DETAILED CONTENTS OF PIC

1. Algorithm and Programming Development (06 Periods)

- 1.1 Steps in development of a program
- 1.2 Flow charts, Algorithm development
- 1.3 Programme Debugging

2. Program Structure (10 Periods)

- 2.1 I/O statements, assign statements
- 2.2 Constants, variables and data types
- 2.3 Operators and Expressions
- 2.4 Unformatted and Formatted IOS
- 2.5 Data Type Casting

3. Control Structures (10 Periods)

- 3.1 Introduction
- 3.2 Decision making with IF – statement
- 3.3 IF – Else and Nested IF
- 3.4 While and do-while, for loop
- 3.5 Break. Continue, goto and switch statements

4 Pointers (08 Periods)

- 4.1. Introduction to pointers
- 4.2 Address operator and pointers
- 4.3 Declaring and initializing pointers,
- 4.4 Single pointer

5. Functions (12 Periods)

- 5.1 Introduction to functions
- 5.2 Global and Local Variables
- 5.3 Function Declaration
- 5.4 Standard functions
- 5.5 Parameters and Parameter Passing
- 5.6 Call - by value/reference

6. Arrays and Strings (08 Periods)

- 6.1. Introduction to Arrays
- 6.2. Array Declaration, Length of array

6.3 Single and Multidimensional Array.
6.4 Arrays of characters
6.5 Introduction of Strings
6.6 String declaration and definition
6.7 String Related function i.e. strlen, strcpy, strcmp
6.8 Passing an array to function
6.9 Pointers to an array and strings.

7. Structures and Unions (10 Periods)

7.1 Declaration of structures
7.2 Accessing structure members
7.3 Structure Initialization
7.4 Pointer to a structures,
7.5 Unions

LIST OF PRACTICALS

1. Programming exercises on executing and editing a C program.
2. Programming exercises on defining variables and assigning values to variables.
3. Programming exercises on arithmetic and relational operators.
4. Programming exercises on arithmetic expressions and their evaluation.
5. Programming exercises on formatting input/output using printf and scanf and their return type values.
6. Programming exercises using if statement.
7. Programming exercises using if – Else.
8. Programming exercises on switch statement.
9. Programming exercises on do – while, statement.
10. Programming exercises on for – statement.
11. Simple programs using pointers.
12. Programs on one-dimensional array.
13. Programs on two-dimensional array.
14. (i) Programs for putting two strings together.
    (ii) Programs for comparing two strings.
15. Simple programs using functions
16. Simple programs using structures.
17. Simple programs using union.

SUGGESTED DISTRIBUTION OF MARKS

Topic No.	Time Allotted (Periods)	Marks Allotted (%)
1	06	10
2	10	16
3	10	16
4	08	12
5	12	20
6	08	12
7	10	14
Total	**64**	**100**

Unit-1

Algorithm and Programming Development

INTRODUCTION:-

C is a structured programming language developed by Dennis Ritchie in 1973 at Bell Laboratories. It is one of the most popular computer languages today because of its structure, high-level abstraction, machine independent feature etc. C language was developed to write the UNIX operating system, hence it is strongly associated with UNIX, which is one of the most popular network operating system in use today and heart of internet data superhighway.

C language is a very good language to introduce yourself to the programming world, as it is a simple procedural language which is capable of doing wonders.

Programs written in C language takes very less time to execute and almost executes at the speed of assembly language instructions.

Initially C language was mainly used for writing system level programs, like designing operating systems, but there are other applications as well which can be very well designed and developed using C language, like Text Editors, Compilers, Network Drivers etc.

Programs :-

A computer program is a set of instructions for computer to perform specified tasks.

1. programmes are set of instructions for computer to perform a specified task
2. programmes are developed by individuals
3. programmes are smaller in size
4. programmes can be developed according to programmer's individual style of development
5. In programmes, user interface are not important, because the programmer is the sole user.
6. In programmes, very little documentation is required.

CHARACTERESTICS OF GOOD PROGRAM

A software product can be judged by what it offers and how well it can be used. This program must satisfy on the following grounds:

- Operational
- Transitional
- Maintenance

Well-engineered and crafted program is expected to have the following characteristics

Operational

This tells us how well software works in operations. It can be measured on:

- Budget
- Usability
- Efficiency
- Correctness
- Functionality
- Dependability
- Security
- Safety

Transitional

This aspect is important when the software is moved from one platform to another:

- Portability
- Interoperability
- Reusability
- Adaptability

Maintenance

This aspect briefs about how well software has the capabilities to maintain itself in the ever- changing

environment:

- Modularity
- Maintainability
- Flexibility
- Scalability

In short, Software engineering is a branch of computer science, which uses well-defined engineering concepts required to produce efficient, durable, scalable, in-budget and on-time software products

Emergence of programming

Starting from early years of computer to present times,there have been significant changes in characteristics of problems,issues and computing practices.

High level programming:-

Computers become fasterwith introduction of the semi conductor technology in early 1960.faster semi conductor transistors replaced the prevalent vacuum tubes based circuit in a computer.at this time, a high level language were introduced.by using a high level language it is possible to solve a large a complex problem writing each high level programming construct enables the programmer to write several machine instructions. The high level language that are more powerful, easier to use and directed towards

specialized classes of problems. A list of some high level language and their feature is given below:

Fortan:-it was developed in 1954 to facilitate the writing of scientific, mathematical and engineering software. Its great strength lies in its facilities for mathematical computations.

Cobol:- it was introduced in early 1960's and it still being used for business applications.it was designed to process large data files with alphanumeric characters which are characteristics of business problems.it can read, write and manipulate records very effectively.

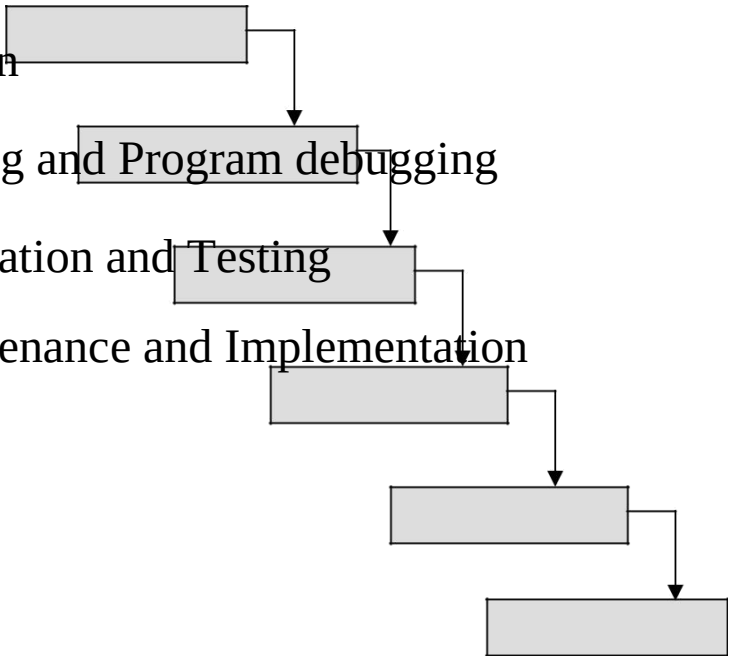
Pl/I: -it was created by IBM in 1964 as a general purpose programming language to support both business and scientific problem solving.it is very powerful but not widely used.

C:- the language C was developed in AT and T bell laboratory in early 1970's it is language in which most of the Unix operating is written. It is easier to learn and is portable. It is starting to be used for business, scientific and technical applications on large computer.

Steps in development of a program:-

The process of developing efficient and correct programs is so huge and lengthy that it has to be divided into steps(phases) known as basics steps for developing a program. These phases of development are so interlinked,interrelated and ordered that the outcome of first step becomes the input for next step. These are as the following phases :

- Feasibility Study
- Requirements Analysis and Specification
- Design
- Coding and Program debugging
- Integration and Testing
- Maintenance and Implementation



Feasibility study:-

- Feasibility Study is the first step/phase of Program development and in this phase financial feasibility or aspect of the program is seen or calculated using a process by which an agent creates a specification of a software artifact, intended to an accomplish goals, using a set of primitive components and subject to constraints.

Software requirement Specification:-

This is the second phase of program development life cycle. After calculating financial aspects of the program ,the programmer is keen to know the detailed specification regarding the requirements of the program and the user has to give the detailed software requirement specification(SRS) to the programmer .so,SRS is the output of this phase, hence a good SRS complete in all aspects is anticipated by the programmer from the user.

Program designing

Program design is a process by which an agent creates a specification of a software artifact, intended to an accomplish goals, using a set of primitive components and subject to constraints. Program design is the process of implementing software solutions to one or more sets of problems.one of the

important parts of program design is software requirement analysis. It is the part of the software development process that lists specifications. Program design may be platform independent or platform specific depending on availability of the technology used in designing.

Program design processes have two levels.

1. **SystemDesign (External design):-**Its is focus on deciding which modules are needed, specification of these module their central relationships and definition of interface among them. This design involves planning and specifying the external behavior of the software product(program).

2. Internal design (detail design):-The focus is on planning and specifying the internal structure and processing detail. Detail design basically expands the system design to include more detail of processing logic and data structure so that design is complete for coding phase.

Here the output of this phase is the **software requirement document** designed in an efficient way.

Coding and debugging:- Coding is the technique of writing the solution of the program designed efficiently in **software requirement document** according to the grammatical rules and syntax of any computer programming language.Writing programs using C language is also known

as programming in C. The output of this phase is the Program.

After creating a program in C, it has to be checked for the errors.Debugging is the process of finding and removing the errors(bugs) from the program. Error is the mistake made by the programmer which violates the grammatical rules of the computer language. Errors can be of three types:

1. Syntax errors or Compile Time Error
2. Run time errors
3. Logical errors.

Program debugging is concerned with the finding and removal of the syntax errors only. After the removal of all syntax errors, our source program is converted into object code by the C Compiler.

Integeration and Testing:- It is a systematic approach to develop error free programs by applying a set of techniques and guidelines . After developing error free program, it has to be tested by giving valid inputs and obtaining required valid output from the program. If the program is developed in various modules, then all these modules are tested and integrated in this phase.

Maintenance and Implementation:- It is a systematic approach to develop error free programs by applying a set of

techniques and guidelines and to maintain the program as per user's dynamic requirement and to implement the program in efficient manner. This is the last step of the program development life cycle.

Design methodology:-

A systematic approach to creating a design by applying a set of technique and guidelines is called design methodology. The focus of most of methodologies of design is on the program design. The purpose of these design mythologies is not to reduced the sequence steps of design activities but to guide the developer of the design such that the design specifications like completeness, consistency, unambiguityetc. can be achieved**.**

There are basically three approaches to software design.

(i) Control flow based design.
(ii) Instruction based design(Algorithms).
(iii) Modular design(Bottom-up and Top-down, structured).

Control flow based design:

With the advent of powerful machine and high level language, the usage of computer grew rapidly. In addition, the nature of programalso changed from simple to complex.

The increased size and complexity could not be managed by the individualstyle. It was analyzed the clarity of control flow (the sequence in which the program's instruction are executed) is a great importance. To help the programmer todesign a program having good control flow structure, flowcharting is developed. In flowcharting technique, the algorithm is represented using flowchart. A flow chart is graphical representation that arrange the sequence of operations to be carried out to solve the given problem.

To provide clarity of control flow chart the use of GOTO construct in flow chart should be avoided.

Definition of flow chart:-

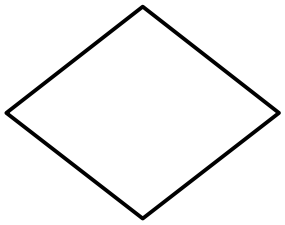
A flowchart is the graphical or pictorial representation of an algorithm with the help of different symbols, shapes and arrows in order to demonstrate a process or a program. With algorithms, we can easily understand a program. The main purpose of a flowchart is to analyze different processes. Several standard graphics are applied in a flowchart:

- Terminal Box - Start / End

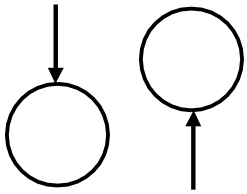
- Input / Output box

- Process / Instruction

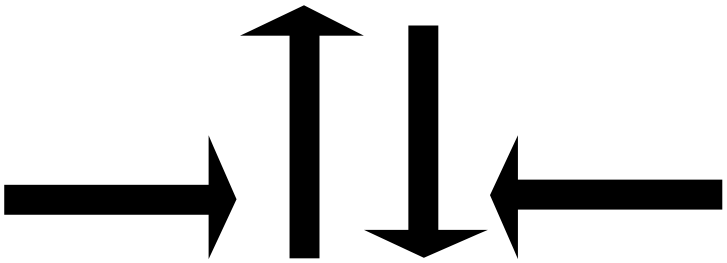
- Decision



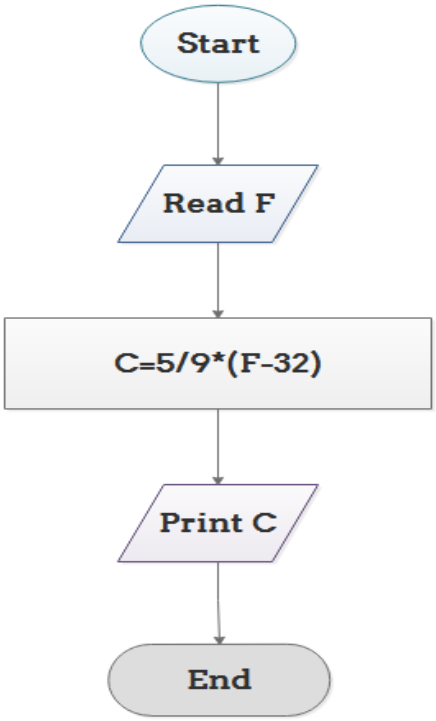
- Connector



- Arrow



The graphics above represent different part of a flowchart. The process in a flowchart can be expressed through boxes and arrows with different sizes and colors. In a flowchart, we can easily highlight a certain element and the relationships between



eachpart.**Example 1:Convert Temperat ure from Fahrenheit (℉) to Celsius(℃)**

Advantages of flow chart:-

Better Communication

Proper program documentation

Efficient coding

Systematic debugging

Systematic testing

Limitations of flow chart:-

Flowcharts are very time consuming and laborious to draw (especially for large complex programs)

Redrawing a flowchart for incorporating changes/ modifications is a tedious task

There are no standards determining the amount of detail that should be included in a flowchart

Definition of Algorithm:-

To write a logical step-by-step method to solve the problem is called algorithm, in other words, an algorithm is a procedure for solving problems. In order to solve a mathematical or computer problem, this is the first step of the procedure. An algorithm includes calculations, reasoning and data processing. Algorithms can be presented by natural languages, pseudo code and flowcharts, etc.

Example 1: Print 1 to 20:

Algorithm:

Step 1: Start/Begin

Step 2: Initialize X as 0,

Step 3: Increment X by 1,

Step 4: Print X,

Step 5: If X is less than 20 then go back to step 2.

Step 6: Stop/End

Example 2: Convert Temperature from Fahrenheit (°F) to Celsius (°C)

Algorithm:

Step 1: Start/Begin

Step 2: Read temperature in Fahrenheit,

Step 3: Calculate temperature with formula C=5/9*(F-32),

Step 4: Print C

Step 5: Stop/End

Program Debugging:-

After creating a program in C, it has to be checked for the errors. Debugging is the process of finding and removing the errors (bugs) from the program. Error is the mistake made by the programmer which violates the grammatical rules of the computer language.

Errors can be of three types:

1. Syntax errors or Compile Time Error
2. Run time errors
3. Logical errors.

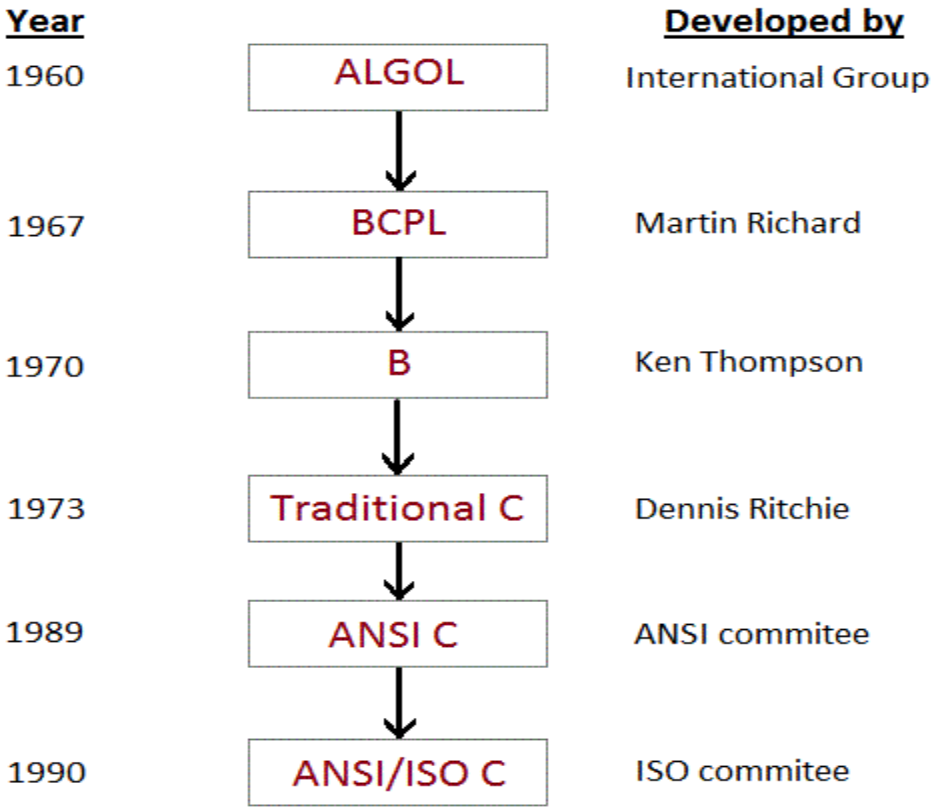
Program debugging is concerned with the finding and removal of the syntax errors only. After the removal of all syntax errors, our source program is converted into object code by the C Compiler.

Unit-2

Program Structure

History of C language

C language has evolved from three different structured language ALGOL, BCPL and B Language. It uses many concepts from these languages while introduced many new concepts such as datatypes, struct, pointer etc. In 1988, the language was formalised by **American National Standard Institute**(ANSI). In 1990, a version of C language was approved by the **International Standard Organisation**(ISO) and that version of C is also referred to as C89.



First C Program and its Structure

Lets see how to write a simple and most basic C program:

```
#include <stdio.h>

int main()
```

```c
{
 printf("Hello,World");  //single line comment
 return 0;
/*
    multi
    line
    comments
/*
}
```

```
Hello,World
```

Different parts of C program

- Pre-processor
- Header file
- Function
- Variables
- Statements & expressions
- Comments

All these are essential parts of a C language program.

Pre-processor

`#include` is the first word of any C program. It is also known as a **pre-processor**. The task of a pre-processor is to initialize the environment of the program, i.e to link the program with the header files required.

So, when we say `#include <stdio.h>`, it is to inform the compiler to include the **stdio.h** header file to the program before executing it.

Header file

A Header file is a collection of built-in(readymade) functions, which we can directly use in our program. Header files contain definitions of the functions which can be incorporated into any C program by using pre-processor `#include` statement with the header file. Standard header files are provided with each compiler, and covers a range of areas like string handling, mathematical functions, data conversion, printing and reading of variables.

With time, you will have a clear picture of what header files are, as of now consider as a readymade piece of function which comes packaged with the C language and you can use them without worrying about how they work, all you have to do is include the header file in your program.

To use any of the standard functions, the appropriate header file must be included. This is done at the beginning of the C source file.

For example, to use the `printf()` function in a program, which is used to display anything on the screen, the line `#include <stdio.h>` is required because the header file **stdio.h** contains the `printf()` function. All header files will have an extension `.h`

---

main() function

`main()` function is a function that must be there in every C program. Everything inside this function in a C program will be executed. In the above example, `int` written before the `main()` function is the **return type** of main() function. we will discuss about it in detail later. The curly braces `{ }` just after the **main()** function encloses the **body** of **main()** function.

We will learn what functions are in upcoming tutorials.

Comments

We can add comments in our program to describe what we are doing in the program. These comments are ignored by the compiler and are not executed.

To add a single line comment, start it by adding two forward slashses `//` followed by the comment.

To add multiline comment, enclode it between `/* .... */`, just like in the program above.

Return statement - return 0;

A return statement is just meant to define the end of any C program.

All the C programs can be written and edited in normal text editors like Notepad or Notepad++ and must be saved with a file name with extension as `.c`

If you do not add the extension `.c` then the compiler will not recognise it as a C language program file.

Compile and Run C Program

To compile and run a C language program, you need a C compiler. To setup a C language compiler in your Computer/laptop, there are two ways:

1. Download a full fledged IDE like Turbo C or Microsoft Visual C++, which comes along with a C language compiler.
2. Or, you use any text editor to edit the program files and download the C compiler separately.

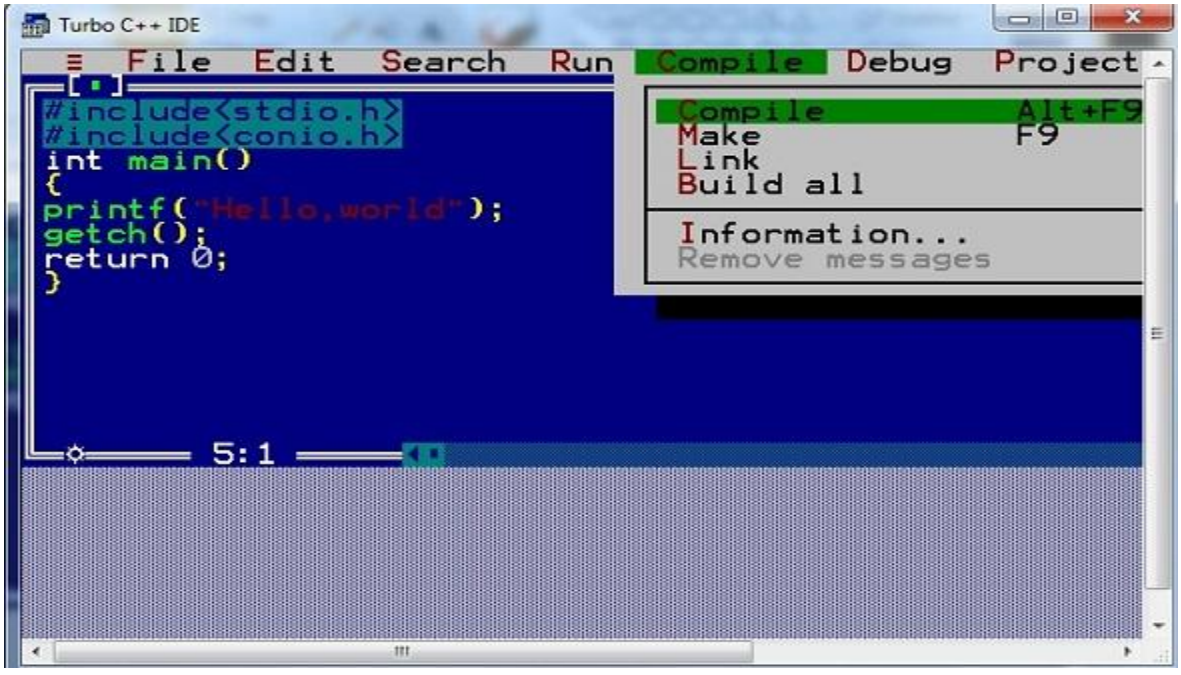
Using an IDE - Turbo C

We will recommend you to use **Turbo C** IDE, oldest IDE for c programming. It is freely available over internet and is good for a beginner.

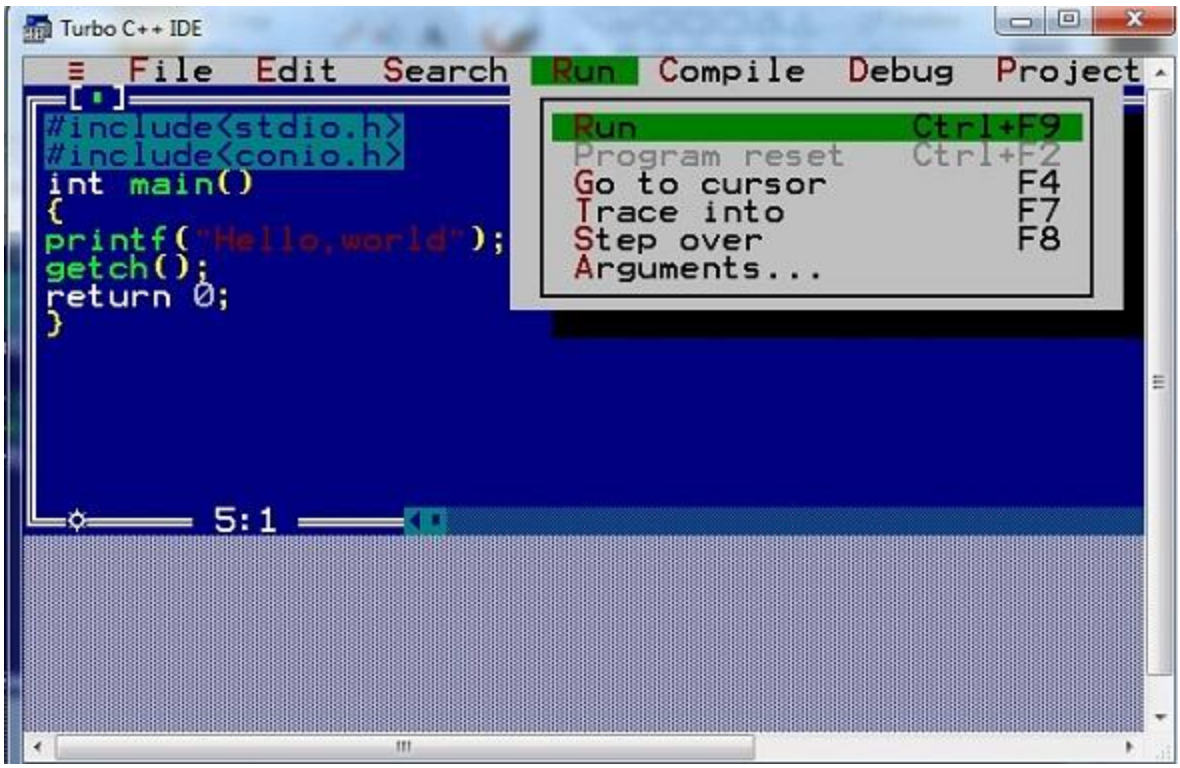
Step 1 : Open turbo C IDE(Integrated Development Environment), click on **File** and then click on New

Step 2 : Write the above example as it is

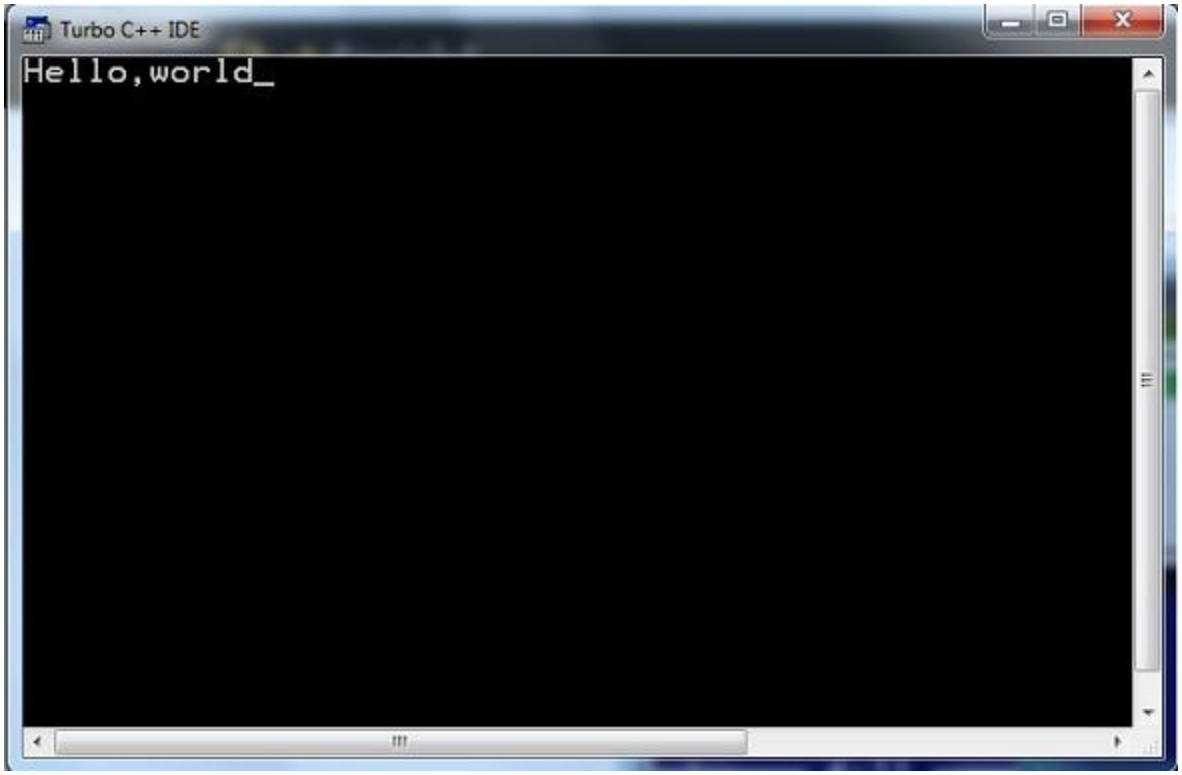
Step 3 : Click on compile or press Alt+f9 to compile the code



Step 4 : Click on Run or press Ctrl+f9 to run the code



Step 5 : Output



Difference between Compile and Run

You must be thinking why it is a 2 step process, first we compile the code and then we run the code. So, compilation is the process where the compiler checks whether the program is correct syntax wise, and there are no errors in the syntax.

When we run a compiled program, then it actually executes the statements inside the `main()`function.

C Language Basic Syntax Rules

C language syntax specify rules for sequence of characters to be written in C language. In simple language it states how to form statements in a C language program - How should the line of code start, how it should end, where to use double quotes, where to use curly brackets etc.

The rule specify how the character sequence will be grouped together, to form **tokens**. A smallest individual unit in C program is known as **C Token**. Tokens are either keywords, identifiers, constants, variables or any symbol which has some meaning in C language. A C program can also be called as a collection of various tokens.

So C tokens are basically the building blocks of a C program.

Semicolon `;`

Semicolon `;` is used to mark the end of a statement and beginning of another statement. Absence of semicolon at the end of any statement, will mislead the compiler to think that this statement is not yet finished and it will add the next consecutive statement after it, which may lead to compilation(syntax) error.

```
#include
int main()
{
    printf("Hello,World")
    return 0;
}
```

In the above program, we have omitted the semicolon from the `printf("...")` statement, hence the compiler will think that starting from `printf` uptill the semicolon after `return 0` statement, is a single statement and this will lead to compilation error.

Comments

Comments are plain simple text in a C program that are not compiled by the compiler. We write comments for better understanding of the program. Though writing comments is not compulsory, but it is recommended to make your program more descriptive. It make the code more readable.

There are two ways in which we can write comments.

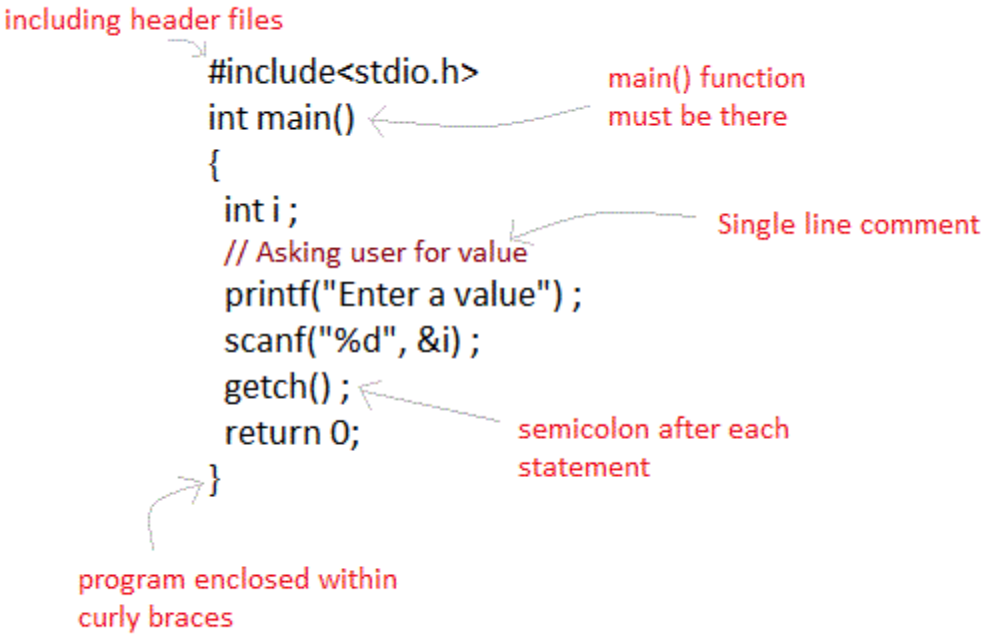
1. Using `//` This is used to write a single line comment.
2. Using `/* */`: The statements enclosed within `/*` and `*/` , are used to write multi-line comments.

Example of comments :

```
// This is a comment

/* This is a comment */

/* This is a long
and valid comment */

// this is not
  a valid comment
```

Some basic syntax rule for C program

- C is a case sensitive language so all C instructions must be written in lower case letter.
- All C statement must end with a semicolon.
- Whitespace is used in C to describe blanks and tabs.
- Whitespace is required between keywords and identifiers. We will learn about keywords and identifiers in the next tutorial.



What are Keywords in C?

Keywords are preserved words that have special meaning in C language. The meaning of C language keywords has already been described to the C compiler. These meaning cannot be changed. Thus, keywords cannot be used as variable names because that would try to change the existing meaning of the keyword, which is not allowed.(Don't worry if you do not know what variables are, you will soon understand.) There are total 32 keywords in C language.

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
const	extern	return	union

char	float	short	unsigned
continue	for	signed	volatile
default	goto	sizeof	void
do	if	static	while

What are Identifiers?

In C language identifiers are the names given to variables, constants, functions and user-define data. These identifier are defined against a set of rules.

Rules for an Identifier

1. An Identifier can only have alphanumeric characters(a-z , A-Z , 0-9) and underscore(_).
2. The first character of an identifier can only contain alphabet(a-z , A-Z) or underscore (_).
3. Identifiers are also case sensitive in C. For example **name** and **Name** are two different identifiers in C.
4. Keywords are not allowed to be used as Identifiers.
5. No special characters, such as semicolon, period, whitespaces, slash or comma are permitted to be used in or as Identifier.

When we declare a variable or any function in C language program, to use it we must provide a name to it, which identified it throughout the program, for example:

```
int myvariable = "Studytonight";
```

Here `myvariable` is the name or identifier for the variable which stores the value "Studytonight" in it.

Character set

In C language characters are grouped into the following catagories,

1. Letters(all alphabets a to z & A to Z).
2. Digits (all digits 0 to 9).
3. Special characters, (such as colon `:`, semicolon `;`, period `.`, underscore `_`, ampersand `&`etc).
4. White spaces.

C Input and Output

Input means to provide the program with some data to be used in the program and **Output** means to display data on screen or write the data to a printer or a file.

C programming language provides many built-in functions to read any given input and to display data on screen when there is a need to output the result.

In this tutorial, we will learn about such functions, which can be used in our program to take input from user and to output the result on screen.

All these built-in functions are present in C header files, we will also specify the name of header files in which a particular function is defined while discussing about it.

`scanf()` and `printf()` functions

The standard input-output header file, named `stdio.h` contains the definition of the functions `printf()` and `scanf()`, which are used to display output on screen and to take input from user respectively.

```
#include<stdio.h>


void main()
```

```
{
    // defining a variable
    int i;
    /*
        displaying message on the screen
        asking the user to input a value
    */
    printf("Please enter a value...");
    /*
        reading the value entered by the user
    */
    scanf("%d", &i);
    /*
        displaying the number as output
    */
    printf( "\nYou entered: %d", i);
}
```

When you will compile the above code, it will ask you to enter a value. When you will enter the value, it will display the value you have entered on screen.

You must be wondering what is the purpose of `%d` inside the `scanf()` or `printf()` functions. It is known as **format string** and this informs the `scanf()` function, what type of input to expect and in `printf()` it is used to give a heads up to the compiler, what type of output to expect.

Format String	Meaning
`%d`	Scan or print an integer as signed decimal number
`%f`	Scan or print a floating point number
`%c`	To scan or print a character
`%s`	To scan or print a character string. The scanning ends at whitespace.

We can also **limit the number of digits or characters** that can be input or output, by adding a number with the format string specifier, like `"%1d"` or `"%3s"`, the first one means a single numeric digit and the second one means 3 characters, hence if you try to input `42`, while `scanf()` has `"%1d"`, it will take only `4` as input. Same is the case for output.

In C Language, computer monitor, printer etc output devices are treated as files and the same process is followed to write output to these devices as would have been followed to write the output to a file.

NOTE : `printf()` function returns the number of characters printed by it, and `scanf()` returns the number of characters read by it.

```
int i = printf("studytonight");
```

In this program `printf("studytonight");` will return `12` as result, which will be stored in the variable `i`, because studytonight has 12 characters.

`getchar()` & `putchar()` functions

The `getchar()` function reads a character from the terminal and returns it as an integer. This function reads only single character at a time. You can use this method in a loop in case you want to read more than one character. The `putchar()` function displays the character passed to it on the screen and returns the same character. This function too displays only a single character at a time. In case you want to display more than one characters, use `putchar()` method in a loop.

```
#include <stdio.h>


void main( )
{
    int c;
    printf("Enter a character");
    /*
        Take a character as input and
        store it in variable c
    */
    c = getchar();
    /*
        display the character stored
        in variable c
    */
```

```
    putchar(c);
}
```

When you will compile the above code, it will ask you to enter a value. When you will enter the value, it will display the value you have entered.

`gets()` & `puts()` functions

The `gets()` function reads a line from **stdin**(standard input) into the buffer pointed to by `str` pointer, until either a terminating newline or EOF (end of file) occurs.
The `puts()` function writes the string `str` and a trailing newline to **stdout**.

`str` → This is the pointer to an array of chars where the C string is stored. (Ignore if you are not able to understand this now.)

```
#include<stdio.h>


void main()
{
    /* character array of length 100 */
    char str[100];
    printf("Enter a string");
    gets( str );
    puts( str );
    getch();
}
```

When you will compile the above code, it will ask you to enter a string. When you will enter the string, it will display the value you have entered.

Difference between `scanf()` and `gets()`

The main difference between these two functions is that `scanf()` stops reading characters when it encounters a space, but `gets()` reads space as character too.

If you enter name as **Study Tonight** using `scanf()` it will only read and store **Study** and will leave the part after space. But `gets()` function will read it completely.

Variables in C Language

When we want to store any information(data) on our computer/laptop, we store it in the computer's memory space. Instead of remembering the complex address of that memory space where we have stored our data, our operating system provides us with an option to create folders, name them, so that it becomes easier for us to find it and access it.

Similarly, in C language, when we want to use some data value in our program, we can store it in a memory space and name the memory space so that it becomes easier to access it.

The naming of an address is known as **variable**. Variable is the name of memory location. Unlike constant, variables are changeable, we can change value of a variable during execution of a program. A programmer can choose a meaningful variable name. Example : average, height, age, total etc.

Datatype of Variable

A **variable** in C language must be given a type, which defines what type of data the variable will hold.

It can be:

- `char`: Can hold/store a character in it.
- `int`: Used to hold an integer.
- `float`: Used to hold a float value.
- `double`: Used to hold a double value.
- `void`

Rules to name a Variable

1. Variable name must not start with a digit.

2. Variable name can consist of alphabets, digits and special symbols like underscore `_`.
3. Blank or spaces are not allowed in variable name.
4. Keywords are not allowed as variable name.
5. Upper and lower case names are treated as different, as C is case-sensitive, so it is suggested to keep the variable names in lower case.

Declaring, Defining and Initializing a variable

Declaration of variables must be done before they are used in the program. Declaration does the following things.

1. It tells the compiler what the variable name is.
2. It specifies what type of data the variable will hold.
3. Until the variable is defined the compiler doesn't have to worry about allocating memory space to the variable.
4. Declaration is more like informing the compiler that there exist a variable with following datatype which is used in the program.
5. A variable is declared using the `extern` keyword, outside the `main()` function.

```
extern int a;
extern float b;
extern double c, d;
```

Defining a variable means the compiler has to now assign a storage to the variable because it will be used in the program. It is not necessary to declare a variable using `extern` keyword, if you want to use it in your program. You can directly define a variable inside the `main()` function and use it.

To define a function we must provide the datatype and the variable name. We can even define multiple variables of same datatype in a single line by using comma to separate them.

```
int a;
float b, c;
```

Initializing a variable means to provide it with a value. A variable can be initialized and defined in a single statement, like:

```
int a = 10;
```

Let's write a program in which we will use some variables.

```
#include <stdio.h>

// Variable declaration(optional)
extern int a, b;
extern int c;

int main () {

    /* variable definition: */
    int a, b;

    /* actual initialization */
    a = 7;
    b = 14;
    /* using addition operator */
    c = a + b;
    /* display the result */
    printf("Sum is : %d \n", c);

    return 0;
}
```

```
Sum is : 21
```

You must be thinking how does this `printf()` works, right? Do not worry, we will learn about it along with other ways to input and output datain C language in the next tutorial.

Difference between Variable and Identifier?

An Identifier is a name given to any variable, function, structure, pointer or any other entity in a programming language. While a variable, as we have just learned in this tutorial is a named memory location to store data which is used in the program.

Identifier	**Variable**
Identifier is the name given to a variable, function etc.	While, variable is used to name a memory location which stores data.
An identifier can be a variable, but not all indentifiers are variables.	All variable names are identifiers.

Example:
`// a variable`
`int studytonight;`
`// or, a function`
`int studytonight() {`
`..`
`}`
	Example:
	`// int variable`
	`int a;`
	`// float variable`
	`float a;`

Data types in C Language

Data types specify how we enter data into our programs and what type of data we enter. C language has some predefined set of data types to handle various kinds of data that we can use in our program. These datatypes have different storage capacities.

C language supports 2 different type of data types:

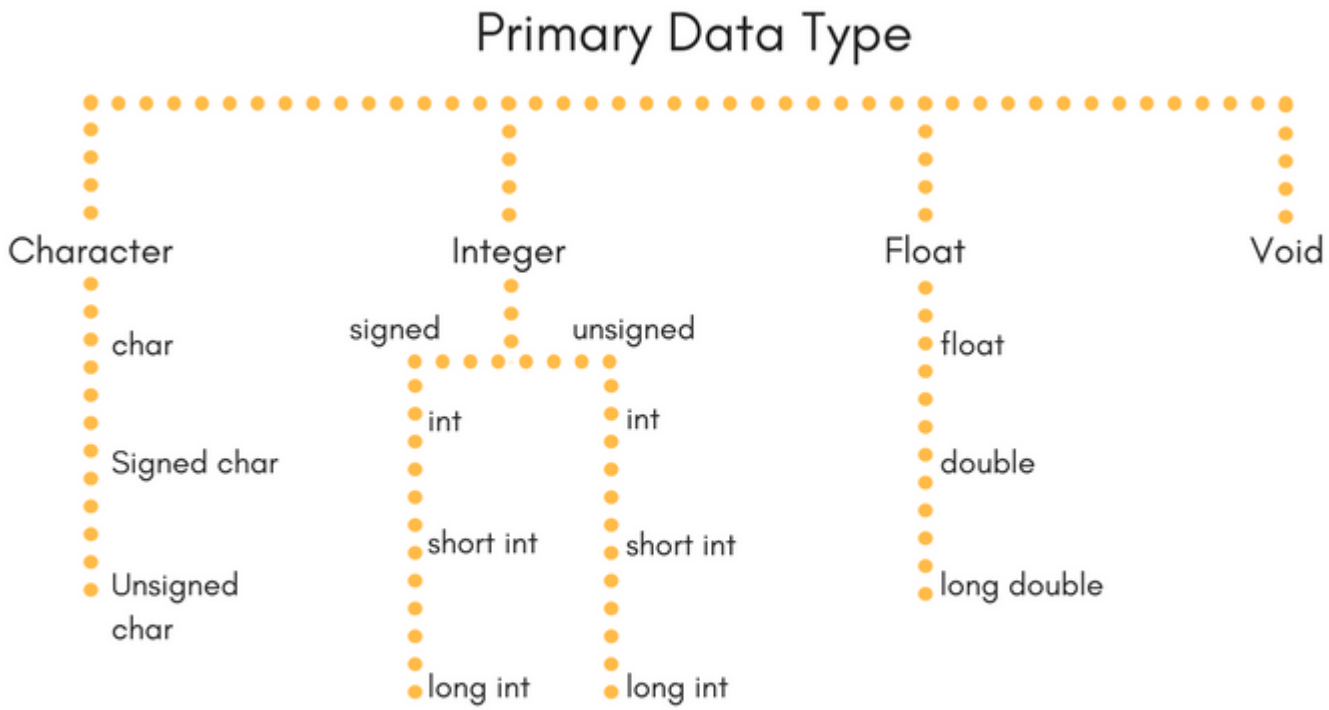
1. **Primary data types**:

   These are fundamental data types in C namely integer(`int`), floating point(`float`), character(`char`) and `void`.

2. **Derived data types**:

   Derived data types are nothing but primary datatypes but a little twisted or grouped together like **array**, **stucture**, **union** and **pointer**. These are discussed in details later.

Data type determines the type of data a variable will hold. If a variable `x` is declared as `int`. it means x can hold only integer values. Every variable which is used in the program must be declared as what data-type it is.



Integer type

Integers are used to store whole numbers.

Size and range of Integer type on 16-bit machine:

Type	Size(bytes)	Range
int or signed int	2	-32,768 to 32767
unsigned int	2	0 to 65535
short int or signed short int	1	-128 to 127
unsigned short int	1	0 to 255
long int or signed long int	4	-2,147,483,648 to 2,147,483,647
unsigned long int	4	0 to 4,294,967,295

Floating point type

Floating types are used to store real numbers.

Size and range of Integer type on 16-bit machine

Type	Size(bytes)	Range
Float	4	3.4E-38 to 3.4E+38
double	8	1.7E-308 to 1.7E+308
long double	10	3.4E-4932 to 1.1E+4932

Character type

Character types are used to store characters value.

Size and range of Integer type on 16-bit machine

Type	Size(bytes)	Range
char or signed char	1	-128 to 127
unsigned char	1	0 to 255

---

void type

`void` type means no value. This is usually used to specify the type of functions which returns nothing. We will get acquainted to this datatype as we start learning more advanced topics in C language, like functions, pointers etc.

Operators in C Language

C language supports a rich set of built-in operators. An operator is a symbol that tells the compiler to perform a certain mathematical or logical manipulation. Operators are used in programs to manipulate data and variables.

C operators can be classified into following types:

- Arithmetic operators
- Relational operators
- Logical operators
- Bitwise operators
- Assignment operators
- Conditional operators
- Special operators

Arithmetic operators

C supports all the basic arithmetic operators. The following table shows all the basic arithmetic operators.

Operator	Description
+	adds two operands
-	subtract second operands from first
*	multiply two operand

/	divide numerator by denominator
%	remainder of division
++	Increment operator - increases integer value by one
--	Decrement operator - decreases integer value by one

Relational operators

The following table shows all relation operators supported by C.

Operator	**Description**
==	Check if two operand are equal
!=	Check if two operand are not equal.
>	Check if operand on the left is greater than operand on the right
<	Check operand on the left is smaller than right operand
>=	check left operand is greater than or equal to right operand
<=	Check if operand on left is smaller than or equal to right operand

---

Logical operators

C language supports following 3 logical operators. Suppose `a = 1` and `b = 0`,

Operator	Description	Example
&&	Logical AND	(a && b) is false
||	Logical OR	(a || b) is true
!	Logical NOT	(!a) is false

Bitwise operators

Bitwise operators perform manipulations of data at **bit level**. These operators also perform **shifting of bits** from right to left. Bitwise operators are not applied to `float` or `double`(These are datatypes, we will learn about them in the next tutorial).

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

Now lets see truth table for bitwise `&`, `|` and `^`

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

The bitwise **shift** operator, shifts the bit value. The left operand specifies the value to be shifted and the right operand specifies the number of positions that the bits in the value have to be shifted. Both operands have the same precedence.

Example :

```
a = 0001000
b = 2
a << b = 0100000
a >> b = 0000010
```

Assignment Operators

Assignment operators supported by C language are as follows.

Operator	Description	Example
=	assigns values from right side operands to left side operand	a=b

+=	adds right operand to the left operand and assign the result to left	a+=b is same as a=a+b
-=	subtracts right operand from the left operand and assign the result to left operand	a-=b is same as a=a-b
*=	mutiply left operand with the right operand and assign the result to left operand	a*=b is same as a=a*b
/=	divides left operand with the right operand and assign the result to left operand	a/=b is same as a=a/b
%=	calculate modulus using two operands and assign the result to left operand	a%=b is same as a=a%b

---

Conditional operator

The conditional operators in C language are known by two more names

1. **Ternary Operator**
2. **? : Operator**

It is actually the `if` condition that we use in C language decision making, but using conditional operator, we turn the `if` condition statement into a short and simple operator.

The syntax of a conditional operator is :

```
expression 1 ? expression 2: expression 3
```

Explanation:

- The question mark **"?"** in the syntax represents the **if** part.

- The first expression (expression 1) generally returns either true or false, based on which it is decided whether (expression 2) will be executed or (expression 3)
- If (expression 1) returns true then the expression on the left side of " **:** " i.e (expression 2) is executed.
- If (expression 1) returns false then the expression on the right side of " **:** " i.e (expression 3) is executed.

Special operator

Operator	**Description**	**Example**
sizeof	Returns the size of an variable	**sizeof(x)** return size of the variable **x**
&	Returns the address of an variable	**&x ;** return address of the variable **x**
*	Pointer to a variable	***x ;** will be pointer to a variable **x**

Unit-3

Control Structures

Decision making in C

Decision making is about deciding the order of execution of statements based on certain conditions or repeat a group of statements until certain specified conditions are met. C language handles decision-making by supporting the following statements,

- `if` statement
- `switch` statement
- conditional operator statement (`? :` operator)
- `goto` statement

Decision making with `if` statement

The `if` statement may be implemented in different forms depending on the complexity of conditions to be tested. The different forms are,

1. Simple `if` statement
2. `if....else` statement
3. Nested `if....else` statement
4. Using `else if` statement

Simple `if` statement

The general form of a simple `if` statement is,

```
if(expression)
{
    statement inside;
}
    statement outside;
```

If the *expression* returns true, then the **statement-inside** will be executed, otherwise **statement-inside** is skipped and only the **statement-outside** is executed.

Example:

```
#include <stdio.h>

void main( )
{
    int x, y;
    x = 15;
    y = 13;
    if (x > y )
    {
        printf("x is greater than y");
    }
}
```

```
x is greater than y
```

`if...else` statement

The general form of a simple `if...else` statement is,

```
if(expression)
{
    statement block1;
}
else
{
    statement block2;
}
```

If the *expression* is true, the **statement-block1** is executed, else **statement-block1** is skipped and **statement-block2** is executed.

Example:

```
#include <stdio.h>

void main( )
{
    int x, y;
    x = 15;
    y = 18;
    if (x > y )
    {
        printf("x is greater than y");
    }
    else
    {
        printf("y is greater than x");
    }
}
```

```
y is greater than x
```

Nested `if....else` statement

The general form of a nested `if...else` statement is,

```
if( expression )
{
    if( expression1 )
    {
        statement block1;
    }
    else
    {
        statement block2;
    }
}
else
{
    statement block3;
}
```

if *expression* is false then **statement-block3** will be executed, otherwise the execution continues and enters inside the first `if` to perform the check for the next `if` block, where if *expression 1* is true the **statement-block1** is executed otherwise **statement-block2** is executed.

Example:

```
#include <stdio.h>

void main( )
{
    int a, b, c;
    printf("Enter 3 numbers...");
    scanf("%d%d%d",&a, &b, &c);
    if(a > b)
    {
        if(a > c)
        {
```

```
            printf("a is the greatest");
        }
        else
        {
            printf("c is the greatest");
        }
    }
    else
    {
        if(b > c)
        {
            printf("b is the greatest");
        }
        else
        {
            printf("c is the greatest");
        }
    }
}
```

`else if` ladder

The general form of else-if ladder is,

```
if(expression1)
{
    statement block1;
}
else if(expression2)
{
    statement block2;
}
else if(expression3 )
{
    statement block3;
}
```

```
else
    default statement;
```

The expression is tested from the top(of the ladder) downwards. As soon as a **true** condition is found, the statement associated with it is executed.

Example :

```
#include <stdio.h>

void main( )
{
    int a;
    printf("Enter a number...");
    scanf("%d", &a);
    if(a%5 == 0 && a%8 == 0)
    {
        printf("Divisible by both 5 and 8");
    }
    else if(a%8 == 0)
    {
        printf("Divisible by 8");
    }
    else if(a%5 == 0)
    {
        printf("Divisible by 5");
    }
    else
    {
        printf("Divisible by none");
    }
}
```

Points to Remember

1. In `if` statement, a single statement can be included without enclosing it into curly braces `{ ... }`

```
int a = 5;
if(a > 4)
    printf("success");
```

No curly braces are required in the above case, but if we have more than one statement inside `if`condition, then we must enclose them inside curly braces.

4. `==` must be used for comparison in the expression of `if` condition, if you use `=` the expression will always return **true**, because it performs assignment not comparison.
5. Other than **0(zero)**, all other values are considered as **true**.

```
if(27)
    printf("hello");
```

In above example, **hello** will be printed.

Switch statement in C

When you want to solve multiple option type problems, for example: Menu like program, where one value is associated with each option and you need to choose only one at a time, then, `switch`statement is used.

Switch statement is a control statement that allows us to choose only one choice among the many given choices. The expression in `switch` evaluates to return an integral value, which is then compared to the values present in different cases. It executes that block of code which matches the case value. If there is no match, then **default** block is executed(if present). The general form of `switch` statement is,

```
switch(expression)
{
    case value-1:
        block-1;
        break;
    case value-2:
        block-2;
        break;
    case value-3:
        block-3;
```

```
        break;
    case value-4:
        block-4;
            break;
    default:
            default-block;
        break;
}
```

Rules for using `switch` statement

1. The expression (after switch keyword) must yield an **integer** value i.e the expression should be an integer or a variable or an expression that evaluates to an integer.
2. The case **label** values must be unique.
3. The case label must end with a colon(:)
4. The next line, after the **case** statement, can be any valid C statement.

Points to Remember

1. We don't use those expressions to evaluate switch case, which may return floating point values or strings or characters.
2. `break` statements are used to **exit** the switch block. It isn't necessary to use `break` after each block, but if you do not use it, then all the consecutive blocks of code will get executed after the matching block.

```
3. int i = 1;
4. switch(i)
5. {
6.     case 1:
```

```
        printf("A");        // No break
    case 2:
        printf("B");        // No break
    case 3:
        printf("C");
        break;
}
```

```
A B C
```

The output was supposed to be only **A** because only the first case matches, but as there is no `break` statement after that block, the next blocks are executed too, until it a `break` statement in encountered or the execution reaches the end of the `switch` block.

13. **default** case is executed when none of the mentioned case matches the `switch` expression. The default case can be placed anywhere in the `switch` case. Even if we don't include the default case, `switch` statement works.
14. Nesting of `switch` statements are allowed, which means you can have `switch` statements inside another `switch` block. However, nested `switch` statements should be avoided as it makes the program more complex and less readable.

Example of `switch` statement

```
#include<stdio.h>
void main( )
{
    int a, b, c, choice;
    while(choice != 3)
    {
```

```
        /* Printing the available options */
        printf("\n 1. Press 1 for addition");
        printf("\n 2. Press 2 for subtraction");
        printf("\n Enter your choice");
        /* Taking users input */
        scanf("%d", &choice);

        switch(choice)
        {
            case 1:
                printf("Enter 2 numbers");
                scanf("%d%d", &a, &b);
                c = a + b;
                printf("%d", c);
                break;
            case 2:
                printf("Enter 2 numbers");
                scanf("%d%d", &a, &b);
                c = a - b;
                printf("%d", c);
                break;
            default:
                printf("you have passed a wrong key");
                printf("\n press any key to continue");
        }
    }
}
```

Difference between `switch` and `if`

- `if` statements can evaluate `float` conditions. `switch` statements cannot evaluate `float`conditions.

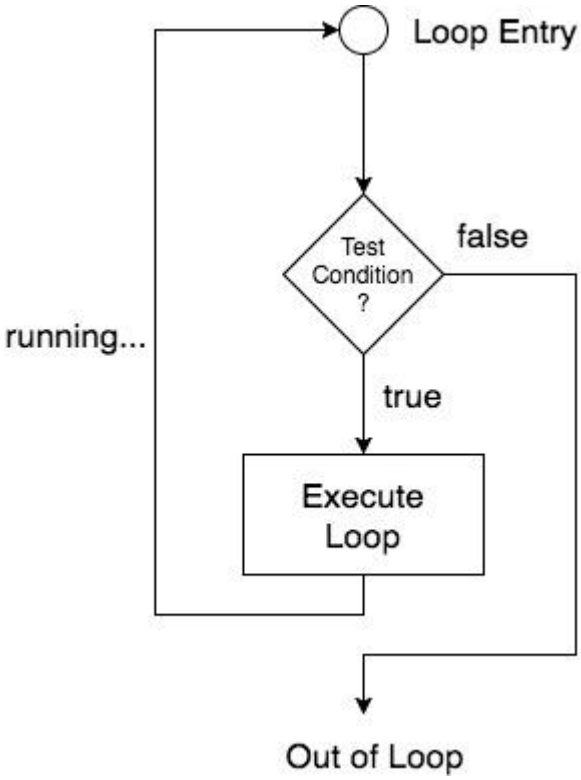
- `if` statement can evaluate relational operators. `switch` statement cannot evaluate relational operators i.e they are not allowed in `switch` statement.

How to use Loops in C

In any programming language including C, loops are used to execute a set of statements repeatedly until a particular condition is satisfied.

How it Works

The below diagram depicts a loop execution,



As per the above diagram, if the Test Condition is true, then the loop is executed, and if it is false then the execution breaks out of the loop. After the loop is successfully executed the execution again starts from the Loop entry and again checks for the Test condition, and this keeps on repeating.

The sequence of statements to be executed is kept inside the curly braces `{ }` known as the **Loop body**. After every execution of the loop body, **condition** is verified, and if

it is found to be **true** the loop body is executed again. When the condition check returns **false**, the loop body is not executed, and execution breaks out of the loop.

Types of Loop

There are 3 types of Loop in C language, namely:

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

`while` loop

`while` loop can be addressed as an **entry control** loop. It is completed in 3 steps.

- Variable initialization.(e.g `int x = 0;`)
- condition(e.g `while(x <= 10)`)
- Variable increment or decrement (`x++` or `x--` or `x = x + 2`)

Syntax :

```
variable initialization;
while(condition)
{
    statements;
    variable increment or decrement;
}
```

Example: Program to print first 10 natural numbers

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

```c
void main( )
{
    int x;
    x = 1;
    while(x <= 10)
    {
        printf("%d\t", x);
        /* below statement means, do x = x+1, increment x by 1*/
        x++;
    }
}
```

```
1 2 3 4 5 6 7 8 9 10
```

`for` loop

`for` loop is used to execute a set of statements repeatedly until a particular condition is satisfied. We can say it is an **open ended loop.**. General format is,

```c
for(initialization; condition; increment/decrement)
{
    statement-block;
}
```

In `for` loop we have exactly two semicolons, one after initialization and second after the condition. In this loop we can have more than one initialization or increment/decrement, separated using comma operator. But it can have only one **condition**.

The `for` loop is executed as follows:

1. It first evaluates the initialization code.
2. Then it checks the condition expression.
3. If it is **true**, it executes the for-loop body.
4. Then it evaluate the increment/decrement condition and again follows from step 2.
5. When the condition expression becomes **false**, it exits the loop.

Example: Program to print first 10 natural numbers

```
#include<stdio.h>

void main( )
{
    int x;
    for(x = 1; x <= 10; x++)
    {
        printf("%d\t", x);
    }
}
```

```
1 2 3 4 5 6 7 8 9 10
```

Nested `for` loop

We can also have nested `for` loops, i.e one `for` loop inside another `for` loop. Basic syntax is,

```
for(initialization; condition; increment/decrement)
{
    for(initialization; condition; increment/decrement)
    {
        statement ;
    }
}
```

Example: Program to print half Pyramid of numbers

```
#include<stdio.h>

void main( )
{
    int i, j;
    /* first for loop */
```

```
    for(i = 1; i < 5; i++)
    {
        printf("\n");
        /* second for loop inside the first */
        for(j = i; j > 0; j--)
        {
            printf("%d", j);
        }
    }
}
```

```
1
21
321
4321
54321
```

`do while` loop

In some situations it is necessary to execute body of the loop before testing the condition. Such situations can be handled with the help of `do-while` loop. `do` statement evaluates the body of the loop first and at the end, the condition is checked using `while` statement. It means that the body of the loop will be executed at least once, even though the starting condition inside `while` is initialized to be **false**. General syntax is,

```
do
{
    .....
    .....
}
while(condition)
```

Example: Program to print first 10 multiples of 5.

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

```c
void main()
{
    int a, i;
    a = 5;
    i = 1;
    do
    {
        printf("%d\t", a*i);
        i++;
    }
    while(i <= 10);
}
```

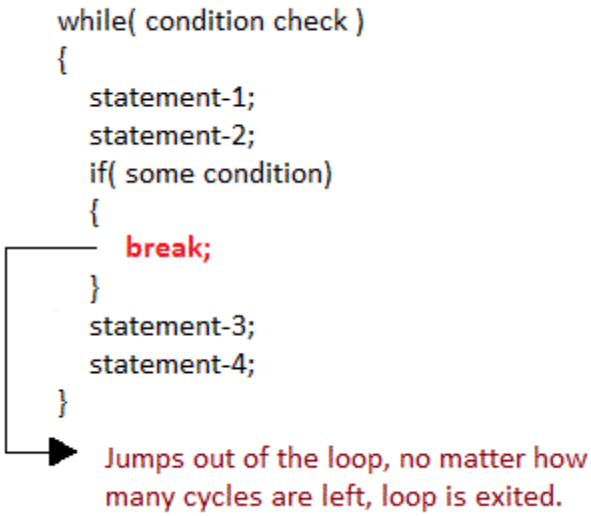
```
5 10 15 20 25 30 35 40 45 50
```

Jumping Out of Loops

Sometimes, while executing a loop, it becomes necessary to skip a part of the loop or to leave the loop as soon as certain condition becomes **true**. This is known as jumping out of loop.

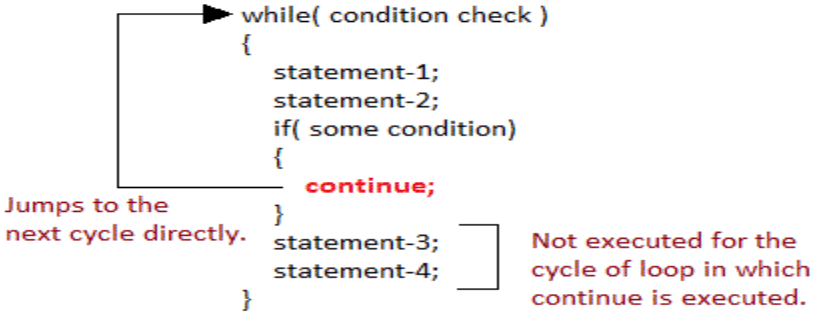
1) break statement

When `break` statement is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop.



2) continue statement

It causes the control to go directly to the test-condition and then continue the loop process. On encountering `continue`, cursor leave the current cycle of loop, and starts with the next cycle.



Unit-4

Pointers

Introduction to Pointers

A Pointer in C language is a variable which holds the address of another variable of same data type.

Pointers are used to access memory and manipulate the address.

Pointers are one of the most distinct and exciting features of C language. It provides power and flexibility to the language. Although pointers may appear a little confusing and complicated in the beginning, but trust me, once you understand the concept, you will be able to do so much more with C language.

Before we start understanding what pointers are and what they can do, let's start by understanding what does "Address of a memory location" means?

Address in C

Whenever a variable is defined in C language, a memory location is assigned for it, in which it's value will be stored. We can easily check this memory address, using the `&` symbol.

If `var` is the name of the variable, then `&var` will give it's address.

Let's write a small program to see memory address of any variable that we define in our program.

```c
#include<stdio.h>


void main()
{
    int var = 7;
    printf("Value of the variable var is: %d\n", var);
    printf("Memory address of the variable var is: %x\n", &var);
}
```

```
Value of the variable var is: 7

Memory address of the variable var is: bcc7a00
```

You must have also seen in the function `scanf()`, we mention `&var` to take user input for any variable `var`.

```
scanf("%d", &var);
```

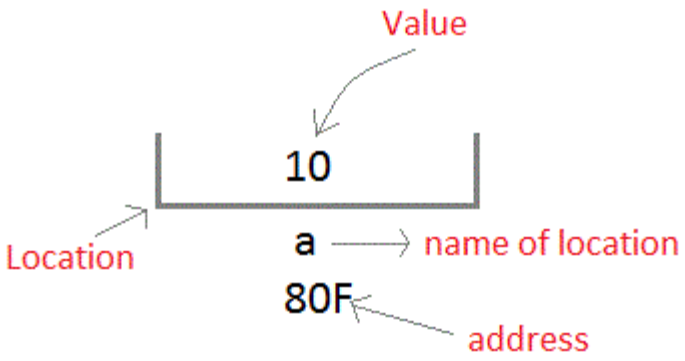
This is used to store the user inputted value to the address of the variable `var`.

Concept of Pointers

Whenever a **variable** is declared in a program, system allocates a location i.e an address to that variable in the memory, to hold the assigned value. This location has its own address number, which we just saw above.

Let us assume that system has allocated memory location `80F` for a variable `a`.

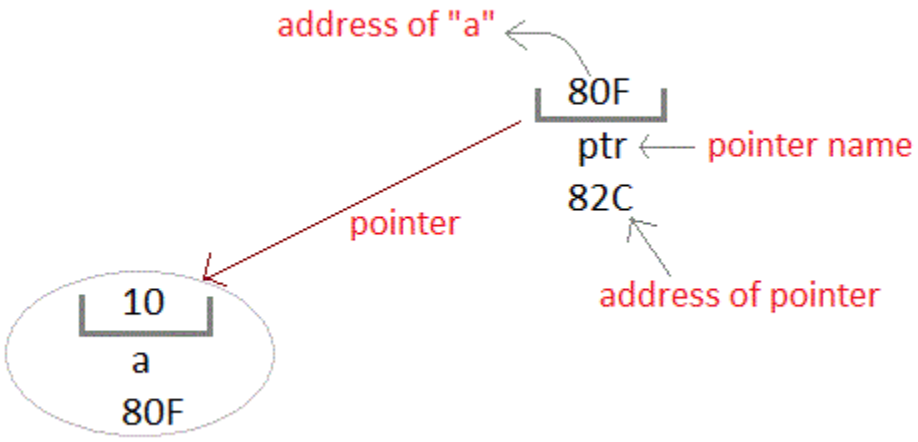
```
int a = 10;
```



We can access the value `10` either by using the variable name `a` or by using its address `80F`.

The question is how we can access a variable using it's address? Since the memory addresses are also just numbers, they can also be assigned to some other variable. The variables which are used to hold memory addresses are called **Pointer variables**.

A **pointer** variable is therefore nothing but a variable which holds an address of some other variable. And the value of a **pointer variable** gets stored in another memory location.



Benefits of using pointers

Below we have listed a few benefits of using pointers:

1. Pointers are more efficient in handling Arrays and Structures.
2. Pointers allow references to function and thereby helps in passing of function as arguments to other functions.
3. It reduces length of the program and its execution time as well.
4. It allows C language to support Dynamic Memory management.

Declaring, Initializing and using a pointer variable

In this tutorial, we will learn how to declare, initialize and use a pointer. We will also learn what NULL pointer are and where to use them. Let's start!

Declaration of Pointer variable

General syntax of pointer declaration is,

```
datatype *pointer_name;
```

Data type of a pointer must be same as the data type of the variable to which the pointer variable is pointing. `void` type pointer works with all data types, but is not often used.

Here are a few examples:

```
int *ip     // pointer to integer variable
float *fp       // pointer to float variable
double *dp      // pointer to double variable
char *cp        // pointer to char variable
```

Initialization of Pointer variable

Pointer Initialization is the process of assigning address of a variable to a **pointer** variable. Pointer variable can only contain address of a variable of the same data type. In C language **address operator** `&` is used to determine the address of a variable. The `&` (immediately preceding a variable name) returns the address of the variable associated with it.

```
#include<stdio.h>


void main()
{
    int a = 10;
    int *ptr;       //pointer declaration
    ptr = &a;       //pointer initialization
}
```

Pointer variablea always point to variables of same datatype. Let's have an example to showcase this:

```
#include<stdio.h>


void main()
{
    float a;
    int *ptr;
    ptr = &a;       // ERROR, type mismatch
}
```

If you are not sure about which variable's address to assign to a pointer variable while declaration, it is recommended to assign a `NULL` value to your pointer variable. A pointer which is assigned a `NULL`value is called a **NULL pointer**.

```
#include <stdio.h>

int main()
{
    int *ptr = NULL;
    return 0;
}
```

Using the pointer or Dereferencing of Pointer

Once a pointer has been assigned the address of a variable, to access the value of the variable, pointer is **dereferenced**, using the **indirection operator** or **dereferencing operator** `*`.

```
#include <stdio.h>

int main()
{
    int a, *p;  // declaring the variable and pointer
    a = 10;
    p = &a;     // initializing the pointer

    printf("%d", *p);    //this will print the value of 'a'

    printf("%d", *&a);   //this will also print the value of 'a'

    printf("%u", &a);    //this will print the address of 'a'

    printf("%u", p);     //this will also print the address of 'a'

    printf("%u", &p);    //this will print the address of 'p'
```

```
    return 0;
}
```

Points to remember while using pointers:

1. While declaring/initializing the pointer variable, `*` indicates that the variable is a pointer.
2. The address of any variable is given by preceding the variable name with Ampersand `&`.
3. The pointer variable stores the address of a variable. The declaration `int *a` doesn't mean that `a` is going to contain an integer value. It means that `a` is going to contain the address of a variable storing integer value.
4. To access the value of a certain address stored by a pointer variable, `*` is used. Here, the `*` can be read as **'value at'**.

Time for an Example

```
#include <stdio.h>

int main()
{
    int i = 10;     // normal integer variable storing value 10
    int *a;     // since '*' is used, hence its a pointer variable

    /*
        '&' returns the address of the variable 'i'
        which is stored in the pointer variable 'a'
    */
    a = &i;

    /*
```

```
        below, address of variable 'i', which is stored
        by a pointer variable 'a' is displayed
    */
   printf("Address of variable i is %u\n", a);


    /*
        below, '*a' is read as 'value at a'
        which is 10
    */
   printf("Value at the address, which is stored by pointer variable a is %d\n",
*a);


   return 0;
}
```

```
Address of variable i is 2686728 (The address may vary)
Value at an address, which is stored by pointer variable a is 10
```

Pointer to a Pointer(Double Pointer)

Pointers are used to store the address of other variables of similar datatype. But if you want to store the address of a pointer variable, then you again need a pointer to store it. Thus, when one pointer variable stores the address of another pointer variable, it is known as **Pointer to Pointer** variable or **Double Pointer**.

Syntax:

```
int **p1;
```

Here, we have used two indirection operator(`*`) which stores and points to the address of a pointer variable i.e, `int *`. If we want to store the address of this (double pointer) variable `p1`, then the syntax would become:

```
int ***p2
```

Simple program to represent Pointer to a Pointer

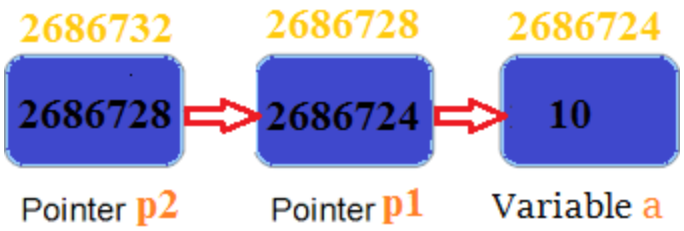
```
#include <stdio.h>


int main() {
```

```
    int  a = 10;
    int  *p1;       //this can store the address of variable a
    int  **p2;
    /*
        this can store the address of pointer variable p1 only.
        It cannot store the address of variable 'a'
    */

    p1 = &a;
    p2 = &p1;

    printf("Address of a = %u\n", &a);
    printf("Address of p1 = %u\n", &p1);
    printf("Address of p2 = %u\n\n", &p2);

    // below print statement will give the address of 'a'
    printf("Value at the address stored by p2 = %u\n", *p2);

    printf("Value at the address stored by p1 = %d\n\n", *p1);

    printf("Value of **p2 = %d\n", **p2); //read this *(*p2)

    /*
        This is not allowed, it will give a compile time error-
        p2 = &a;
        printf("%u", p2);
    */
    return 0;
}
```

```
Address of a = 2686724
Address of p1 = 2686728
Address of p2 = 2686732
Value at the address stored by p2 = 2686724
Value at the address stored by p1 = 10
```

```
Value of **p2 = 10
```

Explanation of the above program



- `p1` pointer variable can only hold the address of the variable `a` (i.e Number of indirection operator(*)-1 variable). Similarly, `p2` variable can only hold the address of variable `p1`. It cannot hold the address of variable `a`.
- `*p2` gives us the value at an address stored by the `p2` pointer. `p2` stores the address of `p1`pointer and value at the address of `p1` is the address of variable `a`.

  Thus, `*p2` prints address of `a`.
- `**p2` can be read as `*(*p2)`. Hence, it gives us the value stored at the address `*p2`.

  From above statement, you know `*p2` means the address of variable a. Hence, the value at the address `*p2` is 10. Thus, `**p2` prints `10`.

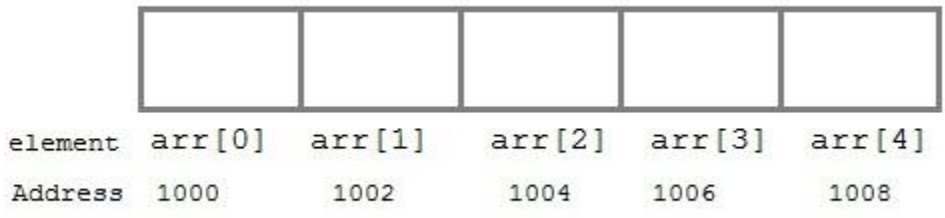
Pointer and Arrays

When an array is declared, compiler allocates sufficient amount of memory to contain all the elements of the array. Base address i.e address of the first element of the array is also allocated by the compiler.

Suppose we declare an array `arr`,

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

Assuming that the base address of `arr` is 1000 and each integer requires two bytes, the five elements will be stored as follows:



Here variable `arr` will give the base address, which is a constant pointer pointing to the first element of the array, `arr[0]`. Hence `arr` contains the address of `arr[0]` i.e `1000`. In short, `arr` has two purpose - it is the name of the array and it acts as a pointer pointing towards the first element in the array.

`arr` is equal to `&arr[0]` by default

We can also declare a pointer of type `int` to point to the array `arr`.

```
int *p;
p = arr;
// or,
p = &arr[0];    //both the statements are equivalent.
```

Now we can access every element of the array `arr` using `p++` to move from one element to another.

NOTE: You cannot decrement a pointer once incremented. `p--` won't work.

Pointer to Array

As studied above, we can use a pointer to point to an array, and then we can use that pointer to access the array elements. Lets have an example,

```
#include <stdio.h>


int main()
{
    int i;
    int a[5] = {1, 2, 3, 4, 5};
    int *p = a;     // same as int*p = &a[0]
```

```
    for (i = 0; i < 5; i++)
    {
        printf("%d", *p);
        p++;
    }

    return 0;
}
```

In the above program, the pointer `*p` will print all the values stored in the array one by one. We can also use the Base address (`a` in above case) to act as a pointer and print all the values.

Replacing the **printf("%d", *p);** statement of above example, with below mentioned statements. Lets see what will be the result.

printf("%d", a[i]); ⟶ **prints the array, by incrementing index**

printf("%d", i[a]); ⟶ **this will also print elements of array**

printf("%d", a+i); ⟶ **This will print address of all the array elements**

printf("%d", *(a+i)); ⟶ **Will print value of array element.**

printf("%d", *a); ⟶ **will print value of a[0] only**

a++; ⟶ **Compile time error, we cannot change base address of the array.**

The generalized form for using pointer with an array,

```
*(a+i)
```

is same as:

```
a[i]
```

Pointer to Multidimensional Array

A multidimensional array is of form, `a[i][j]`. Lets see how we can make a pointer point to such an array. As we know now, name of the array gives its base address.
In `a[i][j]`, `a` will give the base address of this array, even `a + 0 + 0` will also give the base address, that is the address of `a[0][0]`element.

Here is the generalized form for using pointer with multidimensional arrays.

```
*(*(a + i) + j)
```

which is same as,

```
a[i][j]
```

Pointer and Character strings

Pointer can also be used to create strings. Pointer variables of `char` type are treated as string.

```
char *str = "Hello";
```

The above code creates a string and stores its address in the pointer variable `str`. The pointer `str`now points to the first character of the string "Hello". Another important thing to note here is that the string created using `char` pointer can be assigned a value at **runtime**.

```
char *str;
str = "hello";      //this is Legal
```

The content of the string can be printed using `printf()` and `puts()`.

```
printf("%s", str);
puts(str);
```

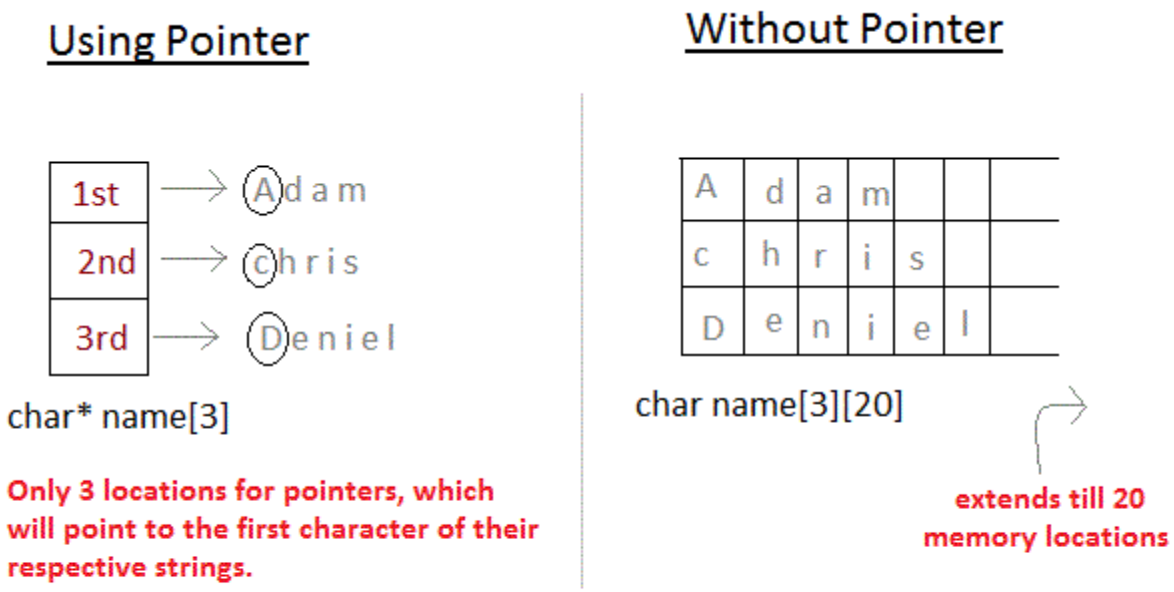
Notice that `str` is pointer to the string, it is also name of the string. Therefore we do not need to use indirection operator `*`.

Array of Pointers

We can also have array of pointers. Pointers are very helpful in handling character array with rows of varying length.

```
char *name[3] = {
    "Adam",
    "chris",
```

```
    "Deniel"
};
//Now lets see same array without using pointer
char name[3][20] = {
    "Adam",
    "chris",
    "Deniel"
};
```



In the second approach memory wastage is more, hence it is prefered to use pointer in such cases.

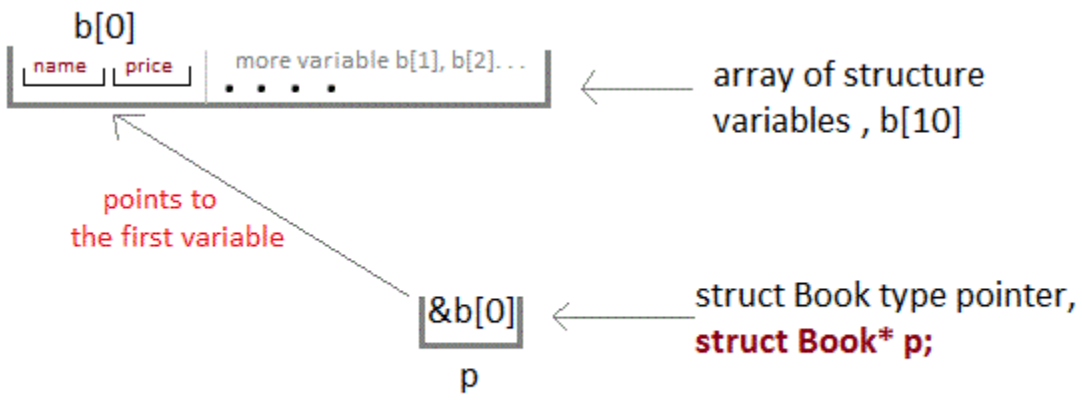
When we say memory wastage, it doesn't means that the strings will start occupying less space, no, characters will take the same space, but when we define array of characters, a contiguos memory space is located equal to the maximum size of the array, which is a wastage, which can be avoided if we use pointers instead.

Pointer to Structure Array

Like we have array of integers, array of pointers etc, we can also have array of structure variables. And to use the array of structure variables efficiently, we

use **pointers of structure type**. We can also have pointer to a single structure variable, but it is mostly used when we are dealing with array of structure variables.

```c
#include <stdio.h>

struct Book
{
    char name[10];
    int price;
}

int main()
{
    struct Book a;      //Single structure variable
    struct Book* ptr;   //Pointer of Structure type
    ptr = &a;

    struct Book b[10];  //Array of structure variables
    struct Book* p;     //Pointer of Structure type
    p = &b;

    return 0;
}
```



Accessing Structure Members with Pointer

To access members of structure using the structure variable, we used the dot `.` operator. But when we have a pointer of structure type, we use arrow `->` to access structure members.

```c
#include <stdio.h>

struct my_structure {
    char name[20];
    int number;
    int rank;
};

int main()
{
    struct my_structure variable = {"StudyTonight", 35, 1};

    struct my_structure *ptr;
    ptr = &variable;

    printf("NAME: %s\n", ptr->name);
    printf("NUMBER: %d\n", ptr->number);
    printf("RANK: %d", ptr->rank);

    return 0;
}
```

```
NAME: StudyTonight
NUMBER: 35
RANK: 1
```

Pointers as Function Argument

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

```c
#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;
}
```

```
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.

```c
#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;
}
```

```
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.

Pointer to functions

It is possible to declare a pointer pointing to a function which can then be used as an argument in another function. A pointer to a function is declared as follows,

```
type (*pointer-name)(parameter);
```

Here is an example :

```
int (*sum)();   //legal declaration of pointer to function
int *sum();     //This is not a declaration of pointer to function
```

A function pointer can point to a specific function when it is assigned the name of that function.

```
int sum(int, int);
int (*s)(int, int);
s = sum;
```

Here `s` is a pointer to a function `sum`. Now `sum` can be called using function pointer `s` along with providing the required argument values.

```
s (10, 20);
```

Example of Pointer to Function

```
#include <stdio.h>


int sum(int x, int y)
```

```
{
    return x+y;
}

int main( )
{
    int (*fp)(int, int);
    fp = sum;
    int s = fp(10, 15);
    printf("Sum is %d", s);

    return 0;
}
```

```
25
```

Complicated Function Pointer example

You will find a lot of complex function pointer examples around, lets see one such example and try to understand it.

```
void *(*foo) (int*);
```

It appears complex but it is very simple. In this case `(*foo)` is a pointer to the function, whose argument is of `int*` type and return type is `void*`.

Pointer Arithmetic

If you want to have complete knowledge of pointers, pointer arithmetic is very important to understand. In this topic we will study how the memory addresses change when you increment a pointer.

---

16 bit Machine (Turbo C)

In a 16 bit machine, size of all types of pointer, be it `int*`, `float*`, `char*` or `double*` is always **2 bytes**. But when we perform any arithmetic function like increment on a pointer, changes occur as per the size of their primitive data type.

Size of datatypes on 16-bit Machine:

Type	Size (in bytes)
int or signed int	2
char	1
long	4
float	4
double	8
long double	10

Examples for Pointer Arithmetic

Now lets take a few examples and understand this more clearly.

```
int* i;
i++;
```

In the above case, pointer will be of 2 bytes. And when we increment it, it will increment by 2 bytes because `int` is also of 2 bytes.

```
float* i;
i++;
```

In this case, size of pointer is still 2 bytes initially. But now, when we increment it, it will increment by 4 bytes because `float` datatype is of 4 bytes.

```
double* i;
i++;
```

Similarly, in this case, size of pointer is still 2 bytes. But now, when we increment it, it will increment by 8 bytes because its data type is `double`.

---

32 bit Machine (Visual Basic C++)

The concept of pointer arithmetic remains exact same, but the size of pointer and various datatypes is different in a 32 bit machine. Pointer in 32 bit machine is of **4 bytes**.

And, following is a table for **Size of datatypes on 32-bit Machine :**

Type	Size (in bytes)
int or signed int	4
char	2
long	8
float	8
double	16

Note: We cannot add two pointers. This is because pointers contain addresses, adding two addresses makes no sense, because you have no idea what it would point to.

But we can subtract two pointers. This is because difference between two pointers gives the number of elements of its data type that can be stored between the two pointers.

Program for pointer arithmetic(32-bit machine)

```c
#include <stdio.h>

int main()
{
    int m = 5, n = 10, o = 0;

    int *p1;
    int *p2;
    int *p3;

    p1 = &m;    //printing the address of m
    p2 = &n;    //printing the address of n

    printf("p1 = %d\n", p1);
    printf("p2 = %d\n", p2);

    o = *p1+*p2;
    printf("*p1+*p2 = %d\n", o);//point 1

    p3 = p1-p2;
    printf("p1 - p2 = %d\n", p3); //point 2

    p1++;
    printf("p1++ = %d\n", p1); //point 3

    p2--;
    printf("p2-- = %d\n", p2); //point 4

    //Below line will give ERROR
    printf("p1+p2 = %d\n", p1+p2); //point 5

    return 0;
```

```
}
```

```
p1 = 2680016
p2 = 2680012
*p1+*p2 = 15
p1-p2 = 1
p1++ = 2680020
p2-- = 2680008
```

Explanation of the above program:

1. **Point 1:** Here, `*` means 'value at the given address'. Thus, it adds the value of m and n which is 15.
2. **Point 2:** It subtracts the addresses of the two variables and then divides it by the size of the pointer datatype (here integer, which has size of 4 bytes) which gives us the number of elements of integer data type that can be stored within it.
3. **Point 3:** It increments the address stored by the pointer by the size of its datatype(here 4).
4. **Point 4:** It decrements the address stored by the pointer by the size of its datatype(here 4).
5. **Point 5:** Addition of two pointers is not allowed.

Unit-5

Functions

Functions in C

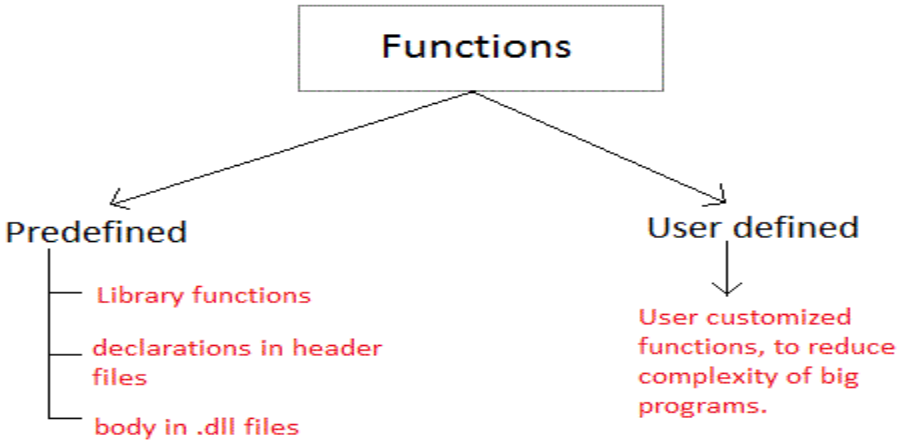
A **function** is a block of code that performs a particular task.

There are many situations where we might need to write same line of code for more than once in a program. This may lead to unnecessary repetition of code, bugs and even becomes boring for the programmer. So, C language provides an approach in which you can declare and define a group of statements once in the form of a function and it can be called and used whenever required.

These functions defined by the user are also know as **User-defined Functions**

C functions can be classified into two categories,

1. **Library functions**
2. **User-defined functions**



Library functions are those functions which are already defined in C library, example `printf()`, `scanf()`, `strcat()` etc. You just need to include appropriate header files to use these functions. These are already declared and defined in C libraries.

A **User-defined functions** on the other hand, are those functions which are defined by the user at the time of writing program. These functions are made for code reusability and for saving time and space.

Benefits of Using Functions

1. It provides modularity to your program's structure.
2. It makes your code reusable. You just have to call the function by its name to use it, wherever required.
3. In case of large programs with thousands of code lines, debugging and editing becomes easier if you use functions.
4. It makes the program more readable and easy to understand.

Function Declaration

General syntax for function declaration is,

```
returntype functionName(type1 parameter1, type2 parameter2,...);
```

Like any variable or an array, a function must also be declared before its used. Function declaration informs the compiler about the function name, parameters is accept, and its return type. The actual body of the function can be defined separately. It's also called as **Function Prototyping**. Function declaration consists of 4 parts.

- returntype
- function name
- parameter list
- terminating semicolon

returntype

When a function is declared to perform some sort of calculation or any operation and is expected to provide with some result at the end, in such cases, a `return` statement is added at the end of function body. Return type specifies the type of value(`int`, `float`, `char`, `double`) that function is expected to return to the program which called the function.

Note: In case your function doesn't return any value, the return type would be `void`.

---

functionName

Function name is an identifier and it specifies the name of the function. The function name is any valid C identifier and therefore must follow the same naming rules like other variables in C language.

---

parameter list

The parameter list declares the type and number of arguments that the function expects when it is called. Also, the parameters in the parameter list receives the argument values when the function is called. They are often referred as **formal parameters**.

Time for an Example

Let's write a simple program with a `main()` function, and a user defined function to multiply two numbers, which will be called from the `main()` function.

```c
#include<stdio.h>

int multiply(int a, int b);     // function declaration

int main()
{
    int i, j, result;
    printf("Please enter 2 numbers you want to multiply...");
    scanf("%d%d", &i, &j);

    result = multiply(i, j);        // function call
    printf("The result of muliplication is: %d", result);

    return 0;
}

int multiply(int a, int b)
{
    return (a*b);       // function defintion, this can be done in one line
}
```

Function definition Syntax

Just like in the example above, the general syntax of function definition is,

```c
returntype functionName(type1 parameter1, type2 parameter2,...)
{
    // function body goes here
}
```

The first line *returntype* **functionName(type1 parameter1, type2 parameter2,...)** is known as **function header** and the statement(s) within curly braces is called **function body**.

Note: While defining a function, there is no semicolon(`;`) after the parenthesis in the function header, unlike while declaring the function or calling the function.

Function body

The function body contains the declarations and the statements(algorithm) necessary for performing the required task. The body is enclosed within curly braces `{ ... }` and consists of three parts.

- **local** variable declaration(if required).
- **function statements** to perform the task inside the function.
- a **return** statement to return the result evaluated by the function(if return type is `void`, then no return statement is required).

Calling a function

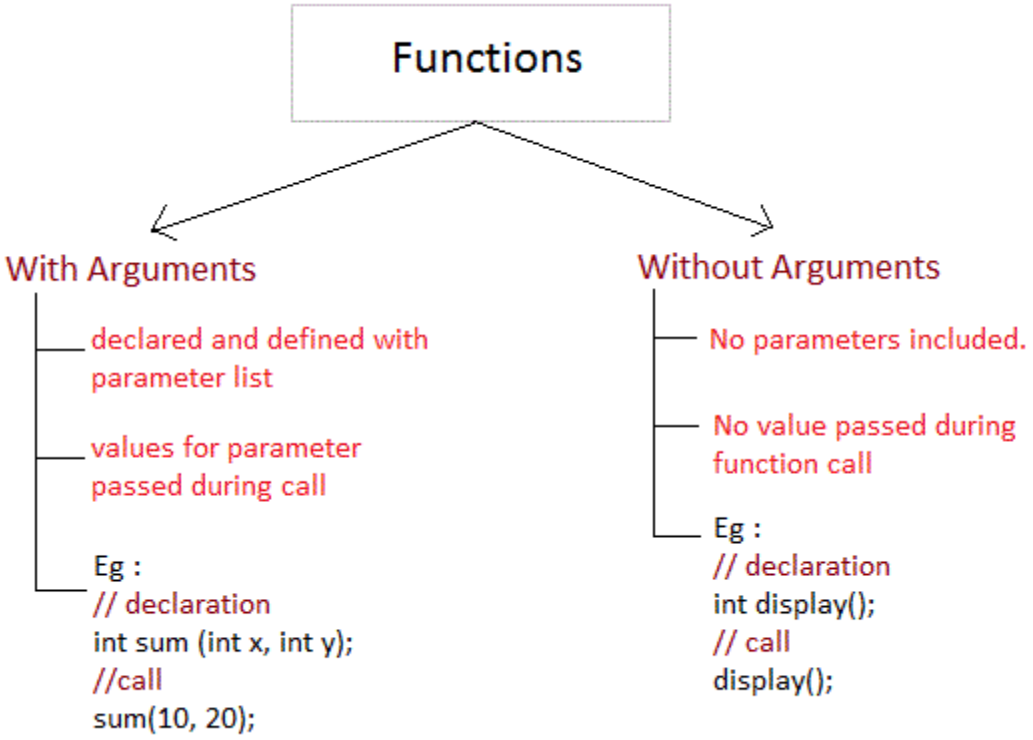
When a function is called, control of the program gets transferred to the function.

```
functionName(argument1, argument2,...);
```

In the example above, the statement `multiply(i, j);` inside the `main()` function is function call.

Passing Arguments to a function

Arguments are the values specified during the function call, for which the formal parameters are declared while defining the function.



It is possible to have a function with parameters but no return type. It is not necessary, that if a function accepts parameter(s), it must return a result too.

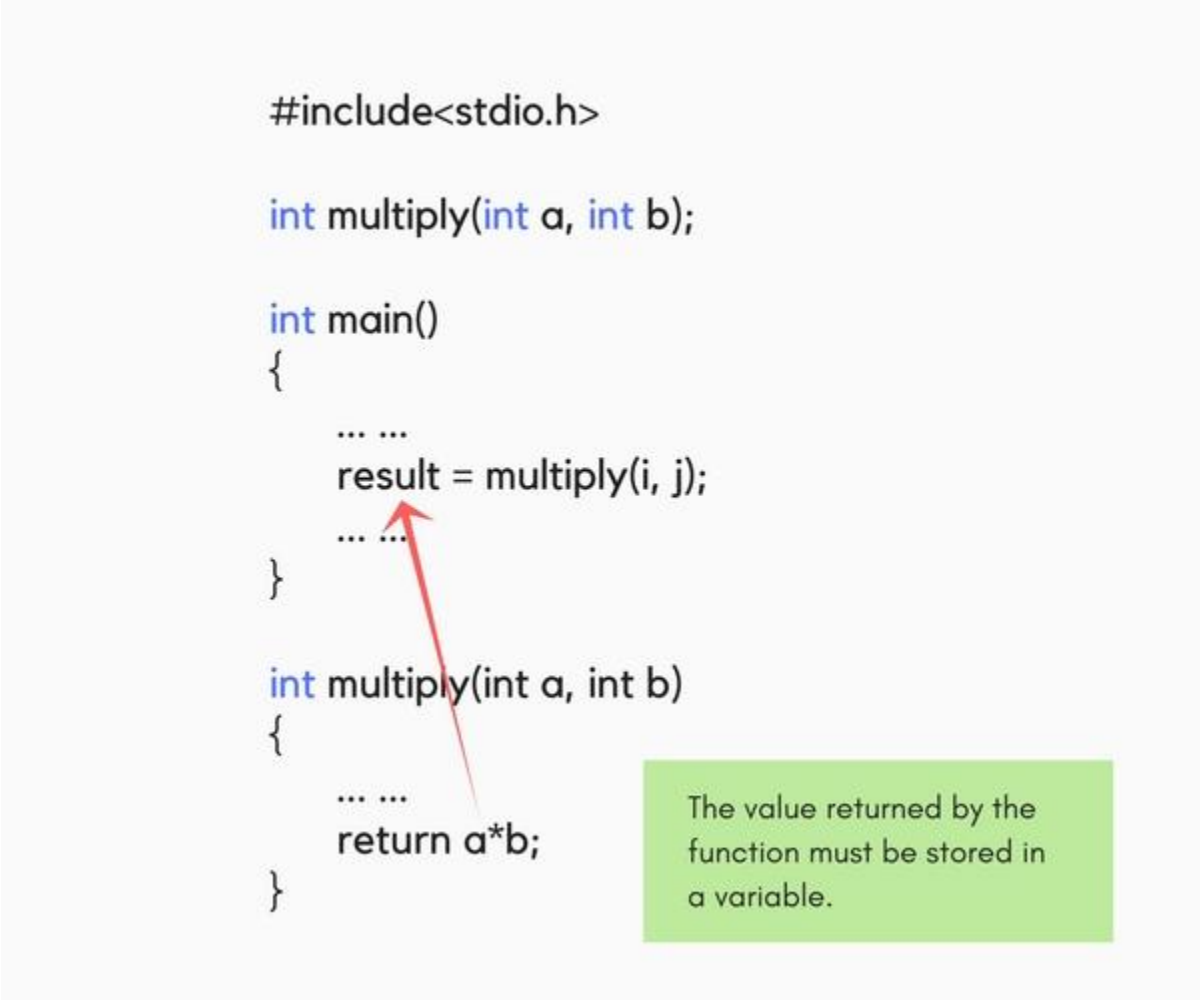
```
#include<stdio.h>

int multiply(int a, int b);

int main()
{
    ... ...
    result = multiply(i, j);
    ... ...
}

int multiply(int a, int b)
{
    ... ...
}
```

providing arguments while calling function

While declaring the function, we have declared two parameters `a` and `b` of type `int`. Therefore, while calling that function, we need to pass two arguments, else we will get compilation error. And the two arguments passed should be received in the function definition, which means that the function header in the function definition should have the two parameters to hold the argument values. These received arguments are also known as **formal parameters**. The name of the variables while declaring, calling and defining a function can be different.

Returning a value from function

A function may or may not return a result. But if it does, we must use the `return` statement to output the result. `return` statement also ends the function execution, hence it must be the last statement of any function. If you write any statement after the `return` statement, it won't be executed.



The datatype of the value returned using the `return` statement should be same as the return type mentioned at function declaration and definition. If any of it mismatches, you will get compilation error.

In the next tutorial, we will learn about the different types of user defined functions in C language and the concept of Nesting of functions which is used in recursion.

Type of User-defined Functions in C

There can be 4 different types of user-defined functions, they are:

1. Function with no arguments and no return value
2. Function with no arguments and a return value
3. Function with arguments and no return value
4. Function with arguments and a return value

Below, we will discuss about all these types, along with program examples.

Function with no arguments and no return value

Such functions can either be used to display information or they are completely dependent on user inputs.

Below is an example of a function, which takes 2 numbers as input from user, and display which is the greater number.

```
#include<stdio.h>

void greatNum();       // function declaration

int main()
{
    greatNum();        // function call
    return 0;
}

void greatNum()        // function definition
{
    int i, j;
```

```
    printf("Enter 2 numbers that you want to compare...");
    scanf("%d%d", &i, &j);
    if(i > j) {
        printf("The greater number is: %d", i);
    }
    else {
        printf("The greater number is: %d", j);
    }
}
```

Function with no arguments and a return value

We have modified the above example to make the function `greatNum()` return the number which is greater amongst the 2 input numbers.

```
#include<stdio.h>

int greatNum();       // function declaration

int main()
{
    int result;
    result = greatNum();        // function call
    printf("The greater number is: %d", result);
    return 0;
}

int greatNum()        // function definition
{
    int i, j, greaterNum;
    printf("Enter 2 numbers that you want to compare...");
    scanf("%d%d", &i, &j);
    if(i > j) {
        greaterNum = i;
    }
    else {
```

```
        greaterNum = j;
    }
    // returning the result
    return greaterNum;
}
```

Function with arguments and no return value

We are using the same function as example again and again, to demonstrate that to solve a problem there can be many different ways.

This time, we have modified the above example to make the function `greatNum()` take two `int`values as arguments, but it will not be returning anything.

```
#include<stdio.h>

void greatNum(int a, int b);       // function declaration

int main()
{
    int i, j;
    printf("Enter 2 numbers that you want to compare...");
    scanf("%d%d", &i, &j);
    greatNum(i, j);        // function call
    return 0;
}

void greatNum(int x, int y)        // function definition
{
    if(x > y) {
        printf("The greater number is: %d", x);
    }
    else {
        printf("The greater number is: %d", y);
    }
}
```

Function with arguments and a return value

This is the best type, as this makes the function completely independent of inputs and outputs, and only the logic is defined inside the function body.

```c
#include<stdio.h>

int greatNum(int a, int b);       // function declaration

int main()
{
    int i, j, result;
    printf("Enter 2 numbers that you want to compare...");
    scanf("%d%d", &i, &j);
    result = greatNum(i, j); // function call
    printf("The greater number is: %d", result);
    return 0;
}

int greatNum(int x, int y)        // function definition
{
    if(x > y) {
        return x;
    }
    else {
        return y;
    }
}
```

Nesting of Functions

C language also allows nesting of functions i.e to use/call one function inside another function's body. We must be careful while using nested functions, because it may lead to infinite nesting.

```
function1()
{
    // function1 body here

    function2();

    // function1 body here
}
```

If function2() also has a call for function1() inside it, then in that case, it will lead to an infinite nesting. They will keep calling each other and the program will never terminate.

Not able to understand? Lets consider that inside the `main()` function, function1() is called and its execution starts, then inside function1(), we have a call for function2(), so the control of program will go to the function2(). But as function2() also has a call to function1() in its body, it will call function1(), which will again call function2(), and this will go on for infinite times, until you forcefully exit from program execution.

What is Recursion?

Recursion is a special way of nesting functions, where a function calls itself inside it. We must have certain conditions in the function to break out of the recursion, otherwise recursion will occur infinite times.

```
function1()
{
    // function1 body
    function1();
    // function1 body
}
```

Example: Factorial of a number using Recursion

```
#include<stdio.h>

int factorial(int x);       //declaring the function
```

```c
void main()
{
    int a, b;

    printf("Enter a number...");
    scanf("%d", &a);
    b = factorial(a);       //calling the function named factorial
    printf("%d", b);
}

int factorial(int x) //defining the function
{
    int r = 1;
    if(x == 1)
        return 1;
    else
        r = x*factorial(x-1);       //recursion, since the function calls itself

    return r;
}
```

Similarly, there are many more applications of recursion in C language. Go to the programs section, to find out more programs using recursion.

Types of Function calls in C

Functions are called by their names, we all know that, then what is this tutorial for? Well if the function does not have any arguments, then to call a function you can directly use its name. But for functions with arguments, we can call a function in two different ways, based on how we specify the arguments, and these two ways are:

1. Call by Value
2. Call by Reference

Call by Value

Calling a function by value means, we pass the values of the arguments which are stored or copied into the formal parameters of the function. Hence, the original values are unchanged only the parameters inside the function changes.

```
#include<stdio.h>

void calc(int x);

int main()
{
    int x = 10;
    calc(x);
    // this will print the value of 'x'
    printf("\nvalue of x in main is %d", x);
    return 0;
}

void calc(int x)
{
    // changing the value of 'x'
    x = x + 10 ;
    printf("value of x in calc function is %d ", x);
}
```

```
value of x in calc function is 20
value of x in main is 10
```

In this case, the actual variable `x` is not changed. This is because we are passing the argument by value, hence a copy of `x` is passed to the function, which is updated during function execution, and that copied value in the function is destroyed when the function ends(goes out of scope). So the variable `x` inside the `main()` function is never changed and hence, still holds a value of `10`.

But we can change this program to let the function modify the original `x` variable, by making the function `calc()` return a value, and storing that value in x.

```
#include<stdio.h>

int calc(int x);
```

```c
int main()
{
    int x = 10;
    x = calc(x);
    printf("value of x is %d", x);
    return 0;
}

int calc(int x)
{
    x = x + 10 ;
    return x;
}
```

```
value of x is 20
```

Call by Reference

In call by reference we pass the address(reference) of a variable as argument to any function. When we pass the address of any variable as argument, then the function will have access to our variable, as it now knows where it is stored and hence can easily update its value.

In this case the formal parameter can be taken as a **reference** or a **pointer**(don't worry about pointers, we will soon learn about them), in both the cases they will change the values of the original variable.

```c
#include<stdio.h>

void calc(int *p);      // functin taking pointer as argument

int main()
{
    int x = 10;
    calc(&x);       // passing address of 'x' as argument
    printf("value of x is %d", x);
    return(0);
```

```
}

void calc(int *p)       //receiving the address in a reference pointer variable
{
    /*
        changing the value directly that is
        stored at the address passed
    */
    *p = *p + 10;
}
```

```
value of x is 20
```

NOTE: If you do not have any prior knowledge of pointers, do study Pointers first. Or just go over this topic and come back again to revise this, once you have learned about pointers.

How to pass Array to a Function

Whenever we need to pass a list of elements as argument to any function in C language, it is prefered to do so using an `array`. But how can we pass an array as argument to a function? Let's see how its done.

Declaring Function with array as a parameter

There are two possible ways to do so, one by using call by value and other by using call by reference.

1. We can either have an array as a parameter.

   ```
   int sum (int arr[]);
   ```

2. Or, we can have a pointer in the parameter list, to hold the base address of our array.

   ```
   int sum (int* ptr);
   ```

   We will study the second way in details later when we will study pointers.

Returning an Array from a function

We don't return an array from functions, rather we return a pointer holding the base address of the array to be returned. But we must, make sure that the array exists after the function ends i.e. the array is not local to the function.

```
int* sum (int x[])
{
    // statements
    return x ;
}
```

We will discuss about this when we will study pointers with arrays.

Passing arrays as parameter to function

Now let's see a few examples where we will pass a single array element as argument to a function, a one dimensional array to a function and a multidimensional array to a function.

Passing a single array element to a function

Let's write a very simple program, where we will declare and define an array of integers in our `main()`function and pass one of the array element to a function, which will just print the value of the element.

```
#include<stdio.h>

void giveMeArray(int a);

int main()
{
    int myArray[] = { 2, 3, 4 };
    giveMeArray(myArray[2]);        //Passing array element myArray[2] only.
    return 0;
```

```
}

void giveMeArray(int a)
{
    printf("%d", a);
}
```

```
4
```

Passing a complete One-dimensional array to a function

To understand how this is done, let's write a function to find out average of all the elements of the array and print it.

We will only send in the name of the array as argument, which is nothing but the address of the starting element of the array, or we can say the starting memory address.

```
#include<stdio.h>

float findAverage(int marks[]);

int main()
{
    float avg;
    int marks[] = {99, 90, 96, 93, 95};
    avg = findAverage(marks);       // name of the array is passed as argument.
    printf("Average marks = %.1f", avg);
    return 0;
}

float findAverage(int marks[])
{
    int i, sum = 0;
    float avg;
    for (i = 0; i <= 4; i++) {
        sum += age[i];
```

```
    }
    avg = (sum / 5);
    return avg;
}
```

```
94.6
```

Passing a Multi-dimensional array to a function

Here again, we will only pass the name of the array as argument.

```
#include<stdio.h>

void displayArray(int arr[3][3]);

int main()
{
    int arr[3][3], i, j;
    printf("Please enter 9 numbers for the array: \n");
    for (i = 0; i < 3; ++i)
    {
        for (j = 0; j < 3; ++j)
        {
            scanf("%d", &arr[i][j]);
        }
    }
    // passing the array as argument
    displayArray(arr);
    return 0;
}

void displayArray(int arr[3][3])
{
    int i, j;
    printf("The complete array is: \n");
    for (i = 0; i < 3; ++i)
```

```
    {
        // getting cursor to new line
        printf("\n");
        for (j = 0; j < 3; ++j)
        {
            // \t is used to provide tab space
            printf("%d\t", arr[i][j]);
        }
    }
}
```

```
Please enter 9 numbers for the array:
1
2
3
4
5
6
7
8
9
The complete array is:
1 2 3
4 5 6
7 8 9
```

Unit-6

Arrays and Strings

Arrays in C

In C language, `arrays` are reffered to as structured data types. An array is defined as **finite ordered collection of homogenous** data, stored in contiguous memory locations.

Here the words,

- **finite** *means* data range must be defined.
- **ordered** *means* data must be stored in continuous memory addresses.
- **homogenous** *means* data must be of similar data type.

Example where arrays are used,

- to store list of Employee or Student names,
- to store marks of students,
- or to store list of numbers or characters etc.

Since arrays provide an easy way to represent data, it is classified amongst the data structures in C. Other data structures in c are **structure**, **lists**, **queues**, **trees** etc. Array can be used to represent not only simple list of data but also table of data in two or three dimensions.

Declaring an Array

Like any other variable, arrays must be declared before they are used. General form of array declaration is,

```
data-type variable-name[size];


/* Example of array declaration */


int arr[10];
```

Here `int` is the data type, `arr` is the name of the array and 10 is the size of array. It means array `arr` can only contain 10 elements of `int` type.

Index of an array starts from `0` to **size-1** i.e first element of `arr` array will be stored at `arr[0]`address and the last element will occupy `arr[9]`.

Initialization of an Array

After an array is declared it must be initialized. Otherwise, it will contain **garbage** value(any random value). An array can be initialized at either **compile time** or at **runtime**.

Compile time Array initialization

Compile time initialization of array elements is same as ordinary variable initialization. The general form of initialization of array is,

```
data-type array-name[size] = { list of values };


/* Here are a few examples */
int marks[4]={ 67, 87, 56, 77 };    // integer array initialization

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

int marks[4]={ 67, 87, 56, 77, 59 };    // Compile time error
```

One important thing to remember is that when you will give more initializer(array elements) than the declared array size than the **compiler** will give an error.

```
#include<stdio.h>


void main()
{
    int i;
    int arr[] = {2, 3, 4};      // Compile time array initialization
    for(i = 0 ; i < 3 ; i++)
    {
        printf("%d\t",arr[i]);
    }
}
```

```
2 3 4
```

Runtime Array initialization

An array can also be initialized at runtime using `scanf()` function. This approach is usually used for initializing large arrays, or to initialize arrays with user specified values. Example,

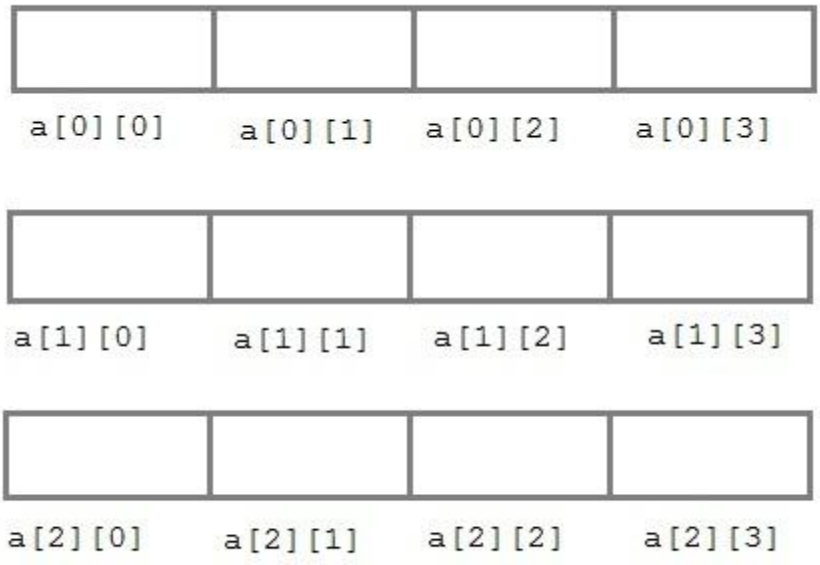
```c
#include<stdio.h>


void main()
{
    int arr[4];
    int i, j;
    printf("Enter array element");
    for(i = 0; i < 4; i++)
    {
        scanf("%d", &arr[i]);    //Run time array initialization
    }
    for(j = 0; j < 4; j++)
    {
        printf("%d\n", arr[j]);
    }
}
```

Two dimensional Arrays

C language supports multidimensional arrays also. The simplest form of a multidimensional array is the two-dimensional array. Both the row's and column's index begins from `0`.

Two-dimensional arrays are declared as follows,

```c
data-type array-name[row-size][column-size]


/* Example */
int a[3][4];
```



An array can also be declared and initialized together. For example,

```
int arr[][3] = {
    {0,0,0},
    {1,1,1}
};
```

Note: We have not assigned any row value to our array in the above example. It means we can initialize any number of rows. But, we must always specify number of columns, else it will give a compile time error. Here, a `2*3` multi-dimensional matrix is created.

Runtime initialization of a two dimensional Array

```
#include<stdio.h>


void main()
{
    int arr[3][4];
    int i, j, k;
    printf("Enter array element");
    for(i = 0; i < 3;i++)
```

```
    {
        for(j = 0; j < 4; j++)
        {
            scanf("%d", &arr[i][j]);
        }
    }
    for(i = 0; i < 3; i++)
    {
        for(j = 0; j < 4; j++)
        {
            printf("%d", arr[i][j]);
        }
    }
}
```

String and Character Array

String is a sequence of characters that is treated as a single data item and terminated by null character `'\0'`. Remember that C language does not support strings as a data type. A **string** is actually one-dimensional array of characters in C language. These are often used to create meaningful and readable programs.

For example: The string "hello world" contains 12 characters including `'\0'` character which is automatically added by the compiler at the end of the string.

Declaring and Initializing a string variables

There are different ways to initialize a character array variable.

```
char name[13] = "StudyTonight";       // valid character array initialization


char name[10] = {'L','e','s','s','o','n','s','\0'};     // valid initialization
```

Remember that when you initialize a character array by listing all of its characters separately then you must supply the `'\0'` character explicitly.

Some examples of illegal initialization of character array are,

```
char ch[3] = "hell";    // Illegal
```

```
char str[4];
str = "hell";   // Illegal
```

String Input and Output

Input function `scanf()` can be used with **%s** format specifier to read a string input from the terminal. But there is one problem with `scanf()` function, it terminates its input on the first white space it encounters. Therefore if you try to read an input string "Hello World" using `scanf()` function, it will only read **Hello** and terminate after encountering white spaces.

However, C supports a format specification known as the **edit set conversion code %[..]** that can be used to read a line containing a variety of characters, including white spaces.

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


void main()
{
    char str[20];
    printf("Enter a string");
    scanf("%[^\n]", &str);  //scanning the whole string, including the white spaces
    printf("%s", str);
}
```

Another method to read character string with white spaces from terminal is by using the `gets()`function.

```
char text[20];
gets(text);
printf("%s", text);
```

String Handling Functions

C language supports a large number of string handling functions that can be used to carry out many of the string manipulations. These functions are packaged in **string.h** library. Hence, you must include **string.h** header file in your programs to use these functions.

The following are the most commonly used string handling functions.

Method	Description
`strcat()`	It is used to concatenate(combine) two strings
`strlen()`	It is used to show length of a string
`strrev()`	It is used to show reverse of a string
`strcpy()`	Copies one string into another
`strcmp()`	It is used to compare two string

---

strcat() function

```
strcat("hello", "world");
```

`strcat()` function will add the string **"world"** to **"hello"** i.e it will ouput helloworld.

---

strlen() function

`strlen()` function will return the length of the string passed to it.

```
int j;
j = strlen("studytonight");
printf("%d",j);
```

strcmp() function

`strcmp()` function will return the ASCII difference between first unmatching character of two strings.

```
int j;
j = strcmp("study", "tonight");
```

```
printf("%d",j);
```

```
-1
```

---

strcpy() function

It copies the second string argument to the first string argument.

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

int main()
{
    char s1[50];
    char s2[50];

    strcpy(s1, "StudyTonight");     //copies "studytonight" to string s1
    strcpy(s2, s1);     //copies string s1 to string s2

    printf("%s\n", s2);

    return(0);
}
```

```
StudyTonight
```

---

strrev() function

It is used to reverse the given string expression.

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

```c
int main()
{
    char s1[50];

    printf("Enter your string: ");
    gets(s1);
    printf("\nYour reverse string is: %s",strrev(s1));
    return(0);
}
```

```
Enter your string: studytonight
Your reverse string is: thginotyduts
```

Unit-6

Structures and Unions

Introduction to Structure

Structure is a user-defined datatype in C language which allows us to combine data of different types together. Structure helps to construct a complex data type which is more meaningful. It is somewhat similar to an Array, but an array holds data of similar type only. But structure on the other hand, can store data of any type, which is practical more useful.

For example: If I have to write a program to store Student information, which will have Student's name, age, branch, permanent address, father's name etc, which included string values, integer values etc, how can I use arrays for this problem, I will require something which can hold data of different types together.

In structure, data is stored in form of **records**.

Defining a structure

`struct` keyword is used to define a structure. `struct` defines a new data type which is a collection of primary and derived datatypes.

Syntax:

```
struct [structure_tag]
{
    //member variable 1
    //member variable 2
    //member variable 3
    ...
}[structure_variables];
```

As you can see in the syntax above, we start with the `struct` keyword, then it's optional to provide your structure a name, we suggest you to give it a name, then inside the curly braces, we have to mention all the member variables, which are nothing but normal C language variables of different types like `int`, `float`, `array` etc.

After the closing curly brace, we can specify one or more structure variables, again this is optional.

Note: The closing curly brace in the structure type declaration must be followed by a semicolon(`;`).

Example of Structure

```
struct Student
{
    char name[25];
    int age;
    char branch[10];
    // F for female and M for male
    char gender;
};
```

Here `struct Student` declares a structure to hold the details of a student which consists of 4 data fields, namely `name`, `age`, `branch` and `gender`. These fields are called **structure elements or members**.

Each member can have different datatype, like in this case, `name` is an array of `char` type and `age` is of `int` type etc. **Student** is the name of the structure and is called as the **structure tag**.

Declaring Structure Variables

It is possible to declare variables of a **structure**, either along with structure definition or after the structure is defined. **Structure** variable declaration is similar to the declaration of any normal variable of any other datatype. Structure variables can be declared in following two ways:

---

1) Declaring Structure variables separately

```
struct Student
{
    char name[25];
    int age;
    char branch[10];
    //F for female and M for male
    char gender;
};

struct Student S1, S2;      //declaring variables of struct Student
```

---

2) Declaring Structure variables with structure definition

```
struct Student
{
    char name[25];
    int age;
    char branch[10];
    //F for female and M for male
    char gender;
}S1, S2;
```

Here `S1` and `S2` are variables of structure `Student`. However this approach is not much recommended.

Accessing Structure Members

Structure members can be accessed and assigned values in a number of ways. Structure members have no meaning individually without the structure. In order to assign a value to any structure member, the member name must be linked with the **structure** variable using a dot `.` operator also called **period** or **member access** operator.

For example:

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

struct Student
{
    char name[25];
    int age;
    char branch[10];
    //F for female and M for male
    char gender;
};

int main()
{
    struct Student s1;
```

```
    /*
        s1 is a variable of Student type and
        age is a member of Student
    */
    s1.age = 18;
    /*
        using string function to add name
    */
    strcpy(s1.name, "Viraaj");
    /*
        displaying the stored values
    */
    printf("Name of Student 1: %s\n", s1.name);
    printf("Age of Student 1: %d\n", s1.age);

    return 0;
}
```

```
Name of Student 1: Viraaj
Age of Student 1: 18
```

We can also use `scanf()` to give values to structure members through terminal.

```
scanf(" %s ", s1.name);
scanf(" %d ", &s1.age);
```

Structure Initialization

Like a variable of any other datatype, structure variable can also be initialized at compile time.

```
struct Patient
{
    float height;
    int weight;
    int age;
```

```
};

struct Patient p1 = { 180.75 , 73, 23 };    //initialization
```

or,

```
struct Patient p1;
p1.height = 180.75;     //initialization of each member separately
p1.weight = 73;
p1.age = 23;
```

Array of Structure

We can also declare an array of **structure** variables. in which each element of the array will represent a **structure** variable. **Example :** `struct employee emp[5];`

The below program defines an array `emp` of size 5. Each element of the array `emp` is of type `Employee`.

```
#include<stdio.h>

struct Employee
{
    char ename[10];
    int sal;
};

struct Employee emp[5];
int i, j;
void ask()
{
    for(i = 0; i < 3; i++)
    {
        printf("\nEnter %dst Employee record:\n", i+1);
        printf("\nEmployee name:\t");
        scanf("%s", emp[i].ename);
        printf("\nEnter Salary:\t");
        scanf("%d", &emp[i].sal);
    }
```

```
    printf("\nDisplaying Employee record:\n");
    for(i = 0; i < 3; i++)
    {
        printf("\nEmployee name is %s", emp[i].ename);
        printf("\nSlary is %d", emp[i].sal);
    }
}
void main()
{
    ask();
}
```

Nested Structures

Nesting of structures, is also permitted in C language. Nested structures means, that one structure has another stucture as member variable.

Example:

```
struct Student
{
    char[30] name;
    int age;
    /* here Address is a structure */
    struct Address
    {
        char[50] locality;
        char[50] city;
        int pincode;
    }addr;
};
```

Structure as Function Arguments

We can pass a structure as a function argument just like we pass any other variable or an array as a function argument.

Example:

```
#include<stdio.h>

struct Student
{
    char name[10];
    int roll;
};

void show(struct Student st);

void main()
{
    struct Student std;
    printf("\nEnter Student record:\n");
    printf("\nStudent name:\t");
    scanf("%s", std.name);
    printf("\nEnter Student rollno.:\t");
    scanf("%d", &std.roll);
    show(std);
}

void show(struct Student st)
{
    printf("\nstudent name is %s", st.name);
    printf("\nroll is %d", st.roll);
}
```

typedef in C

`typedef` is a keyword used in C language to assign alternative names to existing datatypes. Its mostly used with user defined datatypes, when names of the datatypes become slightly complicated to use in programs. Following is the general syntax for using `typedef`,

```
typedef <existing_name> <alias_name>
```

Lets take an example and see how `typedef` actually works.

```
typedef unsigned long ulong;
```

The above statement define a term `ulong` for an `unsigned long` datatype. Now this `ulong` identifier can be used to define `unsigned long` type variables.

```
ulong i, j;
```

Application of typedef

`typedef` can be used to give a name to user defined data type as well. Lets see its use with structures.

```
typedef struct
{
    type member1;
    type member2;
    type member3;
} type_name;
```

Here **type_name** represents the stucture definition associated with it. Now this **type_name** can be used to declare a variable of this stucture type.

```
type_name t1, t2;
```

Example of Structure definition using typedef

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

typedef struct employee
{
    char name[50];
    int salary;
}emp;

void main( )
{
    emp e1;
    printf("\nEnter Employee record:\n");
```

```
    printf("\nEmployee name:\t");
    scanf("%s", e1.name);
    printf("\nEnter Employee salary: \t");
    scanf("%d", &e1.salary);
    printf("\nstudent name is %s", e1.name);
    printf("\nroll is %d", e1.salary);
}
```

`typedef` and Pointers

`typedef` can be used to give an alias name to pointers also. Here we have a case in which use of `typedef` is beneficial during pointer declaration.

In Pointers `*` binds to the right and not on the left.

```
int* x, y;
```

By this declaration statement, we are actually declaring `x` as a pointer of type `int`, whereas `y` will be declared as a plain `int` variable.

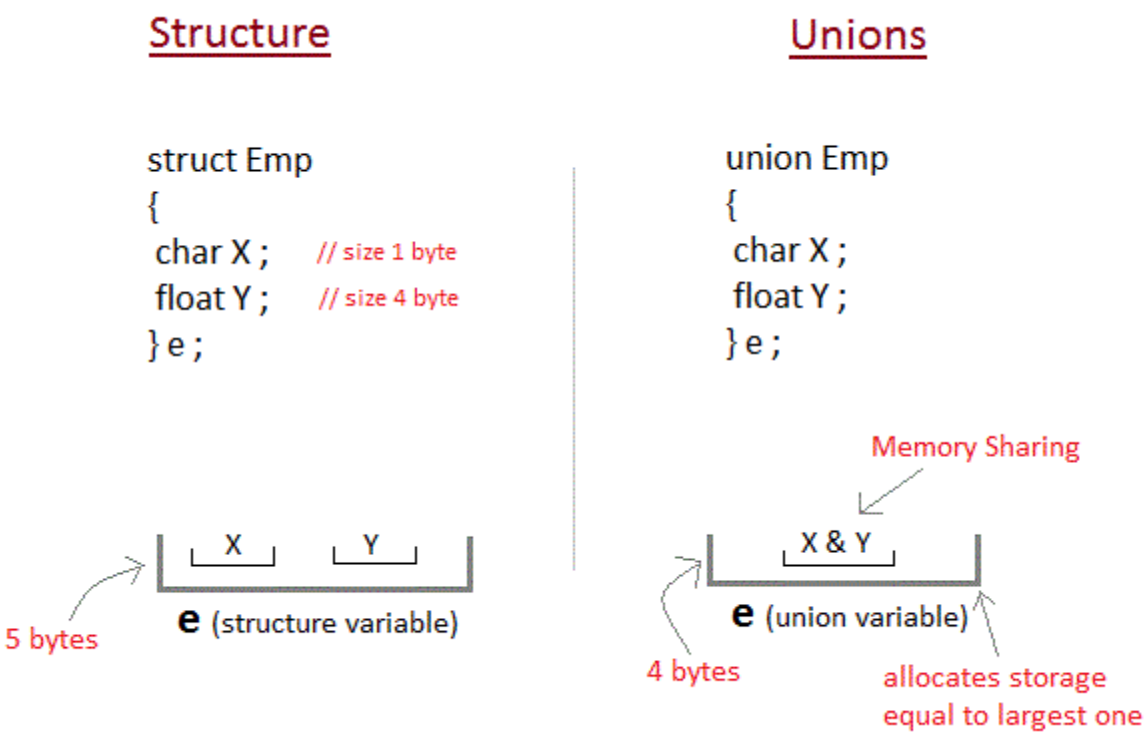
```
typedef int* IntPtr;
IntPtr x, y, z;
```

But if we use `typedef` like we have used in the example above, we can declare any number of pointers in a single statement.

NOTE: If you do not have any prior knowledge of pointers, do study Pointers first.

Unions in C Language

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.

Accessing a Union Member

Syntax for accessing any `union` member is similar to accessing structure members,

```
union test
{
    int a;
    float b;
    char c;
}t;

t.a;    //to access members of union t
t.b;
t.c;
```

Time for an Example

```

#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;
}
```

```
-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.