UNIT I

1

STRUCTURE OF C PROGRAM

Unit Structure

1.0 OBJECTIVES

- Understanding the structure of c programming language
- Understanding the working of header files
- How to use comments in a c program
- Understanding the concept of interpreters and compilers
- Working with compilation and execution of program
- Understanding the structure of formatted input and output functions

1.1 INTRODUCTION

History of C language:

C programming language was developed by Dennis Ritchie in the year 1972 at AT & T Bell laboratories. It is a general purpose, imperative and procedural language. Mainly this language was invented to write UNIX operating system. It means UNIX is totally written in C. C was initially used for making up the operating systems because it produces the code which runs as fast as assembly language.

There is a similarity between any languages eg. English language and any programming language. Because when we learn English language We start learning alphabets first then we start forming words then small sentences and then we write paragraphs. Similarly when we are considering C language first we start learning alphabets, digits and special characters then we come to know keywords, constants and variables then in next

phase we do follow instructions and the last phase where we are able to write the program.

What is Program:

So when we try to define the word Program we can define it as "A set of instructions to be followed with rules in a sequential manner to get the desired output". To understand this definition in a very well manner we have to keep in mind certain things such as

1. It is a set of instructions (not just a single instruction).
2. While writing a program rules related to it must be followed.
3. Sequence of the instructions must be taken care of. If sequence is broken it will not give us the expected output.

To understand the importance of sequence of instructions we will consider the simple example of tea preparation

Eg: If Tea preparation is our program in that case we have to follow following instructions:

1. Switch on the fire and put a vessel on it
2. Add water
3. Add sugar and tea powder
4. Let it boil
5. Add milk and let it boil for a while

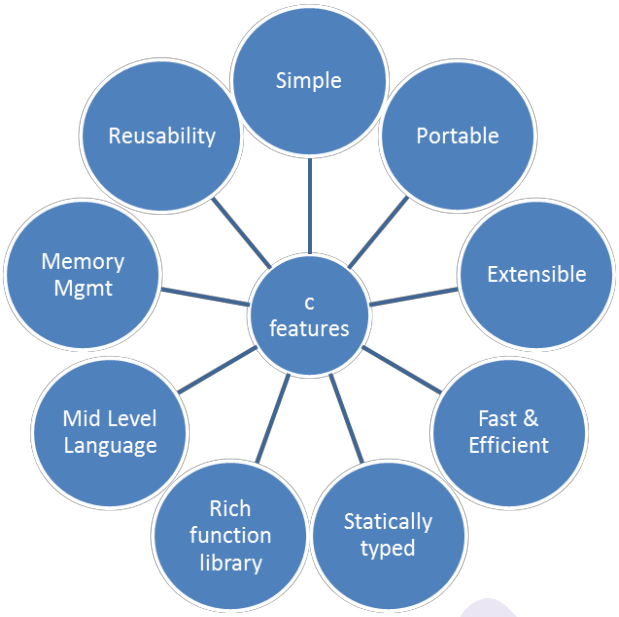
So in these 5 steps we will get our expected output i.e. tea. But if suppose we are following the instructions such as:

1. Add sugar and tea powder
2. Let it boil
3. Add milk and let it boil for a while
4. Add water

In this second case we can see that we have followed all the instructions but not in a proper sequence and hence we will not get the desired output.so one can understand that sequence is most important in any program. The word "**कार्यक्रम"** is most suitable for program. As **"कार्य"** means work or function and "क्रम" means sequence. Hence it says program means the work to be done in a sequence.

Features of C:

C is widely used language and it has following features:



1. **Simple:** The language is simple and easmy to understand as it provides structural approach i.e. it used to break the program in various parts

2. **Portable:** C is a machine independent and portable language and it can be compiled and run on different machine with some changes.

3. **Extensible:** Programs written in C language can be extended means in existing feature and operations of a program new features and operations can be added.

4. **Fast and Efficient:** C language is having lessor inbuilt functions as compared to languages like python and hence overhead is less as well as it directly interact with computer hardware and hence leads to increase in speed

5. **Statically typed:** Whenever a programmer types a program he /she has to mention types of variables used because types of variables are checked at compile time and not run time. Hence it is a statically typed language.

6. **Rich function library:** C provides lot of inbuilt functions which results in fast development

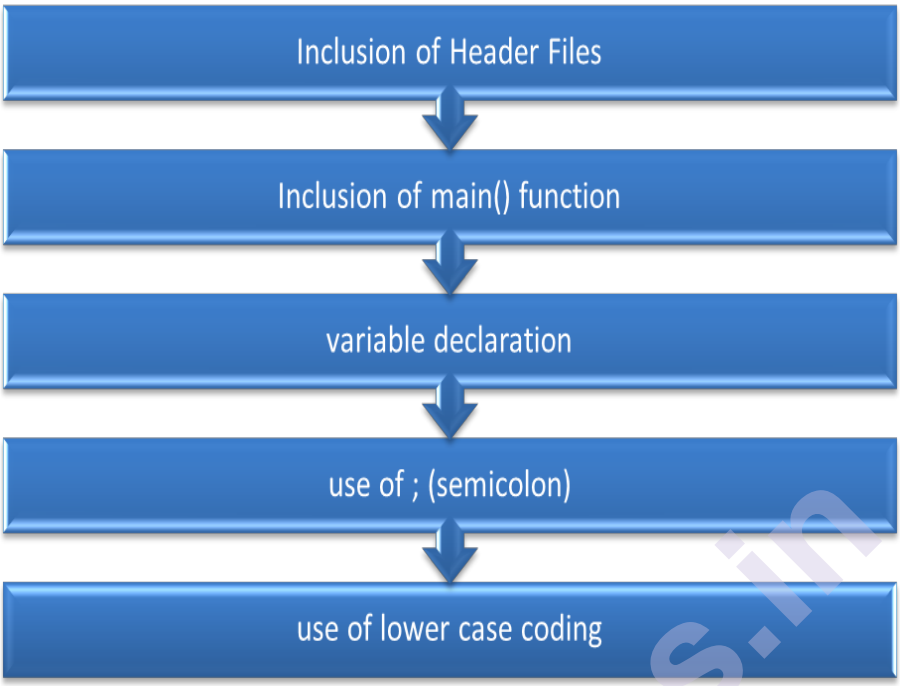
7. **Mid Level Language:** C is intended to do low level programming such as developing system application as well as it also support high level language features and hence it is called as mid level language

8. **Memory Management:** C language support dynamic memory allocation. The allocated memory can be made free whenever we need using free() function.

9. **Reusability:** C enables the feature of function call inside the function i.e. recursion which leads to code reusability through backtracking.

1.2 C HEADER AND BODY

To write a C program one must follow the given below rules:



1. **Inclusion of Header Files:** The header files in C contains C function declaration and macro definitions which are to be shared between several source files. Header files serve two purposes.

- System header files declare the interfaces to parts of the operating system. Whenever they are included in the program they supply the definitions and declarations to invoke system calls and libraries.
- When we create our own header files containing declarations for interfaces between the source files of your program. Each time a group of related declarations and macro definitions all or most of which are needed in several different source files, in such cases instead of writing all those declarations again and again it is a good idea to create a header file for them.

These header files are always included in the C program with a pre-processor directive '# include'.

Conventionally the C header files are having an extension .h and name of the file can contain only letters, digits, dashes, and underscores and at most one dot.

Examples of header files are:

Sr.No.	Header File	Type of Functions
1	<assert.h>	Diagnostics Functions
2	<ctype.h>	Character Handling Functions
3	<locale.h>	Localization Functions
4	<math.h>	Mathematics Functions
5	<setjmp.h>	Nonlocal Jump Functions
6	<signal.h>	Signal Handling Functions
7	<stdarg.h>	Variable Argument List Functions
8	<stdio.h>	Input/Output Functions
9	<stdlib.h>	General Utility Functions
10	<string.h>	String Functions
11	<time.h>	Date and Time Functions

2. Inclusion of main() function:

main () is the only function in C language which is self executable in nature. It is a primary function and acts a starting point for program execution. All the instructions of a program should be enclosed within the scope of main() function only. A program usually starts its execution in the beginning of main() and stops the execution at the end of main(). This is the reason we never write main(); means we are not putting (;) after main(). Because if we put (;) after main() it will terminate the main() function and thereby will not be able to execute the instructions written inside main().

We either can write main() or void main(). However if we just write main() in that case we should return some value at the end. Whereas when we are writing void main() we are making it clear to main() function that there is no need to return any value.

3. Variable Declaration:

The next thing what we have to after opening main() function is declaration of variables. In C language compulsorily the variables are to be declared explicitly with its datatypes before using them. We can declare and use the variable at the same time as we can do in various high level languages.

4. Use of ; (semicolon):

Every language is having its own grammar synonymously in programming languages we used to call it as syntax. As per the syntax every instruction in c language is terminated with the help of (;) semicolon. However some exceptional statements are there which must not be terminated using semicolon. Eg : main() function , conditions, loops etc. These are the statement on which something is dependent and if they are terminated the dependent statements can not be executed. So the statements on which something is dependent can not be terminated using (;) semicolon.

5. Use of lower case coding:

C is not a loosely typed language it directly interacts with hardware and so all the instructions in C language must be compulsorily to be typed in small case only. Certain things such as name of the variables or messages which are to be printed can be in capital case.

1.3 COMMENTS

Comments are non executable statements used as annotation or explanation of the source code which is readable by the programmer only.

C language provides two different types of comments:

1. **Single line comment:** For a single line commet // (double slash) can be used.

Eg: // Comment goes here

2. **Multiline comment:** For multiline comments we use /* symbol for starting and */ for ending the comment

Eg: /* comments are non executable statements

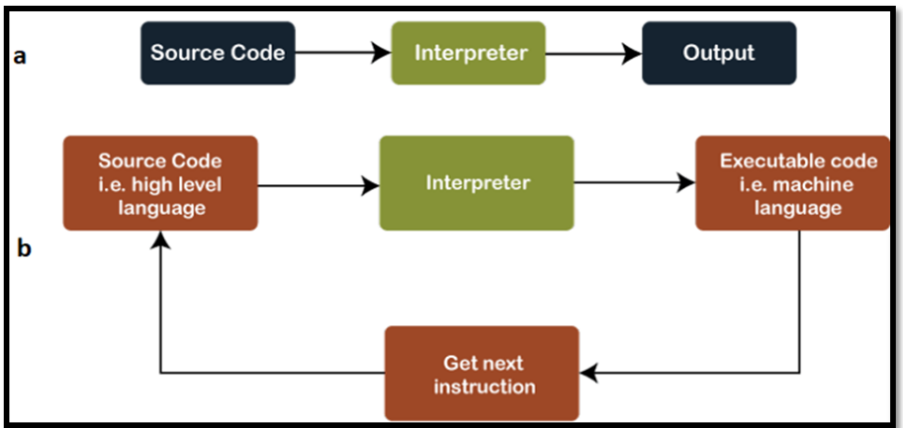
Comment goes here */

1.4 INTERPRETERS V/S COMPILERS

Compilers and interpreters are programs that enables to conversion of source code into machine code. Computer programs are normally written in high level language ie. Human understandable language whereas computers understand only machine language ie. Binary language ie. 0 and 1 only. So to enable the communication between human being (high level language) and computer (machine language) there is a need of language translators. Interpreters and compilers both are working as language translators. However the functioning of both is different.

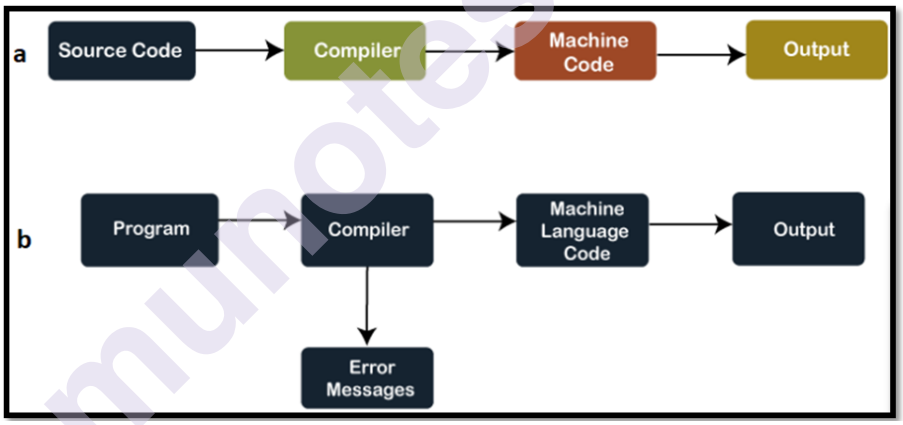
Working of Interpreters:

An interpreter is a software program which translates a program source code into a machine language ie 0 and 1 . However, the source code is interpreted line-by-line whle running the program. If there is any error in any line it leaves that line and moves to the next line and try to interpret it. At the end whichever lines it has interpreted successfully on that basis it generates the output and shown to the user.



Working of Compilers:

A compiler is also a software program that follows the syntax rule of programming language to convert a source code to machine code. It cannot fix any error if present in a program; it generates an error message, and it has to be corrected by the programmer in the program's syntax. If the written program is errorless then the compiler converts entire source code into machine code. A compiler converts complete source code into machine code at once. And finally, the program get executed.



Interprter v/s Compiler:

Sr.No.	Interpreter	Compiler
1	The program code is interpreted one line at a time	The program is compiled at one stretch
2	Line by line code is scanned and encountered errors are shown	All the errors are shown at the end of scanning process
3	Requires more time for execution	Time required for execution is vey less
4	Doesn't convert source code into object code	Converts source code into object code
5	Examples :Python, Ruby, Perl, MATLAB	Examples : C, C++, C#

1.5 PYTHON V/S C

C and python are very similar languages and are used for development of various applications. Sometimes it becomes very difficult to decide when to use python and when to use C. Whereas the difference between c and python is that c is a structured programming language widely used for hardware related applications such operating system development and python is a multi paradigm general purpose language used for machine learning, natural language processing and web development.

Following table will illustrate the difference between python and C in a **more easy way:**

Sr.No.	Python	C
1	Python is an interpreted, high-level, general-purpose programming language	C is a general-purpose, procedural computer programming language.
2	Interpreted programs execute slower as compared to compiled programs.	Compiled programs execute faster as compared to interpreted programs
3	It is easier to write a code in Python as the number of lines is less comparatively	Program syntax is harder than Python
4	There is no need to declare the type of variable. Variables are untyped in Python. A given variable can be stuck on values of different types at different times during the program execution	In C, the type of a variable must be declared when it is created, and only values of that type must be assigned to it.
5	Error debugging is simple. This means it takes only one in instruction at a time and compiles and executes simultaneously. Errors are shown instantly and the execution is stopped, at that instruction	In C, error debugging is difficult as it is a compiler dependent language. This means that it takes the entire source code, compiles it and then shows all the errors
6	Supports function renaming mechanism i.e, the same function can be used by two different names.	C does not support function renaming mechanism. This means the same function cannot be used by two different names

7	Syntax of Python programs is easy to learn, write and read	The syntax of a C program is harder than Python
8	Python uses an automatic garbage collector for memory management	In C, the Programmer has to do memory management on their own.
9	Python is a General-Purpose programming language	C is generally used for hardware related applications
10	Python has a large library of built-in functions	C has a limited number of built-in functions
11	Gives ease of implementing data structures with built-in insert, append functions	Implementing data structures requires its functions to be explicitly implemented
12	No pointers functionality available in Python	Pointers are available in C

1.6 PROGRAM COMPILATION

Installation Process: C software can be downloadable freely and easily available.

Link to download C software : https://developerinsider.co/download-turbo-c-for-windows-7-8-8-1-and-windows-10-32-64-bit-full-screen/

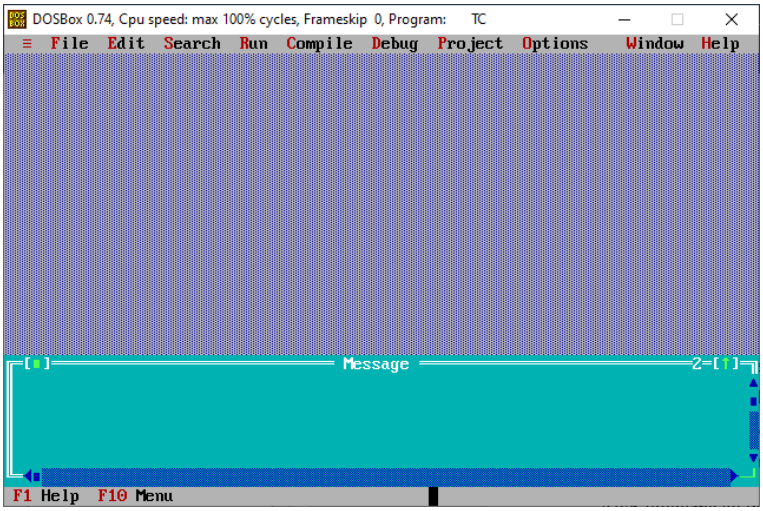
Steps for installation:

1. Download the software using above link
2. Extract downloaded “Turbo C++3.2.zip” file
3. Run the setup.exe file
4. Follow the setup instructions

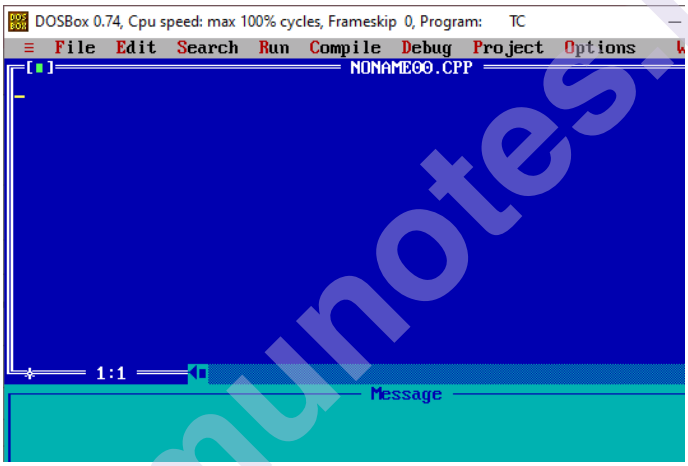
After following all the instructions following icon will get ceated on desktop:

Steps for program compilation and execution:

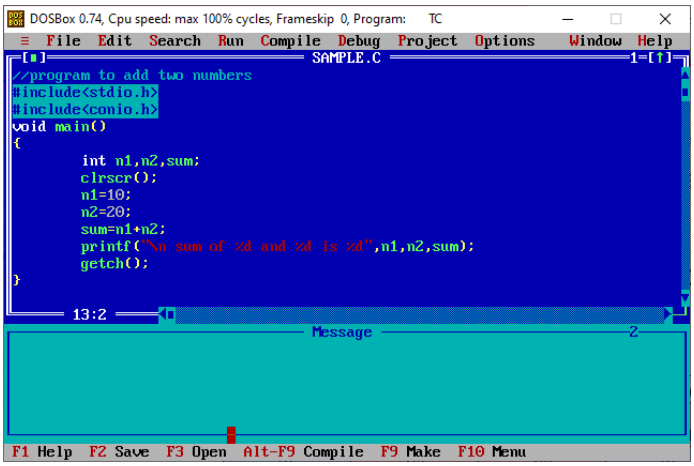
1. **Double click on Turbo C++ icon and then click on start Turbo C++ a text editor will open like this:**



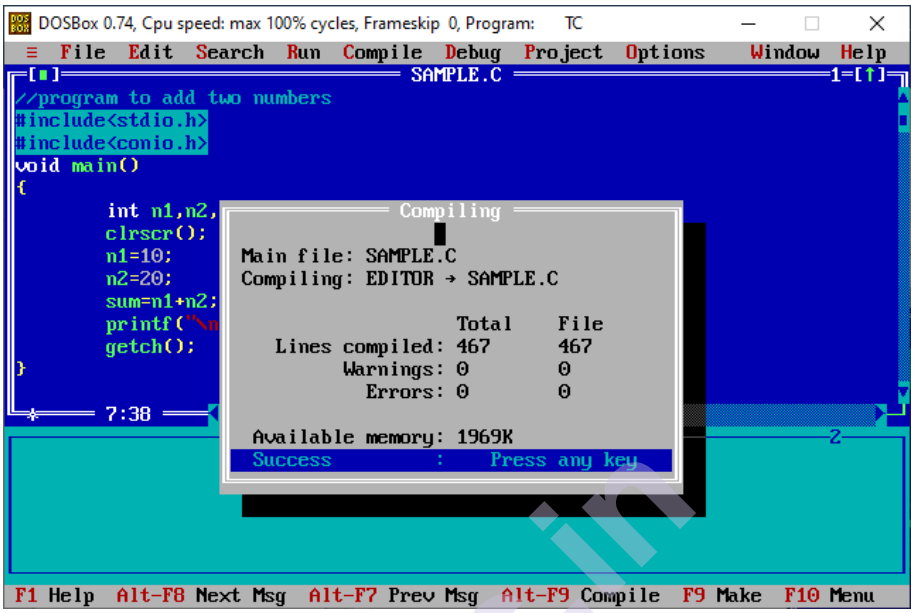
2. Click on File option New menu following screen will appear. It is nothing but a text editor where actually can write the source code. The default name is 'NONAME00' and extension is '.cpp'. However since we are writing program for C, extension should be given as 'c'.



3. Write a source code here and save it with program name. Extension. By default the C programs used to get stored on the path C:\Turboc3\Bin . Here name of the program is sample and extension is c.



4. Compile the program by pressing the shortcut key Alt + F9. The Following screen will appear. Here it is showing success and hence we can execute this program.



5. To execute the program press Ctrl + F9. It will display the output as follows:

1.7 FORMATTED INPUT OUTPUT FUNCTIONS

C provides standard functions scanf() and printf(), for performing formatted input and output .These functions accept, as parameters, a format specification string and a list of variables.

The format specification string is a character string that specifies the data type of each variable to be input or output and the size or width of the input and output.

Formatted input function:

The scanf() function is used for inputs formatted from standard inputs and provides numerous conversion options for the printf() function.

Syntax:

```
scanf(format_specifiers, &data1, &data2,……); // & is address operator
```

The scanf() function reads and converts the characters from the standard input according to the format specification string and stores the input in the memory slots represented by the other arguments.

Example:

```
scanf("%d %c",&data1,&data2);
```

In the case of string data names, the data name is not prefixed with the &.

Formatted Output Function:

The printf() function is used for output formatted as the standard output according to a format specification. The format specification string and the output data are the parameters of the printf() function.

Syntax:

```
printf(format_specifiers, data1, data2,…..... );
```

Example:

```
printf("%d %c", data1, data2);
```

The character specified after % is referred to as a conversion character because it allows a data type to be converted to another type and printed.

1.8 SUMMARY

C is a mother of all languages. It is a general purpose imperative and procedural language. C was initially used for making up the operating systems.

Program is "A set of instructions to be followed with rules in a sequential manner to get the desired output".

C is simple, portable, extensible, fast as well as efficient. It has a rich library of functions.

While writing a c program 5 rules are to be kept in mind such as including header files, including main() function, variable declaration , using lower case coding and for statement terminaton semicolon is used.

Single line comments are denoted by // symbol and multiline comments are denoted by /* */ symbol

Interpreters are the language translators which interprets the code line by line

Compilers are the language translators which compiles the source code at a single stretch

Printf() is a formatted output function and scanf() is a formatted input function

1.9 UNIT END QUESTIONS

1. Explain feature of C
2. Explain in brief the rules to be followed while writing a C program
3. What are header files? Explain its use also list out the header files used in C
4. Write short note on interpreters
5. Write short note on compiler
6. Write difference between interpreter and compiler
7. Compare Python v/s C
8. Write short note of formatted input and output functions

*****

2

DATATYPES IN C

Unit Structure

2.0 OBJECTIVES

- Understanding variables and constants
- Understanding the concept of keyword and identifiers
- Working with datatypes and their conversion characters
- Comparing python and C datatypes
- C and python type checking

2.1 INTRODUCTION

This chapter mainly focuses on the concept of variables, constants, reserved keywords, identifiers and datatypes of C language. Variables are nothing but names allocated to storage areas and are declared with the help of datatypes. There are various categories of datatypes and these datatypes are working with the help of their conversion characters. We will also come to know how C and python datatypes are different form each other and type checking in C as well as python.

2.2 KEYWORDS AND IDENTIFIERS

Identifiers:

C identifiers are the names of variables, functions, arrays, structures, unions, and labels in C program. An identifier is composed of uppercase, lowercase letters, underscore, digits. Whereas starting letter should be either an alphabet or an underscore. The identifiers are of two types i.e. Internal identifier and external identifier. If the identifier is not used in the external linkage, then it is called as an internal identifier. If the identifier is used in the external linkage, then it is called as an external identifier.

Rules for constructing C identifiers:

1. The first character of an identifier should be either an alphabet or an underscore and can be followed by character, digit or an underscore.
2. It must not begin with any digit
3. Commas and blank spaces are not allowed within the identifier
4. Keywords cannot be used as identifiers
5. The length of identifiers must not be more than 31 characters

Keywords:

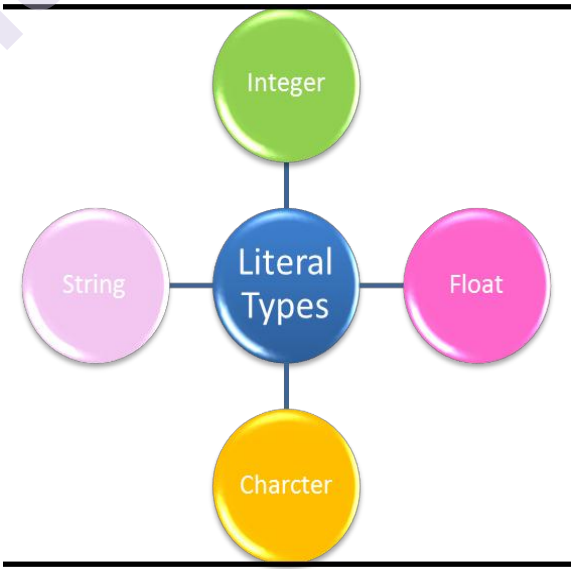
Keywords are predefined reserved words which has special meaning defined by the compiler. In C language there are 32 keywords as listed below:

auto	double	Int	Struct
break	else	long	Switch
case	enum	register	Typedef
char	extern	return	Union
continue	for	signed	Void
do	if	static	While
default	goto	sizeof	Volatile
const	float	short	Unsigned

Literals:

Literals are the constant values assigned to the constant variables. Literals represent the fixed values which can not be modified. It contains memory but does not have references as variables.

Types of Literals:



Integer literal:

It is a numeric literal that represents only integer type values. It represents the value neither in fractional nor exponential part.

An integer literal is suffixed by following two sign qualifiers:

L or l: It is a size qualifier that specifies the size of the integer type as long.

U or u: It is a sign qualifier that represents the type of the integer as unsigned. An unsigned qualifier contains only positive values.

Float literal:

It is a literal that contains only floating-point values or real numbers. These real numbers contain the number of parts such as integer part, real part, exponential part, and fractional part. The floating-point literal must be specified either in decimal or in exponential form.

Decimal form:

The decimal form must contain either decimal point, exponential part, or both. If it does not contain either of these, then the compiler will throw an error. The decimal notation can be prefixed either by '+' or '-' symbol that specifies the positive and negative numbers.

Exponential Form:

The exponential form is useful when we want to represent the number, which is having a big magnitude. It contains two parts, i.e., mantissa and exponent. For example, the number is 2340000000000, and it can be expressed as 2.34e12 in an exponential form.

2.3 VARIABLES AND CONSTANTS

Variables:

The meaning of word variable encapsulated in it. If we segregate the word Variable into two parts i.e. Vari (Vary) and Able it means capacity to change. So the values of variables can be changed as and when required in the program. Variables are the names given to the data storage areas used for program manipulation . Each and every variable in C has its own specific type and size, layout of memory allocation and range of values that can be stored within that memory.

The rules regarding variable declaration are same as we have already seen in rules for declaration of identifiers such as:

1. Variable name must start with and alphabet only and can be followed by digit and can have an underscore
2. Variable name must not contain any blank space.

3. C language is case sensitive so same names of variables in uppercase and lowercase will be treated as different.

Constants:

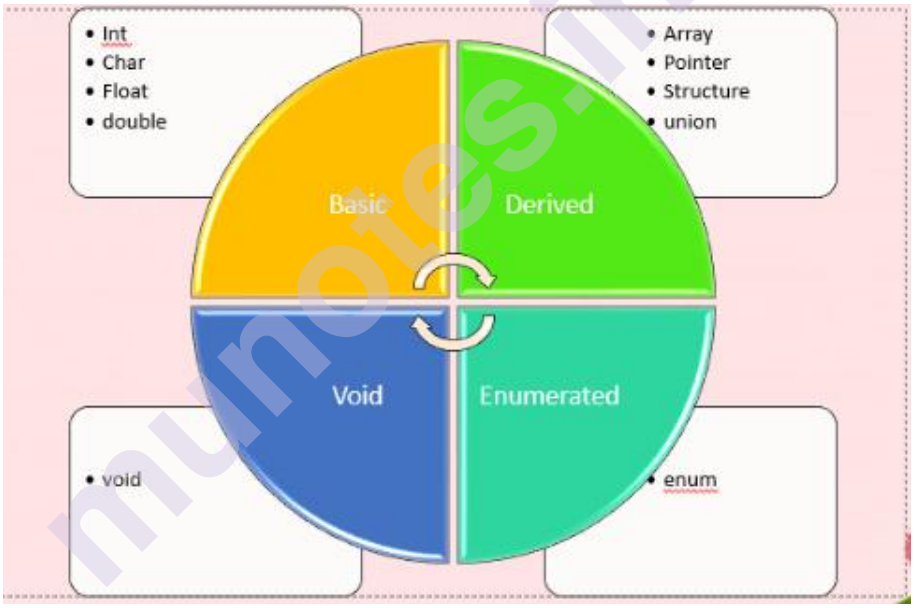
As we have seen the meaning of word variable is capacity to change and hence we can state that value of the variable can be changes as and when it is required in the program. However the constants have the feature not to changes its value anywhere in the program. It remains fixed in the whole program.

The define a constant #define preprocessor can be used.

Eg : #define PI 3.14;

Here the constant PI is having value 3.14 and it cant be changed anywhere in the program.

2.4 DATATYPES IN C



The above mentioned diagram reflects the datatypes in C. There are 4 categories of datatype and that are:

1. **Basic: Basic datatypes are divided into two types:**

   a. Numeric : It contains the datatypes such as integer, float, long and double

   b. Non numeric: it contains single character and string

Following table illustrates the basic datatypes in depth:

Sr. No	Datatype	Conversion Character	Storage Size	Declaration Method
1	Int	%d	2 bytes	Int a=10;
2	Float	%f	4 bytes	Float a=10.87;
3	Double	%ld	8 bytes	Double a=98.65
4	Char	%c	1 byte	Char a='N';
5	String	%s	As per the size	Char a[10]="Nisha";
Sr. No	**Datatype**	**Conversion Character**	**Storage Size**	**Declaration Method**
1	Int	%d	2 bytes	Int a=10;
2	Float	%f	4 bytes	Float a=10.87;
3	Double	%ld	8 bytes	Double a=98.65
4	Char	%c	1 byte	Char a='N';
5	String	%s	As per the size	Char a[10]="Nisha";

2 Derived: Derived datatypes include Arrays, pointer, structures, unions and functions.

a. Arrays are group of similar datatype values stored in contiguous memory locations sharing the same name. It can store primitive datatypes such as int, char, double, float etc. Arrays also can store the collection of derived datatypes such as structures, pointers etc. The array is a simplest data structure where each of its element can be accessed with the help of its index number.

b. A pointer is such a variable which stores address of another variable. This variable can be of any type such as int, char, array, function or any other pointer.

c. Structure is a user defined datatype which is like arrays only but the main difference between structure and array is that structures can store multiple values of different datatypes sharing same name. It can be declared using the keyword struct. Structures are normally used to represent records

d. Union is same as structure which allows to store multiple datatypes but in same memory location. Union can be defined with many members but one member can be accessed at a time. Union is an efficient way to use same memory location for multiple purpose.

2. Enumerated: These are the arithmetic types used to define variables which can assign constant integer values throughout the program. It makes the program easy to read and maintain. It can be defined using keyword enum. Common examples are days or week i.e. Sunday,

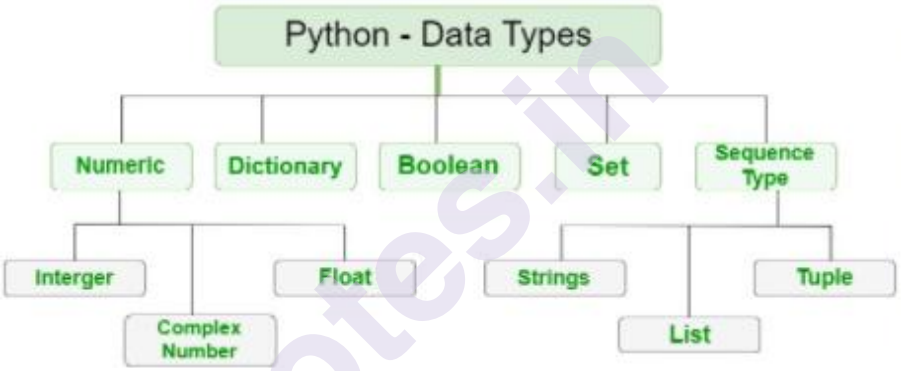
Monday,…. Saturday or compass directions such as North, South, East, West etc.

3. **Void:** These are the types where no value is specified that means it is empty datatype without a value. When we don't want to return any value to the calling function we use void keyword before the name of that function.

2.5 PYTHON DATATYPES V/S C DATATYPES

Python is a dynamically typed language so its not necessary to define type of a variable while declaring. The interpreter implicitly binds the value with its type.

Already we have seen datatypes in C language. However python provides following datatypes:



1. **Numeric:** In Python, numeric data type represent the data which has numeric value. Numeric value can be integer, floating number or even complex numbers. These values are defined as int, float and complex class in Python.

a. **Integers:** This value is represented by int class. It contains positive or negative whole numbers (without fraction or decimal). In Python there is no limit to how long an integer value can be.

b. **Float:** This value is represented by float class. It is a real number with floating point representation. It is specified by a decimal point. Optionally the character e or E followed by a positive or negative integer may be appended to specify scientific notation.

c. **Complex Numbers:** Complex number is represented by complex class. It is specified as (real part) + (imaginary part) I for example - 2+3j

Note: Type() function is used to determine the type of data type:

2. **Dictionary:** Python's dictionaries are kind of Hash table type. They work like associative arrays or hashes found in perl and consist of key-value pairs. A dictionary key can be almost any Python type, but

are usually numbers or strings values, on the other hand can be any arbitrary Python object

3. **Boolean:** Python boolean type is one of the built-in data types provided by Python, which represents one of the two values i.e. True or False. Generally, it is used to represent the truth values of the expressions. For example, 1== 0 is True whereas2<1 is False. We can evaluate values and variables using the Python bool() function. This method is used to return or convert a value to a Boolean value i.e., True or False, using the standard truth testing procedure.

4. **Set:** The collection of unique items that are not in order ae called as sets. { } braces are used to define a set and , comma is used to separate the set values. The items in set are unordered in nature. Duplicate values are not allowed in sets. Operations like union and intersection can be performed on two sets.

5. **Sequence Type:** In python sequence type can be categorized in 3 parts i.e. strings, list and tuple

a. **Strings:** A String is a sequence of Unicode characters. In Python, String is called str. Strings are represented by using Double quotes or single quotes. If the strings are multiple, then it can be denoted by the use of triple quotes “”” or ”’. All the characters between the quotes are items of the string. One can put as many as the character they want with the only limitation being the memory resources of the machine system. Deletion or Updation of a string is not allowed in python programming language because it will cause an error. Thus, the modification of strings is not supported in the python programming language.

b. **List:** An ordered sequence of items is called List. It is a very flexible data type in Python. There is no need for the value in the list to be of the same data type. The List is the data type that is highly used data type in Python. List datatype is the most exclusive datatype in Python for containing versatile data. It can easily hold different types of data in Python. t is effortless to declare a list. The list is enclosed with brackets and commas are used to separate the items.

c. **Tuple:** A Tuple is a sequence of items that are in order, and it is not possible to modify the Tuples. The main difference list and tuples are that tuple is immutable, which means it cannot be altered. Tuples are generally faster than the list data type in Python because it cannot be changed or modified like list datatype. The primary use of Tuples is to write-protect data. Tuples can be represented by using parentheses (), and commas are used to separate the items

2.6 TYPE CHECKING C V/S PYTHON

Type checking is nothing but checking that each operation should receive proper number of arguments with proper data type. There are basically 2

types of checking such as static type checking and dynamic type checking. However Python uses dynamic type checking and C uses static type checking

1. **Static type checking:** Static checking is done at complete time. Information needed at compile time is provided by declaration by language structures. The information includes

a. For each operation: The number, order, and data type of its arguments

b. For each variables: Name and data type of data object

c. For each constant: Name, data type and value

Advantages:

a. Compiler saves information: If the type of data is according to the operation then compiler saves that information for checking later operations which further no need of compilation

b. Checked execution paths: As static type checking includes all operations that appear in program statement, all possible execution paths are checked and further testing for type error is not needed. So no type tag on data objects on run time are not required and no dynamic checking is needed.

Disadvantages:

a. Declarations
b. Data control structures
c. Provisions of compiling separately some subprograms

2. **Dynamic type checking:** Dynamic type checking is done at run time and it uses concept of type tag which is stored in each data objects that indicates the data type of the object

Advantages:

a. It is much flexible in designing programs

b. In this declarations are not required

c. In this type may be changed during execution

d. In this program are free form most concern about data type

Disadvantages:

c. Difficult to debug: We need to check program execution paths for testing and in dynamic type checking, program execution path for an operation is never checked.

d. Extra Storage: Dynamic type checking need extra storage to keep type information during execution

e. **Hardware Support:** As hardware seldom support the dynamic type checking so we have to implement in software which reduces execution speed.

2.7 SUMMARY

C identifiers are the names of variables, functions, arrays, structures, unions, and labels.

Internal:

identifier and external identifier are the two types of indetifiers

Keywords are predefined reserved words which has special meaning defined by the compiler.

Literals are the constant values assigned to the constant variables

variables have the capacity to change whereas constants remains fixed.

C language works with 4 different types of datatypes ie basic, derieved, enumerated and void.

Type checking is nothing but checking that each operation should receive proper number of arguments with proper data type

C uses static type checking and python uses dynamic type checking

2.8 UNIT END QUESTIONS

1. Explain keyword with its list
2. Write the difference between variable and constant
3. write short note on c datatypes write the difference between static type checking and dynamic type checking

*****

3

C VARIABLES

Unit Structure

3.0 OBJECTIVES

- Understanding variable and its scope
- How to use datatypes considering their hierarchy
- Getting knowledge about type of declaration in C and Python

3.1 INTRODUCTION

A variable is simply something which is having the capacity to vary. Each and every variable is having its own datatype. This datatype can either be predefined or can be user defined. Specifically variables represent an objects that can be measured, counted, controlled or manipulated. However scope is that region or area of the program where the variables can be accessed after its declaration.

3.2 VARIABLE DECLARATION

In earlier chapter we have learnt that a variable is nothing but a storage location in which values are stored. Each variable is having specific type which determines the size and layout of the variable's memory. The names of variables can be composed of letters, alphabets, digits, and an underscore. Whereas uppercase and lowercase characters are considered as different from each other as C is a case sensitive language.

Here we will see how the variables can be declared:

```
int a=10;

float b=20.30;

char c='T';
```

double d=66.678;

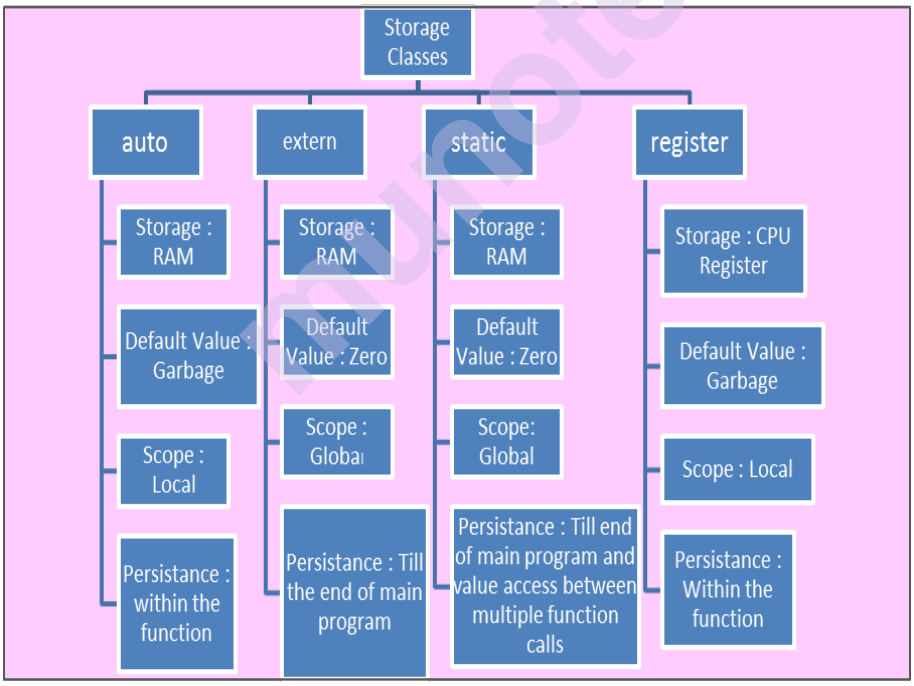
char str[10]="nisha";

Here variables a,b,c,d and str are declared as well as defined with datatypes such as int, float, char double and string and with values 10, 20.30, T, 66.678 and nisha respectively.

A variable declaration provides the surety to the compiler that a variable is in existence with a given type and name so that the compiler can proceed for further compilation process without knowing the completer details of the variable. At the time of compilation process only the variable definition gets its meaning which is required for linking of the program.

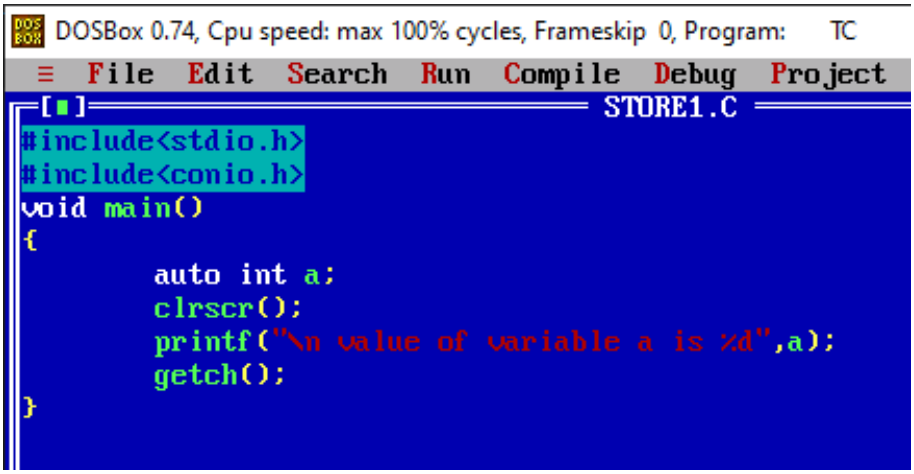
3.3 SCOPE OF VARIABLES

These variables can be declared in one file and can be accessed in multiple blocks, functions or files just we need to define the scope of the variable. To know the scope of variable first it is required to know where the variable is exactly got stored. In this case storage classes play a vital role. These storage classes in C are used to determine the visibility, persistence, initial value and memory location of a variable. There are 4 different types of storage classes in C that are auto, extern, static and register. Following diagram gives more clear picture about the storage classes.

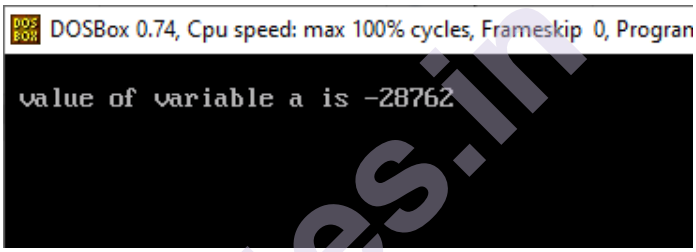


1. **Automatic:** If we do not define any variable using any specific storage class then by default it is automatic storage class. They are stored in The scope and visibility of these types of variables is limited to the block in which that variable is defined. The initial by default value can not be predicted means it is a garbage value. The keyword used for variable declaration is auto.

Declaration method:

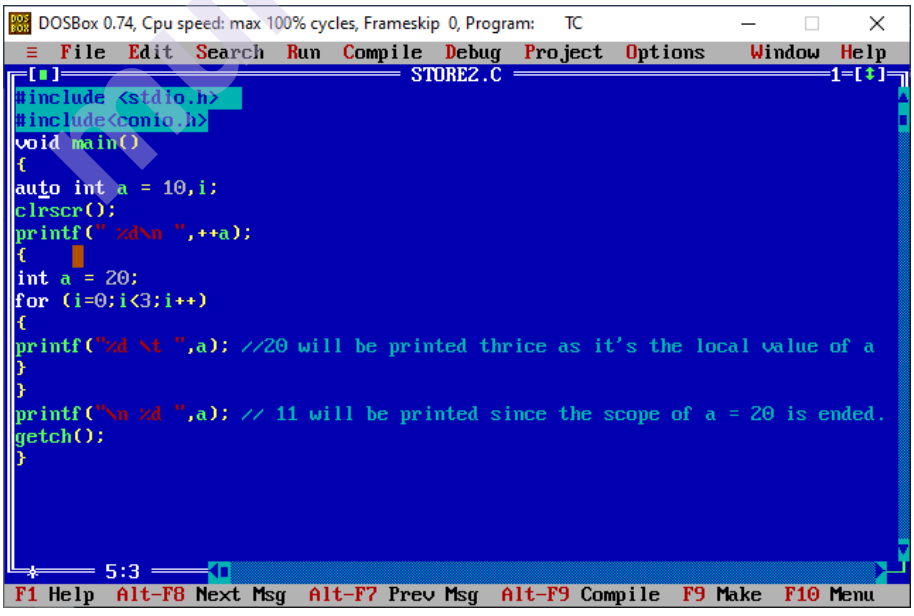


Output:



Here we can see that value of variable a is a garbage value.

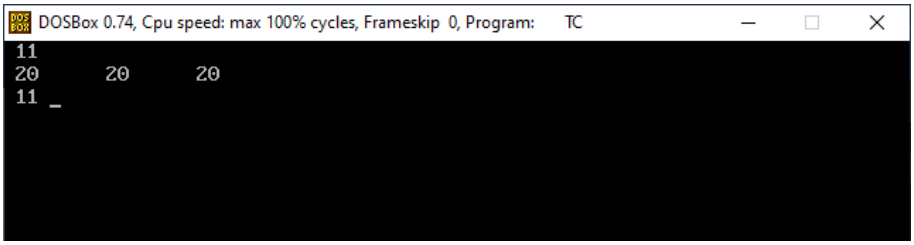
Now we will see scope of automatic type of variables with the help of following example:



In this program initially the value of variable a is defined as 10. Whereas the first printf statement will allow to print the value of a by incrementing it by 1 and hence it will show the value of a as 11. Here is the scope of variable a which we have declared as 10 is started. However in the next

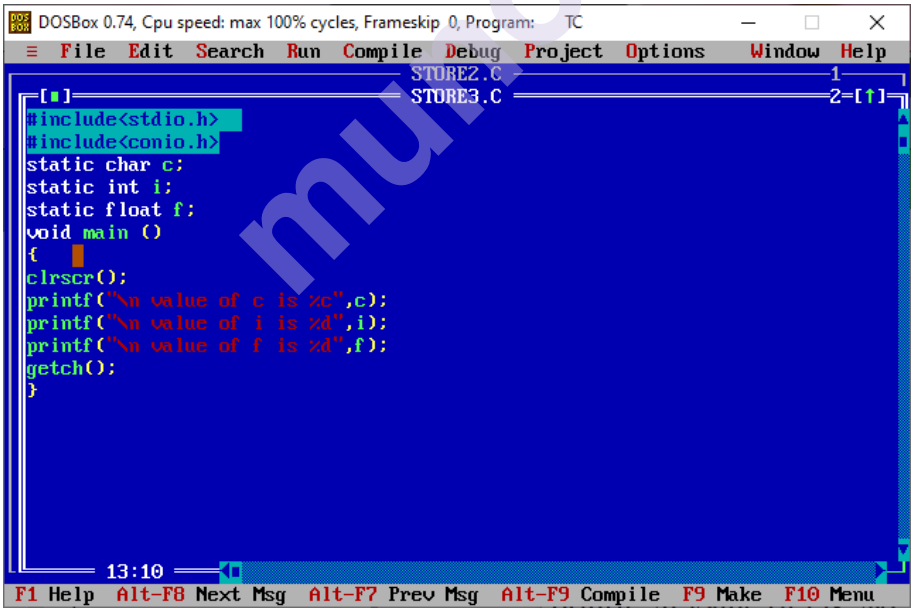
line when we are writing int a=20 and then we are opening a for loop and printing the value of a three times and then closing the for loop. In this case the scope of variable a having value 20 will be in persistence within the block of for loop only. As soon as we come out of for loop once again the variable a will regain its original value as 11. So we will be able to see the following output.

Output:



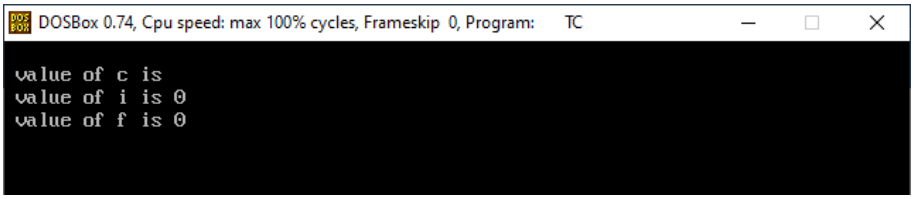
2. **static:** The variables which are declared with static storage class are allowed to hold their values in multiple function calls. A static variable can be declared multiple times but can be assigned with any value only once. The values static local type of variables are accessible only in the function or block in which they are defined. Whereas the values of static global type of variables are accessible in the whole file in which it is defined. The default initial value of integer type of variables is zero and for other types it will be null.

Example:

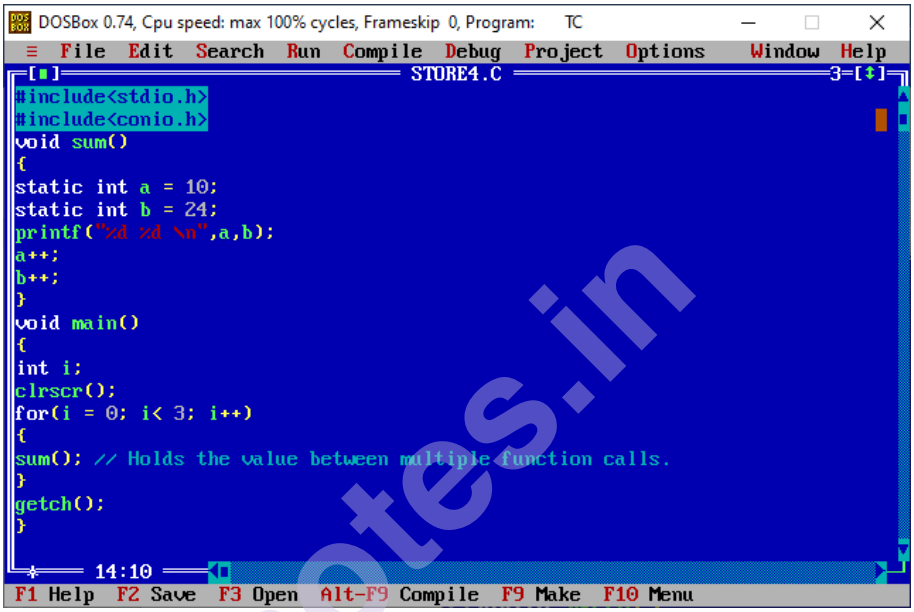


In this program we have declared 3 different static global variables i.e. c, i, f where c is a non numeric type of variable and i and f are numeric types of variables. So whenever we will see their default initial value, for numeric type it will be zero and for non numeric type it will be null.

Output:

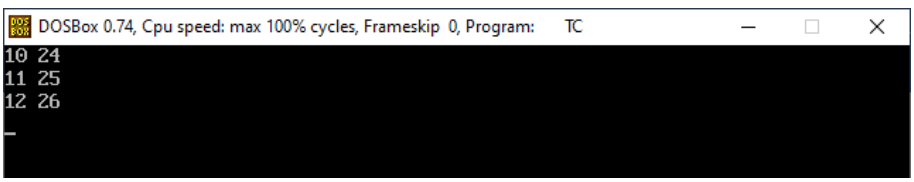


The below example will illustrate the scope of variable and how it holds the value in multiple function calls



In this program there are two functions i.e. sum() and the another one is main(). Now in main() function we have called sum() function 3 times using for() loop. However in the definition of function sum() we have defined the initial values of variable a and b as 10 and 24 respectively. So when we are saying that the function sum() is called 3 times in that case for the first call of the sum() function the value of a and b will be printed as 10 and 24 and will be incremented by 1. Now when the function sum() is called second time, the incremented values of a and b will be considered as the static type of variables are having the capacity to hold the values between different function calls and hence in the second iteration the values of variables a and b will be printed as 11 and 25 and then these values will again be incremented by 1 and in the third iteration the values of a and b will be printed as 12 and 26.

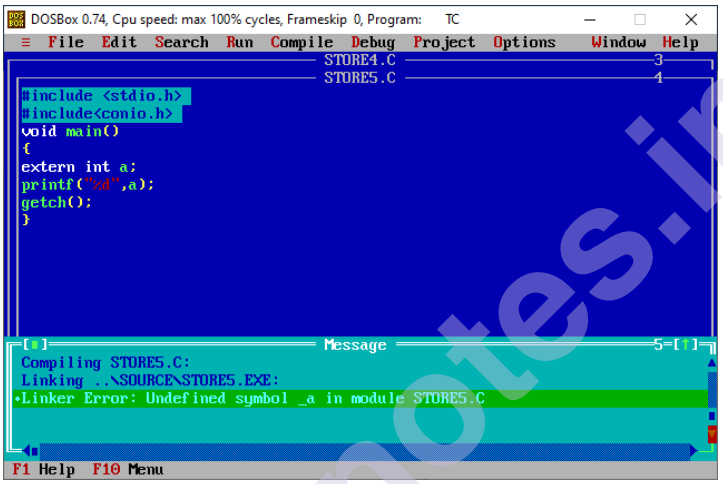
Output:



3. **Extern:** The external storage class tells the compiler that the variable declared using extern are having external linkages elsewhere in the program. These type of variables are global and cannot be initialized within any block or function. They can be declared multiple times but can be initialized only once. The default initial value of extern numeric types of variables is zero and for nonnumeric types it is null. 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, but if it is not initialized then the compiler shows an error.

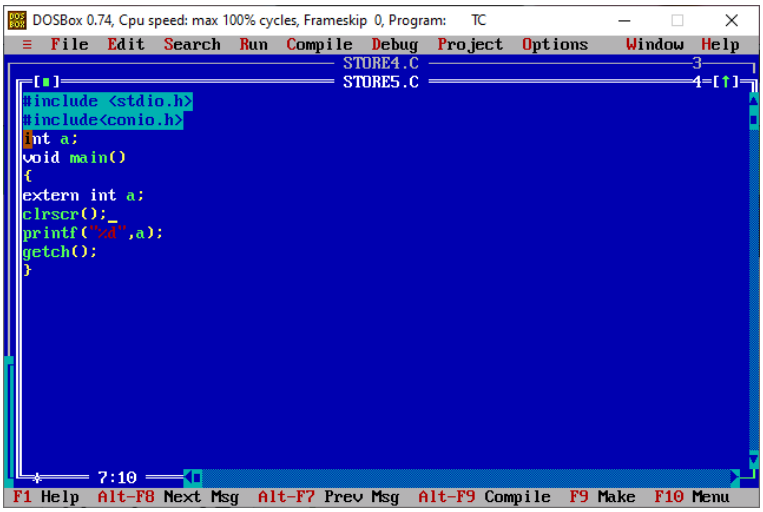
Following examples will illustrate the working of extern types of variables:

Example 1:



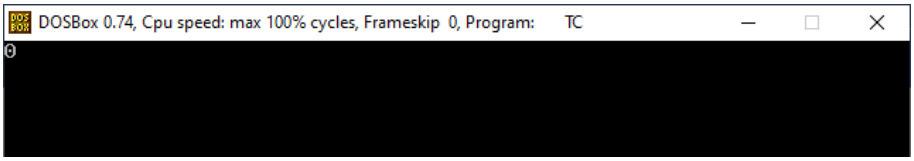
In the above example in the message section it is showing the error that a is not defined because extern type of variables should be declared globally i.e. outside the main() however in the given program we have declared it inside the main() and hence compiler cant recognize the definition of variable a anywhere.

Example 2:

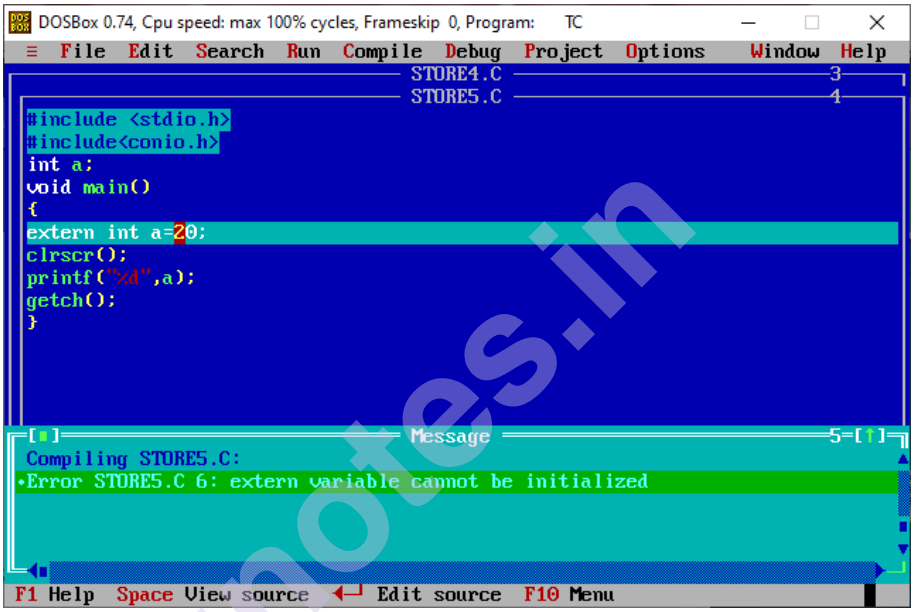


Now here since the variable a is declared outside the main() it will be recognized and since the variable a is of integer type it will show its default value as 0

Output:

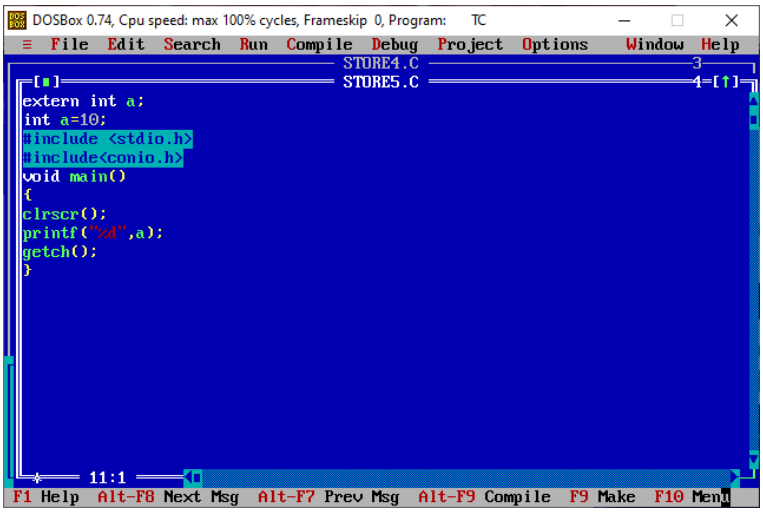


Example 3:



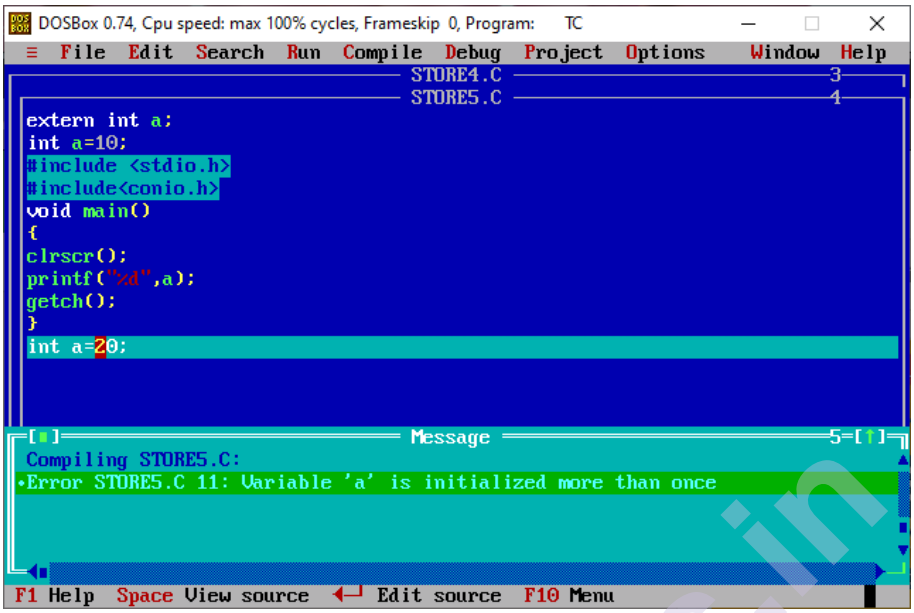
In this example we are trying to initialize the variable a with value 20 inside the main() which is not valid and hence it will show the error that extern variable can not be initialized.

Example 4:



In this program the variable a is declared globally with value 10 and will print the value as 10.

Example 5:

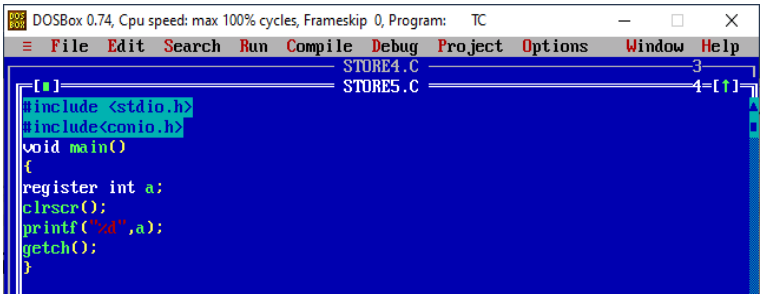


However when we will compare example no 4 and example no 5 we can see that when we want to define variable a with value 20 after main() function it is showing us an error as extern types of variables can not be defined multiple times.

4. **Register:** We have seen all the other types of variables i.e. auto, static and extern are getting stored in computer's memory whereas the variable declared with register used to get store in cpu register depending upon the size of the memory remaining in the CPU. These types of variables cannot be dereferenced means we can not use & (address operator) for such type of variables. Since these types of variables are stored in CPU register their access time is faster. The keyword register is used to store the variable in CPU register however its compiler's choice whether to store then in register or not. The default initial value of register variables is a garbage value

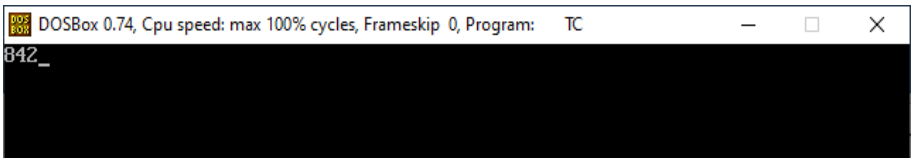
Following example will illustrate the working of register types of variables:

Example 1:



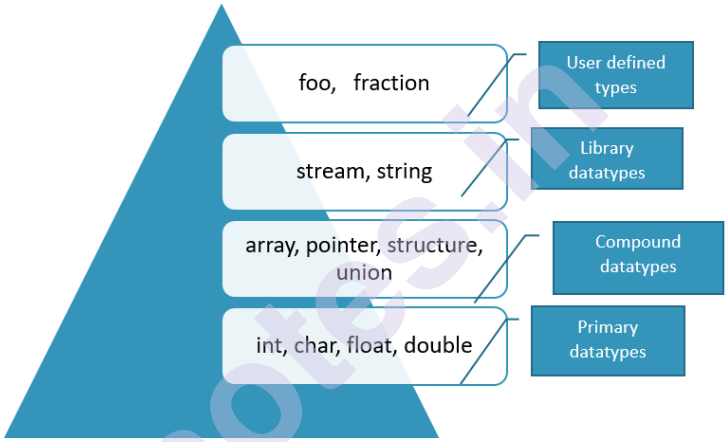
In this program it will show the garbage value of a.

Output:

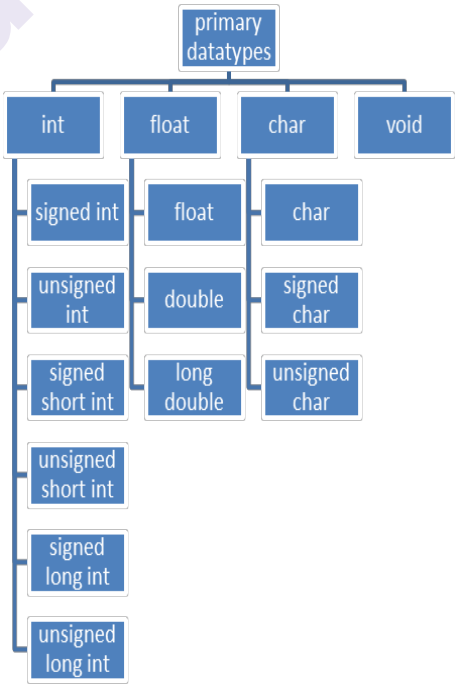


3.4 HIERARCHY OF DATATYPES

Already we have seen the topic datatype in earlier chapter. C provides 4 basic categories of datatypes such as basic, derived, enumeration and void. However the hierarchy of these datatypes can be explained with the help of following diagram.



The hierarchy of primary datatypes will be as follows:



Following table will help us to understand more about all the above mentioned datatypes:

Data type	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	

3.5 TYPE OF DECLARATION C V/S PYTHON

Type declaration in C:

Type definition is a feature of C programming language which allows a programmer to define an identifier which represents existing data types of a variable. The user defined identifier can be used later in the program to declare variables.

Syntax:

```
typedef type identifier;
```

Here, type is an existing data type and identifier is the name given to the data type.

Example:

```
typedef int age;
typedef float weight;
```

Here we can observe that age represents integer type of data and weight represents float type of data.

```
age boy1,boy2;
weight b1,b2;
```

Here, boy1 and boy2 are declared as as integer data type and b1 & b2 are declared as floating integer data type.

The main advantage of using user-defined type declaration is that we can create meaningful data type names for increasing the readability of a program.

Another user-defined data type is enumerated data type. The general syntax of enumerated data type is:

```
enum identifier {value 1,value 2,...value n};
```

Here, identifier is a user-defined enumerated data type which can be used to declare variables that can have one of the values enclosed within the braces. The values inside the braces are known as enumeration constants. After this declaration, we can declare variables to be of this 'new' type as:

```
enum identifier v1, v2, ... vn;
```

The enumerated variables v1, v2, … vn can only have one of the values value1, value2, … valuen. The following kinds of declarations are valid:

```
v1=value5;
v3=value1;
```

User-defined Type Declaration Example:

```
enum mnth {January, February, ..., December};
enum mnth day_st, day_end;
day_st = January;
day_end = December;
if (day_st == February)
day_end = November;
```

The compiler automatically assigns integer digits begriming with 0 to all the enumeration constants. That is, the enumeration constant value1 is assigned 0, value2 is assigned 1, and so on. However, the automatic assignments can be overridden by assigning values explicitly to thee enumeration constants.

For example:

```
enum mnth {January = 1, February, ..., December};
```

Here, the constant January is assigned value 1. The remaining values are assigned values that increase successively by 1.

The definition and declaration of enumerated variables can be combined in one statement. For example;

```
enum mnth {January, ... December} day_st, day_end;
```

Type declaration in Python:

Now we all know that in C variables can be declared as:

```
Int x,y,z;
```

Whereas in python it can be declared as:

```
X=0
Y=0
Z=0
```

In python types of variables and functions can be declared in following way

```
explicit_number: type
```

or,

```
def function(explicit_number: type) -> type:
  pass
```

Following example will illustarate how to use static type checking in python:

```
from typing import Dict
def get_first_name(full_name: str) -> str:
    return full_name.split(" ")[0]
fallback_name: Dict[str, str] = {
    "first_name": "UserFirstName",
    "last_name": "UserLastName"
}
raw_name: str = input("Please enter your name: ")
```

```
first_name: str = get_first_name(raw_name)

# If the user didn't type anything in, use the fallback name
if not first_name:
    first_name = get_first_name(fallback_name)
```

```
print(f"Hi, {first_name}!")
```

3.6 SUMMARY

A variable is simply something which is having the capacity to vary and it represents an objects that can be measured, counted, controlled or manipulated.

A variable declaration provides the surety to the compiler that a variable is in existence with a given type and name

Storage classes allows to understand the scope of variables

There are 4 types of storage classes such as auto, static, extern and register.

C datatypes work with a specific hierarchy and includes primitive datatypes, compound datatypes, library datatypes and user defined datatypes

3.7 UNIT END QUESTIONS

1. Write short note on variables
2. Explain datatypes in c in detail
3. Explain hierarchy of datatypes in c
4. Write the difference between type checking in c and python

Book References

Let us C – Yashwant P Kanetkar – BPB Publication

Web References

https://www.javatpoint.com/data-types-in-c

https://www.tutorialspoint.com/cprogramming/index.htm

*****

4

TYPES OF OPERATORS

Unit Structure

4.0 OBJECTIVES

This chapter would make you understand the following concepts:

- To design programs to perform basic calculations.
- To compare two entity and implement decision making order of evaluation of operators.

4.1 TYPES OF OPERATORS

C Programming language supports a set of built-in operators. An operator is a symbol that operates on the operands and instructs the compiler to perform certain operations. Operands can be any variables that hold value.

The various types of Operators are as follows:

1. Arithmetic Operator
2. Relational Operator
3. Logical Operator
4. Compound Assignment Operator
5. Increment/Decrement Operator
6. Conditional & ternary Operator
7. Bitwise & comma operator

4.1.1 Arithmetic Operator:

This operator supports performing basic mathematical operations. Following are the operators used for arithmetic operations:

Symbol	Description	Example
Assume a=10 and b=5		
+	Addition of two operand values	a+b Output: 15
-	Subtraction of two operand values	a-b Output: 5
*	Multiplication of two operand values	a*b Output: 50
/	Division (Quotient) of two operand values	a/b Output: 2
%	Modulus (Remainder) of two operand values	a%b Output: 0 (Since a is divisible by b)

Example:

Program to calculate Simple Interest

```
#include<stdio.h>

void main()

{

int p, n;

float rate, Simple_interest;

p=10000;

n=10;

r=5.5;

Simple_interest = (p* n * r)/100;

printf("Simple Interest = %f",Simple_interest);

getch();

}
```

4.1.2 Relational Operator:

Relational operator is also known as Comparison operator. It performs comparison between two operand values:

Operator	Description	Example
Assume a=10 and b=5		
==	Check if two operand values are equal	a==b Output: false
!=	Checks if two operands values are not equal	a!=b Output: true
>	Checks if left operand value is greater than right operand value	a>b Output: true
<	Checks if left operand value is smaller than right operand value	a<b Output: false
>=	checks if left operand value is greater than or equal to right operand value	a>=b Output: true
<=	Check if left operand value is smaller than or equal to right operand value	a<=b Output: false

Example:

```
#include<stdio.h>
void main()
{
int a, b;
a=10;
b=20;
printf("Value = %d", a<=b);
printf("Value = %d", a==b);
printf("Value = %d", a>b);
printf("Value = %d", a!=b);
getch();
}
```

Output:

Value = 1

Value = 0

Value = 0

Value = 1

4.1.3 Compound Assignment Operator:

Assignment operators in C language are as follows:

Operator	Description	Example
=	assigns values from right side to left side operand	a=10
Shorthand Assignment Operators		
+=	adds right operand value to the left operand value and assign the result to left operand	a+=b is same as a=a+b
-=	subtract right operand value to the left operand value and assign the result to left operand	a-=b is same as a=a-b
*=	multiply right operand value to the left operand value and assign the result to left operand	a*=b is same as a=a*b
/=	divide right operand value to the left operand value and assign the result to left operand	a/=b is same as a=a/b
%=	calculate modulus right operand value to the left operand value and assign the result to left operand	a%=b is same as a=a%b

Example:

```
#include<stdio.h>

void main()

{

int n;

n=100;

n+=10;

printf("Value of n = %d",n);

getch();

}
```

Output:

Value of n = 110

4.1.4 Conditional & ternary Operator:

The conditional operator is also known as Ternary Operator or ?: operator
It is similar to the C language decision making, the if condition.

Syntax:
condition ? true statement : false statement

Explanation:

- condition is specified in the if followed by question mark "?". Based on the condition, the statements are executed.
- True statements are executed (statement after question mark “?”) only if the condition returns true as the boolean value.
- False statements are executed (statement after question mark “:”) only if the condition returns false as the boolean value.

Example:

```
#include<stdio.h>

void main()

{

int n=30;

if(n%3==0) ? printf(“Number is divisible by 3”) : printf(“Number is not divisible by 3”);

getch();

}
```

Output:

Number is divisible by 3

4.1.5 Increment/Decrement Operator:

C programming supports increment operator ++ and decrement operator --

These two operators operate on a single operand only

- Increment ++ increases the value of the operand by 1
- Decrement -- decreases the value of operand by 1

Example:

```
#include<stdio.h>

int main()

{
```

```
int a,b;
a=10;
b=20;
printf("Value of a = %d", a++);
printf("Value of b = %d", b--);
return 0;
}
```

Output:

Value of a = 11

Value of b = 19

4.1.6 Logical Operator:

Operator	Description	Example
Assume a=1 and b=0		
&&	Logical AND If both the conditions are true then it returns true. If either of the condition is false then it returns false	(A && B) is false
||	Logical OR If both or either of the conditions are true then it returns true. If both the conditions are false then it returns false	(A || B) is true.
!	Logical NOT It gives the negation of an expression ie. if an expression is true then it returns false and vice versa.	!(A && B) is true.

Example:

```
#include<stdio.h>
void main()
{
int n;
n=35;
```

```
if (n%5==0 && n%7==0) ? printf("Number is divisible by both") : printf("Number is divisible by none");

}
```

Output:

Number is divisible by both

4.1.7 Bitwise & comma operator:

Bitwise operator operates on bits and performs bit by bit operation:

Operator	Description
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
<<	left shift
>>	right shift

a	b	a&b	a|b	a^b
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

4.2 PRECEDENCE & ORDER OF EVALUATION

- Operator precedence defines the order of an expression to be evaluated.
- Some operators have higher precedence than other operators. Example, the addition operator has a higher precedence than the subtraction operator.
- The operator with higher precedence is evaluated first and the operator with lower precedence is evaluated.
- For example, x = 10 + 5 * 2; here, x is assigned value as 20 and not 30 because multiplication operator (*) has a higher precedence than addition operator (+)

- Hence, first 5*2 is calculated and then it's added to 10.
- In the table, the operators with the highest precedence appear on the top of the table and those with the lowest precedence appear at the bottom. First the higher precedence operators will be evaluated and then the lower precedence operator.

In the table, associativity determines the precedence value of the list of operators under the categories:

Type	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left

4.3 STATEMENT & EXPRESSIONS

Expressions:

In any programming language, an 'expression' is basically a combination of variables and operators that are interpreted by the compiler.

For example:

c=a+b

a<=b

a++

Here, a and b are the variables having some values and +, <=, ++ are the operators which manipulate the values of the variable. And the complete unit is called an expression.

Statements:

A 'statement' is a standalone unit for execution.

For Examples

A statement can be the jump statements; return, break, continue, and goto.

a=10

4.4 AUTOMATIC & EXPLICIT TYPE CONVERSION

Type conversion means converting one operand of a data type into another one. It is also called type casting or type conversion in C language

C programming provides two types of type conversion:

1. Implicit type (Automatic) casting
2. Explicit type casting

1. Implicit type (Automatic) casting:

In Implicit type conversion, the operand of one data type is converted into another data type automatically but an operand of lower data type is converted into a higher data type automatically.

Example:

```
#include<stdio.h>

int main(){

    //initializing variable of short data type

    short a=10;

    int b; //declaring int variable
```

```
    b=a; //implicit type casting
    printf("Value of a = %d\n",a);
    printf("Value of b = %d\n",b);
    return 0;
}
```

Output:

Value of a = 10

Value of b = 10

2. Explicit type casting:

In Explicit type casting, one datatype is forcefully type casted to another datatype.

Example:

```
#include<stdio.h>
 int main()
{
   double x = 1.2;
  // Explicit conversion from double to int
   int sum = (int)x + 1;
     printf("sum = %d", sum);
   return 0;
}
```

Output:

sum = 2

4.5 MISCELLANEOUS PROGRAMS

Practice Program:

1. Write a program to check whether a number is even or odd

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

```
n=35;

if (n%2==0) ? printf("Number is even") : printf("Number is odd");

}
```

Output:

Number is even

2. Write a program to check whether the alphabet is a vowel or not

```
#include<stdio.h>

void main()

{

char ch;

ch='a';

if (ch=='a' || ch=='e'|| ch=='i'|| ch=='o'||ch=='u' ) ? printf("It is a vowel")
: printf("It is not a vowel");

}
```

Output:

It is a vowel

4.6 SUMMARY

Operators are defined as symbols that operate on operands which help us to perform specific mathematical and logical computations.

C language works with majorly 7 different types of operators namely Arithmetic, Relational, Logical, Assignment, Increment/Decrement, Conditional and Bitwise operator.

The precedence of operators matters in the grouping and evaluation of expressions.

First the expressions of higher-precedence operators are evaluated and if there are several operators having equal precedence, then they are evaluated from left to right.

Typecasting is converting one data type into another one.

C language uses two types of typecasting namely implicit and explicit conversion.

4.7 UNIT END QUESTIONS

1. Explain Arithmetic operator
2. How can we compare two values?
3. Write a note on assignment operator
4. Explain conditional operator
5. How can we increment/decrement value by 1
6. Explain the precedence of operator
7. State the difference between expression and statement

*****

5

CONTROL STATEMENTS FOR DECISION MAKING

Unit Structure

5.0 OBJECTIVE

This chapter would make you understand the following concepts:

- To design programs involving decision structures, loops
- Working of decision making
- How to execute number of statements repeatedly efficiently

5.1 BRANCHING/DECISION MAKING

Decision making is about checking the condition and based on the condition, a decision is made.

The decision making statements supported by c are as follows:

- if statement
- if else statement

- Nested if else statement
- switch statement

5.1.1 if statement:

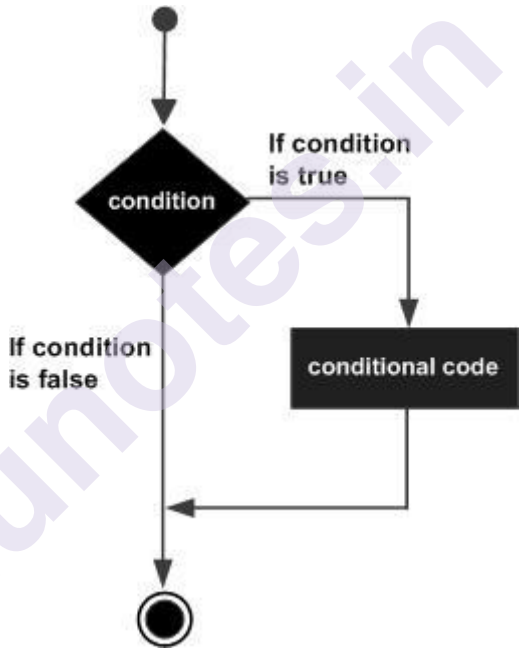
Syntax:

```
if(condition)
{
  statement(s);
}
```

Explanation:

If the condition specified is true then the statement(s) written in curly brackets will be executed otherwise nothing will be printed

Flow Diagram:



Example:

Program to check whether a number is equal to zero or not

```
void main()
{
int n;
n=0;
if(n==0)
{
printf("Number is zero");
}
getch();
```

}

The above code on execution gives following output

Number is zero

5.1.2 if else statement:

An if statement is followed by an else statement, which will be executed when the Boolean result of the condition is false.

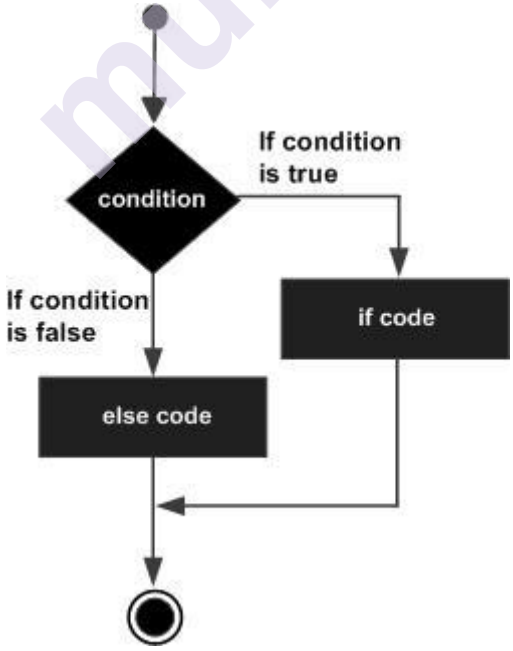
Syntax:

```
if(condition)
{
  If block statement(s;
}
else
{
 else block statement(s);
}
```

Explanation:

If the condition specified is true then the if block statement(s) written in curly brackets will be executed otherwise the control will go to the else and else block statement(s) will be executed

Flow Diagram:



Example:

Program to check whether a number is even or odd

```
void main()
{
int n;
n=20;
if(n%2==0)
{
printf("Number is even");
}
else
{
printf("Number is odd");
}
getch();
}
```

The above code on execution gives following output

Number is even

5.1.3 Nested if else statement:

nested if-else statements means one if else block within another if or else or both. And based on the conditions the statement(s) will be executed.

Syntax:

```
if( Condition 1 )
  {
        if(condition 2)
{
        Statement 1;
    }
      else
      {
      Statement 2;
      }
}
else
{
      Else block statement(s);
}
```

Explanation:

If condition 1 is true then it enters the curly brackets and checks for condition 2.

- If condition 2 is also true then it executes statement 1.
- But if condition 2 is false then it executes statement 2

If condition 1 is false then it goes to else and executes else block statements

Note: else block can also have if else statements within it.

This is called as nested if else statements

Example:

```
#include <stdio.h>
 int main () {
   int a = 35;
      if( a%5 == 0 )
   {
                if(a%7==0)
{
                printf("Number is divisible by both 5 and 7");
     }
        else
        {
        printf("Number is only divisible by 5");
        }
}
else
{
                if(a%7==0)
{
                printf("Number is divisible by Only 7");
     }
        else
        {
        printf("Number is neither divisible by 5 nor by 7");
        }
}
   return 0;
}
```

The above code on execution gives following output

Number is divisible by both 5 and 7

5.1.4 Switch statement:

A switch statement allows a variable value to be checked against a list of values for equality. Every value here is called as a case, and the variable value is checked for each switch case.

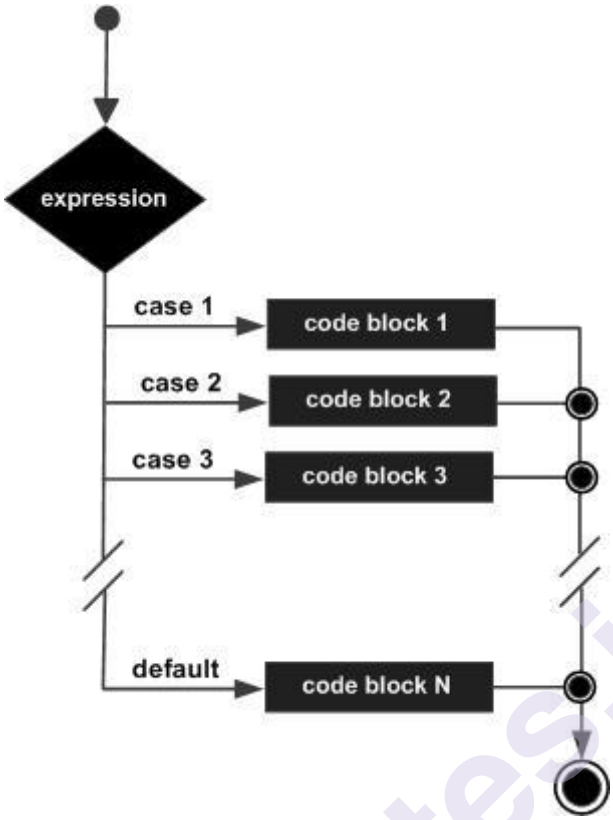
Synta:

```
switch(expression)
{
  case constant-expression  :
     statement(s);
     break;
  case constant-expression  :
     statement(s);
     break;
.
.
.
.
  default :
  statement(s);
}
```

The following rules are to be applied for switch statement:

- Within a switch statement, we can write any number of case statements.
- The datatype of constant-expression of a case should be same as the variable in the switch
- When the variable value is equal to a case then the corresponding statements will be executed until a break statement.
- The switch case is terminated as the break statement is encountered, and the control moves to the next line following the switch statement.
- If neither of the case values matches, then the default statement is executed.

Flow Diagram:



Example:

```
#include <stdio.h>
 int main ()
{
  char grade = 'A';
  switch(grade) {
     case 'A' :
       printf("Excellent!\n" );
       break;
     case 'B' :
              printf("Very Good\n" );
       break;
     case 'C' :
       printf("Good\n" );
       break;
```

```
    case 'D' :
      printf("You have passed\n" );
      break;
    case 'F' :
      printf("Better try again\n" );
      break;
    default :
      printf("Invalid grade\n" );
  }
    return 0;
}
```

The above code on execution gives following output

Excellent

5.2 LOOPS

In computer language, a loop is a sequence of instructions that has to be repeated until the condition is true.

C programming supports 3 types of loops, namely:

1. while loop
2. for loop
3. do while loop

5.2.1 while loop:

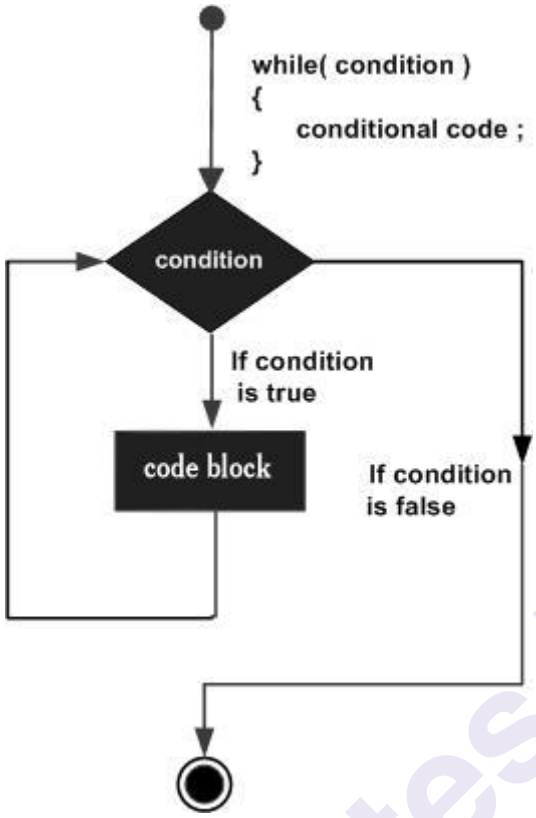
In C programming while loop, the set of statements are executed repeatedly till the condition is true. statement(s) can be a single statement or a block of statements.

As the condition becomes false, the control moves to statement after the while loop block

Syntax:

```
while(condition)
{
statement(s);
}
```

Flow Diagram:



Example:

Program to print first 10 natural numbers

```
#include<stdio.h>
void main()
{
int i;
printf("First 10 natural numbers are:");
i=1;
while(i<=10)
{
printf("%d\n",i);
i++;
}
getch();
}
```

The above code on execution gives following output:

1

2

3

4

5

6

7

8

9

10

5.2.2 for loop:

In C programming for loop, the set of statements are executed repeatedly till the condition is true. statement(s) can be a single statement or a block of statements.

As the condition becomes false, the control moves to statement after the while loop block

Syntax:

for (initialization; condition; increment/decrement)

{

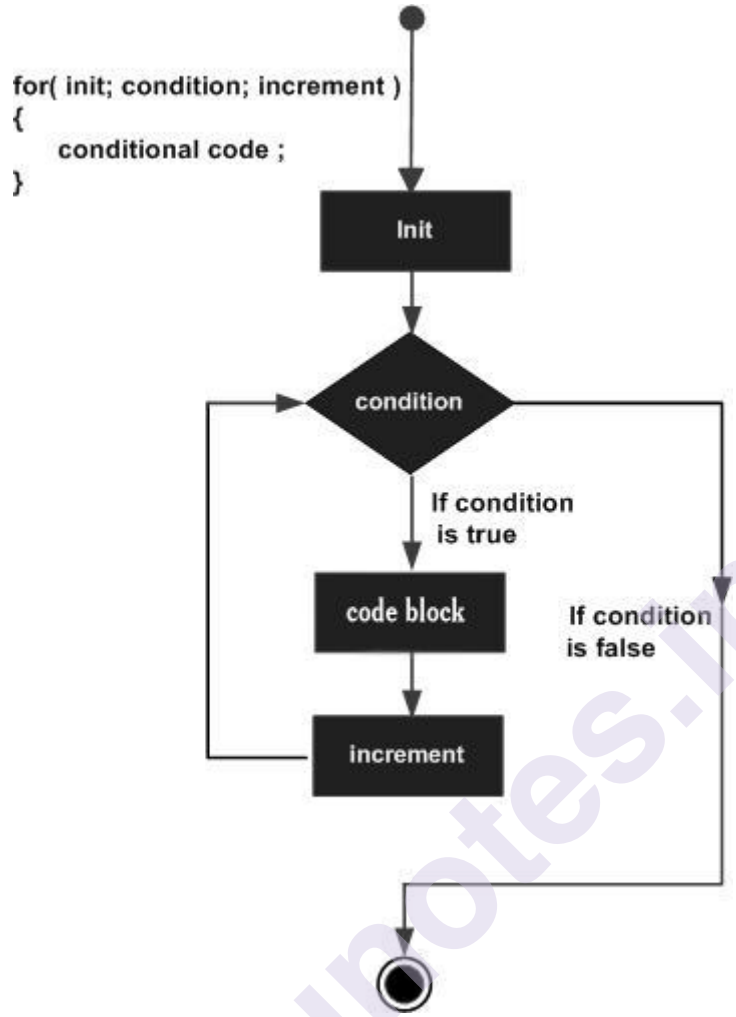
statement(s);

}

The flow of control in a 'for' loop:

- initialization is the first step to be executed and it's done only once. It tells you to declare and initialize variables for the loop.
- Next is the condition that is evaluated. If the condition is true then the body of the loop is executed and If the condition is false, the body of the loop does not execute and the control jumps to the statement just after the 'for' loop.
- As the body of the 'for' loop is executed, the control moves to the increment/decrement statement where accordingly increment/decrement is done.
- Again the condition is evaluated with the new value of the variable. again, if the condition is true, the loop executes and the process keeps on repeating itself until the condition is true.
- As the condition becomes false, the 'for' loop process is terminated.

Flow Diagram:



Example:

Program to print even numbers between 1 to 25

```
#include<stdio.h>
void main()
{
    int i;
    printf("Even numbers between 1 to 25 are:");
    for(i=1; i<=25; i++)
    {
    if(i%2==0)
    {
        printf("%d\n",i);
```

```
        }
        }
        getch();
}
```

The above code on execution gives following output:

2
4
6
8
10
12
14
16
18
20
22
24

5.2.3 do while:

Unlike while and for loops, do while loop first executes the statement and then evaluates the condition as the condition is placed at the bottom.

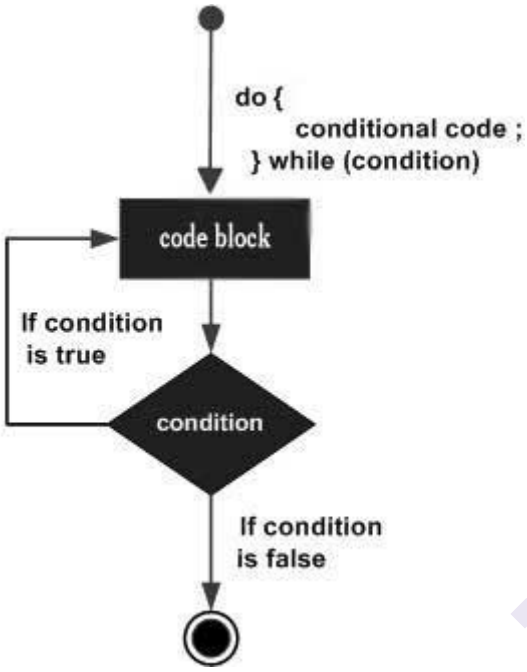
Next iteration will depend on the condition whether true or false. If the condition is true then the statement(s) will be executed again till the condition is true. If the condition is false, the loop is terminated.

With this it implies that the block of statement(s) is executed at least once even if the condition is not true at the first instance itself.

Syntax:

```
do
{
statement(s);
} while (condition);
```

Flow Diagram:



Example:

```
#include <stdio.h>

int main()

{

    int j=0;

    do

    {

        printf("Value of variable j is: %d\n", j);

        j++;

    }while (j>3);

    return 0;

}
```

The above code on execution gives following output:

Value of variable j is 1

5.2.4 Differentiate between while and do while loop:

while	do-while
It is an entry controlled loop	It is an exit controlled loop
Condition is evaluated first and then accordingly statement(s) are executed	Statement(s) is executed at least once and then condition is evaluated
Statement may or may not be executed as it depends on the condition to be true	Statement is executed at least once irrespective of the condition
There is no semicolon at the end of while(condition)	It is mandatory to add a semicolon at the end of while (condition)

Syntax:
while (condition)
{
Block of loop;
}
	Syntax:
	do
	{
	Block of loop;
	}while (condition);

5.3 JUMP STATEMENTS

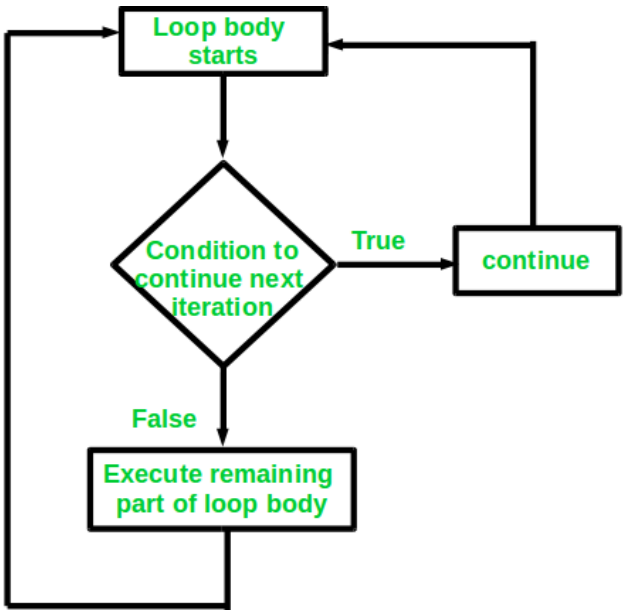
Jump statements are used to control the flow of the program if the specified conditions are satisfied. It is basically used to either continue or terminate the loop.

C Programming supports four jump statements: continue, break, return, and goto.

5.3.1 continue statement:

It is used if we want to execute some portion of the loop skipping some parts based on the condition; instead of terminating the whole loop.

continue statement is practically used with decision-making statements such as nested switch, or if else statement which may be written inside any of the loop.



Example:

```
#include<stdio.h>

int main()

{

   for (int i = 1; i < 10; i++) {

        if (i == 5)

          continue;

      printf("%d\n",i);

   }

   return 0;

}
```

Output:

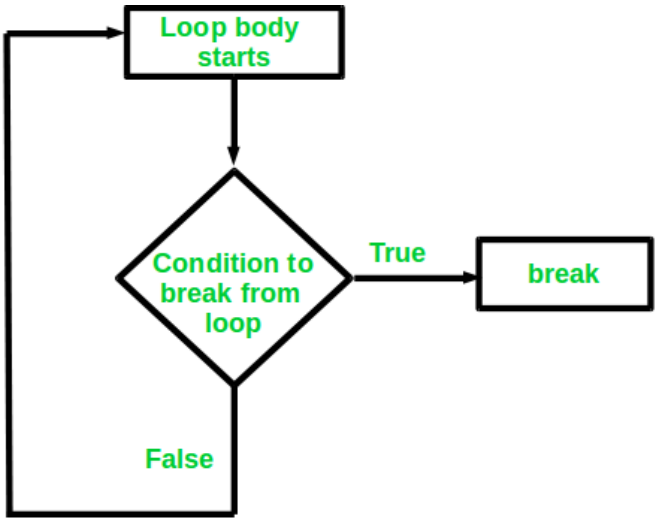
1 2 3 4 6 7 8 9

5.3.2 break statement:

It is used if we want to forcefully break and terminate a loop once the condition is satisfied.

Break statement is practically used with decision-making statements such as nested switch, or if else statement which may be written inside any of the loop.

Flowchart:



Example:

```
#include<stdio.h>

int main()

{

   for (int i = 1; i < 10; i++) {

        // Breaking Condition

      if (i == 5)

         break;

      printf("%d\n",i);

   }

   return 0;

}
```

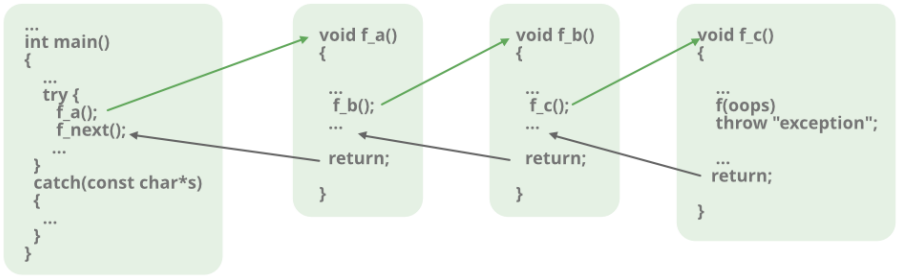
Output:

1 2 3 4

5.3.3 Return:

It used to move the control to the original source.

Every function returns a value unless the function is declared as void().



Example:

```
#include<stdio.h>

int main()

{

   for (int i = 0; i < 10; i++) {

        // Termination condition

      if (i == 5)

         return 0;

      printf("%d\n",i);

   }

      return 0;

}
```

Output:

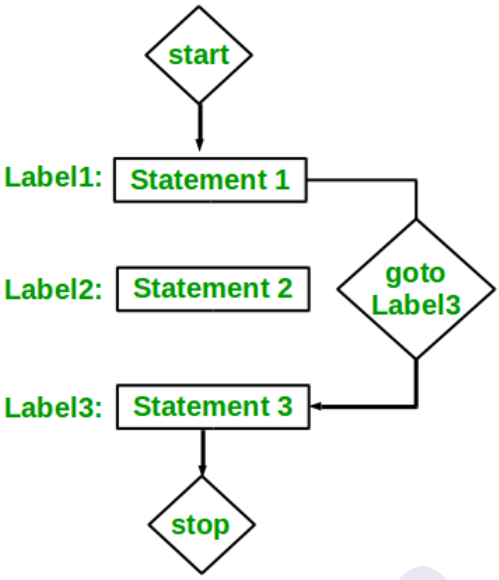
0 1 2 3 4

5.3.4 Goto statement:

This statement allows the control to jump directly to a specific part of the program through the use of labels.

The label statements can be written anywhere in the program.

Flowchart:



Example:

```
#include<stdio.h>
int main()
{
   int n = 4;
   if (n % 2 == 0)
      goto label1;
   else
      goto label2;
 label1:
   printf("Even");
   return 0;
 label2:
   printf("Odd");
}
```

Output:

Even

5.4 MISCELLANEOUS PROGRAMS

1. Program to print factorial of a number

```
#include<stdio.h>
void main()
{
int i, n, fact=1;
n=5;
i=1;
while(i<=n)
{
fact=fact*n;
i++;
}
printf("Factorial = %d", fact);
getch();
}
```

The above code on execution gives following output:

Factorial = 120

2. Program to calculate the sum of first n natural numbers

```
#include <stdio.h>
int main()
{
   int num, count, sum = 0;
   printf("Enter a positive integer: ");
   scanf("%d", &num);
   for(count = 1; count <= num; ++count)
   {
      sum += count;
   }
```

```
    printf("Sum = %d", sum);
    return 0;
}
```

The above code on execution gives following output:

Sum=55

5.5 UNIT END QUESTIONS

1. Explain switch case statement
2. Explain if statement
3. Write a note on nested if else statement
4. Explain if else statement
5. What are loops? What is the need of loops?
6. Explain for loop
7. Explain while loop
8. Explain do while loop
9. Differentiate between while and do while loop

Bibliography:

1. https://www.tutorialspoint.com/cprogramming/
2. https://www.guru99.com/
3. https://www.javatpoint.com/
4. https://www.programiz.com/c-programming/

*****

UNIT II

6

ARRAY

Unit Structure

6.1 OBJECTIVES

a. To understand the concept of the use of arrays in C language.

b. To learn different types of arrays like two dimensional and multidimensional arrays.

c. To understand the declarations and initialization of arrays using C.

d. To learn how to access array elements.

e. To understand the difference between C array, Python list and tuple.

6.2 INTRODUCTION

A **variable** is a memory location that stores a value. Like a box these values are stored. The value held in this variable can change, or be different. But each variable can only hold one item of **data or value**.

An array is a series of memory locations each of which holds a single item of data or values, but with each variable sharing the same name. All data in an array must be of the same **data type and not different**.

For example, a score table in a game needs to record ten scores. One way to do this is to have a variable for each score:

score_0

score_1

score_2

score_3

score_4

score_5

score_6

score_7

score_8

Score_9

This would be a fine, but there is another better way. It is very simple to keep all the related data under one name. by using an array.

Instead of having ten variables, each holding a score, there could be one array that holds all the related data:

score(9)

By using this array, all 10 data items can be stored in one place.

1. Arrays: (One and two dimensional):

An array in C/C++ or be it in any programming language is a collection of similar data items stored at contiguous memory locations and elements can be accessed randomly using indices of an array.

For example, arranging the percentage marks obtained by 100 students in ascending order. In such a case we have two options to store these marks in memory:

(a) Declare 100 variables to store percentage marks obtained by 100 different students, i.e. each variable containing one student's marks.

(b) Declare a one variable (called array or subscripted variable) capacity of storing or holding all the hundred values.

A formal definition of an array, An array is a collective name given to a group of 'similar quantities'. These similar quantities could be percentage marks of 100 students, or salaries of 300 employees, or ages of 50 employees.

For example, assume the following group of numbers, which represent percentage marks obtained by five students.

per = { 48, 88, 34, 23, 96 }

A Simple Program Using Array:

Code:

```
#include <stdio.h>
int main()
{
        int avg, sum = 0 ;
        int i ;
```

```
        int marks[30] ; /* array declaration */
        for ( i = 0 ; i <= 5 ; i++ )
        {
                printf ( "\nEnter marks " ) ;
                scanf ( "%d", &marks[i] ) ; /* store data in array */
        }
        for ( i = 0 ; i <= 5 ; i++ )
        sum = sum + marks[i] ; /* read data from an array*/
        avg = sum / 30 ;
        printf ( "\nAverage marks = %d", avg ) ;
}
```

Output:

```
Enter marks 10
Enter marks 40
Enter marks 30
Enter marks 25
Enter marks 14
Enter marks 45
Average marks = 5
```


Properties of Array:

The array contains the following properties:

- Each element of an array is of the 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:

1. **Code Optimization:** Less code to access the data.
2. **Ease of traversing:** retrieve the elements of an array easily by using the for loop.
3. **Ease of sorting:** To sort the elements of the array, we need a few lines of code.
4. **Random Access:** We can access any element randomly using the array.

Disadvantage:

Fixed Size: Once the size of the array is declared it cannot change later and can't be modified and also cannot exceed the limit or does not grow the size like the linked list in Data Structure.

One dimensional array:

A one-dimensional array (or single dimension array) is a type of linear array. Accessing its elements involves a single subscript which can either represent a row or column index.

The Syntax is as follows:

data type array name [size]

For example, int a[5]

Initialization:

An array can be initialized in two ways, which are as follows:

- **Compile time initialization:**

In compile time initialization, the user has to enter the details in the program itself. Compile time initialization is the same as variable initialization.

Syntax:

type name[size] = { list_of_values };

int rollnum[4]={ 2, 5, 6, 7}; //integer array initialization

float area[5]={ 23.4, 6.8, 5.5,7.3,2.4 }; //float array initialization

char name[9]={ 'T', 'u', 't', 'o', 'r', 'i', 'a', 'l', '\0' }; //character array initialization

Example:

Code:

```
#include<studio.h>
void main()
{
        int array[5]={1,2,3,4,5};
        int i;
        printf("Displaying array of elements :");
        for(i=0;i<5;i++)
        {
                printf("%d ",array[i]);
        }
}
```

Output:

```
Displaying array of elements :1 2 3 4 5
```

Runtime initialization:

Using runtime initialization, users can get a chance of accepting or entering different values during different runs of the program. It is also used for initializing large arrays or arrays with user specified values. An array can also be initialized at runtime using scanf() function.

Example:

Code:

```
#include<stdio.h>
int main ( )
{
        int a[5] = {10,20,30,40,50};
        int i;
        printf ("elements of the array are");
        for ( i=0; i<5; i++)
             printf ("%d", a[i]);
}
```

Output:

```
elements of the array are 10 20 30 40 50
```


Two Dimensional Arrays:

It is also possible for arrays to have two or more dimensions. The two-dimensional array is also called a matrix. program that stores roll numbers and marks obtained by a student side by side in a matrix.

These are used in situations where a table of values have to be stored (or) in matrix applications.

Syntax:

The syntax is given below

datatype arrayname [row size] [column size];

For example

int a[5] [5];

a[0][0] 10	a[0][1] 20	a[0][2] 30
a[1][0] 40	a[1][1] 50	a[1][2] 60
a[2][0]	a[2][1]	a[2][2]

Example:

Code:

```
#include<stdio.h>
int main ( )
{
        int a[3][3] = {10,20,30,40,50,60,70,80,90};
        int i,j;
        printf ("elements of the array are");
        for ( i=0; i<3; i++)
        {
                for (j=0;j<3; j++)
                {
                        printf("%d \t", a[i] [j]);
                }
        printf("\n");
        }
}
```

Output:

```
elements of the array are:
10 20 30
40 50 60
70 80 90
```

Initialization of 2D Array:

To Initialising a 2-Dimensional Array, we need to write like this:

```
int student[4][2] = {
                        { 1234, 56 },
                        { 1212, 33 },
                        { 1434, 80 },
                        { 1312, 78 }
} ;
```

Or even another way also written as

```
int stud[4][2] = { 1234, 56, 1212, 33, 1434, 80, 1312, 78 } ;
```

It is important that, while initializing a 2-D array it is necessary to mention the second (column) dimension, whereas the first dimension (row) is optional.

Declarations:

int arr[2][3] = { 12, 34, 23, 45, 56, 45 } ;

int arr[ ][3] = { 12, 34, 23, 45, 56, 45 } ;

Example:

```
#include<stdio.h>
int main()
{
	/* 2D array declaration*/
	int disp[2][3];
	/*Counter variables for the loop*/
	int i, j;
	for(i=0; i<2; i++)
	{
		for(j=0;j<3;j++)
		{
			printf("Enter value for disp[%d][%d]:", i, j);
			scanf("%d", &disp[i][j]);
		}
	}
	//Displaying array elements
	printf("Two Dimensional array elements:\n");
	for(i=0; i<2; i++)
	{
		for(j=0;j<3;j++)
		{
			printf("%d ", disp[i][j]);
			if(j==2)
			{
				printf("\n");
			}
		}
	}
```

```
        return 0;
}
```

Output:

```
Enter value for disp[0][0]:1
Enter value for disp[0][1]:2
Enter value for disp[0][2]:3
Enter value for disp[1][0]:4
Enter value for disp[1][1]:5
Enter value for disp[1][2]:6
Two Dimensional array elements:
1 2 3
4 5 6
```

Memory View of a 2-Dimensional Array:

The array blueprint shown in Figure 1 is only conceptually true. This is because memory doesn't contain rows and columns. In memory whether it is a one-dimensional or a two-dimensional array the array elements are stored in one continuous chain. The arrangement of array elements of a two-dimensional array in memory is shown below:

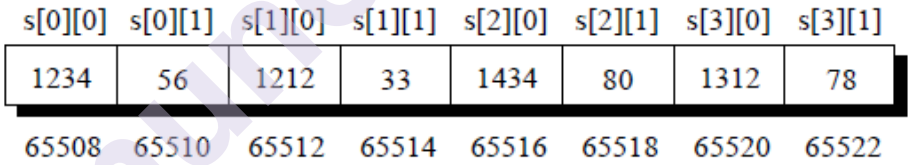


Fig. 1 Memory View of 2D Array

Pointers and 2-Dimensional Arrays:

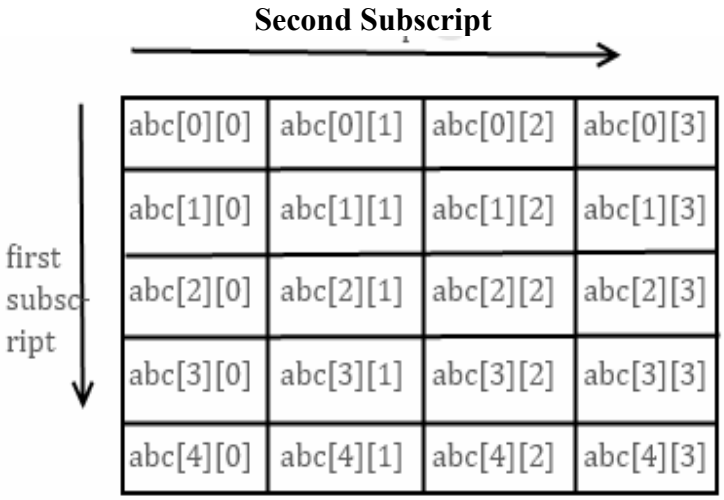


Fig. 2 2D Array Conceptual Memory representation

In a two dimensional array, As Shown in figure 2, accessing each element by using two subscripts, where the first subscript represents the row number and the second subscript represents the column number.

The elements of a 2-D array can be accessed with the help of pointer notation also. Suppose arr is a 2-D array, we can access any element arr[i][j] of the array using the pointer expression

***(*(arr + i) + j)**.

Each row of a two-dimensional array can be thought of as a one-dimensional array. This is a very important fact to access array elements of a two-dimensional array using pointers.

Thus, the declaration,

int s[5][2] ;

which is a one-dimensional array containing 2 integers. We refer to an element of a one-dimensional array using a single subscript. Similarly, if we can imagine s to be a one-dimensional array then we can refer to its zeroth element as s[0], the next element as s[1] and so on. More specifically, s[0] gives the address of the zeroth one-dimensional array, s[1] gives the address of the first one-dimensional array and so on.

Example:

Code:

```
#include<stdio.h>
int main( )
{
        int s[4][2] =
        {
                { 1234, 56 },
                { 1212, 33 },
                { 1434, 80 },
                { 1312, 78 }
            } ;
        int i ;
        for ( i = 0 ; i <= 3 ; i++ )
                printf ( "\nAddress of %d th 1-D array = %u",
i, s[i] ) ;
}
```

Output:

```
Address of 0 th 1-D array = 1065904832
Address of 1 th 1-D array = 1065904840
Address of 2 th 1-D array = 1065904848
Address of 3 th 1-D array = 1065904856
```

Pointer to an Array:

Use a pointer to an array, and then use that pointer to access the array elements.

Example:

```
Code:
#include<stdio.h>
int main( )
{
        int s[5][2] = {
                        { 1234, 56 },
                        { 1212, 33 },
                        { 1434, 80 },
                        { 1312, 78 }
                } ;
        int ( *p )[2] ;
        int i, j, *pint ;
        for ( i = 0 ; i <= 3 ; i++ )
        {
                p = &s[i] ;
                pint = p ;
                printf ( "\n" ) ;
                for ( j = 0 ; j <= 1 ; j++ )
                printf ( "%d ", *( pint + j ) ) ;
        }
}

Output:
1234 56
1212 33
1434 80
1312 78
```

4. Declaring array variables:

To start with, like other variables an array needs to be declared so that the compiler will know what kind of an array and how large an array we need. To declare the array with following statement:

```
int marks[30] ;
```

- Data Type, int specifies the type of the variable, like normal variables and the word marks specifies the name of the variable.

- The [30] however is new. The number 30 tells how many elements of the type int will be stored inside the array.
- This number is often called the 'dimension' of the array.
- The bracket ([ ]) tells the compiler that we are dealing with an array.

5. Initialization of arrays:

Different ways in which all elements of an array can be initialized to the same value:

1. Initializer List:

To initialize an array in C with the same value, the naive way is to provide an initializer list. We use this with small arrays.

int num[5] = {1, 1, 1, 1, 1};

This will initialize the num array with value 1 at all indexes. We may also ignore the size of the array:

int num[ ] = {1, 1, 1, 1, 1}

The array will be initialized to 0 in case we provide an empty initializer list or just specify 0 in the initializer list.

int num[5] = { }; // num = [0, 0, 0, 0, 0]

int num[5] = { 0 }; // num = [0, 0, 0, 0, 0]

2. Designated Initializer:

This initializer is used when we want to initialize a range with the same value. This is used only with GCC compilers.

[first . . . last] = value;

int num[5]={ [0 . . . 4] = 3 };

1. Macros:

For initializing a huge array with the same value we can use macros

2. Using For Loop:

We can also use a for loop to initialize an array with the same value.

6. Accessing array elements:

After declaration of an array, we need to see how individual elements in the array can be referred. This is done with subscript, the number in the brackets following the array name. This number specifies the element's position in the array. All the array elements are numbered, starting with 0. Therefore, marks[2] is not the second element of the array, but the third. In the c program we are using the variable i as a subscript to refer to various elements of the array. This variable can take different values and hence

can refer to the different elements in the array in turn. This ability to use variables as subscripts is what makes arrays so useful.

Inserting Data into an Array:

Here is the section of code that places data into an array:

```
for ( i = 0 ; i <= 29 ; i++ )
{
printf ( "\nEnter marks " ) ;
scanf ( "%d", &marks[i] ) ;
}
```

The above code for loop causes the process of asking for and receiving a student's marks from the user to be repeated 30 times. The first time through the loop, i has a value 0, so the scanf( ) function will cause the value typed to be stored in the array element marks[0], the first element of the array. This process will be repeated until i In scanf( ) function, we have to use the "address of " operator (&) on the element marks[i] of the array. The passing the address of this particular array element to the scanf( ) function, rather than its value; which is what scanf( ) function requires.

5. Reading Data from an Array:

After inserting data into the array, the program reads the data back out of the array and uses it to calculate the average or logic. The for loop is much the same, but now the body of the loop causes each student's marks to be added to a running total stored in a variable called sum.

When all the marks have been added up, the result is divided by 30, the number of students, to get the average.

```
for ( i = 0 ; i <= 29 ; i++ )
sum = sum + marks[i] ;
avg = sum / 30 ;
printf ( "\nAverage marks = %d", avg ) ;
```

7. Compare array types of C with list and tuple types of Python.:

Array:

1. We should always start with an array as it appeared in the programming languages earlier than the other two. Therefore, you would expect its operation to be simple and primitive. An array is a contiguous memory allocation for data storage.

2. The static array always has a predefined size and it is efficient for iterative works as the elements are stored side by side in the dedicated memory locations.

3. It is not that effective when it comes to inserting a new element in between the already present elements. Similarly, it causes inefficient memory management when you delete an element in between the elements present. Therefore, a static array is suitable only when you need to keep a series of elements side by side and you have to do iterative works through loops.

4. In such a scenario, the memory management and data processing will be faster. If you are already running out of memory space, it can give you errors and you can lose certain elements due to out of range scenarios.

5. To avoid the drawbacks of static arrays to a certain extent, the dynamic array was introduced and the data structures like vectors, array list and likewise fall under the dynamic array. These dynamic arrays can be resized and they can be placed in a scattered manner in the memory space as per availability.

List:

1. The concept of the dynamic array led to list or linked list as some like to call it. As a matter of act, after the introduction of the linked list, the dynamic array data structures started to become less popular.

2. Unlike an array, a linked list is not continuous memory allocation. It has a scattered memory allocation technique.

3. Each node of a list consists of two parts. The first part contains the data while the second part is a pointer that has the memory reference of the next node wherever it is placed in the memory.

4. This makes the memory management efficient in all scenarios and you can add or delete nodes in a list effortlessly with extremely high processing speed.

5. Unlike an array, a list can have heterogeneous data. A modified version of a list is called a double linked list where each node has three parts – the data, the reference of the previous node and the reference of the next node.

6. This makes it easy to access any data with a higher speed and perform different iterations swiftly.

Tuple:

1. Tuple is often compared with List as it looks very much like a list.

2. A tuple is actually an object that can contain heterogeneous data. Out of all data structures, a tuple is considered to be the fastest and they consume the least amount of memory.

3. While array and list are mutable which means you can change their data value and modify their structures, a tuple is immutable. Like a

static array, a tuple is fixed in size and that is why tuples are replacing arrays completely as they are more efficient in all parameters.

4. The syntaxes of each one of these data structures are different. If you have a dataset which will be assigned only once and its value should not change again, you need a tuple.

List	Array	Tuple
List is mutable	Array is mutable	Tuple is immutable
A list is ordered collection of items	An array is ordered collection of items	A tuple is an ordered collection of items
Item in the list can be changed or replaced	Item in the array can be changed or replaced	Item in the tuple cannot be changed or replaced
List can store more than one data type	Array can store only similar data types	Tuple can store more than one data type

6.8 SUMMARY

1. An array is similar to an ordinary variable except that it can store multiple elements of similar type.
2. Compiler doesn't perform bounds checking on an array.
3. The array variable acts as a pointer to the zero element of the array. In a 1-D array, the zero element is a single value, whereas, in a 2-D array this element is a 1-D array.
4. On incrementing a pointer it points to the next location of its type.
5. Array elements are stored in contiguous memory locations and so they can be accessed using pointers.
6. Only limited arithmetic can be done on pointers.

6.9 UNIT END QUESTIONS

1. What is an array? Explain the types of arrays?
2. Write a C program to implement the concept of 2D array?
3. Explain the difference between array, list and tuple?
4. What is an array of pointers?

6.10 REFERENCE FOR FURTHER READING

1. Programming in ANSI C (Third Edition) : E Balagurusamy, TMH
2. Yashavant P. Kanetkar. " Let Us C", BPB Publications

*****

7

DATA INPUT AND OUTPUT FUNCTIONS

Unit Structure

7.1 OBJECTIVE

a. To understand the concept of Data input and output function.

b. To learn the different I/O functions.

c. To learn how to read data from the keyboard as an input.

d. To learn how to write data to an output screen like a monitor.

7.2 INTRODUCTION

The screen and keyboard together are called input devices or consoles. Console I/O functions can be further classified into two categories

1. formatted and

2. Unformatted console I/O functions

The basic difference between them is that the formatted functions allow the input read from the keyboard or the output displayed on the screen to be formatted as per our user requirements.

For example, if values of average marks and percentage marks are to be displayed on the screen, then the details like where this output would appear on the screen, how many spaces would be present between the two

values, the number of places after the decimal points, etc. can be controlled using formatted functions.

The functions available under each of these two categories are shown in Figure 1.

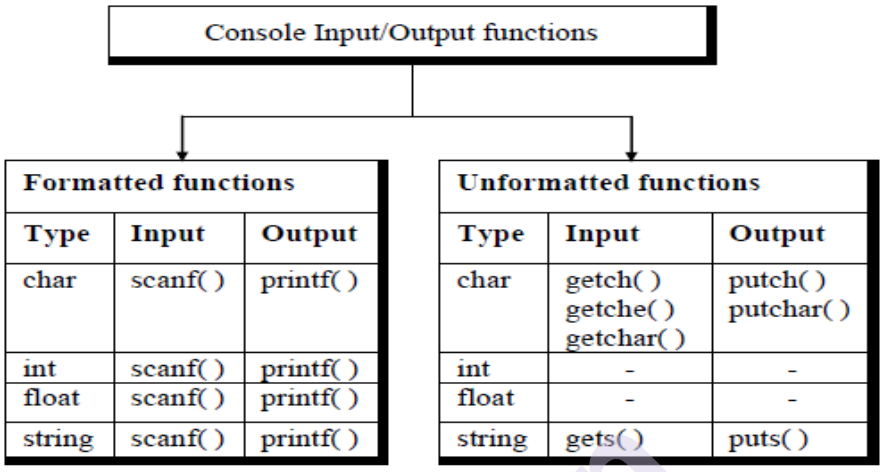


Fig. 1

1. Data Input and Output functions:

The functions printf( ), and scanf( ) fall under the category of formatted console I/O functions. These functions allow us to supply the input in a fixed format and let us obtain the output in the specified form.

Its general form looks like this…

printf ("format string", list of variables) ;

The format string can contain:

1. Characters that are simply printed as they are
2. Conversion specifications that begin with a % sign
3. Escape sequences that begin with a \ sign

Example:

Code:

```
#include <stdio.h>
int main()
{
        int age = 34;
        float marks = 69.2 ;
        printf ( "Average = %d\nPercentage = %f", age, marks ) ;
}
```

Output:

```
Average = 34
Percentage = 69.199997
```

In the above example, printf( ) function interprets the contents of the format string. For this it examines the format string from left to right. So long as it doesn't come across either a % or a \ it continues to dump the characters that it encounters onto the screen.

In this above example

1. Average = is dumped on the screen. The moment it comes across a conversion specification in the format string it picks up the first variable in the list of variables and prints its value in the specified format.
2. The moment %d is met the variable avg is picked up and its value is printed. Similarly, when an escape sequence is met it takes the appropriate action.
3. The moment \n is met it places the cursor at the beginning of the next line. This process continues till the end of the format string is not reached.

Format Specifications:

The %d and %f used in the printf( ) are called format specifiers. They tell printf( ) to print the value of avg as a decimal integer and the value of per as a float. Following is the list of format specifiers that can be used with the printf( ) function.

C also provides the following optional specifiers in the format specifications.

Data type		Format specifier
Integer	short signed	%d or %I
	short unsigned	%u
	long singed	%ld
	long unsigned	%lu
	unsigned hexadecimal	%x
	unsigned octal	%o
Real	float	%f
	double	%lf
Character	signed character	%c
	unsigned character	%c
String		%s

Fig. 2

Specifier	Description
dd	Digits specifying field width
.	Decimal point separating field width from precision (precision stands for the number of places after the decimal point)
dd	Digits specifying precision
-	Minus sign for left justifying the output in the specified field width

Fig. 3

Example:

Code:

```
#include <stdio.h>

int main()
{
        int size = 63 ;
        printf ( "\nsize is %d cm", size ) ;
        printf ( "\nsize is %2d cm", size ) ;
        printf ( "\nsize is %4d cm", size ) ;
        printf ( "\nsize is %6d cm", size ) ;
        printf ( "\nsize is %-6d cm", size ) ;
}
```

Output:

```
size is 63 cm
size is 63 cm
size is   63 cm
size is  63 cm
size is 63     cm
```

The format specifiers could be used even while displaying a string of characters. The following program would clearly understand this point:

Code:

```
int main()
{
        char firstname1[ ] = "Sandy" ;
        char surname1[ ] = "MalyaSingh" ;
        char firstname2[ ] = "Ajay" ;
        char surname2[ ] = "laxmi" ;
        printf ( "\n%20s%20s", firstname1, surname1 ) ;
```

```
        printf ( "\n%20s%20s", firstname2, surname2 ) ;
}

Output:
                Sandy          MalyaSingh
                 Ajay                laxmi
```

Printing Single Characters:

A single character can be displayed in the desired location using the format.

%wc

The character will be displayed right-justified in the field of w columns. We can make the display left-justified by placing a minus sign before the integer w. The default value of w is 1.

Escape Sequences:

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 a 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

Example1: \a escape sequence

Code:

```
// C program to illustrate
// \a escape sequence
#include <stdio.h>
int main(void)
{
        printf("My mobile number " "is 9\a8\a7\a3\a9\a2\a3\a4\a0\a6\a");
        return (0);
}
```

Output:

My mobile number is 9873923406.

Example2: \n New Line

Code:

```
// C program to illustrate
// \n escape sequence
#include <stdio.h>
int main(void)
{
        // Here we are using \n, which
        // is a new line character.
        printf("Hello\n");
        printf("C Programming");
        return (0);
}
```

Output:

Hello
C Programming

Example3: \t Horizontal Tab

Code:

```
#include <stdio.h>
int main ()
{
        printf("\n horizontal tab escape sequence tutorial");
        printf(" \n 34543 \t 345435 ");
        printf(" \n 123 \t 678 ");
        return 0;
}
```

Output:

```
horizontal tab escape sequence tutorial
 34543    345435
 123      678
```

Example4: \b back space

Code:

```
#include <stdio.h>
int main ()
{
   printf("\n backspace escape sequence tutorial");
   printf(" \n watch\b carefully the execution");
   return 0;
}
```

Output:

```
 backspace escape sequence tutorial
 watc carefully the execution
```

Example5: \r Carriage Return

Code:

```
#include <stdio.h>
int main ()
{
   printf("\n demo code below");
   printf("\r remove");
   printf("\n  done with example");
   return 0;
}
```

Output:
removeode below
done with example

7.4 CHARACTER I/O FORMAT

7.4.1 getch():

getch() function is a function in the C programming language which waits for any character input from the keyboard. Please find below the description and syntax for the above file handling function.

File operation	Declaration & Description
getch()	Declaration: int getch(void); This function waits for any character input from the keyboard. But, it won't echo the input character on to the output screen

This is a simple Hello World! C program. After displaying Hello World! In the output screen, this program waits for any character input from the keyboard. After any single character is typed or pressed, this program returns 0. But, please note that the getch() function will just wait for any keyboard input. It won't display the given input character in the output screen.

Example:

Code:

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

Output:
Hello World!

7.4.2 getche():

getche() function is a function in C programming language which waits for any character input from the keyboard and it will also echo the input character onto the output screen. Please find below the description and syntax for the above file handling function.

File operation	Declaration & Description
getche()	Declaration: int getche(void); This function waits for any character input from the keyboard. And, it will also echo the input character onto the output screen.

This is a simple Hello World! program. After displaying Hello World! In the output screen, this program waits for any character input from the keyboard. After any single character is typed or pressed, this program returns 0. But, please note that, the getche() function will wait for any keyboard input and it will display the given input character on the output screen immediately after keyboard input is entered.

Example:

Code:

```
#include <stdio.h>
int main()
{
        char flag;
        /* Our first simple C basic program */
        printf("Hello World! ");
        printf("Do you want to continue Y or N");
        flag = getche(); // It waits for keyboard input.
         if (flag == 'Y')
         {
                printf("You have entered Yes");
        }
        else
        {
                printf("You have entered No");
        }
        return 0;
}
```

Output:

```
Hello World!
Do you want to continue Y or N
Y
You have entered Yes
```

7.4.3 getchar():

A getchar() function is a non-standard function whose meaning is already defined in the <stdin.h> header file to accept a single input from the user. In other words, it is the C library function that gets a single character or unsigned char from the stdin. However, the getchar() function is similar to the getc() function, but there is a small difference between the getchar() and getc() function of the C programming language

A getchar() reads a single character from standard input, while a getc() reads a single character from any input stream.

Example:

Code:

```
#include <conio.h>
#include<stdio.h>
void main()
{
        char c;
        printf ("\n Enter a character \n");
        c = getchar(); // get a single character
        printf(" You have passed ");
        putchar(c); // print a single character using putchar
        getch();
}
```

Output:

```
Enter a character
A
 You have passed A
```

7.4.4 getc():

The C library function int getc(FILE *stream) gets the next character (an unsigned char) from the specified stream and advances the position indicator for the stream.

Code:

```
#include<stdio.h>
int main () {
   char c;
   printf("Enter character: ");
   c = getc(stdin);
   printf("Character entered: ");
   putc(c, stdout);
   return(0);
```

```
}

Output:
Enter character: a
Character entered: a
```

Example:

7.4.5 gets():

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.

Example:

Code:

```
#include<stdio.h>
void main ()
{
        char s[30];
        printf("Enter the string? ");
        gets(s);
        printf("You entered %s",s);
}
```

Output:

```
Enter the string?
C is the best
You entered C is the best
```

7.4.6 putchar():

The putchar(int char) method in C is used to write a character, of unsigned char type, to stdout. This character is passed as the parameter to this method.

Syntax: int putchar(int char)

Example:

Code:

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

```
{
        // Get the character to be written
        char ch = 'G';
        // Write the Character to stdout
        putchar(ch);
        return (0);
}
```

Output:

G

7.4.7 putc():

The C library function int putc(int char, FILE *stream) writes a character (an unsigned char) specified by the argument char to the specified stream and advances the position indicator for the stream.

Syntax: int putc(int char, FILE *stream)

Example:

Code:

```
#include <stdio.h>
int main ()
{
        FILE *fp;
        int ch;
        fp = fopen("file.txt", "w");
        for( ch = 33 ; ch <= 100; ch++ )
        {
                putc(ch, fp);
        }
        fclose(fp);
        return(0);
}
```

Output:

This is C Programming

7.4.8 puts():

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.

Example:

Code:

```
#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: Rahul Shyam
Your name is: Rahul Shyam
```


7.5 SUMMARY

1. There is no keyword available in C for doing input/output.
2. All I/O in C is done using standard library functions.
3. There are several functions available for performing console input/output.
4. The formatted console I/O functions can force the user to receive the input in a fixed format and display the output in a fixed format.
5. There are several format specifiers and escape sequences available to format input and output.
6. Unformatted console I/O functions work faster since they do not have the overheads of formatting the input or output.

7.6 UNIT END QUESTIONS

1. List the different Escape Sequences in C?
2. Explain the different format specifiers in C.
3. Write a short note on:

a. getchar() & putchar()

b. puts() & gets()

7.7 REFERENCE FOR FURTHER READING

1. Programming in ANSI C (Third Edition) : E Balagurusamy, TMH
2. Yashavant P. Kanetkar. " Let Us C", BPB Publications

*****

8

MANIPULATING STRINGS

Unit Structure

8.1 OBJECTIVE

a. To understand the operation on string using C.
b. To learn declaration and initialization of string functions.
c. To understand the different Standard Library String Functions
d. To understand the differences between C and python strings.

8.2 INTRODUCTION

The way a group of integers can be stored in an integer array, similarly a group of characters can be stored in a character array. Character arrays are many times also called strings. Many languages internally treat strings as character arrays, but somehow conceal this fact from the programmer. Character arrays or strings are used by programming languages to manipulate text such as words and sentences.

A string constant is a one-dimensional array of characters terminated by a null ('\0'). For example,

```
char name[ ] = { 'H', 'A', 'E', 'S', 'L', 'E', 'R', '\0' } ;
```

Each character in the array occupies one byte of memory and the last character is always '\0'. \0 looks like two characters, but it is actually only one character, with the \ indicating that what follows it is something special. '\0' is called a null character. Note that '\0' and '0' are not the same. The ASCII value of '\0' is 0, whereas the ASCII value of '0' is 48. Figure 1 shows the way a character array is stored in memory. Note that the elements of the character array are stored in contiguous memory locations.

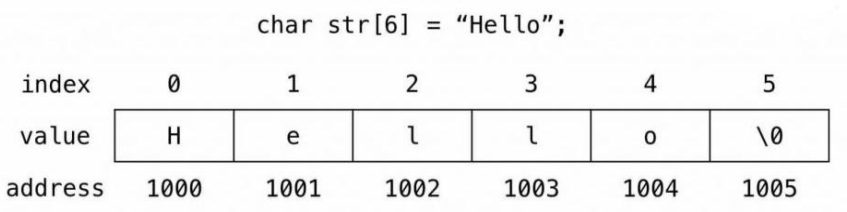


Fig. 1 Character array memory view

The terminating null ('\0') is important, because it is the only way the functions that work with a string can know where the string ends. In fact, a string not terminated by a '\0' is not really a string, but merely a collection of characters

Example:

Code:

```
#include <stdio.h>
int main()
{
        char name[ ] = "Rudra" ;
        int i = 0 ;
        while ( name[i] != '\0' )
        {
                printf ( "%c", name[i] ) ;
                i++ ;
        }
}
```

Output:

Rudra

8.3 DECLARING AND INITIALIZING STRING VARIABLES

A C String is a simple array with char as a data type. 'C' language does not directly support string as a data type. Hence, to display a String in C, you need to make use of a character array.

The general syntax for declaring a variable as a String in C is as follows:

char string_variable_name [array_size];

The classic Declaration of strings can be done as follow:

char string_name[string_length] = "string";

The size of an array must be defined while declaring a C String variable because it is used to calculate how many characters are going to be stored inside the string variable in C.

valid examples of string declaration are as follows:

```
char first_name[30];    //declaration of a string variable
char last_name[30];
```

The above example represents string variables with an array size of 30. This means that the given C string array is capable of holding 30 characters at most. The indexing of the array begins from 0 hence it will store characters from a 0-29 position. The C compiler automatically adds a NULL character '\0' to the character array created.

String initialization in C language. Following example show the initialization of Strings in C,

```
char first_name[15] = "ABCDEFG";

char first_name[15] = {'A','B','C','D','E','F','G','\0'};

// NULL character '\0' is required at end in this declaration

char string1 [6] = "hello";/* string size = 'h'+'e'+'l'+'l'+'o'+"NULL" = 6 */

char string2 [ ] = "world";  /* string size = 'w'+'o'+'r'+'l'+'d'+"NULL" = 6
*/

char string3[6] = {'h', 'e', 'l', 'l', 'o', '\0'} ; /*Declaration as set of characters
,Size 6*/
```

in string3, the NULL character must be added explicitly, and the characters are enclosed in single quotation marks.

'C' also allows us to initialize a string variable without defining the size of the character array. It can be done in the following way,

```
char first_name[ ] = "AMITKUMAR";
```

Reading a String:

The scanf() function is used to read a string. The scanf() function reads the sequence of characters until it encounters whitespace (space, newline, tab, etc.).

Example:

Code:

```
#include <stdio.h>
int main()
{
        char name[20];
        printf("Enter name: ");
        scanf("%s", name);
        printf("Your name is %s.", name);
```

```
    return 0;
}
```

Output

```
Enter name: Amey
Your name is Amey.
```

Read a line of text:

You can use the fgets() function to read a line of string. And, you can use puts() to display the string.

Example:

Code:

```
#include <stdio.h>
int main()
{
    char name[30];
    printf("Enter name: ");
    fgets(name, sizeof(name), stdin);  // read string
    printf("Name: ");
    puts(name);    // display string
    return 0;
}
```

Output:

```
Enter name: Rahul Kumar
Name: Rahul Kumar
```


Passing Strings to Functions:

Strings can be passed to a function in a similar way as arrays. Learn more about passing arrays to a function.

Example:

Code:

```
#include<stdio.h>
void displayString(char str[]);
int main()
{
    char str[50];
    printf("Enter string: ");
    fgets(str, sizeof(str), stdin);
```

```
        displayString(str);     // Passing string to a function.
        return 0;
}
void displayString(char str[])
{
        printf("String Output: ");
        puts(str);
}
```

Output:

Enter string: Rudra
String Output: Rudra

8.4 CHARACTER AND STRING

Strings are defined as an array of characters. The difference between a character array and a string is that the string is terminated with a special character '\0'.

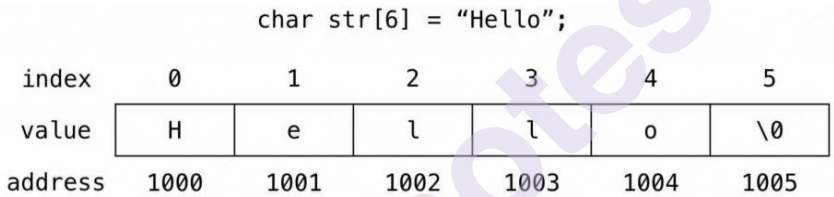


Fig. 2 Array character

Example:

Code:

```
#include<stdio.h>
int main()
{
        // declare and initialize string
        char str[] = "C Program";
        // print string
        printf("%s",str);
        return 0;
}
```

Output:

C Program

8.5 HANDLING FUNCTIONS

Every C compiler a large set of useful string handling library functions are provided. Table 1 lists the more commonly used functions along with their purpose.

Function	Use
strlen	Finds length of a string
strlwr	Converts a string to lowercase
strupr	Converts a string to uppercase
strcat	Appends one string at the end of another
strncat	Appends first n characters of a string at the end of another
strcpy	Copies a string into another
strncpy	Copies first n characters of one string into another
strcmp	Compares two strings
strncmp	Compares first n characters of two strings
strcmpi	Compares two strings without regard to case ("i" denotes that this function ignores case)
stricmp	Compares two strings without regard to case (identical to strcmpi)
strnicmp	Compares first n characters of two strings without regard to case
strdup	Duplicates a string
strchr	Finds first occurrence of a given character in a string
strrchr	Finds last occurrence of a given character in a string
strstr	Finds first occurrence of a given string in another string
strset	Sets all characters of string to a given character
strnset	Sets first n characters of a string to a given character
strrev	Reverses string

Table 1 : List String functions

strlen( ):

This function counts the number of characters present in a string. Its usage is illustrated in the following program.

strlen(): String Length

strlen function returns length of the string without counting null characters.

Example:

Code:

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

int main()
{
        char str[]="www.mu.ac.in";
        int length;
        //string length
        length=strlen(str);
        printf("String Length: %d\n",length);
        return 0;
}
```

Output:

String Length: 12

strupr() & strlwr() - String Upper & Lower:

strupr converts string into uppercase letter.

strlwr converts string into lowercase letter.

Example:

Code:

```
#include <stdio.h>
#include <string.h>
 int main()
{
        char s[100];
        int i;
        printf("\nEnter a string : ");
        gets(s);

        for (i = 0; s[i]!='\0'; i++)
        {
                if(s[i] >= 'a' && s[i] <= 'z')
```

```
            {
                s[i] = s[i] - 32;
            }
        }
        printf("\nString in Upper Case = %s", s);

        for (i = 0; s[i]!='\0'; i++)
        {
            if(s[i] >= 'A' && s[i] <= 'Z')
            {
                s[i] = s[i] + 32;
            }
        }
        printf("\nString in Lower Case = %s", s);
        return 0;
}
```

Output:

```
Enter a string : IDOL Mumbai University

String in UpperCase = IDOL MUMBAI UNIVERSITY
String in LowerCase = idol mumbai university
```

strrev() - String Reverse:

strrev- is a function used to reverse the string.

Example:

Code:

```
#include <stdio.h>
#include <string.h>
int main()
{
    char str[40]; // declare the size of character string
    printf (" \n Enter a string to be reversed: ");
    scanf ("%s", str);

    // use strrev() function to reverse a string
    printf (" \n After the reverse of a string: %s ", strrev(str));
    return 0;
}
```

Output:
Enter a string to be reversed: IDOL
After the reverse of a string: LODI

strcpy() - String Copy:

strcpy-copies one string to another string, in this function there will be two parameters, the second parameter's values will be copied into the first parameter's variable.

Example:

Code:

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

int main()
{
        char str1[30];
        char str2[30];
        printf("Enter string 1: ");
        gets(str1);
        //copy str1 into str2
        strcpy(str2,str1);
        printf("str1: %s \nstr2: %s \n",str1,str2);
        return 0;
}
```

Output:
Enter string 1: Rudra
str1: Rudra
str2: Rudra

strcmp() - String Compare:

strcmp function compares two strings and returns 0, less than 0 and greater than 0 based on strings, if strings are the same function will return 0, other function will return difference of first dissimilar character, difference may be positive or negative.

strcmpi() - String Comparing Ignoring case:

strcmpi function compares two strings ignoring case sensitivity and returns 0, less than 0 and greater than 0 based on strings, if strings are the

same function will return 0, other function will return difference of first dissimilar character, difference may be positive or negative.

Example:

Code:

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

int main()
{
        char str1[30];
        char str2[30];

        printf("Enter string1: "); gets(str1);
        printf("Enter string2: "); gets(str2);

        //using strcmp
        printf("Using strcmp:\n");
        if(strcmp(str1,str2)==0)
                printf("strings are same.\n");
         else
                printf("strings are not same.\n");

        //using strcmp
        printf("Using strcmpi:\n");
        if(strcmpi(str1,str2)==0)
                printf("strings are same.\n");
        else
                printf("strings are not same.\n");
        return 0;
}
```

Output:

```
Enter string1: Hello World
Enter string2: hello world
Using strcmp:
strings are not same.
Using strcmpi:
strings are same.
```

strcat() - String Concatenate:

Concatenation is the process of appending one string to the end of another string.

Example:

Code:

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

int main()
{
   char title[5],fName[30],lName[30];
   char name[100]={0}; //assign null

   printf("Enter title (Mr./Mrs.): ");
   gets(title);
   printf("Enter first name: ");
   gets(fName);
   printf("Enter last name: ");
   gets(lName);
    //create complete name using string concatenate
   strcat(name,title);
   strcat(name," ");

   strcat(name,fName);
   strcat(name," ");

   strcat(name,lName);
   strcat(name," ");
    printf("Hi.... %s\n",name);

   return 0;
}
```

Output:

```
Enter title (Mr./Mrs.): Mr.
Enter first name: Rahul
Enter last name: Sathe
Hi.... Mr. Rahul Sathe
```

8.6 COMPARE WITH PYTHON STRINGS

1. The main difference between C and Python is that C is a structure oriented programming language while Python is an OOP language.
2. In general, C is used for developing hardware operable applications, and python is used as a general purpose programming language.
3. C language is run under a compiler, python on the other hand is run under an interpreter. Python has fully formed built-in and predefined library functions, but C has only few built-in functions available.
4. Python is easy to learn and implement, whereas C needs a detailed understanding to program and implement.
5. String in python work differently from those in other scripting languages, like C. Python strings operate in the same basic fashion as C character arrays-a string is a sequence of single characters.
6. The term sequence is important here because python gives special capabilities to objects that are based on sequences.
7. Other sequence objects include lists, which are sequences of objects and tuples, which are immutable sequences of objects.
8. Strings are also immutable, They cannot be changed in place.
9. Python strings are also your first introduction to the complex objects that python supports, and they form the basis of many of the other object types that python supports.

Operation	C String	Python String
Declaration of strings	char str_name[size];	str_name = value
Initializing a String	char str[] = "hello";	a = "Hello"
read a string from user	scanf("%s",str);	val = input("Enter String: ")
Length of string	strlen	len()
String Comparison	strcmp()	print("hello" == "hello")
String Copy	strcpy()	str2 = str1
String Reverse	strrev()	txt = "Hello World"[::-1]
String Upper & Lower	strupr() & strlwr()	s.upper() & s.lower()

Concatenation of Two or More Strings in python:

Joining two or more strings into a single one is called concatenation. The + operator does this in Python. Simply writing two string literals together also concatenates them. The * operator can be used to repeat the string for a given number of times.

Example:

Code:

```
# Python String Operations
str1 = 'Hello'
str2 ='World!'

# using +
print('str1 + str2 = ', str1 + str2)

# using *
print('str1 * 3 =', str1 * 3)
```

Output:

```
str1 + str2 =  HelloWorld!
str1 * 3 = HelloHelloHello
```

8.7 SUMMARY

1. A string is nothing but an array of characters terminated by '\0'.
2. Being an array, all the characters of a string are stored in contiguous memory locations.
3. Though scanf( ) can be used to receive multi-word strings, gets( ) can do the same job in a cleaner way.
4. Both printf( ) and puts( ) can handle multi-word strings.
5. Strings can be operated upon using several standard library functions like strlen( ), strcpy( ), strcat( ) and strcmp( ) which can manipulate strings. More importantly we imitated some of these functions to learn how these standard library functions are written.

8.8 UNIT END QUESTIONS

1. Compare the C string and python string.
2. Explain different types of string function.
3. Write a short note on Character and string.
4. Write a c program to convert string lower case to upper and vice versa.

8.9 REFERENCE FOR FURTHER READING

1. Programming in ANSI C (Third Edition) : E Balagurusamy, TMH
2. Yashavant P. Kanetkar. " Let Us C", BPB Publications

*****

9

FUNCTION & RECURSION

Unit Structure

9.1 OBJECTIVE

1. To understand the concept of function in the C programming language.
2. To understand how to declare and define the function.
3. To understand the difference between global and local variables.
4. To understand the difference between call by value and call by reference.
5. To solve problems using recursion.
6. To know what is a recursive function and the benefits of using recursive functions

9.2 INTRODUCTION

A function is a having its own block of statements that perform a coherent task of some kind. Every C program can be thought of as a collection of these functions. Using a function is something like hiring a person to do a specific job for you. Sometimes the interaction with this person is very simple & sometimes it's complex.

A function is a group of statements that together perform a task. Every C program has at least one function, which is main(), and all the most small programs can define additional functions.

We can divide up your code into separate functions. Divide up your code among different functions is up to depend on the user, but logically the division is such that each function performs a specific task.

A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function.

The C standard library provides a large number of built-in functions that your program can call. For example, strcat() to concatenate two strings, memcpy() to copy one memory location to another location, and many more functions.

A function can also be referred to as a method or a subroutine or a procedure, etc.

Let us now look at a function that calls or activates the function and the function itself.

Example:

Code:

```
include <stdio.h>
int main( )
{
        message( ) ;
        printf ( "\nMain function" ) ;
}
message( )
{
        printf ( "\ncalling function" ) ;
}
```

Output:

```
calling function
Main function
```

In the above program, main( ) itself is a function and through it we are calling the function message( ). It means that the control passes to the function message( ). The activity of main( ) is temporarily suspended; it falls asleep while the message( ) function wakes up and goes to work. When the message( ) function runs out of statements to execute, the control returns to main( ), which comes to life again and begins executing its code at the exact point where it left off. Thus, main( ) becomes the 'calling' function, whereas message( ) becomes the 'called' function.

A number of conclusions can be drawn:

1. Any C program contains at least one function.
2. If a program contains only one function, it must be main( ).
3. If a C program contains more than one function, then one (and only one) of these functions must be main( ), because program execution always begins with main( )
4. There is no limit on the number of functions that might be present in a C program.
5. Each function in a program is called in the sequence specified by the function calls in main( ).
6. After each function has done its thing, control returns to main( ).When main( ) runs out of function calls, the program ends.

9.3 FUNCTION

9.3.1 Function declaration:

A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function.

Or,

A function declaration tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts:

return_type function_name(parameter list);

For the above defined function max(), the function declaration is as follows:

int max(int num1, int num2);

Parameter names are not important in function declaration only their type is required, so the following is also a valid declaration:

int max(int, int);

Function declaration is required when you define a function in one source file and call that function in another file. In such a type, declare the function at the top of the file calling the function.

2. Function definition:

The general form of a function definition in C programming language is as follows:

return_type function_name(parameter list)

{

body of the function

}

A function definition in C programming consists of a *function header* and a *function body*. Here are all the parts of a function:

- **Return Type:** A function may return a value. The return_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword void.
- **Function Name:** This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters:** A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as an actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- **Function Body:** The function body contains a collection of statements that define what the function does.

9.3.3 Global and local variables:

Local variable:

- It is generally declared inside a function.
- If it isn't initialized, a garbage value is stored inside it.
- It is created when the function begins its execution.
- It is lost when the function is terminated.
- Data sharing is not possible since the local variable/data can be accessed by a single function.
- Parameters need to be passed to local variables so that they can access the value in the function.
- It is stored on a stack, unless mentioned otherwise.
- They can be accessed using a statement inside the function where they are declared.
- When the changes are made to local variables in a function, the changes are not reflected in the other function.

- Local variables can be accessed with the help of statements, inside a function in which they are declared.

Example:

Code:

```
#include <stdio.h>
int main ()
{
        /* local variable declaration */
         int a, b;
        int c;
        /* actual initialization */
        a = 10;
        b = 20;
        c = a + b;
        printf ("value of a = %d, b = %d and c = %d\n", a, b, c);
        return 0;
}
```

Output:

value of a = 10, b = 20 and c = 30

Global variable:

- It is declared outside the function.
- If it isn't initialized, the value of zero is stored in it as default.
- It is created before the global execution of the program.
- It is lost when the program terminates.
- Data sharing is possible since multiple functions can access the global variable.
- They are visible throughout the program, hence passing parameters is not required.
- It can be accessed using any statement within the program.
- It is stored on a specific location inside the program, which is decided by the compiler.
- When changes are made to the global variable in one function, these changes are reflected in the other parts of the program as well.

Example:

Code:

```
#include<stdio.h>
/* global variable declaration */
int g;
int main ()
{
        /* local variable declaration */
        int a, b;
        /* actual initialization */
        a = 10;
        b = 20;
        g = a + b;
        printf ("value of a = %d, b = %d and g = %d\n", a, b, g);
        return 0;
}
```

Output:

```
value of a = 10, b = 20 and g = 30
```

Example:

Code:

```
#include<stdio.h>

// Global variables
int A;
int B;

int Add()
{
        return A + B;
}
int main()
{
        int answer; // Local variable
        A = 5;
        B = 7;
        answer = Add();
        printf("%d\n",answer);
        return 0;
```

```
}
```
Output:
12

In the above program two global variables are declared, A and B. These variables can be used in main() and Add() functions. The local variable answer can only be used in the main() function.

9.3.4 Return statement:

- The return statement terminates or ends the execution of a function and returns control to the calling function.
- Execution resumes in the calling function at the point immediately following the call.
- A return statement can also return a value to the calling function.
- A return statement causes your function to exit and hand back a value to its caller.
- The point of functions, in general, is to take in inputs and return something.
- The return statement is used when a function is ready to return a value to its caller.

Syntax: return (expression);

Example:

Code:
```
#include <stdio.h>
void print() // void method
{
        printf("Return Statement");
}
int main()  // Driver method
{
        print(); // Calling print
        return 0;
}
```
Output:
Return Statement

Working of return statement:

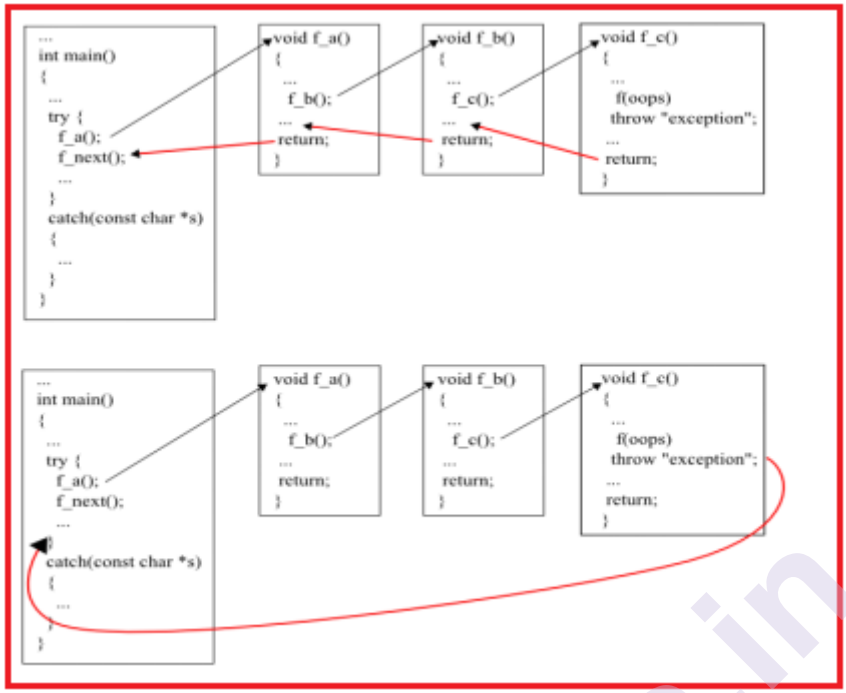


Fig. 1 Return Statement

- **The return statement serves two purposes:**

1. On executing the return statement it immediately transfers the control back to the calling program.

2. It returns the value present in the parentheses after return, to th3e calling program. In the above program the value of sum of three numbers is being returned.

- There is no restriction on the number of return statements that may be present in a function. Also, the return statement need not always be present at the end of the called function

- Whenever the control returns from a function some value is definitely returned. If a meaningful value is returned then it should be accepted in the calling program by equating the called function to some variable.

9.3.5 Calling a function by passing values:

- Functions can be invoked in two ways: Call by Value or Call by Reference. These two ways are generally differentiated by the type of values passed to them as parameters.

- The parameters passed to function are called actual parameters whereas the parameters received by function are called formal parameters.

- **Call By Value:** In this parameter passing method, values of actual parameters are copied to function's formal parameters and the two types of parameters are stored in different memory locations. So any changes made inside functions are not reflected in actual parameters of the caller.
- **Call by Reference:** Both the actual and formal parameters refer to the same locations, so any changes made inside the function are actually reflected in actual parameters of the caller.

Call By Value	Call By Reference
While calling a function, we pass values of variables to it. Such functions are known as "Call By Values".	While calling a function, instead of passing the values of variables, we pass address of variables(location of variables) to the function known as "Call By References.
In this method, the value of each variable in the calling function is copied into corresponding dummy variables of the called function.	In this method, the address of actual variables in the calling function are copied into the dummy variables of the called function.
With this method, the changes made to the dummy variables in the called function have no effect on the values of actual variables in the calling function.	With this method, using addresses we would have access to the actual variables and hence we would be able to manipulate them.

Call by Value:

Example:

Code:

```
#include<studio.h>

// Function Prototype
void swapx(int x, int y);

// Main function
int main()
{
        int a = 10, b = 20;
        // Pass by Values
        swapx(a, b);
        printf("a=%d b=%d\n", a, b);
        return 0;
}
```

```
// Swap functions that swaps
// two values
void swapx(int x, int y)
{
        int t;
        t = x;
        x = y;
        y = t;
        printf("x=%d y=%d\n", x, y);
}
```

Output:

```
x=20 y=10
a=10 b=20
```

Call By References:

Example:

Code:

```
#include<studio.h>
// Function Prototype
void swapx(int*, int*);

// Main function
int main()
{
        int a = 10, b = 20;
        // Pass reference
        swapx(&a, &b);
        printf("a=%d b=%d\n", a, b);
        return 0;
}

// Function to swap two variables
// by references
void swapx(int* x, int* y)
{
        int t;
        t = *x;
        *x = *y;
        *y = t;
        printf("x=%d y=%d\n", *x, *y);
```

```
}
```

Output:
x=20 y=10
a=20 b=10

9.4 RECURSION

9.4.1 Definition of Recursion:

- In C language, it is possible for the functions to call themselves.
- A function is called 'recursive' if a statement within the body of a function calls the same function. Sometimes called 'circular definition', recursion is thus the process of defining something in terms of itself.
- Example of recursion.

Suppose we want to calculate the factorial value of an integer. As we know, the and that number. For example, 4 factorial is 4 * 3 * 2 * 1. This can also be expressed as 4! = 4 * 3! where '!' stands for factorial. Thus the factorial of a number can be expressed in the form of itself.

9.4.2 Recursive functions:

- The recursive functions are a class of functions on the natural numbers studied in computability theory, a branch of contemporary mathematical logic which was originally known as recursive function theory.
- Such functions take their name from the process of recursion by which the value of a function is defined by the application of the same function applied to smaller arguments.

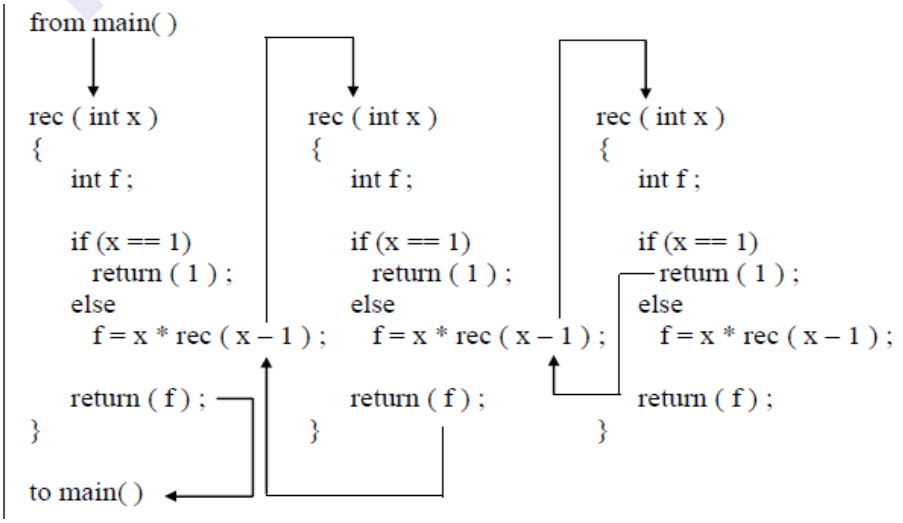


Fig. 2 Recursive function

The following example calculates the factorial of a given number using a recursive function:

Example:

Code:

```
#include <stdio.h>
unsigned long long int factorial(unsigned int i)
{

    if(i <= 1)
    {
        return 1;
    }
    return i * factorial(i - 1);
}
int  main()
{
    int i = 6;
    printf("Factorial of %d is %d\n", i, factorial(i));
    return 0;
}
```

Output:

```
Factorial of 6 is 720
```

The following example generates the Fibonacci series for a given number using a recursive function:

Example:

Code:

```
#include <stdio.h>

int fibonacci(int i)
{

    if(i == 0)
    {
        return 0;
    }
    if(i == 1)
    {
        return 1;
```

```
        }
        return fibonacci(i-1) + fibonacci(i-2);
}

int  main()
{
        int i;
        for (i = 0; i < 10; i++)
        {
                printf("%d\t\n", fibonacci(i));
        }
        return 0;
}
```

Code:

```
0
1
1
2
3
5
8
13
21
34
```


9.5 SUMMARY

1. To avoid repetition of code and bulky programs functionally related statements are isolated into a function.
2. Function declaration specifies what is the return type of the function and the types of parameters it accepts.
3. Function definition defines the body of the function.
4. Variables declared in a function are not available to other functions in a program. So, there won't be any clash even if we give the same name to the variables declared in different functions.

9.6 UNIT END QUESTIONS

1. What is a function? Explain with an example?
2. What is the difference between Global & Local variables?

3. What is the difference between Call by value and Call by reference?
4. What is a Recursive function? Explain with Example.

9.7 REFERENCE FOR FURTHER READING

1. Programming in ANSI C (Third Edition) : E Balagurusamy, TMH
2. Yashavant P. Kanetkar. " Let Us C", BPB Publications.

*****

UNIT III

10

POINTERS

Unit Structure

10.0 OBJECTIVE

In this chapter we will discuss about Pointers. Which is most important and necessary fundamentals of C. Here our objective is give knowledge of pointers. And how its store in memory, How to handle etc.

10.1 INTRODUCTION

In C, a **pointer** holds the address of an object stored in memory. The pointer then simply "points" to the object. The type of the object must correspond with the type of the pointer.

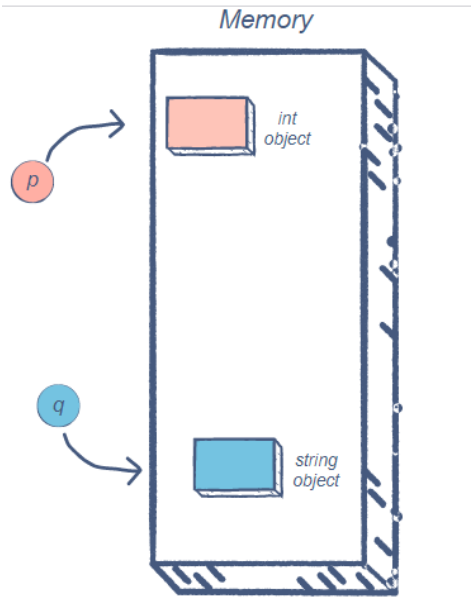
This variable can be of type int, char, array, function, or any other pointer. The size of the pointer depends on the architecture. However, in 32-bit architecture the size of a pointer is 2 byte.

10.2 DECLARING A POINTER

```
type *name;    // points to a value of the specified type
```

Type refers to the data type of the object our pointer points to, and name is just the label of the pointer. The * character specifies that this variable is in fact, a pointer. Here is an example:

```
int *p;   // integer pointer
string *q;   // string pointer
```

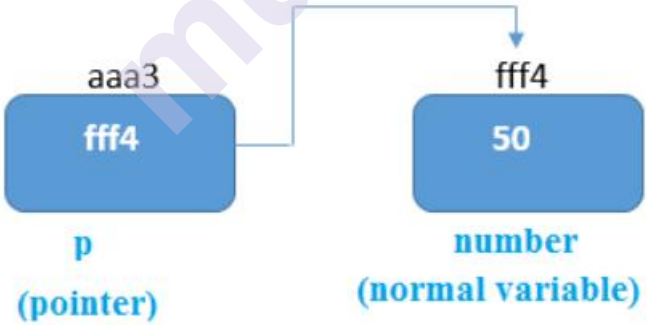


The & character specifies that we are storing the address of the variable succeeding it.

The * character lets us access the value.

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 addre
ss 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
```

10.3 DYNAMIC MEMORY

So, why do we need pointers when we already have normal variables? Well, with pointers we have the power of creating new objects in dynamic memory rather than static memory. A pointer can create a new object in dynamic memory by using the malloc command.

With malloc, we need to specify the amount of bytes we want to reserve in dynamic memory. Since it is a void method, we need to cast the pointer into the data type we want. Here's the template for pointer declaration using malloc:

```
p = (cast type*) malloc(size);
```

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


int main() {
  int *p = (int*) malloc(sizeof(int));   // dynamic memory reserved for an
 integer
  *p = 10;   // the object is assigned the value of 10
```

```
 printf("The value of p: %d\n", *p);

 int *q = p;   // both pointers point to the same object

 printf("The value of q: %d\n", *q);


 int *arr = (int*) malloc(5 * sizeof(int)); // a dynamic array of size 5 is
created.

 free(arr); // releases the designated memory

 free(p);

}
```

Accesses the memory address of the object p points to Pointer Cheat Sheet

Syntax	Purpose
int *p	Declares a pointer p
p = (int*) malloc(sizeof(int))	Creates an integer object in dynamic memory
p = (int*) malloc(n * sizeof(int))	Creates a dynamic array of size n
p = &var	Points p to the var variable
***p**	Accesses the value of the object p points to
***p = 8**	Updates the value of the object p points to
p	Accesses the memory address of the object p points to

10.4 REFERENCING AND DEREFERENCING

Referencing means taking the address of an existing variable (using &) to set a pointer variable. In order to be valid, a pointer has to be set to the address of a variable of the same type as the pointer, without the asterisk:

```
int  c1;
int* p1;
c1 = 5;
p1 = &c1;
```

Dereferencing a pointer means using the * operator (asterisk character) to retrieve the value from the memory address that is pointed by the pointer: NOTE: The value stored at the address of the pointer must be a

value OF THE SAME TYPE as the type of variable the pointer "points" to, but there is **no guarantee** this is the case unless the pointer was set correctly. The type of variable the pointer points to is the type less the outermost asterisk.

```
int n1;
n1 = *p1;
```

Invalid dereferencing may or may not cause crashes:

- Dereferencing an uninitialized pointer can cause a crash
- Dereferencing with an invalid type cast will have the potential to cause a crash.
- Dereferencing a pointer to a variable that was dynamically allocated and was subsequently de-allocated can cause a crash
- Dereferencing a pointer to a variable that has since gone out of scope can also cause a crash.

Invalid referencing is more likely to cause compiler errors than crashes, but it's not a good idea to rely on the compiler for this.

& is the reference operator and can be read as "address of".
** is the dereference operator and can be read as "value pointed by".*

& is the reference operator
** is the dereference operator*

es c1

- & is the reference operator -- it gives you a reference (pointer) to some object
- * is the dereference operator -- it takes a reference (pointer) and gives you back the referred to object;

10.5 POINTER ARITHMETIC

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

1. 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.

32-bit

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

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.

```
#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 ou
r 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
```

Traversing an array by using pointer:

```
#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
```

2. 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)
```

32-bit

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

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.

```
#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
```

3. 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))
```

32-bit

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

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.

```
#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.

4. 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))
```

32-bit

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

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,

```
Address2 -
Address1 = (Subtraction of two addresses)/size of data type which
point er points
```

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

10.6 USING POINTERS WITH ARRAYS

Consider the following program:

```
#include<stdio.h>

int main()
{
int arr[5] = { 1, 2, 3, 4, 5 };
```

```
int *ptr = arr;

printf("%p\n", ptr);
return 0;
}
```

In this program, we have a pointer *ptr* that points to the 0^{th} element of the array. Similarly, we can also declare a pointer that can point to whole array instead of only one element of the array. This pointer is useful when talking about multidimensional arrays.

Syntax:

```
data_type (*var_name)[size_of_array];
```

Example:

```
int (*ptr)[10];
```

Here *ptr* is pointer that can point to an array of 10 integers. Since subscript have higher precedence than indirection, it is necessary to enclose the indirection operator and pointer name inside parentheses. Here the type of ptr is 'pointer to an array of 10 integers'.

Note: The pointer that points to the 0^{th} element of array and the pointer that points to the whole array are totally different. The following program shows this:

```
// C program to understand difference between
// pointer to an integer and pointer to an
// array of integers.
#include<stdio.h>

int main()
{
    // Pointer to an integer
    int *p;

    // Pointer to an array of 5 integers
    int (*ptr)[5];
    int arr[5];

    // Points to 0th element of the arr.
```

```
        p = arr;
        // Points to the whole array arr.
        ptr = &arr;
        printf("p = %p, ptr = %p\n", p, ptr);

        p++;
        ptr++;

        printf("p = %p, ptr = %p\n", p, ptr);

        return 0;
}
```

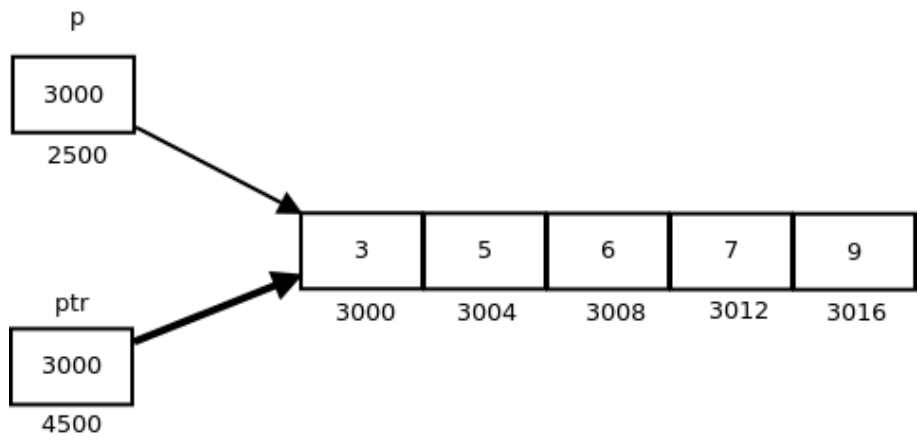
Output:

```
p = 0x7fff4f32fd50, ptr = 0x7fff4f32fd50
p = 0x7fff4f32fd54, ptr = 0x7fff4f32fd64
```

p*:** is pointer to 0th element of the array *arr*, while ***ptr is a pointer that points to the whole array *arr*.

- The base type of *p* is int while base type of *ptr* is 'an array of 5 integers'.
- We know that the pointer arithmetic is performed relative to the base size, so if we write ptr++, then the pointer *ptr* will be shifted forward by 20 bytes.

The following figure shows the pointer p and ptr. Darker arrow denotes pointer to an array.



On dereferencing a pointer expression we get a value pointed to by that pointer expression. Pointer to an array points to an array, so on

dereferencing it, we should get the array, and the name of array denotes the base address. So whenever a pointer to an array is dereferenced, we get the base address of the array to which it points.

```
/ C program to illustrate sizes of
// pointer of array
#include<stdio.h>

int main()
{
    int arr[] = { 3, 5, 6, 7, 9 };
    int *p = arr;
    int (*ptr)[5] = &arr;

    printf("p = %p, ptr = %p\n", p, ptr);
    printf("*p = %d, *ptr = %p\n", *p, *ptr);

    printf("sizeof(p) = %lu, sizeof(*p) = %lu\n",
                    sizeof(p), sizeof(*p));
    printf("sizeof(ptr) = %lu, sizeof(*ptr) = %lu\n",
                    sizeof(ptr), sizeof(*ptr));
    return 0;
}
```

Output:

```
p = 0x7ffde1ee5010, ptr = 0x7ffde1ee5010
*p = 3, *ptr = 0x7ffde1ee5010
sizeof(p) = 8, sizeof(*p) = 4
sizeof(ptr) = 8, sizeof(*ptr) = 20
```

Pointer to Multidimensional Arrays:

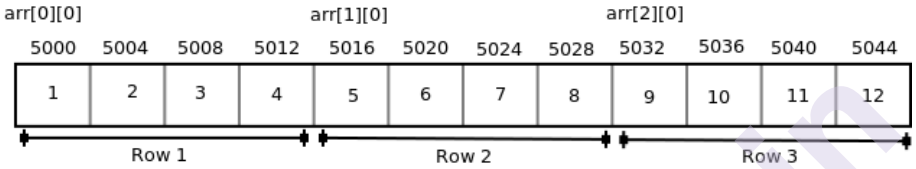
Pointers and two dimensional Arrays: In a two dimensional array, we can access each element by using two subscripts, where first subscript represents the row number and second subscript represents the column number. The elements of 2-D array can be accessed with the help of pointer notation also. Suppose arr is a 2-D array, we can access any element *arr[i][j]* of the array using the pointer expression ***(*(arr + i) + j)**. Now we'll see how this expression can be derived.

Let us take a two dimensional array *arr[3][4]*:

```
int arr[3][4] = { {1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12} };
```

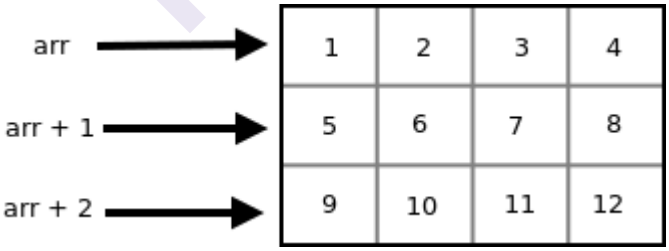
	Col 1	Col 2	Col 3	Col 4
Row 1	1	2	3	4
Row 2	5	6	7	8
Row 3	9	10	11	12

Since memory in a computer is organized linearly it is not possible to store the 2-D array in rows and columns. The concept of rows and columns is only theoretical, actually, a 2-D array is stored in row-major order i.e rows are placed next to each other. The following figure shows how the above 2-D array will be stored in memory.



Each row can be considered as a 1-D array, so a two-dimensional array can be considered as a collection of one-dimensional arrays that are placed one after another. In other words, we can say that 2-D dimensional arrays that are placed one after another. So here *arr* is an array of 3 elements where each element is a 1-D array of 4 integers. We know that the name of an array is a constant pointer that points to 0^{th} 1-D array and contains address 5000. Since *arr* is a 'pointer to an array of 4 integers', according to pointer arithmetic the expression arr + 1 will represent the address 5016 and expression arr + 2 will represent address 5032.

So we can say that *arr* points to the 0^{th} 1-D array, *arr + 1* points to the 1^{st} 1-D array and *arr + 2* points to the 2^{nd} 1-D array.



arr - Points to 0^{th} element of arr - Points to 0^{th} 1-D array - 5000
arr + 1 - Points to 1^{th} element of arr - Points to 1^{nd} 1-D array - 5016
arr + 2 - Points to 2^{th} element of arr - Points to 2^{nd} 1-D array - 5032

In general we can write:

arr + i Points to i^{th} element of arr ->

Points to ith 1-D array

- Since arr + i points to i^{th} element of *arr*, on dereferencing it will get i^{th} element of *arr* which is of course a 1-D array. Thus the expression **(arr + i)* gives us the base address of i^{th} 1-D array.
- We know, the pointer expression **(arr + i)* is equivalent to the subscript expression *arr[i]*. So **(arr + i)* which is same as *arr[i]* gives us the base address of i^{th} 1-D array.
- To access an individual element of our 2-D array, we should be able to access any j^{th} element of i^{th} 1-D array.
- Since the base type of **(arr + i)* is *int* and it contains the address of 0^{th} element of i^{th} 1-D array, we can get the addresses of subsequent elements in the i^{th} 1-D array by adding integer values to **(arr + i)*.
- For example **(arr + i) + 1* will represent the address of 1^{st} element of 1^{st}element of i^{th} 1-D array and **(arr+i)+2* will represent the address of 2^{nd} element of i^{th} 1-D array.
- Similarly *(arr + i) + j will represent the address of j^{th} element of i^{th} 1-D array. On dereferencing this expression we can get the j^{th} element of the i^{th} 1-D array.
- **Pointers and Three Dimensional Arrays**
  In a three dimensional array we can access each element by using three subscripts. Let us take a 3-D array-

```
int arr[2][3][2] = { {{5, 10}, {6, 11}, {7, 12}}, {{20, 30}, {21, 31}, {22, 32}} };
```

We can consider a three dimensional array to be an array of 2-D array i.e each element of a 3-D array is considered to be a 2-D array. The 3-D array *arr* can be considered as an array consisting of two elements where each element is a 2-D array. The name of the array *arr* is a pointer to the 0^{th} 2-D array.

arr	**Points to 0^{th} 2-D array.**
arr + i	**Points to i^{th} 2-D array.**
***(arr + i)**	**Gives base address of i^{th} 2-D array, so points to 0^{th} element of i^{th} 2-D array, each element of 2-D array is a 1-D array, so it points to 0^{th} 1-D array of i^{th} 2-D array.**
***(arr + i) + j**	**Points to j^{th} 1-D array of i^{th} 2-D array.**
***(*(arr + i) + j)**	**Gives base address of j^{th} 1-D array of i^{th} 2-D array so it points to 0^{th} element of j^{th} 1-D array of i^{th} 2-D array.**
***(*(arr + i) + j) + k**	**Reprents the value of j^{th} element of i^{th} 1-D array.**
***(*(arr + i) + j) + k)**	**Gives the value of k^{th} element of j^{th} 1-D array of i^{th} 2-D array.**

Thus the pointer expression ***(*(*(arr + i) + j) + k) is equivalent to the subscript expression arr[i][j][k].**

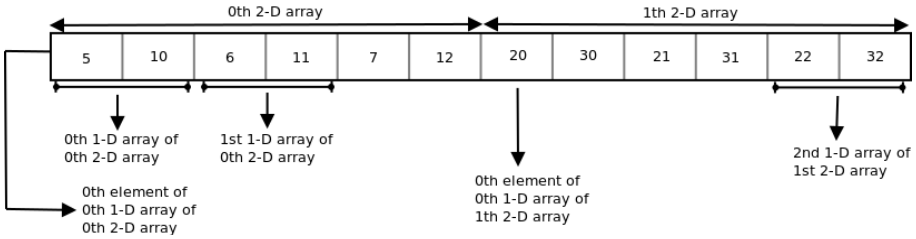
We know the expression *(arr + i) is equivalent to arr[i] and the expression *(*(arr + i) + j) is equivalent arr[i][j]. So we can say that arr[i] represents the base address of i^{th} 2-D array and arr[i][j] represents the base address of the j^{th} 1-D array.

```
// C program to print the elements of 3-D
// array using pointer notation
#include<stdio.h>
int main()
{
  int arr[2][3][2] = {
                      {
                        {5, 10},
                        {6, 11},
                        {7, 12},
                      },
                      {
                        {20, 30},
                        {21, 31},
                        {22, 32},
                      }
                     };
  int i, j, k;
  for (i = 0; i < 2; i++)
  {
    for (j = 0; j < 3; j++)
    {
      for (k = 0; k < 2; k++)
        printf("%d\t", *(*(*(arr + i) + j) +k));
      printf("\n");
    }
  }

  return 0;
}
```

Output:

```
5    10
6    11
7    12
20    30
21    31
22    32
```

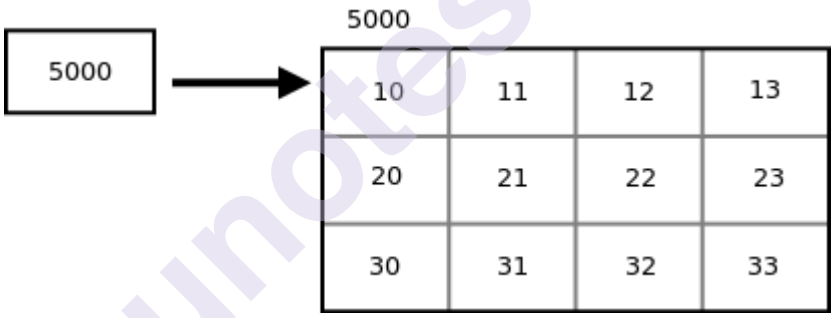
The following figure shows how the 3-D array used in the above program is stored in memory.



Subscripting Pointer to an Array:

Suppose *arr* is a 2-D array with 3 rows and 4 columns and *ptr* is a pointer to an array of 4 integers, and *ptr* contains the base address of array *arr*.

```
int arr[3][4] = {{10, 11, 12, 13}, {20, 21, 22, 23}, {30, 31, 32, 33}};
int (*ptr)[4];
ptr = arr;
```



Since *ptr* is a pointer to an array of 4 integers, *ptr + i* will point to i^{th} row. On dereferencing *ptr + i*, we get base address of i^{th} row. To access the address of j^{th} element of i^{th} row we can add j to the pointer expression **(ptr + i)*. So the pointer expression **(ptr + i) + j* gives the address of j^{th} element of i^{th} row and the pointer expression **(*(ptr + i)+j)* gives the value of the j^{th} element of i^{th} row. We know that the pointer expression *(*(ptr + i) + j) is equivalent to subscript expression ptr[i][j]. So if we have a pointer variable containing the base address of 2-D array, then we can access the elements of array by double subscripting that pointer variable.

```
// C program to print elements of a 2-D array
// by scripting a pointer to an array
#include<stdio.h>

int main()
```

```
{
  int arr[3][4] = {
                    {10, 11, 12, 13},
                    {20, 21, 22, 23},
                    {30, 31, 32, 33}
                  };
  int (*ptr)[4];
  ptr = arr;
  printf("%p %p %p\n", ptr, ptr + 1, ptr + 2);
  printf("%p %p %p\n", *ptr, *(ptr + 1), *(ptr + 2));
  printf("%d %d %d\n", **ptr, *(*(ptr + 1) + 2), *(*(ptr + 2) + 3));
  printf("%d %d %d\n", ptr[0][0], ptr[1][2], ptr[2][3]);
  return 0;
}
```

Output:

```
0x7ffead967560 0x7ffead967570 0x7ffead967580
0x7ffead967560 0x7ffead967570 0x7ffead967580
10 22 33
10 22 33
```

10.7 USING POINTERS WITH STRINGS

In C, a string can be referred to either using a character pointer or as a character array.

Strings as character arrays:

```
char str[4] = "GfG"; /*One extra for string terminator*/
/* OR */
char str[4] = {‘G’, ‘f’, ‘G’, '\0'}; /* '\0' is string terminator */
```

When strings are declared as character arrays, they are stored like other types of arrays in C. For example, if str[] is an auto variable then string is stored in stack segment, if it’s a global or static variable then stored in data segment, etc.

Strings using character pointers:

Using character pointer strings can be stored in two ways:

1) Read only string in a shared segment:

When a string value is directly assigned to a pointer, in most of the compilers, it's stored in a read-only block (generally in data segment) that is shared among functions.

```
char *str = "GfG";
```

In the above line "GfG" is stored in a shared read-only location, but pointer str is stored in a read-write memory. You can change str to point something else but cannot change value at present str. So this kind of string should only be used when we don't want to modify string at a later stage in the program.

2) Dynamically allocated in heap segment:

Strings are stored like other dynamically allocated things in C and can be shared among functions.

p

```
char *str;
int size = 4; /*one extra for '\0'*/
str = (char *)malloc(sizeof(char)*size);
*(str+0) = 'G';
*(str+1) = 'f';
*(str+2) = 'G';
*(str+3) = '\0';
```


Let us see some examples to better understand the above ways to store strings.

Example 1 (Try to modify string):

The below program may crash (gives segmentation fault error) because the line *(str+1) = 'n' tries to write a read only memory.

```
int main()
{
char *str;
str = "GfG";        /* Stored in read only part of data segment */
*(str+1) = 'n'; /* Problem: trying to modify read only memory */
getchar();
return 0;
}
```

The below program works perfectly fine as str[] is stored in writable stack segment.

```
int main()
{
char str[] = "GfG"; /* Stored in stack segment like other auto variables */
*(str+1) = 'n'; /* No problem: String is now GnG */
getchar();
return 0;
}
```

Example 2 (Try to return string from a function):

The below program works perfectly fine as the string is stored in a shared segment and data stored remains there even after return of getString()

```
char *getString()
{
char *str = "GfG"; /* Stored in read only part of shared segment */

/* No problem: remains at address str after getString() returns*/
return str;
}

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

The below program also works perfectly fine as the string is stored in heap segment and data stored in heap segment persists even after the return of getString()

```
char *getString()
{
int size = 4;
char *str = (char *)malloc(sizeof(char)*size); /*Stored in heap segment*/
```

```
*(str+0) = 'G';
*(str+1) = 'f';
*(str+2) = 'G';
*(str+3) = '\0';

/* No problem: string remains at str after getString() returns */
return str;
}
int main()
{
printf("%s", getString());
getchar();
return 0;
}
```

But, the below program may print some garbage data as string is stored in stack frame of function getString() and data may not be there after getString() returns.

```
char *getString()
{
char str[] = "GfG"; /* Stored in stack segment */

/* Problem: string may not be present after getString() returns */
/* Problem can be solved if write static before char, i.e. static char str[] = "GfG";*/
return str;
}
int main()
{
printf("%s", getString());
getchar();
return 0;
}
```

10.8 ARRAY OF POINTERS

Just like we can declare an array of int, float or char etc, we can also declare an array of pointers, here is the syntax to do the same.

Syntax: datatype *array_name[size];

Let's take an example:

```
int *arrop[5];
```

Here arrop is an array of 5 integer pointers. It means that this array can hold the address of 5 integer variables. In other words, you can assign 5 pointer variables of type pointer to int to the elements of this array.

The following program demonstrates how to use an array of pointers.

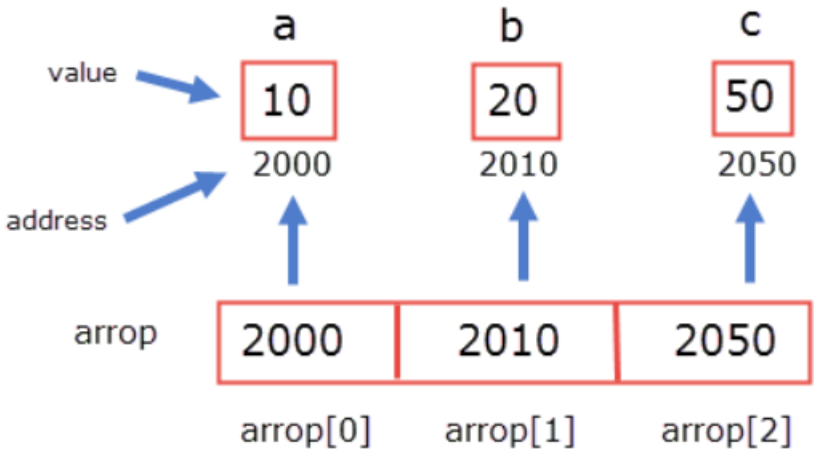
```
#include<stdio.h>
#define SIZE 10

int main()
{
   int *arrop[3];
   int a = 10, b = 20, c = 50, i;

   arrop[0] = &a;
   arrop[1] = &b;
   arrop[2] = &c;

   for(i = 0; i < 3; i++)
   {
      printf("Address = %d\t Value = %d\n", arrop[i], *arrop[i]);
   }

   return 0;
}
```

Expected Output:

```
Address = 387130656     Value = 10
Address = 387130660     Value = 20
Address = 387130664     Value = 50
```

How it works:

Notice how we are assigning the addresses of a, b and c. In line 9, we are assigning the address of variable a to the 0th element of the of the array. Similarly, the address of b and c is assigned to 1st and 2nd element respectively. At this point, the arrop looks something like this:



arrop[i] gives the address of ith element of the array. So arrop[0] returns address of variable a, arrop[1] returns address of b and so on. To get the value at address use indirection operator (*).

```
*arrop[i]
```

So, *arrop[0] gives value at address arrop[0], Similarly *arrop[1] gives the value at address arrop[1] and so on.

10.9 POINTERS AS FUNCTION ARGUMENTS AND FUNCTIONS RETURNING POINTERS.

Pointer as a function parameter is used to hold addresses of arguments passed during function call. This is also known as **call by reference**. When a function is called by reference any change made to the reference variable will effect the original variable.

Example Time: Swapping two numbers using Pointer

```
#include <stdio.h>

void swap(int *a, int *b);

int main()
{
   int m = 10, n = 20;
   printf("m = %d\n", m);
   printf("n = %d\n\n", n);

   swap(&m, &n);      //passing address of m and n to the swap function
   printf("After Swapping:\n\n");
   printf("m = %d\n", m);
```

```
    printf("n = %d", n);
    return 0;
}

/*
    pointer 'a' and 'b' holds and
    points to the address of 'm' and 'n'
*/
void swap(int *a, int *b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

```
Output
m = 10
n = 20
After Swapping:

m = 20
n = 10
```


Functions returning Pointer variables:

A function can also return a pointer to the calling function. In this case you must be careful, because local variables of function doesn't live outside the function. They have scope only inside the function. Hence if you return a pointer connected to a local variable, that pointer will be pointing to nothing when the function ends.

```
#include <stdio.h>

int* larger(int*, int*);

void main()
{
    int a = 15;
```

```
    int b = 92;
    int *p;
    p = larger(&a, &b);
    printf("%d is larger",*p);
}

int* larger(int *x, int *y)
{
    if(*x > *y)
        return x;
    else
        return y;
}
```

```
Output
92 is larger
```

Safe ways to return a valid Pointer:

1. Either use **argument with functions**. Because argument passed to the functions are declared inside the calling function, hence they will live outside the function as well.
2. Or, use static local variables inside the function and return them. As static variables have a lifetime until the main() function exits, therefore they will be available througout the program.

10.10 SUMMARY

- In C, a **pointer** holds the address of an object stored in memory. The pointer then simply "points" to the object. The type of the object must correspond with the type of the pointe
- **Referencing** means taking the address of an existing variable (using &) to set a pointer variable.r.
- **Dereferencing** a pointer means using the * operator (asterisk character) to retrieve the value from the memory address that is pointed by the pointer
- If we increment a pointer by 1, the pointer will start pointing to the immediate next location.
- Like increment, we can decrement a pointer variable. If we decrement a pointer, it will start pointing to the previous location.

- Like pointer addition, we can subtract a value from the pointer variable. Subtracting any number from a pointer will give an address.
- In C, a string can be referred to either using a character pointer or as a character array.

Strings as character arrays:

- Just like we can declare an array of int, float or char etc, we can also declare an array of pointers.
- Pointer as a function parameter is used to hold addresses of arguments passed during function call. This is also known as **call by reference**

10.11 UNIT END QUESTIONS

1. How to declare pointers in C?
2. Why do we need pointers when we already have normal variables?
3. What do you mean by Referencing and Dereferencing?
4. Write a short note on Pointer Arithmetic.
5. Write a C program using Pointers with Arrays.
6. Write a short note on **Pointer to Multidimensional Arrays.**
7. **Write a program** using Pointers with Strings
8. Explain Array of Pointers with example.

10.12 REFERENCE FOR FURTHER READING

- https://www.educative.io/edpresso/what-is-a-pointer-in-c?
- https://newbedev.com/meaning-of-referencing-and-dereferencing-in-c
- http://www.codingunit.com/cplusplus-tutorial-pointers-reference-and-dereference-operators
- http://www.cplusplus.com/doc/tutorial/pointers/
- https://overiq.com/c-programming-101/array-of-pointers-in-c/

*****

11

DYNAMIC MEMORY ALLOCATION

Unit Structure

11.0 OBJECTIVE

In this Chapter, you'll learn to dynamically allocate memory in your C program using standard library functions: malloc(), calloc(), free() and realloc().

11.1 INTRODUCTION

As you know, an array is a collection of a fixed number of values. Once the size of an array is declared, you cannot change it.

Sometimes the size of the array you declared may be insufficient. To solve this issue, you can allocate memory manually during run-time. This is known as dynamic memory allocation in C programming.

To allocate memory dynamically, library functions are malloc(), calloc(), realloc() and free() are used. These functions are defined in the <stdlib.h> header file.

11.2 MALLOC() FUNCTION

The malloc() function stands for memory allocation. It is a function which is used to allocate a block of memory dynamically. It reserves memory space of specified size and returns the null pointer pointing to the memory location. The pointer returned is usually of type void. It means that we can assign malloc function to any pointer.

Syntax:

```
ptr = (cast_type *) malloc (byte_size);
```

Here,

- ptr is a pointer of cast_type.
- The malloc function returns a pointer to the allocated memory of byte_size.

```
Example: ptr = (int *) malloc (50)
```

When this statement is successfully executed, a memory space of 50 bytes is reserved. The address of the first byte of reserved space is assigned to the pointer ptr of type int.

Consider another example of malloc implementation:

```
#include <stdlib.h>
int main(){
int *ptr;
ptr = malloc(15 * sizeof(*ptr)); /* a block of 15 integers */
   if (ptr != NULL) {
     *(ptr + 5) = 480; /* assign 480 to sixth integer */
     printf("Value of the 6th integer is %d",*(ptr + 5));
   }
}
```

Output:

```
Value of the 6th integer is 480
```

1. Notice that sizeof(*ptr) was used instead of sizeof(int) in order to make the code more robust when *ptr declaration is typecasted to a different data type later.
2. The allocation may fail if the memory is not sufficient. In this case, it returns a NULL pointer. So, you should include code to check for a NULL pointer.
3. Keep in mind that the allocated memory is contiguous and it can be treated as an array. We can use pointer arithmetic to access the array elements rather than using brackets [ ]. We advise to use + to refer to array elements because using incrementation ++ or += changes the address stored by the pointer.

How to use "malloc" in C:

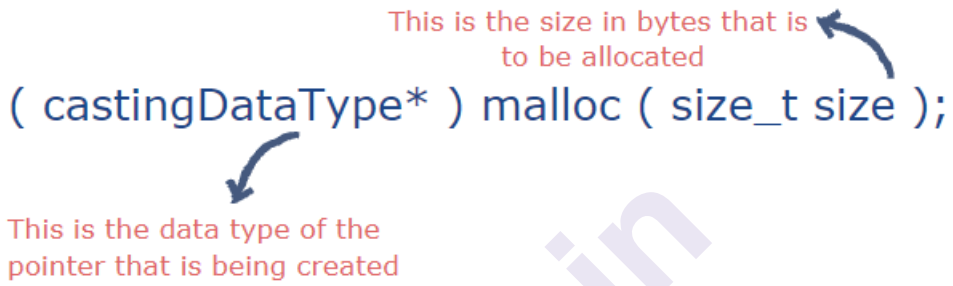
Memory allocation (malloc), is an in-built function in C. This function is used to assign a specified amount of memory for an array to be created. It also returns a pointer to the space allocated in memory using this function.

The need for malloc:

In the world of programming where every space counts, there are numerous times when we only want an array to have a specific amount of space at run time. That is, we want to create an array occupying a particular amount of space, dynamically. We do this using *malloc*.

Syntax

We know what malloc returns and we know what it requires as an input, but how does the syntax of the function work. The illustration below shows that:



Note: malloc will return NULL if the memory specified is not available and hence, the allocation has failed

Example:

Now that we know how malloc is used and why it is needed, let's look at a few code examples to see how it is used in the code.

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

   int* ptr1;
   // We want ptr1 to store the space of 3 integers
   ptr1 = (int*) malloc (3 * sizeof(int));

   if(ptr1==NULL){
     printf("Memory not allocated. \n");
   }
   else{printf("Memory allocated succesfully. \n");
   // This statement shows where memory is allocated
   printf("The address of the pointer is:%u\n ", ptr1);

   // Here we assign values to the ptr1 created
```

```
    for(int i=0;i<3;i++){
      ptr1[i] = i;
    }
    // Printing the vlaues of ptr1 to show memory allocation is done
    for(int i=0;i<3;i++){
      printf("%d\n", ptr1[i]);
    }

    }

}
```

When the amount of memory is not needed anymore, you must return it to the operating system by calling the function free.

Take a look at the following example:

```
#include<stdio.h>

        int main()
        {
                int *ptr_one;

                ptr_one = (int *)malloc(sizeof(int));

                if (ptr_one == 0)
                {
                        printf("ERROR: Out of memory\n");
                        return 1;
                }
                *ptr_one = 25;
                printf("%d\n", *ptr_one);

                free(ptr_one);

                return 0;
        }
```

Note: If you compile on windows the windows.h file should be included to use malloc.

The malloc statement will ask for an amount of memory with the size of an integer (32 bits or 4 bytes). If there is not enough memory available, the malloc function will return a NULL. If the request is granted a block of memory is allocated (reserved). The address of the reserved block will be placed into the pointer variable.

The if statement then checks for the return value of NULL. If the return value equals NULL, then a message will be printed and the programs stops. (If the return value of the program equals one, than that's an indication that there was a problem.)

The number twenty-five is placed in the allocated memory. Then the value in the allocated memory will be printed. Before the program ends the reserved memory is released.

Malloc and structures:

A structure can also be used in a malloc statement.

Take a look at the example:

```
#include<stdio.h>

    typedef struct rec
    {
        int i;
        float PI;
        char A;
    }RECORD;

    int main()
    {
        RECORD *ptr_one;

        ptr_one = (RECORD *) malloc (sizeof(RECORD));

        (*ptr_one).i = 10;
        (*ptr_one).PI = 3.14;
        (*ptr_one).A = 'a';

        printf("First value: %d\n",(*ptr_one).i);
```

```
        printf("Second value: %f\n", (*ptr_one).PI);
        printf("Third value: %c\n", (*ptr_one).A);

        free(ptr_one);

        return 0;
}
```

Note: the parentheses around *ptr_one in the printf statements. This notation is not often used. Most people will use ptr_one->i instead. So (*ptr_one).i = 25 and ptr_one->i = 25 are the same.

If you want to use the structure without the typedef the program will look like this:

```
#include<stdio.h>

struct rec
{
        int i;
        float PI;
        char A;
};

int main()
{
        struct rec *ptr_one;
        ptr_one =(struct rec *) malloc (sizeof(struct rec));

        ptr_one->i = 10;
        ptr_one->PI = 3.14;
        ptr_one->A = 'a';

        printf("First value: %d\n", ptr_one->i);
        printf("Second value: %f\n", ptr_one->PI);
        printf("Third value: %c\n", ptr_one->A);

        free(ptr_one);
        return 0;
}
```

11.3 CALLOC() FUNCTION

The **calloc()** in C is a function used to allocate multiple blocks of memory having the same size. It is a dynamic memory allocation function that allocates the memory space to complex data structures such as arrays and structures and returns a void pointer to the memory. Calloc stands for contiguous allocation.

Malloc function is used to allocate a single block of memory space while the calloc function in C is used to allocate multiple blocks of memory space. Each block allocated by the calloc in C programming is of the same size.

calloc() Syntax:

```
ptr = (cast_type *) calloc (n, size);
```

- The above statement example of calloc in C is used to allocate n memory blocks of the same size.
- After the memory space is allocated, then all the bytes are initialized to zero.
- The pointer which is currently at the first byte of the allocated memory space is returned.

Whenever there is an error allocating memory space such as the shortage of memory, then a null pointer is returned as shown in the below calloc example.

How to use calloc

The below calloc program in C calculates the sum of an arithmetic sequence.

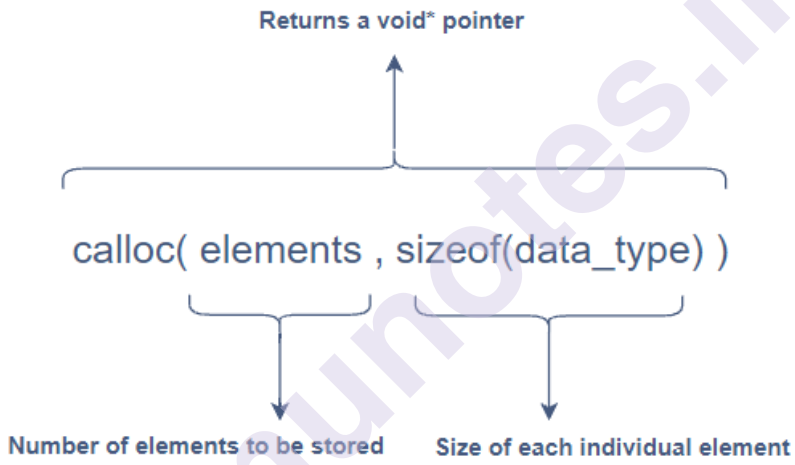
```
#include <stdio.h>
   int main() {
      int i, * ptr, sum = 0;
      ptr = calloc(10, sizeof(int));
      if (ptr == NULL) {
         printf("Error! memory not allocated.");
         exit(0);
      }
      printf("Building and calculating the sequence sum of the first 10 terms \ n ");
      for (i = 0; i < 10; ++i) { * (ptr + i) = i;
         sum += * (ptr + i);
```

```
    }
    printf("Sum = %d", sum);
    free(ptr);
    return 0;
  }
```

Result of the calloc in C example:

```
Building and calculating the sequence sum of the first 10 terms
Sum = 45
```

The calloc() **function in C** is used to allocate a specified amount of memory and then initialize it to *zero*. The function returns a *void pointer* to this memory location, which can then be cast to the desired type. The function takes in two parameters that collectively specify the amount of memory to be allocated.



Code:

Take a look at the code below. Note how `(int*)` is used to convert the void pointer to an `int` pointer.

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

int main() {

   int* a = (int*) calloc(5, sizeof(int));

   return 0;
}
```

11.4 REALLOC() FUNCTION

In the C Programming Language, the **realloc function** is used to resize a block of memory that was previously allocated. The realloc function allocates a block of memory (which be can make it larger or smaller in size than the original) and copies the contents of the old block to the new block of memory, if necessary.

Synta:

The syntax for the realloc function in the C Language is:

```
void *realloc(void *ptr, size_t size);
```

Parameters or Arguments

ptr

The old block of memory.

size

The size of the elements in bytes.

Note

- *ptr* must have been allocated by one of the following functions - calloc function, malloc function, or realloc function.

Returns

The realloc function returns a pointer to the beginning of the block of memory. If the block of memory can not be allocated, the realloc function will return a null pointer.

Required Header

In the C Language, the required header for the realloc function is:

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

Applies To

In the C Language, the realloc function can be used in the following versions:

- ANSI/ISO 9899-1990

realloc Example

Let's look at an example to see how you would use the realloc function in a C program:

```
/* The size of memory allocated MUST be larger than the data you will put in it */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, const char * argv[])
{
   /* Define required variables */
   char *ptr1, *ptr2;
   size_t length1, length2;

   /* Define the amount of memory required */
   length1 = 10;
   length2 = 30;

   /* Allocate memory for our string */
   ptr1 = (char *)malloc(length1);

   /* Check to see if we were successful */
   if (ptr1 == NULL)
   {
      /* We were not successful, so display a message */
      printf("Could not allocate required memory\n");

      /* And exit */
      exit(1);
   }

   /* Copy a string into the allocated memory */
   strcpy(ptr1, "C malloc");

   /* Oops, we wanted to say more but now do not
    have enough memory to store the message! */

   /* Expand the available memory with realloc */
   ptr2 = (char *)realloc(ptr1, length2);
```

```
   /* Check to see if we were successful */
   if (ptr2 == NULL)
   {
      /* We were not successful, so display a message */
      printf("Could not re-allocate required memory\n");

      /* And exit */
      exit(1);
   }

   /* Add the rest of the message to the string */
   strcat(ptr2, " at TechOnTheNet.com");

   /* Display the complete string */
   printf("%s\n", ptr2);

   /* Free the memory we allocated */
   free(ptr2);

   return 0;
}
```

11.5 FREE() FUNCTION

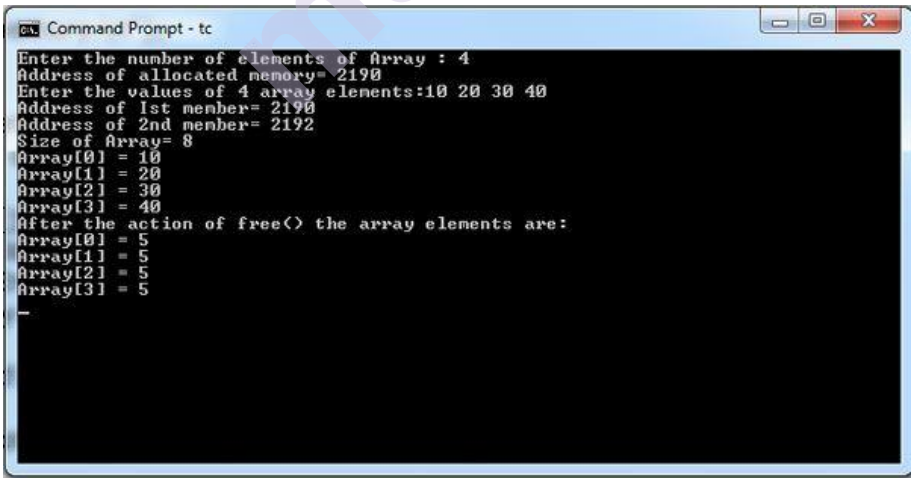
The function free() is used to de-allocate the memory allocated by the functions malloc ( ), calloc ( ), etc, and return it to heap so that it can be used for other purposes. The argument of the function free ( ) is the pointer to the memory which is to be freed. The prototype of the function is as below.

void free(void *ptr);

When free () is used for freeing memory allocated by malloc () or realloc (),the whole allocated memory block is released. For the memory allocated by function calloc () also all the segments of memory allocated by calloc () are de-allocated by free (). This is illustrated by Program. The following program demonstrates the use of function calloc () and the action of function free () on the memories allocated by calloc () .

Illustrates that function free () frees all blocks of memory allocated by calloc () function:

```
#include <stdio.h>
 #include<stdlib.h>
 main()
 {
   int i,j,k, n ;
   int* Array;
   clrscr();
   printf("Enter the number of elements of Array : ");
   scanf("%d", &n );
   Array= (int*) calloc(n, sizeof(int));
   if( Array== (int*)NULL)
    {
       printf("Error. Out of memory.\n");
       exit (0);
    }
     printf("Address of allocated memory= %u\n" , Array);
     printf("Enter the values of %d array elements:", n);
      for (j =0; j<n; j++)
           scanf("%d",&Array[j]);
           printf("Address of Ist member= %u\n", Array);
           printf("Address of 2nd member= %u\n", Array+1);
           printf("Size of Array= %u\n", n* sizeof(int) );
   for ( i = 0 ; i<n; i++)
     printf("Array[%d] = %d\n", i, *(Array+i));
     free(Array);
     printf("After the action of free() the array elements are:\n");
     for (k =0;k<n; k++)
       printf("Array[%d] = %d\n", k, *(Array+i));
        return 0;
}
```



Why do we need to deallocate the dynamic memory?

When we try to create the dynamic memory, the memory will be created in the heap section.

Memory leak

If we don't deallocate the dynamic memory, it will reside in the heap section. It is also called memory leak.

It will reduce the system performance by reducing the amount of available memory.

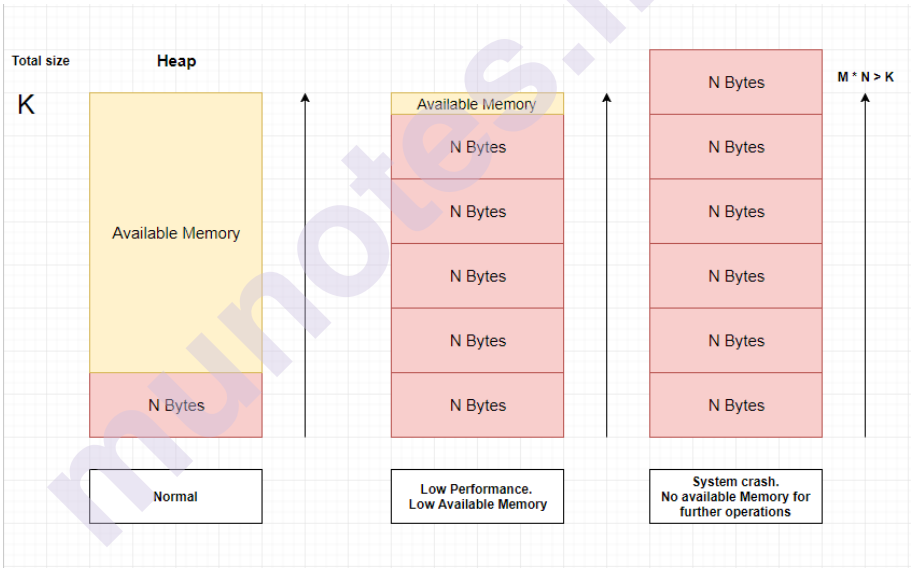
Let's assume total heap size as K.

If we allocate N byte of memory dynamically, it will consume N bytes of memory in heap section.

When the particular piece of code executed M number of time, then M * N bytes of memory will be consumed by our program.

At some point in time (M * N > K), the whole heap memory will be consumed by the program it will result in the system crash due to low available memory.

Pictorial Explanation



So, it is programmers responsibility to deallocate the dynamic memory which is no longer needed.

How to deallocate the dynamic memory?

using free() function, we can deallocate the dynamic memory.

Syntax of free

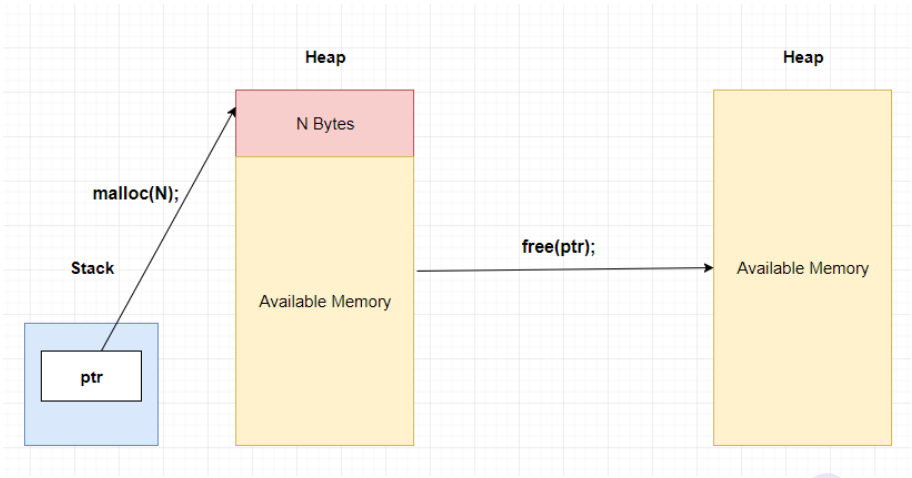
```
free(ptr);
```

Example

```
char *ptr;
ptr = malloc(N);
//do something
```

free(ptr);

Pictorial Explanation



11.6 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.

Note:
The output can vary on different machines such as on 32-bit operating system will show different output, and the 64-bit operating system will show the different outputs of the same data types.

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 character data type.

    printf("\nsize of the floating data type is %d",sizeof(float)); //Displaying the size of floating 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:

```
size of the variable x is 4
size of the integer data type is 4
size of the character data type is 1
size of the floating data type is 4

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

When operand is an expression

```
1.  #include <stdio.h>
2.  int main()
3.  {
4.    double i=78.0; //variable initialization.
5.    float j=6.78; //variable initialization.
6.    printf("size of (i+j) expression is : %d",sizeof(i+j)); //Displaying the size of the expression (i+j).
7.    return 0;
8.  }
```

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
```

11.7 SUMMARY

- Sometimes the size of the array you declared may be insufficient. To solve this issue, you can allocate memory manually during run-time. This is known as dynamic memory allocation in C programming.
- The malloc() function stands for memory allocation. It is a function which is used to allocate a block of memory dynamically
- In the world of programming where every space counts, there are numerous times when we only want an array to have a specific amount of space at run time. That is, we want to create an array occupying a particular amount of space, dynamically. We do this using *malloc*.
- The **calloc()** in C is a function used to allocate multiple blocks of memory having the same size.
- In the C Programming Language, the **realloc function** is used to resize a block of memory that was previously allocated. The realloc function allocates a block of memory (which be can make it larger or smaller in size than the original) and copies the contents of the old block to the new block of memory, if necessary.
- The function free() is used to de-allocate the memory allocated by the functions malloc ( ), calloc ( ), etc, and return it to heap so that it can be used for other purposes.
- 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.

11.8 UNIT END QUESTIONS

1. How to use "malloc" in C?
2. The need for malloc.
3. How to implement calloc()?
4. Write a short note on realloc().
5. What is the use of free()?

6. Explain Sizeof operator.
7. Differentiate between malloc(), calloc(), realloc().
8. Why do we need to deallocate the dynamic memory?

11.9 REFERENCE FOR FURTHER READING

- https://www.guru99.com/malloc-in-c-example.html
- https://www.educative.io/edpresso/how-to-use-malloc-in-c
- https://www.codingunit.com/c-tutorial-the-functions-malloc-and-free
- https://www.educative.io/edpresso/what-is-calloc-in-c
- https://www.guru99.com/calloc-in-c-example.html
- https://www.techonthenet.com/c_language/standard_library_functions/stdlib_h/realloc.php
- https://ecomputernotes.com/data-structures/basic-of-data-structure/free-function
- https://www.log2base2.com/C/pointer/free-in-c.html
- https://www.digi.com/resources/documentation/digidocs/90001537/references/r_python_garbage_coll.htm

*****

12

STRUCTURE

Unit Structure

12.0 OBJECTIVE

In this tutorial, you'll learn about struct types in C Programming with the help of examples.

12.1 INTRODUCTION

In C programming, a struct (or structure) is a collection of variables (can be of different types) under a single name.

Structure is a group of variables of different data types represented by a single name. Lets take an example to understand the need of a structure in C programming.

12.2 DECLARATION OF STRUCTURE

Lets say we need to store the data of students like student name, age, address, id etc. One way of doing this would be creating a different variable for each attribute, however when you need to store the data of multiple students then in that case, you would need to create these several variables again for each student. This is such a big headache to store data in this way.

We can solve this problem easily by using structure. We can create a structure that has members for name, id, address and age and then we can create the variables of this structure for each student. This may sound confusing, do not worry we will understand this with the help of example.

We use **struct** keyword to create a **structure in C**. The struct keyword is a short form of **structured data type.**

```
struct struct_name {
   DataType member1_name;
   DataType member2_name;
   DataType member3_name;
   ...
};
```

Here struct_name can be anything of your choice. Members data type can be same or different. Once we have declared the structure we can use the struct name as a data type like int, float etc.

First we will see the syntax of creating struct variable, accessing struct members etc and then we will see a complete example.

How to declare variable of a structure?:

```
struct  struct_name  var_name;
```

Or,

```
struct struct_name {
   DataType member1_name;
   DataType member2_name;
   DataType member3_name;
   ...
} var_name;
```

How to access data members of a structure using a struct variable?

```
var_name.member1_name;
var_name.member2_name;
...
```

How to assign values to structure members?:

There are three ways to do this.

1) Using Dot(.) operator:

```
var_name.memeber_name = value;
```

2) All members assigned in one statement:

```
struct struct_name var_name =
{value for memeber1, value for memeber2 ...so on for all the members}
```

3) Designated initializers: We will discuss this later at the end of this post.

Example of Structure in C:

```
#include <stdio.h>
/* Created a structure here. The name of the structure is
 * StudentData.
 */
struct StudentData{
   char *stu_name;
   int stu_id;
   int stu_age;
};
int main()
{
    /* student is the variable of structure StudentData*/
    struct StudentData student;

    /*Assigning the values of each struct member here*/
    student.stu_name = "Steve";
    student.stu_id = 1234;
    student.stu_age = 30;
    /* Displaying the values of struct members */
    printf("Student Name is: %s", student.stu_name);
    printf("\nStudent Id is: %d", student.stu_id);
    printf("\nStudent Age is: %d", student.stu_age);
    return 0;
}
```

Output:

```
Student Name is: Steve
Student Id is: 1234
Student Age is: 30
```

12.3 ARRAY OF STRUCTURES

Declaring an array of structure is same as declaring an array of fundamental types. Since an array is a collection of elements of the same type. In an array of structures, each element of an array is of the structure type.

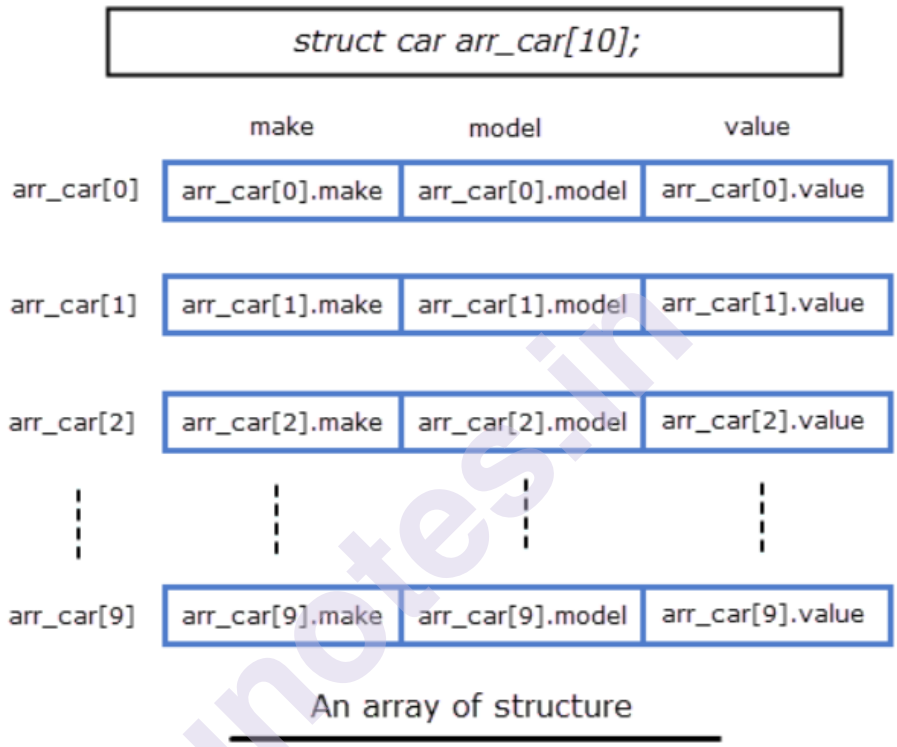
Let's take an example:

```
struct car
{
   char make[20];
```

```
    char model[30];
    int year;
};
```

Here is how we can declare an array of structure car.

```
struct car arr_car[10];
```



An array of structure

Here arr_car is an array of 10 elements where each element is of type struct car. We can use arr_car to store 10 structure variables of type struct car. To access individual elements we will use subscript notation ([]) and to access the members of each element we will use dot (.) operator as usual.

arr_stu[0] : points to the 0th element of the array.

arr_stu[1] : points to the 1st element of the array.

and so on. Similarly,

arr_stu[0].name : refers to the name member of the 0th element of the array.

arr_stu[0].roll_no : refers to the roll_no member of the 0th element of the array.

arr_stu[0].marks : refers to the marks member of the 0th element of the array.

Recall that the precedence of [] array subscript and dot(.) operator is same and they evaluates from left to right. Therefore in the above expression first array subscript([]) is applied followed by dot (.) operator. The array subscript ([]) and dot(.) operator is same and they evaluates from left to right. Therefore in the above expression first [] array subscript is applied followed by dot (.) operator.

Let's rewrite the program we used in the last chapter as an introduction to structures.

```
#include<stdio.h>
#include<string.h>
#define MAX 2

struct student
{
   char name[20];
   int roll_no;
   float marks;
};

int main()
{
   struct student arr_student[MAX];
   int i;

   for(i = 0; i < MAX; i++ )
   {
      printf("\nEnter details of student %d\n\n", i+1);

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

      printf("Enter roll no: ");
      scanf("%d", &arr_student[i].roll_no);

      printf("Enter marks: ");
      scanf("%f", &arr_student[i].marks);
   }
```

```
    printf("\n");

    printf("Name\tRoll no\tMarks\n");

    for(i = 0; i < MAX; i++ )
    {
        printf("%s\t%d\t%.2f\n",
        arr_student[i].name,                    arr_student[i].roll_no,
arr_student[i].marks);
    }

    // signal to operating system program ran fine
    return 0;
}
```

Expected Output:

```
Enter details of student 1

Enter name: Jim
Enter roll no: 1
Enter marks: 44

Enter details of student 2

Enter name: Tim
Enter roll no: 2
Enter marks: 76

Name Roll no Marks
Jim 1 44.00
Tim 2 76.00
```

How it works:

In lines 5-10, we have declared a structure called the student.

In line 14, we have declared an array of structures of type struct student whose size is controlled by symbolic constant MAX. If you want to

increase/decrease the size of the array just change the value of the symbolic constant and our program will adapt to the new size.

In line 17-29, the first for loop is used to enter the details of the student.

In line 36-40, the second for loop prints all the details of the student in tabular form.

Initializing Array of Structures

We can also initialize the array of structures using the same syntax as that for initializing arrays. Let's take an example:

```
pstruct car
{
   char make[20];
   char model[30];
   int year;
};
struct car arr_car[2] = {
                    {"Audi", "TT", 2016},
                    {"Bentley", "Azure", 2002}
                };
```

12.4 ARRAYS WITHIN STRUCTURES

Since the beginning of this chapter, we have already been using arrays as members inside structures. Nevertheless, let's discuss it one more time. For example:

```
struct student
{
   char name[20];
   int roll_no;
   float marks;
};
```

The student structure defined above has a member name which is an array of 20 characters.

Let's create another structure called student to store name, roll no and marks of 5 subjects.

```
struct student
{
```

```
   char name[20];
   int roll_no;
   float marks[5];
};
```

If student_1 is a variable of type struct student then:

student_1.marks[0] - refers to the marks in the first subject

student_1.marks[1] - refers to the marks in the second subject

and so on. Similarly, if arr_student[10] is an array of type struct student then:

arr_student[0].marks[0] - refers to the marks of first student in the first subject arr_student[1].marks[2] - refers to the marks of the second student in the third subject and so on.

The following program asks the user to enter name, roll no and marks in 2 subjects and calculates the average marks of each student.

```
#include<stdio.h>
#include<string.h>
#define MAX 2
#define SUBJECTS 2

struct student
{
   char name[20];
   int roll_no;
   float marks[SUBJECTS];
};

int main()
{
   struct student arr_student[MAX];
   int i, j;
   float sum = 0;

   for(i = 0; i < MAX; i++ )
   {
      printf("\nEnter details of student %d\n\n", i+1);

      printf("Enter name: ");
      scanf("%s", arr_student[i].name);
```

```
        printf("Enter roll no: ");
        scanf("%d", &arr_student[i].roll_no);

        for(j = 0; j < SUBJECTS; j++)
        {
            printf("Enter marks: ");
            scanf("%f", &arr_student[i].marks[j]);
        }
    }
    printf("\n");

    printf("Name\tRoll no\tAverage\n\n");
    for(i = 0; i < MAX; i++ )
    {
        sum = 0;

        for(j = 0; j < SUBJECTS; j++)
        {
            sum += arr_student[i].marks[j];
        }
        printf("%s\t%d\t%.2f\n",
             arr_student[i].name, arr_student[i].roll_no, sum/SUBJECTS);
    }

    // signal to operating system program ran fine
    return 0;
}
```

Expected Output:

```
Enter details of student 1

Enter name: Rick
Enter roll no: 1
Enter marks: 34
Enter marks: 65

Enter details of student 2

Enter name: Tim
```

```
Enter roll no: 2
Enter marks: 35
Enter marks: 85

Name Roll no Average

Rick 1 49.50
Tim 2 60.00
```

How it works:

In line 3 and 4, we have declared two symbolic constants MAX and SUBJECTS which controls the number of students and subjects respectively.

In lines 6-11, we have declared a structure student which have three members namely name, roll_no and marks.

In line 15, we have declared an array of structures arr_student of size MAX.

In line 16, we have declared two int variables i, j to control loops.

In line 17, we have declared a float variable sum and initialized it to 0. This variable will be used to accumulate marks of a particular student.

In line 19-34, we have a for loop which asks the user to enter the details of the student. Inside this for loop, we have a nested for loop which asks the user to enter the marks obtained by the students in various subjects.

In line 40-50, we have another for loop whose job is to print the details of the student. Notice that after each iteration the sum is reinitialized to 0, this is necessary otherwise we will not get the correct answer. The nested for loop is used to accumulate the marks of a particular student in the variable sum. At last the print statement in line 48, prints all the details of the student.

12.5 STRUCTURES WITHIN STRUCTURES

A structure can be nested inside another structure. In other words, the members of a structure can be of any other type including structure. Here is the syntax to create nested structures.

Syntax:

```
structure tagname_1
{
   member1;
   member2;
   member3;
```

```
    ...
    membern;

    structure tagname_2
    {
        member_1;
        member_2;
        member_3;
        ...
        member_n;
    }, var1

} var2;
```

Note: Nesting of structures can be extended to any level.

To access the members of the inner structure, we write a variable name of the outer structure, followed by a dot(.) operator, followed by the variable of the inner structure, followed by a dot(.) operator, which is then followed by the name of the member we want to access.

var2.var1.member_1 - refers to the member_1 of structure tagname_2

var2.var1.member_2 - refers to the member_2 of structure tagname_2

and so on.

Let's take an example:

```
struct student
{
    struct person
    {
        char name[20];
        int age;
        char dob[10];
    } p ;

    int rollno;
    float marks;
} stu;
```

Here we have defined structure person as a member of structure student. Here is how we can access the members of person structure.

stu.p.name - refers to the name of the person

stu.p.age - refers to the age of the person

stu.p.dob - refers to the date of birth of the person

It is important to note that structure person doesn't exist on its own. We can't declare structure variable of type struct person anywhere else in the program.

Instead of defining the structure inside another structure. We could have defined it outside and then declare it's variable inside the structure where we want to use it. For example:

```
struct person
{
   char name[20];
   int age;
   char dob[10];
};
```

We can use this structure as a part of a bigger structure.

```
struct student
{
   struct person info;
   int rollno;
   float marks;
}
```

Here the first member is of type struct person. If we use this method of creating nested structures then you must first define the structures before creating variables of its types. So, it's mandatory for you to first define person structure before using it's variable as a member of the structure student.

The advantage of using this method is that now we can declare a variable of type `struct person` in anywhere else in the program.

Nesting of structure within itself is now allowed. For example:

```
struct citizen
{
   char name[50];
   char address[100];
   int age;
   int ssn;
```

```
    struct citizen relative; // invalid
}
```

Initializing nested Structures

Nested structures can be initialized at the time of declaration. For example:

```
struct person
{
    char name[20];
    int age;
    char dob[10];
};

struct student
{
    struct person info;
    int rollno;
    float marks[10];
}

struct student student_1 = {
                        {"Adam", 25, 1990},
                        101,
                        90
                    };
```

The following program demonstrates how we can use nested structures.

```
#include<stdio.h>
struct person
{
    char name[20];
    int age;
    char dob[10];
};

struct student
```

```
{
   struct person info;
   int roll_no;
   float marks;
};
int main()
{
   struct student s1;

   printf("Details of student: \n\n");

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

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

   printf("Enter dob: ");
   scanf("%s", s1.info.dob);

   printf("Enter roll no: ");
   scanf("%d", &s1.roll_no);

   printf("Enter marks: ");
   scanf("%f", &s1.marks);

   printf("\n*****************************\n\n");
   printf("Name: %s\n", s1.info.name);
   printf("Age: %d\n", s1.info.age);
   printf("DOB: %s\n", s1.info.dob);
   printf("Roll no: %d\n", s1.roll_no);
   printf("Marks: %.2f\n", s1.marks);

   // signal to operating system program ran fine
   return 0;
}
```

Expected Output:

```
Details of student:

Enter name: Phil
Enter age: 27
Enter dob: 23/4/1990
Enter roll no: 78123
Enter marks: 92

******************************

Name: Phil
Age: 27
DOB: 23/4/1990
Roll no: 78123
Marks: 92.00
```

How it works:

In lines 3-8, we have declared a structure called person.

In lines 10-15, we have declared another structure called student whose one of the members is of type struct student (declare above).

In line 19, we have declared a variable s1 of type struct student.

The next five scanf() statements (lines 23-36) asks the user to enter the details of the students which are then printed using the printf() (lines 40-44) statement.

12.6 COMPARING C STRUCTURES WITH PYTHON TUPLES

Tuples let us store several different named values inside a single variable, and a struct does much the same – so what's the difference, and when should you choose one over the other?

When you're just learning, the difference is simple: a tuple is effectively just a struct without a name, like an anonymous struct. This means you can define it as `(name: String, age: Int, city: String)` and it will do the same thing as the following struct:

```
struct User {
    var name: String
```

```
    var age: Int
    var city: String
}
```

However, tuples have a problem: while they are great for one-off use, particularly when you want to return several pieces of data from a single function, they can be annoying to use again and again.

Think about it: if you have several functions that work with user information, would you rather write this:

```
func authenticate(_ user: User) { ... }
func showProfile(for user: User) { ... }
func signOut(_ user: User) { ... }
```

Or this:

```
func authenticate(_ user: (name: String, age:
Int, city: String)) { ... }
func showProfile(for user: (name: String,
age: Int, city: String)) { ... }
func signOut(_ user: (name: String, age: Int,
city: String)) { ... }
```

Think about how hard it would be to add a new property to your `User` struct (very easy indeed), compared to how hard it would be to add another value to your tuple everywhere it was used? (Very hard, and error-prone!)

So, use tuples when you want to return two or more arbitrary pieces of values from a function, but prefer structs when you have some fixed data you want to send or receive multiple times.

12.7 SUMMARY

- In C programming, a struct (or structure) is a collection of variables (can be of different types) under a single name.
- Declaring an array of structure is same as declaring an array of fundamental types. Since an array is a collection of elements of the same type. In an array of structures, each element of an array is of the structure type.

- A structure can be nested inside another structure. In other words, the members of a structure can be of any other type including structure. Here is the syntax to create nested structures.
- Tuples let us store several different named values inside a single variable, and a struct does much the same

12.8 UNIT END QUESTIONS

1. How to declare variable of a structure?
2. Write a short note on Array of Structure.
3. How to define arrays within the structure?
4. Show implementation of Structures within structures.
5. Compare C structures with Python tuples.

12.9 REFERENCE FOR FURTHER READING

- https://beginnersbook.com/2014/01/c-structures-examples/
- https://overiq.com/c-programming-101/array-of-structures-in-c/
- https://staticallytyped.wordpress.com/2011/05/07/c-structs-vs-tuples-or-why-i-like-tuples-more/
- https://www.quora.com/Is-it-a-good-idea-to-replace-Tuple-with-Struct-in-C++
- https://kitchingroup.cheme.cmu.edu/blog/2013/02/27/Some-basic-data-structures-in-python/
- https://www.hackingwithswift.com/quick-start/understanding-swift/whats-the-difference-between-a-struct-and-a-tuple

*****

13

UNIONS

Unit Structure

13.0 OBJECTIVE

In this Chapter, you'll learn about unions in C programming. More specifically, how to create unions, access its members and learn the differences between unions and structures.

13.1 INTRODUCTION

A union is a user-defined type similar to structs in C except for one key difference.Structures allocate enough space to store all their members, whereas unions can only hold one member value at a time.

13.2 HOW TO DEFINE A UNION?

We use the union keyword to define unions. Here's an example:

```
union car
{
  char name[50];
  int price;
};
```

The above code defines a derived type union car.

Create union variables

When a union is defined, it creates a user-defined type. However, no memory is allocated. To allocate memory for a given union type and work with it, we need to create variables.

Here's how we create union variables.

```
union car
{
  char name[50];
  int price;
};

int main()
{
  union car car1, car2, *car3;
  return 0;
}
```

Another way of creating union variables is:

```
union car
{
  char name[50];
  int price;
} car1, car2, *car3;
```


In both cases, union variables *car1*, *car2*, and a union pointer *car3* of `union car` type are created.

Access members of a union

We use the `.` operator to access members of a union. And to access pointer variables, we use the `->` operator.

In the above example,

To access *price* for `car1`, `car1.price` is used.

To access *price* using `car3`, either `(*car3).price` or `car3->price` can be used.

Difference between unions and structures

Let's take an example to demonstrate the difference between unions and structures:

```
#include <stdio.h>
union unionJob
{
   //defining a union
   char name[32];
   float salary;
   int workerNo;
} uJob;

struct structJob
{
   char name[32];
   float salary;
   int workerNo;
} sJob;

int main()
{
   printf("size of union = %d bytes", sizeof(uJob));
   printf("\nsize of structure = %d bytes", sizeof(sJob));
   return 0;
}
```


Output:

```
size of union = 32
size of structure = 40
```

Why this difference in the size of union and structure variables?:

Here, the size of *sJob* is 40 bytes because

the size of `name[32]` is 32 bytes

the size of `salary` is 4 bytes

the size of `workerNo` is 4 bytes

However, the size of *uJob* is 32 bytes. It's because the size of a union variable will always be the size of its largest element. In the above example, the size of its largest element, (`name[32]`), is 32 bytes.

With a union, all members share **the same memory**.

Example: Accessing Union Members

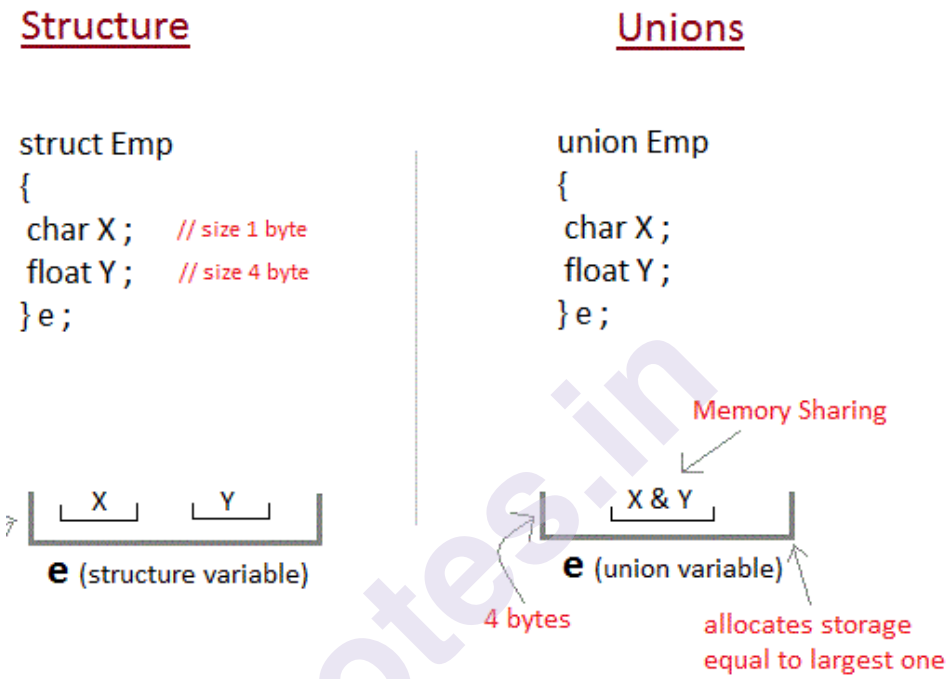
```
#include <stdio.h>

union Job {
   float salary;
   int workerNo;
} j;


int main() {
   j.salary = 12.3;

   // when j.workerNo is assigned a value,
   // j.salary will no longer hold 12.3
   j.workerNo = 100;

   printf("Salary = %.1f\n", j.salary);
   printf("Number of workers = %d", j.workerNo);
   return 0;
}
```


Output:

```
Salary = 0.0
Number of workers = 100
```

13.3 UNION VS STRUCTURE

Unions are conceptually similar to structures. The syntax to declare/define a union is also similar to that of a structure. The only differences is in terms of storage. In **structure** each member has its own storage location, whereas all members of **union** uses a single shared memory location which is equal to the size of its largest data member.



This implies that although a union may contain many members of different types, it cannot handle all the members at the same time. A union is declared using the union keyword.

```
union item
{
    int m;
    float x;
    char c;
}It1;
```

This declares a variable It1 of type union item. This union contains three members each with a different data type. However only one of them can be used at a time. This is due to the fact that only one location is allocated for all the union variables, irrespective of their size. The compiler allocates the storage that is large enough to hold the largest variable type in the union.

In the union declared above the member x requires 4 bytes which is largest amongst the members for a 16-bit machine. Other members of union will share the same memory address.

```
#include <stdio.h>
union item
{
   int a;
   float b;
   char ch;
};

int main( )
{
   union item it;
   it.a = 12;
   it.b = 20.2;
   it.ch = 'z';

   printf("%d\n", it.a);
   printf("%f\n", it.b);
   printf("%c\n", it.ch);

   return 0;
}
```

Output:

```
-26426
20.1999
z
```

As you can see here, the values of `a` and `b` get corrupted and only variable `c` prints the expected result. This is because in union, the memory is shared among different data types. Hence, the only member whose value is currently stored will have the memory.

In the above example, value of the variable `c` was stored at last, hence the value of other variables is lost.

C Structure	C Union
Structure allocates storage space for all its members separately.	Union allocates one common storage space for all its members. Union finds that which of its member needs high storage space over other members and allocates that much space
Structure occupies higher memory space.	Union occupies lower memory space over structure.

We can access all members of structure at a time.	We can access only one member of union at a time.

Structure example:
struct student
{
int mark;
char name[6];
double average;
};
	Union example:
	union student
	{
	int mark;
	char name[6];
	double average;
	};

For above structure, emory allocation will be like below. int mark – 2B char name[6] – 6B double average – 8B Total memory allocation = 2+6+8 = 16 Bytes	For above union, only 8 bytes of memory will be allocated since double data type will occupy maximum space of memory over other data types. Total memory allocation = 8 Bytes

13.4 USING POINTER VARIABLE

Union and structure in C are same in concepts, except allocating memory for their members.

Structure allocates storage space for all its members separately.

Whereas, Union allocates one common storage space for all its members

We can access only one member of union at a time. We can't access all member values at the same time in union. But, structure can access all member values at the same time. This is because, Union allocates one common storage space for all its members. Where as Structure allocates storage space for all its members separately.

Many union variables can be created in a program and memory will be allocated for each union variable separately.

Below table will help you how to form a C union, declare a union, initializing and accessing the members of the union.

Using normal variable	Using pointer variable

Syntax:
union tag_name
{
data type var_name1;
data type var_name2;
data type var_name3;
};
	Syntax:
	union tag_name
	{
	data type var_name1;
	data type var_name2;
	data type var_name3;
	};

Example:
union student
{
int mark;
char name[10];
float average;
};
	Example:
	union student
	{
	int mark;
	char name[10];
	float average;
	};

Declaring union using normal variable: union student report;	**Declaring union using pointer variable:** union student *report, rep;
Initializing union using normal variable: union student report = {100, “Mani”, 99.5};	**Initializing union using pointer variable:** union student rep = {100, “Mani”, 99.5}; report = &rep;
Accessing union members using normal variable: report.mark; report.name; report.average;	**Accessing union members using pointer variable:** report -> mark; report -> name; report -> average;

Example Program For C Union:

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

union student
{
        char name[20];
        char subject[20];
        float percentage;
};

int main()
{
   union student record1;
   union student record2;

   // assigning values to record1 union variable
      strcpy(record1.name, "Raju");
      strcpy(record1.subject, "Maths");
      record1.percentage = 86.50;

      printf("Union record1 values example\n");
      printf(" Name       : %s \n", record1.name);
      printf(" Subject    : %s \n", record1.subject);
      printf(" Percentage : %f \n\n", record1.percentage);

   // assigning values to record2 union variable
      printf("Union record2 values example\n");
      strcpy(record2.name, "Mani");
      printf(" Name       : %s \n", record2.name);
```

```
    strcpy(record2.subject, "Physics");
    printf(" Subject    : %s \n", record2.subject);

    record2.percentage = 99.50;
    printf(" Percentage : %f \n", record2.percentage);
    return 0;
}
```

Output:

```
Union record1 values example
Name :
Subject :
Percentage : 86.500000;
Union record2 values example
Name : Mani
Subject : Physics
Percentage : 99.500000
```

Explanation For Above C Union Program:

There are 2 union variables declared in this program to understand the difference in accessing values of union members.

Record1 union variable:

"Raju" is assigned to union member "record1.name" . The memory location name is "record1.name" and the value stored in this location is "Raju".

Then, "Maths" is assigned to union member "record1.subject". Now, memory location name is changed to "record1.subject" with the value "Maths" (Union can hold only one member at a time).

Then, "86.50" is assigned to union member "record1.percentage". Now, memory location name is changed to "record1.percentage" with value "86.50".

Like this, name and value of union member is replaced every time on the common storage space.

So, we can always access only one union member for which value is assigned at last. We can't access other member values.

So, only "record1.percentage" value is displayed in output. "record1.name" and "record1.percentage" are empty.

Record2 union variable:

If we want to access all member values using union, we have to access the member before assigning values to other members as shown in record2 union variable in this program.

Each union members are accessed in record2 example immediately after assigning values to them.

If we don't access them before assigning values to other member, member name and value will be over written by other member as all members are using same memory.

We can't access all members in union at same time but structure can do that.

Example Program – Another Way Of Declaring C Union:

In this program, union variable "record" is declared while declaring union itself as shown in the below program.

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

union student
{
        char name[20];
        char subject[20];
        float percentage;
}record;

int main()
{

        strcpy(record.name, "Raju");
        strcpy(record.subject, "Maths");
        record.percentage = 86.50;

        printf(" Name       : %s \n", record.name);
        printf(" Subject    : %s \n", record.subject);
        printf(" Percentage : %f \n", record.percentage);
        return 0;
}
```

Output:

```
Name :
Subject :
Percentage : 86.500000
```

Note:

We can access only one member of union at a time. We can't access all member values at the same time in union.

But, structure can access all member values at the same time. This is because, Union allocates one common storage space for all its members. Where as Structure allocates storage space for all its members separately.

More Examples of Union:

```
#include <stdio.h>
union student
{
   char name[50];
   int id;
   char address[50];
};

int main( )
{
   union student stu;
    printf("\nEnter the name of the student: ");
   scanf("%s", &stu.name);
   printf("Enter the id of student: ");
   scanf("%ld", &stu.id);
   printf("Enter the address of the student: ");
   scanf("%s", &stu.address);
   printf("The name of the student entered is %s\n", stu.name);
   printf("The id of the student entered is %d\n", stu.id);
   printf("The address of the student entered is %s\n", stu.address);

   return 0;
}
```

Output:

The above code will display its output as below:

```
Enter the name of the student: jack
Enter the id of student: 3
Enter the address of the student: Boston
The name of the student entered is Boston
The id of the student entered is 1953722210
The address of the student entered is Boston
```

In the above code, we see that the first and the second variable in the union prints the garbage value and only the third variable prints the true value. As in union, different data types will share the same memory. For this reason, the only variables whose value is currently stored will have the memory.

Now let's look at the same example again. But this time we will print one variable at a time which is the main purpose of having unions.

```
#include <stdio.h>

union student
{
   char name[50];
   int id;
   char address[50];
};

int main( )
{
   union student stu;
   printf("\nEnter the name of the student: ");
   scanf("%s", &stu.name);
   printf("The name of the student entered is %s\n", stu.name);
   printf("Enter the id of student: ");
   scanf("%ld", &stu.id);
   printf("The id of the student entered is %d\n", stu.id);
   printf("Enter the address of the student: ");
   scanf("%s", &stu.address);
   printf("The address of the student entered is %s\n", stu.address);
   return 0;
}
```

The above code will generate its output as below:

Enter the name of the student: jack

The name of the student entered is jack

Enter the id of student: 3

The id of the student entered is 3

Enter the address of the student: Boston

The address of the student entered is Boston

Here the output infers that all the members of the union are printed well. As or union will use one variable at a time.

13.5 SUMMARY

- A union is a user-defined type similar to structs in C except for one key difference.Structures allocate enough space to store all their members, whereas unions can only hold one member value at a time.
- When a union is defined, it creates a user-defined type. However, no memory is allocated. To allocate memory for a given union type and work with it, we need to create variables.
- Unions are conceptually similar to structures. The syntax to declare/define a union is also similar to that of a structure.
- Union and structure in C are same in concepts, except allocating memory for their members.
- We can access only one member of union at a time. We can't access all member values at the same time in union.

13.6 UNIT END QUESTIONS

1. How to define a union?
2. Why this difference in the size of union and structure variables?
3. Unions are conceptually similar to structures. Explain.
4. Differentiate between Union And Structure.
5. Explain Union Using normal variable VS Union Using pointer variable

13.7 REFERENCE FOR FURTHER READING

- https://fresh2refresh.com/c-programming/c-union/
- http://tutorialtous.com/c/blockread.php

- https://www.w3schools.in/c-tutorial/
- https://www.javatpoint.com/
- https://fresh2refresh.com/c-programming/
- https://www.educba.com/c-union/
- https://followtutorials.com/2019/04/c-programming-union.html

*****

14

FILE HANDLING

Unit Structure

14.0 OBJECTIVE

File is main data storage in any computer system. Here we will discuss various file operation and function which are responsible for doing all operations on files.

14.1 INTRODUCTION

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.

14.2 FILE OPERATIONS

Five major operations can be performed on file are:

1. Creation of a new file.
2. Opening an existing file.
3. Reading data from a file.
4. Writing data in a file.
5. Closing a file.

14.2.1 Streams:

A Stream refers to the characters read or written to a program. The streams are designed to allow the user to access the files efficiently .A stream is a file or physical device like keyboard, printer and monitor.

The FILE object contains all information about stream like current position, pointer to any buffer, error and EOF(end of file).

14.3 TYPES OF FILES

There are 2 kinds of files in which data can be stored in 2 ways either in characters coded in their ASCII character set or in binary format. They are

1. Text Files.
2. Binary Files

14.3.1 Text Files:

A Text file contains only the text information like alphabets ,digits and special symbols. The ASCII code of these characters are stored in these files.It uses only 7 bits allowing 8 bit to be zero.

1. w(write):

This mode opens new file on the disk for writing.If the file exist,disk for writing.If the file exist, then it will be over written without then it will be over written without any confirmation.

Syntax:

```
fp=fopen("data.txt","w");
"data.txt" is  filename
"w" is writemode.
```

2. r(read):

This mode opens an preexisting file for reading.If the file doesn't Exist then the compiler returns a NULL to the file pointer

Syntax:

```
fp=fopen("data.txt","r");
```

3. w+(read and write)

This mode searches for a file if it is found contents are destroyed If the file doesn't found a new file is created.

Syntax:

```
fp=fopen("data.txt","w+");
```

4. a(append)

This mode opens a preexisting file for appending the data.

Syntax:

```
fp=fopen("data.txt","a");
```

5. a+(append+read):

the end of the file.

Syntax:

```
fp=fopen("data.txt","a+");
```

6. r+(read +write)

This mode is used for both Reading and writing

14.3.2 Binary Files:

A binary file is a file that uses all 8 bits of a byte for storing the information .It is the form which can be interpreted and understood by the computer.

The only difference between the text file and binary file is the data contain in text file can be recognized by the word processor while binary file data can't be recognized by a word processor.

1. wb(write):

this opens a binary file in write mode.

Syntax:

```
fp=fopen("data.dat","wb");
```

2. rb(read):

this opens a binary file in read mode

Syntax:

```
fp=fopen("data.dat","rb");
```

3. ab(append):

this opens a binary file in a Append mode i.e. data can be added at the end of file.

Syntax:

```
fp=fopen("data.dat","ab");
```

4. r+b(read+write):

this mode opens preexisting File in read and write mode.

Syntax:

```
fp=fopen("data.dat","r+b");
```

5. w+b(write+read):

this mode creates a new file for reading and writing in Binary mode.

Syntax:

```
fp=fopen("data.dat","w+b");
```

6. a+b(append+write):

this mode opens a file in append mode i.e. data can be written at the end of file.

Syntax:

```
fp=fopen("data.dat","a+b");
```


14.4 DIFFERENT TYPES OF FUNCTIONS

1. 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") ;
while ( 1 )
{
```

```
ch = fgetc ( fp ) ;
if ( ch == EOF )
break ;
printf("%c",ch) ;
}
fclose (fp ) ;
}
```

Output:

The content of the file will be printed.

C fopen function returns NULL in case of a failure and returns a FILE stream pointer on success.

Example:

```
#include<stdio.h>

int main()
{
   FILE *fp;
   fp = fopen("fileName.txt","w");
   return 0;
}
```

The above example will create a file called fileName.txt.

The w means that the file is being opened for writing, and if the file does not exist then the new file will be created.

2. 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 fclose returns EOF in case of failure and returns 0 on success.

```
#include<stdio.h>
int main()
{
   FILE *fp;
   fp = fopen("fileName.txt","w");
   fprintf(fp, "%s", "Sample Texts");
```

```
   fclose(fp);
   return 0;
}
```

- The above example will create a file called fileName.txt.
- The w means that the file is being opened for writing, and if the file does not exist then the new file will be created.
- The fprintf function writes Sample Texts text to the file.
- The fclose function closes the file and releases the memory stream.

Example:

```
#include <stdio.h>
int main () {
  FILE *fp;
  fp = fopen("file.txt", "w");
  fprintf(fp, "%s", "Hello World!!!");
  fclose(fp);
  return(0);
}
```

Let us compile and run the above program that will create a file file.txt, and then it will write following text line and finally it will close the file using fclose() function.

```
Hello World!!!
```

3. fgetc():

The fgetc() function returns a single character from the file. It gets a character from the stream. It returns EOF at the end of file.

- getc() function returns the next requested object from the stream on success.
- Character values are returned as an unsigned char cast to an int or EOF on the end of the file or error.
- The function feof() and ferror() to distinguish between end-of-file and error must be used.

Syntax:

```
int fgetc(FILE *stream)
```

Example:

```
#include<stdio.h>
#include<conio.h>
void main(){
FILE *fp;
char c;
clrscr();
fp=fopen("myfile.txt","r");
while((c=fgetc(fp))!=EOF){
printf("%c",c);
}
fclose(fp);
getch();
}
```

myfile.txt

```
this is simple text message
```

Example:

```
#include<stdio.h>
int main()
{
  FILE *fp = fopen("fileName.txt", "r");
  int ch = getc(fp);
  while (ch != EOF)
  {
    //To display the contents of the file on the screen
    putchar(ch);
    ch = getc(fp);
  }
  if (feof(fp))
     printf("\n Reached the end of file.");
  else
     printf("\n Something gone wrong.");
  fclose(fp);

  getchar();
  return 0;
}
```

4. fputc():

The fprintf() function is used to write set of characters into file. It sends formatted output to a stream.

Syntax:

```
int fprintf(FILE *stream, const char *format [, argument, ...])
```

Example:

```
#include <stdio.h>
main(){
  FILE *fp;
  fp = fopen("file.txt", "w");//opening file
  fprintf(fp, "Hello file by fprintf...\n");//writing data into file
  fclose(fp);//closing file
}
```

Example:

```
int main (void)
{FILE * fileName;
   char ch;
   fileName = fopen("anything.txt","wt");
      for (ch = 'D' ; ch <= 'S' ; ch++) {
         putc (ch , fileName);}
   fclose (fileName);
   return 0;}
```

5. fgets():

The fgets() function reads a line of characters from file. It gets string from a stream.

Syntax:

```
char* fgets(char *s, int n, FILE *stream)
```

Parameters:

- **s:** This is the pointer to an array of chars where the string read is stored.
- **n:** This is the maximum number of characters to be read (including the final null-character). Usually, the length of the array passed as str is used.

- **Stream:** This is the pointer to a FILE object that identifies the stream where characters are read from.

Example:

```
#include<stdio.h>
#include<conio.h>
void main(){
FILE *fp;
char text[300];
clrscr();
fp=fopen("myfile2.txt","r");
printf("%s",fgets(text,200,fp));
fclose(fp);
getch();
}
```

Output:

```
hello c programming
```

```
#include <stdio.h>
int main () {
  FILE *fp;
  char str[60];
  /* opening file for reading */
  fp = fopen("file.txt" , "r");
  if(fp == NULL) {
    perror("Error opening file");
    return(-1);
  }
  if( fgets (str, 60, fp)!=NULL ) {
    /* writing content to stdout */
    puts(str);
  }
  fclose(fp);
  return(0);
}
```

Let us assume, we have a text file file.txt, which has the following content. This file will be used as an input for our example program:

```
We are in 2021
```

Now, let us compile and run the above program that will produce the following result:

```
We are in 2021
```

6. fputs():

The fputs() function writes a line of characters into file. It outputs string to a stream.

Syntax:

```
int fputs(const char *s, FILE *stream)
```

Parameters:

- **s:** This is an array containing the null-terminated sequence of characters to be written.
- **Stream:** This is the pointer to a FILE object that identifies the stream where the string is to be written.

Example:

```
#include<stdio.h>
#include<conio.h>
void main(){
FILE *fp;
clrscr();
fp=fopen("myfile2.txt","w");
fputs("hello c programming",fp);
fclose(fp);
getch();
}
```

myfile2.txt

```
hello c programming
```

Example

The following example shows the usage of fputs() function.

```
#include <stdio.h>
int main () {
   FILE *fp;
   fp = fopen("file.txt", "w+");
   fputs("This is c programming.", fp);
   fputs("This is a system programming language.", fp);
```

```
  fclose(fp);
  return(0);
}
```

Let us compile and run the above program, this will create a file file.txt with the following content:

```
This is c programming.This is a system programming language.
```

Now let's see the content of the above file using the following program:

```
#include <stdio.h>
int main () {
  FILE *fp;
  int c;

  fp = fopen("file.txt","r");
  while(1) {
    c = fgetc(fp);
    if( feof(fp) ) {
      break ;
    }
    printf("%c", c);
  }
  fclose(fp);
  return(0);
}
```


Let us compile and run the above program to produce the following result.

```
This is c programming.This is a system programming language.
```

7. fscanf():

The fscanf() function is used to read set of characters from file. It reads a word from the file and returns EOF at the end of file.

Syntax:

```
int fscanf(FILE *stream, const char *format [, argument, ...])
```

Example:

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

```
  FILE *fp;
  char buff[255];//creating char array to store data of file
  fp = fopen("file.txt", "r");
  while(fscanf(fp, "%s", buff)!=EOF){
  printf("%s ", buff );
  }
  fclose(fp);
}
```

Output:

```
Hello file by fprintf...
```

Example:

```
int main()
{
   char str1[10], str2[10];
   int yr;
   FILE* fileName;
   fileName = fopen("anything.txt", "w+");
   fputs("Welcome to", fileName);
   rewind(fileName);
   fscanf(fileName, "%s %s %d", str1, str2, &yr);
   printf("---------------------------------------- \n");
   printf("1st word %s \t", str1);
   printf("2nd word  %s \t", str2);
   printf("Year-Name  %d \t", yr);
   fclose(fileName);
   return (0);
}
```


8. fprintf():

The fprintf() function is used to write set of characters into file. It sends formatted output to a stream.

Syntax:

```
int fprintf(FILE *stream, const char *format [, argument, ...])
```

Example:

```
#include <stdio.h>
main(){
  FILE *fp;
```

```
  fp = fopen("file.txt", "w");//opening file
  fprintf(fp, "Hello file by fprintf...\n");//writing data into file
  fclose(fp);//closing file
}
```

Example:

```
int main (void)
{
  FILE *fileName;
  fileName = fopen("anything.txt","r");
  fprintf(fileName, "%s %s %d", "Welcome", "to",  2018);
  fclose(fileName);
  return(0);
}
```

9. getw():

getw( ) function is used for reading a number from a file.

The syntax for getw() function is as follows −

Syntax

```
int getw (FILE *fp);
```

For example,

Example:

```
FILE *fp;
int num;
num = getw(fp);
```

```
fp =fopen ("num.txt", "r");
printf ("file content is\n");
for (i =1; i<= 10; i++){
   i= getw(fp);
   printf ("%d",i);
   printf("\n");
}
fclose (fp);
```

8. putw():

Declaration: int putw(int number, FILE *fp);

putw function is used to write an integer into a file. In a C program, we can write integer value in a file as below.

```
putw(i, fp);
```

where

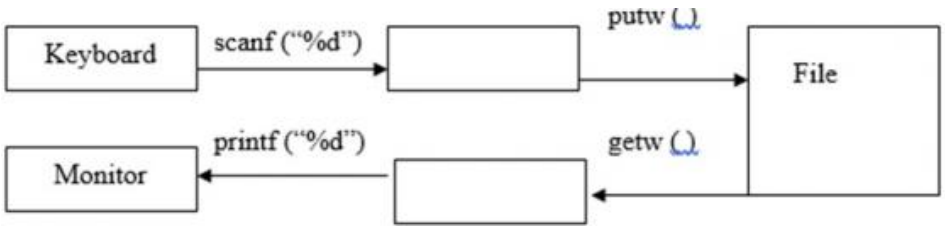
i – integer value

fp – file pointer

```
#include <stdio.h>
int main ()
{
  FILE *fp;
  int i=1, j=2, k=3, num;
  fp = fopen ("test.c","w");
  putw(i,fp);
  putw(j,fp);
  putw(k,fp);
  fclose(fp);
  fp = fopen ("test.c","r");
  while(getw(fp)!=EOF)
  {
    num= getw(fp);
    printf("Data in test.c file is %d \n", num);
  }
  fclose(fp);
  return 0;
}
```

Output:

```
Data in test.c file is
1
2
3
```



Program:

Following is the C program for storing the numbers from 1 to 10 and to print the same:

```
#include<stdio.h>
int main( ){
  FILE *fp;
  int i;
  fp = fopen ("num.txt", "w");
  for (i =1; i<= 10; i++){
    putw (i, fp);
  }
  fclose (fp);
  fp =fopen ("num.txt", "r");
  printf ("file content is\n");
  for (i =1; i<= 10; i++){
    i= getw(fp);
    printf ("%d",i);
    printf("\n");
  }
  fclose (fp);
  return 0;
}
```


Output:

When the above program is executed, it produces the following result:

```
file content is
1
2
3
4
5
6
7
```

```
8
9
10
```

9. fread():

The C library function size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream) reads data from the given stream into the array pointed to, by ptr.

Declaration:

Following is the declaration for fread() function.

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)
```

Parameters:

ptr − This is the pointer to a block of memory with a minimum size of size*nmemb bytes.

size − This is the size in bytes of each element to be read.

nmemb − This is the number of elements, each one with a size of size bytes.

stream − This is the pointer to a FILE object that specifies an input stream.

Example:

The following example shows the usage of fread() function.

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

int main () {
   FILE *fp;
   char c[] = "Hello World!!!";
   char buffer[100];

   /* Open file for both reading and writing */
   fp = fopen("file.txt", "w+");

   /* Write data to the file */
   fwrite(c, strlen(c) + 1, 1, fp);
   /* Seek to the beginning of the file */
```

```
   fseek(fp, 0, SEEK_SET);
   /* Read and display data */
   fread(buffer, strlen(c)+1, 1, fp);
   printf("%s\n", buffer);
   fclose(fp);
   return(0);
}
```

Let us compile and run the above program that will create a file file.txt and write a content this is tutorialspoint. After that, we use fseek() function to reset writing pointer to the beginning of the file and prepare the file content which is as follows –

```
Hello World!!!
```

10. fwrite():

The C library function size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream) writes data from the array pointed to, by ptr to the given stream.

Declaration:

Following is the declaration for fwrite() function.

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)
```

Parameters:

ptr − This is the pointer to the array of elements to be written.

size − This is the size in bytes of each element to be written.

nmemb − This is the number of elements, each one with a size of size bytes.

stream − This is the pointer to a FILE object that specifies an output stream.

Example:

The following example shows the usage of fwrite() function.

```
#include<stdio.h>
int main () {
   FILE *fp;
   char str[] = "This is IDOL";
```

```
   fp = fopen( "file.txt" , "w" );
   fwrite(str , 1 , sizeof(str) , fp );
   fclose(fp);
   return(0);
}
```

Let us compile and run the above program that will create a file file.txt which will have following content –

```
This is IDOL
```

Now let's see the content of the above file using the following program –

```
#include <stdio.h>
int main () {
   FILE *fp;
   int c;
   fp = fopen("file.txt","r");
   while(1) {
      c = fgetc(fp);
      if( feof(fp) ) {
         break ;
      }
      printf("%c", c);
   }
   fclose(fp);
   return(0);
}
```

Let us compile and run the above program to produce the following result:

```
This is IDOL
```

11. fseek():

The C library function int fseek(FILE *stream, long int offset, int whence) sets the file position of the stream to the given offset.

Declaration

Following is the declaration for fseek() function.

```
int fseek(FILE *stream, long int offset, int whence)
```

Parameters:

Stream: This is the pointer to a FILE object that identifies the stream.

Offset: This is the number of bytes to offset from whence.

Whence: This is the position from where offset is added. It is specified by one of the following constants –

Sr.No.	Constant & Description
1	**SEEK_SET** Beginning of file
2	**SEEK_CUR** Current position of the file pointer
3	**SEEK_END** End of file

Example:

The following example shows the usage of fseek() function.

```
#include <stdio.h>

int main () {
   FILE *fp
   fp = fopen("file.txt","w+");
   fputs("This is IDOL", fp);
   fseek( fp, 7, SEEK_SET );
   fputs(" C Programming Language", fp);
   fclose(fp);
   return(0);
}
```


Let us compile and run the above program that will create a file file.txt with the following content. Initially program creates the file and writes This is IDOL but later we had reset the write pointer at 7th position from the beginning and used puts() statement which over-write the file with the following content –

```
This is C Programming Language
```

Now let's see the content of the above file using the following program –

```
#include <stdio.h>
int main () {
   FILE *fp;
   int c;
```

```
  fp = fopen("file.txt","r");
  while(1) {
     c = fgetc(fp);
     if( feof(fp) ) {
        break;  }
     printf("%c", c);
  }
  fclose(fp);
  return(0);
}
```

Let us compile and run the above program to produce the following result:

```
This is C Programming Language
```

14.5 SUMMARY

- In programming, we may require some specific input data to be generated several numbers of times.
- Five major operations can be performed on file are:
  - ✓ Creation of a new file.
  - ✓ Opening an existing file.
  - ✓ Reading data from a file.
  - ✓ Writing data in a file.
  - ✓ Closing a file.
- A Stream refers to the characters read or written to a program. The streams are designed to allow the user to access the files efficiently
- There are 2 kinds of files in which data can be stored in 2 ways either in characters coded in their ASCII character set or in binary format. They are
  - ✓ Text Files.
  - ✓ Binary Files
- A Text file contains only the text information like alphabets ,digits and special symbols.
- A binary file is a file that uses all 8 bits of a byte for storing the information .It is the form which can be interpreted and understood by the computer.

14.6 UNIT END QUESTIONS

1. What are operations on file? Explain in detail.
2. What are types of file?
3. Explain following functions with examples.

   fopen(), fclose(), fgetc(), fputc()
4. Explain following functions with examples.

   fgets(), fputs(), fscanf(), fprintf()
5. Explain following functions with examples

   getw(), putw(), fread(), fwrite(), fseek().

14.7 REFERENCE FOR FURTHER READING

- http://tutorialtous.com/c/blockread.php
- https://www.w3schools.in/c-tutorial/file-handling/fclose/
- https://www.javatpoint.com/file-handling-in-c
- https://fresh2refresh.com/c-programming/c-file-handling/getw-putw-functions-c/

*****