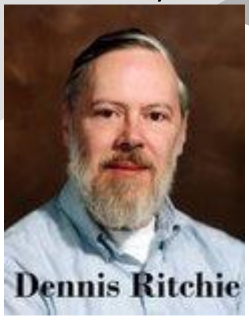
C Programming

History:

History of C language is interesting to know. Here we are going to discuss a brief history of the c language.
C programming language was developed in 1972 by Dennis Ritchie



at bell laboratories of AT&T (American Telephone & Telegraph), located in the U.S.A.
Dennis Ritchie is known as the founder of the c language.
It was developed to overcome the problems of previous languages such as B, BCPL, etc.
Initially, C language was developed to be used in UNIX operating system. It inherits many features of previous languages such as B and BCPL.
Let's see the programming languages that were developed before C language.

Language	Year	Developed By
Algol	1960	International Group
BCPL	1967	Martin Richard
B	1970	Ken Thompson
Traditional C	1972	Dennis Ritchie
K & R C	1978	Kernighan & Dennis Ritchie
ANSI C	1989	ANSI Committee
ANSI/ISO C	1990	ISO Committee

C99	1999	Standardization Committee

Fundemental of C:

First program in C

C programing language is a language that we doesn’t use for communicate with any human or other lives. It is use for communicate with Computer and machine.

So before you go through the tutorials of C programming, we just begin by writing the **first program in C** to print a line.

My first program in C: Hello World !!!

Program to display “Hello world!!!”

```
/* This is my first program in C..*/

#include <stdio.h>                                          // Header Section - Library including

int main()                                                  // main section started from here

{

    printf("Hello world!!!\n");

    return 0;

}
```

Output

```
Hello World!!!
```

This is a simple basic programme that a print a line.

EXPLANATION OF THE PROGRAM

Line 1:

The first line of the program begins with `/*` and ends with `*/` which represents that any text written between these two symbols is a *comment*.

Comments do not affect program while running because C compiler ignores the comments and programmers use these comments for the documentation of the program.

Line 2:

```
#include <stdio.h>
```

This is the directive to the **C Preprocessor** (You might wonder what these words like preprocessor and directives are but don't worry these things will be crystal clear after going through next chapters).

Lines beginning with `#` are called **preprocessor directives** and this specific line tells preprocessor to include the contents of standard input/output header file (stdio.h) in the program while the program is being compiled.

All kind of funtion defination is defined in Library section. This Include in the beggening of the programme. Two types of Function: 1. Built-in which is pre-defined and included by library file, and 2. user Defined Function, initialized by programmer/Coder.

Line 3:

```
int main ( )
```

This is the main functions of the program which is part of every C program.

C programs can have multiple functions which are also called the building blocks, however, one of the function should be `main`.

The code of the function is written between the curly braces and in this program also the body of the function is enclosed by the braces in line 4 and 7.

Line 5:

```
printf("Hello World!!!");
```

This is the standard output functions which instruct the computer to print anything written between the quotes (" ..text..").

This entire line including the `printf`, its arguments within the parenthesis and the semicolon is called a *statement*.

Every statement must end with a semicolon.

Good Programming Practice

While writing C program, commenting is a good idea which makes clear about the purpose of the function.

Line 6:

```
return 0;
```

This is nothing but an exit sequence for the function.

Moreover, it signifies the termination of the function main returning the integer value 0.

Escape sequence in C Programming

Escape sequences are the characters which are not printed. The `\` backslash is called **escape character.**

Escape sequence	**Description**
\n	Newline. This will place a cursor on the beginning of the second line.
\t	Tab. This will insert horizontal tab.
\\	Backslash. Insert backslash in a string.
\a	Alert. This will signify alert sign or sound without changing the position of cursor

C Programming Identifiers / Variable Declaretion

In the real world, when a new baby is born a name is given. Similarly, in C program when a new variable, function or an array is declared a particular name is given to them which is called an identifier. For example

```
int apple;
```

Here, `apple` is an identifier of integer type variable.

So identifiers are the user defined names. However, there are certain rules that must be followed while writing an identifier.

While writing variable name use the meaningful word which makes a program self-documenting. For example: totalMoney

First letter of an identifier must be alphabet (underscore is also allowed)

- Identifier can only contain letters, digits and underscores
- Maximum length of identifier allowed is 31 characters
- White space is not allowed
- Keywords cannot be used as identifier

Declaration types of variable:

1. Camel case:

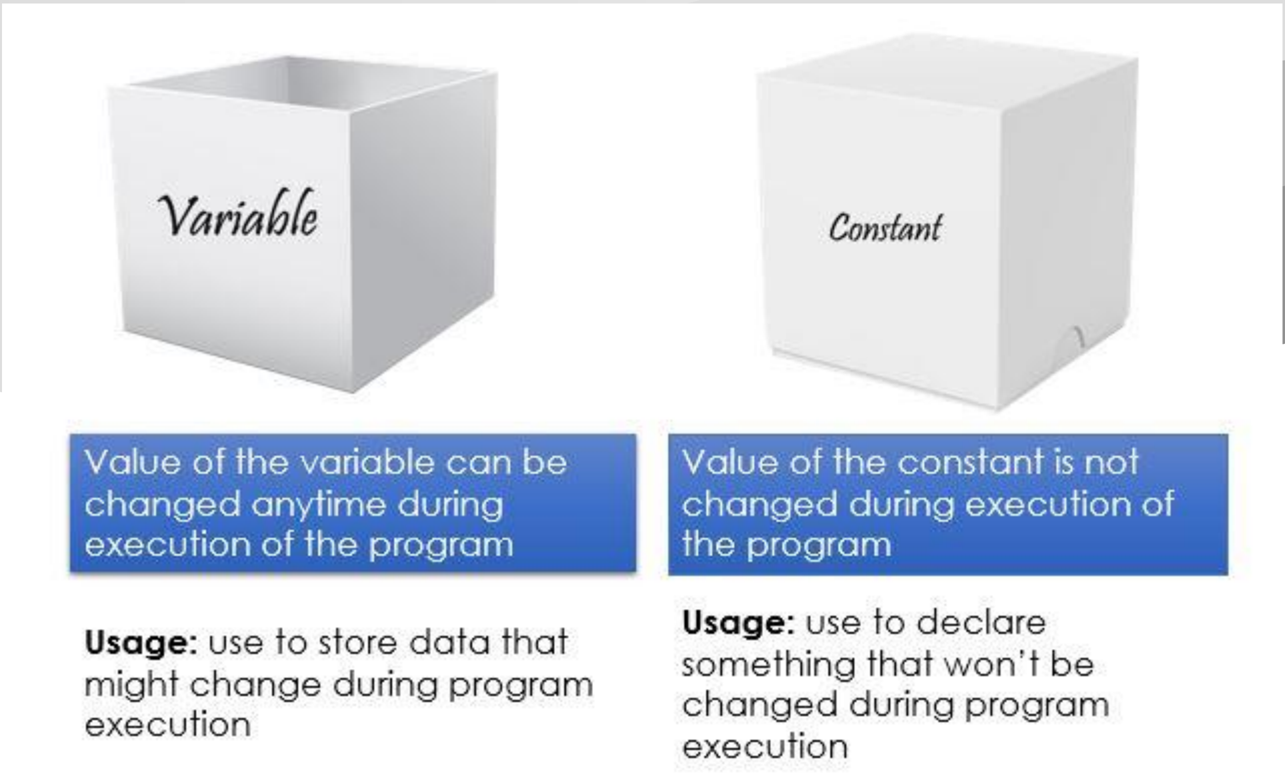
    *Upper Camel Case; is also called **Pascal Case**

    **Lower Camel Case

2. Snake case; using underscore

- Since C is case sensitive language variable name written in uppercase will differ from lowercase.

Difference between variable and constant



Value of the variable can be changed anytime during execution of the program

Value of the constant is not changed during execution of the program

Usage: use to store data that might change during program execution

Usage: use to declare something that won't be changed during program execution

C Programming Constants

Constants are the expressions with fixed values that do not change during the execution of the program. To put it up simply constant means whose value cannot be changed.

Following are the **different types of constants we can use in C**.

1. Integer Constants

Integer constants refer to the sequence of digits with no decimal points. The three kinds of integer constants are:

- **Decimals constant:** Decimal numbers are set of digits from 0 to 9 with +ve or -ve sign. For example: 11, -11 etc.
- **Octal constant:** Octal numbers are any combination of digits from set 0 to 7 with leading 0. For example: 0213, 0123 etc.
- **Hexadecimal constant:** Hexadecimal numbers are the sequence of digits preceding 0x or 0X. For example: 0xBD23, OX543 etc.

Rules for constructing Integer Constants

- Name of an integer constant must have at least one digit.
- Comma or blanks are not allowed in the name of an integer constant.
- It should not have a decimal point.

2. Real or Floating point Constants

They are numeric constants with decimal points. The real constants are further categorized as:

- **Fractional Real constant:** These are the set of digits from 0 to 9 with decimal points. For example: 123.3, 0.765 etc.
- **Exponential Real constant:** In the exponential form, the constants are represented in two form:

  Mantissa E exponent: where the Mantissa is either integer or real constant but the exponent is always in integer form.

  For example, 12345.67 = 1.234567 E4, where E4 = 10^4.

Rules for constructing Real or Floating point Constants

- Name of a real constant must have at least one digit.
- Comma or blanks are not allowed in the name of a real constant.
- It should have a decimal point.

3. Character constant

Character constants are the set of alphabet enclosed in single quotes. For example: ‘A’, ‘f’, ‘i’ etc.

For example: ‘A’, ‘f’, ‘i’ etc…

Rules for constructing Character Constant

- The maximum length of a character constant can be one character.

4. Declared Constant

As same as declaring a variable using the prefix const we can create new constants whose values can’t be altered once defined.

For example:

```
const int b = 100;
```

This signifies that `b` will have a constant integer value of **100**. Not only integer, by using prefix we can declare character constant and string constant as well.

5. String Constant

String constants are the sequence of characters enclosed in a pair of double quotes (" ").

For example: "Hello"

6. Enumerated Constant

Enumerated constant creates set of constants with a single line using keyword `enum`.

Syntax

```
enum type_name{ var1, var2, var3 };
```

Here `var1`, `var2` and `var3` are the values of enumerated datatype `type_name`.

By default the `var1`, `var2` and `var3` will have values 0, 1 and 2.

By using assignment operators, we can assign any value to `var1`, `var2` and `var3`.

Visit C programming operators to learn about different operators used in C.

For example:

```
enum COLOR {RED = 5, GREEN, WHITE = 8};
```

This code sets *red* to 5 and *white* to 8.

The member without assigned value will possess the 1 higher value than the previous one. In this case, *green*will be set to 6.

C Programming Datatype

Datatype, which kind of data we want to use. E.g integer, float, charcter etc.

Datatypes in C programming

Fundamental data types:	integer type
	floating type
	Character type
Derived data types:	Function
	Arrays

User-defined data types:	Pointer
	Structure
	Union

Fundamental data types in C

Fundamental data types are basic built-in data types of C programming language. There are three fundamental data types in C programming. They are an integer data type, floating data type and character data type.

Data type	**Size(in bytes)**	**Range**
character	1	-128 to 127
integer	2	-32,768 to 32,767
floating	4	-3.4e-38 to 3.4e+38

Syntax for fundamental data types

```
int variable_name;     // keyword int is used for integer datatypes

float variable_name;    // keyword float is used for integer datatypes

char variable_name;     // keyword char is used for integer datatypes
```

Difference between float and double

float	**double**
Floating-point number (i.e. a number containing decimal point or an exponent)	Double-precision floating point number (i.e. more significant figures, and an exponent which may be larger in magnitude)
Size: 4 bytes	Size: 8 bytes

Derived data types in C

Those data types which are derived from the fundamental data types are called **derived data types**. Function, arrays, and pointers are derived data types in C programming language.

For example, an array is derived data type because it contains the similar types of fundamental data types and acts as a new data type for C.

User defined data types in C

Those data types which are defined by the user as per his/her will are called **user-defined data types**. Examples of such data types are structure, union and enumeration.

For example, let's define a structure

```
struct student

 {

  char name[100];

  int roll;

  float marks;

 }
```

Here, with this structure we can create our own defined data type in following way:

```
struct student info;
```

Here, student is user-defined data type where info is variable which holds name, roll number, and marks of a student. Combination of variaus type of Data.

C programming operators

List of C programming operators

Operators in C programming

1: Arithmetic operator

2: Relational operator

3: Logical operator

4: Assignment operator

5: Increment/Decrement operator

6: Conditional operator

7: Bitwise operator

8: Special operator

The data items in which any operation is carried out are called **operands**.

Operators which require two operands are called **binary operators** and which takes single operand are called **unary operators**.

Arithmetic operators in C

Operators used in the arithmetic operation like addition, subtraction, multiplication or division are called arithmetic operators.

Arithmetic operator	Meaning
+	Addition or unary plus
–	Subtraction or unary minus

*	Multiplication
/	Division
%	Modulo division

Relational Operator in C

Relational operators are used to compare two operators depending on their relation.

For example, for comparing the price of two things. The value of a relational operator is either `1` or `0`. If the specified relation is true then `1` else `0`.

```
a > b
```

Here, `>` is a relational operator and it will return `1` if `a` is greater than `b` else it will return `0`.

Relational operator	**Meaning**
<	is less than
<=	is less than or equal to
>	is greater than
>=	is greater than or equal to
==	is equal to
!=	is not equal to

Common Programming Error

A syntax error will occur if the two symbols in any of the operators `==`, `>=`, `<=` and `!=` are separated by the space.

Logical Operator in C

Logical operators are used when more than one condition is tested. They are used to combine more

Here `&&` is relational operator used to combine two relational expressions `a > b` and `c == 1`.

Logical operator	Meaning
&&	logical AND
||	logical OR
!	logical NOT

Example: Logical AND (&&) operator

If we need to make sure that both conditions are true before performing a certain task. In this case, logical `AND &&` is used:

```
if ( gender == 1 && age <= 10 )

   ++childNumber;
```

Example: Logical OR (||) operator

If we wish to ensure that either or both of two conditions are true then we use logical `OR (||)` operator.

```
if (mark < 40 || attendance < 15)

     puts("Student is not qualified for exam");
```

In this case, if any of the conditions is true then `if` statement is also true and the message `Student is not qualified for exam` is printed.

Assignment Operator in C

The assignment operator is used to assign a value or a result to a data item or a variable.

`=` is the assignment operator.

For example:

```
a = 5
```

Here, `=` is assignment operator which assigns value 5 to variable `a`.
Assume:

```
int c = 2, d = 3, e = 4, f = 6, g = 8
```

Assignment Operator	Sample expression	Explanation	Assign
+=	c += 2	c = c + 2	4 to c
-=	d -= 1	d = d – 1	1 to d
*=	e *= 2	e = e * 2	8 to e
/=	f /= 3	f = f / 3	2 to f
%=	g %= 4	g = g % 4	2 to g

Note: Don’t get confused between equality operator `==` and assignment operator `=`.

Bitwise Operator in C

Bitwise operators are used to manipulate data at a bit level. They are used for testing or shifting the bit.

Bitwise operator	Meaning
&	bitwise AND
|	bitwise OR

^	bitwise exclusive OR
<<	shift left
>>	shift right

Conditional operator in C

Conditional operator is a ternary operator that takes three operands.

Syntax

```
x = exp1 ? a : b
```

Here, if `exp1` is non zero i.e. `TRUE` then the value of `a` will be assigned to `x` and if `exp1` is zero i.e. `FALSE`then the value of `b` is assigned to `x`.

learn about conditional operator (?:) in detail.

Increment / Decrement operator in C

C provides an increment operator `++` and decrement operator `--`. The functionality of `++` is to add 1 unit to the operand and `--` is to subtract 1 from the operand.

For example

```
++ a;

-- b;
```

Here `++a` is equivalent to `a = a + 1` and `--b` is equivalent to `b = b - 1`.

There are two kinds of increment and decrement operator i.e **prefix and postfix.**

If the operator is used before the variable i.e `++a` then it is called **prefix increment operator.**

If the operator is used after variable i.e `a++` then it is called **postfix increment operator.**

In the prefix operator, first 1 is added and then the value is assigned to the variable.

In postfix operator, first the value is assigned then only 1 is added and the added value is assigned.

Operator	**Sample expression**	**Explanation**
++	++x	x is increased by 1, then use the value of x
++	x++	Use the current value of x and the increment x by 1
– –	– -x	x is decreased by 1, then use the value of x
– –	x- –	Use the current value of x and the decrement x by 1

Special operators in C

Besides these fundamental operators, C provides some other special operators like comma operator and sizeof operator

sizeof operator in C

`sizeof` operator is an operator which when used with operand returns the number of bytes occupied by the operand.

For example

```
x = sizeof( a );
```

Here, the size occupied by variable *a* will be assigned to *x*.

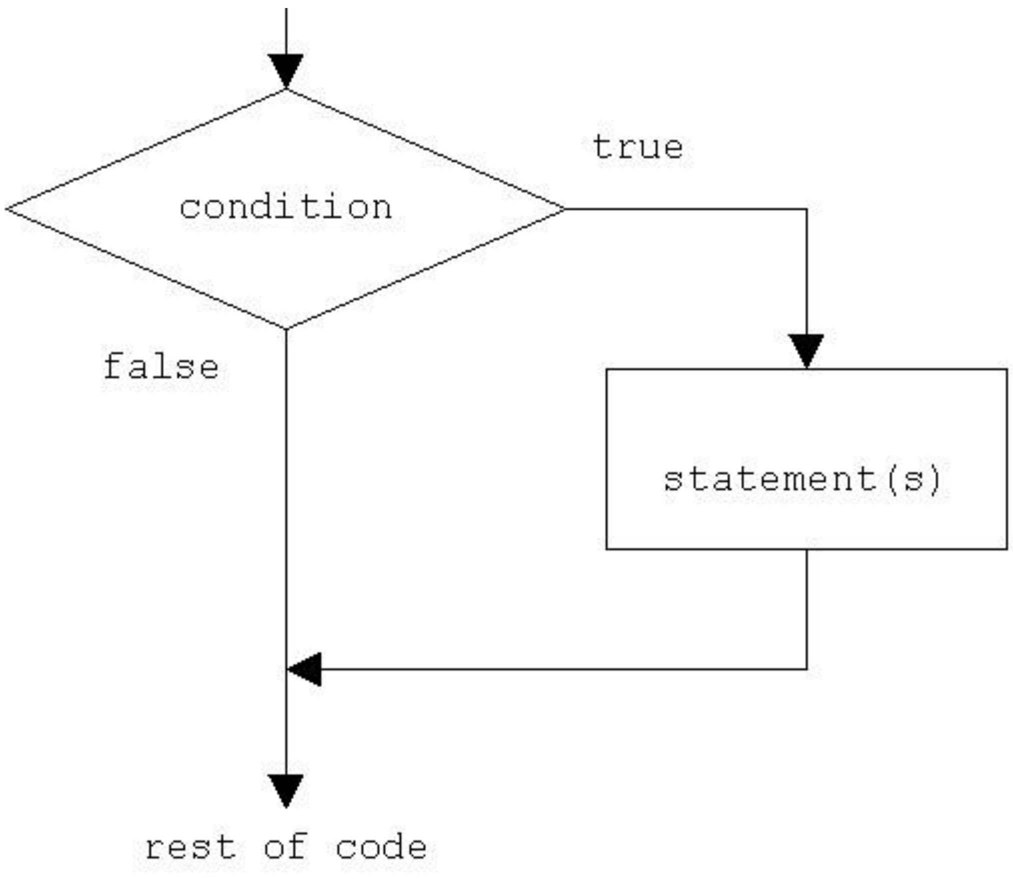
C programming if statement

It’s a one-way branching in which the statements will only execute if the given expression is true.

```
if ( expression)

 {

   statements;

 }
```

Here, statements will be executed only if the expression is true.

Flowchart of if statement



C programming if else statement

`if..else` statement is used if we want to execute some code if the condition is true and another code if the condition is false.

syntax of if..else statement

Here, if an `expression` is true then `statement1` will be executed , otherwise, `statement2` will be executed.

```
if (expression)

 {

   statement1;

 }

else

 {
   statement2;

 }
```

Here, if the `expression` is true then `statement1` will be executed otherwise, `statement2` will be executed.

Good Programming Practice

Properly indent each level of `if...else` statements. Type beginning and ending of braces and then write statements between them.

Example: if grades are less than 40 you will get failed result, otherwise you will get pass result

```
#include <stdio.h>

int main ()

 {

    int grades;

    printf("enter your grades :");

    scanf("%d",&grades);

    if ( grades >= 40 )

      printf("Congratulations you are passed");

    else

      printf("Sorry you are failed");

    return 0;

 }
```

Output

```
enter your grades : 50

Congratulations you are passed
```

if..else if...else statement

`if...else if...else` statement is used to select a particular block of code when several block of codes are there.

structure of if..else if.. else statement

```
 if(expression1)

  {

    statement1;

  }

 else if (expression2)

  {

    statement2;

  }

 else if (expression3)

  {

    statement3;

  }
```

Here, `statement1` will be executed if `expression1` is TRUE, `statement2` will be executed if `expression2` is TRUE and if none of the expressions are true then `statement3` will be executed.

Note: **Syntax error** is detected by the compiler whereas **logic error** has its effect at execution time.

While writing programs, we might need to repeat same code or task again and again.

For this C provides a feature of looping which allows a certain block of code to be executed repeatedly unless or until some sort of condition is satisfied even though the code appears once in a program.

There are three types of loops in C programming:

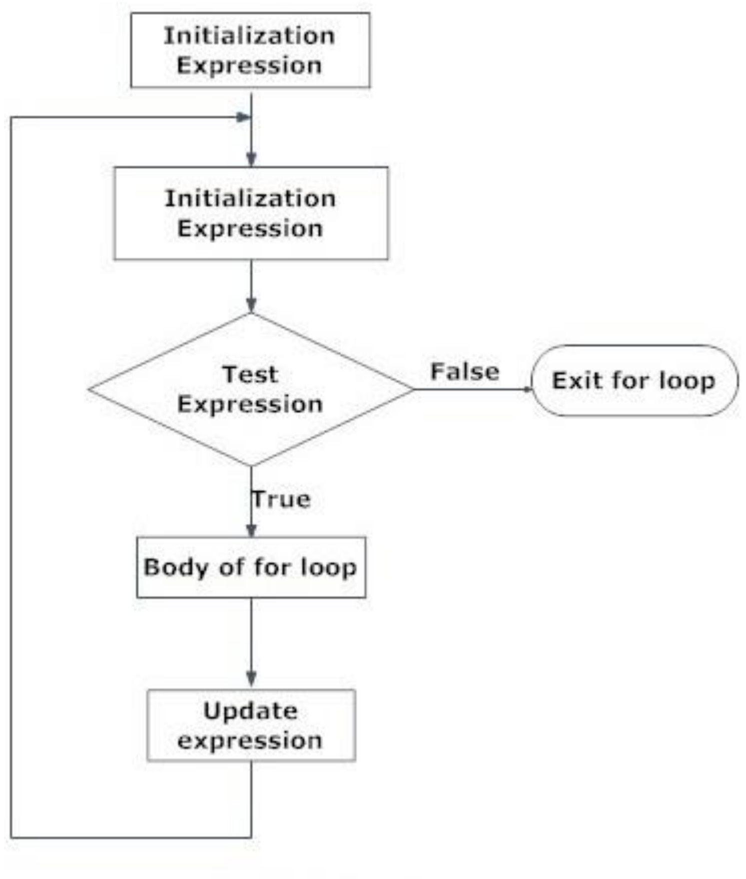
1. for loop
2. while loop
3. do … while loop

Structure of for loop in C

```
for ( statement1; statement2; statement3)

      {

        //body of for loop

      }
```

Here, `statement1` is the initialization of loop, `statement2` is the continuation of the loop. Normally, it is a test condition and `statement3` is increment or decrement of a control variable as per need.

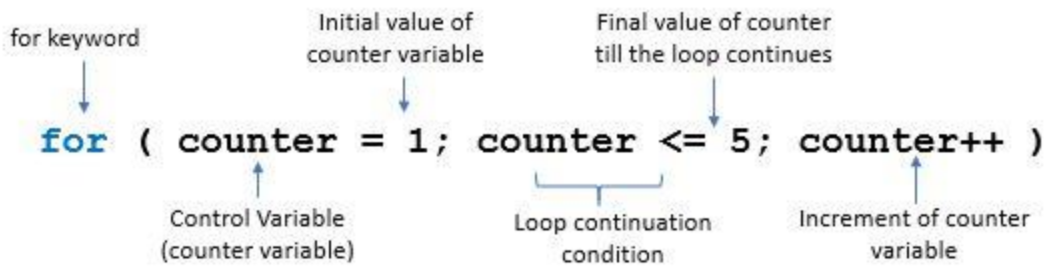
Here is the block diagram to show how a for loop works:



Common Programming Error

Using commas instead of a semicolon in the header expression of **for loop.**

Header format of for loop in C


for keyword
Initial value of counter variable
Final value of counter till the loop continues
for (counter = 1; counter <= 5; counter++)
Control Variable (counter variable)
Loop continuation condition
Increment of counter variable

How for loop works?

```
for ( counter = 1; counter <= 5; counter++)
```

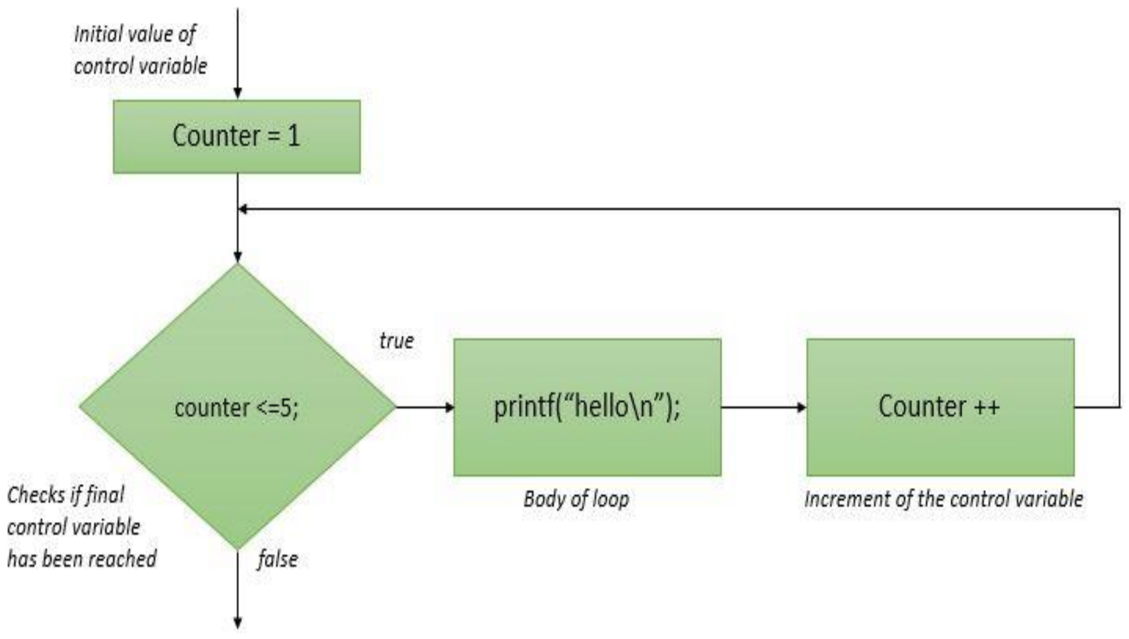
```
         {

          printf("hello\n");  //body of for loop

          }
```

In the above example, `counter` is the control variable and `counter = 1` is the first statement which initializes `counter` to 1.

The second expression, `counter <= 5` is the loop continuation condition which repeats the loop till the value of `counter` exceeds 5.

Finally, the third expression, `counter++` increases the value of `counter` by 1.

So the above piece of code will print `'hello'` five times.


Initial value of control variable
Counter = 1
counter <=5;
true
printf("hello\n");
Counter ++
Body of loop
Increment of the control variable
Checks if final control variable has been reached
false

Good Programming Practice

If you forget to write *statement1*, *statement2* and *statement 3* the loop will repeat forever. To prevent from infinite repetition write conditions carefully and don't forget increment or decrement.

Example of for loop

```
#include<stdio.h>

int main()

{

  int n;

  printf("How many time you want to print?"\n);

  scanf(" %d ", &n);

  for ( i = 1; i <= n; i++) //for loop which will repeat n times

  {

    printf("simple illustration of for loop"\n);

  }

  return 0;

}
```

Output

```
How many time you want to print?

2

simple illustration of for loop

simple illustration of for loop
```

Explanation:

Suppose the user entered value 2. Then the value of `n` will be 2.

The program will now check test expression,`'i <= n'` which is true and the program will print `'simple illustration of for loop'`.

Now the value of `'i'` is incremented by one i.e. `i = 2` and again program will check test expression, which is true and program will print `'simple illustration of for loop'`.

Again the value of `'i'` is incremented by 1 i.e. `i = 3`

This time test expression will be false and the program will terminate.

while loop in C

Structure of while statement

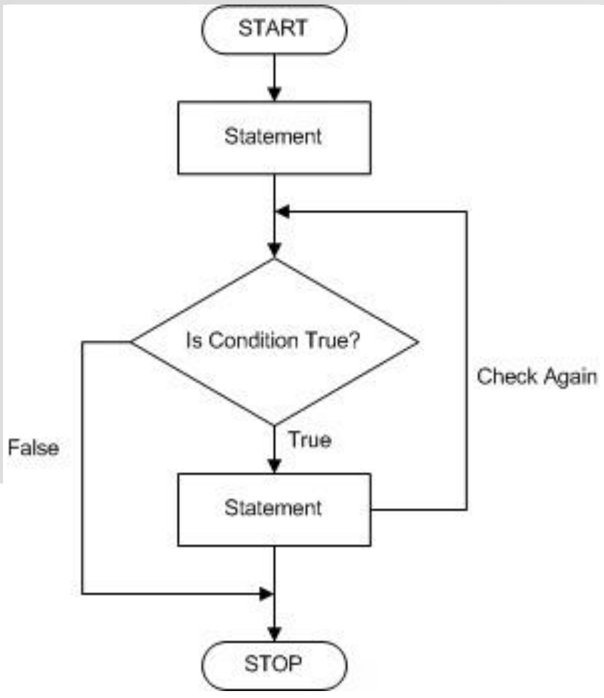
```
while (condition)

{

    //block of code to be executed

}
```

How while loops work in C then?

As shown in the above structure a condition is placed which determines how many times the code is to be repeated.

Before entering inside the `while` loop the condition is checked, and if it is true the code block inside `while`loop is executed and again after the operation condition is checked and the repetition of code is continued until the condition becomes false.

Following flowchart explains more accurately the concept of **while loop in C programming**.



Example to highlight the concept of while loop in C.

Sample Program: C program to print the sum of first 5 natural numbers

```
#include <stdio.h>

int main ()

{

  int sum = 0, i = 1;  //initialization of counter variable i

  while(i <= 5)  //loop to be repeated 5 times

  {

     sum = sum+i;

     i++;  //increment of countervariable

  }

  printf("sum of first 5 natural numbers = %d",sum);

  return 0;

}  //end of program
```

Output

```
sum of first 5 natural number = 15
```

Programming Tips

Counter variable like `i` should be initialized before while loop otherwise compiler will report an error and if you forget to increase/decrease the counter variable used in condition, the loop will repeat forever and there will not be any output.

do while loop in C

`do..while` is a variant of while loop but it is **exit controlled**, whereas, `while` loop was entry controlled.

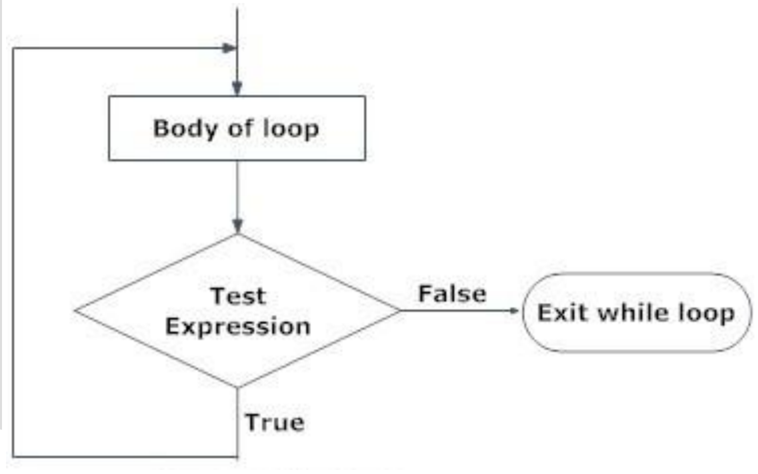
Exit controlled means unlike `while` loop in `do..while` first the code inside the loop will be executed and then the condition is checked.

In this way even if the condition is false the code inside the loop will be executed once which doesn’t happen in while.

Syntax of do while loop

```
do

         {

               //block of code to be executed

         } while (condition);
```

Flowchart of do while loop



Example: Do while loop

C program to print sum of first 5 natural numbers using do..while loop

```
#include <stdio.h>

int main ()

{

 int sum = 0, i = 1;  //initialization of counter variable i

 do

  {

   sum = sum+i;

   i++;  //increment of counter variable

  }while(i <= 5);  //coondition of do while

 printf("sum of first 5 natural numbers = %d",sum);

 return 0;

}  //end of program
```

Output

```
sum of first 5 natural numbers = 15
```

Programming Tips

Note that curly braces can be ignored if the body of loop contains one statement only and don't forget to put a semicolon at the end of while(condition); otherwise, it will be an error.

C PROGRAMMING BREAK AND CONTINUE STATEMENTS

Break and continue statements in c

Till now, we have learned about the looping with which we can repeatedly execute the code such as, for loopand while & do … while loop.

Just think what will you do when you want to jump out of the loop even if the condition is true or continue repeated execution of code skipping some of the parts?

For this C provides `break` and `continue` statements. By the help of these statements, we can jump out of loop anytime and able continue looping by skipping some part of the code.

If we feel in a condition, loop doesn't need to continue, then we use

break;
To jump outside of loop body. [Implement in Prime cheker]

It stop running the loop.

```
#include <stdio.h>
int main ()
{
    int n,a,c=0;
    scanf ("%d",&n);
    for (a=2; a<=n; a++)
    {
        if (n%a==0 )
        {
            c=1;
            break; //stop running the loop and jump out from the loop
               // Because we confirmed this is not prime number
        }
    }
    if (c=1 )
    {
        printf ("%d is a prime number", n);
    }
    else
    {
        printf ("%d is not prime", n);
    }
    return 0;
}
```

on other hand if we feel, in a conditon statement doesn't need to execute. Then we use

continue;

to skip the statement for this time only and jump to first line to the loop as usual.

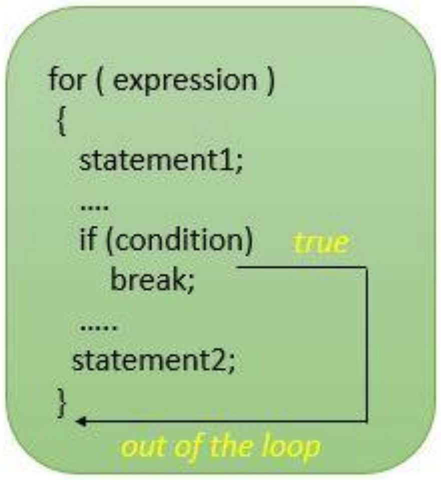
The break statement in C

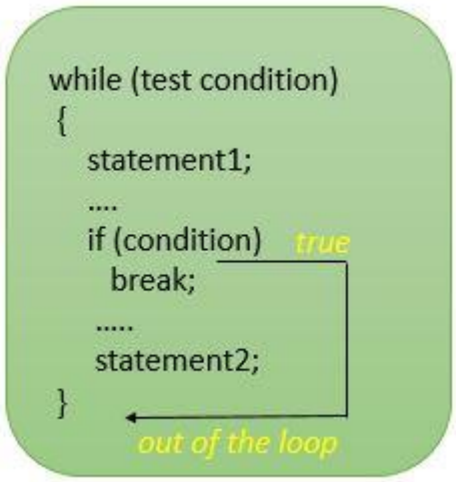
In any loop `break` is used to jump out of loop skipping the code below it without caring about the test condition.

It interrupts the flow of the program by breaking the loop and continues the execution of code which is outside the loop.

The common use of break statement is in switch case where it is used to skip remaining part of the code.

How does break statement works?





Structure of Break statement

In while loop

```
while (test_condition)

{

  statement1;
  if (condition )

     break;

  statement2;

}
```

In do…while loop

```
do

{

  statement1;

  if (condition)

     break;

  statement2;

}while (test_condition);
```

In for loop

```
for (int-exp; test-exp; update-exp)

{

  statement1;

  if (condition)

     break;

  statement2;

}
```

Now in above structure, if `test_condition` is true then the `statement1` will be executed and again if the condition is true then the program will encounter `break` statement which will cause the flow of execution to jump out of loop and `statement2` below `if` statement will be skipped.

Programming Tips

`break` statement is always used with `if` statement inside a loop and loop will be terminated whenever `break` statement is encountered.

Example: C program to take input from the user until he/she enters zero.

```
#include <stdio.h>

int main ()

{

 int a;

 while (1)

 {

   printf("enter the number:");

   scanf("%d", &a);

   if ( a == 0 )

      break;

  }

 return 0;

}
```

Explanation

In above program, `while` is an infinite loop which will be repeated forever and there is no exit from the loop.

So the program will ask for input repeatedly until the user will input 0.

When the user enters zero, the `if` condition will be true and the compiler will encounter the `break` statement which will cause the flow of execution to jump out of the loop.

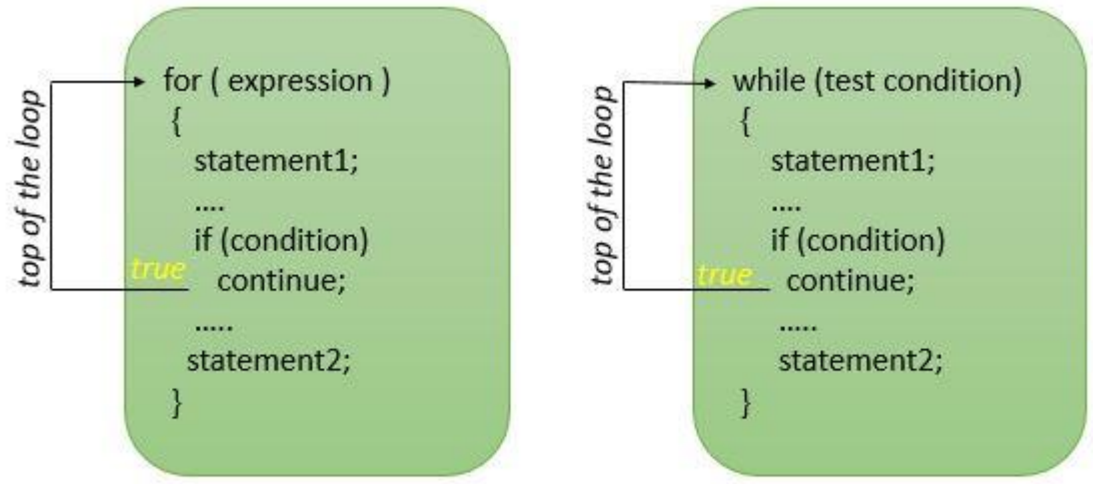
The continue statement in C

Like a `break` statement, `continue` statement is also used with `if` condition inside the loop to alter the flow of control.

When used in `while`, `for` or `do...while` loop, it skips the remaining statements in the body of that loop and performs the next iteration of the loop.

Unlike `break` statement, `continue` statement when encountered doesn't terminate the loop, rather interrupts a particular iteration.

How continue statement work?



Structure of `continue` statement

In while loop

```
while (test_condition)

{

  statement1;

  if (condition )

     continue;

  statement2;

}
```

In do...while loop

```
do

{

  statement1;

  if (condition)

     continue;

  statement2;

}while (test_condition);
```

In for loop

```
for (int-exp; test-exp; update-exp)

{

  statement1;

  if (condition)

     continue;

  statement2;

}
```

Explanation

In above structures, if `test_condition` is true then the `continue` statement will interrupt the flow of control and block of `statement2` will be skipped, however, iteration of the loop will be continued.

Example: C program to print sum of odd numbers between 0 and 10

```
#include <stdio.h>

int main ()

{
 int a,sum = 0;

 for (a = 0; a < 10; a++)

 {



   if ( a % 2 == 0 )

      continue;

  sum = sum + a;

  }

 printf("sum = %d",sum);

 return 0;

}
```

Output

```
sum = 25
```

C PROGRAMMING SWITCH CASE

Normally, we use it in Menu for using multipurposes function.

Structure of switch statement

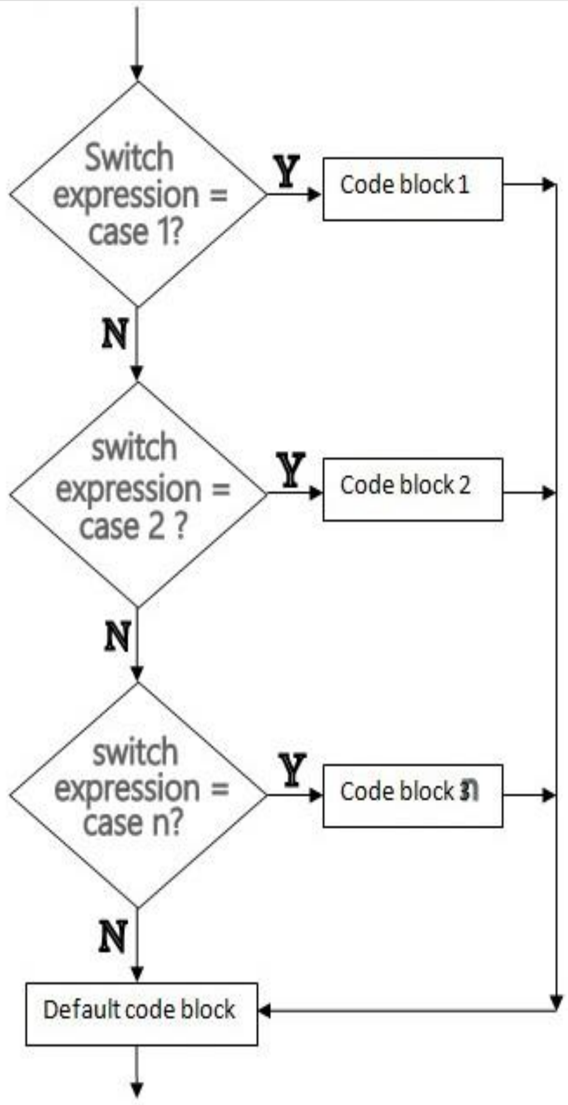
```
switch(expression)

        {

        case exp1: //note that instead of semi-colon a colon is used

               code block1;

               break;
case exp2:

        code block2;

        break;

.......

default:

        default code block;

}
```

How C programming switch case works then?

`switch` tests the value of expression against a list of case values successively. For example, in above structure, the expression is compared with all the case values. When `exp1` matches the expression code block 1 will be executed and the break statement will cause an exit from the switch statement and no further comparison will be done.

However, if `exp1` doesn't matches with expression then `exp2` will be compared and comparisons will be continued until the matching case. If no case matches with the expression the default code block will be executed.

Following block diagram explains more accurately the concept of switch case in C Programming.


Switch expression = case 1?
Y
Code block 1
N
switch expression = case 2 ?
Y
Code block 2
N
switch expression = case n?
Y
Code block n
N
Default code block

Good Programming Practice

In `switch..case` `case` and the `default` statements can occur in any order but, it is good programming practice to write `default` at the end.

Example: C program to print the day of the week according to the number of days entered

```
#include <stdio.h>



int main()

{
```

```
int n;

printf("Enter the number of the day :");

scanf(" %d ", &n);

switch (n)

{

 case 1:

    printf("Sunday");

    break;

 case 2:

    printf("Monday");

    break;

 case 3:

    printf("Tuesday");

    break;

 case 4:

    printf("Wednesday");

    break;

 case 5:

    printf("Thrusday");

    break;

 case 6:

    intf("Friday");

    break;
```

```
   case 7:

      printf("Saturday");

      break;

   default:

      printf("You entered wrong day");

      exit(0);

 }

 return 0;

}
```

Output

```
Enter the number of day : 5

Thrusday
```

Explanation:

The program asks the user to enter a value which is stored in variable *n*. Then the program will check for the respective case and print the value of the matching case. Finally, the `break` statement exit `switch` statement.

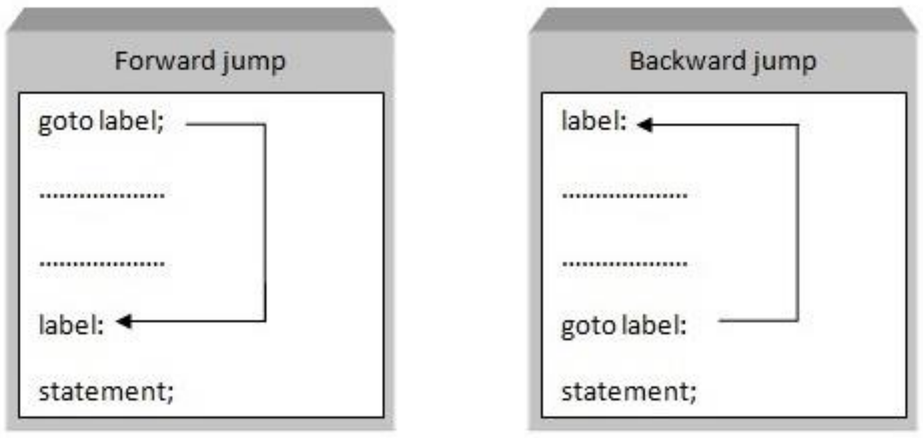
Common Programming Errors

When we forget to write `break` statement in a `switch` statement results in a logic error.

C PROGRAMMING GOTO STATEMENT

In C programming, `goto` statement is used to alter the flow of execution of the program as we wish. Use of `goto` is no recommended as **it is not a good programming practice** though it might be handy in some situations.

structure of c programming goto statement



As shown in above structure of `goto` statement label is an identifier which specifies the place where the flow is to be jumped and it must be followed by colon.

Good Programming Practice

The label used in above structure must be followed by a colon and for good programming practice try avoiding the use of `goto` statement as much as you can.

Example to illustrate the use of goto statement in C

```
#include <stdio.h>



int main ()

{

  goto a;       //instructs compiler to jump to label a

  b:

    printf("gram");

    goto c;        //instructs compiler to jump to label c

  a:
    printf("C pro");

    goto b;       //instructs compiler to jump to label b

  c:

    printf("ming");

   return 0;

}              //end of program
```

Output

```
C programming
```

Explanation of the program

As soon as the compiler encounters `goto` statement with label `'a'` the control is jumped to label `'a:'` and `'C pro'` is printed.

Then compiler encounters second `goto` statement which commands to jump to the section where label ‘b’ exist and `gram` is printed.

Similarly `ming` is printed as the compiler encounters third `goto  c` statement. As an output *‘C programming’ is printed’*.

The big problem with `goto` is that it obscure the flow of control. There are many other ways to get the job done without `goto`.

P.S: Try avoiding `goto` statement instead use function and looping.

FUNCTIONS IN C PROGRAMMING

C programming functions

**A function is a single comprehensive unit (self-contained block) containing a block of code that performs a specific task.

This means function performs the same task when called which avoids the need of rewriting the same code again and again.

Types of functions in C programming

C functions are classified into two categories:

1. Standard Library functions
2. User defined functions

Standard Library functions

Library functions are built-in standard function to perform a certain task. These functions are defined in the header file which needs to be included in the program.

Click here to learn about standard library math function.

For examples: `printf()`, `scanf()`, `gets()`, `puts()` etc… are standard library functions.

User defined functions

The user defined functions are written by a programmer at the time of writing the program. When the function is called, the execution of the program is shifted to the first statement of called function.

Good Programming Practice

Each user-defined function should perform a specific task and function name should express that task which promotes reusability.

Example: C program to calculate the area of square using function

```
/* program to calculate the area of square */

#include <stdio.h>

void area();   //function prototype

int main()     //function main begins program execution

{

  area();    //function call

  return 0;
}           // end main

void area()  //called function

{

  int square_area,square_side;

  printf("Enter the side of square :");

  scanf("%d",&square_side);

  square_area = square_side * square_side;

  printf("Area of Square = %d",square_area);

}          //end function area
```

Explanation:

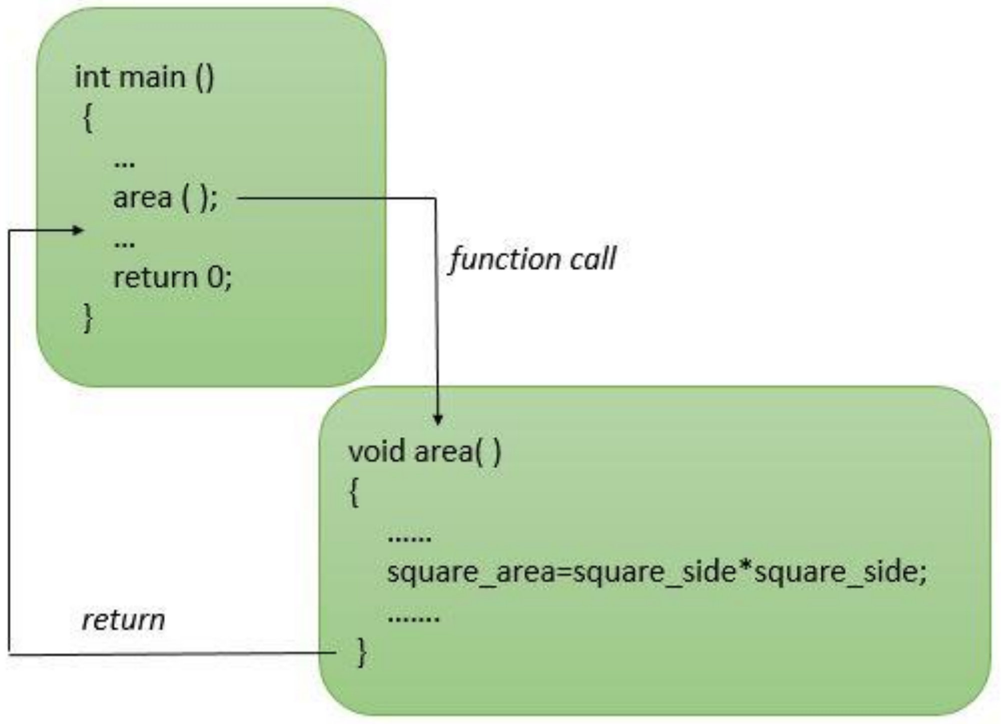
In the above program, function `area` is invoked or called in the `main` function. The execution of program now shifts to called function `area` which calculates the area of square. The `void` in funtion prototype indicates that this function does not return a value.

Syntax of a function definition

```
return_value_type function_name (parameter_list)

{

    definitions

    statements

}
```

The **function_name** is an identifier.

How does function work?



```
int main ()
{
   ...
   area ( );
   ...
   return 0;
}
```

```
void area( )
{
   ......
   square_area=square_side*square_side;
   .......
}
```

Good Programming Practice

To avoid ambiguity, we should not use the same name for functions arguments and the corresponding parameters in the function definition.

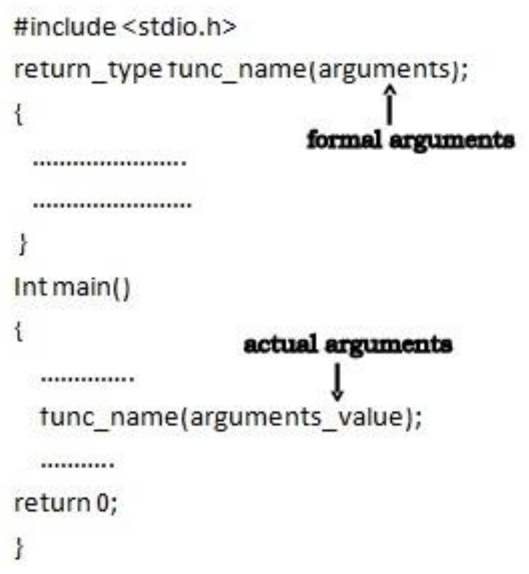
Common Programming Errors

Function definition inside another function results in a syntax error. Remember semicolon at the end of function prototype.

C PROGRAMMING FUNCTION ARGUMENTS

C programming function arguments also known as parameters are the variables that will receive the data sent by the calling program. These arguments serve as input data to the function to carry out the specified task.

Description of C programming function arguments



```
#include <stdio.h>
return_type func_name(arguments);
{
  .......................
  ........................
}
Int main()
{
   ...............
   func_name(arguments_value);
   ............
return 0;
}
```

Here, as shown in the figure above `arguments_value` is used to send values to the called program.

Function arguments in c programming

Basically, there are two types of arguments:

- **Actual arguments**
- **Formal arguments**

The variables declared in the function prototype or definition are known as **Formal arguments** and the values that are passed to the called function from the main function are known as **Actual arguments**.

The actual arguments and formal arguments must match in number, type, and order.

Following are the two ways to pass arguments to the function:

- **Pass by value**
- **Pass by reference**

Pass by Value

Pass by value is a method in which a copy of the value of the variables is passed to the function for the specific operation.

In this method, the arguments in the function call are not modified by the change in parameters of the called function. So the original variables remain unchanged.

Example of passing arguments by value to function in C

```
// arguments pass by value

# include <stdio.h>

int add (int a, int b)

{

    return( a + b );

 }



int main()

{

   int x, y, z;

   x = 5;

   y = 5;

   z = add(x,y); // call by value

return 0;

}

//end of program
```

In this program, function `add()` is called by passing the arguments *x* and *y*.

The copy of the values of `x` and `y` are passed to `a` and `b` respectively and then are used in the function.

So by changing the values of `a` and `b`, there will be no change in the actual arguments `x` and `y` in the function call.

Pass by reference

Pass by reference is a method in which rather than passing direct value the address of the variable is passed as an argument to the called function.

When we pass arguments by reference, the formal arguments in the called function becomes the assumed name or aliases of the actual arguments in the calling function. So the function works on the actual data.

Example of passing arguments by reference to function in C

```
// arguments pass by reference

 #include <stdio.h>

 void swap (int *a, int *b) // a and b are reference variables

 {

    int temp;

    temp = *a;

    *a = *b;

    *b = temp;

 }

 int main()

 {

    int x = 2, y = 4;

    printf("before swapping x = %d and y = %d\n", x, y);

    swap(&x, &y);     // call by reference

    return 0;

 } //end of program
```

In the above program, the formal arguments `a` and `b` becomes the alias of actual arguments `x` and `y` when the function was called.

So when the variables `a` and `b` are interchanged `x` and `y` are also interchanged. So the output becomes like this.

Output

```
before swapping x = 2 and y = 4

after swapping x = 4 and y = 2
```

Now, if the function was defined as:

```
void swap(int a, int b)

 {

    int temp;

    temp = a;

    a = b;

    b = temp;

 }
```

This is the pass by value method so here even if the values are swapped in the function the actual value won’t interchange and output would become like this:

```
before swapping x = 2 and y = 4

after swapping x = 4 and y = 2
```

C PROGRAMMING USER DEFINED FUNCTIONS

A function is a single comprehensive unit (self-contained block) containing a block of code that performs a specific task. In this tutorial, you will learn about **c programming user defined functions**.

C programming user defined functions

In C programming user can write their own function for doing a specific task in the program. Such type of functions in C are called **user-defined functions.**

Let us see how to write C programming user defined functions.

Example: Program that uses a function to calculate and print the area of a square.

```
#include <stdio.h>

int square(int a);   //function prototype

int main()

{

  int x, sqr;

  printf("Enter number to calculate square: ");

  scanf("%d", &x);

  sqr = square (x);    //function call

  printf("Square = %d", sqr);

  return 0;

}                      //end main



int square (int a)     //function definition

{

  int s;

  s = a*a;

  return s;  //returns the square value s

}             //end function square
```

Elements of user-defined function in C programming

There are multiple parts of user defined function that must be established in order to make use of such function.

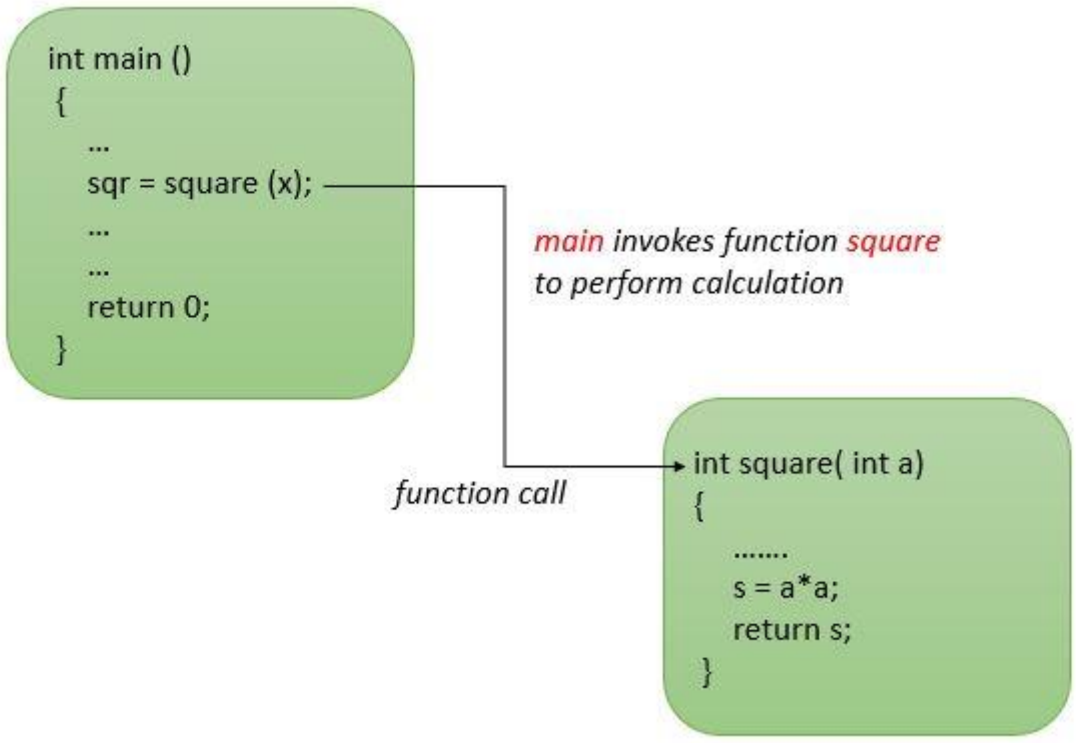
- **Function declaration or prototype**
- **Function call**
- **Function definition**

- **Return statement**

Function Call

Here, function `square` is called in `main`

```
sqr = square (x);    //function call
```



Function declaration or prototype

```
int square(int a);    //function prototype
```

Here, `int` before function name indicates that this function returns **integer value** to the caller while `int`inside parentheses indicates that this function will recieve an integer value from caller.

Function definition

A function definition provides the actual body of the function.

Syntax of function definition

```
return_value_type function_name (parameter_list)

{

   // body of the function

}
```

It consists of a function header and a function body. The `function_name` is an identifier.

The `return_value_type` is the data type of value which will be returned to a caller.

Some functions performs the desired task without returning a value which is indicated by `void` as a `return_value_type`.

All definitions and statements are written inside the body of the function.

Good Programming Practice

To avoid ambiguity, we should not use the same name for functions arguments and the corresponding parameters in the function definition.

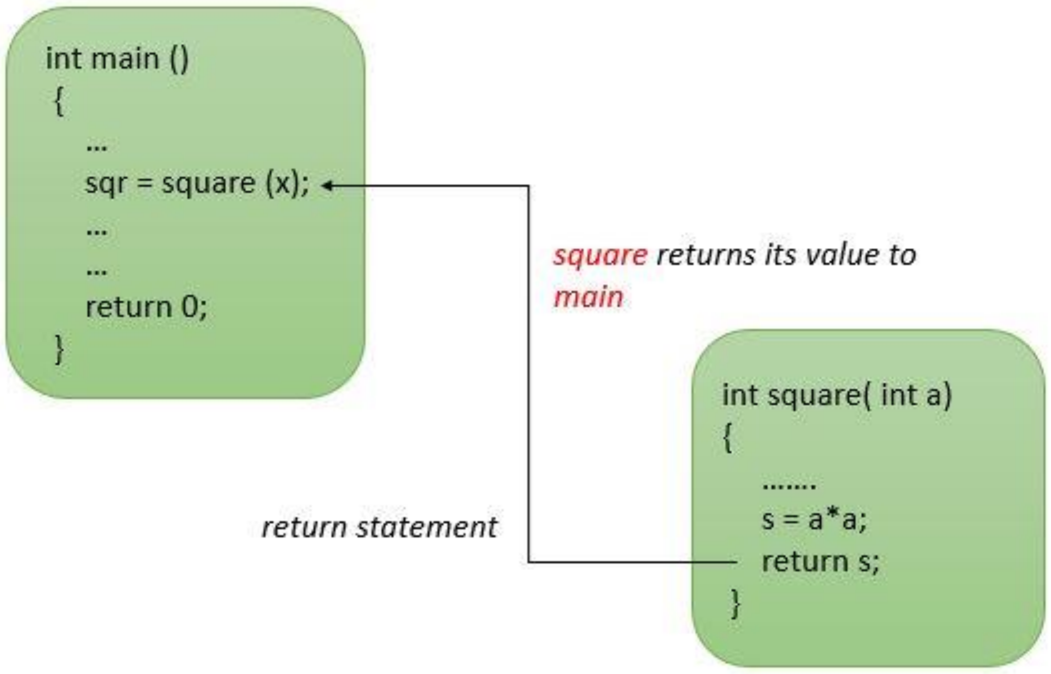
Common Programming Errors

Function definition inside another function results in a syntax error. Remember semicolon at the end of function prototype.

Return statement

Return statement returns the value and transfer control to the caller.

```
return s;  //returns the square value s
```



```
int main ()
{
    ...
    sqr = square (x);
    ...
    ...
    return 0;
}
```

```
int square( int a)
{
    .......
    s = a*a;
    return s;
}
```

There are three ways to return control.

```
return;
```

The above `return` statement does not return value to the caller.

```
return expression;
```

The above `return` statement returns the value of expression to the caller.

```
return 0;
```

The above `return` statement indicate whether the program executed correctly.

Types of user defined functions in C

Depending upon the presence of arguments and the return values, user defined functions can be classified into five categories.

1. **Function with no arguments and no return values**
2. **Function with no arguments and one return value**
3. **Function with arguments and no return values**
4. **Function with arguments and one return value**
5. **Function with multiple return values**

1: Function with no arguments and no return value

Function with no argument means the called function does not receive any data from calling function and **Function with no return value** means calling function does not receive any data from the called function. So there is no data transfer between calling and called function.

C program to calculate the area of square using the function with no arguments and no return values

```
/* program to calculate the area of square */

#include <stdio.h>

void area();   //function prototype

int main()

{

  area();    //function call

  return 0;

}

void area()

{

  int square_area,square_side;

  printf("Enter the side of square :");

  scanf("%d",&square_side);

  square_area = square_side * square_side;

  printf("Area of Square = %d",square_area);

}
```

Explanation

In the above program, `area( );` function calculates area and no arguments are passed to this function. The return type of this function is `void` and hence return nothing.

2: Function with no arguments and one return value

As said earlier function with no arguments means called function does not receive any data from calling function and function with one return value means one result will be sent back to the caller from the function.

C program to calculate the area of square using the function with no arguments and one return values

```
#include <stdio.h>

int area();   //function prototype with return type int

int main()

{

  int square_area;

  square_area = area();    //function call

  printf("Area of Square = %d",square_area);

  return 0;

}

int area()

{

  int square_area,square_side;

  printf("Enter the side of square :");

  scanf("%d",&square_side);

  square_area = square_side * square_side;

  return square_area;
}
```

Explanation
In this function `int area( );` no arguments are passed but it returns an integer value `square_area`.

3: Function with arguments and no return values

Here function will accept data from the calling function as there are arguments, however, since there is no return type nothing will be returned to the calling program. So it's a one-way type communication.

C program to calculate the area of square using the function with arguments and no return values

```
#include <stdio.h>

void area( int square_side);  //function prototype

int main()

{

  int square_side;

  printf("Enter the side of square :");

  scanf("%d",&square_side);

  area(square_side);   //function call

  return 0;

}

void area(int square_side)

{

  int square_area;

  square_area = square_side * square_side;

  printf("Area of Square = %d",square_area);
}
```

Explanation
In this function, the integer value entered by the user in *square_side* variable is passed to `area();` .The called function has `void` as a return type as a result, it does not return value.

This program calculates the area of a square and prints the result.

4: Function with arguments and one return value

Function with arguments and one return value means both the calling function and called function will receive data from each other. It's like a dual communication.

C program to calculate the area of square using the function with arguments and one return values

```
#include <stdio.h>

int area(int square_side);   //function prototype with return type int

int main()

{

  int square_area,square_side;

  printf("Enter the side of square :");

  scanf("%d",&square_side);

  square_area = area(square_side);    //function call

  printf("Area of Square = %d",square_area);

  return 0;

}

int area(int square_side)

{

  int square_area;
  square_area = square_side * square_side;

  return square_area;

}
```

5: Function with multiple return values

So far we have used functions that return only one value because function normally returns a single value. However, we can use functions which can return multiple values by using input parameters and output parameters. Those parameters which are used to receive data are called **input parameters** and the parameters used to send data are called **output parameters**. This is achieved by using **address operator(&)** and **indirection operator(*)**. Moreover, following example will clarify this concept.

C program to calculate the area and volume of square using the function with multiple return values

```
#include <stdio.h>

void area_volume(int l, int *a, int *v);   //function prototype

int main()

{

  int l,a,v;

  printf("Enter the side of square :");

  scanf("%d",&l);

  area_volume(l,&a,&v);    //function call

  printf("Area = %d\n Volume = %d",a,v);

  return 0;

}

void area_volume(int l, int *a, int *v)

{
 *a = l*l;

 *v = l*l*l;

}
```

Explanation

In the above program *l* is input argument, *a* and *v* are output arguments. In the function call, we pass actual value of *l* whereas addresses of *a* and *v* are passed.

C PROGRAMMING RECURSION

C programming recursive functions

Until now, we have used multiple functions that call each other but in some case, it is useful to have functions that call themselves. In C, such function which calls itself is called recursive function and the process is called recursion.

For example:

```
main ()

{

  printf("Recursion \n");

  main();

}
```

In this above example, `main` function is again called from inside the `main` function. So this function will keep on printing **Recursion** until the program run out of memory.

Example of recursive function: C program to find the factorial of first 3 natural numbers using recursion

```
#include <stdio.h>

long int fact( int n )
```

```
 {

     if ( n <= 1 )

        return 1;

     else      //recursive step

        return ( n * fact (n-1) );

 }    //end factorial

int main ()

 {

     int i;

     for ( i = 1; i <=3; i++ )

        printf("%d! = %d\n",i, fact(i) );

     return 0;

}
```

Output

```
1! = 1

2! = 2

3! = 6
```

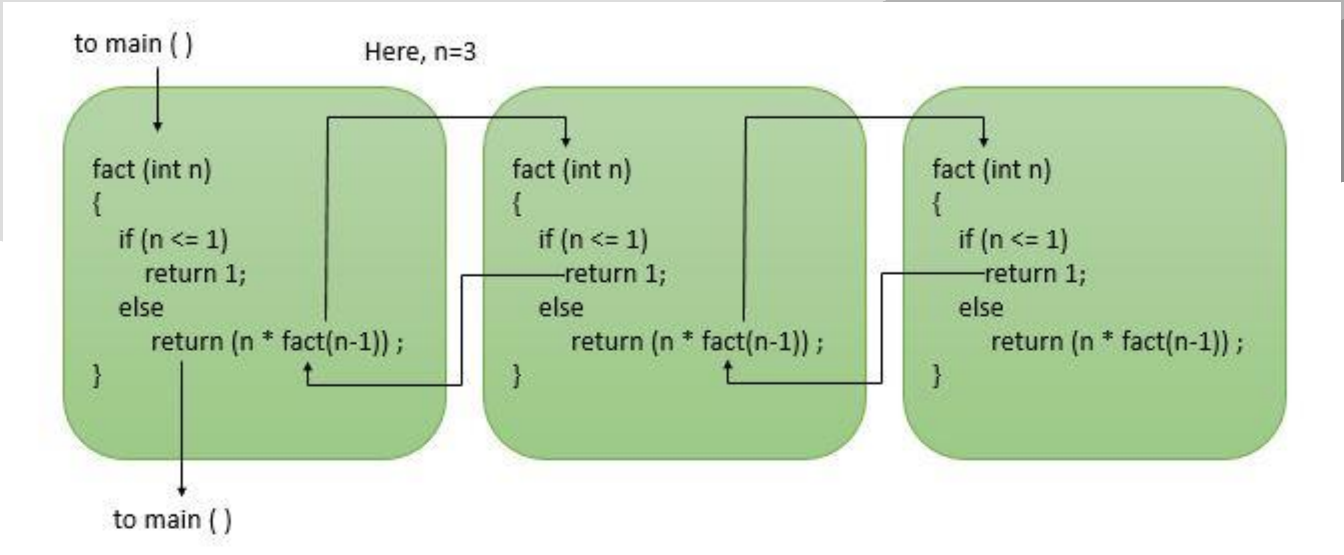
Explanation of output

```
when i = 1 | fact (1) : 1 * fact(0) = 1*1 = 1

when i = 2 | fact (2) : 2 * fact(1) = 2*1 = 2

when i = 3 | fact (3) : 3 * fact(2) = 3*2 = 6
```

Explanation of the program

First of all, the recursive factorial function checks whether `if` condition is true or not i.e whether n is less than or equal to 1. If condition is true, factorial returns 1 and the program terminates otherwise the following statement executes.

```
return ( n * fact (n-1) );
```



Common Programming Errors

Forgetting the base case or incorrect recursion step that does not converge will cause infinite recursion.

It is mandatory to keep a base case for stop recursion. Otherwise it called infinite recursion.

Difference between recursion and iteration

- Iteration uses a repetition statement whereas recursion does repetition through repeated function calls.
- Iteration terminates when loop condition fails whereas recursion terminates when the base case became true.

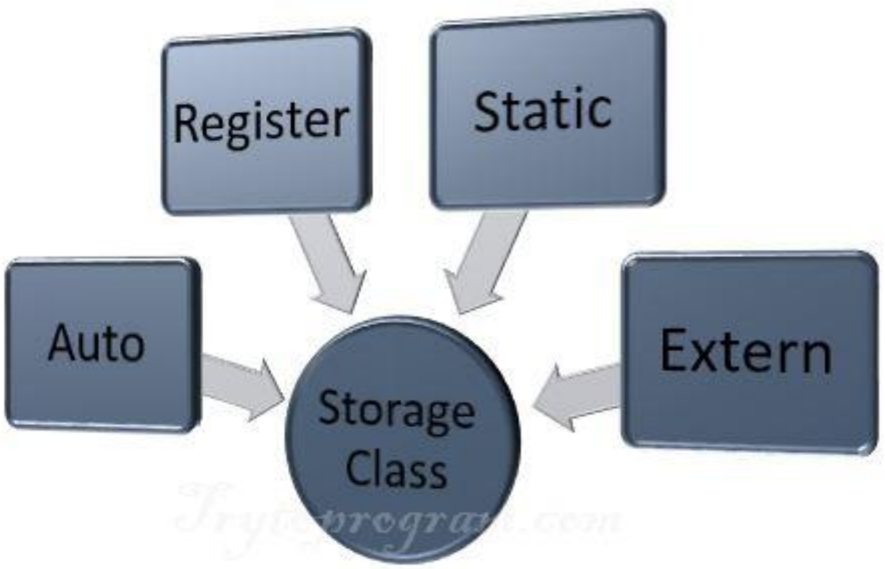
C programming storage class

In C programming, properties of variables include `variable_name`, `variable_type`, `variable_size` and `variable_value`.

We already know that variable in an identifier. Identifier has other properties such as storage class, storage duration, scope and linkage.

In C, there are four types of storage class:

1. **Auto**
2. **Register**
3. **Static**
4. **Extern**



Storage class of variable in C determines following things:

- **Lifetime of the variable** i.e. time period during which variable exist in computer memory.
- **Scope of the variable** i.e. availability of variable value.

Auto Storage Class/Local Variables

Local variables are declared within function body and have automatic storage duration.

For representing automatic storage duration keyword `auto` is used.

Memory for the local variables is created when the function is invoked or active, and destroyed or de-allocated when a block is exited or control moves out of the function.

By default, local variables have automatic storage duration.

```
#include <stdio.h>

int main ()

{

  int x;    //local variable

  ....

  return 0;

}



int func_name ()

{

  int y;   //local variable

  ....

}
```

In the above program, both `x` and `y` are local variables but `x` is only available to `main` function whereas `y`is only available to `func_name` function.

Register Storage Class

Register variables are stored in CPU registers and are available within the declared function.

The lifetime of register variable remains only when control is within the block.

Keyword `register` is used to define register storage class. Register variables are accessed faster because it is stored in CPU register.

```
#include <stdio.h>

int main()

{

   register int x;   //register variable

   for (x = 1; x <= 5; x++)

   {

      printf("\n%d", x);

   }

}
```

Static Storage Class

Static variables are defined by keyword `static`. For static variables, memory is allocated only once and storage duration remains until the program terminates.

How do static and auto variables works?

Auto

```
#include <stdio.h>

int main()

{

  increment();

  increment();

  increment();

  return 0;

}

void increment()
```

```
{

   auto int x = 1;

   printf("%d\n", x);

   x = x + 1;

}
```

Static

```
#include <stdio.h>

int main()

{

  increment();

  increment();

  increment();

  return 0;

}

void increment()

{

   static int x = 1;

   printf("%d\n", x);

   x = x + 1;

}
```

Explanation
In the above programs, `increment()` function is called three times from the `main`.

The only difference is the storage class of variable *x*.

Like `auto` variables, `static` variables are local to the function in which they are declared but the difference is that the storage duration of `static` variables remains until the end of the program as mentioned earlier.

External Storage Class

External variables are declared outside the body of the function. If they are declared in the global declaration section, it can be used by all the functions in the program.

```
#include <stdio.h>

int x = 5;  //global variable

int main ()

{

    ......

    ......

    return 0;

}
```

Note the difference in following

```
extern int x;

int x = 5;
```

In the above example, the first statement is the declaration of the variable and the second statement is the definition.

Note: Automatic storage helps in preserving memory because they are created when the function is started and destroyed when the function is exited.

Arrays in C programming

An **array** is a collection of same types of data items that are placed in the contiguous memory location on computers memory.

**Same name multiple data with a sequential serial

When we need to handle the multiple data of similar types arrays are used.

For example, we need 10 unique variables for 10 data items of integer type which make program bulky and tedious but with an array, we can create a single pack of 10 integer data items.

For example:

```
int num[10];     //num is a variable that will hold 10 integer data items
```

Basically, there are two **types of arrays**:

- One dimensional array
- Multidimensional array

Syntax of one-dimensional array

```
data_type variable_name[array_size];  //declaring an one dimensional
```

In above syntax, the **“data_type”** as the name suggests defines the types of data items that the array will be holding,**“variable_name”** defines the unique name of an array and the **“array_size”** defines the number of elements contained in the array.

Good Programming Practice

The elements field within brackets [ ] which represent the number of array elements that are going to be held by array must be a constant value. This field can’t be left blank while declaring array because array size must be specified before execution.

Array and Memory

Once the array is declared it must be initialized otherwise it will contain a garbage value.

General form of initialization an array

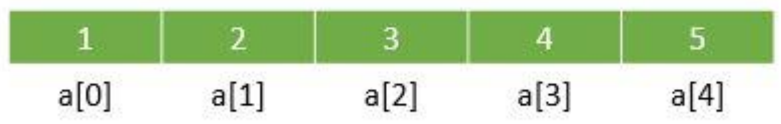
```
data_type variable_name[size] = { list };  //initializing an array
```

The values in the list are separated by commas.

For example:

```
int a[]={1,2,3,4,5};
```

This declaration will create an array like this:



Common Programming Errors

The number of elements inside braces { } must not be larger than the number of elements specified inside braces [ ] while declaring arrays.

Or if we insert size of an array then the number of elements must not exceed the size.

```
int a[5]={0};  //this will assign 0 to all 5 elements
```

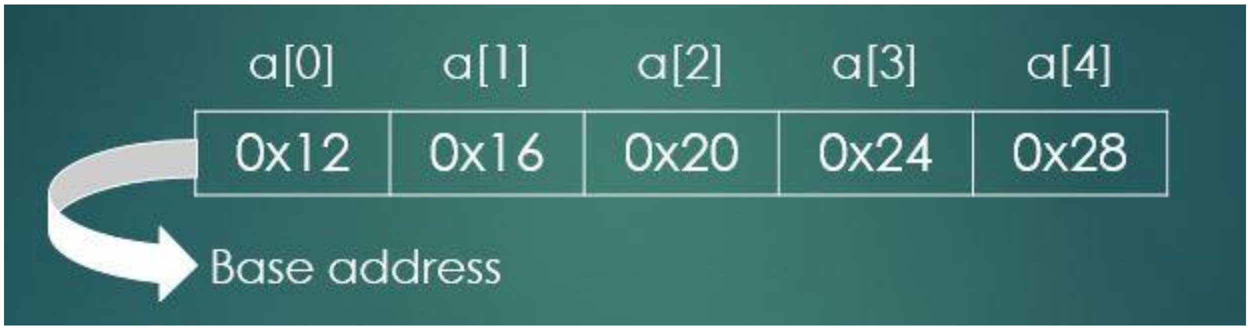
To initialize character arrays

```
char a[]={'a','b','c','d'};
```

Normally, when we declare multiple variables they occupy memory and are randomly placed in computers memory heap, however, when we declare an array of multiple data items these items occupy the continuous memory locations in computers memory heap.

For example, **take an integral constant which takes 4 bytes of computers memory.**

Now, if the first element of an array `a[0]` is stored at memory location 0x12 (say) then the second element `a[1]` of an array will be stored at the memory location adjacent to it which is 0x16 as integer occupies 4 bytes of memory.



Initializing an array with a symbolic constant

```
#include <stdio.h>

#define SIZE 5    //sizi of array

int main()

{

  int a[SIZE] = {1,2,3,4,5};    //array a has SIZE elements

  ....

}
```

Here, `#define` preprocessor directive defines a symbolic constant *SIZE* whose value is 5.

Common Programming Errors

Putting semicolon at the end of `#define` and `#include` because preprocessor directive are not C statements.

Good Programming Practice

Always use uppercase letters for naming symbolic constant in an array.

Array example: C program to print sum of first 5 numbers entered by the user

```
#include <stdio.h>

#include <stdio.h>

int main ()

{

 int i, a[5] ,sum = 0;  //a[5] is array that can hold 5 integer data items

 for(i=0;i<5;i++)  //loop to be repeated 5 times

  {

   printf("enter the number "<< i+1<<":");
   scanf("%d", &a[i]);

   sum = sum + a[i];

  }

 printf("sum = %d",sum);

 return 0;

}  //end of program
```

Output

```
enter the number 1 : 1

enter the number 2 : 2

enter the number 3 : 3

enter the number 4 : 4

enter the number 5 : 5

sum = 15
```

Summary

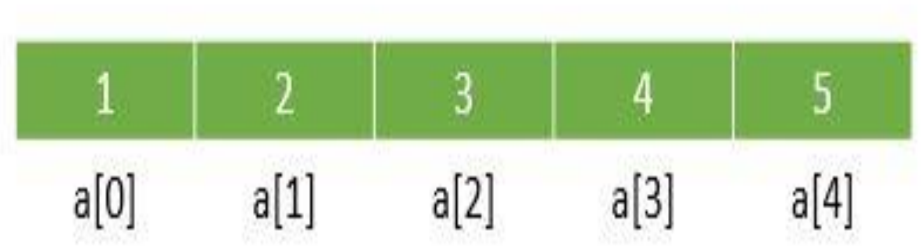
- An array is similar to the variable except that it can store multiple elements of similar data type.
- Array elements are stored in contiguous memory locations and can be accessed using pointers.
- Array variable acts as a pointer to the zeroth element of the array.
- When the pointer is increased it points to next location of its type.

C programming multidimensional arrays

In one-dimensional arrays, elements are arranged in one direction but in multidimensional array data items are arranged in multiple directions.

Like in **two-dimensional array** data items are arranged in two directions i.e rows and columns. Data items arranged in the horizontal direction are called rows and that in vertical directions are referred as columns.

Moreover, the following picture explains the concept of a **C programming multidimensional array**.



An array is a sequenced collection of similar kinds of data items that can be represented by single variable name. The individual data item in an array is called **element**.

Initializing a two-dimensional array

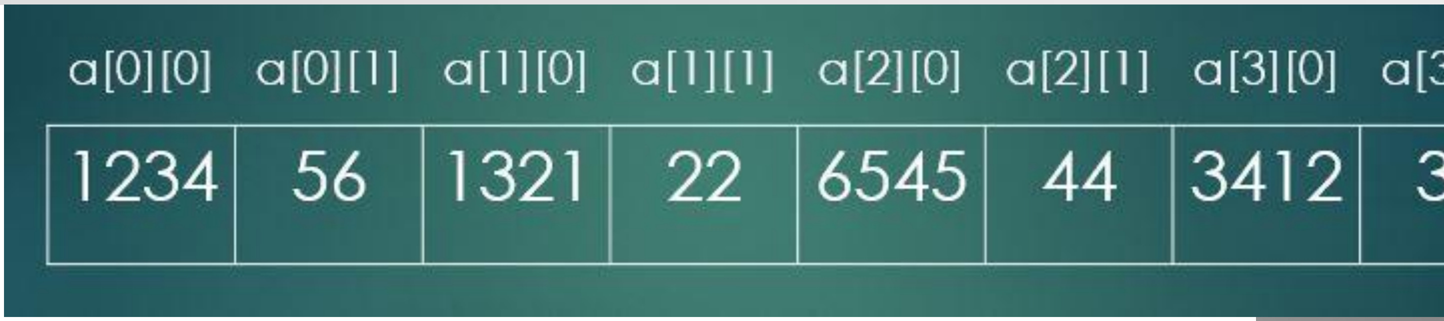
How do we initialize a two-dimensional array?

```
data_type array_name[row_size][columns_size];  //declaring two dimensional array
```

Example:

```
int a[4][2] = {

               {1234, 56}

               {1321, 22}

               {6545, 44}

               {3412, 31}

               };

int a[4][2] = {1234, 56, 1321, 22, 6545, 44, 3412, 31};
```

Memory map of a two-dimensional array

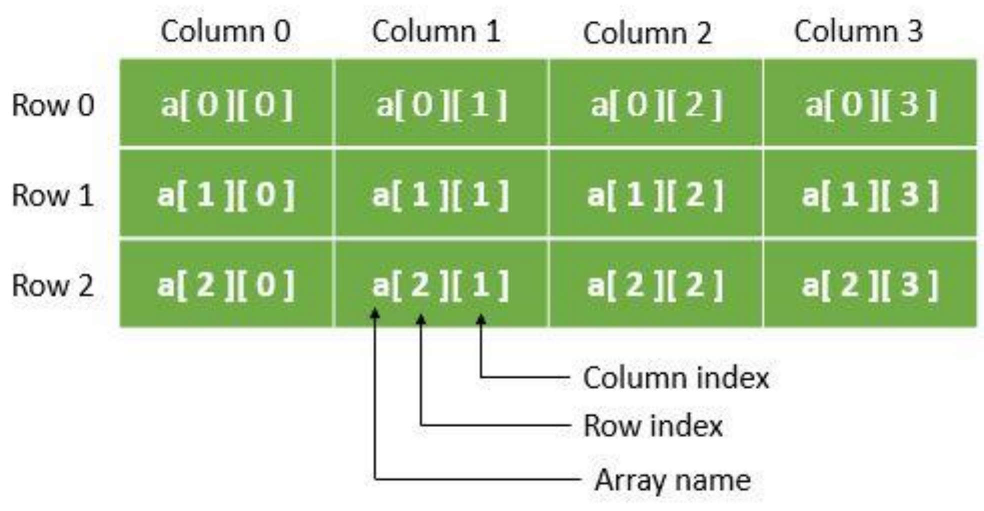


Initializing a multidimensional array

```
data_type array_name[s1][s2][s3]....[sn];;  //declaring multi dimensional array
```

Example:

```
int a[3][3][2] = {

                    {{1,2},{2,3},{3,4}},

                    {{4,5},{5,6},{6,7}},

                    {{7,8},{8,9},{9,0}}

                  };
```



Common Programming Error

Defining double array elements as `a[x,y]` instead of `a[x][y]` is programmer error rather than syntax error because the comma is treated as an operator.

Example: Initializing C programming multidimensional arrays.

```
#include <stdio.h>

void showArray(int x[][3]);  //function prototype

int main ()

{

   int a[2][3] = {{1,2,3}, {4,5,6}};   //initializing array

   int b[2][3] = {1, 2, 3, 4, 5};

   int c[2][3] = {{1, 2}, {4}};

   printf("values in array a by row:\n");

   showArray(a);



   printf("values in array b by row:\n");

   showArray(b);



   printf("values in array c by row:\n");

   showArray(c);

   return 0;

}             // end of main

void showArray(int x[][3])

{

   int i;      //row counter

   int j;      //column counter
   for (i = 0; i <= 1; ++i)

   {

      for (j = 0; j<= 2; ++j)

      {

         printf("%d", x[i][j]);

      }       //end of inner for

      printf("\n");

   }          //end of outer for

}             //end of function
```

Output

```
values in array a by row:

123

456

values in array b by row:

123

450

values in array c by row:

120

400
```

Explanation
There are three arrays in this program. The elements of an array that are not initialized explicitly are initialized zero, as in the second and third array of the above program.

The subscripts of every array are used by the compiler to determine the location of array elements in the memory.

Like other values of variables, arrays can be passed to a function. Let us see how to **pass an entire array to a function**. Both one-dimensional arrays and multidimensional arrays can be passed as function arguments.

Passing one-dimensional array to the function

While passing one-dimensional array to the function name of the array is passed as actual arguments and array variable with subscript is passed as formal arguments.

Ways of passing arrays as formal arguments.

1: Passing formal parameter as pointer

```
void func_name( int *name )

 {

     .......

 }
```

2: Passing formal parameters as sized array

```
void func_name( int name[10] )

 {

     .......

 }
```

3: Passing formal parameters as unsized array

```
void func_name( int name[] )

 {

     .......

 }
```

Example: C program to pass an array to the function that contains marks obtained by the student. Then display the total marks obtained by the student

```
#include <stdio.h>

int total_marks(int a[]);

int main()

{

   int total = 0, marks[]={40,80,75,90,88};

   printf("Total marks = %d",total_marks(marks));

   return 0;

}

int total_marks(int a[])

{

   int sum=0,i;

   for (i=0;i<5;i++)

       sum=sum+a[i];

   return sum;

}
```

Output

```
Total marks = 373
```

Explanation:
Here, the `total_marks( )` function is used to calculate total marks. The `for` loop is used to access the array elements.

Passing Multidimensional array to the function

Like simple arrays, multidimensional arrays can be passed to the function. Two-dimensional array is indicated by two sets of brackets with array variable. The size of second dimension must be specified.

Example: C program to display the 3*3 matrix of given elements.

```
#include <stdio.h>

void matrix( int m[][3] )

{

   int i,j;

   printf("Matrix is :\n");

   for ( i=0;i<3;i++ )

   {

      for ( j=0;j<3;j++)

      {

         printf("%d",m[i][j]);

       }

      printf("\n");

   }

}

int main()

{

   int m[3][3] = {1,2,3,4,5,6,7,8,9};

   matrix(m);

   return 0;

}
```

Output

```
Matrix is :
```

```
1 2 3

4 5 6

7 8 9
```

Note: It is a good idea to pass arrays by reference rather than value for performance reason. In the case of large arrays, passing by value will consume higher storage.

Difference between passing an entire array and individual array elements to a function.

```
#include <stdio.h>

#define SIZE 5



void changeArray( int x[], int s);

void changeElement( int y );



int main ()

{

    int arr[SIZE] = {0, 1, 2, 3, 4};

    int i;   //counter

    printf("passing entire array by reference.\n");

    printf("values of original array:\n");

    for(i = 0; i < SIZE; i++)

    {

        printf("%3d", arr[i]);

    }

    changeArray(arr, SIZE);   //passing array by reference
    printf("\nvalues of changed array:\n");

    for(i = 0; i < SIZE; i++)

    {

        printf("%3d", arr[i]);

    }

    printf("\nlet's see the value by passing array element\n"

           "\nvalue of arr[3] = %d", arr[3]);

    changeElement(arr[3]);  //passing array element arr[3] by value



    printf("\nvalue of arr[3] = %d", arr[3]);



}



void changeArray(int x[], int s)

{

    int j;    //counter variable

    for (j = 0; j<= SIZE; j++)

    {

        x[j] = x[j] * 2;

    }

}



void changeElement( int y)
```

```
{

   y = y * 2;    //multiplying array element by 2

   printf("\nvalue of 3rd element in changeElement = %d ", y);

}
```

Output

```
passing entire array by reference.

values of original array:

  0   1   2   3   4

values of changed array:

  0   2   4   6   8

let's see the value by passing array element



value of arr[3] = 6

value of 3rd element in changeElement = 12

value of arr[3] = 6
```

Explanation
In the above program, we can see the difference between passing an array by reference and element by value.

The program first prints the value of array and it is passed to `changeArray()` function, where elements are multiplied by 2. The modified result is printed.

In the second part, the program first prints the third element of modified array. After this third element of an array is passed to `changeElement()` function by value, where it is multiplied by 2 and value is printed.

Finally, in the `main` function the value of third element of an array is printed again but the value remains same.

Note: We can use `const` (constant) type qualifier to prevent modification of array elements in a function.

Pointers in C programming

**Pointer is a variable of RAM memory address of a variable

Pointer is a striking and most powerful feature of C programming. As the name itself suggests a pointer is something that points something.

It can also be described as a variable that points to an ordinary variable of any type by holding the address of that variable.

For the better understanding of C programming pointers consider following declaration:

```
int x = 1;
```

Above declaration reserves a memory block with name `x` and its value 1.

Assume that `x` is located at address 3000.

Therefore: `&x =` address of `x = 3000`

To print the value and the address the program would be

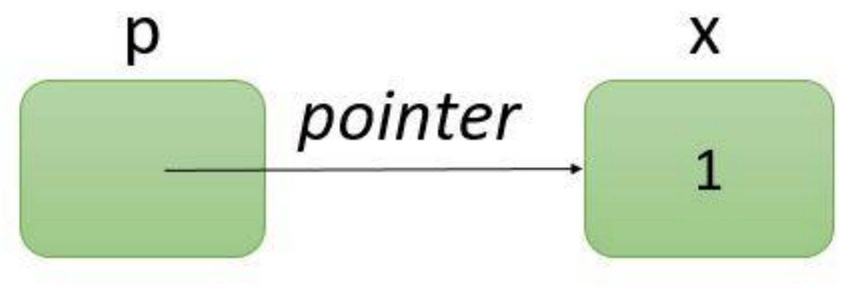
```
#include <stdio.h>

int main()

{

    int x = 1;

    printf("Value of x = %d",x);

    printf("Address of x = %d",&x);

    return 0;

}
```

Output

```
Value of x = 1

Address of x = 3000
```

In above program, the address of `x` can be assigned to another variable `p` as shown below:

```
int x = 1;

int *p; // pointer variable declaration

p = &x;
```



So the pointer variable is declared as:

```
data_type *variable_name;
```

Good Programming Practice

It is a good idea to include `"ptr"` in pointer variable name to distinguish easily.

How to access the value of a variable using pointer variable?

So far we know that pointer variable stores the address of the variable. However, the value of the variable can also be accessed by pointer variable using the syntax:

```
data_type *variable_name ;
```

Common Programming Error

Pointer needs to be handled carefully. If the pointer is not initialized properly dereferencing it cause an error.

For example, if `p` is the pointer variable that points to the address of the variable `x` then `*p` gives the value of variable `x`. The following program will clarify the concept of C programming pointers.

```
#include <stdio.h>

int main()

{

    int x, y, z, *p1, *p2;

    x = 2;

    y = 3;

    p1 = &x;

    p2 = &y;

    z = *p1 + *p2;

    printf("sum = %d",z);

    return 0;

}
```

Output

```
sum = 5
```

Explanation

In the above program, pointers `p1` and `p2` point to the variable `x` and `y`. Dereference operator is used for extracting the value from that memory location.

Reference operator (&) and Dereference operator (*)

The refrence operator `&` returns the address of the variable.

The dereference operator `*` or indirection operator returns the value from the address.

Example to demonstrate the use of & and * pointer operators.

```
#include <stdio.h>

int main()

{

  int x;     //integer variable

  int *y;    //y is pointer to an integer



  x = 5;

  y = &x;    //pointing to address of x



  printf("The address of x = %p", &x);

  printf("\nThe value of y = %p", y);



  printf("\n\nThe value of x = %d", x);

  printf("\nThe value of *y = %d", *y);



  printf("\n\n* and & are complement of each other"

         "\n&*y = %p"

         "\n*&y = %p\n", &*y, *&y );

  return 0;

}
```

Output

```
The address of x = 0060FF0C
The value of y = 0060FF0C



The value of x = 5

The value of *y = 5



* and & are complement of each other

&*y = 0060FF0C

*&y = 0060FF0C
```

`%p` outputs the memory location in hexadecimal.

The value of *y* and address of *x* are same in the output which confirms that the address of *x* is assigned to pointer variable *y*.

```
printf("\n\n* and & are complement of each other"

         "\n&*y = %p"

         "\n*&y = %p\n", &*y, *&y );
```

This proves that `&` and `*` are complements of each other.

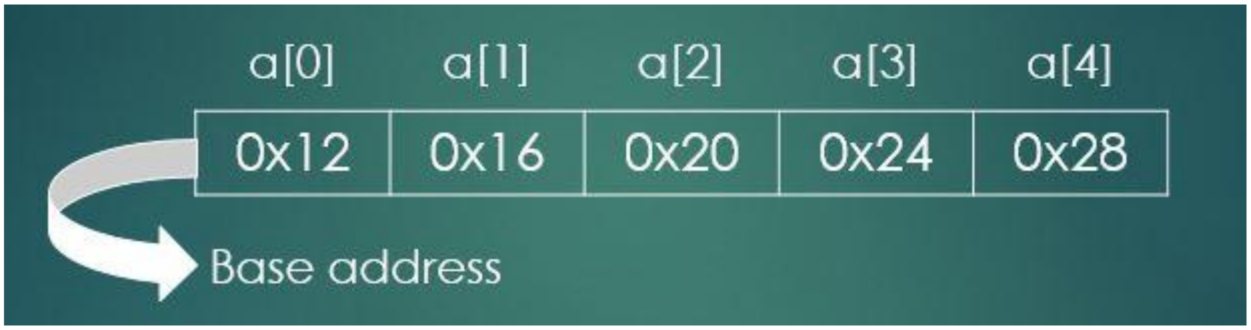
Arrays and Pointers in C

Pointers and Arrays are kind of similar in C programming. As we all know that array is a collection of items of similar data types. And the items in an array are stored in contagious memory locations in computers memory heap. Now if an array is declared like:

```
int a[5];
```

Now in above declaration of the array, if the first element of array `a[0]` is stored at memory location `0x16`(say) then the second element `a[1]` of array will be stored at the memory location adjacent to it which is `0x16`as integer occupies 4 bytes of memory.

In computers memory, it would look like



In C programming, the name of an array always points to the base address i.e above *a* will refer to the address `0x12`. So `&a[0]` and *'a'* are equal.

That is why arrays and pointers are analogous in many ways.

Relation between array and pointer

Consider following declaration

```
int *p;

int a[5];

p = a;
```

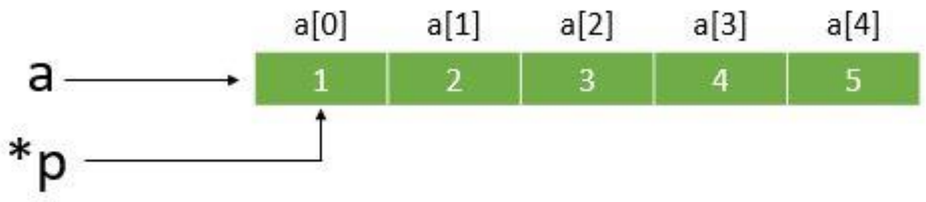
Now in above declaration

`&a[0]` is equivalent to pointer variable *p* and `a[0]` is equivalent to `*p`

`&a[1]` is equivalent to `p+1` and `a[1]` is equivalent to `*p+1`

Analyze following operations:

```
p == &a[ 0 ];     //name of the array is the address of the first element

p + 2 == &a[ 2 ];      //same address

*p == a[ 0 ];         //same value

*(p + 2) == a[ 2 ];  //same value

*p + 3              //3 is added to the value of first element

*(p + 3)            //points to the third element
```



Example: C program to print sum of 2 numbers using pointer to an array

```
#include <stdio.h>

int main()

{

  int i, x[2], sum = 0;

  int *p;

  p = x; //assign the base address

  printf("Enter the number:");

  for( i = 0; i < 2; i++ )
  {

      scanf("%d",( p + i ));

      sum += *(p+i);        // *(p+i) equals x[i]

  }

  printf("Sum = %d", sum);

  return 0;

}
```

Output

```
Enter the numbers: 1

2

sum = 3
```

Difference between pointers and arrays in C programming

As we know that, string are series of characters that are stored in an array. The name of the array acts as a pointer to the base value.

Similarly, pointer variable also points to the specific location in the memory.

For example, we have to store **“trytoprogram”** in the computer memory.

This can be done either by storing it in an array or simply storing it somewhere in computer memory and assigning the address of the string to a `char` pointer.

```
char str[ ] = "trytoprogram";

char *ptr = "trytoprogram";
```

Here, we have used two different methods to store “trytoprogram” as discussed above.

Although both of these look same, there is a subtle difference in usage.

How char str[ ] and *ptr works ?

Let’s explore the difference between pointers and strings in c programming.

```
#include <stdio.h>



int main ()

{

   char str1[ ] = "trytoprogram";

   char str2[ 50 ];



   char *ptr1 = "hello world";

   char *ptr2;



   //look carefully

   str2 = str1; //error

   ptr2 = ptr1; //valid



   str1 = "program"; //error because it is defined and initialized already



   ptr1 = "program"; //valid and works



   return 0;

}
```

Explanation

Let’s discuss line by line of the above program.

We have defined two character `str1` & `str2` type array and initialize `str1`. Similarly, there are two char type pointer `ptr1` and `ptr2`.

Now, we can clearly see from the above example that we cannot assign a string to other. However, we can easily assign pointer to another pointer.

The another difference is that we cannot initialize a string to another set of characters once it has been defined and initialized. But this is totally valid in the case of `char` pointers.

C Strings

A **string** is a sequence of characters which is treated as a single data item in C. It can also be said as an array of characters and any group of characters defined between double quotations is string constants.

“trytoprogram” is an example of string

Now, *try2program* is a string with a group of 11 characters.

Each character occupies 1 byte of memory.

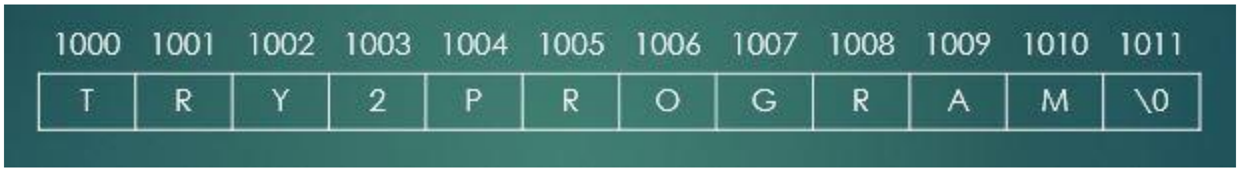
These characters are placed in consecutive memory locations; after all, string is an array of characters.

```
address of "t" = 1000 (say)


address of "r" = 1001


address of "y" = 1002
address of "2" = 1003

address of "p" = 1004

address of "r" = 1005

address of "o" = 1006

address of "g" = 1007

address of "r" = 1008

address of "a" = 1009

address of "m" = 1010
```

String and memory

As we know string is the sequence of arrays and is placed in consecutive memory locations.

To indicate the termination of string a **null character** is always placed at the end of the string in the memory. Now above string will be stored in memory location like this:



Declaring and Initializing string variables

String is not a data type in C, so character arrays are used to represent strings in C and are declared as:

```
char string_name [size];
```

The length of a string **is always greater** than the number of string characters by one because when a compiler assigns a character string to a character array, it automatically supplies a null character (‘\o’) at the end of the string.

Initializing string arrays

Strings in C can be initialized in following ways:

```
  char string_name [12] = "try2program";



  char string_name [12] = {'t','r','y','2','p','r','o','g','r','a','m','\0'};
```

In the above example, the size of an array will be determined automatically by the compiler.

Note: Difference between 0, ‘0’, ‘\0’, “0”.

```
0        //an integer value

'0'      //character value

'\0'     //an escape sequence representing null character

"0"      //string representation
```

How to read strings from users?

Basically, there are two ways for reading the string from users:

- using `scanf` function
- using `getchar` and `gets` functions

using scanf function

The method of reading string using input function `scanf` with *%s* format specifications is the most infamous method.

However, with this `scanf` function, we can only read a word because this function terminates its input on the first white space it encounters.

For example:

```
char name [10];

scanf("%s", name);
```

Note: Normally, an ampersand is used before `scanf` function while reading the value of variables but in the case of character arrays, we don’t need an ampersand (&) because string name itself acts as a pointer to the first variable location.

using getchar and gets functions

`scanf` function is used to read only a word and to read a whole line of text either `getchar` or `gets` function is used.

This function can be used to read successive single characters from the input until a new line character `'\n'` is encountered and then null character is inserted at the end of the string.

Syntax of getchar function

```
char name;

name = getchar( );
```

Note: `getchar` function has no parameters.

Syntax of gets function

```
char name;

getchar(name);
```

Example: C program to read a word and a line of text entered by the user

```
#include <stdio.h>

int main()

{

 char word[20];

 char line[50], ch;

 int a=0;



 printf("Enter name :");

 while(ch!='\n')        //terminates if user hit enter
 {

  ch=getchar();

  line[a]=ch;

  a++;

 }

 line[a]='\0';

 printf("Name =%s",line);

 printf("Enter name :");

 scanf("%s",word);   //only reads a word and terminates at whitespace

 printf("Name = %s \n",word);

 return 0;

}
```

Output

```
Enter name: Denise Ritchie

Name = Denise Ritchie

Enter name : Denise Ritchie

Name = Denise
```

Explanation of the program

In above program, `while` loop will continue until the compiler encounters new line command i.e `\n` .

While user enters strings, `getchar` and `gets` also reads white spaces and only terminates when user will hit enter. So a whole line will be printed.

However, when we read input with `scanf` function no matter how long line user enters it will terminate once it encounters white space.

As a result with `getchar` function compiler can read whole name **Denise Ritchie** and with `scanf` function compiler can only read **Denise** as there is white space after it.

String manipulations in C

C supports a string handling library which provides useful functions that can be used for string manipulations.

All these string handling functions are defined in the header file `string.h`. So every time we use these string handling functions `string.h` header file must be included. Some of the major string handling functions used are tabulated below.

Function	Work
strcat( )	concatenates two strings
strlen( )	finds the length of string
strcpy( )	copies one string in to another another string
strcmp( )	compares two strings
strlwr( )	converts string to lowercase
strupr( )	converts string to uppercase
strstr( )	finds first occurrence of substring in string

strtok( )	splits strings into tokens

Function `strcpy` copies its second argument into its first argument and we must ensure that array is large enough to store strings of second arguments and a null character.

Function `strcmp` compares its first and second arguments character wise. It returns 0 if the strings are equal.
It returns a positive value if the first string is greater than second.
It returns a negative value if the first string is less than second.

Function `strlen` returns the length of input string. It does not include a null character in the length of the string.

puts( ) and putchar( ) function

This both function are defined in input/output library `stdio.h` and are used for manipulating string and character data.

Syntax of putchar

```
int putchar( int x );
```

This will print the character stored in x and returns it as an integer.

Syntax of puts

```
puts("\nHello World!!!");
```

This will print “Hello World!!!”.

C programming structures

*****Structures can be called User Defined Data Type***

Similar to array structures are used to represent a collection of data items but of similar and different types using the single name. **Structure** is a user defined datatype where we have to design and declare a data structure before the variable of that type are declared and used. So it can be said that structures help to organize complex data in a meaningful way.

General format of structure

```
struct tag_name{

  data_type member1;

  data_type member2;

  ..........

  ..........

  data_type member(n);

};
```

Here the keyword `struct` declares a structure that will hold the datafields named members. 'tag_name' is a name of the structure that we have declared. Inside the structure, we can use any kind of datatypes like `int`, `float`, `char` and others.

Things to remember

- The structure template must terminate with a semicolon.

- That tag_name can be used to declare structure variable of its type later in the program

Declaring structure variable

Now that we have defined and designed structure, next step is to declare a structure variable in programs. Here is how it is done:

```
struct tag_name variable1, variable2;
```

Here variable1 and variable2 are variables declared of type struct tag_name. Each one of these variables will hold members defined inside the structure.

Example :

```
  struct product_details

  {

    char product_name[20];

    int product_number;

  };

  struct product_details product1, product2;
```

Here product1 and product2 are variable which will hold members product_name and product_number. NOTE: Remember that members inside the structure are not variables and they do not occupy memory until they are associated with structure variables such as product1 and product2.

Another way to declare structure variable is without using tagname:

```
struct{

    data_type member1;

    data_type member2;
    .........

} name1, name2, name3;
```

Here `name1`, `name2`, and `name3` are structure variables representing members but such declaration has no tag_name.
This method is not recommended because, without a tagname, we cannot use structure variable of its type for future declarations.

Accessing structure members

As we already know members themselves are not variables and they don’t hold any meaning and memory unless they are associated with a structure variable. So to access those members member operator ‘.’ is used which is also called dot operator.
For example:

```
product1.product_name;
```

Example to highlight concept of structure in C programming

Program to take employee, his/her name, age, and salary.

```
#include <stdio.h>

struct employee{    // structure definition

  char name[20];

  int age;

  int salary;

}

int main()

{

  struct employee person; // declaration of structure variable person

  printf("Enter name,age and salary \n");

  scanf("%s %d %d", person.name, person.age, person.salary);

  printf("%s %d %d \n", person.name, person.age, person.salary);
  return 0;

}
```

Output:

```
Enter name,age and salary

Joe 27 90000
```

Structures within structures

Structure within structure means nesting of the structure where one `struct` variable is used in another structure definition or a new structure is defined inside the structure.

Examples:

```
struct book

{

  char name[20];

  char author[20];

  struct

  {

  int price;

  int pages;

 } book_details;

}:
```

Here `book` structure contains a member named `book_details` which are itself structure with two members.

Another way to write nested structures is:

```
struct book_details

  {
```

```
  int price;

  int pages;

 };

struct book

{

  char name[20];

  char author[20];

  struct book_details intro;

};
```

C PROGRAMMING STRUCTURE AND ARRAYS

we used datatypes like `int`, `float` and arrays of characters inside the structure.

Besides those C also permits the use of arrays as elements that can be one-dimensional or multi-dimensional of any type.

In this tutorial, you will learn about the association of **c programming structure and arrays**.

Example of arrays within structure

```
struct car_model{

  char car_name[20];

  int model[5];
} cars[3];
```

Here in this example `model` contains 5 elements `model[0]`, `model[1]`, `model[2]`, `model[3]`, `model[4]` and three struct variable `car` also has 3 elements `car[0]`, `car[1]` and `car[2]`. Those structure elements `model` can be accessed as following:

```
car[0].model[3];
```

Arrays of structures

In C, we can also create an array of structure variable where each element of the array will represent structure variable.

It is same like a multidimensional array with each array element holding structure variable which contains data of different datatypes.

Consider following example:

```
struct car{

  int carnum_1;

  int carnum_2;

};

int main()

{

  struct car numbers[2] = {{22,34},{54,88}};

}
```

In above example, we have declared an array of structure variable called `numbers` which contain 2 elements `numbers[0]` and `numbers[1]`. Here each element is initialized and can be accessed as following:

```
numbers[0].carnum_1 = 22;

numbers[1].carnum_1 = 34;

numbers[0].carnum_2 = 54;

numbers[1].carnum_2 = 86;
```

C programming files IO operations and functions

So far we have used i/o functions like `printf` and `scanf` to write and read data. These are console oriented functions used when data is small.

So for handling large data, a new approach is introduced in c programming called file handling where data is stored on the local machine and can be operated without destroying any data.

A file simply is a collection of data stored in the form of a sequence of bytes in a machine.

Before jumping onto file handling functions we must know about the types of files used in c programming language.

- **Text File**
- **Binary File**

Text files in C

Text files are humanly readable which is displayed in a sequence of characters that can be easily interpreted by humans as it presented in the textual form. Common editors like notepad and others can be used to interpret and edit them.

They can be stored in plain text (.txt) format or rich text format (.rtf).

Binary files in C

In binary files data is displayed in some encoded format (using 0’s and 1’s) instead of plain characters. Typically they contain the sequence of bytes.

They are stored in `.bin` format.

Thus C supports some functions to handle files and performs some operations. It includes:

- opening a file
- reading data from a file
- writing data to a file
- naming a file
- closing a file

File handling in C

For any file handling operations first, a pointer of `FILE` type must be declared in C. For example:

```
FILE *fp;
```

Common I/O functions used for file handling in C

Functions	Operations
fopen()	For creating new file or opening existing file
fclose()	For closing a file
getc()	For reading a character from file
putc()	For writing a character to a file
fprintf()	For writing set of data to a file
fscanf()	For reading set of data from a file
getw()	For reading an integer from a file
putw()	For writing an integer to a file
fseek()	For setting the position to the desired point in a file
ftell()	Gives the current position in a file
rewind()	Sets the position to the beginning of the file

Opening a file

As mentioned in above table `fopen()` is a standard function used to create a new file or open an existing file from our local machine.
The general format for declaring and opening a file is:

```
FILE *ptr;


fp = fopen("filename","mode");
```

First, a pointer `fp` of `FILE` type is declared. In the second statement, `fopen` function is used to open the file named `filename` with a purpose which is defined by the *mode*.

The file is assigned to pointer variable `fp` which is used as the communication link between the system and the program.

The purpose of opening a file is defined by mode. It can be any of the following:

File opening modes in standard file I/O

Mode	Purpose
r	opens a text file in reading mode
w	opens the text file for writing mode
a	opens the text file for appending mode or adding data to it
r+	opens the text file for both reading and writing mode
w+	opens the text file for both reading and writing mode
a+	opens the text file for both reading and writing mode
rb	opens a binary file in reading mode
wb	
ab	
rb+	
wb+	
ab+	

opens a binary file in writing mode
opens the binary file for appending mode or adding data to it
opens the binary file for both reading and writing mode
opens the binary file for both reading and writing mode
opens the binary file for both reading and writing mode

For example:

```
FILE *fp;


fp = fopen("C://myfile.txt","r");
```

Here in this example, ‘myfile.txt’ will be opened in reading mode and if the file doesn’t exist in the specified location, it will return NULL.

NOTE: if the mode is writing mode then a new file will be created if the file doesn’t exist.

Closing a file

A file must be closed once it is opened and all operations are done because of various reasons like it will prevent any accidental misuse of the file. Moreover, sometimes the number of files that can be opened simultaneously is limited. So it is always best to close files once all operations are finished.

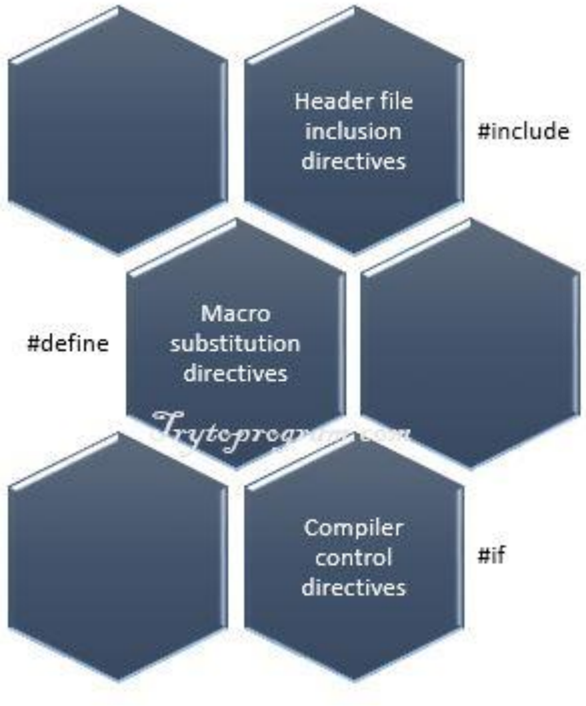
`fclose()` is used for closing a file.

```
fclose(fp); // fp was the pointer to which our previously opened file was assigned
```

Additional Topics

C PROGRAMMING PREPROCESSOR DIRECTIVES

In this tutorial, you will learn in depth about **C programming preprocessor directives** that modify source code before it is compiled by the compiler.


Header file inclusion directives
#include
#define
Macro substitution directives
Trytoprogram.com
Compiler control directives
#if

As the name implies **C programming preprocessor directives** are the instructions that execute our program before it is compiled.

Many of write C program without knowing what actually preprocessor is. In this article, we will explore preprocessor directives in C programming.

C programming Preprocessor directives

While writing C program, the first line of code that starts with `#` is called preprocessor directive and only comments and whitespace character are allowed before it.

#include preprocessor directive

Every C program starts with `#include` preprocessor directive with `.h` extension file known as a header file.

This causes the copy of a file to be placed in place of the directive. There are two ways to write `#include`directive. These are

```
#include <filename>


#include "filename"
```

Following is the difference between these two:

```
#include <filename>
```

This command is used for standard library headers. It searches for header file in the specified list of directories only.

See the list of standard library headers.

```
#include "filename"
```

This command searches for the file in the current directory as well as the specified list of directories. This method is used for **programmer-defined headers.**

#define preprocessor directive (Symbolic Constants)

The `#define` directive is used to create symbolic constants and the statement is called **macro.**

```
#define identifier replacement_text
```

When a symbolic constant is defined in the program, all subsequent identifier or symbol is replaced by `replacement_text`.

For example:

```
#define SIZE 10
```

When symbolic constant `SIZE` appears in the program it is replaced with 10.

Example: C program to calculate the area of a circle.

```
/* Program to calculate the are of a circle */

#include <stdio.h>

#define PI 3.1415



int main()

{

   float r, area;

   printf("Enter radius: ");

   scanf("%f", &r);



   area = PI * r * r;

   printf("\nArea of circle= %.3f", area);



   return 0;

}
```

Output

```
Enter radius: 4.2
```

```
Area of circle= 55.416
```

Explanation

In the above program, we have defined symbolic constant `PI` which value is `3.1415`.

If the value of constant needs to be modified in the program, it can be done simply by changing the value of `PI`in `#define` directive.

Good Programming Practice

In C symbolic constants are defined only using capital letters and try using a meaningful name for it.

Common Programming Error

Terminating macro definition with a semicolon and forgetting to enclose header file with `< >`.

Common example of using #define

```
#include<stdio.h>

#define OR ||

#define AND &&

int main()

{

   int x, y, z;

   printf("Enter values of x, y and z: ");

   scanf("%d %d %d", &x, &y, &z);

   if((x > 5 AND y < 10) OR (z > 100))

       printf("Max will get apple.");

   else
```

```
       printf("Max will get punishment.");

   return 0;

}
```

Macros with arguments

Till now we have used simple macros with no arguments. Interestingly macros can have arguments.

Example: Using macro with arguments

```
#include <stdio.h>

#define CIRCLE_AREA(r) (3.1415 * r * r)



int main()

{

   float x = 4.2;

   printf("Area of circle= %.3f", CIRCLE_AREA(x));

   return 0;

}
```

Output

```
Enter radius: 4.2

Area of circle= 55.416
```

Explanation
In the above program, wherever there is `CIRCLE_AREA(r)` it is replaced with `3.1415 * r * r`. The variable `x`in `CIRCLE_AREA(x)` is subsituted to `r`.

Undefining a Macro

We can easily undefine the defined macro using the following statement.

```
#undef identifier
```

This is very useful when we want to restrict the use of the defined macro in other parts of the program.

Example:

```
#undef SIZE
```

Common Programming Error

Placing space between the macro template and its argument results in error. For example, there should not be space between `CIRCLE_AREA` and `(r)` in the definition.

C PROGRAMMING MATH LIBRARY FUNCTIONS

In this article, you will learn about various types of **C programming math library functions** with their function prototype.

C standard library provides a huge collection of standard functions for performing various common tasks such as mathematical calculations, input/output, character manipulations, string manipulations etc.

Standard C programming math library functions

We should use C standard library function when possible instead of writing new functions.

C programming math library functions allow us to perform common mathematical computations. For example:

```
printf("%.2f", sqrt(16));
```

The above code will calculate square root of 16 using standard `sqrt` library function. For using this function, we should include `math.h` header by using preprocessor directive which is shown below:

```
#include <math.h>
```

The function arguments can be variables, constant or expressions. For example: If a=8, b=8

```
printf("%.2f", sqrt(a + b));
```

The above statement will calculate and print the square root of `a + b`.

Commonly used C programming math library functions

C programming math library functions with description

sqrt(x)
square root of `x`

cbrt(x)
cube root of `x` (C99 and C11 only)

exp(x)
exponential function `x`

log(x)
natural logarithm of `x` (base e)

log10(x)
logarithm of `x` (base 10)

fabs(x)
absolute value of `x` as a floating-point

ceil(x)
rounds `x` to the smallest integer not less than `x`

floor(x)
rounds `x` to the largest integer not greater than `x`

pow(x, y)
`x` raised to power `y` (xy)

fmod(x, y)
remainder of `x/y` as a floating-point number

sin(x)
trigonometric sine of `x`

cos(x)
trigonometric cosine of `x`

tan(x)
trigonometric tangent of `x`

In this article, you will learn about different **C programming character handling library functions** that are used for character processing.

As we know that characters are the fundamental building blocks of every program.

A character constant is represented as a character in a single quote. The value of character constant is an integer.

For example, `a` is represented as `'a'` which is actually the integer value equal to 97 in ASCII.

Similarly, `'\n'` represents the integer value of newline equal to 10 in ASCII.

C provides standard character handling library `<ctype.h>.`

We should include character handling library i.e. `<ctype.h>` for using different character handling function.

This header include various useful functions for performing tests and manipulations of character data in C programming.

The function in `<ctype.h>` recieves an `unsigned char` represented as `int` or `EOF` as argument.

In C programming, character is a one byte integer.

C Character handling library <ctype.h> functions

C programming character library functions with description

int islower(int ch);
It returns a *true* if `ch` is a *lowercase* letter and *0* otherwise.

int isupper(int ch);
It returns a true value if `ch` is a *uppercase* and 0 therwise.

int tolower(int ch);
This function converts *uppercase* letter into *lowercase* letter.

int toupper(int ch);
This function converts *lowercase letter* into *uppercase letter*.

int isblank(int ch);
This function checks whether `ch` is a *blank character* or not.

int isdigit(int ch);
This function checks whether `ch` is a *digit* or not.

int isalpha int ch);
This function checks whether `ch` is *a letter* or not.

int isalnum(int ch);
This function returns a 1 if `ch` is *a digit* or *a letter* and 0 otherwise.

int isxdigit(int ch);
This function checks if `ch` is a *hexadecimal digit* or not.

int isspace(int ch);
This function checks if `ch` is a *whitespace character* or not.

int isgraph(int ch);
This function checks if `ch` is a *printing character* other than a space or not.

int iscntrl(int ch); This function checks if `ch` is a *control character* or not.
int isprint(int ch); This function checks if `ch` is a *printing character* including a *space* or not.
int ispunct(int ch); This function checks if `ch` is a *printing character* other than *a digit, a space, a letter* or not.

Examples:

http://www.trytoprogram.com/c-examples/

User Defined Library in C

USER DEFINED FUNCTIONS IN C:

- As you know, there are 2 types of functions in C. They are, library functions and user defined functions.
- Library functions are inbuilt functions which are available in common place called C library. Where as, User defined functions are the functions which are written by us for our own requirement.

ADDING USER DEFINED FUNCTIONS IN C LIBRARY:

- Do you know that we can add our own user defined functions in C library?
- Yes. It is possible to add, delete, modify and access our own user defined function to or from C library.
- The advantage of adding user defined function in C library is, this function will be available for all C programs once added to the C library.
- We can use this function in any C program as we use other C library functions.
- In latest version of GCC compilers, compilation time can be saved since these functions are available in library in the compiled form.
- Normal header files are saved as "file_name.h" in which all library functions are available. These header files contain source code and this source code is added in main C program file where we add this header file using "#include <file_name.h>" command.
- Where as, precompiled version of header files are saved as "file_name.gch".

STEPS FOR ADDING OUR OWN FUNCTIONS IN C LIBRARY:

STEP 1:

For example, below is a sample function that is going to be added in the C library. Write the below function in a file and save it as "**addition.c**"

```
addition   (int   i, int    j)
{
        int total;
        total = i + j;
        return total;
}
```

STEP 2:

Compile "addition.c" file by using Alt + F9 keys (in turbo C).

STEP 3:

"addition.obj" file would be created which is the compiled form of "addition.c" file.

STEP 4:

Use the below command to add this function to library (in turbo C).

```
c:\> tlib math.lib + c:\ addition.obj
```

+ means adding c:\addition.obj file in the math library.
We can delete this file using – (minus).

STEP 5:

Create a file “addition.h” & declare prototype of addition() function like below.
int addition (int i, int j);
Now, addition.h file contains prototype of the function “addition”.

Note : Please create, compile and add files in the respective directory as directory name may change for each IDE.

STEP 6: Let us see how to use our newly added library function in a C program.

```
# include <stdio.h>
// Including our user defined function.
# include “c:\\addition.h”
int main ()
{
    int total;
    // calling function from library
    total = addition (10, 20);
    printf ("Total = %d \n", total);
}
```

A Matrix Library:

Name: matrix.c

```
/*
   *Maximum Two matrix can be calculated, if you have more than two matrix
   You have calculate two by two.

   **You can Calculate square or non-Square matrix maximum row and column size is 100

*/

// Input function
void input (int m[100][100], int x, int y)
{
    int i,j;

    for(i=0; i<x; i++)
    {
        for(j=0; j<y; j++)
        {
            scanf("%d",&m[i][j]);
        }
    }

  return;
}
```

```
// Print Function

void Print (int m[100][100], int x, int y)
{
   int i,j;

   printf("\n  Printing the matrix: \n");

   for(i=0; i<x; i++)
   {
      printf("\n");
      for(j=0; j<y; j++)
      {
         printf(" %d",m[i][j]);
      }

   }

   return;
}



//Addition
int add (int m1[100][100], int m2[100][100], int m3[100][100], int x, int y  )
{
   int i, j;

   for(i=0; i<x; i++)
   {
      for(j=0; j<y; j++)
      {
         m3[i][j]=m1[i][j]+m2[i][j];
      }
   }

}


//Subtraction of matrix
int sub (int m1[100][100], int m2[100][100], int m3[100][100], int x, int y  )
{
   int i, j;

   for(i=0; i<x; i++)
   {
      for(j=0; j<y; j++)
      {
         m3[i][j] = m1[i][j] - m2[i][j];
```

```
        }
    }

}

//  Matrix Multiplication is a complex term :)


void multiplie (int m1[100][100], int r1, int c1, int m2[100][100], int r2, int c2, int
m3[100][100])
{
    int rem, i, j;

          //elegebility checking;
    if(c1==r2)   rem=1;
    else if (c2==r1)   rem=2;
    else
    {
        printf("Wrong Input \n Multiplication not possible");
        return;
    }

          //operation

    if(rem==1)
    {

        for(i=0; i<r1; i++)
        {
            for(j=0; j<c2; j++)
            {
                m3[i][j]=0;

                for(int k=0; k<c1; k++)
                {
                    m3[i][j] = m3[i][j] + (m1[i][k] * m2[k][j]);
                }
            }
        }



      else  if(rem==2)
    {

        for(i=0; i<r2; i++)
        {
```

```
        for(j=0; j<c1; j++)
        {
          m3[i][j]=0;

          for(int k=0; k<c2; k++)
          {
            m3[i][j] = m3[i][j] + (m1[i][k] * m2[k][j]);
          }
        }
      }

}}
```

Refference:

1. **https://www.javatpoint.com/history-of-c-language**
2. **http://www.trytoprogram.com/c-programming/first-program-in-c/**
3. **https://capitalizemytitle.com/camel-case/**
4. https://fresh2refresh.com/c-programming/c-creating-library-functions/
5.