History of C

The ***milestones*** in C's development as a language are listed below:

Initially developed by Dennis Ritchie between 1969 and 1973 at AT&T Bell Labs.

- UNIX developed c. 1969 -- DEC PDP-7 Assembly Language
- BCPL -- a user friendly OS providing powerful development tools developed from BCPL. Assembler tedious long and error prone.
- A new language ``B'' a second attempt. c. 1970.
- A totally new language ``C'' a successor to ``B''. c. 1971
- By 1973 UNIX OS almost totally written in ``C''.

C is a general-purpose language which has been closely associated with the UNIX operating system for which it was developed - since the system and most of the programs that run it are written in C.

Many of the important ideas of C stem from the language BCPL, developed by Martin Richards. The influence of BCPL on C proceeded indirectly through the language B, which was written by Ken Thompson in 1970 at Bell Labs, for the first UNIX system on a DEC PDP-7. BCPL and B are "type less" languages whereas C provides a variety of data types.

In 1972 Dennis Ritchie at Bell Labs writes C and in 1978 the publication of The C Programming Language by Kernighan & Ritchie caused a revolution in the computing world.

In 1983, the American National Standards Institute (ANSI) established a committee to provide a modern, comprehensive definition of C. The resulting definition, the ANSI standard, or "ANSI C", was completed late 1988.

Characteristics of C

We briefly list some of C's characteristics that define the language and also have lead to its popularity as a programming language. Naturally we will be studying many of these aspects throughout the course.

- Small size
- Extensive use of function calls
- Loose typing -- unlike PASCAL
- Structured language
- Low level (BitWise) programming readily available
- Pointer implementation - extensive use of pointers for memory, array, structures and functions.

C has now become a widely used professional language for various reasons.

- It has high-level constructs.
- It can handle low-level activities.
- It produces efficient programs.
- It can be compiled on a variety of computers.
- Multipurpose and multiplatform

Its main drawback is that it has poor error detection which can make it off putting to the beginner. However diligence in this matter can pay off handsomely since having learned the rules of C we can break them. Not many languages allow this. This if done properly and carefully leads to the power of C programming.

Uses of C Language:

C was initially used for system development work, in particular the programs that make-up the operating system. Why use C? Mainly 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

In recent years C has been used as a general-purpose language because of its popularity with programmers. It is not the world's easiest language to learn and you will certainly benefit if you are not learning C as your first programming language! C is trendy (I nearly said sexy) - many well established programmers are switching to C for all sorts of reasons, but mainly because of the portability that writing standard C programs can offer.

Why use C?

C has been used successfully for every type of programming problem imaginable from operating systems to spreadsheets to expert systems - and efficient compilers are available for machines ranging in power from the Apple Macintosh to the Cray supercomputers. The largest measure of C's success seems to be based on purely practical considerations:

- the portability of the compiler;
- the standard library concept;
- a powerful and varied repertoire of operators;
- an elegant syntax;
- ready access to the hardware when needed;
- and the ease with which applications can be optimised by hand-coding isolated procedures

C is often called a "Middle Level" programming language. This is not a reflection on its lack of programming power but more a reflection on its capability to access the system's low level functions. Most high-level languages (e.g. Fortran) provides everything the programmer might want to do already built into the language. A low level language (e.g. assembler) provides nothing other than access to the machines basic instruction set. A middle level language, such as C, probably doesn't supply all the constructs found in high-languages - but it provides you with all the building blocks that you will need to produce the results you want!

Difference between C & C++:

1. C follows the procedural programming paradigm while C++ is a multi-paradigm language (procedural as well as object oriented)

In case of C, importance is given to the steps or procedure of the program while C++ focuses on the data rather than the process.

Also, it is easier to implement/edit the code in case of C++ for the same reason.

2. In case of C, the data is not secured while the data is secured(hidden) in C++

This difference is due to specific OOP features like Data Hiding which are not present in C.

3. C is a low-level language while C++ is a middle-level language (Relatively, Please see the discussion at the end of the post)

C is regarded as a low-level language(difficult interpretation & less user friendly) while C++ has features of both low-level(concentration on whats going on in the machine hardware) & high-level languages(concentration on the program itself) & hence is regarded as a middle-level language.

4. C uses the top-down approach while C++ uses the bottom-up approach

In case of C, the program is formulated step by step, each step is processed into detail while in C++, the base elements are first formulated which then are linked together to give rise to larger systems.

5. C is function-driven while C++ is object-driven

Functions are the building blocks of a C program while objects are building blocks of a C++ program.

6. C++ supports function overloading while C does not

Overloading means two functions having the same name in the same program. This can be done only in C++ with the help of Polymorphism(an OOP feature)

7. We can use functions inside structures in C++ but not in C.

In case of C++, functions can be used inside a structure while structures cannot contain functions in C.

8. The NAMESPACE feature in C++ is absent in case of C

C++ uses NAMESPACE which avoid name collisions. For instance, two students enrolled in the same university cannot have the same roll number while two students in different universities might have the same roll number. The universities are two different namespace & hence contain the same roll number(identifier) but the same university(one namespace) cannot have two students with the same roll number(identifier)

9. The standard input & output functions differ in the two languages

C uses scanf & printf while C++ uses cin>> & cout<< as their respective input & output functions

10. C++ allows the use of reference variables while C does not

Reference variables allow two variable names to point to the same memory location. We cannot use these variables in C programming.

C Program Structure

A C program basically has the following form:

- Preprocessor Commands
- Type definitions
- Function prototypes -- declare function types and variables passed to function.
- Variables
- Functions

We must have a main() function.

A function has the form:

```
type function_name (parameters)
                    {
                                        local variables

                                        C Statements

                    }
```

If the type definition is omitted C assumes that function returns an **integer** type. **NOTE:** This can be a source of problems in a program.

So returning to our first C program:

```
 /* Sample program */

                     main()
                                             {

                                             printf( ``I Like C \n" );
                                             exit ( 0 );

                                             }
```

NOTE:

- C requires a semicolon at the end of **every** statement.
- printf is a ***standard*** C function -- called from main.
- \n signifies newline. **Formatted output** -- more later.
- exit() is also a standard function that causes the program to terminate. Strictly speaking it is not needed here as it is the last line of main() and the program will terminate anyway.

Let us look at another printing statement:

```
printf(``.\n.1\n..2\n...3\n");
```

The output of this would be:

```
  .
                    .1
                    ..2
                    ...3
```

Variables:

Variable is used to store some value in computer memory; its value can be changed in a program. On the other hand the value of constant cant b changed.

C has the following simple data types:

C type	Size (bytes)	Lower bound	Upper bound
char	1	—	—
unsigned char	1	0	255
short int	2	−32768	+32767
unsigned short int	2	0	65536
(long) int	4	-2^{31}	$+2^{31}-1$
float	4	$-3.2 \times 10^{\pm 38}$	$+3.2 \times 10^{\pm 38}$
double	8	$-1.7 \times 10^{\pm 308}$	$+1.7 \times 10^{\pm 308}$

Rules to Declare Variables

1) A Variable name consists of any combination of alphabets, digits and underscores. Some compiler allows variable names whole length could be up to 247 characters. Still it would be safer to stick to the rule of 31 characters. Please avoid creating long variable name as it adds to your typing effort.

2) The first character of the variable name must either be alphabet or underscore. It should not start with the digit.

3) No commas and blanks are allowed in the variable name.

4) No special symbols other than underscore are allowed in the variable name.

We need to declare the type of the variable name before making use of that name in the program. Type declaration can be done as follows:

To declare a variable as integer, follow the below syntax:

int variable_name;

Here int is the type of the variable named variable_name. ‘int’ denotes integer type.

Following are the examples of type declaration statements:

E.g.: int p, n;

float r;

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 initialise a const with a value as it cannot get a value ***any other way***.

The preprocessor #define is another more flexible method to define ***constants*** in a program.

Types of Constants:

1. Numeric Constant
   a. Integer
   b. Float
   c. Exponential

2. Non Numeric Constant
   a. Character
   b. String

Printing Out and Inputting Variables

C uses formatted output. The printf function has a special formatting character (%) -- a character following this defines a certain format for a variable:

```
%c -- characters
                    %d -- integers
                    %f -- floats
```

e.g. printf(``%c %d %f'',ch,i,x);

NOTE: Format statement enclosed in ``...'', variables follow after. Make sure order of format and variable data types match up.

scanf() is the function for inputting values to a data structure: Its format is similar to printf:

i.e. scanf(``%c %d %f'',&ch,&i,&x);

Arithmetic Operations

As well as the standard arithmetic operators (+ - * /) found in most languages, C provides some more operators. There are some notable differences with other languages, such as Pascal.

Assignment is = ***i.e.*** *i* = 4; ch = `y';

Increment ++, Decrement -- which are more efficient than their long hand equivalents, for example:-- x++ is faster than x=x+1.

The ++ and -- operators can be either in post-fixed or pre-fixed. With pre-fixed the value is computed before the expression is evaluated whereas with post-fixed the value is computed after the expression is evaluated.

In the example below, ++z is pre-fixed and the w-- is post-fixed:

```
int x,y,w;

                    main()
                                          {

                                          x=((++z)-(w--)) % 100;

                                          }
```

This would be equivalent to:

```
int x,y,w;

                    main()
                                          {

                                          z++;
                                          x=(z-w) % 100;
                                          w--;

                                          }
```

The % (modulus) operator only works with integers.

Division / is for both integer and float division. So be careful.

The answer to: $x = 3 / 2$ is 1 even if x is declared a float!!

RULE: If both arguments of / are integer then do integer division.

So make sure you do this. The correct (for division) answer to the above is $x = 3.0 / 2$ or $x= 3 / 2.0$ or (better) $x = 3.0 / 2.0$.

There is also a convenient **shorthand** way to express computations in C.

It is very common to have expressions like: $i = i + 3$ or $x = x*(y + 2)$

This can written in C (generally) in a ***shorthand*** form like this:

$$expr_1 \; op = expr_2$$

which is equivalent to (but more efficient than):

$$expr_1 = expr_1 \; op \; expr_2$$

So we can rewrite $i = i + 3$ as $i += 3$

and $x = x*(y + 2)$ as x *= $y + 2$.

NOTE: that x *= $y + 2$ means $x = x*(y + 2)$ and **NOT** $x = x*y + 2$.

Comparison Operators

To test for equality is = =

A warning: Beware of using ``=" instead of ``= =", such as writing accidentally

```
if ( i = j ) .....
```

This is a perfectly **LEGAL** C statement (syntactically speaking) which copies the value in "j" into "i", and delivers this value, which will then be interpreted as TRUE if j is non-zero. This is called **assignment by value** -- a key feature of C.

Not equals is: !=

Other operators < (less than) , > (grater than), <= (less than or equals), >= (greater than or equals) are as usual.

Logical Operators

Logical operators are usually used with conditional statements.

The two basic logical operators are:

&& for logical AND, || for logical OR.

Beware & and | have a different meaning for bitwise AND and OR.

Order of Precedence

It is necessary to be careful of the meaning of such expressions as a + b * c

We may want the effect as either

```
(a + b) * c
```

or

```
a + (b * c)
```

All operators have a priority, and high priority operators are evaluated before lower priority ones. Operators of the same priority are evaluated from left to right, so that

```
a - b - c
```

is evaluated as

```
( a - b ) - c
```

as you would expect.

From high priority to low priority the order for all C operators is:

```
( ) [ ] -> .
                        ! ~ - * & sizeof cast ++ -
                          (these are right->left)
                        * / %
                        + -
                        < <= >= >
                        == !=
                        &
                        ^              |
                        &&
                        ||
                        ?:     (right->left)
                        = += -= (right->left)
                        ,  (comma)
```

Thus

```
a < 10 && 2 * b < c
```

is interpreted as

```
( a < 10 ) && ( ( 2 * b ) < c )
```

and

```
a = b = spokes / spokes_per_wheel + spares;
```

as

```
a =( b =( spokes / spokes_per_wheel )+ spares );
```

The ? operator

The ? (***ternary condition***) operator is a more efficient form for expressing simple if statements. It has the following form:

expression$_1$? *expression*$_2$: *expression*$_3$

It simply states:

if *expression*$_1$ then *expression*$_2$ else *expression*$_3$

For example to assign the maximum of a and b to z:

```
z = (a>b)? a : b;
```

which is the same as:

```
if (a>b)
   z = a;
 else
   z = b;
```

Expressions:

Combination of operators and operands is called expressions:

X=a+b-d/5

There are 4 types of expressions in C/C++:

Arithmetic:

In which arithmetic operators are used

X=a+b-d/5

Relational/Conditional

In which relational operators are used

If (a>b)   or if(marks<=70)

Logical:

In which logical operators are used

If (marks>=70 && marks<=80)

Incremental/Decremenatal

In which increment and decrement operators are used

A++

a--

Basic Data Types and Operators

The *type* of a variable determines what kinds of values it may take on. An *operator* computes new values out of old ones. An *expression* consists of variables, constants, and operators combined to perform some useful computation.

Types

There are only a few basic data types in C. The first ones we'll be encountering and using are:

- char a character
- int an integer, in the range -32,767 to 32,767
- long int a larger integer (up to +-2,147,483,647)
- float a floating-point number
- double a floating-point number, with more precision and perhaps greater range than float

The ranges listed above for types int and long int are the guaranteed minimum ranges. On some systems, either of these types (or, indeed, any C type) may be able to hold larger values, but a program that depends on extended ranges will not be as portable.

(From the ranges listed above, we can determine that type int must be at least 16 bits, and that type long int must be at least 32 bits. But neither of these sizes is exact; many systens have 32-bit ints, and some systems have 64-bit long ints.)

String

A *string* is represented in C as a sequence or array of characters. (We'll have more to say about arrays in general, and strings in particular, later.) A string constant is a sequence of zero or more characters enclosed in double quotes: "apple", "hello, world", "this is a test".

Escape Sequences

\n	a ``newline" character
\a	beep
\xdd	graphic character
\b	a backspace
\r	a carriage return (without a line feed)
\'	a single quote (e.g. in a character constant)
\"	a double quote (e.g. in a string constant)
\\	a single backslash

For example, "he said \"hi\"" is a string constant which contains two double quotes, and '\'' is a character constant consisting of a (single) single quote. Notice once again that the character constant 'A' is very different from the string constant "A".

Declarations

Informally, a *variable* (also called an *object*) is a place you can store a value. So that you can refer to it unambiguously, a variable needs a name. You can think of the variables in your program as a set of boxes or cubbyholes, each with a label giving its name; you might imagine that storing a value ``in" a variable consists of writing the value on a slip of paper and placing it in the cubbyhole.

A *declaration* tells the compiler the name and type of a variable you'll be using in your program. In its simplest form, a declaration consists of the type, the name of the variable, and a terminating semicolon:

```
char c;
int i;
float f;
```

You can also declare several variables of the same type in one declaration, separating them with commas:

```
int i1, i2;
```

Arithmetic Operators

The basic operators for performing arithmetic are the same in many computer languages:

+	addition
-	subtraction
*	multiplication
/	division
%	modulus (remainder)

The - operator can be used in two ways: to subtract two numbers (as in a - b), or to negate one number (as in -a + b or a + -b).

When applied to integers, the division operator / discards any remainder, so 1 / 2 is 0 and 7 / 4 is 1. But when either operand is a floating-point quantity (type float or double), the division operator yields a floating-point result, with a potentially nonzero fractional part. So 1 / 2.0 is 0.5, and 7.0 / 4.0 is 1.75.

The *modulus* operator % gives you the remainder when two integers are divided: 1 % 2 is 1; 7 % 4 is 3. (The modulus operator can only be applied to integers.)

An additional arithmetic operation you might be wondering about is exponentiation. Some languages have an exponentiation operator (typically ^ or **), but C doesn't. (To square or cube a number, just multiply it by itself.)

Multiplication, division, and modulus all have higher *precedence* than addition and subtraction. The term ``precedence'' refers to how ``tightly'' operators bind to their operands (that is, to the things they operate on). In mathematics, multiplication has higher precedence than addition, so 1 + 2 * 3 is 7, not 9. In other words, 1 + 2 * 3 is equivalent to 1 + (2 * 3). C is the same way.

Assignment Operators

[This section corresponds to K&R Sec. 2.10]

The assignment operator = assigns a value to a variable. For example,

```
x = 1
```

sets x to 1, and

```
a = b
```

sets a to whatever b's value is. The expression

```
i = i + 1
```

is, as we've mentioned elsewhere, the standard programming idiom for increasing a variable's value by 1: this expression takes i's old value, adds 1 to it, and stores it back into i.

Operator Type	**Operator**	**Description**	**Associativity**	**Precedence Level**
Parentheses, Braces	( ) [ ]	Parentheses Brackets	Left to right	1
Unary	++ -- + - ! ~ (type) & Size of	Unary pre increment / pre decrement Unary plus / minus Unary Logical negation / bitwise complement Unary cast Address Determine Size in bytes	Right to left	2
Binary	* / %	Multiplication / Division/ Modules	Left to right	3
	+ -	Addition / Subtraction	Left to right	4
	<< >>	Bitwise shift left, bitwise shift right	Left to right	5
	< <= > >=	Relational less than / less than or equal to Relational greater than / greater than or equal to	Left to right	6
	= = !=	Relational is equal to / is not equal to	Left to right	7
	&	Bitwise AND	Left to right	8
	^	Bitwise XOR	Left to right	9
	|	Bitwise OR	Left to right	10
	&&	Logical AND	Left to right	11
	||	Logical OR	Left to right	12
Ternary	? :	Ternary (Conditional)	Right to left	13

Assignment	= += -= *= /= %= &= ^= != <<= >>=	Assignment Addition/Subtraction Assignment Multiplication/Division Assignment Modulus/bitwise AND assignment Bitwise exclusive/inclusive OR assignment Bitwise Shift left/right assignment	Right to left	14
Comma	,	Comma (separate Expression)	Left to right	15

INPUT & OUTPUT STATEMENTS

Reading data, processing it and writing the processed data known as information or a result of a program are the essential functions of a program. The C is a functional language. It provides a number of macros and functions to enable the programmer to effectively carry out these input and output operations. Some of the input/output functions / macros in C are

1. getchar()	2. putchar()
3. scanf()	4. printf()
5. gets()	6. puts()

The data entered by the user through the standard input device and processed data is displayed on the standard output device.

I/O Functions:

I / O functions are grouped into two categories.

→ Unformatted I/O functions
→ Formatted I/O function.

The Formatted I/O functions allows programmers to specify the type of data and the way in which it should be read in or written out. On the other hand, unformatted I/O functions do not specify the type of data and the way is should be read or written. Amongst the above specified I/O functions scanf() and printf() are formatted I/O functions.

Function	Formatted	Unformatted
Input	scanf()	getchar(),gets()
Output	printf()	putchar(),puts()

Formatted Output – The printf function

C provides the printf function to display the data on the monitor. This function can be used to display any combination of numerical values, single characters and strings. The general form of printf statement

Function name

Function arguments

printf("my roll number is %d. \n", rollno)

control string print list

The %d is known as a placeholder or conversion character or format code or format specifier. At the time of display the value of the specified variable is substituted at the place of placeholder. In out example, value of rollno is substituted in place of %d. The printf uses different placeholders to display different type of data items.

Format Specifieres

Placeholder	Type of Data Item displayed
%c	Single character
%d	Signed decimal integer
%e	Floating point number with an exponent
%f	Floating point number without an exponent
%g	Floating point number either with exponent or without exponent depending on value. Trailing zeros and trailing decimal point will not be displayed.
%i	Singed decimal number
%o	Octal number, without leading zero
%s	String
%u	Unsigned decimal integer
%x	Hexadecimal number, without leading zero

Loops

Looping is a way by which we can execute any some set of statements more than one times continuously .In c there are mainly three types of loops are use :

- while Loop
- do while Loop

- For Loop

While Loop

Loops generally consist of two parts: one or more *control expressions* which (not surprisingly) control the execution of the loop, and the *body*, which is the statement or set of statements which is executed over and over.

The general syntax of a `while` loop is

```
        Initialization

    while( expression )
{
            Statement1
            Statement2
            Statement3

}
```

The most basic *loop* in C is the `while` loop. A `while` loop has one control expression, and executes as long as that expression is true. This example repeatedly doubles the number 2 (2, 4, 8, 16, ...) and prints the resulting numbers as long as they are less than 1000:

```
    int x = 2;

    while(x < 1000)
            {
            printf("%d\n", x);
            x = x * 2;
            }
```

(Once again, we've used braces `{}` to enclose the group of statements which are to be executed together as the body of the loop.)

For Loop

Our second loop, which we've seen at least one example of already, is the `for` loop. The general syntax of a `while` loop is

```
    for( Initialization;expression;Increments/decrements )
{
            Statement1
            Statement2
```

```
                    Statement3

}
```

The first one we saw was:

```
          for (i = 0; i < 10; i = i + 1)
                    printf ("i is %d\n", i);
```

(Here we see that the `for` loop has three control expressions. As always, the *statement* can be a brace-enclosed block.)

Do while Loop

This is very similar to the while loop except that the test occurs at the end of the loop body. This guarantees that the loop is executed at least once before continuing. Such a setup is frequently used where data is to be read. The test then verifies the data, and loops back to read again if it was unacceptable.

```
do
{
  printf("Enter 1 for yes, 0 for no :");
      scanf("%d", &input_value);
} while (input_value != 1 && input_value != 0)
```

The break Statement

We have already met break in the discussion of the switch statement. It is used to exit from a loop or a switch, control passing to the first statement beyond the loop or a switch.

With loops, break can be used to force an early exit from the loop, or to implement a loop with a test to exit in the middle of the loop body. A break within a loop should always be protected within an if statement which provides the test to control the exit condition.

The continue Statement

This is similar to break but is encountered less frequently. It only works within loops where its effect is to force an immediate jump to the loop control statement.

- In a while loop, jump to the test statement.
- In a do while loop, jump to the test statement.
- In a for loop, jump to the test, and perform the iteration.

Like a break, continue should be protected by an if statement. You are unlikely to use it very often.

Take the following example:

```
int i;
for (i=0;i<10;i++)
{

if (i==5)
continue;
printf("%d",i);
if (i==8)
break;
}
```

This code will print 1 to 8 except 5.

Continue means, whatever code that follows the continue statement WITHIN the loop code block will not be exectued and the program will go to the next iteration, in this case, when the program reaches i=5 it checks the condition in the if statement and executes 'continue', everything after continue, which are the printf statement, the next if statement, will not be executed.

Break statement will just stop execution of the look and go to the next statement after the loop if any. In this case when i=8 the program will jump out of the loop. Meaning, it wont continue till i=9, 10.

Comments:

- The compiler is "line oriented", and parses your program in a line-by-line fashion.
- There are two kinds of comments: single-line and multi-line comments.
- The single-line comment is indicated by "//"

  This means everything after the first occurrence of "//", UP TO THE END OF CURRENT LINE, is ignored.

- The multi-line comment is indicated by the pair "/*" and "*/".

  This means that everything between these two sequences will be ignored. This may ignore any number of lines.

Here is a variant of our first program:

```
/* This is a variant of my first program.
* It is not much, I admit.
*/
int main() {
printf("Hello World!\n");    // that is all?
return(0);
}
```

Conditions in C Language:

In programming we use conditions to make and check logic in programming, there are 3 types of conditions in C and C++.

1. If
2. If else
3. Switch

The if statement in C Programming Language

Here, if is the keyword in C.
Syntax:

```
if (condition is true){
/*block of statements to be executed*/
}
```

The keyword if tells the compiler that what follows is decision control statement. The block of statements to be executed must be enclosed in opening and closing braces. If only one statement needs to be executed then braces need not be used. Let us understand if statement using an example.

Example:

```
int a = 10;
if (a == 10){
printf("You are in if block\n");
printf("The value of a is %d\n", a);
}
printf("You are out of if block");
```

In the above example we have assigned value 10 to a. In the next line, there is an if statement. It is read as "If a is equal to 10 then perform the block of statements enclosed within the braces". After the execution of this block normal sequence flow of the program is executed.
The output of the above example is:

```
You are in if block
The value of a is 10
You are out of if block
```

Lets have a look at different expressions that can be used in if statement.

a == b read as a is equal to b
a!=b read as a is not equal to b
a < b read as a is less than b a > b read as a is greater than b

a<=b read as a is less than or equal to b a>=b read as a is greater than or equal to b

Example:

```
/*Demonstration of if statement*/
int marks;
printf("Enter the marks:");
scanf("%d", &marks);
if(marks>=35){
printf("Congrats!!!");
}
```

In the above program, it will prompt you to enter the marks. If you enter marks greater than or equal to 35, it will display Congrats!!! on the screen else it will do nothing.

The if-else Statement in C Programming Language
Here, if and else are the keywords in C.
Syntax:

```
if (condition is true){
/*1st block of statements*/
}
else{
/*2nd block of statements*/

}
```

The keyword if tells the compiler that what follows is decision control statement. The block of statements to be executed must be enclosed in opening and closing braces. If only one statement needs to be executed then braces need not be used. Let us understand if-else statement using an example.

Example:

```
int a = 10;
if (a == 10){
printf("You are in 1st block\n");
printf("The value of a is %d\n", a);
}
else{
printf("You are in 2nd block\n");
printf("The value of a is not equal to %d\n", a);
}
```

In the above example we have assigned value 10 to a. In the next line, there is an if statement. It is read as "If a is equal to 10 then perform the block of statements enclosed within the braces of if part else perform the block of statements enclosed within the braces of else part". After the execution of if-else statement normal sequence flow of the program is followed.
The output of the above example is:

```
You are in if block
You are in 1st block
The value of a is 10
```

The else-if statement

The else-if statement is nothing but the rearrangement of else with the if that follows. Consider the below program.

```
if(condition)
perform this
else{
if(condition)
perform this;
}
```

The above program can be re-written as,

```
if(condition)
perform this
else if(condition)
perform this;
```

Here, the else-if statement is used. Else if reduces the complexity of the program making it easier to understand.

The else-if statement is is used when ever we have multiple if conditions.
e.g.

```
if (condition 1)
{
do this;
}
else if (condition 2)
{
do this;
}
else if(condition 3)
{
do this;
}
else
{
do this;
}
```

The switch Statement in C Language

The switch statement is very powerful decision making statement. It reduces the complexity of the program. Hence increases the readability of the program. Switch statement accepts single input from the user and based on that input executes a particular block of statements.

Syntax of switch statement:

```
switch(expression)
{
 case constant 1:
 perform this;
 case constant 2:
 perform this;
 case constant 3:
 perform this;
 case constant 4:
 perform this;
 .
 .
 .
 default:
 perform this;
 }
```

Control passes to the statement whose case constant-expression matches the value of switch (expression). The switch statement can include any number of case instances, but no two case constants within the same switch statement can have the same value. Execution of the statement body begins at the selected statement and proceeds until the end of the body or until a break statement transfers control out of the body. Please make a note that using default case is optional.

Example 1:

```
 int a = 2;
 switch(a)
{
 case 1:
 printf("1");
case 2:
 printf("2");
}
```

Output:

```
 2
```

Example 2:

```
 int a = 2;
 switch(a)

{
```

```
case 1:
printf("1\n");
break;
case 2:
printf("2\n");
break;
default:
printf("No match")
}
Output:
2
```

Example 3:

```
char a = 'A';
switch(a){
case 'B':
printf("B\n");
break;
case 'A':
printf("A\n");
break;
default:
printf("No match")
}
Output:
A
```

Use of GOTO keyword

The goto statement is used for unconditional jump from one part of the program to another part of the program. It is always suggested not to use goto statement as this reduces the readability of the program. Using goto statement is considered as poor programming approach.

The goto statement consists of two parts: label and goto keyword.

Example:

```
/*use of goto statement*/

#include<stdio.h>

#include<conio.h>

void main(){
```

```
int a;

goto label;

a = 10;

printf("%d", a);

label:

a = 20;

printf("%d", a);

}
```

Output:

```
20
```

Functions:

In structural programming, the program is divided into small independent tasks. These tasks are small enough to be understood easily without having to understand the entire program at once. Each task is designed to perform specific functionality on its own. When these tasks are performed, their outcomes are combined together to solve the problem. Such a structural programming can be implemented using **modular programming**. In modular programming, the program is divided into separate small programs called modules. Each module is designed to perform specific function. Modules make out actual program shorter, hence easier to read and understand. A defined function or set of similar functions is coded in a separate module or sub module, which means that code can be loaded into memory more efficiently and that modules can be reused in other programs. After a module has been tested individually, it is then integrated with other modules into the overall program structure.

There are several advantages of modular / structural programming, some of them are

Reusability: Many programs require that a particular group of instructions accessed repeatedly, from several different places within the program. The repeated instructions can be placed within a single function, which can then be accessed wherever it is needed. This avoids rewriting of group of instructions on every access.

Easy Debugging: Since each function is smaller and has a logical clarity, it is easy to locate and correct errors in it.

Build Library: The use of functions allows a programmer to build a customized library of frequently used routines or system-dependent features. Each routine can programmed as a separate function and stored within a special library file. Building libraries reduce the time and space complexity and promote the portability.

Basics of Functions

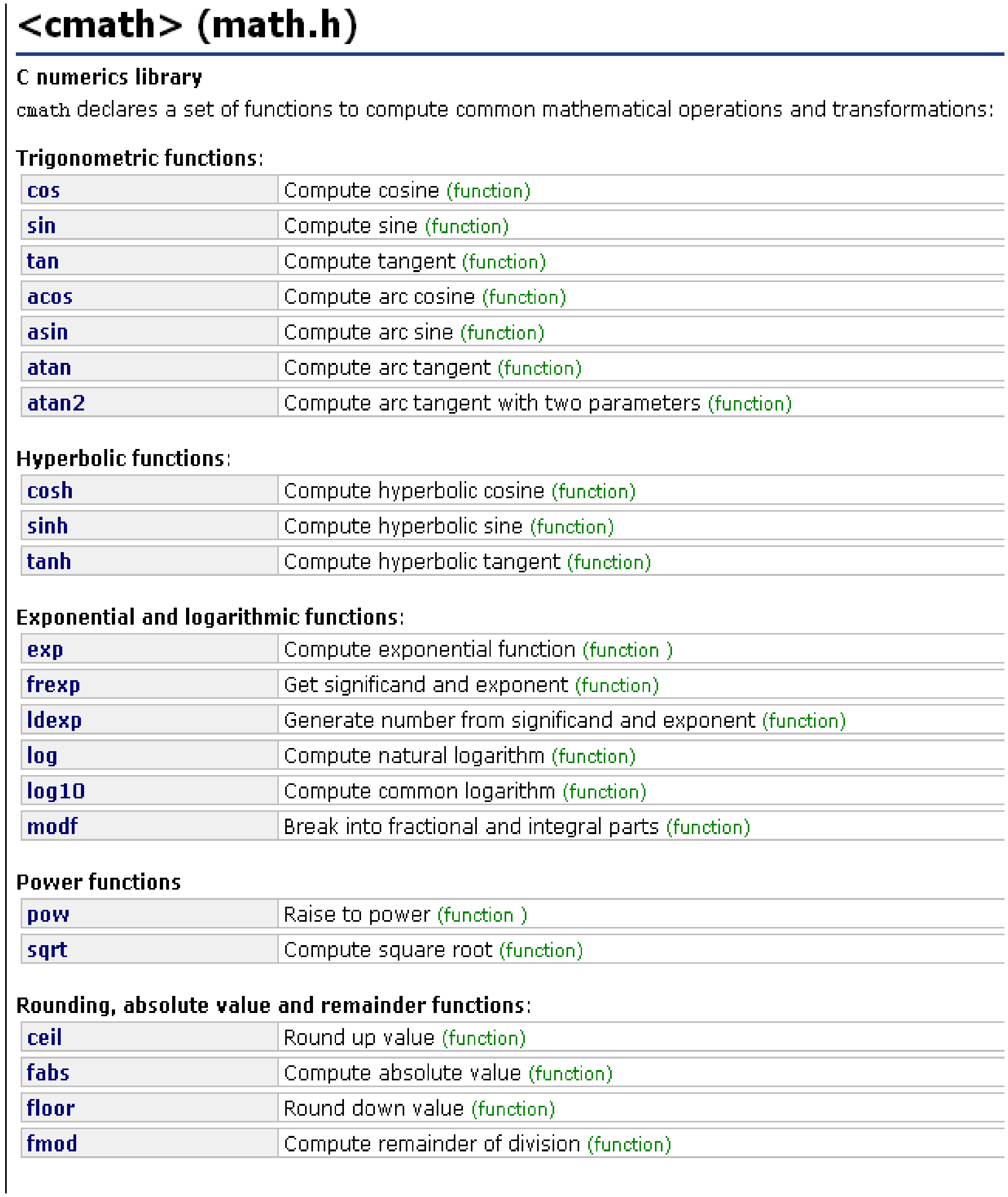
A function by definition, is a procedure or routine. In other words, it's a self-contained block of code that executes a certain task. Although some languages do distinguish between procedure and function, whereas a function returns a value of some kind and a procedure does not, C combines both functionalities into its definition. In order to use function in the program we need to establish 3 elements that are related to the function.

- **Function Declaration:** All identifiers in C need to be declared before they are used. This is true for functions as well as variables. For functions the declarations needs to be before the first call of the function.
- **Function Definition:** It is actual code for the function to perform the specific task.
- **Function Call:** In order to use function in the program we use function call to access the function. The program or function that calls the function is referred to as calling program or calling function.

Ex:

```
#include <stdio.h>

print_msg();      /* The function Declaration */

void main()   /* the main function */
{
print_msg();  /* The function call */
}

print_msg()   /* the function definition */
{

printf("this is a module called print_msg \n");

}
```

Math Library Functions



<cmath> (math.h)

C numerics library

`cmath` declares a set of functions to compute common mathematical operations and transformations:

Trigonometric functions:

cos	Compute cosine (function)
sin	Compute sine (function)
tan	Compute tangent (function)
acos	Compute arc cosine (function)
asin	Compute arc sine (function)
atan	Compute arc tangent (function)
atan2	Compute arc tangent with two parameters (function)

Hyperbolic functions:

cosh	Compute hyperbolic cosine (function)
sinh	Compute hyperbolic sine (function)
tanh	Compute hyperbolic tangent (function)

Exponential and logarithmic functions:

exp	Compute exponential function (function)
frexp	Get significand and exponent (function)
ldexp	Generate number from significand and exponent (function)
log	Compute natural logarithm (function)
log10	Compute common logarithm (function)
modf	Break into fractional and integral parts (function)

Power functions

pow	Raise to power (function)
sqrt	Compute square root (function)

Rounding, absolute value and remainder functions:

ceil	Round up value (function)
fabs	Compute absolute value (function)
floor	Round down value (function)
fmod	Compute remainder of division (function)

Programming Examples:

1. The LeVan Car Rental company charges $0.25/mile if the total mileage does not exceed 100. If the total mileage is over 100, the company charges $0.25/mile for the first 100 miles, then it charges $0.15/mile for any additional mileage over 100.

Write a program so that if the clerk enters the number of miles, the program would display the total price owed.

Source File

```
#include <iostream>
using namespace std;

void main()
{
   unsigned int Miles;
   const double LessThan100 = 0.25;
   const double MoreThan100 = 0.15;
   double PriceLessThan100, PriceMoreThan100, TotalPrice;
   cout << "Enter the number of miles: ";
   cin >> Miles;
   if(Miles <= 100)
   {
      PriceLessThan100 = Miles * LessThan100;
      PriceMoreThan100 = 0;
   }
   else
   {
      PriceLessThan100 = 100 * LessThan100;
      PriceMoreThan100 = (Miles - 100) * MoreThan100;
   }
   TotalPrice = PriceLessThan100 + PriceMoreThan100;
   cout << "\nTotal Price = $" << TotalPrice << "\n\n";
}
```

Here is an example of running the program:

```
  Enter the number of miles: 75
Total Price = $18.75
```

// - Show all odd numbers using a for loop

```
#include <iostream>
using namespace std;
int main()
{
         for (int count = 1; count <= 41; count += 2)
         {
                  cout << count << ", ";
         }
         cout << endl;
         return 0;
}
/*==================[output]==============================================
========
1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41,
Press any key to continue . . .
```

// - Count backwards using a for loop.

```
#include <iostream>
using namespace std;
int main()
{
	for (int count = 15; count > -1; count--)
	{
		cout << count << ", ";
	}
	cout << endl;
	return 0;
}
/*=================[output]============================
15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0,
Press any key to continue . . .
=========================================================*/
```

// - Count by fives using a for loop.

```
#include <iostream>
using namespace std;
int main()
{
	for (int count = 0; count <= 100; count += 5)
	{
		cout << count << ", ";
	}
	cout << endl;
	return 0;
}
/*=====================[output]=========================================
==
0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95, 100,
Press any key to continue . . .
```

```
// While loop in C++ with output show below.

#include <iostream>

using namespace std;
```

```
int main()
{
    int num = 1; // num variable controls the count from 1.
            // initialize counter starting at 1.
     while (num <= 7) // Loop - number controls how many times.
    {
        cout << num; // Show the number each time.
        num++;      // Add 1 to the variable num.
        cout << " - Hello world!\n";
    }
     cout << "\n";
    return 0;
}

/* ====================== output ==============================

1 - Hello world!
2 - Hello world!
3 - Hello world!
4 - Hello world!
5 - Hello world!
6 - Hello world!
7 - Hello world!


*/// ============================================================
```

SUM of series

```
#include<stdio.h>
int main()
{
 int n, sum,i,j;

 printf("Please enter an integer, n = ");
 scanf("%d", &n);

 for(i=1;i<=n;i++)
     for(j=1;j<=i;j++)
         sum = sum + n;
 printf("sum = %d", sum);

 return 0;
}
```

Factorial of a given number

Hey the program is as follows

```
 #include<iostream.h>
 void maun()
 {
 int fact=1,i,n;
 cout<<"enter the number ";
 cin>>"n;
 for(i=1;i<=n;i++)
 {
 fact=fact*i;
 }
 cout<<"the fact is "<<fact;
 }
```

To check whether number is prime or not

```
#include<stdio.h>
 main()
 {
   int n, c = 2;

   printf("Enter a number to check if it is prime\n");
```

```
    scanf("%d",&n);

    for ( c = 2 ; c <= n - 1 ; c++ )
    {
      if ( n%c == 0 )
      {
        printf("%d is not prime.\n", n);
        break;
      }
    }
    if ( c == n )
      printf("%d is prime.\n", n);

    return 0;
  }
```

Swapping two numbers

```
#include <stdio.h>

 int main()
 {
    int x, y, temp;

    printf("Enter the value of x and y\n");
    scanf("%d%d", &x, &y);

    printf("Before Swapping\nx = %d\ny = %d\n",x,y);

    temp = x;
    x = y;
    y = temp;

    printf("After Swapping\nx = %d\ny = %d\n",x,y);

    return 0;
 }
```

Leap Year Program

```
#include <stdio.h>

 int main()
 {
   int year;

   printf("Enter a year to check if it is a leap year\n");
```

```
  scanf("%d", &year);

  if ( year%400 == 0)
    printf("%d is a leap year.\n", year);
  else if ( year%100 == 0)
    printf("%d is not a leap year.\n", year);
  else if ( year%4 == 0 )
    printf("%d is a leap year.\n", year);
  else
    printf("%d is not a leap year.\n", year);

  return 0;
}
```

Program to check that character is vowel or not

```
#include <stdio.h>

 main()
 {
  char ch;

  printf("Enter a character\n");
  scanf("%c", &ch);

  switch(ch)
  {
    case 'a':
    case 'A':
    case 'e':
    case 'E':
    case 'i':
    case 'I':
    case 'o':
    case 'O':
    case 'u':
    case 'U':
      printf("%c is a vowel.\n", ch);
      break;
    default:
      printf("%c is not a vowel.\n", ch);
  }

  return 0;
}
```

C program to perform addition, subtraction, multiplication and division

```
#include <stdio.h>

 int main()
 {
   int first, second, add, subtract, multiply;
   float divide;

   printf("Enter two integers\n");
   scanf("%d%d", &first, &second);

   add      = first + second;
   subtract = first - second;
   multiply = first * second;
   divide   = first / (float)second;   //typecasting

   printf("Sum = %d\n",add);
   printf("Difference = %d\n",subtract);
   printf("Multiplication = %d\n",multiply);
   printf("Division = %.2f\n",divide);

   return 0;
 }
```

//program to shutdown a system

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

 main()
 {
   char ch;

   printf("Do you want to shutdown your computer now (y/n)\n");
   scanf("%c",&ch);

   if (ch == 'y' || ch == 'Y')
     system("C:\\WINDOWS\\System32\\shutdown -s");

   return 0;
 }
```