# FUNCTIONs in C

# WHAT IS C FUNCTION?

- A large C program is divided into basic building blocks called C function. C function contains set of instructions enclosed by “{ }” which performs specific operation in a C program. Actually, Collection of these functions creates a C program.

# USES OF C FUNCTIONS:

- C functions are used to avoid rewriting same logic/code again and again in a program.
- There is no limit in calling C functions to make use of same functionality wherever required.
- We can call functions any number of times in a program and from any place in a program.
- A large C program can easily be tracked when it is divided into functions.
- The core concept of C functions are, re-usability, dividing a big task into small pieces to achieve the functionality and to improve understandability of very large C programs.

# C FUNCTION DECLARATION, FUNCTION CALL AND FUNCTION DEFINITION:

- There are 3 aspects in each C function. They are,
- Function declaration or prototype –
  This informs compiler about the function name, function parameters and return value's data type.
- Function call – This calls the actual function
- Function definition – This contains all the statements to be executed.

| C functions aspects | syntax |
|---|---|
| function definition | Return_type function_name (arguments list)<br>{ Body of function; } |
| function call | function_name (arguments list); |
| function declaration | return_type function_name (argument list); |

# SIMPLE EXAMPLE PROGRAM FOR C FUNCTION

- As you know, functions should be declared and defined before calling in a C program.
- In the below program, function “square” is called from main function.
- The value of “m” is passed as argument to the function “square”. This value is multiplied by itself in this function and multiplied value “p” is returned to main function from function “square”.

# Conti..

```
#include<stdio.h>
// function prototype, also called function declaration
float square ( float x );
// main function, program starts from here

int main( )
{
   float m, n ;
   printf ( "\nEnter some number for finding square \n");
   scanf ( "%f", &m ) ;
   // function call
   n = square ( m ) ;
   printf ( "\nSquare of the given number %f is %f",m,n );
}

float square ( float x )  // function definition
{
   float  p ;
   p = x * x ;
   return ( p ) ;
}
```

# Output

```
Enter some number for finding square
2
Square of the given number 2.000000  is
   4.000000
```

# HOW TO CALL C FUNCTIONS IN A PROGRAM

- There are two ways that a C function can be called from a program.
- Call by value
- Call by reference

# CALL BY VALUE

- In call by value method, the value of the variable is passed to the function as parameter.
- The value of the actual parameter can not be modified by formal parameter.
- Different Memory is allocated for both actual and formal parameters. Because, value of actual parameter is copied to formal parameter.
- Note:
- Actual parameter – This is the argument which is used in function call.
- Formal parameter – This is the argument which is used in function definition

# EXAMPLE PROGRAM FOR C FUNCTION (USING CALL BY VALUE):

- In this program, the values of the variables “m” and “n” are passed to the function “swap”.
- These values are copied to formal parameters “a” and “b” in swap function and used.

# Conti..

```
#include<stdio.h>
// function prototype, also called function declaration
void swap(int a, int b);

int main()
{
   int m = 22, n = 44;
   // calling swap function by value
   printf(" values before swap m = %d \nand n = %d", m, n);
   swap(m, n);
}

void swap(int a, int b)
{
   int  tmp;
   tmp = a;
   a = b;
   b = tmp;
   printf(" \nvalues after swap m = %d\n and n = %d", a, b);
}
```

# Output

- values before swap m = 22  
  and n = 44  
  values after swap m = 44  
  and n = 22

# CALL BY REFERENCE:

- In call by reference method, the address of the variable is passed to the function as parameter.
- The value of the actual parameter can be modified by formal parameter.
- Same memory is used for both actual and formal parameters since only address is used by both parameters.

# EXAMPLE PROGRAM FOR C FUNCTION (USING CALL BY REFERENCE)

- In this program, the address of the variables “m” and “n” are passed to the function “swap”.
- These values are not copied to formal parameters “a” and “b” in swap function.
- Because, they are just holding the address of those variables.
- This address is used to access and change the values of the variables.
-

# Conti..

```
#include<stdio.h>
// function prototype, also called function declaration
void swap(int *a, int *b);

int main()
{
    int m = 22, n = 44;
    //  calling swap function by reference
    printf("values before swap m = %d \n and n = %d",m,n);
    swap(&m, &n);
}

void swap(int *a, int *b)
{
    int tmp;
    tmp = *a;
    *a = *b;
    *b = tmp;
    printf("\n values after swap a = %d \nand b = %d", *a, *b);
}
```

# Output

- values before swap m = 22
- and n = 44  
  values after swap a = 44

# C Program To Check A String Is Palindrome Or Not | C Programs

If the original string is equal to reverse of that string, then the string is said to be a palindrome.

**2)**Read the entered string using gets(s)**.**

**3)** Calculate the string length using string library function strlen(s) and store the length into the variable n.

**4)** i=0,c=0.

Compare the element at s[i] with the element at s[n-i-1].If both are equal then increase the c value.

Compare the remaining characters by increasing i value up t i<n/2.

**5)** If the number of characters compared is equal to the number of characters matched then the given string is the palindrome.

```
#include <string.h>
int main()
{char s[1000];
    int i,n,c=0;
    printf("Enter  the string : ");
    gets(s);
    n=strlen(s);
    for(i=0;i<n/2;i++)
    {
     if(s[i]==s[n-i-1])
     c++;}
if(c==i)
    printf("string is palindrome");
    else
        printf("string is not palindrome");
    return 0;}
```

# Using functions

```
int checkpalindrome(char *s)
{  int i,c=0,n;
   n=strlen(s);
for(i=0;i<n/2;i++)
   { if(s[i]==s[n-i-1])
    c++;}
if(c==i)
      return 1;
   else
      return 0;}
int main()
{char s[1000];
  printf("Enter  the string: ");
   gets(s);
   if(checkpalindrome(s))
   printf("string is palindrome");
   else
```

# Using Recursion

The main() calls the function checkpalindrome(char *s).

**2)** The function checkpalindrome(char *s)

**a)** i=0,c=0.Calculate the string length n using strlen(s).

**b)** if i<length of the string/2

If the element at s[i] is equal to the element at s[n-i-1] then increase the c value and i value.

The function calls itself.

The function calls itself recursively until i<n/2.

**c)** If i!<length of the string/2,

If i=c then this function returns 1 otherwise it returns 0.

**3)** If the returned value is 1 then print the given string is a palindrome. If the returned value is 0 then print the string is not a palindrome.

# Recursion

```
int checkpalindrome(char *s)
{ static int i,c=0,n=strlen(s);
   if(i<n/2)
   {  if(s[i]==s[n-i-1])
    c++;
    i++;
    checkpalindrome(s);}
else
{if(c==i)
      return 1;
      else
      return 0;}}
int main()
{char s[1000];
   printf("Enter  the string: ");
   gets(s);
   if(checkpalindrome(s))
   printf("string is palindrome");
   else
      printf("string is not palindrome");}
```

```
/* function declaration */
int max(int num1, int num2);
 int main ()
{ /* local variable definition */
int a = 100; int b = 200; int ret;
/* calling a function to get max value */
ret = max(a, b);
printf( "Max value is : %d\n", ret );
 return 0; }
/* function returning the max between two numbers */
int max(int num1, int num2)
{ /* local variable declaration */ int result;
if (num1 > num2) result = num1;
else result = num2; return result;
```

# Scope of variables

A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable it cannot be accessed. There are three places where variables can be declared in C programming language −

Inside a function or a block which is called **local** variables.

Outside of all functions which is called **global** variables.

In the definition of function parameters which are called **formal** parameters.

# Local Variables

Variables that are declared inside a function or block are called local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own. The following example shows how local variables are used. Here all the variables a, b, and c are local to main() function.

```
#include <stdio.h>
int main ()
 { /* local variable declaration */
int a, b; int c;
/* actual initialization */
a = 10; b = 20; c = a + b;
printf ("value of a = %d, b = %d and c = %d\n", a, b, c);
return 0; }
```

# Global variables

are defined outside a function, usually on top of the program. Global variables hold their values throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.

A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration. The following program show how global variables are used in a program.

```
#include <stdio.h>
/* global variable declaration */
int g;
int main ()
{ /* local variable declaration */
int a, b;
/* actual initialization */
a = 10; b = 20;
g = a + b;
printf ("value of a = %d, b = %d and g = %d\n", a, b, g);
return 0; }
```

**A program can have same name for local and global variables but the value of local variable inside a function will take preference.**

Here is an example −

```
#include <stdio.h>
/* global variable declaration */
 int g = 20; int main ()
{ /* local variable declaration */
 int g = 10;
printf ("value of g = %d\n", g);
return 0; }
```

When the above code is compiled and executed, it produces the following result −

```
value of g = 10 −
```

## Formal Parameters

**Formal parameters, are treated as local variables with-in a function and they take precedence over global variables. Following is an example**

```
#include <stdio.h>
/* global variable declaration */ int a = 20; int main () { /*
local variable declaration in main function */ int a = 10;
int b = 20; int c = 0; printf ("value of a in main() = %d\n",
a); c = sum( a, b); printf ("value of c in main() = %d\n", c);
return 0; }
/* function to add two integers */
int sum(int a, int b)
 { printf ("value of a in sum() = %d\n", a);
printf ("value of b in sum() = %d\n", b); return a + b; }
```

When the above code is compiled and executed, it produces the following result −

```
value of a in main() = 10 value of a in sum() = 10 value of
b in sum() = 20 value of c in main() = 30
```

# Initializing Local and Global Variables

When a local variable is defined, it is not initialized by the system, you must initialize it yourself. Global variables are initialized automatically by the system when you define them as follows –

| Data Type | Initial Default Value |
|---|---|
| int | 0 |
| char | '\0' |
| float | 0 |
| double | 0 |
| pointer | NULL |

It is a good programming practice to initialize variables properly, otherwise your program may produce unexpected results, because uninitialized variables will take some garbage value already available at their memory location.

# Pass Individual Array Elements to functions

```
#include <stdio.h>
 void display(int age1, int age2)
{ printf("%d\n", age1);
printf("%d\n", age2); }
int main()
{ int ageArray[] = {2, 8, 4, 12};
// pass second and third elements to display()
display(ageArray[1], ageArray[2]); return 0; }
```

# Passing arrays to functions

```
#include <stdio.h>
 float calculateSum(float num[]);
int main()
{ float result, num[] = {23.4, 55, 22.6, 3, 40.5, 18};
// num array is passed to calculateSum()
result = calculateSum(num);
printf("Result = %.2f", result); return 0; }
//function definition
float calculateSum(float num[]) { float sum = 0.0; for (int i
= 0; i < 6; ++i) { sum += num[i]; }
 return sum; }
Result = 162.50
```

# Passing arrays to functions

```
#include <stdio.h>
void displayNumbers(int num[2][2]);
 int main()
{ int num[2][2];
printf("Enter 4 numbers:\n");
for (int i = 0; i < 2; ++i)
{ for (int j = 0; j < 2; ++j)
 { scanf("%d", &num[i][j]); } }
// pass multi-dimensional array to a function
displayNumbers(num); return 0; }
```

```
void displayNumbers(int num[2][2])
{ printf("Displaying:\n");
for (int i = 0; i < 2; ++i)
{ for (int j = 0; j < 2; ++j)
{ printf("%d\n", num[i][j]); } } }
```

```
Enter 4 numbers: 2 3 4 5
Displaying:
2
3
4
5
```

```
double getAverage(int arr[], int size)
{ int i; double avg;
double sum = 0;
for (i = 0; i < size; ++i)
 { sum += arr[i]; }
 avg = sum / size; return avg; }
```

```
#include <stdio.h>
/* function declaration */
double getAverage(int arr[], int size);
int main ()
{ /* an int array with 5 elements */
int balance[5] = {1000, 2, 3, 17, 50};
double avg;
/* pass pointer to the array as an argument */
 avg = getAverage( balance, 5 ) ;
/* output the returned value */
printf( "Average value is: %f ", avg ); return 0; }
```

# Quiz time

What is the o/p

```
main()
{ int i = abc(10);
 printf("%d", --i); }
int abc(int i)
{ return(i++); }
```

Options

9

10

11

12

o/p 9

return(i++) it will first return i and then increment. i.e. 10 will be returned.

**What is the result of compiling and running this code?main() { char string[] = "Hello World"; display(string); } void display(char *string) { printf("%s", string); }**

**A.** will print Hello World

**B.** Compilation Error

**C.** will print garbage value

**D.** None of these.

Compilation error

Type mismatch in redeclaration of function display

As the function **display()** is not defined before use, compiler will assume the return type of the function which is int(default return type). But when compiler will see the actual definition of **display()**, mismatch occurs since the function **display()** is declared as void. Hence the error.

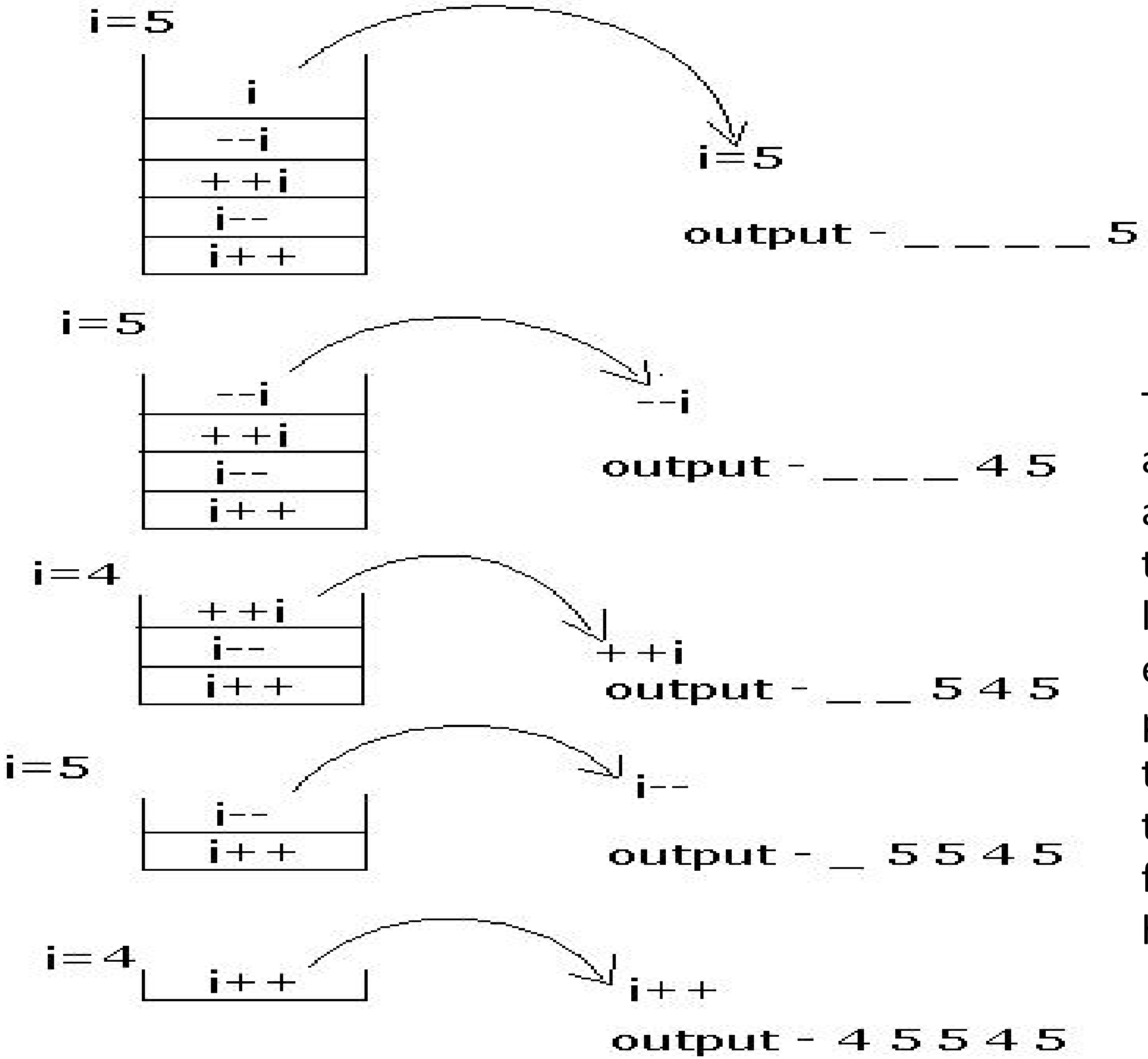
```
main() { int i = 5; printf("%d%d%d%d%d", i++, i--, ++i, --i,
i); }
```

**A.** 54544

**B.** 45445

**C.** 54554

**D.** 45545



i=5

i
--i
++i
i--
i++

i=5

output - _ _ _ _ 5

i=5

--i
++i
i--
i++

--i

output - _ _ _ 4 5

i=4

++i
i--
i++

++i

output - _ _ 5 4 5

i=5

i--
i++

i--

output - _ 5 5 4 5

i=4

i++

i++

output - 4 5 5 4 5

The arguments in a function call are pushed into the stack from left to right. The evaluation is by popping out from the stack. and the evaluation is from right to left, hence the result.

**What will be the output of the following program code?main() { static int var = 5; printf("%d ", var--); if(var) main(); }**

**A.** 5 5 5 5 5

**B.** 5 4 3 2 1

**C.** Infinite Loop

**D.** Compilation Error

**E.** None of these

```
5 4 3 2 1
```

When static storage class is given, it is initialized once. The change in the value of a static variable is retained even between the function calls. Main is also treated like any other ordinary function, which can be called recursively.

How many times the program will print "IndiaBIX" ?#include<stdio.h> int main() { printf("IndiaBIX"); main(); return 0; }

Infinite times

32767 times

65535 times

Till stack overflows

# Till stack overflows

A call stack or function stack is used for several related purposes, but the main reason for having one is to keep track of the point to which each active subroutine should return control when it finishes executing.

A stack overflow occurs when too much memory is used on the call stack.

Here function main() is called repeatedly and its return address is stored in the stack. After stack memory is full. It shows stack overflow error.

What will be the output of the program?#include<stdio.h> void fun(int*, int*); int main() { int i=5, j=2; fun(&i, &j); printf("%d, %d", i, j); return 0; } void fun(int *i, int *j) { *i = *i**i; *j = *j**j; }

5, 2

10, 4

2, 5

25, 4

**Answer:** Option d

**Explanation:**

**Step 1**: int i=5, j=2; Here variable i and j are declared as an integer type and initialized to 5 and 2 respectively.

**Step 2**: fun(&i, &j); Here the function fun() is called with two parameters &i and &j (The & denotes call by reference. So the address of the variable i and j are passed. )

**Step 3**: void fun(int *i, int *j) This function is called by reference, so we have to use * before the parameters.

**Step 4**: *i = *i**i; Here *i denotes the value of the variable i. We are multiplying 5*5 and storing the result 25 in same variable i.

**Step 5**: *j = *j**j; Here *j denotes the value of the variable j. We are multiplying 2*2 and storing the result 4 in same variable j.

**Step 6**: Then the function void fun(int *i, int *j) return back the control back to main() function.

**Step 7**: printf("%d, %d", i, j); It prints the value of variable i and j.

Hence the output is 25, 4.

# Recursion

# What is recursion?

Sometimes, the best way to solve a problem is by solving a **smaller version** of the exact same problem first

Recursion is a technique that solves a problem by solving a **smaller problem** of the same type

```
int f(int x)
{
 int y;

 if(x==0)
    return 1;
 else {
    y = 2 * f(x-1);
    return y+1;
 }
}
```

# Problems defined recursively

There are many problems whose solution can be defined recursively

Example: *n factorial*

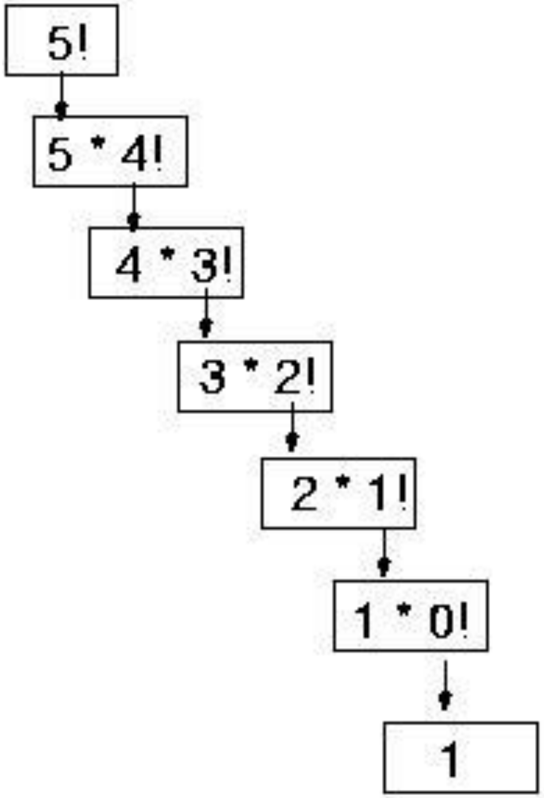
(*recursive* solution)

(*closed form* solution)

# Coding the factorial function

Recursive implementation

```
int Factorial(int n)
{
 if (n==0)  // base case
   return 1;
 else
   return n * Factorial(n-1);
}
```



5!

5 * 4!

4 * 3!

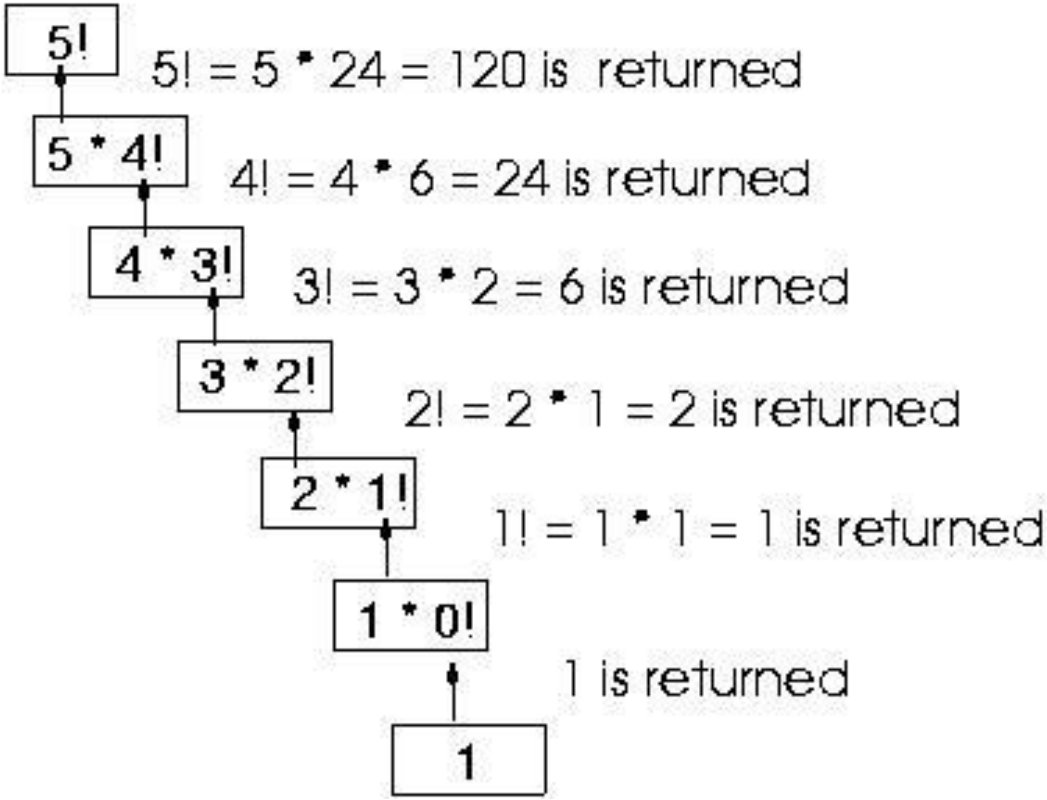
3 * 2!

2 * 1!

1 * 0!

1

Final value = 120



5!

5! = 5 * 24 = 120 is  returned

5 * 4!

4! = 4 * 6 = 24 is returned

4 * 3!

3! = 3 * 2 = 6 is returned

3 * 2!

2! = 2 * 1 = 2 is returned

2 * 1!

1! = 1 * 1 = 1 is returned

1 * 0!

1 is returned

1

# Coding the factorial function (cont.)

Iterative implementation

```
int Factorial(int n)
{
 int fact = 1;

 for(int count = 2; count <= n; count++)
   fact = fact * count;

 return fact;
}
```

# Another example:

## *n* choose *k* (combinations)

Given *n* things, how many different sets of size *k* can be chosen?

$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$, $1 < k < n$ *(recursive solution)*

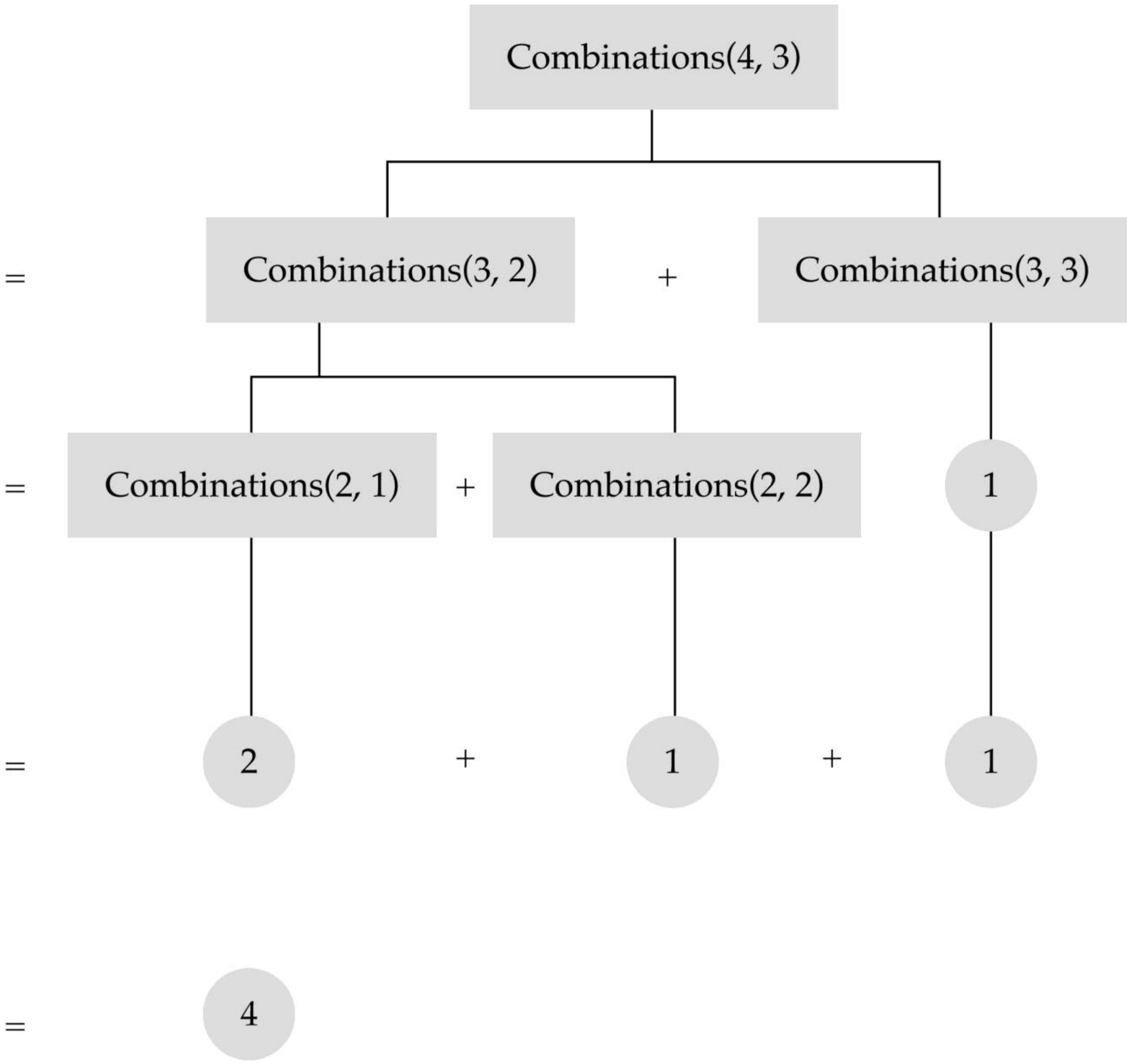
$\binom{n}{k} = \frac{n!}{k!(n-k)!}$, $1 < k < n$ *(closed-form solution)*

with base cases:

$\binom{n}{1} = n$ $(k = 1)$, $\binom{n}{n} = 1$ $(k = n)$

# *n* choose *k* (combinations)

```
int Combinations(int n, int k)
{
 if(k == 1)  // base case 1
   return n;
 else if (n == k)  // base case 2
   return 1;
 else
   return(Combinations(n-1, k) + Combinations(n-1, k-1));
}
```


Combinations(4, 3)
= Combinations(3, 2) + Combinations(3, 3)
= Combinations(2, 1) + Combinations(2, 2) 1
= 2 + 1 + 1
= 4

# Recursion vs. iteration

Iteration can be used in place of recursion

- An iterative algorithm uses a *looping construct*
- A recursive algorithm uses a *branching structure*

Recursive solutions are often less efficient, in terms of both *time* and *space*, than iterative solutions

Recursion can simplify the solution of a problem, often resulting in shorter, more easily understood source code

# How do I write a recursive function?

Determine the size factor

Determine the base case(s)

(the one for which you know the answer)

Determine the general case(s)

(the one where the problem is expressed as a smaller version of itself)

Verify the algorithm

(use the "Three-Question-Method")

# Three-Question Verification Method

The Base-Case Question:

Is there a nonrecursive way out of the function, and does the routine work correctly for this "base" case?

The Smaller-Caller Question:

Does each recursive call to the function involve a smaller case of the original problem, leading inescapably to the base case?

The General-Case Question:

Assuming that the recursive call(s) work correctly, does the whole function work correctly?

# Recursive binary search

## Non-recursive implementation

```
template<class ItemType>
void SortedType<ItemType>::RetrieveItem(ItemType& item, bool& found)
{
 int midPoint;
 int first = 0;
 int last = length - 1;

 found = false;
 while( (first <= last) && !found) {
   midPoint = (first + last) / 2;
   if (item < info[midPoint])
           last = midPoint - 1;
   else if(item > info[midPoint])
      first = midPoint + 1;
   else {
       found = true;
       item = info[midPoint];
    }
  }
}
```

# Recursive binary search (cont'd)

What is the *size factor*?

The number of elements in (*info[first] ... info[last]*)

What is the *base case(s)*?

(1) If *first > last*, return *false*
(2) If *item==info[midPoint]*, return *true*

What is the *general case*?

if *item < info[midPoint]* <u>search the first half</u>
if *item > info[midPoint]*, <u>search the second half</u>

# Recursive binary search (cont’d)

```
 template<class ItemType>
bool BinarySearch(ItemType info[], ItemType& item, int first, int last)
{
 int midPoint;

 if(first > last)  // base case 1
   return false;
 else {
   midPoint = (first + last)/2;
   if(item < info[midPoint])
     return BinarySearch(info, item, first, midPoint-1);
   else if (item == info[midPoint]) { // base case 2
     item = info[midPoint];
     return true;
  }
   else
     return BinarySearch(info, item, midPoint+1, last);
 }
}
```

# Recursive binary search (cont'd)

```
template<class ItemType>
    void SortedType<ItemType>::RetrieveItem (ItemType&
    item, bool& found)
    {
     found = BinarySearch(info, item, 0, length-1);
    }
```

# How is recursion implemented?

## What happens when a function gets called?

```
int a(int w)
{
 return w+w;
}

int b(int x)
{
 int z,y;
 ……………… // other statements
 z = a(x) + y;
 return z;
}
```

# What happens when a function is called? (cont.)

An **activation** record is stored into a stack **(run-time stack)**

1) The computer has to stop executing function ***b*** and starts executing function **a**
2) Since it needs to come back to function ***b*** later, it needs to store everything about function **b** that is going to need (**x, y, z**, and the place to start executing upon return)
3) Then, **x** from **a** is bounded to **w** from **b**
4) Control is transferred to function **a**

# What happens when a function is called? (cont.)

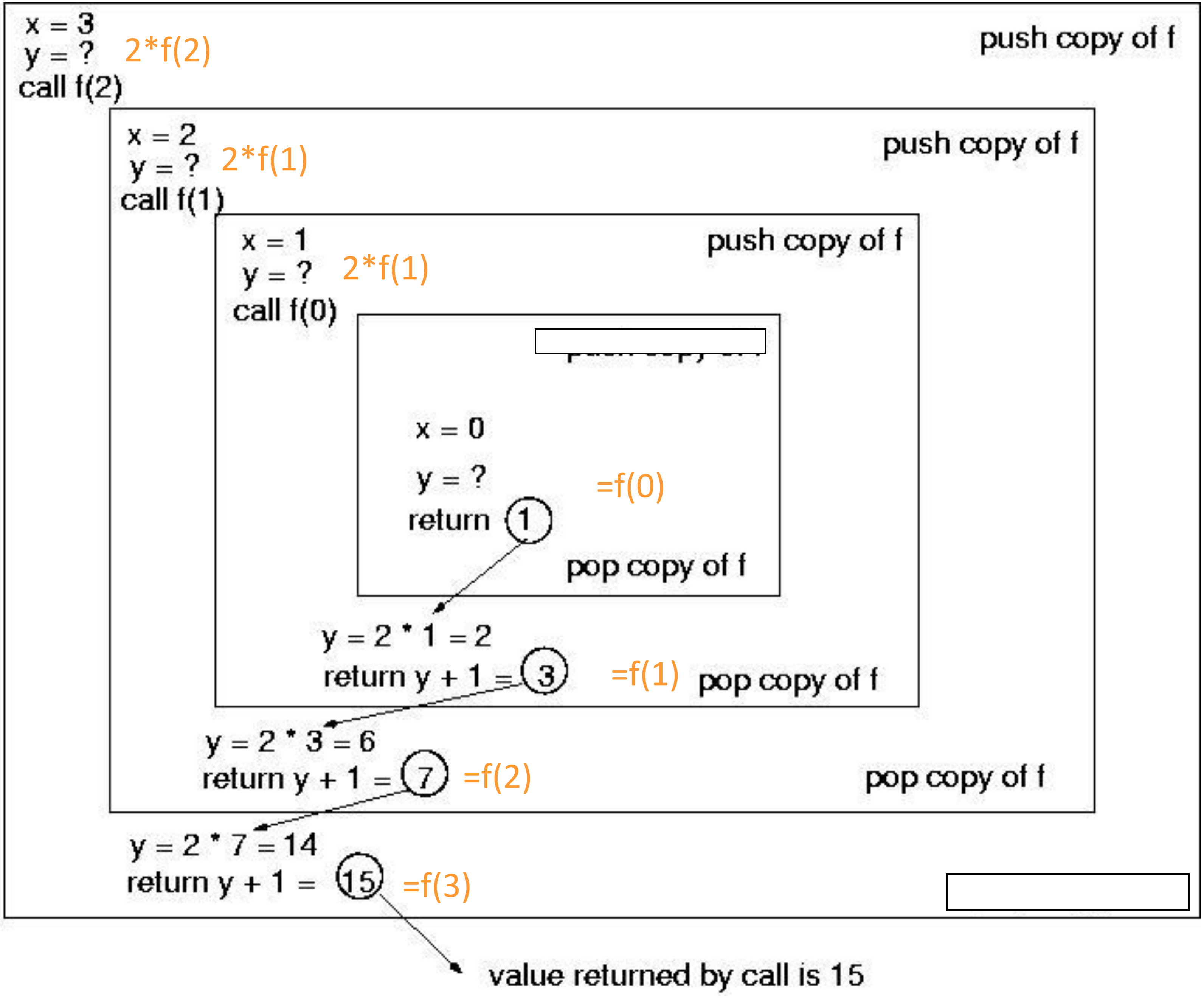
After function a is executed, the activation record is popped out of the run-time stack

All the old values of the parameters and variables in function b are restored and the return value of function a replaces a(x) in the assignment statement

# What happens when a recursive function is called?

Except the fact that the calling and called functions have the same name, there is really no difference between recursive and nonrecursive calls

```
int f(int x)
{
 int y;

 if(x==0)
  return 1;
 else {
   y = 2 * f(x-1);
   return y+1;
 }
}
```



x = 3
y = ? 2*f(2)
call f(2)

push copy of f

x = 2
y = ? 2*f(1)
call f(1)

push copy of f

x = 1
y = ? 2*f(1)
call f(0)

push copy of f

x = 0
y = ? =f(0)
return 1

pop copy of f

y = 2 * 1 = 2
return y + 1 = 3 =f(1)

pop copy of f

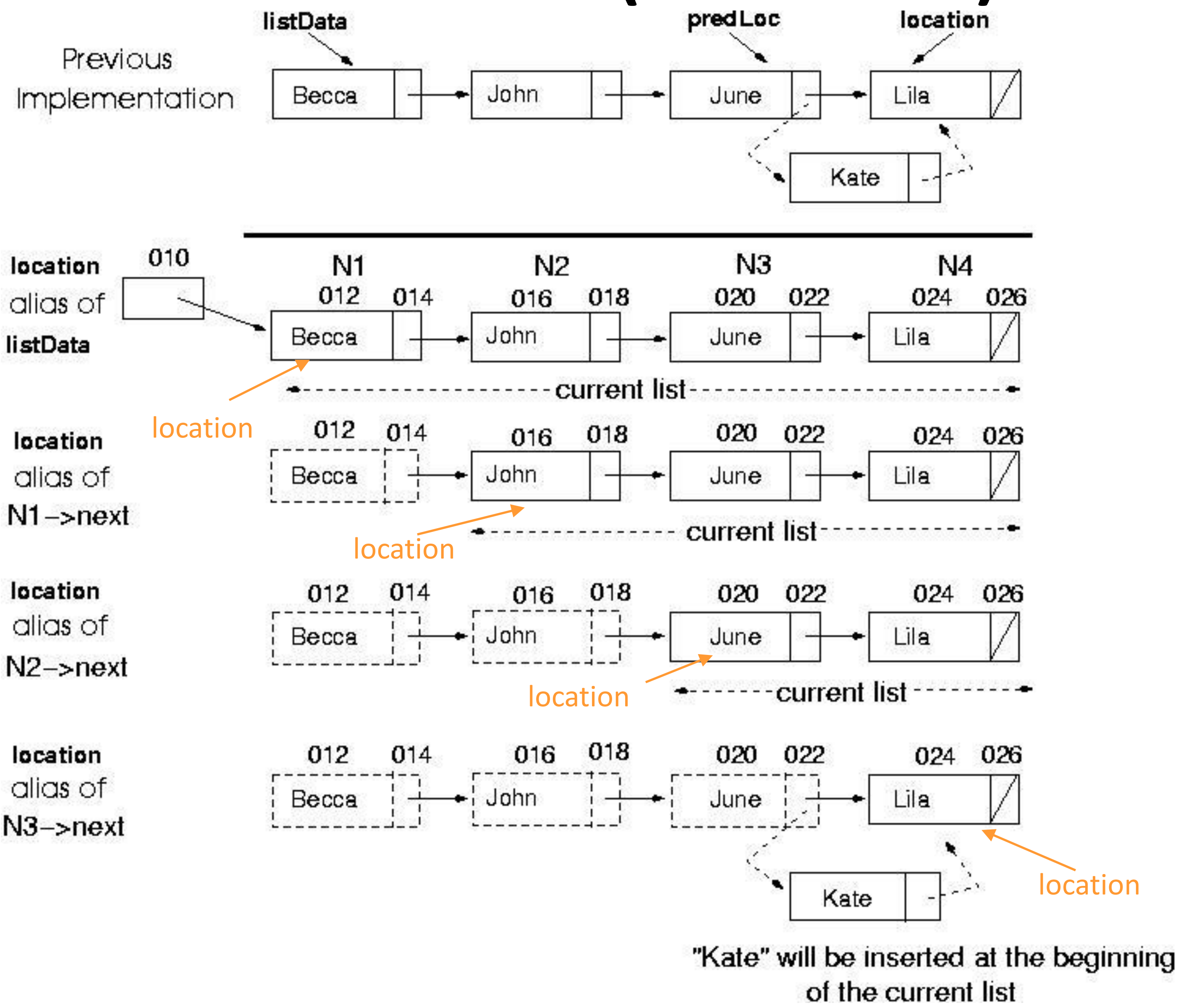
y = 2 * 3 = 6
return y + 1 = 7 =f(2)

pop copy of f

y = 2 * 7 = 14
return y + 1 = 15 =f(3)

value returned by call is 15

# Recursive InsertItem (sorted list)



# Recursive InsertItem (sorted list)

What is the *size factor*?

The number of elements in the current list

What is the *base case(s)*?

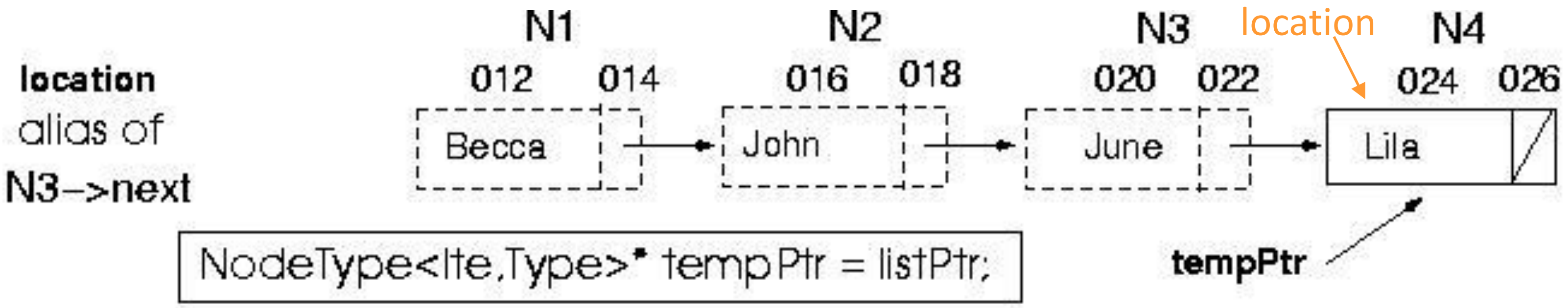
1) If the list is empty, insert item into the empty list
2) If *item* < *location->info,* insert item as the first node in the current list

What is the *general case*?

*Insert(location->next, item)*

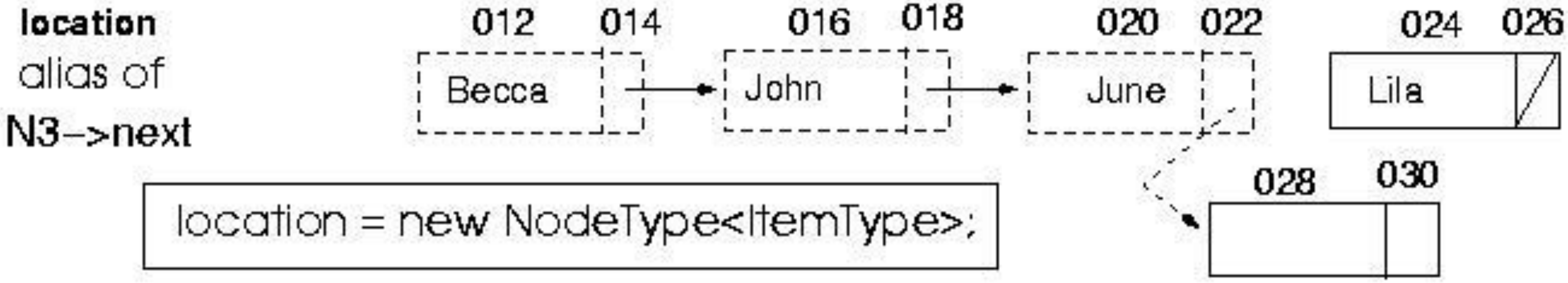
# Recursive InsertItem (sorted list)

```
template <class ItemType>
void Insert(NodeType<ItemType>* &location, ItemType item)
{
 if(location == NULL) || (item < location->info)) {  // base cases

   NodeType<ItemType>* tempPtr = location;
   location = new NodeType<ItemType>;
   location->info = item;
   location->next = tempPtr;
 }
 else
   Insert(location->next, newItem);  // general case
}

template <class ItemType>
void SortedType<ItemType>::InsertItem(ItemType newItem)
{
 Insert(listData, newItem);
}
```



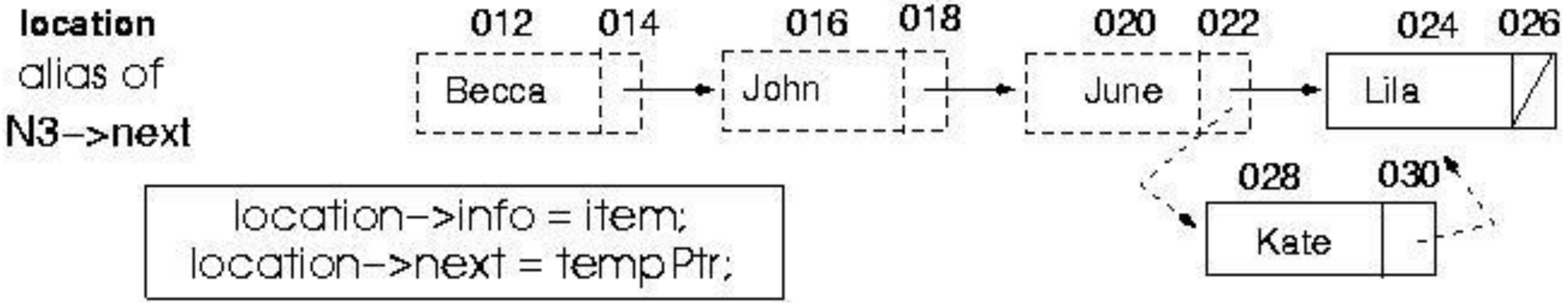
**location** alias of N3->next

```
NodeType<Ite,Type>* tempPtr = listPtr;
```



**location** alias of N3->next

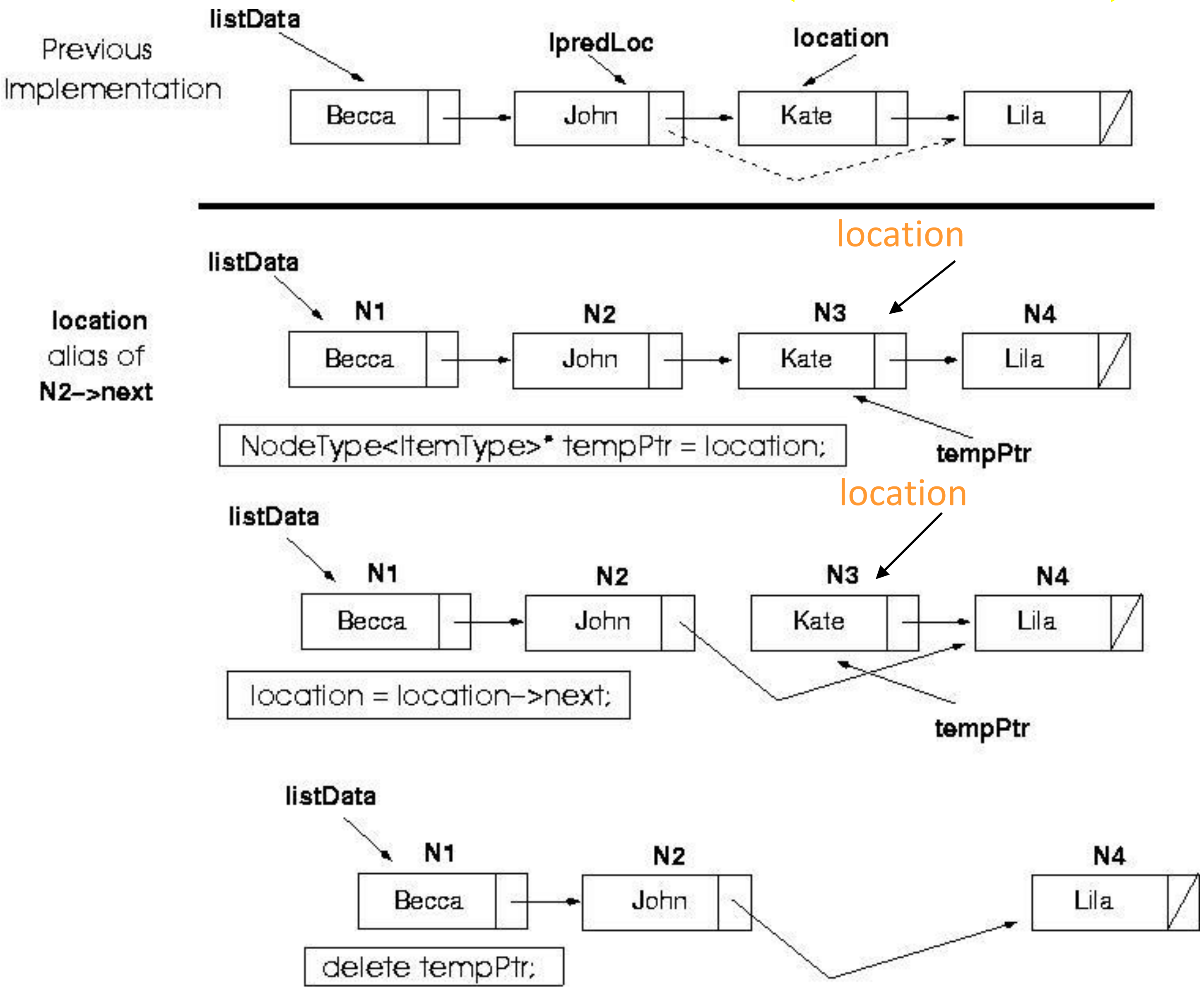
```
location = new NodeType<ItemType>;
```



**location** alias of N3->next

```
location->info = item;
location->next = tempPtr;
```

# Recursive DeleteItem (sorted list)



```
NodeType<ItemType>* tempPtr = location;
```

```
location = location–>next;
```

```
delete tempPtr;
```

# Recursive DeleteItem (sorted list) (cont.)

What is the *size factor*?

The number of elements in the list

What is the *base case(s)*?

If *item == location->info*, delete node pointed by *location*

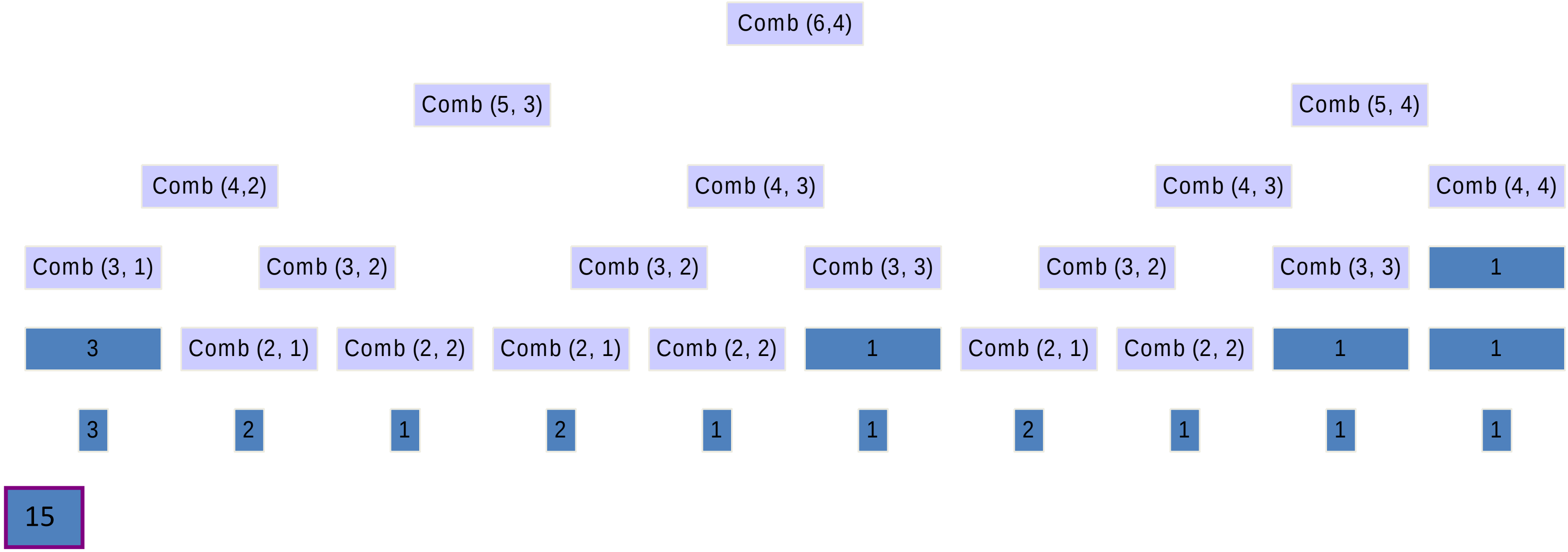
What is the *general case*?

*Delete(location->next, item)*

# Recursive DeleteItem (sorted list) (cont.)

```
template <class ItemType>
void Delete(NodeType<ItemType>* &location, ItemType item)
{
 if(item == location->info)) {

   NodeType<ItemType>* tempPtr = location;
   location = location->next;
   delete tempPtr;
 }
 else
   Delete(location->next, item);
}

template <class ItemType>
void SortedType<ItemType>::DeleteItem(ItemType item)
{
 Delete(listData, item);
}
```

# Recursion can be very inefficient is some cases



Comb (6,4)

Comb (5, 3) Comb (5, 4)

Comb (4,2) Comb (4, 3) Comb (4, 3) Comb (4, 4)

Comb (3, 1) Comb (3, 2) Comb (3, 2) Comb (3, 3) Comb (3, 2) Comb (3, 3) 1

3 Comb (2, 1) Comb (2, 2) Comb (2, 1) Comb (2, 2) 1 Comb (2, 1) Comb (2, 2) 1 1

3 2 1 2 1 1 2 1 1 1

15

# Deciding whether to use a recursive solution

When the depth of recursive calls is relatively "shallow"

The recursive version does about the same amount of work as the nonrecursive version

The recursive version is shorter and simpler than the nonrecursive solution