 0x40401c
Called with: n=4, count= 1
&n=0x7ffe07439284, &count= 0x40401c
Called with: n=3, count= 1
&n=0x7ffe07439264, &count= 0x40401c
Called with: n=2, count= 2
&n=0x7ffe07439244, &count= 0x40401c
Called with: n=1, count= 3
&n=0x7ffe07439224, &count= 0x40401cn=6,

No. of zeros = 3
```

Another Example

```
int Fib (int, int);

int main()
{
   int n;
   scanf("%d", &n);
   if (n == 0 || n ==1)
      printf("F(%d) = %d \n", n, 1);
   else
      printf("F(%d) = %d \n", n,
   Fib(n,2));
   return 0;
}
```

```
int Fib(int n, int i)
{
   static int m1, m2;
   int res, temp;
   if (i==2) {m1 =1; m2=1;}
   if (n == i)  res = m1+ m2;
   else
    {   temp = m1;
        m1 = m1+m2;
        m2 = temp;
       res = Fib(n, i+1);
    }
   return res;
}
```

Static Variables: See the addresses!

```
int Fib(int n, int i)
{
  static int m1, m2;
  int res, temp;
  if (i==2) {m1 =1; m2=1;}
  printf("F: m1=%d, m2=%d, n=%d,
                    i=%d\n", m1,m2,n,i);
  printf("F: &m1=%u, &m2=%u\n",
                              &m1,&m2);
  printf("F: &res=%u, &temp=%u\n",
                        &res,&temp);
  if (n == i)  res = m1+ m2;
  else {  temp = m1;  m1 = m1+m2;
       m2 = temp;
       res = Fib(n, i+1);    }
  return res;
}
```

Output

```
5
F: m1=1, m2=1, n=5, i=2
F: &m1=134518656, &m2=134518660
F: &res=3221224516, &temp=3221224512
F: m1=2, m2=1, n=5, i=3
F: &m1=134518656, &m2=134518660
F: &res=3221224468, &temp=3221224464
F: m1=3, m2=2, n=5, i=4
F: &m1=134518656, &m2=134518660
F: &res=3221224420, &temp=3221224416
F: m1=5, m2=3, n=5, i=5
F: &m1=134518656, &m2=134518660
F: &res=3221224372, &temp=3221224368
F(5) = 8
```

Common Errors in Writing Recursive Functions

- Non-terminating Recursive Function (Infinite recursion)
  - No base case

```
int badFactorial(int x) {
   return x * badFactorial(x-1);
}
```

  - The base case is never reached

```
int anotherBadFactorial(int x) {
   if(x == 0)
      return 1;
   else
      return x*(x-1)*anotherBadFactorial(x-2);
      // When x is odd, base case never reached!!
}
```

```
int badSum2(int x)
{
   if(x==1) return 1;
   return(badSum2(x--));
}
```

Common Errors in Writing Recursive Functions

- Mixing up loops and recursion

```
int anotherBadFactorial(int x) {
   int i, fact = 0;
   if (x == 0)
      return 1;
   else {
      for (i=x; i>0; i=i-1) {
           fact = fact + x*anotherBadFactorial(x-1);
      }
      return fact;
   }
}
```

- In general, if you have recursive function calls within a loop, think carefully if you need it. Most recursive functions you will see in this course will not need this

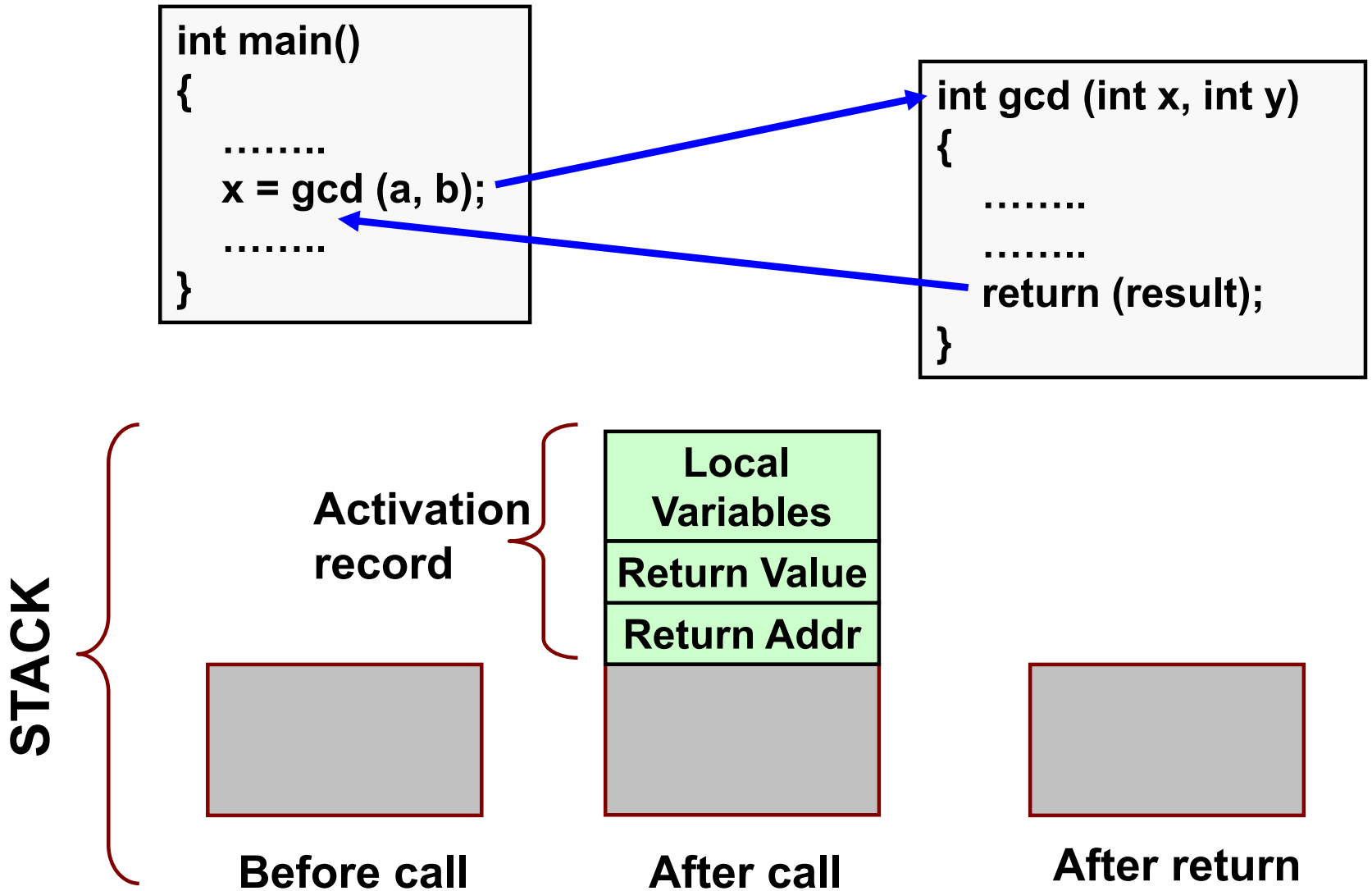
Recursion vs. Iteration

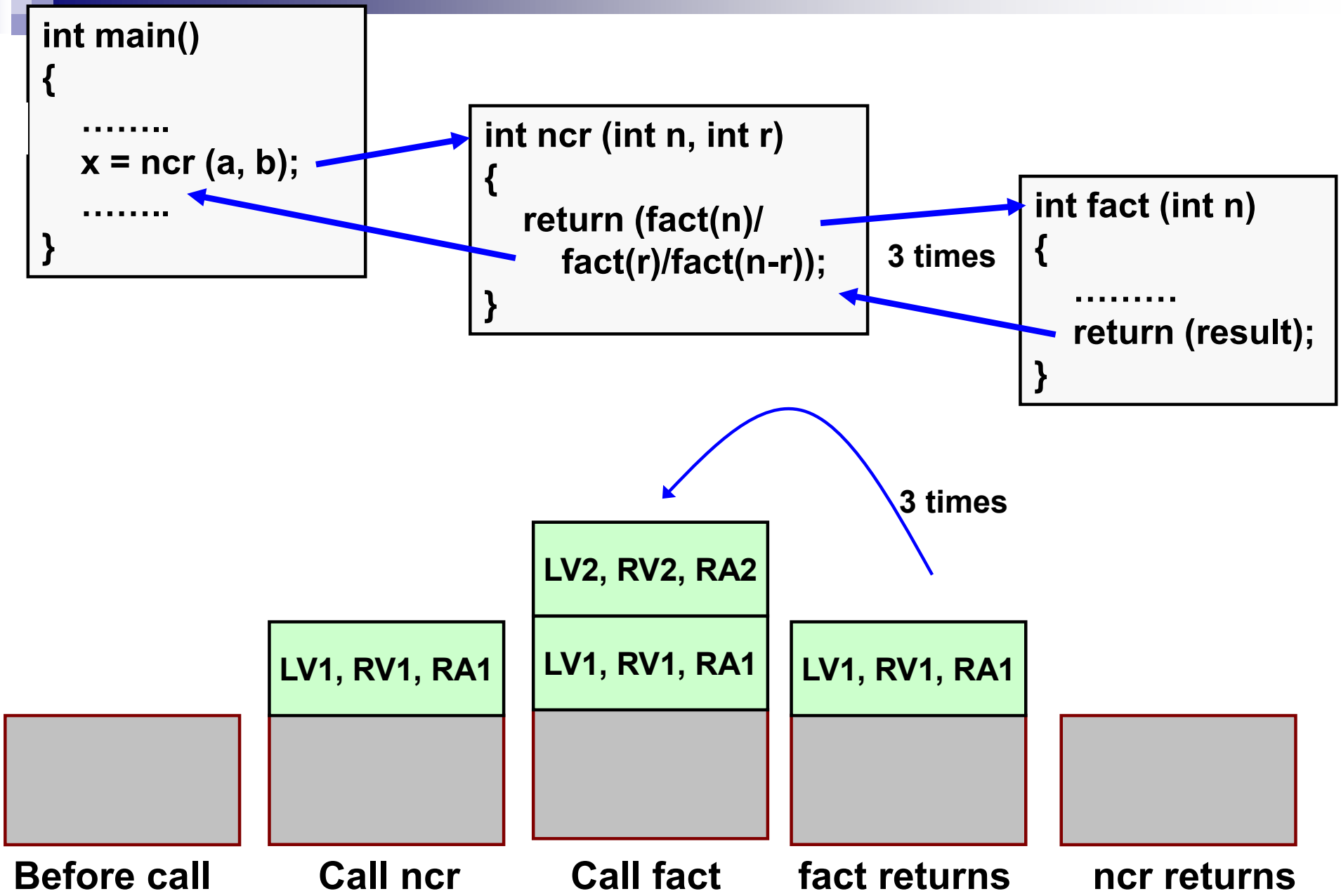
- Repetition
  - Iteration: explicit loop
  - Recursion: repeated function calls
- Termination
  - Iteration: loop condition fails
  - Recursion: base case recognized
- Both can have infinite loops
- Balance
  - Choice between performance (iteration) and good software engineering (recursion).

- Every recursive program can also be written without recursion
- Recursion is used for programming convenience, not for performance enhancement
- Sometimes, if the function being computed has a nice recursive form, then a recursive code may be more readable

How are function calls implemented?

- The following applies in general, with minor variations that are implementation dependent
  - The system maintains a stack in memory
    - Stack is a last-in first-out structure
    - Two operations on stack, push and pop
  - Whenever there is a function call, the activation record gets pushed into the stack
    - Activation record consists of the return address in the calling program, the return value from the function, and the local variables inside the function


int main()
{
……..
x = gcd (a, b);
……..
}
int gcd (int x, int y)
{
……..
……..
return (result);
}
STACK
Activation record
Local Variables
Return Value
Return Addr
Before call
After call
After return

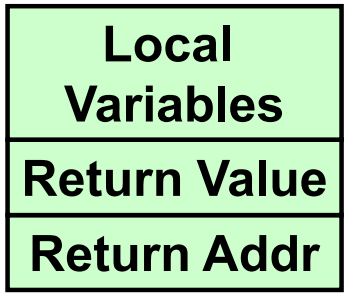

int main()
{
……..
x = ncr (a, b);
……..
}
int ncr (int n, int r)
{
return (fact(n)/
fact(r)/fact(n-r));
}
3 times
int fact (int n)
{
……..
return (result);
}
3 times
LV2, RV2, RA2
LV1, RV1, RA1
LV1, RV1, RA1
LV1, RV1, RA1
Before call
Call ncr
Call fact
fact returns
ncr returns

What happens for recursive calls?

- What we have seen ….
  - Space for activation record is allocated on the stack when a function call is made
  - Space allocated for activation record is de-allocated on the stack when the function returns
- In recursion, a function calls itself
  - Several function calls going on, with none of the function calls returning back
    - Space for activation records allocated on the stack continuously
    - Large stack space required

- Space for activation records are de-allocated, when the termination condition of recursion is reached

- We shall illustrate the process by an example of computing factorial
  - Activation record looks like:

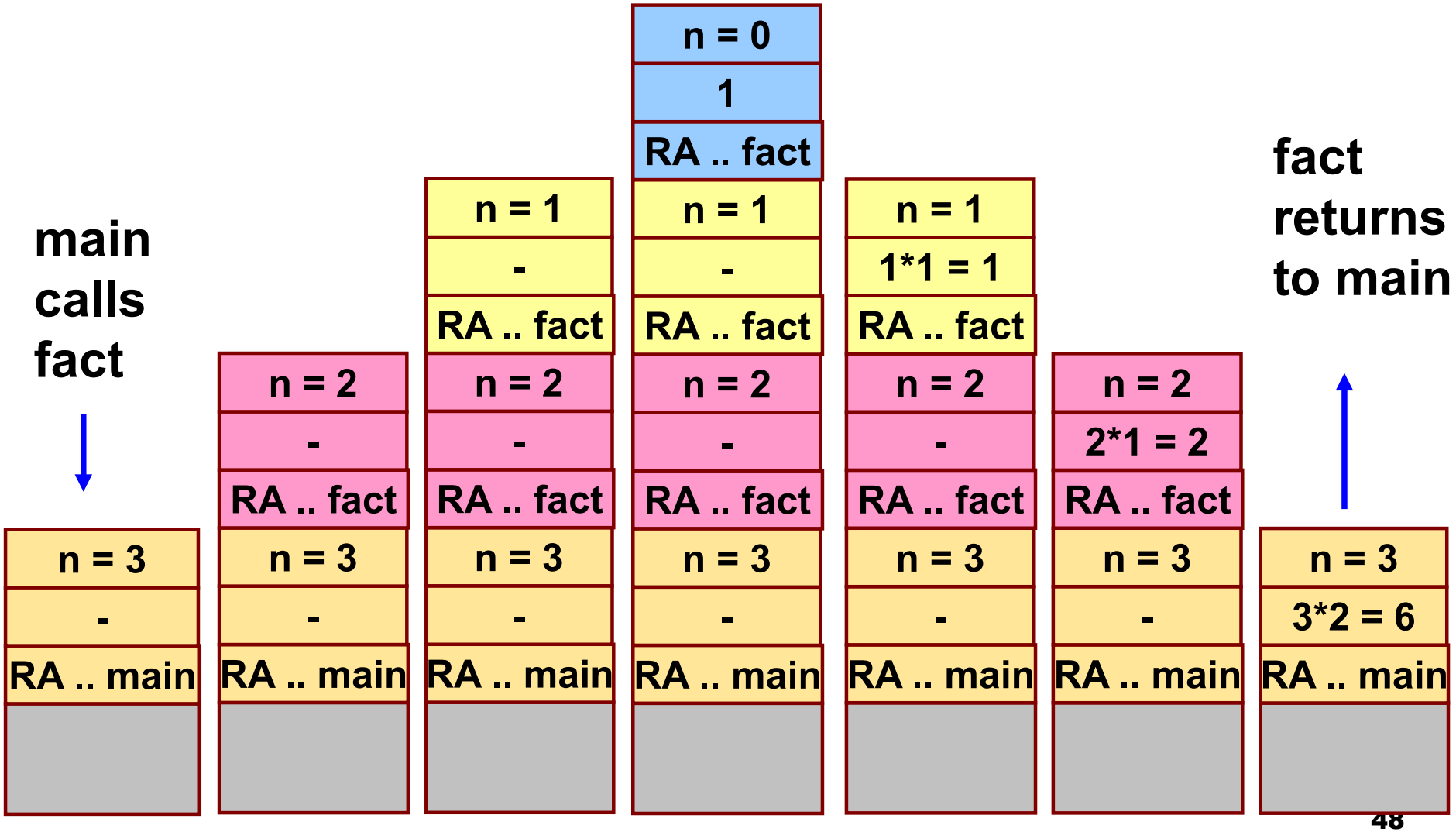


Example:: main() calls fact(3)

```
int main()
{
   int  n;
   n = 3;
   printf (“%d \n”, fact(n) );
   return 0;
}
```

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

TRACE OF THE STACK DURING EXECUTION



Do Yourself

- Trace the activation records for the following version of Fibonacci sequence

```
int   f (int n)
{
    int a, b;
    if  (n  < 2)   return (n);
    else  {
      a = f(n-1);
      b = f(n-2);
      return (a+b);
    }
}

void main() {
    printf("Fib(4) is: %d \n", f(4));
}
```

X → `a = f(n-1);`

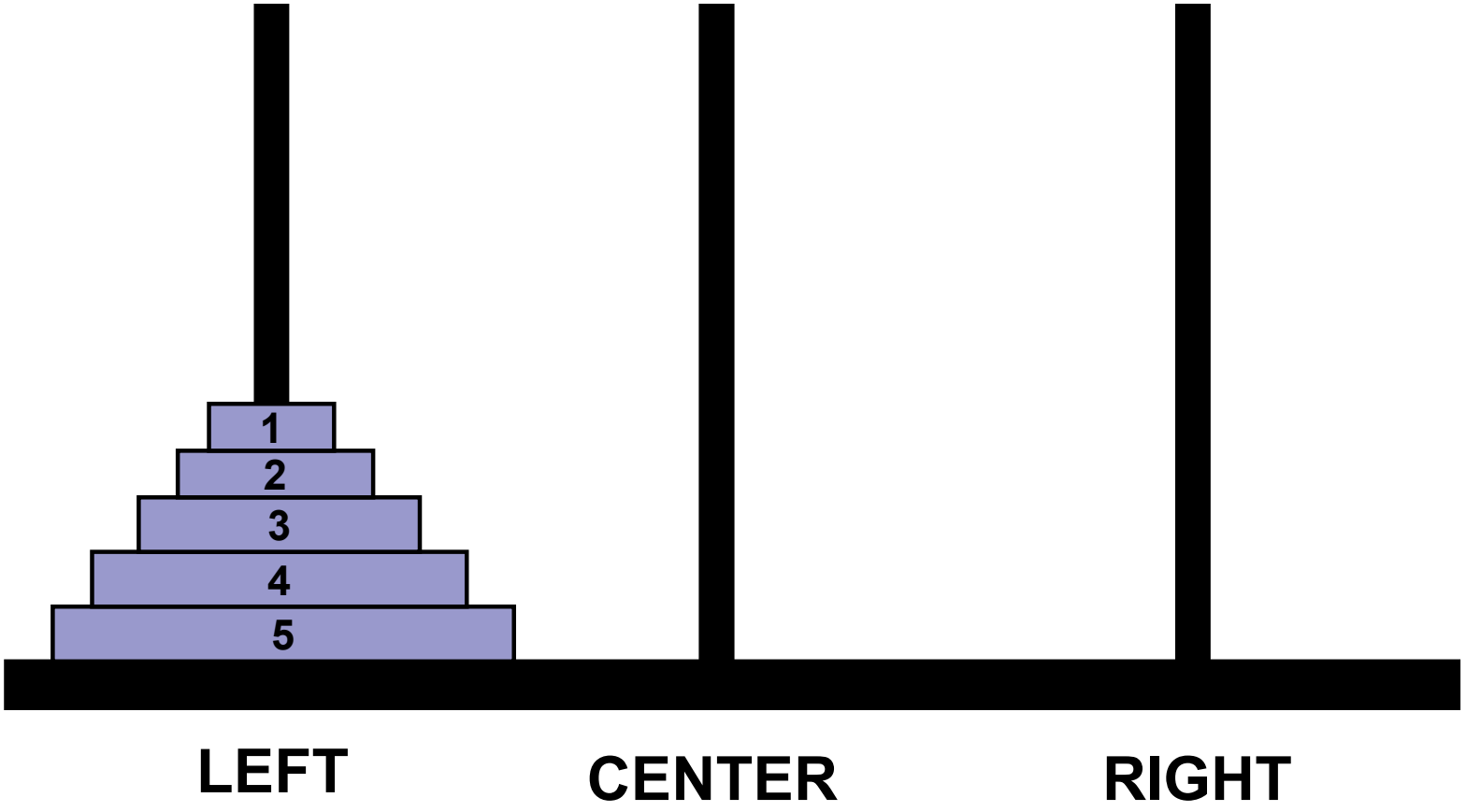
Y → `b = f(n-2);`

main → `void main() {`

Local Variables (n, a, b)
Return Value
Return Addr (either main, or X, or Y)

Additional Example

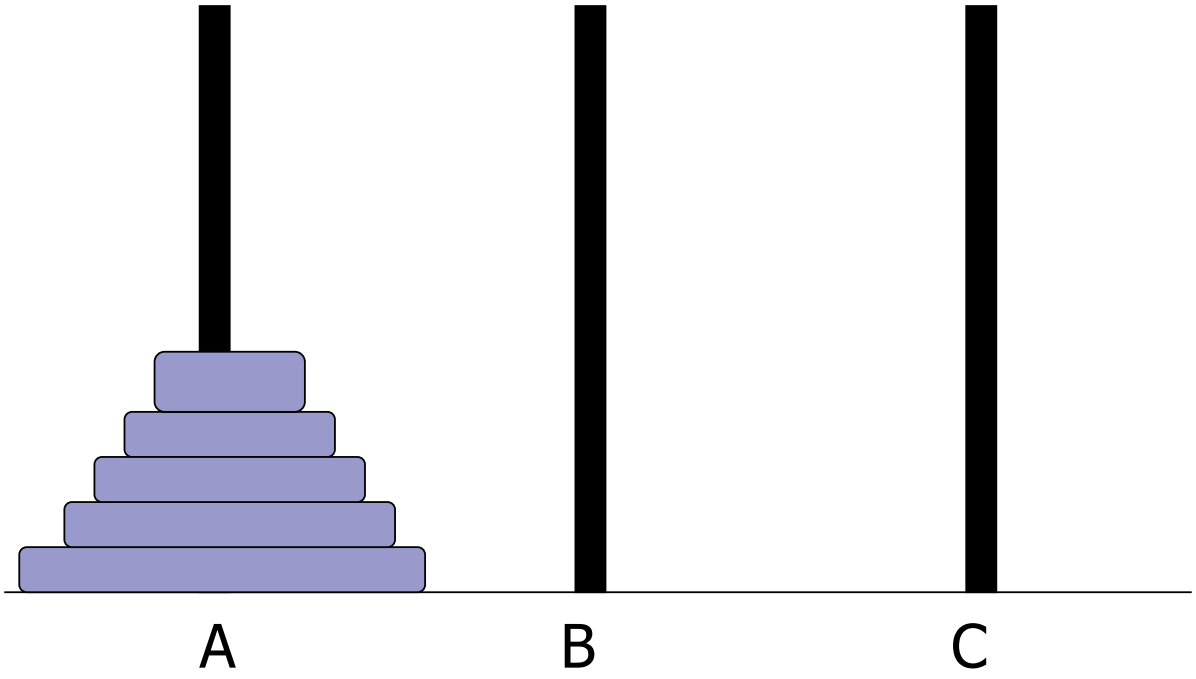
Towers of Hanoi Problem



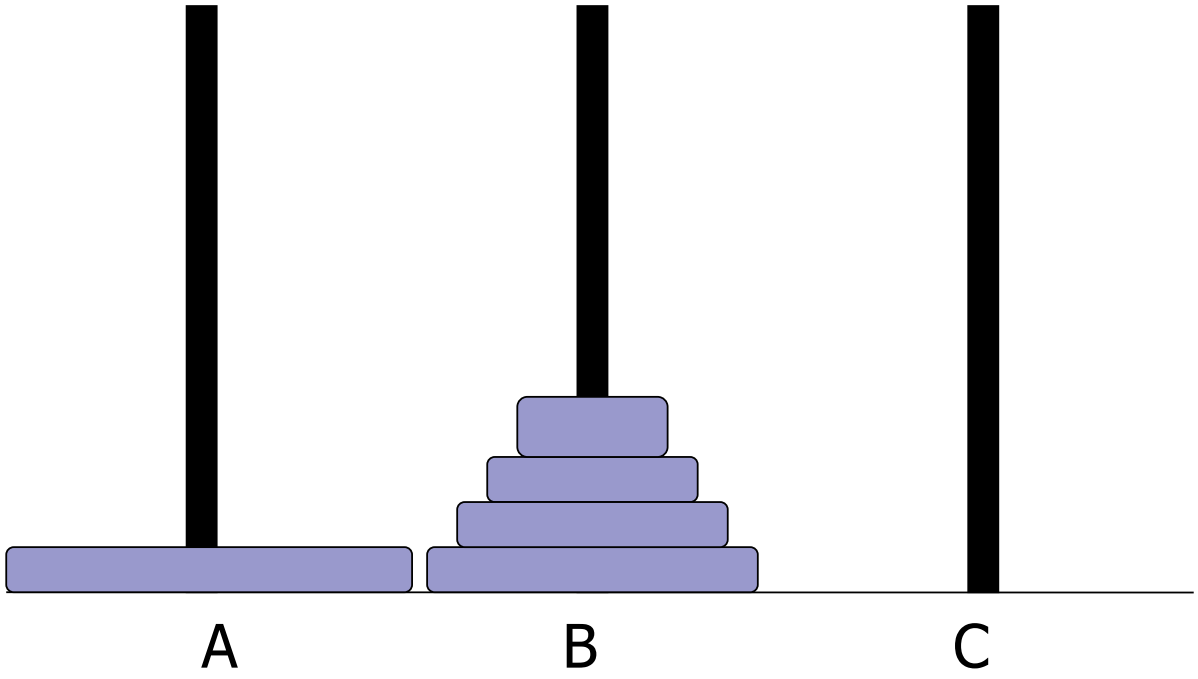
- Initially all the disks are stacked on the LEFT pole
- Required to transfer all the disks to the RIGHT pole
  - Only one disk can be moved at a time.
  - A larger disk cannot be placed on a smaller disk
- CENTER pole is used for temporary storage of disks

- Recursive statement of the general problem of n disks
  - Step 1:
    - Move the top (n-1) disks from LEFT to CENTER
  - Step 2:
    - Move the largest disk from LEFT to RIGHT
  - Step 3:
    - Move the (n-1) disks from CENTER to RIGHT

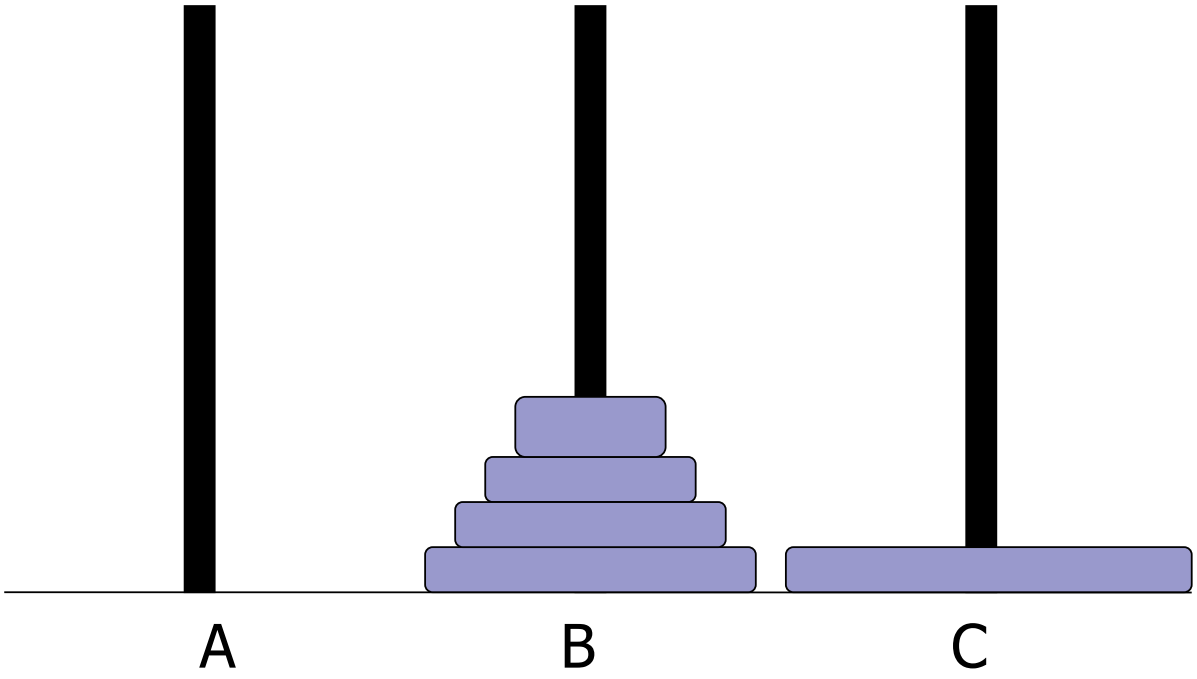
Tower of Hanoi



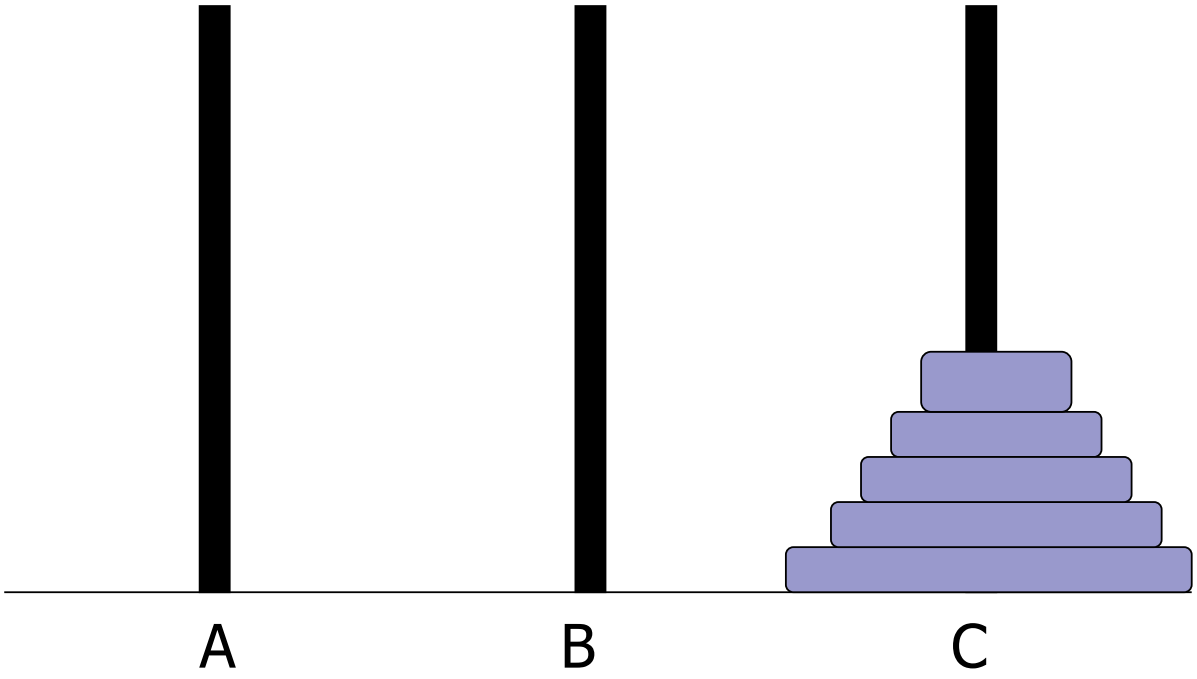
Tower of Hanoi



Tower of Hanoi



Tower of Hanoi



Towers of Hanoi function

```
void towers (int n, char from, char to, char aux)
{
 /* Base Condition */
 if (n==1)   {
          printf (“Disk 1 : %c →  &c \n”, from, to) ;
          return ;
      }
     /* Recursive Condition */
        towers (n-1, from, aux, to) ;
        ……….
        ……….
}
```

Towers of Hanoi function

```
void towers (int n, char from, char to, char aux)
{
 /* Base Condition */
 if (n==1)   {
          printf (“Disk 1 : %c →  &c \n”, from, to) ;
          return ;
      }
     /* Recursive Condition */
        towers (n-1, from, aux, to) ;
        printf (“Disk %d : %c → %c\n”, n, from, to) ;
        ……….
}
```

Towers of Hanoi function

```
void towers (int n, char from, char to, char aux)
{
 /* Base Condition */
 if (n==1)   {
          printf (“Disk 1 : %c →  %c \n”, from, to) ;
          return ;
      }
     /* Recursive Condition */
        towers (n-1, from, aux, to) ;
        printf (“Disk %d : %c → %c\n”, n, from, to) ;
        towers (n-1, aux, to, from) ;
}
```

TOH runs

```
void towers(int n, char from, char to, char aux)
{ if (n==1)
 {  printf ("Disk 1 : %c -> %c \n", from, to) ;
    return ;
 }
   towers (n-1, from, aux, to) ;
   printf ("Disk %d : %c -> %c\n", n, from, to) ;
   towers (n-1, aux, to, from) ;
}
int main()
{ int n;
  scanf("%d", &n);
  towers(n,'A',‘C',‘B');
  return 0;
}
```

Output

```
3
Disk 1 : A -> C
Disk 2 : A -> B
Disk 1 : C -> B
Disk 3 : A -> C
Disk 1 : B -> A
Disk 2 : B -> C
Disk 1 : A -> C
```

More TOH runs

```
void towers(int n, char from, char to, char aux)
{ if (n==1)
 {  printf ("Disk 1 : %c -> %c \n", from, to) ;
    return ;
 }
   towers (n-1, from, aux, to) ;
   printf ("Disk %d : %c -> %c\n", n, from, to) ;
   towers (n-1, aux, to, from) ;
}
int main()
{ int n;
  scanf("%d", &n);
  towers(n,'A',‘C',‘B');
  return 0;
}
```

Output

```
4
Disk 1 : A -> B
Disk 2 : A -> C
Disk 1 : B -> C
Disk 3 : A -> B
Disk 1 : C -> A
Disk 2 : C -> B
Disk 1 : A -> B
Disk 4 : A -> C
Disk 1 : B -> C
Disk 2 : B -> A
Disk 1 : C -> A
Disk 3 : B -> C
Disk 1 : A -> B
Disk 2 : A -> C
Disk 1 : B -> C
```