Date

Introduction:

Before we define what is function, let's first understand why function is needed.
If we look carefully, we find that if-else concept is not just to make convenience in programming, but it is a 'must' tool without which the programs related to condition-testing cannot be implemented. But, if we look at the loop - for, while.., its use is not mandatory, i.e. completing the code & bringing the required result is still possible without the use of loop also, but program writing will be very hectic & lengthy. So, we can say 'loops' are introduced in programming to ease coding especially when repetition required.
Here, also, we should first know whether function is 'must-have' tool or is it 'aiding' in programming. The answer is, 'function' concept actually aids in programming in certain scenario with the exception of main() function in C which is 'must-have' function.

Now, let's understand why & in what scenario function is needed. (Motivation & Importance)

* Very often in computer programming, there is some code that must be executed multiple times at different places in the program.
* It is also common to need the same code in multiple different programs.
* In such cases (as mentioned above), the overall program becomes too lengthy, time consuming, the memory is more consumed, increased processing

time, difficulty in error finding or say, tracking a large program, difficulty in modifying program (removing/adding certain functionality) etc.

All these are the motivation (statement of problem) towards the need of Function.

Definition: A 'function' is a block of code that performs a specific task. We can divide a large program into the basic building blocks known as functions. The function contains the set of programming statements enclosed by {}. Hence, collection of such functions creates a program.
The function is also known as procedure, subroutine, method in other programming languages.

Advantages of Functions:

* By using functions, we can avoid rewriting same logic/code again and again in a program.
* We can call C functions any number of times in a program and from any place in a program.
* We can track a large C program easily when it is divided into multiple functions.
* Reusability is the main achievement of C functions.

Three aspects of a C function:

* Function declaration
* Function definition
* Function call

Date

* Function Declaration: (Function Prototype)

It tells the compiler about
- the name of function
- number of parameters a function takes
- types & order of parameters a function takes
- type of data a function returns

Function declarations do not need (जरूरि छैन) to include parameter names, but function definitions must.

Syntax:

```
return_type name_of_function (data_type_of_parameters);
```

variable names are not compulsory to mention

Function declaration for sum of two numbers, as an example is shown below:

```
int sum(int var1, int var2);
```

As already mentioned that the parameter name/s is not compulsory (var1, var2) while declaring, it can also be done as:

```
int sum(int, int);
```

* Function Definition:

It contains the actual statements which are to be executed. It is the most important aspect to which the control comes when the function is called.
It consists of function header and a function body which is enclosed in curly braces.

Date

* Function header consists of :

- Return_type : The function always starts with a return type of the function. But, if the function is not supposed to return any value, then, the 'void' keyword is used as the return type of the function.

- Function_name: Name of the function that must be unique.

- Parameters : Variable name/s (along with the datatype) which accepts the value/s sent during function call.

Syntax:

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

Example:

```
void sum(int a, int b)
{
    int sum_result;
    sum_result = a+b;
    printf("Sum of two numbers : %d", sum_result);
}
```

Date

* Function Call:

Once function definition is ready, it must then be called inside boundary of main function in order to execute codes defined in the function body. Because the function never comes in action if it is not called. While calling the function, function name with arguments (if any) within the small braces are used.

During function call, if there are arguments (parameters) to pass, then, data type, number & order of parameters must match with what & how these are defined in function declaration & definition.

Types of function:

* Library function
* User-defined function

Library function:

It is also referred to as **built-in function**. There is a compiler package that already ~~exits~~ exists which contains these functions, each of which has a specific meaning & purpose, & is included in the package. Built-in functions have the advantage of being directly usable without being defined, whereas user-defined functions must be declared & defined before being used. Functions falling under a particular category are placed in one library (header file), & there are many such libraries in C. Each library can have more than one functions of similar category. And, combination of all libraries are contained into a single package.

Date

Examples of in-built functions:
printf(), scanf(), gets(), puts()... [stdio.h]
pow(), sqrt().... [math.h]
strcmp(), strcpy()... [string.h]

Advantages of C library functions:
- are easy to use
- are optimized for better performance
- save a lot of time i.e., function development time.
- are convenient as they always work

User-defined function:
The functions that are created by the programmer are known as User-defined functions or "tailor-made functions". These can be improved and modified according to the need of the programmer.

Advantages:
- can be modified as per need.
- codes of these functions can also be used in other programs.
- These functions are easy to understand, debug and maintain.

Categories of function:

Based on Function arguments & Return Values in C, there are 4 types of functions:
* Function with no arguments and no return value
* Function with no arguments and with return value
* Function with argument & with no return value
* Function with argument & with return value.

Date

No arguments & No return value

Calling function	Analysis	Called function

main()
{
...
...
fun();
}
	→ No arguments passed.
	
	No values returned ←
		fun()
		{
		...
		...
		}

No arguments & with return values

main(){
int c;
...
c=fun();
...
}
	→ No arguments passed.
	
	Values are returned back. ←
		fun(){
		...
		...
		return c;
		}

With arguments & without return values

main(){
...
fun(a, b);
...
}
	→ Arguments are passed
	
	No values returned ←
		fun(int x, int y){
		...
		...
		}

With arguments & with return values

main(){
int c;
...
c=fun(a,b);
...
}
	→ Arguments are passed.
	
	Values are returned back. ←
		fun(int x, int y){
		...
		...
		return c;
		}

Date

Examples:

1) No argument and no return value

```
#include<stdio.h>
main(){
    void sum();      // function declaration locally
    sum();           // function is called
    getch();
}

void sum(){          // function definition
    int a,b,c;
    printf("Enter 2 numbers:");
    scanf("%d %d",&a,&b);
    c=a+b;
    printf("sum = %d",c);
}
```

function body

2) No argument and with return value

```
#include<stdio.h>
main(){
    int c;
    int sum();       // function declaration locally
    c=sum();         // function calling
    printf("sum = %d",c);
    getch();
}
int sum(){
    int a,b,c;
    printf("Enter 2 numbers:");
    scanf("%d %d",&a,&b);
    c=a+b;
    return c;        // value is returned.
}
```

Date

3) With arguments & no return value

```
#include<stdio.h>
main(){
    void sum(int, int); // declaration in local scope
    int a,b;
    printf("Enter 2 numbers:");
    scanf("%d %d", &a, &b);
    sum(a,b); //function called & 'a' & 'b' are sent
    getch();
}
void sum(int x, int y){
    int z;
    z = x+y;
    printf("Sum = %d", z);
}
```

4) With arguments and with return value

```
#include<stdio.h>
main(){
    int sum(int, int); // declaration in local scope
    int a,b,c;
    printf("Enter 2 numbers:");
    scanf("%d %d", &a, &b);
    c=a+b
    c = sum(a,b);   // calling function
    printf("sum = %d", c);
    getch();
}
int sum(int a, int b){
    int c;
    c = a+b;
    return c;
}
```

Date

Function Prototype:

A function prototype is simply the declaration of a function that specifies:

* function's name
* function's parameters
* return type

It doesn't contain function body. A function prototype gives information to the compiler that the function may later be used in the program.

Syntax:

```
returnType functionName (type1 argument1, type2 argument2,...);
```

for e.g., `int addNumbers(int a, int b);`

Here,

1. name of the function is addNumbers()
2. return type of the function is int
3. two arguments of type int will be passed to the function.

Note: The function prototype is not needed if the user-defined function is defined before the main() function.

Calling a function:

A function must be called from main() function in order to execute the codes within the function. When it is called, the control of the program is transferred to the user-defined function.

Date

Syntax:

```
functionName (argument1, argument2, ...);
```

This code calls the function named as functionName. The program control arrives to the function and the code within it will be executed. During the call, the values mentioned by argument1, argument2 ... are passed (sent) to the function where there are similar type variables present, in the small braces () of the header of the function, that accepts these passed values of arguments.

Passing arguments to function : (with example)

```
# include <stdio.h>
int addNumbers (int a, int b); // function declared;
                               // 1st 'int' shows that the
                               function returns a value
                               of integer type.
                               // int a & int b indicate that
                               the function is defined with
                               two integer type parameters
                               &, while calling function,
                               two integer values to be
                               passed.

int main () {
   ...
   sum = addNumbers (n1, n2);
   ...
}
int addNumbers (int a, int b)
{
   ...
}
```

n1 moves into 'a' & n2 moves into 'b'.

⇓

Passing arguments

Date

Return Statement

If a function is of return type, then it consists of a return statement at the end which terminates the execution of a function and returns a value to the location from where the function is called.
When a function is called from a calling function (usually main() function), the program control gets transferred to the function being called. Then the codes within the called function start executing. When execution reaches return statement at the end, then the program control gets back to the location in calling function from where it was called.
Lets see through code example to have a visual understanding about what value gets returned where.

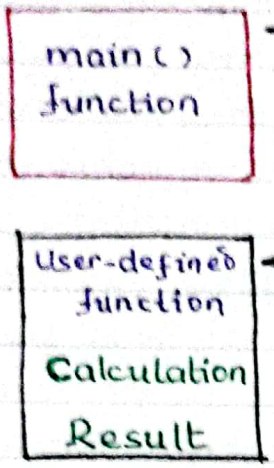
```
# include <stdio.h>
int addNumbers (int a, int b);
int main() {
    ...
    sum = addNumbers (n1, n2);
    ...
}

int addNumbers (int a, int b) {
    ...
    return result;
}
```

means → sum = result

Date

कैहि Tips र याद राख्न पर्ने कुराहरू :



← यो हो main() function र
यो हो हामिले बनाएको function

जुन सुकै case मा पनि calculation र result चाहिं user-defined function मैं हुन्छ।

1) * भने पछि, 'result' जब हाम्रो function मा आउँछ, त्यहि display गर्दिन पनि सकिन्छ र यदि question ले, result चाहिं main() function मा print गर भनेको छ भने स्वभाविक हो कि result लाई return गर्नु पर्छ।

2) * यस्तै संदर्भमा, आएको result मा further कैहि manipulation गर्न पर्ने छ र त्यो manipulation (परिमार्जन यदि main() function मा गर्न पर्ने छ भने पनि return गर्न पछि भनेर फुस्न पर्ने हुन्छ।

3) * अर्को संदर्भ छ, हामिले बनाएको function मा भित्र calculation हुन्छ, त्यो त ठिक छ, तर त्यो calculation गर्न चाहिने data (values) चाहिं कहाँ हुन्छ या भनौं कहाँ बाट आउँछ।

3.1- चाहे program में initialize गर्ने होस् या user संग run-time मा माग्ने होस्, यो कम हामिले user-defined function भित्र पनि गर्न सक्छौं या main() function भित्र पनि गर्न सकिन्छ।

3.1.1 उदाहरण - यदि user-defined function भित्र data लिने हो भने फुर्सै सक्यो, त्यहिं लिने

अनि त्यहि calculation गर्ने।

3.1.2- तर यदि, data चाहिं main() function मा लिएको अनि त्यसलाई argument/s बनाई user-defined function सम्म पुऱ्याउने पर्ने हुन्छ। (situation of passing argument/s to user-defined function)

```
main(){
    data { int a=5;
           int b=8;
    OR
    data { printf, scanf बाट user संग
           value माग्ने
    myFunc(data);
}
```

```
myFunc(received data){
    perform calculation;
    obtain result;
    then
    Either print result here
    Or return result
}
```

main() मा data छ, त्यसलाई user-defined मा पठाएको

Figure:'A'

```
main(){
    ...
    myFunc();
    ...
}
```

"data pass गर्न पर्दैन"

```
myFunc(){
    int x=3;
    int y=8;      } data
    :
    OR
    printf, scanf बाट
    data user संग माग्ने  } data
    then,
    perform calculation &
    obtain result
    then,
    Either print result here
    Or return result
}
```

user-defined function मै data छ, त्यहि calculation हुन्छ, data लाई main() बाट pass गर्न पर्दैन।

Figure:'B'

Date

4) * second वाला case एक चोटि फेरि हेरौं र यसमा चप जानकारि लिऔं। data लाई user-defined function भन्ने define गर्या पनि program ले काम गरेकै छ। तर मानौं कि एस्तो परिस्थिति छ कि जुन data चाहिं function भित्र calculation गर्न लाई चाहिन्छ, त्यहि data चाहिं main() function भित्र पनि अरू कतै आवश्यक परेको छ।

4.1) यो स्थितिमा, second case मा जस्तै गरि user-defined function भित्र यदि variable को नाम दारवी त्यसलाई value दिइन्छ भने, त्यहिं user-defined function भित्र त्यो data लाई यस्ै दिइएको variable को नाम बाट प्रयोग गर्न कुनै समस्या पर्दैन, तर (किनकी त्यो data त main() function मा पनि चाहिएको छ) त्यो variable लाई यदि हामिले main() function मा बेखेर प्रयोग गऱ्यौं भने ERROR आउँछ र program सफल हुन पाउँदैन [Concept of local variable- to be studied later in detail] किनकि एउटा function भित्र define/initialize गरिएको variable लाई अरू function ले, या भनौं त्यो function बाहिर कतै पनि चिन्दैन। त्यसैले user-defined function भित्र define गरिएको variable लाई main() function भित्र कतै बेखेर प्रयोग गर्न साथ ERROR आउँछ। (vice-versa)

4.2) त्यसैले माथि भनेको परिस्थिति (user-defined function भित्रको calculation मा प्रयोग हुने values/variables चाहिं main() function मा पनि चाहिने परिस्थिति) मा चाहिं, त्यो data लाई main() function भित्र define गर्ने र (ताकि त्यो data अब main() भित्र जता पनि use गर्न सकियोस्) र user-defined function सम्म पुऱ्याउन त्यसलाई call गर्दा argument बनाएर पठाउने।

Date

अथवा

4.3) यसरि धेरै ठाउँमा, वा भनौं धेरै वटा function हरूमा same data प्रयोग हुने वाला छ भने एस्तो data लाई एसरि define गर्ने कि यसलाई whole program भरि सबैले चिनोस् → global variable भनिन्छ यस्तोलाई। यसको लाई यस्तो data लाई main() function लैरुन अगाडि (पहिले – main भित्र नपर्ने गरि) declare गर्न पर्छ।

```
#include <stdio.h>
#.....
int x, y;
float a;
main(){
   ...
   ...
}
myFunc(){
   ...
   ...
}
```

यो चाहिं main() बाहिरै सुरुमै define भएपछि यसलाई main() को सार्थे अन्य function हरू (यस program भित्रका) सबैले चिन्धन र प्रयोग गर्न सकिन्छ।

→ global variables

4.4) जुन variables हरू कुनै function भित्र define गरिन्छ, (जसलाई यस function बाहिर कसैले चिन्दैन) यसलाई local variables भनिन्छ।

Date

Some Examples:

★ Write a program to calculate simple interest using function and print the result inside the main.

- Notice that the result is to be printed inside main, i.e., after calculation & obtaining the result in used-defined function, it must be sent back to main function — using return statement.
- This tells us that the user-defined function must be of returning value (int, float...) according to the nature of result, hence to be declared accordingly.
- Next, we need to think about the data required for the program (principle (p), time (t), & rate (r)) & think where to declare & initialize them (i.e., within main() or user-defined function). As question has not mentioned any constraint, it is of our choice, but be careful that:
  - * if we define these data in user-defined function, then, we do not need to send data from main() to user-defined function as arguments, and
  - * if we define them within main(), we will have to send those data in the form of arguments while calling the user-defined function.
- But, it is a good choice to define the required data in main() as they can be re-used anywhere within main() if necessary.

Date

```
#include <stdio.h>                    // global declaration
float calculate_interest (float p, int t, float r);
void main() {
    float pr, ra, in ; // principal, rate, interest
    time
    int ti ; // time
    printf("Enter principal, time and rate";
    scanf (" %f %d %f", &pr, &ti, &ra); // input data
                                          is defined in main(),
                                          so we have to pass it
                                          to used-defined funct"
                                          as p arguments.
    in = calculate_interest (pr, ra, ti, ra); // three data
                                                 sent as arguments.
    printf(" Interest is %f", in);
    getch();
}

// now defining user-defined function which receives
   the three data, calculates interest using it, &
   send back the result to main().

float calculate_interest(float p, int t, float r) {
    float i;
    i = (p * t * r)/100;
    return (i);
}
```

Date

* Write a program to find factorial of a number.

- We will form a function which calculates, or say, does the calculation part for finding factorial.
- We will ask user to enter the number (for which factorial to be found) in main() — hence we have to pass this number as arguments from main() to user-defined function.
- We will print the result in main(), so, the defined function has to return the result.
- The result of factorial can be very big number for which 'int' data type may not be suitable, hence, we define a data type of Long int for the result as well as the return type of the function.

```
#include<stdio.h>
Long int factorial (int num); // function declaration
void main(){
    Long int f; //variable to store result of factorial
    int n; // number of which factorial is to be found.
    printf("Enter a number of which factorial needed:");
    scanf("%d", &n);

    f = factorial (n); // function is called & 'n' is passed.
    printf("\n");
    printf(" Factorial of %d is %ld", n, f);
    getch();
}
```

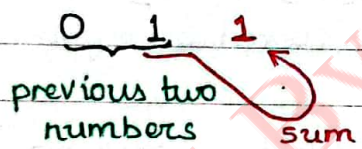
Date

```
long int factorial (int num){ //function header
    long int fact = 1; // calculated result will be saved
                          in this variable in this user-
                          defined function.
    int i; // variable used for 'for'-loop.
    for(i=1; i<=num; i++){
        fact = fact * i;
    }
    return fact;
}
```

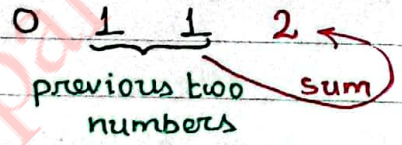
Write a program to generate Fibonacci series upto n terms using function.

This series starts with 0 1 .....

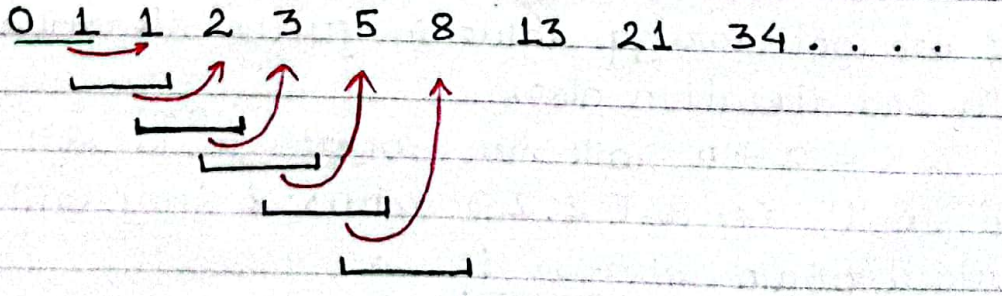
After these two numbers, next number is always the sum of previous two numbers. So, the third number in this series is sum of '0' & '1' which are previous two numbers.



4th number is again sum of previous two numbers, i.e. '1' '1'.



Similarly,



Date

Programming idea:
The first two numbers 0 and 1 are assigned to two variables:

0 → a
1 → b

and we print them first.
Then, we declare third variable 'c' to store the sum of previous 2 numbers. While finding 3rd number, c is a+b. But this statement 'c=a+b' is to be written within for-loop. 'For' loop is required because it is a series & we have to find numbers more and more.
Notice that, the third number which is obtained by adding 0 & 1 is actually first action that happens in for-loop with 1st iteration. When c = a+b is performed, then 'c' has to be displayed using 'printf'.
Till now, i.e., 1st iteration of 'for' loop, we get,

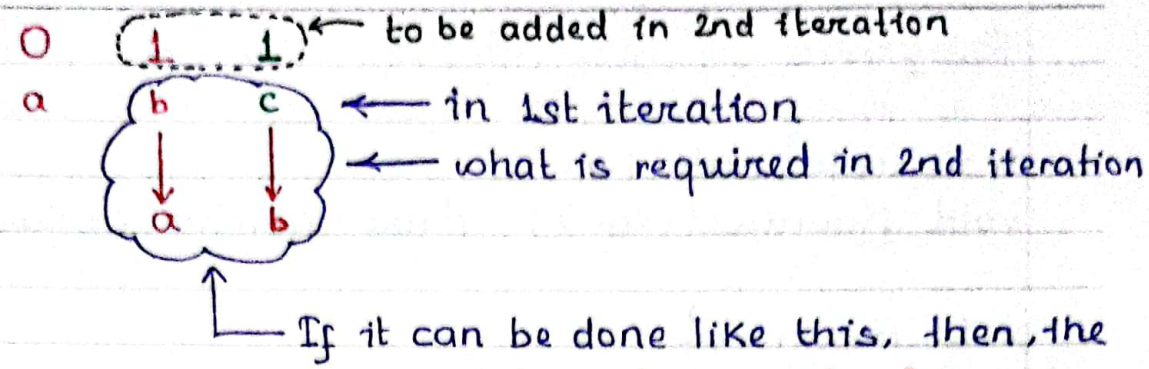
0 1 (1) ← obtained on 1st iteration of for loop.
(a) (b) (c)

Now, it's time to think what happens on 2nd iteration.
Codes written within 'for' Loop are:

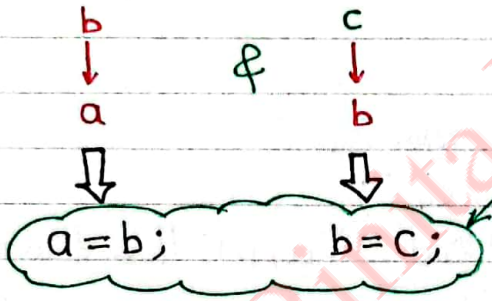
```
c = a+b;
printf("%d", c);
```

If this much only is the code, it works for the 3rd number only in the series (in 1st iteration), & we get wrong value in further iteration.
In 2nd iteration also,
c = a + b will run where 'a' is still '0' & 'b' is '1', i.e. 1st & 2nd values, & they will be added again whereas in →
0 1 → 0 [1 1] → these two must be added but code written is c = a+b; that means:

Date

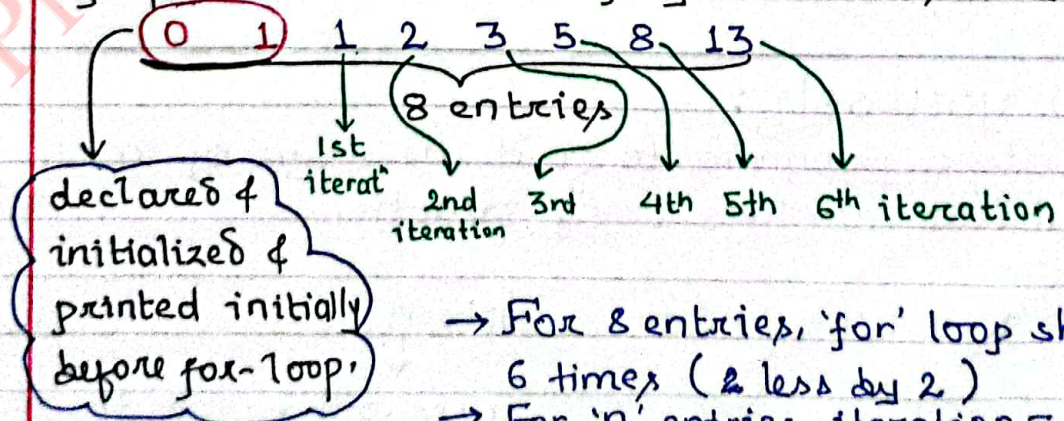


If it can be done like this, then, the same statement c=a+b will add last two values & we get correct answer. So, ~~in 2nd iteration~~ before executing 'c=a+b' in 2nd iteration, we must do these two replacements :



These replacements bring the last two values in 'a' & 'b' everytime so that when c=a+b; is executed, it ~~adds~~ always adds last 2 values.

Now, let's understand the total no. of iterations required in 'for' loop. Question always mentions the total number of values to be displayed in series. If question asks to display 8 entries like-



→ For 8 entries, 'for' loop should iterate 6 times (2 less by 2)
→ For 'n' entries, iteration = n-2 times

Date

Program:

```
# include <stdio.h>
void fibonacci(int);   // user function declared.

void main(){
    int n;
    printf("Enter the number of entries in series:");
    scanf("%d",&n);
    fibonacci(n);
    getch();
}

void fibonacci(int num) {
    int a=0, b=1, c, i;
    printf("%d\t %d\t",a,b); // 'a' & 'b' ⇒ '0 1', 1st two
                                numbers printed
    for(i=0; i<n-2; i++){
        c=a+b;
        printf("%d\t", c); // 3rd & so on...numbers
                              are displayed
        a=b;
        b=c; // 'a' & 'b' are replaced by last two
                (latest) numbers
    }
}
```

★ Do yourself:

Write a program to find the sum of 'n' natural numbers using function:

1 + 2 + 3 + 4 + 5 + 6 + ... upto 'n'

↓

user will provide it.

Date

* Write a program to find area of two circle usin having different radius using same function.

```
#include <stdio.h>
float circle_area(int);

void main() {
        float area1, area2;
        int radius1, radius2;
        printf("Enter two radius:");
        scanf("%d %d", &radius1, &radius2);

 area1 = circle_area(radius1);
        area2 = circle_area(radius2);

        printf("Area of 1st circle is %f", area1);
        printf("Area of 2nd circle is %f", area2);

        getch();
}

float circle_area(int r) {
        float area;
        area = 3.14 * r * r;
        return area;
}
```

Date

* Write a program to find the reverse of a number using function.
If input is 1536, output be 6351.
Understanding:

e.g.

1	5	3	6

Here, we first want the individual digits to be separated. This can be done by taking remainder by dividing given number by 10.

Remainder of $\frac{1536}{10}$ = 6

Remainder of $\frac{8237}{10}$ = 7

In 1st attempt, we will get the last digit of given number. Then, we need to form two digits, i.e., '63' from 1536.
Making '63' involves two steps after the last digit '6' was obtained (as remainder).

- Making '6' in two place digits, i.e. '60' by multiplying '6' with 10.
- Then, '60' to be added with next digit from right side (i.e. 3) which makes it 63 (two digits reversed).

But the question is, how to get that second digit '3' from right side. After completion of 1st attempt, i.e. obtaining last digit '6', we will divide our number by 10 [i.e. 1536/10 ⇒ 153]. Now, again, if we take the remainder of 153 by dividing by 10, we get '3'.
So, previous '6' is multiplied by 10, & added with new remainder '3' gives 63.

```
rev_num = rev_num*10 + rem;
```

Date

rev_num = rev_num * 10 + rem. ——— eqn (i)

'0' in 1st attempt — 6 ← in 1st attempt

rev_num = 6 ← The 'rem' obtained in 1st attempt is saved in rev_num.

As mentioned earlier,
In 2nd attempt, we need to obtain as
rev_num = 63

↓This can be achieved by the same eqn(i) above but before that, we need 2nd remainder, i.e. '3' from 1536. As eqn(i) is to be executed again, it will be under 'loop'.
Before 1st attempt (1st iteration ends) & eqn(i) executes again, we must perform some tasks to get '3'.

→ '3' is remainder of '153' (153 % 10)
so, given a number '1536' will be first converted into 153 as by dividing 1536 by '10' & storing in the same original integer variable.

~~→ finding rem = 153;~~

Now, when loop iterates, the first line is about finding remainder of given a number:
rem = n % 10; but in second iteration, the given number 'n' has already been modified as 153 from 1536. ($n = n/10 = 1536/10 = 153$)
$\therefore$ rem = 153 % 10 = 3

Now, eqn(i) gives
rev_num = rev_num * 10 + rem
= 6 * 10 + 3
rev_num = 63 ← obtained in 2nd iteration & so on.

[Loop will iterate until 'n' becomes zero due to $n = n/10$ as $n = 1536; n = 153; n = 15; n = 1; n = 0$ [loop terminates]

Date

Program:

```
#include <stdio.h>
void reverse(int);

void main(){
    int num;
    printf("Enter a number");
    scanf("%d", &num);
    reverse(num);
    getch();
}

void reverse(int n){
    int rem, rev_num;

    while(n>0){
        rem = n%10;
        rev_num = rev_num*10 + rem;
        n = n/10;
    }

    printf("\n The reverse number is %d:", rev_num);
}
```