B.Tech (CSE) First Year Group A

Course: Computer Programming: Concepts and Practices (CS-1001)

Unit 1: Computer fundamentals, Evolution of programming languages, Syntax and semantics, Concurrency, Number systems, Functional Programming and Logic programming languages, Computational complexity.

Unit 2: Introduction to Programming, Pseudo-code, Character set, Identifiers, Keywords, Data Types, Constant and Variables, Operators, expressions and statements, conditional and looping statements.

Unit 3: Data types, Type Checking and Scopes, Storage Classes, Arrays, Sequential and Linked linear lists, Trees, Trees representations, binary tree traversals, Graphs, Graphs representations.

Unit 4: Functions, Structures, Union, Storage Classes, Pointers, Dynamic memory allocations, file handling in C, Pre-processor directives and macros, I/O handling, Header files.

Unit 5: Sorting and searching algorithms, String algorithms, Pattern search and text editing.

Computer fundamentals-

Basic computer concepts

A ***computer*** is a device that can perform computations and make logical decisions billions of times faster than human beings can.

Today's fastest supercomputers can perform thousands of trillions (quadrillions) of instructions per second!

A ***quadrillion-instruction-per-second*** computer can perform more than 100,000 calculations per second for every person on the planet!

US : a number equal to 1 followed by 15 zeros

British : a number equal to 1 followed by 24 zeros

Computers process **data** under the control of **sets of instructions,** called **computer programs**. These programs guide the computer through **orderly sets of actions specified** by people called **computer programmers**.

Computer fundamentals-

➢ Computers: Hardware and Software

A computer consists of various devices referred to as hardware (e.g., the keyboard, screen, mouse, hard disk, memory, DVDs and processing units). The programs that run on a computer are referred to as software.

➢ Computer Organization

Regardless of differences in physical appearance, virtually every computer may be envisioned as divided into six logical units or sections:

- Input unit
- Output unit
- Memory unit
- Arithmetic and logic unit (ALU)
- Central processing unit (CPU)
- Secondary storage unit

Computer fundamentals-

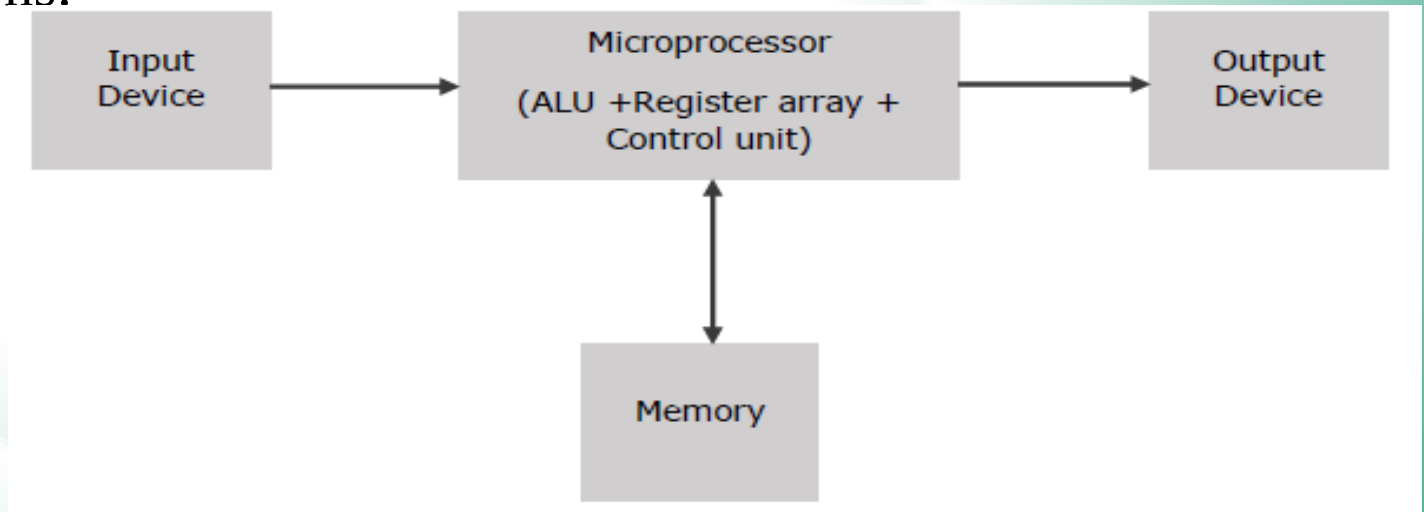
➢ Computers: Hardware and Software

A computer consists of various devices referred to as hardware (e.g., the keyboard, screen, mouse, hard disk, memory, DVDs and processing units). The programs that run on a computer are referred to as software.

➢ Computer Organization

Regardless of differences in physical appearance, virtually every computer may be envisioned as divided into six logical units or sections:

- Input unit
- Output unit
- Memory unit
- Arithmetic and logic unit (ALU)
- Central processing unit (CPU)
- Secondary storage unit



Many of today's computers have multiple CPUs and, hence, can perform many operations simultaneously—such computers are called multiprocessors. A multi-core processor implements multiprocessing on a single integrated circuit chip—for example a dual-core processor has two CPUs and a quad-core processor has four CPUs.

Personal Computer----Distributed Computer----Client/Server Computer----------Internet (WWW)

Evolution of programming languages-

- The different types of programming languages.
- Machine Languages, Assembly Languages and High-Level Languages
- The history of the C programming language.
- The purpose of the C Standard Library.
- The elements of a typical C program development environment.
- How C provides a foundation for further study of programming languages in general and of C++, Java and C# in particular.

Evolution of programming languages-

Programming Language is indeed the fundamental unit of today's tech world. It is considered as the set of commands and instructions that we give to the machines to perform a particular task.

The device was made by Charles Babbage and the code was written by Ada Lovelace for computing Bernoulli's number. 1883

1949: Assembly Language- low-level language consists of instructions (kind of symbols) that only machines could understand.

1952: Autocode- Developed by Alick Glennie and first compiled computer programming language. COBOL and FORTRAN are the languages referred to as Autocode.

1957: FORTRAN- Developed by John Backus and IBM for numeric computation and scientific computing.

•Software for NASA probes voyager-1 (space probe) and voyager-2 (space probe) was originally written in FORTRAN 5.

1958: ALGOL- ALGOrithmic **L**anguage. first language implementing the nested function and has a simple syntax than FORTRAN. having a code block like "begin" that indicates that your program has started and "end" means you have ended your code.

1959: COBOL- COmmon **B**usiness-**O**riented **L**anguage and 1997, 80% of the world's business ran on Cobol.

1964: BASIC- All-purpose symbolic instruction code. 1991 Microsoft released Visual Basic, an updated version of Basic The first microcomputer version of Basic was co-written by Bill Gates, Paul Allen, and Monte Davidoff for their newly-formed company, Microsoft.

1972: C- general-purpose, procedural programming language and the most popular programming language till now. All the code that was previously written in assembly language gets replaced by the C language like operating system, kernel, and many other applications.

•It can be used in implementing an operating system, embedded system, and also on the website using the Common Gateway Interface (CGI). C is the mother of almost all higher-level programming languages like C#, D, Go, Java, JavaScript, Limbo, LPC, Perl, PHP, Python, and Unix's C shell.

• 1972-SQL, 1978 MATLAB, 1983 Objective C, C++, 1991 Python, 1995 JAVA, PHP, JavaScript, 2000 C#, 2014 Swift

History of programming languages-

C evolved from two previous languages, BCPL and B. BCPL was developed in 1967 by Martin Richards as a language for writing operating-systems software and compilers. Ken Thompson modeled many features in his B language after their counterparts in BCPL, and in 1970 he used B to create early versions of the UNIX operating system at Bell Laboratories. B

The C language was evolved from B by Dennis Ritchie at Bell Laboratories and was originally implemented on a DEC PDP-11 computer in 1972. C uses many of the important concepts of BCPL and B while adding data typing and other powerful features. C initially became widely known as the development language of the UNIX operating system. Today, virtually all new major operating systems are written in C and/or C++. C is available for most computers. C is mostly hardware independent. With careful design, it's possible to write C programs that are portable to most computers.

By the late 1970s, C had evolved into what is now referred to as "traditional C." The publication in 1978 of Kernighan and Ritchie's book, The C Programming Language, drew wide attention to the language. This became one of the most successful computer science books of all time.

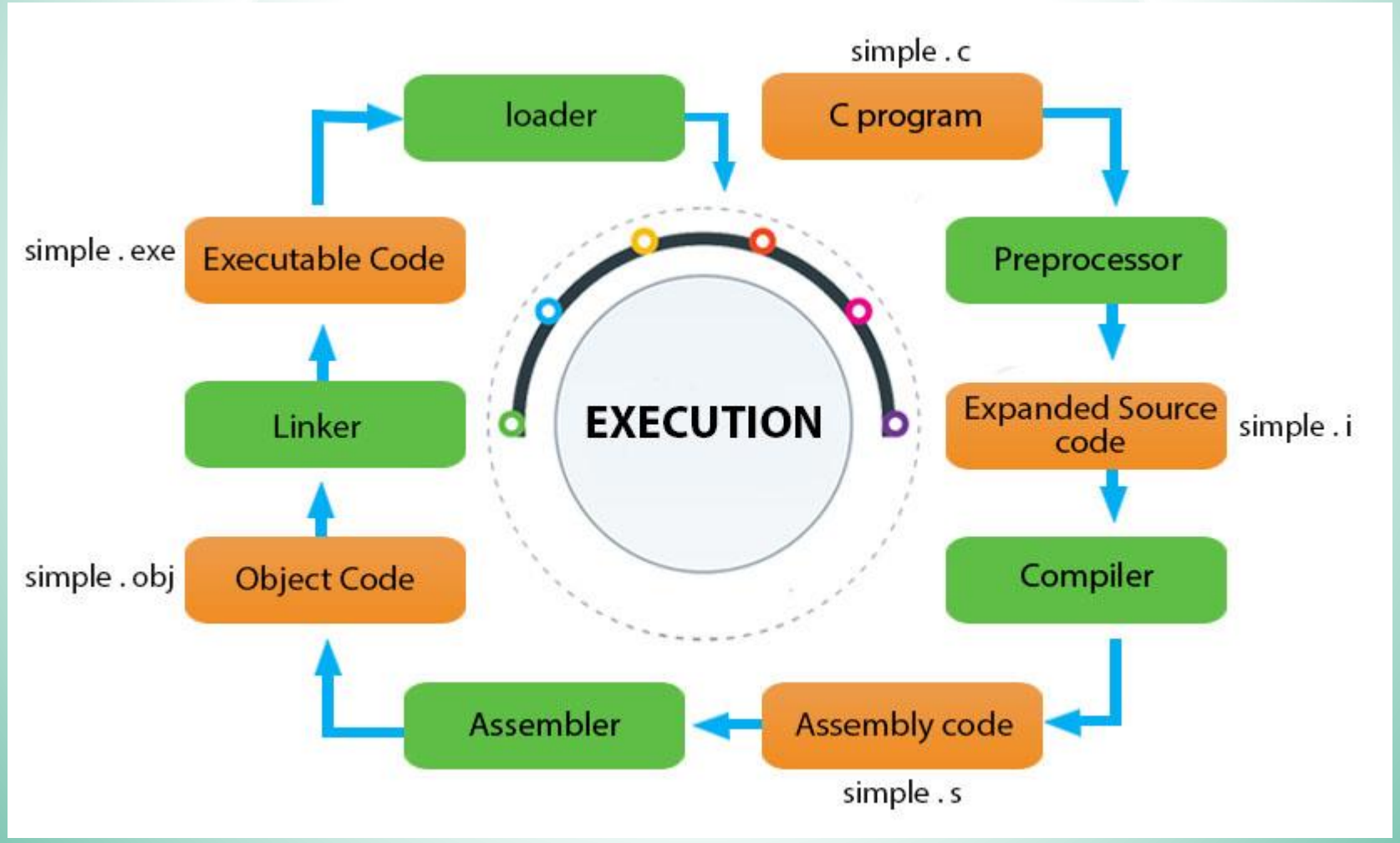
C programming language was developed in 1972 by Dennis Ritchie at bell laboratories of AT&T (American Telephone & Telegraph), located in the U.S.A.

Dennis Ritchie is known as the **founder of the c language**.

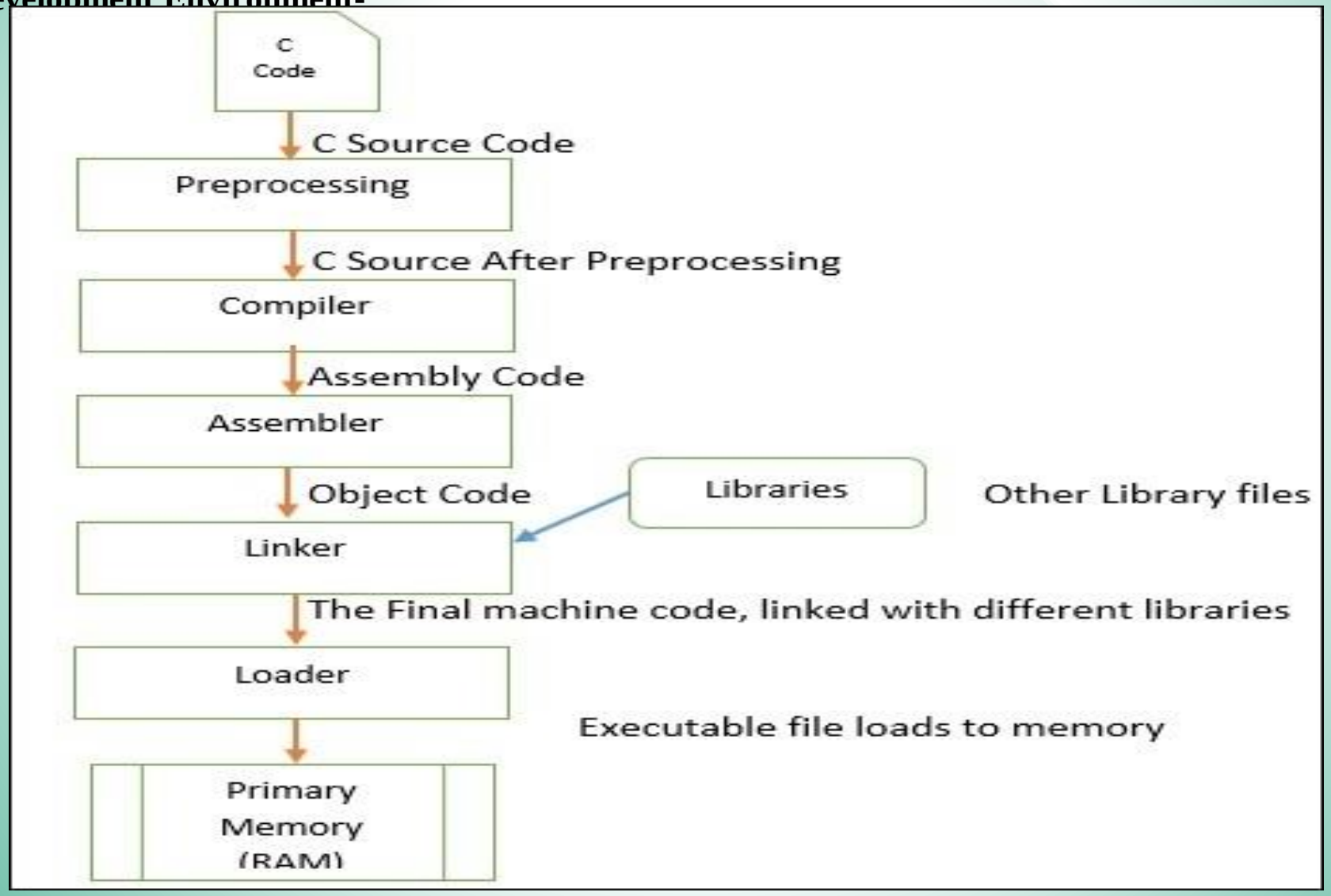
It was developed to overcome the problems of previous languages such as B, BCPL, etc.

C Program Development Environment-

C programs typically go through six phases to be executed. These are: edit, preprocess, compile, link, load and execute.



C Program Development Environment-



Compile and Execute C Program

Let us see how to save the source code in a file, and how to compile and run it. Following are the simple steps −

Open a text editor and add the above-mentioned code.

Save the file as hello.c

Open a command prompt and go to the directory where you have saved the file.

Type gcc hello.c and press enter to compile your code.

If there are no errors in your code, the command prompt will take you to the next line and would generate a.out executable file.

Now, type a.out to execute your program.

You will see the output "Hello World" printed on the screen.

```
$ gcc hello.c
$ ./a.out
Hello, World!
```

Laboratory: CS1101

```
#include <stdio.h>
int main() {
// printf() displays the string inside quotation
printf("Hello, World!");
return 0;
}
```

•The *#include* is a preprocessor command that tells the compiler to include the contents of *stdio.h* (standard input and output) file in the program.
•The *stdio.h* file contains functions such as *scanf()* and *printf()* to take input and display output respectively.
•If you use the *printf()* function without writing *#include <stdio.h>*, the program will not compile.
•The execution of a C program starts from the *main()* function.
•*printf()* is a library function to send formatted output to the screen. In this program, *printf()* displays Hello, World! text on the screen.
•The return 0; statement is the **"Exit status"** of the program. In simple terms, the program ends with this statement.

Laboratory: CS1101

```
#include <stdio.h>
int main() {
    int number;
     printf("Enter an integer: ");
        // reads and stores input
    scanf("%d", &number);
    // displays output
    printf("You entered: %d", number);
        return 0;
}
```

int number;
Then, the user is asked to enter an integer number. This number is stored in the number variable.
printf("Enter an integer: "); scanf("%d", &number);

Finally, the value stored in number is displayed on the screen using *printf().*

Enter two integers: 12 11 12 + 11 = 23

Laboratory: CS1101

```
#include <stdio.h>
int main() {
    int number1, number2, sum, min;
        printf("Enter two integers: ");
    scanf("%d %d", &number1, &number2);
    // calculating sum
    sum = number1 + number2;
   min= number1-number 2;
        printf("%d + %d = %d %d", number1, number2, sum, min);
    return 0;
}
```

In this program, the user is asked to enter two integers. These two integers are stored in variables number1 and number2 respectively.

```
printf("Enter two integers: "); scanf("%d %d", &number1, &number2);
```

Then, these two numbers are added using the + operator, and the result is stored in the sum variable.

```
sum = number1 + number2;
```

Laboratory: CS1101

```
#include <stdio.h>
int main() {
    double a, b, product;
    printf("Enter two numbers: ");
    scanf("%lf %lf", &a, &b);
     // Calculating product
    product = a * b;
    // %.2lf displays number up to 2 decimal point
    printf("Product = %.2lf", product);
     return 0;
}
```

In this program, the user is asked to enter two numbers which are stored in variables a and b respectively.

```
printf("Enter two numbers: "); scanf("%lf %lf", &a, &b);
```

Then, the product of a and b is evaluated and the result is stored in product.

```
product = a * b;
```

Finally, product is displayed on the screen using printf().

```
printf("Product = %.2lf", product);
```

Notice that, the result is rounded off to the second decimal place using %.2lf conversion character.

Laboratory Assignment 1-

1. **C Program to Check Whether a Number is Prime or Not**
2. **C Program to Check Whether a Number is Palindrome or Not**
3. **C Program to Find the Size of int, float, double and char**
4. **C Program to Swap Two Numbers**

High thoughts must have high language – Aristophanes

- **Syntax:**

Like ordinary language English, programming languages have syntax. The Syntax of a (programming) language is a set of rules that define what sequences of symbols are considered to be valid expression (programs) in the language.
or
The Syntax of a programming language is what the program looks like. Syntax provides significant information needed for understanding a program and provides much needed information towards the translation of the source program into the object program.

A valid representation of syntax

X=Y+Z

Invalid representation maybe

XY+-

- **2+3*4 text will be interpreted this expression as having value**

14 or 20.

That is, expression is interpreted as if written (2+3)*4.

We can specify either interpretation, if we wish, by syntax and hence guide the translator into generating the correct operations for evaluating this expression.

- **Semantics: Semantics is the meaning of an expression (program) in a programming language.**
- In C to declare a 10 elements vector V of Integer has declaration

- **int v:{10};**

In Pascal
v:array[0.....9] of integer
Although both creates similar data objects at run time , their syntax is very different. To Understand the meaning of declaration we need to know the semantics of both pascal and c for such array declaration.

- **Another example**

```
scanf("%d %d", &number1, &number2);
;
```

- **While (< boolean_exp>)< statement>**

The semantics of this statement form is that when the current value of the Boolean exp. Is true, the embedded statement is true.

We can conclude – Syntax

- It refers to the rules and regulations for writing any statement in a programming language like C/C++.
- It does not have to do anything with the meaning of the statement.
- A statement is syntactically valid if it follows all the rules.
- It is related to the grammar and structure of the language.

Semantics:

- It refers to the meaning associated with the statement in a programming language.
- It is all about the meaning of the statement which interprets the program easily.
- Errors are handled at runtime.

Computer - Number System

When we type some letters or words, the computer translates them in numbers as computers can understand only numbers. A computer can understand the positional number system where there are only a few symbols called digits and these symbols represent different values depending on the position they occupy in the number.

The value of each digit in a number can be determined using −

The digit

The position of the digit in the number

The base of the number system (where the base is defined as the total number of digits available in the number system)

S.No.	Number System and Description
1	**Binary Number System** Base 2. Digits used : 0, 1
2	**Octal Number System** Base 8. Digits used : 0 to 7
3	**Hexa Decimal Number System** Base 16. Digits used: 0 to 9, Letters used : A- F

Binary Number System

Characteristics of the binary number system are as follows −

•Uses two digits, 0 and 1

•Also called as base 2 number system

•Each position in a binary number represents a **0** power of the base (2). Example 2^0

•Last position in a binary number represents a **x** power of the base (2). Example 2^x where **x** represents the last position - 1.

Example

Binary Number: 10101_2

Calculating Decimal Equivalent −

Step	Binary Number	Decimal Number
Step 1	10101_2	$((1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0))_{10}$
Step 2	10101_2	$(16 + 0 + 4 + 0 + 1)_{10}$
Step 3	10101_2	21_{10}

Note − 10101_2 is normally written as 10101.

Octal Number System

Characteristics of the octal number system are as follows −

- Uses eight digits, 0,1,2,3,4,5,6,7
- Also called as base 8 number system
- Each position in an octal number represents a **0** power of the base (8). Example 8^0
- Last position in an octal number represents a **x** power of the base (8). Example 8^x where **x** represents the last position - 1

Example

Octal Number: 12570_8

Calculating Decimal Equivalent −

Step	Octal Number	Decimal Number
Step 1	12570_8	$((1 \times 8^4) + (2 \times 8^3) + (5 \times 8^2) + (7 \times 8^1) + (0 \times 8^0))_{10}$
Step 2	12570_8	$(4096 + 1024 + 320 + 56 + 0)_{10}$
Step 3	12570_8	5496_{10}

Note − 12570_8 is normally written as 12570.

Hexadecimal Number System

Characteristics of hexadecimal number system are as follows −

•Uses 10 digits and 6 letters, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

•Letters represent the numbers starting from 10. A = 10. B = 11, C = 12, D = 13, E = 14, F = 15

•Also called as base 16 number system

•Each position in a hexadecimal number represents a **0** power of the base (16). Example, 16^0

•Last position in a hexadecimal number represents a **x** power of the base (16). Example 16^x where **x** represents the last position - 1

Example

Hexadecimal Number: $19FDE_{16}$

Calculating Decimal Equivalent −

Note − $19FDE_{16}$ is normally written as 19FDE.

Step	Binary Number	Decimal Number
Step 1	$19FDE_{16}$	$((1 \times 16^4) + (9 \times 16^3) + (F \times 16^2) + (D \times 16^1) + (E \times 16^0))_{10}$
Step 2	$19FDE_{16}$	$((1 \times 16^4) + (9 \times 16^3) + (15 \times 16^2) + (13 \times 16^1) + (14 \times 16^0))_{10}$
Step 3	$19FDE_{16}$	$(65536 + 36864 + 3840 + 208 + 14)_{10}$
Step 4	$19FDE_{16}$	106462_{10}

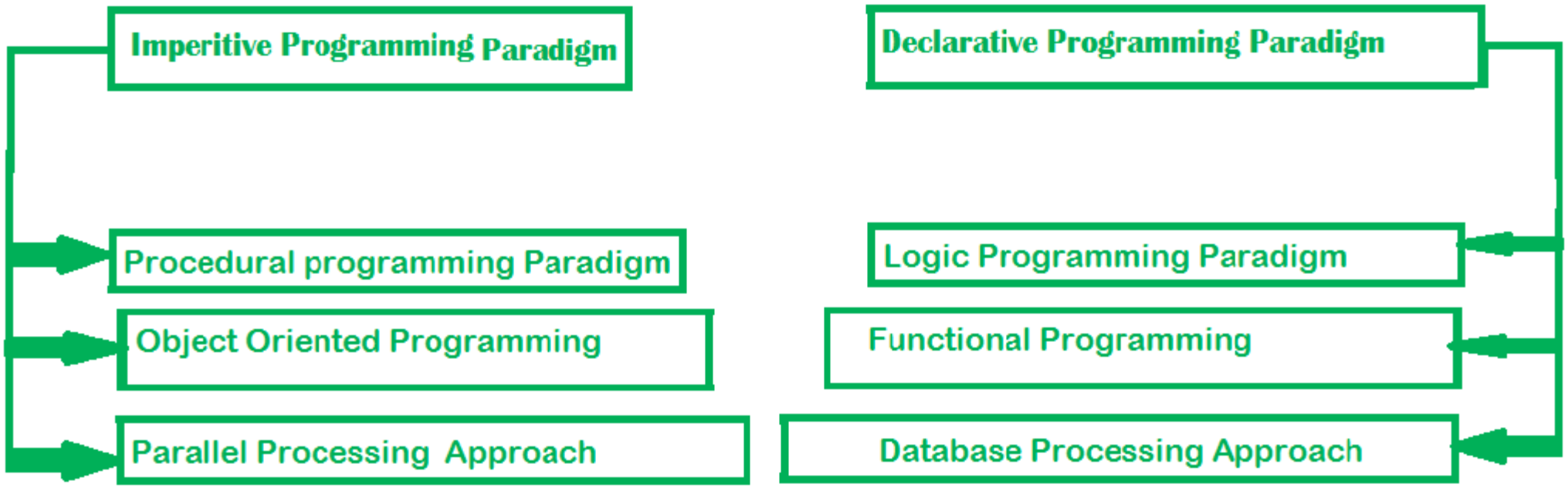
- **Difference Between Functional and Logical Programming**

Programming paradigm is an approach to solve problems using some programming language or also we can say it is a method to solve a problem using tools and techniques that are available to us following some approach.

There are lots of programming languages that are known but all of them need to follow some strategy when they are implemented and this methodology/strategy is paradigms. Apart from varieties of programming languages, there are lots of paradigms to fulfill each and every demand. They are discussed below as follows:

Functional Programming is a type of programming paradigm in which everything is done with the help of functions and using functions as its basic building blocks. In it, we simply try to bind each and everything in a purely mathematical functions' style. Programs are generally written at a higher level and are therefore much easier to comprehend.

Programming Paradigms



- **Logical Programming** is a type of programming paradigm that uses logic circuits to control how facts and rules about the problems within the system are represented or expressed. In it, logic is used to represent knowledge, and inference is used to manipulate it. It tells the model about how to accomplish a goal rather than what goal to accomplish.
- Now let us go through the major key differences between them after going through the basics of both of them. Differences are shown below in a tabular format as follows:

Functional Programming	Logical Programming
It is totally based on functions.	It is totally based on formal logic.
In this programming paradigm, program statements usually express or represent facts and rules related to problems within a system of formal logic.	In this programming paradigm, programs are constructed by applying and composing functions.
These are specially designed to manage and handle symbolic computation and list processing applications.	These are specially designed for fault diagnosis, natural language processing, planning, and machine learning.
Its main aim is to reduce side effects that are accomplished by isolating them from the rest of the software code.	Its main aim is to allow machines to reason because it is very useful for representing knowledge.

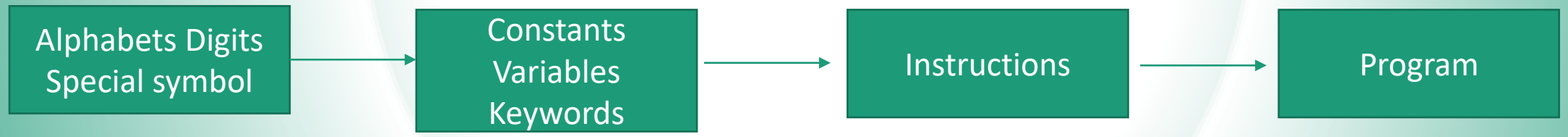
- In computer science, the **computational complexity** or simply **complexity** of an algorithm is the amount of resources required to run it. Particular focus is given to time and memory requirements. The complexity of a problem is the complexity of the best algorithms that allow solving the problem.

Character set, Identifiers, Keywords, Data Types, Constant and Variables, Operators

Steps in learning English language:

Alphabets → Words → Sentences → Paragraphs

Steps in learning C



A character denotes any alphabet, digit or special symbol used to represent information. (valid alphabets, numbers and special symbols allowed in C.)

- Alphabets A, B, ….., Y, Z a, b, ……, y, z
- Digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
- Special symbols ~ ‘ ! @ # % ^ & * ( ) _ - + = | \ { } [ ] : ; " ' < > , . ? /

White space Characters

- Blank space, newline, horizontal tab, carriage return and form feed.

C Identifiers

Identifier refers to name given to entities such as variables, functions, structures etc.

Identifiers must be unique. They are created to give a unique name to an entity to identify it during the execution of the program. For example:

```
int money; double accountBalance;
```

Here, money and accountBalance are identifiers.

Also remember, identifier names must be different from keywords. You cannot use int as an identifier because int is a keyword.

Rules for naming identifiers

1. A valid identifier can have letters (both uppercase and lowercase letters), digits and underscores.
2. The first letter of an identifier should be either a letter or an underscore.
3. You cannot use keywords like int, while etc. as identifiers.
4. There is no rule on how long an identifier can be. However, you may run into problems in some compilers if the identifier is longer than 31 characters.

You can choose any name as an identifier if you follow the above rule, however, give meaningful names to identifiers that make sense.

Constants, Variables and Keywords

The alphabets, numbers and special symbols when properly combined form constants, variables and keywords.

C Constants

Primary Constants: Integer Constant, Real Constant, Character Constant

Secondary Constants :Array, Pointer, Structure, Union, Enum, etc.

Rules for Constructing Integer Constants

- An integer constant must have at least one digit.
- It must not have a decimal point.
- It can be either positive or negative.
- If no sign precedes an integer constant it is assumed to be positive.
- No commas or blanks are allowed within an integer constant.
- The allowable range for integer constants is -32768 to 32767.

- Rules for Constructing Real Constants

Real constants are often called Floating Point constants. The real constants could be written in two forms—Fractional form and Exponential form.

- Following rules must be observed while constructing real
- constants expressed in fractional form:
- A real constant must have at least one digit.
- It must have a decimal point.
- It could be either positive or negative.
- Default sign is positive.
- No commas or blanks are allowed within a real constant.

- Rules for Constructing Character Constants

A character constant is a single alphabet, a single digit or a single special symbol enclosed within single inverted commas. Both the inverted commas should point to the left.

- For example, ’A’ is a valid character constant whereas ‘A’ is not.
- The maximum length of a character constant can be 1 character.

- As we saw earlier, an entity that may vary during program execution is called a variable. Variable names are names given to locations in memory. These locations can contain integer, real or character constants

Rules for Constructing Variable Names (a) (b) (c) (d) A variable name is any combination of 1 to 31 alphabets, digits or underscores. Some compilers allow variable names whose length could be up to 247 characters. Still, it would be safer to stick to the rule of 31 characters. Do not create unnecessarily long variable names as it adds to your typing effort. The first character in the variable name must be an alphabet or underscore. No commas or blanks are allowed within a variable name. No special symbol other than an underscore (as in gross_sal) can be used in a variable name. Ex.: si_int m_hra

- Keywords are the words whose meaning has already been explained to the C compiler (or in a broad sense to the computer). The keywords cannot be used as variable names because if we do so we are trying to assign a new meaning to the keyword, which is not allowed by the computer

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

Literals

Literals are data used for representing fixed values. They can be used directly in the code. For example: 1, 2.5, 'c' etc.

Here, 1, 2.5 and 'c' are literals. Why? You cannot assign different values to these terms.

- **1. Integers**
- An integer is a numeric literal(associated with numbers) without any fractional or exponential part. There are three types of integer literals in C programming:
- decimal (base 10)
- octal (base 8)
- hexadecimal (base 16)
- **1. Integers**
- An integer is a numeric literal(associated with numbers) without any fractional or exponential part. There are three types of integer literals in C programming:
- decimal (base 10)
- octal (base 8)
- hexadecimal (base 16)

String Literals

A string literal is a sequence of characters enclosed in double-quote marks. For example:

```
"good"                  //string constant
""                  //null string constant
"      "            //string constant of six white space
"x"                  //string constant having a single character.
"Earth is round\n"        //prints string with a newline
```

Basic types

Here's a table containing commonly used types in C programming for quick access.

Type	Size (bytes)	Format Specifier
int	at least 2, usually 4	%d, %i
char	1	%c
float	4	%f
double	8	%lf
short int	2 usually	%hd
unsigned int	at least 2, usually 4	%u
long int	at least 4, usually 8	%ld, %li
long long int	at least 8	%lld, %lli
unsigned long int	at least 4	%lu
unsigned long long int	at least 8	%llu
signed char	1	%c
unsigned char	1	%c
long double	at least 10, usually 12 or 16	%Lf

```
#include <stdio.h>
int main() {
  short a;
  long b;
  long long c;
  long double d;

  printf("size of short = %d bytes\n", sizeof(a));
  printf("size of long = %d bytes\n", sizeof(b));
  printf("size of long long = %d bytes\n", sizeof(c));
  printf("size of long double= %d bytes\n", sizeof(d));
  return 0;
}
```

```
size of short = 2 bytes
size of long = 4 bytes
size of long long = 8 bytes
size of long double= 16 bytes
```

Signed and unsigned??

Arithmetic in C

C operation

Arithmetic operator, Algebraic expression, C expression

Addition + f + 7 f + 7

Subtraction – p – c p - c

Multiplication * bm b * m

Division / x / y or or x ÷ y x / y

Remainder % r mod s r % s

Operator(s)	Operation(s)	Order of evaluation (precedence)
( )	Parentheses	Evaluated first. If the parentheses are nested, the expression in the innermost pair is evaluated first. If there are several pairs of parentheses "on the same level" (i.e., not nested), they're evaluated left to right.
*	Multiplication	Evaluated second. If there are several, they're evaluated left to right.
/	Division	
%	Remainder	
+	Addition	Evaluated last. If there are several, they're evaluated left to right.
-	Subtraction	

Algebraic equality or relational operator	C equality or relational operator	Example of C condition	Meaning of C condition
Equality operators			
=	==	x == y	x is equal to y
≠	!=	x != y	x is not equal to y
Relational operators			
>	>	x > y	x is greater than y
<	<	x < y	x is less than y
≥	>=	x >= y	x is greater than or equal to y
≤	<=	x <= y	x is less than or equal to y

```c
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   int num1; /* first number to be read from user  */
   int num2; /* second number to be read from user */

   printf( "Enter two integers, and I will tell you\n" );
   printf( "the relationships they satisfy: " );

   scanf( "%d%d", &num1, &num2 ); /* read two integers */

   if ( num1 == num2 ) {
      printf( "%d is equal to %d\n", num1, num2 );
   } /* end if */

   if ( num1 != num2 ) {
      printf( "%d is not equal to %d\n", num1, num2 );
   } /* end if */

   if ( num1 < num2 ) {
      printf( "%d is less than %d\n", num1, num2 );
   } /* end if */
```

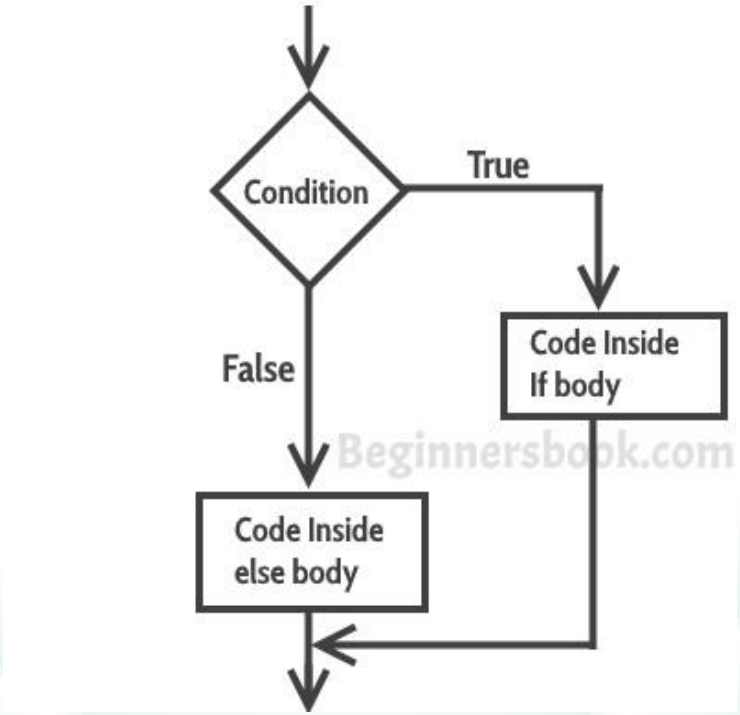
Operators	Associativity
()	left to right
* / %	left to right
+ -	left to right
< <= > >=	left to right
== !=	left to right
=	right to left

```c
#include <stdio.h>
int main() {
    int num;
    printf("Enter an integer: ");
    scanf("%d", &num);

    // true if num is perfectly divisible by 2
    if(num % 2 == 0)
        printf("%d is even.", num);
    else
        printf("%d is odd.", num);

    return 0;
}
```

```
#include <stdio.h>

int main() {

  int year;

  printf("Enter a year: ");

  scanf("%d", &year);  // leap year if perfectly divisible by 400

  if (year % 400 == 0) {

    printf("%d is a leap year.", year);   }

  // not a leap year if divisible by 100

  // but not divisible by 400

  else if (year % 100 == 0) {

    printf("%d is not a leap year.", year);   }    // leap year if not divisible by 100    // but divisible by 4

  else if (year % 4 == 0) {

    printf("%d is a leap year.", year);   }

  // all other years are not leap years

  else {  printf("%d is not a leap year.", year);   }

  return 0;}
```

Control Structures:

Normally, statements in a program are executed one after the other in the order in which they're written------**sequential execution**. Various C statements we'll soon discuss enable you to specify that the next statement to be executed may be other than the next one in sequence ----- **transfer of control**.



Examination-Example

```
if ( marks >= 60 ) {
printf( "Passed\n" );
} /* end if */
else {
Printf ( "Failed\n" );
} /* end else */
```

```
if ( grade >= 90 )
printf( "A\n" );
else
if ( grade >= 80 )
printf("B\n");
else
if ( grade >= 70 )
printf("C\n");
else
if ( grade >= 60 )
printf( "D\n" );
else
printf( "F\n" );
```

```
if ( grade >= 90 )
printf( "A\n" );
else if ( grade >= 80 )
printf( "B\n" );
else if ( grade >= 70 )
printf( "C\n" );
else if ( grade >= 60 )
printf( "D\n" );
else
printf( "F\n" );
```

When an if else statement is present inside the body of another “if” or “else” then this is called nested if else.

The else..if statement is useful when you need to check multiple conditions within the program, nesting of if-else blocks can be avoided using else..if statement

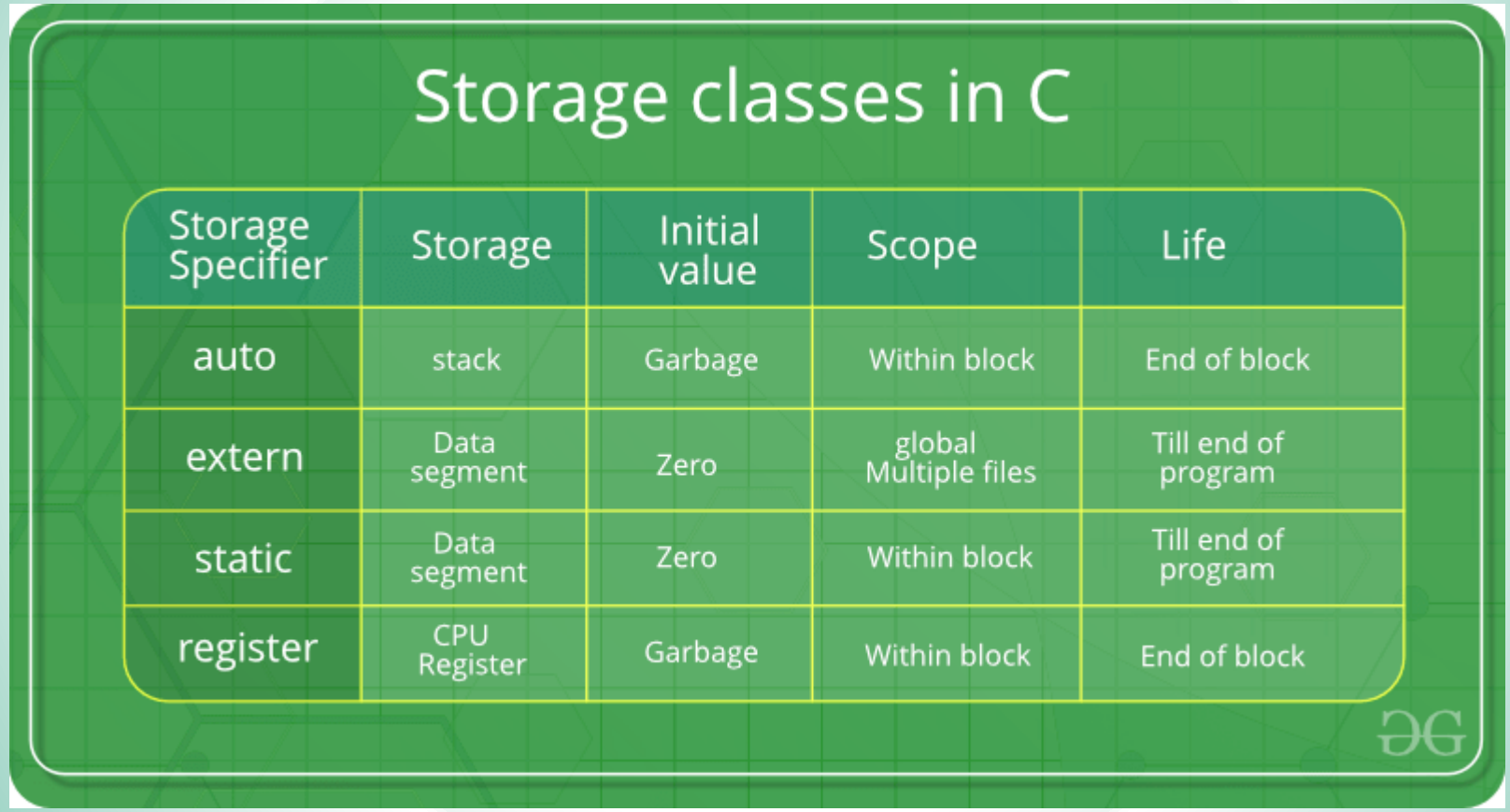
```
#include <stdio.h>
int main()
{
   int year;

   printf("Enter year : ");
   scanf("%d", &year);
   /*
    * If year is exactly divisible by 4  and year is not divisible by 100
    * or year is exactly divisible by 400 then      * the year is leap year.
    * Else year is normal year      */
   if(((year % 4 == 0) && (year % 100 !=0)) || (year % 400==0))
   {
      printf("LEAP YEAR");
   }
   else
   {
      printf("COMMON YEAR");
   }
   return 0;
}
```

```
for ( i = 100; i >= 1; i-- )
for ( i = 7; i <= 77; i += 7 )
for ( i = 20; i >= 2; i -= 2 )
for ( j = 2; j <= 17; j += 3 )
for ( j = 44; j >= 0; j -= 11 )
```

```c
#include <stdio.h>
int main()
{
   char grade ;
   printf("Enter your grade\n");
   scanf(" %c" , &grade);
   switch(grade)
   {
      case 'A':
         printf("Excellent!\n");
         break;
      case 'B':
         printf("Outstanding!\n");
         break;
      case 'C':
         printf("Good!\n");
         break;
      case 'D':
         printf("Can do better\n");
         break;
      case 'E':
         printf("Just passed\n");
         break;
      case 'F':
         printf("You failed\n");
         break;
      default:
         printf("Invalid grade\n");
   }
```

C Storage Classes:

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



auto: This is the default storage class for all the variables declared inside a function or a block.

Hence, the keyword auto is rarely used while writing programs in C language.

Auto variables can be only accessed within the block/function they have been declared and not outside them (which defines their scope).

they can be accessed outside their scope as well using the concept of pointers given here by pointing to the very exact memory location where the variables resides.

They are assigned a garbage value by default whenever they are declared.

Extern storage class simply tells us that the variable is defined elsewhere and not within the same block where it is used.

Basically, the value is assigned to it in a different block and this can be overwritten/changed in a different block as well.

So an extern variable is nothing but a global variable initialized with a legal value where it is declared in order to be used elsewhere. It can be accessed within any function/block.

Also, a normal global variable can be made extern as well by placing the 'extern' keyword before its declaration/definition in any function/block. This basically signifies that we are not initializing a new variable but instead we are using/accessing the global variable only.

The main purpose of using extern variables is that they can be accessed between two different files which are part of a large program. For more information on how extern variables work, have a look at this

Static storage class is used to declare static variables which are popularly used while writing programs in C language.

Static variables have a property of preserving their value even after they are out of their scope!

Hence, static variables preserve the value of their last use in their scope.

So we can say that they are initialized only once and exist till the termination of the program.

Thus, no new memory is allocated because they are not re-declared.

Their scope is local to the function to which they were defined. Global static variables can be accessed anywhere in the program. By default, they are assigned the value 0 by the compiler.

Register storage class declares register variables which have the same functionality as that of the auto variables.

The only difference is that the compiler tries to store these variables in the register of the microprocessor if a free register is available.

This makes the use of register variables to be much faster than that of the variables stored in the memory during the runtime of the program. If a free register is not available, these are then stored in the memory only.

Usually few variables which are to be accessed very frequently in a program are declared with the register keyword which improves the running time of the program. An important and interesting point to be noted here is that we cannot obtain the address of a register variable using pointers.

Array:

An array is a group of memory locations related by the fact that they all have the same name and the same type. To refer to a particular location or element in the array, we specify the name of the array and the position number of the particular element in the array.

Function:

- Functions- Introduction
- Function Definitions
- Function Prototypes
- Function Call Stack and Activation Records
- Calling Functions By Value and By Reference
- Recursion
- Example Using Recursion: Fibonacci Series
- Recursion vs. Iteration

Function:

➢ Functions

Function declaration A function must be declared globally in a c program to tell the compiler about the function name, function parameters, and return type.

Function call Function can be called from anywhere in the program. The parameter list must not differ in function calling and function declaration. We must pass the same number of functions as it is declared in the function declaration.

Function definition It contains the actual statements which are to be executed. It is the most important aspect to which the control comes when the function is called. Here, we must notice that only one value can be returned from the function.

Function:

➢ Functions

1. **Library Functions**: are the functions which are declared in the C header files such as scanf(), printf(), gets(), puts(), ceil(), floor() etc.
2. **User-defined functions**: are the functions which are created by the C programmer, so that he/she can use it many times. It reduces the complexity of a big program and optimizes the code.

Function Declarations –

```
return_type function_name ( parameter list );
```

Defining a Function-

```
return_type function_name( parameter list )
{
body of the function
}
```

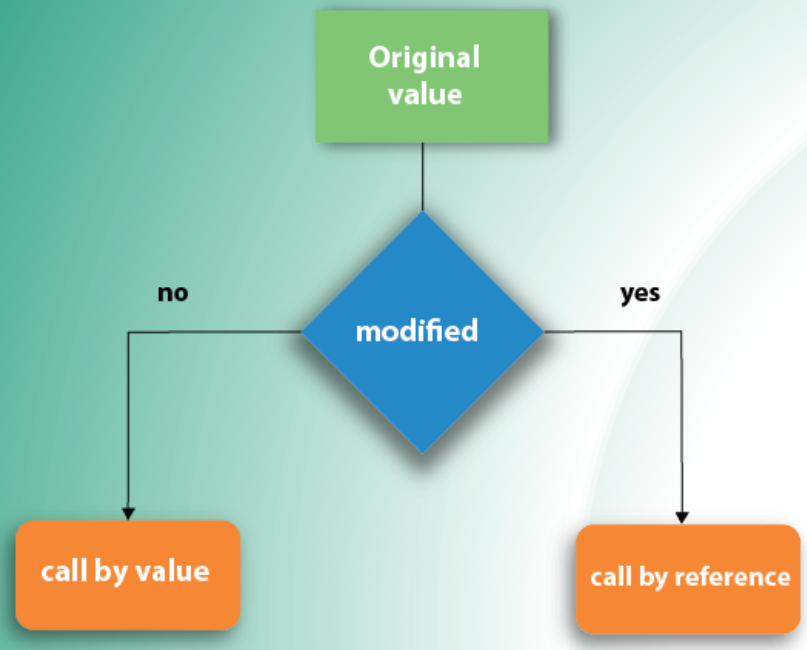
Calling a Function-
When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.

Function Arguments- While calling a function, there are two ways in which arguments can be passed to a function –

This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.- call by value

This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

```
#include <stdio.h>
int max(int num1, int num2);                    /* function declaration */
int main () {
int a = 100;    int b = 200;  int ret;          /* local variable definition */
   ret = max(a, b);                             /* calling a function to get max value */
   printf( "Max value is : %d\n", ret );
   return 0; }
 int max(int num1, int num2)                    /* function definition - returning the max between two numbers */
 {
int result;                                     /* local variable declaration */
  if (num1 > num2)
     result = num1;
  else
     result = num2;
  return result;
}
```



Call by value in C

•In call by value method, the value of the actual parameters is copied into the formal parameters. In other words, we can say that the value of the variable is used in the function call in the call by value method.

•In call by value method, we can not modify the value of the actual parameter by the formal parameter.

•In call by value, different memory is allocated for actual and formal parameters since the value of the actual parameter is copied into the formal parameter.

•The actual parameter is the argument which is used in the function call whereas formal parameter is the argument which is used in the function definition.

Call by reference in C

•In call by reference, the address of the variable is passed into the function call as the actual parameter.

•The value of the actual parameters can be modified by changing the formal parameters since the address of the actual parameters is passed.

•In call by reference, the memory allocation is similar for both formal parameters and actual parameters. All the operations in the function are performed on the value stored at the address of the actual parameters, and the modified value gets stored at the same address.