C Language

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

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

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

Initially, C language was developed to be used in **UNIX operating system**. It inherits many features of previous languages such as B and BCPL.

The C Language is developed by Dennis Ritchie for creating system applications that directly interact with the hardware devices such as drivers, kernels, etc.

C programming is considered as the base for other programming languages, that is why it is known as mother language.

It can be defined by the following ways:

1. Mother language
2. System programming language
3. Procedure-oriented programming language
4. Structured programming language
5. Mid-level programming language

1) C as a mother language

C language is considered as the mother language of all the modern programming languages because **most of the compilers, JVMs, Kernels, etc. are written in C language**, and most of the programming languages follow C syntax, for example, C++, Java, C#, etc.

2) C as a system programming language

A system programming language is used to create system software. C language is a system programming language because it **can be used to do low-level programming (for example driver and kernel)**. It is generally used to create hardware devices, OS, drivers, kernels, etc. For example, Linux kernel is written in C.

It can't be used for internet programming like Java, .Net, PHP, etc.

3) C as a procedural language

A procedure is known as a function, method, routine, subroutine, etc. A procedural language **specifies a series of steps for the program to solve the problem**.

A procedural language breaks the program into functions, data structures, etc.

C is a procedural language. In C, variables and function prototypes must be declared before being used.

4) as a structured programming language

A structured programming language is a subset of the procedural language. **Structure means to break a program into parts or blocks** so that it may be easy to understand.

In the C language, we break the program into parts using functions. It makes the program easier to understand and modify.

5) C as a mid-level programming language

C is considered as a middle-level language because it **supports the feature of both low-level and high-level languages**. C language program is converted into assembly code, it supports pointer arithmetic (low-level), but it is machine independent (a feature of high-level).

A **Low-level language** is specific to one machine, i.e., machine dependent. It is machine dependent, fast to run. But it is not easy to understand.

A **High-Level language** is not specific to one machine, i.e., machine independent. It is easy to understand.

History of C Language

Let's see the programming languages that were developed before C language.

Language	Year	Developed By
Algol	1960	International Group
BCPL	1967	Martin Richard

B	1970	Ken Thompson
Traditional C	1972	Dennis Ritchie
K & R C	1978	Kernighan & Dennis Ritchie
ANSI C	1989	ANSI Committee
ANSI/ISO C	1990	ISO Committee
C99	1999	Standardization Committee

Features of C Language

C is the widely used language. It provides many **features** that are given below.

1. Simple
2. Machine Independent or Portable
3. Mid-level programming language
4. structured programming language
5. Rich Library
6. Memory Management
7. Fast Speed
8. Pointers
9. Recursion
10. Extensible

1) Simple

C is a simple language in the sense that it provides a **structured approach** (to break the problem into parts), **the rich set of library functions**, **data types**, etc.

2) Machine Independent or Portable

Unlike assembly language, c programs **can be executed on different machines** with some machine specific changes. Therefore, C is a machine independent language.

3) Mid-level programming language

Although, C is **intended to do low-level programming**. It is used to develop system applications such as kernel, driver, etc. It **also supports the features of a high-level language**. That is why it is known as mid-level language.

4) Structured programming language

C is a structured programming language in the sense that **we can break the program into parts using functions**. So, it is easy to understand and modify. Functions also provide code reusability.

5) Rich Library

C **provides a lot of inbuilt functions** that make the development fast.

6) Memory Management

It supports the feature of **dynamic memory allocation**. In C language, we can free the allocated memory at any time by calling the **free()** function.

7) Speed

The compilation and execution time of C language is fast since there are lesser inbuilt functions and hence the lesser overhead.

8) Pointer

C provides the feature of pointers. We can directly interact with the memory by using the pointers. We **can use pointers for memory, structures, functions, array**, etc.

9) Recursion

In C, we **can call the function within the function**. It provides code reusability for every function. Recursion enables us to use the approach of backtracking.

10) Extensible

C language is extensible because it **can easily adopt new features**.

First C Program:

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

- printf("Hello C Language");
- **return** 0;
- }

#include <stdio.h> includes the **standard input output** library functions. The printf() function is defined in stdio.h .

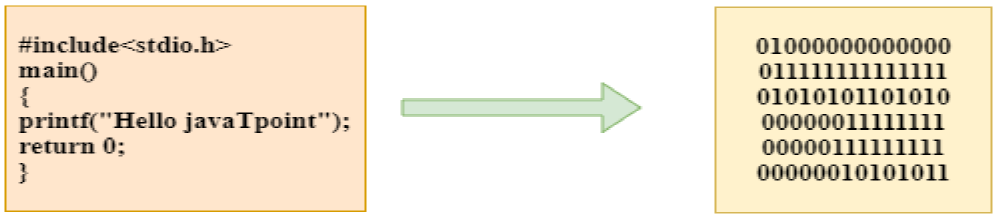
int main() The **main() function is the entry point of every program** in c language.

printf() The printf() function is **used to print data** on the console.

return 0 The return 0 statement, returns execution status to the OS. The 0 value is used for successful execution and 1 for unsuccessful execution.

What is a compilation?

The compilation is a process of converting the source code into object code. It is done with the help of the compiler. The compiler checks the source code for the syntactical or structural errors, and if the source code is error-free, then it generates the object code.



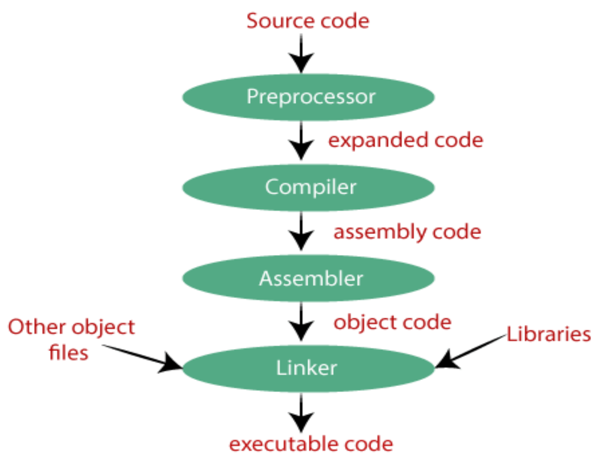
The c compilation process converts the source code taken as input into the object code or machine code. The compilation process can be divided into four steps, i.e., Pre-processing, Compiling, Assembling, and Linking.

The preprocessor takes the source code as an input, and it removes all the comments from the source code. The preprocessor takes the preprocessor directive and interprets it. For example, if **<stdio.h>,** the directive is available in the program, then the preprocessor interprets the directive and replace this directive with the content of the **'stdio.h'** file.

The following are the phases through which our program passes before being transformed into an executable form:

- **Preprocessor**
- **Compiler**
- **Assembler**

- **Linker**



Preprocessor

The source code is the code which is written in a text editor and the source code file is given an extension ".c". This source code is first passed to the preprocessor, and then the preprocessor expands this code. After expanding the code, the expanded code is passed to the compiler.

Compiler

The code which is expanded by the preprocessor is passed to the compiler. The compiler converts this code into assembly code. Or we can say that the C compiler converts the pre-processed code into assembly code.

Assembler

The assembly code is converted into object code by using an assembler. The name of the object file generated by the assembler is the same as the source file. The extension of the object file in DOS is '.obj,' and in UNIX, the extension is 'o'. If the name of the source file is **'hello.c',** then the name of the object file would be 'hello.obj'.

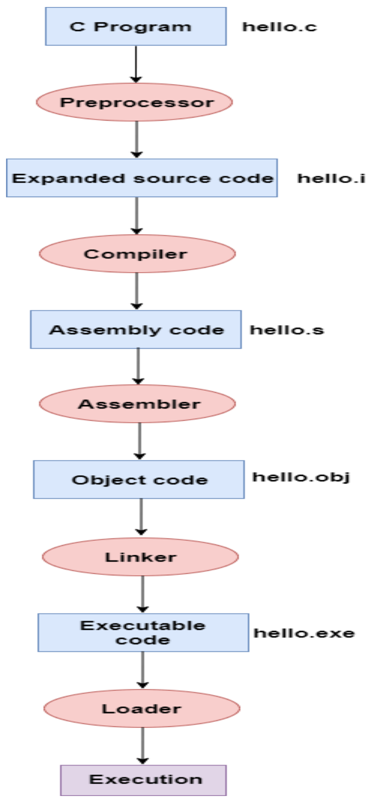
Linker

Mainly, all the programs written in C use library functions. These library functions are pre-compiled, and the object code of these library files is stored with '.lib' (or '.a') extension. The main working of the linker is to combine the object code of library files with the object code of our program. Sometimes the situation arises when our program refers to the functions defined in other files; then linker plays a very important role in this. It links the object code of these files to our program. Therefore, we conclude that the job of the linker is to link the object code of our program with the object code of the library files and other files. The output of the linker is the executable file. The name of the executable file is the same as the source file but differs only in their extensions. In DOS, the extension of the executable file is '.exe', and in UNIX, the executable file can be named as 'a.out'. For example, if we are using printf() function in a program, then the linker adds its associated code in an output file.

hello.c

- #include <stdio.h>
- **int** main()
- {
-     printf("Hello SAURABH");
-     **return** 0;
- }

Now, we will create a flow diagram of the above program:



In the above flow diagram, the following steps are taken to execute a program:

- Firstly, the input file, i.e., **hello.c,** is passed to the preprocessor, and the preprocessor converts the source code into expanded source code. The extension of the expanded source code would be **hello.i.**
- The expanded source code is passed to the compiler, and the compiler converts this expanded source code into assembly code. The extension of the assembly code would be **hello.s.**
- This assembly code is then sent to the assembler, which converts the assembly code into object code.
- After the creation of an object code, the linker creates the executable file. The loader will then load the executable file for the execution.

printf() and scanf() in C

The printf() and scanf() functions are used for input and output in C language. Both functions are inbuilt library functions, defined in stdio.h (header file).

printf() function

The **printf() function** is used for output. It prints the given statement to the console.

The syntax of printf() function is given below:
printf("format string",argument_list);

The **format string** can be %d (integer), %c (character), %s (string), %f (float) etc.

scanf() function

The **scanf() function** is used for input. It reads the input data from the console.
scanf("format string",argument_list);

Program to print cube of given number

```
#include<stdio.h>
int main(){
int number;
printf("enter a number:");
scanf("%d",&number);
printf("cube of number is:%d ",number*number*number);
return 0;
}
```

Variables in C

A **variable** is a name of the memory location. It is used to store data. Its value can be changed, and it can be reused many times.
It is a way to represent memory location through symbol so that it can be easily identified.
the syntax to declare a variable:
type variable_list;

Rules for defining variables

- A variable can have alphabets, digits, and underscore.
- A variable name can start with the alphabet, and underscore only. It can't start with a digit.
- No whitespace is allowed within the variable name.
- A variable name must not be any reserved word or keyword, e.g. int, float, etc.

	Valid variable names	**Invalid variable names**
1	**int** a;	**int** 2;
2	**int** _ab;	**int** a b;
3	**int** a30;	**int long**;

Types of Variables in C

There are many types of variables in c:

1. local variable
2. global variable
3. static variable
4. automatic variable
5. external variable

Local Variable

A variable that is declared inside the function or block is called a local variable. It must be declared at the start of the block.

- **void** function1(){
- **int** x=10;//local variable
- }

You must have to initialize the local variable before it is used.

Global Variable

A variable that is declared outside the function or block is called a global variable. Any function can change the value of the global variable. It is available to all the functions. It must be declared at the start of the block.

- **int** value=20;//global variable
- **void** function1(){
- **int** x=10;//local variable
- }

Static Variable

A variable that is declared with the static keyword is called static variable. It retains its value between multiple function calls.

- **void** function1(){
- **int** x=10;   //local variable
- **static int** y=10; //static variable
- x=x+1;
- y=y+1;
- printf("%d, %d", x, y);
- }

If you call this function many times, the **local variable will print the same value** for each function call, e.g., 11, 11, 11 and so on. But the **static variable will print the incremented value** in each function call, e.g. 11, 12, 13 and so on.

Automatic Variable

All variables in C that are declared inside the block, are automatic variables by default. We can explicitly declare an automatic variable using **auto keyword**.

- **void** main(){
- **int** x=10;//local variable (also automatic)
- auto **int** y=20;//automatic variable
- }

External Variable

We can share a variable in multiple C source files by using an external variable. To declare an external variable, you need to use **extern keyword**.

myfile.h

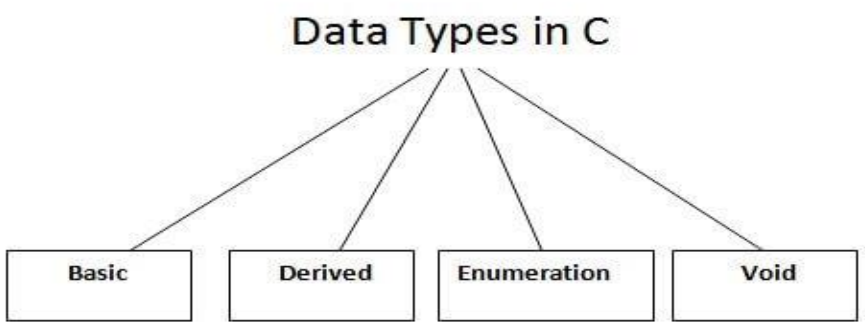
extern int x=10;//external variable (also global)

program1.c

- #include "myfile.h"
- #include <stdio.h>
- **void** printValue(){
-     printf("Global variable: %d", global_variable);
- }

Data Types in C

A data type specifies the type of data that a variable can store such as integer, floating, character, etc.



There are the following data types in C language.

Types	**Data Types**
Basic Data Type	int, char, float, double
Derived Data Type	array, pointer, structure, union
Enumeration Data Type	Enum
Void Data Type	Void

Basic Data Types

The basic data types are integer-based and floating-point based. C language supports both signed and unsigned literals.

The memory size of the basic data types may change according to 32 or 64-bit operating system. Its size is given **according to 32-bit architecture**.

Data Types	**Memory Size**	**Range**
Char	1 byte	−128 to 127
signed char	1 byte	−128 to 127
unsigned char	1 byte	0 to 255
Short	2 byte	−32,768 to 32,767
signed short	2 byte	−32,768 to 32,767
unsigned short	2 byte	0 to 65,535
Int	2 byte	−32,768 to 32,767
signed int	2 byte	−32,768 to 32,767
unsigned int	2 byte	0 to 65,535
short int	2 byte	−32,768 to 32,767
signed short int	2 byte	−32,768 to 32,767
unsigned short int	2 byte	0 to 65,535
long int	4 byte	-2,147,483,648 to 2,147,483,647
signed long int	4 byte	-2,147,483,648 to 2,147,483,647
unsigned long int	4 byte	0 to 4,294,967,295
Float	4 byte	
Double	8 byte	
long double	10 byte	

Comments in C

Comments in C language are used to provide information about lines of code. It is widely used for documenting code. There are 2 types of comments in the C language.

1. Single Line Comments
2. Multi-Line Comments

Single Line Comments

Single line comments are represented by double slash \\. Let's see an example of a single line comment in C.

- #include<stdio.h>
- **int** main(){
-     //printing information
-     printf("Hello C");
- **return** 0;
- }

Multi-Line Comments

Multi-Line comments are represented by slash asterisk * ... *\. It can occupy many lines of code, but it can't be nested. Syntax:

- /*
- code
- to be commented
- */

An example of a multi-Line comment in C.

- #include<stdio.h>
- **int** main(){
-     /*printing information
-       Multi-Line Comment*/
-     printf("Hello C");
- **return** 0;
- }

Format Specifier

The Format specifier is a string used in the formatted input and output functions. The format string determines the format of the input and output. The format string always starts with a '%' character. The commonly used format specifiers in printf() function are:

Format specifier	**Description**
%d or %i	It is used to print the signed integer value where signed integer means that the variable can hold both positive and negative values.
%u	It is used to print the unsigned integer value where the unsigned integer means that the variable can hold only positive value.
%o	It is used to print the octal unsigned integer where octal integer value always starts with a 0 value.
%x	It is used to print the hexadecimal unsigned integer where the hexadecimal integer value always starts with a 0x value. In this, alphabetical characters are printed in small letters such as a, b, c, etc.
%X	It is used to print the hexadecimal unsigned integer, but %X prints the alphabetical characters in uppercase such as A, B, C, etc.
%f	It is used for printing the decimal floating-point values. By default, it prints the 6 values after '.'.
%e/%E	It is used for scientific notation. It is also known as Mantissa or Exponent.
%g	It is used to print the decimal floating-point values, and it uses the fixed precision, i.e., the value after the decimal in input would be exactly the same as the value in the output.
%p	It is used to print the address in a hexadecimal form.
%c	It is used to print the unsigned character.
%s	It is used to print the strings.
%ld	It is used to print the long-signed integer value.

Let's understand the format specifiers in detail through an example.

- **%d**

```
int main()
{
  int b=6;
  int c=8;
  printf("Value of b is:%d", b);
  printf("\nValue of c is:%d",c);

    return 0;
}
```

In the above code, we are printing the integer value of b and c by using the %d specifier.

- **%u**

```
int main()
{
  int b=10;
  int c= -10;
  printf("Value of b is:%u", b);
  printf("\nValue of c is:%u",c);
    return 0;
}
```

In the above program, we are displaying the value of b and c by using an unsigned format specifier, i.e., %u. The value of b is positive, so %u specifier prints the exact value of b, but it does not print the value of c as c contains the negative value.

- **%o**

```
int main()
{
  int a=0100;
  printf("Octal value of a is: %o", a);
  printf("\nInteger value of a is: %d",a);
  return 0;
}
```

In the above code, we are displaying the octal value and integer value of a.

- **%x and %X**

- **int** main()
- {
- **int** y=0xA;
- printf("Hexadecimal value of y is: %x", y);
- printf("\nHexadecimal value of y is: %X",y);
- printf("\nInteger value of y is: %d",y);
- **return** 0;
- }

In the above code, y contains the hexadecimal value 'A'. We display the hexadecimal value of y in two formats. We use %x and %X to print the hexadecimal value where %x displays the value in small letters, i.e., 'a' and %X displays the value in a capital letter, i.e., 'A'.

- **%f**

- **int** main()
- {
- **float** y=3.4;
- printf("Floating point value of y is: %f", y);
- **return** 0;
- }

The above code prints the floating value of y.

- **%e**

- **int** main()
- {
- **float** y=3;
- printf("Exponential value of y is: %e", y);
- **return** 0;
- }

- **%E**

- **int** main()
- {
- **float** y=3;
- printf("Exponential value of y is: %E", y);
- **return** 0;
- }

- **%g**

```
int main()
{
  float y=3.8;
  printf("Float value of y is: %g", y);
  return 0;
}
```

In the above code, we are displaying the floating value of y by using %g specifier. The %g specifier displays the output same as the input with a same precision.

- **%p**

```
int main()
{
  int y=5;
  printf("Address value of y in hexadecimal form is: %p", &y);
  return 0;
}
```

- **%c**

```
int main()
{
  char a='c';
  printf("Value of a is: %c", a);
  return 0;
}
```

- **%s**

```
int main()
{
  printf("%s", "javaTpoint");
  return 0;
}
```

Minimum Field Width Specifier

Suppose we want to display an output that occupies a minimum number of spaces on the screen. You can achieve this by displaying an integer number after the percent sign of the format specifier.

```
int main()
```

- {
- **int** x=900;
-   printf("%8d", x);
-   printf("\n%-8d",x);
-   **return** 0;
- }

In the above program, %8d specifier displays the value after 8 spaces while %-8d specifier will make a value left-aligned.

Now we will see how to fill the empty spaces. It is shown in the below code:

- **int** main()
- {
- **int** x=12;
-   printf("%08d", x);
-   **return** 0;
- }

In the above program, %08d means that the empty space is filled with zeroes.

Specifying Precision

We can specify the precision by using '.' (Dot) operator which is followed by integer and format specifier.

- **int** main()
- {
- **float** x=12.2;
-   printf("%.2f", x);
-   **return** 0;
- }

Escape Sequence in C

An escape sequence in C language is a sequence of characters that doesn't represent itself when used inside string literal or character. It is composed of two or more characters starting with backslash \. For example: \n represents new line.

List of Escape Sequences in C

Escape Sequence	Meaning
\a	Alarm or Beep
\b	Backspace
\f	Form Feed
\n	New Line
\r	Carriage Return
\t	Tab (Horizontal)
\v	Vertical Tab
\\	Backslash
\'	Single Quote
\"	Double Quote
\?	Question Mark
\nnn	octal number
\xhh	hexadecimal number
\0	Null

Escape Sequence Example

```
#include<stdio.h>
int main(){
    int number=50;
   printf("You\nare\nlearning\n\'c\' language\n\"Do you know C language\"");
return 0;
}
```

ASCII value in C

The full form of ASCII is the **American Standard Code for information interchange**. It is a character encoding scheme used for electronics communication. Each character or a special character is represented by some ASCII code, and each ascii code occupies 7 bits in memory.

In C programming language, a character variable does not contain a character value itself rather the ascii value of the character variable. The ascii value represents the character variable in numbers, and each character variable is assigned with some number range from 0 to 127. For example, the ascii value of 'A' is 65.

In the above example, we assign 'A' to the character variable whose ascii value is 65, so 65 will be stored in the character variable rather than 'A'.

We will create a program which will display the ascii value of the character variable.

```
#include <stdio.h>
int main()
{
    char ch;    // variable declaration
    printf("Enter a character");
    scanf("%c",&ch);  // user input
    printf("\n The ascii value of the ch variable is : %d", ch);
    return 0;
}
```

In the above code, the first user will give the character input, and the input will get stored in the 'ch' variable. If we print the value of the 'ch' variable by using %c format specifier, then it will display 'A' because we have given the character input as 'A', and if we use the %d format specifier then its ascii value will be displayed, i.e., 65. Now, we will create a program which will display the ascii value of all the characters.

```
#include <stdio.h>
int main()
{
 int k;   // variable declaration
 for(int k=0;k<=255;k++)  // for loop from 0-255
 {
    printf("\nThe ascii value of %c is %d", k,k);
 }
return 0;
}
```

The above program will display the ascii value of all the characters. As we know that ascii value of all the characters starts from 0 and ends at 255, so we iterate the for loop from 0 to 255.

Now we will create the program which will sum the ascii value of a string.

```
#include <stdio.h>
int main()
{
   int sum=0;  // variable initialization
   char name[20];  // variable initialization
   int i=0;  // variable initialization
   printf("Enter a name: ");
   scanf("%s", name);
   while(name[i]!='\0')  // while loop
   {
      printf("\nThe ascii value of the character %c is %d", name[i],name[i]);
      sum=sum+name[i];
      i++;
   }
   printf("\nSum of the ascii value of a string is : %d", sum);
   return 0;
}
```

In the above code, we are taking user input as a string. After taking user input, we execute the **while** loop which adds the ascii value of all the characters of a string and stores it in a '**sum**' variable.

Classification of tokens in C

Tokens in C language can be divided into the following categories:

- Keywords in C
- Identifiers in C
- Strings in C
- Operators in C
- Constant in C
- Special Characters in C

Keywords in C

Keywords in C can be defined as the **pre-defined** or the **reserved words** having its own importance, and each keyword has its own functionality. Since keywords are the pre-defined words used by the compiler, so they cannot be used as the variable names. If the keywords are used as the variable names, it means that we are assigning a different meaning to the keyword, which is not allowed. A keyword is a **reserved word**. You cannot use it as a variable name, constant name, etc. There are only 32 reserved words (keywords) in the C language.

A list of 32 keywords in the c language is given below:

auto	Break	Case	Char	const	continue	default	do
double	Else	Enum	Extern	float	for	goto	if
int	Long	Register	Return	short	signed	sizeof	static
struct	Switch	Typedef	Union	unsigned	void	volatile	while

Identifiers in C

Identifiers in C are used for naming variables, functions, arrays, structures, etc. Identifiers in C are the user-defined words. It can be composed of uppercase letters, lowercase letters, underscore, or digits, but the starting letter should be either an underscore or an alphabet. Identifiers cannot be used as keywords. Rules for constructing identifiers in C are given below:

- The first character of an identifier should be either an alphabet or an underscore, and then it can be followed by any of the character, digit, or underscore.
- It should not begin with any numerical digit.
- In identifiers, both uppercase and lowercase letters are distinct. Therefore, we can say that identifiers are case sensitive.
- Commas or blank spaces cannot be specified within an identifier.
- Keywords cannot be represented as an identifier.
- The length of the identifiers should not be more than 31 characters.
- Identifiers should be written in such a way that it is meaningful, short, and easy to read.

Rules for constructing C identifiers

- The first character of an identifier should be either an alphabet or an underscore, and then it can be followed by any of the character, digit, or underscore.
- It should not begin with any numerical digit.
- In identifiers, both uppercase and lowercase letters are distinct. Therefore, we can say that identifiers are case sensitive.
- Commas or blank spaces cannot be specified within an identifier.
- Keywords cannot be represented as an identifier.
- The length of the identifiers should not be more than 31 characters.
- Identifiers should be written in such a way that it is meaningful, short, and easy to read.

Example of valid identifiers

total, sum, average, _m _, sum_1, etc.

Example of invalid identifiers

- 2sum (starts with a numerical digit)
- **int** (reserved word)
- **char** (reserved word)
- m+n (special character, i.e., '+')

Types of identifiers

- Internal identifier
- External identifier

Internal Identifier

If the identifier is not used in the external linkage, then it is known as an internal identifier. The internal identifiers can be local variables.

External Identifier

If the identifier is used in the external linkage, then it is known as an external identifier. The external identifiers can be function names, global variables.

Differences between Keyword and Identifier

Keyword	Identifier
Keyword is a pre-defined word.	The identifier is a user-defined word
It must be written in a lowercase letter.	It can be written in both lowercase and uppercase letters.
Its meaning is pre-defined in the c compiler.	Its meaning is not defined in the c compiler.
It is a combination of alphabetical characters.	It is a combination of alphanumeric characters.
It does not contain the underscore character.	It can contain the underscore character.

```
int main()
{
   int a=10;
   int A=20;
   printf("Value of a is : %d",a);
   printf("\nValue of A is :%d",A);
   return 0;
}
```

The above output shows that the values of both the variables, 'a' and 'A' are different. Therefore, we conclude that the identifiers are case sensitive.

Strings in C

Strings in C are always represented as an array of characters having null character '\0' at the end of the string. This null character denotes the end of the string. Strings in C are enclosed within double quotes, while characters are enclosed within single characters. The size of a string is a number of characters that the string contains.

Now, we describe the strings in different ways:

char a[10] = "javatpoint"; // The compiler allocates the 10 bytes to the 'a' array.

char a[] = "javatpoint"; // The compiler allocates the memory at the run time.

char a[10] = {'j','a','v','a','t','p','o','i','n','t','\0'}; // String is represented in the form of characters.

Constants in C

A constant is a value assigned to the variable which will remain the same throughout the program, i.e., the constant value cannot be changed, for example: 10, 20, 'a', 3.4, "c programming" etc.

Types of constants in C

Constant	Example
Integer constant	10, 11, 34, etc.
Floating-point constant	45.6, 67.8, 11.2, etc.
Octal constant	011, 088, 022, etc.
Hexadecimal constant	0x1a, 0x4b, 0x6b, etc.
Character constant	'a', 'b', 'c', etc.
String constant	"java", "c++", ".net", etc.

Two ways to define constant in C

There are two ways to define constant in C programming.

1. const keyword
2. #define preprocessor

(1) C const keyword

The const keyword is used to define constant in C programming.

const float PI=3.14;

Now, the value of PI variable can't be changed.

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

- **const float** PI=3.14;
- printf("The value of PI is: %f",PI);
- **return** 0;
- }

Output: The value of PI is: 3.140000

If you try to change the the value of PI, it will render compile time error.

- #include<stdio.h>
- **int** main(){
- **const float** PI=3.14;
- PI=4.5;
- printf("The value of PI is: %f",PI);
- **return** 0;
- }

Output: Compile Time Error: Cannot modify a const object

(2) C #define preprocessor

The #define preprocessor is also used to define constant. We will learn about #define preprocessor dire

10.100000

Special characters in C

Some special characters are used in C, and they have a special meaning which cannot be used for another purpose.

- **Square brackets [ ]:** The opening and closing brackets represent the single and multidimensional subscripts.
- **Simple brackets ( ):** It is used in function declaration and function calling. For example, printf() is a pre-defined function.
- **Curly braces { }:** It is used in the opening and closing of the code. It is used in the opening and closing of the loops.
- **Comma (,):** It is used for separating for more than one statement and for example, separating function parameters in a function call, separating the variable when printing the value of more than one variable using a single printf statement.

- **Hash/pre-processor (#):** It is used for pre-processor directive. It basically denotes that we are using the header file.
- **Asterisk (*):** This symbol is used to represent pointers and also used as an operator for multiplication.
- **Tilde (~):** It is used as a destructor to free memory.
- **Period (.):** It is used to access a member of a structure or a union.

C Boolean

In C, Boolean is a data type that contains two types of values, i.e., 0 and 1. Basically, the bool type value represents two types of behavior, either true or false. Here, '0' represents false value, while '1' represents true value.

In C Boolean, '0' is stored as 0, and another integer is stored as 1. We do not require to use any header file to use the Boolean data type in C++, but in C, we have to use the header file, i.e., stdbool.h. If we do not use the header file, then the program will not compile.

Syntax

bool variable_name;

In the above syntax, **bool** is the data type of the variable, and **variable_name** is the name of the variable.

Example 1

```
#include <stdio.h>
#include<stdbool.h>
int main()
{
bool x=false; // variable initialization.
if(x==true) // conditional statements
{
printf("The value of x is true");
}
else
printf("The value of x is FALSE");
return 0;
}
```

In the above code, we have used **<stdbool.h>** header file so that we can use the bool type variable in our program. After the declaration of the header file, we create the bool type variable '**x**' and assigns a '**false**' value to it. Then, we add the conditional statements, i.e., **if..else**, to determine whether the value of 'x' is true or not.

Output The value of x is FALSE

Boolean Array

Now, we create a bool type array. The Boolean array can contain either true or false value, and the values of the array can be accessed with the help of indexing.

- #include <stdio.h>
- #include<stdbool.h>
- **int** main()
- {
- **bool** b[2]={**true**,**false**}; // Boolean type array
- **for**(**int** i=0;i<2;i++) // for loop
- {
- printf("%d,",b[i]); // printf statement
- }
- **return** 0;
- }

In the above code, we have declared a Boolean type array containing two values, i.e., true and false.

Output 1,0,

Typedef

There is another way of using Boolean value, i.e., **typedef**. Basically, typedef is a keyword in C language, which is used to assign the name to the already existing datatype.

```
#include <stdio.h>
typedef enum{false,true} b;
int main()
{
b x=false; // variable initialization
if(x==true) // conditional statements
{
printf("The value of x is true");
}
else
{
printf("The value of x is false");
}
return 0;
}
```

In the above code, we use the Boolean values, i.e., true and false, but we have not used the bool type. We use the Boolean values by creating a new name of the 'bool' type. In order to achieve this, **the typedef** keyword is used in the program.

typedef enum{**false**,**true**} b;

The above statement creates a new name for the '**bool**' type, i.e., 'b' as 'b' can contain either true or false value. We use the 'b' type in our program and create the 'x' variable of type 'b'.

Output The value of x is false

Operators in C

Operators in C is a special symbol used to perform the functions. The data items on which the operators are applied are known as operands. Operators are applied between the operands. Depending on the number of operands, operators are classified as follows:

Unary Operator

A unary operator is an operator applied to the single operand. For example: increment operator (++), decrement operator (--), sizeof, (type)*.

Binary Operator

The binary operator is an operator applied between two operands. The following is the list of the binary operators:

- Arithmetic Operators
- Relational Operators
- Shift Operators
- Logical Operators
- Bitwise Operators
- Conditional Operators

- Assignment Operator
- Misc Operator

Arithmetic Operators

Arithmetic operators are used to perform common mathematical operations.

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

Boolean with Logical Operators

The Boolean type value is associated with logical operators. There are three types of logical operators in the C language:

&&(AND Operator): It is a logical operator that takes two operands. If the value of both the operands are true, then this operator returns true otherwise false

||(OR Operator): It is a logical operator that takes two operands. If the value of both the operands is false, then it returns false otherwise true.

!(NOT Operator): It is a NOT operator that takes one operand. If the value of the operand is false, then it returns true, and if the value of the operand is true, then it returns false.

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

- **bool** x=**false**;
- **bool** y=**true**;
- printf("The value of x&&y is %d", x&&y);
- printf("\nThe value of x||y is %d", x||y);
- printf("\nThe value of !x is %d", !x);
- }

Conditional Operator in C

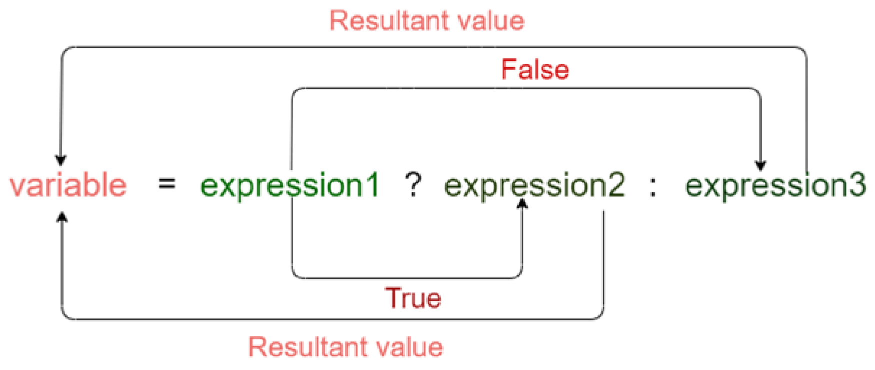
The conditional operator is also known as a **ternary operator**. The conditional statements are the decision-making statements which depends upon the output of the expression. It is represented by two symbols, i.e., '?' and ':'.

As conditional operator works on three operands, so it is also known as the ternary operator. The behavior of the conditional operator is similar to 'if-else' statement is also a decision-making statement.

Syntax of a conditional operator

Expression1? expression2: expression3;

The pictorial representation of the above syntax is shown below:



Meaning of the above syntax.

- In the above syntax, the expression1 is a Boolean condition that can be either true or false value.
- If the expression1 results into a true value, then the expression2 will execute.
- The expression2 is said to be true only when it returns a non-zero value.
- If the expression1 returns false value then the expression3 will execute.
- The expression3 is said to be false only when it returns zero value.

```
#include <stdio.h>
int main()
{
    int age;  // variable declaration
    printf("Enter your age");
    scanf("%d",&age);   // taking user input for age variable
    (age>=18)? (printf("eligible for voting")) : (printf("not eligible for voting"));  //
conditional operator
    return 0;
}
```

In the above code, we are taking input as the 'age' of the user. After taking input, we have applied the condition by using a conditional operator. In this condition, we are checking the age of the user. If the age of the user is greater than or equal to 18, then the statement1 will execute, i.e., (printf("eligible for voting")) otherwise, statement2 will execute, i.e., (printf("not eligible for voting")).

Example.2

```
#include <stdio.h>
int main()
{
   int a=5,b;  // variable declaration
   b=((a==5)?(3):(2)); // conditional operator
   printf("The value of 'b' variable is : %d",b);
    return 0;
}
```

In the above code, we have declared two variables, i.e., 'a' and 'b', and assign 5 value to the 'a' variable. After the declaration, we are assigning value to the 'b' variable by using the conditional operator. If the value of 'a' is equal to 5 then 'b' is assigned with a 3 value otherwise 2.

NOTE: As we know that the behavior of conditional operator and 'if-else' is similar but they have some differences. Let's look at their differences.

- A conditional operator is a single programming statement, while the 'if-else' statement is a programming block in which statements come under the parenthesis.
- A conditional operator can also be used for assigning a value to the variable, whereas the 'if-else' statement cannot be used for the assignment purpose.

- It is not useful for executing the statements when the statements are multiple, whereas the 'if-else' statement proves more suitable when executing multiple statements.
- The nested ternary operator is more complex and cannot be easily debugged, while the nested 'if-else' statement is easy to read and maintain.

Bitwise Operator in C

The bitwise operators are the operators used to perform the operations on the data at the bit-level. When we perform the bitwise operations, then it is also known as bit-level programming. It consists of two digits, either 0 or 1. It is mainly used in numerical computations to make the calculations faster.

We have different types of bitwise operators in the C programming language. The following is the list of the bitwise operators:

Operator	Meaning of operator
&	Bitwise AND operator
|	Bitwise OR operator
^	Bitwise exclusive OR operator
~	One's complement operator (unary operator)
<<	Left shift operator
>>	Right shift operator

Let's look at the truth table of the bitwise operators.

X	Y	X&Y	X|Y	X^Y
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Bitwise AND operator

Bitwise AND operator is denoted by the single ampersand sign (&). Two integer operands are written on both sides of the (&) operator. If the corresponding bits of both the operands are 1, then the output of the bitwise AND operation is 1; otherwise, the output would be 0.

For example,

- We have two variables a and b.
- a =6;
- b=4;
- The binary representation of the above two variables are given below:
- a = 0110
- b = 0100
- When we apply the bitwise AND operation in the above two variables, i.e., a&b, t he output would be:
- Result = 0100

As we can observe from the above result that bits of both the variables are compared one by one. If the bit of both the variables is 1 then the output would be 1, otherwise 0.

Let's understand the bitwise AND operator through the program.

```
#include <stdio.h>
int main()
{
   int a=6, b=4;  // variable declarations
   printf("The output of the Bitwise AND operator a&b is %d",a&b);
   return 0;
}
```

In the above code, we have created two variables, i.e., 'a' and 'b'. The values of 'a' and 'b' are 6 and 14 respectively. The binary value of 'a' and 'b' are 0110 and 1110, respectively. When we apply the AND operator between these two variables,

a AND b = 0110 && 1110 = 0110

Output: is 6

Bitwise OR operator

The bitwise OR operator is represented by a single vertical sign (|). Two integer operands are written on both sides of the (|) symbol. If the bit value of any of the operand is 1, then the output would be 1, otherwise 0.

For example,

- We consider two variables,
- a = 23;
- b = 10;
- The binary representation of the above two variables would be:
- a = 0001 0111
- b = 0000 1010
- When we apply the bitwise OR operator in the above two variables, i.e., a|b , then the output would be:
- Result = 0001 1111

As we can observe from the above result that the bits of both the operands are compared one by one; if the value of either bit is 1, then the output would be 1 otherwise 0.

```
#include <stdio.h>
int main()
{
   int a=23,b=10;  // variable declarations
   printf("The output of the Bitwise OR operator a|b is %d",a|b);
   return 0;
}
```

Output: is 31

Bitwise exclusive OR operator

Bitwise exclusive OR operator is denoted by (^) symbol. The result of bitwise XOR operator is **1** if the corresponding bits of two operands are opposite. For example,

```
12 = 00001100 (In Binary)
25 = 00011001 (In Binary)
Bitwise XOR Operation of 12 and 25
  00001100
^ 00011001
 ________
 00010101 = 21 (In decimal)
```

Let's understand the bitwise exclusive OR operator through a program.

```
#include <stdio.h>
int main()
{
  int a=12,b=10;  // variable declarations
  printf("The output of the Bitwise exclusive OR operator a^b is %d",a^b);
  return 0;
}
```

Output: is 6

Bitwise complement operator

The bitwise complement operator is also known as one's complement operator. It is represented by the symbol tilde (~). It takes only one operand or variable and performs complement operation on an operand. When we apply the complement operation on any bits, then 0 becomes 1 and 1 becomes 0.

For example,

- If we have a variable named 'a',
- a = 8;
- The binary representation of the above variable is given below:
- a = 1000
- When we apply the bitwise complement operator to the operand, then the output would be:
- Result = 0111
- As we can observe from the above result that if the bit is 1, then it gets changed to 0 else 1.

Let's understand the complement operator through a program.

```
#include <stdio.h>
int main()
{
  int a=8;  // variable declarations
  printf("The output of the Bitwise complement operator ~a is %d",~a);
  return 0;
}
```

Bitwise shift operators

Two types of bitwise shift operators exist in C programming. The bitwise shift operators will shift the bits either on the left-side or right-side. Therefore, we can say that the bitwise shift operator is divided into two categories:

- Left-shift operator
- Right-shift operator

Left-shift operator

It is an operator that shifts the number of bits to the left-side.
Syntax of the left-shift operator is given below:
Operand << n
Where,
Operand is an integer expression on which we apply the left-shift operation.
n is the number of bits to be shifted.

In the case of Left-shift operator, 'n' bits will be shifted on the left-side. The 'n' bits on the left side will be popped out, and 'n' bits on the right-side are filled with 0.

For example,

- Suppose we have a statement:
- **int** a = 5;
- The binary representation of 'a' is given below:
- a = 0101
- If we want to left-
  shift the above representation by 2, then the statement would be:
- a << 2;
- 0101<<2 = 00010100

Program

- #include <stdio.h>
- **int** main()
- {
-     **int** a=5; // variable initialization
-     printf("The value of a<<2 is : %d ", a<<2);
-     **return** 0;
- }

Output: is 20

Right-shift operator

It is an operator that shifts the number of bits to the right side.

Syntax of the right-shift operator is given below:

Operand >> n;

Where,

Operand is an integer expression on which we apply the right-shift operation.

N is the number of bits to be shifted.

In the case of the right-shift operator, 'n' bits will be shifted on the right-side. The 'n' bits on the right-side will be popped out, and 'n' bits on the left-side are filled with 0.

For example,

- Suppose we have a statement,
- **int** a = 7;
- The binary representation of the above variable would be:
- a = 0111
- If we want to right-
  shift the above representation by 2, then the statement would be:
- a>>2;
- 0000 0111 >> 2 = 0000 0001

Let's understand through a program.

- #include <stdio.h>
- **int** main()
- {
-     **int** a=7; // variable initialization
-     printf("The value of a>>2 is : %d ", a>>2);
-     **return** 0;
- }

Output: The value of a>>2 is: 1

Right shift operator shifts all bits towards right by certain number of specified bits. It is denoted by >>.

212 = 11010100 (In binary)

212 >> 2 = 00110101 (In binary) [Right shift by two bits]

212 >> 7 = 00000001 (In binary)

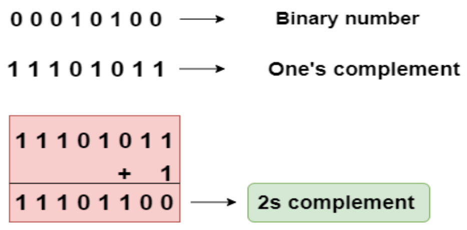
212 >> 8 = 00000000

212 >> 0 = 11010100 (No Shift)

2s complement

The 2s complement in C is generated from the 1s complement in C. As we know that the 1s complement of a binary number is created by transforming bit 1 to 0 and 0 to 1; the 2s complement of a binary number is generated by adding one to the 1s complement of a binary number.

In short, we can say that the 2s complement in C is defined as the sum of the one's complement in C and one.



In the above figure, the binary number is equal to 00010100, and its one's complement is calculated by transforming the bit 1 to 0 and 0 to 1 vice versa. Therefore, one's complement becomes 11101011. After calculating one's complement, we calculate the two's complement by adding 1 to the one's complement, and its result is 11101100.

Let's create a program of 2s complement.

- #include <stdio.h>
- **int** main()
- {
-   **int** n; // variable declaration
-   printf("Enter the number of bits do you want to enter :");
-   scanf("%d",&n);

```
  char binary[n+1];  // binary array declaration;
  char onescomplement[n+1]; // onescomplement array declaration
  char twoscomplement[n+1]; // twoscomplement array declaration
  int carry=1; // variable initialization
  printf("\nEnter the binary number : ");
  scanf("%s", binary);
  printf("%s", binary);
  printf("\nThe ones complement of the binary number is :");
  // Finding onescomplement in C
  for(int i=0;i<n;i++)
  {
     if(binary[i]=='0')
     onescomplement[i]='1';
     else if(binary[i]=='1')
     onescomplement[i]='0';
  }
  onescomplement[n]='\0';
  printf("%s",onescomplement);
printf("\nThe twos complement of a binary number is : ");
// Finding twoscomplement in C
for(int i=n-1; i>=0; i--)
   {
      if(onescomplement[i] == '1' && carry == 1)
      {
         twoscomplement[i] = '0';
      }
      else if(onescomplement[i] == '0' && carry == 1)
      {
         twoscomplement[i] = '1';
         carry = 0;
      }
      else
      {
         twoscomplement[i] = onescomplement[i];
      }
   }
twoscomplement[n]='\0';
printf("%s",twoscomplement);
return 0;
}
```

Analysis of the above program,

- First, we input the number of bits, and it gets stored in the '**n**' variable.
- After entering the number of bits, we declare character array, i.e., **char binary[n+1],** which holds the binary number. The '**n**' is the number of bits which we entered in the previous step; it basically defines the size of the array.
- We declare two more arrays, i.e., **onescomplement[n+1]**, and **twoscomplement[n+1].** The **onescomplement[n+1]** array holds the ones complement of a binary number while the **twoscomplement[n+1]** array holds the two's complement of a binary number.
- Initialize the **carry** variable and assign 1 value to this variable.
- After declarations, we input the binary number.
- Now, we simply calculate the one's complement of a binary number. To do this, we create a **loop** that iterates throughout the binary array, **for(int i=0;i<n;i++)**. In for loop, the condition is checked whether the bit is 1 or 0. If the bit is 1 then **onescomplement[i]=0** else **onescomplement[i]=1**. In this way, one's complement of a binary number is generated.
- After calculating one's complement, we generate the 2s complement of a binary number. To do this, we create a **loop** that iterates from the last element to the starting element. In for loop, we have three conditions:
    - If the bit of onescomplement[i] is 1 and the value of carry is 1 then we put 0 in twocomplement[i].
    - If the bit of onescomplement[i] is 0 and the value of carry is 1 then we put 1 in twoscomplement[i] and 0 in carry.
    - If the above two conditions are false, then onescomplement[i] is equal to twoscomplement[i].

If else Statement

The if-else statement in C is used to perform the operations based on some specific condition. The operations specified in if block are executed if and only if the given condition is true.

There are the following variants of if statement in C language.

- If statement
- If-else statement
- If else-if ladder
- Nested if

If Statement

The if statement is used to check some given condition and perform some operations depending upon the correctness of that condition. It is mostly used in the scenario where we need to perform the different operations for the different conditions. The syntax of the if statement is given below.

if(expression){
//code to be executed
}

Example:

- #include<stdio.h>
- #include<conio.h>
- **int** main()
- {
- **int** num;
- printf("Enter a number:");
- scanf("%d",&num);
- **if**(num%2==0)
- {
- printf(" given number is even number");
- }
- getch();
- **return** 0;
- }

1. **Program** to find the largest number of the three.

- #include <stdio.h>
- **int** main()
- {
-     **int** a, b, c;
-      printf("Enter three numbers?");
-     scanf("%d %d %d",&a,&b,&c);
-     **if**(a>b && a>c)
-     {
-         printf("%d is largest",a);
-     }
-     **if**(b>a && b > c)
-     {

-         printf("%d is largest",b);
-     }
-     **if**(c>a && c>b)
-     {
-         printf("%d is largest",c);
-     }
-     **if**(a == b && a == c)
-     {
-         printf("All are equal");
-     }
- }

If-else Statement

The if-else statement is used to perform two operations for a single condition. The if-else statement is an extension to the if statement using which, we can perform two different operations, i.e., one is for the correctness of that condition, and the other is for the incorrectness of the condition. Here, we must notice that if and else block cannot be executed simultaneously. Using if-else statement is always preferable since it always invokes an otherwise case with every if condition. The syntax of the if-else statement is given below.

if(expression){
//code to be executed if condition is true
}**else**{
//code to be executed if condition is false
}

Example:

- #include<stdio.h>
- **int** main(){
- **int** num;
- printf("enter a number:");
- scanf("%d",&num);
- **if**(num%2==0){
- printf("Number is even number");
- }
- **else**{
- printf("Number is odd number");
- }
- **return** 0;

```
}
```

Program to check whether a person is eligible to vote or not.

```
#include <stdio.h>
int main()
{
    int age;
    printf("Enter your age?");
    scanf("%d",&age);
    if(age>=18)
    {
        printf("You are eligible to vote...");
    }
    else
    {
        printf("Sorry ... you can't vote");
    }
}
```

If else-if ladder Statement

The if-else-if ladder statement is an extension to the if-else statement. It is used in the scenario where there are multiple cases to be performed for different conditions. In if-else-if ladder statement, if a condition is true then the statements defined in the if block will be executed, otherwise if some other condition is true then the statements defined in the else-if block will be executed, at the last if none of the condition is true then the statements defined in the else block will be executed. There are multiple else-if blocks possible. It is similar to the switch case statement where the default is executed instead of else block if none of the cases is matched.

```
if(condition1){
//code to be executed if condition1 is true
}else if(condition2){
//code to be executed if condition2 is true
}
else if(condition3){
//code to be executed if condition3 is true
}
...
else{
```

```
//code to be executed if all the conditions are false
}
```

The example of an if-else-if statement in C language is given below.

```
#include<stdio.h>
int main(){
int number=0;
printf("enter a number:");
scanf("%d",&number);
if(number==10){
printf("number is equals to 10");
}
else if(number==50){
printf("number is equal to 50");
}
else if(number==100){
printf("number is equal to 100");
}
else{
printf("number is not equal to 10, 50 or 100");
}
return 0;
}
```

Program to calculate the grade of the student according to the specified marks.

```
#include <stdio.h>
int main()
{
    int marks;
    printf("Enter your marks?");
    scanf("%d",&marks);
    if(marks > 85 && marks <= 100)
    {
        printf("Congrats ! you scored grade A ...");
    }
    else if (marks > 60 && marks <= 85)
    {
        printf("You scored grade B + ...");
    }
```

- else if (marks > 40 && marks <= 60)
- {
- printf("You scored grade B ...");
- }
- else if (marks > 30 && marks <= 40)
- {
- printf("You scored grade C ...");
- }
- else
- {
- printf("Sorry you are fail ...");
- }
- }

Switch Statement

The switch statement in C is an alternate to if-else-if ladder statement which allows us to execute multiple operations for the different possible values of a single variable called switch variable. Here, we can define various statements in the multiple cases for the different values of a single variable.

<u>The syntax of switch statement:</u>

- **switch**(expression) {
- **case** value1:
- //code to be executed;
- **break**; //optional
- **case** value2:
- //code to be executed;
- **break**; //optional
- ......
- **default**:
- code to be executed **if** all cases are not matched;
- }

Rules for switch statement in C language

1) The *switch expression* must be of an integer or character type.

2) The *case value* must be an integer or character constant.

3) The *case value* can be used only inside the switch statement.

4) The *break statement* in switch case is not must. It is optional. If there is no break statement found in the case, all the cases will be executed present after the matched case. It is known as *fall through* the state of C switch statement.

We are assuming that there are following variables.

int x, y, z;
char a, b;
float f;

Valid Switch	Invalid Switch	Valid Case	Invalid Case
switch(x)	switch(f)	case 3;	case 2.5;
switch(x>y)	switch(x+2.5)	case 'a';	case x;
switch(a+b-2)		case 1+2;	case x+2;
switch(func(x,y))		case 'x'>'y';	case 1,2,3;

Functioning of switch case statement

First, the integer expression specified in the switch statement is evaluated. This value is then matched one by one with the constant values given in the different cases. If a match is found, then all the statements specified in that case are executed along with the all the cases present after that case including the default statement. No two cases can have similar values. If the matched case contains a break statement, then all the cases present after that will be skipped, and the control comes out of the switch. Otherwise, all the cases following the matched case will be executed.

Example 1

```
#include<stdio.h>
int main(){
int number=0;
printf("enter a number:");
scanf("%d",&number);
switch(number){
case 10:
printf("number is equals to 10");
break;
case 50:
printf("number is equal to 50");
break;
case 100:
printf("number is equal to 100");
break;
default:
printf("number is not equal to 10, 50 or 100");
}
return 0;
}
```

Example: 2

```
#include <stdio.h>
int main()
{
    int x = 10, y = 5;
    switch(x>y && x+y>0)
    {
        case 1:
        printf("hi");
        break;
        case 0:
        printf("bye");
        break;
        default:
        printf(" Hello bye ");
    }
}
```

Nested switch case statement

We can use as many switch statement as we want inside a switch statement. Such type of statements is called nested switch case statements. Consider the following example.

```
#include <stdio.h>
int main () {
   int i = 10;
   int j = 20;
   switch(i) {
        case 10:
          printf("the value of i evaluated in outer switch: %d\n",i);
        case 20:
          switch(j) {
            case 20:
              printf("The value of j evaluated in nested switch: %d\n",j);
          }
    }
    printf("Exact value of i is : %d\n", i );
    printf("Exact value of j is : %d\n", j );
    return 0;
  }
```

Let's summarize the above differences in a tabular form.

	If-else	**switch**
Definition	Depending on the condition in the 'if' statement, 'if' and 'else' blocks are executed.	The user will decide which statement is to be executed.
Expression	It contains either logical or equality expression.	It contains a single expression which can be either a character or integer variable.
Evaluation	It evaluates all types of data, such as integer, floating-point, character or Boolean.	It evaluates either an integer, or character.
Sequence of execution	First, the condition is checked. If the condition is true then 'if' block is executed otherwise 'else' block	It executes one case after another till the break keyword is not found, or the default statement is executed.
Default execution	If the condition is not true, then by default, else block will be executed.	If the value does not match with any case, then by default, default statement is executed.
Editing	Editing is not easy in the 'if-else' statement.	Cases in a switch statement are easy to maintain and modify. Therefore, we can say that the removal or editing of any case will not interrupt the execution of other cases.

Speed	If there are multiple choices implemented through 'if-else', then the speed of the execution will be slow.	If we have multiple choices then the switch statement is the best option as the speed of the execution will be much higher than 'if-else'.

C Loops

The looping can be defined as repeating the same process multiple times until a specific condition satisfies. There are three types of loops used in the C language.

Loops in C language

The looping simplifies the complex problems into the easy ones. It enables us to alter the flow of the program so that instead of writing the same code again and again, we can repeat the same code for a finite number of times. For example, if we need to print the first 10 natural numbers then, instead of using the printf statement 10 times, we can print inside a loop which runs up to 10 iterations.

Advantage of loops in C

1) It provides code reusability.

2) Using loops, we do not need to write the same code again and again.

3) Using loops, we can traverse over the elements of data structures (array or linked lists).

Types of C Loops

There are three types of loops in C language that is given below:

1. do while
2. while
3. for

Do while loop

The do-while loop continues until a given condition satisfies. It is also called post tested loop. It is used when it is necessary to execute the loop at least once. The do-while loop is mostly used in menu-driven programs where the termination condition depends upon the end user.

The syntax of the C language do-while loop is given below:

do{
//code to be executed
}**while**(condition);

Example 1

- #include<stdio.h>
- **int** main(){
- **int** i=1;
- **do**{
- printf("%d \n",i);
- i++;
- }**while**(i<=10);
- **return** 0;
- }

Program to print table for the given number using do while loop

- #include<stdio.h>
- **int** main(){
- **int** i=1,number=0;
- printf("Enter a number: ");
- scanf("%d",&number);
- **do**{
- printf("%d \n",(number*i));
- i++;
- }**while**(i<=10);
- **return** 0;
- }

Infinitive do while loop

The do-while loop will run infinite times if we pass any non-zero value as the conditional expression.

- **do**{
- //statement
- }**while**(1);

While loop

While loop is also known as a pre-tested loop. In general, a while loop allows a part of the code to be executed multiple times depending upon a given boolean condition. It can be viewed as a repeating if statement. The while loop is mostly used in the case where the number of iterations is not known in advance.

The syntax of while loop in c language is given below:

while(condition){
//code to be executed
}

Example 1:

- #include<stdio.h>
- **int** main(){
- **int** i=1;
- **while**(i<=10){
- printf("%d \n",i);
- i++;
- }
- **return** 0;
- }

Program to print table for the given number using while loop in C

- #include<stdio.h>
- **int** main(){
- **int** i=1,number=0;
- printf("Enter a number: ");
- scanf("%d", &number);
- **while**(i<=10){
- printf("%d \n",(number*i));
- i++;
- }
- **return** 0;
- }

Properties of while loop

- A conditional expression is used to check the condition. The statements defined inside the while loop will repeatedly execute until the given condition fails.
- The condition will be true if it returns 0. The condition will be false if it returns any non-zero number.
- In while loop, the condition expression is compulsory.

- Running a while loop without a body is possible.
- We can have more than one conditional expression in while loop.
- If the loop body contains only one statement, then the braces are optional.

Example 1

```
#include<stdio.h>
void main ()
{
    int j = 1;
    while(j+=2,j<=10)
    {
        printf("%d ",j);
    }
    printf("%d", j);
}
```

Output: 3 5 7 9 11

Example 2

```
#include<stdio.h>
void main ()
{
    while()
    {
        printf("hello");
    }
}
```

Example 3

```
#include<stdio.h>
void main ()
{
    int x = 10, y = 2;
    while(x+y-1)
    {
        printf("%d %d", x--, y--);
    }
}
```

Infinitive while loop in C

If the expression passed in while loop results in any non-zero value then the loop will run the infinite number of times.

```
while(1){
//statement
}
```

For loop

The for loop is used in the case where we need to execute some part of the code until the given condition is satisfied. The for loop is also called as a per-tested loop. It is better to use for loop if the number of iteration is known in advance.

The syntax of for loop in c language is given below

```
for(Expression 1; Expression 2; Expression 3){
//code to be executed
}
```

Examples 1:

```
#include<stdio.h>
int main(){
int i=0;
for(i=1;i<=10;i++){
printf("%d \n",i);
}
return 0;
}
```

Program: Print table for the given number using C for loop

```
#include<stdio.h>
int main(){
int i=1,number;
printf("Enter a number: ");
scanf("%d", &number);
for(i=1;i<=10;i++){
printf("%d \n",(number*i));
}
```

- **return** 0;
- }

Example 2:

- #include <stdio.h>
- **int** main()
- {
-     **int** a, b, c;
-     **for**(a=0,b=12,c=23;a<2;a++)
-     {
-         printf("%d ",a+b+c);
-     }
- }

Example 3:

- #include <stdio.h>
- **int** main()
- {
-     **int** i=1;
-     **for**(;i<5;i++)
-     {
-         printf("%d ",i);
-     }
- }

Example 4

- #include <stdio.h>
- **int** main()
- {
-     **int** i, j, k;
-     **for**(i=0,j=0,k=0;i<4,k<8,j<10;i++)
-     {
-         printf("%d %d %d\n", i, j, k);
-         j+=2;
-         k+=3;
-     }
- }

Example 5

```
#include <stdio.h>

int main()
{
    int i;
    for(i=0;;i++)
    {
        printf("%d",i);
    }
}  Output: go to Infinite loop
```

Example 6

```
#include<stdio.h>
void main ()
{
    int i=0,j=2;
    for(i = 0;i<5;i++,j=j+2)
    {
        printf("%d %d\n",i, j);
    }
}
```

Loop body

The braces {} are used to define the scope of the loop. However, if the loop contains only one statement, then we don't need to use braces. A loop without a body is possible. The braces work as a block separator, i.e., the value variable declared inside for loop is valid only for that block and not outside. Consider the following example.

```
#include<stdio.h>
void main ()
{
    int i;
    for(i=0;i<10;i++)
    {
        int i = 20;
        printf("%d ",i);
    }
}
```

Output: 20 20 20 20 20 20 20 20 20…

Infinitive for loop

To make a for loop infinite, we need not give any expression in the syntax. Instead of that, we need to provide two semicolons to validate the syntax of the for loop. This will work as an infinite for loop.

```
#include<stdio.h>
void main ()
{
    for(;;)
    {
        printf("welcome to javatpoint");
    }
}
```

Nested Loops in C

C supports nesting of loops in C. **Nesting of loops** is the feature in C that allows the looping of statements inside another loop. Let's observe an example of nesting loops in C.

Any number of loops can be defined inside another loop, i.e., there is no restriction for defining any number of loops. The nesting level can be defined at n times. You can define any type of loop inside another loop; for example, you can define '**while**' loop inside a '**for**' loop.

Syntax of Nested loop

```
Outer_loop
{
    Inner_loop
   {
         // inner loop statements.
   }
       // outer loop statements.
}
```

Outer_loop and **Inner_loop** are the valid loops that can be a 'for' loop, 'while' loop or 'do-while' loop.

Nested for loop

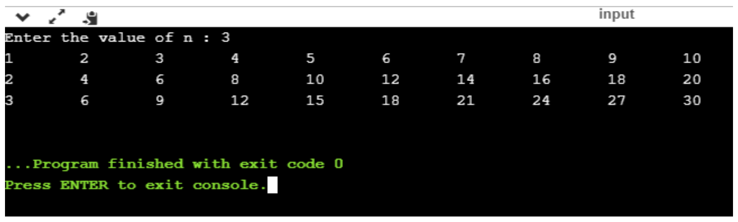
The nested for loop means any type of loop which is defined inside the 'for' loop.

```
for (initialization; condition; update)
{
   for(initialization; condition; update)
   {
        // inner loop statements.
   }
   // outer loop statements.
}
```

Example of nested for loop

```
• #include <stdio.h>
• int main( )
• {
•   int n;// variable declaration
•   printf("Enter the value of n :");
•   // Displaying the n tables.
•   for(int i=1; i<=n; i++)  // outer loop
•   {
•      for(int j=1 ;j<=10; j++)  // inner loop
•      {
•         printf("%d\t", (i*j)); // printing the value.
•      }
•      printf("\n");
• }
• return 0;
•   }
```

Output:



```
Enter the value of n : 3
1       2       3       4       5       6       7       8       9       10
2       4       6       8       10      12      14      16      18      20
3       6       9       12      15      18      21      24      27      30


...Program finished with exit code 0
Press ENTER to exit console.
```

Nested while loop

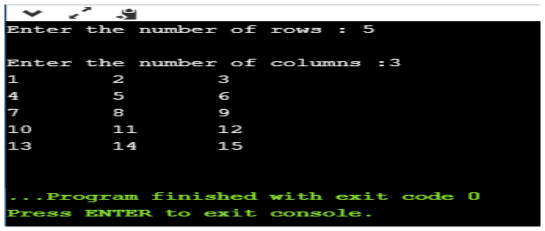
The nested while loop means any type of loop which is defined inside the 'while' loop.

- **while**(condition)
- {
-     **while**(condition)
-     {
-         // inner loop statements.
-     }
- // outer loop statements.
- }

Example of nested while loop

- #include <stdio.h>
- **int** main()
- {
-    **int** rows; // variable declaration
-    **int** columns; // variable declaration
-    **int** k=1; // variable initialization
-    printf("Enter the number of rows :"); // input the number of rows.
-    scanf("%d", &rows);
-    printf("\nEnter the number of columns :"); // input the number of columns.
-    scanf("%d", &columns);
-       **int** i=1;
-    **while**(i<=rows) // outer loop
-    {
-        **int** j=1;
-       **while**(j<=columns) // inner loop
-        {
-            printf("%d\t", k); // printing the value of k.
-            k++;   // increment counter
-            j++;
-        }
-        i++;
-        printf("\n");
-    }
- }

Output:



```
Enter the number of rows : 5

Enter the number of columns :3
1        2        3
4        5        6
7        8        9
10       11       12
13       14       15


...Program finished with exit code 0
Press ENTER to exit console.
```

Nested do.. while loop

The nested do..while loop means any type of loop which is defined inside the 'do..while' loop.

- **do**
- {
-    **do**
-   {
-       // inner loop statements.
-    }**while**(condition);
- // outer loop statements.
- }**while**(condition);

Example of nested do.. while loop.

- #include <stdio.h>
- **int** main()
- {
-   /*printing the pattern
-      ********
-      ********
-      ********
-      ******** */
- **int** i=1;
- **do**   // outer loop
- {
-     **int** j=1;
-     **do**   // inner loop
-    {
-       printf("*");

```
        j++;
     }while(j<=8);
      printf("\n");
      i++;
       }while(i<=4);
}
```

Infinite Loop in C

An infinite loop is a looping construct that does not terminate the loop and executes the loop forever. It is also called an **indefinite** loop or an **endless** loop. It either produces a continuous output or no output.

All the games also run in an infinite loop. The game will accept the user requests until the user exits from the game.

For loop

Let's see the **infinite 'for'** loop. The following is the definition for the **infinite** for loop:

```
for(; ;)
{
   // body of the for loop.
}
```

As we know that all the parts of the **'for' loop** are optional, and in the above for loop, we have not mentioned any condition; so, this loop will execute infinite times.

```
#include <stdio.h>
int main()
{
   for(;;)
   {
     printf("Hello");
   }
return 0;
}
```

In the above code, we run the 'for' loop infinite times, so **"Hello"** will be displayed infinitely.

while loop

Now, we will see how to create an infinite loop using a while loop. The following is the definition for the infinite while loop:

```
while(1)
{
  // body of the loop..
}
```

In the above while loop, we put '1' inside the loop condition. As we know that any non-zero integer represents the true condition while '0' represents the false condition.

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

In the above code, we have defined a while loop, which runs infinite times as it does not contain any condition. The value of 'i' will be updated an infinite number of times.

do..while loop

The **do..while** loop can also be used to create the infinite loop. The following is the syntax to create the infinite **do..while** loop.

```
do
{
   // body of the loop.
}while(1);
```

The above do..while loop represents the infinite condition as we provide the '1' value inside the loop condition. As we already know that non-zero integer represents the true condition, so this loop will run infinite times.

goto statement

We can also use the goto statement to define the infinite loop.

- infinite_loop;
- // body statements.
- **goto** infinite_loop;

In the above code, the goto statement transfers the control to the infinite loop.

Macros

We can also create the infinite loop with the help of a macro constant. Let's understand through an example.

- #include <stdio.h>
- #define infinite for(;;)
- **int** main()
- {
-   infinite
-   {
-       printf("hello");
-   }
-     **return** 0;
- }

Unintentional infinite loops

Sometimes the situation arises where unintentional infinite loops occur due to the bug in the code. If we are the beginners, then it becomes very difficult to trace them. Below are some measures to trace an unintentional infinite loop:

We should examine the semicolons carefully. Sometimes we put the semicolon at the wrong place, which leads to the infinite loop.

- #include <stdio.h>
- **int** main()
- {
- **int** i=1;
- **while**(i<=10);
- {
- printf("%d", i);
- i++;
- }

```
return 0;
}
```

We use the wrong loop condition which causes the loop to be executed indefinitely.

```
#include <stdio.h>
int main()
{
  for(int i=1;i>=1;i++)
  {
      printf("hello");
  }
return 0;
}
```

Break statement

The break is a keyword in C which is used to bring the program control out of the loop. The break statement is used inside loops or switch statement. The break statement breaks the loop one by one, i.e., in the case of nested loops, it breaks the inner loop first and then proceeds to outer loops. The break statement in C can be used in the following two scenarios:

1. With switch case
2. With loop

- **Syntax:**

```
//loop or switch case
break;
```

Example

```
#include<stdio.h>
#include<conio.h>
void main ()
{
    int i;
    for(i = 0; i<10; i++)
    {
        printf("%d ",i);
        if(i == 5)
        break;
    }
    printf("came outside of loop i = %d",i);

}
```

Example of C break statement with switch case

In such case, it breaks only the inner loop, but not outer loop.

```
#include<stdio.h>
int main(){
int i=1,j=1;//initializing a local variable
for(i=1;i<=3;i++){
for(j=1;j<=3;j++){
printf("%d &d\n",i,j);
if(i==2 && j==2){
break;//will break loop of j only
}
}//end of for loop
return 0;
}
```

As you can see the output on the console, 2 3 is not printed because there is a break statement after printing i==2 and j==2. But 3 1, 3 2 and 3 3 are printed because the break statement is used to break the inner loop only.

break statement with while loop

Consider the following example to use break statement inside while loop.

```
#include<stdio.h>
void main ()
{
    int i = 0;
    while(1)
    {
        printf("%d  ",i);
        i++;
        if(i == 10)
        break;
    }
    printf("came out of while loop");
}
```

Output

```
0  1  2  3  4  5  6  7  8  9  came out of while loop
```

break statement with do-while loop

Consider the following example to use the break statement with a do-while loop.

```
#include<stdio.h>
void main ()
{
   int n=2,i,choice;
   do
   {
       i=1;
       while(i<=10)
       {
           printf("%d X %d = %d\n",n,i,n*i);
           i++;
       }
       printf("do you want to continue with the table of %d , enter any non-zero value to continue.",n+1);
       scanf("%d",&choice);
    if(choice == 0)
       {
           break;
       }
       n++;
   }while(1);
}
```

Continue statement

The **continue statement** in C language is used to bring the program control to the beginning of the loop. The continue statement skips some lines of code inside the loop and continues with the next iteration. It is mainly used for a condition so that we can skip some code for a particular condition.

Syntax:

```
//loop statements
continue;
```

- //some lines of the code which is to be skipped

Continue statement example 1

- #include<stdio.h>
- **void** main ()
- {
-     **int** i = 0;
-     **while**(i!=10)
-     {
-         printf("%d", i);
-         **continue**;
-         i++;
-     }
- }

Output: infinite loop

Continue statement example 2

- #include<stdio.h>
- **int** main(){
- **int** i=1;//initializing a local variable
- //starting a loop from 1 to 10
- **for**(i=1;i<=10;i++){
- **if**(i==5){//if value of i is equal to 5, it will continue the loop
- **continue**;
- }
- printf("%d \n",i);
- }//end of for loop
- **return** 0;
- }

Goto statement

The goto statement is known as jump statement in C. As the name suggests, goto is used to transfer the program control to a predefined label. The goto statment can be used to repeat some part of the code for a particular condition. It can also be used to break the multiple loops which can't be done by using a single break statement. However, using goto is avoided these days since it makes the program less readable and complecated.

Syntax:

- label:
- //some part of the code;
- **goto** label;

goto example

An example to use goto statement in C language.

- #include <stdio.h>
- **int** main()
- {
-   **int** num, i=1;
-   printf("Enter the number whose table you want to print?");
-   scanf("%d",&num);
-   table:
-   printf("%d x %d = %d\n",num, i, num*i);
-   i++;
-   **if**(i<=10)
-   **goto** table;
- }

Output: Enter the number whose table you want to print?10

```
10 x 1 = 10
10 x 2 = 20
10 x 3 = 30
10 x 4 = 40
10 x 5 = 50
10 x 6 = 60
10 x 7 = 70
10 x 8 = 80
10 x 9 = 90
10 x 10 = 100
```

Type Casting in C

Typecasting allows us to convert one data type into other. In C language, we use cast operator for typecasting which is denoted by (type).

Syntax:

(type)value;

Note: It is always recommended to convert the lower value to higher for avoiding data loss.

Without Type Casting:

- **int** f= 9/4;
- printf("f : %d\n", f);   //Output: 2

With Type Casting:

- **float** f=(**float**) 9/4;
- printf("f : %f\n", f);   //Output: 2.250000

Type casting example

An example to cast int value into the float.

- #include<stdio.h>
- **int** main(){
- **float** f= (**float**)9/4;
- printf("%f\n", f);
- **return** 0;
- }

Output: 2.250000

Static in C

Static is a keyword used in C programming language. It can be used with both variables and functions, i.e., we can declare a static variable and static function as well. An ordinary variable is limited to the scope in which it is defined, while the scope of the static variable is throughout the program.

Static keyword can be used in the following situations:

- **Static global variable**
  When a global variable is declared with a static keyword, then it is known as a static global variable. It is declared at the top of the program, and its visibility is throughout the program.
- **Static function**
  When a function is declared with a static keyword known as a static function. Its lifetime is throughout the program.

- **Static local variable**
  When a local variable is declared with a static keyword, then it is known as a static local variable. The memory of a static local variable is valid throughout the program, but the scope of visibility of a variable is the same as the automatic local variables. However, when the function modifies the static local variable during the first function call, then this modified value will be available for the next function call also.
- **Static member variables**
  When the member variables are declared with a static keyword in a class, then it is known as static member variables. They can be accessed by all the instances of a class, not with a specific instance.
- **Static method**
  The member function of a class declared with a static keyword is known as a static method. It is accessible by all the instances of a class, not with a specific instance.

```
#include <stdio.h>
int main()
{
 printf("%d",func());
printf("\n%d",func());
 return 0;
}
int func()
{
   int count=0; // variable initialization
   count++; // incrementing counter variable

  return count; }
```

In the above code, the func() function is called. In func(), count variable gets updated. As soon as the function completes its execution, the memory of the count variable will be removed. If we do not want to remove the count from memory, then we need to use the count variable as static. If we declare the variable as static, then the variable will not be removed from the memory even when the function completes its execution.

Output: 1 1

Static variable

A static variable is a variable that persists its value across the various function calls.

Syntax: The syntax of a static variable is given below:

static data_type variable_name;

```
#include <stdio.h>
int main()
{
   printf("%d",func());
   printf("\n%d",func());
    return 0;
}
int func()
{
    static int count=0;
    count++;
    return count;
}
```

In the above code, we have declared the count variable as static. When the func() is called, the value of count gets updated to 1, and during the next function call, the value of the count variable becomes 2. Therefore, we can say that the value of the static variable persists within the function call.

Output 1 2

Static Function

As we know that non-static functions are global by default means that the function can be accessed outside the file also, but if we declare the function as static, then it limits the function scope. The static function can be accessed within a file only.

The static function would look like as:

```
static void func()
{
    printf("Hello");
}
```

Differences b/w static and global variable

Global variables are the variables that are declared outside the function. These global variables exist at the beginning of the program, and its scope remains till the end of the program. It can be accessed outside the program also.

Static variables are limited to the source file in which they are defined, i.e., they are not accessible by the other source files.

Both the static and global variables have static initialization. Here, static initialization means if we do not assign any value to the variable then by default, 0 value will be assigned to the variable.

Differences b/w static local and static global variable

Static global variable

If the variable declared with a static keyword outside the function, then it is known as a static global variable. It is accessible throughout the program.

Static local variable

The variable with a static keyword is declared inside a function is known as a static local variable. The scope of the static local variable will be the same as the automatic local variables, but its memory will be available throughout the program execution. When the function modifies the value of the static local variable during one function call, then it will remain the same even during the next function call.

Properties of a static variable

The following are the properties of a static variable:

- The memory of a static variable is allocated within a static variable.
- Its memory is available throughout the program, but the scope will remain the same as the automatic local variables. Its
- value will persist across the various function calls.
- If we do not assign any value to the variable, then the default value will be 0.
- A global static variable cannot be accessed outside the program, while a global variable can be accessed by other source files.

Programming Errors in C

Errors are the problems or the faults that occur in the program, which makes the behavior of the program abnormal, and experienced developers can also make these faults. Programming errors are also known as the bugs or faults, and the process of removing these bugs is known as **debugging**.

These errors are detected either during the time of compilation or execution. Thus, the errors must be removed from the program for the successful execution of the program.

There are mainly five types of errors exist in C programming:

- Syntax error
- Run-time error
- Linker error
- Logical error
- Semantic error

Syntax error

Syntax errors are also known as the compilation errors as they occurred at the compilation time, or we can say that the syntax errors are thrown by the compilers. These errors are mainly occurred due to the mistakes while typing or do not follow the syntax of the specified programming language. These mistakes are generally made by beginners only because they are new to the language. These errors can be easily debugged or corrected.

For example:

1. If we want to declare the variable of type integer,
2. **int** a; // this is the correct form
3. Int a; // this is an incorrect form.

Commonly occurred syntax errors are:

- If we miss the parenthesis (}) while writing the code.
- Displaying the value of a variable without its declaration.
- If we miss the semicolon (;) at the end of the statement.

```
#include <stdio.h>
int main()
{
   a = 10;
   printf("The value of a is : %d", a);
```

- **return** 0;
- }

In the above output, we observe that the code throws the error that 'a' is undeclared. This error is nothing but the syntax error only.

There can be another possibility in which the syntax error can exist, i.e., if we make mistakes in the basic construct. Let's understand this scenario through an example.

- #include <stdio.h>
- **int** main()
- {
- **int** a=2;
- **if**(.) // syntax error
-
- printf("a is greater than 1");
- **return** 0;
- }

In the above code, we put the (.) instead of condition in 'if', so this generates the syntax error as shown in the below screenshot.

Run-time error

Sometimes the errors exist during the execution-time even after the successful compilation known as run-time errors. When the program is running, and it is not able to perform the operation is the main cause of the run-time error. The division by zero is the common example of the run-time error. These errors are very difficult to find, as the compiler does not point to these errors.

- #include <stdio.h>
- **int** main()
- {
- **int** a=2;
- **int** b=2/0;
- printf("The value of b is : %d", b);
- **return** 0;
- }

In the above output, we observe that the code shows the run-time error, i.e., division by zero.

Linker error

Linker errors are mainly generated when the executable file of the program is not created. This can be happened either due to the wrong function prototyping or usage of the wrong header file. For example, the **main.c** file contains the **sub()** function whose declaration and definition is done in some other file such as **func.c**. During the compilation, the compiler finds the **sub()** function in **func.c** file, so it generates two object files, i.e., **main.o** and **func.o**. At the execution time, if the definition of **sub()** function is not found in the **func.o** file, then the linker error will be thrown. The most common linker error that occurs is that we use **Main()** instead of **main().**

```
#include <stdio.h>
int Main()
{
   int a=78;
   printf("The value of a is : %d", a);
   return 0;
}
```

Logical error

The logical error is an error that leads to an undesired output. These errors produce the incorrect output, but they are error-free, known as logical errors. These types of mistakes are mainly done by beginners. The occurrence of these errors mainly depends upon the logical thinking of the developer. If the programmers sound logically good, then there will be fewer chances of these errors.

```
#include <stdio.h>
int main()
{
  int sum=0; // variable initialization
  int k=1;
  for(int i=1;i<=10;i++); // logical error, as we put the semicolon after loop
  {
     sum=sum+k;
     k++;
  }
printf("The  value of sum is %d", sum);
```

-    **return** 0;
- }

In the above code, we are trying to print the sum of 10 digits, but we got the wrong output as we put the semicolon (;) after the for loop, so the inner statements of the for loop will not execute. This produces the wrong output.

Semantic error

Semantic errors are the errors that occurred when the statements are not understandable by the compiler.

The following can be the cases for the semantic error:

- Use of a un-initialized variable.
  int i;
  i=i+2;
- Type compatibility
  int b = "java";
- Errors in expressions
  int a, b, c;
  a+b = c;
- Array index out of bound
  int a[10];
  a[10] = 34;

- #include <stdio.h>
- **int** main()
- {
- **int** a,b,c;
- a=2;
- b=3;
- c=1;
- a+b=c; // semantic error
- **return** 0;
- }

In the above code, we use the statement **a+b =c**, which is incorrect as we cannot use the two operands on the left-side.

Compile time vs Runtime

Compile-time and Runtime are the two programming terms used in the software development. Compile-time is the time at which the source code is converted into an executable code while the run time is the time at which the executable code is started running. Both the compile-time and runtime refer to different types of error.

Compile-time errors

Compile-time errors are the errors that occurred when we write the wrong syntax. If we write the wrong syntax or semantics of any programming language, then the compile-time errors will be thrown by the compiler. The compiler will not allow to run the program until all the errors are removed from the program. When all the errors are removed from the program, then the compiler will generate the executable file.

The compile-time errors can be:

- Syntax errors
- Semantic errors

Syntax errors

When the programmer does not follow the syntax of any programming language, then the compiler will throw the syntax error.

For example,

int a, b:

The above declaration generates the compile-time error as in C, every statement ends with the semicolon, but we put a colon (:) at the end of the statement.

Semantic errors

The semantic errors exist when the statements are not meaningful to the compiler.

For example,

a+b=c;

The above statement throws a compile-time errors. In the above statement, we are assigning the value of 'c' to the summation of 'a' and 'b' which is not possible in C programming language as it can contain only one variable on the left of the assignment operator while right of the assignment operator can contain more than one variable.

The above statement can be re-written as:

c=a+b;

Runtime errors

The runtime errors are the errors that occur during the execution and after compilation. The examples of runtime errors are division by zero, etc. These errors are not easy to detect as the compiler does not point to these errors.

the differences between compile-time and runtime:

Compile-time	Runtime
The compile-time errors are the errors which are produced at the compile-time, and they are detected by the compiler.	The runtime errors are the errors which are not generated by the compiler and produce an unpredictable result at the execution time.
In this case, the compiler prevents the code from execution if it detects an error in the program.	In this case, the compiler does not detect the error, so it cannot prevent the code from the execution.
It contains the syntax and semantic errors such as missing semicolon at the end of the statement.	It contains the errors such as division by zero, determining the square root of a negative number.

Example of Compile-time error

```
#include <stdio.h>
int main()
{
    int a=20;
    printf("The value of a is : %d",a):
    return 0;
}
```

In the above code, we have tried to print the value of 'a', but it throws an error. We put the colon at the end of the statement instead of a semicolon, so this code generates a compile-time error.

Example of runtime error

```
#include <stdio.h>
int main()
{
   int a=20;
   int b=a/0; // division by zero
   printf("The value of b is : %d",b):
   return 0;
}
```

In the above code, we try to divide the value of 'b' by zero, and this throws a runtime error.

UNIT-2

FUNCTIONS

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.

Advantage of functions in C

There are the following advantages of C functions.

- By using functions, we can avoid rewriting same logic/code again and again in a program.
- We can call C functions any number of times in a program and from any place in a program.
- We can track a large C program easily when it is divided into multiple functions.
- Reusability is the main achievement of C functions.
- However, Function calling is always a overhead in a C program.

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.

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.

An example of 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.

Example with return value:

- **int** get(){
- **return** 10;
- }

In the above example, we have to return 10 as a value, so the return type is int. If you want to return floating-point value (e.g., 10.2, 3.1, 54.5, etc), you need to use float as the return type of the method.

- **float** get(){
- **return** 10.2;
- }

Now, you need to call the function, to get the value of the function.

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.

1. **function without arguments and without return value**
2. **function without arguments and with return value**
3. **function with arguments and without return value**
4. **function with arguments and with return value**

1. Example for Function without argument and without return value

Example 1

```
#include<stdio.h>
void printName();
void main ()
{
    printf("Hello ");
    printName();
}
void printName()
{
    printf("C programming");
}
```

Example 2

```
#include<stdio.h>
void sum();
void main()
{
    printf("\nGoing to calculate the sum of two numbers:");
    sum();
}
void sum()
{
    int a,b;
    printf("\nEnter two numbers");
    scanf("%d %d", &a, &b);
    printf("The sum is %d",a+b);
}
```

2. Example for Function without argument and with return value

Example 1

```
•  #include<stdio.h>
•  int sum();
•  void main()
•  {
•      int result;
•      printf("\n Going to calculate the sum of two numbers:");
•      result = sum();
•      printf("%d", result);
•  }
•  int sum()
•  {
•      int a, b;
•      printf("\n Enter two numbers");
•      scanf("%d %d", &a, &b);
•      return a + b;
•  }
```

Example 2: program to calculate the area of the square

```
•  #include<stdio.h>
•  int square();
•  void main()
•  {
•      printf("Going to calculate the area of the square\n");
•      float area = square();
•      printf("The area of the square: %f\n",area);
•  }
•  int square()
•  {
•      float side;
•      printf("Enter the length of the side in meters: ");
•      scanf("%f",&side);
•      return side * side;
•  }
```

3. Example for Function with argument and without return value

Example 1

```
#include<stdio.h>
#include<conio.h>
void sum(int, int);
void main()
{
   int a, b;
   printf("\n Going to calculate the sum of two numbers:");
   printf("\n Enter two numbers:");
   scanf("%d %d", &a, &b);
   sum(a, b);
getch();
}
void sum(int a, int b)
{
   printf("\nThe sum is %d",a+b);
}
```

Example 2: program to calculate the average of five numbers.

```
#include<stdio.h>
void average(int, int, int, int, int);
void main()
{
   int a, b, c, d, e;
   printf("\n Going to calculate the average of five numbers:");
   printf("\n Enter five numbers:");
   scanf("%d %d %d %d %d", &a, &b, &c, &d, &e);
   average(a, b, c, d, e);
}
void average(int a, int b, int c, int d, int e)
{
   float avg;
   avg = (a+b+c+d+e)/5;
   printf("The average of given five numbers : %f",avg);
}
```

4. Example for Function with argument and with return value

Example 1

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

```
int sum(int, int);
void main()
{
   int a,b,result;
   printf("\nGoing to calculate the sum of two numbers:");
   printf("\nEnter two numbers:");
   scanf("%d %d",&a,&b);
   result = sum(a,b);
   printf("\nThe sum is : %d",result);
getch();
}
int sum(int a, int b)
{
   return a+b;
}
```

Example 2: Program to check whether a number is even or odd

```
#include<stdio.h>
int even_odd(int);
void main()
{
 int n,flag=0;
 printf("\nGoing to check whether a number is even or odd");
 printf("\nEnter the number: ");
 scanf("%d",&n);
 flag = even_odd(n);
 if(flag == 0)
 {
   printf("\nThe number is odd");
 }
 else
 {
   printf("\nThe number is even");
 }
}
int even_odd(int n)
{
   if(n%2 == 0)
   {
      return 1;
```

- }
- **else**
- {
- **return** 0;
- }
- }

Library Functions

Library functions are the inbuilt function in C that are grouped and placed at a common place called the library. Such functions are used to perform some specific operations. For example, printf is a library function used to print on the console. The library functions are created by the designers of compilers. All C standard library functions are defined inside the different header files saved with the extension **.h**. We need to include these header files in our program to make use of the library functions defined in such header files. For example, To use the library functions such as printf/scanf we need to include stdio.h in our program which is a header file that contains all the library functions regarding standard input/output.

The list of mostly used header files is given in the following table.

SN	Header file	Description
1	stdio.h	This is a standard input/output header file. It contains all the library functions regarding standard input/output.
2	conio.h	This is a console input/output header file.
3	string.h	It contains all string related library functions like gets(), puts(),etc.
4	stdlib.h	This header file contains all the general library functions like malloc(), calloc(), exit(), etc.
5	math.h	This header file contains all the math operations related functions like sqrt(), pow(), etc.
6	time.h	This header file contains all the time-related functions.
7	ctype.h	This header file contains all character handling functions.
8	stdarg.h	Variable argument functions are defined in this header file.
9	signal.h	All the signal handling functions are defined in this header file.

10	setjmp.h	This file contains all the jump functions.
11	locale.h	This file contains locale functions.
12	errno.h	This file contains error handling functions.
13	assert.h	This file contains diagnostics functions.

Call by value and Call by reference in C

There are two methods to pass the data into the function in C language, i.e., *call by value* and *call by reference*.

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.

Call by Value Example: Swapping the values of the two variables

```
#include <stdio.h>
#include<conio.h>
void swap(int , int); //prototype of the function
int main()
{
    int a = 10;
    int b = 20;
    printf("Before swapping the values in main a = %d, b = %d\n",a,b);
    swap(a, b);
    printf("After swapping values in main a = %d, b = %d\n",a,b);    // The value of
actual parameters do not change by changing the formal parameters in call by val
ue, a = 10, b = 20
}
void swap (int a, int b)
{
   int temp;
   temp = a;
   a=b;
   b=temp;
   printf("After swapping values in function a = %d, b = %d\n",a,b); // Formal parameters, a = 20, b = 10
}
```

Output:
```
Before swapping the values in main a = 10, b = 20
After swapping values in function a = 20, b = 10
After swapping values in main a = 10, b = 20
```

Call by reference in C

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

Call by reference Example: Swapping the values of the two variables

```
#include <stdio.h>
void swap(int *, int *); //prototype of the function
int main()
{
   int a = 10;
   int b = 20;
   printf("Before swapping the values in main a = %d, b = %d\n",a,b); // printing the value of a and b in main
   swap(&a,&b);
   printf("After swapping values in main a = %d, b = %d\n",a,b); // The values of actual parameters do change in call by reference, a = 10, b = 20
}
void swap (int *a, int *b)
{
   int temp;
   temp = *a;
   *a=*b;
```

- *b=temp;
- printf("After swapping values in function a = %d, b = %d\n",*a,*b); // Formal parameters, a = 20, b = 10
- }

OUTPUT:
```
Before swapping the values in main a = 10, b = 20
After swapping values in function a = 20, b = 10
After swapping values in main a = 20, b = 10
```

Difference between call by value and call by reference in c

No.	Call by value	Call by reference
1	A copy of the value is passed into the function	An address of value is passed into the function
2	Changes made inside the function is limited to the function only. The values of the actual parameters do not change by changing the formal parameters.	Changes made inside the function validate outside of the function also. The values of the actual parameters do change by changing the formal parameters.
3	Actual and formal arguments are created at the different memory location	Actual and formal arguments are created at the same memory location

Recursion in C

Recursion is the process which comes into existence when a function calls a copy of itself to work on a smaller problem. Any function which calls itself is called recursive function, and such function calls are called recursive calls. Recursion involves several numbers of recursive calls. However, it is important to impose a termination condition of recursion. Recursion code is shorter than iterative code however it is difficult to understand.

Recursion cannot be applied to all the problem, but it is more useful for the tasks that can be defined in terms of similar subtasks. For Example, recursion may be applied to sorting, searching, and traversal problems.

Generally, iterative solutions are more efficient than recursion since function call is always overhead. Any problem that can be solved recursively, can also be solved iteratively. However, some problems are best suited to be solved by the recursion, for example, tower of Hanoi, Fibonacci series, factorial finding, etc.

In the following example, recursion is used to calculate the factorial of a number.

```
#include <stdio.h>
int fact (int);
int main()
{
    int n,f;
    printf("Enter the number whose factorial you want to calculate?");
    scanf("%d",&n);
    f = fact(n);
    printf("factorial = %d",f);
}
int fact(int n)
{
    if (n==0)
    {
        return 0;
    }
    else if ( n == 1)
    {
        return 1;
    }
    else
    {
        return n*fact(n-1);
    }
}
```

Output

```
Enter the number whose factorial you want to calculate?5
factorial = 120
```

We can understand the above program of the recursive method call by the figure given below:

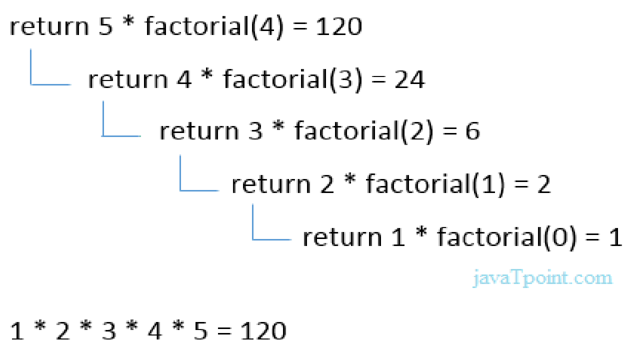

return 5 * factorial(4) = 120
return 4 * factorial(3) = 24
return 3 * factorial(2) = 6
return 2 * factorial(1) = 2
return 1 * factorial(0) = 1
javaTpoint.com
1 * 2 * 3 * 4 * 5 = 120

Fig: Recursion

Recursive Function

A recursive function performs the tasks by dividing it into the subtasks. There is a termination condition defined in the function which is satisfied by some specific subtask. After this, the recursion stops and the final result is returned from the function.

The case at which the function doesn't recur is called the base case whereas the instances where the function keeps calling itself to perform a subtask, is called the recursive case. All the recursive functions can be written using this format.

Pseudocode for writing any recursive function is given below.

```
if (test_for_base)
{
    return some_value;
}
else if (test_for_another_base)
{
    return some_another_value;
}
else
{
    // Statements;
    recursive call;
}
```

Example of recursion in C

Let's see an example to find the nth term of the Fibonacci series.

```
#include<stdio.h>
int fibonacci(int);
void main ()
{
   int n, f;
   printf("Enter the value of n?");
   scanf("%d", &n);
   f = fibonacci(n);
   printf("%d", f);
}
int fibonacci (int n)
{
   if (n==0)
   {
   return 0;
   }
   else if (n == 1)
   {
      return 1;
   }
   else
   {
      return fibonacci(n-1)+fibonacci(n-2);
   }
}
```

```
Output: Enter the value of n? 12                    144
```

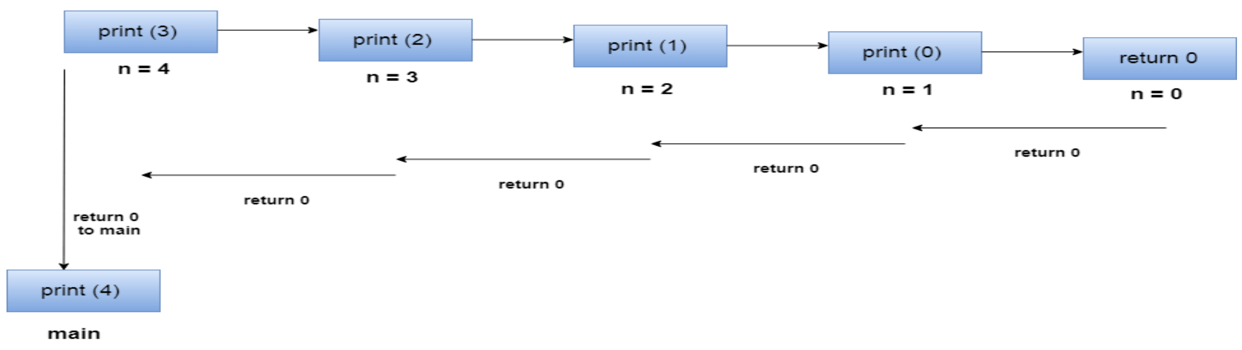
Memory allocation of Recursive method

Each recursive call creates a new copy of that method in the memory. Once some data is returned by the method, the copy is removed from the memory. Since all the variables and other stuff declared inside function get stored in the stack, therefore a separate stack is maintained at each recursive call. Once the value is returned from the corresponding function, the stack gets destroyed. Recursion involves so much complexity in resolving and tracking the values at each recursive call. Therefore we need to maintain the stack and track the values of the variables defined in the stack.

Let us consider the following example to understand the memory allocation of the recursive functions.

```
int display (int n)
{
   if(n == 0)
      return 0; // terminating condition
   else
   {
      printf("%d",n);
      return display(n-1); // recursive call
   }
}
```

Explanation: Let us examine this recursive function for n = 4. First, all the stacks are maintained which prints the corresponding value of n until n becomes 0, Once the termination condition is reached, the stacks get destroyed one by one by returning 0 to its calling stack. Consider the following image for more information regarding the stack trace for the recursive functions.



Stack tracing for recursive function call

Storage Classes in C

Storage classes in C are used to determine the lifetime, visibility, memory location, and initial value of a variable. There are four types of storage classes in C

- Automatic
- External
- Static
- Register

Storage Classes	Storage Place	Default Value	Scope	Lifetime

Auto	RAM	Garbage Value	Local	Within function
Extern	RAM	Zero	Global	Till the end of the main program Maybe declared anywhere in the program
Static	RAM	Zero	Local	Till the end of the main program, Retains value between multiple functions call
Register	Register	Garbage Value	Local	Within the function

Automatic

- Automatic variables are allocated memory automatically at runtime.
- The visibility of the automatic variables is limited to the block in which they are defined.

  The scope of the automatic variables is limited to the block in which they are defined.
- The automatic variables are initialized to garbage by default.
- The memory assigned to automatic variables gets freed upon exiting from the block.
- The keyword used for defining automatic variables is auto.
- Every local variable is automatic in C by default.

Example 1

```
#include <stdio.h>
int main()
{
int a; //auto
char b;
float c;
printf("%d %c %f",a,b,c); // printing initial default value of automatic variables a, b, and c.
return 0;
}
```

Output: garbage garbage garbage

Example 2

- #include <stdio.h>
- **int** main()
- {
- **int** a = 10,i;
- printf("%d ",++a);
- {
- **int** a = 20;
- **for** (i=0;i<3;i++)
- {
- printf("%d ",a); // 20 will be printed 3 times since it is the local value of a
- }
- }
- printf("%d ",a); // 11 will be printed since the scope of a = 20 is ended.
- }

Output: 11 20 20 20 11

Static

- The variables defined as static specifier can hold their value between the multiple function calls.
- Static local variables are visible only to the function or the block in which they are defined.
- A same static variable can be declared many times but can be assigned at only one time.
- Default initial value of the static integral variable is 0 otherwise null.
- The visibility of the static global variable is limited to the file in which it has declared.
- The keyword used to define static variable is static.

Example 1

- #include<stdio.h>
- **static char** c;
- **static int** i;
- **static float** f;
- **static char** s[100];
- **void** main ()
- {

- printf("%d %d %f %s",c,i,f); // the initial default value of c, i, and f will be printe d.
- }

Output: 0 0 0.000000 (null)

Example 2

- #include<stdio.h>
- **void** sum()
- {
- **static int** a = 10;
- **static int** b = 24;
- printf("%d %d \n",a,b);
- a++;
- b++;
- }
- **void** main()
- {
- **int** i;
- **for**(i = 0; i< 3; i++)
- {
- sum(); // The static variables holds their value between multiple function calls.
- }
- }

Output:

```
10 24
11 25
12 26
```

Register

- The variables defined as the register is allocated the memory into the CPU registers depending upon the size of the memory remaining in the CPU.
- We can not dereference the register variables, i.e., we can not use &operator for the register variable.
- The access time of the register variables is faster than the automatic variables.
- The initial default value of the register local variables is 0.

- The register keyword is used for the variable which should be stored in the CPU register. However, it is compiler?s choice whether or not; the variables can be stored in the register.
- We can store pointers into the register, i.e., a register can store the address of a variable.
- Static variables can not be stored into the register since we can not use more than one storage specifier for the same variable.

Example 1

- #include <stdio.h>
- **int** main()
- {
- **register int** a; // variable a is allocated memory in the CPU register. The initial de fault value of a is 0.
- printf("%d",a);
- }

Output: 0

Example 2

- #include <stdio.h>
- **int** main()
- {
- **register int** a = 0;
- printf("%u",&a); // This will give a compile time error since we can not access the address of a register variable.
- }

Output:

```
main.c:5:5: error: address of register variable ?a? requested

  printf("%u",&a);
```

External

- The external storage class is used to tell the compiler that the variable defined as extern is declared with an external linkage elsewhere in the program.
- The variables declared as extern are not allocated any memory. It is only declaration and intended to specify that the variable is declared elsewhere in the program.
- The default initial value of external integral type is 0 otherwise null.

- We can only initialize the extern variable globally, i.e., we can not initialize the external variable within any block or method.
- An external variable can be declared many times but can be initialized at only once.
- If a variable is declared as external then the compiler searches for that variable to be initialized somewhere in the program which may be extern or static. If it is not, then the compiler will show an error.

Example 1

- #include <stdio.h>
- **int** main()
- {
- **extern** **int** a;
- printf("%d",a);
- }

Output: main.c:(.text+0x6): undefined reference to `a'

collect2: error: ld returned 1 exit status

Example 2

- #include <stdio.h>
- **int** a;
- **int** main()
- {
- **extern** **int** a; // variable a is defined globally, the memory will not be allocated to a
- printf("%d",a);
- }

Output: 0

Example 3

- #include <stdio.h>
- **int** a;
- **int** main()
- {
- **extern** **int** a = 0; // this will show a compiler error since we can not use extern and initializer at same time

- printf("%d",a);
- }

Output

compile time error
main.c: In function ?main?:
main.c:5:16: error: ?a? has both ?extern? and initializer
extern int a = 0;

Example 4

- #include <stdio.h>
- **int** main()
- {
- **extern int** a; // Compiler will search here for a variable a defined and initialized s omewhere in the pogram or not.
- printf("%d",a);
- }
- **int** a = 20;

Output: 20

Example 5

- **extern int** a;
- **int** a = 10;
- #include <stdio.h>
- **int** main()
- {
- printf("%d",a);
- }
- **int** a = 20; // compiler will show an error at this line

Output: compile time error

ARRAY

An array is defined as the collection of similar type of data items stored at contiguous memory locations. Arrays are the derived data type in C programming language which can store the primitive type of data such as int, char, double, float, etc. It also has the capability to store the collection of derived data types, such as pointers, structure, etc. The array is the simplest data structure where each data element can be randomly accessed by using its index number.

C array is beneficial if you have to store similar elements. For example, if we want to store the marks of a student in 6 subjects, then we don't need to define different variables for the marks in the different subject. Instead of that, we can define an array which can store the marks in each subject at the contiguous memory locations.

By using the array, we can access the elements easily. Only a few lines of code are required to access the elements of the array.

Properties of Array

The array contains the following properties.

- Each element of an array is of same data type and carries the same size, i.e., int = 4 bytes.
- Elements of the array are stored at contiguous memory locations where the first element is stored at the smallest memory location.
- Elements of the array can be randomly accessed since we can calculate the address of each element of the array with the given base address and the size of the data element.

Advantage of C Array

1) Code Optimization: Less code to the access the data.

2) Ease of traversing: By using the for loop, we can retrieve the elements of an array easily.

3) Ease of sorting: To sort the elements of the array, we need a few lines of code only.

4) Random Access: We can access any element randomly using the array.

Disadvantage of C Array

1) Fixed Size: Whatever size, we define at the time of declaration of the array, we can't exceed the limit. So, it doesn't grow the size dynamically like LinkedList which we will learn later.

Declaration of C Array

We can declare an array in the c language in the following way.

data_type array_name[array_size];

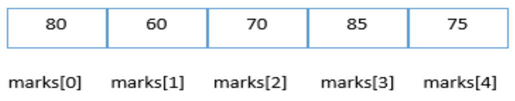
int marks[5];

Here, int is the *data_type*, marks are the *array_name*, and 5 is the *array_size*.

Initialization of C Array

The simplest way to initialize an array is by using the index of each element. We can initialize each element of the array by using the index. Consider the following example.

- marks[0]=80;//initialization of array
- marks[1]=60;
- marks[2]=70;
- marks[3]=85;
- marks[4]=75;



Initialization of Array

Array example

- #include<stdio.h>
- **int** main(){
- **int** i=0;
- **int** marks[5];//declaration of array
- marks[0]=80;//initialization of array
- marks[1]=60;
- marks[2]=70;
- marks[3]=85;
- marks[4]=75;
- //traversal of array

- **for**(i=0;i<5;i++){
- printf("%d \n",marks[i]);
- }//end of for loop
- **return** 0;
- }

Output: 80 60 70 85 75

C Array: Declaration with Initialization

We can initialize the c array at the time of declaration. Let's see the code.

int marks[5]={20,30,40,50,60};

In such case, there is **no requirement to define the size**. So it may also be written as the following code.

int marks[]={20,30,40,50,60};

Let's see the C program to declare and initialize the array in C.

- #include<stdio.h>
- **int** main(){
- **int** i=0;
- **int** marks[5]={20,30,40,50,60};//declaration and initialization of array
- //traversal of array
- **for**(i=0;i<5;i++){
- printf("%d \n",marks[i]);
- }
- **return** 0;
- }

Output: 20 30 40 50 60

C Array Example: Sorting an array

In the following program, we are using bubble sort method to sort the array in ascending order.

- #include<stdio.h>
- **void** main ()
- {

```
    int i, j,temp;
    int a[10] = { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
    for(i = 0; i<10; i++)
    {
        for(j = i+1; j<10; j++)
        {
            if(a[j] > a[i])
            {
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }
    printf("Printing Sorted Element List ...\n");
    for(i = 0; i<10; i++)
    {
        printf("%d\n",a[i]);
    }
}
```

Program to print the largest and second largest element of the array.

```
#include<stdio.h>
void main ()
{
    int arr[100],i,n,largest,sec_largest;
    printf("Enter the size of the array?");
    scanf("%d",&n);
    printf("Enter the elements of the array?");
    for(i = 0; i<n; i++)
    {
        scanf("%d",&arr[i]);
    }
    largest = arr[0];
    sec_largest = arr[1];
    for(i=0;i<n;i++)
    {
        if(arr[i]>largest)
        {
            sec_largest = largest;
•               largest = arr[i];
•           }
•           else if (arr[i]>sec_largest && arr[i]!=largest)
•           {
•               sec_largest=arr[i];
•           }
•       }
•       printf("largest = %d, second largest = %d",largest,sec_largest);
•
•   }
```

Two Dimensional Array in C

The two-dimensional array can be defined as an array of arrays. The 2D array is organized as matrices which can be represented as the collection of rows and columns. However, 2D arrays are created to implement a relational database lookalike data structure. It provides ease of holding the bulk of data at once which can be passed to any number of functions wherever required.

Declaration of two dimensional Array in C

The syntax to declare the 2D array is given below.

data_type array_name[rows][columns];

Consider the following example.

int twodimen[4][3];

Here, 4 is the number of rows, and 3 is the number of columns.

Initialization of 2D Array in C

In the 1D array, we don't need to specify the size of the array if the declaration and initialization are being done simultaneously. However, this will not work with 2D arrays. We will have to define at least the second dimension of the array. The two-dimensional array can be declared and defined in the following way.

1. **int** arr[4][3]={{1,2,3},{2,3,4},{3,4,5},{4,5,6}};

Two-dimensional array example in C

1. #include<stdio.h>

```
int main(){
int i=0,j=0;
int arr[4][3]={{1,2,3},{2,3,4},{3,4,5},{4,5,6}};
//traversing 2D array
for(i=0;i<4;i++){
 for(j=0;j<3;j++){
   printf("arr[%d] [%d] = %d \n",i,j,arr[i][j]);
 }//end of j
}//end of i
return 0;
}
```

Output

```
arr[0][0] = 1
arr[0][1] = 2
arr[0][2] = 3
arr[1][0] = 2
arr[1][1] = 3
arr[1][2] = 4
arr[2][0] = 3
arr[2][1] = 4
arr[2][2] = 5
arr[3][0] = 4
arr[3][1] = 5
arr[3][2] = 6
```

1. C 2D array example: Storing elements in a matrix and printing it.

```
#include <stdio.h>
void main ()
{
    int arr[3][3],i,j;
    for (i=0;i<3;i++)
    {
        for (j=0;j<3;j++)
        {
            printf("Enter a[%d][%d]: ",i,j);
            scanf("%d",&arr[i][j]);
        }
•     }
•     printf("\n printing the elements ....\n");
•     for(i=0;i<3;i++)
•     {
•         printf("\n");
•         for (j=0;j<3;j++)
•         {
•             printf("%d\t",arr[i][j]);
•         }
•     }
• }
```

Output

```
Enter a[0][0]: 56
Enter a[0][1]: 10
Enter a[0][2]: 30
Enter a[1][0]: 34
Enter a[1][1]: 21
Enter a[1][2]: 34
Enter a[2][0]: 45
Enter a[2][1]: 56
Enter a[2][2]: 78

 printing the elements ....

56	10	30
34	21	34
45	56	78
```

Return an Array in C

Passing array to a function

```
• #include <stdio.h>
• void getarray(int arr[])
• {
•     printf("Elements of array are : ");
•     for(int i=0;i<5;i++)
•     {
•         printf("%d ", arr[i]);
•     }
}
int main()
{
  int arr[5]={45,67,34,78,90};
  getarray(arr);
  return 0;
}
```

In the above program, we have first created the array **arr[]** and then we pass this array to the function getarray(). The **getarray()** function prints all the elements of the array arr[].

Passing array to a function as a pointer

Now, we will see how to pass an array to a function as a pointer.

```
#include <stdio.h>
void printarray(char *arr)
{
    printf("Elements of array are : ");
    for(int i=0;i<5;i++)
    {
        printf("%c ", arr[i]);
    }
}
int main()
{
  char arr[5]={'A','B','C','D','E'};
  printarray(arr);
  return 0;
}
```

In the above code, we have passed the array to the function as a pointer. The function **printarray()** prints the elements of an array.

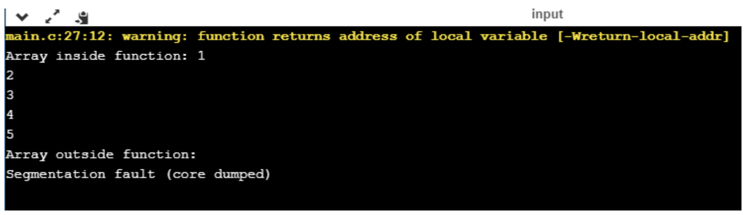
How to return an array from a function

Returning pointer pointing to the array

```
#include <stdio.h>
int *getarray()
{
    int arr[5];
    printf("Enter the elements in an array : ");
    for(int i=0;i<5;i++)
    {
        scanf("%d", &arr[i]);
    }
    return arr;
}
int main()
{
  int *n;
  n=getarray();
  printf("\nElements of array are :");
  for(int i=0;i<5;i++)
    {
        printf("%d", n[i]);
    }
    return 0;
}
```

In the above program, **getarray()** function returns a variable 'arr'. It returns a local variable, but it is an illegal memory location to be returned, which is allocated within a function in the stack. Since the program control comes back to the **main()** function, and all the variables in a stack are freed. Therefore, we can say that this program is returning memory location, which is already de-allocated, so the output of the program is a **segmentation fault**.

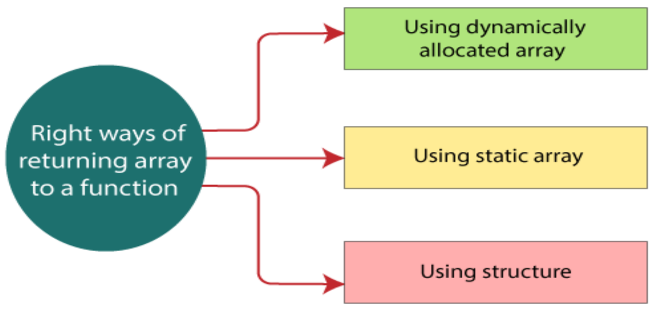
Output



```
main.c:27:12: warning: function returns address of local variable [-Wreturn-local-addr]
Array inside function: 1
2
3
4
5
Array outside function:
Segmentation fault (core dumped)
```

There are three right ways of returning an array to a function:

- **Using dynamically allocated array**
- **Using static array**
- **Using structure**

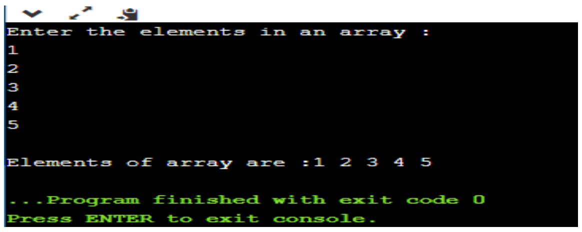

Right ways of returning array to a function
Using dynamically allocated array
Using static array
Using structure

Returning array by passing an array which is to be returned as a parameter to the function.

```
#include <stdio.h>
int *getarray(int *a)
{

    printf("Enter the elements in an array : ");
    for(int i=0;i<5;i++)
    {
        scanf("%d", &a[i]);
    }
    return a;
}
int main()
{
  int *n;
  int a[5];
  n=getarray(a);
  printf("\nElements of array are :");
  for(int i=0;i<5;i++)
    {
        printf("%d", n[i]);
    }
    return 0;
}
```

Output



```
Enter the elements in an array :
1
2
3
4
5

Elements of array are :1 2 3 4 5

...Program finished with exit code 0
Press ENTER to exit console.
```

Returning array using malloc() function.

```
#include <stdio.h>
#include<malloc.h>
int *getarray()
{
    int size;
    printf("Enter the size of the array : ");
    scanf("%d",&size);
    int *p= malloc(sizeof(size));
    printf("\nEnter the elements in an array");
    for(int i=0;i<size;i++)
    {
        scanf("%d",&p[i]);
    }
    return p;
}
int main()
{
   int *ptr;
   ptr=getarray();
   int length=sizeof(*ptr);
   printf("Elements that you have entered are : ");
   for(int i=0;ptr[i]!='\0';i++)
    {
      printf("%d ", ptr[i]);
    }
  return 0;
}
```

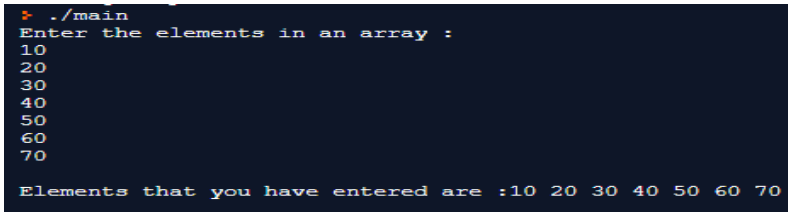
Using Static Variable

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

- **static int** arr[7];
- printf("Enter the elements in an array : ");
- **for**(**int** i=0;i<7;i++)
- {
- scanf("%d",&arr[i]);
- }
- **return** arr;
-
- }
- **int** main()
- {
- **int** *ptr;
- ptr=getarray();
- printf("\nElements that you have entered are :");
- **for**(**int** i=0;i<7;i++)
- {
- printf("%d ", ptr[i]);
- }
- }

In the above code, we have created the variable **arr[]** as static in **getarray()** function, which is available throughout the program. Therefore, the function getarray() returns the actual memory location of the variable '**arr**'.

Output



```
> ./main
Enter the elements in an array :
10
20
30
40
50
60
70

Elements that you have entered are :10 20 30 40 50 60 70
```

Using Structure

The structure is a user-defined data type that can contain a collection of items of different types. Now, we will create a program that returns an array by using structure.

- #include <stdio.h>
- #include<malloc.h>
- **struct** array
- {
- **int** arr[8];

```
};
struct array getarray()
{
    struct array y;
    printf("Enter the elements in an array : ");
    for(int i=0;i<8;i++)
    {
        scanf("%d",&y.arr[i]);
    }
    return y;
}
int main()
{
  struct array x=getarray();
  printf("Elements that you have entered are :");
  for(int i=0;x.arr[i]!='\0';i++)
  {
      printf("%d ", x.arr[i]);
  }
    return 0;
}
```

Passing Array to Function in C

In C, there are various general problems which requires passing more than one variable of the same type to a function. For example, consider a function which sorts the 10 elements in ascending order. Such a function requires 10 numbers to be passed as the actual parameters from the main function. Here, instead of declaring 10 different numbers and then passing into the function, we can declare and initialize an array and pass that into the function. This will resolve all the complexity since the function will now work for any number of values.

As we know that the array_name contains the address of the first element. Here, we must notice that we need to pass only the name of the array in the function which is intended to accept an array. The array defined as the formal parameter will automatically refer to the array specified by the array name defined as an actual parameter.

Consider the following syntax to pass an array to the function.

functionname(arrayname);//passing array

Methods to declare a function that receives an array as an argument

There are 3 ways to declare the function which is intended to receive an array as an argument.

First way:

return_type function(type arrayname[])

Declaring blank subscript notation [] is the widely used technique.

Second way:

return_type function(type arrayname[SIZE])

Optionally, we can define size in subscript notation [].

Third way:

return_type function(type *arrayname)

You can also use the concept of a pointer. In pointer chapter, we will learn about it.

C language passing an array to function example

```
#include<stdio.h>
int minarray(int arr[],int size){
int min=arr[0];
int i=0;
for(i=1;i<size;i++){
if(min>arr[i]){
min=arr[i];
}
}//end of for
return min;
}//end of function
int main(){
int i=0,min=0;
int numbers[]={4,5,7,3,8,9};//declaration of array
min=minarray(numbers,6);//passing array with size
printf("minimum number is %d \n",min);
return 0;
}
```

Output: minimum number is 3

Function to sort the array

```
#include<stdio.h>
void Bubble_Sort(int[]);
void main ()
{
    int arr[10] = { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
    Bubble_Sort(arr);
}
void Bubble_Sort(int a[]) //array a[] points to arr.
{
int i, j,temp;
    for(i = 0; i<10; i++)
    {
        for(j = i+1; j<10; j++)
        {
            if(a[j] < a[i])
            {
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }
    printf("Printing Sorted Element List ...\n");
    for(i = 0; i<10; i++)
    {
        printf("%d\n",a[i]);
    }
}
```

Output

```
Printing Sorted Element List ...
7
9
10
12
23
```

```
23
34
44
78
101
```

Returning array from the function

As we know that, a function cannot return more than one value. However, if we try to write the return statement as return a, b, c; to return three values (a,b,c), the function will return the last mentioned value which is c in our case. In some problems, we may need to return multiple values from a function. In such cases, an array is returned from the function.

Returning an array is similar to passing the array into the function. The name of the array is returned from the function. To make a function returning an array, the following syntax is used.

```
int * Function_name() {
//some statements;
return array_type;
}
```

To store the array returned from the function, we can define a pointer which points to that array. We can traverse the array by increasing that pointer since pointer initially points to the base address of the array. Consider the following example that contains a function returning the sorted array.

```
#include<stdio.h>
int* Bubble_Sort(int[]);
void main ()
{
    int arr[10] = { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
    int *p = Bubble_Sort(arr), i;
    printf("printing sorted elements ...\n");
    for(i=0;i<10;i++)
    {
        printf("%d\n",*(p+i));
    }
}
int* Bubble_Sort(int a[]) //array a[] points to arr.
{
int i, j,temp;
```

```
    for(i = 0; i<10; i++)
    {
        for(j = i+1; j<10; j++)
        {
            if(a[j] < a[i])
            {
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }
    return a;
}
```

Output

```
Printing Sorted Element List ... 7 9 10 12 23 23 34 44 78 101
```

C POINTERS

The pointer in C language is a variable which stores the address of another variable. This variable can be of type int, char, array, function, or any other pointer. The size of the pointer depends on the architecture. However, in 32-bit architecture the size of a pointer is 2 byte.

Consider the following example to define a pointer which stores the address of an integer.

- **int** n = 10;
- **int*** p = &n; // Variable p of type pointer is pointing to the address of the variable n of type integer.

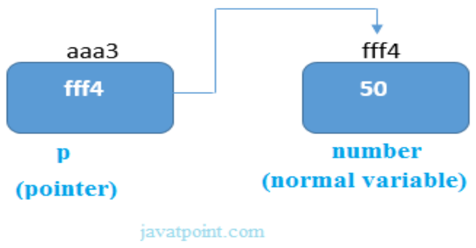
Declaring a pointer

The pointer in c language can be declared using * (asterisk symbol). It is also known as indirection pointer used to dereference a pointer.

- **int** *a;//pointer to int
- **char** *c;//pointer to char

Pointer Example

An example of using pointers to print the address and value is given below.



As you can see in the above figure, pointer variable stores the address of number variable, i.e., fff4. The value of number variable is 50. But the address of pointer variable p is aaa3.

By the help of * (**indirection operator**), we can print the value of pointer variable p.

Let's see the pointer example as explained for the above figure.

- #include<stdio.h>
- int main(){
- int number=50;
- int *p;
- p=&number;//stores the address of number variable
- printf("Address of p variable is %x \n",p); // p contains the address of the number therefore printing p gives the address of number.
- printf("Value of p variable is %d \n",*p); // As we know that * is used to dereference a pointer therefore if we print *p, we will get the value stored at the address contained by p.
- return 0;
- }

Output

```
Address of number variable is fff4
Address of p variable is fff4
Value of p variable is 50
```

Pointer to array

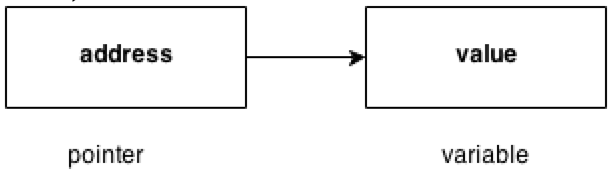
- int arr[10];
- int *p[10]=&arr; // Variable p of type pointer is pointing to the address of an integer array arr.

Pointer to a function

- void show (int);
- void(*p)(int) = &show; // Pointer p is pointing to the address of a function

Pointer to structure

- struct st {
- int i;
- float f;
- }ref;
- struct st *p = &ref;



Advantage of pointer

1) Pointer **reduces the code** and **improves the performance**, it is used to retrieving strings, trees, etc. and used with arrays, structures, and functions.

2) We can **return multiple values from a function** using the pointer.

3) It makes you able to **access any memory location** in the computer's memory.

Usage of pointer

There are many applications of pointers in c language.

1) Dynamic memory allocation

In c language, we can dynamically allocate memory using malloc() and calloc() functions where the pointer is used.

2) Arrays, Functions, and Structures

Pointers in c language are widely used in arrays, functions, and structures. It reduces the code and improves the performance.

Address Of (&) Operator

The address of operator '&' returns the address of a variable. But, we need to use %u to display the address of a variable.

```
#include<stdio.h>
int main(){
int number=50;
printf("value of number is %d, address of number is %u",number,&number);
return 0;
}
```

Output: value of number is 50, address of number is fff4

NULL Pointer

A pointer that is not assigned any value but NULL is known as the NULL pointer. If you don't have any address to be specified in the pointer at the time of declaration, you can assign NULL value. It will provide a better approach.

```
int *p=NULL;
```

In the most libraries, the value of the pointer is 0 (zero).

Pointer Program to swap two numbers without using the 3rd variable.

```
#include<stdio.h>
int main(){
int a=10, b=20;
int *p1=&a,*p2=&b;
printf("Before swap: *p1=%d *p2=%d",*p1,*p2);
*p1=*p1+*p2;
*p2=*p1-*p2;
*p1=*p1-*p2;
printf("\n After swap: *p1=%d *p2=%d",*p1,*p2);
return 0;
}
```

Output:

```
Before swap: *p1=10 *p2=20
After swap: *p1=20 *p2=10
```

Reading complex pointers

There are several things which must be taken into the consideration while reading the complex pointers in C. Let’s see the precedence and associativity of the operators which are used regarding pointers.

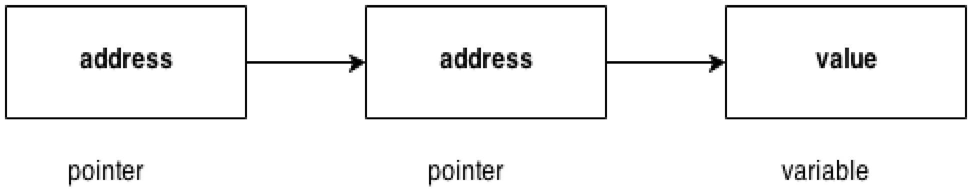
Operator	Precedence	Associativity
(), []	1	Left to right
*, identifier	2	Right to left
Data type	3	-

Here, we must notice that,

- (): This operator is a bracket operator used to declare and define the function.
- []: This operator is an array subscript operator
- * : This operator is a pointer operator.
- Identifier: It is the name of the pointer. The priority will always be assigned to this.
- Data type: Data type is the type of the variable to which the pointer is intended to point. It also includes the modifier like signed int, long, etc).

DOUBLE POINTER (POINTER TO POINTER)

As we know that, a pointer is used to store the address of a variable in C. Pointer reduces the access time of a variable. However, In C, we can also define a pointer to store the address of another pointer. Such pointer is known as a double pointer (pointer to pointer). The first pointer is used to store the address of a variable whereas the second pointer is used to store the address of the first pointer. Let's understand it by the diagram given below.



The syntax of declaring a double pointer is given below.

- int **p; // pointer to a pointer which is pointing to an integer.
- Consider the following example.
- #include<stdio.h>
- void main ()
- {
-     int a = 10;

- int *p;
- int **pp;
- p = &a; // pointer p is pointing to the address of a
- pp = &p; // pointer pp is a double pointer pointing to the address of pointer p
- printf("address of a: %x\n",p); // Address of a will be printed
- printf("address of p: %x\n",pp); // Address of p will be printed
- printf("value stored at p: %d\n",*p); // value stoted at the address contained by p i.e. 10 will be printed
- printf("value stored at pp: %d\n",**pp); // value stored at the address contained by the pointer stoyred at pp
- }

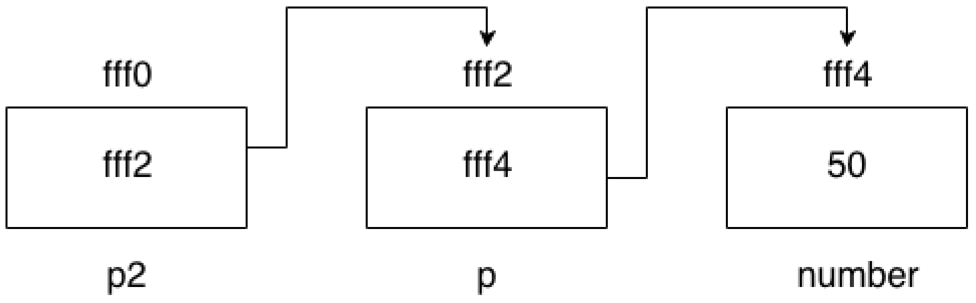
OUTPUT

```
address of a: d26a8734
address of p: d26a8738
value stored at p: 10
value stored at pp: 10
```

C double pointer example

Let's see an example where one pointer points to the address of another pointer.

Backward Skip 10sPlay VideoForward Skip 10s



As you can see in the above figure, p2 contains the address of p (fff2), and p contains the address of number variable (fff4).

1. #include<stdio.h>
2. int main(){
3. int number=50;
4. int *p;//pointer to int

```c
int **p2;//pointer to pointer
p=&number;//stores the address of number variable
p2=&p;
printf("Address of number variable is %x \n",&number);
printf("Address of p variable is %x \n",p);
printf("Value of *p variable is %d \n",*p);
printf("Address of p2 variable is %x \n",p2);
printf("Value of **p2 variable is %d \n",*p);
return 0;
}
```

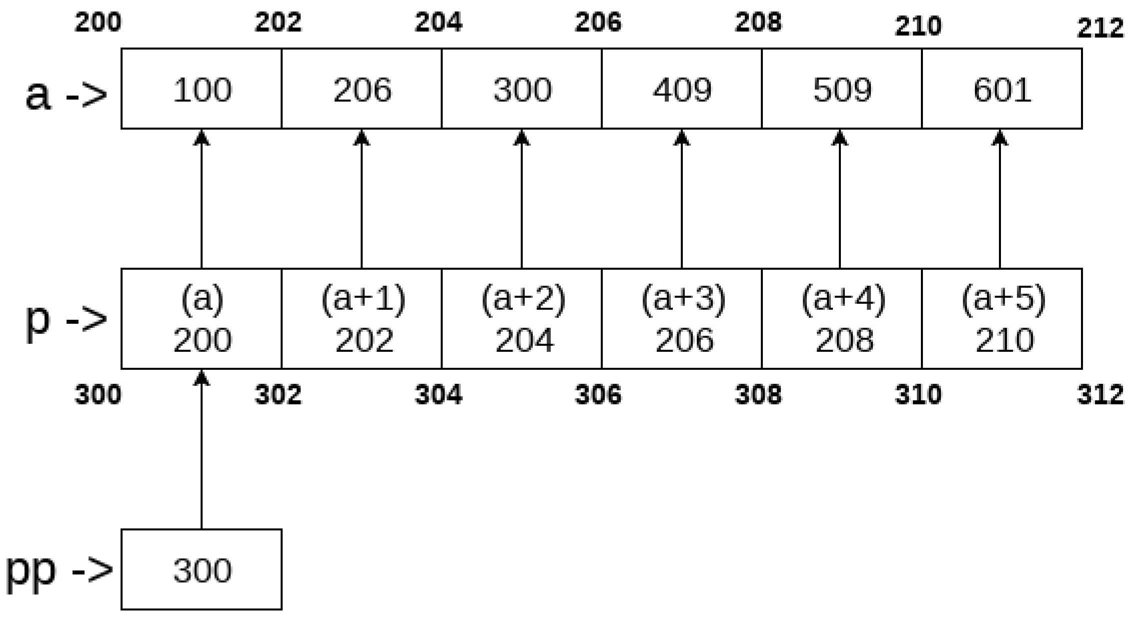
OUTPUT

```
Address of number variable is fff4
Address of p variable is fff4
Value of *p variable is 50
Address of p2 variable is fff2
Value of **p variable is 50
```

Q. What will be the output of the following program?

```c
#include<stdio.h>
void main ()
{
    int a[10] = {100, 206, 300, 409, 509, 601}; //Line 1
    int *p[] = {a, a+1, a+2, a+3, a+4, a+5}; //Line 2
    int **pp = p; //Line 3
    pp++; // Line 4
    printf("%d %d %d\n",pp-p,*pp - a,**pp); // Line 5
    *pp++; // Line 6
    printf("%d %d %d\n",pp-p,*pp - a,**pp); // Line 7
    ++*pp; // Line 8
    printf("%d %d %d\n",pp-p,*pp - a,**pp); // Line 9
    ++**pp; // Line 10
    printf("%d %d %d\n",pp-p,*pp - a,**pp); // Line 11
}
```

EXPLANATION



To access a[0] $\longrightarrow$ a[0] = * (a) = *p[0] = **(p+0) = **(pp+0) = 10

In the above question, the pointer arithmetic is used with the double pointer. An array of 6 elements is defined which is pointed by an array of pointer p. The pointer array p is pointed by a double pointer pp. However, the above image gives you a brief idea about how the memory is being allocated to the array a and the pointer array p. The elements of p are the pointers that are pointing to every element of the array a. Since we know that the array name contains the base address of the array hence, it will work as a pointer and can the value can be traversed by using *(a), *(a+1), etc. As shown in the image, a[0] can be accessed in the following ways.

- a[0]: it is the simplest way to access the first element of the array
- *(a): since a store the address of the first element of the array, we can access its value by using indirection pointer on it.
- *p[0]: if a[0] is to be accessed by using a pointer p to it, then we can use indirection operator (*) on the first element of the pointer array p, i.e., *p[0].
- **(pp): as pp stores the base address of the pointer array, *pp will give the value of the first element of the pointer array that is the address of the first element of the integer array. **p will give the actual value of the first element of the integer array.

Coming to the program, Line 1 and 2 declare the integer and pointer array relatively. Line 3 initializes the double pointer to the pointer array p. As shown in the image, if the address of the array starts from 200 and the size of the integer is 2, then the pointer array will contain the values as 200, 202, 204, 206, 208, 210. Let us consider that the base address of the pointer array is 300; the double pointer pp contains the address of pointer array, i.e., 300. Line number 4 increases the value of pp by 1, i.e., pp will now point to address 302.

Line number 5 contains an expression which prints three values, i.e., pp - p, *pp - a, **pp. Let's calculate them each one of them.

- pp = 302, p = 300 => pp-p = (302-300)/2 => pp-p = 1, i.e., 1 will be printed.
- pp = 302, *pp = 202, a = 200 => *pp - a = 202 - 200 = 2/2 = 1, i.e., 1 will be printed.
- pp = 302, *pp = 202, *(*pp) = 206, i.e., 206 will be printed.

Therefore as the result of line 5, The output 1, 1, 206 will be printed on the console. On line 6, *pp++ is written. Here, we must notice that two unary operators * and ++ will have the same precedence. Therefore, by the rule of associativity, it will be evaluated from right to left. Therefore the expression *pp++ can be rewritten as (*(pp++)). Since, pp = 302 which will now become, 304. *pp will give 204.

On line 7, again the expression is written which prints three values, i.e., pp-p, *pp-a, *pp. Let's calculate each one of them.

- pp = 304, p = 300 => pp - p = (304 - 300)/2 => pp-p = 2, i.e., 2 will be printed.
- pp = 304, *pp = 204, a = 200 => *pp-a = (204 - 200)/2 = 2, i.e., 2 will be printed.
- pp = 304, *pp = 204, *(*pp) = 300, i.e., 300 will be printed.

Therefore, as the result of line 7, The output 2, 2, 300 will be printed on the console. On line 8, ++*pp is written. According to the rule of associativity, this can be rewritten as, (++(*(pp))). Since, pp = 304, *pp = 204, the value of *pp = *(p[2]) = 206 which will now point to a[3].

On line 9, again the expression is written which prints three values, i.e., pp-p, *pp-a, *pp. Let's calculate each one of them.

- pp = 304, p = 300 => pp - p = (304 - 300)/2 => pp-p = 2, i.e., 2 will be printed.
- pp = 304, *pp = 206, a = 200 => *pp-a = (206 - 200)/2 = 3, i.e., 3 will be printed.
- pp = 304, *pp = 206, *(*pp) = 409, i.e., 409 will be printed.

Therefore, as the result of line 9, the output 2, 3, 409 will be printed on the console. On line 10, ++**pp is writen. according to the rule of associativity, this can be rewritten as,

(++(*(*(pp)))). pp = 304, *pp = 206, **pp = 409, ++**pp => *pp = *pp + 1 = 410. In other words, a[3] = 410.

On line 11, again the expression is written which prints three values, i.e., pp-p, *pp-a, *pp. Let's calculate each one of them.

- pp = 304, p = 300 => pp - p = (304 - 300)/2 => pp-p = 2, i.e., 2 will be printed.
- pp = 304, *pp = 206, a = 200 => *pp-a = (206 - 200)/2 = 3, i.e., 3 will be printed.
- On line 8, **pp = 410.

Therefore as the result of line 9, the output 2, 3, 410 will be printed on the console.

At last, the output of the complete program will be given as:

Output

```
1 1 206
2 2 300
2 3 409
2 3 410
```

POINTER ARITHMETIC IN C

We can perform arithmetic operations on the pointers like addition, subtraction, etc. However, as we know that pointer contains the address, the result of an arithmetic operation performed on the pointer will also be a pointer if the other operand is of type integer. In pointer-from-pointer subtraction, the result will be an integer value. Following arithmetic operations are possible on the pointer in C language:

- Increment
- Decrement
- Addition
- Subtraction
- Comparison

Incrementing Pointer in C

If we increment a pointer by 1, the pointer will start pointing to the immediate next location. This is somewhat different from the general arithmetic since the value of the pointer will get increased by the size of the data type to which the pointer is pointing.

We can traverse an array by using the increment operation on a pointer which will keep pointing to every element of the array, perform some operation on that, and update itself in a loop.

The Rule to increment the pointer is given below:

```
new_address= current_address + i * size_of(data type)
```

Where i is the number by which the pointer get increased.

1. 32-bit

For 32-bit int variable, it will be incremented by 2 bytes.

2. 64-bit

For 64-bit int variable, it will be incremented by 4 bytes.

Let's see the example of incrementing pointer variable on 64-bit architecture.

```c
#include<stdio.h>
int main(){
int number=50;
int *p;//pointer to int
p=&number;//stores the address of number variable
printf("Address of p variable is %u \n",p);
p=p+1;
printf("After increment: Address of p variable is %u \n",p); // in our case, p will get incremented by 4 bytes.
return 0;
}
```

Output

```
Address of p variable is 3214864300
After increment: Address of p variable is 3214864304
```

3. Traversing an array by using pointer

```c
#include<stdio.h>
void main ()
{
    int arr[5] = {1, 2, 3, 4, 5};
    int *p = arr;
    int i;
    printf("printing array elements...\n");
    for(i = 0; i< 5; i++)
    {
        printf("%d  ",*(p+i));
    }
}
```

Output

```
printing array elements...
1  2  3  4  5
```

Decrementing Pointer in C

Like increment, we can decrement a pointer variable. If we decrement a pointer, it will start pointing to the previous location. The formula of decrementing the pointer is given below:

```
new_address= current_address - i * size_of(data type)
```

4. 32-bit

For 32-bit int variable, it will be decremented by 2 bytes.

5. 64-bit

For 64-bit int variable, it will be decremented by 4 bytes.

Let's see the example of decrementing pointer variable on 64-bit OS.

```c
#include <stdio.h>
void main(){
int number=50;
int *p;//pointer to int
p=&number;//stores the address of number variable
printf("Address of p variable is %u \n",p);
p=p-1;
printf("After decrement: Address of p variable is %u \n",p); // P will now point to the immidiate previous location.
}
```

Output

```
Address of p variable is 3214864300
After decrement: Address of p variable is 3214864296
```

C Pointer Addition

We can add a value to the pointer variable. The formula of adding value to pointer is given below:

```
new_address= current_address + (number * size_of(data type))
```

6. 32-bit

For 32-bit int variable, it will add 2 * number.

7. 64-bit

For 64-bit int variable, it will add 4 * number.

Let's see the example of adding value to pointer variable on 64-bit architecture.

```c
#include<stdio.h>
int main(){
int number=50;
int *p;//pointer to int
p=&number;//stores the address of number variable
printf("Address of p variable is %u \n",p);
p=p+3;   //adding 3 to pointer variable
printf("After adding 3: Address of p variable is %u \n",p);
return 0;
}
```

Output

```
Address of p variable is 3214864300
After adding 3: Address of p variable is 3214864312
```

As you can see, the address of p is 3214864300. But after adding 3 with p variable, it is 3214864312, i.e., 4*3=12 increment. Since we are using 64-bit architecture, it increments 12. But if we were using 32-bit architecture, it was incrementing to 6 only, i.e., 2*3=6. As integer value occupies 2-byte memory in 32-bit OS.

C Pointer Subtraction

Like pointer addition, we can subtract a value from the pointer variable. Subtracting any number from a pointer will give an address. The formula of subtracting value from the pointer variable is given below:

```
new_address= current_address - (number * size_of(data type))
```

8. 32-bit

For 32-bit int variable, it will subtract 2 * number.

9. 64-bit

For 64-bit int variable, it will subtract 4 * number.

Let's see the example of subtracting value from the pointer variable on 64-bit architecture.

```
#include<stdio.h>
int main(){
int number=50;
int *p;//pointer to int
p=&number;//stores the address of number variable
printf("Address of p variable is %u \n",p);
p=p-3; //subtracting 3 from pointer variable
printf("After subtracting 3: Address of p variable is %u \n",p);
return 0;
}
```

Output

```
Address of p variable is 3214864300
After subtracting 3: Address of p variable is 3214864288
```

You can see after subtracting 3 from the pointer variable, it is 12 (4*3) less than the previous address value.

However, instead of subtracting a number, we can also subtract an address from another address (pointer). This will result in a number. It will not be a simple arithmetic operation, but it will follow the following rule.

If two pointers are of the same type,

1. Address2 - Address1 = (Subtraction of two addresses)/size of data type which pointer poi nts

Consider the following example to subtract one pointer from an another.

```
#include<stdio.h>
void main ()
{
    int i = 100;
    int *p = &i;
    int *temp;
    temp = p;
    p = p + 3;
    printf("Pointer Subtraction: %d - %d = %d",p, temp, p-temp);
}
```

Output

```
Pointer Subtraction: 1030585080 - 1030585068 = 3
```

Illegal arithmetic with pointers

There are various operations which can not be performed on pointers. Since, pointer stores address hence we must ignore the operations which may lead to an illegal address, for example, addition, and multiplication. A list of such operations is given below.

- Address + Address = illegal
- Address * Address = illegal
- Address % Address = illegal
- Address / Address = illegal
- Address & Address = illegal
- Address ^ Address = illegal
- Address | Address = illegal

- ~Address = illegal

Pointer to function in C

As we discussed in the previous chapter, a pointer can point to a function in C. However, the declaration of the pointer variable must be the same as the function. Consider the following example to make a pointer pointing to the function.

```c
#include<stdio.h>
int addition ();
int main ()
{
    int result;
    int (*ptr)();
    ptr = &addition;
    result = (*ptr)();
    printf("The sum is %d",result);
}
int addition()
{
    int a, b;
    printf("Enter two numbers?");
    scanf("%d %d",&a,&b);
    return a+b;
}
```

Output

```
Enter two numbers?10 15
The sum is 25
```

Pointer to Array of functions in C

To understand the concept of an array of functions, we must understand the array of function. Basically, an array of the function is an array which contains the addresses of functions. In other words, the pointer to an array of functions is a pointer pointing to an array which contains the pointers to the functions. Consider the following example.

```c
#include<stdio.h>
int show();
int showadd(int);
int (*arr[3])();
int (*(*ptr)[3])();

int main ()
{
    int result1;
    arr[0] = show;
    arr[1] = showadd;
    ptr = &arr;
    result1 = (**ptr)();
    printf("printing the value returned by show : %d",result1);
    (*(*ptr+1))(result1);
}
int show()
{
    int a = 65;
    return a++;
}
int showadd(int b)
{
    printf("\nAdding 90 to the value returned by show: %d",b+90);
}
```

Output

```
printing the value returned by show : 65
Adding 90 to the value returned by show: 155
```

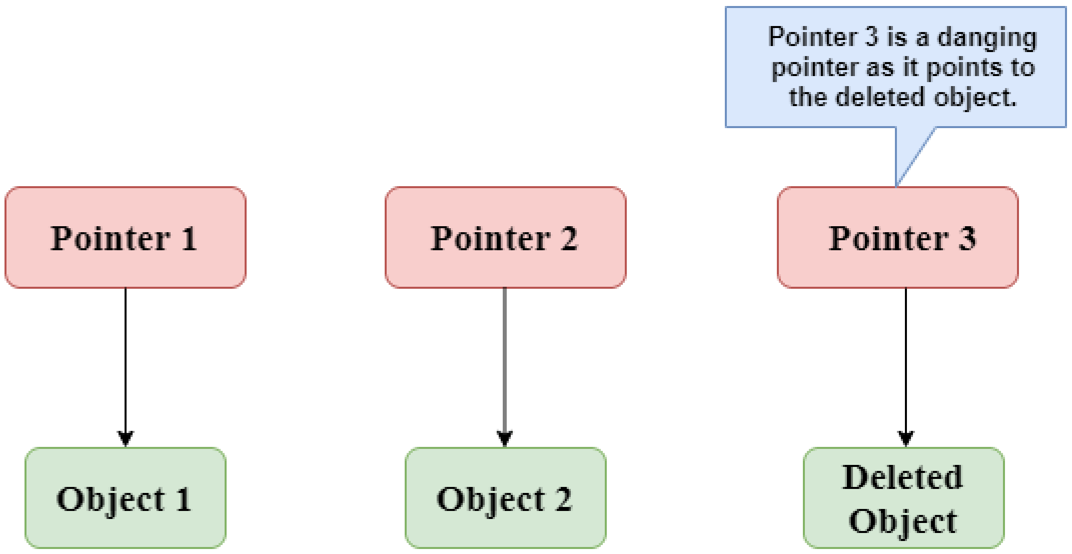
DANGLING POINTERS IN C

The most common bugs related to pointers and memory management is dangling/wild pointers. Sometimes the programmer fails to initialize the pointer with a valid address, then this type of initialized pointer is known as a dangling pointer in C.

Dangling pointer occurs at the time of the object destruction when the object is deleted or de-allocated from memory without modifying the value of the pointer. In this case, the pointer is pointing to the memory, which is de-allocated. The dangling pointer can point to the memory, which contains either the program code or the code of the operating system.

If we assign the value to this pointer, then it overwrites the value of the program code or operating system instructions; in such cases, the program will show the undesirable result or may even crash. If the memory is re-allocated to some other process, then we dereference the dangling pointer will cause the segmentation faults.

Let's observe the following examples.


Pointer 3 is a danging pointer as it points to the deleted object.
Pointer 1
Pointer 2
Pointer 3
Object 1
Object 2
Deleted Object

In the above figure, we can observe that the **Pointer 3** is a dangling pointer. **Pointer 1** and **Pointer 2** are the pointers that point to the allocated objects, i.e., Object 1 and Object 2, respectively. **Pointer 3** is a dangling pointer as it points to the de-allocated object.

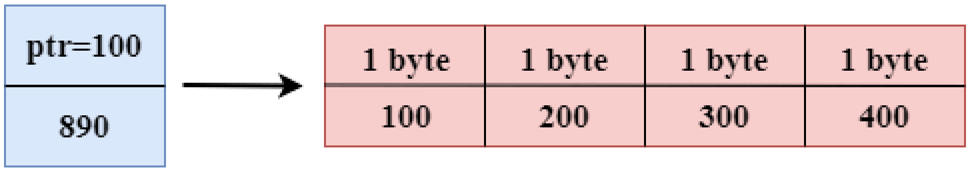
Let's understand the dangling pointer through some C programs.

Using free() function to de-allocate the memory.

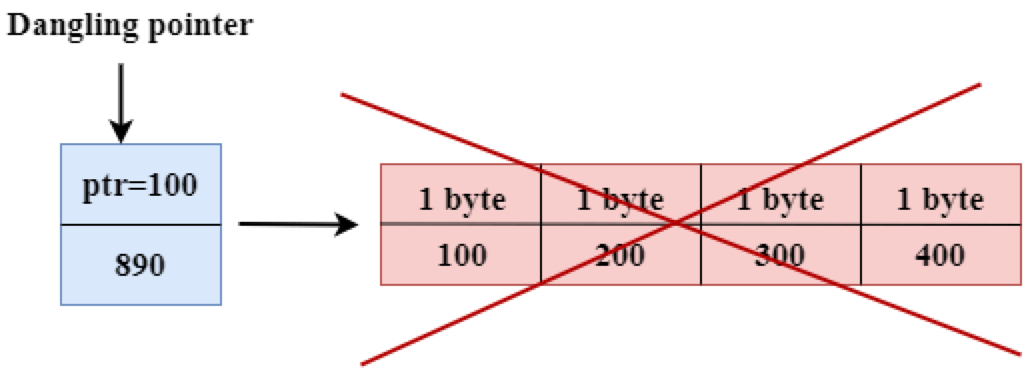
```
#include <stdio.h>
int main()
{
  int *ptr=(int *)malloc(sizeof(int));
  int a=560;
  ptr=&a;
  free(ptr);
  return 0;
}
```

In the above code, we have created two variables, i.e., *ptr and a where 'ptr' is a pointer and 'a' is a integer variable. The *ptr is a pointer variable which is created with the help of **malloc()** function. As we know that malloc() function returns void, so we use int * to convert void pointer into int pointer.

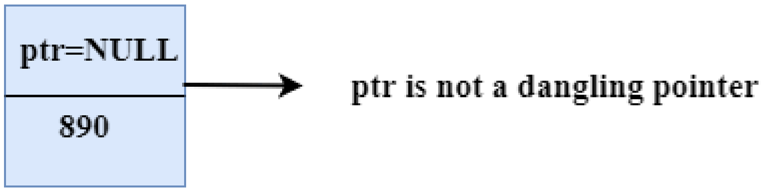
The statement **int *ptr=(int *)malloc(sizeof(int));** will allocate the memory with 4 bytes shown in the below image:



The statement **free(ptr)** de-allocates the memory as shown in the below image with a cross sign, and 'ptr' pointer becomes dangling as it is pointing to the de-allocated memory.



If we assign the NULL value to the 'ptr', then 'ptr' will not point to the deleted memory. Therefore, we can say that ptr is not a dangling pointer, as shown in the below image:



Variable goes out of the scope

When the variable goes out of the scope then the pointer pointing to the variable becomes a **dangling pointer.**

```
#include<stdio.h>
int main()
{
   char *str;
   {
       char a = ?A?;
       str = &a;
   }
   // a falls out of scope
   // str is now a dangling pointer
   printf("%s", *str);
}
```

In the above code, we did the following steps:

- First, we declare the pointer variable named 'str'.
- In the inner scope, we declare a character variable. The str pointer contains the address of the variable 'a'.
- When the control comes out of the inner scope, 'a' variable will no longer be available, so str points to the de-allocated memory. It means that the str pointer becomes the dangling pointer.

Function call

Now, we will see how the pointer becomes dangling when we call the function.

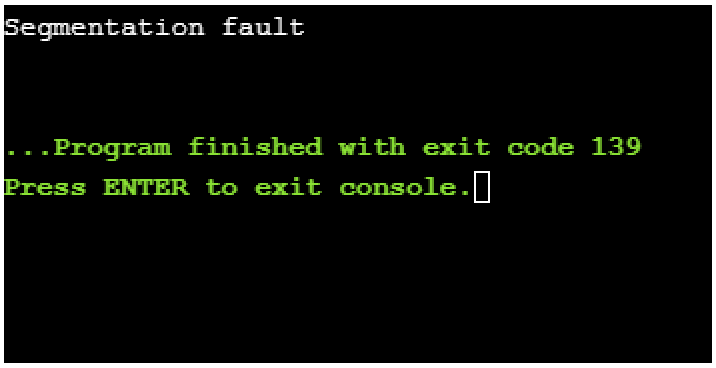
Let's understand through an example.

```
 #include <stdio.h>
  int *fun(){
  int y=10;
  return &y;
 }
int main()
{
int *p=fun();
printf("%d", *p);
return 0;
}
```

In the above code, we did the following steps:

- First, we create the **main()** function in which we have declared **'p'** pointer that contains the return value of the **fun()**.
- When the **fun()** is called, then the control moves to the context of the **int *fun(),** the **fun()** returns the address of the 'y' variable.
- When control comes back to the context of the **main()** function, it means the variable **'y'** is no longer available. Therefore, we can say that the **'p'** pointer is a dangling pointer as it points to the de-allocated memory.

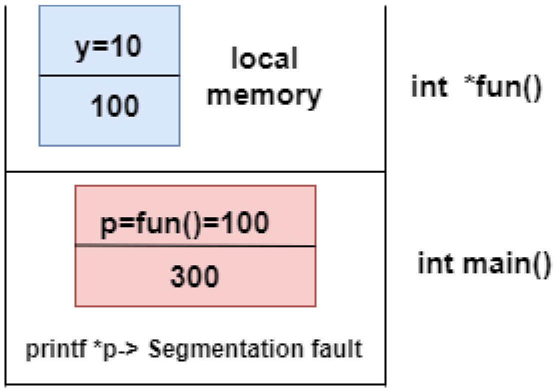
Output



```
Segmentation fault


...Program finished with exit code 139
Press ENTER to exit console.
```

Let's represent the working of the above code diagrammatically.

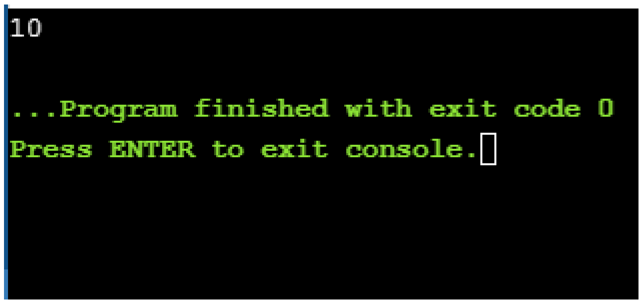


Let's consider another example of a dangling pointer.

```
#include <stdio.h>
int *fun()
{
    static int y=10;
    return &y;
}
int main()
{
  int *p=fun();
  printf("%d", *p);
   return 0;
}
```

The above code is similar to the previous one but the only difference is that the variable 'y' is static. We know that static variable stores in the global memory.

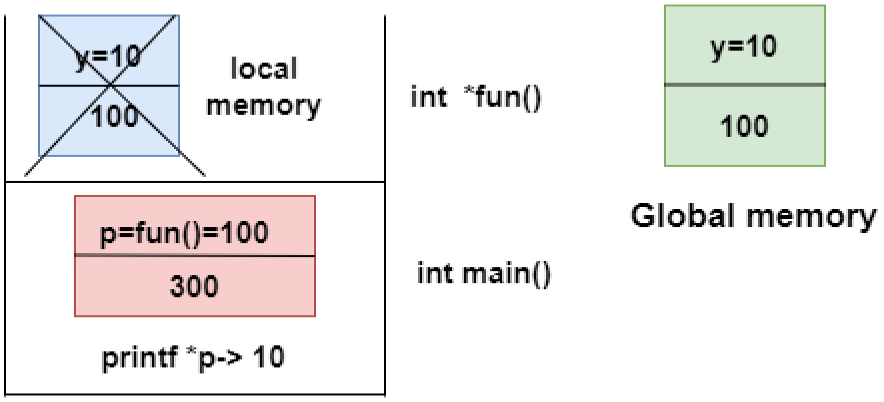
Output



```
10

...Program finished with exit code 0
Press ENTER to exit console.
```

Now, we represent the working of the above code diagrammatically.



The above diagram shows the stack memory. First, **the fun()** function is called, then the control moves to the context of the **int *fun().** As 'y' is a static variable, so it stores in the global memory; Its scope is available throughout the program. When the address value is returned, then the control comes back to the context of the **main().** The pointer 'p' contains the address of 'y', i.e., 100. When we print the value of '*p', then it prints the value of 'y', i.e., 10. Therefore, we can say that the pointer 'p' is not a dangling pointer as it contains the address of the variable which is stored in the global memory.

Avoiding Dangling Pointer Errors

The dangling pointer errors can be avoided by initializing the pointer to the **NULL** value. If we assign the **NULL** value to the pointer, then the pointer will not point to the de-allocated memory. Assigning **NULL** value to the pointer means that the pointer is not pointing to any memory location.

SIZEOF() OPERATOR

The **sizeof()** operator is commonly used in C. It determines the size of the expression or the data type specified in the number of char-sized storage units. The **sizeof()** operator contains a single operand which can be either an expression or a data typecast where the cast is data type enclosed within parenthesis. The data type cannot only be primitive data types such as integer or floating data types, but it can also be pointer data types and compound data types such as unions and structs.

Need of sizeof() operator

Mainly, programs know the storage size of the primitive data types. Though the storage size of the data type is constant, it varies when implemented in different platforms. For example, we dynamically allocate the array space by using **sizeof()** operator:

```
int *ptr = malloc(10*sizeof(int));
```

In the above example, we use the sizeof() operator, which is applied to the cast of type int. We use **malloc()** function to allocate the memory and returns the pointer which is pointing to this allocated memory. The memory space is equal to the number of bytes occupied by the int data type and multiplied by 10.

The **sizeof()** operator behaves differently according to the type of the operand.

- **Operand is a data type**
- **Operand is an expression**

When operand is a data type.

```
    #include <stdio.h>
    int main()
    {
       int x=89;    // variable declaration.
printf("size of the variable x is %d", sizeof(x));  // Displaying the size of ?x? variable.
printf("\nsize of the integer data type is %d",sizeof(int)); //Displaying the size of integer
data type.
printf("\nsize of the character data type is %d",sizeof(char)); //Displaying the size of char
acter data type.
printf("\nsize of the floating data type is %d",sizeof(float)); //Displaying the size of floati
ng data type.
    return 0;
    }
```

In the above code, we are printing the size of different data types such as int, char, float with the help of **sizeof()** operator.

Output: 4 4 1 4

When operand is an expression

```
• #include <stdio.h>
• int main()
• {
•   double i=78.0; //variable initialization.
•   float j=6.78; //variable initialization.
•   printf("size of (i+j) expression is : %d",sizeof(i+j)); //Displaying the size of the e
  xpression (i+j).
•   return 0;
• }
```

In the above code, we have created two variables 'i' and 'j' of type double and float respectively, and then we print the size of the expression by using **sizeof(i+j)** operator.

Output

```
size of (i+j) expression is : 8
```

CONST POINTER IN C

10. Constant Pointers

A constant pointer in C cannot change the address of the variable to which it is pointing, i.e., the address will remain constant. Therefore, we can say that if a constant pointer is pointing to some variable, then it cannot point to any other variable.

11. Syntax of Constant Pointer

```
1. <type of pointer> *const <name of pointer>;
```

Declaration of a constant pointer is given below:

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

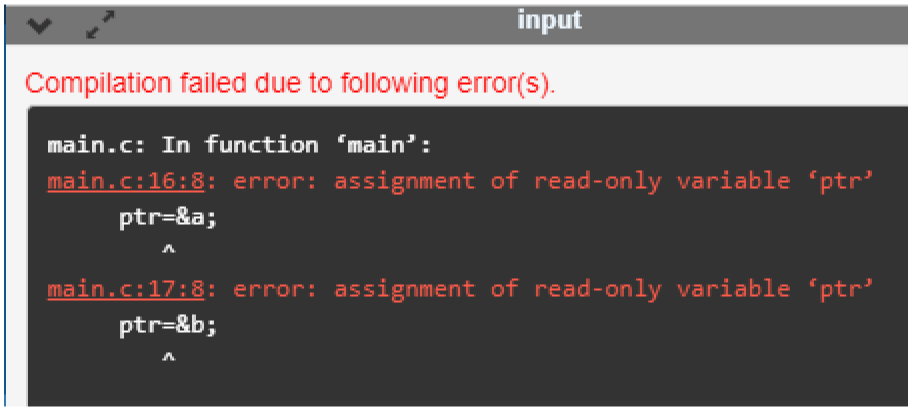
Let's understand the constant pointer through an example.

```c
#include <stdio.h>
int main()
{
    int a=1;
    int b=2;
    int *const ptr;
    ptr=&a;
    ptr=&b;
    printf("Value of ptr is :%d",*ptr);
    return 0;
}
```

In the above code:

- We declare two variables, i.e., a and b with values 1 and 2, respectively.
- We declare a constant pointer.
- First, we assign the address of variable 'a' to the pointer 'ptr'.
- Then, we assign the address of variable 'b' to the pointer 'ptr'.
- Lastly, we try to print the value of the variable pointed by the 'ptr'.

Output



```
Compilation failed due to following error(s).
main.c: In function 'main':
main.c:16:8: error: assignment of read-only variable 'ptr'
     ptr=&a;
        ^
main.c:17:8: error: assignment of read-only variable 'ptr'
     ptr=&b;
        ^
```

In the above output, we can observe that the above code produces the error "assignment of read-only variable 'ptr'". It means that the value of the variable 'ptr' which 'ptr' is holding cannot be changed. In the above code, we are changing the value of 'ptr' from &a to &b, which is not possible with constant pointers. Therefore, we can say that the constant pointer, which points to some variable, cannot point to another variable.

12. Pointer to Constant

A pointer to constant is a pointer through which the value of the variable that the pointer points cannot be changed. The address of these pointers can be changed, but the value of the variable that the pointer points cannot be changed.

13. Syntax of Pointer to Constant

```
const <type of pointer>* <name of pointer>
```

Declaration of a pointer to constant is given below:

```
const int* ptr;
```

Let's understand through an example.

- **First, we write the code where we are changing the value of a pointer**

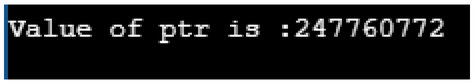
```
#include <stdio.h>
int main()
{
   int a=100;
   int b=200;
   const int* ptr;
   ptr=&a;
```

```
    ptr=&b;
    printf("Value of ptr is :%u",ptr);
    return 0;
}
```

In the above code:

- We declare two variables, i.e., a and b with the values 100 and 200 respectively.
- We declare a pointer to constant.
- First, we assign the address of variable 'a' to the pointer 'ptr'.
- Then, we assign the address of variable 'b' to the pointer 'ptr'.
- Lastly, we try to print the value of 'ptr'.

Output



```
Value of ptr is :247760772
```

The above code runs successfully, and it shows the value of 'ptr' in the output.

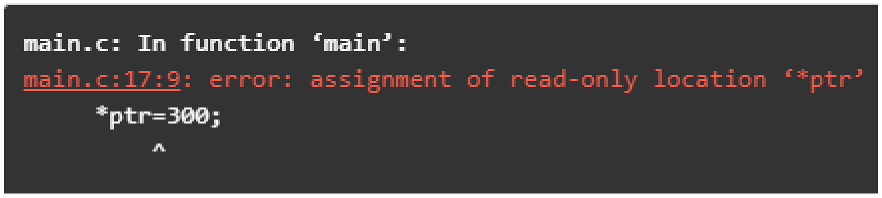
- Now, we write the code in which we are changing the value of the variable to which the pointer points.

```
#include <stdio.h>
int main()
{
    int a=100;
    int b=200;
    const int* ptr;
    ptr=&b;
    *ptr=300;
    printf("Value of ptr is :%d",*ptr);
    return 0;
}
```

In the above code:

- We declare two variables, i.e., 'a' and 'b' with the values 100 and 200 respectively.
- We declare a pointer to constant.
- We assign the address of the variable 'b' to the pointer 'ptr'.
- Then, we try to modify the value of the variable 'b' through the pointer 'ptr'.
- Lastly, we try to print the value of the variable which is pointed by the pointer 'ptr'.

Output



```
main.c: In function ‘main’:
main.c:17:9: error: assignment of read-only location ‘*ptr’
     *ptr=300;
         ^
```

The above code shows the error "assignment of read-only location '*ptr'". This error means that we cannot change the value of the variable to which the pointer is pointing.

14. Constant Pointer to a Constant

A constant pointer to a constant is a pointer, which is a combination of the above two pointers. It can neither change the address of the variable to which it is pointing nor it can change the value placed at this address.

15. Syntax

```
const <type of pointer>* const <name of the pointer>;
```

Declaration for a constant pointer to a constant is given below:

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

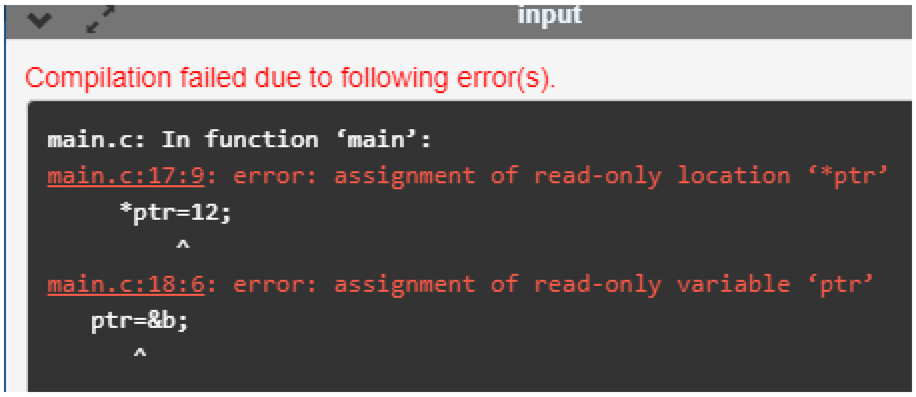
Let's understand through an example.

```
#include <stdio.h>
int main()
{
    int a=10;
    int b=90;
    const int* const ptr=&a;
   *ptr=12;
    ptr=&b;
    printf("Value of ptr is :%d",*ptr);
    return 0;
}
```

In the above code:

- We declare two variables, i.e., 'a' and 'b' with the values 10 and 90, respectively.
- We declare a constant pointer to a constant and then assign the address of 'a'.
- We try to change the value of the variable 'a' through the pointer 'ptr'.
- Then we try to assign the address of variable 'b' to the pointer 'ptr'.
- Lastly, we print the value of the variable, which is pointed by the pointer 'ptr'.

Output



Compilation failed due to following error(s).

```
main.c: In function ‘main’:
main.c:17:9: error: assignment of read-only location ‘*ptr’
     *ptr=12;
         ^
main.c:18:6: error: assignment of read-only variable ‘ptr’
   ptr=&b;
      ^
```

The above code shows the error "assignment of read-only location '*ptr'" and "assignment of read-only variable 'ptr'". Therefore, we conclude that the constant pointer to a constant can change neither address nor value, which is pointing by this pointer.

VOID POINTER IN C

Till now, we have studied that the address assigned to a pointer should be of the same type as specified in the pointer declaration. For example, if we declare the int pointer, then this int pointer cannot point to the float variable or some other type of variable, i.e., it can point to only int type variable. To overcome this problem, we use a pointer to void. A pointer to void means a generic pointer that can point to any data type. We can assign the address of any data type to the void pointer, and a void pointer can be assigned to any type of the pointer without performing any explicit typecasting.

16. Syntax of void pointer

```c
void *pointer name;
```

Declaration of the void pointer is given below:

```c
void *ptr;
```

In the above declaration, the void is the type of the pointer, and 'ptr' is the name of the pointer.

Let us consider some examples:

```c
int i=9;        // integer variable initialization.

int *p;        // integer pointer declaration.

float *fp;        // floating pointer declaration.

void *ptr;        // void pointer declaration.

p=fp;        // incorrect.

fp=&i;        // incorrect

ptr=p;        // correct

ptr=fp;        // correct

ptr=&i;        // correct
```

17. Size of the void pointer in C

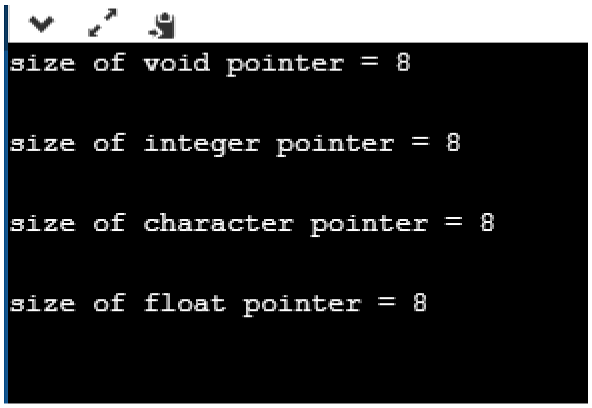
The size of the void pointer in C is the same as the size of the pointer of character type. According to C perception, the representation of a pointer to void is the same as the pointer of character type. The size of the pointer will vary depending on the platform that you are using.

Let's look at the below example:

```c
#include <stdio.h>
int main()
{
   void *ptr = NULL; //void pointer
```

```
    int *p  = NULL;// integer pointer
    char *cp = NULL;//character pointer
    float *fp = NULL;//float pointer
    //size of void pointer
    printf("size of void pointer = %d\n\n",sizeof(ptr));
    //size of integer pointer
    printf("size of integer pointer = %d\n\n",sizeof(p));
    //size of character pointer
    printf("size of character pointer = %d\n\n",sizeof(cp));
    //size of float pointer
    printf("size of float pointer = %d\n\n",sizeof(fp));
    return 0;
}
```

Output



```
size of void pointer = 8

size of integer pointer = 8

size of character pointer = 8

size of float pointer = 8
```

18. Advantages of void pointer

Following are the advantages of a void pointer:

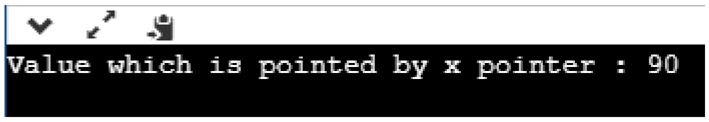
- The malloc() and calloc() function return the void pointer, so these functions can be used to allocate the memory of any data type.

```
#include <stdio.h>
#include<malloc.h>
int main()
{
   int a=90;
```

```

    int *x = (int*)malloc(sizeof(int)) ;
    x=&a;
    printf("Value which is pointed by x pointer : %d",*x);
     return 0;
}
```

Output



```
Value which is pointed by x pointer : 90
```

- The void pointer in C can also be used to implement the generic functions in C.

Some important points related to void pointer are:

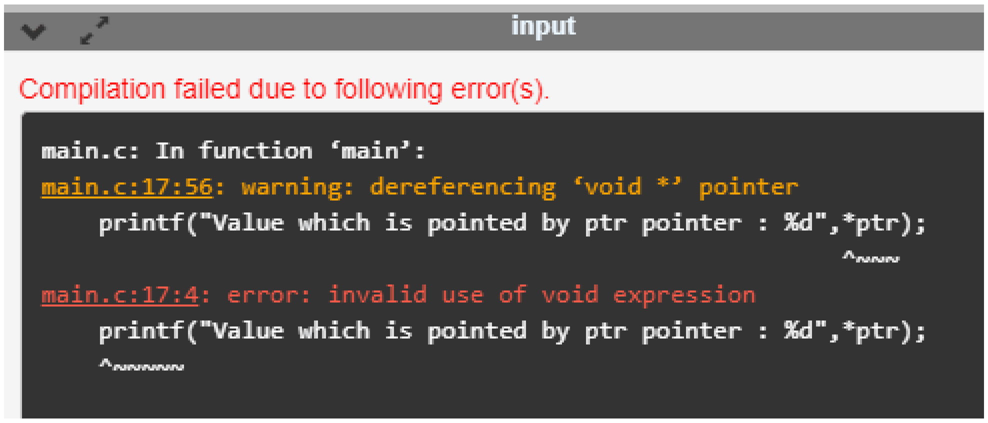
- **Dereferencing a void pointer in C**

The void pointer in C cannot be dereferenced directly. Let's see the below example.

```
#include <stdio.h>
int main()
{
   int a=90;
   void *ptr;
   ptr=&a;
   printf("Value which is pointed by ptr pointer : %d",*ptr);
   return 0;
}
```

In the above code, *ptr is a void pointer which is pointing to the integer variable 'a'. As we already know that the void pointer cannot be dereferenced, so the above code will give the compile-time error because we are printing the value of the variable pointed by the pointer 'ptr' directly.

Output



Compilation failed due to following error(s).

```
main.c: In function ‘main’:
main.c:17:56: warning: dereferencing ‘void *’ pointer
    printf("Value which is pointed by ptr pointer : %d",*ptr);
                                                        ^~~~
main.c:17:4: error: invalid use of void expression
    printf("Value which is pointed by ptr pointer : %d",*ptr);
    ^~~~~~
```

Now, we rewrite the above code to remove the error.

```
#include <stdio.h>
int main()
{
  int a=90;
  void *ptr;
  ptr=&a;
  printf("Value which is pointed by ptr pointer : %d",*(int*)ptr);
   return 0;
}
```

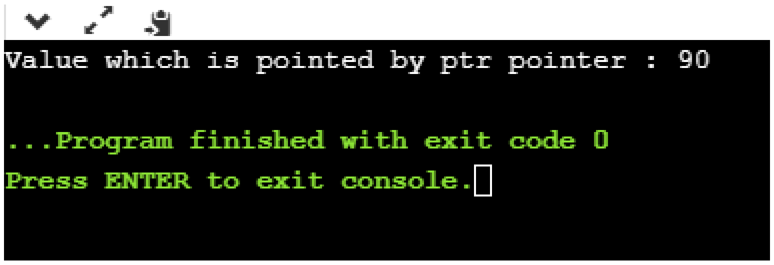
In the above code, we typecast the void pointer to the integer pointer by using the statement given below:

(int*)ptr;

Then, we print the value of the variable which is pointed by the void pointer 'ptr' by using the statement given below:

***(int*)ptr;**

Output



```
Value which is pointed by ptr pointer : 90

...Program finished with exit code 0
Press ENTER to exit console.
```

- **Arithmetic operation on void pointers**

We cannot apply the arithmetic operations on void pointers in C directly. We need to apply the proper typecasting so that we can perform the arithmetic operations on the void pointers.

Let's see the below example:

```
#include<stdio.h>
int main()
{
  float a[4]={6.1,2.3,7.8,9.0};
  void *ptr;
  ptr=a;
  for(int i=0;i<4;i++)
 {
    printf("%f,",*ptr);
    ptr=ptr+1;        // Incorrect.

}}
```

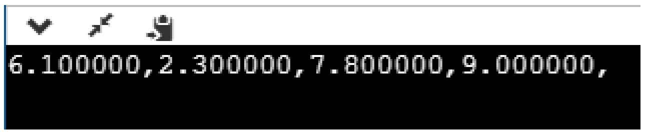
The above code shows the compile-time error that "**invalid use of void expression**" as we cannot apply the arithmetic operations on void pointer directly, i.e., ptr=ptr+1.

Let's rewrite the above code to remove the error.

```
#include<stdio.h>
int main()
{
  float a[4]={6.1,2.3,7.8,9.0};
  void *ptr;
  ptr=a;
  for(int i=0;i<4;i++)
 {
    printf("%f,",*((float*)ptr+i));
  }}
```

The above code runs successfully as we applied the proper casting to the void pointer, i.e., (float*)ptr and then we apply the arithmetic operation, i.e., *((float*)ptr+i).

Output



```
6.100000,2.300000,7.800000,9.000000,
```

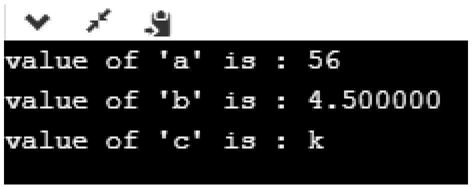
19. Why we use void pointers?

We use void pointers because of its reusability. Void pointers can store the object of any type, and we can retrieve the object of any type by using the indirection operator with proper typecasting.

Let's understand through an example.

```
#include<stdio.h>
int main()
{
  int a=56; // initialization of a integer variable 'a'.
  float b=4.5; // initialization of a float variable 'b'.
  char c='k'; // initialization of a char variable 'c'.
   void *ptr; // declaration of void pointer.
   // assigning the address of variable 'a'.
   ptr=&a;
   printf("value of 'a' is : %d",*((int*)ptr));
   // assigning the address of variable 'b'.
   ptr=&b;
   printf("\nvalue of 'b' is : %f",*((float*)ptr));
   // assigning the address of variable 'c'.
   ptr=&c;
    printf("\nvalue of 'c' is : %c",*((char*)ptr));
    return 0;
}
```

Output



```
value of 'a' is : 56
value of 'b' is : 4.500000
value of 'c' is : k
```

C DEREFERENCE POINTER

As we already know that **"what is a pointer"**, a pointer is a variable that stores the address of another variable. The dereference operator is also known as an indirection operator, which is represented by (*). When indirection operator (*) is used with the pointer variable, then it is known as **dereferencing a pointer.** When we dereference a pointer, then the value of the variable pointed by this pointer will be returned.

Why we use dereferencing pointer?

Dereference a pointer is used because of the following reasons:

- It can be used to access or manipulate the data stored at the memory location, which is pointed by the pointer.
- Any operation applied to the dereferenced pointer will directly affect the value of the variable that it points to.

Let's observe the following steps to dereference a pointer.

- First, we declare the integer variable to which the pointer points.

```
1. int x =9;
```

- Now, we declare the integer pointer variable.

```
1. int *ptr;
```

- After the declaration of an integer pointer variable, we store the address of 'x' variable to the pointer variable 'ptr'.

```
1. ptr=&x;
```

- We can change the value of 'x' variable by dereferencing a pointer 'ptr' as given below:

```
1. *ptr =8;
```

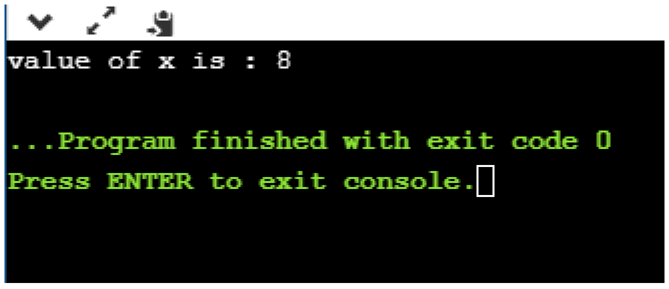
The above line changes the value of 'x' variable from 9 to 8 because 'ptr' points to the 'x' location and dereferencing of 'ptr', i.e., *ptr=8 will update the value of x.

Backward Skip 10sPlay VideoForward Skip 10s

Let's combine all the above steps:

```c
#include <stdio.h>
int main()
{
    int x=9;
    int *ptr;
    ptr=&x;
    *ptr=8;
    printf("value of x is : %d", x);
    return 0;}
```

Output



```
value of x is : 8

...Program finished with exit code 0
Press ENTER to exit console.
```

Let's consider another example.

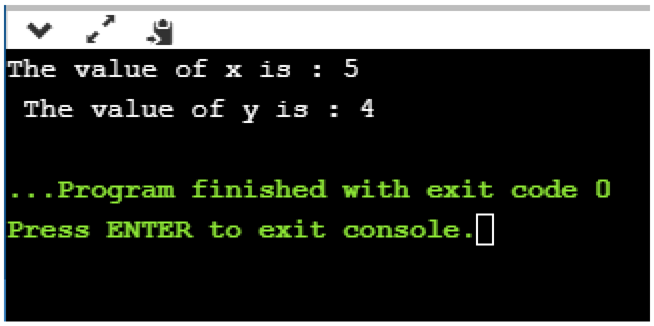
```c
#include <stdio.h>
int main()
{
    int x=4;
    int y;
    int *ptr;
    ptr=&x;
    y=*ptr;
    *ptr=5;
    printf("The value of x is : %d",x);
    printf("\n The value of y is : %d",y);
    return 0;
}
```

In the above code:

- We declare two variables 'x' and 'y' where 'x' is holding a '4' value.
- We declare a pointer variable 'ptr'.
- After the declaration of a pointer variable, we assign the address of the 'x' variable to the pointer 'ptr'.
- As we know that the 'ptr' contains the address of 'x' variable, so '*ptr' is the same as 'x'.
- We assign the value of 'x' to 'y' with the help of 'ptr' variable, i.e., y=***ptr** instead of using the 'x' variable.

> NOTE: ACCORDING TO US, IF WE CHANGE THE VALUE OF 'X', THEN THE VALUE OF 'Y' WILL ALSO GET CHANGED AS THE POINTER 'PTR' HOLDS THE ADDRESS OF THE 'X' VARIABLE. BUT THIS DOES NOT HAPPEN, AS 'Y' IS STORING THE LOCAL COPY OF VALUE '5'.

Output



```
The value of x is : 5
 The value of y is : 4

...Program finished with exit code 0
Press ENTER to exit console.
```

Let's consider another scenario.

```
#include <stdio.h>
int main()
{
   int a=90;
```

```
    int *ptr1,*ptr2;
    ptr1=&a;
    ptr2=&a;
    *ptr1=7;
    *ptr2=6;
     printf("The value of a is : %d",a);
     return 0;
}
```

In the above code:

- First, we declare an 'a' variable.
- Then we declare two pointers, i.e., ptr1 and ptr2.
- Both the pointers contain the address of 'a' variable.
- We assign the '7' value to the *ptr1 and '6' to the *ptr2. The final value of 'a' would be '6'.

> NOTE: IF WE HAVE MORE THAN ONE POINTER POINTING TO THE SAME LOCATION, THEN THE CHANGE MADE BY ONE POINTER WILL BE THE SAME AS ANOTHER POINTER.

DYNAMIC MEMORY ALLOCATION IN C

The concept of **dynamic memory allocation in c language** *enables the C programmer to allocate memory at runtime*. Dynamic memory allocation in c language is possible by 4 functions of stdlib.h header file.

static memory allocation	dynamic memory allocation
memory is allocated at compile time.	memory is allocated at run time.
memory can't be increased while executing program.	memory can be increased while executing program.
used in array.	used in linked list.

Now let's have a quick look at the methods used for dynamic memory allocation.

malloc()	allocates single block of requested memory.
calloc()	allocates multiple block of requested memory.
realloc()	reallocates the memory occupied by malloc() or calloc() functions.
free()	frees the dynamically allocated memory.

Malloc() function in C

The malloc() function allocates single block of requested memory. It doesn't initialize memory at execution time, so it has garbage value initially. It returns NULL if memory is not sufficient.

The syntax of malloc() function is given below:

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

```
#include<stdio.h>
#include<stdlib.h>
int main(){
 int n, i, *ptr, sum=0;
  printf("Enter number of elements: ");
  scanf("%d", &n);
  ptr=(int*)malloc(n*sizeof(int));  //memory allocated using malloc
  if(ptr==NULL)
  {
    printf("Sorry! unable to allocate memory");
    exit(0);
  }
  printf("Enter elements of array: ");
  for(i=0; i<n; ++i)
  {
    scanf("%d", ptr+i);
    sum+=*(ptr+i);
  }
•      printf("Sum=%d", sum);
•      free(ptr);
•  return 0;
•  }
```

Output

```
Enter elements of array: 3
Enter elements of array: 10 10   10
Sum=30
```

Calloc() function in C

The calloc() function allocates multiple block of requested memory. It initially initialize all bytes to zero. It returns NULL if memory is not sufficient.

The syntax of calloc() function is given below:

ptr=(cast-type*)calloc(number, byte-size)

Let's see the example of calloc() function.

```
•  #include<stdio.h>
•  #include<stdlib.h>
•  int main(){
•   int n, i,*ptr, sum=0;
•      printf("Enter number of elements: ");
•      scanf("%d", &n);
•      ptr=(int*)calloc(n,sizeof(int));  //memory allocated using calloc
•      if(ptr==NULL)
•      {
•          printf("Sorry! unable to allocate memory");
•          exit(0);
•      }
•      printf("Enter elements of array: ");
•      for(i=0; i<n; ++i)
•      {
•          scanf("%d", ptr+i);
```

```
            sum+=*(ptr+i);
        }
        printf("Sum=%d", sum);
        free(ptr);
    return 0;
    }
```

Output

```
Enter elements of array: 3
Enter elements of array: 10 10 10
Sum=30
```

Realloc() function in C

If memory is not sufficient for malloc() or calloc(), you can reallocate the memory by realloc() function. In short, it changes the memory size.

Let's see the syntax of realloc() function.

```
ptr = realloc(ptr, new-size)
```

free() function in C

The memory occupied by malloc() or calloc() functions must be released by calling free() function. Otherwise, it will consume memory until program exit.

Let's see the syntax of free() function.

```
free(ptr)
```

C STRINGS

The string can be defined as the one-dimensional array of characters terminated by a null ('\0'). The character array or the string is used to manipulate text such as word or sentences. Each character in the array occupies one byte of memory, and the last character must always be 0. The termination character ('\0') is important in a string since it is the only way to identify where the string ends. When we define a string as char s[10], the character s[10] is implicitly initialized with the null in the memory.

There are two ways to declare a string in c language.

1. By char array
2. By string literal

Let's see the example of declaring **string by char array** in C language.

```
char ch[6]={'i', 'n', 'd', 'i', 'a', '\0'};
```

While declaring string, size is not mandatory. So we can write the above code as given below:

```
char ch[]={'i', 'n', 'd', 'i', 'a', '\0'};
```

We can also define the **string by the string literal** in C language. For example:

```
char ch[]="india";
```

In such case, '\0' will be appended at the end of the string by the compiler.

Difference between char array and string literal

There are two main differences between char array and literal.

- We need to add the null character '\0' at the end of the array by ourself whereas, it is appended internally by the compiler in the case of the character array.
- The string literal cannot be reassigned to another set of characters whereas, we can reassign the characters of the array.

String Example in C

A simple example where a string is declared and being printed. The '%s' is used as a format specifier for the string in c language.

```
#include<stdio.h>
#include <string.h>
int main(){
  char ch[6]={‘I’,’n’,’d’,’i’,’a’, '\0'};
   char ch2[6]="India";
   printf("Char Array Value is: %s\n", ch);
   printf("String Literal Value is: %s\n", ch2);
 return 0;
}
```

Output

```
Char Array Value is: India
String Literal Value is: India
```

Traversing String

Traversing the string is one of the most important aspects in any of the programming languages. We may need to manipulate a very large text which can be done by traversing the text. Traversing string is somewhat different from the traversing an integer array. We need to know the length of the array to traverse an integer array, whereas we may use the null character in the case of string to identify the end the string and terminate the loop.

Hence, there are two ways to traverse a string.

- By using the length of string
- By using the null character.

Let's discuss each one of them.

Using the length of string

Let's see an example of counting the number of vowels in a string.

```c
#include<stdio.h>
void main ()
{
    char s[11] = "javatpoint";
    int i = 0;
    int count = 0;
    while(i<11)
    {
        if(s[i]=='a' || s[i] == 'e' || s[i] == 'i' || s[i] == 'u' || s[i] == 'o')
        {
            count ++;
        }
        i++;
    }
    printf("The number of vowels %d",count);
}
```

Output: The number of vowels 4

Using the null character

Let's see the same example of counting the number of vowels by using the null character.

```c
#include<stdio.h>
void main ()
{
    char s[11] = "javatpoint";
    int i = 0;
    int count = 0;
    while(s[i] != NULL)
    {
        if(s[i]=='a' || s[i] == 'e' || s[i] == 'i' || s[i] == 'u' || s[i] == 'o')
        {
            count ++;
        }
        i++;
    }
    printf("The number of vowels %d",count);
}
```

Output: The number of vowels 4

Accepting string as the input

Till now, we have used scanf to accept the input from the user. However, it can also be used in the case of strings but with a different scenario. Consider the below code which stores the string while space is encountered.

```c
#include<stdio.h>
void main ()
{
    char s[20];
    printf("Enter the string?");
    scanf("%s",s);
    printf("You entered %s",s);
}
```

Output

```
Enter the string? C programming is the best
You enter c programming
```

It is clear from the output that, the above code will not work for space separated strings. To make this code working for the space separated strings, the minor changed required in the scanf function, i.e., instead of writing scanf("%s",s), we must write: scanf("%[^\n]s",s) which instructs the compiler to store the string s while the new line (\n) is encountered. Let's consider the following example to store the space-separated strings.

```
#include<stdio.h>
void main ()
{
    char s[20];
    printf("Enter the string?");
    scanf("%[^\n]s",s);
    printf("You entered %s",s);
}
```

Output

```
Enter the string? india is the best
You entered india is the best
```

Here we must also notice that we do not need to use address of (&) operator in scanf to store a string since string s is an array of characters and the name of the array, i.e., s indicates the base address of the string (character array) therefore we need not use & with it.

Some important points

However, there are the following points which must be noticed while entering the strings by using scanf.

- The compiler doesn't perform bounds checking on the character array. Hence, there can be a case where the length of the string can exceed the dimension of the character array which may always overwrite some important data.
- Instead of using scanf, we may use gets() which is an inbuilt function defined in a header file string.h. The gets() is capable of receiving only one string at a time.

Pointers with strings

We have used pointers with the array, functions, and primitive data types so far. However, pointers can be used to point to the strings. There are various advantages of using pointers to point strings. Let us consider the following example to access the string via the pointer.

```
#include<stdio.h>
void main ()
{
    char s[6] = "india";
    char *p = s; // pointer p is pointing to string s.
    printf("%s",p); // the string india is printed if we print p.
}
```

Output: india

As we know that string is an array of characters, the pointers can be used in the same way they were used with arrays. In the above example, p is declared as a pointer to the array of characters s. P affects similar to s since s is the base address of the string and treated as a pointer internally. However, we can not change the content of s or copy the content of s into another string directly. For this purpose, we need to use the pointers to store the strings. In the following example, we have shown the use of pointers to copy the content of a string into another.

```
#include<stdio.h>
void main ()
{
    char *p = "hello";
    printf("String p: %s\n",p);
    char *q;
    printf("copying the content of p into q...\n");
    q = p;
    printf("String q: %s\n",q);
}
```

Output

```
String p: hello
copying the content of p into q...
String q: hello
```

Once a string is defined, it cannot be reassigned to another set of characters. However, using pointers, we can assign the set of characters to the string. Consider the following example.

```
#include<stdio.h>
void main ()
{
    char *p = "hello c";
    printf("Before assigning: %s\n",p);
    p = "hello";
    printf("After assigning: %s\n",p);
}
```

Output

```
Before assigning: hello c
After assigning: hello
```

GETS() AND PUTS() FUNCTIONS

The gets() and puts() are declared in the header file stdio.h. Both the functions are involved in the input/output operations of the strings.

gets() function

The gets() function enables the user to enter some characters followed by the enter key. All the characters entered by the user get stored in a character array. The null character is added to the array to make it a string. The gets() allows the user to enter the space-separated strings. It returns the string entered by the user.

Declaration

```
char[] gets(char[]);
```

READING STRING USING GETS()

```
#include<stdio.h>
void main ()
{
    char s[30];
    printf("Enter the string? ");
    gets(s);
```

- printf("You entered %s",s);
- }

OUTPUT

```
Enter the string?
jndia is the best
You entered india is the best
```

The gets() function is risky to use since it doesn't perform any array bound checking and keep reading the characters until the new line (enter) is encountered. It suffers from buffer overflow, which can be avoided by using fgets(). The fgets() makes sure that not more than the maximum limit of characters are read. Consider the following example.

- #include<stdio.h>
- void main()
- {
-   char str[20];
-   printf("Enter the string? ");
-   fgets(str, 20, stdin);
-   printf("%s", str);
- }

OUTPUT

```
Enter the string? india is the best website
india is the best
```

puts() function

The puts() function is very much similar to printf() function. The puts() function is used to print the string on the console which is previously read by using gets() or scanf() function. The puts() function returns an integer value representing the number of characters being printed on the console. Since, it prints an additional newline character with the string, which moves the cursor to the new line on the console, the integer value returned by puts() will always be equal to the number of characters present in the string plus 1.

Declaration

```
int puts(char[])
```

Let's see an example to read a string using gets() and print it on the console using puts().

- #include<stdio.h>

```
#include <string.h>
int main(){
char name[50];
printf("Enter your name: ");
gets(name); //reads string from user
printf("Your name is: ");
puts(name);  //displays string
return 0;
}
```

OUTPUT:

```
Enter your name: Sonoo Jaiswal
Your name is: Sonoo Jaiswal
```

C STRING FUNCTIONS

There are many important string functions defined in "string.h" library.

No.	Function	Description
1)	strlen(string_name)	returns the length of string name.
2)	strcpy(destination, source)	copies the contents of source string to destination string.
3)	strcat(first_string, second_string)	concats or joins first string with second string. The result of the string is stored in first string.
4)	strcmp(first_string, second_string)	compares the first string with second string. If both strings are same, it returns 0.
5)	strrev(string)	returns reverse string.
6)	strlwr(string)	returns string characters in lowercase.
7)	strupr(string)	returns string characters in uppercase.

STRING LENGTH: STRLEN() FUNCTION

The strlen() function returns the length of the given string. It doesn't count null character '\0'.

```
#include<stdio.h>
#include <string.h>
int main(){
char ch[20]={'j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't', '\0'};
   printf("Length of string is: %d",strlen(ch));
 return 0;
}
```

Output: Length of string is: 10

C COPY STRING: STRCPY()

The strcpy(destination, source) function copies the source string in destination.

```
#include<stdio.h>
#include <string.h>
int main(){
 char ch[20]={'i', 'n', 'd', 'i', 'a', 'p', 'o', 'i', 'n', 't', '\0'};
   char ch2[20];
   strcpy(ch2,ch);
   printf("Value of second string is: %s",ch2);
 return 0;
}
```

Output: Value of second string is: indiapoint

C STRING CONCATENATION: STRCAT()

The strcat(first_string, second_string) function concatenates two strings and result is returned to first_string.

```
#include<stdio.h>
#include <string.h>
int main(){
char ch[10]={'h', 'e', 'l', 'l', 'o', '\0'};
char ch2[10]={'c', '\0'};
strcat(ch,ch2);
printf("Value of first string is: %s",ch);
return 0;
}
```

Output: Value of first string is: helloc

C COMPARE STRING: STRCMP()

The strcmp(first_string, second_string) function compares two string and returns 0 if both strings are equal.

Here, we are using *gets()* function which reads string from the console.

```
#include<stdio.h>
#include <string.h>
int main(){
  char str1[20],str2[20];
  printf("Enter 1st string: ");
  gets(str1);//reads string from console
  printf("Enter 2nd string: ");
  gets(str2);
  if(strcmp(str1,str2)==0)
      printf("Strings are equal");
  else
      printf("Strings are not equal");
 return 0;
}
```

Output:

```
Enter 1st string: hello
Enter 2nd string: hello
Strings are equal
```

C REVERSE STRING: STRREV()

The strrev(string) function returns reverse of the given string. Let's see a simple example of strrev() function.

```
#include<stdio.h>
#include <string.h>
int main(){
  char str[20];
  printf("Enter string: ");
  gets(str);//reads string from console
  printf("String is: %s",str);
  printf("\nReverse String is: %s",strrev(str));
 return 0;
}
```

Output:

```
Enter string: indian
String is: indian
Reverse String is: naidni
```

C STRING LOWERCASE: STRLWR()

The strlwr(string) function returns string characters in lowercase. Let's see a simple example of strlwr() function.

```
#include<stdio.h>
#include <string.h>
int main(){
  char str[20];
  printf("Enter string: ");
  gets(str);//reads string from console
  printf("String is: %s",str);
  printf("\nLower String is: %s",strlwr(str));
 return 0;
}
```

Output:

```
Enter string: MYname
String is: MYname
```

```
Lower String is: myname
```

C STRING UPPERCASE: STRUPR()

The strupr(string) function returns string characters in uppercase. Let's see a simple example of strupr() function.

```c
#include<stdio.h>
#include <string.h>
int main(){
  char str[20];
  printf("Enter string: ");
  gets(str);//reads string from console
  printf("String is: %s",str);
  printf("\nUpper String is: %s",strupr(str));
 return 0;
}
```

Output:

```
Enter string: indian
String is: indian
Upper String is: INDIAN
```

C STRING STRSTR()

The strstr() function returns pointer to the first occurrence of the matched string in the given string. It is used to return substring from first match till the last character.

Syntax:

```c
char *strstr(const char *string, const char *match)
```

String strstr() parameters

string: It represents the full string from where substring will be searched.

match: It represents the substring to be searched in the full string.

String strstr() example

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

- int main(){
-   char str[100]="programming with c and java";
-   char *sub;
-   sub=strstr(str,"java");
-   printf("\nSubstring is: %s",sub);
- return 0;
- }

Output: programming with c and java

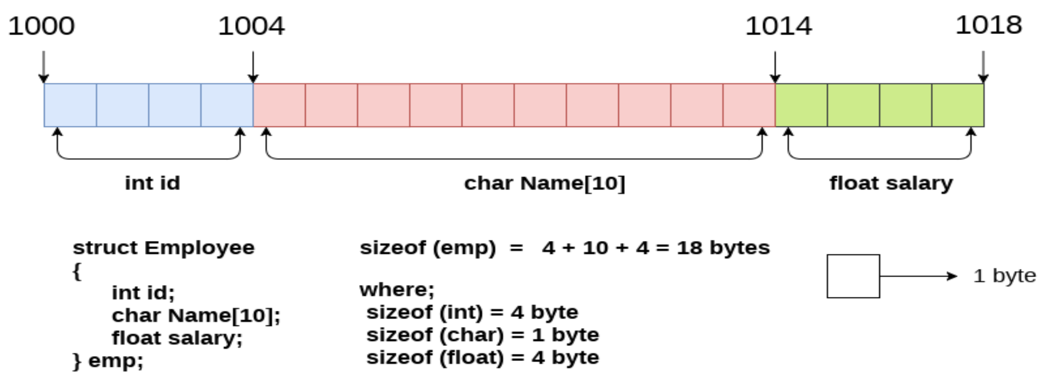
STRUCTURE

Structure in c is a user-defined data type that enables us to store the collection of different data types. Each element of a structure is called a member. Structures ca; simulate the use of classes and templates as it can store various information

The**, struct** keyword is used to define the structure. Let's see the syntax to define the structure in c.

- struct structure_name
- {
-     data_type member1;
-     data_type member2;
-     .
-     .
-     data_type memeberN;
- };
- Let's see the example to define a structure for an entity employee in c.
- struct employee
- {   int id;
-     char name[20];
-     float salary;
- };

The following image shows the memory allocation of the structure employee that is defined in the above example.



Declaring structure variable

We can declare a variable for the structure so that we can access the member of the structure easily. There are two ways to declare structure variable:

1. By struct keyword within main() function
2. By declaring a variable at the time of defining the structure.

1st way:

The example to declare the structure variable by struct keyword. It should be declared within the main function.

```
struct employee
{   int id;
    char name[50];
    float salary;
};
```

Now write given code inside the main() function.

struct employee e1, e2;

The variables e1 and e2 can be used to access the values stored in the structure. Here, e1 and e2 can be treated in the same way as the objects in C++ and Java.

2nd way:

Let's see another way to declare variable at the time of defining the structure.

```
struct employee
```

```
{   int id;
    char name[50];
    float salary;
}e1,e2;
```

Which approach is good

If number of variables are not fixed, use the 1st approach. It provides you the flexibility to declare the structure variable many times.

If no. of variables are fixed, use 2nd approach. It saves your code to declare a variable in main() function.

Accessing members of the structure

There are two ways to access structure members:

1. By . (member or dot operator)
2. By -> (structure pointer operator)

Let's see the code to access the *id* member of *p1* variable by. (member) operator.

```
p1.id
```

Structure example

Let's see a simple example of structure in C language.

```
#include<stdio.h>
#include <string.h>
struct employee
{   int id;
    char name[50];
}e1;  //declaring e1 variable for structure
int main( )
{
   //store first employee information
   e1.id=101;
   strcpy(e1.name, "Krishana");//copying string into char array
   //printing first employee information
```

```
  printf( "employee 1 id : %d\n", e1.id);
  printf( "employee 1 name : %s\n", e1.name);
return 0;
}
```

Output:

```
employee 1 id : 101
employee 1 name : Krishana
```

Let's see another example of the structure in C language to store many employees information.

```
#include<stdio.h>
#include <string.h>
struct employee
{   int id;
    char name[50];
    float salary;
}e1,e2;  //declaring e1 and e2 variables for structure
int main( )
{
  //store first employee information
  e1.id=101;
  strcpy(e1.name, "Sonoo Jaiswal");//copying string into char array
  e1.salary=56000;
 //store second employee information
  e2.id=102;
  strcpy(e2.name, "James Bond");
  e2.salary=126000;
  //printing first employee information
  printf( "employee 1 id : %d\n", e1.id);
  printf( "employee 1 name : %s\n", e1.name);
  printf( "employee 1 salary : %f\n", e1.salary);
  //printing second employee information
  printf( "employee 2 id : %d\n", e2.id);
```

- printf("employee 2 name : %s\n", e2.name);
- printf("employee 2 salary : %f\n", e2.salary);
- **return** 0;
- }

Output:

```
employee 1 id : 101
employee 1 name : Sonoo Jaiswal
employee 1 salary : 56000.000000
employee 2 id : 102
employee 2 name : James Bond
employee 2 salary : 126000.000000
```

TYPEDEF

The **typedef** is a keyword used in C programming to provide some meaningful names to the already existing variable in the C program. It behaves similarly as we define the alias for the commands. In short, we can say that this keyword is used to redefine the name of an already existing variable.

Syntax of typedef

typedef <existing_name> <alias_name>

In the above syntax, '**existing_name'** is the name of an already existing variable while '**alias name'** is another name given to the existing variable.

For example, suppose we want to create a variable of type **unsigned int**, then it becomes a tedious task if we want to declare multiple variables of this type. To overcome the problem, we use **a typedef** keyword.

typedef unsigned **int** unit;

In the above statements, we have declared the **unit** variable of type unsigned int by using **a typedef** keyword.

Now, we can create the variables of type **unsigned int** by writing the following statement:

unit a, b;

instead of writing:

unsigned **int** a, b;

Till now, we have observed that the **typedef** keyword provides a nice shortcut by providing an alternative name for an already existing variable. This keyword is useful when we are dealing with the long data type especially, structure declarations.

```
#include <stdio.h>
int main()
{
typedef unsigned int unit;
unit i,j;
i=10;
j=20;
printf("Value of i is :%d",i);
printf("\nValue of j is :%d",j);
return 0;
}
```

Output

```
Value of i is :10
Value of j is :20
```

Using typedef with structures

Consider the below structure declaration:

```
struct student
{
char name[20];
int age;
};
struct student s1;
```

In the above structure declaration, we have created the variable of **student** type by writing the following statement:

struct student s1;

The above statement shows the creation of a variable, i.e., s1, but the statement is quite big. To avoid such a big statement, we use the **typedef** keyword to create the variable of type **student**.

- **struct** student
- {
- **char** name[20];
- **int** age;
- };
- **typedef struct** student stud;
- stud s1, s2;

In the above statement, we have declared the variable **stud** of type struct student. Now, we can use the **stud** variable in a program to create the variables of type **struct student**.

The above typedef can be written as:

- **typedef struct** student
- {
- **char** name[20];
- **int** age;
- } stud;
- stud s1,s2;

From the above declarations, we conclude that **typedef** keyword reduces the length of the code and complexity of data types. It also helps in understanding the program.

- #include <stdio.h>
- **typedef struct** student
- {
- **char** name[20];
- **int** age;
- }stud;
- **int** main()
- {
- stud s1;
- printf("Enter the details of student s1: ");
- printf("\nEnter the name of the student:");
- scanf("%s",&s1.name);
- printf("\nEnter the age of student:");
- scanf("%d",&s1.age);

- printf("\n Name of the student is : %s", s1.name);
- printf("\n Age of the student is : %d", s1.age);
- **return** 0;
- }

Output

```
Enter the details of student s1:
Enter the name of the student: Peter
Enter the age of student: 28
Name of the student is : Peter
Age of the student is : 28
```

Using typedef with pointers

We can also provide another name or alias name to the pointer variables with the help of **the typedef**.

For example, we normally declare a pointer, as shown below:

```
int* ptr;
```

We can rename the above pointer variable as given below:

```
typedef int* ptr;
```

In the above statement, we have declared the variable of type **int***. Now, we can create the variable of type **int*** by simply using the '**ptr'** variable as shown in the below statement:

```
ptr p1, p2 ;
```

In the above statement, **p1** and **p2** are the variables of type '**ptr'**.

C ARRAY OF STRUCTURES

Consider a case, where we need to store the data of 5 students. We can store it by using the structure as given below.

- #include<stdio.h>
- **struct** student
- {
-     **char** name[20];

```
    int id;
    float marks;
};
void main()
{
    struct student s1,s2,s3;
    int dummy;
    printf("Enter the name, id, and marks of student 1 ");
    scanf("%s %d %f",s1.name,&s1.id,&s1.marks);
    scanf("%c",&dummy);
    printf("Enter the name, id, and marks of student 2 ");
    scanf("%s %d %f",s2.name,&s2.id,&s2.marks);
    scanf("%c",&dummy);
    printf("Enter the name, id, and marks of student 3 ");
    scanf("%s %d %f",s3.name,&s3.id,&s3.marks);
    scanf("%c",&dummy);
    printf("Printing the details....\n");
    printf("%s %d %f\n",s1.name,s1.id,s1.marks);
    printf("%s %d %f\n",s2.name,s2.id,s2.marks);
    printf("%s %d %f\n",s3.name,s3.id,s3.marks);
}
```

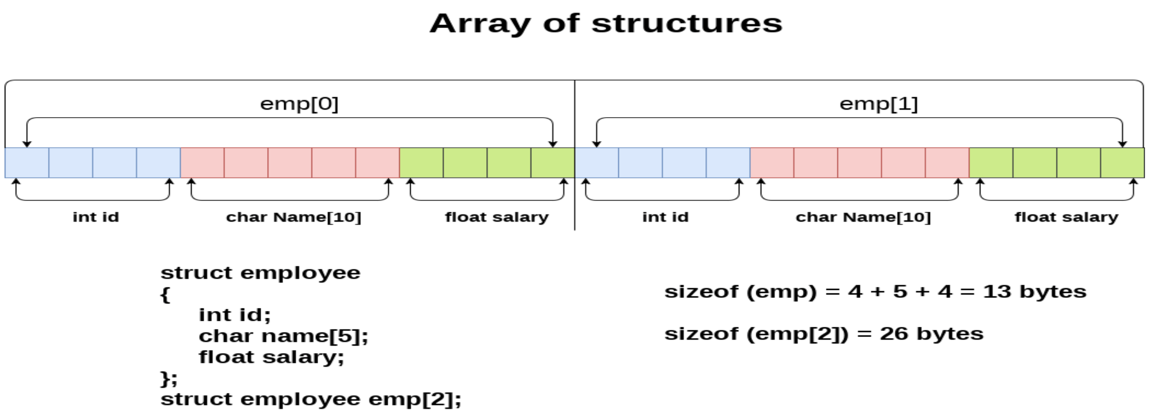
Output

```
Enter the name, id, and marks of student 1 James 90 90
Enter the name, id, and marks of student 2 Adoms 90 90
Enter the name, id, and marks of student 3 Nick 90 90
Printing the details....
James 90 90.000000
Adoms 90 90.000000
Nick 90 90.000000
```

In the above program, we have stored data of 3 students in the structure. However, the complexity of the program will be increased if there are 20 students. In that case, we will have to declare 20 different structure variables and store them one by one. This will always be tough since we will have to declare a variable every time we add a student. Remembering the name of all the variables is also a very tricky task. However, c enables us to declare an array of structures by using which, we can avoid declaring the different structure variables; instead we can make a collection containing all the structures that store the information of different entities.

Array of Structures in C

An array of structres in C can be defined as the collection of multiple structures variables where each variable contains information about different entities. The array of structures in C are used to store information about multiple entities of different data types. The array of structures is also known as the collection of structures.



```
struct employee
{
    int id;
    char name[5];
    float salary;
};
struct employee emp[2];
```

sizeof (emp) = 4 + 5 + 4 = 13 bytes

sizeof (emp[2]) = 26 bytes

Let's see an example of an array of structures that stores information of 5 students and prints it.

- #include<stdio.h>
- #include <string.h>
- struct student{
- int rollno;
- char name[10];
- };
- int main(){
- int i;
- struct student st[5];
- printf("Enter Records of 5 students");
- for(i=0;i<5;i++){
- printf("\nEnter Rollno:");
- scanf("%d",&st[i].rollno);
- printf("\nEnter Name:");
- scanf("%s",&st[i].name);

- }
- printf("\nStudent Information List:");
- **for**(i=0;i<5;i++){
- printf("\nRollno:%d, Name:%s",st[i].rollno,st[i].name);
- }
- **return** 0;
- }

Output:

```
Enter Records of 5 students
Enter Rollno:1
Enter Name:Sonoo
Enter Rollno:2
Enter Name:Ratan
Enter Rollno:3
Enter Name:Vimal
Enter Rollno:4
Enter Name:James
Enter Rollno:5
Enter Name:Sarfraz

Student Information List:
Rollno:1, Name:Sonoo
Rollno:2, Name:Ratan
Rollno:3, Name:Vimal
Rollno:4, Name:James
Rollno:5, Name:Sarfraz
```

NESTED STRUCTURE IN C

C provides us the feature of nesting one structure within another structure by using which, complex data types are created. For example, we may need to store the address of an entity employee in a structure. The attribute address may also have the subparts as street number, city, state, and pin code. Hence, to store the address of the employee, we need to store the address of the employee into a separate structure and nest the structure address into the structure employee. Consider the following program.

- #include<stdio.h>
- **struct** address
- {
- **char** city[20];
- **int** pin;

```
•     char phone[14];
•  };
•  struct employee
•  {
•     char name[20];
•     struct address add;
•  };
•  void main ()
•  {
•     struct employee emp;
•     printf("Enter employee information?\n");
•     scanf("%s %s %d %s",emp.name,emp.add.city, &emp.add.pin, emp.add.phone)
   ;
•     printf("Printing the employee information....\n");
•     printf("name: %s\nCity: %s\nPincode: %d\nPhone: %s",emp.name,emp.add.cit
   y,emp.add.pin,emp.add.phone);
•  }
```

Output

```
Enter employee information?

Arun

Delhi

110001

1234567890

Printing the employee information....

name: Arun

City: Delhi

Pincode: 110001

Phone: 1234567890
```

The structure can be nested in the following ways.

1. By separate structure
2. By Embedded structure

1) Separate structure

Here, we create two structures, but the dependent structure should be used inside the main structure as a member. Consider the following example.

```
struct Date
{
  int dd;
  int mm;
  int yyyy;
};
struct Employee
{
  int id;
  char name[20];
  struct Date doj;
}emp1;
```

As you can see, doj (date of joining) is the variable of type Date. Here doj is used as a member in Employee structure. In this way, we can use Date structure in many structures.

2) Embedded structure

The embedded structure enables us to declare the structure inside the structure. Hence, it requires less line of codes but it can not be used in multiple data structures. Consider the following example.

```
struct Employee
{
  int id;
  char name[20];
  struct Date
   {
```

- int dd;
- int mm;
- int yyyy;
- }doj;
- }emp1;

Accessing Nested Structure

We can access the member of the nested structure by Outer_Structure.Nested_Structure.member as given below:

- e1.doj.dd
- e1.doj.mm
- e1.doj.yyyy

C Nested Structure example

Let's see a simple example of the nested structure in C language.

```
#include <stdio.h>
#include <string.h>
struct Employee
{
   int id;
   char name[20];
   struct Date
    {
      int dd;
      int mm;
      int yyyy;
    }doj;
}e1;
int main( )
{
   //storing employee information
   e1.id=101;
   strcpy(e1.name, "Sonoo Jaiswal");//copying string into char array
   e1.doj.dd=10;
```

- e1.doj.mm=11;
- e1.doj.yyyy=2014;
- //printing first employee information
- printf("employee id : %d\n", e1.id);
- printf("employee name : %s\n", e1.name);
- printf("employee date of joining (dd/mm/yyyy) : %d/%d/%d\n", e1.doj.dd,e1.doj.mm,e1.doj.yyyy);
- return 0;
- }

Output:

```
employee id : 101
employee name : Sonoo Jaiswal
employee date of joining (dd/mm/yyyy) : 10/11/2014
```

Passing structure to function

Just like other variables, a structure can also be passed to a function. We may pass the structure members into the function or pass the structure variable at once. Consider the following example to pass the structure variable employee to a function display() which is used to display the details of an employee.

- #include<stdio.h>
- struct address
- {
-     char city[20];
-     int pin;
-     char phone[14];
- };
- struct employee
- {
-     char name[20];
-     struct address add;
- };
- void display(struct employee);
- void main ()
- {

```
•     struct employee emp;
•     printf("Enter employee information?\n");
•     scanf("%s %s %d %s",emp.name,emp.add.city, &emp.add.pin, emp.add.phone)
  ;
•     display(emp);
•  }
•  void display(struct employee emp)
•  {
•    printf("Printing the details....\n");
•    printf("%s %s %d %s",emp.name,emp.add.city,emp.add.pin,emp.add.phone);
•  }
```

STRUCTURE PADDING IN C

Structure padding is a concept in C that adds the one or more empty bytes between the memory addresses to align the data in memory.

Let's first understand the structure padding in C through a simple scenario which is given below:

Suppose we create a user-defined structure. When we create an object of this structure, then the contiguous memory will be allocated to the structure members.

```
•  struct student
•  {
•     char a;
•     char b;
•     int c;
•  } stud1;
```

In the above example, we have created a structure of type **student**. We have declared the object of this structure named as '**stud1**'. After the creation of an object, a contiguous block of memory is allocated to its structure members. First, the memory will be allocated to the '**a**' variable, then '**b**' variable, and then '**c**' variable.

What is the size of the struct student?

Now, we calculate the size of the **struct student**. We assume that the size of the int is 4 bytes, and the size of the char is 1 byte.

- **struct** student
- {
-   **char** a; // 1 byte
-   **char** b; // 1 byte
-   **int** c; // 4 bytes
- }

In the above case, when we calculate the size of the **struct student**, size comes to be 6 bytes. But this answer is wrong. Now, we will understand why this answer is wrong? We need to understand the concept of structure padding.

Structure Padding

The processor does not read 1 byte at a time. It reads 1 word at a time.

What does the 1 word mean?

If we have a 32-bit processor, then the processor reads 4 bytes at a time, which means that 1 word is equal to 4 bytes.

1 word = 4 bytes

If we have a 64-bit processor, then the processor reads 8 bytes at a time, which means that 1 word is equal to 8 bytes.

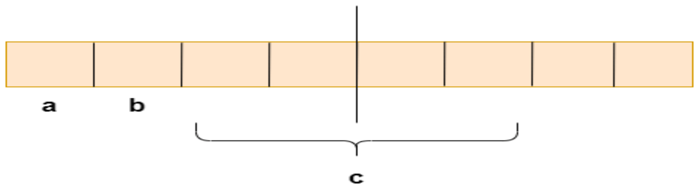
1 word = 8 bytes

Therefore, we can say that a 32-bit processor is capable of accessing 4 bytes at a time, whereas a 64-bit processor is capable of accessing 8 bytes at a time. It depends upon the architecture that what would be the size of the word.

Why structure padding?

- **struct** student
- {
-   **char** a; // 1 byte
-   **char** b; // 1 byte
-   **int** c; // 4 bytes
- }

If we have a 32-bit processor (4 bytes at a time), then the pictorial representation of the memory for the above structure would be:



As we know that structure occupies the contiguous block of memory as shown in the above diagram, i.e., 1 byte for char a, 1 byte for char b, and 4 bytes for int c, then what problem do we face in this case.

Changing order of the variables

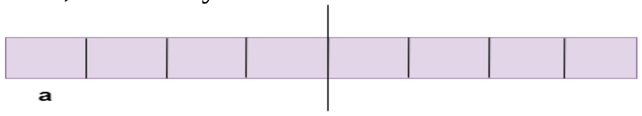
Now, we will see what happens when we change the order of the variables, does it affect the output of the program. Let's consider the same program.

```
#include <stdio.h>
struct student
{
   char a;
   int b;
  char c;
};
int main()
{
   struct student stud1; // variable declaration of the student type..
   // Displaying the size of the structure student.
   printf("The size of the student structure is %d", sizeof(stud1));
   return 0;
}
```

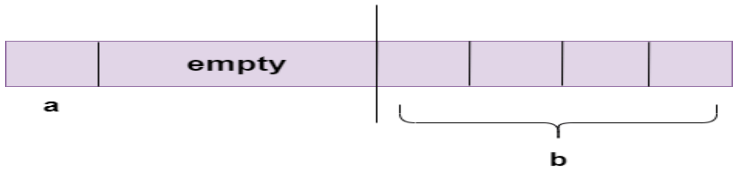
The above code is similar to the previous code; the only thing we change is the order of the variables inside the **structure student**. Due to the change in the order, the output would be different in both the cases. In the previous case, the output was 8 bytes, but in this case, the output is 12 bytes, as we can observe in the below screenshot.

Now, we need to understand "**why the output is different in this case**".

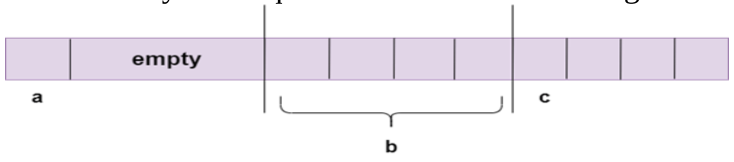
- First, memory is allocated to the **char a** variable, i.e., 1 byte.



- Now, the memory will be allocated to the **int b** Since the **int** variable occupies 4 bytes, but on the left, only 3 bytes are available. The empty row will be created on these 3 bytes, and the int variable would occupy the other 4 bytes so that the integer variable can be accessed in a single CPU cycle.



- Now, the memory will be given to the **char c** At a time, CPU can access 1 word, which is equal to 4 bytes, so CPU will use 4 bytes to access a 'c' variable. Therefore, the total memory required is 12 bytes (4 bytes +4 bytes +4 bytes), i.e., 4 bytes required to access **char a** variable, 4 bytes required to access **int b** variable, and other 4 bytes required to access a single character of '**c**' variable.



How to avoid the structure padding in C?

The structural padding is an in-built process that is automatically done by the compiler. Sometimes it required to avoid the structure padding in C as it makes the size of the structure greater than the size of the structure members.

We can avoid the structure padding in C in two ways:

- **Using #pragma pack(1) directive**
- **Using attribute**

Using #pragma pack(1) directive

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

```
#pragma pack(1)
struct base
{
    int a;
    char b;
    double c;
};
int main()
{
  struct base var; // variable declaration of type base
  // Displaying the size of the structure base
  printf("The size of the var is : %d", sizeof(var));
return 0;
}
```

In the above code, we have used the **#pragma pack(1)** directive to avoid the structure padding. If we do not use this directive, then the output of the above program would be 16 bytes. But the actual size of the structure members is 13 bytes, so 3 bytes are wasted. To avoid the wastage of memory, we use the **#pragma pack(1)** directive to provide the 1-byte packaging.

By using attribute

```
#include <stdio.h>
struct base
{
    int a;
    char b;
    double c;
}__attribute__((packed));  ;
int main()
{
  struct base var; // variable declaration of type base
  // Displaying the size of the structure base
  printf("The size of the var is : %d", sizeof(var));
    return 0;
}
```

UNION IN C

Union can be defined as a user-defined data type which is a collection of different variables of different data types in the same memory location. The union can also be defined as many members, but only one member can contain a value at a particular point in time.

Union is a user-defined data type, but unlike structures, they share the same memory location.

```
struct abc
{
  int a;
  char b;
}
```

The above code is the user-defined structure that consists of two members, i.e., 'a' of type **int** and 'b' of type **character**. When we check the addresses of 'a' and 'b', we found that their addresses are different. Therefore, we conclude that the members in the structure do not share the same memory location.

When we define the union, then we found that union is defined in the same way as the structure is defined but the difference is that union keyword is used for defining the union data type, whereas the struct keyword is used for defining the structure. The union contains the data members, i.e., 'a' and 'b', when we check the addresses of both the variables then we found that both have the same addresses. It means that the union members share the same memory location.

In union, members will share the memory location. If we try to make changes in any of the member then it will be reflected to the other member as well. Let's understand this concept through an example.

```
union abc
{
  int a;
char b;
}var;
int main()
{
  var.a = 66;
```

- printf("\n a = %d", var.a);
- printf("\n b = %d", var.b);
- }

In the above code, union has two members, i.e., 'a' and 'b'. The 'var' is a variable of union abc type. In the **main()** method, we assign the 66 to 'a' variable, so var.a will print 66 on the screen. Since both 'a' and 'b' share the memory location, **var.b** will print '**B**' (ascii code of 66).

Deciding the size of the union

The size of the union is based on the size of the largest member of the union.

- **union** abc{
- **int** a;
- **char** b;
- **float** c;
- **double** d;
- };
- **int** main()
- {
-   printf("Size of union abc is %d", **sizeof**(**union** abc));
-   **return** 0;
- }

As we know, the size of int is 4 bytes, size of char is 1 byte, size of float is 4 bytes, and the size of double is 8 bytes. Since the double variable occupies the largest memory among all the four variables, so total 8 bytes will be allocated in the memory. Therefore, the output of the above program would be 8 bytes.

Accessing members of union using pointers

We can access the members of the union through pointers by using the (->) arrow operator.

Let's understand through an example.

- #include <stdio.h>
- **union** abc
- {
-     **int** a;

```
    char b;
};
int main()
{
    union abc *ptr; // pointer variable declaration
    union abc var;
    var.a= 90;
    ptr = &var;
    printf("The value of a is : %d", ptr->a);
    return 0;
}
```

In the above code, we have created a pointer variable, i.e., *ptr, that stores the address of var variable. Now, ptr can access the variable 'a' by using the (->) operator. Hence the output of the above code would be 90.

Why do we need C unions?

Consider one example to understand the need for C unions. Let's consider a store that has two items:

- Books
- Shirts

Store owners want to store the records of the above-mentioned two items along with the relevant information. For example, Books include Title, Author, no of pages, price, and Shirts include Color, design, size, and price. The 'price' property is common in both items. The Store owner wants to store the properties, then how he/she will store the records.

Initially, they decided to store the records in a structure as shown below:

```
struct store
{
   double price;
   char *title;
   char *author;
   int number_pages;
   int color;
```

```
  int size;
  char *design;
};
```

The above structure consists of all the items that store owner wants to store. The above structure is completely usable but the price is common property in both the items and the rest of the items are individual. The properties like price, *title, *author, and number_pages belong to Books while color, size, *design belongs to Shirt.

```
int main()
{
  struct store book;
  book.title = "C programming";
book.author = "Paulo Cohelo";
book.number_pages = 190;
book.price = 205;
printf("Size is : %ld bytes", sizeof(book));
return 0;
}
```

In the above code, we have created a variable of type **store**. We have assigned the values to the variables, title, author, number_pages, price but the book variable does not possess the properties such as size, color, and design. Hence, it's a wastage of memory. The size of the above structure would be 44 bytes.

We can save lots of space if we use unions.

```
#include <stdio.h>
struct store
{
   double price;
   union
   {
      struct{
      char *title;
      char *author;
      int number_pages;
      } book;
```

```

        struct {
        int color;
        int size;
        char *design;
        } shirt;
      }item;
  };
   int main()
  {
     struct store s;
     s.item.book.title = "C programming";
     s.item.book.author = "John";
     s.item.book.number_pages = 189;
     printf("Size is %ld", sizeof(s));
     return 0;
  }
```

In the above code, we have created a variable of type store. Since we used the unions in the above code, so the largest memory occupied by the variable would be considered for the memory allocation. The output of the above program is 32 bytes. In the case of structures, we obtained 44 bytes, while in the case of unions, the size obtained is 44 bytes. Hence, 44 bytes is greater than 32 bytes saving lots of memory space.

FILE HANDLING IN C

In programming, we may require some specific input data to be generated several numbers of times. Sometimes, it is not enough to only display the data on the console. The data to be displayed may be very large, and only a limited amount of data can be displayed on the console, and since the memory is volatile, it is impossible to recover the programmatically generated data again and again. However, if we need to do so, we may store it onto the local file system which is volatile and can be accessed every time. Here, comes the need of file handling in C.

File handling in C enables us to create, update, read, and delete the files stored on the local file system through our C program. The following operations can be performed on a file.

- Creation of the new file
- Opening an existing file

- Reading from the file
- Writing to the file
- Deleting the file

Functions for file handling

There are many functions in the C library to open, read, write, search and close the file. A list of file functions are given below:

No.	Function	Description
1	fopen()	opens new or existing file
2	fprintf()	write data into the file
3	fscanf()	reads data from the file
4	fputc()	writes a character into the file
5	fgetc()	reads a character from file
6	**fclose()**	closes the file
7	fseek()	sets the file pointer to given position
8	fputw()	writes an integer to file
9	fgetw()	reads an integer from file
10	ftell()	returns current position
11	rewind()	sets the file pointer to the beginning of the file

Opening File: fopen()

We must open a file before it can be read, write, or update. The fopen() function is used to open a file. The syntax of the fopen() is given below.

```
FILE *fopen( const char * filename, const char * mode );
```

The fopen() function accepts two parameters:

- The file name (string). If the file is stored at some specific location, then we must mention the path at which the file is stored. For example, a file name can be like **"c://some_folder/some_file.ext"**.
- The mode in which the file is to be opened. It is a string.

We can use one of the following modes in the fopen() function.

Mode	Description
R	opens a text file in read mode
W	opens a text file in write mode
A	opens a text file in append mode
r+	opens a text file in read and write mode
w+	opens a text file in read and write mode
a+	opens a text file in read and write mode
Rb	opens a binary file in read mode
Wb	opens a binary file in write mode
Ab	opens a binary file in append mode
rb+	opens a binary file in read and write mode
wb+	opens a binary file in read and write mode

ab+	opens a binary file in read and write mode

The fopen function works in the following way.

- Firstly, It searches the file to be opened.
- Then, it loads the file from the disk and place it into the buffer. The buffer is used to provide efficiency for the read operations.
- It sets up a character pointer which points to the first character of the file.

Consider the following example which opens a file in write mode.

```
#include<stdio.h>
void main( )
{
FILE *fp ;
char ch ;
fp = fopen("file_handle.c", "r") ;
ch = fgetc (fp);
printf("%c",ch) ;
}
fclose (fp ) ;
}
```

Output

The content of the file will be printed.

```
void main( )
{
FILE *fp; // file pointer
char ch;
fp = fopen("file_handle.c","r");
while ( 1 )
{
ch = fgetc ( fp ); //Each character of the file is read and stored in the character file.
if ( ch == EOF )
break;
printf("%c",ch);
}
fclose (fp );
}
```

Closing File: fclose()
The fclose() function is used to close a file. The file must be closed after performing all the operations on it. The syntax of fclose() function is given below:

```
int fclose( FILE *fp );
```

C fprintf() and fscanf()
C fprintf() and fscanf() example

C fputc() and fgetc()
C fputc() and fgetc() example

C fputs() and fgets()
C fputs() and fgets() example

C fseek()
C fseek() example

LIBRARY FUNCTION IN C

Introduction:

In the world of programming, ***libraries*** are one of the most important and useful tools available. They are prewritten pieces of code that can be used to perform specific tasks, such as ***sorting*** or ***searching data***, without the need to write the code from scratch. C is a popular programming language that has a wide range of libraries available for use. In this article, we will explore the concept of library functions in C and how they can be used to simplify programming tasks.

What is a library function in C?

In C programming language, a ***library function*** is a prewritten piece of code that performs a specific task. These functions are included in ***precompiled libraries***, which can be linked to a program to provide additional functionality. ***Library functions*** can be categorized into two types: ***Standard Library Functions*** and ***User-defined Library Functions***.

Standard Library Functions:

The ***standard library functions*** are a set of functions that are provided by the C programming language. These functions are included in the ***standard library***, which is part of the C language specification.

The ***standard C library*** is also known as the ***libc library***. It is a collection of functions and macros that are part of the C programming language. These functions provide a wide range of functionality, including ***input/output operations***, ***string manipulation***, ***memory allocation, mathematical calculations,*** and more.

Some of the common standard library functions in C include:

1. **Input/output functions:** These ***input/output functions*** are used to read input from the user or write output to the screen or a file. Examples include ***printf(), scanf(),*** and ***gets()***.
2. **String manipulation functions:** These ***string manipulation functions*** are used to ***manipulate strings*** in C. Examples include ***strlen()***, ***strcpy(),*** and ***strcat()***.
3. **Mathematical functions:** These ***mathematical functions*** are used to perform ***mathematical operations*** in C. Examples include ***sin(), cos(),*** and ***sqrt()***.
4. **Time functions:** These ***time functions*** are used to retrieve the current time or perform time-related calculations. Examples include ***time(), localtime(),*** and ***strftime()***.

User-defined Library Functions:

User-defined library functions are functions that are created by the programmer and added to a library for later use. These functions can be written in C or any other programming language that can be compiled into a library. User-defined library functions can be used to simplify complex programming ***tasks***, ***reuse code***, and ***improve code maintainability***.

Creating a User-defined Library Function:

To create a user-defined library function, the following steps can be taken:

1. **Write the function code:** Write the code for the function that performs the required task.
2. **Create a header file:** Create a header file for the function that contains the function prototype.
3. **Compile the code:** Compile the code into an object file using a compiler.
4. **Create the library:** Create a library by combining the object file with other required object files.
5. **Link the library:** Link the library to the program that will use the function.

Advantages of Using Library Functions:

There are several advantages of using library functions which includes:

1. **Saves time:** Library functions provide prewritten code that can be used to perform common programming tasks, saving programmers time and effort.
2. **Increases productivity:** With library functions, programmers can quickly develop complex programs without worrying about the low-level details.
3. ***Improves code readability:*** Library functions provide a standardized way of performing common tasks, making the code more readable and easier to maintain.
4. **Reduces bugs:** The use of library functions reduces the chances of introducing bugs into the program since the code has already been tested and debugged.

Library Functions:

In C, functions can be defined within the code of a program, but these are not considered library functions. ***Library functions*** are pre-written functions that are included in a library and can be used in multiple programs without the need to write the code again.

Linking Libraries:

To use ***library functions*** in a C program, the library needs to be linked with the program during the compilation process. It can be done in two ways: ***static linking*** and ***dynamic linking***.

Static Linking: In ***static linking***, the library functions are copied into the executable file during compilation. It means that the executable file includes all the necessary code and can be run independently of the library.

Dynamic Linking: In ***dynamic linking***, the library functions are not copied into the executable file. Instead, the program references the library functions at runtime. It means that the library file needs to be present on the system when the program is run.

Header Files:

Header files are used to declare the prototypes for the functions included in the library. The ***header file*** contains the function declarations, as well as any necessary ***macros***, ***constants***, and ***types***. Header files are included in the source code using the ***#include directive***.

For example, if we wanted to use the ***printf() function*** from the standard C library, we would include the ***stdio.h header*** file at the top of our code:

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

In this example, we include the ***stdio.h*** header file, which declares the prototype for the ***printf() function***. It allows us to use the ***printf() function*** without having to write the code ourselves.

Custom Library Functions:

On the other hand, ***custom functions*** are functions that are written specifically for a program and cannot be used in other programs without copying the code. In addition to ***standard library functions***, programmers can also create their own custom library functions to encapsulate common functionality and reduce code duplication. ***Custom library functions*** can be defined in separate source files and compiled into a library, which can then be linked with multiple programs.

To create a custom library function, the function needs to be defined in a source file with a ***.c extension***, and the ***function prototype*** needs to be declared in a header file with a ***.h extension***. After that, the ***source file*** is compiled into an object file with a ***.o extension***, and the ***object files*** for all the functions in the library are combined into a single library file with a ***.a*** or ***.so extension***, depending on whether static or dynamic linking is used.

Example:

Here's an example of a custom library function that calculates the factorial of a number:

```
int factorial(int n)
{
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

In this example, we define a recursive function called ***factorial()*** that calculates the factorial of a given number. After that, we can compile this source file into an object file using a command like:

```
gcc -c factorial.c -o factorial.o
```

We can then combine this object file with other object files to create a library file using a command like:

```
arrcslibmylib.afactorial.o
```

It creates a static library file called ***libmylib.a*** that can be linked with other programs using the ***-lmylib*** option during compilation.

Header Files:

Header files are used to declare the function prototypes for library functions, both ***standard*** and ***custom***. A function prototype is a declaration of a ***function's name***, ***return type***, and ***parameter types***, but without the function body. Header files also include any necessary ***macros***, ***constants***, and ***types*** that are needed by the functions.

For example, here's the header file for our ***factorial() function*:**

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

In this example, we declare the prototype for the ***factorial() function***, which takes an integer ***parameter*** and returns an integer value.

When using library functions in a program, the corresponding header files need to be included using the ***#include directive***.

For example:

```
#include "factorial.h"

int main()
{
    int n = 5;
    int result = factorial(n);
printf("Factorial of %d is %d\n", n, result);
    return 0;
}
```

In this example, we include the ***factorial.h header file***, which declares the prototype for the ***factorial() function***. After that, we call the ***factorial() function*** with a value of **5** and print the result using ***printf()***.

Common Library Functions:

There are many standard library functions available in C, including:

1. **string.h:** functions for manipulating strings, such as ***strcpy()*** and ***strlen().***
2. **stdio.h:** ***input/output functions***, such as ***printf()*** and ***scanf().***
3. **math.h:** mathematical functions, such as ***sin()*** and ***sqrt().***
4. **time.h:** functions for working with dates and times, such as ***time()*** and ***localtime().***
5. **ctype.h:** functions for working with characters, such as ***isalpha()*** and ***toupper().***

Some examples of using library functions in C:

Example 1: Let's take an example using the ***string.h*** library to ***manipulate strings***.

```
#include <stdio.h>
#include <string.h>
int main()
{
    char str1[20] = "Hello";
    char str2[20] = "World";
    // concatenate str2 to str1
strcat(str1, str2);
    // print the concatenated string
printf("%s", str1);
    return 0;
}
```

Output: HelloWorld

In this example, we include the ***string.h library*** and use the ***strcat() function*** to concatenate the ***str2 string*** onto the end of the ***str1 string***. After that, we use ***printf()*** to print the resulting concatenated string.

Example 2: Let's take an example using the ***math.h library*** to perform mathematical calculations.

```
#include <stdio.h>
#include <math.h>
int main()
{
    double x = 4.0;
    // calculate the square root of x
    double y = sqrt(x);
    // print the result
printf("The square root of %f is %f", x, y);
    return 0;
}
```

Output: The square root of 4.000000 is 2.000000

Example 3: Let's take an example using the ***time.h library*** to work with ***dates*** and ***times.***

```
#include <stdio.h>
#include <time.h>
int main()
{
time_t t = time(NULL);
    struct tm *tm = localtime(&t);
    // print the current date and time
printf("Current date and time: %s", asctime(tm));
    return 0;
}
```

Output: Current date and time: Mon May 8 17:07:05 2023

Explanation: In this example, we include the ***time.h library*** and use the ***time() function*** to get the current time, which is then passed to the ***localtime() function*** to convert it to a local time structure. After that, we use ***printf()*** and the ***asctime() function*** to print the ***current date*** and ***time*** in a readable format.

#DEFINE

The #define preprocessor directive is used to define constant or micro substitution. It can use any basic data type.

Syntax: #define token value

Let's see an example of #define to define a constant.

- #include <stdio.h>
- #define PI 3.14
- main() {
-    printf("%f",PI);
- }

Output: 3.140000

Let's see an example of #define to create a macro.

- #include <stdio.h>
- #define MIN(a,b) ((a)<(b)?(a):(b))
- **void** main() {
-    printf("Minimum between 10 and 20 is: %d\n", MIN(10,20));
- }

Output: Minimum between 10 and 20 is: 10