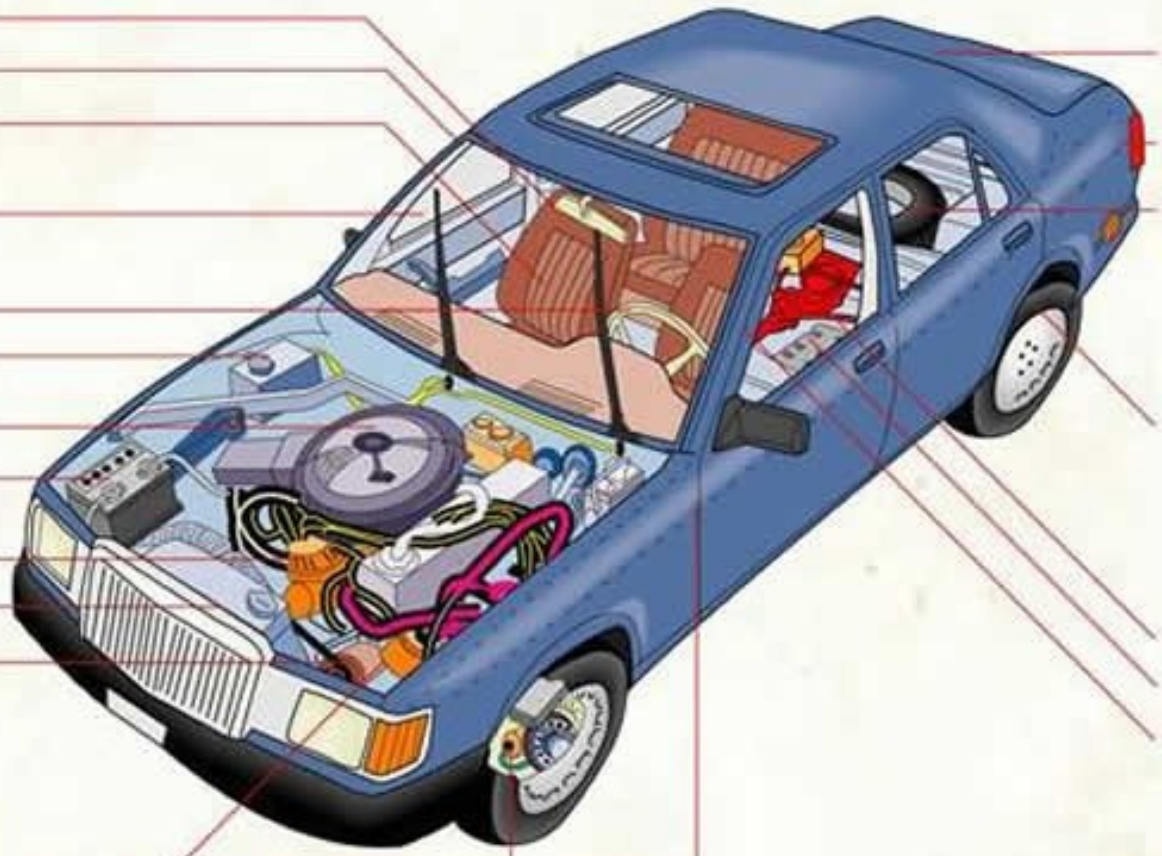
Functions in C

ANATOMY OF AN AUTOMOBILE



rearview mirror
window frame
seat
windshield wiper
sterring wheel
windshield washer
air filter
battery
distributor
radiator
alternator
trunk
tail light
spare wheel
wheel
transmission
muffler
line shaft
oil filter
disk brake
body side moulding

Modularity in Car

- Each component in car is independently manufactured and tested
- Can be fitted to any car

Functions in C

- Modularity is supported in C by functions
- All variables declared inside functions are local variables
  - Known only in function defined
- Parameters
  - Communicate information between functions
  - Local variables

Syntax for C

- ftype fname (void);

  void draw_circle(void);

- Declarations and statements: function body (block)
- Functions can not be defined inside other functions

Syntax

function header

{

statements

}

Syntax of function header

return_type function_Name(parameter_List)

Passing Arguments

- Call by value
  - Copy of argument passed to function
  - Changes in function do not effect original
  - Use when function does not need to modify argument
    - Avoids accidental changes
- Call by reference
  - Passes original argument
  - Changes in function effect original
  - Only used with trusted functions

Return Statement

return;

return expression;

```
return;
return (a + b);
```

```
int maximum(int value1, int value2)
{
   return (value1 > value2 ? value1 : value2);
}
```

Unlike Python C can return only value

Example

```c
#include <stdio.h>

/* function prototype declaration */
int max(int num1, int num2);

int main () {

   /* local variable definition */
   int a = 100;
   int b = 200;
   int r;

   /* calling a function to get max value */
   r = max(a, b);

   printf( "Max value is : %d\n", r );

   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;
}
```

Output : 200

Local Variables Vs Global Variables

- Local Variables - Variables that are declared inside a function or block are called local variables

```c
#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 - A global variable can be accessed by any function.

```c
#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;
}
```

Formal Parameters Vs Actual Parameters

- Formal Parameters -The Formal Parameters are the variables defined by the function that receives values when the function is called.
- The formal parameters are in the called function.
- In formal parameters, the data types of the receiving values should be included.

- Actual Parameters - The Actual parameters are the values that are passed to the function when it is invoked.
- The actual parameters are passed by the calling function.
- In actual parameters, only the value is mentioned.

`

Example

```c
# include <stdio.h>
int calsum ( int x, int y, int z ) ;
int main( )
{
int a, b, c, sum ;
printf ( "Enter any three numbers " ) ;
scanf ( "%d %d %d", &a, &b, &c ) ;
sum = calsum ( a, b, c ) ;
printf ( "Sum = %d\n", sum ) ;
return 0 ;
}
int calsum ( int x, int y, int z )
{
int d ;
d = x + y + z ;
return ( d ) ;
}
```

Types of Function calls

Arguments can generally be passed to functions in one of the two ways:

(a) sending the values of the arguments -

- Copy of the parameter is passed
- the formal arguments in the called function have no effect on the values of actual arguments in the calling function.

(b) sending the addresses of the arguments -

- the addresses of actual arguments in the calling function are copied into the formal arguments of the called function.
- using these addresses, we would have an access to the actual arguments and hence we would be able to manipulate them.

Call by Value vs Call by Reference

- The process of calling function by actually sending or passing the copies of data.
- At most one value at a time can be returned to the calling function with an explicit return statement.
- Here formal parameters are normal variable names that can receive actual parameters/argument value's copy.

- The process of calling function using pointers to pass the address of variables .
- Multiple values can be returned to calling function and explicit return statement is not required .
- Here formal parameters are pointer variables that can receive actual parameter or arguments as address of variables .

Call by Value

```c
#include <stdio.h>

void swapv ( int x, int y ) ;
int main( )
{
int a = 10, b = 20 ;
swapv ( a, b ) ;
printf ( "a = %d b = %d\n", a, b ) ;
return 0 ;
}
void swapv ( int x, int y )
{
int t ;
t = x ;
x = y ;
y = t ;
printf ( "x = %d y = %d\n", x, y ) ;
}
```

Output:
x = 20 y = 10
a = 10 b = 20

Note: values of a and b remain unchanged even after exchanging the values of x and y.

Pass by Reference

- Address of variable is passed
- Pointer variables are used in function declaration and definition
- Address of variable is got by prefixing variable name with an '&'
- Pointer variables are declared with a '*' in front
- An integer pointer variable is declared as

  int* a;
- Value of an address is got by using '*' operator

Call by reference

```c
#include <stdio.h>

void swapr ( int *, int * ) ;
int main( )
{
int a = 10, b = 20 ;
swapr ( &a, &b ) ;
printf ( "a = %d b = %d\n", a, b ) ;
return 0 ;
}

void swapr ( int *x, int *y )
{
int t ;
t = *x ;
*x = *y ;
*y = t ;
}
```

Output:
a = 20 b = 10

Note: program manages to exchange the values of a and b using their addresses stored in x and y

Returning more than one values

```c
#include <stdio.h>

void areaperi ( int, float *, float * ) ;
int main( )
{
int radius ;
float area, perimeter ;
printf ( "Enter radius of a circle " ) ;
scanf ( "%d", &radius ) ;
areaperi ( radius, &area, &perimeter ) ;
printf ( "Area = %f\n", area ) ;
printf ( "Perimeter = %f\n", perimeter ) ;
return 0 ;
}

void areaperi ( int r, float *a, float *p )
{
*a = 3.14 * r * r ;
*p = 2 * 3.14 * r ;
}
```

OUTPUT:

Enter radius of a circle 5
Area = 78.500000
Perimeter = 31.400000

```c
#include<stdio.h>
void swap(int* a, int* b)
{
int t;
t = *a;
*a=*b;
*b = t;
}
void main()
{
int a,b;
scanf("%d%d",&a,&b);
swap(&a,&b);
printf("%d\t%d",a,b);
}
```

Default Passing Mechanisms

- Primitive data types such as int, float, long, double, char are passed by value – so changes do not reflect in calling function
- Arrays are passed by address – so changes reflect in calling function

Pass a Single Dimensional Array as Argument

- Passed by reference or address by default
- Changes made in function gets reflected in main()
- Three ways to pass arrays in C all are same

Way1	Way2	Way3
void myFunction(int *a) { . . . }	void myFunction(int a[10]) { . . . }	void myFunction(int a[]) { . . . }

Passing Arrays to Functions

```c
# include <stdio.h>

void display ( int *, int ) ;
int main( )
{
    int num[ ] = { 24, 34, 12, 44, 56, 17 } ;
    dislpay ( &num[ 0 ], 6 ) ;
    return 0 ;
}

void display ( int *j, int n )
{
    int i ;
    for ( i = 0 ; i <= n - 1 ; i++ )
    {
        printf ( "element = %d\n", *j ) ;
        j++ ; /* increment pointer to point to next element */
    }
}
```

- The following two function calls are same:

display (&num[0], 6) ;
display (num, 6) ;

Passing 2D Arrays

```c
#include <stdio.h>
const int n = 3;

void print(int arr[3][3], int m)
{
    int i, j;
    for (i = 0; i < m; i++)
      for (j = 0; j < n; j++)
        printf("%d ", arr[i][j]);
}

int main()
{
    int arr[][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    print(arr, 3);
    return 0;
}
```

As a Single Dimensional Array

```c
#include <stdio.h>
void print(int *arr, int m, int n)
{
    int i, j;
    for (i = 0; i < m; i++)
      for (j = 0; j < n; j++)
        printf("%d ", *((arr+i*n) + j));
}

int main()
{
    int arr[][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    int m = 3, n = 3;

    print(&arr[0][0], m, n);
    //print((int *)arr, m, n);
    return 0;
}
```

Recursion

- Recursive functions

➢ function is called ‘recursive’ if a statement within the body of a function calls the same function.

➢ Called ‘circular definition’, recursion is thus the process of defining something in terms of itself.

Recursion

- Factorial - the factorial of a number is the product of all the integers between 1 and that number.
- For example, 4 factorial is 4 * 3 * 2 * 1. This can also be expressed as 4! = 4 * 3!
- Thus factorial of a number can be expressed in the form of itself.

Non-Recursive Factorial Program

```c
# include <stdio.h>
int factorial ( int ) ;
int main( )
{
    int a, fact ;
    printf ( "Enter any number " ) ;
    scanf ( "%d", &a ) ;
    fact = factorial ( a ) ;
    printf ( "Factorial value = %d\n", fact ) ;
    return 0 ;
}

int factorial ( int x )
{
    int f = 1, i ;
    for ( i = x ; i >= 1 ; i-- )
    f = f * i ;
    return ( f ) ;
}
```

OUTPUT:

Enter any number 3
Factorial value = 6

Factorial using Recursion

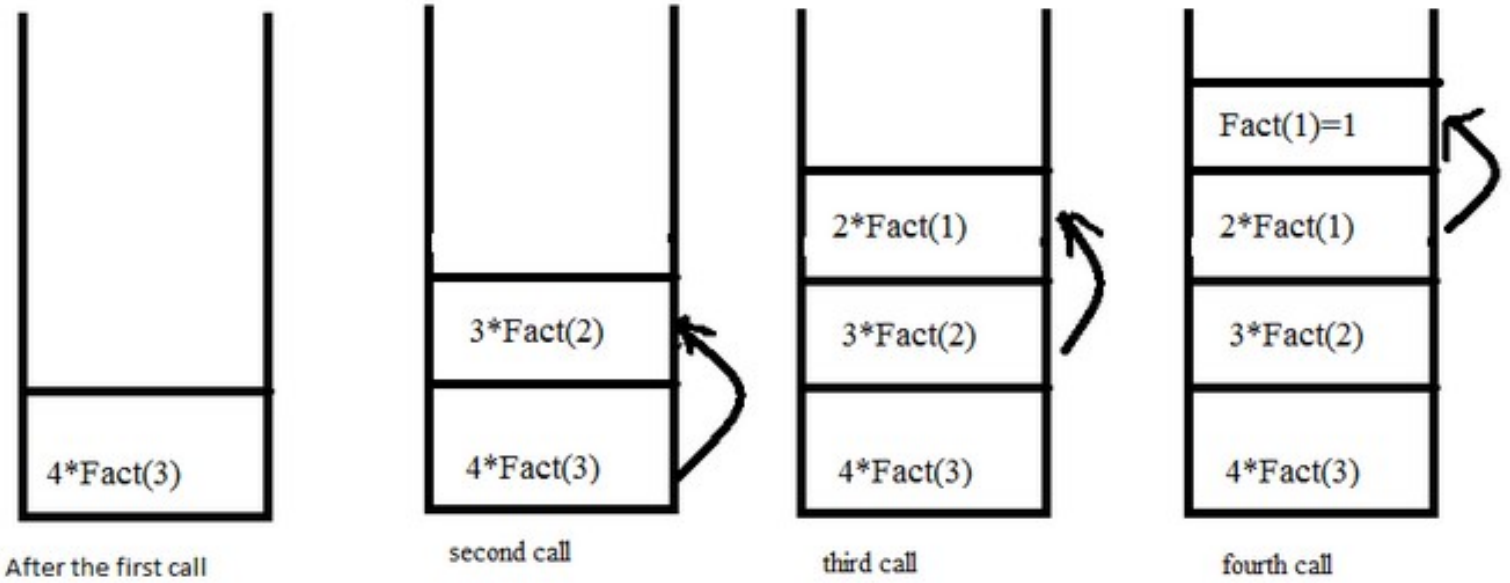
```c
#include<stdio.h>
int fact(int);
void main()
{
 int num;
 scanf("%d",&num);
 printf("%d",fact(num));
}

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

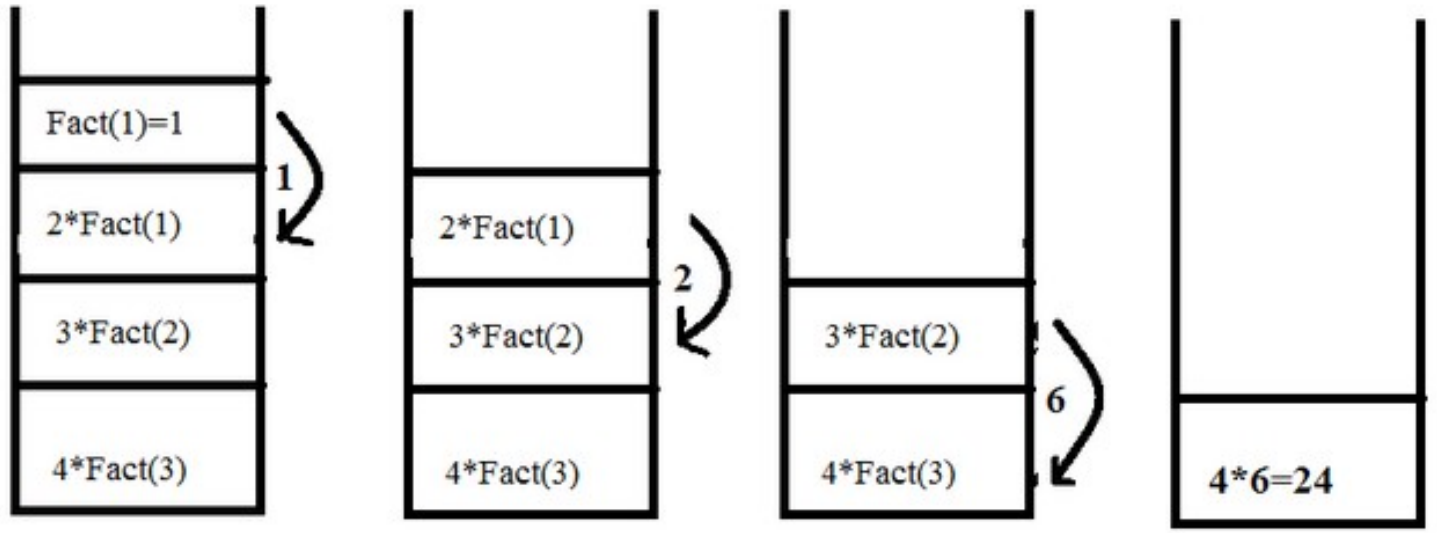
Base Case

Recursive Call

When function call happens previous variables gets stored in stack



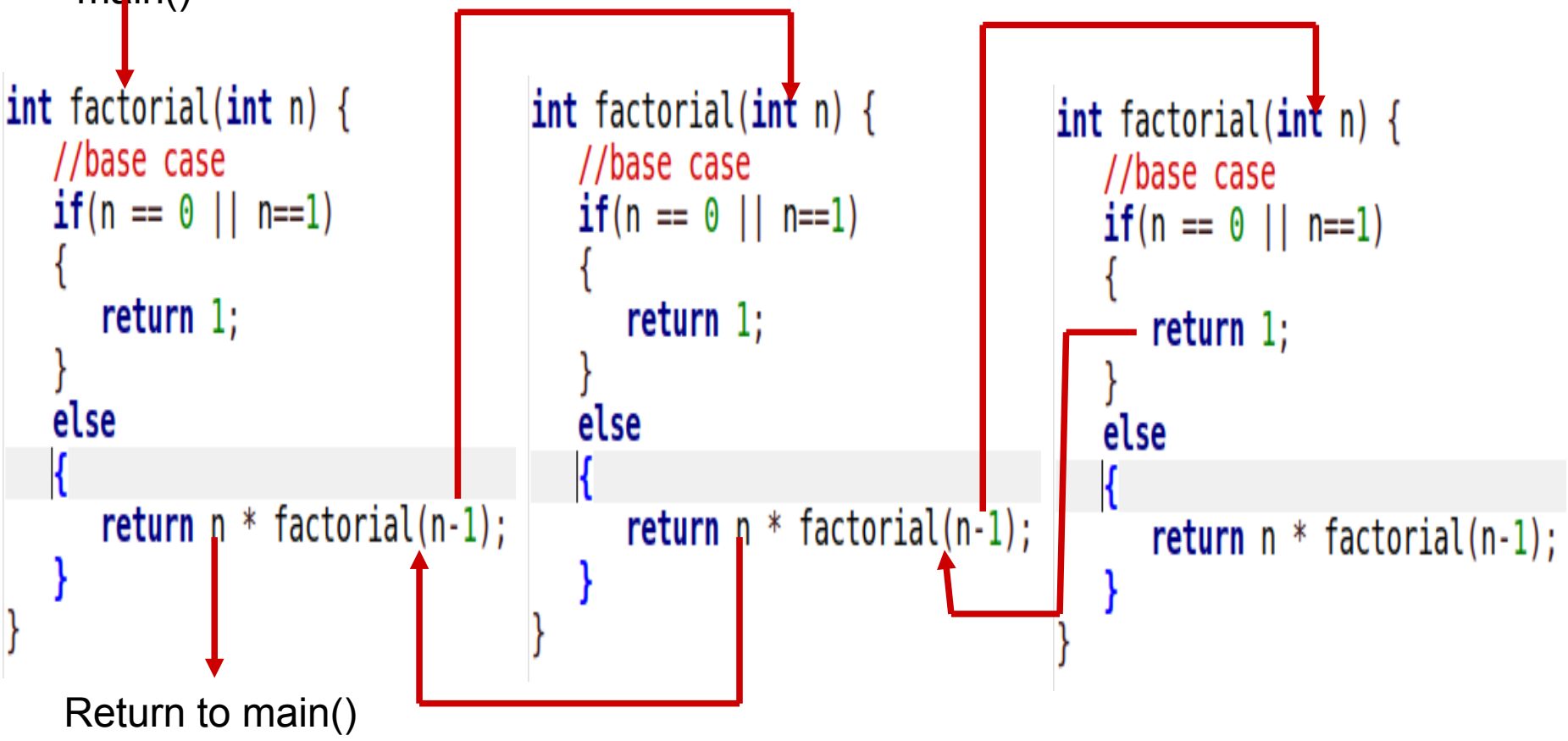
Returning values from base case to caller function



Trace of Factorial Code

factorial (3) which returns (3 times factorial (2),
which returns (2 times factorial (1),
which returns (1)))

main()



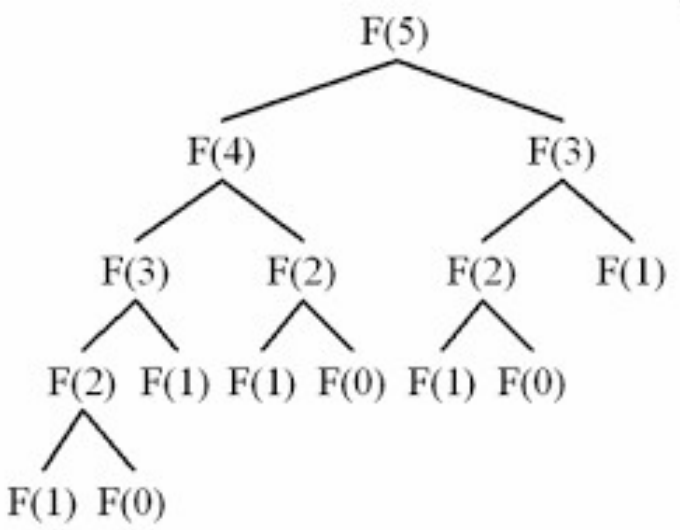
Return to main()

Fibonacci Series

0, 1, 1, 2, 3, 5, 8...

```
int fibonacci( int n )
{
   if (n == 0 || n == 1)  // base case
     return n;
   else
     return fibonacci( n - 1) + fibonacci( n – 2 );
}
```

$$
\begin{aligned}
F(5) &= F(4) + F(3) \\
&= F(3) + F(2) + F(2) + F(1) \\
&= F(2) + F(1) + F(1) + F(0) + F(1) + F(0) + 1 \\
&= F(1) + F(0) + 1 + 1 + 0 + 1 + 0 + 1 \\
&= 1 + 0 + 1 + 1 + 0 + 1 + 0 + 1 \\
&= 5
\end{aligned}
$$



String Reverse

```c
char* reverse(char* str);
void main()
{
    int i, j, k;
    char str[100];
    char *rev;
    printf("Enter the string:\t");
    scanf("%s", str);
    printf("Original string  %s\n", str);
    rev = reverse(str);
    printf("Reversed string %s\n", rev);

}

char* reverse(char *str)
{
    int i = 0;
    char rev[100];
    if(*str)
    {
        reverse(str+1);
        rev[i++] = *str;
    }
    return rev;
}
```

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)