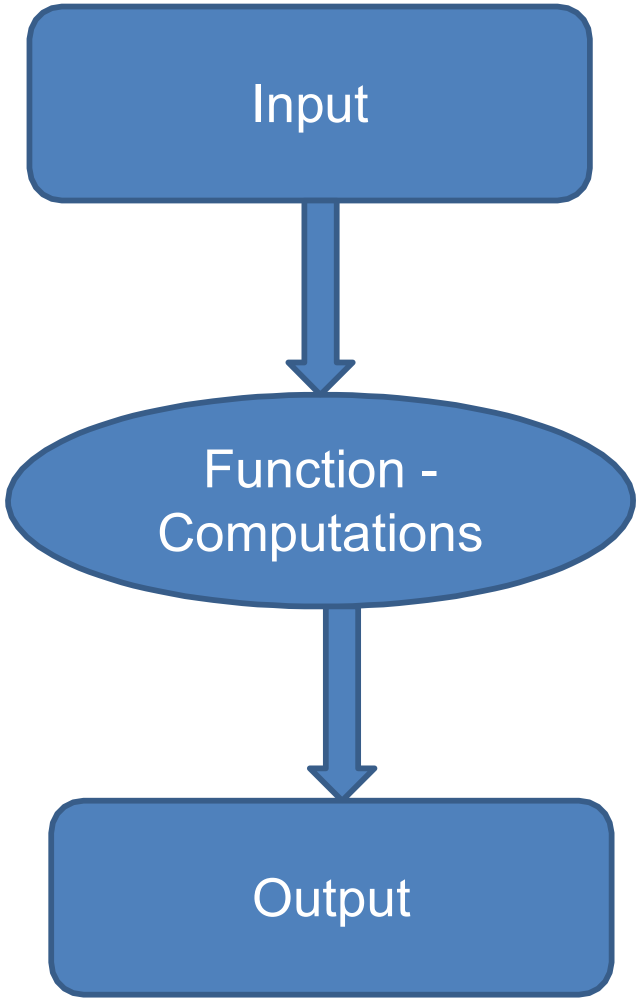
“AdvancedC Programming”

Functions

Functions

- **Definition:** A function is a self-contained program segment that carries out some specific, well-defined task.
- Every C program consists of one or more functions.
- Function should be declared or prototyped before it is being used anywhere
- One of these functions must be called "main" * (conditions apply)
- Function will carry out its intended action whenever it is called.



Defining a Function

- **Function Definition:** A function definition has two principal components:
  - the first line - Return type, Function Name, Parameters or Arguments
  - the body of the function
- Every C program consists of one or more functions.

Syntax:

```
return_type function_name( parameter list )
{
   body of the function
}
```

Function Prototypes

- **Function Declaration:** A function **declaration** tells the compiler about a function name and how to call the function.
- Parameter names are not compulsory, but data types must be mentioned.

Syntax:

return_type function_name(parameter list);

- **Function Call:** To use a function, you will have to call that function to perform the defined task.
- When a program calls a function, program control is transferred to the called function.
- A called function performs defined task and when its return statement is executed

Function prototypes…

Function prototype at the beginning of C code:

- describe what the function returns and what it takes as input
- Prototype should be followed by semicolon
- Parameter list is optional
- Default return type is integer

Example:

```
double max( double param1, double param2)

    {
     if (param1 > param2)
       {
         return param1;
                  }
     else
       {
         return param2;
    }  }
```

Examples:

```
float sqrt( float );

int maximium( int , int );
```

Return type of a C

- Every C function must specify the type of data being generated by the function.
- If the function does not generate any data, its return type can be “void”.
- The type of expression returned must match the type of the function, or be capable of being converted, otherwise we can see undefined behaviour.

Example:

```
void print_happy_birthday( int age )
  {
     printf("Congratulations on your %d th Birthday\n", age);
     return;
    }
```

Functions - Example

Example:

```
#include <stdio.h>
long int facrorial( int ); → Function prototype
int main()                                Actual Argument
{
    int n;     long int fact;
    fact = factorial( n ); → Function Call
    printf("Factorial of %d is %ld", n, fact);
    return 0;
}                                          Formal Argument
long int factorial( int n )      /* calculate the factorial of n */
{
    int i;
    long int prod = 1;  if
    (n > 1)
    for (i = 2; i <= n; ++i)
    prod *= i;
    return(prod);
}
```

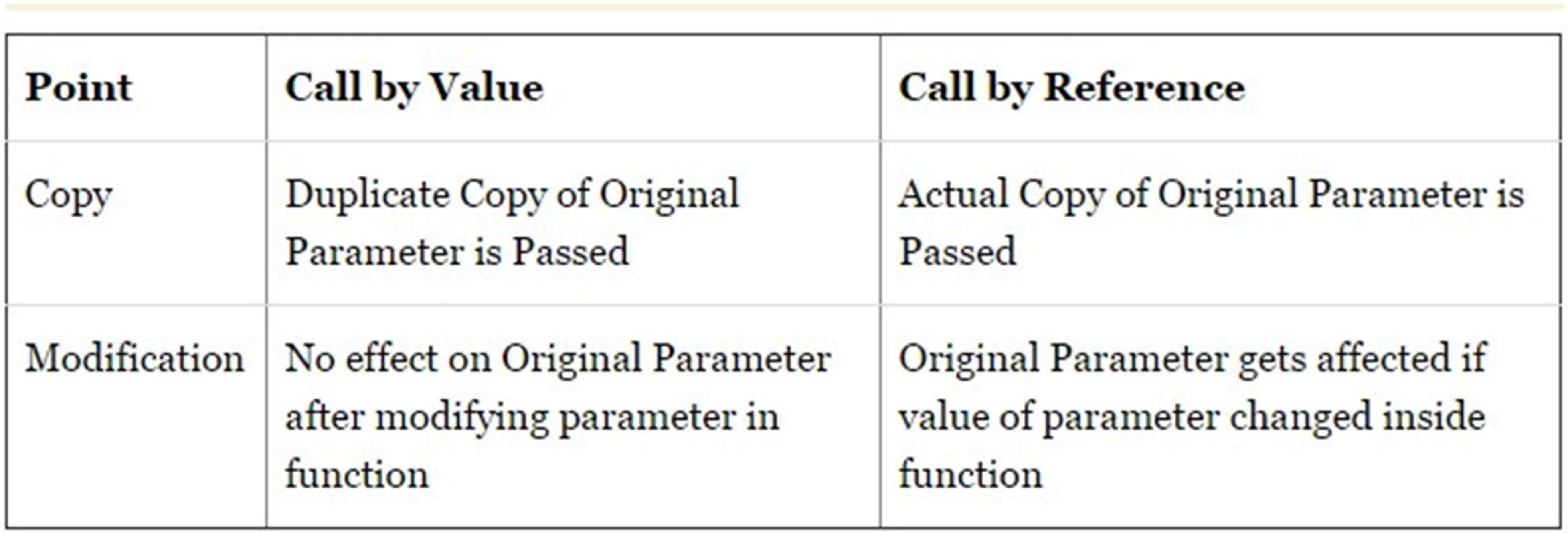
Function Parameters

- Parameters are the symbolic name for "data" that are passed to a function
  - Formal Arguments - Called function
  - Actual Arguments - Calling function
- Two ways that arguments can be passed to a function:
  - ➢ Call by value
    - o Copy of data is sent to the function being called
    - o Doesn’t affect the original data
  - ➢ Call by Reference
    - o Reference to the data is sent
    - o Reference parameter "refers" to the original data in the calling function.
    - o Changes affect the original data

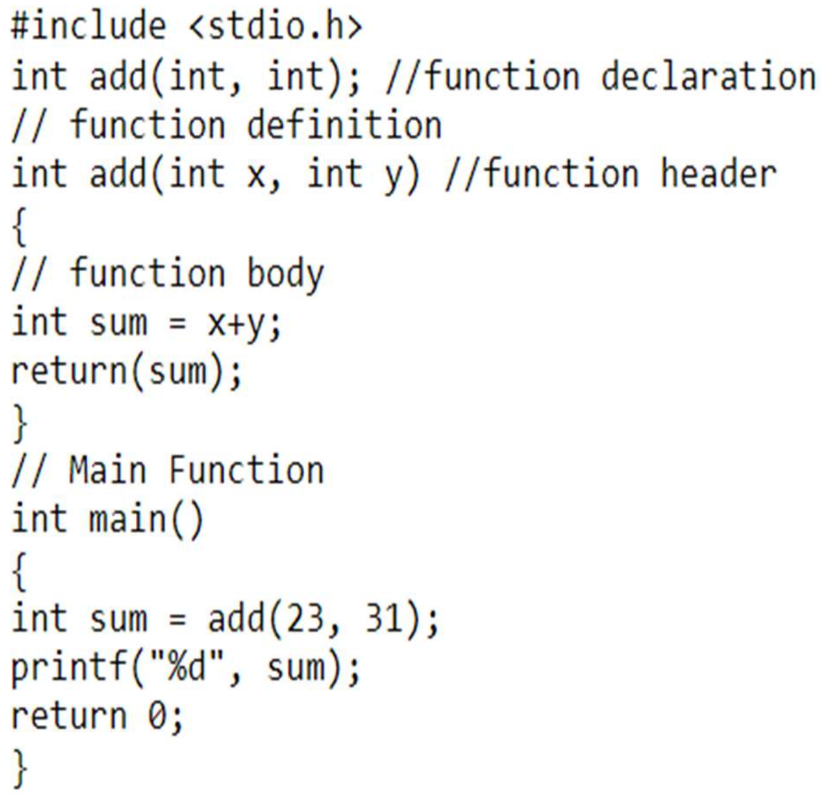
```
void swap(int x, int y)

// call by value
Swap(x,y);
```

```
void swap( int *x, int *y)

//call by reference
Swap( &x, &y);
```

Function Parameters…



Point	Call by Value	Call by Reference
Copy	Duplicate Copy of Original Parameter is Passed	Actual Copy of Original Parameter is Passed
Modification	No effect on Original Parameter after modifying parameter in function	Original Parameter gets affected if value of parameter changed inside function



```
#include <stdio.h>
int add(int, int); //function declaration
// function definition
int add(int x, int y) //function header
{
// function body
int sum = x+y;
return(sum);
}
// Main Function
int main()
{
int sum = add(23, 31);
printf("%d", sum);
return 0;
}
```

- As we can see in the above example that we are calling the function using int sum = add(23, 31); statement. The returned value from the function is stored in sum variable.
- **Actual Parameter**: Those parameters which are passed to functions while calling them is are known as actual parameter. For example, 23 & 31 in the above example are the actual parameters.
- **Formal Parameter**: Those parameters which are received by the functions are known as formal parameters. For example, x & y in the above example are the formal parameters.

Calling a Function

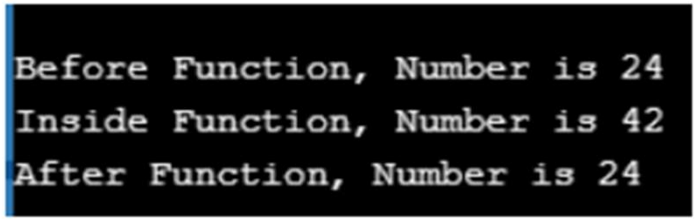
There are two ways in which we can call a function:

- Call by value
- Call by reference

Call by value

- In call by value method, the value of the actual parameter is passed as an argument to the function. The value of the actual parameter cannot be changed by the formal parameters.
- In call be value method, different memory address is allocated to formal & actual parameters. Just the value of actual parameter is copied to formal parameter.

```
#include <stdio.h>
void Call_By_Value(int num1)
{
num1=42;
printf("nInside Function, Number is %d", num1);
}
int main()
{
int num;
num=24;
printf("nBefore Function, Number is %d", num);
Call_By_Value(num);
printf("nAfter Function, Number is %dn", num);
return 0;
}
```

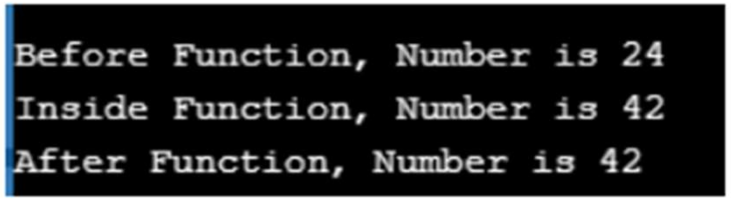


```
Before Function, Number is 24
Inside Function, Number is 42
After Function, Number is 24
```

Call by reference

- In call by reference, the memory address of the actual parameter is passed to the function as argument. Here, the value of the actual parameter can be changed by the formal parameter.
- Same memory address is used for both the actual & formal parameter. So, if the value of formal parameter is modified, it is reflected by the actual parameter as well.
- The address operator & is used to get the address of a variable of any data type.
- So in the function call statement ‘Call_By_Reference(&num);’, the address of num is passed so that num can be modified using its address.

```
#include <stdio.h>
// function definition
void Call_By_Reference(int *num1)
{
*num1=42;
printf("nInside Function, Number is %d", *num1);
}
// Main Function
int main()
{
int num;
num=24;
printf("nBefore Function, Number is %d", num);
Call_By_Reference(&num);
printf("nAfter Function, Number is %dn", num);
return 0;
}
```



```
Before Function, Number is 24
Inside Function, Number is 42
After Function, Number is 42
```

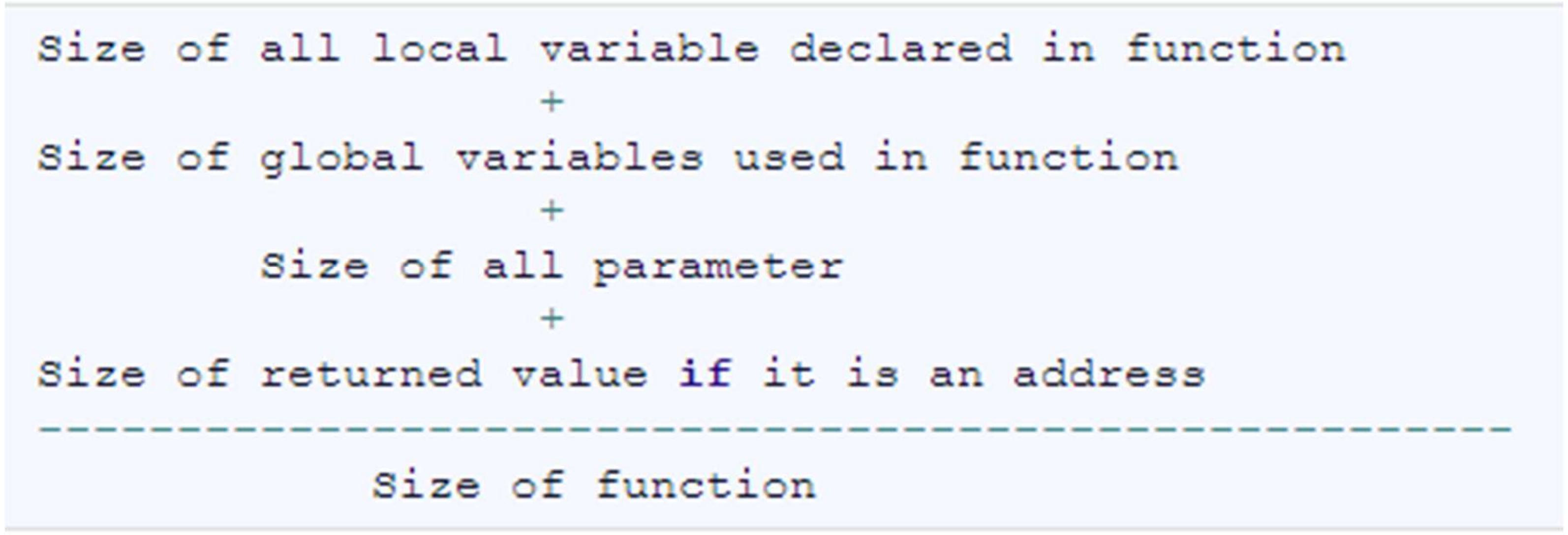
Function Parameters - Rules

- The number of arguments in a function call must be the same as the number of parameters in the function definition. This number can be zero.
- The max no. of arguments is 253 for a single function. (Machine dependent)
- If a function is called with arguments of the wrong type, **the presence of a prototype** means that the actual argument is converted to the type of the formal argument ‘as if by assignment’. If prototype is not present then it may give garbage values or undefined behaviour.
- In the old style, parameters that are not explicitly declared are assigned a default type of int.
- The scope of function parameters is the function itself. Therefore, parameters of the same name in different functions are unrelated.

Functions - Rules

- **Function Declaration:** A function **declaration** tells the compiler about a function name and how to call the function.
- C program can have one or more functions.
- Any function can be called from any function
- A function can be called any number of times.
- A function can call itself.
- A function cannot be defined in another function.

Size of a function



```
Size of all local variable declared in function
                    +
Size of global variables used in function
                    +
        Size of all parameter
                    +
Size of returned value if it is an address
-----------------------------------------------------
            Size of function
```

Functions with no arguments No return value

```
#include<stdio.h>
 void area(); // Prototype Declaration
void main()
 {
          area();
}
void area()
 {
          float area_circle;
          float rad;
          printf("\nEnter the radius of circle : ");
          scanf("%f",&rad);
          area_circle = 3.14 * rad * rad ;
          printf("Area of Circle = %f",area_circle);
 }
```

Functions with arguments No return value

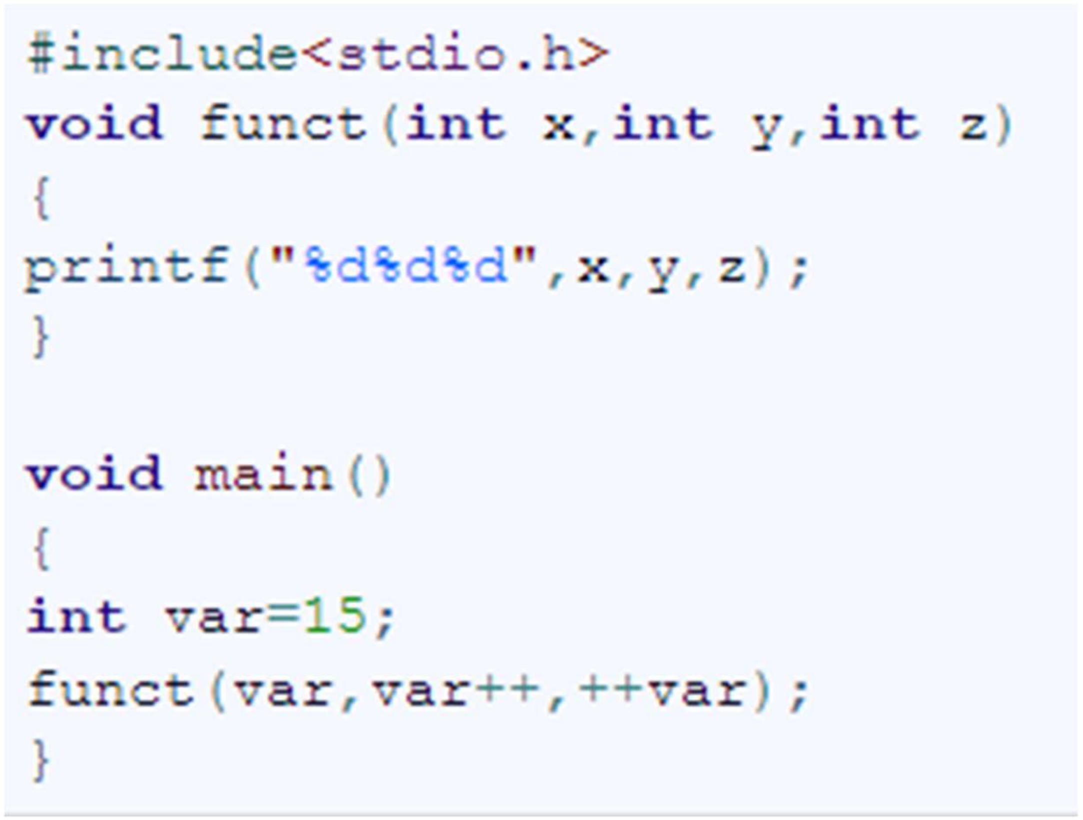
```
#include<stdio.h>
 void area(float rad);

void main()
 {
     float rad;
    printf("nEnter the radius : ");
    scanf("%f",&rad);
    area(rad);
}
 void area(float rad)
 {
    float ar;
    ar = 3.14 * rad * rad ;
    printf("Area of Circle = %f",ar);
}
```

Functions with arguments with return value

```
#include<stdio.h>
float calculate_area(int);
int main()
 {
            int radius;
            float area;
            printf("\nEnter the radius of the circle : ");
             scanf("%d",&radius);
            area = calculate_area(radius);
            printf("\nArea of Circle : %f ",area);
            return(0);
 }
 float calculate_area(int radius) {
            float areaOfCircle;
            areaOfCircle = 3.14 * radius * radius;
            return(areaOfCircle);
 }
```

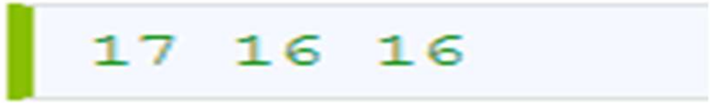
Functions – Parameter Parsing

- In a function **parameters** are passed/parsed in **Right to Left** direction.

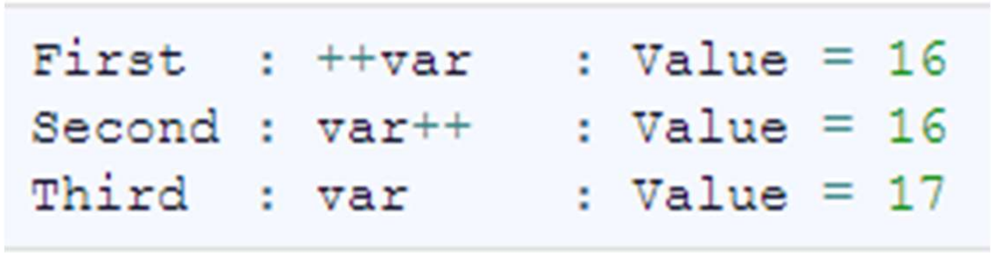


```c
#include<stdio.h>
void funct(int x,int y,int z)
{
printf("%d%d%d",x,y,z);
}

void main()
{
int var=15;
funct(var,var++,++var);
}
```

Output :



```
17 16 16
```



```
First  : ++var    : Value = 16
Second : var++    : Value = 16
Third  : var      : Value = 17
```

Functions - Advantages

Advantages:

- Modularity
- Readability
- Reusability
- Easy to develop, debug and test
- Allows test-driven development
- Allows unit testing
- Allows top-down modular approach

What is the Value of the a and b ?

```
void swapping(int c, int d)
 {
            Int tmp;
            tmp = c;
            c = d;
            d = tmp;
            printf("In function: %d %d\n", c , d);
}
Void main( )
{            int a,b;
             a=5; b=10;
             printf("input: %d %d\n", a, b);
            swapping(a,b);
            printf("output: %d %d\n", a, b);
}
```

What is the Value of the a and b ?

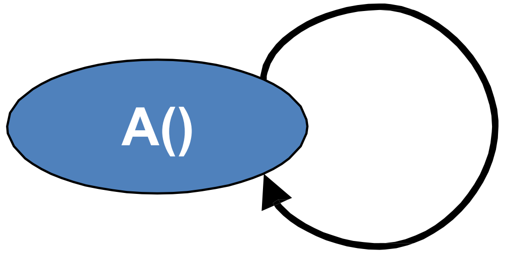
```
void swapping(int *ptr_c, int *ptr_d)
{            int tmp;
            tmp = *ptr_c;
            *ptr_c = *ptr_d;
             *ptr_d = tmp;
            printf("In function: %d %d\n", *ptr_c , *ptr_d);
                }
Void        main( )
{
            int a,b;
            a=5;
            b=10;
            printf("input: %d %d\n", a, b);
            swapping(&a,&b);
            printf("output: %d %d\n", a, b);
}
```

Recursion

- **Definition:** Recursion is a process by which **a function calls itself** repeatedly, until some specified condition has been satisfied.
- To solve a problem recursively
  - The problem must be written in a recursive form.
  - The problem statement must include a stopping condition.
- A recursive function must have the following type of statements :
  - A statement to test and determine whether the function is calling itself again.
  - A statement that calls the function itself and must be argument.
  - A conditional statement (if-else).
  - A return statement.

Recursion

Syntax:



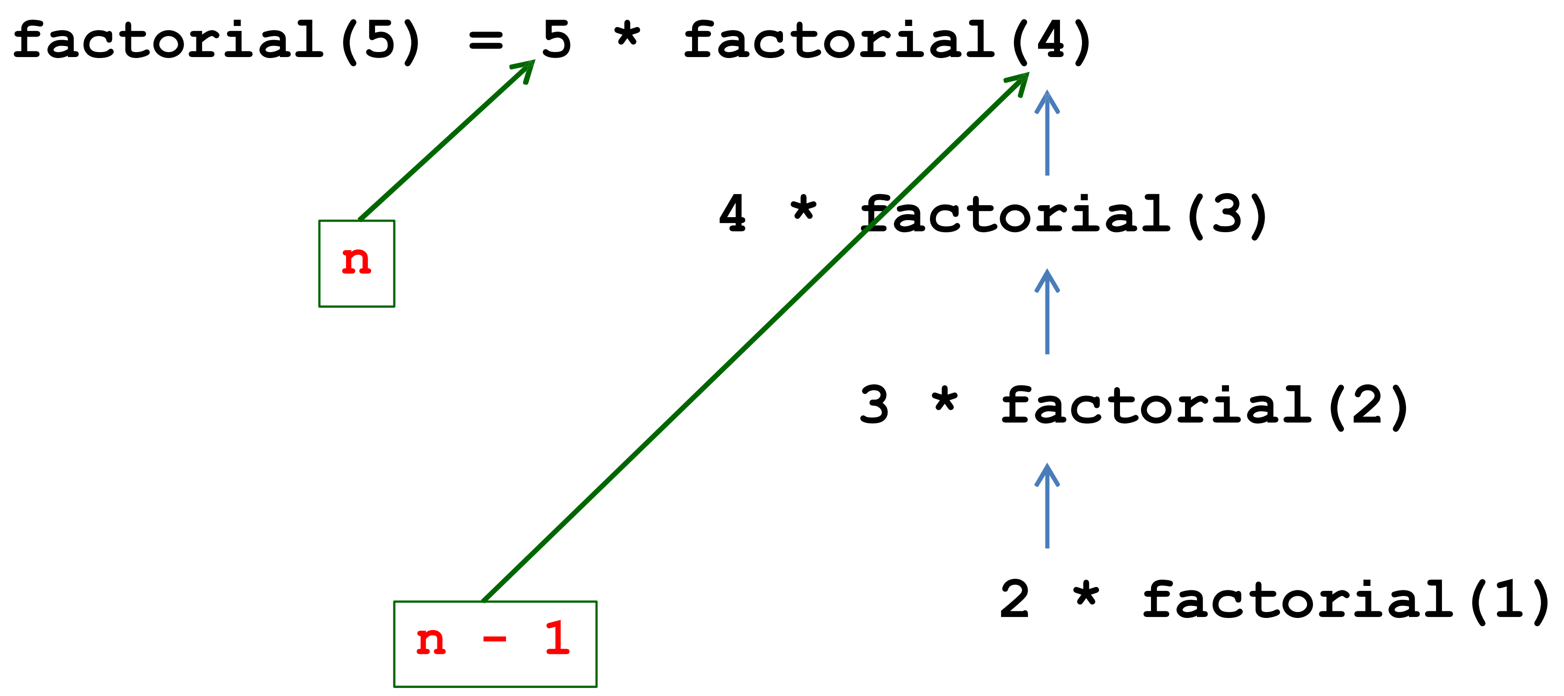
```
void main()
{
    A();
    return;
}
```

```
Void A()
{
      if (stopping condition)
          return;
      else
           A();
      return;
}
```

Recursion – Example1 - Factorial

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

Can we compute factorial using recursion?



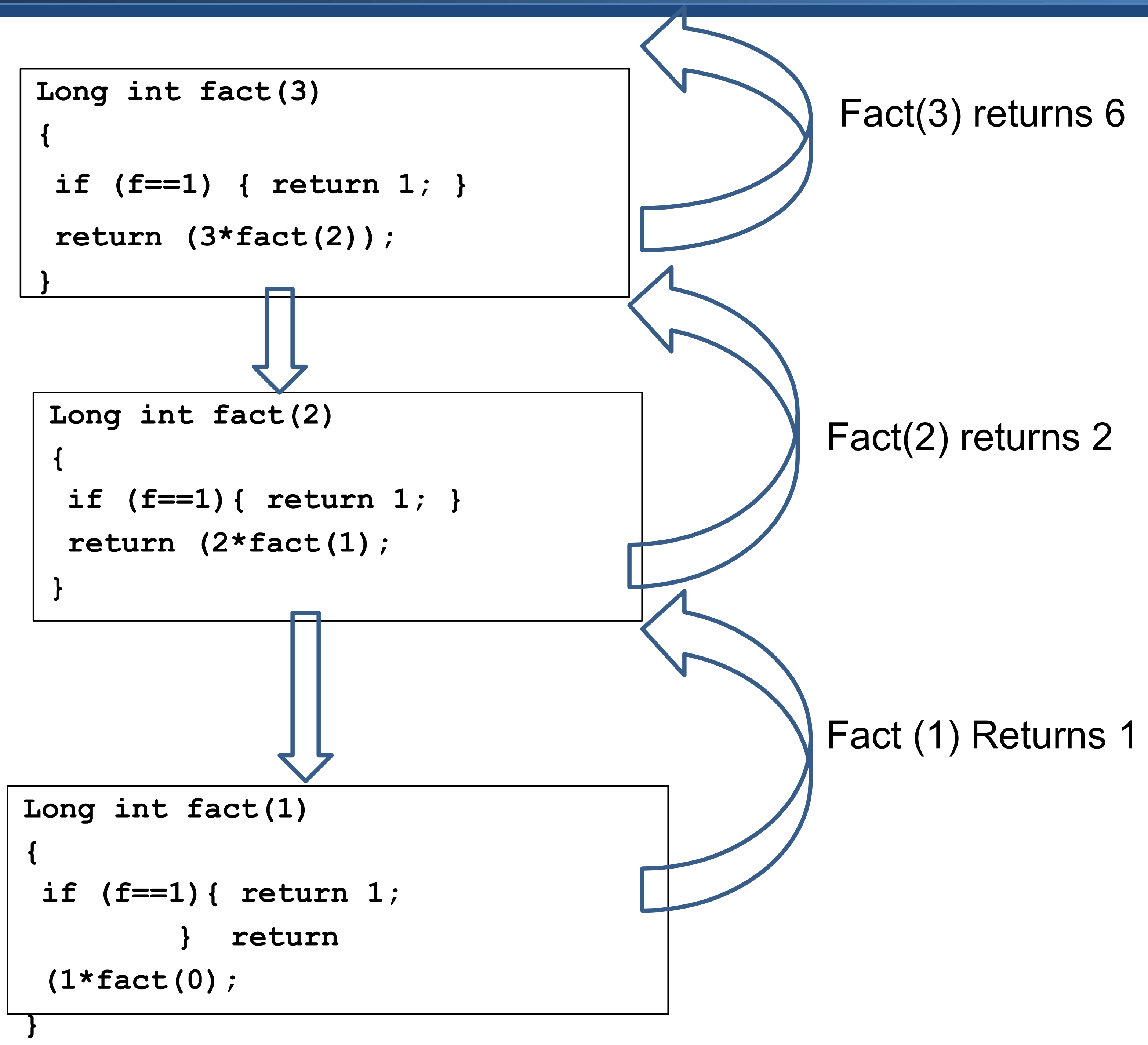
Recursion – Example1 - Factorial

Example: Recursive function to find factorial

```
int main()
{
 long int fact;
 printf(“Enter n value:”);
 scanf(“%d”, &n);
 fact = fact(n);
 printf("Factorial=%ld", fact);
 return 0;
}
```

```
long int fact(int f)
{
 if (f==1 || f==0)
     return 1;
 return (f*fact(f-1);
}
```

Working of Recursion

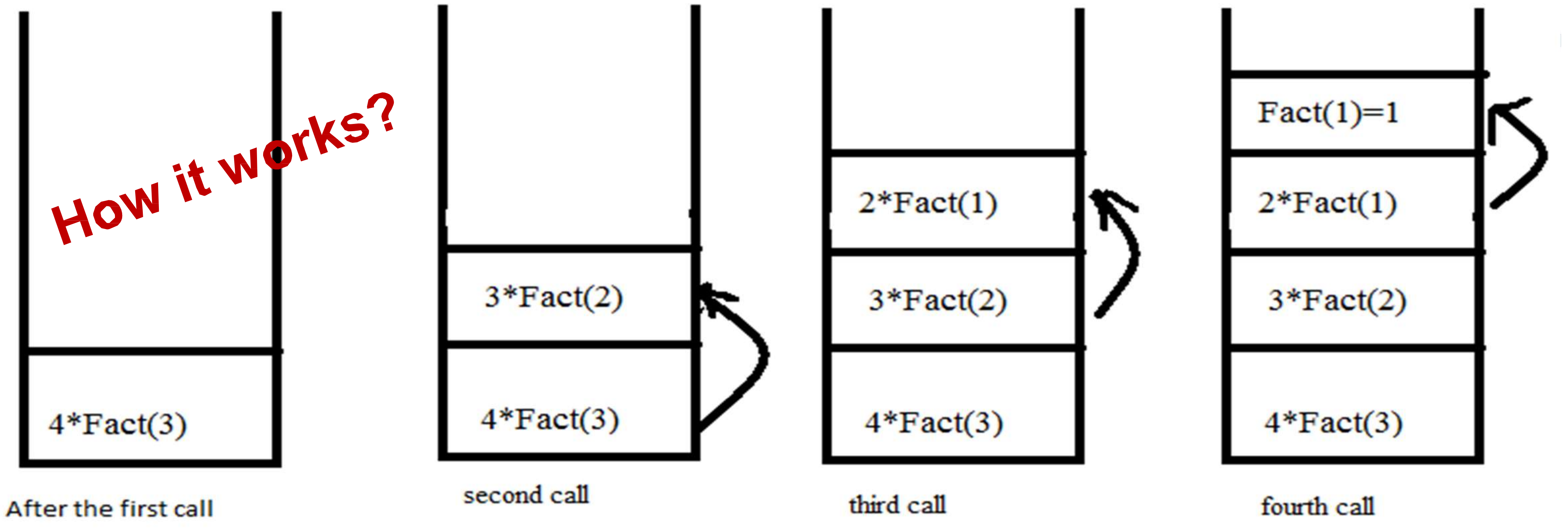


```
Long int fact(3)
{
 if (f==1) { return 1; }
 return (3*fact(2));
}
```

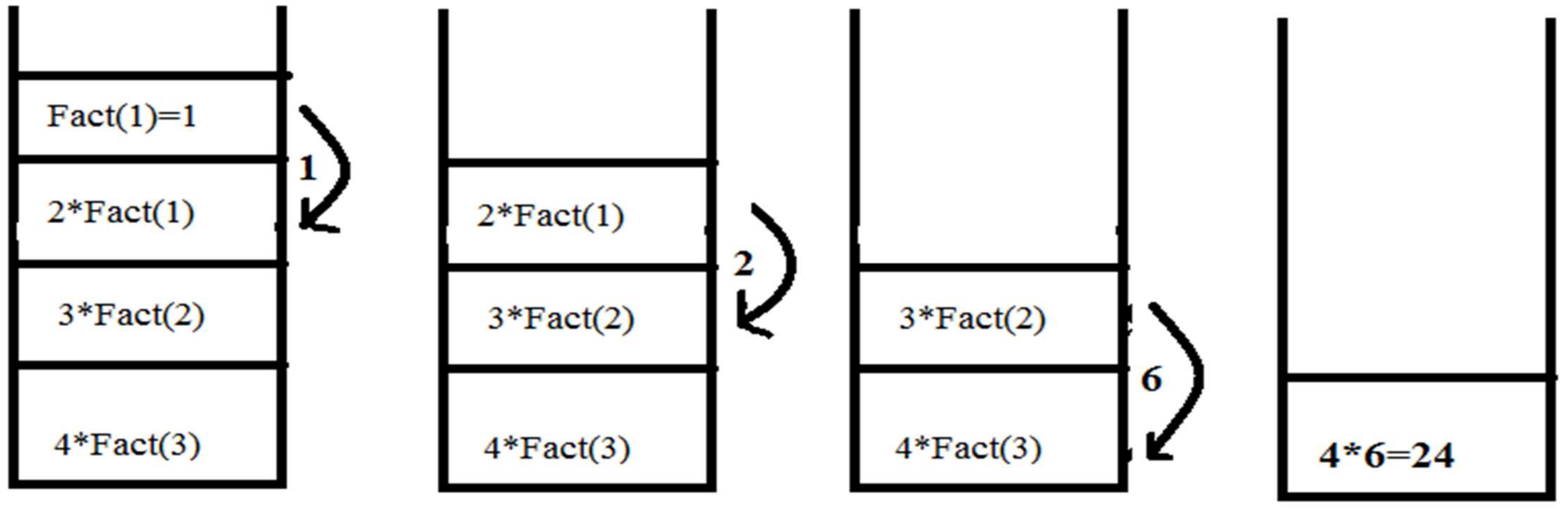
```
Long int fact(2)
{
 if (f==1){ return 1; }
 return (2*fact(1);
}
```

```
Long int fact(1)
{
 if (f==1){ return 1;
         }  return
 (1*fact(0);
}
```

When function call happens previous variables gets stored in stack



Returning values from base case to caller function



Recursion – Example2 - Fibonacci

$$f(n) = \begin{cases} n = 0 & 0 \\ n = 1 & 1 \\ n > 1 & f(n-1) + f(n-2) \end{cases}$$

Can we compute Fibonacci series using recursion?

```
1+1=2
   1+2=3
      2+3=5
         3+5=8
            5+8=13
               8+13=21
                  13+21=34
                     21+34=55
                                • • •
```

Do it Now: Write a program to find ‘n’ Fibonacci numbers using recursion

Recursion – Example2 - Fibonacci

```
#include<stdio.h>
int  main()
{
  int i;
  for (i = 0; i < 10; i++)
  printf("%d\t", fibonacci(i));
  return 0;
}
```

```
int fibonacci(int i)
{
   if(i == 0) return 0;
   if(i == 1) return 1;
   return (fibonacci(i-1) + fibonacci(i-2));
}
```

Recursion

Advantages:

- Simple to code (Not Always)
- Size of the code will be less
- Readable

- **Disadvantages:**
  - Difficult   to understand in some algorithms
  - Stack Overflow in case of deep recursion
  - Less portable
  - Writing parallel recursive functions is error prone

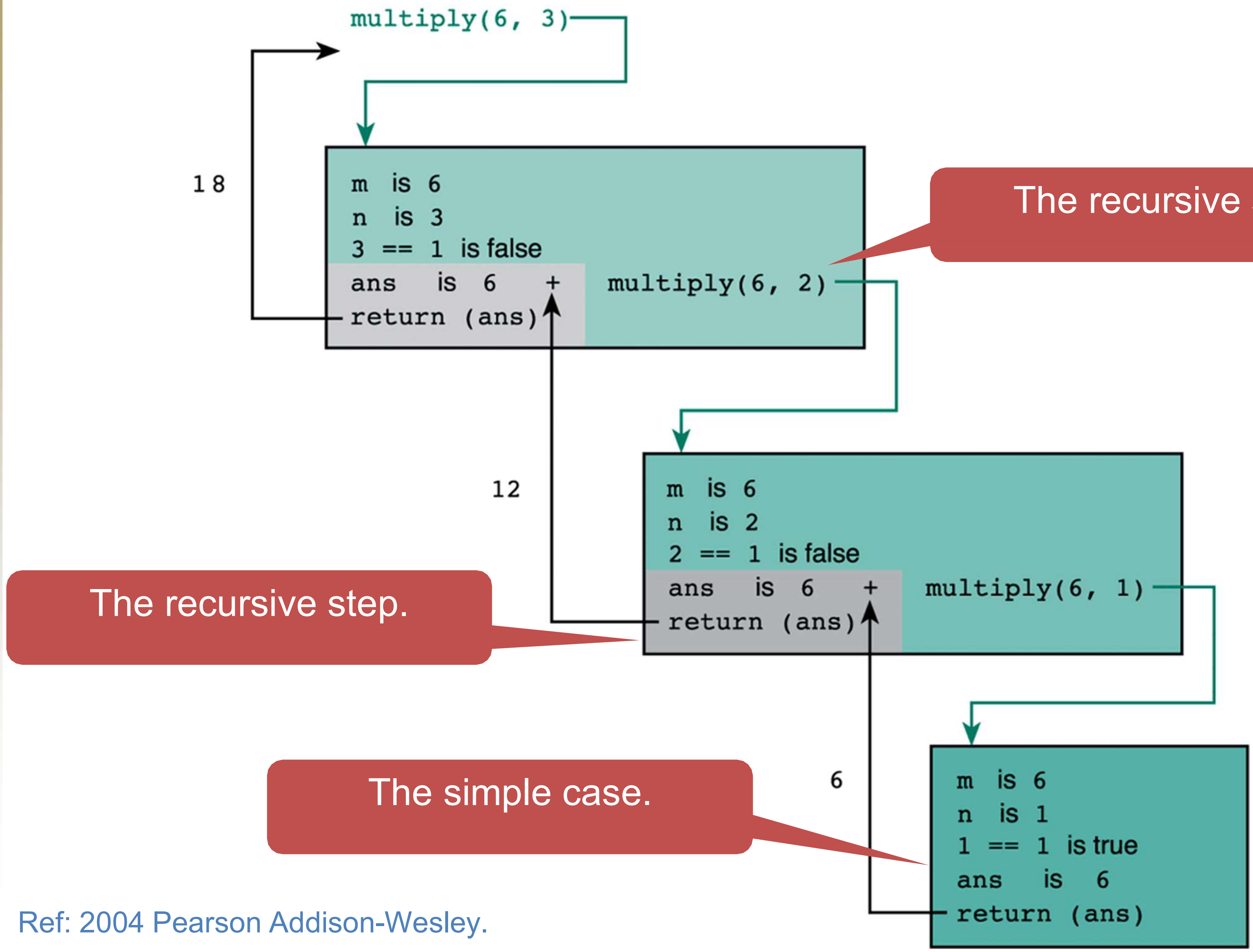
Class room Exercise - 6

1. Write a program to implement multiplication using addition. Use recursion.
2. Write a program to swap two numbers.
3. Write a recursive function to find the sum of n integers.

Class room Exercise - 6

```
/*
 *  Performs integer multiplication using + operator.
 *  Pre:   m and n are defined and n > 0
 *  Post:  returns m * n
 */
int
multiply(int m, int n)
{
      int ans;

      if (n == 1)
            ans = m;      /* simple case */
      else
            ans = m + multiply(m, n - 1);  /* recursive step */

      return (ans);
}
```

Trace of Function multiply(6,3)



Ref: 2004 Pearson Addison-Wesley.

THANK YOU

YOU