Date

Recursive Function

The function which calls itself is called Recursive function.
Recursion is the process through which a function calls itself within its body until some specified condition is satisfied.
A terminating condition must be defined properly within recursive function.

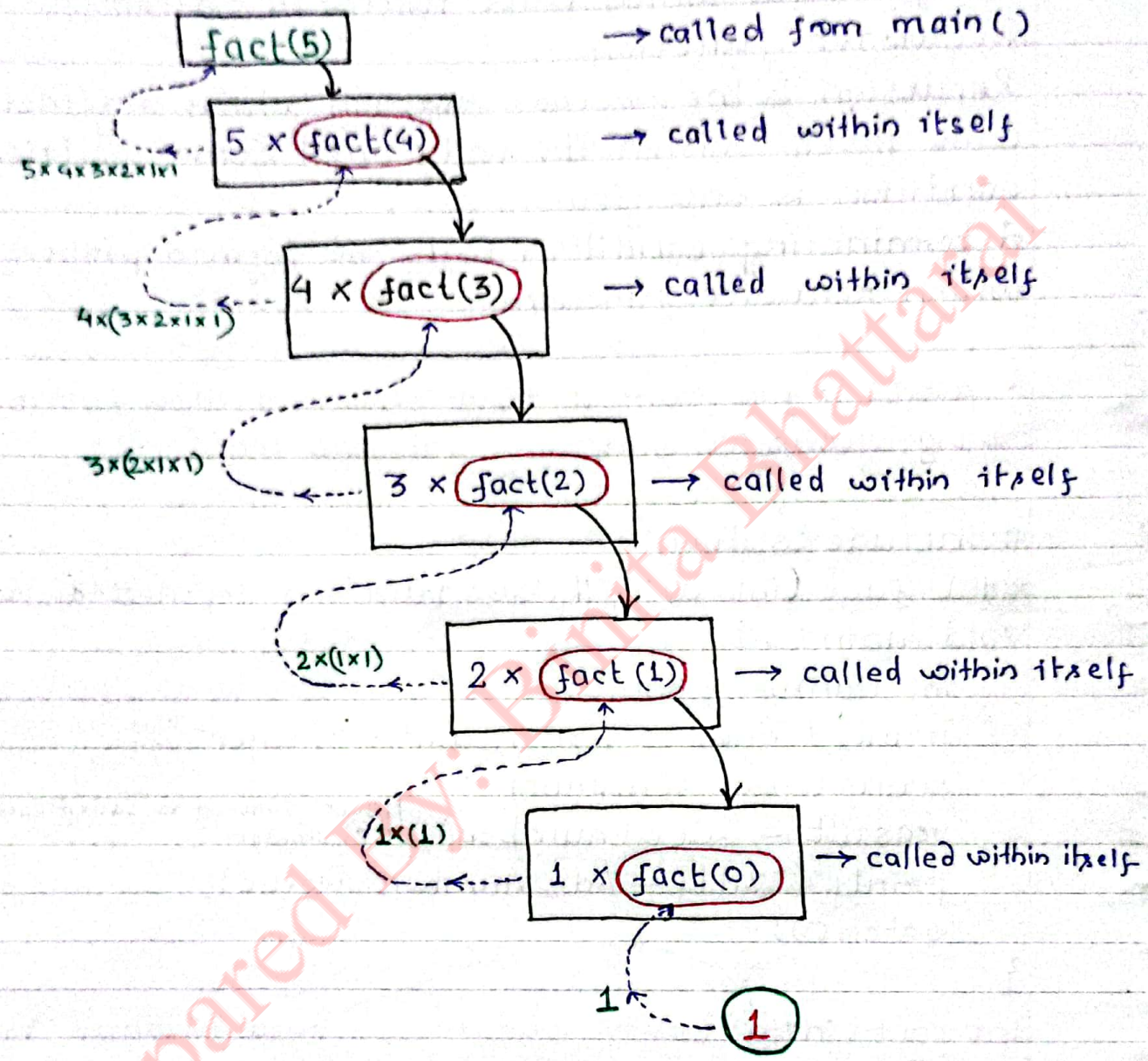
* Write a program to find factorial of a number using recursive function and not loops.

```
#include<stdio.h>
int fact(int a); // User function defined (declared)
void main(){
    int number, result;
    printf("Enter a non-negative integer:");
    scanf("%d", &number);
    result = fact(number); // 'fact' function is called with parameter.
    printf("%d! = %d", number, result);
    getch();
}
int fact(int a){
    int x=1; // to store the result
    if(a<=0){
        return 1;
    }
    else{
        x = a * fact(a-1);
        return x;
    }
}
```

'a' becomes 'a-1' i.e. if initial value is 5, a is 5 initially. When funct^n is called with (a-1), then, a becomes 4. Again, when funct^n is called with a-1, 'a' becomes 3 & so on.
When 'a' becomes 0, then it return '1' & program moves out of the function.

Date

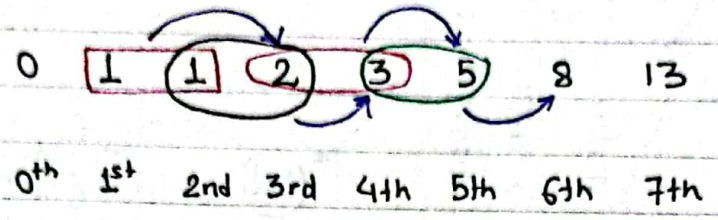
Understanding the program:



Date

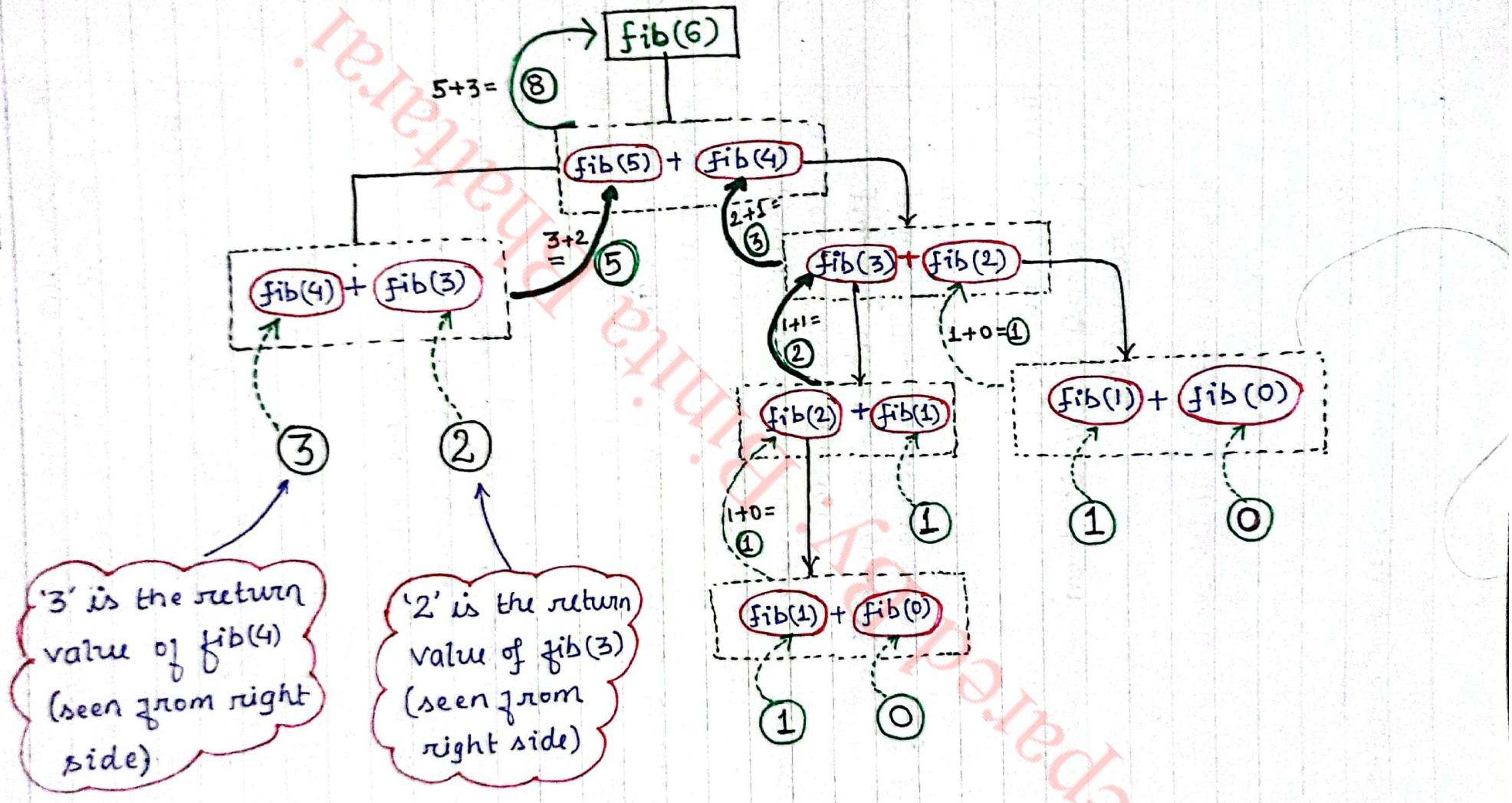
* Program to find particular (n^{th}) term of Fibonacci series using recursive function.

Logic:



Any term, for e.g. 6th term, is obtained by adding previous two terms, i.e.; 5^{th} & 4^{th}.
∴ If we want to find n^{th} term, we need to add $(n-1)^{th}$ & $(n-2)^{th}$ term.

```
#include <stdio.h>
int fibo(int); // user function declaration
void main() {
    int num, fib_result;
    printf("Enter which term to generate");
    scanf("%d", &num);
    fib_result = fibo(num);
    printf("The particular term is %d", fib_result);
    getch();
}
int fibo(int n) {
    if(n==0){   // 0th term
        return(0);
    }
    else if(n==1){ // 1st term
        return(1);
    }
    else{
        return(fibo(n-1) + fibo(n-2));
}
```



Date

Storage Class:

हामिले variable declare गर्दा त्यसको अगाडि datatype लगाउने गर्दौं (जस्तै int, float, char...) जसले के मात्र भन्छ भन्दा, त्यो variable मा कस्तो किसिमको data राख्न मिल्छ। तर एउटा variable को बारेमा datatype बाहेक अरू पनि information हुन्छन् जस्तै :

* त्यो variable को data बस्ने location
* declare मात्र गरेर छोड्दा (initialize नगर्दा) bydefault कस्तो/कति हुन्छ
* एउटा complete program मा main() function बाहिर को ठाउँ हुन्छ, अनि अरू सबै function हरू पनि हुन सक्छ :

```
#include<stdio.h>          } main() बाहिर तर कुनै funct" भित्र होइन
void main(){               }
   ....                    }
   ....                    } main() भित्रको ठाउँ।
}                          }
int add(){                 } main() बाहिर तर एउटा funct" भित्रको ठाउँ
   ....                    }
}                          }
int area(){                }
   ....                    }
}                          }
```

अब, variable (e.g. x) लाई कतै declare गरियो भने, त्यो variable लाई कता-कता वा भनौं कति सम्मको ठाउँमा प्रयोग गर्न सकिन्छ भन्ने कुरा आउँछ (i.e. called 'scope' of variable)। एउटा variable लाई हुन सक्छ कि केहि सिमित ठाउँ सम्म मात्र चीन्छ र त्यो ठाउँ भन्दा बाहिर चिन्दैन (चिन्दैन भने त्यहाँ प्रयोग गर्न मिलदैन)

Date

* त्यसरिनें 4th information चाहिं, त्यो variable को value चाहिं कति बेर सम्म रहिरहन्छ भन्ने बारे हो – i.e. Life of the variable.
मानौं int x=5; गरिएको छ भने, x भन्ने variable र त्यसमा 5 भन्ने value कति time देखी कति time सम्म रहन्छ र त्यस पछि erase हुन्छ त - भन्ने बारेमा हो।
"Timing of diff. section of program" लाई example बाट हेरौं।

```
#include <stdio.h>
#include <math.h>
int add(int,int);
int area(flo
float area(float, float);
int p=3;

void main() {
    int c;
    ....
    ....
}

int add(int x, int y){
    int sum;
    .....
}

float area(float a, float b){
    int result;
    ...
}
```

3 sec
15 sec
4 sec
5 sec
'4sec'
'5 sec'
whole program. (3+15) = (18 sec)

कुनै variable 4 sec लाई exist गर्ने होला, कुनै 5 sec लाई, कुनै 15 sec लाई, कुनै 18 sec लाई)

⇒ So, variable (साथै कुनै funct^n) को यो चार वटा information चाहिं "storage class" ले दिन्छ र त्यो 'word' लाई datatype अगाडि लेखिन्छ (परेको खण्डमा)

Date

In C language, storage class is used for determining the

- storage location
- Visibility (scope)
- Initial Default value
- Lifetime

of any given variable.

Types of storage classes :

* Automatic → auto
* External → extern
* Static → static
* Register → register

Summary of storage class:

Class	storage Location	Scope	Default Value	Lifetime
auto	RAM	Local (within the block)	Garbage Value	Till execution remains within the block
extern	RAM	Global (whole program)	Zero	Till the program ends. One can declare it anywhere in program
static	RAM	Local	Zero	Till the program ends. Value of variable persists between various function calls.
register	CPU register	Local	Garbage Value	Till execution remains within the block

Date

Automatic Storage class: (keyword- auto)

It is also known as auto storage class. It acts as the default storage class (यानि, variable declare गर्दा यदि कुनै पनि storage class (auto, static, register, extern) mention गरिएन भने by default त्यसको class चाहिं auto हुन जान्छ) for all the variables that are local (within a block) in nature.

Regarding storage location & allocating memory space for variable:

The memory space that these auto variables use is the main memory (RAM). The memory space gets automatically allocated during the run time.

Regarding the scope/visibility:

The scope of these variables is limited to the block within which they are defined. (त्यो block बाहिर त्यो variable झन्लाई चिन्दैन; बाहिर use गऱ्यो भने error आउँछ — undefined variable)

Initial Default Value:

After these variables are only declared, the initial value of those variables is a garbage value.

Lifetime of the variable:

When the program execution comes into the specific block (जहाँ त्यो variable चाहिं declared छ), then the variable comes into existence & remains existing till the execution of that block is end. So, when the program execution exits from a block, the memory space that was allocated to the variable gets free.

Date

External Storage Class (Keyword- extern)

It tells us that the variable is defined elsewhere and not within the same block where it is used. Basically, the value is assigned to it in a different block & this can be overwritten/changed in a different block as well.

So, an extern variable is nothing but a global variable initialized with a legal value where it is declared in order to be used elsewhere. It can be accessed within any function/block.

The main purpose of using extern variables is that they can be accessed between two different files which are a part of a large program.

Static :

The speciality of static variables can be understood via a program (given below) & compared with auto.

'Static'

```
void fun(){
    static int x=1;
    printf("%d \n",x);
    x = x+1;
}
void main(){
    fun();
    fun();
    fun();
}
```

OUTPUT:

1
2
3

'auto'

```
void fun(){
    auto int x=1;
    printf("%d \n",x);
    x = x+1;
}
void main(){
    fun();
    fun();
    fun();
}
```

OUTPUT:

1
1
1

} seems normal

Date

Static variables have the property of preserving (सांचेर राख्ने, मर्न नदिने) their value of their last use in their scope.

{
.....
.....
.....
}

यानौं यो block मा एउटा static variable define गरिएको छ। यो variable को पनि पहिचान (scope) यहि block भित्र मात्र हो तर के चाहिं हुने रहेछ भन्दा : यानौं यसको value '5' भनेर initialize गरियो र यहि block भित्र यसमा '3' add गरेर यसैं variable मा store गरियो। यसको नयाँ value '8' हो। Execution जब यो block सकेर बाहिर आउँछ, यसपछि पनि यो variable को value (last वाला) '8' नैं टिकिरहन्छ (हराएर जाँदैन) तर यसको पहिचान चाहिं यहि block भित्र नैं हुन्छ। ~~अगाडिको program मा~~ तर यो variable यदि auto हुन्थ्यो भने, execution चाहिं यो block बाट बाहिर आइना साथ पुरानो value हराएर जान्थ्यो। यहि difference लाई अगाडिको program ले देखाएको छ।

~~अब~~ static variable दोहोरो पटक initialize हुँदैन, यो कुरा पनि अगाडिको program ले देखाएको छ।

```
void fun(){
  static int x = 1;
  printf("%d\n", x);
  x = x+1;
}
```

एसो है हेर्दा, यो func() second or third or चोटि call गर्दा, हरेक चोटि, first line → static int x=1; execute भई हरेक पटक x को value '1' हुने र यस पछिको line ले यहि '1' लाई print गर्न पर्ने तर यस्तो नभई o/p चाहिं 1, 2, 3 गरेर आयो, यानि static variable को initialization वाला line static int x=1; चाहिं मात्र एक पटक चल्ने रहेछ, र पछि नचल्ने — यसैले x को incremented value नै o/p मा आयो — meaning — static variable मात्र एक चोटि initialize हुन्छ [पटक-पटक हैन]

Date

So, we can say that static variables are initialized only once and exist till the termination of the program.

Register (Keyword - register)

Register variables have the same functionality as that of the auto variables. The only difference is, the compiler stores these variables in the register of the microprocessor.

Access time of register variables during run time is much faster than the variables stored in RAM.

If free register is not available, then they get stored in RAM (memory).

Usually, few variables, which are to be accessed very frequently in a program, are declared as register variables.

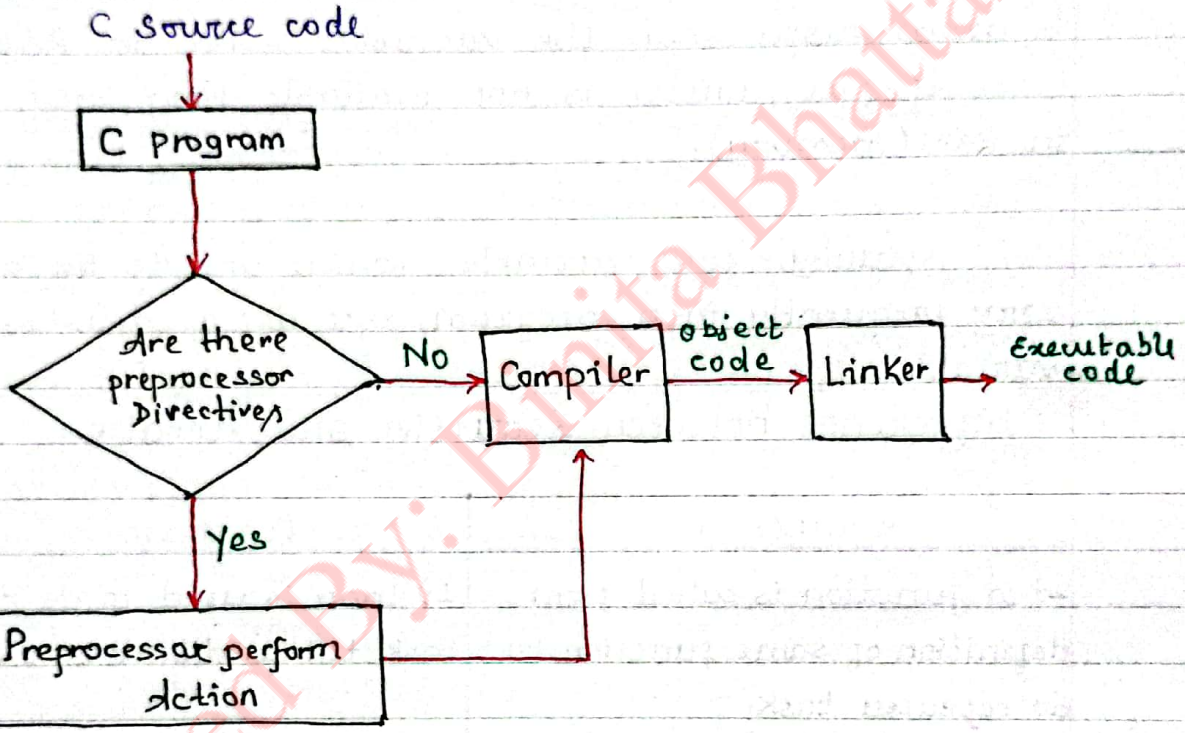
Difference between Recursive and Iteration

Recursive	Iteration
1.) A function is called from definition of same function to do repeated task.	1.) Loop is used to do repeated task till condition is satisfied.
2.) A function calls itself until some condition is satisfied.	2.) A function doesn't call itself.
3.) All problems cannot be solved through recursion.	3.) All problems can be solved through iteration.
4.) Recursive function are usually shorter in length.	4.) Length of iteration are longer.
5.) There must be an exclusive if statement specifying stopping condition.	5.) Iteration involves 4 steps: declaration, initialization, condition & updating.

Date

Pre-processor directive :

Preprocessor directives are lines in the program that starts with #. The # is followed by an identifier that is the directive name.
Preprocessors are programs that process our source code before compilation. Lets see the steps involved between writing a program & executing the program.



The source code is first stored in a file, let the name be program.c. This file is then processed by preprocessors and an expanded source code file is generated named program.i. This expanded file is compiled by the compiler & an object file is generated named program.obj. Finally, the linkerlinks object code file to the object code of the library functions to generate the executable file program.exe.

Date

Preprocessor programs provide preprocessor directives that tell the compiler to preprocess the source code before compiling.
As the preprocessor is not the part of compiler, it can be different in various compiler.

As mentioned earlier, all of the preprocessor directives start with '#' which indicates that whatever statement starts with a # will go to the preprocessor program to get executed.

We use preprocessor directive for:
1.) File inclusion
2.) Macro expansion
3.) Conditional compilation
4.) Other directives

File Inclusion:
This type of preprocessor directive tells the compiler to include a file in the source code program. Two types of files can be included by the user:
* Header files or standard files
* User defined files.

Standard files contain definitions of pre-defined functions like printf(), scanf(), strlen() etc.
Syntax:
```
#include <file_name>
```
Here, '<' & '>' brackets tell the compiler to look for that file in the standard directory.
User defined files: When a program becomes very large, it is a good practice to divide it into smaller files & include them whenever required.
Syntax:
```
#include "filename"
```

Date

Macro (#define)

A macro is a segment of code which is replaced by the value of macro. Macro is defined by #define directive.

There are two types of Macros:

1) Object-like Macros
2) Function-like Macros

1) Object-like Macros:

It is an identifier that is replaced by value. It is widely used to represent numeric constants. For e.g:

```
#define PI 3.1415
```

Here, PI is the macro name which will be replaced by the value 3.1415.

2) Function-like Macros:

It looks like function call.

```
#define macro_name macro_expansion
```

for e.g.,

```
#define ADD(a,b) (a+b)
```

Here, ADD is the name of macro which seems accepting two values.

(a+b) is the activity performed by this macro.

Within main() function, if we write ADD(3,5), the '3' & '5' get added as mentioned by macro_expansion & the result '8' appears in place of ADD(3,5). (as if value returned)

Date

Examples:

* Example 1:

```
#define VALUE 10
#define PI 3.1415
#define PLUS +

void main () {
    int x=5, y=7, z;
    printf ("%d \n", VALUE);
    printf ("%f \n", PI);

    z = x PLUS y;
    printf ("sum = %d", z);
    getch();
}
```

* Example 2: (Macro with argument)

```
#define AREA(r) (3.14*r*r)

void main () {
    int x,y,z;
    float a, r;
    printf ("Enter radius: \n");
    scanf ("%f", &r);

    a = AREA(r);
    printf ("Area = %f", a);
    getch();
```

Date

Other preprocessor directives:

- # undef
- # ifdef
- # ifndef
- # if
- # else
- # elif
- # endif
- # error
- # pragma

Difference between Macro & Function:

Macro	Function
1.) It is defined using # directive. It is preprocessed before compilation of source code.	1.) It is defined outside main program. It is processed after submission of source code to compiler.
2.) When a macro is expanded, control is not transferred to macro.	2.) When a function is expanded, control is transferred to function.
3.) Macro makes program size large and needs more memory.	3.) It makes program size small, & compact & needs less memory.
4.) Macro can't call itself.	4.) Can call itself.
5.) Execution is faster than function.	5.) slower compared to macro.
6.) Used for small task	6.) Used for larger task.

Date

Passing Array to function:

1.) Passing single (individually) elements at a time:

We will call a user function from main() function & pass one argument (single element) of an array. The user function will accept the value in its parameter & can use it as required.

Let's write a program to pass each element of an array as argument using 'for' loop & then user function will accept & display the elements. So, user function will be named as display() as it is meant for this purpose (नाम जे पनि राख्न मिल्छ, तर काम अनुसार नाम राख्नु उचित हो)

```
#include<stdio.h>
void display(int x);
void main(){
    int i;
    int arr[5]={5, 3, 6, 12, 7};
    for(i=0; i<5; i++){
        display(a[i]);
    }
    getch();
}
void display(int x){
    printf("%d \t", x);
}
```

Date

2) Passing whole array at a time:

While calling the user function, there requires two arguments: "array_name" and the size of the array.

While defining the function, there will be two parameters: declared array with datatype & name (for e.g. int x[]) and the secondone is the size of the array.

Example 1 (Display array using funct")

```
#include <stdio.h>
void display (int d[], int n);

void main() {
    int a[5] = {5, 10, 15, 20, 25};
    display (a, 5); //name of array, & size as arguments
    getch();
}

void display (int d[], int n) {
    int i;
    for(i=0; i<n; i++){
        printf("%d\t", d[i]);
    }
}
```

Date

Example 2:

Program to input n numbers in an array and display it in reverse order using function.

We will use two user defined function: one to display original array, & another to reverse and display it.

```
# include<stdio.h>
int i; // defined globally – variable used in 'for' loop
void display(int a[], int m);
void reverse(int b[], int n);

void main(){
    int x[100], num;
    printf("Provide the number of elements to enter:");
    scanf("%d", &num);
    printf("Enter %d numbers", num);
    for(i=0; i<num; i++) {
        scanf("%d", &x[i]);
    }
    display(x, num);
    reverse(x, num);
    getch();
}
```

```
void display(int a[], int m)
{
   printf("\n Array elements are\n");
   for(i=0; i<m; i++){
      printf("%d\t", a[i]);
   }
}
```

```
void reverse(int b[], int n){
   printf("Reverse elements are \n");
   for(i=n-1; i>=0; i--){
      printf("%d\t", b[i]);
   }
}
```

Date

Passing 2-D array into function:

Example 1:

Program to input mxn order matrix & display it using function.

```
#include <stdio.h>
int i, j; // variables for 'for-loop' globally defined
void display (int a[][20], int row, int col);

void main () {
    int arr[20][20], row, col;
    printf ("Enter row & column size:");
    scanf ("%d %d", &row, &col);
    printf ("Enter %d elements", row*col);
    for (i=0; i<row; i++) {
        for (j=0; j<col; j++) {
            scanf ("%d", &arr[i][j]);
        }
    }
    display (a, row, col);
    getch ();
}

void display (int a[20][20], int r, int c) {
    for (i=0; i<r; i++) {
        for (j=0; j<c; j++) {
            printf ("%d \t", a[i][j]);
        }
        printf ("\n");
    }
}
```