Chapter 1: Introduction & C Programming Fundamentals

In this chapter you will learn all about:

Chapter 1: C Programming Fundamentals

- General Introduction to C Programming
- C Program Identifiers and constants
- Fundamental data types, Declarations
- Standard data input and output functions
- Arithmetic operators
- Expressions and statements
- Annexure

==========

.

Chapter 1: C Programming Fundamentals

A. What is C?

- **A general purpose programming language –** *What does this mean?*

  Originally used as a systems programming language

- **Closely associated with UNIX and LINUX(** 94% written in C)

- **Convenient and effective for many tasks -Widely used in several industrial applications and in real-life engineering applications;**

- **Does not have many restrictions in using the language;**

Application Domains

- **Computer graphics, CAD/CAM**
- **Artificial intelligence and expert systems**
- **Communications and networks**
- **Real-time systems**
- **Database management systems**
- **Financial and accounting packages**
- **Languages: C, PHP, Prolog, Python**

The C standards:

Kernighan and Ritchie C

- "The C Programming Language", Prentice-Hall 1978
- "The C Programming Language", Second Edition, Prentice-Hall 1988

American National Standard

- For the C Programming Language
- American National Standards Institute (ANSI)
    * C language
    * standard library
    * compilation and execution environments

Portability

- It is the separation of the language and library that makes C portable
- When porting to another computer

    the source code can be recompiled with ease

- The library will be supplied by the compiler vendor

B. General Structure of a C program

In general, **all** C Programs follow this structure given below.

```
#include<stdio.h>  /* library that has I/O functions*/
#include<math.h> /* library that has math functions */
/* Above is called "header" in general.  */
< other declarations can come here >/* global variables */
void main (< arguments >) /* main is a function for all C
programs to begin with; main function may or may not have
arguments */
{  /*  start of the code in the main */
     < argument declarations >     /* argument types
     < declarations >         /* local variables */
     < statements >
     return; /* We will come this later. Stay tuned */
}  /* This bracket marks the end of the code */
```

Example 1.1

```
#include<stdio.h>
int num; /* <-What is this line? */
void main() /* need not have any arguments */
{
     int x;
     num = 6;  /* <-What does this mean? */
     x = num*5;  /* computation */
     printf("%d is my lucky number!\n", num);
     /* What does this prog do upon execution? */
     return;
} /* end of my c code */
```

Result of the above program after executing:

```
>> 6 is my lucky number!
```

C. Program Identifiers

Sequence of letters, digits and underscore:

- first character must be either a letter or an underscore

- some compilers permit use of other characters like the dollar sign ($)

Upper and lower case different

The first 31 characters are significant

- some compilers allow up to 128 characters

Example 1.2 (Identifiers)

```
seminar_room          _hdb_block24
i, j, k               _hdb_block25
end_of_file           MAXCHARS
End_of_file           2_good_2_be_true (illegal)
My name (illegal)     my-name (illegal)
```

D. Fundamental data types

Unlike Python, we have to explicitly declare what type of data we want to use, in advance, in C!

Integers: int [2 or 4 bytes – varies w.r.t machine]

- qualifiers: *short, long, signed, unsigned*

Characters: char [single character – 1 byte]

- qualifiers: *signed, unsigned*

Floating point: float [4 bytes or 1 word]

- **double** [8 bytes or 2 words for large size floating point numbers]
- **qualifier: *long***

Others: enum, void, const, volatile ***(****we will revisit this later when required****)***

D.1 Data types - A quick start!

(Start making your hands dirty from now on…)

Integers: (Example – Say, 4 bytes per integer)

- **int** x; /* declares x as an integer */

- int x, Y, y, z; /* declare more than one variable */
- int age, Age, Num_of_books;
- **long int** distance_to_Neptune;
- Examples of integers: 25, -12, 1025, 20040, 8000000

Floating Point Numbers: (Example – Say, 4 bytes per float)

- **float** Q; /* declares x as a float */
- float Q, q, P; /* declare more than one variable */
- float Volume, Cost_of_a_book;
- **long float** radius_of_an_atom;
- Examples of Float: 25.032, -12.1, 0.001025, 200.40, 0.000002123;

Remarks: A quantity like 5.026 x 10-17 is represented as: 5.026E-17; 0.5026e-16; 50.26e-18; 0.0005026E-13; Note that you can use E or e for base 10 numbers; So, 1.2 x 10-3 can be written as: 1.2E-3 or 1.2e-3

Character variables: (1 byte per character)

- **char** A; /* declares x as a character variable */
- char A, my_initials, her_initials; /* declare more than one variable */
- Examples of Char: ‘a’, ‘b’, ‘A’, ‘Z’, ‘<’, ‘&’, ‘%’

Array data structure:

- **int** myarray[10], herarray[];
- int data[5] = {0,1,2,3} /* first 4 elements will be initialized */

Note: Array index starts from 0 /* *same as in python* */

D.2 What about strings?

***Rule* - Enclose a string of characters within double-quotes:**

Example 1.3:

`"Jennifer O'Neil"` "John's account number is: "

char myname[10] = "Jessica"; /* Max of 10 characters can be stored */

char myname[10] = "Jessica Michael"

char mytext[] = "air conditioning facility"; /* any length string can be stored */

/* In the last case, '\0' will be appended automatically – null character denoting the end of the String. *We will revisit this soon to learn an important concept!* */

Character Set comprises the following.

- **Alphabetic characters (letters):** A-Z, a-z
- **Numerical characters (digits):** 0-9
- **Graphic characters:**

& ampersand	, comma	. period or dot
' apostrophe	" double quote	+ plus
* asterisk	= equal	? question
\ backslash	! exclamation	; semicolon
{ brace left	> greater than	/ slash
} brace right	- hyphen or minus	~ tilde
[bracket left	< less than	# number sign
] bracket right	(parenthesis left	_ underscore
^ circumflex	) parenthesis right	| vertical bar
: colon	% percent	

- **Non-graphic characters** *(list not complete…)*

\n	newline	\f	form feed
\r	carriage return	\\	backslash
\t	horizontal tab	\v	vertical tab

\b	backspace	\a	bell (alert)
\’	single quote	\”	double quotes
\?	question mark		
\xhh	hex number	\ooo	octal number

D.3 Standard Keywords - Reserved program words (*Refer to C manual for a complete list):*

auto while continue if for printf break else double int float void char const struct sizeof union switch **…**

E. Data Input and Output

- **Single character input** – *getchar()* function

```
char c; /* declares that c is a character type variable */
<……>
c = getchar();  /* single character through keyboard is assigned to c */
```

- **Single Character Output** – *putchar()* function

```
char h; /* declares that h is a character type variable */
<……>
putchar(h);  /* current value of h is transmitted to the standard display device */
```

- **Entering Input Data** – *scanf()* function – (*This is an example of a function that takes arbitrary number of arguments*)

scanf("format", &arg1, &arg2, ...)

- **reads** arg1, arg2, etc values from the standard input (stdin)
- format string specifies type and nature of values
- blanks, tabs or newlines serve as delimiters
- common conversion chars:
  - % d decimal integer, % o octal integer,
  - % x hexadecimal integer, % c single character
  - % s character string, % f floating point number
  - % e exponential form
  - d, o, x, e, and f may be preceded by 1
  - * is used as suppression character *(used for skipping input field)*

Note: Conflict occurs if next input character is not matched

Example 1.4: (Formatted data input – Declarations)

```
int       i;
float     x;
char      lady_name[30];
```

Input line

```
10     256.87556       Jennifer
```

Invoking scanf

```
scanf("%3d %5f %s", &i, &x, lady_name);
Result:  i = 10     x = 256.8 lady_name = "Jennifer"

scanf("%d %*f %5s", &i, lady_name);
Result:  i = 10 lady_name = "Jenni"
```

Q: What is the effect of

```
scanf("%d %f %s", &i, x, lady_name);
```

?

- **Writing Output Data** – *printf()* function - (*This is an example of a function that takes arbitrary number of arguments* **)**

printf("format", arg1, arg2, ...)

- writes arg1, arg2, etc to the standard output
- format string specifies type and nature of values
- conversion chars – Same formatting rules apply as scanf()

Example 1.5: (Formatted data input – Declarations)

```
float x = 123.456789;
int i = 9876;
char lady_desc[10] = "pretty";
```

Invoking printf (*I am using* **b** *to indicate a blank space below)*

`printf("%f\n", x);`	`123.456789`	*left justified*
`printf("%10.2f\n", x);`	**bbbb**`123.46`	*right justified*
`printf("%10.3e\n", x);`	**b**`1.235e+02`	
`printf("%d\n", i);`	`9876`	*left justified*
`printf("%7d\n", i);`	**bbb**`9876`	*right justified*

What is the result of

`printf("Janet is a %s girl\n", lady_desc);` **?**

Example 1.6: Play with this simple code to understand the printing formats. Vary all the data types, values, range of values, and observe the printing formats.

```
#include<stdio.h>
void main() {
int x;
float y, Area;
x = 2;
y = 0.5467;
Area = x*y;
printf("The value of x is: %d\n", x);  /* Int */
printf("The value of y is: %.6f\n",y); /* Float */
printf("Area is: %.6f\n",Area);  /* formatted float  */
printf("Area (in exp) is: %1.3e\n",Area);  /* expo */
return;
}
```

Tutorial Problem 1.1: Consider the inputs, double x = 5000, y = 0.0025. **Print** the respective outputs if you use: %e for printing x, y, x*y and x/y.

Tutorial Problem 1.2: Consider the input float x = 123.456. **Print** the respective outputs if you use %7.1f, %12.5e, and %12.3e, for printing x.

E.1. Input-Output between Computer & Standard input/output devices – gets() and puts() functions

==========

```
#include <stdio.h>
main() { /* code to read and write a line of text */
char name[80]; /* declares that c is a character type variable */
gets(name); /* works like scanf */
puts(name); /* works like printf */
}
```

E.2 Measuring the size of the data types

As the sizes of the data types used to store in computer's memory may differ depending on the type of CPU (32-bit, 64-bit, 128-bit, etc), it is important to get to know the sizes before you use the platform. *Why is this important?*

In C we can dynamically allocate memory (during run time) to store any amount of data in the memory. Unless we know the sizes of the data we are storing in, CPU cannot allocate the required space! We can get to know the sizes using sizeof() library function.

Run through this tutorial first!

Tutorial 1.3: Print the sizes of different data types – int, char, float, double, long int, an integer array.

F. Simple/Regular C Statements

- an instruction terminated with semicolon
- most statements are expression statements

  < expression > **;**

Example 1.7:

- Assignment operator: = /* right to left assignment */
- **Arithmetic operators:** **+, -, *, /, %**
- **int** x,y,z,q; /* declares x,y,z,q as integers */
- x = y+ z; /* adds y and z and assigns the result to x */
- q = x / y; /* x divided by y and the result is assigned to q */

Compound statement

- block of statements
  * optional declarations
  * followed by optional statements
  * enclosed in parentheses { }

Example 1.8: (Compound statement)

```
{
        x = i+t;
        q = 2*x – 3*t + 45;
}
```

Compound statements appear at a variety of instances. These include, writing decision-making logic, writing control loops, nested loops, etc.

G. Arithmetic operators

Basic binary operators

+ plus	/ divide
- minus	% modulus (Mod operator)
* multiply	

Unary operators

- unary minus	! logical negation
+ unary plus	

- do not confuse the unary and the binary minus (plus)
- they are interpreted differently in terms of their ***precedence***

Tutorial Problem 1.4: Write the following arithmetic expressions in C

(a) $Y = ax^2 + bx + c$ / * try using library functions wherever possible */

(b) $R = (ax^2 + bx + c)$.Q divided by $(Mx + Nz)$

(c) Multiply the expression $(x^2 + mx)$ by an integer 25. Assign it to a float variable T. Then, subtract T from Y.

(d) Write the expression for the roots of the quadratic equation $ax^2 + bx + c = 0$

H. Relational operators

Relational operators

>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
= =	comparison equal to *do not be confused with a single = assignment*
!=	not equal to

Tutorial Problem 1.5: Let int i =1,j=2, k=3; float f = 5.5; char c= 'w'; Interpret the following logical expressions:

Expression	Interpretation (True/False)	Value (1=True / 0=False)
i<j		
(i+j) >= k		
(j+k) > (i+5)		
k!= 3		
j == 2		
c = 119		
Let i=7; Then, c >=10*(i+f)		
Let i=7; Then, (i+f) <= 10		

Note: Decimal value of w=119 and '0' = 48 are taken from the ASCII table

I. Logical operators

Connectives

&& logical **AND** || logical **OR**

Negation

! **negate**

Tutorial Problem 1.6: Let int i = 7; float f=5.5; char c = 'w'; Interpret the following logical expressions:

Expression	Interpretation (True/False)	Value
((f>4) && (c <= 'w'))		
((i*3 <=20) && (f+2.5 > i))		
((c*3 <=i*2) || (f > 0))		

I.1. Other unary operators

++	incremental
--	decremental
cast	(datatype) expression

i++; (*What is this?*) ++***i***; *(What is this?)*

(--j)* price; (*What is this?)*

(i++) * price; (float) i;

Tutorial 1.7: (Unary Operators) ***Pay attention to an example program presented in the lecture.***

Important Note: We will introduce BITWISE OPERATORS in the last chapter when we attempt to understand Microprocessor fundamentals.

STAY TUNED!

J. Combining assignment operator with Arithmetic operators – *cryptic writing!*

General form:

exp1 op= exp2

where **op** can be any of: + - * / % << >> & ^ |

Equivalent to: **exp1 = exp1 op (exp2)**

Tutorial Problem 1.8: int x = 1, y = 20; int marilyn = 4; **Evaluate**:

(i) x += 2; Equivalent expression: _____ Value of x = _____

(ii) x *= (y / marilyn); Eq. Exp: _____ Value of x = _____

K.1. Conditional operator

General form:

exp1 ? exp2 : exp3

Equivalent to:

```
if ( exp1 ) {
        ( exp2 )
}
else {
        ( exp3 )
}
```

Tutorial Problem 1.9: **Interpret** the following expressions.

(i) (c = getchar()) != EOF && c != '\n'

(ii) **if** ((c >= 10 && c < 15) || (c < -10 && c >= -15)) {

printf("Range of c is within the target\n");

(iii) Write a C conditional expression to check if user has input an alphabetical character using the logical operators.

L. Symbolic Constants

#define *NAME* **text**

"NAME" represents a symbolic name (*usually written in uppercase letters*) and "text" represents sequence of characters associated with the symbolic name; **Note** that there is **NO SEMICOLON** at the end of this definition!

This is part of what is called as a "C Pre-processor" component. We will revisit this concept when we see functions.

Example 1.9:

Observe the following definitions

#define TAXRATE 0.23

#define TRUE 1

#define FALSE 0

#define PI 3.141593

#define FRIEND 'Susan'

. . . . . .

Tutorial Problem 1.10: Demonstrate an example on how you use the above symbolic constants declared above.

Recall! Note on Flow-Charts

Do-it-Yourself - You have seen how to draw flow-charts to capture your logic in Lecture #1. Draw a flow-chart to determine the roots of a given quadratic equation. Assume A, B, and C are real positive values such that the *discriminant function* returns only positive values. You can make your design in such a way that user can repeat the computation as long as he/she wants.

M. Standard Library

Standard input/output;	Character processing;	String processing
Mathematical functions;	Storage allocation;	Error handling

- Application Libraries

  Btree indexing; Database management; Graphics
  Screen management; Communications; Multi-tasking
  Scientific functions;
  Each can be purchased with source code from vendors

N. Commenting your code

- Comments in Code can significantly help in many ways – tracking your logic, tracking where all a variable gets affected
- Comments before a complex logic may reduce errors and provide warning to the user/programmer
- Comments before a function can help in identifying what the function is meant for and its limitations; Sometimes it is better to tell what one cannot expect from a function;
- At any place, comments can be placed to warn / caution a programmer about program behaviour if certain inputs are presented to the program;
- If a variable has any maximum value or a minimum value it can be told to the user at the place where it is declared;
- Comments can be provided to explain a complex logic captured in a cryptic statement to facilitate a quick understanding by the user;
- Facilitating code sharing and in handling third party codes
- Facilitates Group Projects, especially in integration phase

ANNEX

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

To get the exact size of a type or a variable on a particular platform, you can use the **sizeof** operator. The expressions *sizeof(type)* yields the storage size of the object or type in bytes. Following is an example to get the size of int type on any machine:

Type	Storage size	Value range	Precision
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

1. About Constants

General rules – no commas & blank spaces within constants; a constant can be preceded by a minus sign, value of a constant can't exceed its minimum and maximum bounds (compiler dependency)

I.1 Integer point constants, floating point constants, character constants, string constants

Integer Point Constants – Three different number systems – decimal (base 10), octal (base 8), hexadecimal (base 16);

Base 10 - 0 through 9 allowed **with first digit must not be 0;**

Base 8 – 0 through 7: first digit **must begin with 0** to identify it as an octal #;

Base 16 – 0 through 9, A – F; **must begin with either 0x or 0X** to identify the number as a hexadecimal number;

==========

Example: 32768 (OK) 32, 768 (Not OK) **0**325 (Not OK)

32.5 (Not OK for integer) 07582 (Not OK for both *decimal and octal* – why?)

0745.76 (Not OK for *decimal*); 0xabd (OK)

0BE20 (Not OK)

==========

II. Operator precedence & complex expressions

( )	left to right (association)
* /	left to right
+ -	left to right
=	right to left (right exp is assigned to the left)

Among the arithmetic operators ***, / and %** fall into one category (group 1) and **+, -** fall into other category (group 2). Group 1 has higher precedence than Group 2.

Order in which consecutive operations are carried out is important – follows ***associativity*** property. Within each of the groups, associativity is from left to right; Always complete an expression within parentheses before you consider parameters outside parentheses. Parentheses have the highest precedence in a single expression;

Expression: **a – b/c * d** = a-[(b/c) x d] (Check: 10-10/2*2 = **?**)

Division is carried out first since this has higher precedence than subtraction; The result is then multiplied by **d** due to left to right associativity property; The product is then subtracted from **a**.

Expressions:

- combination of operators and operands
- every expression has a value

(***very important, will be stressed throughout the course***)

Example:

a * (b + c / d) / 20	(arithmetic)
(q > (x*y))	(relational)

!found (logical)

Other expression *(later in the course, if time permits)*

- shift, bit manipulation; - conditional, cast, pointer

Do-it-Yourself - Observe the following expressions using operators; Interpret them with the following values: p=3, i=7, seconds = 3200

(i) Q = p * (i + 1); (ii) time = seconds / (60.0 * 60 * 24 * 365);

(iii) x = 40 %19 - number; (iv) p*2 / seconds + i;

***DIY*:** Run the following code and observe the result.

```
#include<stdio.h>
main() {
        float y;
        y = - 2+5*6 + 4 + 3*2+3/5;
        printf("Result: %2.2f\n",y);
}
```