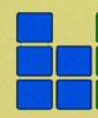
LEARN C

PROGRAMMING

FOR BEGINNERS

By



The Crazy Programmer

About this EBook

These tutorials are made for those people who have absolutely no pre knowledge of programming. In these tutorials you will need only a basic knowledge of computers. All the topics are covered from scratch and in the last we will cover some advanced topics too.

I have tried my best to provide correct and useful information in this eBook. Still if you find any mistake or anything missing then please contact me. Send you feedback at **sareneeru94@gmail.com**.

You can also contact me if you have any queries regarding any topic or concept of C language. I will try my best to help you.

For more tutorials and programs visit **www.thecrazyprogrammer.com**

Reference

I have taken Let Us C by Yashavant P. Kanetkar and Programming in ANSI C by E Ballagurusamy as a reference for writing this eBook.

Copyright

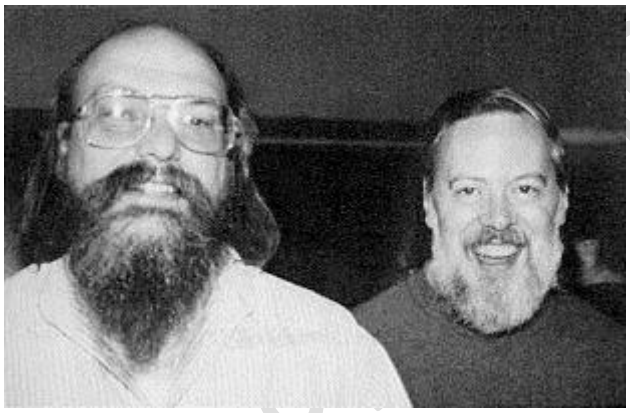
All the Best and Happy Coding!! ☺ ☺

Table of Contents

C Programming Overview

What is C language?

Well the answer is quite simple. It's a procedural programming language which was designed and written by Dennis Ritchie at AT & T's Bell Labs in 1972. In early 70s very frequently new programming languages were introduced. But after the launch of C, it slowly replaced the old languages like ALGOL. Actually no one advertises this language and even Ritchie was surprised that so many peoples were using C language. Till now many languages has come. But the popularity of C is not changed till now. The main reason behind it is that it is still fast, reliable and easy to use.



Ken Thompson and Dennis Ritchie

Why C should be your first programming language?

Seriously many people claim that now one should start its programming journey through C++, C# or JAVA. Well I think nobody will be comfortable while studying the advanced concepts like OOPS from the start.

Applications

C language has widely uses, some main uses are given below.

1. Used to develop softwares that control embedded systems. Examples of some embedded systems are washing machine, microwave oven, etc.
2. For creating computer applications.
3. UNIX operating system is developed in C language.

Compilers

The first question which will arise in our mind. What is a Compiler?

Well as everybody knows Computer is a machine which process raw data into meaningful information. It cannot understand our language. It can only understand machine language. Even if we write programs in any language (like C). At the end they will be converted to machine language to process through computer.

What does Compiler do?

Compiler converts your program into machine language or binary language.

Install IDE first

What is IDE?

IDE means Integrated Development Environment. To write any program in C language you will need an editor. After that to convert that program into machine language, you will need a compiler. IDE will combine editor and compiler into same environment. So you can easily write the program and also compile it easily.

Well I would suggest you not to use Turbo C. Because it is very old compiler (more than 20 years). I don't know about other countries but in Indian most of the institutes and colleges are still using it. Start your C programming journey with some modern compiler like GCC. Its available free of cost and you can download it from http://www.thecrazyprogrammer.com/p/downloads.html. It doesn't matter a lot which compiler or ide you are using. The main thing is to learn the logic.

Variables, Constants and Keywords

Variables

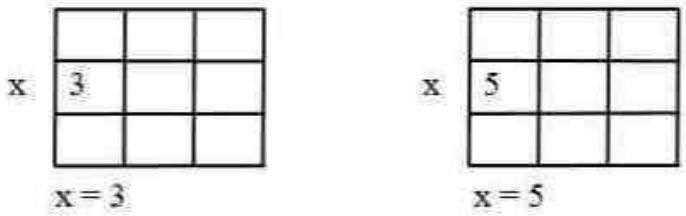
In a typical C program we have to do a lot of computation. Of course there will be storing of some data in different locations of memory in computer. These memory locations are identified by their address like 56234. Suppose if programmer wants to access the particular locations like 10 times in a program to store another value at that location. So It will become a tedious job for a programmer if he have to memorise the numerical address name. So to make this job easy we use "Variables".

So "Variables are nothing but the name of the locations or addresses of memory which is given by programmer."

Constants

As its name suggests constants are the values which will never change during the execution of program.

Sounds confusing? Let's try to make things more clear with a simple example.



In the above picture (1st) we have stored the constant value 3 at x location. The name of that location is x. It's a variable. We can also store another value at x location.

Here X = variable (location or memory address name)
3 = constant

Try to understand the second example yourself if you have any doubt, do comment below.

There are two type of Constants

1. Primary constants
2. Secondary constants (We will learn them later)

At this stage we will only discuss primary constants. Primary constants are of three types.

a. Integer constants
b. Real constants
c. Character constants

Let's discuss them one by one.

Integer Constant

Yes it will contain only integers. Remember an integer constant will never contain any decimal point. Eg: 1, 2, -43 etc

Character Constant

It is single (remember) alphabet, number or any special symbol which is enclosed in an inverted commas. Eg: '+', '1', 'a', etc.

Real Constant or Floating Point Constant

A real constant may have any digit but it must contain one decimal point. Eg: 1.22, -54.5, 3432.13

Types of Variables

As I said earlier Variable are name of locations in memory. In that location we can store any constant like integer, character or Real. But there is some limit that an integer variable can only store integer constant, character variable can only store character constant and real variable can only store real constant.

So it is quite obvious types of variables is similar types of constants. Eg: int x= 1;

Here "int" is a keyword, "x" is an integer variable and it can only store integer constant, "1" is a integer constant.

Rules for writing variable names

1. A variable name may contain alphabets, numbers and underscores.
2. No other special character (other than underscore) can be used for writing variable name.
3. A variable name should start with either underscore or alphabets.
4. No space is allowed while writing variables.

Keywords

Keywords are the words whose meaning is already explained to the compiler. They cannot be used as a variable name.

A question which may arise in your mind that, how computer will know that its integer variable or character variable or anything else?

The simple answer is with the help of keywords. In one of the above example I have used "int" keyword. Eg: int x=1

In this example "int" is a keyword and it will tell the computer that "x" will be an integer variable and it will only store integer constant.

There are 32 keywords used in C language which are given below. We will discuss them in later tutorials.

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

First C Program - Hello World

We have gained lots of theoretical knowledge. Now its time to move on and write our first C program and understand it.

Hello World C Program

This is a program to print "Hello World"

```
#include<stdio.h>

void main()
{
 printf("Hello World");
}
```

Output
Hello World

Now try to understand this program step by step.

1. **#include<stdio.h>:** First statement started with #, it is called pre-processor directive. We will learn about them thoroughly in later tutorials. #include<stdio.h> is used to include the stdio.h header file in our program. Header files contains the functions that we use in our program. Here we have used printf() function which is present in stdio.h header file.

2. **void main():** Here main() is the function. A program always starts with the main() function and every program must have main(). Here void is the return type of this function. It means main() function will not return anything. The opening curly braces ({) and closing curly braces (}) shows the body of the function.

main() can also be called as a collection of statements.

3. **printf():** Here printf() is another function. This is used to print the values on the screen. Its general form is

printf("Statement you want to print on screen");

4. Semicolon (;) is used for denoting the termination of statement. It is also called statement terminator in C Language.

How to compile and execute?

Here we are using TURBO C compiler. If you have not downloaded it yet than download it from http://www.thecrazyprogrammer.com/p/downloads.html

1. Open Turbo C and type the program.
2. After that press F2 to save the program with .c extension. Eg: program.c
3. Now Press ALT + F9 to compile and CTRL + F9 to run the program.
4. Press ALT + F5 to view the output.

Well this is the easiest C program. Now lets move on and try slightly complicated program of multiplication of two numbers.

C Program to multiply two numbers

```
#include<stdio.h>

void main()
{
 int a, b, c;
 a=3;
 b=4;
 c=a*b;
 printf("Answer is %d",c);
}
```

Output
Answer is 12

Now lets try to understand this program.

1. First two statements are same as above program. In the third statement I have written

int a, b, c;

Here int is the keyword for integer and a, b and c are the integer variables. So they can only store integer values.

2. In the next two statements I have written

a=3;
b=4;

In these statements we are storing the values 3 and 4 in a and b variables respectively.

3. In the next statement

c=a*b;

We are multiplying the values in a and b, and storing them in the variable c.

4. Now in the last statement we are printing the value which is in c. A new thing %d which we have used in this program. It is called format specifier. It usually tell the compiler that it has to print the integer value on screen which is present in a variable.

Some common format specifiers are given below

a. %d for integers
b. %c for characters
c. %f for floating point numbers or real numbers

A bit elaborated form of printf() is given below

printf("string you want to print ", variable);

Suppose we have to print the value in b so we will use the function.

printf("%d",b);

Things to keep in mind

1. Use semicolon at the end of each statement.
2. Type the statements in main program as the way you want them to be executed.
3. Never try to memorise any program. Just try to understand it.
4. Learning programming is all about practical. So start doing practical from now.

Structure of C Programs

Basic Structure of C Programs
Documentation Section
Link Section
Definition Section
Global Declaration Section

main() Function Section
{
Declaration Part
Executable Part
}

Subprogram Section Function 1 Function 2 Function 3 - - - Function n

Documentation Section

This section consists of comment lines which include the name of programmer, the author and other details like time and date of writing the program. Documentation section helps anyone to get an overview of the program.

Link Section

The link section consists of the header files of the functions that are used in the program. It provides instructions to the compiler to link functions from the system library.

Definition Section

All the symbolic constants are written in definition section. Macros are known as symbolic constants.

Global Declaration Section

The global variables that can be used anywhere in the program are declared in global declaration section. This section also declares the user defined functions.

main() Function Section

It is necessary have one main() function section in every C program. This section contains two parts, declaration and executable part. The declaration part declares all the variables that are used in executable part. These two parts must be written in between the opening and closing braces. Each statement in the declaration and executable part must end with a semicolon (;). The execution of program starts at opening braces and ends at closing braces.

Subprogram Section

The subprogram section contains all the user defined functions that are used to perform a specific task. These user defined functions are called in the main() function.

Don’t worry if you are finding difficulty to understand these sections. They will be covered one by one in later tutorials.

printf(), scanf() and comments

Hello everyone, I hope you must have done the practical test of our previous programs. Remember practical knowledge is utmost important in learning c language.

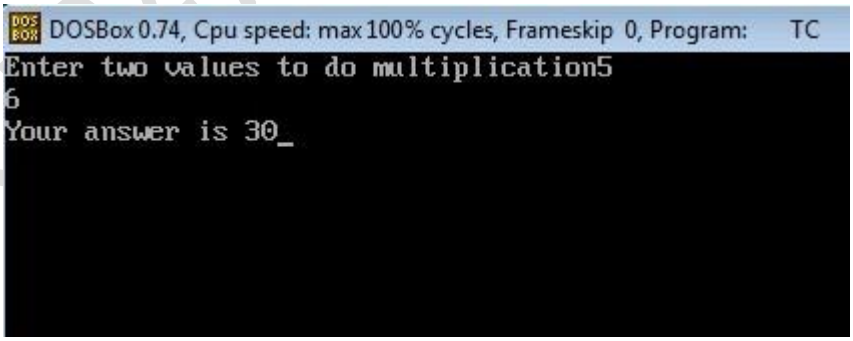
Anyways till we have covered the basic use of printf() function by which we can print values on the screen. Today we will learn how to take values from the user.

Note: Read previous article to know more about printf() function.

scanf()
scanf() is used to take data from the user. Till now we have wrote programs in which we declared variables with some values. But in practice we need those programs which are general enough to make computations.

So with the help of scanf() function now we will make a general program for multiplication of two numbers. In this program we will ask the user to enter the values.

```
#include<stdio.h>

void main()
{
 int a,b,c;
 printf("Enter two values to do multiplication");
 scanf("%d%d",&a,&b);
 c=a*b;
 printf("Your answer is %d",c);
}
```



Now lets try to understand this program.

1. First two instructions are same like our previous programs.

2. In the third instruction we are declaring three variables of integer type.

3. In the fourth instruction we are printing the statement using printf() function.

4. In the fifth instruction we are taking input from the user through scanf() function.

In this scanf() function we have done two things.

a. We have given the format specifier %d to instruct the compiler that we want to input integer value.

b. We have used ampersand (&) which is also called "address of operator". By using this we instruct the compiler we want to store that input in that variable (a and b).

Why do we use ampersand operator (&)?

As I have said already it is a "address of operator". By using this operator we specify the address of variable to the compiler.

A bit confusion..? Ok, checkout the example below.

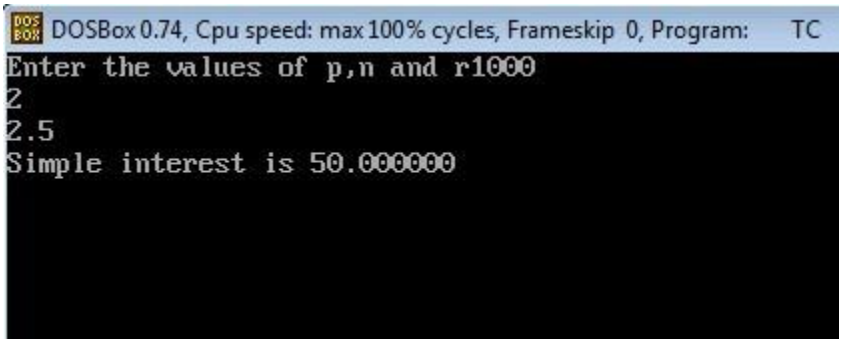
Suppose we have used &a. Now C compiler will receive the input and go to the address of a (which can be anything like 7635). After that it will store that value on that particular address. That's it.

Lets write another program which is slightly complicated i.e. program to calculate simple interest.

C Program to Calculate Simple Interest

In this program I am assuming that you must know the formula and working of simple interest in mathematics. So I will not explain that formula to you.

```
/*Program to calculate simple interest
TheCrazyProgrammer date 21/12/14*/

#include<stdio.h>

void main()
{
 int p,n; //Here p is principle amount and n is number of years
 float r,si; //Here r is rate of interest and si is simple interest
 printf("Enter the values of p,n and r");
 scanf("%d%d%f",&p,&n,&r);
 si=(p*n*r)/100;
 printf("Simple interest is %f",si);
}
```



Lets try to understand this program step by step.

1. First two statements are comments.

Comments

Comments are generally used to increase the readability of program. At present we are making very small programs. But when we develop big programs then the program has to go through a long process of testing and debugging. Comments are not the part of program code and are not read by compiler.

It is very important to write comments in programs. So that other programmers can also read and understand your program easily. Writing comments is also a good programming practice. Start writing comments in the programs from the beginning itself.

C allows two types of comments

a. **Single line comment**: // first type of comment
b. **Multiline comment**: /* second type of comment*/

Single line comment is used to write comments in one line only. Multiline comment is used to write comments in multiple lines. All things that comes in between /* and */ is considered as comment. We can use anyone according to requirement.

2. After that next three instructions are same which includes C pre-processor directives, main() function, declaration of integer variables.

3. In the fourth instruction we have declared float variable r and si. Because rate of interest can be a floating point number. And to stay on safe side we also declared si variable as float. As the answer may come in floating point.

4. After that using printf() function we print a message to instruct the user to insert the values of p, n and r.

5. Using scanf() we are taking input from the user. Checkout we have used %f format specifier for r variable. We have declared r as float variable. So we have to use %f format specifier to print as well as receive values in r variable.

6. Now in the next statement we have calculated the simple interest using the formula.

7. In the last we have print the answer using printf() function. Notice we have used %f format specifier there. As si is also a float variable.

So these are the basic use of printf() and scanf() functions. These are one of the most used functions in C language. So you can estimate the importance of them. Now you can make 100s of programs by using these two functions.

Try making these programs yourself (take values from user)
1. Make a program to add two numbers.
2. Make a program which will convert distance in km to meter.

C Instructions

I hope before continuing to this tutorial you must have written the basic printf() and scanf() programs. If you didn't then I strongly recommend you to do it.

Anyways so far we have covered the very basic programs of C. We have used many instructions in it. So today I will tell you about the C instructions.

What are C Instructions?

There are basically three types of C instructions.

1. Type Declaration Instruction
2. Arithmetic Instruction
3. Control Instruction

We have covered first two types of instructions already in our previous turorials but I will tell you about certain nuances of these instructions now. So lets try to analyse them one by one.

Type Declaration Instruction

As its name suggests it is used to declare type of variables in C language. It is must that you have to declare the type of variable before using it. And it is also must that these type declaration instruction should be provided in the beginning of the program (just after the main()).

Now lets learn about some variations in them.

a) We can initialize the variable at the time of its declaration.

Example
```
int a=3;
char a='d';
int a=7*3*2;
```

b) Remember the order of variables while declaring them.

Example
int i=3,j=5; is same as int j=5,i=3;

However,
float a=5.5,b=a+7.1; is alright, but
float b=a+7.1,a=5.5; is not valid.

It is because in the above declaration we are trying to use variable a even before defining it.

c) Another interesting point while declaring variables is

Below instruction will work
int a,b,c,d;
a=b=c=10;

However, the following statement would not work.
int a=b=c=d=10 ;

Now again we are trying to use variables even before defining them.

I hope now you must have understand the importance of type declaration instruction. So we can only use variables after defining them. And these definition should be written at the starting of main() body. We cannot define variables anywhere else in the program. If you do so, it will result in an error.

Arithmetic Instruction

General form of an arithmetic instruction follow rules given below.

i) It should contain one assignment operator i.e. =.
ii) On the left side of =, there should be one variable.
iii) On the right side of =, there should be variables and constants.
iv) And those variables and constant will be connected by some arithmetic operators like +,-,*,/,%.

Example
int a=12;
float b,c=2.2;
b=c+a/6.1*8;

What are operands?
These variables and constants together are called operands. These operands are connected by arithmetic operators.

How the execution of arithmetic instructions takes place?
Firstly all the variables and constants on the right hand side of assignment operator (=) is calculated. After that the result will be stored in the variable on the left hand side of assignment operator.

Now let's checkout certain nuances of arithmetic operators

1. It is compulsory to use only one variable on the left hand side of =.

Example
c=a*b is correct,
whereas a*b=c is incorrect.

2. A operator named modular operator (%) is also used in C. This modular operator will return the remainder to left hand side variable of assignment operator. It cannot be used with floats. It will return the same sign as the numerator has.

Example
-5%2=-1
5 %-2=1

Control Instruction

This instruction is used to shift the control of program as per the user or programmer wants. This instruction contains decision making, looping and more. This is a bit advance topic. We will cover this topic in our later tutorials.

Type Conversion, Precedence and Associativity of Operators

The process of converting one data type into another data type is known as type conversion.

The automatic conversion of one data type into another data type is known as implicit type conversion. It is done by the compiler.

The type conversion can also be done manually. The type conversion done by the programmer is known as explicit type conversion.

We will discuss about these two types in detail in our next tutorials.

It is quite common that we have to use both integer and float numbers. While doing some arithmetic operations between them we should take care of few things which we will discuss in this tutorial.

1. Arithmetic operation between integer and integer will always result in an integer.
2. Arithmetic operation between float and float will always give float number.
3. Arithmetic operation between float and integer will always give float number. In this case, first integer will be promoted to float after that the arithmetic operation will take place.

Checkout the table below to understand these operations.

Operation	Result	Operation	Result
5 / 2	2	2 / 5	0
5.0 / 2	2.5	2.0 / 5	0.4
5 / 2.0	2.5	2 / 5.0	0.4
5.0 / 2.0	2.5	2.0 / 5.0	0.4

Type Conversion in Assignments

It is quite often that the variable on the left hand side of assignment operator (=) does not match the type of variable on its right hand side. In such a case, the value of the whole expression will be promoted or demoted on the basis of variable present on left hand side of assignment operator.

To make things a bit more clear lets take an example.

float a;

```
int b;
b=4.2;
a=3;
```

In the above example we are trying to store the float value (4.2) in integer variable b. So the value will be demoted to (4) and it will store in the integer variable. Same thing will happen with 3. As 3 is an integer, so it will be converted to float (3.00000) and then it will be stored in float variable.
Complexity of the arithmetic operation doesn't matter. You should stick with the three basic rules. That's it.

To understand things better. Checkout below examples.

Arithmetic Instruction	Result	Arithmetic Instruction	Result
k = 2 / 9	0	a = 2 / 9	0.0
k = 2.0 / 9	0	a = 2.0 / 9	0.2222
k = 2 / 9.0	0	a = 2 / 9.0	0.2222
k = 2.0 / 9.0	0	a = 2.0 / 9.0	0.2222
k = 9 / 2	4	a = 9 / 2	4.0
k = 9.0 / 2	4	a = 9.0 / 2	4.5
k = 9 / 2.0	4	a = 9 / 2.0	4.5
k = 9.0 / 2.0	4	a = 9.0 / 2.0	4.5

Precedence (Hierarchy) of Operators

BODMAS is the basic rule to solve complex arithmetic problems in maths. Unfortunately BODMAS doesn't work in C. Suppose you wrote one expression.

```
x*y+z*a
```

So how you will solve this expression?

This way (x*y)+(z*a)
Or this way x*(y+z)*a

To solve this problem in C you have to learn about hierarchy or precedence of operators. Hierarchy of some common arithmetic operators is given below.

Priority	Operators	Description
1st	* / %	multiplication, division, modular division
2nd	+ -	addition, subtraction
3rd	=	assignment

Some Quick Tips

- Within nested parentheses, arithmetic operations will be done according to above priority. Expression under innermost parentheses will be solved first.
- It is advised to use parenthesis in the program carefully. You should type both parentheses first ( ). After that start typing expression inside it. This will remove the errors caused by missed parenthesis.

Associativity of Operators

Associativity of operators is used when two operators of equal priority makes a tie. In that case the arithmetic operation is solved using the associativity table of C language.

Complete table of precedence and associativity of operators in C is given below.

	Operator	Associativity	Precedence
()	Function call	Left-to-Right	Highest 14
[]	Array subscript		
.	Dot (Member of structure)		
−>	Arrow (Member of structure)		
!	Logical NOT	Right-to-Left	13
~	One's-complement		
−	Unary minus (Negation)		
++	Increment		
−−	Decrement		
&	Address-of		
*	Indirection		
(type)	Cast		
sizeof	Sizeof		
*	Multiplication	Left-to-Right	12
/	Division		
%	Modulus (Remainder)		
+	Addition	Left-to-Right	11
−	Subtraction		
<<	Left-shift	Left-to-Right	10
>>	Right-shift		
<	Less than	Left-to-Right	8
<=	Less than or equal to		
>	Greater than		
>=	Greater than or equal to		
==	Equal to	Left-to-Right	8
!=	Not equal to		
&	Bitwise AND	Left-to-Right	7
^	Bitwise XOR	Left-to-Right	6
|	Bitwise OR	Left-to-Right	5
&&	Logical AND	Left-to-Right	4
||	Logical OR	Left-to-Right	3
? :	Conditional	Right-to-Left	2
=, += *=, etc.	Assignment operators	Right-to-Left	1
,	Comma	Left-to-Right	Lowest 0

if Statement – Part 1

Generally we take decisions by looking at the current circumstances.

Example:
If she says yes then I will dance.
If you like this tutorial then I will write the next one too.

So, mostly decisions depend on some kind of conditions. C language also uses decision control instructions. It has mainly three decision control instructions.

- if
- if-else
- switch

What are decisions in programming?

Till now we have only used sequence control instruction. In that statements are executed in the order they appear. But during serious programming we often need to run one set of instructions under one condition and entirely different set of instructions if the condition fails.

So the selection of instructions on the basis of some condition is called decision in programming.

Now we will proceed further to understand the first decision control instruction i.e. if

if Statement

Here if is a keyword. It is used to check conditions before executing a set of statements. Its general form is given below.

```
if(Condition)
{
Statement 1
Statement 2
…. And so on
}
```

The keyword if tells the compiler to follow decision control instruction. So the compiler checks the condition, if it turns out to be true then it execute the statements which are present in the body of if statement. But if the condition is false then compiler just skip body of if statement.

Now some questions may arise in your mind.

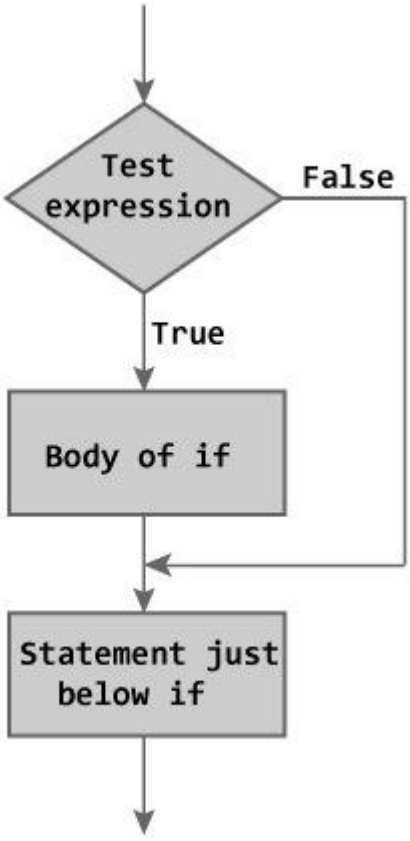
What are conditions? How can I show conditions?

In C, conditions are expressed using relational operators. By using relational operators we can compare two values – if it is greater than, less than, equal to and not equal to the other value.

Checkout below table to understand relational operators.

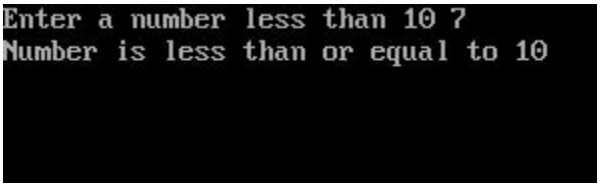
this expression	is true if
x == y	x is equal to y
x != y	x is not equal to y
x < y	x is less than y
x > y	x is greater than y
x <= y	x is less than or equal to y
x >= y	x is greater than or equal to y

A flow chart of if statement is given below.



```
/*Program of if statement */

#include<stdio.h>

void main( )
{
int n;
printf("Enter a number less than 10" );
scanf("%d",&n);
if (n<=10)
{
 printf ("Number is less than or equal to 10" );
}
}
```

Output



Lets try to understand this basic program.

Note: Now we will start analysing the program after main().

1. In the first statement we have declared an integer variable n.

2. In the second statement we just print message to enter a number less than 10 using printf() function.

3. In the third statement we are accepting value from the user and storing it in n variable.

4. Now in the fourth statement we have used a condition which is n<=10. It means the compiler will check whether the value in n variable is less than or equal to 10. In our test run we have entered 7, so the condition turns out to be true and a message is printed.

5. In case if you will enter value greater than 10 then it will skip the body of if statement and the program will end.

if statement is most frequently used in C programming. I hope you must have understand the basic concept of using if statement.

if Statement – Part 2

In the last tutorial we have learnt about the basic use of decision control instruction using if statement. Today we will extend that tutorial with some advanced examples.

A bit more about conditions

As I told you earlier that if statement is used with some conditions. Truly speaking it can also be used with some expression.
So the general form of if statement is

```
if(expression)
{
Statement 1
Statement 2
. . . . .
. . . . .
}
```

What is expression?

This expression can be any valid expression including relational and arithmetic expression. Some common examples are given below.

```
if(3+2)
printf("This will print everytime");

if(6/2+4)
printf("This will print everytime");

if(0)
printf("This will never execute");
```

Conclusion

Truly speaking in C language any non-zero number is considered to be TRUE whereas zero is considered as FALSE.

During the running of decision control instruction, a compiler checks the expression either it evaluates 0 or not. If the result is 0 then it will never execute the body of if statement. But if the expression evaluate to any non-zero number then it will execute the body of if statement.

In the first two examples, expression under if evaluates a non-zero value. So it will print the message.

In the last example, I have written 0. So the printf() statement is not executed.

Note: Usage of expression is very important in C language. While writing big programs it is used very frequently. So I recommend you to understand the basic logic behind it correctly. If you have any problems then you can also post your queries in comments.

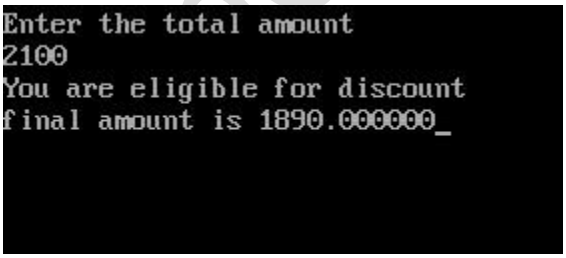
Now lets take one slightly complicated program to understand all the basics associated with if statement.

Sample Program

Make one program which will display the total amount after deducting the discount of 10%, if the total is more than 2000.

```
/*Program to show discount*/
#include <stdio.h>

void main()
{
 float amnt,famnt,dis;
 printf("Enter the total amount\n");
 scanf("%f",&amnt);
 famnt=amnt;

 if (amnt>2000)
 {
  printf("You are eligible for discount\n");
  dis=0.1*amnt;
  famnt=famnt-dis;
 }
 printf("final amount is %f",famnt);
}
```

Output



Now let's try to understand this program

1. I hope you must understand the basic initial steps of this program. I will give the explanation for only if statement.

2. In if statement I have given the condition of (amnt>2000), it means it will check the amount if it is greater than 2000. Say, if it is greater than the amount then it will execute the statements in the body of if. Otherwise it will skip it.

3. In our test run we have given the value 2100, as it is greater than 2000. So the control passes inside the if statement body and do following things.

- First it will display the message "You are eligible for discount"
- In the next step it will calculate the discount i.e. 10% of the total amount.
- In the final step it will deduct the discount from the final amount.

4. Now in the last step it will display the final amount.

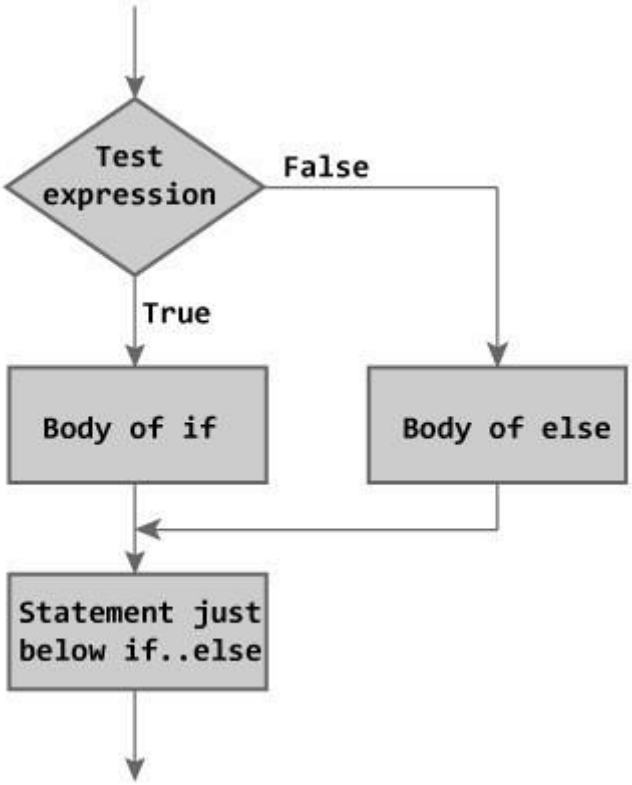
if-else Statement

Every coin has two faces and this is also true for our problems. Till now we have learnt about if statement in which we can execute a set of statements by giving some condition or expression.

But most of time we need to execute one set of statements if the condition is true, and an entirely different set of statements if the condition is false. This task can be accomplished easily in C language by using if-else statement.

Syntax of if-else statement

```
if(condition)
{
Statement 1
Statement 2 and so on
}
else
{
Statement 1
Statement 2 and so on
}
```



Few points about if-else block

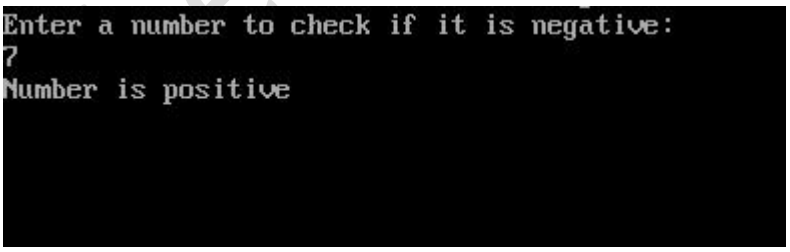
- The statements inside if keyword is collectively called if block.
- The statements inside else keyword is collectively called else block.
- The parenthesis inside if-else block can be dropped, if there is only one statement in it. As the default scope of these keywords is only one statement.
- No condition is used with else keyword. The statements under else will be executed only if the condition with if statement is turn out false. So it is treated as default choice.

The best way to understand a topic is through a program. Let's make one program using if-else statement.

Program to check a negative number

```
#include <stdio.h>

void main()
{
 int num;
 printf("Enter a number to check if it is negative:\n");
 scanf("%d",&num);
 if(num<0)
 {
  printf("Number is negative");
 }
 else
 {
  printf("Number is positive");
 }
}
```

Output



Explanation

Statements in beginning are self-explainable. I hope till now you are also familiar with them.

- The main logic of the program lies inside if-else block. In the if block i have written a condition which is checking, if the number is negative. If it turns out to be true then the message "Number is negative" is printed, otherwise compiler will skip this block.
- In the next else block I have written only one printf() function which will print the message "Number is positive".
- Checkout I have written no condition with else keyword. As I said earlier else block is used as the default block.
- In our test run I have entered the number 7 which is positive. So after taking the input, compiler checked the condition. It turns out false so it executed the statements under else block.

Nested if-else

In the previous tutorial we have learnt about if-else statements. Those statements provide the flexibility to check for two possible faces of answer. But it is also possible that there will be more than two options for the answer. To make things more precise, take an example of school grading system. Suppose if you want to create a program that will print the grade of student based on his/her percentage marks. Then in that case you will need at least 4 conditions to check.

So to put those things on work we have to use nested if-else statements.

What are nested if-else statements?

As the name suggests, nesting means writing if-else statements inside another if or else block. Basically there is no limit of nesting. But generally programmers nest up to 3 blocks only.

General form of nested if-else statements is given below.

```
if (condition)
{
Statement 1
Statement 2 and so on
}

else
{
 if (condition)
  {
  Statement a
  Statement b
  }
 else
  {
  Statement c
  Statement d
  }
}
```

It's a bit complex structure for beginners so lets try to understand its general form.

As you can see, I have nested another if-else block inside one else block. So while executing in this form. The compiler first check the condition if (primary or first) block. If it fails then it will move on to the else block. In the else block it will check the condition of secondary if statement, if it also fails then it will execute the statements under else block.

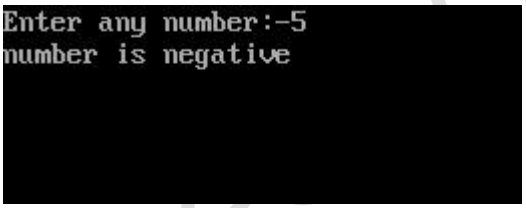
In our general form I have nested if-else block in else block. You can also nest that if-else statement under first if statement. This will also work.

Now lets try implement our knowledge in one program.

Question: Take one number from the user. Check it whether it is negative, zero or positive and print the message for it.

```
#include <stdio.h>

void main()
{
 int num;
 printf("Enter any number:");
 scanf("%d",&num);

 if (num < 0)
  printf("number is negative");

 else
 {
  if (num==0)
   printf("number is 0");
  else
   printf("number is positive");
 }
}
```

Output



Explanation

I have deliberately written that program which is similar to the last one. As I want to show the importance of if-else nesting inside C programming.

Initial statements of the program are self-explainable. So I will start my explanation with if-else statement.

- In the first if statement, I have given a condition i.e. num<0, it will check the number whether it is negative or not. If it is negative then it will print the message

"number is negative". But if condition fails then it will skip the statements under if block.

- Am I forget to write curly braces {} after if statement? The program will also work by dropping them in our case. This is because the default scope of if statement is one statement after it. As I have written only one statement, so there is no need to use curly braces in it. However if you write more than one statements under that block then you have to insert curly braces.
- After that I gave another else block. And inside that else block I have nested one if-else block. It means after entering the control inside else block it will check the condition of 2nd if block. If the condition turns out to be true then it will print the message "number is 0". Otherwise the control will transfer to the else block and it will print "number is positive".
- In the above program you can clearly see the secondary if-else block. Because I have indented the 2nd if-else block to increase the readability of program. So it is advised to use indentation while using nested if-else.

Logical Operators

Nested if-else statement helps in creating multiple choices inside a program. But there are also some disadvantages of using them which are given below.

- Indentation is most important while using nested if-else statements. But if we add 2 layer nesting inside on if-else block. Then you will notice that your program is creeping towards right. And it will make difficult to read that program.
- Braces are used to show the scope of statements. There are chances that you may commit a mistake while writing those braces. One single mistake will fill your program with lots errors.
- As there are multiple if-else statements so you also have to take care of matching the correct else block with if statement.

So these are some serious problems while writing nested if-else statements. But wait! It is not compulsory to use nested if-else statements every time. We can accomplish the same task by using Logical operators.

What are logical operators?

Basically C uses three logical operators which are AND (&&), OR (||) and NOT (!). I hope you must have learnt about the Boolean Algebra. These three operators are also used in that.

Remember while writing AND and OR operator, it should be written with two symbols which are && for AND and || for OR. Single symbol has completely different meaning in C programming.

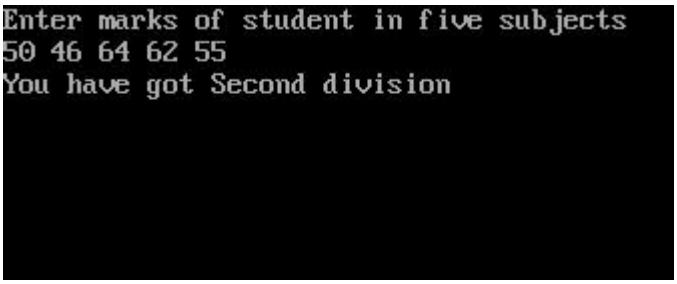
So now lets take one example to understand the use of logical operators.

Question: Make a program to print the division of student. Take marks from the user. Calculate the percentage and calculate their division.

```
#include<stdio.h>

void main( )
{
 int  n1,n2,n3,n4,n5,perc;
 printf("Enter marks of student in five subjects\n");
 scanf("%d %d %d %d %d",&n1,&n2,&n3,&n4,&n5);
 perc=(n1+n2+n3+n4+n5)/5;

 if(perc>=60)
  printf ("You have got First division");
```

```
 if((perc >=50)&&(perc<60))
  printf ("You have got Second division");

 if((perc>= 40)&&(perc<50))
  printf("You have got Third division");

 if(perc<40)
  printf("Sorry you are Fail");
}
```

Output



Explanation

- In the initial steps it is asked to enter student's marks. After that I have calculated the percentage of the student based on the marks. Now I have to check the division of student.
- Now in the first if statement I have given the condition if (perc>=60) it will check if the percentage is greater than 60 then it will print the message "You have got first division".
- Now in the next if statement I have written the condition if ((perc>=50) && (perc<60)). So basically there are two condition inside it and they are joined through logical operators. Remember the statements under this if block will only execute when both the conditions is true because they are joined with AND operator.
- Similar structure is adopted in the next if statement too.
- In the last if statement percentage is checked whether it is less than 40. If it turns out to be true. Then it will display the message "Sorry you are fail".

else if Clause

Till now we have learnt about the if statement, if-else statement, if-else nested statements and logical operators. Logical operators are good alternatives of nested if-else statements. But there are still some cases when we have to use nested if-else clause. So to make those situations a bit easy, Dennis Ritchie introduced else if clause.

What are else if clause?

Well they are no major difference between nested if-else and else if clause. As I told you earlier, the main problem with nesting is that it makes the program difficult to read. So to avoid that issue, else if clause is introduced.

General Syntax of else if clause is given below.

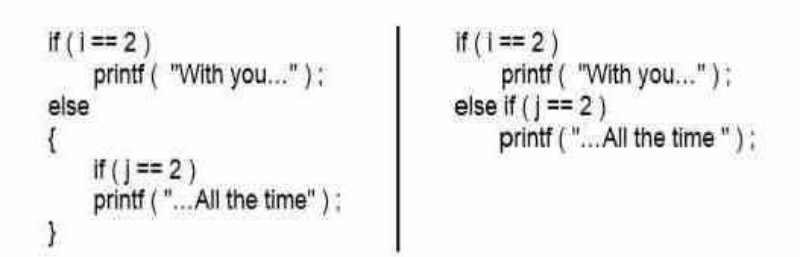
```
if (condition)
{
Statement 1
Statement 2 and so on
}

else if (condition)
{
Statement 1
Statement 2 and so on
}

else if (condition)
{
Statement 1
Statement 2 and so on
}

else
{
Statement 1
Statememt 2 and so on
}
```

Things to remember while using else if clause

- You can use any number of else if clause in this ladder.
- Usage of else in the last is completely optional. It means its up to you that you either want to include it or not in your program.
- This does not destroy the readability of the program.

- By using else-if clause, our program does not creep to the right due to indentation.

Remember by using else if clause, no change will occur in the execution of program. It is just the re-write of nested if-else clause. You can understand this point by watching below example.



```
if ( i == 2 )
    printf ( "With you..." ) ;
else
{
    if ( j == 2 )
    printf ( "...All the time" ) ;
}
```

```
if ( i == 2 )
    printf ( "With you..." ) ;
else if ( j == 2 )
    printf ( "...All the time " ) ;
```

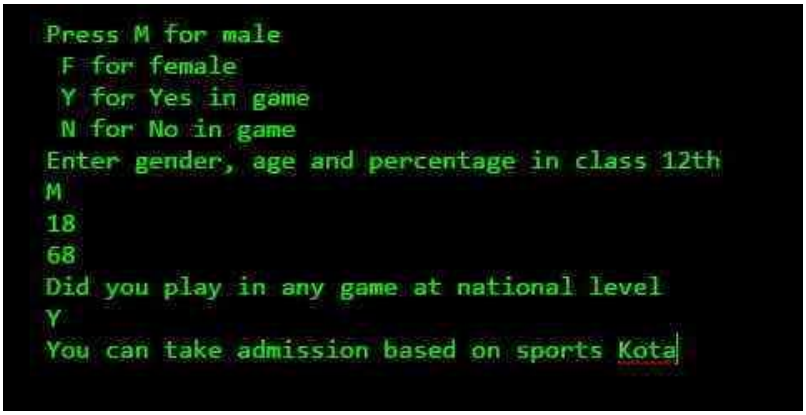
Lets try to understand this clause with one program.

Question: Make one program to check the eligibility of student to take admission in a college. The student must fulfil at least one condition to take admission in the college.

- Student should be male. His marks should be more than 80% in 12th and his age should be at least 18 years.
- Student should be female. Her marks should be more than 75% in 12th and her age should be at least 17 years.
- Student should be played at any national level game. Age and qualification doesn't matter in this case.

```
#include <stdio.h>

void main ()
{
 int per, age;
 char gen, game;

 printf("Press M for male \n F for female \n Y for Yes in game \n N for No in game \n");
 printf("Enter gender, age and percentage in class 12th \n");
 scanf("%c %d %d",&gen, &age, &per);
 printf("Did you play in any game at national level \n");
 scanf("%c",&game);

 if ((age>=18 && per>=80 && gen=='M') || (age>=17 && per>=75 && gen=='F'))
  prinf("You can take admission in our college.");
 else if (game=='Y')
  printf("You can take admission based on sports Kota");
 else
  printf("You cannot take admission in our college.");
}
```

Output



Explanation

- In the beginning we gave some instructions to user that how he/she should enter the data in our program.
- After that we printed the message and take some details from the student.
- Now basically we have 2 conditions. At first we have to check the student if he/she eligible for general kota and in the last we have to check if he/she is eligible for sports kota.
- To make things a bit clear I checked the results for general kota by using if keyword. I combined the conditions by using logical operators to make the program a bit compact.
- After that by using else if clause I have checked if the student is eligible for sports kota.
- If nothing works then I have also given the default else block to print the message "You cannot take admission in our college."

Operators Revisited - Hierarchy of Operators, NOT and Conditional Operator

Today we will re-visit the operators once again. In the tutorial of logical operators deliberately missed the NOT operator.

Why? Ah… it's a bit confusing and I don't want to ruin the next important topics due to that operator.

NOT Operator (!)

The NOT operator (!) is used to reverse the results. This operator is mainly used as a key in big complex programs. By using this operator we can reverse the condition easily. Lets try to understand it with an example.

If (!(y>6))

In the above statement I am writing a condition that y should be lesser than or equal to 6. I can also write the same condition as

If (y<=6)

Both the statements will give the same results. You can use anyone of them.

Hierarchy of Operators

I have given the hierarchy of operators after arithmetic operators. Now we have learnt about the logical operators (AND OR NOT) too. So the new hierarchy of operators is given below.

Operators	Type
!	Logical NOT
* / %	Arithmetic and modulus
+ -	Arithmetic
< > <= >=	Relational
== !=	Relational
&&	Logical AND
||	Logical OR
=	Assignment

Conditional Operators

They are also called ternary operators. As we have to use three arguments to use this operator.

General form of Conditional/Ternary operator

(Expression 1 ? expression 2 : expression 3)

It is generally used to avoid small if-else statement. Remember it is not the alternative of if-else clauses. It can used at some places.

Lets try to understand it with some simple example.

```
if (x==10)
   Y=3;
else
   Y=9;
```

In the above we are basically checking if x is equal to 10. If condition turns true then it will assign y as 3. Otherwise it will assign y as 9. The same task can be completed using ternary operator.

```
Y=(x==10 ? 3 : 9);
```

I hope everyone will agree with the fact that above example is very much compact than the earlier version.

Another example to use ternary operators is given below.

```
( x > 4 ? printf ( "Value is greater than 4" ) : printf ( "Value is less than 4" ) ) ;
```

Nested Conditional Operator

Well nested conditional operators are used very rarely but they are good to make the program compact.
A small example of nested ternary operator is given below

```
Small = ( x < y ? ( x > z ? 9: 10 ) : ( y > z ? 14: 16 ) ) ;
```

In the above example small is the variable and it will store

9 if x<y and x>z
10 if x<y and x<z
14 if x>y and y>z
16 if x>y and y<z

Sounds confusing? Well that's why they are used rarely. But like our example, it can sometimes make the program compact.

So that's all for decision control instructions. I recommend you to make programs and practice for at least 2 days before proceeding further. In the next tutorial I will cover an overview to loops in C programming.

while loop – Part 1

Till now we have learnt about decision control instructions or the programs which will execute as the way they are programmed i.e. sequentially. However its not compulsory that we want to do some task inside a program for only one time.

Suppose we want to make a program which will print your name 5 times. Then it will be very ineffective way to use the printf() function for 5 times. Instead of it we should look for something that will do that task for specific number of times automatically.

Well this is indeed possible. This is where loops comes in action. Loops are generally used to perform some task specific number of times. Similar to decision control instructions, in loops we also have to give some condition. And the program will execute task until the condition turns to be true.

There are three types of loops in C programming.

1. while
2. for
3. do while

while loop

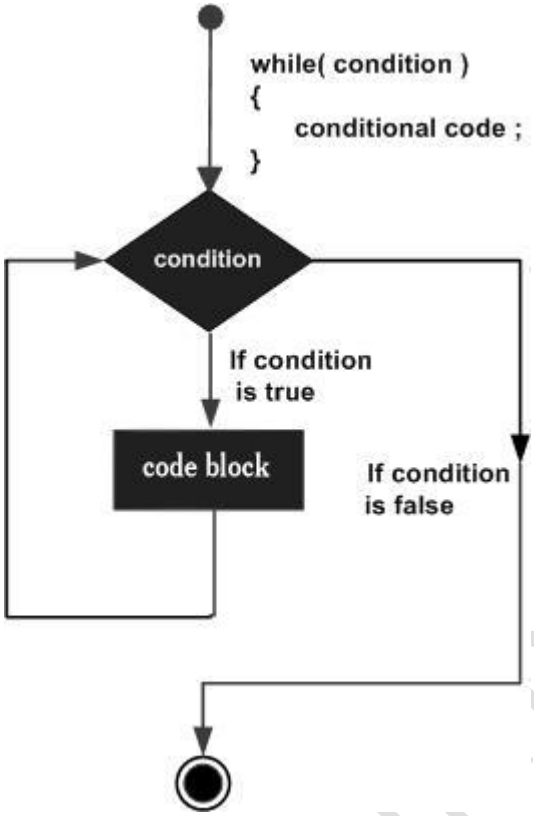
This is the first loop and the second most commonly used loop in programs. It is generally used to execute a set of statements fixed number of times. Generally programmers use this loop when they already know how many times the task should be executed.

Syntax of while loop

```
initialize loopcounter
while (condition)
{
execute this
and this...
increment loop counter
}
```

What is loop counter?

As I told you earlier that compiler doesn't know how many times you want to perform some task. So to instruct the compiler we use loop counter. As its name suggests, it counts and instruct the while loop to stop executing the task.



Lets try to understand while loop with a simple program.

C program to display a name 5 times on the screen

```
#include<stdio.h>

void main()
{
 int x=0;
 while (x<5)
 {
  printf("TheCrazyProgrammer.com \n");
  x=x+1;
 }
}
```

Output

Explanation

- In our program x is working as a loop counter. While declaring as a int variable I assigned a value 0 to it.
- After that I wrote the while loop with condition x<5. It means the loop will be executed till x is less than 5. To stop the loop x should contain the value 5.
- In the while loop I have given a printf() function to display the name.
- After that I have increased the loop counter by adding 1 to it.

Execution of Program

- In the beginning compiler will check the condition to enter the while loop. But as I have assign the value 0 to it. So the condition will turn true.
- After that the control will shift inside the loop and it will display the message.
- Now the loop counter which is x increased by 1. Then the new value of x will be 1.
- Now the compiler again check the condition of while loop. Now x contains the value 1 which is still true. Now it will again print the message and the value of x will again increase.
- The process will be carried out until x becomes 5. When x will become 5 then compiler again check the condition but this time it will return false and the while loop will end.

while loop – Part 2

In the last tutorial we have learnt about the while loops. Loops are used very frequently while writing big programs. And in the last tutorial we have learnt only the basics of the loop control structure. In this tutorial I will not write any program. Instead of it I will tell you about some tips and traps of using while loop in C.

If you have write any program using while loop for the first time. Then it is quite possible that you will get many errors. This tutorial mainly focuses to make you confident while writing loop control structure in programs.

Tips and traps of while loop

1. As I told you earlier the general form of while loop is.

```
initialise loop counter;
while(condition)
{  do this;
  and this;
  increment loop counter;
}
```

We can add any valid condition or expression with while keyword. Any expression which evaluates as non zero is said to be true and the expression which evaluates zero is said to be false.

2. We can use logical operators to describe condition. This is perfectly fine for all loops.

```
while(x<=60)
while(x>=50&&y<=75)
```

3. If you want to execute only one statement in while loop then you can also drop the curly braces { }. Default scope of while loop is one statement below the while keyword.

```
while(x<=20)
 i=i+1;
```

is same as

```
while(x<=10)
{
 i=i+1;

}
```

However if you want to execute multiple statements in while loop then it is compulsory to use curly braces.

4. Remember to increment or decrement the loop counter. Otherwise it will result in an infinite loop. One common mistake is given below.

```
main()
{
 int i=1;
 while(i<=10)
 {
  printf("%d", i);
 }
}
```

The above program will result in an infinite loop because we are not changing the value of loop counter. Condition is always true as value of loop counter remains same and the loop will get executed infinitely.

Correction of above code is given below.

```
main()
{
 int i=1;
 while(i<=10)
 {
  printf("%d", i);
  i=i+1;
 }
}
```

5. Never write a semicolon after while keyword. It will result in an infinite loop. A common error code is given below.

```
main()
{
 int i=1;
 while(i<=10);
 {
  printf("%d", i);
  i=i+1;
 }
}
```

Checkout I have given semicolon after while keyword. So do not make this mistake.

Few more operators

Post Increment/Decrement Operator

This operator is commonly used with loops. You must have noticed that most of the time we use expression i=i+1 to increment the loop counter. To make this a bit easy to write we can use post increment and decrement operator.

So i=i+1 is same as i++ (post increment operator).

and i=i-1 is same as i—(post decrement operator).

Pre Increment/Decrement Operator

This operator is similar to post. But with a small difference. This operator gives priority to incrimination in the function.

i=i+1 is same as ++i (pre increment operator).

i=i-1 is same as --i (pre decrement operator).

Sounds confusing? As both are same.

Well write one program with below function to understand difference between them.

```
i=1;
j=10;
prinf("%d %d \n", ++i, i++);
prinf("%d %d \n",--i, i--);
prinf("%d %d \n", ++j, j++);
prinf("%d %d \n",--j, j--);
```

Compound Assignment Operator

It is also similar to the above operators. The best way to understand them is by syntax.

i=i+5 is same as i+=5.

Or

i=i-5 is same as i-=5.

Or

i=i*6 is same as i*=6.

This operator can be used with arithmetic operators like +, -, *, / and %.

for loop – Part 1

Till now we have learnt about the while loop. We have also seen some common errors made by beginners while write loop control structure. If you see some big complex program then you will notice that program mainly contains only for loop. Now an obvious question which may hit your mind.

Why for loops are used more frequently than while loops?

Well the answer of this question resides in its syntax. For loop is generally used to implement a loop control structure, when programmer knows how many times he wants to execute a set of statements. Well it is not compulsory to use for loop in only those conditions. You can implement it anytime.

General Syntax of for loop in C is given below.

Syntax

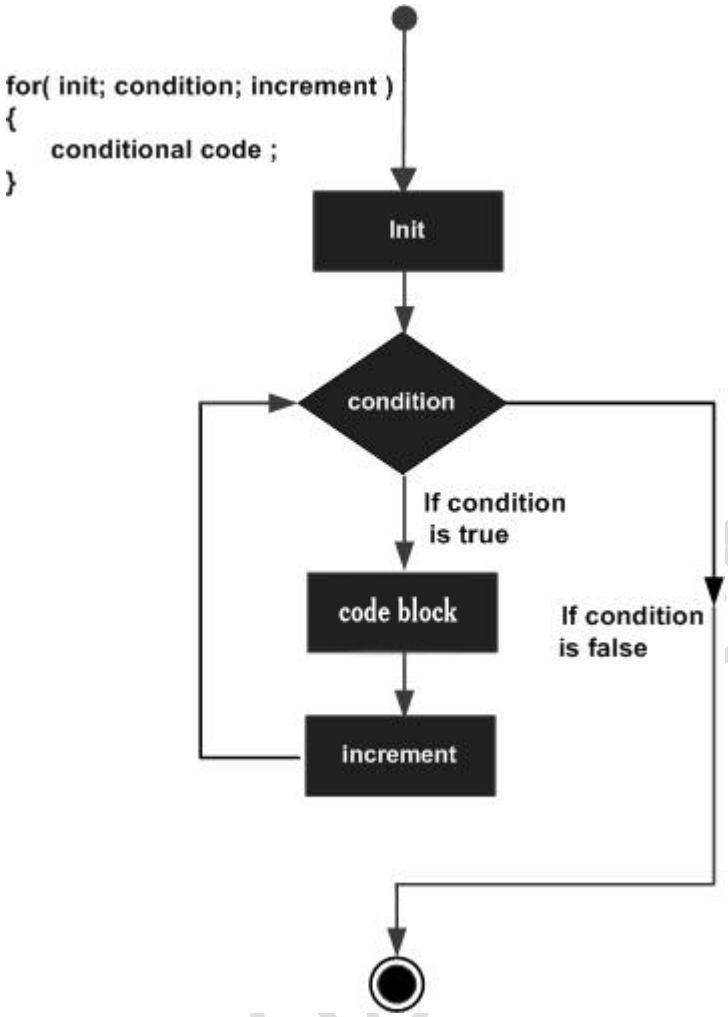
```
for(loop_counter_initialize; Condition; Loop_Counter_incremented)
{
 do this..;
 and this..;
}
```

As you can see the syntax of for loop is quite compact. While using for loop, one doesn't want have to write the loop counter explicitly.

So for loop is divided into three parts.

1. Initialization of loop counter
2. Condition of expression to check
3. Increment of loop counter

Truly speaking these 3 parts can replaced with any valid expression. It means we can also use functions like scanf() and printf() in spite of these 3 parts. We will learn about it in the next tutorial.



Anyways let's try to understand this loop with one example.

```
#include<stdio.h>

void main()
{
 int x;
 for(x=0;x<10;x++)
 printf("Say %d\n",x);
}
```

Output

Lets try to understand this program.

The program is quite easy to understand. In the beginning I have declared x variable as int. After that I have started for loop with 3 arguments (like in syntax). After that I have written printf() function. As I want to execute only one function after for keyword, that's why I dropped the braces too.

Explanation

- It is important to understand the execution of this program. As I have used for loop for this time. So lets get started.
- Program starts and entered in the for loop. I have initialize the for loop with the value x=0.
- After initialization the control will shift to check the condition. The condition turns to be true, as x is smaller than 10.
- Now remember after the condition checked, the control will shift to the printf() function (body of for loop). It will print the message on screen.
- Then in last the control will shift to the incrimination part. So the value of x will be incremented to 1.
- Now it will again check the condition. It will again turns true. So it will again print the message.
- This process will go on until the value of x becomes 10. When x holds the value 10 then it will fail with the condition. So the control will come outside the loop.

As I have said earlier for loops are used more widely than while loops. Its just due to its compact syntax. It is very important to understand the execution of for loop to write good programs using it.

For practice of this loop you can try below program.

Write a program to print your name 10 times. Try to analyse the execution of program by giving different conditions.

for loop – Part 2

Before proceeding to this tutorial I am assuming that you are familiar with for loop. If you didn't read my last tutorial then I strongly recommend you to read that. Anyways till now we have learnt about the basic use of for loop and its execution.

Today I will tell you about the basic variations while using for loop. All three things initialization, condition and loop counter increment are resides within the syntax of for loop. This is one of the key advantages of using it. Giving below are some variations while using it.

1. It is not compulsory to initialize the for loop with the syntax. I can also initialise it before.

```
int i=1;
for(;i<10;i++)
{
 printf("This is working");
}
```

Consider the above example carefully. You will notice I have not initialized variable i within for loop. But I have left the space for it. I have also given semi-colon for it. It is compulsory even if you do not want to initialize a variable within for loop.

2. The same condition is also true if I do not mention the third part in it. Consider the below example.

```
int i;
for(i=1;i<10;)
{
 printf("This is working");
 i++;
}
```

Notice that I have not increment the loop counter within for loop. Instead of it I am incrementing it inside the body of the loop. Again it is compulsory to give semicolon.

3. What if I drop both the parts? Consider the below example carefully.

```
int i=1;
for(;i<10;)
{
 printf("Is it correct?");
 i++;
}
```

Will it work or not? Well its perfectly fine. So from the above three points we conclude that the initialization and increment part is completely optional in for loop.

4. Use of post increment/decrement operator within for loop.

```
int i=1;
for(;i++<10;)
{
 printf("This is working");
}
```

Checkout the execution of the above program stepwise.

- i will initialized with value 1
- Condition is checked within for loop. It turns true.
- After that i will increment to 2.
- Now printf() function will display the message "This is working".
- Now again condition is checked with value i=2. It will turn true.
- After that i will increment to 3.

Note: Post increment operator is used with i. So first the condition will be checked. After that i will increment to 2. The reverse is also true for pre-increment/decrement operator.

5. Use of logical operators in for loop.

```
int i=1;
for(;i>0&&i<10;i++)
{
 printf("This is working");
}
```

You can check the loop for multiple conditions by using logical operators in it.

6. In the last tutorial I have told you that we can replace any valid expressions in these three parts. One good example of that is given below.

Shortest C program to print 10 numbers

```
#include<stdio.h>

void main()
{
 int i;
 for(i=1;i<11;printf("%d\n",i++));
}
```

Consider the above example carefully. I have inserted the printf() function in it. And I have also given a post increment operator with i. After the for loop I have given a semicolon. Because I don't want to execute a single statement in loop after that.

Loops are important for this C programming tutorial. So I recommend you to try your hands on for loop. Try to write programs with some variations to obtain on good conclusions.

You can try the below programs for practice.
Write the shortest program to print reverse counting from 10 to 1.

Nested loops

Nesting is one of the most complex topics of C programming. Just like decision control instructions, a loop control instruction can also be nested easily. But sometimes understanding the execution of such programs could be a bit tricky. So it is important to make the basics clear.

As I said in my earlier tutorials, nesting means defining statement under the scope of another similar statement. In case of loops, when we nest two loops then it generally multiplies the execution frequency of loops.

We can nest for loop inside while loop and vice versa is also true. Generally programmer nest up to 3 loops. But there is no limit of nesting in C.

Now let's try a small program with nested loops.

```
#include<stdio.h>

void main()
{
 int row,col;
 for(row=1;r<4;r++)
 {
  for(col=1;col<4;col++)
  {
   printf("%d\t%d\n",row,col);
  }
 }
}
```

Output

Explanation

- Every row executes a column at least 3 times. Therefore you are seeing all three values with every row.
- Once the control reaches inside the inner loop, it came out only when the loop is fully executed.
- We cannot alter the control once it reaches inside the inner loop.

Note: I have deliberately shown a simple program to show nesting of loops. Execution of nested loops can't be summarized in words. So it will be better if you try your hands on the nesting of loops to understand them better.

Odd Loop

There are situations when we need to execute a set of instructions until the user deny it. So we need to check if the user wants to repeat the set of instructions or not.

In that case programmer has no idea about the executing time and frequency of the program. This is called odd loop in C programming. So basically we have to make a program which will ask to the user about the re-execution of program.

```
#include<stdio.h>

void main()
{
 char yes='Y';
 int x,y;

 while(yes=='Y')
 {
  printf("Enter two values to perform addition\n");
  scanf("%d%d",&x,&y);
  printf("Sum is %d \n",x+y);
  printf("Do you want to continue ? Press Y for Yes and N for No\n");
  scanf("%c", &yes);
 }
}
```

Output

```
Enter two values to perform addition
5
6
Sum is 11
Do you want to continue ? Press Y for Yes and N for No
Y
Enter two values to perform addition
65
5
Sum is 70
Do you want to continue ? Press Y for Yes and N for No
N
```

Explanation

- In the beginning I have character variable yes with character initial value 'Y'. After that I have declared two more integer variables.
- Now I have started while loop with condition yes=='Y'. It means the loop will be executed until it receives value 'Y'.
- After that I have displayed the message to enter two values. By using scanf() function I have stored data inside the variables.
- Now instead of using a third variable to calculate sum. I have used a simple argument to calculate sum inside printf() function.
- In the end the program will display the message for user. If he wants to continue then he will press "Y". Otherwise he will press "N" and the program will be terminated.

Break and Continue Statement

Hello all, I hope till now you must be impressed with the power of loops in C programming. Well today I will tell you about the two keywords which are used very frequently inside loops.

These two keywords are given below.

1. break
2. continue

Lets learn about them one by one.

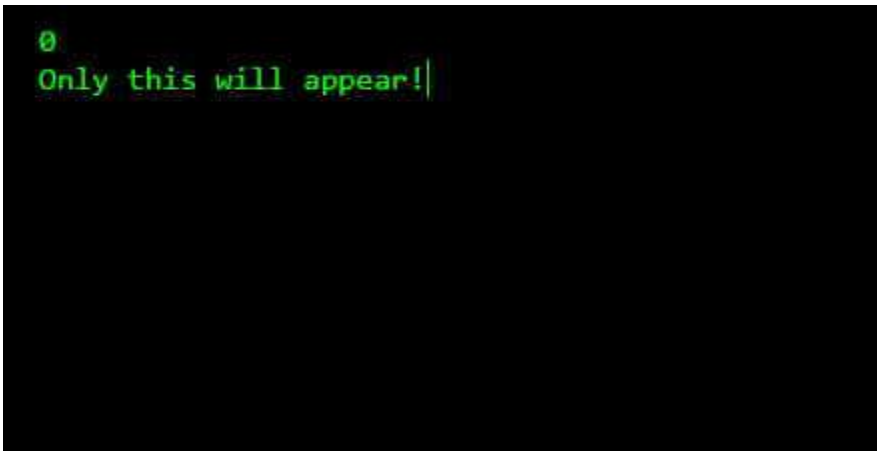
Break Statement

It is the keyword which is generally used to break the execution of loop. Sometimes we want to terminate the loop after getting the desired results. In those conditions, break keyword serves our purpose. After encountering the break keyword, control of program shifts to the next statement after the loop. They are used very frequently in the odd loops.

Lets take one example for better explanation.

```
#include<stdio.h>

void main()
{
 int i;

 for(i=0;i<5;i++)
 {
  printf("%d",i);
  break;
  printf("This will not appear!");
 }

 printf("\nOnly this will appear!");
}
```

Output



Explanation

- In the above program I have started a for loop. Inside the body of the loop I have written few statements. First of all the printf() function is executed which will display the first loop counter value.
- After that I have written break keyword. And just after that I have written another printf() function.
- But during the execution of program complier will not execute the second printf() function below break keyword. As I said earlier it will take the control of the program outside the loop.
- And in the last I have another printf() function which is outside the loop. This is executed because it lies outside the loop.

Continue Statement

It is a counter part of break keyword. Sometime while writing complex program we need a keyword which will take the control to the beginning. Continue keyword serves this purpose and it is used with loop. So whenever compiler encounters the continue keyword it takes the control to the starting of loop.

A program for continue keyword is given below.

```
#include<stdio.h>

void main()
{
 int i;

  for(i=0;i<5;i++)
 {
  printf("%d\n",i);
  continue;
  printf("This will not appear!");
 }
```

```
  printf("\nOnly this will appear!");
}
```

Output

Explanation

- I have just replaced the break keyword with continue. And it makes the big difference to the program.
- Execution of the program is similar to the earlier one. Except one thing that this time the loop will be executed fully. But it will not execute the printf() function after continue keyword.
- Because after the continue keyword the control moves to the beginning. So it will never reach to that function.
- In the end of the loop last printf() function will get executed.

Try This

You can try your hands on these two keywords by making simple programs.

- Write a program to check a prime number which is entered by user.
- Write a program to check composite number which is entered by user.

do while loop

Till now we have learnt the most frequently used loops in C programming which are for and while loop. Now lets learn about the third loop control instruction which is do while loop.

do while loop is just like a normal loop control instruction which executes a set of statements until the condition turns false.

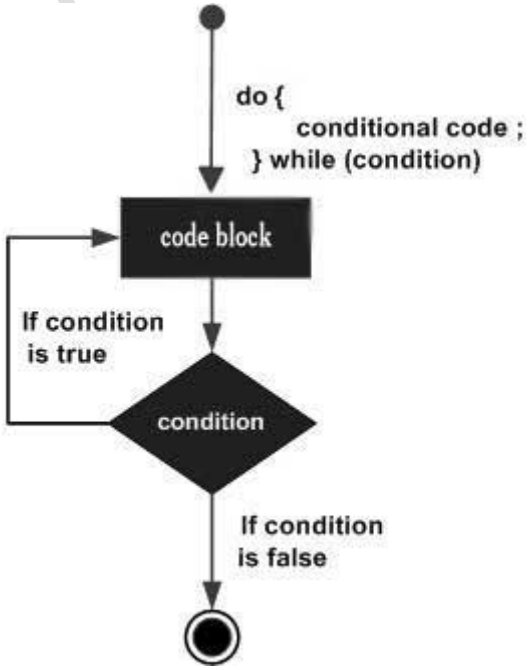
do while loop

Till now we have learnt that loops checks the condition first before executing the set of statements inside its body. But this loop is a bit different than others. It is compulsory that the statements under the body of do while loop will get executed at least once in a program. This loop control instruction executes its statements first, after that it checks the condition for it.

Why do we use do while loops?
Well that's a really silly question. You cannot predict anything before programming some big thing. While writing some complex big programs, there is a need of a loop which will execute statements at least once. That's why do while loop is introduced in C.

Syntax

```
do
{
        do this
        and this atleast once!
}while(condition);
```



Note: Remember writing semi colon after while keyword in do while loop. This is the most common mistake of a beginner C programmer.

Lets make one program to grasp this loop too.

```
#include<stdio.h>

void main()
{
	do
	{
		printf("OMG its working!");
	}while(7>10);
}
```

Output

Explanation

Well it's a very simple program to demonstrate the use of do while loops. As you can see the condition is wrong with while keyword. But still the printf() function will get executed once.

Usage of break and continue with do while loop

break: It will work similar to other loops. By introducing break keyword it will take control outside the loop. Remember it is not recommended to use break keyword inside loops without if statement.

continue: The functioning of continue will also work similar to other ones. However do while loop can make some good twist to the programs, if used properly with continue keyword.

This is the last loop control instruction of C programming. I recommend to try your hands on do while loops too. In the next tutorial I will tell you the best way to make Menu in C programming.

switch statement – Part 1

It is quite common that in day to day life we have to choose one option from several choices. Say if you have to choose only one desert out of the menu. Suppose if I give you a task to write a program for one menu. Probably you will use if-else ladder to make that. Well its good for now. But after reading this tutorial, I would suggest you to go with switch statements rather than anything else.

switch statement

This is the decision control instruction which allows the programmer to build a menu type program, so that the user can choose one options from multiple choices. It is also called switch-case-default instructions too because these are three keywords involved in making this instruction. Now lets head on to the syntax of switch statement.

Syntax

```
switch(integer expression)
{
 case constant 1: do this;
 case constant 2: do this;
 case constant 3: do this;
 default: do this;
}
```

Explanation of this syntax

integer expression: It is used to check if programmer wants to execute the set of statements under switch or not. It works similar to conditions but we can only give integer expressions insider. It means an expression which is a integer or which evolves an integer.

case: It is the keyword to show multiple conditions inside switch statement. We can have any number of cases.

constant 1,2 and 3: As its name suggests these values should be either integer constant or character constant. Each value must be unique.

do this: We can use any legal C programming statement at that place.

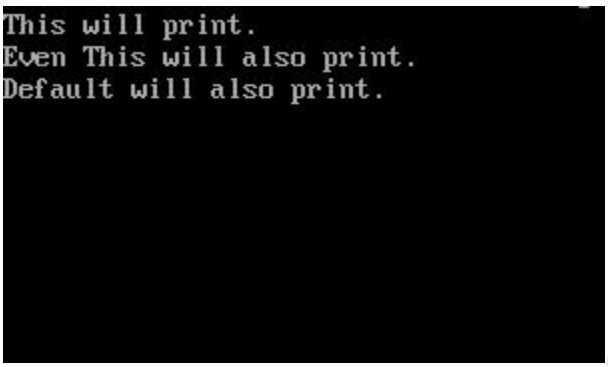
default: It is another keyword which is used to execute some default statements if all conditions fails inside switch.

Now lets apply the above knowledge in some program.

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

```
void main ()
{
 int i=2;

 switch(i)
 {
  case 0: printf("No Message");
  case 1: printf("No Message");
  case 2: printf("This will print. \n");
  case 3: printf("Even This will also print. \n");
  default: printf("Default will also print.");
 }
}
```

Output



Explanation

- The program starts with integer variable i with value 2 in it.
- After that I have written a switch keyword with one integer expression. i.e. switch(i). It means full switch statements will get executed if i=2 turns true.
- I think you should have an idea that case 2 will get executed.
- Now the turning point is after case 2, every statement get executed.
- Even the last expression under default is also get executed.

Now an obvious question which should hit your mind.

Why is it so?
Well this is not the demerit of this program. In fact it's the speciality of switch statements. It means if one condition turns true then all the subsequent conditions will also get executed inside switch statement. This is because it does not check any condition after getting a true value with one condition.

Switch statements are used very frequently in C programming. So I would recommend you to go through this tutorial at least once and make some programs by using switch keyword. In the next tutorial I will tell you about the method by which we can stop executing all the statements inside switch.

switch statement – Part 2

In the last tutorial I told you about the syntax and working of a program using switch keyword. Well in day to day programming we generally don't use that syntax in C. This is because if we use the earlier method then it will execute all the subsequent statements after the condition turns true.

switch statement

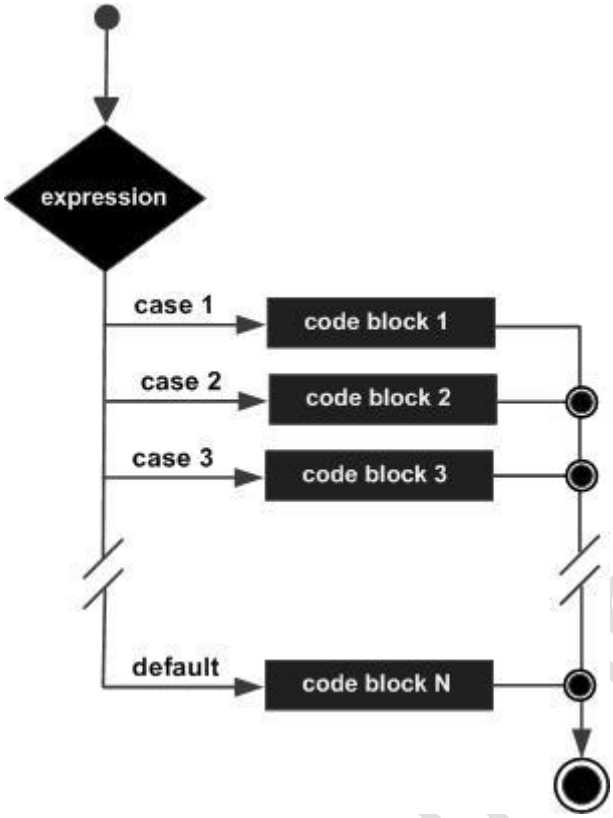
So we need a syntax that will execute only a certain set of statements when the condition turns true. So lets checkout the advance modification of switch statement with break keyword.

Syntax

```
switch(integer expression)
{
 case constant 1: Statement 1;
   break;

 case constant 2; Statement 2;
   break;
 . . . . . .
 . . . . . .

 default: Statement 3;
}
```

Explanation of the above syntax
Well the whole syntax is almost same. But in the above syntax we have added break keyword after the cases. Remember we generally do not use continue keyword with switch.

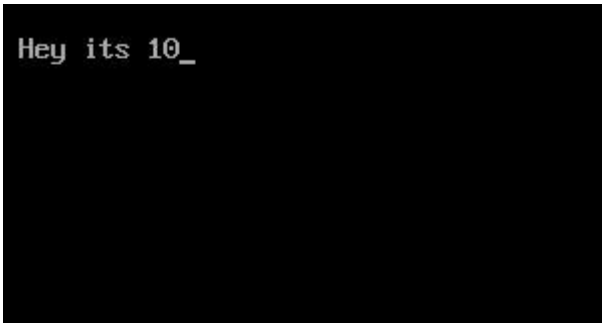
Working of break keyword in switch statement
As you can see I have given break keyword after every case. Its working in switch case is almost same as in loops. If the compiler encounters break keyword in switch then it will take the control to the outside of switch block.



The perfect way to understand it is through a program.

```
#include<stdio.h>

void main()
{
 int i=10;

 switch(i)
 {
  case 1: printf("Hey its 1");
   break;

  case 10: printf("Hey its 10");
    break;

  case 7: printf("Hey its 7");
   break;

  default: printf("Hey its default");
 }
}
```

Output



Explanation

- In the beginning of the program I have declared an integer variable with value 10.
- After that I have written switch keyword with integer expression i. It means the compiler will check the value of i (which is 10) with all the cases.
- In the first case I have written integer constant '1' which turns false. So compiler will skip that case.
- In the second case I have written integer constant '10' which turns true. So compiler will execute the statements under that case.
- Now just after the printf() function I have written a break keyword. So it will take the control of the program outside the switch block.

Points to remember

- You can also write multiple statements under each case to execute them. And you will not need to give separate braces for them, as the switch statement executes all statements once the condition turns true.
- Remember default keyword is optional. Its completely depend on us whether we want it or not in our program.
- The order of cases and default in the switch block do no matter. You can write them in any order but you must take care of proper use of break after each case.

switch statement – Part 3

In the last tutorial I told you about the practical use of switch keyword. As I said earlier a programmer generally use this keyword for menu driven programs. Today I will tell you about the tips and traps while using switch keyword. I will also compare switch with if-else ladder.

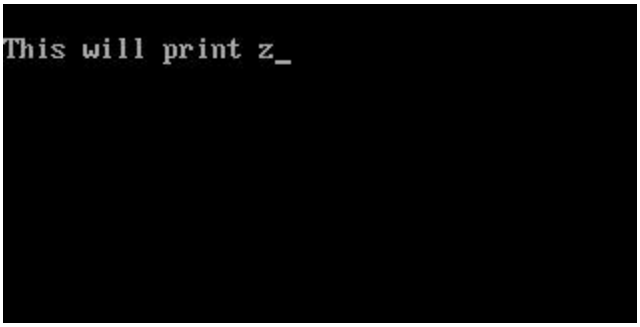
So lets start it one by one.

switch statement

1. It is not compulsory to make switch-case statement for only integer expression. We can also make it for characters too.

```
#include<stdio.h>

void main()
{
 char i='z';
 switch(i)
 {
  case 'a':
  printf("This will print a");
  break;

  case 'z':
  printf("This will print z");
  break;

  case 'p':
  printf("This will print p");
  break;

  default:
  printf("Sorry its a mismatch");
 }
}
```

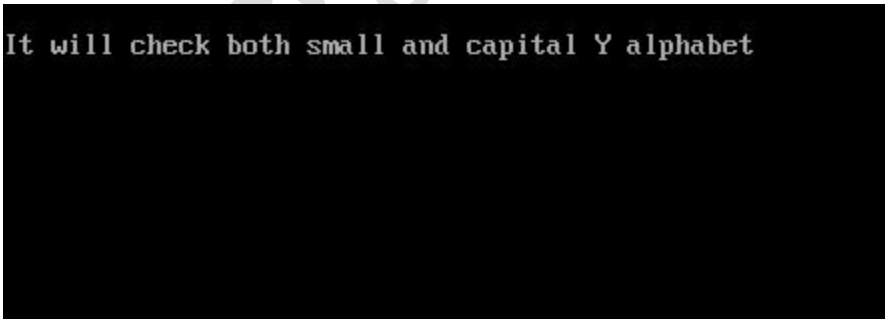
Output



2. We are not bound to write only one statement inside each case. And we can also put multiple conditions inside each case.

```
#include<stdio.h>

void main()
{
 char ans='Y';
 switch(ans)
 {
  case 'y':
  case 'Y':
  printf("It will check both small and capital Y alphabet");
  break;

   case 'n':
  case 'N':
  printf("It will check both small and capital N alphabet");
  break;
 }
}
```

Output



In the above program we have excluded the default column. As I said earlier its completely optional. You can also write multiple statements inside each case without worrying about the braces.

3. In case if you write a instruction inside switch case but it does not belong to any case. Then the compiler will skip that instruction at run time.

4. Switch statement can also be used to check the result of particular integer expression. E.g. switch(1*5*0+8*0)

5. You can also write nested switch statement but in practice it is used very rarely.

switch vs if-else

Advantages of using switch statement

1. It is a far better way of writing a program and it also gives well structured way to programs.

2. Switch statements works faster than if-else ladder because while the compilation of program, compiler generally generates a jump table by which it can easily check the answer instead of checking each condition.

Disadvantages of using switch

1. It doesn't allow to write conditions with operators.

2. It doesn't allow to write even floating point expression. e.g. switch (0.5) is not allowed.

3. A programmer cannot write multiple cases which will give same result.

```
case 8:
x=y;
break;

case 7+1:
k=2;
break;
```

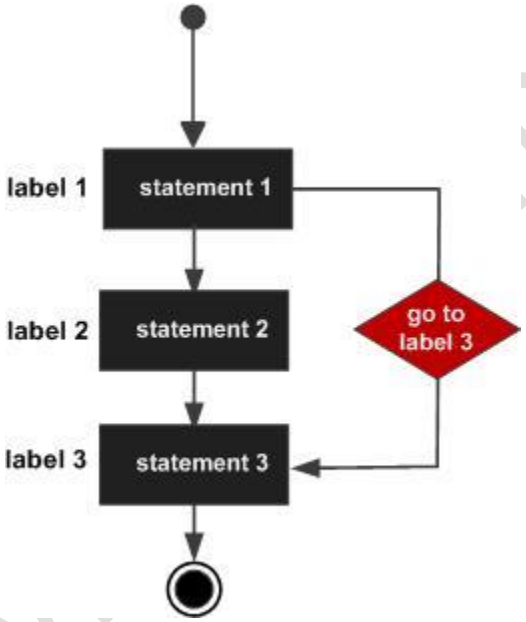
Above code will give an error. Because two cases will give the same result i.e. 8.

goto statement

By using goto statement you can transfer the control of program anywhere. This keyword is not recommended to use in any situation.

Syntax

goto label;
. . . .
. . . .
label: statement;

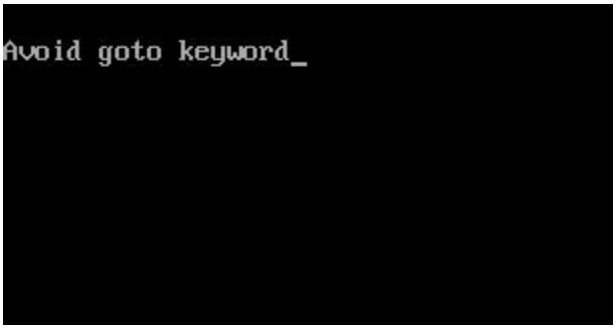


Lets make one program to understand it.

```
#include<stdio.h>

void main()
{
 int i=1;

 if(i==1)
  goto one;
 else if(i==2)
  printf("2");
 else
  printf("Its nothing");

 one:
 printf("Avoid goto keyword");
```

```
}
```

Output



Why you should avoid goto statement in C?
It takes the control of the program anywhere and anytime which is not good for software development life cycle. In this cycle a program has to debug and re-test a lot of times. And with the use of goto keyword it makes very hard to debug the program.

This keyword is generally used to take the control outside the loop. That's it. Otherwise you should avoid this keyword.

Functions – Part 1

It's a good approach if we build a program by dividing it into small modules known as functions. In today's tutorial, I will tell you about the basic use of functions.

So lets begin our quest to learn functions in C programming. The very first question that will hit your mind should be.

What are functions?

A function is a set of statements which are aggregated to perform some specific task. Generally we use functions to perform basic tasks in a generic way.
A good C programmer avoids writing the same set of statements repeatedly. Instead of it, a programmer makes a function and writes all the statements there and call that function whenever needed.

There are two types of functions.

Inbuilt Function
These functions are already defined to perform specific task. For example printf() to print value on screen while scanf() to read value. There are many other inbuilt functions.

User-defined function
The functions that are defined by the programmer or user are called as user-defined functions. In this tutorial you will learn how to define and use such functions.

In functions we have three parts.

Function declaration

return_type function_name(argument list);

Function declaration tells the compiler about the value that it will return, the name of the function and the arguments or values that will be passed to the function. Passing the values is optional so you can skip argument list passed. If you don't want to return any value then just write *void* instead of *return_type*.

Function Definition

```
return_type function_name(argument_list)
{
 Body_of_funtion;
 . . . . . .
 . . . . . .
}
```

It defines the actual body of the function and the task that it will perform.

Function calling

function_name(argument list);

This statement will call the function and the control of the program will go to body of the function. After executing all the statements in the function it will come back where calling was done.

Lets checkout the simple C program with two functions.

```
#include<stdio.h>

//funtion declaration
void msg();

void main()
{
 printf("Hello All");

 //funtion calling
 msg();
}

//funtion definition
void msg()
{
 printf("\nTheCrazyProgrammer");
}
```

Output

Explanation

- As I said in earlier tutorials, main() is also a function. Every C program starts with the main() function. It is also called starting function and we cannot alter the control from it in the beginning. Our above program also starts with main() function.
- In the main() function I have printed the message "Hello All" using printf() function.

- After that I have called the function msg() which is created by me. Carefully look I called the msg() function by writing msg();
- After encountering the call to msg() function, the control shifts to the msg() function.
- Now a message "TheCrazyProgrammer" is printed on the screen.
- Again control reaches to the main() function. As there are no statements left in the main() function. So the program comes to end.

While transferring the control from main() function to msg() function, the activity of main() function is temporarily suspended. In our above program main() is calling function and msg() is called function.

Function is one of the most important topics in C programming. You cannot write efficient programs without the proper knowledge of functions in C programming. So I recommend you to go through this tutorial at least once to make everything clear. In the next tutorial I will tell you about the multiple calls within one function.

Functions – Part 2

In my last tutorial I gave an overview to the functions in C programming. Today I will tell you about the multiple calls within one function and its nuances. So without wasting any time lets head over to a program.

```
#include<stdio.h>

void firstfu();
void secondfu();
void thirdfu();
void fourthfu();


void main()
{
 printf(" Control is in main\n");
 firstfu();
}

void firstfu()
{
 printf(" Now the control is in firstfu function\n");
 thirdfu();
}

void secondfu()
{
 printf(" Now the control is in secondfu function\n");
 fourthfu();
}

void thirdfu()
{
 printf(" Now the control is in thirdfu function \n");
 secondfu();
}

void fourthfu()
{
 printf(" Now the control is in fourthfu function \n");
}
```

Output

```
Control is in main
Now the control is in firstfu function
Now the control is in thirdfu function
Now the control is in secondfu function
Now the control is in fourthfu function
```

A simple concept to learn
Suppose main() function calls another function msg(). When the control reach to the msg() function then the activity of main() will be suspended temporarily. After executing all the statements from the msg() function, the control will automatically goes to the original calling function (main() in our case).

Explanation

- The program starts with main() function and it will print the message inside it. After that I have called the firstfu() function from main(). Remember here main() is calling function and firstfu() is called function.
- Now the control reaches to firstfu() function and the activity of main() function is suspended temporarily. Inside firstfu() function, printf() will print a message. Now I have called thirdfu() function. Remember here firstfu() is calling function and thirdfu() is called function
- Now the control reaches to the thirdfu() function and the activity of firstfu() function is suspended temporarily. And it will execute its printf() function. Now I have called the secondfu() function.
- Same thing will happened to the secondfu() function and it will call the fourthfu() function.
- Now the printf() statement will be executed inside the fourthfu() function. As there is no further call to other function there. So the control will goes back to the calling function. Calling function of fourthfu() is secondfu().
- No more statements is left in secondfu(). So the control will reach to the calling function. thirdfu() is the calling function of secondf().
- Same execution will be done for all of them. And the control will reach to the main() function again. Now no statement is left in main() function, so the program will be terminated.

Points to Remember

1. Any C program contains at least one function and it should be main().
The execution of the program always starts with main() function. We cannot alter it in any case.

2. A typical C program may contain any number of functions. But it should contain one main() function and the program will start executing with main() function.

3. Any function can call any other function. Even main() function can be called from other functions. To call a function we have to write the function name followed by a semi colon.

```
void main()
{
  msg();
}
```

4. We should define a function before calling it. A function can be defined by following below syntax.

5. We can call one function any number of times.

```
void main()
{
  msg();
  msg();
}
```

6. A function can also called by itself. Such an execution is called recursion.

7. We cannot define one function inside another. We can only call one function from another.

Functions – Part 3

So far we have learnt about the simplest use of functions in C. In serious C programming functions are not used in that way. We have to make them flexible so that we can customize the results as per our requirements. To make generic function we have to pass some values to them. These values are also called parameters or arguments. Based on these parameter our function should return the value to the calling functions.

To make things a bit clear, we want to make such functions which can communicate to its calling function. And it should return the results as per the customization.

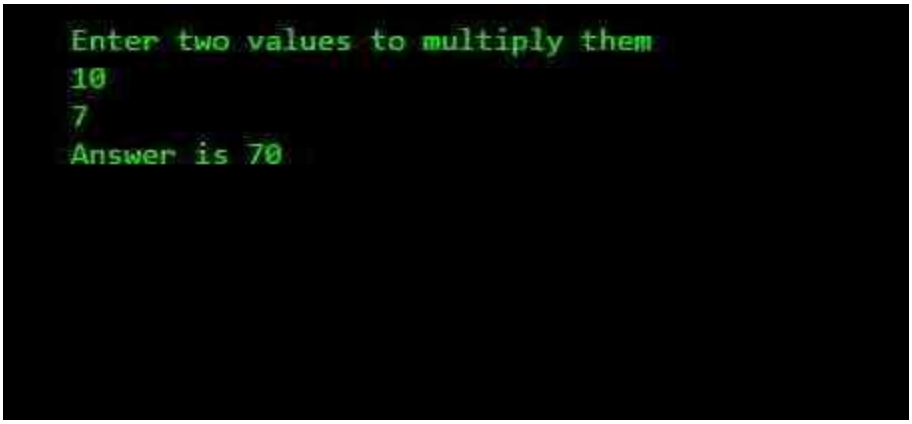
Till now we have used the functions like printf() and scanf() in which unknowingly we have passed some arguments like variable names to print it on the screen. We have to obtain similar results in our function. So today I will tell you about passing the values to the functions.

Passing Values to Functions

Lets understand this concept through a program.

```
#include<stdio.h>

int multi(int,int);

void main()
{
 int x,y,mul;
 printf("Enter two values to multiply them\n");
 scanf("%d%d",&x,&y);
 mul=multi(x,y);
 printf("Answer is %d",mul);
}

int multi(int a,int b)
{
 int ans;
 ans=a*b;
 return(ans);
}
```

Output



Explanation

1. In the statement above main() function I have declared the function multi() by writing the instruction *int multi(int , int);*

int: It is return type. It means which type of value the function should return to the calling function. In this function I have declared that it will return integer value.

multi: It is the name of the function. You can give any name to this function (valid identifier).

(int,int): These are the number of arguments that I will take from the calling functions. I have declared the data type of two arguments as integer. Here I am taking only two arguments, you can take any number of arguments.

2. It is compulsory to declare the function before using it. So that compiler should understand that we will define some custom functions in it.

3. In the first three statements of main() function I have declared some variables and taken some values in it from the user.

4. Now I have passed two parameters or arguments to the my function multi() with the statement *mul=multi(x, y);*

Here, *multi* is the name of the function, *(x, y)* is the arguments that I am passing to the multi() function. These should be integers because as I have declared in the definition of multi() function that I will receive two integer values in it. *mul* is the variable which will store the value returned by multi() function.

5. Now the control goes to multi() function and the values of variables x and y will automatically be copied in the a and b variables.

6. Now the multiplication takes place inside the multi() function and the result will be stored in *ans* integer variable.

7. In the last statement I am returning the value stored in *ans* variable to the calling

function i.e. main(). It is done by using the statement *return(ans);*. Here *return* is a keyword that returns a single value. It can be also written as *return ans*.

8. After returning the value the control will again come back to main(). You must remember that as the return statement is encountered the control immediately come back to calling function.

9. Now in last I am printing the answer using printf() function.

I would recommend you to go through the above at least twice to make your basic concepts clear. It is very necessary to understand this concept before proceeding to the further tutorials. If you are finding difficulty in understanding anything then you can ask your question by commenting below.

Functions – Part 4

In the last tutorial I told you about the passing values to the functions. I hope you had read that tutorial at least twice till now. So today I will tell you about some typical nuances of using functions in C language. I will also tell you about the scope rule of functions. So lets start with our first objective

Rules for using functions in C

1. The variables that are passed by main() function are called actual arguments or parameters and the variables in which the user defined function receives those values is called formal arguments or parameters.

2. We can pass any number of actual arguments. But it should be in the same number and order to the formal arguments.

3. We can also use the same variable name for both formal and actual arguments but the compiler will treat them different.

4. Compiler automatically passes the control to the calling function when it encounters the closing brace of user defined function. There is no need for some return statement for it.

5. However if you want to return some value to the calling function then you should use return statement for it.

6. You can use any number of return statements in your program. There is no restriction for that.

7. Some variations while using return statement

```
return(b);
return(4);
return(2.67);
return;
```

Last return statement will return any garbage value to the calling function. And we can drop the parenthesis in that case.

8. We can only return one value at a time. It's a restriction in C. So below statements will not work.

```
return(x, y);
return(6,4);
```

9. IMPORTANT – When we change the value of formal arguments then those values will not change in actual arguments.

E.g. Suppose we pass two variables to some function. And in that function I received the values in that variables in some local variable. So when we change the values of those local variables the it will not affect to the values of actual variables.

Scope Rule of Functions

In this rule I will tell you about the reason behind point number 9 in above rules. Till now you must be asking that why we should pass the values to some function explicitly?

The simple reason is that the default scope of variables inside some functions is local to it. It means the variables inside main() function can only be accessed in main() function. We cannot access them in any other function. So that is why we have to pass values to the user defined functions explicitly.

Functions – Part 5

I hope till now you must be having a good grip on the basic concepts of functions. If its not then I would suggest you to again read our earlier tutorials. In today's tutorial I will introduce to the first advance feature of functions.

Basically there are three advanced topics of functions

1. Function declaration and prototypes
2. Call by value and call by reference
3. Recursion

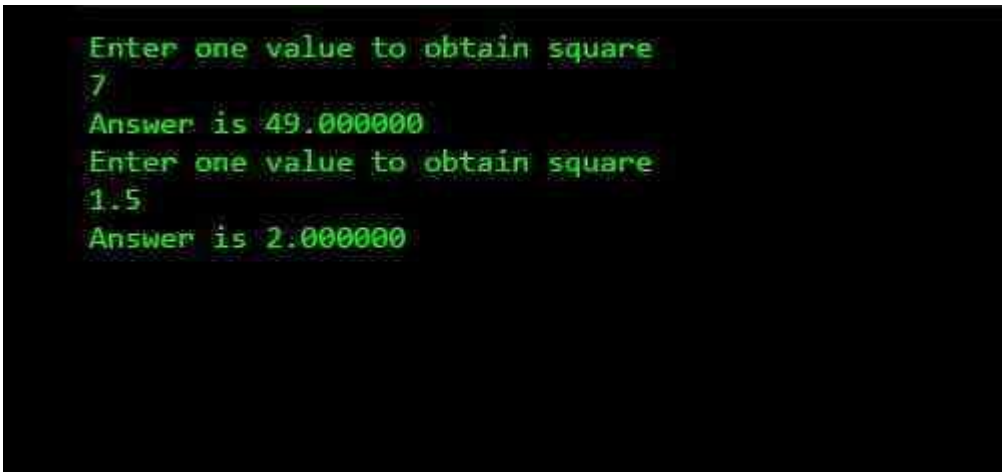
Today I will tell you about very first topic "Function declaration and prototypes". I have already covered the half of this topic in my earlier tutorials but I will give you an overview of that again.

Function Declaration and Prototypes

By default any C function will return an integer value to its calling function. Even if we don't specify any prototype for the function then it is considered that it will only return integer value. To make some change in the return value we have to declare its prototype first. Lets understand with one simple example.

```
#include<stdio.h>

squareval(float);

void main()
{
 float x, y;
 printf("Enter one value to obtain square \n");
 scanf("%f",&x);
 y=squareval(x);
 printf("Answer is %f", y);
}

squareval(float a)
{
 float b;
 b=a*a;
 return(b);
}
```

Output



I have executed above program with two values. At first I have entered 2 and it gave me correct answer 4 for it. But in the next run when I entered 1.5, it gave me wrong result by replying 2 as an answer.

Can you tell reason why it is giving wrong results for float value?

As I said earlier that by default a function always return an integer value. In our program we have not given any return type for the function squareval(). So it is returning an integer value to the main() function.

To rectify this we have to specify the return type as float in the prototype of the squareval() function. Correct version of above program is given below.

```
#include<stdio.h>

float squareval(float);

void main()
{
 float x, y;
 printf("Enter one value to obtain square \n");
 scanf("%f",&x);
 y=squareval(x);
 printf("Answer is %f", y);
}

float squareval(float a)
{
 float b;
 b=a*a;
 return(b);
}
```

Output

```
Enter one value to obtain square
9
Answer is 81.000000
Enter one value to obtain square
1.5
Answer is 2.250000
```

Checkout now its giving correct results for float values too. Above the main() function I have declared the function squareval() with return type float. After that while defining the sqaureval() function I have again written the return type as float.

So by giving the prototype of the function we can return values as per our desired data type.

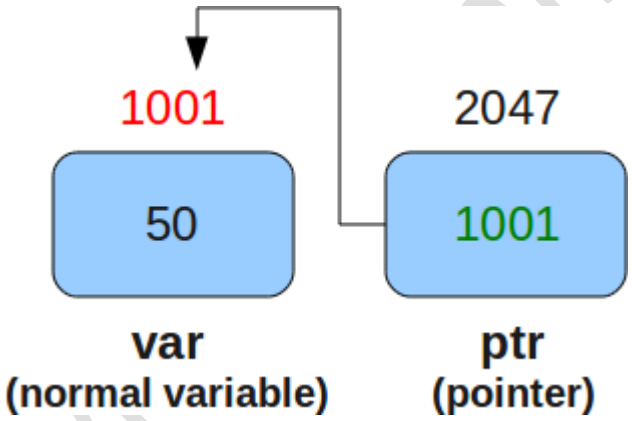
We can also declare function prototype with user defined data types like structure. We will learn it in our upcoming tutorials.

Pointers

Today I will not introduce you to any advance feature of function. Before proceeding further to our next tutorial about call by value and call by reference, it is compulsory to learn the basic concept of pointers. It is used in the advance feature of function that is call by reference. Pointer is one of the most difficult concept in C programming. So I recommend you to read this tutorial with concentration. In this tutorial I will only give you a basic overview of pointers.

What are Pointers?

In the first tutorial I told you how values are stored in C. I told you that every variable has an address. So a pointer is variable which stores address of another variable. Remember here address term is quite important. Pointers can only store addresses of other variable.



int x=10, y=20;
printf("%u %u", &x, &y);

Here %u is a format specifier. It stands for unsigned, so it will only display positive values.

You will get output of the above program like below.
605893 605897

Well these are the addresses of the variable x and y. Here & is the "address of" operator. It is used to take the compiler to the address of variables. Address of any variable can't be negative. This is the reason we have used %u format specifier to print the address of variables on the screen.

value at address (*) Operator

This is the second operator used for pointers. It is used to access the value present at some address. And it is used to declare a pointer.

Note: Pointer values and integer values are entirely different. We cannot store any address in integer variable. To store address of any variable we have to declare one variable as pointer.

Declaration and initialization of pointers

```
int x=10;
int *y; // Declaration of Pointer variable
y=&x; // Storing address of x variable in y pointer variable
```

Usage of pointers

```
int a=3;
int *b;
b=&a;
printf("Value at address %u is %d", b, *b);
```

The output of above code will be something like given below.
Value at address 605764 is 3

Explanation

- In the first statement I have declared the integer variable i and stored the value 3 in it.
- In the second statement I have declared b variable as pointer by using (*) operator. Remember that it can store only the address of integer type variable.
- After that I have stored address of variable a in pointer variable b.
- Now in printf() function I have printed the address inside b pointer and value at address inside pointer variable b.

Note: In our program pointer variable b contains the address of integer variable a. To store address of float, char or any other type variable, we have to declare a pointer of that type.

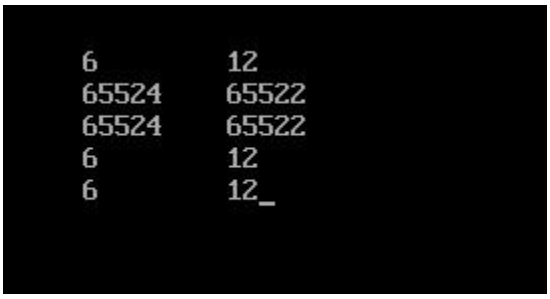
To better understand the concept of pointers I would recommend you to run the below program at home and try to figure out the result by your own.

```
#include<stdio.h>

void main()
{
 int a=6,b=12;
 int *x,*y;
```

```
 x=&a;
 y=&b;
 printf("%d \t %d \n",a,b);
 printf("%u \t %u \n",&a,&b);
 printf("%u \t %u \n",x,y);
 printf("%d \t %d \n",*x,*y);
 printf("%d \t %d",*(&a),*(&b));
}
```

Predict the output of above program first and run it to check the results of it. Comment below if you find any difficulty in understanding this program.

Output



As I said earlier concept of pointer is compulsory before learning the topic call by reference. I hope till now you must have a good grip on basic concepts of pointer. So in the next tutorial I will give you a tutorial on call by value and call by reference feature of functions.

Call by Value and Call by Reference

To understand this tutorial you must have knowledge of pointers in C. So read previous article about pointers: Pointers in C Programming

Now armed with the knowledge of pointers lets move our quest to learn C programming one step forward. Today I will tell you about the second advance feature of functions i.e. call by value and call by reference in C. Knowingly or unknowingly we have used the call by value feature in many programs till now. So it will be easy to understand the logic behind it. On the other side call by reference is used with pointers.

Call by Value

In previous tutorials I told you about the actual and formal arguments (parameters). In this method, whenever we make changes in formal arguments then it doesn't change the value in actual arguments. As its name suggests, call by value feature means calling the function and passing some values in it. The values of actual arguments are copied into formal arguments. We have done it so many times till now. Lets take a fresh example to understand it quickly.

```
#include<stdio.h>

void fun(int a);

void main()
{
 int a=10;
 printf(" Before calling\n a=%d",a);
 fun(a); //here a is actual argument or parameter
 printf("\n\n After calling\n a=%d",a);
}

void fun(int a)
{
 a=a+5; //here a is formal argument or parameter
 printf("\n\n In function fun()\n a=%d",a);
}
```

Output

```
Before calling
a=10

In function fun()
a=15

After calling
a=10
```

Explanation

- The program code is almost self-explanatory. First I have printed the value of variable a before calling of function and then I have passed one argument in the function fun().
- The value of variable a of main() is copied into variable a of function fun().
- Now I have changed the value of variable a by adding 5 in it. After that I have printed the variable a in function fun().
- At last the value of variable a is again printed in main(). You can observe in output that the change that was done in variable a in function fun() is not affected the value of variable a in main().

Call by Reference

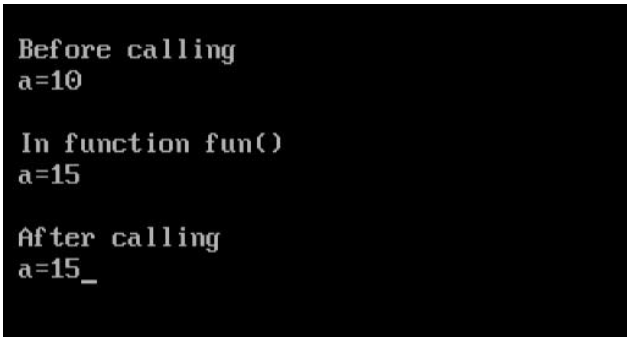
As I told earlier while applying call by value feature of functions we cannot change the values in actual arguments by changing the values in formal arguments. Many times we stick in a condition when we need to change the values of actual arguments in any other function. In those cases we use call by reference method.

In this method we will send the address of variables to the function. Any changes in formal arguments causes change in actual arguments. Lets build a simple program to understand this topic.

```
#include<stdio.h>

void fun(int *a);

void main()
{
 int a=10;
 printf("\n\n Before calling\n a=%d",a);
 fun(&a); //a is actual argument or parameter
 printf("\n\n After calling\n a=%d",a);
}
```

```
void fun(int *a)
{
 *a=(*a)+5; //a is formal argument or parameter
 printf("\n\n In function fun()\n a=%d",*a);
}
```

Output



Explanation

- In the beginning I have declared the function fun() with its argument as integer pointer.
- In the function call I have passed the address of variable a instead of passing its value.
- Now I have the address in pointer variable a of function fun().
- Then I have done some changes in the pointer variable and printed its value. Finally the variable a is again printed in main()
- You can see in the output that the change done in the pointer variable a of function fun() causes change in variable a of main().

You may face difficulty in understanding the concept of call by reference. You can ask your queries by commenting below.

Recursion

In the last tutorial I told you about the function calls i.e. call by value and call by reference. Today I will tell about the last advance feature of function i.e. recursion in C. So lets straight away starts it with a simple example.

```
#include<stdio.h>

void crashp();

int main()
{
        crashp();

        return 0;
}

void crashp()
{
        crashp();       //calling funtion itself
}
```

This is the simplest example of a recursive function. A function is said to be recursive function when it calls itself. In the above you can see the function crashp() is calling itself again and again.

What will be the output of the above program?
As you can see crashp() function is calling itself again and again. So this program will create an infinite loop and it will never end. In this case press Ctrl+PauseBreak button to stop the execution.

Note: It is quite difficult to show the working of recursive function. Therefore, I recommend you to read the basic concepts of functions first before proceeding further. Working of recursive function totally depends on the basic concepts.

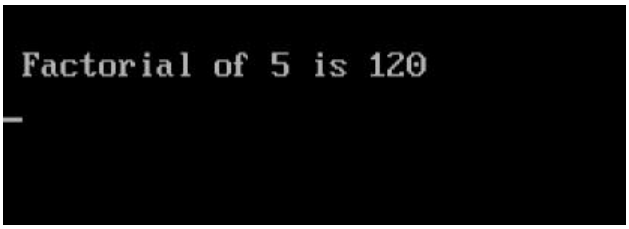
Lets take another example of a recursive function to calculate factorial of a number.

Program for factorial using recursion

```
#include<stdio.h>

int factorial(int x)
{
  if(x<=1)
  {
```

```
    return 1;
  }
  return (x * factorial(x-1));
}

int main()
{
   int x=5;
   printf("Factorial of %d is %d\n",x,factorial(x));
   return 0;
}
```

Output



Explanation

1. Program execution starts from the main() function. I have already assigned integer variable x with value 5.

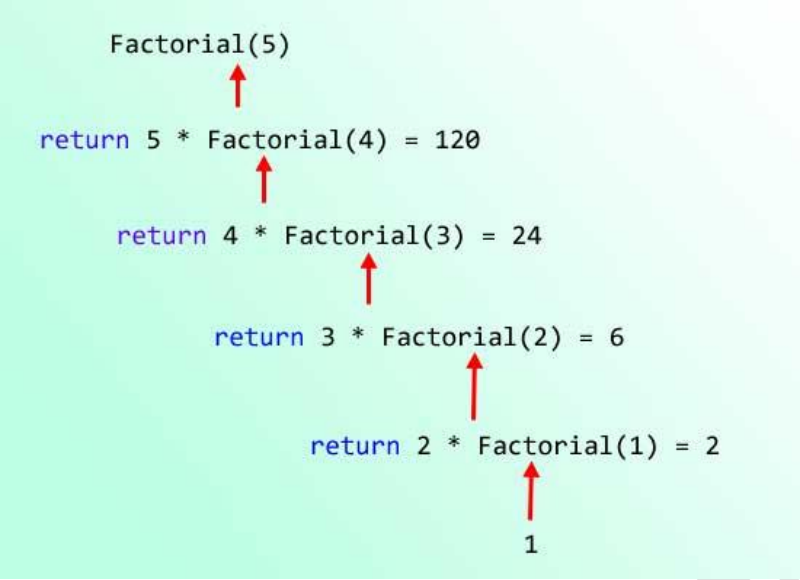
2. After that I have written the printf() function.

3. Inside printf() function I have called factorial() function to calculate the factorial value.

4. Now the function factorial() will call itself until value of x makes this condition true (x<=1)

5. Consider the return statement of factorial() function carefully i.e. return x * factorial(x-1);

It will solve the answer in this way 5*4*3*2*1=120



Its quite tricky to understand the execution of a recursive function. But go through the above tutorial at least 2-3 times for better understanding. So this brings us to the end of functions. Now in the next tutorial I will tell you about the extended data types.

Data Types

In some of our tutorials in the beginning, I told you about the three primary data types which are used in C language. Today I will introduce you to some variations in these primary data types in C. We can make unlimited data types as per our requirement in C. So C language is quite rich in data types.

The primary data types which we have used till now has some limits. For example, we can only store values in the range of –32768 to 32767 for an int data type. Remember this range is for 16 bit compiler Turbo C. For 32 bit compiler like VC++ this range is –2147483648 to +2147483647. You can't store more values which exceed this limit in int data type. To overcome those limits we have to use some extra keywords like unsigned, signed etc.

What are 16 bit and 32 bit compilers?
Turbo C is a 16 bit compiler. After converting the C code into machine language, it will target that code to run on 16 bit processors like intel 8086.

On the other hand VC++ is a 32 bit compiler. It will target to run the code on 32 bit processors like intel Pentium processors.

Compiler	short	int	long
16-bit (Turbo C/C++)	2	2	4
32-bit (Visual C++)	2	4	4

short and long int

As I said earlier C language provides variation in its primary data types. short and long are such variations of integer data type. As its name suggests long has bigger range than int.

Note: Here short and long are both keywords.

The range of both short and long are given below.

Range of long is -2147483648 to 2147483647.
Range of short is -32,768 to +32,767

signed and unsigned int

Generally it happened when we are confirmed that the value will not be negative in any case. So to utilize those conditions efficiently we often use unsigned int. With the use of unsigned keyword the range of int data type shifts from negative to positive.

New range will be 0 to 65535

By default normal int is also called signed int.

signed and unsigned char

Range of normal char is -128 to +127. You must have known that char data type always stores the ASCII value of character. Many times we emerge in a condition when we have print or access the ASCII character who value is more than +127. So in that case we generally use unsigned char. Similar to int it also makes the range almost twice. Both signed and unsigned char occupies 1 byte in memory.

New range will be 0 to 255

float and double

float variable occupies 4 bytes in memory and it also provides a good range to store values in it. It gives the range of -3.4e38 to +3.4e38.

If you want to increase this limit then you can also use double data type. It takes 8 bytes in the memory and it also provided a very big range to store values. Its range is -1.7e4932 to +1.7e4932.

Data Type	Range	Bytes	Format
signed char	-128 to + 127	1	%c
unsigned char	0 to 255	1	%c
short signed int	-32768 to +32767	2	%d
short unsigned int	0 to 65535	2	%u
signed int	-32768 to +32767	2	%d
unsigned int	0 to 65535	2	%u
long signed int	-2147483648 to +2147483647	4	%ld
long unsigned int	0 to 4294967295	4	%lu
float	-3.4e38 to +3.4e38	4	%f
double	-1.7e308 to +1.7e308	8	%lf
long double	-1.7e4932 to +1.7e4932	10	%Lf
Note: The sizes and ranges of int, short and long are compiler dependent. Sizes in this figure are for 16-bit compiler.			

Declaration of all data types

Given below are declarations of all the data types which we have learned so far.

```
int a;
unsigned int b;
long int c;
short int d;
unsigned short int e;
unsigned long int f;
char g;
unsigned char f;
float g;
double h;
```

Storage Classes

Till now we have declared the variables in its simplest form. Declaration of variables contains two parts.

- Data type of the variable
- Storage of the variable

Till now we are just declaring the data type of the variable.

Does it mean there is no storage class of variables which we have declared till now?
Well the answer is definitely No, because we cannot declare a variable without its storage class. By default certain storage classes automatically added, if we don't declare storage classes explicitly.

What are storage classes?

As I told you earlier, during the declaration of variables some space is reserved for that variable. Storage class is the property which decides that reserved space. There are two places where space is reserved for variables. One is memory and the other is register.

Storage class of the variable defines

- Place where the variable should store
- Default or initial value of the variable
- Scope of the variable
- Life of the variable

There are four storage classes in C language and I will define all of them based on the above factors.

Automatic Storage Class

Storage: Memory
Initial Value: Any garbage value
Scope: Local to the block in which the variable is defined
Life: Till the control remains in the block in which it is defined

Syntax: auto int x;

```
int main()
{
 auto int x;  //automatic variable
```

```
 int y;   //this is also automatic variable
 return 0;
}
```

If we do not specify the storage class of a variable then by default it is in automatic storage class. You can see this in above example.

Register Storage Class

Storage: CPU registers.
Initial value: Any garbage value
Scope: Local to the block in which the variable is defined
Life: Till the control remains in the block in which it is defined

Syntax: register int x;

A variable defined as register is stored in CPU registers. We define a variable as register when we need to access it frequently in our program. Since registers are limited so when no register is available then the variable is stored in memory.

Static Storage Class

Storage: Memory
Initial value: 0 (zero)
Scope: Local to the block in which the variable is defined
Life: Variable exists during the various function calls

Syntax: static int x;

```
#include<stdio.h>

void fun()
{
 static int x;  // x is static variable
 printf("%d ",x);
 x=x+2;
}

int main()
{
 fun();
 fun();
 fun();
 return 0;
}
```

Output

0 2 4

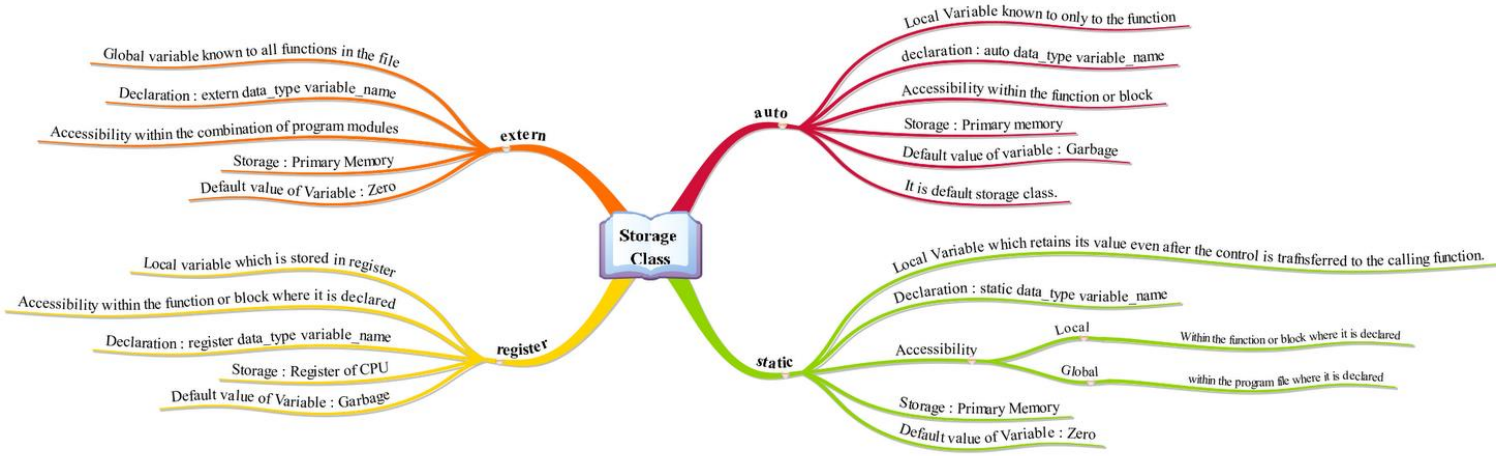
In the above program I have declared x variable as static, its initial value is 0. I have called the fun() function there times. I am increasing the value of x by 2. You can see in the output that the value of x variable is not lost. It remains in the memory during different function calls.

External Storage Class

Storage: Memory
Initial value: 0 (zero)
Scope: Global.
Life: During the whole program execution

Syntax: extern int x;

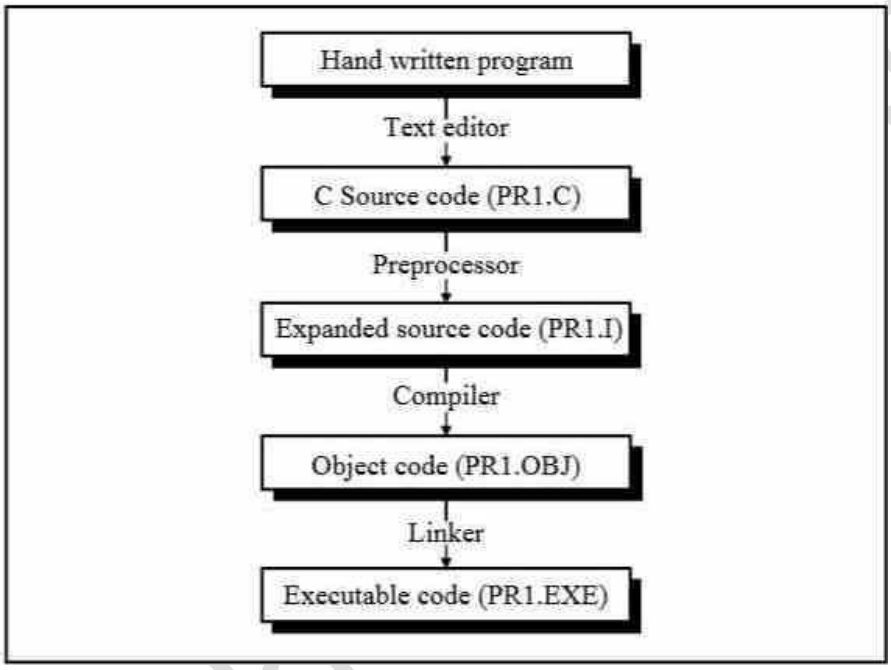
A variable with external storage class can be defined outside all functions. So that it will be available for every function throughout the program execution. It can also be defined inside certain function.



Note: static, register, auto and extern are the keywords which are used to specify some storage class. These storage classes can also be used to define functions.

Preprocessor Directives – Part 1

A program goes from several stages before getting executed. The program we write is converted into source code by using text editor which is stored with extension **.C**. After that it is converted to expanded source code by using preprocessor which is stored with extension **.I**. Now this expanded source code is converted to object code by using compiler which is stored with extension **.OBJ**. After that it is converted into executable code by using Linker and stored with extension **.EXE**.



Preprocessor Directives

Preprocessor directive is a text substitution tool that instruct the compiler to pre-processor our program before its actual compilation. Each C preprocessor starts with # symbol. Generally they are defined at the beginning of the program just after the header files. But you can define them anywhere in the program.

Types of Preprocessor Directives

There are four types of preprocessor directives which are given below.

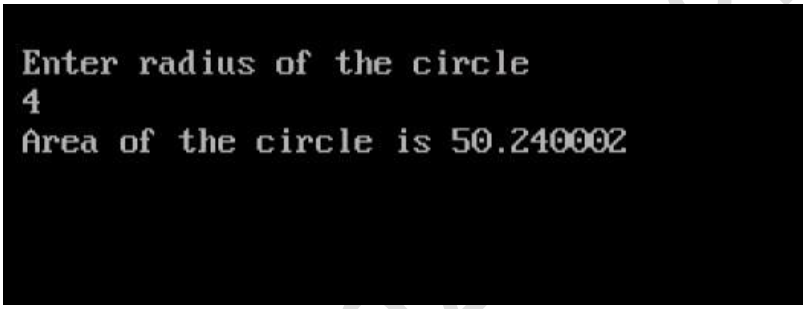
1. Macros
2. File inclusion
3. Conditional compilation
4. Miscellaneous directives

Macros

Macros are generally used to give a common name to the values which are used many times in a program. Or we can say that, macros are used to define symbolic constants. Consider below program to understand it.

```
//C program to calculate area of circle

#include <stdio.h>

#define PI 3.14  //macro

int main()
{
 float r, a;
 printf("Enter radius of the circle\n");
 scanf("%f",&r);
 a=PI*r*r;
 printf("Area of the circle is %f",a);

 return 0;
}
```

Output



Points to remember

- A macros always start with *#define*. In our program *#define PI 3.14* is the macros definition. Wherever PI is encountered in the program it is replaced with 3.14.
- In the compilation of the program all the macros definition replaced with the values which are used with #define. It is done in the process when source code is converted into expanded source code.
- Do not add semicolon at the end.
- Generally macros are written in single line. If you want to write it in multiple lines then you have to use macro continuation operator i.e. \. An example for this is given below.

Examples

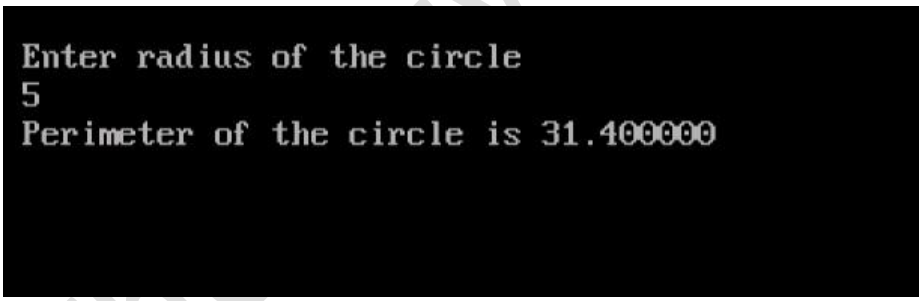
```
#define OR ||
#define CHECK (a>10 && a<20)
#define MSG \
printf("It is working fine")
```

Macros with arguments

Macros with arguments are similar to functions with arguments. Consider the below program to understand its working.

```
//C program to calculate perimeter of circle

#include<stdio.h>

#define PERI(x) (2*3.14*x)  //macro with argument

int main()
{
 float r,per;
 printf("Enter radius of the circle\n");
 scanf("%f",&r);
 per=PERI(r);
 printf("Perimeter of the circle is %f",per);

  return 0;
}
```

Output



Points to remember

- In the above program I have used macros with argument with the statement *#define PERI(x) (2*3.14*x)*
- Be careful while defining macros with arguments. In above example there must be no space between PERI and (x). The body of the macro should be placed in parentheses.
- Whenever the macro calling is done in our program, its calling statement in replaced by its body.

- They are fast as compared to functions because control do not goes to macro definition, the whole body comes at the place where calling is done.

File Inclusion

As its name suggests this preprocessor directive is used to include all the content of file to be included in the source code. Till now we have used preprocessor directives like *#include<stdio.h>* which includes all the contents from the header file stdio.h. These are predefined header files, you can also make your own and then include it in your program.

Examples

#include<math.h>
#include<stdlib.h>

Why it is used?
Its certainly a good question. It is not a good programming practice to write full program into a single file. While writing a very complex program it is recommend to break the program into files and include them in the beginning. Sometimes the macros of the program is also quite large which can only be stored in some files. So it is basically used for better management of the C program.

Preprocessor Directives – Part 2

In the last tutorial I told you about some preprocessor directives which are macros and file inclusion. Two more preprocessor directives are left which I will complete in this tutorial.

So today I will tell you about the two preprocessor directives in C language which are

- Conditional compilation
- Miscellaneous directives

Conditional Compilation

The usage of this preprocessor directive is similar to its name. It is often used to exclude some set of statements through the process of compilation. One question which will hit on your mind.

Why should I write those statements which I don't want to compile?

- Well programmers often write programs for some clients. Suppose at one stage client demands the older version of program which you have deleted. Then at those cases it is used very widely. So it is often used to omit the working of certain functions easily without touching the code. It is safest way to remove certain code from the program.
- It is also used to test the working of program. While writing complex C programs it is quite often that the C program gives wrong results at last moment. So we can use conditional compilation to rectify errors by using hit and trial method.
- Portability is one of the main feature which is quite famous these days. With the use of conditional compilation we can also make the program portable.

Now lets take one example to explain the conditional compilation.

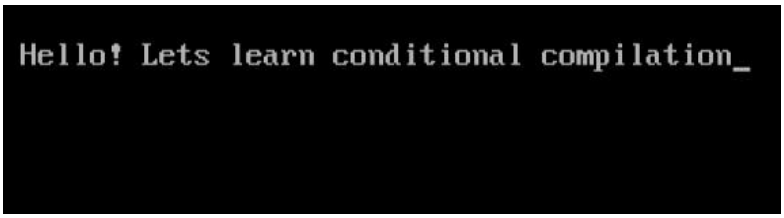
```
#ifdef MACRONAME
   Statement 1;
   Statement 2;
   And so on…
#endif
```

It means the statements 1, 2 and so on will only work when a macro is defined in the program with MACRONAME. You can also use #ifndef, it is just opposite of #ifdef. Here #else can be used to show else part of #ifdef or #ifndef. #endif shows the end of conditional compilation preprocessor directives.

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

```
int main()
{
 printf(" Hello! Lets learn conditional compilation");

 #ifdef WORK
  printf("This will not work");
 #endif

 return 0;
}
```

Output



Explanation

- As you can see, the second printf() does not work. Its because we have not defined any macro with WORK.
- To make the second printf() work, I just have to add #define WORK before main() function. After that it will also print the message inside second printf().

Miscellaneous Directives

The two preprocessor directives that are not commonly used falls inside the category of miscellaneous directives.

1. #undef
2. #pragma

#undef

This preprocessor directive is generally used to undefine certain macros at some stage. It is not commonly used but we can undefined any macros by using #undef followed by the macros name.

#pragma

This preprocessor directive is generally used in three ways which are

a. #pragma startup
b. #pragma exit
c. #pragma warn

#pragma startup: This is used to call some function right from the start up (before executing main() function). We can call any function by using this preprocessor directive.

Syntax: *#pragma startup funcall*

In the above code "funcall" function will be called from the start up of the program.

#pragma exit: It is a counter part of start up. It is used to call some function when program ends.

Syntax: *#pragma exit funcall*

In the above code "funcall" function is called at the exit of the program.

#pragma warn: This preprocessor directive is used very rarely. It is generally used to supress a specific warning inside a program.

Arrays – Part 1

In the last two tutorials I told you about the preprocessor directives. Today I will start giving you some advance lessons in C programming. And these tutorials starts from Arrays. So in this tutorial I will tell you about the arrays in C and its use. Lets start it with one basic question.

What are Arrays?

An array is a collection of similar type of data. Remember here similar means it belongs to one data type. Array is generally used to store many values which belong to one data type.

Why Array is used?

Till now we have some very basic programs. Suppose when we have to make a program to store the marks details of one student. Then we can easily make it by using simple variables. But when we asked to store details of 100 students then it will be very tedious task. We have to take a lot of variables to store those values. And that will be not an efficient way.

In those cases array is used quite easily. It provides an opportunity to store all the values in single array with single name. So we can easily call and receive data with single name.

Declaration of Array

```
int x[10];
```

In our above code int is the data type of an array. "x" is the variable name (array name in our case). "[10]" is something new. Well it defines the size of array. "10" means that this array will store 10 integer values in it. Square brackets shows that it is an array. We can create array of any other data type also. Some more examples are given below.

Examples

```
char a[5];
float b[15];
```

Initialization of Array

We have declared an array. Now its time to initialize it or storing some values in it. We can do it at the time of its declaration.

```
int x[5]={1,5,3,5,6};
```

or

```
int x[]={5,6,3,5,5};
```

In the first statement I have initialized the array with five values. And in the second statement I have initialized it without defining the size of an array. Remember when we declare and initialize the array at the same time then mentioning the size of array is optional.

Accessing Array Elements

Subscript is used to access the specific position element inside an array. Subscript is also known as index.

Remember that a subscript or index always starts with 0 end at n-1, here n is the maximum size of the array. If there is an array of size 10 then its subscript will be from 0 to 9.

For example if we want to access fourth element of the array x that we have declared above then it can be done in following way.

```
int a=x[3];
```

See here for accessing fourth element we are using index 3 because the subscript or index starts from 0. In the same way for accessing fifth element we will use index 4 i.e. x[4].

Printing Data of Array

Now it's a perfect time to build our first program which will print all the values of an array.

```
#include<stdio.h>

void main()
{
 int x[5]={33,66,43,63,67};
 int i;
 for(i=0;i<5;i++)
 {
  printf("%d\n",x[i]);
 }

}
```

Output

Explanation

The program is almost self explanatory. But note down the few points.

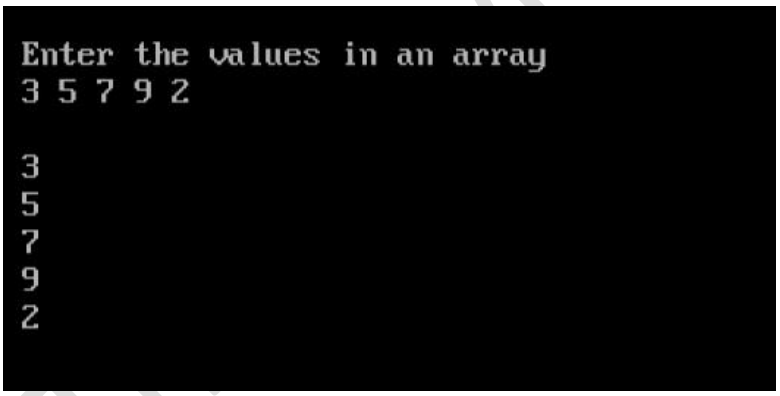
- I have declared an integer array x at the beginning and I have initialized it with some values.
- Now I have used a for loop to print the values inside it.
- Remember, subscript of array always starts with 0. So I have initialized variable i with 0.
- Now I have used x[i] in printf() function to print each value on the screen.

Arrays – Part 2

In the last tutorial I gave you an overview of arrays in C and I also told you about the declaration and initialization of arrays. Till now we have also completed the topic to print the values of array on the screen. Today I will tell you about how to take values inside an array by user. So lets start it with some program.

```
#include<stdio.h>

void main()
{
 int x[5];
 int i;
 printf(" Enter the values in an array\n ");

 for(i=0;i<5;i++)
 {
  scanf("%d",&x[i]); //taking values inside an array
 }

 for(i=0;i<5;i++)
 {
  printf("\n %d ",x[i]); //printing values on the screen
 }
}
```

Output



Explanation

- In our program I have declared "x" as an integer array with 5 elements.
- By using a for loop I have taken 5 values from the user.
- Consider carefully the subscript I have used inside scanf() function which is x[i].
- Almost similar loop is used to print the values inside that array.

To easily understand the working of arrays you should know about the fact.

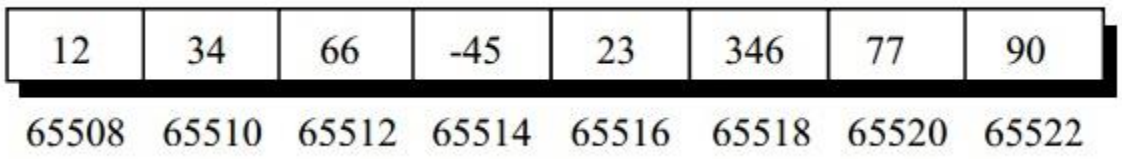
How array elements are stored in memory?

Consider the below declaration of an array.

int x[8];

When compiler detects some array declaration then it immediately reserves the space for it. In our case I have declared the integer array with 8 elements. When compiler detects it, it will immediately reserve 16 bytes in memory (2 byte for each element). In our case we do not initialize the array with its declaration so it will contain some garbage value by default. The reason behind the garbage values are due to its storage class. By default array has "auto" storage class.

Array consists of contiguous memory locations. It means all the values inside an array will be stored in contiguous memory parts. Checkout the below figure.



Bounds Checking

Consider the below code

```
int x[8];
x[10]=98;
```

The size of our array is only 8 elements. But we are storing the value 98 at 10th location. How is it possible? Its certainly not possible. But remember for this situation the compiler will not obstruct you by giving some error. It will not give any warning to you. Because there is no bounds checking is done in arrays. Compiler never checks the limit of the array while storing the values.

So where will this value be stored?
It can be stored anywhere. Probably just after contiguous location of the array. Or top of the array. It can stored anywhere. And remember sometimes small mistakes like that can completely ruin your program and can even hang computer. So I would strongly suggest you to remember this mistake while storing values inside an array.

Arrays – Part 3

So far I told you about some basic topics of arrays in C like initialization of arrays, accepting elements in array, printing the values from an array and so on. Today I will tell you about one advance use of arrays like passing the values of arrays to a function. It can be done in two ways, call by value and call by reference.

Passing Array Elements to a Function

Call by Value

Lets straight away starts with one program.

```
#include<stdio.h>

void printarr(int);

int main()
{
 int x;
 int nums[]={43,54,64,56,65};
 for(x=0;x<5;x++)
  printarr(nums[x]);

 return 0;
}

void printarr(int n)
{
 printf ("\n %d",n) ;
}
```

Output

Explanation

- I have declared an integer array nums and inserted 10 elements into it.

- After that I have started one for loop. Inside the loop I have written one statement i.e. printarr(nums[x]). It will call the function printarr().
- At first the value of x is 0. Then the first element of nums array will be passed to the printarr() function.
- That value will be received in the formal argument n of the function. After that it will display the element on the screen with printf() function.
- Now the control again moves to the main() function. The printarr() function is again called and second element of array nums is passed to it. This process will continue until the loop stops executing. And at last the program will stop.

Call by Reference

Lets understand it with one program.

```
#include<stdio.h>

void printarr(int*);

int main()
{
 int x;
 int nums[]={43,54,64,56,65};

  for(x=0;x<5;x++)
  printarr(&nums[x]);

  return 0;
}

void printarr(int *n)
{
 printf("\n %d",*n) ;
}
```

Explanation

- The code of the program is almost similar to the previous one. But in this case we are passing the address of array elements to the printarr() function.
- Note that I have passed the address of array elements with the help of address of operator i.e. &.
- The address of the array element is received by an integer pointer variable n.
- And in last I have printed the value by using printf() function. The output will be same as previous program.

Which one is better?

As you can see both the approaches are giving the same results. However I need to do little modifications in each program. Remember that the second program is better than the first one. Because it is using the pointer. Usage of pointer decreases the execution time of program. This is the reason people use to prefer pointers while using arrays in C.

Arrays – Part 4

In the last tutorial I told you about the memory allocation for arrays. If you are still not comfortable with that topic then I would suggest you to go through that tutorial at least once to understand this tutorial. Today I will tell you about the usage of pointers with arrays. As I said earlier pointer works faster in accessing elements of arrays then by accessing using its subscript. For convenience programmers generally adopt the subscript method but for making an efficient program it is always recommended to make use of pointers.

Note: This tutorial is all about programs, so I will give some good examples to explain the topic. Now I am assuming you are comfortable with the basic C language statements, so I will not explain about them explicitly.

Access Array Elements using Pointer

```
#include<stdio.h>

int main( )
{
 int x,*y;
 int nums[]={4,53,44,785,124};

 for(x=0;x<5;x++)
 {
  printf("\naddress=%u",&nums[x]);
  printf(" element=%d",nums[x]);
 }

 printf("\n");
 y=&nums[0];

 for(x=0;x<5;x++)
 {
  printf("\naddress=%u",y);
  printf(" element=%d",*y);
  y++;  /* Pointer is incrementing */
 }

 return 0;
}
```

Output

```
address=65516  element=4
address=65518  element=53
address=65520  element=44
address=65522  element=785
address=65524  element=124

address=65516  element=4
address=65518  element=53
address=65520  element=44
address=65522  element=785
address=65524  element=124
```

Explanation

- First of all I have declared an integer array with name nums. The size of the array is five elements. I have also declared an integer variable x and an integer pointer y.
- After that I have used two for loops. In the first one I have printed the elements and address of each element of the array.
- Now after the first for loop I have assigned the address of first array element into an integer pointer y.
- In the second loop I am displaying the address of array elements and its values by using that integer pointer y.
- To access all the values inside an array I am incrementing the integer point continuously within loop.

Points to Remember

- Now with the above program I have again verified that array elements are always stored in contiguous memory locations.
- With the second for loop, consider carefully that an integer pointer y always points to the next element after incrementing it to 1 (this is one of the core concept of pointers in C).
- To access all the values we have used y integer pointer with printf() function.

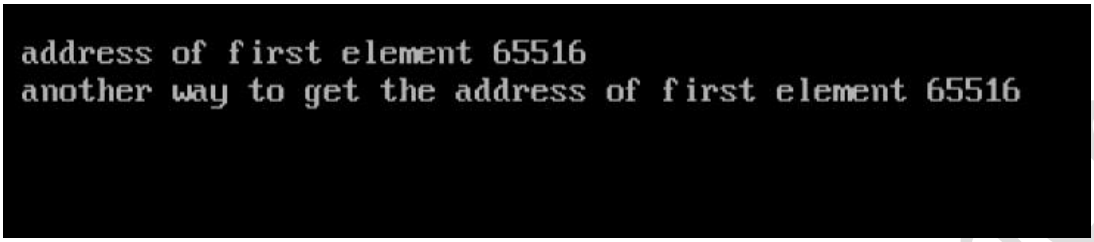
What we have learnt?
With the program we have learnt that we can access the whole array by using pointers if we know the base address or address of first element of array.

Another way to get the base address of an array

```
#include<stdio.h>

int main()
```

```
{
 int x;
 int nums[]={4,53,44,785,124};
 printf(" address of first element %u",&nums[0]);
 printf("\n another way to get the address of first element %u",nums);

 return 0;
}
```

Output



Explanation
With the above program we conclude that we can also access the base address of an array by just typing its name.

Arrays – Part 5

Till now I told you about the 1D arrays but today I will discuss about the 2D arrays or two dimensional arrays in c. Array is a collection of elements with similar data type. We can make arrays with any dimension. However programmers rarely go beyond 3D arrays. I will also give you an overview on 3D arrays in the subsequent tutorials. But today lets discuss the 2D arrays briefly.

2D Arrays

As its name suggests, 2D arrays are the arrays having 2 dimensions. Those two dimensions are generally called rows and columns. 2D arrays are also called matrix.

Declaration of 2D Array

A two dimensional array can be declared in following way.

```
int a[3][3];
```

This is a 2D array with 3 rows and 3 columns. The total elements in the array are 9 (3x3).

Initialization of 2D Array

Similar to 1D arrays, a 2D array can also be initialize at the time of its declaration. Given below are some of the initialization methods that are used very frequently.

```
int num[3][2] = {
  {43,56},
  {56,54},
  {65,98}
  };
```

This is one of the simplest way of initializing a 2D array.

```
int num[3][2] = {43, 56, 56, 54, 65, 98};
```

This method will also work but it will decrease the readability of the array too.

```
int  arr[ ][3] = { 12, 34, 23, 45, 56, 45 } ;
int num[][2] = {
  {43,56},
  {56,54},
  {65,98}
```

```
 {87,86}
 };
```

It is optional to provide the row dimension of a 2D array if we initialize it. Remember giving column dimension is always compulsory.

Lets take one simple program to understand 2D array in C.

```
#include<stdio.h>

int main()
{
 int student[6][2];
 int x;

 for(x=0;x<6;x++)
 {
  printf(" Enter roll no. and marks\n ");
  scanf("%d %d",&student[x][0],&student[x][1]);
 }

 for(x=0;x<6;x++)
  printf("\n %d %d",student[x][0],student[x][1]);

 return 0;

}
```

Output

```
Enter roll no. and marks
1 25
Enter roll no. and marks
2 24
Enter roll no. and marks
3 27
Enter roll no. and marks
4 30
Enter roll no. and marks
5 27
Enter roll no. and marks
6 22

1 25
2 24
3 27
4 30
5 27
6 22
```

Explanation
1. In the first statement I have declared the 2D array with name student. Remember 2D array also stores elements with index 00. Elements will be stored in this way.

00 01
10 11
20 21
And so on.

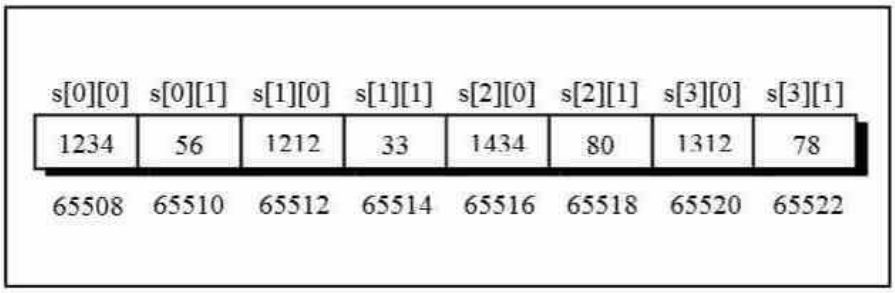
2. Now by using two for loops I have stored the values inside 2D array and display it on the screen.

3. Consider carefully the printf() and scanf() function with arguments "&stud[x][0], &student[x][1]". We are storing and accessing values inside 2D array by row wise. We can also store values and access them column wise too.

Memory Allocation of 2D Array

2D arrays also stores its values similar to 1D arrays. They also store elements in contiguous memory locations.

Elements of first row will be stored first, after that elements of second row will be stored. This procedure will continue until the array elements ends. Given below is the memory allocation of a 2D integer array s[4][2].



Arrays – Part 6

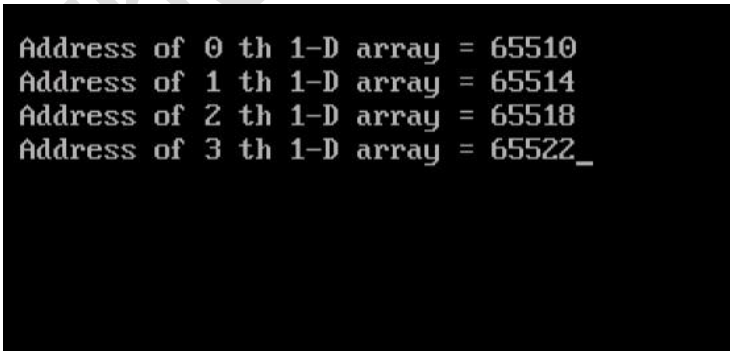
As I told you earlier the concept of arrays is very closely related to pointers. Arrays will work faster by using pointers rather than using a subscript. So it is always recommended to access arrays by using pointers. In the last tutorial I gave a brief overview on 2D arrays in C. Today I will discuss the usage of 2D arrays with pointers. This is one of the most complex topics of C programming, so I would like to suggest you to read this tutorial with full concentration.

2D Arrays with Pointers

A 2D array is nothing but a combination of 1D arrays. To prove my point I would like to show you an example.

```
#include<stdio.h>

int main( )
{
 int s[4][2]={
   {542,43},
   {154,354},
   {432,54},
   {435,435}
   };
 int x;

 for(x=0;x<=3;x++)
  printf("\n Address of %d th 1-D array = %u",x,s[x]);

 return 0;

}
```

Output



Explanation

- In the beginning I have declared a 2D array with 4 rows and 2 columns and also initialized it. So we can say that this 2D array is a collection of four 1D arrays having 2 elements each.
- After that I have declared integer variable x which will act as a loop counter.
- Now I have started the for loop with one printf() function. Consider carefully the arguments in printf() function. I have given two arguments which are x and s[x].
- As you can see I am only accessing the 2D array with one dimension. So it will give the addresses of only 1D arrays. s[0] will show the address of first element of first 1D array, s[1] will show address of first element of second 1D array and so on.

Access 2D Array Using Pointer Notation

The best way to learn it is through a program.

```
#include<stdio.h>

int main()
{
 int s[4][2]={
   { 542, 43 },
   { 154, 354 },
   { 432, 54 },
   { 435, 435 }
   } ;
 int x;

  printf(" %d\n",s[2][1]);
 printf(" It will give you address of 1st element of 3rd 1D array - %u \n",s[2]);
 printf(" It will give you address of 2st element of 3rd 1D array - %u \n",s[2]+1);
 printf(" Value at that address %d \n", *(s[2]+1));
 printf(" Alternate way of accessing that address %d \n",*(*(s+2)+1));

  return 0;
}
```

Output

```
54
It will give you address of 1st element of 3rd 1D array - 65518
It will give you address of 2st element of 3rd 1D array - 65520
Value at that address 54
Alternate way of accessing that address 54
```

Arrays – Part 7

In this tutorial I will tell you about two topics i.e. array of pointers and 3D arrays. So without wasting any time lets head on to the first topic of this tutorial.

Array of Pointers

As I told you earlier "Array is a collection of elements of similar data types". We have already created array of integer, floats and so on. So one question that arises. Can we make array of pointers? Well the answer is yes, of course. To make an array we have to fulfil just one condition i.e it should contain elements of same data type. So we can store pointers of isolated variables inside an array. Lets take one example to understand this.

```
#include<stdio.h>

int main()
{
 int *a[4]; //declaration of array of pointers
 int x=23,y=54,z=65,p=45,q;

 a[0]=&x;
 a[1]=&y;
 a[2]=&z;
 a[3]=&p;

 for(q=0;q<4;q++)
  printf("%d ",*a[q]);

 return 0;

}
```

Output
23 54 65 45

Explanation

- In the beginning of the program I have declared a 1D array of pointers with size 4. It means it can store addresses of four isolated pointer variables.
- After that I have declared some integer variables and I have stored some values in it.
- Now I have stored the addresses of integer variables inside array of pointers a.
- In the last I have printed all the values at the addresses stored in that array by using for loop.

Note: We can even store the addresses of other arrays inside array of pointers.

3D Array

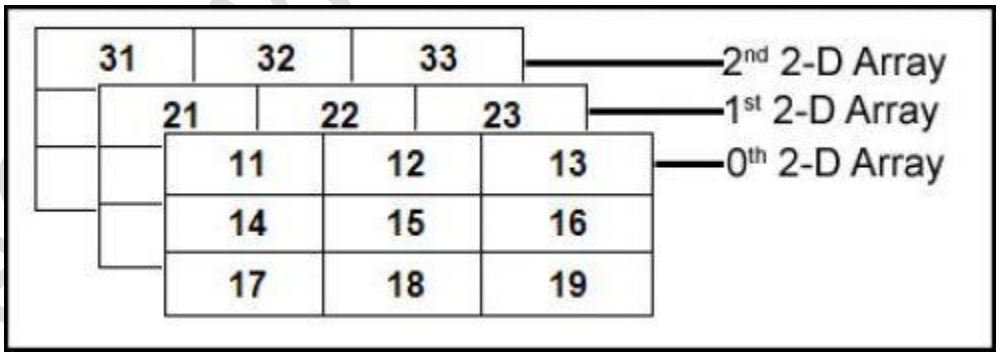
As the name suggests these are arrays having three dimensions. Generally a programmer rarely use 3D arrays. They are mostly used for some game programming. So I will only give you an overview about that.

Declaration and Initialization of 3D Array

Lets take one example to understand it.

```
int a[2][2][2]={
  {
   {13, 56},
   {54, 67}
  },
  {
   {64, 87},
   {23, 678}
  }
 };
```

In C language, 3D arrays treated as a collection of 2D arrays. In the above example we can also say that we are creating two 2D arrays. An example of 3D array is given below which is also a combination of three 2D arrays.



Memory Allocation of 3D Array

Allocation of memory of 3D array is similar to 2D array. As the elements inside 3D array always stored in contiguous memory locations.

String – Part 1

Till now I told you about the arrays in C language. I have also explained the close relation between arrays and pointers. Now lets move this journey to one step forward. Today I will tell you about the strings in C language. In this tutorial I will cover all the basics related to strings.

Earlier we used integer arrays to store group of integer elements. Similarly in C language we use strings to store group of characters. They are used to manipulate text such as words and sentences. Remember strings are similar to arrays but they are not same. Sometimes strings are also called character arrays.

A string always ends with a null character i.e. '\0'. Basically this character is used by the compiler to judge the length of the string. Remember '\0' and '0' are different characters. Both have different ASCII value and have different meaning.

Declaration of String

String can be declared in the same way we declare an array. We just need to use char data type followed by string name and its size. See below example.

```
char a[10];
```

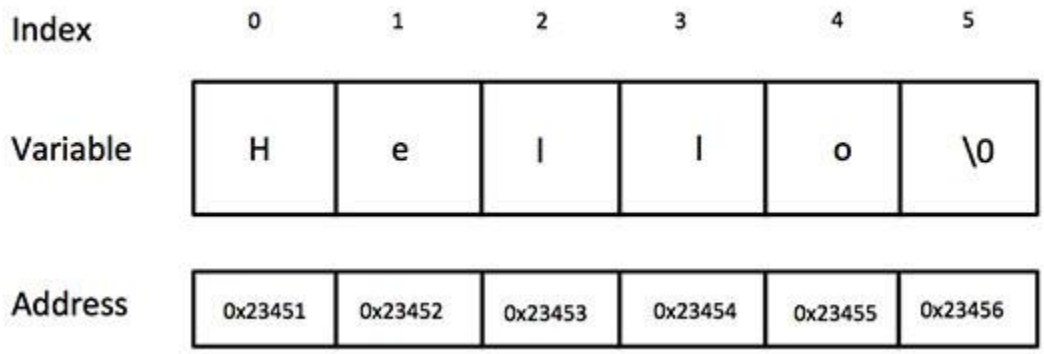
Initialization of String

```
char name[]={'H', 'e', 'l', 'l', 'o', '\0'};
```

or

```
char name[]="Hello";
```

Obviously the second method is far better than the first one. But consider carefully I haven't added '\0' at the end of "Hello". Well I don't need to worry about that in the second method because compiler will automatically add null character at the end of it.



Printing the String Elements

There are many ways to print the string elements on the screen. I will tell you the programs which are used very commonly.

```
#include<stdio.h>

int main()
{
 char name[]="TheCrazyProgrammer";
 int x=0;

 while(x<18)
 {
  printf("%c",name[x]);
  x++;
 }

 return 0;

}
```

Output

TheCrazyProgrammer

The above program is self-explanatory. I have initialized one string and I have printed the character elements by using while loop and subscript. Its similar to the program which I have used earlier to print the array elements. Now lets move on to a better version of this.

```
#include<stdio.h>

int main()
{
 char name[]="TheCrazyProgrammer";
 int x=0;

  while(name[x]!='\0')
 {
  printf("%c",name[x]);
  x++;
 }

  return 0;

}
```

Output of the program is same but the program is different. In this program I have omitted the boundation of writing the length of the string. It will print all the elements until it will get null character '\0'. So it's a better way of accessing string elements. Now lets make one program by using pointers.

```
#include <stdio.h>

int main( )
{
 char name[]="TheCrazyProgrammer";
 char *p;
 p=name;

 while(*p!='\0')
 {
  printf("%c",*p);
  p++;
 }

 return 0;
}
```

This program is similar to the last one. But in this program I have used pointer p for accessing the string elements instead of using subscript. The statement p=name; will store the address of first character of string in the pointer p. Now in the while loop each character is accessed by incrementing the address stored in pointer p.

Strings is one of the most important topics in C programming, as all the text manipulation can be done only through strings. So I would strongly suggest you to read the basics of strings at least twice. So that you can easily understand the advance topics.

String – Part 2

In the last tutorial I gave you an overview of the strings in C language. I told you about the basic programs to print the elements inside a string. In today's tutorial I will tell you about the best way to print the string on the screen. I will also tell you about the limitation of scanf() function with strings. So lets get started.

Text manipulation is one of the most frequently used function in any language. In the last program I have printed the string elements by accessing them one by one. However in serious programing it is very difficult to run loops when you have to print some string of characters on screen. To overcome this problem Dennis Ritchie introduced %s format specifier. It turns out to be very handy while printing the string on the go. Lets make one program to understand it.

```
#include<stdio.h>

void main()
{
 char name[]="TheCrazyProgrammer";
 printf("%s",name);
}
```

Output
TheCrazyProgrammer

Above program is self explanatory. I have only used %s format specifier to print the name string.

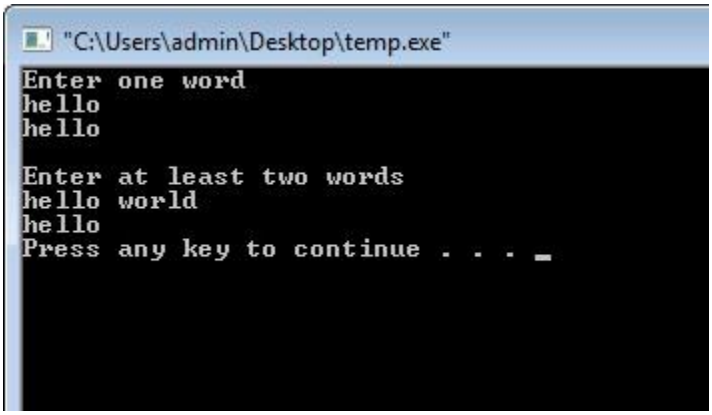
Reading String from User

So far I told you about printing the output on the screen. However taking input from the user to a string is equally important too. To perform this task, first answer that will hit your mind is scanf() function. Yes scanf() can be used to take the input. But it is not perfect for this task. Lets understand it with one program

```
#include<stdio.h>

void main( )
{
 char name[100];
 printf("Enter one word\n");
 scanf("%s",name);
 printf("%s\n",name);
 printf("\nEnter at least two words\n");
```

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

Output



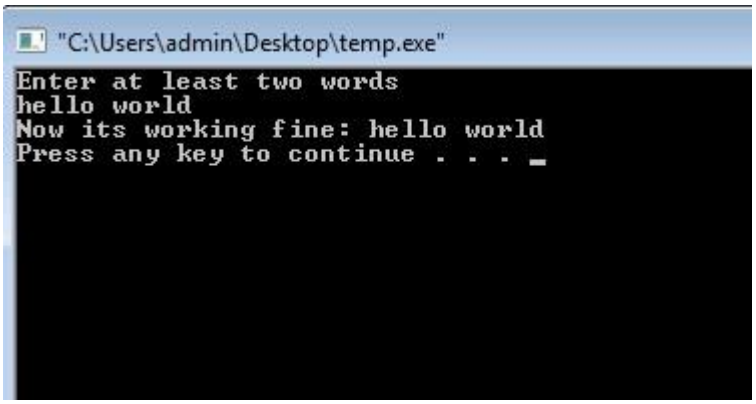
Explanation
As you can see I have used scanf() function two times. First time I have asked the user to enter only one word. So scanf() function is able to perform that task easily. But in the second time I have asked the user to enter at least two words. But in that case printf() has only printed "I". Well the reason behind it is that, scanf() function can only take single word. It closes the string when it founds any blank space. To overcome this problem we have to use another function.

gets() function

This function is used to take the input from the user to the string. It overcomes the problem of storing only single word. With the usage of this function user can store any number of words or even sentences. Lets understand it with one program.

```
#include<stdio.h>

void main( )
{
 char name[100];
 printf("Enter at least two words\n");
 gets(name);
 printf("Now its working fine: %s",name);
}
```

Output



In the above program I have replaced scanf() function with gets function. As you can see now everything is working fine.

puts() function

As its name suggest puts() function is a counter part of gets(). It is used to print one string at a time on the screen. Remember we can only use one string at a time to print it on the screen. On the other hand printf() function can be used to print any number of strings. Lets make one simple program to understand its working.

```
#include<stdio.h>

void main()
{
 char name[100]="Hello World!!!";
 puts(name);
}
```

Output
Hello World!!!

String – Part 3

In the last tutorial I told you about the subtle difference between gets() and scanf() function. Apart from this I also told you about the two ways to print strings on the screen. Armed with the basic concepts of strings lets move our journey forward to learn some advance topics of strings. Today I will tell you about the usage of strings with pointers and I will tell two-dimensional array of characters.

Strings and Pointers

Strings can be stored in two ways. The first one is quite obvious that we have used till now. In the second method we can store strings in character pointer. Given below is one small example.

```
char name[]="TheCrazyProgrammer";
char *p="TheCrazyProgrammer";
```

First one is quite familiar to you. But the second one is bit different. In the second one we are storing the address of first character of the string in character pointer. When compiler encounters this, it immediately store the entire string at some place and store the base address of it in character pointer.

Now one million dollar question which will hit your mind. Which one is better and why?

Well the answer depends on the condition or situation in which we have to use strings. But still programmers generally prefer the second method because it gives some flexibility to them.

Given below are two examples of such conditions.

```
void main()
{
        char name[]="Hello";
        char name1[20];
        char *a ="Good Morning";

        char *b;
        name1=name;//error
        b=a;    //works
}
```

Explanation
As you can see we cannot directly copy one string to another. It will result in an error. Why? This is because by accessing the name of the string we will only get the base address of the string. So when we directly copy one string to another. It will only try to copy the base address of one string to another string that will result in an error.

On the other hand by using character pointer we are only storing base address of string which can be easily copied to another character pointer.

```
void main()
{
	char name[]="Hello";
	char *name2="Hello";
	name="Bye";  //error
	name2="Bye";//works
}
```

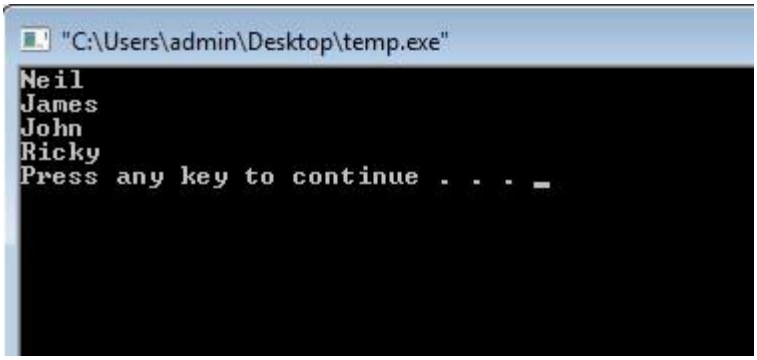
Explanation
When we store string in the normal way then we cannot initialize it multiple times. On the other hand we can again initialize the array when we store it in character pointer.

2D Array of Characters

In the earlier tutorials I already told you about the 2D arrays. 2D arrays of characters are also similar to them. So consider the below program to understand its working.

```
#include<stdio.h>

void main()
{
	char name[4][10]={
				"Neil",
				"James",
				"John",
				"Ricky"
			};
	int x;

	for(x=0;x<4;x++)
	{
		printf("%s\n",name[x]);
	}

}
```

Output



The program is self explainable. I have only used character 2D arrays and I have initialized it with some names. To print those names of the screen I have used one for loop. In the printf() function I have just incremented the subscript row wise to access all the names.

String – Part 4

In the last tutorial I gave a brief introduction of the relation between strings and pointers. I also told you about the 2D array of characters. Today I will tell you about some commonly used standard library string functions. These functions are made by the compiler vendors to speed up the manipulation of strings in C programming. So lets get started.

Standard Library String Functions

There are more than 18 standard library string function but only 4 of them are used very frequently. So I will only discuss those 4 functions which are strlen(), strcpy(), strcat(), strcmp().

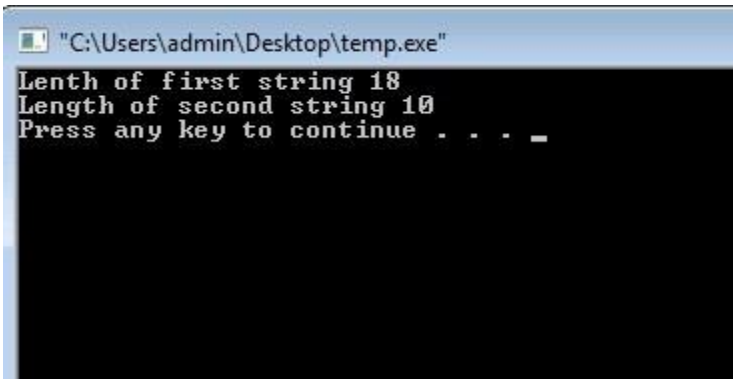
Note: The header file that we will use for these functions is string.h.

strlen()

As its name suggest, this function is made to calculate the length of the string. By using this function we can calculate the number of characters in any string.

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

int main()
{
	char name[]="TheCrazyProgrammer";
	int x,y;
	x=strlen(name);
	y=strlen("Hello dear");
	printf("Lenth of first string %d\nLength of second string %d",x,y);

	return 0;
}
```

Output



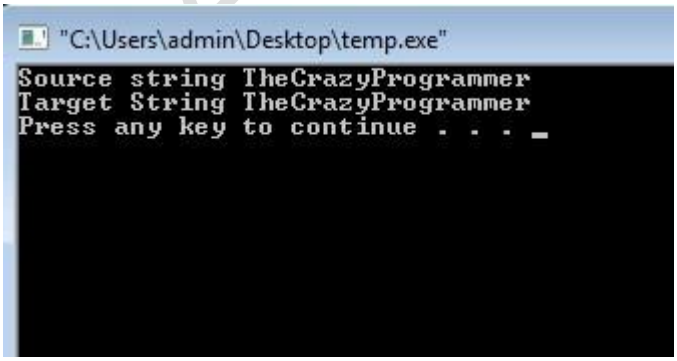
Explanation
Here you can see I have calculated the length of the string by two ways. So you can either pass string or string variable. Remember while calculating the length it doesn't count '\0' character.

strcpy()

This function is used to copy contents of one string to another. To use this function we just have to pass the base address of source and target string. Consider the below example.

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

int main()
{
	char source[]="TheCrazyProgrammer";
	char target[20];
	strcpy(target,source);
	printf("Source string %s\nTarget String %s\n",source,target);

	return 0;
}
```

Output

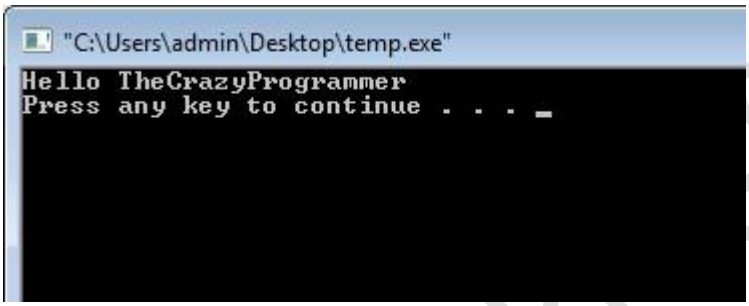


strcat()

This string function is used to concatenate (combine) two strings. To use this function we have to pass the base address of the two strings. Consider carefully the below program code.

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

int main()
{
	char str1[50]="Hello ";
	char str2[20]="TheCrazyProgrammer";
	strcat(str1,str2);
	printf("%s\n",str1);

	return 0;
}
```

Output



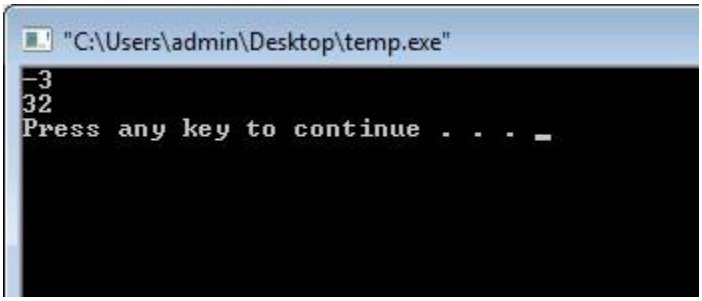
strcmp()

Do you remember I told you that we can't compare two strings directly just like common integer variables. This is because it will always give base address of the string to compare. Strcmp() function is used to compare two strings. This function returns 0 if the strings are identical and returns the difference of ASCII values when a mismatch occurs. Consider the below program carefully.

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

int main()
{
	char source[]="Hello ";
	char target[]="Khello";
	int x,y;
	x=strcmp(source,target);
	y=strcmp(source,"Hello");
```

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

    return 0;

}
```

Output



Explanation
In the first call I have compared source string with target string that returns -3 which is numeric difference of their mismatch ASCII characters. In the second call, I have passed two identical strings. Therefore 0 is returned by the function.

Structure – Part 1

So far I told you about the data types which can store only one type of data. E.g. int can only store integers, char can only store characters and so on. But in real life programming we hardly find some data which can be stored by using only one data type. Suppose we want to store the information of a student in a school. His information may contain his name which will be a string, his class which will be integer, his marks which will be float and so on.

Obviously we can also use arrays to store this information but it would be inefficient if I want to store information of 1000-2000 students. So basically we need a data type which can store values of dis-similar types. We need a data type which can be used to access those dis-similar type values easily. To solve this problem the concept of structures was introduced. So today I will give a basic overview of structure, its declaration and initialization.

Structure is the user defined data types that hold variables of different data types. Basically it gives a common name to the collection of different data types. We can also say that structure is a collection of dissimilar type of elements.

Defining a Structure

We have to define a structure first before using it. Given below is one example that will show how structure is defined.

```
struct student
{
 char name[10];
 int marks;
 int roll_no;
};
```

Here struct is keyword and student is the name of structure. name, marks and roll_no are the elements of this structure. You can clearly see that all these elements are of different data type. Remember declaration of structure does not reserve any space in memory.

Declaring Structure Variables

Look below code that will declare structure variables.

```
struct student s1,s2,s3;
```

Here s1, s2 and s3 are student type variables. We can also declare structure variables while defining the structure. This is shown in below example.

```
struct student
{
 char name[10];
 int marks;
 int roll_no;
}s1,s2,s3;
```

Structure Initialization

```
struct book
{
 char  name[10] ;
 float  price ;
 int  pages ;
};

struct book  a1={"Chemistry",190.00,210};
struct book  a2={"GK",250.80,559};
```

In the above code a1 and a2 are the structure variables. Consider carefully I have stored name, price and pages one by one.

Tips while using Structure

- Generally people forget to write semi colon at the end of closing brace of structure. It may give some big errors in your program. So write semi colon every time at the end of structure definition.
- Usually people declare structure variables at the top of the program. While making some big programs, programmers prefer to make separate header file for structure declaration. They include them at the beginning of the program.

Structure – Part 2

In the last tutorial I told you about the basic use of structure in C programming. I gave an overview of declaration and definition of structures. Armed with that basic knowledge, today I will tell you about how to access structure elements. Apart from this I will also tell you about the memory allocation of structure elements and array of structure. So lets get started.

Accessing Elements of Structure

It is important to define the structure before accessing it. In arrays we use subscript to access the array elements. But this concept is slightly different from them. To access the structure elements we use dot operator or member access operator (.). We have to write the name of structure variable followed by a dot and structure element. One small example is given below.

```
//structure definition
struct student
{
 char name[10];
 int roll;
 int marks;
}s1;

struct student s1={"Rishi",4,56}; //storing values

//accessing structure elements
printf("%s%d%d",s1.name,s1.roll,s1.marks);
```

Above code will display the name, roll number and marks of student s1. Consider carefully I have accessed the elements using dot operator.

Memory Allocation of Structure Elements

Structure elements are stored similar to array elements. It means the structure elements are also stored in contiguous memory locations. As I have said earlier, at the time of structure variable declaration, memory is allotted to it. In our above example 14 bytes (10 for string and 4 for two integers) will be allocated for s1.

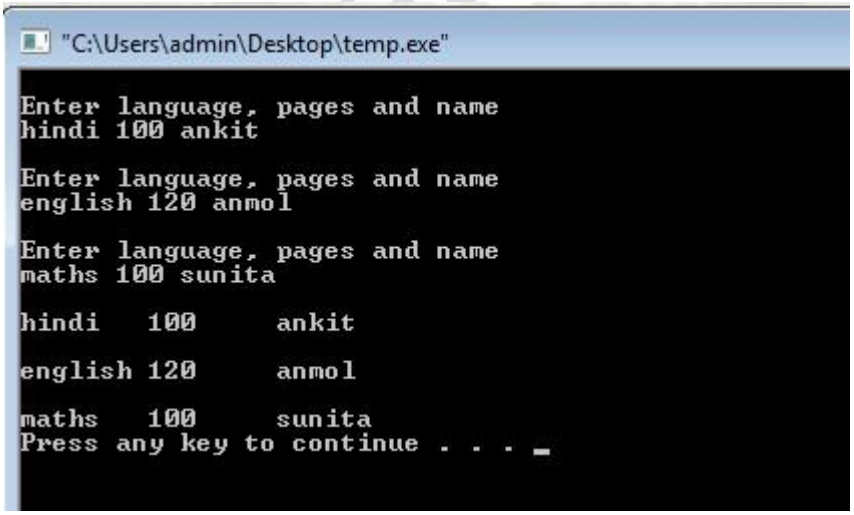
Array of Structure

Before proceeding to this topic I want to ask you – Is it possible to create array of structure? Well of course Yes. If we can make array of pointers, array of chars and so on then we can also make array of structure.

Making an array of structure is similar to creating a normal array. For this we only need to define a structure. After that we have to make an array of structure. Consider below program to understand it properly.

```
#include<stdio.h>

int main()
{
 struct library
 {
  char lang[10];
  int pages;
  char name[20];
 };

 struct library lib[3];
 int i;

 for(i=0;i<3;i++)
 {
  printf("\nEnter language, pages and name\n");
  scanf("%s%d%s",&lib[i].lang,&lib[i].pages,&lib[i].name);
 }

 for(i=0;i<3;i++)
  printf ( "\n%s\t%d\t%s\n",lib[i].lang,lib[i].pages,lib[i].name);

 return 0;
}
```

Output



Explanation

In this example I am storing the records of books in a library system. You can see I have created an array of structure of size 3 and that is storing 3 records from the user and then displaying the same on the screen.

Structure – Part 3

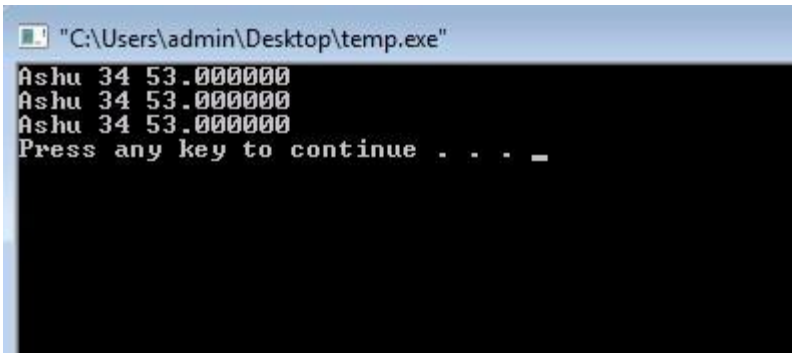
In the last two tutorials I gave a brief overview and functions of structures in C programming. Structure is one of the most important topic of C programming. The concept of structure is re-modified to make the object oriented language C++. Today I will tell you about the various advance features of structures in C programming. I will also tell you about the uses of structures at the end of this tutorial. So lets get started.

1. C language gave user the power to create your own variables. Structures are generally used to create user defined data types. To make the functionality of structures similar to other variables, the elements of structures are stored in contiguous memory locations. Due to this, one can easily assign all the values of one structure variable to other structure variable.

Remember we can also do piece meal copy by copying every element of structure one by one. Lets make one program to understand this.

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

void main()
{
 struct member
 {
  char name[10];
  int age;
  float mem_no;
 };

 struct member a1={"Ashu",34,53};
 struct member a2,a3;

 /* piece-meal copying */
 strcpy(a2.name,a1.name);
 a2.age=a1.age;
 a2.mem_no=a1.mem_no;

 /* copying all elements at one go */
 a3=a2;

 printf("%s %d %f",a1.name,a1.age,a1.mem_no);
 printf("\n%s %d %f",a2.name,a2.age,a2.mem_no);
 printf("\n%s %d %f\n",a3.name,a3.age,a3.mem_no);
}
```

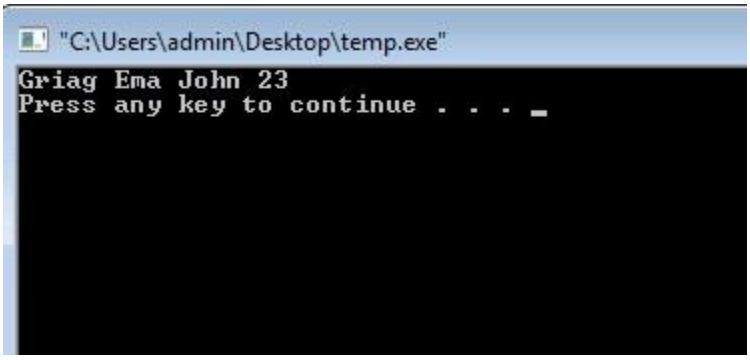
Output



2. Nesting is one main feature in C which gives the flexibility to create complex programs. Structures can also be nested to create some complex structures. Generally programmers prefer to stay within 3 layers of nesting. But remember practically there is no limit of nesting in C. Consider the below program to understand this.

```
#include<stdio.h>

void main()
{
 struct stud
 {
  char name[10];
  int age;
 };

 struct parent_name
 {
  char fath_name[10];
  char moth_name[10];
  struct stud a1;
 };

 struct parent_name s1={"Griag","Ema","John",23};

 printf("%s %s %s %d\n",s1.fath_name,s1.moth_name,s1.a1.name,s1.a1.age);
}
```

Output

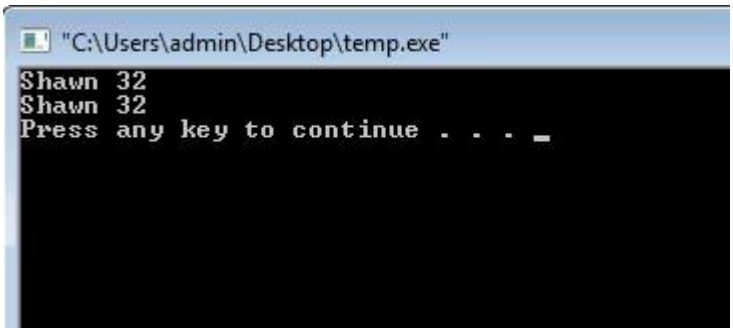


Explanation
Nesting of structure is quite simple. Consider carefully the two dot operators I have used inside printf function to access the elements of first structure.

3. So far I told you about the pointer to integer, pointer to character, pointer to array and so on. Similar to this we can also make pointer to a structure. It is used to store the address of a structure. Remember we cannot use (.) dot operator to access the structure elements using structure pointer. We have to use arrow operator (->) for that purpose. Consider the below code.

```
#include<stdio.h>

void main()
{
 struct stud
 {
  char name[10];
  int age;
 };

 struct stud s1={"Shawn",32};
 struct stud *s2;
 s2=&s1;

 printf("%s %d",s1.name,s1.age);
 printf("\n%s %d\n",s2->name,s2->age);
}
```

Output



Explanation
Consider carefully I have used arrow operator to access elements at that address

Uses of Structure

Structures are generally used to maintain databases in some organisation. One separate concept of Data Structures is also made to maintain the databases using structures. Apart from this some very common uses are given below.

a) Formatting a floppy
b) Receiving a key from the keyboard
c) Moving the cursor on the display
d) Making small games like Tic Tac Toe
e) Sending the output to printer

Console Input/Output Functions

Dennis Ritchie develop the C language without compromising to its compact feature. To attain compactness he deliberately not provided everything related to input output in the definition of the language. So C language doesn't contain any code to receive data from keyboard and send it on the screen. Then how are we using scanf() and printf() functions in C? Dennis Ritchie used the input/output functions of Operating System and linked them with C language. It means the printf() and scanf() function will work according to the OS you are using. Programmer does not have to bother about the working of those functions anyway.

Various input/output functions are available in C language. They are classified into two broad categories.

1. **Console Input/Output Functions** – These functions receive input from keyboard and write them on the VDU (Visual Display Unit).
2. **File Input/Output Functions** – These functions perform input/output operations on a floppy or hard disk.

Console Input/Output Functions

Keyboard and screen together called console. This is the behind the name of these functions. Console I/O functions further classified into

1. Formatted Input/Output Functions
2. Unformatted Input/Output Functions

Lets learn about them one by one.

Formatted Input/Output Functions

printf() and scanf() functions comes under this category. They provide the flexibility to receive the input in some fixed format and to give the output in desired format. As I already explained them in one previous article so I will not discuss them here.

sprintf() and sscanf() Function

These formatted console I/O functions works a bit different to printf() and scanf() functions. sprintf() function is quite similar to printf() function but instead of printing the output on screen, it stores it in the character array. Consider below example to understand this.

```
#include<stdio.h>

void main()
{
 int j=32;
 char cha='p';
 float a=123.2;
 char str[20];
 sprintf(str,"%d %c %f",j,cha,a);
 printf("%s\n",str);

}
```

Explanation
As I said earlier sprintf() doesn't print the output on screen. So I have printed the value of str using printf(). It just stores the data in string. In the above program, str will store the values of "j", "cha" and "a".

sscanf() is the counter part of sprintf() function. It allows the programmer to store the characters of string in some other variable. These two functions are used very rarely in C.

Unformatted Input/Output Functions

Functions like getch(), getche() and getchar() comes under this category. These functions store only one character. Till now we have used scanf() function to store values. Unfortunately we have to press enter key while using scanf() function to store the values in memory. In a condition when we have to store only one character these unformatted function comes very handy.

The header file used for these three functions is conio.h.

getch() function

This function is used to store only one character in memory. It does not echo or display that character on the screen while program execution.

getche() function

This function works similar to getch() function. However it just echo or display that character on screen.

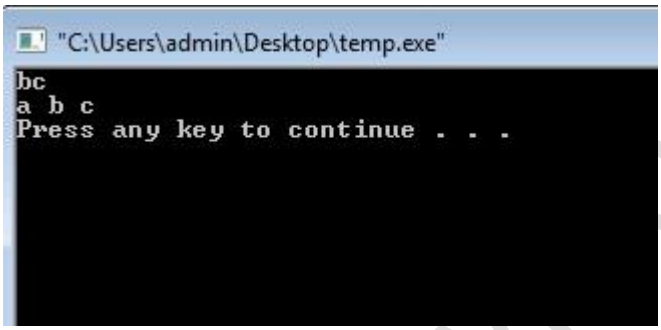
getchar() function

This function works entirely similar to getche() function. It stores one character and display it on the screen. But we have to press the enter key to store one character while using this function.

Consider below example to understand these functions.

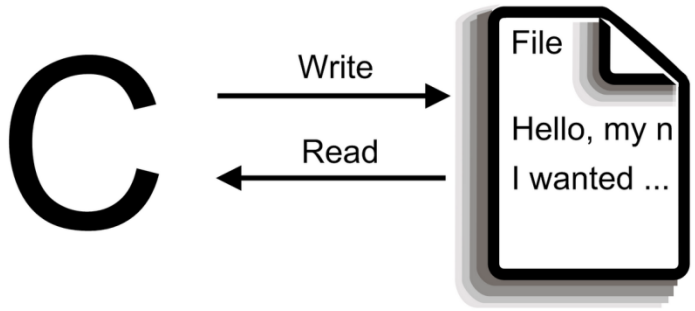
```
#include<stdio.h>
#include<conio.h>

void main()
{
 char ch1,ch2,ch3;
 ch1=getch(); // it does not echo the character on screen
 ch2=getche(); //echo's character on screen
 ch3=getchar();   // Use Enter to store the value

       printf("%c %c %c\n",ch1,ch2,ch3);

}
```

Output



File Handling - Part 1

In the last tutorial I told you about the basic console input and output functions. Today I will tell you about the second type of input output functions i.e. file input output functions. We will discuss about concept of file handling in C. So without making any delay lets get started.



Need of file input output functions

During serious C programming we often want to access huge amount of data. Such data cannot be displayed on the screen by once. We cannot store that data into memory due to its large size. Also it would be inappropriate to store such a large data in memory, due to its volatile nature. So either we have to store that data through keyboard again or regenerate it programmatically. Obviously both the methods are quite inefficient. In such cases we store data in a file. To manipulating those files we use file input output functions.

Data organization in a file

Data is always stored in binary form. However its technique in storing that binary data varies from one operating system to another. But as a programmer we don't have to bother about that. Compiler vendors do this task by writing the proper library functions according to targeted operating system.

Operations on File

Given below are the operations carried out on file through C programming.

1. Creating file
2. Opening an existing file
3. Reading from a file
4. Writing to a file
5. Moving to a specific location in file
6. Closing a file

Now its time to make a program that will display the content of a file on screen.

```
#include<stdio.h>

void main()
{
 FILE *fp;
 char ch;
 fp=fopen("demo.txt","r"); //opening file in read mode

 while(ch!=EOF)  //loop will continue till end of file
 {
  ch=fgetc(fp); //reading character from file one by one
  printf("%c",ch);
 }

 fclose(fp);  //closing the file
}
```

Output
Above program will display the contents of the file demo.txt on screen.

Explanation
1. In the beginning of the program I have declared a structure pointer fp. But wait, where is the structure FILE? Well it is already defined in the header file stdio.h. We need to use structure variable fp that will point to the current position in the file.

2. After that I have created a character variable ch that will store the character that we read from file.

3. Now I have written a statement fp=fopen("demo.txt","r"). fopen() is the function which is used to open the file. Inside that function I have passed two arguments. One with the name of the file and "r" to specify that file is opened in read mode. This function does three tasks which are as follows

- It search for the desired file on the disk.
- After that it loads the content of the file in memory which is called buffer.
- Then it points the structure variable fp to the first character of file.

Note: Here usage of buffer is quite important as it would be inefficient to load every character from the file every time. It will also take a lot of time. Buffer is also used while writing content in some file. Make sure that the file is already created on this disk.

4. After that I have used while loop and inside that I have written one statement ch=fgetc(fp) which is used to fetch one character from the file to the variable ch.

5. If the content of the file comes an end then value of ch will be EOF and the while loop will end.

6. At the end of the loop I have used printf() function to print that character on the screen.

7. In last I have closed the file using fclose() function.

File handling in C is very important concept and difficult too. Many people find difficulty in learning this concept. So I would recommend you to read this tutorial properly and practice the program. If you have any queries then fell free to ask it in comment section.

File Handling - Part 2

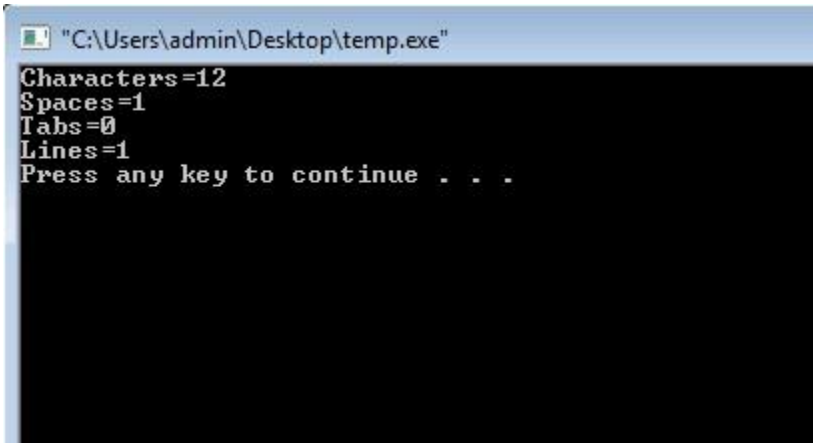
In the last tutorial I told you about the basic file input output functions. If you are reading this tutorial by skipping the last tutorial then I would strongly recommend you to read that tutorial first. So today I will tell you one program that will count all the characters, spaces, blanks and tabs inside one text file.

C Program to count characters, spaces, tabs and lines in a text file

Lets straight away write the program first.

```
#include<stdio.h>

void main()
{
 FILE *np;
 char ch;
 int lines=1,tabs=0,blanks=0,characters=0; //the last line is not counted so default value of lines is 1
 np=fopen("demo.txt","r");

 while(ch!=EOF)
 {
  ch=fgetc(np);
  characters++;

  if(ch==' ')
   blanks++;

  if(ch=='\n')
   lines++;

  if(ch=='\t')
   tabs++ ;
 }

 fclose(np);
 printf("Characters=%d",characters);
 printf("\nSpaces=%d",blanks);
 printf("\nTabs=%d",tabs);
 printf("\nLines=%d\n",lines);
}
```

Output



Note: I have opened the demo.txt in my computer. So the output will vary from file to file.

Explanation
1. First of all I have declared one FILE structure pointer np and initialized four integers which are lines, tabs, blanks and characters.

2. After that I have opened the file demo.txt using fopen() function. I have used the same function in my last tutorial too.

3. After that I have started one while loop. Inside it I am fetching the characters of that file using fgetc() function one by one. This while loop will continue till end of file. The end of file is encountered when value of ch become EOF.

5. Now I have used if blocks to count the number of characters, blanks, lines and tabs.

6. At the end of the loop I have closed the file using fclose() function.

7. The I have printed the values inside the integer variables which I have used to count the characters, spaces, blanks and tabs.

Note: This is one of the most frequently used practical application of file input output functions. By introducing some variables inside the program I have used it to count the characters inside the file.

Till now we have seen how to read from file and then display it on console. In our next tutorials we will learn about how to write something into a file.

File Handling - Part 3

In the last tutorial I told you about the program to calculate the number of spaces, blanks, tabs and characters. You might have noticed that I have used the similar logic of file reading program in that tutorial. Today I will continue the quest of learning file input output function by showing you a program to copy contents from one file to another.

C program to copy contents of one file to another file

Before writing the program I would like to introduce you to one new function fputc(). It's a counter part of fgetc() function. As its name suggests this function is used to write some content in the file. Consider the below program.

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

void main()
{
 FILE *fsource,*ftarget;
 char ch;

 fsource=fopen("demo.txt","r");
 if(fsource==NULL)
 {
  puts("Error while opening source file");
  exit(0);
 }

 ftarget=fopen("new.txt","w");

 while(ch!=EOF)
 {
  ch=fgetc(fsource);
  fputc(ch,ftarget);
 }

 fclose(fsource);
 fclose( ftarget);
}
```

Output
This program will copy contents of demo.txt file to new.txt.

Explanation

1. In this program I have created two file structure pointer named as fsource and ftarget. As its name suggests they are used for source and target files. I have also declared one character variable ch.

2. After that I have checked whether the source file is successfully opened for reading or not. I have used two if statements for that. This can be checked by putting the condition (file_pointer_name==NULL). The same checking can be done for second file which we are using for writing purpose.

3. Now I have started while loop and inside it fgetc() function is used to fetch one character from the source file and the writing that character to target file using fputc() function.

4. The while loop will continue till the file pointer fsource reaches to end of file demo.txt. The end is encountered when value of ch become EOF.

5. In the last I have closed both the files using fclose() functions.

Note: Generally we use buffer to write some content into some file because it will be inefficient to access disk repeatedly to write just one character. Functionality of buffer will be similar to the file reading program.

File Opening Modes

In the above example I have opened the source file with reading mode and target file with writing mode. However there are some more modes that are available in C. Lets study their use too.

"r" – It is used for reading from file. It searches for the file on the disk. In this mode fopen() returns NULL if it can't open the file otherwise it setup the pointer to point the first character of the file.

"w" – It is used for writing into file. It searches the file inside the disk, if it is absent then it will create one on the disk.

"a" – It does similar task like "w". Used for appending, that means it points the pointer at the end of the file for writing.

"r+" – Open file for both reading and writing. The file must already exist.

"w+" – Open file for both reading and writing. Creates a new file if file doesn't exist.

"a+" – Open file for reading and appending purpose.

Note: In all the above modes if fopen() function is unable to open the file due to some reason then it will return NULL.

File Handling - Part 4

In the last tutorial I told you about the file copy program using file input output functions. From the last two tutorials you must have noticed that the base of the program is same for all but we are making just simple twist to do our desired task. There are unlimited possibilities using file input output function. I have already discussed two in the last two tutorials. Today I will tell you about another useful program to do file input output using Strings. And this will be the last program of this topic.

String Input/Output in Files

Till now we are doing input output by using character. However in practical we often use strings instead of characters one by one. In this program I will use a new function whose name is fputs(). Oh Yes you are right! This function will be used to write strings of characters on files. So lets build one program for that.

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

void main()
{
 FILE *fp;
 char p[80];
 fp=fopen("demo.txt","w");

 if(fp==NULL)
 {
  puts("Cannot open file");
  exit(0);
 }

 printf("\nEnter the text to write on file\n");

 while(strlen(gets(p))>0)
 {
  fputs(p,fp);
  fputs("\n",fp);
 }

 fclose(fp);
}
```

Output
Above program will write strings of characters in the file Demo.txt

Explanation
1). In starting I have declared one FILE pointer fp and string p() whose length is 80 characters.

2. After that I have opened the file in "w" mode. This will check the file on the disk. If it is present then it will open it otherwise it will create a file with given name and extension.

3. If the file can't be accessed then it will display a message "cannot open a file".

4. After that I have placed one printf() function to inform the user to write down some strings.

5. Now I have created one infinite loop and I have used strlen() function to exit the program if the user press Enter key two times simultaneously.

6. Inside the while loop I have used fputs() function to write down the strings in file.

Let us make one program to see how to read string from file.

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

void main()
{
 FILE *fp;
 char p[80];
 fp=fopen("demo.txt","r");

 if(fp==NULL)
 {
  puts("Cannot open file");
  exit(0);
 }

 while(fgets(p,'\n',fp)!=NULL)
 {
  puts(p);
 }

 fclose(fp);
}
```

Output
It will display the content of file demo.txt.

Explanation

I have opened the file demo.txt in read mode. Now I am reading the strings from the file using fgets() function. As you can see I have used '\n', it means fgets() will read a string till new line is encountered. When fgets() can't find anymore string then it will return NULL and this will be the end of file.

You can find more C programs on file handling **here**.

Dynamic Memory Allocation

While doing programming we may come across a situation when the number of data items keeps on changing during the program execution. For example, we have made a program that stores the details for employees in a company. The list of employees keeps changing as new employees join or old employees leave the company. When this list increases we need more memory to store the information of new employees. Similarly when some employee leaves we need to free the memory so that it can be utilized to store some other information. Such condition can be managed effectively with the help of concept of dynamic memory allocation.

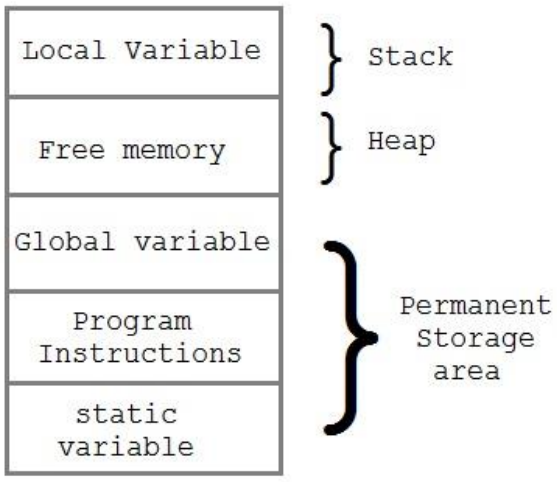
Dynamic Memory Allocation in C

The process of allocation of memory at the run time is known as dynamic memory allocation. In C it is done using four memory management functions that are given below. In this tutorial we will discuss them one by one in detail.

1. malloc()
2. calloc()
3. free()
4. realloc()

Note: The header file used for these four functions is stdlib.h.

In below image you can see that there is a free space between local variables and global variables region. This free space is known as heap and it is used for dynamic memory allocation.



Storage of C Program

malloc()

A block of memory can be allocated using malloc() function. Each allocated byte contains garbage value. Check its syntax below.

Syntax
pointer_name = (caste_type *) malloc(size_in_bytes);

Example
p = (int *) malloc (100 * sizeof (int));

Above statement will allocate a memory space of 100 times the size of int data type. It means if integer takes 2 bytes then 200 bytes will be allocated. Here sizeof() is an operator that will give the size of int data type. The malloc() function will return the address of first byte of memory allocated and this address will be stored in pointer p.

Lets take one more example.

p = (char *) malloc(10);

This will allocate 10 bytes of memory which can store only character type values.

malloc() allocates block of contiguous bytes. If it is unable to find the specified size space in the heap then it will return NULL. It is good to check the pointer variable to ensure that the memory allocation was successful.

calloc()

It is used for allocating multiple blocks of memory of same size. Each byte is automatically set to zero. This function is normally used to allocate memory for derived data types like arrays and structures.

Syntax
pointer_name = (cast_type *) calloc(n, size_in-bytes);

Example

```
. . . . .
struct student
{
 char name[20];
 int roll_no;
}*p;

p = (struct student*) calloc(5, sizeof(struct student) );
. . . . .
```

This will allocate 5 blocks of memory of same size that can store five records of structure student. If calloc() can't find the specified memory space then it will return NULL.

free()

When we allocate memory dynamically then it's our responsibility to free the memory when it is no longer in use, so that it can be used to store some other information. In C it can be done using free() function.

Syntax
free(pointer_name);

Example
free(p);

Here p is the pointer variable that contains the address of first byte of the memory block allocated by either malloc() or calloc() function.

realloc()

To change the size of the memory allocated dynamically we use realloc() function. We may face a condition that the previously allocated memory is not sufficient to store the data and more space is required. Or we have allocated a memory space larger than our need. In these cases we can easily alter the size of allocated memory using realloc().

Syntax
pointer_name = realloc(pointer_name, new_size);

Example
p = malloc(10);
p = realloc(p,20);

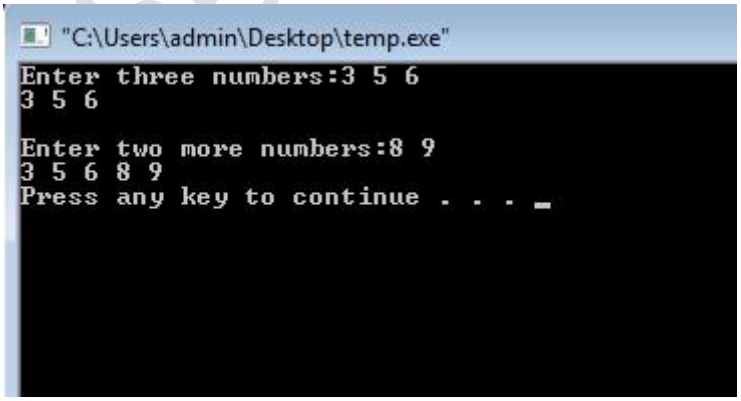
realloc() allocates new memory block and copy the data present in previous allocated block to new block. The previously allocated memory block is automatically freed. It also return NULL if can't find the specified memory space in heap.

Lets make one program to understand the use of all these memory management functions in C.

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

void main()
{
 int *p,i;
```

```
p=(int*)malloc(3*sizeof(int));

//checking the memory allocation was successful
if(p==NULL)
{
 printf("\nInsufficient memory");
 exit(0);
}

printf("Enter three numbers:");
for(i=0;i<3;++i)
 scanf("%d",p+i);

for(i=0;i<3;++i)
 printf("%d ",*(p+i));

//altering the memory
p=realloc(p,5*sizeof(int));

//checking the memory allocation was successful
if(p==NULL)
{
 printf("\nInsufficient memory");
 exit(0);
}

printf("\n\nEnter two more numbers:");
scanf("%d%d",p+3,p+4);

for(i=0;i<5;++i)
 printf("%d ",*(p+i));

//free the memory
free(p);
}
```

Output



Explanation
1. In the above program I am first allocating memory space to store three integer values using malloc() function. You can clearly see that I am checking the pointer p before using to ensure that the memory allocation was successful.

2. After that I have taken three integer values and then stored them in memory and displayed them.

3. Now I am altering the size of memory that I have previously allocated. I am changing the memory size using realloc() function so that I can store two more integer values.

4. After reading two more values I am displaying all the values again. You can see in the output, the three values that I stored earlier remain unchanged.

5. At last I have freed the memory using free() function.

So this is all about dynamic memory allocation in C. If you find anything missing or incorrect in above tutorial then please inform me. Feel free to ask if you have any doubts regarding above tutorial.