2024

C PROGRAMMING

`

EMBEDDED LAB, SMEC

SMEC LABS | Kochi

Introduction

C programming is a general-purpose, procedural, imperative computer programming language developed in 1972 by Dennis M. Ritchie at the Bell Telephone Laboratories to develop the UNIX operating system. C is the most widely used computer language. It keeps fluctuating at number one scale of popularity along with Java programming language, which is also equally popular and most widely used among modern software programmers. It is developed for the UNIX operating system at Bell Labs.

The UNIX operating system, the C compiler, and essentially all UNIX application programs have been written in C. C has now become a widely used professional language for various reasons −

- Easy to learn
- Structured language
- It produces efficient programs
- It can handle low−level activities
- It can be compiled on a variety of computer platforms

Advantages of C Language

The following are the advantages of C language −

- Efficiency and speed − C is known for being high−performing and efficient. It can let you work with memory at a low level, as well as allow direct access to hardware, making it ideal for applications requiring speed and economical resource use.
- Portable − C programs can be compiled and executed on different platforms with minimal or no modifications. This portability is due to the fact that the language has been standardized and compilers are available for use on various operating systems globally.

- Close to Hardware – C allows direct manipulation of hardware through the use of pointers and low–level operations. This makes it suitable for system programming and developing applications that require fine-grained control over hardware resources.
- Standard Libraries – For common tasks such as input/output operations, string manipulation, and mathematical computations, C comes with a large standard library which helps developers write code more efficiently by leveraging pre–built functions.
- Structured Programming – C helps to organize code into modular and easy–to–understand structures. With functions, loops, and conditionals, developers can produce clear code that is easy to maintain.
- Procedural Language – C follows a procedural paradigm that is often simpler and more straightforward for some types of programming tasks.
- Versatility – C language is a versatile programming language and it can be used for various types of software such as system applications, compilers, firmware, application software, etc.

Drawbacks of C Language

The following are the disadvantages/drawbacks of C language –

- Manual Memory Management – C languages need manual memory management, where a developer has to take care of allocating and deallocating memory explicitly.
- No Object–Oriented Feature – Nowadays, most of the programming languages support the OOPs features. But the C language does not support it.
- No Garbage Collection – C language does not support the concept of Garbage collection. A developer needs to allocate and deallocate memory manually and this can be error-prone and lead to memory leaks or inefficient memory usage.
- No Exception Handling – C language does not provide any library for handling exceptions. A developer needs to write code to handle all types of expectations.

Applications of C Language

The following are the applications of C language –

- System Programming – C language is used to develop system software which are close to hardware such as operating systems, firmware, language translators, etc.
- Embedded Systems – C language is used in embedded system programming for a wide range of devices such as microcontrollers, industrial controllers, etc.
- Compiler and Interpreters – C language is very common to develop language compilers and interpreters.
- Database Systems – Since C language is efficient and fast for low-level memory manipulation. It is used for developing DBMS and RDBMS engines.
- Networking Software – C language is used to develop networking software such as protocols, routers, and network utilities.

History

C programming is a general-purpose, procedure-oriented programming language. It is both machine-independent and structured. C is a high-level programming language developed by Dennis Ritchie in the early 1970s. It is now one of the most popular and influential programming languages worldwide.

C is popular for its simplicity, efficiency, and versatility. It has powerful features including low-level memory access, a rich set of operators, and a modular framework.

Apart from its importance with respect to the evolution of computer programming technologies, the design of C language has a profound influence on most of the other programming languages that are in use today. The languages that are influenced by C include Java, PHP, JavaScript, C#, Python and many more. These languages have designed their syntax, control structures and other basic features from C.

C supports different hardware and operating systems due to its portability. Generally, it is considered as a basic language and influenced many other computer languages. It is most widely used in academia and industry. C's relevance and extensive acceptance make it crucial for prospective programmers.

The history of the C programming language is quite fascinating and pivotal in the development of computer science and software engineering.

Overview of C Language History

A brief overview of C language history is given below –

Origin of C Programming

'ALGOL' was the foundation or progenitor of programming languages. It was first introduced in 1960. 'ALGOL' was widely used in European countries. The ALGOL had introduced the concept of structured programming to the developer community. The year 1967 marked the introduction of a novel computer programming language known as 'BCPL', an acronym for Basic Combined Programming Language. BCPL was designed by Martin Richards in the mid-1960s.

Dennis Ritchie created C at Bell Laboratories in the early 1970s. It developed from an older language named B that Ken Thompson created. The main purpose of C's creation was to construct the Unix operating system, which was crucial in the advancement of contemporary computers. BCPL, B, and C all fit firmly in the traditional procedural family typified by Fortran and Algol 60. BCPL, B and C differ syntactically in many details, but broadly they are similar.

Development of C Programming

In 1971, Dennis Ritchie started working on C, and he and other Bell Labs developers kept improving it. The language is appropriate for both system programming and application development because it was made to be straightforward, effective, and portable.

Standardization of C Programming

Dennis Ritchie commenced development on C in 1971 and, in collaboration with other developers at Bell Labs, proceeded to refine it. The language was developed with portability, simplicity, and efficiency in mind, rendering it applicable to both application and system programming.

Basics

A C program basically consists of the following parts:

- Preprocessor Commands
- Functions Variables
- Statements & Expressions
- Comments

Let us look at a simple code that would print the words "Hello World":

```
#include <stdio.h>
int main()
 {
/* my first program in C */
printf("Hello, World! \n");
 return 0;
}
```

Let us take a look at the various parts of the above program:

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

2. The next line int main() is the main function where the program execution begins.

3. The next line /*...*/ will be ignored by the compiler and it has been put to add additional comments in the program. So such lines are called comments in the program.

4. The next line printf(.nother f..) is a function available in C which causes the message "Hello, World!" to be displayed on the screen.

5. The next line returns 0; terminates the main() function and returns the value 0.

Compile and Execute C Program

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

2. Save the file as hello.c

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

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

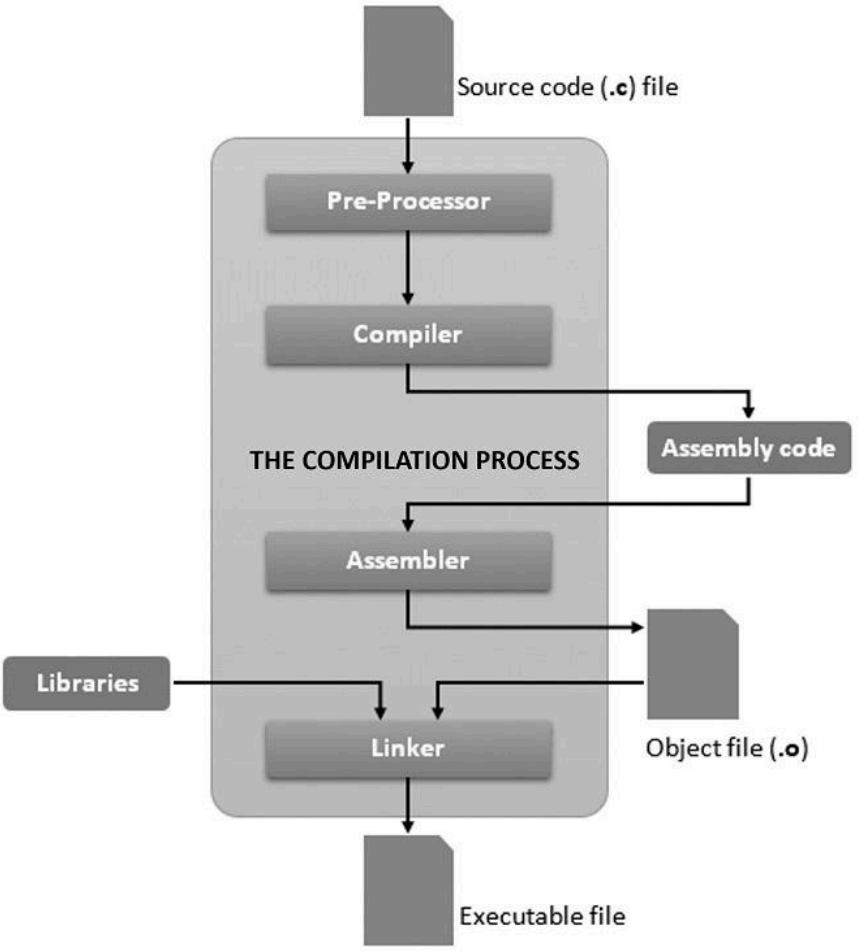
5. If there are no errors in your code, the command prompt will take you to the next line and will generate a.out executable file.

6. Now, type a.out to execute your program.

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

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

The following diagram illustrates the compilation process.



1. Preprocessing

The preprocessing stage handles directives (instructions) given to the preprocessor, which starts with `#`. This stage prepares the source code for the compilation by performing the following actions:

- Including Header Files: Replaces `#include` directives with the content of the specified header files.
- Macro Substitution: Replaces defined macros with their values using `#define`.
- Conditional Compilation: Handles `#if`, `#ifdef`, `#ifndef`, `#else`, and `#endif` to include or exclude parts of the code based on conditions.

Example:

```
#include <stdio.h>
#define PI 3.14
int main() {
    printf("Value of PI: %f\n", PI);
    return 0;
}
```

After preprocessing, the code will have the content of `stdio.h` included and `PI` replaced with `3.14`.

2. Compiling

In the compiling stage, the preprocessed source code is translated into assembly language code. This stage involves syntax and semantic checks to ensure the code adheres to the language rules.

Example:

```
int main() {
    printf("Hello, World!\n");
    return 0;
}
```

The compiler converts this into assembly language, generating a file with an `.s` extension (e.g., `main.s`).

3. Assembling

The assembling stage converts the assembly language code into machine code (binary instructions) specific to the target processor architecture. This stage produces an object file with an `.o` or `.obj` extension (e.g., `main.o`).

Example:

assembly

```
.globl _main
_main:
    pushq   %rbp
    movq    %rsp, %rbp
    leaq    L_.str(%rip), %rdi
    callq   _printf
    movl    $0, %eax
    popq    %rbp
    retq
L_.str:
    .asciz  "Hello, World!\n"
```

4. Linking

The linking stage combines one or more object files and libraries into a single executable file. It resolves references to external symbols (e.g., library functions like `printf`) and assigns final memory addresses to code and data.

Example Workflow:

```
#include <stdio.h>
int main()
 {
   printf("Hello, World! \n");
 return 0;
}
```

The workflow is:

1. Preprocessing: gcc -E main.c -o main.i (produces preprocessed file main.i).
2. Compiling: gcc -S main.i -o main.s (produces assembly file main.s).
3. Assembling: gcc -c main.s -o main.o (produces object file main.o).
4. Linking: gcc main.o -o main (produces executable file main).

Running ./main (on Unix-like systems) or main.exe (on Windows) will then print "Hello, World!" to the console.

4. Exercise

Write a C program to print the following personal details directly using `printf` statements:

- Name:
- Place:
- Date:
- Class:

Instructions:

1. Use `printf` to display each piece of information on a new line.
2. Ensure that each detail is printed with the appropriate label.

TOKENS

Tokens

A **token** is referred to as the smallest unit in the source code of a computer language such as C. Just as a certain piece of text in a language (like English) comprises words (collection of alphabets), digits, and punctuation symbols. A compiler breaks a C program into **tokens** and then proceeds ahead to the next stages used in the compilation process.

The first stage in the compilation process is a tokenizer. The tokenizer divides the source code into individual tokens, identifying the **token** type, and passing tokens one at a time to the next stage of the compiler.

The parser is the next stage in the compilation. It is capable of understanding the language's grammar. identifies syntax errors and translates an error-free program into the machine language.

A C source code also comprises **tokens** of different types. The tokens in C are of the following types –

- Character set
- Keyword tokens
- Literal tokens
- Identifier tokens
- Operator tokens
- Special symbol tokens

C Character set

The C language identifies a character set that comprises English alphabets – upper and lowercase (A to Z, as well as a to z), digits 0 to 9, and certain other symbols with a special meaning attached to them. In C, certain combinations of characters also have a special meaning attached to them. For example, \n is known as a newline character. Such combinations are called escape sequences.

Here is the character set of C language –

- Uppercase: A to Z

- Lowercase: a to z
- Digits: 0 to 9
- Special characters: ! " # $ % & ' ( ) * + - . : , ; ` ~ = < > { } [ ] ^ _ \ /

A sequence of any of these characters inside a pair of double quote symbols " and '' are used to represent a string literal. Digits are used to represent numeric literal. Square brackets are used for defining an array. Curly brackets are used to mark code blocks. Backslash is an escape character. Other characters are defined as operators.

C Keywords

In C, a predefined sequence of alphabets is called a keyword. Compared to human languages, programming languages have fewer keywords. To start with, C had 32 keywords, later on, few more were added in subsequent revisions of C standards. All keywords are in lowercase. Each keyword has rules of usage (in programming it is called syntax) attached to it.

The C compiler checks whether a keyword has been used according to the syntax, and translates the source code into the object code.

C Literals

In computer programming terminology, the term literal refers to a textual representation of a value to be assigned to a variable, directly hard-coded in the source code.

A numeric literal contains digits, a decimal symbol, and/or the exponentiation character E or e.

The string literal is made up of any sequence of characters put inside a pair of double quotation symbols. A character literal is a single character inside a single quote.

Arrays can also be represented in literal form by putting a comma-separated sequence of literals between square brackets. In C, escape sequences are also a type of literal. Two or more characters, the first being a backslash **** character,

put inside a single quote form an escape sequence. Each escape sequence has a predefined meaning attached to it.

C Identifiers

In contrast to the keywords, the identifiers are the user-defined elements in a program. You need to define various program elements by giving them an appropriate name. For example, variable, constant, label, user-defined type, function, etc.

There are certain rules prescribed in C, to form an identifier. One of the important restrictions is that a reserved keyword cannot be used as an identifier. For example, **for** is a keyword in C, and hence it cannot be used as an identifier, i.e., name of a variable, function, etc.

Valid C Identifiers

```
age, Age, AGE, average_age, __temp, address1,
phone_no_personal, _my_name
```

Invalid C Identifiers

```
Average-age, my name, $age, #phone, 1mg, phy+maths
```

Example:

```
#include <stdio.h>
int main () {
   /* variable definition: */
   int marks = 50;         //marks is an identifier
   printf("%d",marks);
  return 0;
}
```

Global Identifiers

If an identifier has been declared outside before the declaration of any function, it is called as an global (external) identifier.

Example:

```
#include <stdio.h>
int marks= 100; // external identifier
int main() {
printf("The value of marks is %d\n", marks);
}
```

Output ☑️

```
The value of marks is 100
```

Local Identifiers

On the other hand, an identifier inside any function is an local (internal) identifier.

Example:

```
#include <stdio.h>
int main() {
int marks= 100; // internal identifier
printf("The value of marks is %d\n", marks);
}
```

C Operators

C is a computational language. Hence a C program consists of expressions that perform arithmetic and comparison operations. The special symbols from the character set of C are mostly defined as operators. For example, the well-known symbols, **+**, **−**, ***** and **/** are the arithmetic operators in C. Similarly, **<** and **>** are used as comparison operators.

C Special symbols

Apart from the symbols defined as operators, the other symbols include punctuation symbols like commas, semicolons, and colons. In C, you find them used differently in different contexts.

Similarly, the parentheses **(** and **)** are used in arithmetic expressions as well as in function definitions. The curly brackets are employed to mark the scope of functions, code blocks in conditional and looping statements, etc.

Here consider the following example:

```
#include <stdio.h>
int main() {
    printf("Hello, World!\n");
    return 0;
}
```

Breakdown:

Keyword Tokens:

- `int` - Specifies the return type of the function.
- `return` - Exits the function and returns a value to the calling function.

Literal Tokens:

- `"Hello, World!\n"` - A string literal representing the text to be printed.
- `0` - An integer literal representing the return value of `main`.

Identifier Tokens:

- `main` - The name of the main function where execution starts.
- `printf` - The name of the standard library function used to print formatted output.

Operator Tokens:

- `(`, `)` - Parentheses used for function calls and grouping expressions.

Special Symbol Tokens:

- # - Preprocessor directive symbol.
- <, > - Angle brackets used for including header files.
- {, } - Curly braces used to define the beginning and end of function bodies.
- ; - Semicolon used to terminate statements.
- , - Comma used to separate elements in function calls.

KEYWORDS

Keywords

Keywords are those predefined words that have special meaning in the compiler and they cannot be used for any other purpose. As per the C99 standard, C language has 32 keywords. Keywords cannot be used as identifiers.

The following table has the list of all keywords (reserved words) available in the C language:

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

All the keywords in C have lowercase alphabets, although the keywords that have been newly added in C, do have uppercase alphabets in them. C is a

case-sensitive language. Hence, int is a keyword but INT, or Int are not recognized as a keyword. The new keywords introduced from C99 onwards start with an underscore character. The compiler checks the source code for the correctness of the syntax of all the keywords and then translates it into the machine code.

Example of C Keywords

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

 int a = 5;
 int b = 10;
 int sum;
 sum = a + b;

 if (sum > 10)
 {
  printf("Sum is greater than 10\n");
 }
 else
 {
  printf("Sum is 10 or less\n");
 }
 return 0;
}
```

Output ☑️

```
Sum is greater than 10
```

Explanation of Keywords Used:

1. **int**:
    - Used to declare integer variables (a, b, and sum).
2. **if**:
    - Used to make a decision based on a condition (if (sum > 10)).
3. **else**:
    - Used to specify an alternative path if the if condition is false.
4. **return**:
    - Used to return a value from the main function (return 0).
5. **#include**:
    - A preprocessor directive, not a keyword, but important to include standard headers like <stdio.h>.

IDENTIFIERS

Identifiers

A C identifier is a name used to identify a variable, function, or any other user defined item. An identifier starts with a letter A to Z, a to z, or an underscore '_' followed by zero or more letters, underscores, and digits (0 to 9).

C does not allow punctuation characters such as @, $, and % within identifiers. C is a case-sensitive programming language. Thus, Manpower and manpower are two different identifiers in C. Here are some examples of acceptable identifiers:

Here are examples of acceptable and not acceptable identifiers in C:

Acceptable Identifiers:

1. `variable`
2. `_temp`
3. `data1`
4. `count_`
5. `MAX_VALUE`
6. `minValue`
7. `numberOfItems`
8. `totalSum`
9. `flag`
10. `x`
11. `Y`
12. `user_input`
13. `speed2`
14. `_result`

Not Acceptable Identifiers:

1. `123abc` **(cannot start with a digit)**
2. `@name` **(cannot contain @)**
3. `var$` **(cannot contain $)**
4. `total%value` **(cannot contain %)**
5. `user-name` **(cannot contain -)**
6. `first name` **(cannot contain space)**
7. `void` **(reserved keyword in C)**
8. `int` **(reserved keyword in C)**
9. `double#value` **(cannot contain #)**

These examples illustrate the rules and restrictions for naming identifiers in C.

```
#include <stdio.h>
int main() {
    int number1 = 5;
    int number2 = 10;
    int sum;
    sum = number1 + number2;
    printf("The sum of %d and %d is %d\n",
number1,number2,sum);
    return 0;
}
```

Output ☑️

The sum of 5 and 10 is 15

Explanation:

1. **Identifiers for Variables**:
    - number1, number2, and sum are identifiers for integer variables.
2. **Identifiers for Functions**:
    - main is the identifier for the main function where execution starts.
    - printf is the identifier for the standard library function used to print to the console.

DATA TYPES

Data Types

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	8 bytes	-9223372036854775808 to 9223372036854775807
unsigned long	8 bytes	0 to 18446744073709551615

What is a Data Type?

- A data type is a classification that specifies which type of value a variable can hold. It determines the range of values that can be stored and the operations that can be performed on those values. Common data types in C include integers, floating-point numbers, characters, and more.

Size

- The size of a data type defines the amount of memory allocated to store a value of that type. The size varies between different data types and systems. For instance:
    - int typically occupies 4 bytes.
    - float usually occupies 4 bytes.
    - double generally occupies 8 bytes.
    - char usually occupies 1 byte.

Number System

- The number system refers to the way numbers are represented. Common number systems include:
    - **Binary (Base-2):** Uses digits 0 and 1.
    - **Octal (Base-8):** Uses digits 0 to 7.
    - **Decimal (Base-10):** Uses digits 0 to 9.
    - **Hexadecimal (Base-16):** Uses digits 0 to 9 and letters A to F.
- **Video Resource:** You can watch a detailed video on the number system here.

VARIABLE

Variable

A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in C has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable
To indicate the storage area, each variable should be given a unique name (identifier). Variable names are just the symbolic representation of a memory location. For example:

```
int age = 25;
```

Here, age is a variable of int type and we have assigned an integer value 25 to it.

Variable Definition in C

A variable definition tells the compiler where and how much storage to create for the variable. A variable definition specifies a data type and contains a list of one or more variables of that type as follows −

```
 type variable_list;
```

Here, **type** must be a valid C data type including char, w_char, int, float, double, bool, or any user-defined object; and **variable_list** may consist of one or more identifier names separated by commas.

Some valid variable declarations are shown here −

```
int    i, j, k;
char   c, ch;
float  f, salary;
double d;
```

The line **int i, j, k;** declares and defines the variables i, j, and k; which instruct the compiler to create variables named i, j and k of type **int**.

Variables can be initialized (assigned an initial value) in their declaration. The initializer consists of an equal sign followed by a constant expression as follows

```
type variable_name = value;
```

Variable Definition and Initialization:

```
// definition and initializing d and f
int d = 3, f = 5;
```

C User Input Function: The scanf()

The C language recognizes the standard input stream as **stdin** and is represented by the standard input device such as a keyboard. C always reads the data from the input stream in the form of characters.

The scanf() function converts the input to a desired data type with appropriate format specifiers.

Syntax of Scanf()

This is how you would use the scanf() function in C –

```
int scanf(const char *format, &var1, &var2, . . .);
```

The first argument to the scanf() function is a format string. It indicates the data type of the variable in which the user input is to be parsed. It is followed by one or more pointers to the variables. The variable names prefixed by & gives the address of the variable.

User Input Format Specifiers

Following format specifiers are used in the format string –

Format Specifier	**Type**
%c	Character
%d	Signed integer
%f	Float values
%i	Unsigned integer
%l or **%ld** or **%li**	Long
%lf	Double
%Lf	Long double
%lu	Unsigned int or unsigned long
%lli or **%lld**	Long long
%llu	Unsigned long long

Example of scanf():

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

```
    printf("Enter a character: ");
    scanf(" %c", &c);
    printf("You entered integer: %d\n", a);
    printf("You entered float: %.2f\n", b);
    printf("You entered character: %c\n", c);
    return 0;
}
```

Output ☑️

```
Enter an integer: 42
Enter a float: 3.14
Enter a character: A
You entered integer: 42
You entered float: 3.14
You entered character: A
```

OPERATORS

Operators

An operator is simply a symbol that is used to perform operations. There can be many types of operations like arithmetic, logical, bitwise, etc

An operator is simply a symbol that is used to perform operations. There can be many types of operations like arithmetic, logical, bitwise, etc

Depending on how many operands are required to perform the operation, operands are called as unary, binary or ternary operators. They need one, two or three operands respectively.

Unary operators – ++ (increment), -- (decrement), ! (NOT), ~ (compliment), & (address of), * (dereference)
Binary operators – arithmetic, logical and relational operators except !
Ternary operators – The ? operator

C divides the operators into the following groups:

- **Arithmetic operators**
- **Assignment operators**
- **Comparison operators**
- **Logical operators**
- **Bitwise operators**

We are most familiar with the arithmetic operators. These operators are used to perform arithmetic operations on operands. The most common arithmetic operators are addition (+), subtraction (-), multiplication (*), and division (/).

In addition, the modulo (%)

Arithmetic Operators

We are most familiar with the arithmetic operators. These operators are used to perform arithmetic operations on operands. The most common arithmetic operators are addition (+), subtraction (-), multiplication (*), and division (/).

In addition, the modulo (%)

Example, take a look at this simple expression −

```
a + b
```

The following table shows all the arithmetic operators supported by the C language. Assume variable **A** holds 10 and variable **B** holds 20 then −

Operator	Name	Description	Example
+	Addition	Adds together two values	x + y
-	Subtraction	Subtracts one value from another	x - y
*	Multiplication	Multiplies two values	x * y
/	Division	Divides one value by another	x / y

%	Modulus	Returns the division remainder	x % y
++	Increment	Increases the value of a variable by 1	++x
--	Decrement	Decreases the value of a variable by 1	--x

Example: Arithmetic Operators in C

The following example demonstrates how to use these arithmetic operators in a C program –

```
#include <stdio.h>
int main() {
    // Declare variables
    int a = 10, b = 5;
    int sum, difference, product, quotient, remainder;
    // Perform arithmetic operations
    sum = a + b;            // Addition
    difference = a - b;     // Subtraction
    product = a * b;        // Multiplication
    quotient = a / b;       // Integer Division
    remainder = a % b;      // Remainder (Modulus)
    a++; // Increment 'a' by 1
    b--; // Decrement 'b' by 1
```

```
    printf("Sum: %d\n", sum);
    printf("Difference: %d\n", difference);
    printf("Product: %d\n", product);
    printf("Quotient: %d\n", quotient);
    printf("Remainder: %d\n", remainder);
    printf("Incremented 'a': %d\n", a);
    printf("Decremented 'b': %d\n", b);

    return 0;
}
```

Output ☑️

```
Sum: 15
Difference: 5
Product: 50
Quotient: 2
Remainder: 0
Incremented 'a': 11
Decremented 'b': 4
```

Relational Operators

Relational operators in C are defined to perform comparison of two values. The familiar angular brackets **<** and **>** are the relational operators in addition to a few more as listed in the table below.

These relational operators are used in Boolean expressions. All the relational operators evaluate to either True or False.

For example, in the Boolean expression –

a > b

Here, ">" is a relational operator.

A list of all comparison operators:

Operators	Name	Example	Description
==	Equal to	x == y	Returns 1 if the values are equal
!=	Not equal	x != y	Returns 1 if the values are not equal
>	Greater than	x > y	Returns 1 if the first value is greater than the second value
<	Less than	x < y	Returns 1 if the first value is less than the second value
>=	Greater than or equal to	x >= y	Returns 1 if the first value is greater than, or equal to, the second value
<=	Less than or equal to	x <= y	Returns 1 if the first value is less than, or equal to, the second value

Example: Relational Operators in C

The following example demonstrates how to use these arithmetic operators in a C program –

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

    int a = 5, b = 10;
```

```
    printf("%d > %d is %d\n", a, b, a > b);

    printf("%d < %d is %d\n", a, b, a < b);

    printf("%d >= %d is %d\n", a, b, a >= b);

    printf("%d <= %d is %d\n", a, b, a <= b);

    printf("%d == %d is %d\n", a, b, a == b);

    printf("%d != %d is %d\n", a, b, a != b);

    return 0;
}
```

Output ☑️

```
5 > 10 is 0
5 < 10 is 1
5 >= 10 is 0
5 <= 10 is 1
5 == 10 is 0
5 != 10 is 1
```

Logical Operators

These operators are used to combine two or more boolean expressions. We can form a compound Boolean expression by combining Boolean expression with these operators. An example of logical operator is as follows –

The logical AND operator (**&&**) and the logical OR operator (**||**) are both binary in nature (require two operands). The logical NOT operator (**!**) is a unary operator.

Here is a table showing the logical operators in C –

Operator	Name	Example	Description
&&	AND	x < 5 && x < 10	Returns 1 if both statements are true
||	OR	x < 5 || x < 4	Returns 1 if one of the statements is true
!	NOT	!(x < 5 && x < 10)	Reverse the result, returns 0 if the result is 1

The result of a logical operator follows the principle of Boolean algebra. The logical operators follow the following truth tables.

Logical AND (&&) Operator

The && operator in C acts as the logical AND operator. It has the following truth table –

a	**b**	**a&&b**

true	true	True
true	false	False
false	true	False
false	false	False

Logical OR (||) Operator

C uses the double pipe symbol (||) as the logical OR operator. It has the following truth table –

a	b	a||b
true	true	True
true	false	True
false	true	true
false	false	false

The above truth table shows that the result of **||** operator is True when either of the operands is True, and False if both operands are false.

Logical NOT (!) Operator

The logical NOT **!** operator negates the value of a Boolean operand. True becomes False, and False becomes True. Here is its truth table –

A	!a
True	False

False	True

Unlike the other two logical operators **&&** and **||**, the logical NOT operator **!** is a unary operator.

Example: Logical Operators in C

The following example demonstrates how to use these logical operators in a C program −

```
#include <stdio.h>
int main() {
    int a = 1, b = 0;
   // Logical operators
   printf("%d && %d is %d\n", a, b, a && b);   // Logical AND
   printf("%d || %d is %d\n", a, b, a || b);   // Logical OR
   printf("!%d is %d\n", a, !a);               // Logical NOT
    return 0;
}
```

Output ☑️

```
1 && 0 is 0
1 || 0 is 1
!1 is 0
```

Bitwise Operators

Bitwise operators in C allow low-level manipulation of data stored in the computer's memory.

Bitwise operators contrast with logical operators in C. For example, the logical AND operator (**&&**) performs AND operation on two Boolean expressions, while

the bitwise AND operator (**&**) performs the AND operation on each corresponding bit of the two operands.

For the three logical operators **&&**, **||**, and **!**, the corresponding bitwise operators in C are **&**, **|** and **~**.

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B)
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B)
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B)
~	Binary One's Complement Operator is unary and has the effect of 'flipping' bits.	(~A
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2

Additionally, the symbols **^** (XOR), **<<** (left shift) and **>>** (right shift) are the other 4 bitwise operators.

Bitwise AND Operator (&) in C

The bitwise AND (&) operator performs as per the following truth table −

bit a	bit b	a & b

0	0	0
0	1	0
1	0	0
1	1	1

Bitwise binary AND performs logical operation on the bits in each position of a number in its binary form.

Assuming that the two int variables "a" and "b" have the values 60 (equivalent to 0011 1100 in binary) and 13 (equivalent to 0000 1101 in binary), the "a & b" operation results in 13, as per the bitwise ANDing of their corresponding bits illustrated below –

```
 0011 1100
& 0000 1101
  ---------
= 0000 1100
```

The binary number 00001100 corresponds to 12 in decimal.

Bitwise OR (|) Operator

The bitwise OR (|) operator performs as per the following truth table –

bit a	**bit b**	**a | b**

0	0	0
0	1	1
1	0	1
1	1	1

Bitwise binary OR performs logical operation on the bits in each position of a number in its binary form.

Assuming that the two int variables "a" and "b" have the values 60 (equivalent to 0011 1100 in binary) and 13 (equivalent to 0000 1101 in binary), then "**a | b**" results in 61, as per the bitwise OR of their corresponding bits illustrated below −

```
 0011 1100

| 0000 1101

  ---------

= 0011 1101
```

The binary number 00111101 corresponds to 61 in decimal.

Bitwise XOR (^) Operator

The bitwise XOR (^) operator performs as per the following truth table −

bit a	bit b	a ^ b
0	0	0
0	1	1
1	0	1
1	1	0

Bitwise binary XOR performs logical operation on the bits in each position of a number in its binary form. The XOR operation is called "exclusive OR".

Note: The result of XOR is 1 if and only if one of the operands is 1. Unlike OR, if both bits are 1, XOR results in 0.

Assuming that the two int variables "a" and "b" have the values 60 (equivalent to 0011 1100 in binary) and 13 (equivalent to 0000 1101 in binary), the "**a ^ b**" operation results in 49, as per the bitwise XOR of their corresponding bits illustrated below –

```
 0011 1100
^ 0000 1101
 ---------
= 0011 0001
```

The binary number 00110001 corresponds to 49 in decimal.

The Left Shift (<<) Operator

The left shift operator is represented by the **<<** symbol. It shifts each bit in its left-hand operand to the left by the number of positions indicated by

the right-hand operand. Any blank spaces generated while shifting are filled up by zeroes.

Assuming that the int variable "a" has the value 60 (equivalent to 0011 1100 in binary), the "**a << 2**" operation results in 240, as per the bitwise left-shift of its corresponding bits illustrated below –

```
0011 1100 << 2 = 1111 0000
```

The binary number 11110000 corresponds to 240 in decimal.

The Right Shift (>>) Operator

The right shift operator is represented by the **>>** symbol. It shifts each bit in its left-hand operand to the right by the number of positions indicated by the right-hand operand. Any blank spaces generated while shifting are filled up by zeroes.

Assuming that the int variable a has the value 60 (equivalent to 0011 1100 in binary), the "**a >> 2**" operation results in 15, as per the bitwise right-shift of its corresponding bits illustrated below –

```
0011 1100 >> 2 = 0000 1111
```

The binary number 00001111 corresponds to 15 in decimal.

The 1's Complement (~) Operator

The 1's complement operator (~) in C is a unary operator, needing just one operand. It has the effect of "flipping" the bits, which means 1s are replaced by 0s and vice versa.

a	~a
0	1
1	0

Assuming that the int variable "a" has the value 60 (equivalent to 0011 1100 in binary), then the "**~a**" operation results in -61 in 2's complement form, as per the bitwise right-shift of its corresponding bits illustrated below –

```
~ 0011 1100 = 1100 0011
```

The binary number 1100 0011 corresponds to -61 in decimal.

Example: Bitwise Operators in C

The following example demonstrates how to use these logical operators in a C program –

```
#include <stdio.h>

int main() {
    unsigned int a = 5;   // Binary: 0000 0101
    unsigned int b = 3;   // Binary: 0000 0011
    unsigned int result;

   // Bitwise AND
    result = a & b;       // 0000 0101 & 0000 0011 = 0000 0001
    printf("a & b = %u\n", result);
     // Bitwise OR
```

```
    result = a | b;        // 0000 0101 | 0000 0011 = 0000 0111
    printf("a | b = %u\n", result);

    // Bitwise XOR
    result = a ^ b;       // 0000 0101 ^ 0000 0011 = 0000 0110
    printf("a ^ b = %u\n", result);

    // Bitwise NOT (Unary)
    result = ~a;     // ~0000 0101 = 1111 1010 (2's complement)
    printf("~a = %u\n", result);

    // Left Shift
    result = a << 1;       // 0000 0101 << 1 = 0000 1010
    printf("a << 1 = %u\n", result);

    // Right Shift
    result = a >> 1;       // 0000 0101 >> 1 = 0000 0010
    printf("a >> 1 = %u\n", result);
    return 0;
}
```

Output ☑️

```
a & b = 1
a | b = 7
a ^ b = 6
~a = 4294967290
a << 1 = 10
a >> 1 = 2
```

Assignment Operators

In C language, the assignment operator stores a certain value in an already declared variable. A variable in C can be assigned the value in the form of a literal, another variable, or an expression.

The value to be assigned forms the right-hand operand, whereas the variable to be assigned should be the operand to the left of the "=" symbol, which is defined as a simple assignment operator in C.

The following table lists the assignment operators supported by the C language –

Operator	**Example**	**Same As**
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
&=	x &= 3	x = x & 3

|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Example: Assignment Operators in C

The following example demonstrates how to use these logical operators in a C program −

```
#include <stdio.h>

int main(){
   int a = 21;
   int c ;
 c =  a;
 printf("Line 1 - =  Operator Example, Value of c = %d\n", c );
 c +=  a;
 printf("Line 2 - += Operator Example, Value of c = %d\n", c );
 c -=  a;
 printf("Line 3 - -= Operator Example, Value of c = %d\n", c );
 c *=  a;
 printf("Line 4 - *= Operator Example, Value of c = %d\n", c );
 c /=  a;
 printf("Line 5 - /= Operator Example, Value of c = %d\n", c );
 c  = 200;
```

```
 printf("Line 6 - %%= Operator Example, Value of c = %d\n", c );
c <<=  2;
 printf("Line 7 - <<= Operator Example, Value of c = %d\n", c );
 c >>=  2;
printf("Line 8 - >>= Operator Example, Value of c = %d\n", c );
 c &=  2;
 printf("Line 9 - &= Operator Example, Value of c = %d\n", c );
 c ^=  2;
 printf("Line 10 - ^= Operator Example, Value of c = %d\n", c );
 c |=  2;
 printf("Line 11 - |= Operator Example, Value of c = %d\n", c );
   return 0;
}
```

Output ☑️

```
Line 1 - =  Operator Example, Value of c = 21

Line 2 - += Operator Example, Value of c = 42

Line 3 - -= Operator Example, Value of c = 21

Line 4 - *= Operator Example, Value of c = 441

Line 5 - /= Operator Example, Value of c = 21

Line 6 - %= Operator Example, Value of c = 11

Line 7 - <<= Operator Example, Value of c = 44

Line 8 - >>= Operator Example, Value of c = 11

Line 9 - &= Operator Example, Value of c = 2

Line 10 - ^= Operator Example, Value of c = 0

Line 11 - |= Operator Example, Value of c = 2
```

Unary operator

Some operators in C are binary as well as unary in their usage. Examples of unary operators in C include ++, --, !, etc.

The Increment Operator in C

The increment operator (++) adds 1 to the value of its operand variable and assigns it back to the variable.

The statement "a++;" is equivalent to writing "a = a +1;"

prefix increment

In this method, the operator precedes the operand (e.g., ++a). The value of the operand will be altered *before* it is used.

Example:

```
int a = 1;
int b = ++a;  // b = 2
```

postfix increment

In this method, the operator follows the operand (e.g., a++). The value operand will be altered *after* it is used.

Example:

```
int a = 1;
int b = a++;   // b = 1
int c = a;     // c = 2
```

The Decrement Operator in C

The decrement operator (--) subtracts 1 from the value of its operand variable and assigns it back to the variable.

The statement "a--;" is equivalent to writing "a = a - 1;"

Prefix decrement

In this method, the operator precedes the operand (e.g., – -a). The value of the operand will be altered *before* it is used.

```
int a = 1;
int b = --a;  // b = 0
```

Example:

```
int main()
{
int a=10;
a++;
printf("%d",a);
}
```

Postfix Decrement

In this method, the operator follows the operand (e.g., a- -). The value of the operand will be altered *after* it is used.

```
int a = 1;
int b = a--;   // b = 1
int c = a;     // c = 0
```

Example:

```
int main()
{

int a=10;
++a;
printf("%d",a);//
}
```

NOT (!)

The logical NOT operator (!) is used to reverse the logical state of its operand. If a condition is true, then the Logical NOT operator will make it false.

Logic:

```
If x is true, then !x is false
  If x is false, then !x is tru
```

Example:

```
#include<stdio.h>
int main()
{
    int a = 10;
    int b = 5;
    int c;
    c=!(a > b);
        printf("%d",c);
    return 0;
}
```

Output ☑️

0

Address of operator (&)

The **address of operator (&)** gives an address of a variable. It is used to return the memory address of a variable. These addresses returned by the address-of operator are known as pointers because they "point" to the variable in memory.

```
//& gives an address on variable n
   int a;
   int *ptr;
   ptr = &a; // address of a is copied to the location ptr.
```

Example:

```
#include <stdio.h>
int main()
{
    int a = 20;
    printf("Address of a = %p", &a);
     return 0;
}
```

Output ☑️

Address of a = 0061FF1C

sizeof()

This operator returns the size of its operand, in bytes. The *sizeof() operator* always precedes its operand. The operand is an expression, or it may be a cast.

```
#include <stdio.h>

int main()
{
    // printing the size of double and int using sizeof
    printf("Size of double: %d\n", sizeof(double));
    printf("Size of int: %d\n", sizeof(int));

    return 0;
}
```

Output ☑️

```
Size of double: 8
Size of int: 4
```

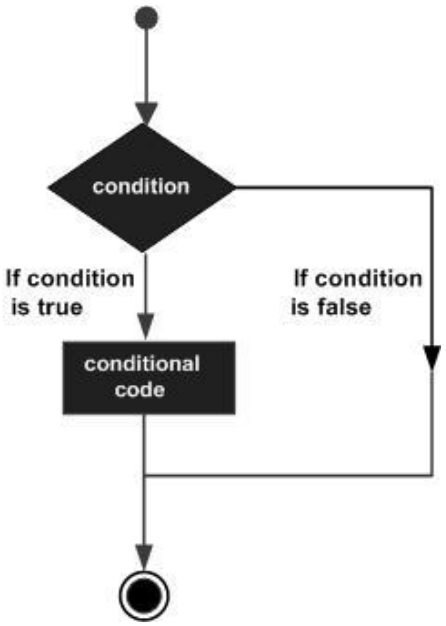
DECISION MAKING

Decision Making Statements

Decision-making statements, in the context of programming, refer to constructs that allow a program to make decisions based on certain conditions or expressions. These statements are crucial for controlling the flow of execution in a program, enabling it to perform different actions or follow different paths depending on the input or state of the program at runtime.

if Statement

The if statement in C is a fundamental decision-making statement that allows you to execute a block of code if a specified condition is true. Here's a detailed explanation and example of how if statements work:



Syntax:

```
if (condition) {
    // Code to be executed if condition is true
}
```

Condition: The condition inside the parentheses (()) is an expression that evaluates to either true or false. If the condition is true, the block of code inside the curly braces ({}) is executed. If the condition is false, the block of code is skipped, and execution continues with the next statement after the if block.

Example 1:

```
#include <stdio.h>

int main() {
    int num = 10;

    // Example of if statement
    if (num > 0) {
        printf("%d is a positive number.\n", num);
    }

    return 0;
}
```

Output ☑️

```
10 is a positive number.
```

Example 2:

```
#include <stdio.h>

int main() {
    int number;
 // Prompt the user to enter a number
    printf("Enter an integer: ");
    scanf("%d", &number);
    if (number > 0) {
```

```
        printf("The number is positive.\n");
    }
    if (number < 0) {
        printf("The number is negative.\n");
    }

    if (number == 0) {
        printf("The number is zero.\n");
    }

    return 0;
}
```

Output ☑️

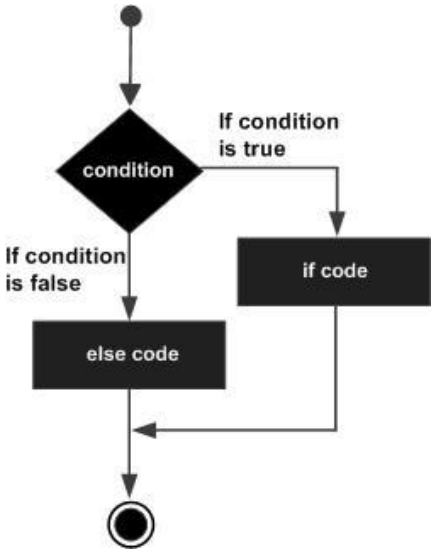
```
Enter an integer: 5
The number is positive.


Enter an integer: -3
The number is negative.


Enter an integer: 0
The number is zero.
```

Explanation of the example:

- **Condition**: num > 0 checks if the variable num is greater than zero.
- **Execution**: Since num is 10, which is greater than 0, the condition is true, and the printf statement inside the if block executes, printing 10 is a positive number. to the console.

if..else Statement

The if–else statement offers an alternative path when the condition isn't met.



Syntax:

```
if (Boolean expr){
   expression;
   . . .
}
else{
   expression;
   . . .
}
```

An **if** statement can be followed by an optional **else** statement, which executes when the Boolean expression is false.

Example to demonstrate if ... else

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

```
  // Prompt user for input
    printf("Enter an integer: ");
    scanf("%d", &number);

    // Check if the number is positive
    if (number > 0) {
        printf("The number is positive.\n");
    }
    // If the number is not positive, check if it's negative
    else {
        printf("The number is zero or negative.\n");
    }

    return 0;
}
```

Output ☑️

Enter an integer: 5
The number is positive.

Enter an integer: -3
The number is zero or negative.

Enter an integer: 0
The number is zero or negative.

If..else if..else Statement

An if statement can be followed by an optional else if...else statement, which is very useful to test various conditions using single if...else if statement. If a condition is true, run the given block that follows. If it

isn't, run the next block instead. However, if none of the above is true and all else fails, finally run another block. In such cases, you would use an **else-if** clause.

Syntax:

```
if (condition){
  // if the condition is true,
  // then run this code
} else if(another_condition){
  // if the above condition was false
  // and this condition is true,
  // then run the code in this block

} else{
  // if both the above conditions are false,
  // then run this code
}
```

Example: Program to find if the entered number is positive or not

```c
#include <stdio.h>

int main() {
   int number;

   // Prompt the user to enter a number
   printf("Enter an integer: ");
   scanf("%d", &number);
   // Check if the number is positive, negative, or zero
   if (number > 0) {
      printf("The number is positive.\n");
   } else if (number < 0) {
      printf("The number is negative.\n");
   } else {
      printf("The number is zero.\n");
   }
 return 0;
}
```

Output ☑️ :

```
Enter an integer: 5
The number is positive.

Enter an integer: -3
The number is negative.

Enter an integer: 0
The number is zero.
```

Nested If Statement

It is always legal in C programming to nest **if-else** statements, which means you can use one **if** or **else-if** statement inside another **if** or **else-if** statement(s).

In the programming context, the term "nesting" refers to enclosing a particular programming element inside another similar element. For example, nested loops, nested structures, nested conditional statements,

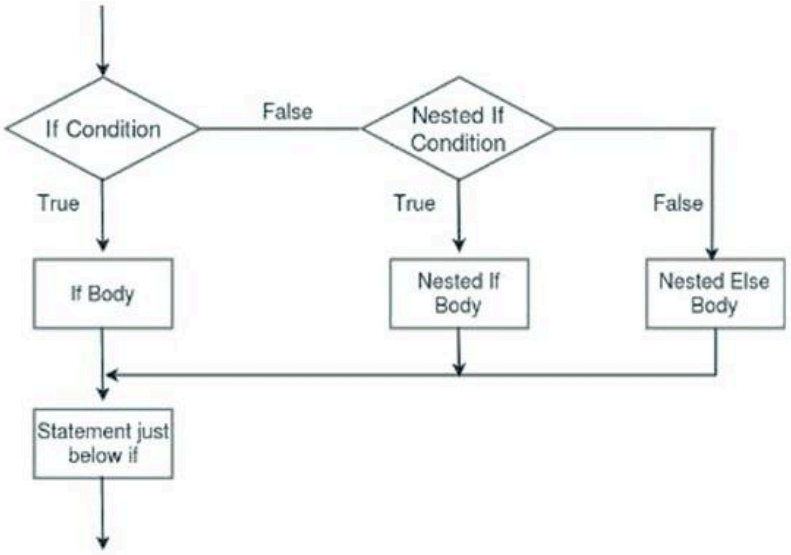
etc. If an **if** statement in C is employed inside another **if** statement, then we call it as a **nested if statement** in C.

Syntax:

The syntax of **nested if** statements is as follows –

```
if (expr1){
  if (expr2){
    block to be executed when expr1 and expr2 are true
  }
  else{
    block to be executed when expr1 is true but expr2 is false
  }
}
```

The following flowchart represents the nesting of **if** statements –



You can compound the Boolean expressions with **&&** or **||** to get the same effect. However, for more complex algorithms, where there are different combinations of multiple Boolean expressions, it becomes difficult to form the compound conditions. Instead, it is recommended to use nested structures.

Another **if** statement can appear inside a top-level **if** block, or its **else** block, or inside both.

Example 1: Here we express a logic for finding if a given number is less than 100, between 100 to 200, or above 200

```
#include <stdio.h>
int main (){
  // local variable definition
  int a = 274;
  printf("Value of a is : %d\n", a);
  if (a < 100){
    printf("Value of a is less than 100\n");
  }
 if (a >= 100 && a < 200){
 printf("Value of a is between 100 and 200\n" );
}
```

```
if (a >= 200){
    printf("Value of a is more than 200\n" );
  }
}
```

Output ☑️

```
Value of a is : 274
Value of a is more than 200
```

Example 1:Here let's use nested conditions for the same problem. It will make the solution more understandable when we use nested conditions.

First, check if "a >= 100". Inside the true part of the if statement, check if it is <200 to decide if the number lies between 100-200, or it is >200. If the first condition (a >= 100) is false, it indicates that the number is less than 100.

```
#include <stdio.h>
 int main (){
  // local variable definition
  // check with different values 120, 250 and 74
  int a = 120;
  printf("value of a is : %d\n", a );

  // check the boolean condition
  if(a >= 100){
    // this will check if a is between 100-200
   if(a < 200){
       // if the condition is true, then print the following
       printf("Value of a is between 100 and 200\n" );
      }
     else{
 printf("Value of a is more than 200\n");
 }
}
```

```
  else{
     // executed if a < 100
     printf("Value of a is less than 100\n");
  }

  return 0;
}
```

Output ☑️

Value of a is : 120
Value of a is between 100 and 200

Exercise

Write a C program that checks whether a given year is a leap year. A year is a leap year if:

1. It is divisible by 4, but not divisible by 100, or
2. It is divisible by 400.

Switch Statement

A switch statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each switch case.

Switch-case statement

The **switch-case** statement is a decision-making statement in C. The **if-else** statement provides two alternative actions to be performed, whereas the **switch-case** construct is a multi-way branching statement. A **switch** statement in C simplifies multi-way choices by evaluating a single variable against multiple values, executing specific code based on the match. It allows a variable to be tested for equality against a list of values.

Syntax:

The flow of the program can switch the line execution to a branch that satisfies a given case. The schematic representation of the usage of switch-case construct is as follows –

```
switch(expression)
{
   case constant-expression :
         statement(s);
         break; /* optional */
   case constant-expression :
         statement(s);
         break; /* optional */

   /* you can have any number of case statements */

   default : /* Optional */
         statement(s);
```

How does switch-case statement work?:

The parenthesis in front of the **switch** keyword holds an expression. The expression should evaluate to an integer or a character. Inside the curly brackets after the parenthesis, different possible values of the expression form the case labels.

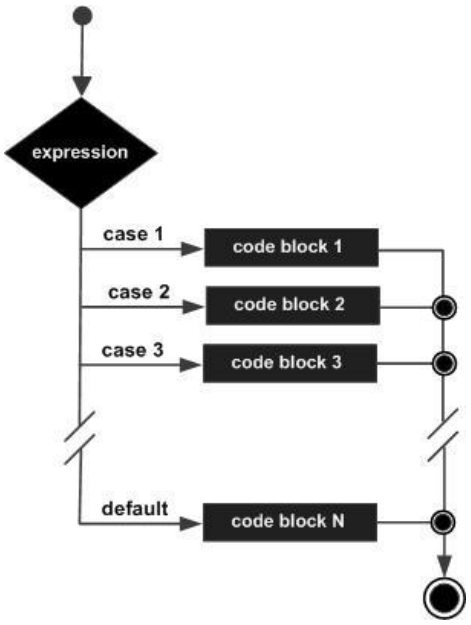
One or more statements after a colon(:) in front of the case label forms a block to be executed when the expression equals the value of the label.

You can literally translate a switch-case as "in case the expression equals value1, execute the block1", and so on.

C checks the expression with each label value, and executes the block in front of the first match. Each case block has a **break** as the last statement. The break statement takes the control out of the scope of the switch construct.

You can also define a **default case** as the last option in the switch construct. The default case block is executed when the expression doesn’t match with any of the earlier case values.

The flowchart that represents the switch-case construct in C is as follows –



Rules for using switch - case:

The following rules apply to a switch statement –

- The expression used in a **switch** statement must have an integral or enumerated type, or be of a class type in which the class has a single conversion function to an integral or enumerated type.
- You can have any number of **case** statements within a switch. Each **case** is followed by the value to be compared to and a colon.
- The constant-expression for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a break statement is reached.
- When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a break. If no break appears, the flow of control will fall through to subsequent cases until a break is reached.
- A switch statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a

task when none of the cases is true. No break is needed in the default case.

The **switch-case statement** acts as a compact alternative to cascade of if-else statements, particularly when the Boolean expression in "if" is based on the "=" operator.

If there are more than one statements in an **if** (or **else**) block, you must put them inside curly brackets. Hence, if your code has many **if** and **else** statements, the code with that many opening and closing curly brackets appears clumsy. The switch-case alternative is a compact and clutter-free solution.

Example 1:To print three different greeting messages based on the value of a "ch" variable ("m", "a" or "e" for morning, afternoon or evening).

```
#include <stdio.h>
 nt main (){
  // local variable definition
  char ch = 'm';
  printf("Time code: %c\n\n", ch);
  switch (ch){
    case 'a':
      printf("Good Afternoon\n");
      break;
    case 'e':
      printf("Good Evening\n");
      break;
    case 'm':
      printf("Good Morning\n");
  }
  return 0;
}
```

Output ☑️

Time code: m

Good Morning

Note

The use of break is very important here. The block of statements corresponding to each case ends with a break statement. What if the break statement is not used?

The switch-case works like this: As the program enters the switch construct, it starts comparing the value of switching expression with the cases, and executes the block of its first match. The break causes the control to go out of the switch scope. If a break is not found, the subsequent block also gets executed, leading to incorrect results.

Example 2:To print three different greeting messages without **break.**

```
#include <stdio.h>
int main (){
  /* local variable definition */
  char ch = 'a';
  printf("Time code: %c\n\n", ch);
```

```
  switch (ch){
    case 'a':
      printf("Good Afternoon\n");
      // break;

    case 'e':
      printf("Good Evening\n");
      // break;

    case 'm':
      printf("Good Morning\n");
  }
  return 0;
}
```

Output ☑️

```
Time code: a

Good Afternoon
Good Evening
Good Morning
```

You expect the "Good Morning" message to be printed, but you find all the three messages printed!

This is because C falls through the subsequent case blocks in the absence of **break** statements at the end of the blocks.

Example:

```
#include <stdio.h>
int main(){
  /* local variable definition */
  char grade = 'B';
```

```
  switch(grade){
    case 'A' :
       printf("Outstanding!\n" );
       break;
    case 'B':
      printf("Excellent!\n");
      break;
    case 'C':
       printf("Well Done\n" );
       break;
    case 'D':
       printf("You passed\n" );
       break;
    case 'F':
       printf("Better try again\n" );
       break;
    default :
       printf("Invalid grade\n" );
  }
  printf("Your grade is  %c\n", grade);

  return 0;
}
```

Output ☑️

```
Excellent!
Your grade is  B
```

Nested `switch` Statement

It is possible to have a **switch** as a part of the statement sequence of an outer **switch**. Even if the case constants of the inner and outer **switch** contain common values, no conflicts will arise.You can have nested switch-case constructs. You may have a different switch-case construct

each inside the code block of one or more case labels of the outer switch scope.

Syntax:

```
switch(ch1){
  case 'A':
  printf("This A is part of outer switch" );
  switch(ch2) {
    case 'A':
      printf("This A is part of inner switch" );
      break;
    case 'B':  /* case code */
  }
  break;
  case 'B':  /* case code */
}
```

Example 1:

```
#include <stdio.h>
int main (){
  /* local variable definition */
  int a = 100;
  int b = 200;
  switch(a){
    case 100:
    printf("This is part of outer switch\n", a);
switch(b){
      case 200:
      printf("This is part of inner switch\n", a);
    }
      }
  printf("Exact value of a is: %d\n", a);
  printf("Exact value of b is: %d\n", b);
  return 0;
}
```

Output ☑️

```
This is part of outer switch
This is part of inner switch
Exact value of a is : 100
Exact value of b is : 200
```

Nested `switch-case` Statement

Just like nested if–else, you can have nested switch-case constructs. You may have a different switch-case construct each inside the code block of one or more case labels of the outer switch scope.

The nesting of switch-case can be done as follows –

```
switch (exp1){
  case val1:
  switch (exp2){
    case val_a:
      stmts;
      break;
    case val_b:
      stmts;
      break;
  }
```

```
  case val2:
  switch (expr2){
    case val_c:
      stmts;
      break;
    case val_d:
      stmts;
      break;
  }
```

Example:

```
#include <stdio.h>
int main() {
 int day = 1;
 int isWeekend = 0;

 switch (day) {
  case 1:
  case 2:
  case 3:
  case 4:
   switch (isWeekend) {
    case 0:
     printf("It's a weekday.\n");
     break;
    case 1:
     printf("It's a weekday, but it's a weekend for you!\n");
     break;
     }
     break;
  case 5:
     printf("It's Friday! Weekend is almost here.\n");
     break;
 default:
     printf("Unknown day.\n");

  return 0;
}
```

Output ☑️

It's a weekday.

LOOPS

Loops

while Loop

A `while` loop in C (and many other programming languages) is used to execute a block of code repeatedly as long as a specified condition remains true. Here's the basic syntax of a `while` loop in C:

```
while (condition) {
 // Code to be executed while condition is true
}
```

How it works:

1. **Condition**: The loop starts with evaluating the `condition`. If the condition is true, the block of code inside the `while` loop is executed. If the condition is false initially, the block of code is skipped, and the program continues with the next statement after the `while` loop.
2. **Execution**: Once inside the loop, the code block executes. After the code block completes execution, the condition is checked again.
3. **Repetition**: If the condition is still true, the code block executes again. This process repeats until the condition becomes false. Once the condition is false, the loop terminates, and control passes to the statement immediately following the `while` loop.

Example:

Let's consider a simple example where we use a `while` loop to print numbers from 1 to 5:

```
#include <stdio.h>
 int main() {
 int i = 1;
while (i <= 5) {
 printf("%d ", i);
 i++;
}
return 0;
 }
```

Output ☑️

1 2 3 4 5

Explanation of the example:

- **Initialization**: `int i = 1;` initializes `i` to 1.
- **Condition**: `i <= 5;` checks if `i` is less than or equal to 5 before each iteration.
- **Increment**: `i++` increments `i` by 1 after each iteration.

For Loop

A for loop in C (and in many other programming languages) is a control flow statement that allows you to execute a block of code repeatedly a fixed number of times. It's particularly useful when you know exactly how many times you want to execute the loop.

Here's the basic syntax of a for loop in C:

```
for (initialization; condition; increment/decrement) {
   // Code to be executed
}
```

Components of a for loop:

1. **Initialization**: This part is executed once before the loop starts. It usually initializes a loop control variable, often i or j, to a starting value.
2. **Condition**: The loop continues to execute as long as this condition is true. It's evaluated before every iteration of the loop. If the condition evaluates to false initially, the loop will not execute even once.
3. **Increment/Decrement**: This part is usually used to update the loop control variable (i++ for incrementing or i-- for decrementing) after each iteration of the loop body.

How it works:

1. **Initialization**: The initialization part is executed first. It typically initializes a loop control variable to a starting value.
2. **Condition**: The condition is checked before each iteration. If it evaluates to true, the loop body executes. If it evaluates to false, the loop terminates, and control passes to the next statement after the for loop.
3. **Execution**: If the condition is true, the loop body executes. After the execution of the loop body, the increment/decrement part updates the loop control variable.
4. **Repetition**: Steps 2 and 3 repeat until the condition becomes false.

Example:

Let’s use a for loop to print numbers from 1 to 5:

```c
#include <stdio.h>

int main() {
   // Example of a for loop to print numbers from 1 to 5
   for (int i = 1; i <= 5; i++) {
      printf("%d ", i);
   }

   return 0;
}
```

Output ☑️

```
1 2 3 4 5
```

Explanation of the example:

- **Initialization**: `int i = 1;` initializes `i` to 1.
- **Condition**: `i <= 5;` checks if `i` is less than or equal to 5 before each iteration.
- **Increment**: `i++` increments `i` by 1 after each iteration.

Do-while Loop

The `do-while` loop in C (and many other programming languages) is similar to the `while` loop, but with one crucial difference: it guarantees that the block of code inside the loop is executed at least once, even if the condition is initially false.Here's the basic syntax of a `do-while` loop in C:

```c
do {
   // Code to be executed
} while (condition);
```

How it works:

1. **Execution**: The block of code inside the `do` statement is executed first, regardless of the condition.
2. **Condition**: After executing the block of code, the condition is checked. If the condition is true, the block of code will execute again. If the condition is false, the loop terminates, and control passes to the next statement after the `do-while` loop.

Example:

Let's use a `do-while` loop to print numbers from 1 to 5:

```
#include <stdio.h>

int main() {
   int i = 1;

// Example of a do-while loop to print numbers from 1 to 5

do {
      printf("%d ", i);
      i++;
   } while (i <= 5);

   return 0;
}
```

Output ☑️

1 2 3 4 5

Explanation of the example:

- **Execution**: `printf("%d ", i);` prints the value of `i` followed by a space. Then, `i++` increments `i` by 1.

- **Condition**: while (i <= 5); checks if i is less than or equal to 5 after each iteration.

Nested for Loop

Nested loops refer to the situation where one loop (inner loop) is placed inside another loop (outer loop). This creates a loop within a loop structure, allowing for more complex iterations over multiple dimensions or nested data structures.

In C (and many other programming languages), the syntax for nested loops is straightforward. Here's a general structure:

```
for (initialization; condition; increment/decrement) {
   // Outer loop body

   for (initialization; condition; increment/decrement) {
      // Inner loop body
      // Code to be executed
   }
}
```

How it works:

1. **Outer Loop**: The outer loop executes normally. Each time the outer loop iterates, the inner loop is re-initialized.
2. **Inner Loop**: The inner loop executes completely each time the outer loop runs one iteration. It iterates through its entire range for each iteration of the outer loop.

Example:

Let's use nested loops to create a multiplication table up to 5x5:

```c
#include <stdio.h>

int main() {
   // Example of nested loops to create a multiplication table
   for (int i = 1; i <= 5; i++) {
      for (int j = 1; j <= 5; j++) {
         printf("%d\t", i * j);
      }
      printf("\n");
   }

   return 0;
}
```

Explanation of the example:

- **Outer Loop (`i` loop)**: Runs from 1 to 5. Each iteration of `i` corresponds to a row in the multiplication table.
- **Inner Loop (`j` loop)**: For each value of `i`, runs from 1 to 5. Each iteration of `j` corresponds to a column in the multiplication table for the current `i`.
- **Printing**: Inside the inner loop, `printf("%d\t", i * j);` calculates and prints the product of `i` and `j`, formatted as a table cell with a tab (`\t`) separator.
- **Newline**: After each complete iteration of the inner loop (`j` loop), a newline (`\n`) is printed to move to the next row in the table.

Break Statement

The `break` statement in C is used to terminate the current loop or switch statement and transfer control to the statement immediately following the terminated statement.

- **Usage in Loops**: In loops (`for`, `while`, `do-while`), `break` is typically used to exit the loop prematurely based on certain conditions.

Example:

```
#include <stdio.h>
int main() {
   // Example of break in a for loop
   for (int i = 1; i <= 10; i++) {
      if (i == 5) {
         break; // Exit the loop when i equals 5
      }
      printf("%d ", i);
   }
   printf("\nOutside the loop\n");
   return 0;
}
```

Output ☑️

```
1 2 3 4
Outside the loop
```

Continue Statement

The `continue` statement in C is used to skip the current iteration of a loop and continue with the next iteration.

- **Usage in Loops**: It's typically used when you want to skip certain iterations based on specific conditions without exiting the loop entirely.

Example:

```
#include <stdio.h>

int main() {
   // Example of continue in a for loop
   for (int i = 1; i <= 5; i++) {
      if (i == 3) {
         continue; // Skip printing when i equals 3
      }
      printf("%d ", i);
   }
   return 0;
}
```

Output ☑️

1 2 4 5

Goto Statement

The goto statement in C allows transferring control to the labeled statement within the same function.

- **Usage**: It's generally discouraged due to its potential to create spaghetti code (complex and hard-to-follow code paths).

Example:

```
#include <stdio.h>
int main() {
   int i = 0;
loop:
   printf("%d ", i);
    i++;
```

```
    if (i < 5) {
        goto loop; // Jump to the 'loop' label
    }

    return 0;
}
```

Output ☑️

1 2 3 4 5

.

ARRAYS

Arrays

Arrays in C are a kind of data structure that can store a fixed-size sequential collection of elements of the same data type. Arrays are used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

What is an Array in C?

An array in C is a collection of data items of similar data type. One or more values of the same data type, which may be primary data types (int, float, char), or user-defined types such as struct or pointers can be stored in an array. In C, the type of elements in the array should match with the data type of the array itself.

The size of the array, also called the length of the array, must be specified in the declaration itself. Once declared, the size of a C array cannot be changed. When an array is declared, the compiler allocates a continuous block of memory required to store the declared number of elements.

Introduction with RAM Organization

Concept of Arrays:

- An array is a collection of variables that are accessed with an index number.
- All elements in an array are of the same data type (e.g., all integers or all floats).

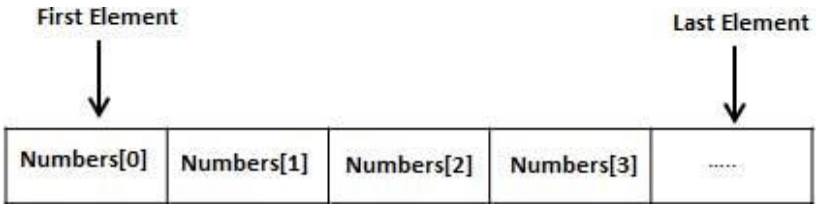
RAM Organization:

- When an array is declared, the compiler allocates a block of memory to store the array elements.
- The size of this block is the number of elements multiplied by the size of each element.

- Memory addresses in this block are contiguous, which means they are adjacent to each other.

```
int arr[5] = {1, 2, 3, 4, 5}; // Array of 5 integers
// Memory Layout (Assuming an int is 4 bytes and arr starts at address 0x1000):
// Address   Value
// 0x1000    1
// 0x1004    2
// 0x1008    3
// 0x100C    4
// 0x1010    5
```

Each element is identified by an index starting with "0". The lowest address corresponds to the first element and the highest address to the last element.



Declaration of an Array in C

To declare an array in C, you need to specify the type of the elements and the number of elements to be stored in it.

Syntax to Declare an Array:

```
type arrayName[size];
```

The "size" must be an integer constant greater than zero and its "type" can be any valid C data type. There are different ways in which an array is declared in C.

Example: Declaring an Array in C

In the following example, we are declaring an array of 5 integers and printing the indexes and values of all array elements.But here only memory was allocated not the elements are initialized –

```
#include <stdio.h>
int main(){
  int arr[5];
  int i;

  for (i = 0; i <= 4; i++){
    printf("a[%d]: %d\n", i, arr[i]);
  }
  return 0;
}
```

Output ☑️

```
a[0]: 2015508889
a[1]: 32765
a[2]: 100
a[3]: 0
a[4]: 4096
```

Initialization of an Array in C

At the time of declaring an array, you can initialize it by providing the set of comma-separated values enclosed within the curly braces {}.

Syntax to Initialize an Array

```
data_type array_name [size] = {value1, value2, value3, ...};
```

Declaration & Definition Methods

Declaration:

- Tells the compiler to reserve space for an array.
- Example: `int arr[10];` declares an array named `arr` that can hold 10 integers.

Definition with Initialization:

- An array can be defined with or without initialization.

```
int arr[5]; // Declaration (memory is allocated, but elements are not initialized)

int arr2[5] = {1, 2, 3, 4, 5}; // Declaration and initialization
```

Example:

```
#include <stdio.h>
int main(){
  int arr2[5] = {1, 2, 3, 4, 5};;
  int i;

  for (i = 0; i <= 4; i++){
    printf("a[%d]: %d\n", i, arr2[i]);
  }
  return 0;
}
```

Output ☑️

a[0]: 1

a[1]: 2

a[2]: 3

a[3]: 4

a[4]: 5

Partial Initialization:

- If fewer initializers are provided, the remaining elements are initialized to zero.

```
int arr3[5] = {1, 2}; // Elements are {1, 2, 0, 0, 0}
```

Example:

```
#include <stdio.h>
int main(){
  int arr3[5] = {1,2};
  int i;

  for(i = 0; i <= 4; i++){
    printf("a[%d]: %d\n", i, arr3[i]);
  }
  return 0;
}
```

Output ☑️

```
a[0]: 1
a[1]: 2
a[2]: 0
a[3]: 0
a[4]: 0
```

Accessing Array Elements with Index

Each element in an array is identified by a unique incrementing index, starting with "0". To access the element by its index, this is done by placing the index of the element within square brackets after the name of the array.

The elements of an array are accessed by specifying the index (offset) of the desired element within the square brackets after the array name. For example –

```
double salary = balance[9];
```

The above statement will take the 10th element from the array and assign the value to the "salary".

Example:

```
#include <stdio.h>

int main() {
   int arr[5] = {10, 20, 30, 40, 50};

   // Accessing elements
   printf("First element: %d\n", arr[0]); // Output: 10
   printf("Third element: %d\n", arr[2]); // Output: 30

   // Modifying elements
   arr[1] = 25;
   printf("Modified second element: %d\n", arr[1]); // Output: 25

   return 0;
}
```

Output ☑️

```
First element: 10
Third element: 30
Modified second element: 25
```

Example:

```
#include <stdio.h>

int main(){
  int marks[10] = {50, 55, 67, 73, 45, 21, 39, 70, 49, 51};
  int i, sum = 0;
  float avg;
```

```
  for (i = 0; i <= 9; i++){
    sum += marks[i];
  }

  avg = (float)sum / 10;
  printf("Average: %f", avg);
  return 0;
}
```

Output ☑️

```
Average: 52.000000
```

Multidimensional Arrays in C

Multi-dimensional arrays can be termed as nested arrays. In such a case, each element in the outer array is an array itself. Such type of nesting can be upto any level. If each element in the outer array is another one-dimensional array, it forms a two-dimensional array. In turn, the inner array is an array of another set of one dimensional array, it is a three-dimensional array, and so on.

Declaration of Multidimensional Arrays

Depending on the nesting level, the multi-dimensional array is declared as follows −

```
type name[size1][size2]...[sizeN];
```

Two-dimensional Array in C

Each element of a two-dimensional array is an array itself. It is like a table or a matrix. The elements can be considered to be logically arranged in rows and columns. Hence, the location of any element is characterized by its row number and column number. Both row and column index start from 0.

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

To declare a two-dimensional integer array of size [x][y], you would write something as follows:

```
type arrayName [ x ][ y ];
```

Declaration and Initialization of Two-dimensional Array

The following statement declares and initializes a two-dimensional array

```
int arr[3][5] = {1,2,3,4,5, 10,20,30,40,50, 5,10,15,20,25};
```

Example of printing two dimensional array elements-

```
#include <stdio.h>

int main() {
   int matrix[3][3] = {
      {1, 2, 3},
      {4, 5, 6},
      {7, 8, 9}
   };
   // Accessing elements
   printf("Element at (0,1): %d\n", matrix[0][1]); // Output:2
   printf("Element at (2,2): %d\n", matrix[2][2]); // Output:9
```

```
    // Modifying elements
    matrix[1][1] = 50;
    printf("Modified element at (1,1): %d\n", matrix[1][1]); // Output: 50

    return 0;
}
```

Output ☑️

```
Element at (0,1): 2
Element at (2,2): 9
Modified element at (1,1): 50
```

Example of printing two dimensional array elements-

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

  /* an array with 5 rows and 2 columns*/
  int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};
  int i, j;

  /* output each array element's value */
  for ( i = 0; i < 5; i++ ) {
    for ( j = 0; j < 2; j++ ) {
      printf("a[%d][%d] = %d\n", i,j, a[i][j] );
    }
  }

  return 0;
}
```

Output ☑️

```
a[0][0] = 0
a[0][1] = 0
a[1][0] = 1
a[1][1] = 2
a[2][0] = 2
a[2][1] = 4
```

a[3][0] = 3
a[3][1] = 6
a[4][0] = 4
a[4][1] = 8

Length Calculation of Array

Using sizeof Operator:

- The sizeof operator can be used to determine the total size of the array in bytes and the size of an individual element.
- By dividing these two, we can get the number of elements in the array.

Example:

```
#include <stdio.h>

int main() {
   int arr[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

   // Calculate length
   int length = sizeof(arr) / sizeof(arr[0]);
   printf("Length of the array: %d\n", length); // Output: 10

   return 0;
}
```

Output ☑️

Length of the array: 10

STRINGS

String

A string in C is a one-dimensional array of char type, with the last character in the array being a "null character" represented by '\0'. Thus, a string in C can be defined as a null-terminated sequence of char type values.

Creating a String in C

Let us create a string "Hello". It comprises five char values. In C, the literal representation of a char type uses single quote symbols – such as 'H'. These five alphabets put inside single quotes, followed by a null character represented by '\0' are assigned to an array of char types. The size of the array is five characters plus the null character – six.

Example

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

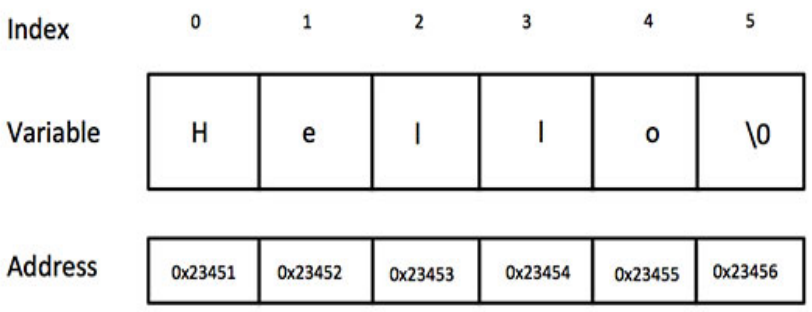
Initializing String Without Specifying Size

C lets you initialize an array without declaring the size, in which case the compiler automatically determines the array size**.**

Example

```
char greeting[] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

The array created in the memory can be schematically shown as follows –



If the string is not terminated by "\0", it results in unpredictable behavior.

Loop Through a String

You can loop through a string (character array) to access and manipulate each character of the string using the for loop or any other loop statements.

Example

In the following example, we are printing the characters of the string.

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

int main (){

  char greeting[] = {'H', 'e', 'l', 'l', 'o', '\0'};

  for (int i = 0; i < 5; i++) {
    printf("%c", greeting[i]);
  }

  return 0;
}
```

Output ☑️

Hello

C String Functions

C also has many useful string functions, which can be used to perform certain operations on strings.

To use them, you must include the `<string.h>` header file in your program:

String Length

For example, to get the length of a string, you can use the `strlen()` function:

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

int main() {
  char alphabet[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
  printf("%d", strlen(alphabet));
  return 0;
}
```

Output ☑️

```
26
```

We used `sizeof` to get the size of a string/array. Note that `sizeof` and `strlen` behaves differently, as `sizeof` also includes the `\0` character when counting:

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

int main() {
  char alphabet[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
  printf("Length is: %d\n", strlen(alphabet));
  printf("Size is: %d\n", sizeof(alphabet));
  return 0;
}
```

Output ☑️

Length is: 26

Size is: 27

It is also important that you know that `sizeof` will always return the memory size (in bytes), and not the actual string length:

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

int main() {
  char alphabet[50] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
  printf("Length is: %d\n", strlen(alphabet));
  printf("Size is: %d\n", sizeof(alphabet));
  return 0;
}
```

Output ☑️

Length is: 26

Size is: 50

Concatenate Strings

To concatenate (combine) two strings, you can use the `strcat()` function:

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

int main() {
  char str1[20] = "Hello ";
  char str2[] = "World!";

  // Concatenate str2 to str1 (the result is stored in str1)
  strcat(str1, str2);

  // Print str1
  printf("%s", str1);

  return 0;
}
```

Output ☑️

```
Hello World!
```

Copy Strings

To copy the value of one string to another, you can use the `strcpy()` function:

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

int main() {
  char str1[20] = "Hello World!";
  char str2[20];

  // Copy str1 to str2
  strcpy(str2, str1);

  // Print str2
  printf("%s", str2);

  return 0;
}
```

Output ☑️

Hello World!

Compare Strings

To compare two strings, you can use the strcmp() function.

It returns 0 if the two strings are equal, otherwise a value that is not 0:

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

int main() {
  char str1[] = "Hello";
  char str2[] = "Hello";
  char str3[] = "Hi";
```

```
    // Compare str1 and str2, and print the result
    printf("%d\n", strcmp(str1, str2));

    // Compare str1 and str3, and print the result
    printf("%d\n", strcmp(str1, str3));

    return 0;
}
```

Output ☑️

0

-4

POINTERS

Pointers

Introduction to pointers with single variable.

What is a Pointer?

A pointer is a variable that stores the memory address of another variable. Pointers are powerful features in C and C++ that provide a way to access and manipulate data stored in memory.

Why Use Pointers?

- Efficient array and string manipulation
- Dynamic memory allocation
- Passing large structures to functions efficiently
- Direct memory access

Syntax

```
type *pointerName;
```

`type` is the data type of the variable that the pointer will point to.

`*` indicates that the variable is a pointer.

`pointerName` is the name of the pointer variable.

Basic Example

```
#include <stdio.h>

int main() {
   int var = 10;     // Declare an integer variable
   int *ptr;        // Declare a pointer to an integer

   ptr = &var;      // Assign the address of var to the pointer ptr

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

   // Output the address of var
   printf("Address of var: %p\n", &var);

   // Output the address stored in ptr (which is the address of var)
   printf("Value of ptr (address of var): %p\n", ptr);

   // Output the value pointed to by ptr (which is the value of var)
   printf("Value pointed to by ptr: %d\n", *ptr);

   return 0;
}
```

Output ☑️

Value of var: 10
Address of var: 0x7ffc4a396c5c
Value of ptr (address of var): 0x7ffc4a396c5c
Value pointed to by ptr: 10

Pointers to Array

Definition:

- A pointer to an array is a pointer that points to the first element of the array. Array names in C are essentially pointers to their first element.

Example:

```
#include <stdio.h>

int main() {
   int arr[5] = {10, 20, 30, 40, 50};
   int *p = arr; // p points to the first element of arr

   for (int i = 0; i < 5; i++) {
      printf("Element %d: %d\n", i, *(p + i));
   }

   return 0;
}
```

Output ☑️

```
Element 0: 10
Element 1: 20
Element 2: 30
Element 3: 40
Element 4: 50
```

FUNCTIONS

Function in C

In c, we can divide a large program into the basic building blocks known as function. The function contains the set of programming statements enclosed by {}. A function can be called multiple times to provide reusability and modularity to the C program. In other words, we can say that the collection of functions creates a program. The function is also known as *procedure* or *subroutine* in other programming languages.

Function Aspects

There are three aspects of a C function.

- **Function declaration** A function must be declared globally in a c program to tell the compiler about the function name, function parameters, and return type.
- **Function call** Function can be called from anywhere in the program. The parameter list must not differ in function calling and function declaration. We must pass the same number of functions as it is declared in the function declaration.
- **Function definition** It contains the actual statements which are to be executed. It is the most important aspect to which the control comes when the function is called. Here, we must notice that only one value can be returned from the function.

SN	C function aspects	Syntax
1	Function declaration	return_type function_name (argument list);
2	Function call	function_name (argument_list)
3	Function definition	return_type function_name (argument list) {function body;}

The syntax of creating function in c language is given below:

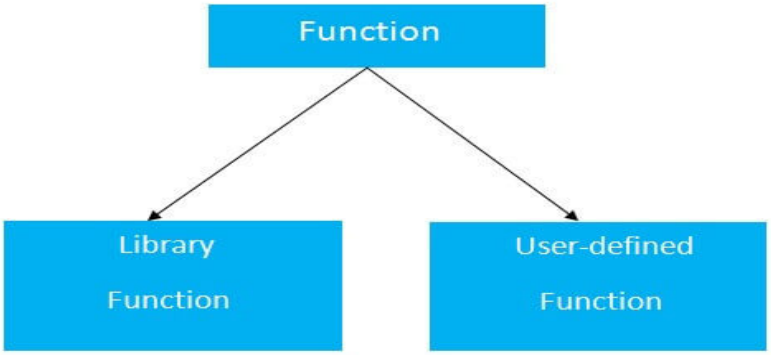
```
return_type function_name(data_type parameter...){
//code to be executed
}
```

Types of Functions

There are two types of functions in C programming:

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

the complexity of a big program and optimizes the code



Function Parameters

Function parameters are the variables listed inside the parentheses in the function definition. They are placeholders for the actual values that will be passed to the function when it is called.

Function Arguments

- **Definition:** Arguments are the actual values or expressions passed to a function when it is called. These values are assigned to the corresponding parameters defined in the function.
- **Scope:** Arguments are evaluated and passed to the function; within the function, they are accessed through the parameters.
- **Usage:** They provide the actual data that the function will process.

Example:

```
#include <stdio.h>

// Function definition with parameters x and y
void multiply(int x, int y) {
   int product = x * y;
   printf("Product: %d\n", product);
}
int main() {
   int a = 4, b = 7;
   // Function call
   multiply(a, b); // Here, a and b are arguments passed to the function
   return 0;
}
```

Output ☑️

Product: 28

In this example, **a** and **b** are arguments passed to the function **multiply** when it is called.

Key Differences

- Parameters are part of the function definition. They specify what kind of arguments the function expects.
- Arguments are the actual values supplied to the function when it is called.

Return Value

A C function may or may not return a value from the function. If you don't have to return any value from the function, use void for the return type.

Let's see a simple example of a C function that doesn't return any value from the function.

Example without return value:

```
void hello(){
printf("hello c");
}
```

If you want to return any value from the function, you need to use any data type such as int, long, char, etc. The return type depends on the value to be returned from the function.

Let's see a simple example of a C function that returns int value from the function.

Example with return value:

```
int get(){
return 10;
   }
```

return and exit(0)

return

- **Usage**: The **return** statement is used within a function to terminate its execution and optionally return a value to the calling function. In the **main** function, **return** can be used to indicate the status of program termination.
- **Effect**: When **return** is called in the **main** function, it ends the execution of the program and returns control to the operating system. The value returned (commonly **0** for success and non-zero for failure) can be used by the operating system to determine the result of the program execution.
- **Scope**: Only terminates the current function, not the entire program unless it is used in **main.**

  Example:

```
#include <stdio.h>

int main() {
   printf("Program is running...\n");

  return 0; // Terminates the main function and returns 0 to the operating system
}
```

Output ☑️

Program is running...

`exit(0)`

- **Usage:** The **`exit`** function, defined in **`stdlib.h`**, terminates the entire program immediately, regardless of which function or block it is called from. It can be called from any function, not just **`main`**.

- **Effect:** **`exit`** performs cleanup (like flushing buffers, closing files, etc.) and then terminates the program. The argument passed to **`exit`** (typically **`0`** for success and non-zero for failure) is returned to the operating system.
- **Scope:** Terminates the entire program, no matter where it is called from.

Example:

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

void someFunction() {
    printf("Inside someFunction. Exiting program...\n");
    exit(0); // Terminates the entire program immediately
}

int main() {
    printf("Program is starting...\n");
    someFunction(); // Calls someFunction which will terminate the program
    printf("This line will not be executed.\n");
    return 0;
}
```

Output ☑️

```
Program is starting...
Inside someFunction. Exiting program...
```

Difference between `return` and `exit(0)`

`return`	**`exit(0)`**
used to exit a function and optionally return a value to the calling function.	used to terminate the entire program and return a value to the operating system.
only terminates the function it is called in.	terminates the entire program, regardless of where it is called.
commonly used in functions to	used when you need to

provide results or status.	terminate the program due to an error, a specific condition, or a user request, from anywhere in the code.

The main() Function

The **main()** function in C is an entry point of any program. The program execution starts with the ***main()* function**. It is designed to perform the main processing of the program and clean up any resources that were allocated by the program. In a C code, there may be any number of functions, but it must have a ***main()* function**. Irrespective of its place in the code, it is the first function to be executed.

Syntax:

```
int main(){
  //one or more statements;
  return 0;
}
```

1.Function Declaration and Definition

- **int** indicates that the function returns an integer value.
- **main** is the name of the function. In C, **main** is a special function name that signifies the starting point of the program.
- () indicates that this function does not take any arguments

2.Function Body

- **`{ ... }`** braces enclose the body of the **`main`** function.
- Inside the braces, you write the statements that you want to execute when the program runs.

3.Statements

- This is a placeholder for any statements or code you want to execute. For example, you might print a message, perform calculations, or call other functions.

4.Return Statement

- **return 0;** indicates that the program terminated successfully.

Actual Parameters

- **Definition**: Actual parameters, also known as arguments, are the real values or variables that are passed to the function when it is called.

Example:

```
int main() {
   int x = 5;
   int y = 10;
   int result = add(x, y);  // x and y are actual parameters
   printf("Result: %d\n", result);
   return 0;
}
```

Output ☑️

Result: 15

Formal Parameters

- **Definition:** Formal parameters are the variables defined in the function declaration/definition that receive the values of the actual parameters.

Example:

```
int add(int a, int b) {  // a and b are formal parameters
    return a + b;
}
```

Example in Detail.

Let's put this all together with a detailed example:

Function Definition with Formal Parameters:

```
#include <stdio.h>

// Function declaration/definition with formal parameters a and b
int add(int a, int b) {
    return a + b;
}

int main() {
    int x = 5;
    int y = 10;

    // Function call with actual parameters x and y
    int result = add(x, y);

    printf("Result: %d\n", result);  // Output: Result: 15
    return 0;
}
```

Output ☑️

Result: 15

Explanation

1.Function Declaration/Definition:

```
int add(int a, int b) {
   return a + b;
}
```

- **a** and **b** are formal parameters. They act as placeholders for the values that will be passed to the function.

2.Function call

```
int result = add(x, y);
```

- **x** and **y** are actual parameters. They provide the actual values that are passed to the function.

3.Function Execution:

- When `add(x, y)` is called, the values of `x` and `y` (which are `5` and `10` respectively) are passed to the formal parameters `a` and `b`.
- The function `add` then performs its operation using these values and returns the result.

Local and Global Variables in Functions

In C programming, variables can be defined in different scopes, which determine their visibility and lifetime. The two most common types of variables based on their scope are **local variables** and **global variables**.

Local Variables

Local variables are declared inside a function or a block (e.g., within `{}` braces). They are only accessible within that function or block, and they exist only during the execution of that function or block.

Characteristics of Local Variables:

- **Scope**: Limited to the function or block in which they are declared.
- **Lifetime**: Exist only during the execution of the function or block.
- **Storage**: Typically stored on the stack.

Example:

```
#include <stdio.h>

void printLocalVariable() {
   int localVar = 10; // local variable
   printf("Local variable: %d\n", localVar);
}

int main() {
   int localVar = 20; // another local variable with the same name but different scope
   printLocalVariable();
   printf("Local variable in main: %d\n", localVar);
   return 0;
}
```

Output ☑️

```
Local variable: 10
Local variable in main: 20
```

Global Variables

Global variables are declared outside of all functions, usually at the top of the program file. They are accessible from any function within the same file, and their lifetime is the entire duration of the program.

Characteristics of Global Variables:

- **Scope**: Accessible from any function within the same file (and other files if declared as extern).
- **Lifetime**: Exist for the entire duration of the program.
- **Storage**: Typically stored in the data segment of memory.

Example:

```
#include <stdio.h>

int globalVar = 30; // global variable

void printGlobalVariable() {
   printf("Global variable: %d\n", globalVar);
}

int main() {
   printGlobalVariable();
   globalVar = 40; // modify global variable
   printf("Global variable in main: %d\n", globalVar);
   return 0;
}
```

Output ☑️
Global variable: 30
Global variable in main: 40

Example Using Both Local and Global Variables

```
#include <stdio.h>

int globalVar = 50; // global variable

void function1() {
   int localVar = 10; // local variable
 printf("Inside function1 - Local variable: %d\n", localVar);
 printf("Inside function1 - Global variable: %d\n", globalVar);
}
void function2() {
   int localVar = 20; // local variable
   globalVar = 60; // modifying global variable
printf("Inside function2 - Local variable: %d\n", localVar);
printf("Inside function2 - Global variable: %d\n", globalVar);
}
```

```
int main() {
    function1();
    function2();
    printf("Inside main - Global variable: %d\n", globalVar);
    return 0;
}
```

Output ☑️

```
Inside function1 - Local variable: 10
Inside function1 - Global variable: 50
Inside function2 - Local variable: 20
Inside function2 - Global variable: 60
Inside main - Global variable: 60
```

Call by Value

- In **call by value** method, the value of the **actual** parameters is copied into the **formal** parameters. In other words, we can say that the value of the variable is used in the function call in the **call by value** method.
- In **call by value** method, we can not modify the value of the **actual** parameter by the **formal** parameter.
- In **call by value**, different memory is allocated for **actual** and **formal** parameters since the value of the **actual parameter is copied into the formal parameter.**
- The **actual parameter** is the argument which is used in the function call whereas **formal parameter** is the argument which is used in the function definition.

Example to demonstrate call by value by

```
#include <stdio.h>

/* function declaration */
void swap(int x, int y);

int main(){

  /* local variable definition */
  int a = 100;
  int b = 200;

  printf("Before swap, value of a: %d\n", a);
  printf("Before swap, value of b: %d\n", b);
  /* calling a function to swap the values */
  swap(a, b);
  printf("After swap, value of a: %d\n", a);
  printf("After swap, value of b: %d\n", b);
  return 0;
}

void swap(int x, int y){
  int temp;
  temp = x; /* save the value of x */
  x = y;    /* put y into x */
  y = temp; /* put temp into y */
  return;
}
```

Output ☑️

Before swap, value of a: 100
Before swap, value of b: 200
After swap, value of a: 100
After swap, value of b: 200

Explanation of the Program

1. Function Declaration:

```
void swap(int x, int y);
```

The program declares a function swap that takes two integers

2.Main Function:

- Initializes two variables, **a** and **b**, with values 100 and 200.
- Prints the values of **a** and **b** before calling **swap**.
- Calls the **swap** function to swap the values of **a** and **b**.
- Prints the values of **a** and **b** after calling **swap**.

```
int main() {
   int a = 100;
   int b = 200;

   printf("Before swap, value of a: %d\n", a);
   printf("Before swap, value of b: %d\n", b);

   swap(a, b);

   printf("After swap, value of a: %d\n", a);
   printf("After swap, value of b: %d\n", b);

   return 0;
}
```

3.Swap Function:

- Takes two integers, **x** and **y**.
- Swaps the values of **x** and **y** using a temporary variable **temp**.

```
void swap(int x, int y) {
   int temp;

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

   return;
}
```

Why the Swap Doesn't Work

- Call by Value: When **swap(a, b)** is called, it passes copies of a and b to the function. The **swap** function swaps the copies, not the original values.
- Therefore, the values of **a** and **b** in the **main** function remain unchanged.

4.Output:

- **Before swap**: a is 100, b is 200.
- **After swap**: a is still 100, b is still 200, demonstrating that the original values were not changed.

Call by Reference

- In call by reference, the address of the variable is passed into the function call as the actual parameter.
- The value of the actual parameters can be modified by changing the formal parameters since the address of the actual parameters is passed.
- In call by reference, the memory allocation is similar for both formal parameters and actual parameters. All the operations in the function are performed on the value stored at the address of the actual parameters, and the modified value gets stored at the same address.

Example to demonstrate call by reference:

```
#include <stdio.h>

/* Function definition to swap the values */
/* It receives the reference of two variables whose values are to be swapped */

int swap(int *x, int *y){

  int temp;

  temp = *x;   /* save the value at address x */
  *x = *y;  /* put y into x */
  *y = temp;   /* put z into y */

  return 0;
}

/* The main() function has two variables "a" and "b" */
/* Their addresses are passed as arguments to the swap() function. */

int main(){

  /* local variable definition */
  int a = 10;
  int b = 20;

  printf("Before swap, value of a: %d\n", a );
  printf("Before swap, value of b: %d\n", b );

  /* calling a function to swap the values */
  swap(&a, &b);

  printf("After swap, value of a: %d\n", a);
  printf("After swap, value of b: %d\n", b);

  return 0;
}
```

Explanation of the Program

1. Function Definition (swap)

The function **swap** takes two integer pointers (**int *x** and **int *y)** as parameters. These pointers allow the function to modify the values of **a** and **b** in the **main** function directly.

```
int swap(int *x, int *y) {
   int temp;

   temp = *x;   // Save the value at address x in temp
   *x = *y;     // Assign the value at address y to x
   *y = temp;   // Assign the saved value (temp) to y

   return 0;    // Return statement (not necessary for this function)
}
```

2. Main Function

In the **main** function:

- Two local variables a and b are defined and initialized to **10** and **20**, respectively.
- The values of **a** and **b** are printed before calling the **swap** function.
- The **swap** function is called with the addresses of **a** and **b** (**&a** and **&b**) as arguments.
- After the **swap** function call, the values of **a** and **b** are printed again to show the swapped values.

```
int main() {
   int a = 10;
   int b = 20;

   printf("Before swap, value of a: %d\n", a);
   printf("Before swap, value of b: %d\n", b);

   swap(&a, &b);  // Pass the addresses of a and b to swap

   printf("After swap, value of a: %d\n", a);
   printf("After swap, value of b: %d\n", b);

   return 0;
}
```

3. Explanation

- **Pointer Parameters**: By passing **&a** and **&b** to **swap**, the function **swap** receives the addresses of **a** and **b**. Inside **swap**, ***x** and ***y** represent the values stored at those addresses (which are **a** and **b**).
- **Swapping Mechanism**: The function swap uses a temporary variable **temp** to store the value of ***x**, then assigns ***y** to ***x**, and finally assigns **temp** (which holds the original value of ***x**) to ***y**. This effectively swaps the values of **a** and **b**.
- **Output**: Before the swap, **a** is **10** and **b** is **20**. After the swap, **a** becomes **20** and **b** becomes **10**, demonstrating that the values have been successfully swapped.

Passing array in a function

If you want to pass an array to a function, you can use either **call by value** or **call by reference** method. In **call by value** method, the argument to the function should be an initialized array, or an array of fixed size equal to the size of the array to be passed. In **call by reference** method, the function argument is a pointer to the array.

Pass array with call by value method

In the following code, the **main() function** has an array of integers. A user−defined function average () is called by passing the array to it. The average() function receives the array, and adds its elements using a **for loop**. It returns a float value representing the average of numbers in the array.

Example:

```
#include <stdio.h>
float average(int arr[5]);
int main(){
  int arr[] = {10, 34, 21, 78, 5};
  float avg = average(arr);
  printf("average: %f", avg);
}
float average(int arr[5]){
  int sum=0;
  int i;
  for (i=0; i<5; i++){
    printf("arr[%d]: %d\n", i, arr[i]);
    sum+=arr[i];
  }
  return (float)sum/5;
}
```

Output ☑️

```
arr[0]: 10
arr[1]: 34
arr[2]: 21
arr[3]: 78
arr[4]: 5
average: 29.600000
```

In the following variation, the average() function is defined with two arguments, an uninitialized array without any size specified. The length of

the array declared in **main() function** is obtained by divising the size of the array with the size of int **data type**.

Example:

```
#include <stdio.h>
float average(int arr[], int length);
int main(){
  int arr[] = {10, 34, 21, 78, 5};
  int length = sizeof(arr)/sizeof(int);

  float avg = average(arr, length);

  printf("average: %f", avg);
}
float average(int arr[], int length){
  int sum=0;
  int i;
  for (i=0; i<length; i++){
    printf("arr[%d]: %d\n", i, arr[i]);
    sum+=arr[i];
  }
  return (float)sum/length;
}
```

Output ☑️

```
arr[0]: 10
arr[1]: 34
arr[2]: 21
arr[3]: 78
arr[4]: 5
average: 29.600000
```

Pass array with call by reference method

To use this approach, we should understand that elements in an array are of similar data type, stored in **continuous** memory locations, and the array size depends on the data type. Also, the **address of the 0th element** is the **pointer to the array**.

In the following example –

```
int a[5] = {1,2,3,4,5};
```

The size of the array is 20 bytes (4 bytes for each int)

```
int *x = a;
```

Here x is the pointer to the array. It points to the 0th element. If the pointer is incremented by 1, it points to the next element.

Example:

```
#include <stdio.h>
int max(int *arr, int length);
int main(){
  int arr[] = {10, 34, 21, 78, 5};
  int length = sizeof(arr)/sizeof(int);
  int maxnum = max(arr, length);
  printf("max: %d", maxnum);
}
int max(int *arr, int length){
  int max=*arr;
  int i;
  for (i=0; i<length; i++){
    printf("arr[%d]: %d\n", i, (*arr));
    if ((*arr)>max)
      max = (*arr);
    arr++;
  }
  return max;
}
```

Output ☑️

```
arr[0]: 10
arr[1]: 34
arr[2]: 21
arr[3]: 78
arr[4]: 5
max: 78
```

The **max() function** receives the address of the array from main() in the pointer arr. Each time, when it is **incremented**, it **points** to the **next element** in the original array.

Example:

```
#include <stdio.h>

void reverseArray(int arr[], int size) {
   int temp;
   for (int i = 0; i < size / 2; i++) {
      temp = arr[i];
      arr[i] = arr[size - 1 - i];
      arr[size - 1 - i] = temp;
   }
}
int main() {
   int myArray[] = {1, 2, 3, 4, 5};
   int size = sizeof(myArray) / sizeof(myArray[0]);

   printf("Original array: ");
   for (int i = 0; i < size; i++) {
      printf("%d ", myArray[i]);
   }
 printf("\n");

   reverseArray(myArray, size);

   printf("Reversed array: ");
   for (int i = 0; i < size; i++) {
      printf("%d ", myArray[i]);
   }
     printf("\n");
   return 0;
}
```

Output ☑️

Original array: 1 2 3 4 5
Reversed array: 5 4 3 2 1

Different aspects of function calling

A function may or may not accept any argument. It may or may not return any value. Based on these facts, There are four different aspects of function calls.

- function without arguments and without return value
- function without arguments and with return value
- function with arguments and without return value
- function with arguments and with return value

- **Function without argument and without return value**

This type of function neither takes any arguments nor returns any value. It performs a specific task within its body and then exits.

Example:

```
#include <stdio.h>

void printMessage();

void main() {
   printf("Hello ");
   printMessage(); // Calling the function
}

void printMessage() {
   printf("World!\n"); // Function definition
}
```

Output ☑️
Hello World!

Explanation:

- **printMessage()** is called in the **main()** function.
- It prints **"World!"** when called.

- **Function without argument and with return value**

This type of function does not take any arguments but returns a value after execution.

Example:

```
#include <stdio.h>

int getNumber();

void main() {
   int num = getNumber(); // Calling the function and storing the return value
   printf("The number is %d\n", num);
}

int getNumber() {
   return 42; // Function returns an integer value
}
```

Output ☑️
The number is 42

Explanation:

- **getNumber()** is called in the **main()** function.
- It returns the number **42**, which is then printed.

- **Function with arguments and without return value**

This type of function takes arguments but does not return any value.

Example:

```
#include <stdio.h>

void printSum(int a, int b);

void main() {
   int x = 5, y = 10;
   printSum(x, y); // Calling the function with arguments
}

void printSum(int a, int b) {
   printf("The sum is %d\n", a + b); // Function prints the sum of two integers
}
```

Output ☑️
The sum is 15

Explanation:

- **printSum(int a, int b)** is called in the **main()** function with two arguments.
- It prints the sum of **a** and **b**.

- **Function with arguments and with return value**

This type of function takes arguments and also returns a value after execution.

Example:

```
#include <stdio.h>
int multiply(int a, int b);
void main() {
   int x = 5, y = 10;
   int result = multiply(x, y); // Calling the function with arguments and storing the return
value

   printf("The product is %d\n", result);
}

int multiply(int a, int b) {
   return a * b; // Function returns the product of two integers
}
```

Output ☑️
The product is 50

Explanation:

- **multiply(int a, int b)** is called in the **main()** function with two arguments.
- It returns the product of **a** and **b**, which is then printed.

C Function pointer

Create a pointer of any data type such as int, char, float, we can also create a pointer pointing to a function. The code of a function always resides in memory, which means that the function has some address. We can get the address of memory by using the function pointer.

Declaration of a function pointer

functions have addresses, so we can create pointers that can contain these addresses, and hence can point them.

Syntax of function pointer

```
return type (*ptr_name)(type1, type2…);
```

For example

```
int (*ip) (int);
```

Calling a function through a function pointer

Example

```
result = (*fp)( a , b);    // Calling a function using function pointer.
```

Example

```
result = (*fp)( a , b);    // Calling a function using function pointer
```

Example program

```
#include <stdio.h>
int add(int,int);
int main()
{
 int a,b;
 int (*ip)(int,int);
 int result;
 printf("Enter the values of a and b : ");
 scanf("%d %d",&a,&b);
 ip=add;
 result=(*ip)(a,b);
 printf("Value after addition is : %d",result);
 return 0;
}
int add(int a,int b)
{
 int c=a+b;
 return c;
}
```

Output ☑️

```
Enter the values of a and b : 2 6
Value after addition is : 8
```

Passing a function's address as an argument to other function

We can pass the function's address as an argument to other functions in the same way

we send other arguments to the function.

```
#include <stdio.h>
void func1(void (*ptr)());
void func2();
int main()
{
   func1(func2);
    return 0;
}
void func1(void (*ptr)())
{
   printf("Function1 is called");
   (*ptr)();
}
void func2()
{
   printf("\nFunction2 is called");
}
```

Output ☑️

```
Function1 is called
Function2 is called
```

In the above code, we have created two functions, i.e., func1() and func2(). The func1() function contains the function pointer as an argument. In the main() method, the func1() method is called in which we pass the address of func2. When func1() function is called, 'ptr' contains the address of 'func2'. Inside the func1() function, we call the func2() function by dereferencing the pointer 'ptr' as it contains the address of func2.

Array of Function Pointers

Function pointers are used in those applications where we do not know in advance which function will be called. In an array of function pointers, array takes the addresses of different functions, and the appropriate function will be called based on the index number.

Example:

```
#include <stdio.h>
float add(float,int);
float sub(float,int);
float mul(float,int);
float div(float,int);

int main()
{
  float x;            // variable declaration.
  int y;
  float (*fp[4]) (float,int);       // function pointer declaration.
  fp[0]=add;            // assigning addresses to the elements of an array of a function   pointer.
  fp[1]=sub;
  fp[2]=mul;
  fp[3]=div;
  printf("Enter the values of x and y :");
  scanf("%f %d",&x,&y);
 float r=(*fp[0]) (x,y);       // Calling add() function.
  printf("\nSum of two values is : %f",r);
    r=(*fp[1]) (x,y);           // Calling sub() function.
   printf("\nDifference of two values is : %f",r);
    r=(*fp[2]) (x,y);           // Calliung sub() function.
   printf("\nMultiplication of two values is : %f",r);
    r=(*fp[3]) (x,y);          // Calling div() function.
   printf("\nDivision of two values is : %f",r);
```

```
 float sub(float x,int y)
{
   float a=x-y;
   return a;
}
float mul(float x,int y)
{
   float a=x*y;
   return a;
}
float div(float x,int y)
{
   float a=x/y;
   return a;
}
```

Output ☑️

```
Enter the values of x and y :3 6

Sum of two values is : 9.000000
Difference of two values is : -3.000000
Multiplication of two values is : 18.000000
Division of two values is : 0.500000
```

Return Array from function

Functions in C help the programmers to adapt modular program design. A function can be defined to accept one or more than one argument, it is able to return a single value to the calling environment. However, the function can be defined to return an array of values. In C, a function can be made to return an array by one of following methods –

- Passing the **array as argument** and **returning the pointer**
- Declaring a **static array** in a function and returning its **pointer**
- Using **malloc()** function

Passing Array by reference

In the following example, we declare an uninitialized array in main() and pass it to a function, along with an integer. Inside the function, the array is filled with the square, cube and square root. The function returns the pointer of this array, using which the values are access and printed in main() function.

```
#include <stdio.h>
#include <math.h>
void  arrfunction(int, float *);
int main(){
   int x=100;
   float arr[3];
   arrfunction(x, arr);
   printf("Square of %d: %f\n", x, arr[0]);
   printf("cube of %d: %f\n", x, arr[1]);
   printf("Square root of %d: %f\n", x, arr[2]);
   return 0;
}
void arrfunction(int x, float *arr){
   arr[0]=pow(x,2);
   arr[1]=pow(x, 3);
   arr[2]=pow(x, 0.5);
}
```

Output ☑️

```
Square of 100: 10000.000000
cube of 100: 1000000.000000
Square root of 100: 10.000000
```

Return static array

You can use a static array within the function and return a pointer to that array. However, remember that static arrays have limitations, such as being shared across function calls and potentially being overwritten in multithreaded environments.

```
#include <stdio.h>

int* getArray() {
    static int arr[5] = {1, 2, 3, 4, 5};  // Static array
    return arr;  // Returning the pointer to the static array
}

int main() {
    int *array;
    array = getArray();

    printf("Array elements: ");
    for (int i = 0; i < 5; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");

    return 0;
}
```

Output ☑️
Array elements: 1 2 3 4 5

Returning a Dynamically Allocated Array

Another method is to use dynamic memory allocation. This approach ensures that each call to the function gets a separate array, but it requires manual memory management.

The **malloc()** function is available as a library function in the **stdlib.h** header file. It dynamically allocates a block of memory during the runtime of a program. Normal declaration of variables causes the memory to be allocated at the compile time.

The malloc() function returns a generic void pointer. To assign values of a certain data type in the allocated memory, it must be typecast to the required type. For example, to store an int data, it must be typecast to int * as follows –

```
int *x = (int *)malloc(sizeof(int);
```

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

int* createArray(int size) {
   int *arr = (int*)malloc(size * sizeof(int));  // Dynamically allocate memory
   if (arr == NULL) {
      printf("Memory allocation failed\n");
      exit(1);  // Exit if memory allocation fails
   }
 for (int i = 0; i < size; i++) {
      arr[i] = i + 1;  // Initialize the array
   }

   return arr;  // Return the pointer to the allocated memory
}

int main() {
   int size = 5;

   int *array = createArray(size);
   printf("Array elements: ");
```

```
for (int i = 0; i < size; i++) {
      printf("%d ", array[i]);
   }
   printf("\n");

   free(array);  // Free the allocated memory

   return 0;
}
```

Output ☑️
Array elements: 1 2 3 4 5

Command line arguments with main()

Command-line arguments in C are passed to the **main()** function. The **main()** function can take two arguments: **argc** and **argv**. Here, **argc** (argument count) represents the number of command-line arguments, and **argv** (argument vector) is an array of pointers to the arguments, which are strings.

Structure of main() with Command-Line Arguments

```
int main(int argc, char *argv[]) {
   // Your code here
}
```

- **argc**: An integer that represents the number of command-line arguments. This includes the name of the program itself.
- **argv**: An array of pointers to strings (character arrays). Each element points to a command-line argument.

Example:

```
#include <stdio.h>

int main(int argc, char *argv[]) {
   printf("Number of arguments: %d\n", argc);

   for (int i = 0; i < argc; i++) {
      printf("Argument %d: %s\n", i, argv[i]);
   }

   return 0;
}
```

Save this file as cmdargs.c

Explanation

1. **Include Standard I/O Library**:
    - **#include <stdio.h>**: This includes the standard input-output library needed for **printf**.
2. **Main Function with Arguments**:
    - **int main(int argc, char *argv[])**: The **main()** function is defined with **argc** and **argv** parameters to handle command-line arguments.
3. **Print Number of Arguments**:
    - **printf("Number of arguments: %d\n", argc);**: This prints the total number of command-line arguments, including the program name.
4. **Loop Through Arguments**:
    - **for (int i = 0; i < argc; i++)**: This loop iterates over each command-line argument.
    - **printf("Argument %d: %s\n", i, argv[i]);**: This prints each argument along with its index.

Running the Program

To compile and run the program, follow these steps:

1.Compile the Program:

```
gcc -o cmdargs cmdargs.c
```

2.Run the Program with Command-Line Arguments:

```
./cmdargs arg1 arg2 arg3
```

Output

If you run the program as **`./cmdargs arg1 arg2 arg3`,** the output will be:

```
Number of arguments: 4
Argument 0: ./cmdargs
Argument 1: arg1
Argument 2: arg2
Argument 3: arg3
```

Lexical Scoping

Lexical scoping (also known as static scoping) in C refers to the region of the source code where a particular variable binding is valid. This scope is determined by the physical structure of the code, specifically the blocks

(enclosed in curly braces `{}`) in which the variables are declared. Understanding lexical scoping is essential for correctly managing variable lifetimes and avoiding name conflicts in C.

Key Concepts of Lexical Scoping

1. **Block Scope**:
    - Variables declared within a block (a pair of curly braces `{}`) are only accessible within that block and any nested blocks.
    - Once the block ends, the variable goes out of scope and cannot be accessed outside of it.
2. **Function Scope**:
    - Labels (used in `goto` statements) are the only identifiers that have function scope, meaning they can be accessed anywhere within the function in which they are declared.
3. **File Scope**:
    - Variables declared outside of any function have file scope, meaning they are accessible from the point of declaration to the end of the file.
4. **Function Prototype Scope**:
    - Parameters in function prototypes have function prototype scope, meaning they are only accessible within the function prototype itself.

Examples of Lexical Scoping

Block Scope

```
#include <stdio.h>

int main() {
   int x = 10;  // x has block scope within main()

   if (x > 5) {
      int y = 20;  // y has block scope within this if block
      printf("x = %d, y = %d\n", x, y);
   }
```

```
 // y is not accessible here, it would cause a compile error
 // printf("y = %d\n", y);  // Uncommenting this line would cause an error

   return 0;
}
```

Output ☑️
x = 10, y = 20

File Scope

```
#include <stdio.h>

int globalVar = 100;  // globalVar has file scope

void function1() {
   printf("globalVar in function1: %d\n", globalVar);
}
int main() {
   printf("globalVar in main: %d\n", globalVar);
   function1();
   return 0;
}
```

Output ☑️
globalVar in main: 100
globalVar in function1: 100

Function Scope (Labels)

```
#include <stdio.h>

int main() {
   int x = 1;

   if (x == 1) {
      goto label;  // label has function scope
   }

label:
   printf("This is the label\n");

   return 0;
}
```

Output ☑️
This is the label

Differences Between Lexical and Dynamic Scoping

- Lexical Scoping:
    - Determined at compile-time.
    - The scope of a variable is based on the program text and is defined by the blocks in which the variable is declared.
    - Most programming languages, including C, use lexical scoping.
- Dynamic Scoping:
    - Determined at runtime.
    - The scope of a variable is based on the call stack at the time the variable is accessed.
    - Less common in modern programming languages.

Benefits of Lexical Scoping

1. Predictability:
    - Lexical scoping makes it easier to reason about and predict the behavior of code, as the scope of variables is fixed and can be determined by simply reading the code.
2. Encapsulation:
    - Lexical scoping helps in encapsulating variables within specific blocks, reducing the likelihood of unintended side effects and name conflicts.
3. Optimization:
    - Compilers can more easily optimize code with lexical scoping because the scope of each variable is known at compile time.

Nested Function in C

We may think that defining a function inside another function is known as "nested function". But the reality is that it is not a nested function, it is treated as lexical scoping. Lexical scoping is not valid in C because the compiler cant reach/find the correct memory location of the inner function.

Nested functions are **not supported** by C because we cannot define a function within another function in C. We can declare a function inside a function, but it's not a nested function.

Because nested function definitions can not access local variables of the surrounding blocks, they can access only global variables of the containing module. This is done so that lookup of global variables doesn't have to go through the directory. As in C, there are two nested scopes: local and global (and beyond this, built-ins). Therefore, nested functions have only a limited use. If we try to approach nested function in C, then we will get compile time error.

```
// C program to illustrate the
// concept of Nested function.
#include <stdio.h>
int main(void)
{
        printf("Main");
        int fun()
        {
                printf("fun");

                // defining view() function inside fun() function.
                int view()
                {
                        printf("view");
                }
                return 1;
        }
        view();
}
```

Output ☑️

Compile time error: undefined reference to `view'

An extension of the GNU C Compiler allows the declarations of nested functions. The declarations of nested functions under GCC's extension need to be prefix/start with the auto keyword.

```
// C program of nested function
// with the help of gcc extension
#include <stdio.h>
int main(void)
{
        auto int view(); // declare function with auto keyword
        view(); // calling function
        printf("Main\n");

        int view()
        {
                printf("View\n");
                return 1;
        }

        printf("Hello");
        return 0;
}
```

Output ☑️
view
Main
Hello

Variadic Functions in C

Variadic functions in C are functions that accept a variable number of arguments. The standard library provides mechanisms to handle these variable arguments, allowing functions like **printf** and **scanf** to work with a flexible number of parameters.

To create a variadic function, you need to include the <stdarg.h> header, which provides the macros needed to handle the variable arguments. These macros are:

- **va_list**: This type is used to declare a variable that will refer to each argument in turn.
- **va_start**: This macro initializes a **va_list** variable to point to the first variable argument.
- **va_arg**: This macro retrieves the next argument in the parameter list.
- **va_end**: This macro ends the traversal of the variable arguments.

Example:

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

// Function that takes a variable number of arguments and returns their sum

int sum(int count, ...) {
   va_list args;
   int total = 0;

   // Initialize the argument list
   va_start(args, count);

   // Iterate over the arguments
   for (int i = 0; i < count; i++) {
      total += va_arg(args, int);
   }
// Clean up the argument list
 va_end(args);
   return total;
}
int main() {
   printf("Sum of 1, 2, 3, 4, 5: %d\n", sum(5, 1, 2, 3, 4, 5));
 printf("Sum of 10, 20, 30: %d\n", sum(3, 10, 20, 30));
   printf("Sum of 7, 14: %d\n", sum(2, 7, 14));
     return 0;
}
```

Output ☑️

```
Sum of 1, 2, 3, 4, 5: 15
Sum of 10, 20, 30: 60
Sum of 7, 14: 21
```

Explanation:

1. **Function Definition**:
    - **int sum(int count, ...)**: The **sum** function takes an integer **count** followed by a variable number of arguments (denoted by the ellipsis ...).
2. **Initialize va_list**:
    - **va_list args;**: Declares a **va_list** variable **args** to iterate over the additional arguments.
    - **va_start(args, count);**: Initializes **args** to point to the first argument after **count**.
3. **Retrieve Arguments**:
    - va_arg(args, int);: Retrieves the next argument in the list, treating it as an **int**.
    - The loop iterates count times, adding each argument to **total**.
4. **Clean Up**:
    - **va_end(args);**: Ends the traversal of the arguments and performs necessary clean-up.
5. **Main Function**:
    - Demonstrates calling the **sum** function with different numbers of arguments.

Variadic Function Best Practices:

- **Type Safety**: Ensure that the type of each argument matches what the function expects, as **va_arg** requires the correct type to function properly.

- **Argument Count**: The first parameter typically indicates the number of subsequent arguments to help the function know how many arguments to process.
- **Default Arguments**: Unlike languages like C++ or Python, C does not support default arguments directly. You must manage argument counts and defaults manually.

Callback Function

A callback function is a function that is passed as an argument to another function, which can then call the callback function at a suitable time. This is a powerful technique in C programming to allow for customizable code behavior, such as handling events, implementing callbacks for asynchronous operations, or designing flexible APIs.

Example: Simple Callback Function

Below is a simple example to demonstrate the concept of a callback function in C. In this example, we will create a perform_operation function that takes two integers and a function pointer (callback function) as arguments. The callback function will define the operation to be performed on the two integers.

```
#include <stdio.h>

// Define a type for the callback function
typedef int (*operation_callback)(int, int);

// Function that adds two numbers
int add(int a, int b) {
   return a + b;
}
```

```
// Function that multiplies two numbers
int multiply(int a, int b) {
    return a * b;
}

// Function that performs an operation on two numbers using a callback
void perform_operation(int x, int y, operation_callback op) {
    int result = op(x, y);
    printf("Result: %d\n", result);
}


int main() {
    int num1 = 10, num2 = 5;

    // Perform addition
    perform_operation(num1, num2, add);

    // Perform multiplication
    perform_operation(num1, num2, multiply);

    return 0;
}
```

Output ☑️
Result: 15
Result: 50

Explanation

1. **Define a Type for the Callback Function**:
    - **typedef int (*operation_callback)(int, int);**: This defines a new type **operation_callback** that represents a function pointer to a function taking two **int** arguments and returning an **int**.
2. **Callback Functions**:

- **int add(int a, int b)**: A function that adds two integers.
- **int multiply(int a, int b)**: A function that multiplies two integers.

3. **Function that Uses the Callback**:
   - **void perform_operation(int x, int y, operation_callback op)**: This function takes two integers and a function pointer as arguments. It calls the callback function **op** with **x** and **y** as arguments and prints the result.
4. **Main Function**:
   - **perform_operation(num1, num2, add);**: Calls **perform_operation** with the **add** function as the callback.
   - **perform_operation(num1, num2, multiply);**: Calls **perform_operation** with the **multiply** function as the callback.

Callback Function with Arguments

In C, you can pass arguments to a callback function by defining the callback function to accept specific parameters and then passing the arguments when calling the callback. Below is an example to illustrate how you can achieve this:

Example: Callback Function with Arguments

In this example, we will create a function that accepts a callback function and arguments to pass to the callback function.

```
#include <stdio.h>
// Define a type for the callback function
typedef void (*callback_t)(int, int);
// Callback function to add two numbers
void add(int a, int b) {
    printf("Addition: %d + %d = %d\n", a, b, a + b);
}

// Callback function to subtract two numbers
void subtract(int a, int b) {
    printf("Subtraction: %d - %d = %d\n", a, b, a - b);
}

// Function that uses a callback function
void perform_operation(callback_t callback, int x, int y) {
    // Call the callback function with provided arguments
    callback(x, y);
}

int main() {
    int num1 = 10, num2 = 5;

    // Perform addition using callback
    perform_operation(add, num1, num2);

    // Perform subtraction using callback
    perform_operation(subtract, num1, num2);

    return 0;
}
```

Output ☑️
Addition: 10 + 5 = 15
Subtraction: 10 - 5 = 5

Explanation

1. **Define a Type for the Callback Function**:
    - **typedef void (*callback_t)(int, int);**: This defines a new type **callback_t** that represents a function pointer to a function taking two **int** arguments and returning **void**.
2. **Callback Functions**:
    - **void add(int a, int b)**: A function that adds two integers and prints the result.
    - **void subtract(int a, int b)**: A function that subtracts two integers and prints the result.
3. **Function that Uses the Callback**:
    - **void perform_operation(callback_t callback, int x, int y)**: This function takes a callback function and two integers as arguments. It calls the callback function with **x** and **y** as arguments.
4. **Main Function**:
    - **perform_operation(add, num1, num2);**: Calls **perform_operation** with the **add** function as the callback and passes **num1** and **num2** as arguments.
    - **perform_operation(subtract, num1, num2);**: Calls **perform_operation** with the **subtract** function as the callback and passes **num1** and **num2** as arguments.

Recursive Function in C

A recursive function in C is a function that calls itself in order to solve a smaller instance of the same problem. This approach is commonly used

for tasks that can be broken down into similar sub-tasks, such as calculating factorials, Fibonacci numbers, or solving problems .

Here's an example of a recursive function in C that calculates the factorial of a number:

```
#include <stdio.h>

// Function prototype
int factorial(int n);

int main() {
    int number;
    printf("Enter a positive integer: ");
    scanf("%d", &number);

    // Calculate factorial using the recursive function
    printf("Factorial of %d = %d\n", number, factorial(number));
    return 0;
}

// Recursive function to calculate factorial
int factorial(int n) {
    if (n <= 1) { // Base case: factorial of 0 or 1 is 1
        return 1;
    } else { // Recursive case
        return n * factorial(n - 1);
    }
}
```

Output ☑️
Enter a positive integer: 5
Factorial of 5 = 120

Explanation

1.Function Prototype:

```
int factorial(int n);
```

This line declares the prototype of the factorial function, which takes an integer n as an argument and returns an integer.

2.Main Function:

```
int main() {
   int number;
   printf("Enter a positive integer: ");
   scanf("%d", &number);

   // Calculate factorial using the recursive function
   printf("Factorial of %d = %d\n", number, factorial(number));
   return 0;
}
```

Variable Declaration:

- **int number;**: Declares an integer variable **number** to store the user input.

User Input:

- **printf("Enter a positive integer: ");**: Prompts the user to enter a positive integer.
- **scanf("%d", &number);**: Reads the integer input from the user and stores it in number.

Factorial Calculation and Output:

- **printf("Factorial of %d = %d\n", number, factorial(number));**: Calls the **factorial** function with **number** as the argument, calculates the factorial, and prints the result.

Return Statement:

- **return 0;**: Returns 0 to indicate that the program has ended successfully.

3.Recursive Function:

```
int factorial(int n) {
   if (n <= 1) { // Base case: factorial of 0 or 1 is 1
      return 1;
   } else { // Recursive case
      return n * factorial(n - 1);
   }
}
```

- **Base Case**:
  - **`if (n <= 1) { return 1; }`**: If n is 0 or 1, the function returns 1. This is the base case for the recursion. The factorial of 0 or 1 is defined as 1.
- **Recursive Case**:
  - **`else { return n * factorial(n - 1); }`**: If **`n`** is greater than 1, the function calls itself with **`n - 1`** and multiplies the result by **`n`**. This reduces the problem size with each recursive call until it reaches the base case.

Example

Let's walk through an example with **`number = 4`**:

- **`factorial(4)`**:
  - **`4 * factorial(3)`**
    - **`3 * factorial(2)`**
      - **`2 * factorial(1)`**
        - **`1 (base case)`**

So, the calculation is as follows: 4∗3∗2∗1=244 * 3 * 2 * 1 = 244∗3∗2∗1=24

Therefore, **`factorial(4)`** returns 24, and the program prints:

Storage Classes

Auto

- **Scope**: Local to the block (typically a function) in which the variable is defined. It cannot be accessed outside of this block.
- **Lifetime**: The variable exists only during the execution of the block in which it is defined. Once the block is exited, the variable is destroyed.
- **Default Behavior**: If you do not explicitly specify a storage class for a local variable, it is treated as auto by default.
- **Storage**: Variables declared with the auto storage class are stored in the stack.
- **Usage**: Primarily used for local variables within functions, to store temporary data.

Syntax:

extern int d;

Example:

```
auto x = 10;      // x is inferred to be of type int
auto y = 3.14;    // y is inferred to be of type double
```

Static

The static keyword in C is used to define variables with a specific scope and lifetime. It can be applied to both local and global variables, and even to functions. Here are the main uses of static:

Static Variables

1. **Static Local Variables**:

- **Scope**: Local to the block in which the variable is defined.
- **Lifetime**: Retains its value between function calls, existing for the entire duration of the program.
- **Usage**: Useful for preserving state information across multiple function calls.

2. **Example**:

```
#include <stdio.h>

void counter() {
   static int count = 0; // Static local variable
   count++;
   printf("Count: %d\n", count);

}

int main() {
   counter(); // Output: Count: 1
   counter(); // Output: Count: 2
   counter(); // Output: Count: 3
   return 0;
}
```

Output ☑️

```
Count: 1
Count: 2
Count: 3
```

Register

The register keyword in C is used to suggest to the compiler that the variable should be stored in a CPU register instead of RAM. This is to optimize the performance of the program by allowing faster access to frequently used variables.

Key Points about register

1. **Scope**: Local to the block in which the variable is defined.

2. **Lifetime**: Exists only during the execution of the block in which it is defined.
3. **Optimization**: The variable may be stored in a CPU register for faster access, but this is not guaranteed.
4. **Restrictions**:
    - You cannot take the address of a register variable. Therefore, you cannot use the & operator with a register variable.
    - Typically used for variables that are heavily used within a loop or function.

Syntax

register int counter;

Example

```
#include <stdio.h>

void countToN(int n) {
   register int i;
   for (i = 0; i < n; i++) {
      printf("%d\n", i);
   }
}

int main() {
   countToN(10);
   return 0;
}
```

Output ☑️

```
0
1
2
3
4
5
6
7
8
9
```

In this example:

- The variable i is declared as a register variable.
- The compiler is advised to store i in a CPU register for quicker access, especially since it is used in a loop.

Important Notes

- **Compiler Discretion**: Modern compilers are very good at optimizing code and may ignore the register keyword because they have their own optimization strategies. In many cases, the compiler's decision will be more efficient than manual suggestions.
- **Limited Number of Registers**: CPUs have a limited number of registers, so not all register variables can actually be placed in registers. The compiler decides which variables will be placed in registers.
- **Legacy**: In modern C programming, the register keyword is less commonly used because compilers have become better at automatically optimizing variable storage.

Extern

The extern keyword in C is used to declare a global variable or function in another file. It extends the visibility of the C variables and C functions across multiple files. It is commonly used to share variables and functions among different files of a program.

Key Points about extern

1. **Scope**: Global; it can be accessed by any file that includes its declaration.
2. **Lifetime**: Exists for the entire duration of the program.
3. **Usage**: Used to declare a variable or a function that is defined in another file.

Syntax

```
extern int var;    // Declaration of a global variable
extern void func(); // Declaration of a function
```

Example

File1.c (Defining the variable and function)

```
#include <stdio.h>

int sharedVar = 42; // Definition of the global variable

void display() {    // Definition of the function
   printf("sharedVar = %d\n", sharedVar);
}

int main() {
   display();     // Call the function
   return 0;
}
```

File2.c (Using the variable and function defined in File1.c)

```
#include <stdio.h>

extern int sharedVar; // Declaration of the global variable
extern void display(); // Declaration of the function

int main() {
   sharedVar = 100; // Modify the global variable
   display();      // Call the function
   return 0;
}
```

Explanation

- In File1.c:
  - int sharedVar = 42; defines a global variable sharedVar.
  - void display() defines a function display that prints the value of sharedVar.
- In File2.c:
  - extern int sharedVar; declares the global variable sharedVar defined in File1.c.
  - extern void display(); declares the function display defined in File1.c.
  - The main function in File2.c modifies sharedVar and calls the display function, which reflects the updated value of sharedVar.

Points to Remember

1. **Declaration vs. Definition**:
   - **Declaration**: Tells the compiler about the existence and type of a variable or function, without allocating storage.

    - **Definition**: Allocates storage for the variable or function.
    - extern is used for declarations.
2. **Multiple Files**:
    - extern is useful when a project is split across multiple files.
    - It allows sharing of global variables and functions across files.
3. **Global Scope**:
    - Variables declared with extern have global scope, meaning they can be accessed from any file that declares them with extern.

In large projects, it is common to separate the definition and declaration of global variables and functions. The global variables and functions are defined in one file, and other files access them using extern. This helps in organizing code better and reusing global variables and functions across multiple files.

ADVANCED POINTERS

Null Pointers

A null pointer is a pointer that doesn't point to any valid memory location. It's often used to indicate that the pointer is not currently referencing any object or memory. The null pointer is defined in several standard header files, such as stddef.h, stdlib.h, and stdio.h, and is typically defined as NULL.

Example

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

int main() {
   int *ptr = NULL; // Initialize a pointer to NULL

   if (ptr == NULL) {
      printf("The pointer is null.\n");
   } else {
      printf("The pointer is not null.\n");
   }
   return 0;
}
```

Output ☑️

```
The pointer is null.
```

Void pointers

A void pointer, also known as a generic pointer, is a special type of pointer in C that can point to any data type. It is declared using the void keyword. Void pointers are versatile because they can be assigned to any other type of pointer without needing an explicit cast, although dereferencing them requires casting to the appropriate type.

Example

```
#include <stdio.h>
int main () {
int i = 10;
   float f = 5.5;
   char c = 'A';
  void *ptr; // Declare a void pointer

   // Point to an integer
   ptr = &i;
   printf("The value of the integer is: %d\n", *((int *)ptr));

   // Point to a float
   ptr = &f;
   printf("The value of the float is: %.1f\n", *((float *)ptr));

   // Point to a char
   ptr = &c;
   printf("The value of the char is: %c\n", *((char *)ptr));

   return 0;
}
```

Output ☑️

The value of the integer is: 10

The value of the float is: 5.5

The value of the char is: A

Dangling Pointers

A dangling pointer is a pointer that continues to reference a memory location even after the memory has been deallocated or freed. Accessing a dangling pointer leads to undefined behavior, including potential program crashes or data corruption. It's important to handle pointers carefully to avoid creating dangling pointers.

Example

```
// C program to demonstrate Deallocating a memory pointed by
// ptr causes dangling pointer
#include <stdio.h>
#include <stdlib.h>

int main()
{
   int* ptr = (int*)malloc(sizeof(int));

   // After below free call, ptr becomes a dangling pointer
   free(ptr);
   printf("Memory freed\n");

   // removing Dangling Pointer
   ptr = NULL;

   return 0;
}
```

Dereference Pointers

Dereferencing a pointer means accessing the value stored at the memory location to which the pointer points. This is done using the dereference operator *

```
#include <stdio.h>

int main() {
   int value = 42; // A regular integer variable
   int *ptr = &value; // Pointer to the integer variable

    // Print the value using the pointer
    printf("Value using pointer: %d\n", *ptr);
    // Change the value using the pointer
   *ptr = 100;
   printf("Modified value using pointer: %d\n", *ptr);
   printf("Modified value of the original variable: %d\n", value);

   return 0;
}
```

Output ☑️

Value using pointer: 42

Modified value using pointer: 100

Modified value of the original variable: 100

Near, Far and Huge Pointers

In the context of C programming, especially on older DOS and Windows systems running on x86 architecture, there were different types of pointers: near, far, and huge. These types were used to manage different memory models due to the segmented memory architecture of the x86 processors.

Near Pointers

- **Definition**: Near pointers are 16-bit pointers that can address memory within the current segment.
- **Usage**: They are used for accessing data within a single segment (64 KB).
- **Limitation**: Limited to a 64 KB segment because they don't specify a segment, only an offset.

Example of Near Pointers

```
// C Program to demonstrate the use of near pointer
#include <stdio.h>

int main()
{
        // declaring a near pointer
        int near* ptr;

        // size of the near pointer
        printf("Size of Near Pointer: %d bytes", sizeof(ptr));
        return 0;
}
```

Output ☑️

Size of Near Pointer: 2 bytes

Far Pointers

- **Definition**: Far pointers are 32-bit pointers that include both a segment and an offset.
- **Usage**: They can access memory across different segments.
- **Components**: Consist of a 16-bit segment selector and a 16-bit offset, enabling access to a 1 MB address space (20-bit address).

Example

```
// C Program to find the size of far pointer
#include <stdio.h>

int main()
{
        // declaring far pointer
        int far* ptr;

        // Size of far pointer
        printf("Size of Far Pointer: %d bytes", sizeof(ptr));
        return 0;
}
```

Output ☑️

Size of Far Pointer: 4 bytes

Huge Pointers

- **Definition**: Huge pointers are similar to far pointers but are normalized.
- **Usage**: They are used for accessing large data structures that might span multiple segments.
- **Normalization**: When a huge pointer is incremented, it is automatically normalized to ensure that the segment part of the address remains within a canonical form, eliminating segmentation issues.

Example

```
// C Program to find the size of the huge pointer
#include <stdio.h>

int main()
{
        // declaring the huge pointer
        int huge* ptr;

        // size of huge pointer
        printf("Size of the Huge Pointer: %d bytes",
                sizeof(ptr));
        return 0;
}
```

In modern C programming, particularly on contemporary systems like x86-64 architectures and standard compilers, the concept of "huge" pointers is outdated. The terminology of near, far, and huge pointers originated from the MS-DOS and 16-bit Windows days when different memory models were used to manage segmented memory.

Note that huge is not a standard qualifier in modern C, and modern compilers will likely treat huge as a custom identifier rather than a special pointer type. Therefore, this code will behave just like it would with a regular pointer

2024

C STRUCTURES

C Structures

A structure (struct) in C is a derived or user-defined data type which allows the grouping of variables of different types under a single name. Structures are useful for organizing complex data in a readable and manageable way.

The difference between an array and a structure is that an array is a homogenous collection of similar types, whereas a structure can have elements of different types stored adjacently and identified by a name.

Syntax of Structure Declaration

```
struct [structure tag]{
  member definition;
  member definition;
  ...
  member definition;
} [one or more structure variables];
```

The structure tag is optional and each member definition is a normal variable definition,

such as "int i;" or "float f;" or any other valid variable definition.

At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional.

Example

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

// Define a structure to represent a point in 2D space
struct Point {
   int x;
   int y;
}
```

```
// Define a structure to represent a person
struct Person {
    char name[50];
    int age;
};
int main() {
    // Declare and initialize a Point structure variable
    struct Point p1;
    p1.x = 10;
    p1.y = 20;

    // Print the values of the Point structure
    printf("Point p1: (%d, %d)\n", p1.x, p1.y);

// Declare and initialize a Person structure variable
    struct Person person1;
    person1.age = 30;
    // Use strcpy to copy the name since person1.name is an array
    strcpy(person1.name, "John Doe");

    // Print the values of the Person structure
    printf("Person: %s, Age: %d\n", person1.name, person1.age);

    return 0;
}
```

Output ☑️

Point p1: (10, 20)

Person: John Doe, Age: 30

Array of structures

An array of structures in C allows you to store multiple instances of a structure, making it easier to manage collections of related data. This is particularly useful when dealing with records or objects that share the same attributes.

Usually, a struct type is defined at the beginning of the code so that its type can be used inside any of the functions. You can declare an array of structures and later on fill data in it or you can initialize it at the time of declaration itself.

Example 1: Array of Structures for Student Records

```
#include <stdio.h>
#include <string.h>
// Define a structure to represent a student
struct Student {
    char name[50];
    int age;
    float grade;
};

int main() {
    // Declare an array of 3 Student structures
    struct Student students[3];

    // Initialize the first student
    strcpy(students[0].name, "Alice");

    students[0].age = 20;
    students[0].grade = 88.5;

// Initialize the second student
    strcpy(students[1].name, "Bob");
    students[1].age = 21;
    students[1].grade = 91.2;
```

```
// Initialize the third student
   strcpy(students[2].name, "Charlie");
   students[2].age = 19;
   students[2].grade = 79.4;

   // Print the details of all students
   for (int i = 0; i < 3; i++) {
      printf("Student %d: %s, Age: %d, Grade: %.2f\n", i + 1, students[i].name, students[i].age,
students[i].grade);
   }
   return 0;
}
```

Output ☑️

Student 1: Alice, Age: 20, Grade: 88.50

Student 2: Bob, Age: 21, Grade: 91.20

Student 3: Charlie, Age: 19, Grade: 79.40

Example 2: Array of Structures for Book Records

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

// Define a structure to represent a book
struct Book {
   char title[100];
   char author[50];
   int year;
};

int main() {
   // Declare an array of 5 Book structures
   struct Book library[5];

   // Initialize the books in the library
   strcpy(library[0].title, "The Catcher in the Rye");
   strcpy(library[0].author, "J.D. Salinger");
   library[0].year = 1951;
   strcpy(library[1].title, "To Kill a Mockingbird");
   strcpy(library[1].author, "Harper Lee");
     library[1].year = 1960;

   strcpy(library[2].title, "1984");
   strcpy(library[2].author, "George Orwell");
   library[2].year = 1949;

   strcpy(library[3].title, "Pride and Prejudice");
   strcpy(library[3].author, "Jane Austen");
   library[3].year = 1813;

   strcpy(library[4].title, "The Great Gatsby");
   strcpy(library[4].author, "F. Scott Fitzgerald");
   library[4].year = 1925;
```

```
    // Print the details of all books in the library
    printf("Library Catalog:\n");
    for (int i = 0; i < 5; i++) {
        printf("Book %d:\n", i + 1);
        printf("Title: %s\n", library[i].title);
        printf("Author: %s\n", library[i].author);
        printf("Year of Publication: %d\n", library[i].year);
        printf("\n");
    }

    return 0;
}
```

Output ☑️

Library Catalog:

Book 1:

Title: The Catcher in the Rye

Author: J.D. Salinger

Year of Publication: 1951

Book 2:

Title: To Kill a Mockingbird

Author: Harper Lee

Year of Publication: 1960

Book 3:

Title: 1984

Author: George Orwell

Year of Publication: 1949

Book 4:

Title: Pride and Prejudice

Author: Jane Austen

Year of Publication: 1813

Book 5:

Title: The Great Gatsby

Author: F. Scott Fitzgerald

Year of Publication: 1925

Enumeration

Enumeration (enum) is a user-defined data type used to assign names to integral constants, making the code more readable and easier to maintain. Enumerations are particularly useful when you have a fixed set of related constants that represent different states or options.

Example

```
#include <stdio.h>

// Define an enumeration for days of the week
enum Weekday {
   Sunday,    // 0
   Monday,    // 1
   Tuesday,   // 2
   Wednesday, // 3
   Thursday,  // 4
   Friday,    // 5
   Saturday   // 6
};
```

```
int main() {
    enum Weekday today = Wednesday;
    // Using switch statement with enum
    switch (today) {
        case Sunday:
            printf("Today is Sunday.\n");
            break;
        case Monday:
            printf("Today is Monday.\n");
            break;
        case Tuesday:
            printf("Today is Tuesday.\n");
            break;
        case Wednesday:
            printf("Today is Wednesday.\n");
            break;
        case Thursday:
            printf("Today is Thursday.\n");
            break;
        case Friday:
            printf("Today is Friday.\n");
            break;
        case Saturday:
            printf("Today is Saturday.\n");
            break;
        default:
            printf("Invalid day.\n");
            break;
    }
    // Enumerations are essentially integers, so you can use them in arithmetic operations
    enum Weekday tomorrow = (today + 1) % 7;
    printf("Tomorrow is %d (which is %s).\n", tomorrow, (tomorrow == Sunday) ?
"Sunday" : "another day");
    return 0;
}
```

Today is Wednesday.

Tomorrow is 4 (which is another day).

UNION

Union

A union is a user-defined data type similar to a struct, but with a key difference: all members of a union share the same memory location. This means that a union can store different data types in the same memory location, but only one type at a time. Unions are useful for memory-efficient storage when only one member will be used at a time.

Example 1

```
#include <stdio.h>
#include <string.h>
// Define a union to represent a data value that can be either an int, a float, or a char array
union Data {
   int i;
   float f;
   char str[20];
};

int main() {
   // Declare a union variable
   union Data data;

   // Assign an integer value to the union
   data.i = 10;
   printf("data.i: %d\n", data.i);

   // Assign a float value to the union (this will overwrite the previous value)
   data.f = 220.5;
   printf("data.f: %.1f\n", data.f);
   // Assign a string value to the union (this will overwrite the previous value)
   strcpy(data.str, "Hello");
   printf("data.str: %s\n", data.str);
   // Note that the previously stored values are now corrupted
   printf("data.i after writing string: %d\n", data.i);
   printf("data.f after writing string: %.1f\n", data.f);
   return 0;.
}
```

Output

```
data.i: 10

data.f: 220.5

data.str: Hello

data.i after writing string: 1819043144

data.f after writing string: 1143139122437582505939828736.0
```

Size of a Union

The size of a union is the size of its largest member. For example, if a union contains two members of char and int types. In that case, the size of the union will be the size of int because int is the largest member.

```c
#include <stdio.h>

// Define a union
union Data {
  int a;
  float b;
  char c[20];
};

int main() {
  union Data data;

  // Printing the size of the each member of union
  printf("Size of a: %lu bytes\n", sizeof(data.a));
  printf("Size of b: %lu bytes\n", sizeof(data.b));
  printf("Size of c: %lu bytes\n", sizeof(data.c));

  // Printing the size of the union
  printf("Size of union: %lu bytes\n", sizeof(data));

  return 0;
}
```

Output ☑️

Size of a: 4 bytes

Size of b: 4 bytes

Size of c: 20 bytes

Size of union: 20 bytes

Real world Scenario Example

```c
#include <stdio.h>
#include <string.h>
// Define a union for sensor data
union SensorData {
   int intValue;
   float floatValue;
   char stringValue[20];
};

// Define a struct that includes the union
struct Sensor {
   char type[10];
   union SensorData data;
};
int main() {
   // Create a sensor variable
   struct Sensor sensor1;

   // Set the sensor type to "int"
   strcpy(sensor1.type, "int");
   sensor1.data.intValue = 100;

   printf("Sensor type: %s\n", sensor1.type);
   printf("Sensor int value: %d\n", sensor1.data.intValue);
   // Change the sensor type to "float"
   strcpy(sensor1.type, "float");
   sensor1.data.floatValue = 99.99;
   printf("Sensor type: %s\n", sensor1.type);
   printf("Sensor float value: %.2f\n", sensor1.data.floatValue);
   return 0;
}
```

Output ☑️

Sensor type: int

Sensor int value: 100

Sensor type: float

Sensor float value: 99.99

Difference between Structure and Union

- In a structure, each member has its own memory location. The size of a structure is at least the sum of the sizes of its individual members, plus any padding needed for alignment purposes. This means that if you have a structure with an int, a float, and a char[10], the memory allocated will be sufficient to store all three members separately.
- In a union, all members share the same memory location. The size of a union is the size of its largest member. This means that if you have a union with an int, a float, and a char[10], the memory allocated will be enough to store the largest member, which in this case is the char[10].
- Structures are ideal for situations where you need to store multiple related data items together and access them independently. For example, in an employee database, you might use a structure to group an employee's ID, name, and salary. Each of these pieces of data can be accessed and modified without affecting the others.
- Unions are useful when you need to store different types of data in the same memory location but only one type at a time. For example, if you are writing a program that needs to process different types of data (such as integers, floats, or strings) but only one type at any given time, a union can help save memory by using the same space for all these types.
- In a structure we can access all members of a structure simultaneously. This allows to read or modify each member independently, which is particularly useful when dealing with related data that needs to be processed together.
- Only one member of a union can holds a valid value at a time. Accessing a different member of the union will overwrite the current value with the new data. This means that when we assign a value to one member, the previously stored value in any other member is lost.

BIT FIELDS

Bit-fields allow you to specify the size (in bits) of structure and union members, which can be especially useful when you want to use memory efficiently. They are typically used when you know that the values of certain fields or groups of fields will never exceed a certain limit or are within a small range.

Bit-fields help conserve memory, especially when dealing with embedded systems or when you have limited storage available in your program. This feature is valuable for optimizing memory usage in situations where it's essential to save space.

Syntax

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

- data_type: This is the data type used to specify the bit-field's value. It can be an integer type, typically int, signed int, or unsigned int, that determines how the bits in the bit-field are to be interpreted.
- member_name: The member name is the identifier that you give to the bit-field, which serves as the name of the bit-field within the structure or union.
- width_of_bit-field: This refers to the number of bits allocated to the bit-field. The width must be less than or equal to the bit width of the specified data type (e.g., int). The width determines how many bits the bit-field can use to represent its value.

```
struct {
  unsigned int widthValidated;
  unsigned int heightValidated;
} status;
```

This structure requires 8 bytes of memory space but in actual, we are going to store either "0" or "1" in each of the variables. The C programming language offers a better way to utilize the memory space in such situations.

If you are using such variables inside a structure, then you can define the width of the variables which tells the C compiler that you are going to use only those many number of bytes. For example, the above structure can be re-written as follows –

```
struct {
  unsigned int widthValidated : 1;
  unsigned int heightValidated : 1;
} status;
```

The above structure requires 4 bytes of memory space for the status variable, but only 2 bits will be used to store the values. Example

Task : Try to print the size of two examples with bit field and without bit field.

Example: Managing Device Status Using Bit Fields

```
#include <stdio.h>

// Define a structure with bit fields to represent device status
struct DeviceStatus {
   unsigned int powerOn : 1;      // 1 bit for power status
   unsigned int error : 1;        // 1 bit for error status
   unsigned int sensorActive : 1; // 1 bit for sensor active status
   unsigned int actuatorActive : 1; // 1 bit for actuator active status
   unsigned int reserved : 4;     // 4 bits reserved for future use
};
int main() {
   // Declare and initialize a structure variable
   struct DeviceStatus status = {0};
   // Set individual status bits
   status.powerOn = 1;
   status.error = 0;
   status.sensorActive = 1;
   status.actuatorActive = 0;
```

```
    // Print the status of each field
    printf("Device Status:\n");
    printf("Power On: %d\n", status.powerOn);
    printf("Error: %d\n", status.error);
    printf("Sensor Active: %d\n", status.sensorActive);
    printf("Actuator Active: %d\n", status.actuatorActive);

    // Modify status bits
    status.error = 1;
    status.actuatorActive = 1;

// Print the modified status
    printf("\nModified Device Status:\n");
    printf("Power On: %d\n", status.powerOn);
    printf("Error: %d\n", status.error);
    printf("Sensor Active: %d\n", status.sensorActive);
    printf("Actuator Active: %d\n", status.actuatorActive);

    // Display the size of the structure
    printf("\nSize of DeviceStatus: %zu bytes\n", sizeof(status));

    return 0;
}
```

Output ☑️

Device Status:

Power On: 1

Error: 0

Sensor Active: 1

Actuator Active: 0

Modified Device Status:

Power On: 1

Error: 1

Sensor Active: 1

Actuator Active: 1

Size of DeviceStatus: 4 bytes

FILE HANDLING

In C, files are used for input and output operations. You can read from and write to files using standard library functions provided in the stdio.h header. Here's an overview of how to handle files in C, along with examples for reading from and writing to files.

Basic File Operations

- Opening a File: Use fopen() to open a file.
- Reading from a File: Use functions like fgetc(), fgets(), fscanf(), or fread().
- Writing to a File: Use functions like fputc(), fputs(), fprintf(), or fwrite()`.
- Closing a File: Use fclose() to close a file.
- Error Handling: Check for errors using ferror() and handle them appropriately.

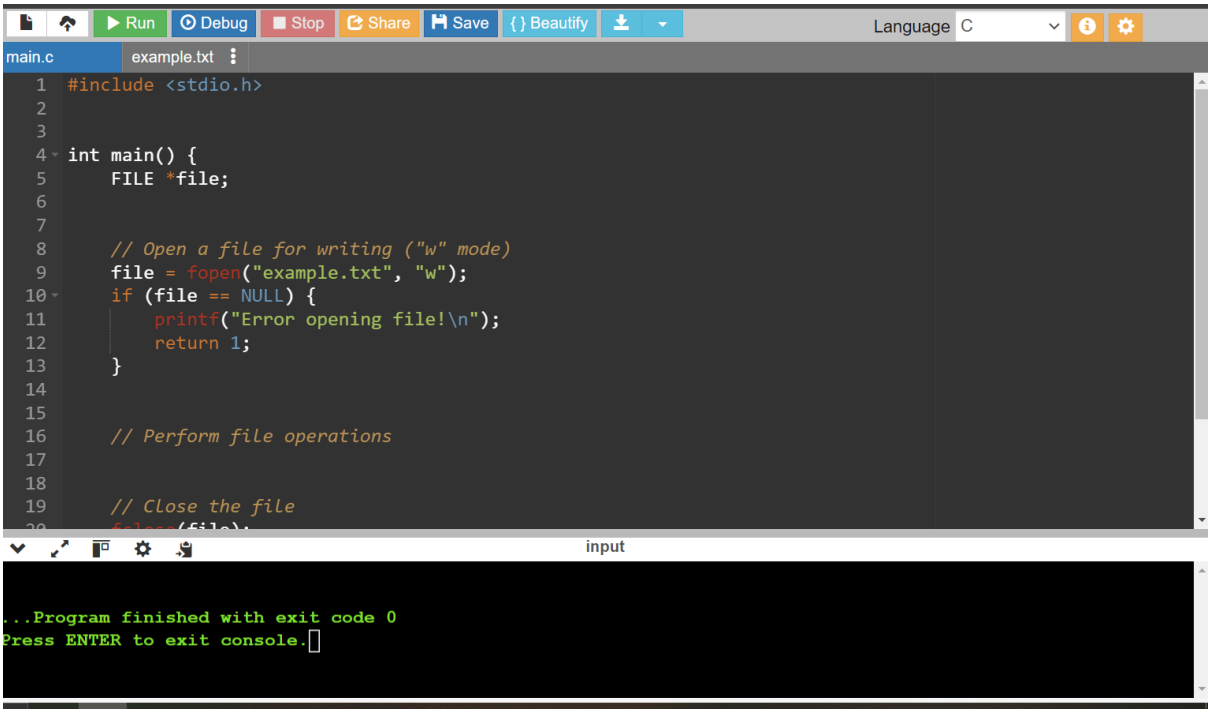
Need of File Handling in C

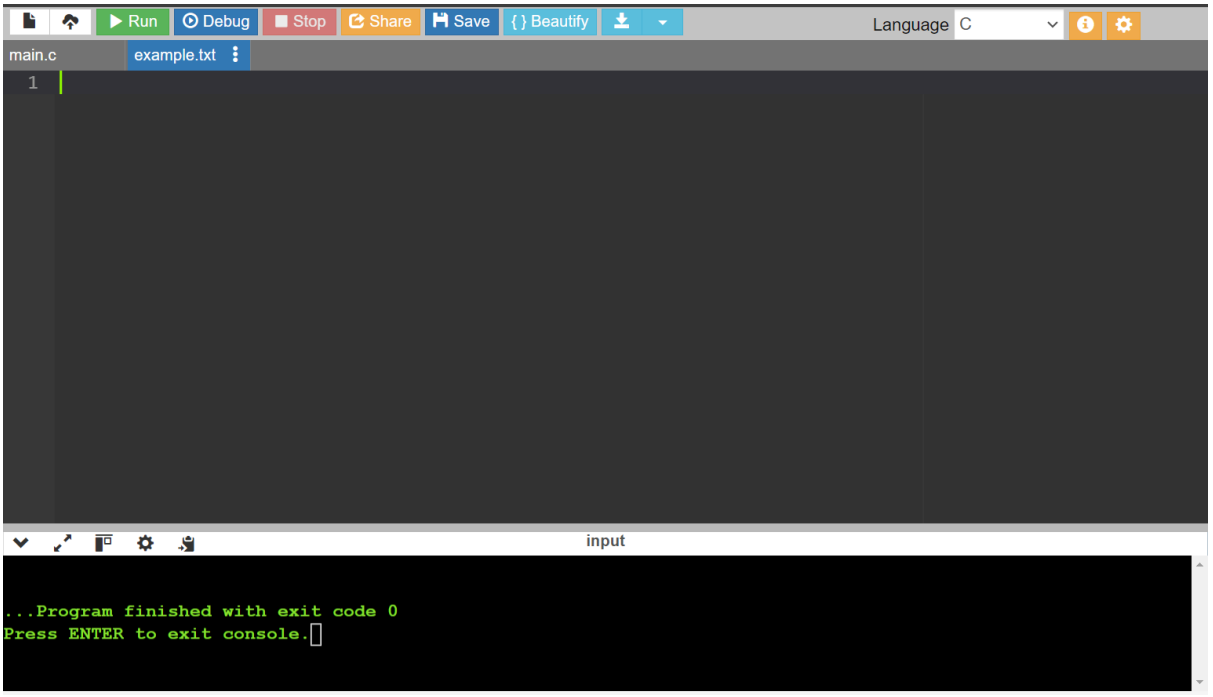
If we perform input and output operations using the C program, the data exists as long as the program is running, when the program is terminated, we cannot use that data again. File handling is required to work with files stored in the external memory i.e., to store and access the information to/from the computer's external memory. You can keep the data permanently using file handling.

Opening and Closing a File

```
#include <stdio.h>

int main() {
   FILE *file;

   // Open a file for writing ("w" mode)
   file = fopen("example.txt", "w");
   if (file == NULL) {
      printf("Error opening file!\n");
      return 1;
   }

   // Perform file operations

   // Close the file
   fclose(file);

   return 0;
}
```

Output ☑️

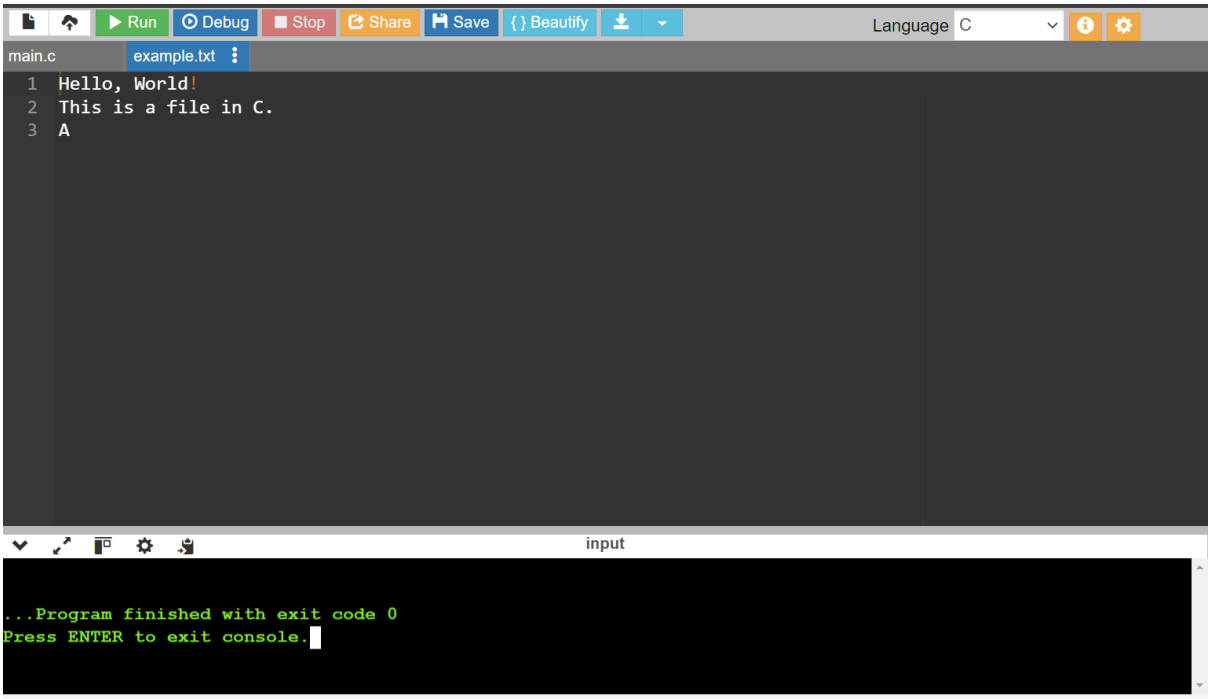

Run
Debug
Stop
Share
Save
{ } Beautify
Language C
main.c
example.txt
#include <stdio.h>
int main() {
FILE *file;
// Open a file for writing ("w" mode)
file = fopen("example.txt", "w");
if (file == NULL) {
printf("Error opening file!\n");
return 1;
}
// Perform file operations
// Close the file
input
...Program finished with exit code 0
Press ENTER to exit console.


Run
Debug
Stop
Share
Save
{ } Beautify
Language C
main.c
example.txt
1
input
...Program finished with exit code 0
Press ENTER to exit console.

Writing to a File

```
#include <stdio.h>

int main() {
   FILE *file;

   // Open a file for writing ("w" mode)
   file = fopen("example.txt", "w");
   if (file == NULL) {
      printf("Error opening file!\n");
      return 1;
   }

   // Write to the file
   fprintf(file, "Hello, World!\n");
   fputs("This is a file in C.\n", file);
   fputc('A', file);

   // Close the file
   fclose(file);

   return 0;
}
```

Output ☑️


Run
Debug
Stop
Share
Save
{ } Beautify
Language C
main.c
example.txt
1 Hello, World!
2 This is a file in C.
3 A
input
...Program finished with exit code 0
Press ENTER to exit console.

Reading from a File

```
#include <stdio.h>

int main() {
   FILE *file;
   char buffer[100];

   // Open a file for reading ("r" mode)
   file = fopen("example.txt", "r");
   if (file == NULL) {
      printf("Error opening file!\n");
      return 1;
   }

   // Read from the file
   while (fgets(buffer, sizeof(buffer), file) != NULL) {
      printf("%s", buffer);
   }

   // Close the file
   fclose(file);

   return 0;
}
```

Example: Complete Program with Error Handling

This example demonstrates a complete program that writes to a file, then reads from it, with proper error handling.

```
#include <stdio.h>
int main() {
   FILE *file;
   char buffer[100];
   // Open a file for writing
   file = fopen("example.txt", "w");
   if (file == NULL) {
      printf("Error opening file for writing!\n");
      return 1;
   }

   // Write to the file
   if (fprintf(file, "Hello, World!\n") < 0) {
      printf("Error writing to file!\n");
      fclose(file);
      return 1;
   }

   // Close the file
   fclose(file);
   // Open the file for reading
   file = fopen("example.txt", "r");
   if (file == NULL) {
      printf("Error opening file for reading!\n");
      return 1;
   }
   // Read from the file
   while (fgets(buffer, sizeof(buffer), file) != NULL) {
      printf("%s", buffer);
   }
   // Close the file
   fclose(file);
   return 0;
}
```

PREPROCESSORS

Header FIles

Header files in C are essential for organizing and managing code, especially in larger projects. They allow you to separate declarations from implementations and make your code more modular and reusable. Header files typically contain declarations of functions, macros, constants, and data types, while the actual definitions (implementations) are placed in source files.

Key Concepts of Header Files

- **Function Declarations**: Declare functions that are defined in other source files.
- **Macros**: Define constants or macro functions.
- **Data Type Definitions**: Define structures, enums, and typedefs.
- **External Variable Declarations**: Declare variables that are defined in other source files.
- **Include Guards**: Prevent multiple inclusions of the same header file.

Example: Creating and Using a Header File

Step 1: Create a Header File

Create a header file named `my_header.h` with the following content:

```
// Include guard to prevent multiple inclusions
#ifndef MY_HEADER_H
#define MY_HEADER_H

// Function declaration
void printMessage();

// Macro definition
#define PI 3.14159

// Structure definition
typedef struct {
    int x;
    int y;
} Point;

#endif // MY_HEADER_H
```

Step 2: Create a Source File

Create a source file named `my_header.c` that implements the function declared in the header file:

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

// Function definition
void printMessage() {
    printf("Hello from the header file!\n");
}
```

Step 3: Create a Main Program

Create a main program file named `main.c` that uses the header file:

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

int main() {
    // Use the function declared in the header file
    printMessage();

    // Use the macro defined in the header file
    printf("Value of PI: %f\n", PI);

    // Use the structure defined in the header file
    Point p = {10, 20};
    printf("Point coordinates: (%d, %d)\n", p.x, p.y);

    return 0;
}
```

Output ☑️

```
Hello from the header file!
Value of PI: 3.141590
Point coordinates: (10, 20)
```

Preprocessors in C are a powerful feature that allows you to perform text substitution and file inclusion before the actual compilation process. They are handled by the preprocessor, which is a part of the compilation process that runs before the compiler itself. Preprocessor directives begin with a # symbol.

1. **`#include`**

Used to include the contents of a file into the current file.

```
#include <stdio.h> // Includes the standard I/O library
#include "myheader.h" // Includes a user-defined header file
```

2. **`#define`**

Used to define macros, which are constants or expressions that can be used throughout the code.

```
#define PI 3.14159
#define MAX(a, b) ((a) > (b) ? (a) : (b))

int main() {
    printf("Value of PI: %f\n", PI);
    printf("Max of 3 and 5: %d\n", MAX(3, 5));
    return 0;
}
```

3. `#undef`

Used to undefine a previously defined macro.

```
#define PI 3.14159
#undef PI

int main() {
   // PI is no longer defined
   return 0;
}
```

4. `Conditional Compilation Directives`

These directives are used to include or exclude parts of the code based on certain conditions.

`#ifdef` and `#ifndef`

Checks if a macro is defined or not.

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

#ifdef DEBUG
   #define LOG(msg) printf("DEBUG: %s\n", msg)
#else
   #define LOG(msg)
#endif


int main() {
   LOG("This is a debug message");
   return 0;
}
```

Task 👍 Remove the line `#define DEBUG` from the code and see the output.

`#if`, `#elif`, `#else`, and `#endif`

Checks if a condition is true or false.

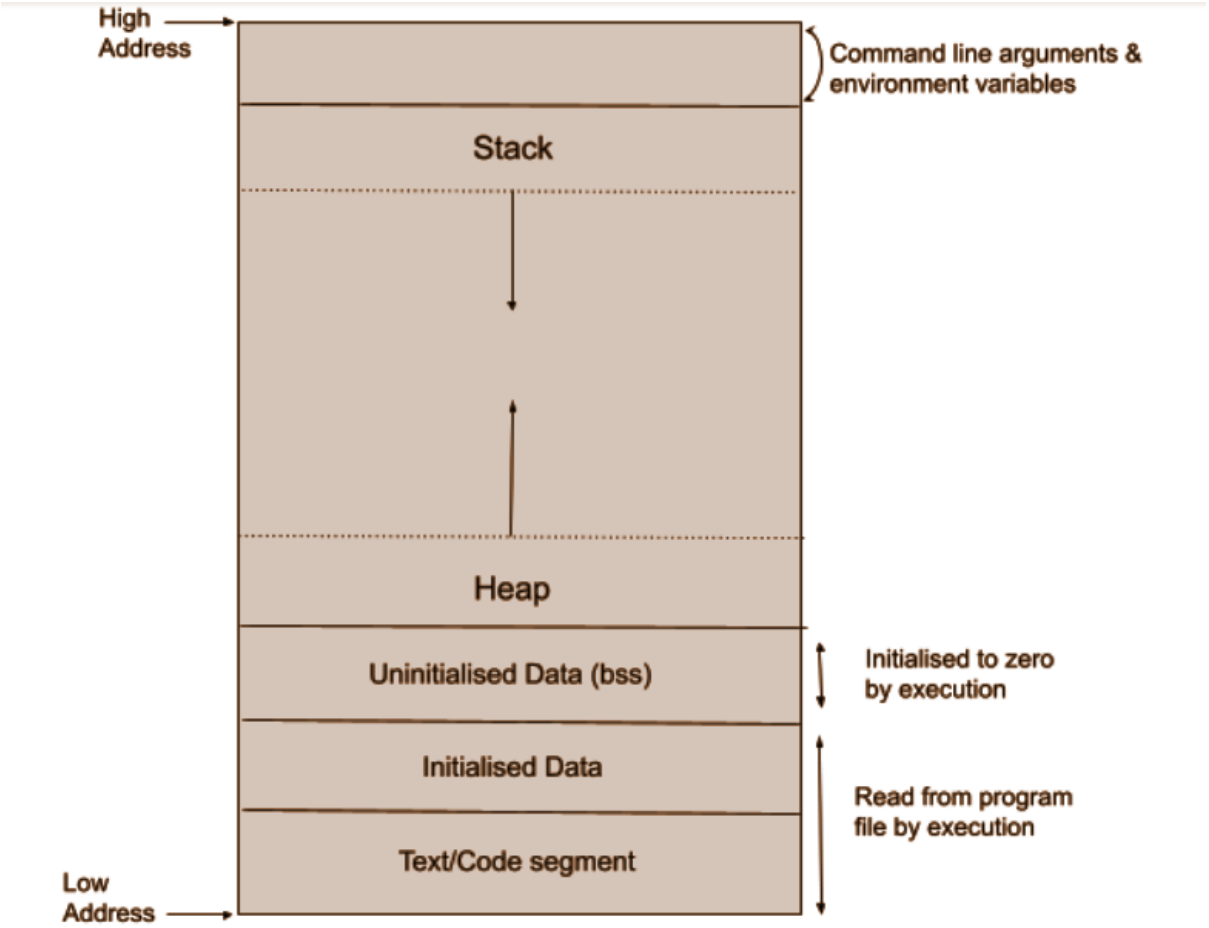
```
#include <stdio.h>
#define VERSION 2

#if VERSION == 1
   #define FEATURE "Version 1 Feature"
#elif VERSION == 2
   #define FEATURE "Version 2 Feature"
#else
   #define FEATURE "Unknown Version"
#endif

int main() {
   printf("Feature: %s\n", FEATURE);
   return 0;
}
```

Task 👍 Change the value of VERSION in line #define VERSION 2 for different values like 2 ,3 etc and see the output.

MEMORY LAYOUT

In C, the memory layout of a program is divided into several segments, each serving a different purpose. Understanding this layout is crucial for writing efficient and safe code, especially when dealing with low-level programming or embedded systems.

When we create a C program and run the program, its executable file is stored in the RAM of the computer in an organized manner.



1. Text/Code 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. If we store the instructions in the hard disk, then the speed for accessing the instructions from the hard disk becomes slower as hard disk works on the serial communication so taking the data from the hard disk will be slower, whereas the RAM is directly connected to the data and address bus so accessing the data from the RAM is faster.

2. Data section

The data which we use in our program will be stored in the data section. Since the variables declared inside the main() function are stored in the stack, but the variables declared outside the main() method will be stored in the data section. The variables declared in the data section could be stored in the form of initialized, uninitialized, and it could be local or global. Therefore, the data section is divided into four categories, i.e., initialized, uninitialized (BSS), global, or local.

Initialized Data Segment

- **Contains**: Global and static variables that are initialized by the programmer.
- **Properties**: Read-write segment.
- **Example**: `int initialized_global = 5;`

BSS (Block Started by Symbol) Segment

- **Contains**: Global and static variables that are not initialized by the programmer.
- **Properties**: Read-write segment.
- **Example**: `int uninitialized_global;`

3. Heap

Heap memory is used for dynamic memory allocation. Heap memory begins from the end of the uninitialized data segment and grows upwards to the higher addresses. The malloc() and calloc() functions are used to allocate the memory in the heap. The heap memory can be used by all the shared libraries and dynamically loaded modules. The free() function is used to deallocate the memory from the heap.

4. Stack

When we define a function and call that function then we use the stack frame. The variables which are declared inside the function are stored in the stack. The function arguments are also stored in the function as the arguments are also a part of the function. Such a type of memory allocation is known as static memory allocation because all the variables are defined in the function, and the size of the variables is also defined at the compile time. The stack section plays a very important role in the memory because whenever the function is called, a new stack frame is created.

Stack is also used for recursive functions. When the function is called itself again and again inside the same function which causes the stack overflow condition and it leads to the segmentation fault error in the program.

- **Contains**: Local variables, function parameters, and return addresses.
- **Properties**: Grows downwards (towards lower memory addresses). Managed automatically by the compiler.
- **Example**: Local variables inside functions.

Note: Stack overflow can occur if too much memory is allocated on the stack, leading to undefined behavior or program crashes.

Here's an example C program that demonstrates variables in different segments:

```
#include <stdio.h>
#include <stdlib.h>
// Global variables (initialized and uninitialized)
int initialized_global = 42;
int uninitialized_global;
// Static variables (initialized and uninitialized)

void function() {
   static int initialized_static = 99;
   static int uninitialized_static;
   printf("Initialized static: %d\n", initialized_static);
   printf("Uninitialized static: %d\n", uninitialized_static);
}
int main() {
   // Local variables (on the stack)
   int local_variable = 10;

   // Dynamic memory allocation (on the heap)
   int *heap_variable = (int *)malloc(sizeof(int) * 5);
   if (heap_variable == NULL) {
      printf("Memory allocation failed!\n");
      return 1;
   }
   for (int i = 0; i < 5; i++) {
      heap_variable[i] = i * 10;
   }
```

```
 // Print addresses to see memory segments
   printf("Address of initialized global: %p\n", (void *)&initialized_global);
   printf("Address of uninitialized global: %p\n", (void *)&uninitialized_global);
   printf("Address of local variable: %p\n", (void *)&local_variable);
   printf("Address of heap variable: %p\n", (void *)heap_variable);

   // Call function to see static variables
   function();

   // Free allocated memory
   free(heap_variable);
   return 0;
}
```

Output ☑️

Address of initialized global: 0x55564c55d010

Address of uninitialized global: 0x55564c55d01c

Address of local variable: 0x7ffc902243f8

Address of heap variable: 0x55564dc0f2a0

Initialized static: 99

Uninitialized static: 0

DYNAMIC MEMORY ALLOCATION

Dynamic memory is memory that is allocated after the program starts running. Allocation of dynamic memory can also be referred to as *runtime* memory allocation.

Unlike with static memory, you have full control over how much memory is being used at any time. You can write code to determine how much memory you need and allocate it.

Dynamic memory does not belong to a variable, it can only be accessed with pointers.

To allocate dynamic memory, you can use the `malloc()` or `calloc()` functions. It is necessary to include the `<stdlib.h>` header to use them. The `malloc()` and `calloc()` functions allocate some memory and return a pointer to its address.

The "malloc" or "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 of malloc() in C

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

For Example:

```
ptr = (int*) malloc(100 * sizeof(int));
```

Since the size of int is 4 bytes, this statement will allocate 400 bytes of memory. And, the pointer ptr holds the address of the first byte in the allocated memory.

Example

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

int main()
{
   // This pointer will hold the
   // base address of the block created
   int* ptr;
   int n, i;

   // Get the number of elements for the array
   printf("Enter number of elements:");
   scanf("%d",&n);
   printf("Entered number of elements: %d\n", n);

   // Dynamically allocate memory using malloc()
   ptr = (int*)malloc(n * sizeof(int));

   // Check if the memory has been successfully
   // allocated by malloc or not
   if (ptr == NULL) {
      printf("Memory not allocated.\n");
      exit(0);
   }
   else {

      // Memory has been successfully allocated
      printf("Memory successfully allocated using malloc.\n");

      // Get the elements of the array
      for (i = 0; i < n; ++i) {
         ptr[i] = i + 1;
      }
```

```
    // Print the elements of the array
    printf("The elements of the array are: ");
    for (i = 0; i < n; ++i) {
        printf("%d, ", ptr[i]);
    }
  }

  return 0;
}
```

Output ☑️

Enter number of elements:10

Entered number of elements: 10

Memory successfully allocated using malloc.

The elements of the array are: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,

C calloc() method

“calloc” or “contiguous allocation” method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. it is very much similar to malloc() but has two different points and these are:

It initializes each block with a default value ‘0’.

It has two parameters or arguments as compare to malloc().

Syntax of calloc() in C

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

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

For Example:

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

This statement allocates contiguous space in memory for 25 elements each with the size of the float.

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

int main()
{

   // This pointer will hold the
   // base address of the block created
   int* ptr;
   int n, i;
   // Get the number of elements for the array

   printf("Enter number of elements:\n");
   scanf("%d",&n);
   // Dynamically allocate memory using calloc()
   ptr = (int*)calloc(n, sizeof(int));
   // Check if the memory has been successfully
   // allocated by calloc or not
   if (ptr == NULL) {
      printf("Memory not allocated.\n");
      exit(0);
   }
   else {

      // Memory has been successfully allocated
      printf("Memory successfully allocated using calloc.\n");

      // Get the elements of the array
      for (i = 0; i < n; ++i) {
         ptr[i] = i + 1;
      }
```

```
// Print the elements of the array
      printf("The elements of the array are: ");
      for (i = 0; i < n; ++i) {
         printf("%d, ", ptr[i]);
      }
   }

   return 0;
}
```

Output ☑️

Enter number of elements:

12

Memory successfully allocated using calloc.

The elements of the array are: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,

C free() method

“free” method in C is used to dynamically de-allocate the memory. The memory allocated using functions malloc() and calloc() is not de-allocated on their own. Hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.

Syntax of free() in C

```
free(ptr);
```

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

int main()
{

   // This pointer will hold the
   // base address of the block created
   int *ptr, *ptr1;
   int n, i;

   // Get the number of elements for the array

   printf("Enter number of elements:\n" );
   scanf("%d",&n);
   // Dynamically allocate memory using malloc()
   ptr = (int*)malloc(n * sizeof(int));

   // Dynamically allocate memory using calloc()
   ptr1 = (int*)calloc(n, sizeof(int));

   // Check if the memory has been successfully
   // allocated by malloc or not
   if (ptr == NULL || ptr1 == NULL) {
      printf("Memory not allocated.\n");
      exit(0);
   }
   else {

      // Memory has been successfully allocated
      printf("Memory successfully allocated using malloc.\n");

      // Free the memory
      free(ptr);
      printf("Malloc Memory successfully freed.\n");
```

```
    // Memory has been successfully allocated
    printf("\nMemory successfully allocated using calloc.\n");

    // Free the memory
    free(ptr1);
    printf("Calloc Memory successfully freed.\n");
  }

  return 0;
}
```

Output ☑️

Enter number of elements:

2

Memory successfully allocated using malloc.

Malloc Memory successfully freed.

Memory successfully allocated using calloc.

Calloc Memory successfully freed.

C realloc() method

“realloc” or “re-allocation” method in C is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to dynamically re-allocate memory. re-allocation of memory maintains the already present value and new blocks will be initialized with the default garbage value.

Syntax of realloc() in C

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

where ptr is reallocated with new size 'newSize'.

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

int main()
{

    // This pointer will hold the
    // base address of the block created
    int* ptr;
    int n, i;

    // Get the number of elements for the array
    n = 5;
    printf("Enter number of elements: %d\n", n);

    // Dynamically allocate memory using calloc()
    ptr = (int*)calloc(n, sizeof(int));

    // Check if the memory has been successfully
    // allocated by malloc or not
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {

        // Memory has been successfully allocated
        printf("Memory successfully allocated using calloc.\n");

        // Get the elements of the array
        for (i = 0; i < n; ++i) {
            ptr[i] = i + 1;
        }

        // Print the elements of the array
        printf("The elements of the array are: ");
```

```
    for (i = 0; i < n; ++i) {
        printf("%d, ", ptr[i]);
    }

    // Get the new size for the array
    n = 10;
    printf("\n\nEnter the new size of the array: %d\n", n);

    // Dynamically re-allocate memory using realloc()
    ptr = (int*)realloc(ptr, n * sizeof(int));

     if (ptr == NULL) {
     printf("Reallocation Failed\n");
     exit(0);
    }

    // Memory has been successfully allocated
    printf("Memory successfully re-allocated using realloc.\n");

    // Get the new elements of the array
    for (i = 5; i < n; ++i) {
        ptr[i] = i + 1;
    }

    // Print the elements of the array
    printf("The elements of the array are: ");
    for (i = 0; i < n; ++i) {
        printf("%d, ", ptr[i]);
    }

    free(ptr);
  }

  return 0;
}
```

Output ☑️

Enter number of elements:

5

Memory successfully allocated using calloc.

The elements of the array are: 1, 2, 3, 4, 5,

Enter the new size of the array:

3

Memory successfully re-allocated using realloc.

The elements of the array are: 1, 2, 3,

.