C LANGUAGE

1 Types of Programming Languages

The programming languages can be classified into two types.

A. Low level languages
B. High level languages

1.1 Low Level Languages

A. Machine level language
B. Assembly language

1.1.1 Machine Level Language

- The machine language contains only two symbols 1 & 0.
- All the instructions of machine language are written in the form of binary numbers 1's & 0's, because computer can understand only digital signals.

1.1.2 Assembly Language

- In assembly language it uses mnemonics which are in English words such as MOV, ADD, SUB…etc. to represent the operations that a processor has to do.
- This is an intermediate language between high-level languages like C and the binary language.
- It uses hexadecimal and binary values, and it is readable by humans.

1.2 High Level Languages

- High level languages are programming languages which are used for writing programs or software which could be understood by the humans and computer.
- These languages are easier to understand for humans because it uses lot of symbols letters phrases to represent logic and instructions in a program.
- It contains high level of abstraction compared to low level languages.

  Ex: FORTAN, COBAL, BASIC, Pascal, Python, Java…. etc.

2 Difference b/w Compiler, Interpreter and Assembler

- **Assembler** is used for converting the code of assembly language into machine level language.
- **Compilers** and **interpreters** are used to convert the code of high-level language into machine language. Although both compilers and interpreters perform the same task but there is a difference in their working.
- A **compiler** searches all the errors of program and lists them. If the program is error free then it converts the code of program into machine code and then the program can be executed by separate commands.

  Ex: C, C++, C#, Scala.

- An **interpreter** checks the errors of program statement by statement. After checking one statement, it converts that statement into machine code and then executes that statement. This process continues until the last statement of program or an erroneous statement occurs.

  Ex: Ruby, Perl, Python

COMPARISON	COMPILER	INTERPRETER
Input	It takes an entire program at a time.	It takes a single line of code or instruction at a time.
Output	It generates intermediate object code.	It does not produce any intermediate object code.
Working mechanism	The compilation is done before execution.	Compilation and execution take place simultaneously.
Speed	Comparatively faster	Slower
Memory	Memory requirement is more due to the creation of object code.	It requires less memory as it does not create intermediate object code.
Errors	Display all errors after compilation, all at the same time.	Displays error of each line one by one.
Error detection	Difficult	Easier comparatively
Pertaining Programming languages	C, C++, C#, Scala, typescript uses compiler.	PHP, Perl, Python, Ruby uses an interpreter.

3 Introduction to C Programming Language

- C was invented and first implemented by Dennis Ritchie in 1970's at bell labs.
- C has been standardized by American National Standard Institutes (ANSI) since 1989 and by the International Organization for Standardization (ISO).
- It is widely used for developing desktop applications, developing browsers and their extensions and also widely used in embedded systems.

3.1 Features of C

- C is a structured programming language in which the program is divided into various modules.
- C is a Procedural Oriented Programming Language and also it is a middle level language.
- C contains 32 keywords, various data types, and a set of powerful built-in functions that make programming very efficient.
- C89 standards are used in every compiler.

3.2 Structure of C Programs

```
Comments
Preprocessor directives
Macros
Global variables;
function declarations;
int main(void)
{
    Local variables;
    function calls ();
    Statements;
    return 0;
}
function definition ()
{
    Local variables;
    Statements;
}
```

4 How do we compile and run a C program using Ubuntu machine with GCC compiler?

A. First we create a C program using an editor (vi, vim) and save the file as filename.c

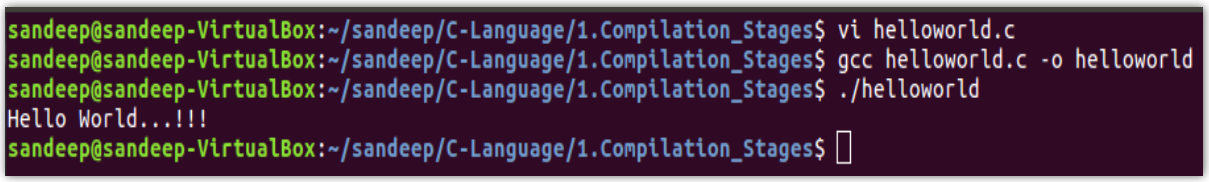
$ vi filename.c

B. We use the command in the terminal for compiling our filename.c source file. And the -o is used to specify the output file name.

$ gcc filename.c –o filename (or) $ gcc filename.c

C. After compilation executable is generated and we run the generated executable using the below command. If -o is not used in compilation use ./a.out command.

$ ./filename (or) ./a.out



5 Compilation Stages in C

A compiler converts a C program into an executable. There are four phases for a C program to become an executable:

A. Pre-processor
B. Compiler
C. Assembler
D. Linker

By executing this **$gcc -Wall -save-temps filename.c –o filename** command, we get all intermediate files in the current directory along with the executable.

```
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/1.Compilation_Stages$ gcc -Wall -save-temps helloworld.c -o helloworld
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/1.Compilation_Stages$ ls
helloworld  helloworld.c  helloworld.i  helloworld.o  helloworld.s
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/1.Compilation_Stages$
```

5.1 Pre-processor

- Removal of Comments.
- Expansion of Macros.
- Expansion of the included files (It contains macro definitions, declarations of enum, structure, union, typdef, external functions and global variables).
- Conditional compilation (#ifdef , #ifndef, #if, #else, #elif , #endif).
- Pre-processed file (.i).

In the filename.i, you can find the preprocessed output.

5.2 Compiler

- The compiler translates the preprocessed source code into assembly language (.s) files.

In the filename.s, you can find the compiled output file.

5.3 Assembler

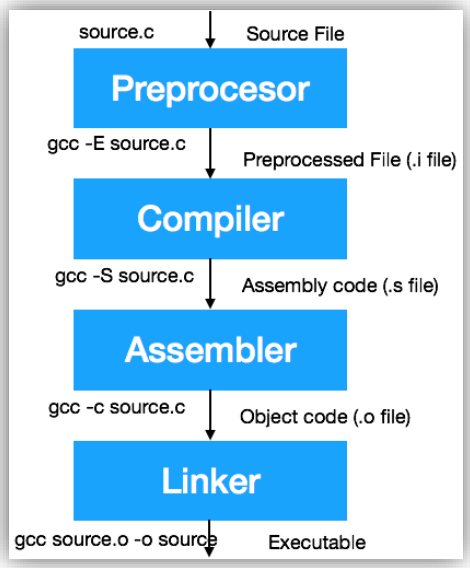
- The assembler translates the assembly language code into machine code (.o, .obj) files.
- At this phase, only existing code is converted into machine language, and the function calls like **printf()** are not resolved.

In the filename.o, you can find the machine-level instructions.

5.4 Linker

- Linking is a process of including the library files into our program.
- Library files are some predefined files that contain the definition of the functions in the machine language and these files have an extension of (.lib, .a static library) files.
- The linking process generates an executable file with an **extension of .exe in DOS and .out in UNIX OS**.
- In linker stage it adds some extra code to our program, it can be verified by using the command **$size filename.o and $size filename**. We will know that how the output file increases from an object file to an executable file.

```
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/1.Compilation_Stages$ size helloworld.o
   text    data     bss     dec     hex filename
     95       0       0      95      5f helloworld.o
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/1.Compilation_Stages$ size helloworld
   text    data     bss     dec     hex filename
   1188     552       8    1748     6d4 helloworld
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/1.Compilation_Stages$
```



6 What are Standard Library Functions in C?

- C Standard library functions or simply C Library functions are inbuilt functions in C programming.
- The prototype and data definitions of these functions are present in their respective header files.
- To use these functions we need to include the header file in our program. For example, If you want to use the printf() function, the header file <stdio.h> should be included.

7 Header Files in C

- In C language, header files contain a set of predefined standard library functions.
- The .h is the extension of the header files in C and we request to use a header file in our program by including it with the C preprocessing directive "#include".
- C Header files offer the features like library functions, data types, macros, etc by importing them into the program with the help of a preprocessor directive "#include".
- There are two types of header files:
    A. Standard / Pre-existing header files in system directory
        #include <filename.h>
    B. Non-standard / User defined header file in source file directory
        #include "filename.h"
- There are 31 standard header files in the latest version of C language. Below the list of some commonly used header files in C:

Header File	Description
<stdio.h>	It is used to perform input and output operations using functions like scanf(), printf(), etc.
<errorno.h>	It is used to perform error handling operations like errno(), strerror(), perror(), etc.
<math.h>	It is used to perform mathematical operations like sqrt(), log2(), pow(), etc.
<string.h>	It is used to perform various functionalities related to string

	manipulation like strlen(), strcmp(), strcpy(), size(), etc.
stdlib.h	This header file contains functions for dynamic memory allocation, such as malloc() and free(), and various other general-purpose functions, such as exit() and system().

Note

In stdio.h, stdlib.h contains the declarations for these functions, while their definitions are part of the C standard library and are linked to your program during the linking phase.

8 Keywords

- Keywords are predefined or reserved words that have special meanings to the compiler.
- These are part of the syntax and cannot be used as identifiers in the program.
- They are always written in lower case.
- There are only 32 keywords available in C which is given below.

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

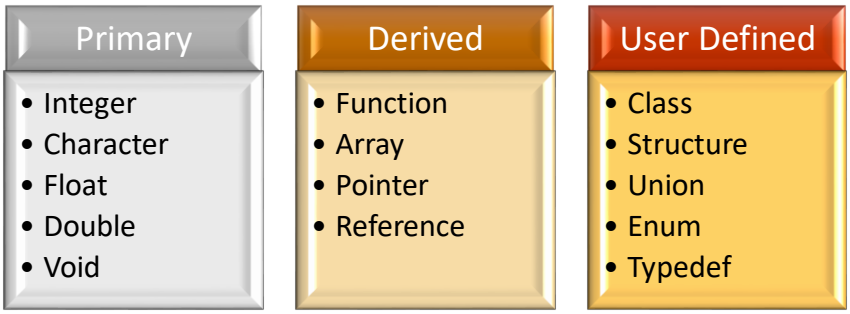
9 Identifiers

- In C, the names of variables, functions, labels, and various other user-defined items are called identifiers.
- The length of these identifiers can vary from one to several characters. Rules for naming identifiers are given below.
    A. The name should consist of only uppercase and lowercase letters, digits and underscore sign (_).
    B. The first character must be an alphabet or underscore (_).
    C. The name should not be a keyword.
    D. Since C is case sensitive, the uppercase and lowercase letters are considered different. Like code, Code and CODE are three different identifiers.

Correct	Incorrect
count test23 high_balance	1count hi!there high . . . balanc

10 Data Types

- Each variable in C has an associated data type. Each data type requires different amounts of memory and has some specific operations which can be performed over it.
- It specifies the type of data that the variable can store like integer, character, floating, double, etc. The data type is a collection of data with values having fixed values, meaning as well as its characteristics.
- The data types in C can be classified as follows:
    A. Primitive data types.
    B. User defined data types.
    C. Derived types.



10.1 Primary Data Types

BASIC DATA TYPES	DATA TYPES WITH TYPE QUALIFIERS	SIZE IN BYTES	RANGE	FORMAT SPECIFIERS
CHAR	Signed Char	1	-128 to 127	%c
	Unsigned Char	1	0 to 255	%c
INT	Int or Signed Int	4	-2,147,483,648 to 2,147,483,647	%d
	Unsigned Int	4	0 to 4,294,967,295	%u
	Short Int or Signed Short Int	2	-32,768 to 32,767	%d
	Unsigned Short Int	2	0 to 65,535	%d or %hu
	Long Int or Signed Long Int	4	–2,147,483,647 to 2,147,483,647	%ld
	Unsigned Long Int	4	0 to 4,294,967,295	%lu
FLOAT	Float	4	1E–37 to 1E+37 with six digits of precision	%f
DOUBLE	Double	8	1E–37 to 1E+37 with ten digits of precision	%lf
	Long Double	16	3.4E-4932 to 1.1E+4932 with ten digits of precision	%Lf

Note

These ranges may vary from compiler to compiler. Below is a list of ranges along with the memory requirement and format specifiers on the 32-bit GCC compiler.

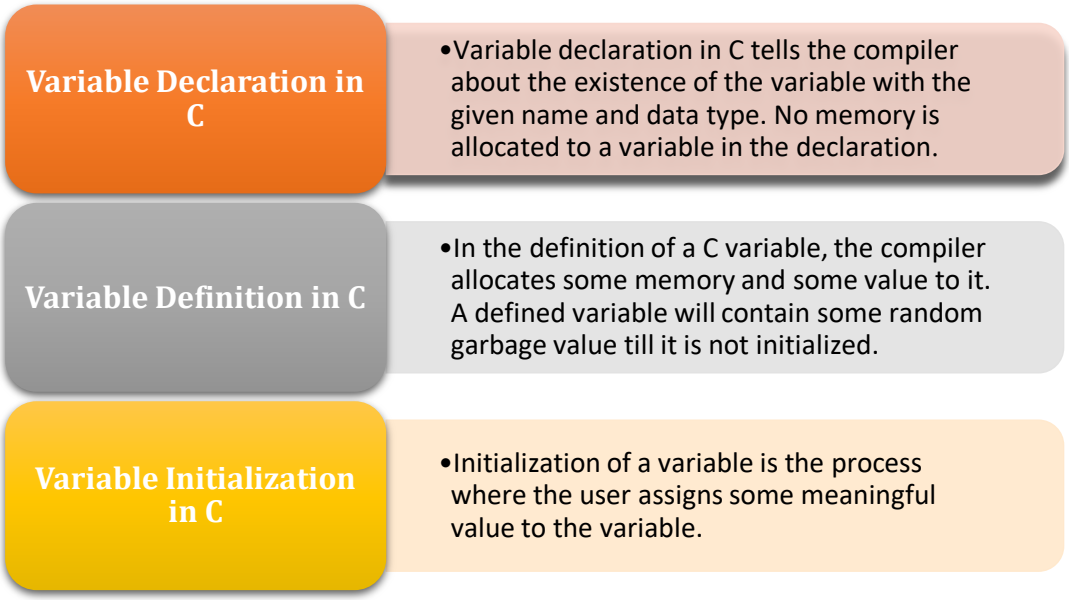
11 Variables

- Variables are used in C programming to store data values.
- Every variable in C has a specific data type, which determines the size and layout of the memory allocated to the variable.
- The range of values that can be stored in the variable, and the set of operations that can be performed on the variable.

Syntax

```
<Data Type> < Variable Name> = Value;                    // defining single variable
                          Or
<Data Type> <Variable Name1, Variable Name2>;     // defining multiple   variable
```

- There are 3 aspects of defining a variable
  A. Variable Declaration
  B. Variable Definition
  C. Variable Initialization



11.1 Local Variables

- A Local variable in C is a variable that is declared inside a function or a block of code. Its scope is limited to the block or function in which it is declared.

11.2 Global Variables

- A Global variable in C is a variable that is declared outside the function or a block of code. Its scope is the whole program i.e. we can access the global variable anywhere in the C program after it is declared.

Example

```
#include<stdio.h>
int x = 20;                          //Global Variable
int main( ) {
        int y = 30;                  //Local Variable
        printf("%d  is the global variable\n",x);
```

```
printf("%d  is the local variable",y);
return 0;
    }
```

12 Scope & Life Time

12.1 Scope

- The scope of a variable defines the region within which it can be used. Once outside of this region, the variable is inaccessible. In C, variables have three scopes:
    A. **Block Scope:** Variables are accessible only within the block where they are declared.
    B. **Function Scope:** Labels are accessible throughout the function they are declared in.
    C. **File Scope:** Variables and functions are accessible throughout the file from their point of declaration.
- Additionally, there's a concept of **local variables** and **global variables**:
    A. The scope of a variable is confined to the code block or function in which it is declared.
    B. The global scope encompasses the entire program, allowing global variables to be accessed from any part of it.

12.2 Life Time

- The lifetime of a variable refers to the duration during which the variable exists in memory and retains its value.
- The lifetime of a variable depends on its storage class and scope.

13 Memory Layout

- A typical memory representation of a C program consists of the following sections.
    A. Text segment.
    B. Initialized data segment.
    C. Uninitialized data segment.
    D. Stack.
    E. Heap.

13.1 Text Segment

- The text segment is also known as the code segment. When we compile any program, it creates an executable file like a.out, .exe, etc., that gets stored in the text or code section of the RAM memory.
- Text segment has read-only permission that prevents the program from accidental modifications.
- Text segment in RAM is shareable so that a single copy is required in the memory for frequent applications like a text editor, C compiler, shells…. etc.

13.2 Initialized Data Segment / Data Segment

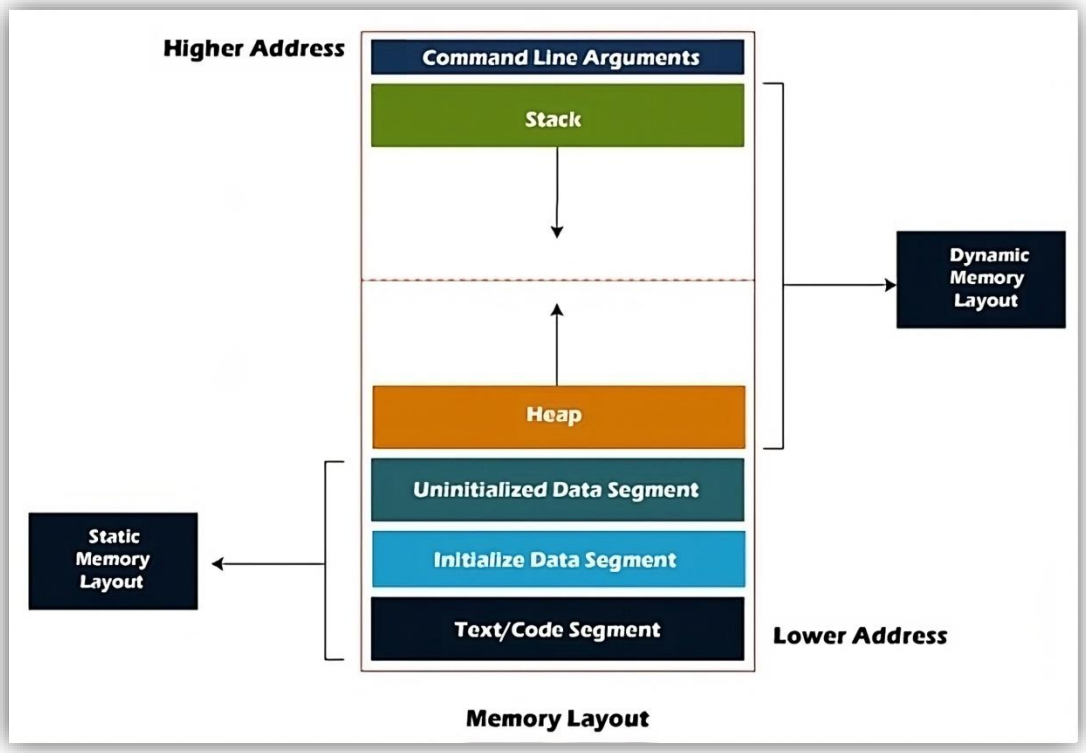
- A data segment is a portion of the virtual address space of a program, which contains the values of all external, global, static, and constant variables that are initialized by the programmer.

- The values of variables in this segment are not read-only, i.e., they can be modified at run time.
- This data segment can be further classified into categories:
    A. **Initialized read-only area:** It is an area where the values of variables cannot be modified.
    B. **Initialized read-write area:** It is an area where the values of variables can also be altered.

Example

```
#include<stdio.h>
int global_var = 50;
const int global_var2 = 30;
int main () {
        static int a = 10;
        return 0;
    }
```

- The variable **global_var** is declared outside the scope of the main () function, so it is stored in the **read-write (.data)** part of the **initialized data segment**.
- The global variable **global_var2**, declared with the const keyword, is stored in the **read-only (.rodata)** section of the **initialized data segment**.
- The static variable is stored in the **read-write** part of the **initialized data segment**, but since it is a local static variable, its lifetime extends for the entire duration of the program.



13.3 Uninitialized Data Segment

- An uninitialized data segment is also known as **"Block Started by Symbol (BSS)"**.
- The kernel initializes every data in the BSS segment to arithmetic 0 and sets pointers to the null pointer before the C program executes.

- The BSS segment also includes all static and global variables initialized with the arithmetic value of 0. Since the values of variables stored in BSS can be changed, this data segment has **read-write** permissions.

Example

```
#include <stdio.h>
//Global variable
int global_variable;
int main () {
        //Static varibles
        static int static_variable;
        //Print global variable
        printf("global_variable = %d\n", global_variable);
        //Print static variable
        printf("static_variable = %d\n", static_variable);
        return 0;
    }
```

  - In the above example, both the variables **"global_variable"** and **"static_variable"** are uninitialized.

13.4 Stack Segment

- The stack segment is used to store local variables, function parameters, return addresses, and other function call-related information during program execution.
- It follows the **"Last-In-First-Out (LIFO)"** structure and grows toward lower memory addresses, although this direction depends on the computer architecture. Additionally, it grows in the opposite direction to the heap.
- Stack pointer register keeps track of the top of the stack and its value change when push/pop actions are performed on the segment. The values are passed to stack when a function is called stack frame.
- Stack frame stores the value of function temporary variables and some automatic variables that store extra information like the return address and details of the caller's environment (memory registers).
- Each time function calls itself recursively, a new stack frame is created, which allows a set of variables of one stack frame to not interfere with other variables of a different instance of the function.

Example

```
#include<stdio.h>
void foo() {
        int a, b;
    }
int main() {
        int local = 5;
        char name[26];
        foo();
        return 0;
    }
```

  - In the above example, the first "main()" starts its execution, and a stack frame for main() is made and pushed into the program stack with data of variables local and name.
  - Then in main, we call foo, then another stack frame is made and pushed for it separately, which contains data of variables a and b.

- After the execution of foo, its stack frame is popped out, and its variable gets unallocated, and when the program ends, main's stack frame also gets popped out.

13.5 Heap Segment

- The heap is used for dynamic memory allocation.
- Memory allocation in Heap segment is done using malloc, calloc , realloc and free functions. It's typically located between the data segment and the stack.

14 Storage Classes in C

- C Storage Classes are used to describe the features of a variable/function. These features basically include the scope, visibility, and lifetime which help us to trace the existence of a particular variable during the runtime of a program.
- There are four types of storage classes in C:
    A. Auto
    B. Extern
    C. Static
    D. Register

14.1 Automatic (Auto)

- The variables declared within a function without any storage class specifier are by default automatic. The keyword **"auto"** can be used while declaring the variables.
- They are created when a function is called and destroyed when the function is exited. Their scope is limited to the block in which they are defined.

Example

```
void func() {
        auto int x;
            int y;
}
```

14.2 Static

- Static variables declared with the **"static"** keyword and retain their value between function calls.
- They are initialized only once, at program startup, and retain their value until the program terminates.
- These variables have a local scope within the block in which they are defined, but their lifetime extends throughout the program.

Example

```
void func() {
        static int y;
}
```

14.3 Extern

- The **"extern"** keyword is used to declare a variable that is defined in another file of the program or in another compilation unit.
- It is typically used to declare global variables that are defined elsewhere.

Example

```
// File1.c
        int globalVar = 10;
// File2.c
        extern int globalVar;
```

14.4 Register

- In C programming, the **"register"** storage class is used to declare variables that are intended for frequent access and are typically stored in CPU registers for faster access during runtime.
- If a free register is available, the compiler assigns the variable to it; otherwise, it's stored in memory. However, one cannot obtain the address of a register variable using pointers.
- This feature enhances program efficiency by minimizing memory access for frequently used variables.
- In C99 and later versions, the **"register"** keyword has been deprecated because compilers optimize register usage automatically.

Example

```
void func() {
        register int z;
    }
```

Example

```
/***************C program to illustrate the static & auto variables*****************/
#include <stdio.h>
void function(void);
int main() {
        function();
        function();
        function();
        return 0;
    }
void function() {
            auto int a = 0;
            static int b = 0;
            a++;
            b++;
            printf("Auto variable a = %d\t", a);
            printf("Static variable b = %d\n", b);
     }
```

Output

```
Auto variable a = 1    Static variable b = 1
Auto variable a = 1    Static variable b = 2
Auto variable a = 1    Static variable b = 3
```

Note:

Both normal global and static global variables exist throughout the entire program's execution. However, normal global variables have external linkage, making them

accessible from other source files, while static global variables have internal linkage, restricting their accessibility to the file in which they are declared.

Storage Specifier	Storage	Initial Value	Scope	Where	Life
Auto	Stack	Garbage	End of block	Local	End of block
Extern	Data Segment	Zero	End of file & extendable	Global	Till end of program
Static	Data Segment	Zero	End of block & End of file but not extendable	Local & Global	Till end of program
Register	CPU registers	Garbage	End of block	Global	End of block

15 Operators

- In C programming, operators are symbols that perform operations on operands. Some operators require two operands, while others act upon only one operand. Here's an overview of the main types of operators in C:
    A. Arithmetic Operators.
    B. Assignment Operators.
    C. Increment and Decrement Operators.
    D. Relational Operators.
    E. Logical Operators.
    F. Conditional Operator.
    G. Comma Operator.
    H. Size of Operator.
    I. Bitwise Operators.
    J. Other Operators.

15.1 Arithmetic Operators

- Arithmetic operators are used for numeric calculations. They are of two types –
    A. Unary arithmetic operators
    B. Binary arithmetic operators

15.1.1 Unary Arithmetic Operators

- Unary operators require only one operand.

    Example: - **+x.** **-y.**

15.1.2 Binary Arithmetic Operators

- Binary operators require two operands. There are five binary arithmetic operators.
    a. '+' Addition.
    b. '–'Subtraction.

c. '*'Multiplication.

d. '/' Division.

e. '%' Modulus (Returns the remainder in division).

15.2 Assignment Operators

- Assignment operators are used to assign value to a variable. The left side operand of the assignment operator is a variable and the right side operand of the assignment operator is a value.
- The value on the right side must be of the same data type as the variable on the left side otherwise the compiler will raise an error.
- The assignment operators can be combined with some other operators in C to provide multiple operations using single operator. These operators are called compound operators.

Symbol	Operator	Syntax
=	Simple Assignment	a = b
+=	Plus and assign	a += b
-=	Minus and assign	a -= b
*=	Multiply and assign	a *= b
/=	Divide and assign	a /= b
%=	Modulus and assign	a %= b

15.3 Increment and Decrement Operators

- C has two useful operators increment (++) and decrement (- -). These are unary operators because they operate on a single operand.
- The increment operator (++) increments the value of the variable by 1 and decrement operator (- -) decrements the value of the variable by 1.
- These operators should be used only with variables; they can't be used with constants or expressions. For example the expressions ++5 or ++(x+y+z) are invalid.
- These operators are of two types:-

A. Prefix increment / decrement - operator is written before the operand (e.g. ++x or - -x).

B. Postfix increment / decrement - operator is written after the operand (e.g. x++ or x - -).

15.3.1 Prefix Increment / Decrement

- In prefix increment / decrement the value of variable is incremented / decremented then the new value is used in the operation.

Example

```
y = ++x;          //prefix increment
  x = x + 1;
```

```
        y = x;
Y = --x                 //prefix decrement
        x = x - 1;
        y = x;
```

15.3.2 Postfix Increment / Decrement

- In postfix increment / decrement the value of variable is used in the operation and then increment/decrement is performed.

Example:

```
y = x++;                //postfix increment
        y = x;
        x = x+1;
y = x--;                //postfix decrement
        y = x;
        x = x-1;
```

15.4 Relational Operators

- Relational operators are used to compare values of two expressions depending on their relations.

OPERATOR	MEANING
<	Less than
>	Greater than
= =	Equal to
<=	Less than or equal to
>=	Greater than or equal to
!=	Not equal to

15.5 Logical or Boolean Operators

- An expression that combines two or more expressions is termed as a logical expression. For combining these expressions we use logical operators.
- These operators return 0 for false and 1 for true. The operands may be constants, variables or expressions. C has three logical operators:

OPERATOR	MEANING
&&	AND
||	OR
!	NOT

15.5.1 AND Operator (&&)

- This operator gives the net result true, if both the conditions are true, otherwise the result is false.

CONDITION1	CONDITION2	RESULT
True	True	True
True	False	False
False	True	False
False	False	False

15.5.2 OR Operator (||)

- This operator gives the net result false, if both the conditions have the value false, otherwise the result is true.

CONDITION1	CONDITION2	RESULT
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

15.5.3 NOT Operator (!)

- This is a unary operator and it negates the value of the condition. If the value of the condition is false then it gives the result true. If the value of the condition is true then it gives the result false.

CONDITION	RESULT
TRUE	FALSE
FALSE	TRUE

15.6 Conditional Operator

- The conditional operator in C is kind of similar to the if-else statement as it follows the same algorithm as of if-else statement but the conditional operator takes less space and helps to write the if-else statements in the shortest way possible.
- It is also known as the ternary operator in C as it operates on three operands.

Syntax

```
Variable = (condition)? Expression2: Expression3;
```

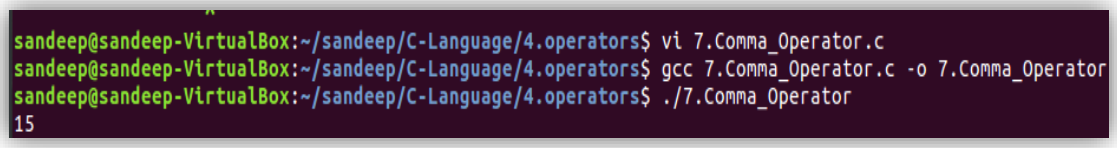
15.7 Comma Operator

- The comma operator evaluates the first operand and discards the result, then evaluates the second operand and returns the result.
- The comma operator has the lowest precedence of any C operator.
- Comma acts as both operator and separator.

Example

```
/************comma as operator***************/
                #include <stdio.h>
                int main() {
                                int x = 10;
                                int y = 15;
                                printf("%d", (x, y));
                                return 0;
                        }
```

Output



15.8 Sizeof operator

- Sizeof is a compile-time unary operator which can be used to compute the size of its operand in terms of bytes.
- It can be applied to any data type, including primitive types such as integer and floating-point types, pointer types, or compound data types such as Structure, union, etc.

Syntax

```
sizeof(Expression);
```

Example

```
/**************C program to illustrate the sizeof operator****************/
        #include <stdio.h>
        int main() {
                        printf("%lu\n", sizeof(char));
                        printf("%lu\n", sizeof(int));
                        printf("%lu\n", sizeof(float));
                        printf("%lu", sizeof(double));
                        return 0;
                }
```

Output

```
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/4.operators$ vi 8.Size_Of_Operator.c
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/4.operators$ gcc 8.Size_Of_Operator.c -o 8.Size_Of_Operator
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/4.operators$ ./8.Size_Of_Operator
1
4
4
8
```

Note

- The output can be different on different machines like a 32-bit system can show different output while a 64-bit system can show different of same data types.

15.9 Bitwise Operators

- C has the ability to support the manipulation of data at the bit level. Bitwise operators are used for operations on individual bits. Bitwise operators operate on integers only. The bitwise operators are as:

BITWISE OPERATOR	MEANING
&	Bitwise AND
|	Bitwise OR
~	One's Complement
<<	Left Shift
>>	Right Shift
^	Bitwise XOR

Example

```
/**************C program to illustrate the bitwise operators****************/
#include <stdio.h>
int main() {
        int a = 25, b = 5;
         printf("a & b: %d\n", a & b);
        printf("a | b: %d\n", a | b);
        printf("a ^ b: %d\n", a ^ b);
        printf("~a: %d\n", ~a);
        printf("a >> b: %d\n", a >> b);
        printf("a << b: %d\n", a << b);
        return 0;
}
```

Output

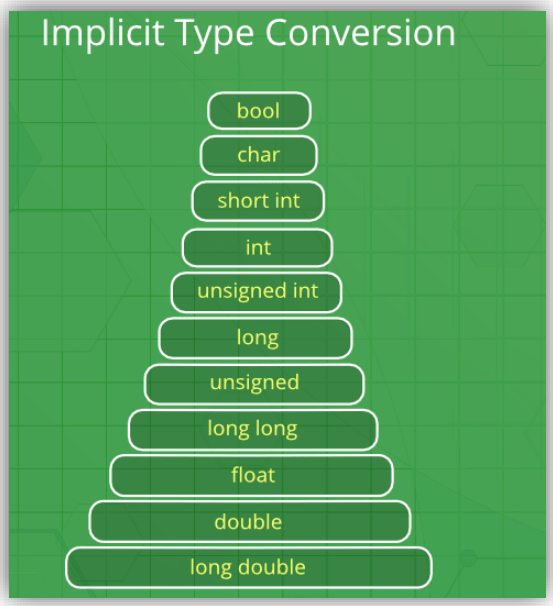
```
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/4.operators$ vi 9.Bitwise_Operators.c
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/4.operators$ gcc 9.Bitwise_Operators.c -o 9.Bitwise_Operators
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/4.operators$ ./9.Bitwise_Operators
a&b:1
a|b:29
a^b:28
~a:-26
a>>b:0
a<<b:800
```

15.10 Type Conversion

- Type conversion, also known as type casting, is the process of converting a value from one data type to another in C programming. There are two main types of type conversion:
    A. Implicit Type Conversion
    B. Explicit Type Conversion

15.10.1 Implicit Type Conversion

- It is also known as automatic type conversion, it occurs automatically when the compiler handles assignments between compatible data types without the need for any explicit instructions from the programmer. It typically occurs in these scenarios:
    - Assigning a value of one data type to a variable of another data type, where the destination type can accommodate the source type without loss of data.
    - When performing arithmetic operations involving different data types, C promotes smaller data types to larger data types before performing the operation to prevent loss of data.



Example

```
/**************C program to illustrate the implicit type conversion****************/

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

                                int a = 10;
```

```
        float b = 4.5;
        float result ;
        result = a+b;
        printf("result = %f",result);
        return 0;
}
```

15.10.2 Explicit Type Conversion

- Also known as type casting, it involves the programmer explicitly specifying the data type to which a value should be converted.
- This is done by placing the desired data type in parentheses before the value or variable to be converted.

Example:

```
/**************C program to illustrate the explicit type conversion***************/

#include <stdio.h>
int main() {
        int a = 10;
        float b = 4.5;
        int result ;
        result = (int)(a+b);
        printf("result = %d",result);
        return 0;
}
```

Output

```
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/4.operators$ vi 10.Type_C0nversion.c
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/4.operators$ gcc 10.Type_C0nversion.c -o 10.Type_Conversion
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/4.operators$ ./10.Type_Conversion
The result of implicit type conversion is:14.500000
The result of explicit type conversion is:14
```

15.11 Operator Precedence and Associativity Table

Operator	*Description*	*Associativity*
() **[]** **.** **->** **++ --**	Parentheses or function call Brackets or array subscript Dot or Member selection operator Arrow operator Postfix increment/decrement	left to right
++ -- **+ -** **! ~** **(type)** ***** **&** **sizeof**	Prefix increment/decrement Unary plus and minus not operator and bitwise complement type cast Indirection or dereference operator Address of operator Determine size in bytes	right to left
*** / %**	Multiplication, division and modulus	left to right
+ -	Addition and subtraction	left to right
<< >>	Bitwise left shift and right shift	left to right
< <= **> >=**	relational less than/less than equal to relational greater than/greater than or equal to	left to right
== !=	Relational equal to or not equal to	left to right
&&	Bitwise AND	left to right
^	Bitwise exclusive OR	left to right
|	Bitwise inclusive OR	left to right
&&	Logical AND	left to right
||	Logical OR	left to right
? :	Ternary operator	right to left
= **+= -=** ***= /=** **%= &=** **^= |=** **<<= >>=**	Assignment operator Addition/subtraction assignment Multiplication/division assignment Modulus and bitwise assignment Bitwise exclusive/inclusive OR assignment	right to left
,	comma operator	left to right

16 Types of Errors in C

- There are five different types of errors in C
  A. Syntax Error
  B. Runtime Error
  C. Logical Error
  D. Semantic Error
  E. Linker Error

16.1 Syntax Error

- Syntax errors in C occur due to mistakes or typos made by the programmer while typing the code, violating the defined rules for the language's syntax
- Syntax errors, also known as compilation errors, are detected by the compiler and are easily identifiable and rectifiable by programmers.
- The most commonly occurring syntax errors in C language are:
    a. Missing semi-colon (;).
    b. Missing parenthesis ({}).
    c. Assigning value to a variable without declaring it.

Example

```
/**************C program to illustrate the syntax error***************/

        #include <stdio.h>
        void main() {
                        var = 5;    // we did not declare the data type of variable
                        printf("The variable is: %d", var);
                }
```

Output

```
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/5.Type_Of_Errors$ vi 1.Syntax_Error.c
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/5.Type_Of_Errors$ gcc 1.Syntax_Error.c -o 1.Syntax_Error
1.Syntax_Error.c: In function ‘main’:
1.Syntax_Error.c:5:1: error: ‘var’ undeclared (first use in this function)
 var=10;
 ^
1.Syntax_Error.c:5:1: note: each undeclared identifier is reported only once for each function it appears in
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/5.Type_Of_Errors$
```

16.2 Runtime Error

- Runtime errors occur during the execution of a program after it has been compiled successfully.
- Runtime errors can be a little tricky to identify because the compiler cannot detect these errors. They can only be identified once the program is running.
- These errors are typically caused by invalid input, number not divisible by zero, array index out of bounds, string index out of bounds, uninitialized variables, etc.
- Runtime errors can occur because of various reasons. Some of the reasons are:
    a. **Mistakes in the Code:** Let us say during the execution of a while loop, the programmer forgets to enter a break statement. This will lead the program to run infinite times, hence resulting in a run time error.
    b. **Memory Leaks:** If a programmer creates an array in the heap but forgets to delete the array's data, the program might start leaking memory, resulting in a run time error.
    c. **Undefined Variables:** If a programmer forgets to define a variable in the code, the program will generate a run time error.

Example

```
/**************C program to illustrate the runtime error***************/

            #include<stdio.h>
            void main() {
                            int var = 2147483649;
                            printf("%d", var);
```

```
}
```

Output

```
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/5.Type_Of_Errors$ vi 2.Run_Time_Error.c
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/5.Type_Of_Errors$ gcc 2.Run_Time_Error.c -o 2.Run_Time_Error
2.Run_Time_Error.c: In function 'main':
2.Run_Time_Error.c:5:11: warning: overflow in implicit constant conversion [-Woverflow]
 int var = 59899979695;
           ^
```

16.3 Logical Error

- Logical errors occur when a program compiles and runs without syntax or runtime errors, yet fails to produce the expected output due to flawed logic.
- Detecting and fixing these errors require careful debugging and a deep understanding of the program's intended behavior.

Example

```
/**************C program to illustrate the runtime error****************/

        #include <stdio.h>
        void main() {
        float a = 10;
        float b = 5;
                if (b = 0) {                                    // we wrote = instead of ==
                                printf("Division by zero is not possible");
                }
                 else {
                                 printf("The output is: %f", a/b);
                        }
                          }
```

Output

```
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/5.Type_Of_Errors$ vi 3.Logical_Error.c
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/5.Type_Of_Errors$ gcc 3.Logical_Error.c -o 3.Logical_Error
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/5.Type_Of_Errors$ ./3.Logical_Error
The output is:inf
```

16.4 Semantic Error

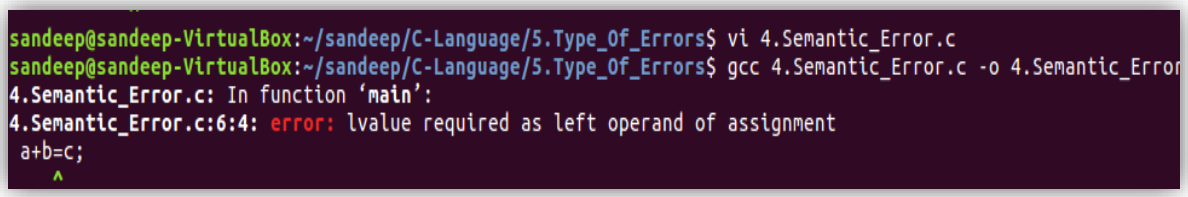
- When a sentence is syntactically correct but has no meaning, semantic errors occur. This is similar to grammatical errors.
- If an expression is entered on the left side of the assignment operator, a semantic error may occur.

Example

```
/**************C program to illustrate the semantic error****************/

        #include <stdio.h>
        int main() {
                        int x = 10,b = 20, c;
                        x + y = c;
                        printf("%d", c);
```

```
        return 0;
    }
```

Output



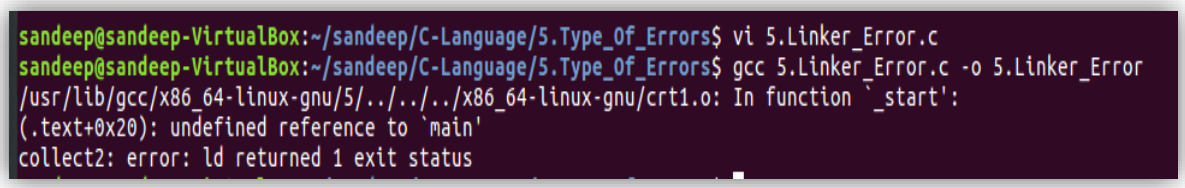
16.5 Linker Error

- Linker errors occur during the linking phase of the compilation process in C programming.
- When attempting to link object files with the main object file, errors may arise if there are issues such as incorrect function prototyping, wrong header files, or other factors.
- For instance, if "main()" is written as "Main()", a linker error will occur, preventing the generation of the executable.

Example

```
/***************C program to illustrate the linker error*****************/
    #include <stdio.h>
    void Main() {
        int var = 10;
        printf("%d", var);
    }
```

Output



17 Control Statements

- In C programming, control statements are used to guide the flow of execution of the program.
- They enable decision-making, looping, and branching, which are essential for writing dynamic and efficient programs. Here are the main types of control statements in C:
    A. Conditional Statements
    B. Looping Statements
    C. Jump Statements

17.1 Conditional Statements

- Conditional statements allow the program to make decisions and execute specific code blocks based on certain conditions. They are:

A. If statement
B. If-else statement
C. if-else-if Ladder
D. switch statement

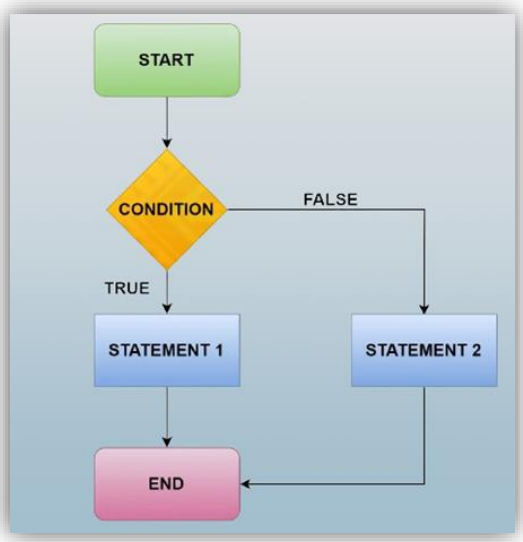
17.1.1 if Statement

- The if statement executes a block of code if a specified condition is true.

Syntax

```
if(condition)
        {
                // Statements to execute if
                // condition is true
        }
```

Block Diagram



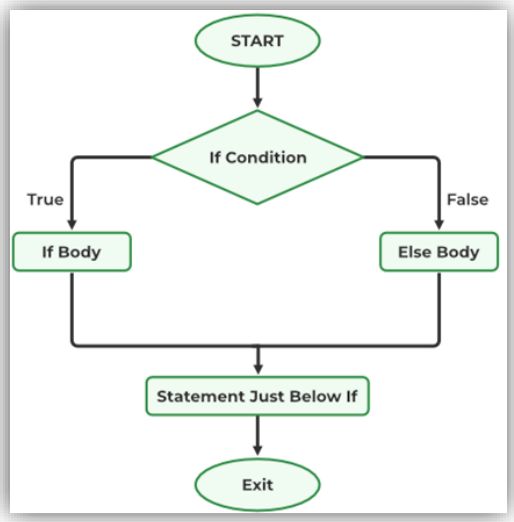
17.1.2 if-else Statement

- The if-else statement executes one block of code if the condition is true, and another block of code if the condition is false.

Syntax

```
if (condition) {
        // Code to execute if the condition is true
} else {
        // Code to execute if the condition is false
}
```

Block Diagram



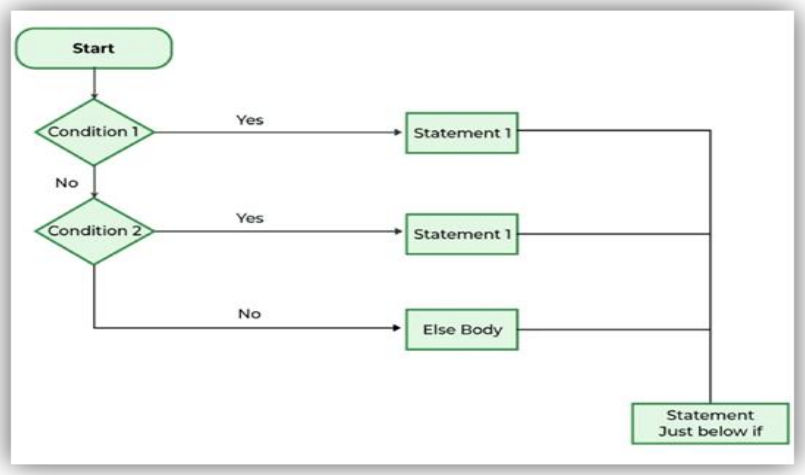
17.1.3 if-else-if ladder

- The if-else-if ladder allows for multiple conditions to be checked in sequence. The first true condition's block of code will be executed.

Syntax

```
if (condition1) {
                    // Code to execute if condition1 is true
} else if (condition2) {
                    // Code to execute if condition2 is true
} else if (condition3) {
                    // Code to execute if condition3 is true
} else {
                    // Code to execute if none of the above conditions are true
}
```

Block Diagram



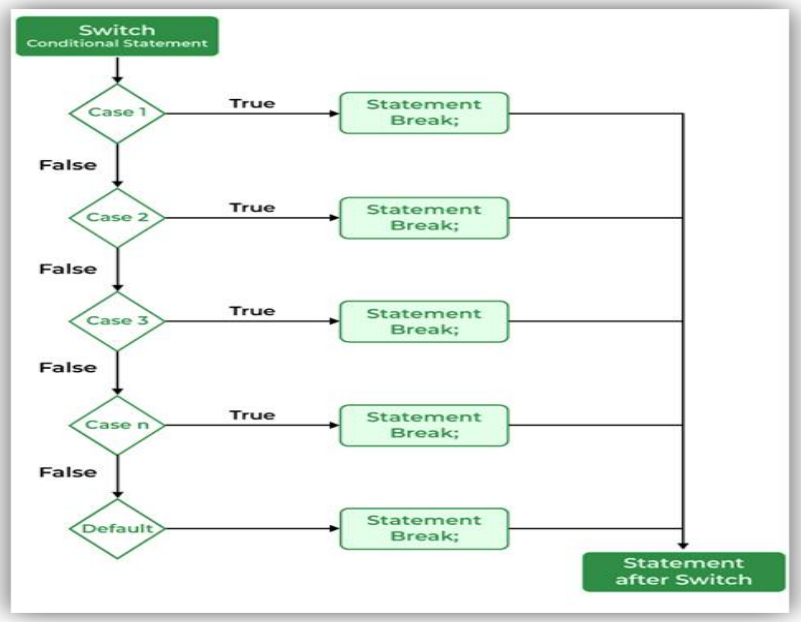
17.1.4 Switch Statement

- The switch statement is used for multiple conditional branches based on the value of an expression.

Syntax

```
switch (expression) {
        case value1:
                statements;
        case value2:
                statements;
                ....
                ....
                ....
        default:
                statements;
            }
```

Block Diagram



17.2 Looping Statements

- Looping statements repeat a block of code as long as a specified condition is true.
- There are mainly two types of loops in C Programming:
    - **Entry Controlled loops:** In Entry controlled loops the test condition is checked before entering the main body of the loop. **for Loop** and **while Loop** are Entry-controlled loops.
    - **Exit Controlled loops:** In Exit controlled loops the test condition is evaluated at the end of the loop body. The loop body will execute at least once, irrespective of whether the condition is true or false. **do-while Loop** is Exit Controlled loop.

17.2.1 for Loop

- The for loop is typically used when the number of iterations is known beforehand. It consists of three parts: initialization, condition, and increment/decrement.

Example

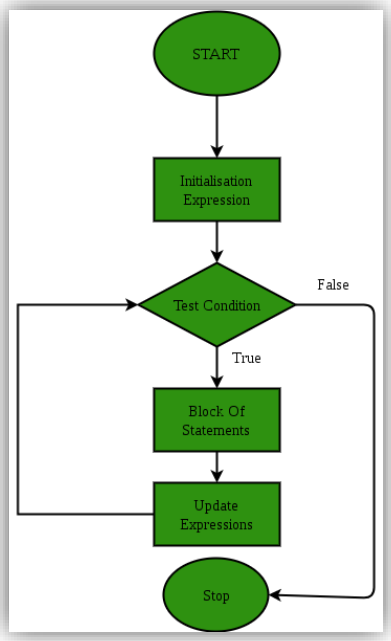
```
/**************C program to illustrate the for loop***************/

    #include <stdio.h>
```

```
int main() {
        for (int i = 0; i < 5; i++) {
                printf("Iteration %d\n", i);
                }
        return 0;
    }
```

Output

```
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/6.Control_Statements/2.Looping_Statements$ vi 1.for_Loop.c
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/6.Control_Statements/2.Looping_Statements$ gcc 1.for_Loop.c -o 1.for_Loop
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/6.Control_Statements/2.Looping_Statements$ ./1.for_Loop
i is 0
i is 1
i is 2
i is 3
i is 4
```

Block Diagram



17.2.2 while Loop

- The while loop is used when the number of iterations is not known beforehand. The loop continues as long as the condition is true.

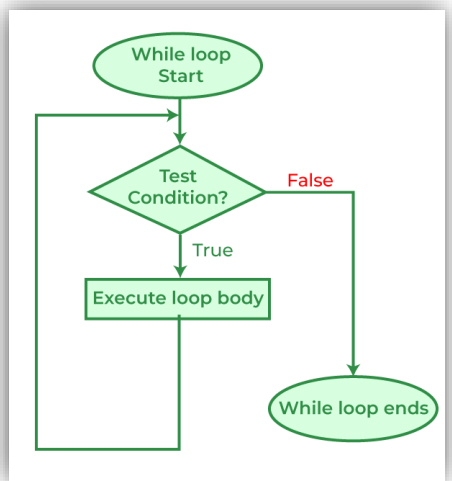
Example

```
/**************C program to illustrate the while loop****************/

    #include <stdio.h>
    int main() {
            int i = 0;
            while (i < 5) {
                    printf("Iteration %d\n", i);
                    i++;
                }
        return 0;
        }
```

Output

```
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/6.Control_Statements/2.Looping_Statements$ vi 2.while_Loop.c
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/6.Control_Statements/2.Looping_Statements$ gcc 2.while_Loop.c -o 2.while_Loop
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/6.Control_Statements/2.Looping_Statements$ ./2.while_Loop
i is 0
i is 1
i is 2
i is 3
i is 4
```

Block Diagram



17.2.3 do-while Loop

- The do-while loop ensures that the loop body is executed at least once before the condition is tested. If the condition is true, the loop body executes again.

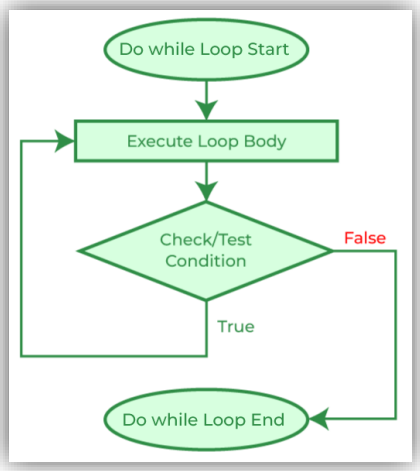
Example

```
/**************C program to illustrate the do - while loop****************/
        #include <stdio.h>
        int main() {
                        int i = 0;
                                do {
                                        printf("Iteration %d\n", i);
                                         i++;
                                  } while (i < 5);
    return 0;
                }
```

Output

```
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/6.Control_Statements/2.Looping_Statements$ vi 3.do_while_Loop.c
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/6.Control_Statements/2.Looping_Statements$ gcc 3.do_while_Loop.c -o 3.do_while_Loop
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/6.Control_Statements/2.Looping_Statements$ ./3.do_while_Loop
i is 0
i is 1
i is 2
i is 3
i is 4
```

Block Diagram



17.3 Jump Statements

- Jump statements are used to alter the flow of execution by transferring control to another part of the program.

17.3.1 break Statement

- The break statement is used to exit from a loop (for, while, or do-while) or to terminate a switch statement.
- It causes the program to jump to the statement immediately following the loop or switch.

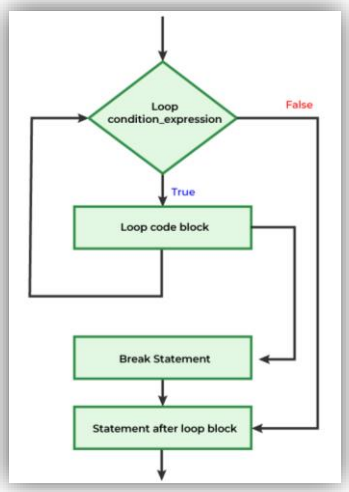
Example

```
/***************C program to illustrate the break statement*****************/
                #include <stdio.h>
                int main() {
                        for (int i = 0; i < 10; i++) {
                                if (i == 7) {
                                                break; // Exit the loop when i equals 5
                                        }
                                        printf("%d\n", i);
                                                }
                        return 0;
                          }
```

Output

```
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/6.Control_Statements/3.Jump_Statements$ vi 1.break_Statement.c
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/6.Control_Statements/3.Jump_Statements$ gcc 1.break_Statement.c -o 1.break_Statement
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/6.Control_Statements/3.Jump_Statements$ ./1.break_Statement
0
1
2
3
4
5
6
```

Block Diagram



17.3.2 continue Statement

- The continue statement skips the remaining code inside the current iteration of the loop and proceeds with the next iteration.

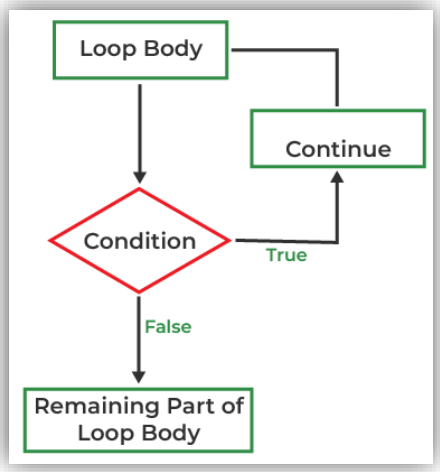
Example

```
/**************C program to illustrate the continue statement*****************/
   #include <stdio.h>
   int main() {
          for (int i = 0; i < 10; i++) {
                 if (i % 2 == 0) {
                        continue; // Skip the current iteration if i is even
                    }
          printf("%d\n", i);
                     }
                 return 0;
              }
```

Output

```
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/6.Control_Statements/3.Jump_Statements$ vi 2.continue_Statement.c
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/6.Control_Statements/3.Jump_Statements$ gcc 2.continue_Statement.c -o 2.continue_Statement
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/6.Control_Statements/3.Jump_Statements$ ./2.continue_Statement
1
3
5
7
9
```

Block Diagram



17.3.3 goto Statement

- The goto statement transfers control to the labeled statement within the same function.

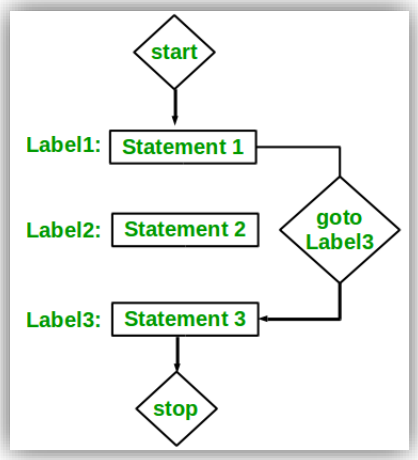
Example

```
/**************C program to illustrate the goto statement****************/
   #include <stdio.h>
   int main() {
    int i = 0;
          start:
                if (i < 5) {
                            printf("%d\n", i);
                             i++;
                            goto start;
                      }
                return 0;
           }
```

Output

```
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/6.Control_Statements/3.Jump_Statements$ vi 3.goto_Statement.c
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/6.Control_Statements/3.Jump_Statements$ gcc 3.goto_Statement.c -o 3.goto_Statement
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/6.Control_Statements/3.Jump_Statements$ ./3.goto_Statement
0
1
2
3
4
```

Block Diagram



18 Functions

- In C programming, functions are blocks of code that perform a specific task. They allow for modular programming, code reuse, and better organization of code. Here's an overview of functions in C:

18.1 Function Definition

- The function definition consists of actual statements which are executed when the function is called (i.e. when the program control comes to the function).

Syntax

```
return_type function_name (para1_type para1_name, para2_type para2_name)
{
                    // function body
}
```

  - **return_type:** The type of data the function returns. If the function does not return a value, the return type is void.
  - **function_name:** The name of the function, which should be descriptive of what the function does.
  - **parameters:** A list of variables passed to the function, enclosed in parentheses. If no parameters are needed, the parentheses are empty.

18.2 Function Declaration

- Function declarations (or prototypes) inform the compiler about the function's name, return type, and parameters. This is typically placed before the main function or in a header file.

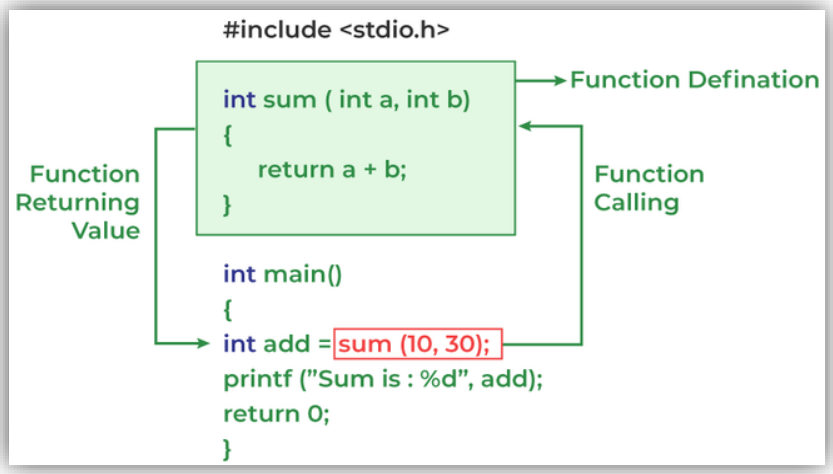
Syntax

```
return_type name_of_the_function (parameter_1, parameter_2);
```

18.3 Function Call

- A function call is a statement that instructs the compiler to execute the function. We use the function name and parameters in the function call.

Block Diagram



Example

```
#include <stdio.h>
int add(int a, int b);                    // Function declaration
int main() {
        int sum = add(5, 3);              //Function call
        printf("Sum: %d\n", sum);
        return 0;
    }
int add(int a, int b) {                   // Function definition
        return a + b;
    }
```

Output

```
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/7.Functions$ vi 1.Basic_Program.c
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/7.Functions$ gcc 1.Basic_Program.c -o 1.Basic_Program
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/7.Functions$ ./1.Basic_Program
sum:8
```

19 Arrays

- Array is a collection of homogeneous elements, stored in contiguous memory locations.

Syntax

```
<data type> varname[<no of elements>];
```

19.1 Accessing Elements

- Each individual element will get a unique identification number called index. Indexes will always start with 0.

```
int first = numbers[0];  // Accesses the first element
numbers[2] = 10;         // Sets the third element to 10
```

19.2 Size of Array

- In C, the size of an array can be determined using the sizeof operator, which gives the total size in bytes.
- If you divide the total size of an array by the size of an element, you can determine the number of elements in an array.

Example

```
#include <stdio.h>
int main() {
        int arr[] = {10, 20, 30, 40, 50};

        int totalSize = sizeof(arr);
        int elementSize = sizeof(arr[0]);
        int numElements = totalSize / elementSize;

       printf("Total size of the array (in bytes): %d\n", totalSize);
        printf("Size of a single element (in bytes): %d\n", elementSize);
        printf("Number of elements in the array: %d\n", numElements);
        return 0;
      }
```

Output

```
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/8.Arrays$ vi 1.Size_Of_Array.c
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/8.Arrays$ gcc 1.Size_Of_Array.c -o 1.Size_Of_Array
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/8.Arrays$ ./1.Size_Of_Array
Totsl size of the array (in bytes): 20
Size of a single element (in bytes): 4
Number of elements in the array: 5
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/8.Arrays$
```

19.3 Passing Arrays to Functions

Syntax

Function Definition

```
void printArray(int arr[], int size)
  {
        for (int i = 0; i < size; i++)
        {
                printf("%d ", arr[i]);
        }
  }
```

Function Call

```
printArray (arr,10);
```

20 Strings

- A character array, which ends with a null character ('\0'). It can be considered as a special kind of data type, that has a special format specifier, "%s" for reading and printing.
- **To Read a String**
  - scanf("%s",arr);

- scanf with %s will read the characters, store them in the array and place a null character at the end automatically.

- **To Print a String**
  - printf("%s",arr);
  - printf with %s will start printing from the given address, stops at null character.

Synatax

```
#include<stdio.h>
int main()
        {
                char str[50];
                scanf("%s",str);
                printf("%s",str);
                return 0;
        }
```

Note:

- When **'scanf'** reads any input (except for characters), it treats spaces and newline characters as delimiters. This means it will stop reading a string as soon as it encounters a space or a newline character, making it impossible to read a string with spaces using **'scanf'**.

20.1 Declaring and Initialization

- **Declaration**

```
char str[50];  // Declares a character array with space for 50 characters
```

- **Initialization**

```
char str1[] = "Hello, World!";  // Size is implicitly determined by the compiler
char str2[50] = "Hello, World!";  // Size is explicitly set, but only part of the array is
                                                                    used
```

20.2 fgets, gets & puts

- In C, **gets**, **fgets**, and **puts** are functions used for handling strings and input/output operations. Here's a detailed explanation of each:

20.2.1 gets

- Reads a line of text from the standard input (stdin) into a buffer until a newline character or EOF is encountered.
- **gets** does not perform bounds checking, making it prone to buffer overflow attacks. Avoid using gets in any modern C program.

20.2.2 fgets

- Reads a line of text from a specified input stream (stdin, file, etc.) into a buffer, up to a specified number of characters or until a newline character or EOF is encountered.
- The fgets function can perform bounds checking, preventing buffer overflow, and it can read from different input streams, not just stdin.

Example

```
#include <stdio.h>
#define MAX 50
int main()
{
```

```
        char str[MAX];
        // MAX Size if 50 defined
        fgets(str, MAX, stdin);
        gets(str);
        printf("String is: \n");
        puts(str);  // Displaying Strings using Puts
        return 0;
}
```

20.3 Scanset

- Scansets in scanf allow you to specify custom sets of characters to be accepted or rejected when reading input.
- A scanset is defined within square brackets ([]) in the format string, and it allows you to define a custom set of characters that scanf will match.
    - **Basic scanset:** %[set] reads characters in the set.
      Example

```
#include <stdio.h>
int main()
 {
        char str[100];
        printf("Enter a string: ");
        scanf("%[a-zA-Z]", str);  // Reads only letters
        printf("You entered: %s\n", str);
        return 0;
}
```

    - **Negated scanset:** %[^set] reads characters until one in the set is encountered.
      Example

```
#include <stdio.h>
int main()
     {
        char name[100];
        printf("Enter your name: ");
        scanf("%[^\n]", name);  // Reads until a newline is encountered
        printf("Your name is: %s\n", name);
        return 0;
  }
```

20.4 String Library Functions

- In C, the standard library provides a variety of functions to manipulate and handle strings. These functions are declared in the `<string.h>` header file. Here some examples of the most commonly used string functions:

Name	Description
strlen	return the length of string not counting \0
strcopy	copies string from source to dest
strncopy	copies n chars from source to dest
strcat	appends string from source to end of dest
strncat	appends n chars from source to end of dest
strcmp	compares two strings alphabetically
strncmp	compares the first n chars of two strings
strstr	finds a string inside another
strtok	breaks string into tokens using delimiters

21 Pointers

- A pointer is a variable that stores the memory address of another variable. Instead of holding a data value directly, a pointer points to the location in memory where the value is stored.

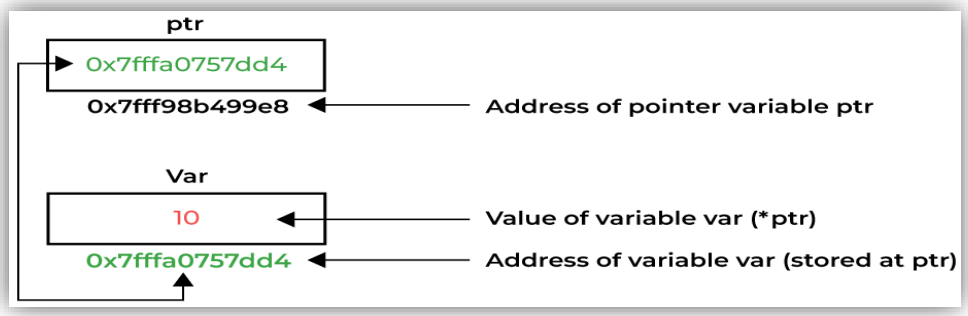
Syntax

```
type *pointerName;
```

- **type**: The data type of the value that the pointer points to.
- ***:** Indicates that the variable is a pointer.
- **pointerName:** The name of the pointer variable.

21.1 Declaring and Initializing Pointers

1. **Declaration:** A pointer is declared by using the * operator.
2. **Address-of Operator (&)**: This operator is used to get the memory address of a variable.
3. **Dereference Operator (*)**: This operator is used to access the value stored at the Pointers can be initialized to the address of another variable using the address-of operator.



Example

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

```
int num = 10;      // Declare an integer variable
int *ptr;        // Declare a pointer to an integer
ptr = &num;       // Store the address of 'num' in the pointer 'ptr'

// Print the value of num
printf("Value of num: %d\n", num);    // Output: 10

// Print the value of num using the pointer
printf("Value at *ptr: %d\n", *ptr);    // Output: 10

// Print the address of num using the pointer
printf("Address of num: %p\n", ptr);   // Output: (Address in hexadecimal)

// Print the address of num using the address-of operator
printf("Address of num (using &): %p\n", &num);  // Output: (Same
address as above)

return 0;
}
```

Output

```
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/10.Pointers$ vi 1.Dec_Init.c
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/10.Pointers$ gcc 1.Dec_Init.c -o 1.Dec_Init
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/10.Pointers$ ./1.Dec_Init
Value of num: 10
Value at *ptr: 10
Address of num: 0x7ffd0d94612c
Address of num (using &): 0x7ffd0d94612c
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/10.Pointers$
```

Explanation

- `'int num = 10;'`: This declares an integer variable `num` and initializes it to 10.
- `'int *ptr;'`: This declares a pointer `ptr` that can hold the address of an integer.
- `'ptr = &num;'`: The pointer `ptr` is assigned the address of the variable `num`.
- `'*ptr'`: This dereferences the pointer, meaning it accesses the value stored at the address contained in `ptr`. In this case, it accesses the value of `num`, which is 10.
- `'%p'`: This format specifier is used in `printf` to print memory addresses.

21.2 Size of a Pointer

The size of a pointer depends on the architecture of the system (i.e., whether it's 32-bit or 64-bit). Here are the details:

32-bit systems: Pointers are typically 4 bytes (32 bits).

64-bit systems: Pointers are typically 8 bytes (64 bits).

The size of a pointer is the same regardless of the data type it points to (e.g., int*, char*, float*, etc.), as the pointer itself only stores the address of a memory location.

Example

```
#include <stdio.h>
int main() {
                char *ptr;
```

```
        int *pointer;
        printf("Size of int pointer: %lu bytes\n", sizeof(ptr));
        printf("Size of int pointer: %lu bytes\n", sizeof(pointer));
        return 0;
}
```

21.3 Wild Pointers & Null Pointers

- A **wild pointer** is a pointer that has not been initialized and therefore points to a random or undefined memory location. Wild pointers are dangerous because they can lead to unpredictable behaviour, including crashes and data corruption.
- **Syntax**

```
datatype *ptrName ;
```

Example

```
#include <stdio.h>
int main() {
        int *ptr;  // Declaring a pointer without initializing it (wild pointer)
        *ptr = 10; // Attempting to dereference a wild pointer
        printf("Value: %d\n", *ptr);
        return 0;
    }
```

Output

```
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/10.Pointers$ vi 2.Wild_Pointer.c
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/10.Pointers$ gcc 2.Wild_Pointer.c -o 2.Wild_Pointer
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/10.Pointers$ ./2.Wild_Pointer
Segmentation fault (core dumped)
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/10.Pointers$
```

- To avoid wild pointers, always initialize pointers before using them by setting them to `NULL` or a valid memory address.
- A **null pointer** is a pointer that is explicitly initialized to **NULL**, indicating that it is not pointing to any valid address in memory.

Syntax

```
datatype *ptrName = NULL;
```

Example

```
#include <stdio.h>
int main() {
        int *ptr = NULL;  // Initializing a pointer to NULL
        if (ptr == NULL)
                {
                        printf("Pointer is NULL.\n");
                }
         return 0;
           }
```

Output

```
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/10.Pointers$ vi 3.Null_Pointer.c
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/10.Pointers$ gcc 3.Null_Pointer.c -o 3.Null_Pointer
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/10.Pointers$ ./3.Null_Pointer
Pointer is NULL.
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/10.Pointers$
```

21.4 Call by Value and Call by Reference in C

Call by value

- In call by value method of parameter passing, the values of actual parameters are copied to the function's formal parameters.
    1. There are two copies of parameters stored in different memory locations.
    2. One is the original copy and the other is the function copy.
    3. Any changes made inside functions are not reflected in the actual parameters of the caller.

Example

```
#include <stdio.h>
// Function Prototype
        void swapx(int x, int y);
// Main function
        int main()
                {
                        int a = 10, b = 20;
                        printf("Before swapping: a = %d b = %d\n",a,b);

                        // Pass by Values
                        swapx(a, b); // Actual Parameters

                        printf("In the Caller: a = %d b = %d\n", a, b);
                        return 0;
                }

        // Swap functions that swaps
        // two values
        void swapx(int x, int y) // Formal Parameters
                {
                        int t;

                        t = x;
                        x = y;
                        y = t;

                        printf("After swapping is done Inside the function: x = %d y
        = %d\n", x, y);
                }
```

Output

```
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/10.Pointers$ vi 4.Call_By_Value.c
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/10.Pointers$ gcc 4.Call_By_Value.c -o 4.Call_By_Value
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/10.Pointers$ ./4.Call_By_Value
Before swapping: a = 10 b = 20
After Swapping is done inside the function: x = 20 y = 10
In the caller: a = 10 b = 20
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/10.Pointers$
```

Call by Reference

- In call by reference method of parameter passing, the address of the actual parameters is passed to the function as the formal parameters. In C, we use pointers to achieve call-by-reference.
    1. Both the actual and formal parameters refer to the same locations.
    2. Any changes made inside the function are actually reflected in the actual parameters of the caller.

Example

```
// C program to illustrate Call by Reference
        #include <stdio.h>
// Function Prototype
        void swapx(int*, int*);
// Main function
        int main()
                {
                        int a = 10, b = 20;
                        printf("Before swapping: a = %d b = %d\n",a,b);

                        // Pass reference
                        swapx(&a, &b);          // Actual Parameters

                        printf("In the Caller: a = %d b = %d\n", a, b);
                        return 0;
                }

// Function to swap two variables
// By references
        void swapx(int* x, int* y) // Formal Parameters
                {
                        int t;

                        t = *x;
                        *x = *y;
                        *y = t;

                        printf("Inside the Function: x = %d y = %d\n", *x, *y);
                }
```

Output

```
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/10.Pointers$ vi 5.Call_By_Reference.c
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/10.Pointers$ gcc 5.Call_By_Reference.c -o 5.Call_By_Reference
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/10.Pointers$ ./5.Call_By_Reference
Before swapping: a = 10 b = 20
After Swapping is done inside the function: x = 20 y = 10
In the caller: a = 20 b = 10
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/10.Pointers$
```

Key Differences

1. **Modification Impact**
   - Call by Value: Modifications inside the function do not affect the original variable.
   - Call by Reference: Modifications inside the function do affect the original variable.
2. **Memory Usage**
   - Call by Value: A copy of the variable is created, which uses additional memory.
   - Call by Reference: No additional memory is used for copying; only the address is passed.
3. **Safety**
   - Call by Value: Safer since the original data cannot be altered.
   - Call by Reference: Less safe since the original data can be altered, potentially leading to unintended side effects.

21.5 Pointer Arithmetic

- Pointer arithmetic allows you to perform operations on pointers, such as incrementing, decrementing, or subtracting them. This is particularly useful when working with arrays, as pointers can be used to iterate through array elements.
- **Basic arithmetic operations you can do with pointers**
  1. **Incrementing (ptr++):** Moving the pointer to the next memory location.
  2. **Decrementing (ptr--):** Moving the pointer to the previous memory location.
  3. **Addition (ptr+n):** Moving the pointer forward by a specific number of elements.
  4. **Subtraction (ptr-n):** Moving the pointer backward by a specific number of elements.
  5. **Pointer differences (**ptr2 - ptr1**):** Calculating the number of elements between two pointers.

Example

```
#include <stdio.h>
int main() {
        int arr[] = {10, 20, 30, 40, 50};
        int *ptr = arr;  // Pointing to the first element of the array

        // Printing the initial pointer value
        printf("Initial pointer address: %p, value: %d\n", ptr, *ptr);

        // Incrementing the pointer
        ptr++;
        printf("After ptr++: address: %p, value: %d\n", ptr, *ptr);

        // Adding 2 to the pointer (moves forward by 2 elements)
```

```
    ptr += 2;
    printf("After ptr += 2: address: %p, value: %d\n", ptr, *ptr);

    // Subtracting 1 from the pointer (moves backward by 1 element)
    ptr--;
    printf("After ptr--: address: %p, value: %d\n", ptr, *ptr);

    // Difference between pointers
    int *ptr2 = &arr[4];  // Pointing to the last element of the array
    int diff = ptr2 - ptr;
    printf("Difference between ptr2 and ptr: %d elements\n", diff);

    return 0;
}
```

Output

```
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/10.Pointers$ vi 6.Pointer_Arithmetic.c
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/10.Pointers$ gcc 6.Pointer_Arithmetic.c -o 6.Pointer_Arithmetic
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/10.Pointers$ ./6.Pointer_Arithmetic
Intial pointer address: 0x7ffd9ae4ce50, Value: 10
After ptr++:
Address: 0x7ffd9ae4ce54, Value : 20
After ptr += 2:
Address: 0x7ffd9ae4ce5c, Value : 40
After ptr--:
Address: 0x7ffd9ae4ce58, Value : 30
Difference between ptr2 and ptr: 2 elements
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/10.Pointers$
```

- **Arithmetic operations you cannot do with pointers**
  1. **Adding two pointers (ptr1 + ptr2 if not in the same array)**

```
int x, y;
int *ptr1 = &x;
int *ptr2 = &y;
int sum = ptr1 + ptr2; // Invalid adding two pointers is not allowed
```

  2. **Subtracting Two Pointers (ptr1 - ptr2 if not in the same array)**

```
int x, y;
int *ptr1 = &x;
int *ptr2 = &y;
int diff = ptr1 - ptr2;  // Invalid Subtracting two pointers is not allowed
```

  3. **Multiplying/Dividing pointers (ptr * n or ptr / n)**

```
int *ptr;
mul = ptr * 2; // Invalid multiplication of pointers is not allowed.
div = ptr / 2; // Invalid division of pointers is not allowed.
```

  4. **Adding/Subtracting pointers of different types (int* + float*)**

```
int *intPtr = &x; // Pointer to an integer
float *floatPtr = &y; // Pointer to a float
 int *resultPtr = intPtr + floatPtr; // This would cause a compilation error
int *resultPtr = intPtr - floatPtr; // This would also cause a compilation
error
```

  5. **Pointer Arithmetic on (void*) Pointers**

```
void *ptr;
ptr++;  // Invalid arithmetic on void pointers is not allowed because the
size of the pointed-to type is unknown.
```

21.6 What is Void Pointer?

- A void pointer in C is a special type of pointer that can point to any data type. It is declared using the keyword void. Since a void pointer is a generic pointer type, it does not have a specific data type associated with it, making it versatile for various pointer operations.

Syntax

```
void *ptrName;
```

Example

```
#include <stdlib.h>
int main()
        {
                int x = 4;
                float y = 5.5;

                //Void pointer
                void* ptr;
                ptr = &x;

                // (int*)ptr - does type casting of void
                // *((int*)ptr) dereferences the typecasted

                // void pointer is now integer.
                printf("Integer variable is = %d, Address of integer variable=%p",
                *((int*)ptr),ptr);

                // void pointer is now float
                ptr = &y;
                printf("Float variable is = %.2f, Address of float variable=%p",
                *((float*)ptr),ptr);

                return 0;
        }
```

Output

```
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/10.Pointers$ vi 7.Void_Pointer.c
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/10.Pointers$ gcc 7.Void_Pointer.c -o 7.Void_Pointer
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/10.Pointers$ ./7.Void_Pointer
Interger Variable=5, Address of integer variable=0x7ffd64077f58
Float Variable=5.50, Address of float variable=0x7ffd64077f5c
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/10.Pointers$
```

21.7 Function Pointers

- Function pointers in C are pointers that point to functions instead of data.
- They allow programs to dynamically choose which function to execute at runtime, providing flexibility, particularly in cases like call back functions, event-driven programming, or implementing state machines.

Syntax

```
return_type (*pointer_name)(parameter_types);
```

Example

```
#include <stdio.h>
// A normal function with an int parameter
// and void return type
void fun(int a)
{
        printf("Value of a is %d\n", a);
}

int main()
{
        // fun_ptr is a pointer to function fun()
        void (*fun_ptr)(int) = &fun;

        /* The above line is equivalent of following two
        void (*fun_ptr)(int);
        fun_ptr = &fun;
        */

        // Invoking fun() using fun_ptr
        (*fun_ptr)(10);

        return 0;
}
```

- A function's name can be used to get its address. In the following program, the address operator '&' has been removed in the assignment, and the function call works even without the dereference operator '*'.

Example

```
#include <stdio.h>
// A normal function with an int parameter
// and void return type
void fun(int a)
{
        printf("Value of a is %d\n", a);
}

int main()
{
        void (*fun_ptr)(int) = fun; // & removed
        fun_ptr(10); // * removed
        return 0;
}
```

Example

```
#include <stdio.h>

void add(int a, int b)
{
        printf("Addition is %d\n", a+b);
}

void subtract(int a, int b)
```

```
{
        printf("Subtraction is %d\n", a-b);
}

void multiply(int a, int b)
{
        printf("Multiplication is %d\n", a*b);
}

int main()
{
        // fun_ptr_arr is an array of function pointers
        void (*fun_ptr_arr[])(int, int) = {add, subtract, multiply};
        unsigned int ch, a , b ;

        printf("Enter Choice: 0 for add, 1 for subtract and 2 for multiply\n");
        scanf("%d", &ch);
        if (ch > 2)
        {
                printf("Invalid Entry\n");
                return 0;
        }
        printf("Enter two numbers\n");
        scanf("%d", &a);
        scanf("%d", &b);
         (*fun_ptr_arr[ch])(a, b);

        return 0;
}
```

Output

```
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/10.Pointers$ vi 10.Function_Pointer.c
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/10.Pointers$ gcc 10.Function_Pointer.c -o 10.Function_Pointer
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/10.Pointers$ ./10.Function_Pointer
Enter Choice: 0 for Addition, 1 for Subtraction and 2 for Multiplication
0
Enter Any Two Numbers:
45
65
Addition of 45 and 65 is: 110
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/10.Pointers$
```

22 Dynamic Memory Allocation

- Dynamic memory allocation in C allows you to allocate memory at runtime using a set of library functions provided by the C standard library.
- This is particularly useful when the size of a data structure (like an array) cannot be determined at compile-time.
- There are four library functions provided by C, defined in the **<stdlib.h>** header file, to facilitate dynamic memory allocation in C programming. They are:
    1. malloc() - memory allocation
    2. calloc() - contiguous allocation
    3. free() - de-allocate

4. realloc() - re-allocation

22.1 malloc (memory allocation)

- The malloc (memory allocation) method in C is used to dynamically allocate a single large block of memory with the specified size.
- It returns a pointer of type void which can be cast into a pointer of any form.
- It doesn't initialize memory at execution time so that it has initialized each block with the default garbage value initially.

Syntax

```
ptr = (cast-type*) malloc (size_t size)
```

Example

```
int *ptr;
ptr = (int*)malloc(10 * sizeof(int));  // Allocates memory for an array of 10 integers
```

22.2 calloc (contiguous allocation)

- The calloc (contiguous allocation) function in C is used to dynamically allocate a specified number of memory blocks of a specified type, similar to malloc().
- It initializes each block with a default value of 0.
- Unlike malloc(), calloc() takes two parameters or arguments.

Syntax

```
ptr = (cast-type*)calloc(n, element-size);
```

Here, n is the no. of elements and element-size is the size of each element.

Example

```
ptr = (float*) calloc(25, sizeof(float));
```

22.3 free (De - allocation)

- The free function in C is used to de - allocate dynamically allocated memory.
- Memory allocated using functions like malloc() and calloc() is not automatically de - allocated; it remains allocated until the program ends or `free()` is explicitly called.
- Therefore, whenever dynamic memory allocation is used, it is important to use the free() function to release the memory, helping to prevent memory leaks and reduce memory wastage.

Syntax

```
free(ptr);
```

22.4 Realloc (Re - allocation)

- The realloc (re - allocation) function in C is used to dynamically adjust the size of a previously allocated memory block.
- If the memory allocated by malloc or calloc is insufficient, realloc can be used to resize the memory block while preserving the existing data.
- When the memory block is reallocated, the existing values are maintained, but any newly allocated memory will contain uninitialized data, often referred to as a "garbage value."

Syntax

```
ptr = realloc(ptr, newSize);
```

Where ptr is re - allocated with new size 'newSize'.

Example

```
ptr = (int*)realloc(ptr, 10 * sizeof(int));  // Re - allocates memory for an array of 10 integers
```

23 Dangling Pointer & Double Pointer

23.1 Dangling Pointer

- A dangling pointer in C is a pointer that still points to a memory location that has been freed or deallocated.
- Accessing or using a dangling pointer leads to undefined behaviour, which can cause program crashes, data corruption, or other unpredictable issues.

Example

```
#include <stdio.h>
#include <stdlib.h>
int main()
        {
                // Allocate 4 bytes of int memory block (64-bit compiler) using
                malloc() during runtime
                int *ptr = (int *)malloc(sizeof(int));  // normal pointer
                 *ptr = 10;

                 // Memory block deallocated using free() function
                free(ptr);

                // Here ptr acts as a dangling pointer
                // Prints garbage value and adress in the output console
                printf("%d\n", *ptr);
                printf("%p\n", ptr);
                // Removing Dangling Pointer
                ptr = NULL;
                printf("After removing dangling pointer.....!!!\n");

                // Check whether if ptr is NULL
                if (ptr == NULL)
                {
                        printf("Pointer is NULL and safe to use.\n");
                }
                        return 0;
        }
```

Output

```
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/10.Pointers$ vi 8.Dangling_Pointer.c
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/10.Pointers$ gcc 8.Dangling_Pointer.c -o 8.Dangling_Pointer
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/10.Pointers$ ./8.Dangling_Pointer
0
0x1c1c010
After removing dangling pointer.....!!!
Pointer is NULL and safe to use.
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/10.Pointers$
```

23.2 Double Pointer

- A double pointer in C is a pointer that holds the address of another pointer, enabling the creation and management of more complex data structures like multidimensional arrays, linked lists, and dynamic memory allocation.

Syntax

```
data_type_of_pointer **name_of_variable = & normal_pointer_variable;
```

Example

```
#include <stdio.h>
int main()
{
        int var = 789;

        // pointer for var
        int* ptr2;

        // double pointer for ptr2
        int** ptr1;

        // storing address of var in ptr2
        ptr2 = &var;

        // storing address of ptr2 in ptr1
        ptr1 = &ptr2;

        // Displaying value of var using both single and double pointers
        printf("Value of var = %d\n", var);
        printf("Value of var using single pointer = %d\n", *ptr2);
        printf("Value of var using double pointer = %d\n", **ptr1);

        return 0;
}
```

Output

```
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/10.Pointers$ vi 9.Double_Pointer.c
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/10.Pointers$ gcc 9.Double_Pointer.c -o 9.Double_Pointer
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/10.Pointers$ ./9.Double_Pointer
Value of var: 789
Value of var using single pointer: 789
Value of var using double pointer: 789
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/10.Pointers$
```

24 Const (Constant) Qualifier in C

- In C, the const keyword is used to declare variables that should not be modified after their initialization.
- When you declare a variable as const, you are indicating that its value is constant and cannot be changed during the execution of the program.
- Here are some different use cases of the const qualifier in C:
    1. Constant Variables

2. Pointer to Constant
3. Constant Pointer to Variable
4. Constant Pointer to Constant Data

24.1 Constant Variables

- When a variable is declared as const, it means that the value of this variable cannot be modified.

Syntax

```
const int var = 100;
```

Example

```
#include <stdio.h>
int main()
  {
        const int var = 100;
        printf("The value is : %d",var);

        //Compilation error assignment of read-only variable 'var'
              var = 200;

              return 0;
}
```

Output

```
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/11.Constant_Qualifier$ vi 1.Const_Variable.c
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/11.Constant_Qualifier$ gcc 1.Const_Variable.c -o 1.Const_Varible
1.Const_Variable.c: In function ‘main’:
1.Const_Variable.c:5:5: error: assignment of read-only variable ‘var’
  var=200;
    ^
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/11.Constant_Qualifier$ vi 1.Const_Variable.c
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/11.Constant_Qualifier$ gcc 1.Const_Variable.c -o 1.Const_Varible
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/11.Constant_Qualifier$ ./1.Const_Varible
The value is: 100
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/11.Constant_Qualifier$
```

24.2 Pointer to Constant

- The data pointed to by the pointer cannot be modified, but the pointer itself can point to a different address.

Syntax

```
const int* ptr;
```

(OR)

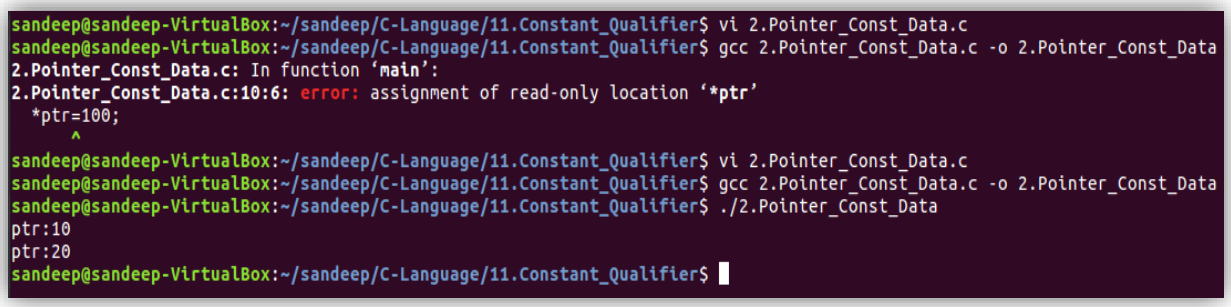
```
int const *ptr;
```

Example

```
#include <stdio.h>
int main()
{
        int i = 10;
        int j = 20;

        //ptr is pointer to constant
        const int* ptr = &i;
        printf("ptr: %d\n", *ptr);
```

```
        //Error object pointed cannot be modified using the pointer ptr *
        *ptr = 100;

        ptr = &j;          // Valid
        printf("ptr: %d\n", *ptr);

        return 0;
}
```

Output



24.3 Constant Pointer to Variable

- The pointer cannot point to a different address, but the data at that address can be modified.

Syntax

```
int* const ptr;
```

Example

```
#include <stdio.h>
int main(void)
        {
                int i = 10;
                int j = 20;

                //constant pointer to integer
                int* const ptr = &i;
                printf("ptr: %d\n", *ptr);

                *ptr = 100;        // Valid
                printf("ptr: %d\n", *ptr);

                ptr = &j;          // Error
                return 0;
        }
```

Output

```
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/11.Constant_Qualifier$ vi 3.Const_Pointer_Variable.c
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/11.Constant_Qualifier$ gcc 3.Const_Pointer_Variable.c -o 3.Const_Pointer_Variable
3.Const_Pointer_Variable.c: In function ‘main’:
3.Const_Pointer_Variable.c:13:5: error: assignment of read-only variable ‘ptr’
  ptr=&j;
     ^
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/11.Constant_Qualifier$ vi 3.Const_Pointer_Variable.c
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/11.Constant_Qualifier$ gcc 3.Const_Pointer_Variable.c -o 3.Const_Pointer_Variable
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/11.Constant_Qualifier$ ./3.Const_Pointer_Variable
ptr=10
ptr=100
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/11.Constant_Qualifier$
```

24.4 Constant Pointer to Constant Data

- In constant pointer to a constant variable which means we cannot change the value pointed by the pointer as well as we cannot point the pointer to other variables.

Syntax

```
const int* const ptr;
```

Example

```
#include <stdio.h>
int main(void)
        {
                int i = 10;
                int j = 20;

                //constant pointer to constant integer
                const int* const ptr = &i;
                printf("ptr: %d\n", *ptr);

                ptr = &j;          //Error
                *ptr = 100;        //Error

                return 0;
        }
```

Output

```
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/11.Constant_Qualifier$ vi 4.Const_Pointer_Const_Variable.c
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/11.Constant_Qualifier$ gcc 4.Const_Pointer_Const_Variable.c -o 4.Const_Pointer_Const_Variable
4.Const_Pointer_Const_Variable.c: In function ‘main’:
4.Const_Pointer_Const_Variable.c:11:13: error: assignment of read-only location ‘*ptr’
        *ptr=100;
            ^
4.Const_Pointer_Const_Variable.c:12:5: error: assignment of read-only variable ‘ptr’
  ptr=&j;
     ^
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/11.Constant_Qualifier$ vi 4.Const_Pointer_Const_Variable.c
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/11.Constant_Qualifier$ gcc 4.Const_Pointer_Const_Variable.c -o 4.Const_Pointer_Const_Variable
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/11.Constant_Qualifier$ ./4.Const_Pointer_Const_Variable
ptr=10
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/11.Constant_Qualifier$
```

25 Structures

25.1 Defining a Structure

- In C, a structure is a user-defined data type that groups variables of different types into a single entity.
- The **struct** keyword is used to declare a structure in the C programming language.
- The variables within a structure are called members, and they can be of any valid data type.
- The values within a structure are stored in contiguous memory locations.

Syntax

```
struct StructureName {
        dataType member1;
        dataType member2;
        ----------// more members
};
```

Example

```
#include <stdio.h>
// Defining the structure
struct Student {
        char name[100];
        int age;
        float percent;
};

int main() {
                // Declaring a variable of type struct Student
                struct Student s1;

                // Asking for user input and storing values in the structure members
                printf("Please enter student details of s1\n");

                printf("Enter Student Name: ");
                scanf("%s", s1.name);

                printf("Enter Student Age: ");
                scanf("%d", &s1.age);

                printf("Enter Percentage of Student: ");
                scanf("%f", &s1.percent);

                // Displaying the entered student details
                printf("Entered student details of s1 are\n");
                printf("Student Name:  %s\n", s1.name);
                printf("Student Age: %d\n", s1.age);
                printf("Total Percentage of Student: %.2f\n", s1.percent);

                return 0;
        }
```

Output

```
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/13.Structures$ vi 1.Basic_Structure.c
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/13.Structures$ gcc 1.Basic_Structure.c -o 1.Basic_Structure
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/13.Structures$ ./1.Basic_Structure
Please enter student details of s1
Enter Student Name: sandeep
Enter Student Age: 26
Enter Percentage of Student: 70.355
Entered student details of s1 are
Student Name: sandeep
Student Age: 26
Total Percentage of Student: 70.36
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/13.Structures$
```

Key Points:

1. **Structure Declaration:** It defines a new data type but does not allocate memory.
2. **Structure Variable:** You create variables of the structure type (e.g., struct Point p1;).
3. **Accessing Members:** Use the dot (.) operator to access individual members (e.g., p1.x).

25.2 Finding the Size of a Structure

- Use the **sizeof** operator to determine the size of a structure in bytes, including any padding bytes added for memory alignment.

Example

```
#include <stdio.h>
struct Student {
                int age;
                char name;
                double percent;
        };
int main() {
        struct Student s1;
        // Finding the size of the structure using sizeof operator
        printf("Size of structure student: %lu bytes\n", sizeof(s1));

        //Alternatively, you can also use the structure type directly
        printf("Size of structure student: %lu bytes\n", sizeof(struct Student));

        //Finding size of each variable inside the structure
        printf("Size of each variable inside the structure (char): %lu bytes\n",
        sizeof(s1.name));
        printf(("Size of each variable inside the structure (int):%lu bytes\n",
        sizeof(s1.age));
        printf(("Size of each variable inside the structure (double):%lu bytes\n",
        sizeof(s1.percent));
        return 0;
}
```

Output

```
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/13.Structures$ vi 2.Sizeof_Structure.c
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/13.Structures$ gcc 2.Sizeof_Structure.c -o 2.Sizeof_Structure
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/13.Structures$ ./2.Sizeof_Structure
Size of structure student: 16 bytes
Size of structure student: 16 bytes
Size of each variable inside the structure (char): 1 bytes
Size of each variable inside the structure (int):4 bytes
Size of each variable inside the structure (double):8 bytes
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/13.Structures$
```

Explanation of Padding

- The structure Student has a total of 13 bytes of data (4 + 8 + 1).
- However, the actual size of the structure is likely 16 bytes due to padding. Memory alignment rules might insert padding bytes between the members so that each member starts at an address aligned to its type:
    - int (4 bytes) is stored at a 4-byte boundary.
    - double (8 bytes) is stored at an 8-byte boundary.
    - char (1 byte) is padded to the nearest multiple of 4 or 8 bytes to maintain alignment for future structures.
- Thus, the structure size becomes 16 bytes instead of 13.

25.3 Structure Padding

- Structure padding is the addition of empty bytes in a structure to naturally align the data members in memory.
- It is done to minimize CPU read cycles when retrieving different data members in the structure.
- Structure padding generally improves performance by aligning data efficiently for the processor.
- However, it can negatively affect memory efficiency, depending on the specific use case.

25.3.1 How to Avoid Structure Padding

1. **Using #pragma pack Directive**
    - This directive can modify the default alignment and packing behaviour of the compiler.
    - #pragma pack (1) tells the compiler to use 1-byte alignment for all members, effectively preventing padding.

Example

```
#include <stdio.h>
#pragma pack (1) // Disable padding
struct Student {
                int age;
                char name;
                double percent;
        };
int main() {
        struct Student s1;
        // Finding the size of the structure using sizeof operator
        printf("Size of structure student: %lu bytes\n", sizeof(s1));
        return 0;
    }
```

Output

```
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/13.Structures$ vi 3.#pragama_Structure.c
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/13.Structures$ gcc 3.#pragama_Structure.c -o 3.#pragama_Structure
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/13.Structures$ ./3.#pragama_Structure
Size of student:13 bytes
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/13.Structures$
```

2. **Reordering Structure Members**
   - Another way to reduce padding is by grouping larger data types together and smaller ones afterward.

Example

```
#include <stdio.h>
struct Student {
        double percent;
        int age;
        char name;
    };
int main() {
        struct Student s1;
        // Finding the size of the structure using sizeof operator
        printf("Size of structure student: %lu bytes\n", sizeof(s1));
        return 0;
    }
```

Output

```
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/13.Structures$ vi 4.Reorder_Structure.c
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/13.Structures$ gcc 4.Reorder_Structure.c -o 4.Reorder_Structure
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/13.Structures$ ./4.Reorder_Structure
Size of student:13 bytes
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/13.Structures$
```

3. **Using __attribute__((packed)) (GCC Specific)**:
   - In GCC, you can use the packed attribute to disable padding. It is equivalent to #pragma pack (1) and ensures that there is no padding between the structure members.

Example

```
#include <stdio.h>
struct Student {
        int age;
        char name;
        double percent;
    } __attribute__((packed));
int main() {
        struct Student s1;
        // Finding the size of the structure using sizeof operator
        printf("Size of structure student: %lu bytes\n", sizeof(s1));
        return 0;
    }
```

Output

```
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/13.Structures$ vi 5.GCC_Attribute_Structure.c
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/13.Structures$ gcc 5.GCC_Attribute_Structure.c -o 5.GCC_Attribute_Structure
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/13.Structures$ ./5.GCC_Attribute_Structure
Size of student:13 bytes
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/13.Structures$
```

25.4 Array of Structure

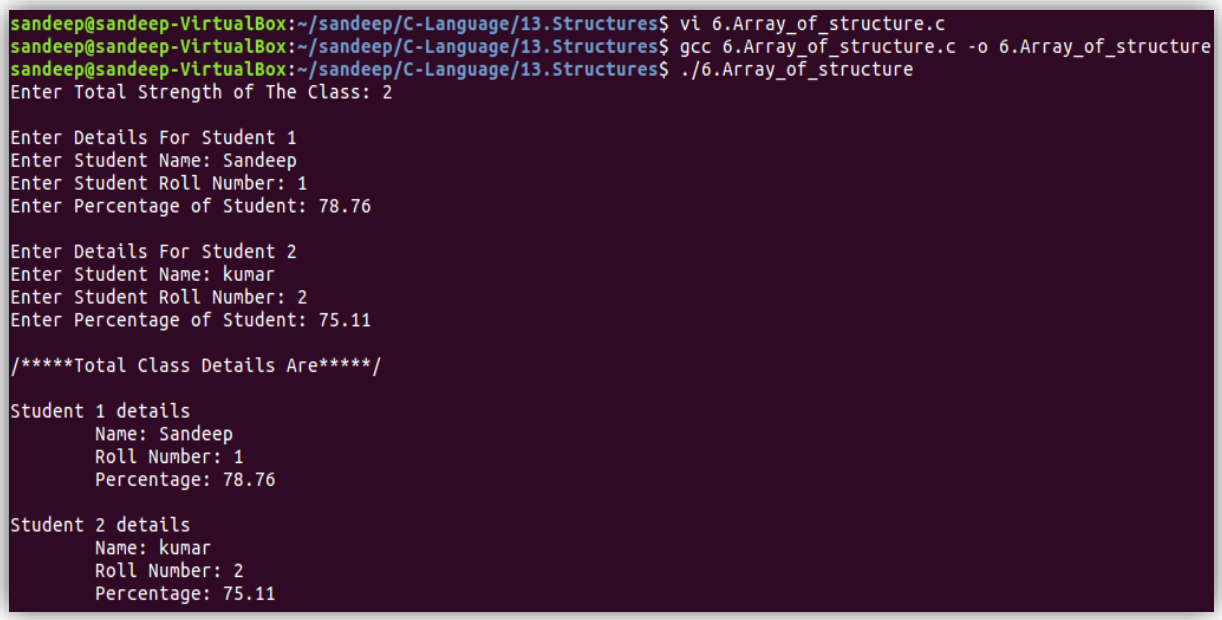
- An array whose elements are of type structure is called array of structure. It is generally useful when we need multiple structure variables in our program.

Example

```
#include <stdio.h>
#define MAX_SIZE 100
struct Student {
                    char name[100];
                    int rollno;
                    float percent;
            };

int main() {
            struct Student class[MAX_SIZE];
            int strength;

            //Get the total strength of the class
            printf("Enter total strength of the class: ");
            scanf("%d", &strength);

            //Input details for each student
            for (int i = 1; i <= strength; i++)
            {
                    printf("\nEnter details for Student %d\n", i);

                    printf("Enter Student Name: ");
                    scanf("%s", class[i].name);

                    printf("Enter Student Roll Number: ");
                    scanf("%d", &class[i].rollno);

                    printf("Enter Percentage of Student: ");
                    scanf("%f", &class[i].percent);
             }

                    // Display the entered details
                    printf("\n/******Total Class Details are******/\n\n");
                    for (int i = 1; i <= strength; i++)
                    {
                            printf("Student %d details\n",i);
                            printf("\tName: %s\n", class[i].name);
```

```
                printf("\tRoll Number: %d\n", class[i].rollno);
                printf("\tPercentage: %.2f\n\n", class[i].percent);
            }

        return 0;
    }
```

Output



25.5 Structure Pointer

- A structure pointer is a pointer that points to the memory address of a structure.
- Structure pointers are used to create complex data structures such as linked lists, trees, and graphs.
- The structure pointer provides the memory address of a structure by pointing to the structure variable.

Syntax

```
struct StructName {
        int member1;
        float member2;
};

struct StructName *ptr; // Pointer to structure
```

Key Points:

- **Accessing Members via Pointer:** Use the arrow operator (->) to access members of the structure through a pointer.
- **Modifying Members via Pointer:** You can also modify structure members via pointers, which allows you to efficiently manipulate structures.
- **Passing Structures to Functions:** Passing large structures directly by value can be inefficient. Using pointers reduces overhead, as only the address is passed.

Example

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

```c
// Defining the structure
struct Student {
        char name[100];
        int age;
        float percent;
};

int main() {
    // Declaring a variable of type struct Student
    struct Student class;
    struct Student *ptr;
    ptr = &class;

    // Asking for user input and storing values in the structure members
    printf("Please enter student details\n");

    printf("Enter Student Name: ");
    scanf("%s", ptr->name);  // Accessing name via pointer

    printf("Enter Student Age: ");
    scanf("%d", &ptr->age);  // Need to pass address (&) to scanf

    printf("Enter Percentage of Student: ");
    scanf("%f", &ptr->percent);  // Accessing percentage via pointer

    // Displaying the entered student details
    printf("\nEntered student details are:\n");
    printf("Student Name:  %s\n", ptr->name);
    printf("Student Age: %d\n", ptr->age);
    printf("Total Percentage of Student: %.2f\n", ptr->percent);

     return 0;
}
```

Key Points:

- **Structure:** The structure Student groups related data types (name, age, and percent) into a single type.
- **Pointer Access:** The code demonstrates accessing and modifying structure members using both direct access (class.name) and pointer notation (ptr->age, ptr->percent).
- **Input Handling:** The scanf function is used to take input, and when accessing members through a pointer, you must pass the address of the member (e.g., &(ptr->age)). Also Here, ptr->age is equivalent to (*ptr).age.

Output

```
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/13.Structures$ vi 7.Pointer_structure.c
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/13.Structures$ gcc 7.Pointer_structure.c -o 7.Pointer_structure
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/13.Structures$ ./7.Pointer_structure
Please Enter Student Details
Enter Student Name:sandeep
Enter Student Age:26
Enter Percentage of Student:78.4567

/*****Entered Student Details are*****/
        Student Name: sandeep
        Student Age: 26
        Total Percentage of Student: 78.46
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/13.Structures$
```

25.6 Structures Using Functions

- We can use structures (struct) along with functions to organize data and manipulate it more efficiently.
- Structures allow you to group different types of data together.
- Functions help encapsulate operations that can be performed on the data.

Example

```
#include <stdio.h>
struct student {
                        char name[20];
                        int age;
                        float marks;
                };
// function to return a structure
struct student get_student_data()
{
        struct student s;

        printf("Enter name: ");
        scanf("%s", s.name);
        printf("Enter age: ");
        scanf("%d", &s.age);
        printf("Enter marks: ");
        scanf("%f", &s.marks);

        return s;
}

int main()
{
        //structure variable s1 which has been assigned the
        //returned value of get_student_data
        struct student s1 = get_student_data();
        //displaying the information
        printf("Name: %s\n", s1.name);
        printf("Age: %d\n", s1.age);
        printf("Marks: %.1f\n", s1.marks);
```

```
        return 0;
    }
```

Output

```
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/13.Structures$ vi 8.Function_Structure.c
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/13.Structures$ gcc 8.Function_Structure.c -o 8.Function_Structure
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/13.Structures$ ./8.Function_Structure
Please Enter Student Details
Enter Student Name:sandeep
Enter Student Age:26
Enter Percentage of Student:78.9654

/*****Entered Student Details are*****/
        Student Name: sandeep
        Student Age: 26
        Total Percentage of Student: 78.97
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/13.Structures$
```

26 Unions

26.1 Defining a Union

- A union in C is a user-defined data type that stores different data types in the same memory location.
- Unlike structures, all members of a union share the same memory space.
- Only one member can hold a valid value at a time, as they all use the same memory address.

Syntax

```
uinon UnionName {
        dataType member1;
        dataType member2;
        ----------------// more members
};
```

Example

```
#include <stdio.h>
//Defining the union
uinon Student {
        char name[100]; // occupies the largest memory space in the union
        int age;
        float percent;
};

int main()
{
        // Declaring a variable of type union Student
        union Student s1;

        // Asking for and displaying one member at a time due to union behavior
        printf("\tPlease enter student details of s1\n");

        // Enter and display Student Name
        printf("\nEnter Student Name: ");
        scanf("%s", s1.name);
        printf("Student Name: %s\n", s1.name);
```

```
    // Enter and display Student Age (note this will overwrite name)
    printf("\nEnter Student Age: ");
    scanf("%d", &s1.age);
    printf("Student Age: %d\n", s1.age);

    // Trying to access name after assigning age leads to undefined behavior
    printf("\nAfter overwriting student name: %s\n", s1.name); // Undefined behavior

    // Enter and display Student Percentage (this will overwrite age)
    printf("\nEnter Percentage of Student: ");
    scanf("%f", &s1.percent);
    printf("Total Percentage of Student: %.2f\n", s1.percent);

    // Trying to access age after assigning percent leads to undefined behavior
    printf("\nAfter overwriting student age: %d\n", s1.age); // Undefined behavior

    return 0;
}
```

Output

```
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/15.Unions$ vi 1.Basicunion.c
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/15.Unions$ gcc 1.Basicunion.c -o 1.Basicunion
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/15.Unions$ ./1.Basicunion
        Please enter student details of s1

Enter Student Name: sandeep
Student Name: sandeep

Enter Student Age: 26
Student Age: 26

After overwriting student name:

Enter Percentage of Student: 78.76
Total Percentage of Student: 78.76

After overwriting student age: 1117619487
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/15.Unions$
```

26.2 Unions & Structures Differences

Feature	Structure	Union
Memory Allocation	Each member has its own memory location.	All members share the same memory location.
Memory Size	Total size is the sum of all member sizes.	Size is equal to the size of the largest member.
Member Access	All members can store distinct values simultaneously.	Only one member can hold a value at a time.
Usage	Used when you need to store multiple values at once.	Used to save memory when only one value is needed.
Initialization	All members can be initialized separately.	Only the first member can be initialized at declaration.
Example Syntax	struct Example { int a; float b; };	union Example { int a; float b; };

Data Integrity	All member values remain intact.	Assigning to one member overwrites others.
Efficiency	Less memory-efficient (more overhead).	More memory-efficient (less overhead).

26.3 Finding the Size of a Union

- The size of a union in C is determined by the size of its largest member.
- All members of a union share the same memory space.
- The memory allocated to the union must be large enough to accommodate its largest member.

Example

```
#include <stdio.h>
union Data {
        int i;          // 4 bytes
        float f;        // 4 bytes
        double d;       // 8 bytes
};
int main() {
        union Data data;
        printf("Size of union: %lu bytes\n", sizeof(data));
        return 0;
}
```

Output

```
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/15.Unions$ vi 2.Sizeofunion.c
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/15.Unions$ gcc 2.Sizeofunion.c -o 2.Sizeofunion
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/15.Unions$ ./2.Sizeofunion
Size of union: 8 bytes
```

27 Bit Fields

- Bit fields in C allow specific allocation of bits for struct or union members.
- They provide control over memory usage, crucial for low-level programming and embedded systems.
- Useful for storing flags, small numbers, or hardware registers.
- Specify the size (in bits) of structure and union members to optimize memory.
- Bit fields are beneficial when values are known to be within a small range, helping minimize memory consumption.
- Particularly useful when program storage is limited.

Syntax

```
struct
{
  data_type member_name : width_of_bit-field;
};
```

Example

```
#include <stdio.h>
#include <stdio.h>
#pragma pack(1)
```

```
struct WithoutBitfield {
                    unsigned int flag1;   // 4 bytes (assuming int is 4 bytes)
                    unsigned int flag2;   // 4 bytes
                    unsigned int flag3;   // 4 bytes
                }; // without using bitfields

struct WithBitfield {
                    unsigned int flag1 : 1;  // 1 bit
                    unsigned int flag2 : 1;  // 1 bit
                    unsigned int flag3 : 1;  // 1 bit
                }; // using bitfields

int main() {
        struct WithoutBitfield noBitfield = {1,0,1}; // Without bitfields
        struct WithBitfield withBitfield = {1,0,1}; // With bitfields

        // Checking sizes
        printf("Size without bitfields: %lu bytes\n", sizeof(noBitfield));
        printf("Size with bitfields: %lu bytes\n", sizeof(withBitfield));
        printf("\n");
        return 0;
    }
```

Output

```
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/16.Bit_Fields$ vi 1.BasicProgram.c
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/16.Bit_Fields$ gcc 1.BasicProgram.c -o 1.BasicProgram
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/16.Bit_Fields$ ./1.BasicProgram
Size without bitfields: 12 bytes
Size with bitfields: 1 bytes
```

28 Data Structures

- Data Structures in C are a way of storing and organizing data in computer memory so that it can be processed efficiently. By using data structures in C, we can optimize memory usage and improve the performance of our programs.
- Types of Data Structures:
    1. Primitive Data Structures
    2. Non-Primitive Data Structures

28.1 Types of Data Structures

28.1.1 Primitive Data Structure

- These are the data types defined by the C programming language. Primitive types can store only a single type of value. They are also known as system-defined data types. int, float, char, and double are examples of primitive data types.

28.1.2 Non-Primitive Data Structure

- These are data structures in C that are derived from primitive data types. Non-primitive data structures are also known as user-defined data types. They can store values of

multiple data types. Arrays, trees, stacks, and queues are some examples of user-defined data structures in C.

- Non-primitive data structures in C can be further classified into two categories:
    1. Linear data structure
    2. Non-linear data structure

28.1.2.1 Linear data structures

- Data structure in which data elements are arranged sequentially or linearly, where each element is attached to its previous and next adjacent elements is called a linear data structure.

  Example: Array, Stack, Queue, Linked List…..etc.
- In linear data structures also further classified as:
    1. **Static data structure:** It is a type of data structure where the size is allocated at the compile time. Therefore, the maximum size is fixed.

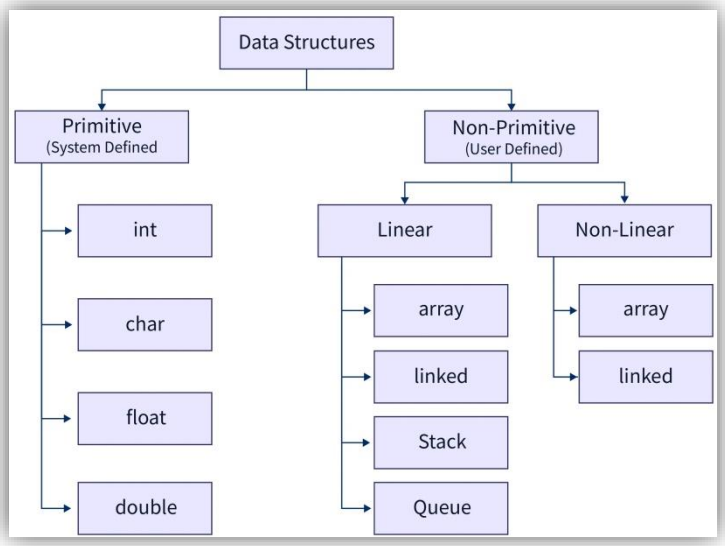
       Example: Array
    2. **Dynamic data structure:** It is a type of data structure where the size is allocated at the run time. Therefore, the maximum size is flexible.

       Example: Queue, Stack….etc.

28.1.2.2 Non-Linear data structures

- Data structures where data elements are not placed sequentially or linearly are called non-linear data structures. In a non-linear data structure, we can't traverse all the elements in a single run only.

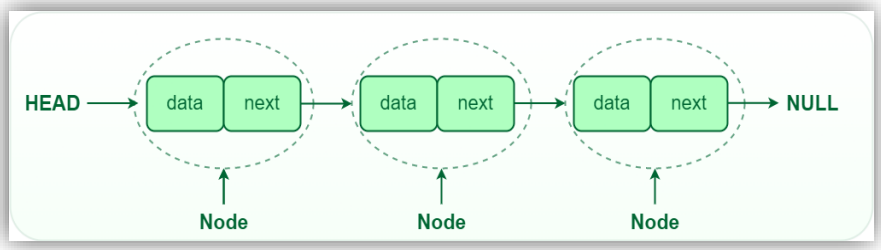
  Examples: Trees and Graphs



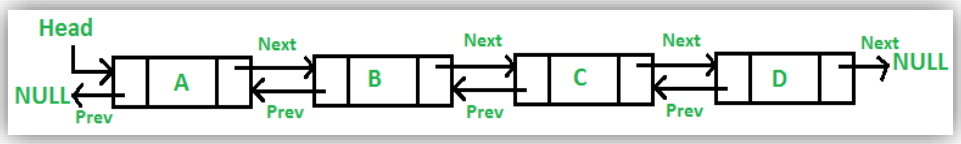
29 Linked List

- A Linked List is a linear data structure where elements are not stored in contiguous memory locations but are instead linked using pointers. It consists of a series of connected nodes, with each node containing data and a reference (pointer) to the next node.
- Linked Lists support three primary operations:
    - **Insert:** Adding or inserting a new node to the list.
    - **Delete:** Removing and returning a node from a specified position in the list.
    - **Traversal:** Iterating through the list to access each node.

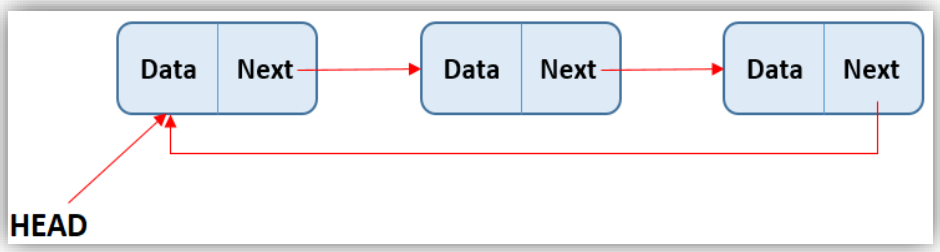
- There are three types of Linked Lists:
  - **Singly Linked List:** Each node contains a data field and a pointer to the next node. The pointer in the last node is set to null, indicating the end of the list.



  - **Doubly Linked List:** Each node contains a data field and two pointers: one pointing to the next node and the other to the previous node, allowing bidirectional traversal.



  - **Circular Linked List:** This can be either singly or doubly linked. In a circular linked list, the last node's pointer references the first node, forming a circular structure with no null pointers.



29.1 Array Vs Linked List

Feature	Array	Linked List
Structure	Elements are stored in a fixed-size, continuous block of memory	Elements are stored in nodes that are scattered in memory, with each node pointing to the next
Memory Allocation	Allocated either statically or dynamically but requires a continuous block of memory	Allocated dynamically, allowing it to grow or shrink as needed
Access Time	Accessing an element is fast, as you can directly use the index to reach any element	Accessing an element requires starting from the beginning and following the pointers to the desired element
Insertion/Deletion	Inserting or deleting elements, especially in the middle, is slow because it may require shifting other elements to maintain the continuous memory block	Inserting or deleting elements is generally faster since only the pointers need to be updated, with no need to shift other elements

Memory Utilization	Memory must be allocated upfront, which could lead to wasted space or insufficient space if the size changes unexpectedly	Memory is used efficiently, as it only allocates as many nodes as needed and can grow or shrink as necessary
Data Locality	Since all elements are stored together in memory, accessing them is generally faster due to better cache utilization	Elements are scattered in memory, which can result in slower access times as it doesn't benefit from cache locality
Extra Memory	No extra memory is needed for each element other than the space for the data itself	Each element requires extra memory to store pointers (to the next element, and possibly the previous one in a doubly linked list)
Traversal	You can directly jump to any element if you know its index	Must traverse nodes one by one
Use Cases	Best for when size is known and frequent access is needed	Best for when frequent insertions/deletions are needed

29.2 Self-referential Structure

- A self-referential structure in C is a structure that contains a pointer to another structure of the same type. This is often used in data structures like linked lists, trees, and graphs.\

Syntax

```
strut typename {
                    type var1;
                    type var2;
                    .......
                    .......
                    struct typename *var3;
                };
```

Example

```
#include <stdio.h>
struct mystruct {
                    int a;
                    struct mystruct *b;  // Added pointer to next struct
                };
int main() {
          //Create struct instances and pointers to them
           struct mystruct x = {10, NULL}, y = {20, NULL}, z = {30, NULL};
           struct mystruct *p1, *p2, *p3;

           // Assign pointers
           p1 = &x;
           p2 = &y;
           p3 = &z;

           // Set the 'b' field to point to the next struct
           x.b = p2;
           y.b = p3;

           // Print the details using correct format specifiers
```

```
    printf("Address of x: %p a: %d Address of next: %p\n", (void *)p1, x.a, (void
*)x.b);
    printf("Address of y: %p a: %d Address of next: %p\n", (void *)p2, y.a, (void
*)y.b);
    printf("Address of z: %p a: %d Address of next: %p\n", (void *)p3, z.a, (void
*)z.b);

    return 0;
}
```

Output

```
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/14.Linked_List/1.Single_Linked_List$ vi 5.Self_Referential.c
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/14.Linked_List/1.Single_Linked_List$ gcc 5.Self_Referential.c -o 5.Self_Referential
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/14.Linked_List/1.Single_Linked_List$ ./5.Self_Referential
Address of x: 0x7ffe223c43b0 a: 10 Address of next: 0x7ffe223c43c0
Address of y: 0x7ffe223c43c0 a: 20 Address of next: 0x7ffe223c43d0
Address of z: 0x7ffe223c43d0 a: 30 Address of next: (nil)
```

29.3 Single Linked List

29.3.1 Create the List

```
#include <stdio.h>
#include <stdlib.h>
//Define the structure for a node in the linked list
struct Node {
        int data;
        struct Node* next;
    };

struct Node* head = NULL;
// Function to create a linked list
struct Node* createLinkedList(int n) {
struct Node* tail = NULL;
struct Node* newnode = NULL;

for (int i = 1; i <= n; i++)
{
//Allocate memory for new node
    newnode = (struct Node*)malloc(sizeof(struct Node));
//Take input from the user
    printf("Enter data for node %d: ", i );
    scanf("%d", &newnode->data);
    newnode->next = NULL; // Set the next pointer of the new node to NULL
//If this is the first node, set it as the head
if (head == NULL)
        {
            head = newnode;
        }
else
    {
```

```
                        //Link it to the previous node
                        tail->next = newnode;
                }
        //Move tail to the current node
                tail = newnode;
        }// loop end
                return head;
        }//Function end
int main() {
                int nodes;

                //Input the number of nodes
                printf("Enter the number of nodes: ");
                scanf("%d", &nodes);
                printf("\n");
                //Create the linked list
                head = createLinkedList(nodes);
                //Display the linked list
                printf("The linked list is: ");
                displayLinkedList();

                return 0;
        }
```

29.3.2 Traverse and display the list

```
        void displayLinkedList()
        {
                struct Node* temp = head;
                //Traverse and display the linked list
                while (temp != NULL)
                {
                        printf("%d -> ", temp->data);
                        temp = temp->next;
                }
                printf("NULL\n");
        }
```

Output

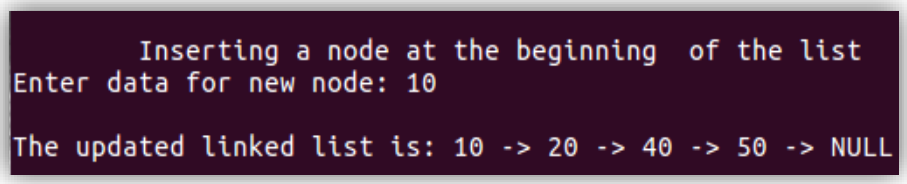
```
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/14.Linked_List/1.Single_Linked_List$ vi 1.Create_Display_Insert_List.c
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/14.Linked_List/1.Single_Linked_List$ gcc 1.Create_Display_Insert_List.c -o 1.Create_Display_Ins
ert_List
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/14.Linked_List/1.Single_Linked_List$ ./1.Create_Display_Insert_List
Enter the number of nodes: 3

Enter data for node 1: 20
Enter data for node 2: 40
Enter data for node 3: 50

The linked list is: 20 -> 40 -> 50 -> NULL
```

29.3.3 Insert a node at the beginning of the list

```
        void insertatbeginning ()
        {
                int newdata;
```

```
        struct Node *newnode;

        printf("\n\tInserting a node at the beginning  of the list\n");
        printf("Enter data for new node: ");
        scanf("%d",&newdata);

        newnode = (struct Node*)malloc(sizeof(struct Node));
        newnode->data = newdata;
        newnode->next = head;
        head = newnode;
        printf("\nThe updated linked list is: ");
        displayLinkedList();
}
```

Output



29.3.4 Insert a node at the end of the list

```
void insertatending()
{
        int newdata;
        struct Node *ptr,*newnode,*qtr=NULL;

        printf("\n\tInserting a node at the end of the list\n");
        printf("Enter data for new node: ");
        scanf("%d",&newdata);

        newnode = (struct Node*)malloc(sizeof(struct Node));
        newnode->data = newdata;
        newnode->next = NULL;
        ptr = head;

        while(ptr!=NULL)
        {
                qtr = ptr;
                ptr = ptr->next;
        }
        qtr->next = newnode;

        printf("\nThe updated linked list is: ");
        displayLinkedList(head);
}
```

Output

```
        Inserting a node at the end of the list
Enter data for new node: 60

The updated linked list is: 10 -> 20 -> 40 -> 50 -> 60 -> NULL
```

29.3.5 Insert a node at specified position in the list

```
void insertatposition()
{
        int position,newdata,i;
        struct Node *temp=NULL,*newnode;

        printf("\n\tInserting a node at specified position in the list\n");
        printf("\nEnter a position to insert a new node: ");
        scanf("%d",&position);
        printf("Enter data for new node: ");
        scanf("%d",&newdata);

        newnode = (struct Node*)malloc(sizeof(struct Node));
        newnode->data = newdata;
        if (position == 1)
        {
                newnode->next = head;
                head = newnode;
        }
        else
        {
                temp = head;
                for(i=1;i<position-1 ;i++)
                {
                        temp = temp->next;
                }
                newnode->next = temp->next;
                temp->next = newnode;
        }
        printf("\nThe updated linked list is: ");
        displayLinkedList(head);
}
```

Output

```
        Inserting a node at specified position in the list
Enter a position to insert a new node: 3
Enter data for new node: 30

The updated linked list is: 10 -> 20 -> 30 -> 40 -> 50 -> 60 -> NULL
```

29.3.6 Delete a node at the beginning of the list

```
void deleteatbeginning ()
{
        struct Node *temp;

        printf("\n\tDeleting a node at the beginning  of the list\n");

        temp = head;
        head = head->next;
        free(temp); // Free the memory of the deleted node
        temp = NULL;
        printf("\nThe updated linked list is: ");
        displayLinkedList();
}
```

Output

```
        Deleting a node at the beginning  of the list

The updated linked list is: 20 -> 30 -> 40 -> 50 -> 60 -> NULL
```

29.3.7 Delete a node at the end of the list

```
void deleteatending()
{
        struct Node *temp1,*temp2=NULL;

        printf("\n\tDeleting a node at the end of the list\n");

        temp1 = head;
        while(temp1->next != NULL)
        {
                temp2 = temp1;
                temp1 = temp1->next;
        }
        temp2->next = NULL; //Update the second last node to point to NULL
        free(temp1);    //Free the last node

        printf("\nThe updated linked list is: ");
        displayLinkedList(head);
}
```

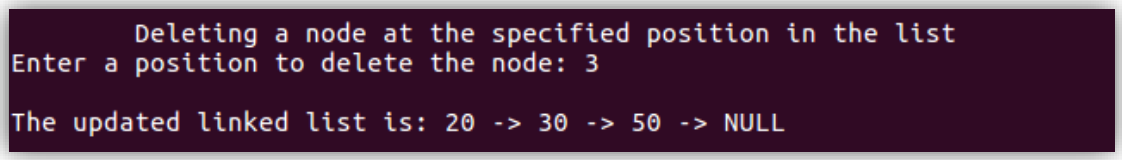
Output

```
        Deleting a node at the end of the list

The updated linked list is: 20 -> 30 -> 40 -> 50 -> NULL
```

29.3.8 Delete a node at a specified position in the list

```
void deleteatposition()
{
        int position,i;
        struct Node *temp=NULL,*toDelete=NULL;

        printf("\n\tDeleting a node at the specified position in the list\n");
        scanf("%d",&position);

        if(position ==1)
        {
                deleteatbeginning();    // Reuse the logic of deleting the first node
        }
        else
        {
                temp = head;


                for(i=1;i<position-1 ;i++)
                {
                        temp = temp->next;
                }
                toDelete = temp->next;
                temp->next = toDelete->next; // Bypass the node to delete
                free(toDelete); // Free memory
                toDelete = NULL;

                printf("\nThe updated linked list is: ");
                displayLinkedList(head);
        }
}
```

Output



29.3.9 Count total nodes in the list

```
void countLinkedList()
{
        int nodecount = 0;
        struct Node* temp = head;
        // Traverse, count and display the linked list
        while (temp != NULL)
        {
                nodecount++;
                printf("%d -> ", temp->data);
                temp = temp->next;
        }
```

```
            printf("NULL\n");
            printf("Number of nodes in the linked list: %d\n", nodecount);
        }
```

Output

```
The linked list is: 12 -> 34 -> 56 -> 78 -> 98 -> NULL
Number of nodes in the linked list: 5
```

29.3.10 Reverse the linked list

```
        struct Node* reverselinkedlist()
        {
            struct Node* prev = NULL;
            struct Node* current = head;
            struct Node* next = NULL;

            while (current != NULL)
            {
                next = current->next;  // Store the next node
                current->next = prev;  // Reverse the current node's pointer
                prev = current;  // Move the prev and current pointers one step
                                                    forward
                current = next;
            }
            return prev;      // New head of the reversed list
        }
```

Output

```
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/14.Linked_List/1.Single_Linked_List$ vi 4.Reverselist.c
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/14.Linked_List/1.Single_Linked_List$ gcc 4.Reverselist.c -o 4.Reverselist
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/14.Linked_List/1.Single_Linked_List$ ./4.Reverselist
Enter the number of nodes: 5

Enter data for node 1: 10
Enter data for node 2: 20
Enter data for node 3: 30
Enter data for node 4: 40
Enter data for node 5: 50

The linked list is: 10 -> 20 -> 30 -> 40 -> 50 -> NULL

The reversed linked list is: 50 -> 40 -> 30 -> 20 -> 10 -> NULL
```

29.4 Circular Linked List

29.4.1 Create the list

```
#include <stdio.h>
#include <stdlib.h>
//Define the structure for a node in the linked list
struct Node {
   int data;
   struct Node* next;
}*head = NULL,*tail = NULL,*newnode = NULL;
```

```
/*** Create a linked list ***/
struct Node* createLinkedList(int n)
{
	for (int i = 1; i <= n; i++)
	{
		// Allocate memory for a new node
		newnode = (struct Node*)malloc(sizeof(struct Node));
		// Take input from the user
		printf("Enter data for node %d: ", i);
		scanf("%d", &newnode->data);
		newnode->next = NULL; // Set the next pointer of the new node to NULL
		// If this is the first node, set it as the head
		if (head == NULL)
		{
			head = newnode;
			tail = newnode;
		}
		else
		{
			// Link it to the previous node
			tail->next = newnode;
			tail = newnode;
		}
	}
		// Make the list circular
		tail->next = head; // Set the last node's next to the head
		return head;
}

int main()
{
	int nodes;

	// Input the number of nodes
	printf("Enter the number of nodes: ");
	scanf("%d", &nodes);
	printf("\n");

	// Create the linked list
	head = createLinkedList(nodes);

	// Display the linked list
	printf("\nThe linked list is: ");
	displayLinkedList();

	return 0;
}
```

29.4.2 Traverse and display the list

```
void displayLinkedList()
{
```

```
        struct Node* temp = head;

        // Traverse and display the linked list
        do {
        printf("%d -> ",temp->data);
        temp = temp->next;
        } while (temp != head);

        printf("%d(Head)",temp->data); // To indicate the circular nature of the list
        printf("\n");
}
```

Output

```
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/14.Linked_List/2.Circular_Linked_List$ vi 1.Create_Insert.c
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/14.Linked_List/2.Circular_Linked_List$ gcc 1.Create_Insert.c -o 1.Create_Insert
sandeep@sandeep-VirtualBox:~/sandeep/C-Language/14.Linked_List/2.Circular_Linked_List$ ./1.Create_Insert
                Circular Linked List

Enter the number of nodes: 5

Enter data for node 1: 10
Enter data for node 2: 20
Enter data for node 3: 30
Enter data for node 4: 40
Enter data for node 5: 50

The linked list is: 10 -> 20 -> 30 -> 40 -> 50 -> 10(Head)
```

29.4.3 Insert a node at the beginning of the list

```
        void insertatbeginning()
        {
                int newdata;

                printf("\n\tInserting a node at the beginning  of the list\n");
                printf("\nEnter data for new node: ");
                scanf("%d",&newdata);

                newnode = (struct Node*)malloc(sizeof(struct Node));
                newnode->data = newdata;
                newnode->next = head;
                tail->next = newnode;
                head = newnode;
                printf("\nThe updated linked list is: ");
                displayLinkedList();
        }
```

Output

```
        Inserting a node at the beginning  of the list

Enter data for new node: 0

The updated linked list is: 0 -> 10 -> 20 -> 30 -> 40 -> 50 -> 0(Head)
```

29.4.4 Insert a node at the end of the list

```
void insertatending()
{
        int newdata;

        printf("\n\tInserting a node at the end of the list\n");
        printf("\nEnter data for new node: ");
        scanf("%d",&newdata);

        newnode = (struct Node*)malloc(sizeof(struct Node));
        newnode->data = newdata;
        tail->next = newnode;
        newnode->next = head;
        tail = newnode;
        printf("\nThe updated linked list is: ");
        displayLinkedList();
}
```

Output

```
        Inserting a node at the end of the list

Enter data for new node: 70

The updated linked list is: 0 -> 10 -> 20 -> 30 -> 40 -> 50 -> 70 -> 0(Head)
```

29.4.5 Insert a node at specified position in the list

```
void insertatposition()
{
        int position,newdata,i;
        struct Node *temp=NULL;

        printf("\n\tInserting a node at specified position in the list\n");
        printf("\nEnter a position to insert a new node: ");
        scanf("%d",&position);
        printf("Enter data for new node: ");
        scanf("%d",&newdata);

        newnode = (struct Node*)malloc(sizeof(struct Node));
        newnode->data = newdata;
        if (position == 1)
        {
                newnode->next = head;
                head = newnode;
        }
        else
        {
                temp = head;
                for(i=1;i<position-1 ;i++)
                {
                        temp = temp->next;
```

```
            }
            newnode->next = temp->next;
            temp->next = newnode;
        }
        printf("\nThe updated linked list is: ");
        displayLinkedList(head);
    }
```

Output

```
        Inserting a node at specified position in the list

Enter a position to insert a new node: 7
Enter data for new node: 60

The updated linked list is: 0 -> 10 -> 20 -> 30 -> 40 -> 50 -> 60 -> 70 -> 0(Head)
```

29.4.6 Delete a node at the beginning of the list

```
void deleteatbeginning ()
{
    struct Node *temp;

    printf("\n\tDeleting a node at the beginning  of the list\n");

    temp = head;
    head = head->next;
    tail->next = head;
    temp->next = NULL;

    free(temp); // Free the memory of the deleted node
    temp = NULL;

    printf("\nThe updated linked list is: ");
    displayLinkedList();
}
```

Output

```
        Deleting a node at the beginning  of the list

The updated linked list is: 10 -> 20 -> 30 -> 40 -> 50 -> 60 -> 70 -> 10(Head)
```

29.4.7 Delete a node at the end of the list***/

```
void deleteatending()
{
    struct Node *temp = NULL;

    printf("\n\tDeleting a node at the end of the list\n");

    temp = head;
```

```
        while(temp->next != tail)
        {
                temp = temp->next;
        }
        free(tail);
        tail = temp;
        temp->next = head;

        printf("\nThe updated linked list is: ");
        displayLinkedList(head);
}
```

Output

```
        Deleting a node at the end of the list

The updated linked list is: 10 -> 20 -> 30 -> 40 -> 50 -> 60 -> 10(Head)
```

29.4.8 Delete a node at a specified position in the list

```
void deleteatposition()
{
        int position,i;
        struct Node *temp=NULL,*toDelete=NULL;

        printf("\n\tDeleting a node at the specified position in the list\n");
        scanf("%d",&position);

        if(position ==1)
        {
                deleteatbeginning();    // Reuse the logic of deleting the first node
        }
        else
        {
                temp = head;
                for(i=1;i<position-1 ;i++)
                {
                        temp = temp->next;
                }
                toDelete = temp->next;
                temp->next = toDelete->next; // Bypass the node to delete
                free(toDelete); // Free memory
                toDelete = NULL;

                printf("\nThe updated linked list is: ");
                displayLinkedList(head);
        }
}
```

Output

```
        Deleting a node at the specified position in the list

Enter a position to delete the node: 6

The updated linked list is: 10 -> 20 -> 30 -> 40 -> 50 -> 10(Head)
```