

SUBJECT TITLE AND CODE : INTRODUCTION TO C PROGRAMMING – 1BPLC105E

MODULE 1

Art of Programming through Algorithms & Flowcharts

Algorithm:

An Algorithm is a finite set of unambiguous instructions which, when executed, performs a task correctly. There are three characteristic features in this description of an algorithm. They are:

- The number of tasks required to perform the tasks should be finite.
- Each of the instruction in the algorithm should be unambiguous in nature, meaning on the execution of each such instruction, the outcome should be definite and predictable
- Finally, the algorithm should solve the problem correctly. In other words, on the execution of the algorithm, the objective should be fulfilled There are no defined rules to write an algorithm.

EXAMPLE 1 :Develop an algorithm to find the average of three numbers taken as input from the user.

Algorithm AVERAGE

1. Step 0 START
2. Step 1 INPUT first number into variable A
3. Step 2 INPUT second number into variable B
4. Step 3 INPUT third number into variable C
5. Step 4 COMPUTE SUM A B C
6. Step 5 COMPUTE AVG SUM/3
7. Step 6 DISPLAY AVG
8. Step 7 END

Steps 1, 2 and 3 are statements that will enable the input of three different numbers. Step 6 is used to display the result. Steps 4 and 5 are of particular interest to us as these are the two imperative statements which instruct/command to perform the computation. Thus, imperative statements are the ones which command the processor to perform a task.

Algorithms – Conditional constructs

Algorithm MAXIMUM

```
Step 0  START
Step 1  INPUT A
Step 2  INPUT B
Step 3  IF A > B THEN
Step 4           MAX = A
Step 5           ELSE
Step 6           MAX = B
Step 7  END IF
Step 8  DISPLAY MAX
Step 9  END
```

Problem: Develop an algorithm to find the maximum of 2 user input numbers

EXAMPLE 3

Algorithms – Conditional constructs

Algorithm DIVISION

```
Step 0  START
Step 1  INPUT A
Step 2  INPUT B
Step 3  IF B != 0 THEN
Step 4           Q = A / B
Step 5           DISPLAY Q
Step 6  END IF
Step 7  END
```

Problem: Develop an algorithm to find the quotient when a user input number is divided by another

Flowchart

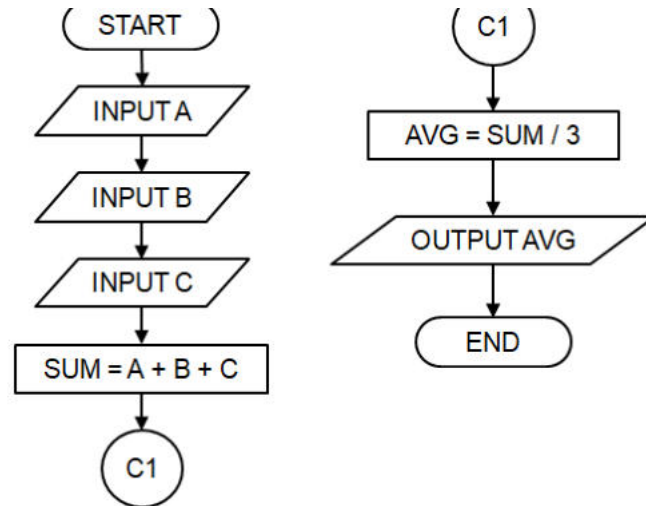
A **flowchart** is a **graphical representation of an algorithm** that uses standardized symbols to show the **sequence of operations**, **flow of control**, and **logic** of a program or process in a clear and visual manner.

Just like an algorithm, a **flowchart** is also a tool used to represent the logic of a program or process. As the name suggests, a flowchart helps us **visually understand the flow of control** through a sequence of statements. Many times, understanding logic becomes easier when it is shown in a **pictorial form**, and a flowchart is very helpful in such situations.

A flowchart uses **standard symbols** to represent different operations, decisions, and flow of control. These symbols provide a clear and uniform way to illustrate the logic.

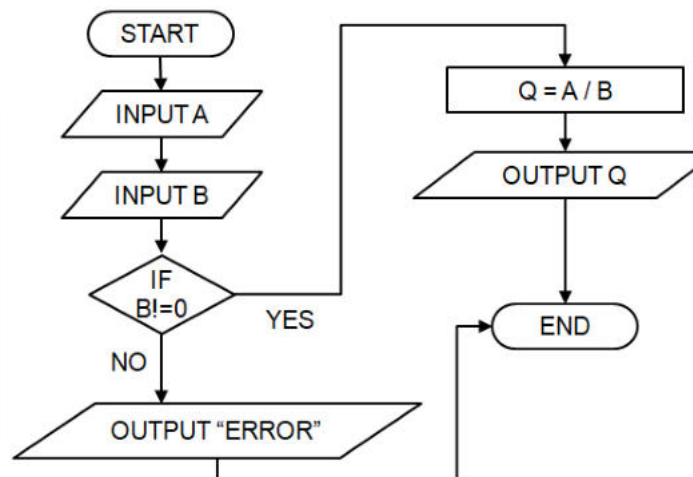
Algorithm AVERAGE

Step 0 START
Step 1 INPUT A
Step 2 INPUT B
Step 3 INPUT C
Step 4 $SUM = A + B + C$
Step 5 $AVG = SUM / 3$
Step 6 DISPLAY AVG
Step 7 END



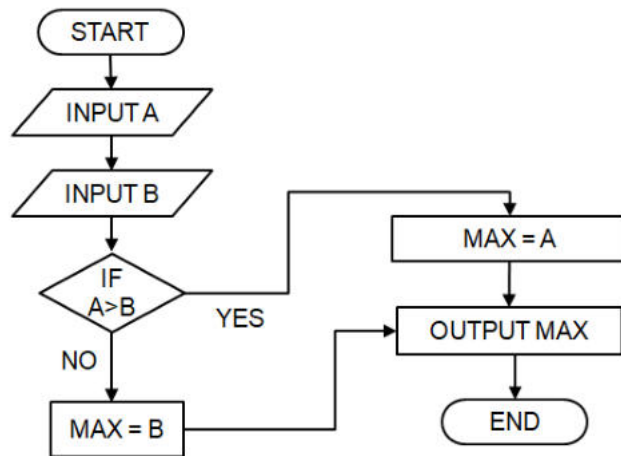
Algorithm DIVISION

Step 0 START
Step 1 INPUT A
Step 2 INPUT B
Step 3 IF B $\neq$ 0 THEN
Step 4 $Q = A / B$
Step 5 DISPLAY Q
Step 6 ELSE
Step 7 DISPLAY "ERROR"
Step 8 END IF
Step 9 END



Algorithm MAXIMUM

```
Step 0  START
Step 1  INPUT A
Step 2  INPUT B
Step 3  IF A > B THEN
Step 4      MAX = A
Step 5      ELSE
Step 6      MAX = B
Step 7  END IF
Step 8  DISPLAY MAX
Step 9  END
```



OVERVIEW OF C

HISTORY OF 'C'

- **ALGOL (Early 1960s)**

The root of modern programming languages is **ALGOL**, introduced in the early 1960s. It was the first computer language to use **block structure** and introduced the concept of **structured programming**.

- **BCPL (1967)**

In **1967**, *Martin Richards* developed **BCPL (Basic Combined Programming Language)**. It influenced the development of later system programming languages.

- **B Language (1970)**

In **1970**, *Ken Thompson* created a language called **B**, which was used to develop the early versions of the **UNIX operating system**.

- **C Language (1972)**

In **1972**, *Dennis Ritchie* introduced a new language called **C** at Bell Labs. It became one of the most widely used system programming languages and the successor of **B**.

Traditional C

C language was developed in the early **1970s**. Over the next decade, it grew and evolved significantly. Before standardization, the main reference for the language was the first edition of "*The C Programming Language*" by **Kernighan and Ritchie**, published in **1978**.

This original version of C is known as "**Classic C**" or "**K&R C**".

Standard C

C was standardized in **1989** by **ANSI (American National Standards Institute)**, and this standardized version is known as **ANSI C** or **C89**.

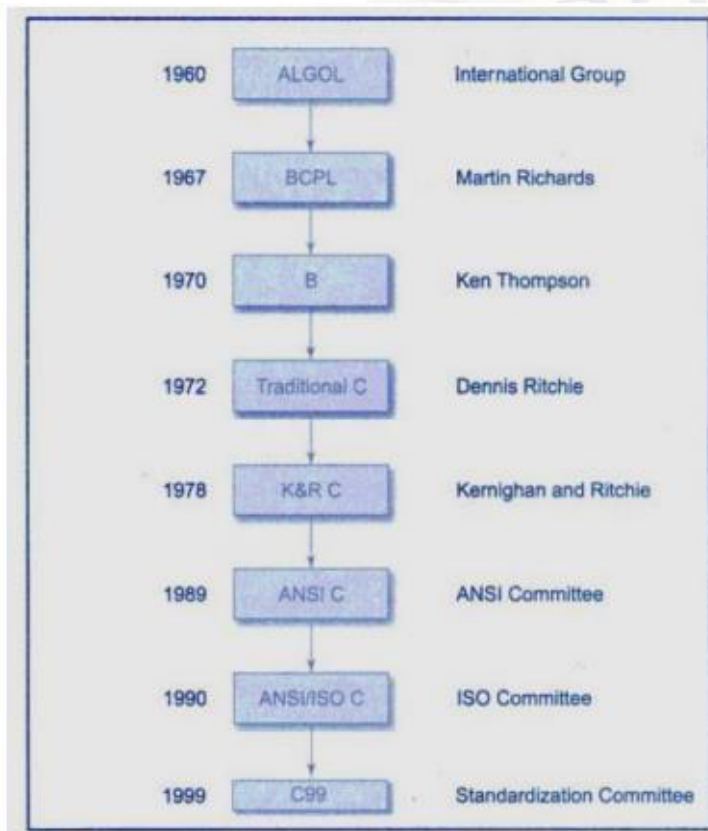
In **1990**, this standard was adopted by **ISO**, making it an **International Standard**.

As part of its evolution, the language standard was updated in:

- **1995 – C95**
- **1999 – C99**

During the **1990s**, **C++** extended C by adding support for **Object-Oriented Programming (OOP)** and other features useful for large-scale software development.

In the same period, **Sun Microsystems** developed **Java**, a new language inspired by both **C** and **C++**.



Features of 'C'

- **Robust Language**

C is a robust language with a rich collection of built-in functions and operators, which allow programmers to write even very complex programs efficiently.

- **Efficient and Fast**

Programs written in C are highly efficient and fast due to its variety of data types and powerful operators.

C is often much faster than languages like BASIC.

Although ANSI C has only **32 keywords**, its strength lies in its **built-in library functions**, which support the development of a wide range of applications.

- **Highly Portable**

C is a highly portable language. A program written on one computer system can run on another system with little or no modification.

This portability is useful when moving to new hardware or operating systems.

- **Well Suited for Structured Programming**

C supports structured programming, allowing users to break a problem into **functions (modules/blocks)**.

This modular structure makes **debugging, testing, and maintenance** easier.

- **Ability to Extend Itself**

A C program is a collection of functions supported by the C library.

Programmers can **create and add their own functions** to the existing library.

With a large number of reusable functions, the programming task becomes simpler and more organized.

BASIC STRUCTURES OF C PROGRAMS

Documentation Section: It has set of comment lines giving the name of program, author details.

Link Section: It provides instructions to the compiler to link function from the system library. # include Directive To access the functions which are stored in the library, it is necessary to tell the compiler, about the file to be accessed.

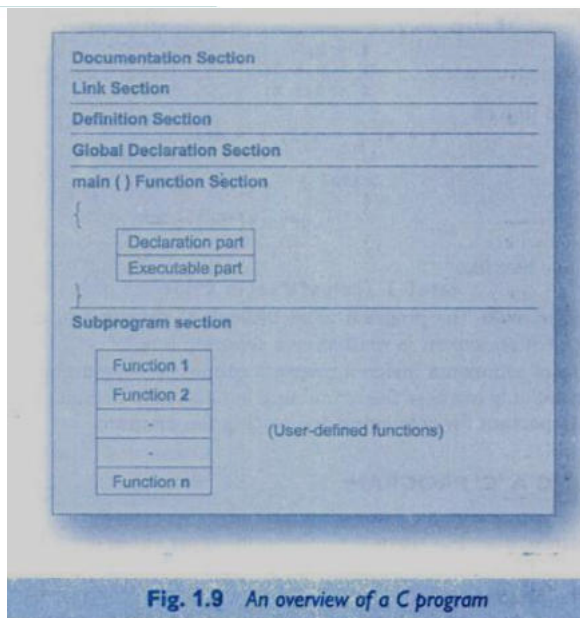


Fig. 1.9 An overview of a C program

Syntax: `#include<stdio.h>` . The first line of the program `#include <stdio.h>` is a preprocessor command, which tells a C compiler to include `stdio.h` file before going to actual compilation.

Definition Section: It defines all symbolic constants.

1. **#define** instruction defines value to a symbolic constant.
2. It is a preprocessor compiler directive, not a statement.
3. Therefore it should not end with a semicolon.
4. Generally written in uppercase.

Global Declaration Section: Some variables that are used in more than one function, such variables (global variables) declared in the global declaration section that is outside of all the functions. It also declares all the user-defined function.

Main() function section: Every 'C' program must have one `main()` function section. It contains two parts

1. **Declaration part:** It declares all variables used in the executable part.
2. **Executable part:** It has at least one statement.
3. Every C program should have a pair of curly braces (`{, }`). The left braces indicates the beginning of the `main ( )` function and the right braces indicates the end of the `main ( )` function. All statements in the declaration and executable parts end with a semicolon (`;`).

Sub Program Section: A subprogram is a function and these functions are written by the user

are called user defined functions. These functions are performed by user specific tasks and this also contains set of program statements. They may be written after a main () function and called within main () function.

Example Program

```
/* print hello*/  
#include<stdio.h> /* link section*/  
void main()          /* execution of program*/  
{  
    print("hello world"); /*executable statement*/  
}
```

Output

```
hello world  
*****
```

How to run (executing) a 'C' program

Executing a program written in C involves a series of steps. These are

1. Creating the program
2. Compiling the program
3. Linking the program with functions that are needed from the C library
4. Executing the program

1. Creating the Program

- A C program must be written and stored in a **file**.
- The file name can contain letters, digits, and special characters, but it must end with the extension **.c**
Example: hello.c
- The program is created and edited using a **text editor**.
- After editing, the file is **saved on disk**.
- This file containing the C program is called the **source program** or **source code**.

2. Compiling and Linking

- The source program must be **translated** into a form that the computer can execute.
- This translation is done by a **compiler**.
- The compiler checks each instruction for **syntax and semantic errors**.

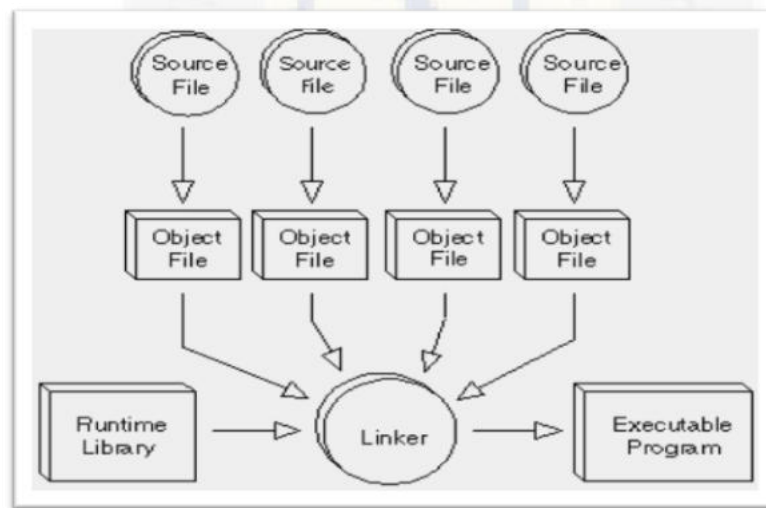
- If no errors are found, the compiler produces an **object file**, usually named:
`hello.o` → *object code*
- A **linker** then combines:
 - the object file(s)
 - required library functions
 to produce an **executable program**.
- The final executable file is normally named:
`a.out`

If errors are found during compilation, they are listed.

You must correct them in the source code using the editor and compile again.

3. Executing the Program

- The executable program is run using the command:
`a.out`
- This loads the program into memory and executes it.
- During execution, the program may ask for **input from the keyboard**



CONSTANTS, VARIABLES AND DATA TYPES

The **character set** refers to all the valid characters that can be used to write programs in C. It includes the following categories:

1. Letters

- **Uppercase:** A – Z
- **Lowercase:** a – z

2. Digits

- All decimal digits: 0 – 9

3. Special Characters

These include symbols such as:

. : ; ? " , / ~ ` ! @ # \$ % ^ & * () { } [] -

4. White Spaces

White spaces are used to separate words and improve readability.

They include:

1. Blank space
2. Horizontal tab
3. Carriage return
4. Newline
5. Form feed

Trigraph characters

Many non-English keyboards do not support all the characters. For that C introduces the concept of “trigraph” sequences to provide a way to enter certain characters that are not available on some keyboards. Each trigraph sequence consists of three characters (two question mark followed by another character).

Table 2.2 ANSI C Trigraph Sequences	
Trigraph sequence	Translation
??=	# number sign
??(	[left bracket
??)	] right bracket
??<	{ left brace
??>	} right brace
??!	vertical bar
??/	\ back slash
??^	^ caret
??~	~ tilde

C Tokens

In a passage of text, individual words and punctuation marks are called *tokens*. Similarly, in C programming, the smallest individual units of a program are known as **C tokens**.

The different types of C tokens are:

- **Keywords**
Examples: `int`, `while`
- **Identifiers**
Examples: `main`, `total`
- **Constants**
Examples: `10`, `20`
- **Strings**
Examples: `"total"`, `"hello"`
- **Special Symbols**
Examples: `()`, `{}`
- **Operators**
Examples: `+`, `/`, `-`, `*`

KEYWORDS AND IDENTIFIERS

- Keywords have **fixed, predefined meanings**.
- Their meanings **cannot be changed** by the programmer.
- They are reserved by the compiler for special purposes.
- Keywords form the basic building blocks of a C program.

Table 2.3 ANSI C Keywords

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

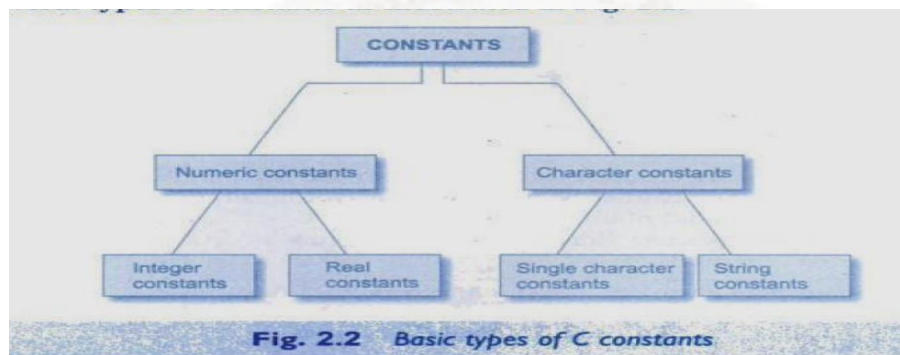
Identifiers refer to the names of variable, functions and arrays. These are user defined names and consist of sequence of letters and digits. Both uppercase and lowercase letters are permitted. But uppercase and lowercase letters are not having same meaning e.g. RAM & ram are not same. The underscore character (_) can also be included and is considered to be a letter.

Rules

1. First character must be alphabet.
2. Consist of letters, digits, underscore
3. Only first 31 characters are significant
4. Cannot use a keyword
5. Must not contain white space

Constants

Constant in C refers to fixed values that do not change during the execution of a program. C supports four types of constants.



Integer constant

An **integer constant** is a sequence of digits without a fractional part.

There are **three types** of integer constants in C:

1. Decimal Integer Constants

- Consist of digits **0–9**
- May have an optional **+** or **–** sign
- Must *not* contain spaces, commas, or special symbols

Valid examples:

123, -87, 0, +56

Invalid examples:

\$1000, 15 750, (20, 000)

2. Octal Integer Constants

- Consist of digits **0–7**
- Must begin with a leading **0**

Examples:

075, 051

3. Hexadecimal Integer Constants

- Must begin with **0x** or **0X**
- Followed by digits **0–9** and letters **A–F** or **a–f**
(A–F represent values 10–15)

Examples:

0x2, 0x3A, 0Xe5

Integer Size Limits

- Maximum value depends on the machine:
 - **16-bit machine:** 32,767
 - **32-bit machine:** 2,147,483,647
- Larger values can be represented using suffixes:
 - U → unsigned
 - L → long
 - UL → unsigned long

Examples:

56789U (unsigned integer)

876543217UL (unsigned long integer)

9765467L (long integer)

Real Constants (Floating-Point Constants)

Real constants include a **fractional part**, used to represent quantities that vary continuously.

Examples:

0.0026, -0.97, 435.29, +487.0

Exponential (Scientific) Notation

A real constant may also be written in exponential form:

General format:
mantissa e exponent

- **Mantissa:** integer or real number
- **Exponent:** integer (with optional + or – sign)
- e or E represents “× 10 to the power of”

Examples:

0.65e4

12e–2

–1.2e–1

Single Character Constants

- A **single character constant** represents **one character**.
- It is enclosed in **single quotation marks** ' '.

Examples:

'5', 'x', ';', ' ' (blank space)

ASCII Values

- Every character has a unique **ASCII (American Standard Code for Information Interchange)** value.
- Example:

```
printf("%d", 'a'); // Output: 97
```

Here, 97 is the ASCII value of 'a'.

```
printf("%c", 97); // Output: a
```

Here, 97 is printed as the character 'a'.

2. String Constants

- A **string constant** is a sequence of characters enclosed in **double quotation marks** " ".
- It may include letters, digits, special characters, or spaces.

Examples:

"hello!"

"1986"

"Hi well done"

"a" (Note: This is still a string because it uses " ")

3. Backslash Character Constants (Escape Sequences)

- Backslash character constants are used in output functions.
- They start with a **backslash** \ and represent **special characters**.
- Although written using two characters, they represent **one single character** in memory.

These combinations are called **escape sequences**.

Constant	Meaning
'\a'	audible alert (bell)
'\b'	back space
'\f'	form feed
'\n'	new line
'\r'	carriage return
'\t'	horizontal tab
'\v'	vertical tab
'\''	single quote
'\"'	double quote
'\?'	question mark
'\\'	backslash
'\0'	null

Variables

A **variable** is a data name used to store a value.

Each variable has a unique **memory location** where its value is stored.

1. Every variable has a name, a type, a size and a value
2. Whenever a new value is placed into a variable, it replaces (and destroys) the previous value Reading variables from memory does not change them Rules
3. They must begin with a letter
4. ANSI standard recognizes a length of 31 characters. However, the length should not be normally more than eight characters, since only first eight characters are treated as significant by many compilers.
5. The uppercase & lowercase letters are different e.g., Marry, MARRY, marry are different variables names.
6. Variable name should not be a keyword.

7. No special characters, such as period, comma, semicolon, blanks are permitted in variable names.

Table 2.6 Examples of Variable Names

Variable name	Valid?	Remark
First_tag	Valid	
char	Not valid	char is a keyword
Price\$	Not valid	Dollar sign is illegal
group one	Not valid	Blank space is not permitted
average_number	Valid	First eight characters are significant
int_type	Valid	Keyword may be part of a name

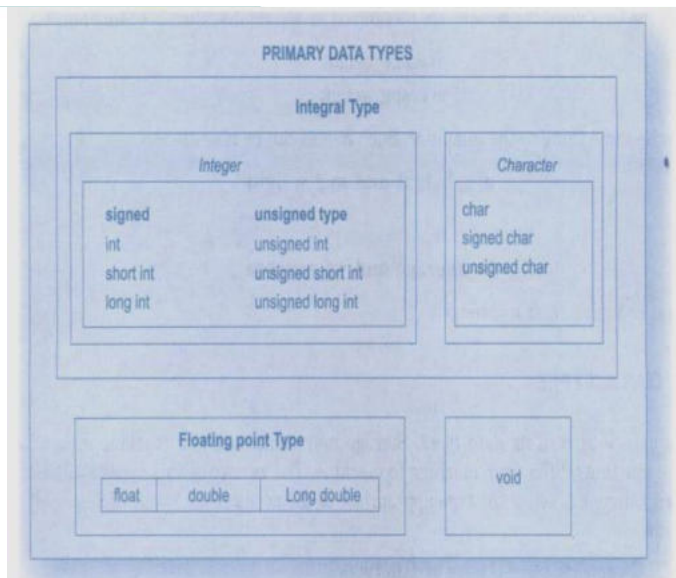
Data Types

Data types in C define the type of data a variable can store.

They specify the storage size, the range of values that can be stored, and the operations that can be performed on the data. C language data types can be broadly classified as

- a) Primary data type
- b) Derived data type
- c) User-defined data type

Basically C supports five Primary data types :-integer (int) , character (char), floating point (float) , double precision floating point (double) and void.



PRIMARY DATA TYPES		
TYPE	SIZE	RANGE
Char	1 byte	-128 to 127
Int	2 byte	-32,768 to 32,767
Float	4 byte	3.4e-38 to 3.4e+38
Double	8 byte	1.7e-308 to 1.7e+308
Long int	4 byte	-2^{32} to 2^{32}
Long double	10 byte	$\pm 3.4 \times 10^{4932}$

Integer Types

Integers in C are whole numbers without any fractional part.

They can be positive or negative, and their size (memory allocation) and range depend on the machine architecture.

Notes:

1. Integers store **whole numbers**, and their **range of values is machine-dependent**.
2. Generally, an **integer occupies 2 bytes**, and the typical value range is **-32768 to +32767** (-2^{15} to $+2^{15} - 1$).
3. A **signed integer** uses **one bit for the sign** and the remaining **15 bits for the number**.
4. C provides **three classes of integer storage** to control range and memory usage: **short int, int, and long int**.
5. All three storage classes have **signed** and **unsigned** versions.
6. A **short int** requires **half the storage** of a normal integer.
7. **Unsigned integers** can store **only positive values** and use **all bits for magnitude**.
8. Therefore, the typical range of an **unsigned integer** is **0 to 65535**.
9. **Long integers** are used when a **larger range** is required and generally occupy **4 bytes**.

Integer Data Types and Memory Allocation

Data Type	Memory Size	Range (Typical)
short int	1 or 2 bytes	–128 to +127 / –32768 to +32767
unsigned short	1 or 2 bytes	0 to 255 / 0 to 65535
int	2 or 4 bytes	–32768 to +32767 / wider on 32-bit
unsigned int	2 or 4 bytes	0 to 65535 / wider on 32-bit
long int	4 bytes	Very large positive/negative range
unsigned long	4 bytes	Large positive range

Void Types

The void has no values. This is usually used to specify the type of functions. The type of a function said to be void when it does not return any value to the calling function.

Character Types

A single character can be defined as a character (char) type data. Characters are usually stored in 8 bits of internal storage. While unsigned chars have values between 0 to 255, signed chars have values from -128 to 127.

Declaration of Variables

After designing suitable variable names, we must declare them to the compiler. Declaration does two things

1. It tells the compiler what the variable name is
2. It specifies what type of data the variable will hold

The declaration of variables must be done before they are used in the program.

Primary Type Declaration

A variable can be used to store a value of any data type.

Syntax

data-type v1,v2,....,vn;

Where v1,v2, ... vn are variable names. Variables are separated by commas. A declaration statement must end with a semicolon.

Example

```
int count;i
int number,total;
double ratio;
```

int and double are the keywords to represent integer type and real type data values respectively.

Table 2.9 Data Types and Their Keywords

<i>Data type</i>	<i>Keyword equivalent</i>
Character	char
Unsigned character	unsigned char
Signed character	signed char
Signed integer	signed int (or int)
Signed short integer	signed short int (or short int or short)
Signed long integer	signed long int (or long int or long)
Unsigned integer	unsigned int (or unsigned)
Unsigned short integer	unsigned short int (or unsigned short)
Unsigned long integer	unsigned long int (or unsigned long)
Floating point	float
Double-precision floating point	double
Extended double-precision floating point	long double

Declaration of variables is usually done immediately after the opening brace of the program. The variable can also be declared outside of the main function.

Example

```
main()
{
int count;
int number,total;
double ratio;
}
```

Assigning values to variable

Values can be assigned to variables using the assignment operator = as follows:

Syntax: Variable_name=constant

An assignment statement in C is used to give a value to a variable.
Initialization means assigning a value to a variable at the time of declaration.

1. Multiple Assignments

C allows multiple assignment statements to be written in a single line.

Example:

```
initial_value = 0;  
final_value = 100;
```

2. Meaning of an Assignment Statement

An assignment statement sets the left-hand side variable equal to the right-hand side value or expression.

Example:

```
year = year + 1;
```

This means:

The new value of year becomes old value + 1.

3. Automatic Type Conversion

C converts the right-hand side value to the type of the left-hand side variable during assignment.

- If a floating-point value is assigned to an integer variable, truncation may occur (decimal part is removed).

4. Initialization of Variables

You can assign a value to a variable at the time of declaration.

Syntax:

```
data_type variable_name = constant;
```

Examples:

```
int final_value = 100;  
double balance = 75.84;
```

```
char yes = 'x';
```

The process of giving initial values to variables is called initialization.

5. Multiple Variable Initialization in One Statement

C permits initialization of several variables in a single line using the assignment operator.

Examples:

```
p = q = s = 0;  
x = y = z = max;
```

Here, the value is assigned from right to left.

User Defined Type Declaration Type def

`typedef` in C is used to give a **new name (alias)** to an existing data type.

It **does not create a new data type**, but only provides another name for an already existing one.

General Form:

```
typedef existing_type new_name;
```

- **existing_type** → an already defined data type
- **new_name** → the new identifier that will represent the existing type

Example:

```
typedef int units;  
typedef float marks;  
  
units batch1, batch2;    // batch1 and batch2 become int type variables  
marks m1, m2;           // m1 and m2 become float type variables
```

Advantage of typedef

- Helps in creating **meaningful and readable** data type names
- Improves clarity of the program
- Useful in structures, arrays, and complex declarations

Enumerated Data Type (enum)

Definition:

`enum` (enumeration) is a user-defined data type that allows a variable to be assigned **only one value** from a predefined list of constants.

Syntax:

```
enum identifier {value1, value2, value3, ...};
```

- **identifier** → name of the enum type
- **value1, value2...** → enumeration constants

Example:

```
enum day {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday};  
enum day week_st, week_end;  
  
week_st = Monday;  
week_end = Friday;
```

Default Values Assigned by Compiler

The compiler automatically assigns **integer values** starting from **0**.

Constant	Value
Monday	0
Tuesday	1
Wednesday	2
Thursday	3
Friday	4
Saturday	5
Sunday	6

Explicit Value Assignment

You can assign values manually:

```
enum day {Monday = 1, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday};
```

Here:

- Monday = 1
- Tuesday = 2
- ... and so on

Combined Definition and Declaration

You can define an enum and create variables in one statement:

```
enum day {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday}  
week_st, week_end;
```

Defining Symbolic Constants

In many programs, the same constant value may be used several times. For example, the mathematical constant π (pi) is often written as **3.142**. Using such values repeatedly causes two major problems:

1. Problems with Using Numeric Constants Directly

a) Modifiability Issue

If we want to change the value of **pi** from 3.142 to 3.14159 for better accuracy, we must search the entire program and modify every occurrence.

Missing even one value may produce **wrong output**.

b) Understandability Issue

When a number appears in the code, its purpose may not be clear, especially if the same number is used for different reasons in different places.

Example:

100 may mean **maximum value** in one place and **array size** in another.

Solution → Use Symbolic Constants with #define

#define is a **preprocessor directive** used to define symbolic (named) constants.

General Form:

```
#define symbolic_name constant_value
```

Examples:

```
#define PI 3.14159
#define MAX 100
```

- Here, **PI** and **MAX** are *symbolic names* (also called constant identifiers).
- Wherever PI appears in the program, the compiler replaces it with **3.14159**.

Rules for Using #define

1. Symbolic names follow the same rules as variable names.
(They cannot start with a digit, no special symbols except underscore.)
2. No space is allowed between # and define.
3. #define PI 3.14 ✓
4. # define PI 3.14 ✗
5. # must be the first character in the line.
6. One blank space is required between:
 - #define and symbolic name
 - symbolic name and value
7. #define statements do NOT end with a semicolon.
(Adding ; is an error.)
8. Symbolic names must not be assigned new values using assignment statements.
They cannot be changed during program execution.
9. Symbolic constants have no data type declared in the #define.
Their type depends on the value given.
10. #define may appear anywhere in the program,
but must be placed before it is used.

DECLARING A VARIABLE AS CONSTANT

We may want the value of some variables to **remain unchanged** throughout the execution of a program.

To ensure this, the variable is declared using the qualifier **const** at the time of initialization.

Example

```
const int class_size = 40;
```

Explanation

This tells the compiler that the value of the integer variable `class_size` must **not be modified** by the program.

Variable as Volatile

The qualifier `volatile` is used to explicitly tell the compiler that the value of a variable **may change at any time** due to **external sources**, i.e., from outside the program.

Example

```
volatile int date;
```

Explanation

- The value of **date** may be altered by external factors.
- When a variable is declared as **volatile**, the compiler will always re-check its value each time it is encountered to see whether an external operation has changed it.
- A volatile variable **can also be modified by the program**, unless it is declared as `const`.

Using both const and volatile

If a value **must not** be modified by the program but **may** be altered by some external process, the variable can be declared using **both** qualifiers.

Example

```
volatile const int location = 100;
```