C Programming Tutorials

Basic Introduction

C is a general-purpose high level language that was originally developed by Dennis Ritchie for the Unix operating system. It was first implemented on the Digital Equipment Corporation PDP-11 computer in 1972.

The Unix operating system and virtually all Unix applications are written in the C language. C has now become a widely used professional language for various reasons.

- Easy to learn
- Structured language
- It produces efficient programs.
- It can handle low-level activities.
- It can be compiled on a variety of computers.

Facts about C

- C was invented to write an operating system called UNIX.
- C is a successor of B language which was introduced around 1970
- The language was formalized in 1988 by the American National Standard Institute (ANSI).
- By 1973 UNIX OS almost totally written in C.
- Today C is the most widely used System Programming Language.
- Most of the state of the art software have been implemented using C

Why to use C ?

C was initially used for system development work, in particular the programs that make-up the operating system. C was adopted as a system development language because it produces code that runs nearly as fast as code written in assembly language. Some examples of the use of C might be:

- Operating Systems
- Language Compilers
- Assemblers
- Text Editors
- Print Spoolers
- Network Drivers
- Modern Programs
- Data Bases
- Language Interpreters
- Utilities

C Program File

All the C programs are written into text files with extension ".c" for example hello.c. You can use "vi" editor to write your C program into a file.

This tutorial assumes that you know how to edit a text file and how to write programming instructions inside a program file.

C Compilers

When you write any program in C language then to run that program you need to compile that program using a C Compiler which converts your program into a language understandable by a computer. This is called machine language (ie. binary format). So before proceeding, make sure you have C Compiler available at your computer. It comes along with all flavors of Unix and Linux.

If you are working over Unix or Linux then you can type gcc -v or cc -v and check the result. You can ask your system administrator or you can take help from anyone to identify an available C Compiler at your computer.

If you don't have C compiler installed at your computer then you can use below given link to download a GNU C Compiler and use it.

Program Structure

A C program basically has the following form:

- Preprocessor Commands
- Functions
- Variables
- Statements & Expressions
- Comments

The following program is written in the C programming language. Open a text file hello.c using vi editor and put the following lines inside that file.

```
#include <stdio.h>

int main()
{
/* My first program */
printf("Hello, TechPreparation! \n");

return 0;
}
```

Preprocessor Commands:

These commands tells the compiler to do preprocessing before doing actual compilation. Like #include <stdio.h> is a preprocessor command which tells a C compiler to include stdio.h file before going to actual compilation. You will learn more about C Preprocessors in C Preprocessors session.

Functions:

Functions are main building blocks of any C Program. Every C Program will have one or more functions and there is one mandatory function which is called main() function. This function is prefixed with keyword int which means this function returns an integer value when it exits. This integer value is returned using return statement.

The C Programming language provides a set of built-in functions. In the above example printf() is a C built-in function which is used to print anything on the screen.

Variables:

Variables are used to hold numbers, strings and complex data for manipulation. You will learn in detail about variables in C Variable Types.

Statements & Expressions :

Expressions combine variables and constants to create new values. Statements are expressions, assignments, function calls, or control flow statements which make up C programs.

Comments:

Comments are used to give additional useful information inside a C Program. All the comments will be put inside /*...*/ as given in the example above. A comment can span through multiple lines.

Note the followings

- C is a case sensitive programming language. It means in C printf and Printf will have different meanings.
- C has a free-form line structure. End of each C statement must be marked with a semicolon.
- Multiple statements can be one the same line.
- White Spaces (ie tab space and space bar) are ignored.
- Statements can continue over multiple lines.

C Program Compilation

To compile a C program you would have to Compiler name and program files name. Assuming your compiler's name is cc and program file name is hello.c, give following command at Unix prompt.

```
$cc hello.c
```

This will produce a binary file called a.out and an object file hello.o in your current directory. Here a.out is your first program which you will run at Unix prompt like any other system program. If you don't like the name a.out then you can produce a binary file with your own name by using -o option while compiling C program. See an example below

```
$cc -o hello hello.c
```

Now you will get a binary with name hello. Execute this program at Unix prompt but before executing / running this program make sure that it has execute permission set. If you don't know what is execute permission then just follow these two steps

```
$chmod 755 hello
$./hello

This will produce following result
Hello, TechPreparation!
```

Congratulations!! you have written your first program in "C". Now believe me its not difficult to learn "C".

Basic Datatypes

C has a concept of 'data types' which are used to define a variable before its use. The definition of a variable will assign storage for the variable and define the type of data that will be held in the location.

The value of a variable can be changed any time.

C has the following basic built-in datatypes.

- int
- float
- double
- char

Please note that there is not a Boolean data type. C does not have the traditional view about logical comparison, but that's another story.

int - data type

int is used to define integer numbers.

```
{
int Count;
Count = 5;
}
```

float - data type

float is used to define floating point numbers.

```
{
float Miles;
Miles = 5.6;
}
```

double - data type

double is used to define BIG floating point numbers. It reserves twice the storage for the number. On PCs this is likely to be 8 bytes.

```
{
double Atoms;
Atoms = 2500000;
}
```

char - data type

char defines characters.

```
{
char Letter;
Letter = 'x';
}
```

Modifiers

The data types explained above have the following modifiers.

- short
- long
- signed
- unsigned

The modifiers define the amount of storage allocated to the variable. The amount of storage allocated is not cast in stone. ANSI has the following rules:

```
short int <= int <= long int
float <= double <= long double
```

What this means is that a 'short int' should assign less than or the same amount of storage as an 'int' and the 'int' should be less or the same bytes than a 'long int'. What this means in the real world is:

Type	Bytes	Range	
short int	2	-32,768 -> +32,767	(32kb)
unsigned short int	2	0 -> +65,535	(64Kb)
unsigned int	4	0 -> +4,294,967,295	(4Gb)
int	4	-2,147,483,648 -> +2,147,483,647	(2Gb)
long int	4	-2,147,483,648 -> +2,147,483,647	(2Gb)
signed char	1	-128 -> +127	
unsigned char	1	0 -> +255	
float	4		
double	8		
long double	12		

These figures only apply to today's generation of PCs. Mainframes and midrange machines could use different figures, but would still comply with the rule above.

You can find out how much storage is allocated to a data type by using the sizeof operator discussed in Operator Types Session.

Here is an example to check size of memory taken by various datatypes.

```
int
main()
{
printf("sizeof(char) == %d\n", sizeof(char));
printf("sizeof(short) == %d\n", sizeof(short));
printf("sizeof(int) == %d\n", sizeof(int));
printf("sizeof(long) == %d\n", sizeof(long));
printf("sizeof(float) == %d\n", sizeof(float));
printf("sizeof(double) == %d\n", sizeof(double));
printf("sizeof(long double) == %d\n", sizeof(long double));
printf("sizeof(long long) == %d\n", sizeof(long long));
```

```
return 0;
}
```

Qualifiers

A type qualifier is used to refine the declaration of a variable, a function, and parameters, by specifying whether:

- The value of a variable can be changed.
- The value of a variable must always be read from memory rather than from a register

Standard C language recognizes the following two qualifiers:

- const
- volatile

The const qualifier is used to tell C that the variable value can not change after initialization.
const float pi=3.14159;

Now pi cannot be changed at a later time within the program.

Another way to define constants is with the #define preprocessor which has the advantage that it does not use any storage

The volatile qualifier declares a data type that can have its value changed in ways outside the control or detection of the compiler (such as a variable updated by the system clock or by another program). This prevents the compiler from optimizing code referring to the object by storing the object's value in a register and re-reading it from there, rather than from memory, where it may have changed. You will use this qualifier once you will become expert in "C". So for now just proceed.

What are Arrays:

We have seen all basic data types. In C language it is possible to make arrays whose elements are basic types. Thus we can make an array of 10 integers with the declaration.

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

The square brackets mean subscripting; parentheses are used only for function references. Array indexes begin at zero, so the elements of x are:

Thus Array are special type of variables which can be used to store multiple values of same data type. Those values are stored and accessed using subscript or index.

Arrays occupy consecutive memory slots in the computer's memory.

```
x[0], x[1], x[2], ..., x[9]
```

If an array has n elements, the largest subscript is n-1.

Multiple-dimension arrays are provided. The declaration and use look like:

```
int name[10] [20];
n = name[i+j] [1] + name[k] [2];
```

Subscripts can be arbitrary integer expressions. Multi-dimension arrays are stored by row so the rightmost subscript varies fastest. In above example name has 10 rows and 20 columns.

Same way, arrays can be defined for any data type. Text is usually kept as an array of characters. By convention in C, the last character in a character array should be a `\0' because most programs that manipulate character arrays expect it. For example, printf uses the `\0' to detect the end of a character array when printing it out with a `%s'.

Here is a program which reads a line, stores it in a buffer, and prints its length (excluding the newline at the end).

```
 main( )
{
int n, c;
char line[100];
n = 0;
while( (c=getchar( )) != '\n' )
{
if( n < 100 )
line[n] = c;
n++;
}
printf("length = %d\n", n);
}
```

Array Initialization

- As with other declarations, array declarations can include an optional initialization
- Scalar variables are initialized with a single value
- Arrays are initialized with a list of values
- The list is enclosed in curly braces

```
int array [8] = {2, 4, 6, 8, 10, 12, 14, 16};
```

The number of initializes cannot be more than the number of elements in the array but it can be less in which case, the remaining elements are initialized to 0.if you like, the array size can be inferred from the number of initializes by leaving the square brackets empty so these are identical declarations:

```
int array1 [8] = {2, 4, 6, 8, 10, 12, 14, 16};
int array2 [] = {2, 4, 6, 8, 10, 12, 14, 16};
```

An array of characters ie string can be initialized as follows:

```
char string[10] = "Hello";
```

Variable Types

A variable is just a named area of storage that can hold a single value (numeric or character). The C language demands that you declare the name of each variable that you are going to use and its type, or class, before you actually try to do anything with it.

The Programming language C has two main variable types

- Local Variables
- Global Variables

Local Variables

- Local variables scope is confined within the block or function where it is defined. Local variables must always be defined at the top of a block.
- When a local variable is defined - it is not initialized by the system, you must initialize it yourself.
- When execution of the block starts the variable is available, and when the block ends the variable 'dies'.

Check following example's output

```
 main()
{
int i=4;
int j=10;

i++;

if (j > 0)
{
/* i defined in 'main' can be seen */
printf("i is %d\n",i);
}

if (j > 0)
{
/* 'i' is defined and so local to this block */
int i=100;
printf("i is %d\n",i);
}/* 'i' (value 100) dies here */

printf("i is %d\n",i); /* 'i' (value 5) is now visable.*/
}

This will generate following output
i is 5
i is 100
```

```
i is 5
```

Here ++ is called incremental operator and it increase the value of any integer variable by 1. Thus i++ is equivalent to i = i + 1;

You will see -- operator also which is called decremental operator and it decrease the value of any integer variable by 1. Thus i-- is equivalent to i = i - 1;

Global Variables

Global variable is defined at the top of the program file and it can be visible and modified by any function that may reference it.

Global variables are initialized automatically by the system when you define them!

Data Type	Initialser
int	0
char	'\0'
float	0
pointer	NULL

If same variable name is being used for global and local variable then local variable takes preference in its scope. But it is not a good practice to use global variables and local variables with the same name.

```
int i=4;                /* Global definition */

main()
{
i++;                   /* Global variable */
func();
printf( "Value of i = %d -- main function\n", i );
}

func()
{
int i=10;               /* Local definition */
i++;                    /* Local variable */
printf( "Value of i = %d -- func() function\n", i );
}

This will produce following result
Value of i = 11 -- func() function
Value of i = 5 -- main function
```

i in main function is global and will be incremented to 5. i in func is internal and will be incremented to 11. When control returns to main the internal variable will die and and any reference to i will be to the global.

Storage Classes

A storage class defines the scope (visibility) and life time of variables and/or functions within a C Program.

There are following storage classes which can be used in a C Program

- auto
- register
- static
- extern

auto - Storage Class

auto is the default storage class for all local variables.

```
{
int Count;
auto int Month;
}
```

The example above defines two variables with the same storage class. auto can only be used within functions, i.e. local variables.

register - Storage Class

register is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and cant have the unary '&' operator applied to it (as it does not have a memory location).

```
{
register int Miles;
}
```

Register should only be used for variables that require quick access - such as counters. It should also be noted that defining 'register' goes not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register - depending on hardware and implementation restrictions.

static - Storage Class

static is the default storage class for global variables. The two variables below (count and road) both have a static storage class.

```
static int Count;
int Road;

{
printf("%d\n", Road);
}
```

static variables can be 'seen' within all functions in this source file. At link time, the static variables defined here will not be seen by the object modules that are brought in.

static can also be defined within a function. If this is done the variable is initialized at run time but is not reinitialized when the function is called. This inside a function static variable retains its value during various calls.

```
 void func(void);

static count=10;    /* Global variable - static is the default */

main()
{
while (count--)
{
func();
}

}

void func( void )
{
static i = 5;
i++;
printf("i is %d and count is %d\n", i, count);
}
```

This will produce following result

```
i is 6 and count is 9
i is 7 and count is 8
i is 8 and count is 7
i is 9 and count is 6
i is 10 and count is 5
i is 11 and count is 4
i is 12 and count is 3
i is 13 and count is 2
i is 14 and count is 1
i is 15 and count is 0
```

static variables can be 'seen' within all functions in this source file. At link time, the static variables defined here will not be seen by the object modules that are brought in.

static can also be defined within a function. If this is done the variable is initialized at run time but is not reinitialized when the function is called. This inside a function static variable retains its value during various calls.

```
 void func(void);

static count=10;     /* Global variable - static is the default */

main()
{
while (count--)
{
func();
}

}
```

```
void func( void )
{
static i = 5;
i++;
printf("i is %d and count is %d\n", i, count);
}

This will produce following result

i is 6 and count is 9
i is 7 and count is 8
i is 8 and count is 7
i is 9 and count is 6
i is 10 and count is 5
i is 11 and count is 4
i is 12 and count is 3
i is 13 and count is 2
i is 14 and count is 1
i is 15 and count is 0
```

NOTE : Here keyword void means function does not return anything and it does not take any parameter. You can memories void as nothing. static variables are initialized to 0 automatically.

Definition vs. Declaration :

Before proceeding, let us understand the difference between definition and declaration of a variable or function. Definition means where a variable or function is defined in reality and actual memory is allocated for variable or function. Declaration means just giving a reference of a variable and function. Through declaration we assure to the complier that this variable or function has been defined somewhere else in the program and will be provided at the time of linking. In the above examples char *func(void) has been put at the top which is a declaration of this function where as this function has been defined below to main() function.

There is one more very important use for 'static'. Consider this bit of code.

```
char *func(void);

main()
{
char *Text1;
Text1 = func();
}

char *func(void)
{
char Text2[10]="martin";
return(Text2);
}
```

Now, 'func' returns a pointer to the memory location where 'text2' starts BUT text2 has a storage class of 'auto' and will disappear when we exit the function and could be overwritten but something else. The answer is to specify

```
static char Text[10]="martin";
```

The storage assigned to 'text2' will remain reserved for the duration if the program.

extern - Storage Class

extern is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern' the variable cannot be initialized as all it does is point the variable name at a storage location that has been previously defined.

When you have multiple files and you define a global variable or function which will be used in other files also, then extern will be used in another file to give reference of defined variable or function. Just for understanding extern is used to declare a global variable or function in another files.

File 1: main.c

```
int count=5;

main()
{
write_extern();
}
```

File 2: write.c

```
void write_extern(void);

extern int count;

void write_extern(void)
{
printf("count is %i\n", count);
}
```

Here extern keyword is being used to declare count in another file.

Now compile these two files as follows

```
gcc main.c write.c -o write
```

This fill produce write program which can be executed to produce result.

Count in 'main.c' will have a value of 5. If main.c changes the value of count - write.c will see the new value

Using Constants

A C constant is usually just the written version of a number. For example 1, 0, 5.73,

12.5e9. We can specify our constants in octal or hexadecimal, or force them to be treated as long integers.

- Octal constants are written with a leading zero - 015.
- Hexadecimal constants are written with a leading 0x - 0x1ae.
- Long constants are written with a trailing L - 890L.

Character constants are usually just the character enclosed in single quotes; 'a', 'b', 'c'. Some characters can't be represented in this way, so we use a 2 character sequence as follows.

'\n'	newline
'\t'	tab
'\\'	backslash
'\"	single quote
'\0'	null (Used automatically to terminate character string)

In addition, a required bit pattern can be specified using its octal equivalent.

'\044' produces bit pattern 00100100.

Character constants are rarely used, since string constants are more convenient. A string constant is surrounded by double quotes eg "Brian and Dennis". The string is actually stored as an array of characters. The null character '\0' is automatically placed at the end of such a string to act as a string terminator.

A character is a different type to a single character string. This is important point to note.

Defining Constants

ANSI C allows you to declare constants. When you declare a constant it is a bit like a variable declaration except the value cannot be changed.

The const keyword is to declare a constant, as shown below:

```
int const a = 1;
const int a =2;
```

Note:
You can declare the const before or after the type. Choose one an stick to it.
It is usual to initialize a const with a value as it cannot get a value any other way.

The preprocessor #define is another more flexible (see Preprocessor Chapters) method to define constants in a program.

```
#define TRUE        1
#define FALSE       0
#define NAME_SIZE   20
```

Here TRUE, FALSE and NAME_SIZE are constant

You frequently see const declaration in function parameters. This says simply that the function is not going to change the value of the parameter.

The following function definition used concepts we have not met (see chapters on functions, strings, pointers, and standard libraries) but for completeness of this section it is is included here:

```
void strcpy(char *buffer, char const *string)
```

The enum Data type

enum is the abbreviation for ENUMERATE, and we can use this keyword to declare and initialize a sequence of integer constants. Here's an example:

```
enum colors {RED, YELLOW, GREEN, BLUE};
```

I've made the constant names uppercase, but you can name them which ever way you want.

Here, colors is the name given to the set of constants - the name is optional. Now, if you don't assign a value to a constant, the default value for the first one in the list - RED in our case, has the value of 0. The rest of the undefined constants have a value 1 more than the one before, so in our case, YELLOW is 1, GREEN is 2 and BLUE is 3.

But you can assign values if you wanted to:

```
enum colors {RED=1, YELLOW, GREEN=6, BLUE };
```

Now RED=1, YELLOW=2, GREEN=6 and BLUE=7.

The main advantage of enum is that if you don't initialize your constants, each one would have a unique value. The first would be zero and the rest would then count upwards.

You can name your constants in a weird order if you really wanted...

```
#include <stdio.h>

int main() {
enum {RED=5, YELLOW, GREEN=4, BLUE};

printf("RED = %d\n", RED);
printf("YELLOW = %d\n", YELLOW);
printf("GREEN = %d\n", GREEN);
printf("BLUE = %d\n", BLUE);
return 0;
}

This will produce following results

RED = 5
YELLOW = 6
GREEN = 4
```

BLUE = 5

Operator Types

What is Operator? Simple answer can be given using expression 4 + 5 is equal to 9. Here 4 and 5 are called operands and + is called operator. C language supports following type of operators.

- Arithmetic Operators
- Logical (or Relational) Operators
- Bitwise Operators
- Assignment Operators
- Misc Operators

Lets have a look on all operators one by one.

Arithmetic Operators:

There are following arithmetic operators supported by C language:

Assume variable A holds 10 and variable holds 20 then:
Show Examples

Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiply both operands	A * B will give 200
/	Divide numerator by denominator	B / A will give 2
%	Modulus Operator and remainder of after an integer division	B % A will give 0
++	Increment operator, increases integer value by one	A++ will give 11
--	Decrement operator, decreases integer value by one	A-- will give 9

Logical (or Relational) Operators:

There are following logical operators supported by C language

Assume variable A holds 10 and variable holds 20 then:
Show Examples

Operator	Description	Example
==	Checks if the value of two operands is equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the value of two operands is equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the	(A <= B) is

	value of right operand, if yes then condition becomes true.	true.
&&	Called Logical AND operator. If both the operands are non zero then then condition becomes true.	(A && B) is true.
||	Called Logical OR Operator. If any of the two operands is non zero then then condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is false.

Bitwise Operators:

Bitwise operator works on bits and perform bit by bit operation.

Assume if B = 60; and B = 13; Now in binary format they will be as follows:

A = 0011 1100

B = 0000 1101

A&B = 0000 1000

A|B = 0011 1101

A^B = 0011 0001

~A = 1100 0011
Show Examples

There are following Bitwise operators supported by C language

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -60 which is 1100 0011
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 0000 1111

Assignment Operators:

There are following assignment operators supported by C language:
Show Examples

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator	C |= 2 is same as C = C | 2

Short Notes on L-VALUE and R-VALUE:

x = 1; takes the value on the right (e.g. 1) and puts it in the memory referenced by x. Here x and 1 are known as L-VALUES and R-VALUES respectively L-values can be on either side of the assignment operator where as R-values only appear on the right.

So x is an L-value because it can appear on the left as we've just seen, or on the right like this: y = x; However, constants like 1 are R-values because 1 could appear on the right, but 1 = x; is invalid.

Misc Operators

There are few other operators supported by C Language.
Show Examples

Operator	Description	Example
sizeof()	Returns the size of an variable.	sizeof(a), where a is integer, will return 4.
&	Returns the address of an variable.	&a; will give actual address of the variable.
*	Pointer to a variable.	*a; will pointer to a variable
? :	Conditional Expression	If Condition is true ? Then value X : Otherwise value Y

Operators Categories:

All the operators we have discussed above can be categorized into following categories:

- Postfix operators, which follow a single operand.
- Unary prefix operators, which precede a single operand.
- Binary operators, which take two operands and perform a variety of arithmetic and logical operations.
- The conditional operator (a ternary operator), which takes three operands and evaluates either the second or third expression, depending on the evaluation of the first expression.
- Assignment operators, which assign a value to a variable.
- The comma operator, which guarantees left-to-right evaluation of comma-separated expressions.

Precedence of C Operators:

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator:

For example x = 7 + 3 * 2; Here x is assigned 13, not 20 because operator * has higher precedence than + so it first get multiplied with 3*2 and then adds into 7.

Here operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type) * & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= |=	Right to left
Comma	,	Left to right

Flow Control Statements

C provides two styles of flow control:

- Branching
- Looping

Branching is deciding what actions to take and looping is deciding how many times to take a certain action.

Branching:
Branching is so called because the program chooses to follow one branch or another.

if statement
This is the most simple form of the branching statements.

It takes an expression in parenthesis and an statement or block of statements. if the expression is true then the statement or block of statements gets executed otherwise these statements are skipped.

NOTE: Expression will be assumed to be true if its evaluated values is non-zero.

if statements take the following form:
Show Example

```
if (expression)
statement;

or

if (expression)
{
Block of statements;
}

or

if (expression)
{
Block of statements;
}
else
{
Block of statements;
}

or

if (expression)
{
Block of statements;
}
else if(expression)
{
Block of statements;
}
else
{
Block of statements;
```

```
}
```

? : Operator

The ? : operator is just like an if ... else statement except that because it is an operator you can use it within expressions.

? : is a ternary operator in that it takes three values, this is the only ternary operator C has.

? : takes the following form:
Show Example

```
if condition is true ? then X return value : otherwise Y value;
```

switch statement:

The switch statement is much like a nested if .. else statement. Its mostly a matter of preference which you use, switch statement can be slightly more efficient and easier to read.
Show Example

```
switch( expression )
{
case constant-expression1: statements1;
[case constant-expression2: statements2;]
[case constant-expression3: statements3;]
[default : statements4;]
}
```

Using break keyword:

If a condition is met in switch case then execution continues on into the next case clause also if it is not explicitly specified that the execution should exit the switch statement. This is achieved by using break keyword.

What is default condition:

If none of the listed conditions is met then default condition executed.

Looping

Loops provide a way to repeat commands and control how many times they are repeated. C provides a number of looping way.

while loop

The most basic loop in C is the while loop. A while statement is like a repeating if statement. Like an If statement, if the test condition is true: the statements get executed. The difference is that after the statements have been executed, the test condition is checked again. If it is still true the statements get executed again. This cycle repeats until the test condition evaluates to false.

Basic syntax of while loop is as follows:
Show Example

```
while ( expression )
{
```

```
Single statement
or
Block of statements;
}
```

for loop

for loop is similar to while, it's just written differently. for statements are often used to process lists such a range of numbers:

Basic syntax of for loop is as follows:
Show Example

```
for( expression1; expression2; expression3)
{
Single statement
or
Block of statements;
}
```

In the above syntax:

- expression1 - Initializes variables.
- expression2 - Conditional expression, as long as this condition is true, loop will keep executing.
- expression3 - expression3 is the modifier which may be simple increment of a variable.

do...while loop

do ... while is just like a while loop except that the test condition is checked at the end of the loop rather than the start. This has the effect that the content of the loop are always executed at least once.

Basic syntax of do...while loop is as follows:
Show Example

```
do
{
Single statement
or
Block of statements;
}while(expression);
```

break and continue statements

C provides two commands to control how we loop:

- break -- exit form loop or switch.
- continue -- skip 1 iteration of loop.

You already have seen example of using break statement. Here is an example showing usage of continue statement.

```
#include

main()
{
int i;
int j = 10;

for( i = 0; i <= j; i ++ )
{
if( i == 5 )
{
continue;
}
printf("Hello %d\n", i );
}
}
```

This will produce following output:

```
Hello 0
Hello 1
Hello 2
Hello 3
Hello 4
Hello 6
Hello 7
Hello 8
Hello 9
Hello 10
```

Input and Output

Input : In any programming language input means to feed some data into program. This can be given in the form of file or from command line. C programming language provides a set of built-in functions to read given input and feed it to the program as per requirement.

Output : In any programming language output means to display some data on screen, printer or in any file. C programming language provides a set of built-in functions to output required data.

printf() function

This is one of the most frequently used functions in C for output. (we will discuss what is function in subsequent chapter.).

Try following program to understand printf() function.

```
#include <stdio.h>

main()
{
int dec = 5;
char str[] = "abc";
```

```
char ch = 's';
float pi = 3.14;

printf("%d %s %f %c\n", dec, str, pi, ch);
}
```

The output of the above would be:

```
5 abc 3.140000 c
```

Here %d is being used to print an integer, %s is being usedto print a string, %f is being used to print a float and %c is being used to print a character.

scanf() function

This is the function which can be used to to read an input from the command line.

Try following program to understand scanf() function.

```
#include <stdio.h>

main()
{
int x;
int args;

printf("Enter an integer: ");
if (( args = scanf("%d", &x)) == 0)
{
printf("Error: not an integer\n");
} else {
printf("Read in %d\n", x);
}
}
```

Here %d is being used to read an integer value and we are passing &x to store the vale read input. Here &indicates the address of variable x.

This program will prompt you to enter a value. Whatever value you will enter at command prompt that will be output at the screen using printf() function. If you enter a non-integer value then it will display an error message.

```
Enter an integer: 20
Read in 20
```

Pointing to Data

A pointer is a special kind of variable. Pointers are designed for storing memory address i.e. the address of another variable. Declaring a pointer is the same as declaring a normal variable except you stick an asterisk '*' in front of the variables identifier.

- There are two new operators you will need to know to work with pointers. The "address of" operator '&' and the "dereferencing" operator '*'. Both are prefix unary operators.
- When you place an ampersand in front of a variable you will get it's address, this can be stored in a pointer variable.
- When you place an asterisk in front of a pointer you will get the value at the memory address pointed to.

Here is an example to understand what I have stated above.

```
#include <stdio.h>

int main()
{
int my_variable = 6, other_variable = 10;
int *my_pointer;

printf("the address of my_variable is : %p\n", &my_variable);
printf("the address of other_variable is : %p\n", &other_variable);

my_pointer = &my_variable;

printf("\nafter \"my_pointer = &my_variable\":\n");
printf("\tthe value of my_pointer is %p\n", my_pointer);
printf("\tthe value at that address is %d\n", *my_pointer);

my_pointer = &other_variable;

printf("\nafter \"my_pointer = &other_variable\":\n");
printf("\tthe value of my_pointer is %p\n", my_pointer);
printf("\tthe value at that address is %d\n", *my_pointer);

return 0;
}
```

This will produce following result.

```
the address of my_variable is : 0xbfffdac4
the address of other_variable is : 0xbfffdac0

after "my_pointer = &my_variable":
the value of my_pointer is 0xbfffdac4
the value at that address is 6

after "my_pointer = &other_variable":
the value of my_pointer is 0xbfffdac0
the value at that address is 10
```

Pointers and Arrays

The most frequent use of pointers in C is for walking efficiently along arrays. In fact, in the implementation of an array, the array name represents the address of the zeroth element of the array, so you can't use it on the left side of an expression. For example:

```
char *y;
char x[100];
```

y is of type pointer to character (although it doesn't yet point anywhere). We can make y point to an element of x by either of

```
y = &x[0];
y = x;
```

Since x is the address of x[0] this is legal and consistent. Now `*y' gives x[0]. More importantly notice the following:

```
*(y+1) gives x[1]
*(y+i) gives x[i]

and the sequence

y = &x[0];
y++;

leaves y pointing at x[1].
```

Pointer Arithmetic:

C is one of the few languages that allows pointer arithmetic. In other words, you actually move the pointer reference by an arithmetic operation. For example:

```
int x = 5, *ip = &x;

ip++;
```

On a typical 32-bit machine, *ip would be pointing to 5 after initialization. But ip++; increments the pointer 32-bits or 4-bytes. So whatever was in the next 4-bytes, *ip would be pointing at it.

Pointer arithmetic is very useful when dealing with arrays, because arrays and pointers share a special relationship in C.

Using Pointer Arithmetic With Arrays:

Arrays occupy consecutive memory slots in the computer's memory. This is where pointer arithmetic comes in handy - if you create a pointer to the first element, incrementing it one step will make it point to the next element.

```
#include <stdio.h>

int main() {
int *ptr;
int arrayInts[10] = {1,2,3,4,5,6,7,8,9,10};

ptr = arrayInts; /* ptr = &arrayInts[0]; is also fine */

printf("The pointer is pointing to the first ");
printf("array element, which is %d.\n", *ptr);
```

```
printf("Let's increment it.....\n");

ptr++;

printf("Now it should point to the next element,");
printf(" which is %d.\n", *ptr);
printf("But suppose we point to the 3rd and 4th: %d %d.\n",
*(ptr+1),*(ptr+2));

ptr+=2;

printf("Now skip the next 4 to point to the 8th: %d.\n",
*(ptr+=4));

ptr--;

printf("Did I miss out my lucky number %d?!\n", *(ptr++));
printf("Back to the 8th it is then..... %d.\n", *ptr);

return 0;
}
```

This will produce following result:

```
The pointer is pointing to the first array element, which is 1.
Let's increment it.....
Now it should point to the next element, which is 2.
But suppose we point to the 3rd and 4th: 3 4.
Now skip the next 4 to point to the 8th: 8.
Did I miss out my lucky number 7?!
Back to the 8th it is then..... 8.
```

See more examples on Pointers and Array

Modifying Variables Using Pointers:

You know how to access the value pointed to using the dereference operator, but you can also modify the content of variables. To achieve this, put the dereferenced pointer on the left of the assignment operator, as shown in this example, which uses an array:

```
#include <stdio.h>

int main() {
char *ptr;
char arrayChars[8] = {'F','r','i','e','n','d','s','\0'};

ptr = arrayChars;

printf("The array reads %s.\n", arrayChars);
printf("Let's change it..... ");

*ptr = 'f'; /* ptr points to the first element */

printf(" now it reads %s.\n", arrayChars);
```

```
printf("The 3rd character of the array is %c.\n",
*(ptr+=2));
printf("Let's change it again..... ");

*(ptr - 1) = ' ';

printf("Now it reads %s.\n", arrayChars);
return 0;
}
```

This will produce following result:

```
The array reads Friends.
Let's change it..... now it reads friends.
The 3rd character of the array is i.
Let's change it again..... Now it reads f iends.
```

Generic Pointers: (void Pointer)

When a variable is declared as being a pointer to type void it is known as a generic pointer. Since you cannot have a variable of type void, the pointer will not point to any data and therefore cannot be dereferenced. It is still a pointer though, to use it you just have to cast it to another kind of pointer first. Hence the term Generic pointer. This is very useful when you want a pointer to point to data of different types at different times.

Try the following code to understand Generic Pointers.

```
#include <stdio.h>

int main()
{
int i;
char c;
void *the_data;

i = 6;
c = 'a';

the_data = &i;
printf("the_data points to the integer value %d\n",
*(int*) the_data);

the_data = &c;
printf("the_data now points to the character %c\n",
*(char*) the_data);

return 0;
}
```

NOTE-1 : Here in first print statement, the_data is prefixed by *(int*). This is called type casting in C language. Type is used to caste a variable from one data type to another datatype to make it compatible to the lvalue.

NOTE-2 : lvalue is something which is used to left side of a statement and in which we can assign some value. A constant can't be an lvalue because we can not assign any value in contact. For example x = y, here x is lvalue and y is rvalue.

However, above example will produce following result:

```
the_data points to the integer value 6
the_data now points to the character a
```

Using Functions

A function is a module or block of program code which deals with a particular task. Making functions is a way of isolating one block of code from other independent blocks of code.

Functions serve two purposes.

- They allow a programmer to say: `this piece of code does a specific job which stands by itself and should not be mixed up with anything else',
- Second they make a block of code reusable since a function can be reused in many different contexts without repeating parts of the program text.

A function can take a number of parameters, do required processing and then return a value. There may be a function which does not return any value.

You already have seen couple of built-in functions like printf(); Similar way you can define your own functions in C language.

Consider the following chunk of code

```
 int total = 10;
printf("Hello World");
total = total + 1;
```

To turn it into a function you simply wrap the code in a pair of curly brackets to convert it into a single compound statement and write the name that you want to give it in front of the brackets:

```
Demo()
{
int total = 10;
printf("Hello World");
total = total + 1;
}
```

curved brackets after the function's name are required. You can pass one or more parameters to a function as follows:

```
Demo( int par1, int par2)
{
int total = 10;
printf("Hello World");
```

```
total = total + 1;
}
```

By default function does not return anything. But you can make a function to return any value as follows:

```
int Demo( int par1, int par2)
{
int total = 10;
printf("Hello World");
total = total + 1;

return total;
}
```

A return keyword is used to return a value and datatype of the returned value is specified before the name of function. In this case function returns total which is int type. If a function does not return a value then void keyword can be used as return value.

Once you have defined your function you can use it within a program:

```
main()
{
Demo();
}
```

Functions and Variables:

Each function behaves the same way as C language standard function main(). So a function will have its own local variables defined. In the above example total variable is local to the function Demo.

A global variable can be accessed in any function in similar way it is accessed in main() function.

Declaration and Definition

When a function is defined at any place in the program then it is called function definition. At the time of definition of a function actual logic is implemented with-in the function.

A function declaration does not have any body and they just have their interfaces.

A function declaration is usually declared at the top of a C source file, or in a separate header file.

A function declaration is sometime called function prototype or function signature. For the above Demo() function which returns an integer, and takes two parameters a function declaration will be as follows:

```
int Demo( int par1, int par2);
```

Passing Parameters to a Function

There are two ways to pass parameters to a function:

- Pass by Value: mechanism is used when you don't want to change the value of passed parameters. When parameters are passed by value then functions in C create copies of the passed in variables and do required processing on these copied variables.
- Pass by Reference mechanism is used when you want a function to do the changes in passed parameters and reflect those changes back to the calling function. In this case only addresses of the variables are passed to a function so that function can work directly over the addresses.

Here are two programs to understand the difference: First example is for Pass by value:

```
#include <stdio.h>

/* function declaration goes here.*/
void swap( int p1, int p2 );

int main()
{
int a = 10;
int b = 20;

printf("Before: Value of a = %d and value of b = %d\n", a, b );
swap( a, b );
printf("After: Value of a = %d and value of b = %d\n", a, b );
}

void swap( int p1, int p2 )
{
int t;

t = p2;
p2 = p1;
p1 = t;
printf("Value of a (p1) = %d and value of b(p2) = %d\n", p1, p2 );
}
```

Here is the result produced by the above example. Here the values of a and b remain unchanged before calling swap function and after calling swap function.

```
Before: Value of a = 10 and value of b = 20
Value of a (p1) = 20 and value of b(p2) = 10
After: Value of a = 10 and value of b = 20
```

Following is the example which demonstrate the concept of pass by reference

```
#include <stdio.h>

/* function declaration goes here.*/
void swap( int *p1, int *p2 );
```

```
int main()
{
int a = 10;
int b = 20;

printf("Before: Value of a = %d and value of b = %d\n", a, b );
swap( &a, &b );
printf("After: Value of a = %d and value of b = %d\n", a, b );
}

void swap( int *p1, int *p2 )
{
int t;

t = *p2;
*p2 = *p1;
*p1 = t;
printf("Value of a (p1) = %d and value of b(p2) = %d\n", *p1, *p2 );
}
```

Here is the result produced by the above example. Here the values of a and b are changes after calling swap function.

```
Before: Value of a = 10 and value of b = 20
Value of a (p1) = 20 and value of b(p2) = 10
After: Value of a = 20 and value of b = 10
```

C Programming Tutorials

Strings

- In C language Strings are defined as an array of characters or a pointer to a portion of memory containing ASCII characters. A string in C is a sequence of zero or more characters followed by a NULL '\0' character:
- It is important to preserve the NULL terminating character as it is how C defines and manages variable length strings. All the C standard library functions require this for successful operation.
- All the string handling functions are prototyped in: string.h or stdio.h standard header file. So while using any string related function, don't forget to include either stdio.h or string.h. May be your compiler differs so please check before going ahead.
- If you were to have an array of characters WITHOUT the null character as the last element, you'd have an ordinary character array, rather than a string constant.
- String constants have double quote marks around them, and can be assigned to char pointers as shown below. Alternatively, you can assign a string constant to a char array - either with no size specified, or you can specify a size, but don't forget to leave a space for the null character!

```
char *string_1 = "Hello";
```

```
char string_2[] = "Hello";
char string_3[6] = "Hello";
```

Reading and Writing Strings:

One possible way to read in a string is by using scanf. However, the problem with this, is that if you were to enter a string which contains one or more spaces, scanf would finish reading when it reaches a space, or if return is pressed. As a result, the string would get cut off. So we could use the gets function

A gets takes just one argument - a char pointer, or the name of a char array, but don't forget to declare the array / pointer variable first! What's more, is that it automatically prints out a newline character, making the output a little neater.

A puts function is similar to gets function in the way that it takes one argument - a char pointer. This also automatically adds a newline character after printing out the string. Sometimes this can be a disadvantage, so printf could be used instead.

```
#include <stdio.h>

int main() {
char array1[50];
char *array2;

printf("Now enter another string less than 50");
printf(" characters with spaces: \n");
gets(array1);

printf("\nYou entered: ");
puts(array1);

printf("\nTry entering a string less than 50");
printf(" characters, with spaces: \n");
scanf("%s", array2);

printf("\nYou entered: %s\n", array2);

return 0;
}
```

This will produce following result:

```
Now enter another string less than 50 characters with spaces:
hello world

You entered: hello world

Try entering a string less than 50 characters, with spaces:
hello world

You entered: hello
```

String Manipulation Functions:

- char *strcpy (char *dest, char *src);
  Copy src string into dest string.
- char *strncpy(char *string1, char *string2, int n);
  Copy first n characters of string2 to stringl .
- int strcmp(char *string1, char *string2);
  Compare string1 and string2 to determine alphabetic order.
- int strncmp(char *string1, char *string2, int n);
  Compare first n characters of two strings.
- int strlen(char *string);
  Determine the length of a string.
- char *strcat(char *dest, const char *src);
  Concatenate string src to the string dest.
- char *strncat(char *dest, const char *src, int n);
  Concatenate n characters from string src to the string dest.
- char *strchr(char *string, int c);
  Find first occurrence of character c in string.
- char *strrchr(char *string, int c);
  Find last occurrence of character c in string.
- char *strstr(char *string2, char string*1);
  Find first occurrence of string string1 in string2.
- char *strtok(char *s, const char *delim) ;
  Parse the string s into tokens using delim as delimiter.

Structured Datatypes

- A structure in C is a collection of items of different types. You can think of a structure as a "record" is in Pascal or a class in Java without methods.
- Structures, or structs, are very useful in creating data structures larger and more complex than the ones we have discussed so far.
- Simply you can group various built-in data types into a structure.
- Object concepts was derived from Structure concept. You can achieve few object oriented goals using C structure but it is very complex.

Following is the example how how to define a structure.

```
struct student
{
char firstName[20];
char lastName[20];
char SSN[9];
float gpa;
};
```

Now you have a new datatype called student and you can use this datatype define your variables of student type:

```
struct student student_a, student_b;

or an array of students as

struct student students[50];
```

Another way to declare the same thing is:

```
struct {
char firstName[20];
char lastName[20];
char SSN[10];
float gpa;
} student_a, student_b;
```

All the variables inside an structure will be accessed using these values as student_a.firstName will give value of firstName variable. Similarly we can access other variables.

Structure Example:

Try out following example to understand the concept:

```
#include <stdio.h>
struct student {
char firstName[20];
char lastName[20];
char SSN[10];
float gpa;
};

main()
{
struct student student_a;

strcpy(student_a.firstName, "Deo");
strcpy(student_a.lastName, "Dum");
strcpy(student_a.SSN, "2333234" );
student_a.gpa = 2009.20;

printf( "First Name: %s\n", student_a.firstName );
printf( "Last Name: %s\n", student_a.lastName );
printf( "SNN : %s\n", student_a.SSN );
printf( "GPA : %f\n", student_a.gpa );
}
```

This will produce following results:

```
First Name: Deo
Last Name: Dum
SSN : 2333234
GPA : 2009.
```

Pointers to Structs:

Sometimes it is useful to assign pointers to structures (this will be evident in the next section with self-referential structures). Declaring pointers to structures is basically the same as declaring a normal pointer:

```
struct student *student_a;
```

To dereference, you can use the infix operator: ->.

```
printf("%s\n", student_a->SSN);
```

typedef Keyword

There is an easier way to define structs or you could "alias" types you create. For example:

```
typedef struct{
char firstName[20];
char lastName[20];
char SSN[10];
float gpa;
}student;
```

Now you can use student directly to define variables of student type without using struct keyword. Following is the example:

```
student student_a;
```

You can use typedef for non-structs:

```
typedef long int *pint32;

pint32 x, y, z;
```

x, y and z are all pointers to long ints

Unions Datatype

Unions are declared in the same fashion as structs, but have a fundamental difference. Only one item within the union can be used at any time, because the memory allocated for each item inside the union is in a shared memory location.

Here is how we define a Union

```
union Shape
{
int circle;
int triangle;
int ovel;
};
```

We use union in such case where only one condition will be applied and only one variable will be used.

Conclusion:

You can create arrays of structs.
Structs can be copied or assigned.
The & operator may be used with structs to show addresses.
Structs can be passed into functions. Structs can also be returned from functions.
Structs cannot be compared!

Structures can store non-homogenous data types into a single collection, much like an array does for common data (except it isn't accessed in the same manner).
Pointers to structs have a special infix operator: -> for dereferencing the pointer.
typedef can help you clear your code up and can help save some keystrokes.

Working with Files

When accessing files through C, the first necessity is to have a way to access the files. For C File I/O you need to use a FILE pointer, which will let the program keep track of the file being accessed. For Example:

```
FILE *fp;
```

To open a file you need to use the fopen function, which returns a FILE pointer. Once you've opened a file, you can use the FILE pointer to let the compiler perform input and output functions on the file.

```
FILE *fopen(const char *filename, const char *mode);
```

Here filename is string literal which you will use to name your file and mode can have one of the following values

```
w - open for writing (file need not exist)
a - open for appending (file need not exist)
r+ - open for reading and writing, start at beginning
w+ - open for reading and writing (overwrite file)
a+ - open for reading and writing (append if file exists)
```

Note that it's possible for fopen to fail even if your program is perfectly correct: you might try to open a file specified by the user, and that file might not exist (or it might be write-protected). In those cases, fopen will return 0, the NULL pointer.

Here's a simple example of using fopen:

```
FILE *fp;

fp=fopen("/home/techpreparation/test.txt", "r");
```

This code will open test.txt for reading in text mode. To open a file in a binary mode you must add a b to the end of the mode string; for example, "rb" (for the reading and writing modes, you can add the b either after the plus sign - "r+b" - or before - "rb+")

To close a function you can use the function:

```
int fclose(FILE *a_file);
```

fclose returns zero if the file is closed successfully.

An example of fclose is:

```
fclose(fp);
```

To work with text input and output, you use fprintf and fscanf, both of which are similar to their friends printf and scanf except that you must pass the FILE pointer as first argument.

Try out following example:

```
#include <stdio.h>

main()
{
FILE *fp;

fp = fopen("/tmp/test.txt", "w");
fprintf(fp, "This is testing...\n");
fclose(fp;);
}
```

This will create a file test.txt in /tmp directory and will write This is testing in that file.

Here is an example which will be used to read lines from a file:

```
#include <stdio.h>

main()
{
FILE *fp;
char buffer[20];

fp = fopen("/tmp/test.txt", "r");
fscanf(fp, "%s", buffer);
printf("Read Buffer: %s\n", %buffer );
flcose(fp;);

}
```

It is also possible to read (or write) a single character at a time--this can be useful if you wish to perform character-by-character input. The fgetc function, which takes a file pointer, and returns an int, will let you read a single character from a file:

```
int fgetc (FILE *fp);
```

The fgetc returns an int. What this actually means is that when it reads a normal character in the file, it will return a value suitable for storing in an unsigned char (basically, a number in the range 0 to 255). On the other hand, when you're at the very end of the file, you can't get a character value--in this case, fgetc will return "EOF", which is a constant that indicates that you've reached the end of the file.

The fputc function allows you to write a character at a time--you might find this useful if you wanted to copy a file character by character. It looks like this:

```
int fputc( int c, FILE *fp );
```

Note that the first argument should be in the range of an unsigned char so that it is a valid character. The second argument is the file to write to. On success, fputc will return the value c, and on failure, it will return EOF.

Binary I/O

There are following two functions which will be used for binary input and output:

```
size_t fread(void *ptr, size_t size_of_elements,
size_t number_of_elements, FILE *a_file);

size_t fwrite(const void *ptr, size_t size_of_elements,
size_t number_of_elements, FILE *a_file);
```

Both of these functions deal with blocks of memories - usually arrays. Because they accept pointers, you can also use these functions with other data structures; you can even write structs to a file or a read struct into memory.

Bits Manipulation

Bit manipulation is the act of algorithmically manipulating bits or other pieces of data shorter than a byte. C language is very efficient in manipulating bits.

Here are following operators to perform bits manipulation:
Bitwise Operators:

Bitwise operator works on bits and perform bit by bit operation.

Assume if B = 60; and B = 13; Now in binary format they will be as follows:

A = 0011 1100

B = 0000 1101

A&B = 0000 1000

A|B = 0011 1101

A^B = 0011 0001

~A = 1100 0011

Show Examples

There are following Bitwise operators supported by C language

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one	(A ^ B) will give 49

	operand but not both.	which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -60 which is 1100 0011
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 0000 1111

The shift operators perform appropriate shift by operator on the right to the operator on the left. The right operator must be positive. The vacated bits are filled with zero.

For example: x << 2 shifts the bits in x by 2 places to the left.

```
if x = 00000010 (binary) or 2 (decimal)

then:
x >>= 2 => x = 00000000 or just 0 (decimal)

Also: if x = 00000010 (binary) or 2 (decimal)
then
x <<= 2 => x = 00001000 or 8 (decimal)
```

Therefore a shift left is equivalent to a multiplication by 2. Similarly a shift right is equal to division by 2. Shifting is much faster than actual multiplication (*) or division (/) by 2. So if you want fast multiplications or division by 2 use shifts.

To illustrate many points of bitwise operators let us write a function, Bitcount, that counts bits set to 1 in an 8 bit number (unsigned char) passed as an argument to the function.

```
int bitcount(unsigned char x)
{
int count;

for ( count=0; x != 0; x>>=1);
{
if ( x & 01)
count++;
}

return count;
}
```

This function illustrates many C program points:

- for loop not used for simple counting operation.
- x >>= 1 => x = x>> 1;
- for loop will repeatedly shift right x until x becomes 0
- use expression evaluation of x & 01 to control if

- x & 01 masks of 1st bit of x if this is 1 then count++

Bit Fields

Bit Fields allow the packing of data in a structure. This is especially useful when memory or data storage is at a premium. Typical examples:

- Packing several objects into a machine word. e.g. 1 bit flags can be compacted.
- Reading external file formats -- non-standard file formats could be read in. E.g. 9 bit integers.

C allows us do this in a structure definition by putting :bit length after the variable. For example:

```
struct packed_struct {
unsigned int f1:1;
unsigned int f2:1;
unsigned int f3:1;
unsigned int f4:1;
unsigned int type:4;
unsigned int my_int:9;
} pack;
```

Here the packed_struct contains 6 members: Four 1 bit flags f1..f3, a 4 bit type and a 9 bit my_int.

C automatically packs the above bit fields as compactly as possible, provided that the maximum length of the field is less than or equal to the integer word length of the computer. If this is not the case then some compilers may allow memory overlap for the fields whilst other would store the next field in the next word.

Pre-Processors

The C Preprocessor is not part of the compiler, but is a separate step in the compilation process. In simplistic terms, a C Preprocessor is just a text substitution tool. We'll refer to the C Preprocessor as the CPP.

All preprocessor lines begin with #

- The unconditional directives are:
  #include - Inserts a particular header from another file
  #define - Defines a preprocessor macro
  #undef - Undefines a preprocessor macro
- The conditional directives are:
  #ifdef - If this macro is defined
  #ifndef - If this macro is not defined
  #if - Test if a compile time condition is true
  #else - The alternative for #if
  #elif - #else an #if in one statement
  #endif - End preprocessor conditional
- Other directives include:
  # - Stringization, replaces a macro parameter with a string constant
  ## - Token merge, creates a single token from two adjacent ones

Pre-Processors Examples:

Analyze following examples to understand various directives

```
#define MAX_ARRAY_LENGTH 20
```

Tells the CPP to replace instances of MAX_ARRAY_LENGTH with 20. Use #define for constants to increase readability.

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

Tells the CPP to get stdio.h from System Libraries and add the text to this file. The next line tells CPP to get myheader.h from the local directory and add the text to the file.

```
#undef FILE_SIZE
#define FILE_SIZE 42
```

Tells the CPP to undefined FILE_SIZE and define it for 42.

```
#ifndef MESSAGE
#define MESSAGE "You wish!"
#endif
```

Tells the CPP to define MESSAGE only if MESSAGE isn't defined already.

```
#ifdef DEBUG
/* Your debugging statements here */
#endif
```

Tells the CPP to do the following statements if DEBUG is defined. This is useful if you pass the -DDEBUG flag to gcc. This will define DEBUG, so you can turn debugging on and off on the fly!

Stringize (#):

The stringize or number-sign operator ('#'), when used within a macro definition, converts a macro parameter into a string constant. This operator may be used only in a macro that has a specified argument or parameter list.

When the stringize operator immediately precedes the name of one of the macro parameters, the parameter passed to the macro is enclosed within quotation marks and is treated as a string literal. For example:

```
#include <stdio.h>

#define message_for(a, b) \
printf(#a " and " #b ": We love you!\n")

int main(void)
{
message_for(Carole, Debra);
return 0;
```

```
}
```

This will produce following result using stringization macro message_for

```
Carole and Debra: We love you!
```

Token Pasting (##):

The token-pasting operator (##) within a macro definition combines two arguments. It permits two separate tokens in the macro definition to be joined into a single token.

If the name of a macro parameter used in the macro definition is immediately preceded or followed by the token-pasting operator, the macro parameter and the token-pasting operator are replaced by the value of the passed parameter. Text that is adjacent to the token-pasting operator that is not the name of a macro parameter is not affected. For example:

```
#define tokenpaster(n) printf ("token" #n " = %d", token##n)

tokenpaster(34);
```

This example results in the following actual output from the preprocessor:

```
printf ("token34 = %d", token34);
```

This example shows the concatenation of token##n into token34. Both the stringize and the token-pasting operators are used in this example.

Parameterized Macros:

One of the powerful functions of the CPP is the ability to simulate functions using parameterized macros. For example, we might have some code to square a number:

```
int square(int x)
{
return x * x;
}
```

We can instead rewrite this using a macro:

```
#define square(x) ((x) * (x))
```

Macros with arguments must be defined using the #define directive before they can be used. The argument list is enclosed in parentheses and must immediately follow the macro name. Spaces are not allowed between and macro name and open parenthesis. For example:

```
#define MAX(x,y) ((x) > (y) ? (x) : (y))
```

Macro Caveats:

- Macro definitions are not stored in the object file. They are only active for the duration of a single source file starting when they are defined and ending when they are undefined (using #undef), redefined, or when the end of the source file is found.
- Macro definitions you wish to use in multiple source files may be defined in an include file which may be included in each source file where the macros are required.
- When a macro with arguments is invoked, the macro processor substitutes the arguments into the macro body and then processes the results again for additional macro calls. This makes it possible, but confusing, to piece together a macro call from the macro body and from the macro arguments.
- Most experienced C programmers enclose macro arguments in parentheses when they are used in the macro body. This technique prevents undesired grouping of compound expressions used as arguments and helps avoid operator precedence rules overriding the intended meaning of a macro.
- While a macro may contain references to other macros, references to itself are not expanded. Self-referencing macros are a special feature of ANSI Standard C in that the self-reference is not interpreted as a macro call. This special rule also applies to indirectly self-referencing macros (or macros that reference themselves through another macro).

Useful Concepts

Error Reporting:

Many times it is useful to report errors in a C program. The standard library perror() is an easy to use and convenient function. It is used in conjunction with errno and frequently on encountering an error you may wish to terminate your program early. We will meet these concepts in other parts of the function reference chapter also.

void perror(const char *message) - produces a message on standard error output describing the last error encountered.

errno: - is a special system variable that is set if a system call cannot perform its set task. It is defined in #include <errno.h>.

Predefined Streams:

UNIX defines 3 predefined streams ie. virtual files

```
stdin, stdout, stderr
```

They all use text a the method of I/O. stdin and stdout can be used with files, programs, I/O devices such as keyboard, console, etc.. stderr always goes to the console or screen.

The console is the default for stdout and stderr. The keyboard is the default for stdin.

Dynamic Memory Allocation:

Dynamic allocation is a pretty unique feature to C. It enables us to create data types and structures of any size and length to suit our programs need within the program. We use dynamic memory allocation concept when we don't know how in advance about memory requirement.

There are following functions to use for dynamic memory manipulation:

void *calloc(size_t num elems, size_t elem_size) - Allocate an array and initialise all elements to zero .

void free(void *mem address) - Free a block of memory.

void *malloc(size_t num bytes) - Allocate a block of memory.

void *realloc(void *mem address, size_t newsize) - Reallocate (adjust size) a block of memory.

Command Line Arguments:

It is possible to pass arguments to C programs when they are executed. The brackets which follow main are used for this purpose. argc refers to the number of arguments passed, and argv[] is a pointer array which points to each argument which is passed to mainA simple example follows, which checks to see if a single argument is supplied on the command line when the program is invoked.

```
#include <stdio.>h
main( int argc, char *argv[] )
{
if( argc == 2 )
printf("The argument supplied is %s\n", argv[1]);
else if( argc > 2 )
printf("Too many arguments supplied.\n");
else
printf("One argument expected.\n");
}
```

Note that *argv[0] is the name of the program invoked, which means that *argv[1] is a pointer to the first argument supplied, and *argv[n] is the last argument. If no arguments are supplied, argc will be one. Thus for n arguments, argc will be equal to n + 1. The program is called by the command line:

```
$myprog argument1
```

More clearly, Suppose a program is compiled to an executable program myecho and that the program is executed with the following command.

```
$myeprog aaa bbb ccc
```

When this command is executed, the command interpreter calls the main() function of the myprog program with 4 passed as the argc argument and an array of 4 strings as the argv argument.

```
argv[0]  -  "myprog"
argv[1]  -  "aaa"
argv[2]  -  "bbb"
argv[3]  -  "ccc"
```

Multidimensional Arrays:

The array we used in the last example was a one dimensional array. Arrays can have

more than one dimension, these arrays-of-arrays are called multidimensional arrays. They are very similar to standard arrays with the exception that they have multiple sets of square brackets after the array identifier. A two dimensional array can be though of as a grid of rows and columns.

```
#include <stdio.h>

const int num_rows = 7;
const int num_columns = 5;

int
main()
{
int box[num_rows][num_columns];
int row, column;

for(row = 0; row < num_rows; row++)
for(column = 0; column < num_columns; column++)
box[row][column] = column + (row * num_columns);

for(row = 0; row < num_rows; row++)
{
for(column = 0; column < num_columns; column++)
{
printf("%4d", box[row][column]);
}
printf("\n");
}
return 0;
}
```

This will produce following result:

```
 0   1   2   3   4
 5   6   7   8   9
10  11  12  13  14
15  16  17  18  19
20  21  22  23  24
25  26  27  28  29
30  31  32  33  34
```

The above array has two dimensions and can be called a doubly subscripted array. GCC allows arrays of up to 29 dimensions although actually using an array of more than three dimensions is very rare.

String Manipulation Functions:

- char *strcpy (char *dest, char *src);
  Copy src string into dest string.
- char *strncpy(char *string1, char *string2, int n);
  Copy first n characters of string2 to stringl .

- int strcmp(char *string1, char *string2);
  Compare string1 and string2 to determine alphabetic order.
- int strncmp(char *string1, char *string2, int n);
  Compare first n characters of two strings.
- int strlen(char *string);
  Determine the length of a string.
- char *strcat(char *dest, const char *src);
  Concatenate string src to the string dest.
- char *strncat(char *dest, const char *src, int n);
  Concatenate n characters from string src to the string dest.
- char *strchr(char *string, int c);
  Find first occurrence of character c in string.
- char *strrchr(char *string, int c);
  Find last occurrence of character c in string.
- char *strstr(char *string2, char string*1);
  Find first occurrence of string string1 in string2.
- char *strtok(char *s, const char *delim) ;
  Parse the string s into tokens using delim as delimiter.

Memory Management Functions:

- void *calloc(int num elems, int elem_size);
  Allocate an array and initialise all elements to zero .
- void free(void *mem address);
  Free a block of memory.
- void *malloc(int num bytes);
  Allocate a block of memory.
- void *realloc(void *mem address, int newsize);
  Reallocate (adjust size) a block of memory.

Buffer Manipulation:

- void* memcpy(void* s, const void* ct, int n);
  Copies n characters from ct to s and returns s. s may be corrupted if objects overlap.
- int memcmp(const void* cs, const void* ct, int n);
  Compares at most (the first) n characters of cs and ct, returning negative value if cs<ct, zero if cs==ct, positive value if cs>ct.
- void* memchr(const void* cs, int c, int n);
  Returns pointer to first occurrence of c in first n characters of cs, or NULL if not found.
- void* memset(void* s, int c, int n);
  Replaces each of the first n characters of s by c and returns s.
- void* memmove(void* s, const void* ct, int n);
  Copies n characters from ct to s and returns s. s will not be corrupted if objects overlap.

Character Functions:

- int isalnum(int c);
  The function returns nonzero if c is alphanumeric

- int isalpha(int c);
  The function returns nonzero if c is alphabetic only
- int iscntrl(int c);
  The function returns nonzero if c is a control character
- int isdigit(int c);
  The function returns nonzero if c is a numeric digit
- int isgraph(int c);
  The function returns nonzero if c is any character for which either isalnum or ispunct returns nonzero.
- int islower(int c);
  The function returns nonzero if c is a lower case character.
- int isprint(int c);
  The function returns nonzero if c is space or a character for which isgraph returns nonzero.
- int ispunct(int c);
  The function returns nonzero if c is punctuation
- int isspace(int c);
  The function returns nonzero if c is space character
- int isupper(int c);
  The function returns nonzero if c is upper case character
- int isxdigit(int c);
  The function returns nonzero if c is hexa digit
- int tolower(int c);
  The function returns the corresponding lowercase letter if one exists and if isupper(c); otherwise, it returns c.
- int toupper(int c);
  The function returns the corresponding uppercase letter if one exists and if islower(c); otherwise, it returns c.

Error Handling Functions:

- void perror(const char *s);
  produces a message on standard error output describing the last error encountered.
- char *strerror(int errnum);
  returns a string describing the error code passed in the argument errnum.