Unit-I

Introduction to programming

What is a Programming?

* Programming is a way to " instruct the computer to perform various tasks or to solve a problem"
* Instruct the computer: This basically means that you need to provide the computer with a set of instructions that are written in a language that the computer can understand.
* Perform various tasks: This basically means finding the solution to a problem. The task could be small and simple involving few instructions to obtain their solution or large and complex involving hundreds of instructions
* Hence, the "programming is a way to tell Computers to do a specific task”.

How to write programs?

* Before attempting to write program, one should learn problem solving to write a program, because one will not be able to solve any problem if he doesn’t know the steps involved in solving the problem.

Problem solving:

* In terms of programming "problem solving is the process of identifying a problem, developing an algorithm for the identified problem and finally implementing the algorithm to develop a computer program."
* Thus, programmer cannot write proper instruction to the computer unless programmer know how to solve the problem.

Approaches to Problem Solving

There are two approaches to Problem Solving:

1.Top-down approach

2. Bottom-up approach

Top-down Approach

* In this Approach we break the program or problem into smaller parts. This process is also called "decomposition."
* The top-down way of solving a program is the step-by-step process of breaking down the problem into smaller parts for organizing and solving the sole problem.
* The programming languages like COBOL, Fortran, C use the top-down approach to solve a problem in which the flow of control Is in downward direction.

Bottom-Up Approach

* This approach works exactly opposite to the top-down approach.
* In this approach we start working from the basic level of problem solving in smaller problems or modules and adding or integrating thew & them up together to achieve the complete solution.
* The programming languages like C++, C#, python use the bottom-up approach to solve a problem.

Structured Programming

* Structured programming is a technique developed to improve the reliability and clarity of programs. This programming is also known as "modular programming."
* The structured program mainly consists of three types of elements:

1) Sequence

2) Repetition

3) Selection

1). Sequence structure

* In sequence structure, Instructions are followed or executed one after another in Sequential order in which they appear.
* The flow of logic is from top to bottom.

2) Selection structure

* Selection structure is used for making decision. It is used for selecting a proper path out of the alternative paths in the Program logic.
* Selection structure may take the form as either if or, if else or if-else if structure.

3) Iterative or Repetition

* The iterative structure is used to produce loops when one or more Instructions are to be executed either a given number of times or fill a certain condition is met.
* The following are the three iterative structures that are used most

1. While loop
2. Do-while loop and
3. For loop.

Tools to design program

The tools that help in planning a program are:

1) Algorithm

2) Flowchart

3) Pseudocode.

Algorithms

* The Algorithm is a set of well-defined finite & Sequence of
* Instructions to solve a particular problem, where each instruction numbered.

Characteristics of algorithm

There are five important characteristics of an algorithm that should be considered while designing an algorithm for a problem

1. Input: An algorithm must have some finite number of input values, which are externally supplied.

2) Output: At the end of an algorithm, it must have at least one desirable output.

3) Definiteness (Now ambiguity): A perfect algorithm is defined as unambiguous, which means that its instructions should be clear and straight forward.

4) Finiteness: An algorithm must be terminated after a finite number of steps. Finiteness means that algorithm should have a limited no. Of instructions which are countable.

5) Effectiveness - Because each instruction is an algorithm affects the overall process, it should be adequate.

6)Language-independence: An algorithm must be language-independent, which means that its instructions can be implemented in any language and produce the same results.

Example

Algorithm to add 3 numbers and print their sum:

Step 1: Begin

Step 2: Declare 3 integer variables. Num1, num2 and num3

Steps 3: Take the input of three numbers, to be added in variables num1, num2 and num3 respectively

Step 4: Declare an integer variable sum to store the resultant sum of the 3 numbers

Steps 5: Add the 3 numbers and store the result in the variable sum

Step 6: Print the value of the variable sum

Step 7: END

Flowchart

* A flowchart is a type of diagram that represents a work flow or process
* A Flowchart is a pictorial representation of an algorithm, a step-by -step approach to solve a problem.
* A flowchart uses different shapes to denote different types of instructions.
* The actual Instructions are written within the shapes using clear and Concise statements
* There shapes are connected by directed lines to indicate the sequence which instructions are to be executed.
* Generally, an algorithm is first represented in the form of a flowchart and then the flowchart is expressed in Some programming language to prepare a program.
* The two main advantages of flowchart are
* One is not concerned with the details of the syntax of the programming language
* Since it is in pictorial form, any error in logic can be detected more easily. Thus, it ensures an error-free program.
* The flow chart is made of symbols that have standardized meanings.
* These symbols are standardized by "American National standards Institute" (ANSI)
* The following is the table showing various symbols used in flowcharts along with their name and brief description.

Pseudocode

* The pseudocode is an informal way of writing program human understanding. Fox better It is written in simple English, making the complex program easier to understand.
* The word "pseudo" means imitation or false and
* The word "code" means the instruction written in a programming language.
* Therefore," "Pseudo code is an imitation of actual computer instruction" \* Because pseudocode emphasized the design of the program it is also called as "Program Design Language (POL)."
* Pseudocode cannot be compiled or interpreted.
* As it doesn't follow the programming language', syntax.

Pseudocode Description

* Comments - Each instruction may be followed by a comment. The Comment begin with double //. Use of comment enhances the readability of the pseudocode.
* Variable names: For variable names, we use italicized lowercase letters for defined constants we will use uppercase letters.
* Assignment statement: The assignment statement will use the notations as E.g.: Set Sum = a+b (or) b=a
* Input/output: Data may be input and assigned to variables by means of a read statement with the following format
* Read: Variable list
* Where Variable list consists one or more variables separated by comma may be output by means of a point or statement. With the following format
* Print: message / variable list

Pseudocode to add 3 numbers and print their sum

Begin

Read: num1, num2, num3

Set: Sum=num1+num2+num3

Print: Sum

End.

Anatomy of Computer Languages

The programming language can be divided into two broad categories, on their level of interaction with the underlying hardware of the computer:

1) low level language

2) High Level language

1)Low level language

* A low level language is a programming language that deals with a computer's hardware components and constraints. These languages permit the efficient use of the Computer.
* The Machine language and assembly language are popular example of low-level languages.
* But problem with these languages is:
* These languages are hardware dependent
* I.e., Programs Written Using These Languages Cannot Be used on other computers.
* Programming using these languages is not an easy job. One must have thorough knowledge of the architecture of the computer

High Level language

* These languages are designed for better programming efficiency
* The advantages for these languages are:
* The Syntax for writing program instructions is very much English statements. This enables the reader to learn Like High level languages quickly
* The Programs written in High level languages can be easily
* Understood.
* The Programs Written in High level languages are not hardware Dependent.
* I.e., The Program Written for One machine can be transferred to another machine.

Programming Environment

* The typical tools in a programming environment include editors, language translators, linkers, loader, test data generators etc.
* Editor - Editor is a program that allows creating and editing of program i.e., to type and edit you need a program called editor.
* Language translators- The assemble language and high-level languages requires translators to translate instructions written in assembly language before they can be executed.
* Source code: a Program written using assembly language or high-level language is known as source code
* Object code: The translated version of the source code in machine language is known as object code.
* Assembler: an assembler is a program that translates a program in assemble language (source code) to machine language (object code)
* Compiler: A Compiler is a program that translates the program
* Written in a high-level language (source code) to Machine language (object code)
* The process of converting a source code to machine code or object Code is known as compilation.
* Interpreter - An interpreter is a computer program that directly executes Instructions written in a high-level language, without compiling or translating them into Machine language or object
* Code.
* Loader: loader is a program That is responsible for transferring the executable file from the Secondary storage to memory
* Note: Integrated Development Environment (2Ds) Such as Turbo C/C++. Dev C++ etc integrate various tools (editor, compiler etc) so that programmer can work with all these fools with much ease.

Programming Paradigms.

* Paradigm Can also be formed as method to solve some problem or do some Task
* The Programming paradigm is a method to solve a problem using Tools and techniques that are available to us following are some Approach.
* There are varieties of paradigms:

The two basic Paradigms are

1) procedural oriented programming

2) Object oriented programming

Procedural Oriented Programming

* Procedural oriented programming is a programming language that follows a step-by-step approach to break down a task into a Collection of variables and functions.
* In this programming, each step is executed in a systematic manner So that the computer can understand what to do.
* The procedural oriented programming is derived from structured programming. And it follows the Top-down approach to solve problem.
* Examples of procedural oriented programming language C, Pascal, ALGOL, COBOL, BASIC etc.

Object Oriented Programming

* The Object-Oriented programming is a programming language that Uses Objects and classes for creating models based on real world.
* The objects contain data in form of attributes and the code in form of methods and It follows A Bottom-Up approach
* Example of object-oriented programming languages are C++, C#, Java, Python etc.

Program Development Life Cycle

* Program development life cycle is a systematic way of developing quality programs.
* It provides an organized plan for breaking down the task of program development into small parts, each of which must be Successfully completed before Moving To next step.
* Program development process are divided into following steps

1) Defining the problem: Requirement gathering and analysis

2) Designing the program: approach to be used

3) Building The program: using right programming language instructions

4)Testing the program: test program by providing a different input

5) Documentation of program

6) Deploying and maintaining the program

Step1: Defining the problem

* Define the Program specifications precisely
* Define the problem, define the input data, The Processing that should take place, the format of output and the constraints etc.

Step2: Designing the program

* Program Design begins by focusing on the main goal that the program Is trying to achieve
* Then we break the program into smaller parts, each part
* Contribute to the output.
* In designing the program, we select the appropriate approaches i.e., top-down approach and bottom-up approach
* The first step is to identify the main function, i.e., the program's major activity.
* Then try to break down the major activity into smaller units.
* For designing the program, we use an appropriate the program design tool to visualize how the module will work and what will be the output.
* The various designing tools used can be

1) Algorithm 2) Flowchart and 3) pseudocode.

Step 3: Building the program

* Once the design of the program is ready, the next step is to convert the program design into a computer program.
* Each step of the program design is coded as one or more programming language (C, C++, Java etc) Instructions.
* Then we use a proper ides to create, edit. Compile, link and execute the program.

Step 4: Testing the program

* When the program is executing, the output of the program may not be correct. Because of logical errors in the program.
* The Programmes must find the correct logical errors by carefully examining the program output for a set of data for which results are already known.
* Here we most provide different range of inputs to test the output.

Steps 5: Documenting the program.

* All the details like the algorithms, pseudocodes, Flowcharts developed during the design phase, list of variable name and definitions, description of files that the program need to work with, and the format of output that the program produces, in depth explanations of major program features, reference documentation of all program commands, etc all of this must be documented and saved or placed together for future reference.

Step 6: Deploying and Maintaining the program

* Even after the program is completed, it needs to be maintained and evaluated regularly.
* In program maintenance, the programming team fixes program errors that users discover during its day-to-day use.

Note: The Four important aspects of any languages are

1) The way it stores data

2) The way It operates upon this data,

3) How it accomplishes input and output

4) How it lets you control the sequence of execution of instructions in a program.

-------------------------------------------------------------------------------------------------------------------------------------------------------------------------

UNIT-II

C language fundamentals and Input/ output Handling

Introduction to C Programming language

Introduction

History

* C programming language was developed by " Dennis Ritchie" In 1972 at Bell Laboratories, located in the USA.
* The C programming was developed as a System Implementation language for UNIX operating system.
* It is derived from the languages:
* Basic Combined programming language (BCPL) and B Language.
* The first C language was released in 1978, in book named "The C programming language" published by Brain Kernighan and Dennis Ritchie.
* The C language was formally standardized in December 1989, when the American National standards Institute (ANSS) Standard for C was adopted.

Anatomy

* The C language Stands between low level and high level languages
* Hence C is often called "Middle level language", Since it was designed to have both a good programming efficiency as well as a good machine efficiency.

Paradigm

* The C is a "procedural oriented programming language"
* C programming follows a step-by-step approach to break the program or problem into smaller parts, called functions.
* in order to execute the program.
* It follows a “top-down approach” to solve a problem in which the flow of control is in downward direction.
* Each step in C programming is executed in sequential and Systematic manner.
* Note: C programming cannot be called as "object-oriented programming because it does not support OOP features such as polymorphism, encapsulation and inheritance programming.

Characteristics

Some of the key features of C language are

* It is a "General purpose programming language", therefore It can be used to solve wide variety of problems.
* It is a "Structured programming language", therefore Programs developed in it can be easily understood.
* The C language is a case sensitive i.e. it will treat the lower-case alphabets and upper-case alphabets differently.
* It was the first language to provide rich set of operators
* The C allow to you to use pointers, to access any storage location
* It is a "platform independent" or "Portable" language, that means any program written in it can be used on any Computer

without or with little modifications.

* C allows you to develop your own library of functions, even it allows you to link it to your any program like Standard library functions.

Importance.

1. if one needs to extend the operating system to work with new devices one needs to write device driver programs. These programs are exclusively written in C.
2. programs not only have to run fast but also have to work in limited amount of memory. No wonder that such programs are written in C. C is the language of choice while building such operating systems and programs.
3. react fast to the user inputs.
4. Major parts of popular operating systems like Windows, UNIX, Linux is still written in C.

Applications

* C language can be used for system programming that include writing software for language translators, device drivers,
* editors, Inkers, loaders etc.
* It can be used in Network software to implement different Communication protocols.
* It can also be used in graphics programming that include writing Software for graphical user interfaces (GUIS), presentation graphics etc.

General Structure of C program

* The Structure of a C program makes the program easy to modify, easy to document and make it consistent in format.
* The structure of a C program can be mainly divided into Six Sections and each section having its purpose.
* The Six sections of in structure of a C program are as follows:

1. Documentation section
2. Link / preprocessor section
3. Definition section
4. Global declaration section.
5. Main section.
6. Sub program section.

1)Documentation section:

* This section is optional. This section Contain Comments (// single line comment (or) /\*\_\_multi line comment -- \*/)
* It contains the description about the program.
* It can also contain the name of the programmer, the date on which it was written or modified etc.

2)Link / preprocessor section :

* This section contains the preprocessor directives. All the header files are included in this section. The frequently used preprocessor directive is #include, The Include tells which header files are to be included In the program
* the following are two example and ways for including a file.

#include <system file.h > --- e.g., #include <stdio.h>

#include "user file.h" --- e.g., #include “Factorial.h"

3)Definition Section:

* It contains preprocessor directive, which Contain Symbolic constants. The frequently used preprocessor directive Is #define. The define directive is used to associate or replace an identifier with a Constant value (literal) that is to be used at many places in the program.

for example, #define PI 3.14

4)Global Declaration Section:

* It is optional, it contains the global declaration. These declarations usually include the declaration of the global variables which are to be shared between many functions in the program.
* These declarations can also include the declarations of function, to be used in the program.

5)Main Section:

* This section contains the main () function. The execution of the program always starts from the main () function. It is mandatory to include a main () function in every c program.
* The main section includes local variable declarations, declaration statements, assignment statements, input/output Statement, Arithmetic statements etc.
* The first section in main () function, as well as other functions, Contains local declaration. i.e., these declarations include the declaration of variables which are to be shared to that function only.
* The second section in main () function, contain the statements that defines the actions to be performed by the function.

6)Subprogram section:

* This section is optional. It contains the other user-defined functions. These functions are called in the main () function.

|  |
| --- |
| Section 1: Comments |
| Section 2: preprocessor directives |
| Section3: Definition section |
| Section4: Global declarations |
| Section 5:  int main ()  {  Local declaration  Statements  } |
| Section 6: other user defined functions |

Steps in building C Program

* The process of building the C program include the Five Steps which are as follows

1. Designing the program
2. Creating the program
3. Compiling program
4. Linking the program
5. Executing the program.

1) Designing the program

* In designing the C program, we use top-down approach
* first, we define the program specification i.e., what the problem is, what type of input is provided, what processing should take place and what will be the format of the output etc.
* Then we identify the main process or routine to achieve main goal.
* Then we break down the whole routine or program into smaller parts and this individual parts are designed separated in such a way that each part contributes to main goal.
* For designing we use program design tools to visualize how the Program works.
* The familiar program design tools are

1. Algorithm
2. flowchart and
3. Pseudocode.

* Using these tools, we design the program step by step to achieve the a solution.

2)Creating the program

* Once the design of the program is ready, we convert the program design into a computer program. Where each step of the program design is Coded as one or more C language instructions.
* Once the program code is ready, we create or enter the code into Computer memory using text editor. Then we make changes in code if necessary, using text editor.
* Then we save the program code on the disk with extension ".c".
* This Stored file is known as "Source file", and its contents are known as "Source Code"

3) Compiling the program

* The Source code in the source file, must be translated into machine Code to execute.
* To carry out this conversion of Source code to machine code we use a Program called "Compiler", and this process of translation is known as, Compilation"
* The C compiler is a combination of preprocessor and translator.
* The preprocessor reads the source code and it removes all the Comment and it takes the preprocessor directive and substitute directives with content of Specified header file. The output of the preprocessor is intermediate file Known as translation unit.
* The translator reads the translation unit and checks the translation unit for Syntax error. If there is an error it displays the error message. then after correcting the error, we repeat the compilation. If there is no error then translator translate the source code into machine code.
* The translated version of the source code to machine code is known as "Object code"
* This object code is stored on the disk with extension “.obj”.

4)Linking the program

* Once the source code is translated into object code, though it is in machine language still it will not be in executable form.
* Some more information like library files or user defined files are also need to be included in the object code to make it in a form that can be executed by the computer.
* The tool called " Linker "is used to link the library files and object code and produce a final machine code which is in the executable form, known as "executable code"
* This executable code is stored on the disk with extension "\*.exe".
* This executable code is the final form of the program that is ready for execution.

5) Executing the program

* To execute a program, we give an operating system command Called "run". to load the program into computer memory and execute it.
* Then the operating system uses a program called "loader" to load the program into memory, once it is loaded into memory the program get executed.
* After execution, output is sent to console.

The First C Program

Rules:

1. No blank spaces are allowed within a variable, constant or keyword.
2. Every C statement must end with a semicolon (;). Thus “;” acts as a statement terminator.
3. All statements are entered in small case letters.
4. The statements in a program must appear in the same order in which we wish them to be executed;

Four important aspects of any language are

1. the way it stores data,
2. the way it operates upon this data,
3. how it accomplishes input and output and
4. how it lets you control the sequence of execution of instructions in a program.

C Instructions

* There are basically three types of instructions in C:

1. Type Declaration Instruction
2. Arithmetic Instruction
3. Control Instruction

* The purpose of each of these instructions is given below:

1. Type declaration instruction − To declare the type of variables used in a C program.
2. Arithmetic instruction − To perform arithmetic operations between constants and variables.
3. Control instruction − To control the sequence of execution of various statements in a C program.

1.Type Declaration Instruction

* This instruction is used to declare the type of variables being used in the program.
* Any variable used in the program must be declared before using it in any statement.
* The type declaration statement is written at the beginning of main () function.

For e.g.: char name;

* There are several subtle variations of the type declaration instruction.

1. While declaring the type of variable we can also initialize it.

For example, int i = 10, j = 25;

1. The order in which we define the variables is sometimes important sometimes not.

For example,

int i = 10, j = 25; is same as ; int j = 25, j = 10;

However,

float a = 1.5, b = a + 3.1; is alright, but float b = a + 3.1, a = 1.5; is not. This is because here we are trying to use even before defining it.

1. The following statements would work

int a, b, c, d;

a = b = c = 10;

However,

the following statement would not work

int a = b = c = d = 10; Once again we are trying to use b (to assign to a) before defining it.

1. Arithmetic Instruction

* A C arithmetic instruction consists of a variable name on the left-hand side of “=” and variable names & constants on the right-hand side of “=”.
* The variables and constants appearing on the right-hand side of “=” are connected by arithmetic operators like +, -, \*, and /.

For e.g.:

int a, b;

int c = b;

int d=a+ c; /\*here a, c is operand and + is arithmetic operator. the arithmetic operation result of a and c variables is assigned to the variable d on Left-hand side\*/

* The variables and constants together are called ‘operands’ that are operated upon by the ‘arithmetic operators’ and the result is assigned, using the assignment operator, to the variable on lefthand side.

Rules:

1. C allows only one variable on left-hand side of “=”.

That is, z = k \* l is correct, whereas k \* l = z is error.

1. No operator is assumed to be present. It must be written explicitly.

In the example below, the multiplication operator after b must be explicitly written.

//a = c.d.b(xy) usual arithmetic statement

//b = c \* d \* b \* (x \* y) C statement

1. In C there is no operator for performing exponentiation operation. Thus, following statement is invalid.

b = 3 ^ 2;

1. Control instructions

* The ‘Control Instructions’ enable us to specify the order in which the various instructions in a program are to be executed by the computer.
* In other-words the control instructions determine the ‘flow of control’ in a program.
* There are four types of control instructions in C. They are:

1. Sequence Control Instruction
2. Selection or Decision Control Instruction
3. Repetition or Loop Control Instruction
4. Case Control Instruction

1.Sequence Control Instruction

* The Sequence control instruction ensures that the instructions are executed in the same order in which they appear in the program.
* In sequence control structure the various steps are executed sequentially,
* i.e., in the same order in which they appear in the program.
* In fact to execute the instructions sequentially, we don’t have to do anything at all. By default, the instructions in a program are executed sequentially.

2.Selection or Decision Control Instruction

* Decision and Case control instructions allow the computer to take a decision as to which instruction is to be executed next.
* Many a times, we want a set of instructions to be executed in one situation, and an entirely different set of instructions to be executed in another situation. This kind of situation is dealt in C programs using a decision control instruction.
* As mentioned earlier, a decision control instruction can be implemented in C using:

1. The if statement.
2. The if-else statement
3. The conditional operators
4. The switch statement.

Repetition or Loop Control Instruction

* In General cases we need to perform an action over and over, often with variations in the details each time. The mechanism, which meets this need, is the loop.
* The versatility of the computer lies in its ability to perform a set of instructions repeatedly.
* This involves repeating some portion of the program either a specified number of times or until a particular condition is being satisfied.
* This repetitive operation is done through loop control Instruction.
* The Loop control instruction helps computer to execute a group of statements repeatedly.
* There are three methods by way of which we can repeat a part of a program. They are

1. Using a for loop
2. Using a while loop
3. Using a do-while loop

4.Case Control Instruction

* In real life we are often faced with situations where we are required to make a choice between a number of alternatives rather than only one or two.
* In C programming the choice we are asked to make is more complicated than merely selecting between two alternatives.
* C provides a special control statement that allows us to handle such cases effectively.
* The control statement that allows us to make a decision from the number of choices is called a switch, or more correctly a switch case-default,since these three keywords go together to make up the control statement.

Tokens

* The Smallest individual or independent element are Known as "tokens"
* The token is a smallest entity that has a meaning in itself.
* Tokens are the basic building blocks in C language which are constructed together to write a program.
* The Tokens in C language are divided into six categories

1. Keywords
2. Identifiers
3. Constants/literals.
4. strings
5. Special Symbols
6. operators.

1) Key words

* Every word in C is classified into a keyword or an identifier.
* Keywords are basically those words that have a predefined and fixed meaning and these meanings cannot be changed.
* The Keywords are also called as "Reserved words".
* All Keywords are written in lower case.
* There incorrect usage results in a syntax error.
* There are 32 Keywords used in C, as per American National Standard Institute (ANSI) standards.
* The list of Keywords of C language are as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| Int | unsigned | break | register |
| Char | if | continue | static |
| Float | else | return | sizeof |
| Double | switch | default | typedef |
| Void | case | go to | enum |
| Short | while | const | struct |
| Long | for | auto | union |
| Signed | do | extern | volatile |

2) Identifiers

* An identifier is basically a name in a program.
* An identifier can be used to denote variables, arrays and functions.
* These are user defined names consist of sequence of characters.
* The good identifiers are specific, descriptive and short.
* The only valid characters and symbols for identifiers are as follows:

1. The capital /upper case letters A to Z
2. The lower-case letters a to z
3. digits 0 to 9 and underscore (\_)

Rules for identifiers

* The first character of identifies must be an alphabet or underscore, it cannot be a digit.
* The word we use can't be a Keyword
* The identifier Cannot contain white space.

Note

* Since C language is case sensitive it distinguishes between lowercase and uppercase, therefore the identifiers
* for example, Sum and sum will be treated as two different identifiers.
* We use the lowercase letters for identifiers commonly
* The underscores are usually used to link the two words to create a meaning full names.

3) Literals/Constants

* The Constants refer to fixed value that do not change during the execution of a program.
* A constant is an entity that doesn’t change

Syntax: const datatype variable\_name = value;

* C constants can be divided into two major categories:

1. Primary Constants: Integer, Real and Character constants.
2. Secondary Constants: Array, Pointer, Structure, Union constants

* C primary constants can be divided into three types

1. Integer constant
2. Real constant
3. Character constant.
4. Integer constant

* Integer constant are numbers without a decimal point.

Rules for Constructing Integer Constants

* An integer constant must have at least one digit.
* An integer value can be either positive or negative.
* Even “0” is also considered as integer constant
* If no sign precedes an integer constant it is assumed to be positive.
* Negative constants are preceded by “- “sign.
* No Commas or blanks are allowed within an integer constant
* the range of integer constants is between -32768 to 32767
* Example of integer constant

ex: 426, -760, +7827, -8000

* An integer constant can be expressed in decimal, binary, octal or hexadecimal notation.
* An integer without leading 0 and consisting of sequence of decimal digits (0-9) is treated a decimal integer

e.g.: 750, -124, +124

* An integer with leading OB and followed by sequence of binary digits (0,0) is treated as binary integer

e.g.: Ob10101, OB111000, -0b110100

* An integer with leading 0 and followed by sequence of octal digits (0-7) is treated as octal integer.

e.g.: 0165, 0315, -0250

* An integer with leading OX and followed by sequence of Hexadecimal digits (0-7, A-F) is treated as hexadecimal integer

e.g.: 0x575, -OXABCD, OX75A

2)Floating / Real Constants

* Real Constants are number having fractional part.
* The real constants could be written in two forms

1. Exponential form
2. Fractional form

Real constant in fractional form

Rules for constructing real constants expressed in fractional form:

* The real constant must have at least one digit.
* It must have a decimal point.
* The real constant could be either positive or negative
* Default Sign is positive
* No commas or blank are allowed within a real constant

for e.g.: +825.34, 426.0, -32.76

Real constant in exponential form.

* The exponential form of representation of real constants is usually used if the value of the Constant is either too small or too large.
* In exponential form of representation, the real constant is represented in two parts
* The part appearing before ‘e’ is called "mantissa"
* The part following ‘e' is called "exponent".

Rules for constructing real constants expressed in exponential form:

* The mantissa part and the exponential part should be separated by a letter e
* The mantissa part may have a positive or negative sign, the default Sign is positive
* The exponent must have at least one digit
* The exponent may have a positive or negative integer, the default Sign is positive
* The range of real constants expressed in exponential form is. -3.4e38 to 3.4e38

for e.g., ÷ +3.2e-5, 4.1e8, -0.2e+3, -3.2e-5

3)Character Constant

* A character constant is a single alphabet, single digit or a single special symbol enclosed within single inverted Commas (` ́) single
* for example, 'A'
* The maximum length of a character constant can be one character
* Each character is represented by an integer value that represents the ASCII code of the character.

For e.g., 'B', 'I', '5', `='

* Local constant [Syntax of Constant variable declaration]

const datatype variable\_name = Value;

for e.g., const float pie = 3.14;

* Global Constan [Syntax for defining or declaring global constant]

#define variable\_name value

for e.g., ÷ #define PI 8.14

4)String

* A string is a sequence of characters enclosed in double quotes (“ “)
* The characters can be from any characters in the character set.

For e.g., “Hello!",” A”, “you are welcome".

* Declaring String Variable

char name[size]

for e.g., char c [20];

* Initializing string variable

char name[size]="value";

for e.g., Char c [10] = {'H' 'e', 'l', 'l', 'o', ` \o'};

(OR)

char c [10] = "Hello";

5)Special Symbols

* The Special Symbols are also called "Punctuators" or "separators."
* The following are the list of special symbols and their description.

|  |  |
| --- | --- |
| Symbols | Description |
| Brackets [ ] | used to enclose array subscripts. |
| Parentheses () | These are used to indicate in function declaration as well as function definition, function parameters and expressions. |
| Braces { } | These are used to enclose the block of statements. It is also used to enclose list of elements while initializing arrays. |
| Comma , | used to separate arguments and parameters |
| Semicolon ; | used to terminate a statement |
| Equal to sign = | used to initialize a variable and as assignment operator |
| Pound sign # | Used for preprocessor directive. |
| double quotes “ ” | used to enclose the string of characters. |
| Single quote ‘ ‘ | used to enclose the single character. |

6) Operators

* The operators are special Symbols which are used to perform logical or mathematic operations on data and variables.
* The data or variable on which the operations que performed is called "Operand"

e.g.: x + y

x and y: operands

+: Operator

* The following are the various type of operators used in c

|  |  |
| --- | --- |
| Operator | Type |
| ++ , -- | Unary operators |
| +, - , \* , / , % | Arithmetic operators |
| <, > , <==, >== , ==,!= | Relational operators |
| && , || , ! | logical operators |
| +=, -= , \*= , /= , %= ,= | Assignment operators |
| &, | , ~ , ^ , >> ,<< | Bitwise operators |
| ? , : | Ternary/Conditional operator. |
| , (Comma operator)  Size of operator | other operators |

Comments

* Comments are non-executable code which are used to provide the Information to programmer.
* Comments are also known as Internal program documentation.
* The comments are solely given for understanding of the programmer and are completely ignored by the compiler.
* Any no. of comments can be written at any place in the program.
* The comments in C are two forms

1. line comment or single line comment
2. Block Comment or multiline comment

Line Comment:

* It is used to provide one liner description about the program
* the single line comment about the program should begin with "// " and ending token is not required. the end of line automatically ends the comment.

Multi line comment:

* It is used when the Comments will spend several lines.
* the multi-line comments about the program should be enclosed within “/\* \*/”

Escape Sequences / Back slash sequences in C

* The Escape Sequences are used for Special output instructions.
* The format of escape sequences is a black slash (`\') followed by characters.
* Only the certain characters can be used as escape sequences.
* The following are the Common escape sequences used in C

|  |  |  |
| --- | --- | --- |
| Escape sequence | Represents | Effect |
| \a | Alarm (or) Beep | Produce one audio alert. |
| \b | Back space | Moves the cursor one space left. |
| \f | Form feed | Moves the active position to top of the next page. |
| \n | New line | It takes the cursor to the new line |
| \r | Carriage return | Moves the active position to initial position of current line. |
| \t | Horizontal tab | Moves over to next wide space on current line. |
| \v | Vertical tab | Moves over to the initial position of next vertical position. |
| \’ | Single quote | Display character ‘ in the output. |
| \” | Double quote | Display character “ in the output. |
| \\ | Backslash | Display character \ in the output. |
| \? | Question mark | Display character ? in the output. |
| \0 | Null | Used to terminate a string. |

Concept of Datatype

* Data type is the type of data which is used in the program.
* Datatype Specifies the type of data that the variable can store like integer, character, float etc.
* The data type in C can be classified as follows

1) Primitive/Built in Data types

* These are the most basic datatypes in c.
* They are predefined in c and can be used directly in a program.
* The primitive datatypes are further classified into

1. integer datatype
2. character datatype
3. Float datatype
4. double datatype
5. void datatype
6. Boolean datatype.

2)Derived Datatypes

* The datatypes that are derived from primitive or built in data type are called as derived data types.
* The derived datatypes are further classified into

1. function
2. Array
3. Pointer
4. Reference.

3)User defined data types

* The datatypes that are first declared and defined by the user before use called the user defined datatype.
* The user derived datatypes are further classified into

1. class
2. Structure
3. Union
4. Enum
5. Typedef.

1) Primitive datatypes

* This data types are also called as built-in, fundamental or basic data types.
* These are predefined datatypes in C and can be used directly in a program
* The following are the five primitive data types supported by C

1. Integer datatype
2. character datatype
3. floating point drape
4. Double datatype
5. void datatype
6. Boolean datatype

1) Integer datatype

* The Integer datatype is used to store integer value with no decimal points.
* The integer value consists of a sequence of digits preceded by Optional sign, plus (+) or minus (-)
* A decimal point and Comma are not allowed in an integer value
* The integer datatype is referred by the key word " int"
* The integer data type is available in two forms

1. signed and
2. unsigned

* The integer data type consists of a subset of integers represented as short, int and long
* The following is the table showing different types of integers, their memory requirement, range of values and format specifiers.

|  |  |  |  |
| --- | --- | --- | --- |
| Data type | Memory (in bytes) | Range of values | Format specifier |
| Short int | 2 | -32768 to +32767 | %d |
| Unsigned short int | 2 | 0 to 65535 | %hu |
| Int | 4 | -2147483648 to +2147483647 | %d |
| Unsigned int | 4 | 0 to 4294967295 | %u |
| Long int | 4 | -2147483648 to +2147483647 | %ld |
| Unsigned long int | 4 | 0 to 4294967295 | %lu |

2) Character datatype

* This datatype is used to store single characters.
* The characters can be letters, digits and symbols.
* The character data type occupies 1 byte of memory.
* The character datatype is referred by the key word "Char".
* This data type available in two forms

1. signed and
2. unsigned.

* The following is the table showing different types of character, their memory requirement, Range of values and format specifiers.

|  |  |  |  |
| --- | --- | --- | --- |
| Data type | Memory (in bytes) | Range of values | Format specifiers |
| Signed char | 1 | -128 to 127 | %c |
| Unsigned char | 1 | 0 to 255 | %c |

3)Floating Point datatype

* The float data type is used for numbers that have a decimal point which are commonly called "Real number"
* The float is used to store decimal and exponential values.
* The floating datatype is used to store the real number with 6 digits precision.
* It is referred by the key word 'float'.

|  |  |  |  |
| --- | --- | --- | --- |
| Data type | Memory (in bytes) | Range of values | Format specifiers |
| float | 4 | 1.2e^-38 to 3.4e^+38 | %f |

4)Double datatype

* It is used to store decimal and exponential values
* The size of double datatype is twice as floating-point datatype.
* It can easily store 16 to 17 digits after or before a decimal point.
* double datatype is more accurate as it uses 12 digits precision.
* This data type is divided into two data types

1. double and
2. long double.

* It is referred by the key word 'double'

|  |  |  |  |
| --- | --- | --- | --- |
| Data type | Memory (in bytes) | Range of values | Format specifier |
| Double | 8 | 1.7e^-308 to 1.7e^+308 | %lf |
| Long double | 16 | 3.4e^-4932 to 1.1e^+4932 | %lf |

5) Void datatype

* The void means empty, the void datatype does not Contain any value
* It is used as return type for functions that do not return a value.
* It is referred by the key word "void".

6) Boolean data type

* The Boolean datatype can store only two values 0 and 1

1. True to represent all non-zero numbers.
2. false to represent zero.

* It is referred by the Key word "Bool"

VARIABLES

* Variable is a name given to locations in memory.
* The variable is an entity that may vary during program execution.
* In any program we typically do lots of calculations. The calculated values are stored in these memory cells.
* To make the retrieval and usage of these values easy these memory cells (also called memory locations) are given names.
* Since the value stored in each location may change the names given to these locations are called variable names.
* Variables hold different kind of data and the Same variable might hold different values during the execution of a program.
* They are user defined names given to memory locations and are used to store values.
* Each variable is associated with specific datatype, which determines the size. The range of values that can be stored within the memory and the set of operations that can be applied to the variable.
* So basically, variables are location where data values of integer, real or character type are stored, and to these variables you can assign value at time of declaration and change value at any point and as many times.
* But and when this variable are prefixed with const keyword then this const variable are used to store constant values of integer, real or character type, and to these variables you can assign value only at the place of declaration and the can't be changed in between.

Rules

1. variable should start with a letter or an underscore character.
2. letter, digits and the underscore character are allowed to name a variable.
3. The sizeof the variable name can be up-to 32 digits.
4. The Keywords should not be used as a variable name.
5. lowercase and uppercase letters are distinct, because C is a Case sensitive.

Declaring Variables

* Each variable in a program must be declared and defined.
* The declaration is used to name a variable
* The definition is used to create (allocate memory) a variable.
* A variable can be declared and defined at the same time.
* when we create variable, the declaration gives a name and the definition reserves memory.
* Only after variables are defined, they can be used to hold the data that are required by the program for its operations.
* To create a variable, we first specify the data type and then its name.
* If two or more variable of same datatype are to be created, their names are separated using comma.

Syntax for declaring and defining a variable

datatype Variable\_name

(OR).

datatype Var\_name1, Var\_name2, - - - - - -Var\_name n;

* Where var\_name1, var\_name2 are the names of the variables separated by comma
* The datatype can be any of the valid data type
* The variable cannot be of void data type

for e.g.: int num;

This declaration tells the C compiler to reserve space in memory to hold the integer value and name the memory location as 'num'.

Initializing Variables

* The Variables can be assigned or initialized by an initial value during its declaration and definition.
* To initialize a variable, a variable is followed by character ‘=’ and then the value to be assigned to the variable.
* Syntax for Initializing variable

datatype variable name = value;

(OR)

type Var\_name1 = value, Var\_name2= value;

(OR)

datatype variable name;

var\_name = value;

for e.g.: int num=8;

* This tells the C compiler to

1. reserve space in memory to hold the integer value
2. Associate or give the name num to the reserved memory location
3. store the value 8 at this location

* We may represent num’s location by following memory map

num 🡪 location name

8 🡪 Value at location

65524 🡪Location number.

Errors

* Error is an illegal operation performed by the user which results in abnormal working of program
* Programming errors are also known as the bugs and the process of identifying and eliminating these errors is known as debugging.
* The errors are detected either during the time of compilation or execution.
* The programming errors must be removed from the program for the successful execution of the program.
* There are mainly five types of errors exist in C which are as follows:

1. Syntax error
2. Runtime error
3. Linker error
4. Logical error
5. Semantic error

1)Syntax error

* The syntax errors are mainly occurred due to the mistakes while typing or when you violate the rules of writing Syntaxes of c
* The syntax errors are detected by the compiler, thus these errors are also known as compile time errors.
* The most commonly occurred syntax errors are

1. missing the parenthesis {} while writing the code
2. printing the value of a variable without declaring.
3. Missing the semicolon (;) at the end of the Statement
4. Syntax of a basic construct is written wrong.

For e.g.: while loop

#include <stdio.h>

int main ()

{

while (.)

{

printf("hello");

}

return 0;

}

Output: error expected expression before '.' token while (.)

The error is generated, because we put (.) instead of condition in ‘while’

2)Runtime error

* The errors which occur during the program execution even after the successful compilation known as “runtime errors”.
* The main cause of runtime error is the program is not able to perform the operation.
* The most Common example of the runtime error is the division by zero. also known as division error

for e.g., #include <stdio.h>

int main ()

{

int num=9, div=0;

div=num/0;

printf (" result = "%d", div);

}

Output:

warning: division by zero [-wdiv-by- zero] div=num/0

* These errors are very difficult to find, as the compiler does not point to these errors.

3)Linker error

* Linker error are mainly generated when the executable file of the program is not created.
* These errors occur after compilation when linker link the object files.
* These can be generated either due to the wrong function prototype or usage of the wrong header file.
* The most common linker error that occur is that when we use Main () instead of main ()

for e.g.,

#include <stdio.h>

int Main ()

{

printf ("\n Hello");

return 0;

}

Output:

(.text+0x20): undefined reference to 'main'

4)Logical Error

* The logical error is an error that leads to an undesired output.

i.e., on compilation and execution of a program the expected output is not obtained when certain input values are given.

* These type of errors produce the incorrect output, but they are error-free, Known as logical errors.

For e.g.,

#include <stdio.h>

int main ()

{

int sum=0;

int a = 1;

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

// logical error as we put the (;) after loop

{

sum =sum +a;

a++;

}

printf ("sum =%d", sum);

return 0;

}

Output :

sum = 1

* Here we are trying to print the sum of 10 digits, but we got the wrong output as we put the semicolon after for loop, so

inner statements of for loop will not executed.

5)Semantic error

* Semantic errors are the errors that occurred when the statements are not understandable by the compiler.
* The most common examples of semantic error are as follows

1. Use of an un-initialized variable

for e.g.,

int i;

i=i+23;

1. Errors in expressions

For e.g.,

int a, b, c;

a + b =c;

output error: 1 value required as left operand of assignment

a + b = c;

1. Array index out of bound.

For e.g.,

int a [10];

a [10] =34;

Operators in C

* The operator is a special symbol which is used to perform Computations or operations on a value or a variable.
* An operand is a data item on which the operations is to be performed.

for e.g.,

a + b

where a, b are operands and + is an operator.

* C language has wide range of operator to perform various operations
* The following are the different operators used in c'

1. Arithmetic operators
2. Relational operators
3. Logical operators
4. Bitwise operators
5. Assignment operators
6. Conditional operator/Ternary operator
7. other operators
   1. Size of operator
   2. Comma operator

1) Arithmetic operators

* These operators are used to perform mathematical or arithmetic Operation on variables such as addition (+), subtraction (-), multiplication (\*), division (), modulo division (%)
* Arithmetic operators are of two types

1. Unary operators
2. Binary operators

a) Unary operators

An operator that operates with a single operand is called Unary operator.

|  |  |  |
| --- | --- | --- |
| Operator | Description | Operator name |
| ++ | increment of a variable | increment operator |
| -- | Decrement of a variable | Decrement operator |

* These operators are most frequently used in while and for loops as control variables.

Note:

* Increment operator (++), increases the value by 1.
* The increment operator can be represented as pre-increment (++a) and post-increment (a++).
* Decrement operator (--), decreases the value by 1.
* The decrement operator can be represented as pre-decrement (--a) and post-decrement (a--).
* When these operators are used in prefix notation in an expression, then first value of the variable is changed accordingly and the new value is used in evaluating the expression.
* when these operators are used in postfix notation in an expression, then first existing value of variable is used in evaluating expression and then value is changed accordingly.

b) Binary operators

* The operator that operate with two operands are called Binary operators.

|  |  |  |
| --- | --- | --- |
| Operator | Operation performed | Example (for x=13, y=5) |
| + | Perform Addition | x + y =18 |
| - | Perform Subtraction | x - y =8 |
| \* | Perform Multiplication | x \* y =65 |
| / | Perform Division | x / y =2 (decimal part truncated) |
| % | Perform Modulus | x % y =3 (remainder of division) |

Note:

* Modulus % operator, give remainder.
* In modulus operation, the sign of the result is always the Sign of the first operand

For e.g., -13 % 5 = -3, -13 % -5 = -3, 13 % -5=3

* The modulus division can only be used with integer it Cannot be used with real operands.
* There is no operator for exponentiation operation in 'C'
* when one of the operands is real and other is integer, the operation always yields a real value.

for e.g., 15.0/10=1.5

2) Relational Operators

* These operators are used for checking the relation between two Operands
* These operators are used to carry out the comparison between different operands and they help in decision making.
* These operators are used to compare the values of operands, if the Comparison succeeds or true, they return a value 1 and if the comparison fails or false, then return value 0.

|  |  |  |  |
| --- | --- | --- | --- |
| Operators | Operation Performed | Example | Result |
| == | Equal to | 15 == 13 | 0 |
| < | less than | 13 < 20 | 1 |
| <= | less than or equal to | 5 <= 3 | 0 |
| > | greater than | 20 > 10 | 1 |
| >= | greater than or equal to | 20 >= 35 | 0 |
| != | not equal to | 5!= 2 | 1 |

3) Logical Operates

* The logical operators are used to form a compound condition by joining two or more conditions formed using relational operators
* An expression containing logical operator returns either 0 or 1 depending upon whether expression result true or false.
* The logical operators are commonly used in decision making in C programming

|  |  |  |  |
| --- | --- | --- | --- |
| Operator | Operations performed | Example (for c=5, d=2) | Result |
| && | Logical AND | ((c==5) &&(d>5)) | 0 |
| || | Logical OR | ((c==5) ||(d>5)) | 1 |
| ! | Logical NOT | ! (c==5) | 0 |

* The logical OR operator return 1 only if either one operand is true and result is 0 when both the operands are false
* The logical AND operator return 1 only when both the operands true, in all other cases it returns false i.e., 0
* The logical NOT operator changes a true value to false and false value to true
* i.e., if false result is true, if true result is false.

4) Bitwise operators

* These operators operate at bit level and allow us to manipulate the individual bits.
* These operators are used for testing, shifting and complimenting data bits

|  |  |  |
| --- | --- | --- |
| Symbol | Operation | Description |
| & | Bitwise AND | Perform AND operation on the bits of operands |
| | | Bitwise OR | Perform OR operation on the bits of operands |
| ^ | Bitwise XOR | Perform XOR operation on the bits of operands |
| ~ | Bitwise NOT | Used to Calculate one's complement of the bits |
| >> | Shift right | moves bits of the operands to Right |
| << | Shift Left | moves bits of the operands to left. |

* The bitwise AND takes two numbers as operands and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1.
* The bitwise OR takes two numbers as operands and does OR on every bit of two numbers. The result of OR is 1 if any of the two bits is 1.
* The bitwise XOR takes two numbers as operands and does XOR on every bit of two numbers. The result of XOR is 1 if the two bits are different.
* The Bitwise Not calculates one's complement and it takes one number and inverts all bits of it.
* Right shift operator takes two numbers, it right shifts the bits of the first operand by specified number of bits by second Operand. The second operand decides the number of places to Shift
* It is denoted by >>

Syntax: Operand >> n;

for example

let a=7 = 0111🡪in binary representation.

a>>2

=> 0000 0111 >> 2 = 0000 0001

For e.g.,

#include <stdio.h>

int main ()

{

int a=7;

printf (" The value of a>>2 is: %d", a >> 2);

return 0;

}

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

* Left shift operator It takes two numbers; it left shifts the bits of the first operand by specified number of bits by second Operand. This second operand decides the number of places to Shift.
* It is denoted by <<

Syntax: Operand << n;

for example

let a=5 = 0101🡪in binary representation.

a>>2

=> 0101 >> 2 = 0001 0100

5) Assignment Operator

* These operators are used to evaluate an expression and to assign the value present at the right-hand side to the left-hand side.
* Assignment operator is used for assigning a value to a variable.

|  |  |  |  |
| --- | --- | --- | --- |
| Operator | Description | Examples | Same as |
| = | Assign the RHS to LHS of expression | a= b | a= b |
| += | Addition assignment | a+= b | a=a + b |
| -= | Subtraction assignment | a-=b | a=a - b |
| \*= | Multiplication assignment | a\*= b | a=a \* b |
| /= | Division assignment | a/= b | a=a / b |
| %= | Division assignment | a%= b | a=a % b |

6) Conditional Operator

* The conditional operator is also known as ternary operator.
* There operators are used to perform certain conditional test on the expression.
* As Conditional operator operates on 3 operands, so it is also Known as ternary operator
* It is represented by 2 Symbols (? and :)

Syntax of Conditional operators

exp1? exp 2: exp3

* It means that if expression1 is true then executes expression2 otherwise execute expression3.

7)Other operators

a) Comma operator

* A Comma operator is used to combine several related expressions.
* The group of expressions separated by comma`,' are evaluated left to right.

for example: int a, c=5, d;

* Comma Operator can be used in while and for loop to separate expressions.

B)Sizeof Operator

* The sizeof is a unary operator that returns the size of the given operand (i.e., variable, constant etc) in bytes.

Syntax of Sizeof operator

sizeof (exp)

For example:

#include <stdio.h>

int main ()

{

int a;

printf ("size of int = %u bytes", sizeof (a));

return 0;

}

Output:

size of int = 2 bytes.

Expressions

* An expression is a combination of operators, constants, variables, or array element.
* An expression is a formula in which operands are linked to each other by the use of operators to compute a value.

**for e.g.,** result = a + b \* c;

* Here the entire line is a statement, but the portion after “=” is an
* Expressions are evaluated based ob precedence and associativity of operators
* These are four types of expressions in C. They are as follows
  1. Arithmetic expressions
  2. Relational expressions
  3. Logical expressions
  4. Conditional expressions.
* Each type of expression takes certain types of operands and uses specific set of operations.
* Evaluation of every expression produces a value of specific type.
* Note that Expressions are not statements, it may be components of statements.

1) Arithmetic Expressions

* An arithmetic expression is an expression that consists of operands and arithmetic operators.
* An arithmetic expression computes a value of type int, float or double.
* The expressions are evaluated by performing one operation at a time. The precedence and associativity of operators decide the order of the evaluation of individual operations.

Evaluation of Arithmetic expressions

* when individual operations are performed. The following three cases can happen

1. When both of the operands are of type integer, then arithmetic will be performed and the result of operation would be an integer value.

for e.g., 5/2 Yield 2 [Here even the result is 2.5 the fraction part is ignored]

1. When both of the operands are of type flout or double, then arithmetic will be performed and the result of operation would be a real value.

for e.g., 2.0/2.0 Yield 1.0 [Not 1]

1. When one operand is of type integer and another is of type real, then mixed arithmetic will be performed.

In this case, the first operand is converted into a real operand, and then arithmetic is preformed and the result of operation would be a real value

For e.g., 6/2.0 yield 3.0

* For example: The way arithmetic expression is evaluated.

5\*4/ (1+5\*2/3+6)

|  |  |
| --- | --- |
| Evaluation | Description |
| 5\*4/ (1+5\*2/3+6) | 5\*2 = 10 |
| 5\*4/ (1+10/3+6) | 10/3 =3 (3.33 Considered as 3) |
| 5\*4/ (1+3+6) | 1+3=4 |
| 5\*4/ (4+6) | 4+6=10 |
| 5\*4/ 10 | 5\*4=20 |
| 20/10 | 20/10 = 2 |

2)Relational Expressions

* A relational expression is an expression used to compare the value of two operands.
* A relational expression is used to test a condition in order to decide whether the action should be taken or not.
* Note that in relational expressions, a numeric value cannot be compared with string value.
* The result of the relational expression is either zero or non-zero value. Here zero value evaluate to Boolean false and non-zero to a Boolean true.
* Note that if relational expressions have one operand of int type and other of float type first, the int operand is converted to float type before relation expression is evaluated.
* example a sample relational Expressions

|  |  |
| --- | --- |
| Relational Expression | Description |
| X%2==0 | Used to test whether the x is an even number or not. The result in value 1 if x is an even otherwise result in value 0. |
| a! = b | used to test whether a is not equal to b. The results in 1 if a is not equal to b otherwise 0. |
| a + b == x + y | used to test whether expression "a + b" is equal to expression" x + y". |
| a>=5 | used to test whether the value of a is greater than or equal to 5. |

3) Logical Expressions

* Logical expressions combine two or more relational expressions and produces Boolean results.
* A logical expression is used to form a test condition in order to take a decision.

for example: x>y && x ==10

* The Sample logical expressions.

|  |  |
| --- | --- |
| Logical expression | Description |
| (x>2) && (x%2==0) | Used to test condition whether x is greater than 2 and x is even. The result of test will be 1 only if both conditions are true simultaneously. |
| x>10 || y < 11 | used to check whether X is greater than to 10 or y is less than 11. The result of test will be 1 if either or both conditions hold true. |
| ! (x>5) && (y ==0) | used to check whether x is not greater than 5 and y is equal to 0. The result of both condition is true it both conditions are true simultaneously. |

4) Conditional Expressions

* A conditional expression is an expression that return 1 if the condition is true otherwise 0.
* A conditional expression is also known as ternary operator.

Syntax: exp1? exp 2: exp3.

It means if expression 1 is true it executes expression 2 otherwise executes expression 3.

For e.g.,

big = a >b? a: b;

* Note: the use of parentheses is not necessary around the first expression of conditional expression. but use of parentheses is recommended as they make condition part of expression easier to use.

For e.g., big = (a>b)? a: b;

* Sample Conditional Expressions

|  |  |
| --- | --- |
| logical expression | Description |
| (a%2==0)?1: 0 | The conditional expression takes value 1 if a is divisible by 2 otherwise it takes value 0. |
| (x>y)? x--: y++ | The expression decrement x if x is greater than y else increment y. |
| (a>b)? ((a>c)? a:c): ((b>c)? b: c) | The conditional expression takes largest of the a, b and c and its value. |

Precedence and Associativity of Operators in C

* Precedence is used to determine the order in which different operators are evaluated in expression.
* Associativity is used to determine the order in which different operators with equal precedence are evaluated in an expression
* The associativity of operators determines the direction in which an expression is evaluated.
* Precedence is applied before associativity to determine the order in which expressions are valuated.
* The precedence is well defined by a rule BODMAS - Brackets, of Division, Multiplication, Addition and Subtraction.
* Division and multiplication is performed before addition and subtraction

for e.g.,

(5+ (3\*4))

=> (5+12)

=> 17

* Associativity can be left to right or right be left right
* Left to right associativity evaluates the expression by starting on the left and moving to the right.
* The right to left associativity evaluates the expression by starting on the right and moving to the left.

Rules / Note+

* Associativity is only used when there are two of more operators of same precedence.
* All operators with the same precedence have same associativity.
* Precedence and associativity of postfix ++ prefix ++ are different
* Comma has the least precedence among all operators and should be used carefully.
* There is no chaining of Comparison operators in c.
* For e.g.,

#include <stdio.h>

int main ()

{

int a=10, b=20, C=30;

if (c>b>a)

printf("TRUE");

else

printf("FALSE");

return 0;

}

Output: FALSE

|  |  |  |  |
| --- | --- | --- | --- |
| operator | description | Precedence level | Associativity |
| ( ) | functional call | 1 | Left to right |
| [ ] | Array element reference |
| 🡪 | Structure operator used with pointer to a structure. |
| . | Structure operator used with structure variable |
| ! | Logical NOT operator | 2 | Left to right |
| ~ | Bitwise NOT Operator (1's complement) |
| ++ | Increment |
| -- | Decrement |
| + | Unary plus |
| - | Unary minus |
| \* | Pointer reference |
| & | Address of |
| type | Type cast (conversion) |
| sizeof | Size of an operand |
| \* | multiplication | 3 | Left to right |
| / | Division |
| % | modulus (remainder) |
| + | Binary plus (Addition) | 4 | Left to right |
| - | Binary minus (subtraction) |
| << | left shift | 5 | Left to right |
| >> | right shift |
| < | less than | 6 | Left to right |
| <= | less than or equal to |
| > | Greater than |
| >= | Greater than or equal to |
| == | Equal to | 7 | Left to right |
| != | Not equal to |
| & | Bitwise AND | 8 | Left to right |
| ^ | Bitwise exclusive OR | 9 | Left to right |
| | | Bitwise OR | 10 | Left to right |
| && | Logical AND | 11 | Left to right |
| || | Logical OR | 12 | Left to right |
| ?: | Conditional operator (ternary Operator) | 13 | Left to right |
| = | Simple assignment | 14 | Right to left |
| += | Assign Sum |
| -= | Assign difference |
| \*= | Assign Product |
| /= | Assign division (quotient) |
| %= | Assign remainder |
| &= | Assign bitwise AND |
| ^= | Assign bitwise XOR |
| != | Assign bitwise OR |
| <<= | Assign left shift |
| >>= | Assign right shift |
| , | Separator of expressions (comma) | 15 | Left to right |

Storage Classes

* *To fully define a variable, one needs to mention not only its ‘type’ but also its ‘storage class’. In other words, not only do all variables have a data type, they also have a ‘storage class’.*
* *The storage classes have defaults. If we don’t specify the storage class of a variable in its declaration, the compiler will assume a storage class depending on the context in which the variable is used.*
* *From C compiler’s point of view, a variable name identifies some physical location within the computer where the string of bits representing the variable’s value is stored.*
* *There are basically two kinds of locations in a computer where such a value may be kept—Memory and CPU registers.*
* *It is the variable’s storage class that determines in which of these two locations the value is stored.*
* Thus, every variable in c language has two properties:

1. type and
2. storage class

* Type refers to the datatype of a variable.
* *A variable’s storage class tells us:*

1. *Where the variable would be stored.*
2. *What will be the initial value of the variable, if initial value is not specifically assigned.(i.e. The default initial value).*
3. *What is the scope of the variable; i.e. In which functions the value of the variable would be available.*
4. *What is the life of the variable; i.e. How long would the variable exist*

* *The storage class precedes the variable's declaration.*

Syntax: Storageclass datatype Variable name;

* This declaration tells the compiler regarding where the variable should be stored, what will be its scope and for how much time it will be available for use.

Note:

* If the storage class in the variable declaration is omitted then.
  + 1. variables declared within a function are considered as an auto by default.
    2. functions declared within a function are considered as an extern by default.
    3. variables and function declared outside a function are considered static with external linkage.
* There are four types of storage classes in c
  + 1. Automatic storage class -Auto
    2. Static storage class -Static
    3. Register storage class-Register
    4. Register storage class-Extern.

1)The auto / Automatic storage class (Local variable)

* The Auto is a default storage class for all the variables declared inside a function or a block.
* The visibility and scope of the automatic variable is limited to the block in which they are defined.
* The automatic variables are assigned a garbage value by default whenever they are declared.
* The automatic variable retains its value till the control remains in the block in which the variable is declared.
* Note: Every local variable is automatic in C by default.
* *Always make it a point that you initialize the automatic variables properly, otherwise you are likely to get unexpected results.*

Syntax: auto datatype variable name;

e.g.: auto int i;

2) The Static storage classes

* The static variables are initialized only once and they exist till the termination of the program.
* The same static variable can be declared many times to allocate the new memory, but can be assigned or initialized only one time.
* The static variables are visible to the block in which the variable is declared.
* The default initial value of the static variable is 0 for integer /flout Variable otherwise its Null.

Syntax: Static datatype variable\_name;

e.g.: static int i;

* *The difference between auto and static is*
* *Static variables don’t disappear when the function is no longer active. Their values persist. If the control comes back to the same function again the static variables have the same values, they had last time around.*
* *If the storage class is static then the statement of initialization of variable in a function is executed only once, irrespective of how many times the same function is called.*
* *Note: avoid using static variables unless you really need them. Because their values are kept in memory when the variables are not active, which means they take up space in memory that could otherwise be used by other variables.*

3) The Register storage class

* The Register storage class is used to define the local variables that should be stored in a register instead of Memory (RAM)
* The value stored in register can be accessed much faster than the value stored in memory.
* So, register should only be used for variables that require quick access.
* Note that we can-not use “&” operator for the register variable.
* The initial default value of the register local variables is 0.
* Note that there are only limited number of registers in a CPU, so not all variables can be a register variables.it is the compiler's choice whether or not, the variable can be stored in the register.
* The Register variable are visible to the block in which they are declared.

Syntax: register datatype variable name;

e.g., register int i;

* Note: *Not every type of variable can be stored in a CPU register.*
* *For example, if the microprocessor has 16-bit registers then they cannot hold a float value or a double value, which require 4 and 8 bytes respectively.*
* *However, if you use the register storage class* *for a float or a double variable you won’t get any error messages.*
* *All that would happen is the compiler would treat the variables to be of auto storage class.*

4) The Extern storage class.

* The Extern Variable is a global variable and used to give a reference of a global variable that is visible to all the program files.
* The extern variables are declared outside all functions and they are accessible from any function inside the program.
* The extern variable retains its value till the program execution terminates.
* The extern keyword signifies that we are not initializing a new Variable but we are accessing the global variable which is declared elsewhere in the program or file
* Note we can-not initialize the external variable within any block.
* An external variable can be declared many times but can be initialized only once.

Syntax: extern datatype variable name;

e.g., extern int i;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Storage specifier | Storage | Initial value | Scope | Life |
| Auto | Memory | Garbage | Within block | End of block |
| Extern | Memory | Zero | Global multiple files | Till end of program |
| Static | Memory | Zero | Within block | Till end of Program |
| Register | CPU register | Garbage | Within block | End of block |

*Which to Use When*

* *Use static storage class only if you want the value of a variable to persist between different function calls.*
* *Use register storage class for only those variables that are being used very often in a program. Reason is, there are very few CPU registers at our disposal and many of them might be busy doing something else.*
* *Use extern storage class for only those variables that are being used by almost all the functions in the program. This would avoid unnecessary passing of these variables as arguments when making a function call. Declaring all the variables as extern would amount to a lot of wastage of memory space because these variables would remain active throughout the life of the program.*
* *If you don’t have any of the express needs mentioned above, then use the auto storage class.*

Type Conversion

* Type conversion in C is the process of converting one date type to another type
* In C there are two types of type conversion

1. Implicit Type Conversion
2. Explicit Type Conversion.

* Note that in type conversion, the destination datatype can't be smaller than the source datatype.

1) Implicit type Conversion

* This is also known as "Automatic type conversion".
* An implicit type conversion is automatically performed by the compiler.
* The value of one type is automatically converted to the valve of another data type.

for example

#include <stdio.h>

int main ()

{

float a=9; // automatic Conversion: int to float

printf ("%f", a);

return 0;

}

Output: 9.000000

* Other example of converting char to int

#include <stdio.h>

Int main ()

{

int num=1;

char a=`K'; //ASCII value of k is 107

int sum;

sum = num + a; //automatic conversion : char to int

printf (" value of sum: %d", sum);

return 0;

}

Output: value of sum:108

Note÷

1) Arithmetic conversion rank

char -> unsigned char → Short → unsigned short → int → unsigned int→ long→ unsigned long→ float→ double →long double

2)We do not require any keyword or special statements in implicit type conversion

3) The implicit type conversion always happens with the, Compatible data types.

4) Converting the data from larger type to smaller, can result in data loss such as

* 1. Converting float to an int will truncate the fraction part
  2. Converting double to float will round up the digits-

for e.g.,

#include <stdio.h>

int main () {

double a=415.12;

int num= a;

printf ("integer value: "%d", num);

printf ("Double value: % 21f", a);

return 0;

}

Output:

integer value: 415

Double value: 415.12

Here data after the decimal (.12) is lost

1) Explicit Type Conversion

* The Explicit type Conversion is user defined. Here the user manually Convert values of one data type to another type
* Explicit conversion is done manually by placing the type parentheses () in front of the value.
* The Syntax for explicit conversion is.

Syntax (type) operand (0R) (type) expression.

for e.g., (float) x

* Example program

#include <stdio.h>

int main ()

{

int num=15;

printf ("integer value: %d\n", num); //explicit type Conversion

float value = (float) num;

printf ("float value: %.2f," value);

return 0;

}

Output:

integer value: 15

float value: 15.00

Standard I/O

* C Programming treats all the devices as files.
* when a c program begins execution, it has access to the following predefined files.
  + 1. Standard input (stdin)- [Key board]
    2. Standard output (stdout) - [screen]
    3. Standard error (stderr) - [Screen]
* Standard Input, is the file from which input is received; this file is usually a keyboard.
* Standard output, is the file to which output is directed; this file is usually a computer screen.
* Standard error; is the file used to keep error messages separate for program's output; this file is usually a computer screen; the user Sees error messages immediately when error occurred.
* The C programming provide a set of standards I/O functions
* These functions are divided into two categories

1. Unformatted I/O functions
2. Formatted I/O functions

1) Unformatted I/O functions

* Unformatted input and output function read a single input sent by the user and display the value as the output at the screen.
* In the unformatted I/O functions we have.

1. Character I/O functions
2. String I/O functions

1) Character I/O functions

* This function performs Input/ output of one character at a time.
* For character input, the function is getchar ()
* For character output, the function is putchar ()
* In addition, C provides two more-character input functions they are getche () and getch ()

1. getchar ()

* getchar () function is the function of standard input (stdin).
* It takes a single character as input at a time and it does not take any parameter.
* The function reads the character and returns it as an integer.
* This function is declared in stdio.h header file.
* And the function inputs the character till the enter is clicked

Syntax

Variable\_name = getchar ();

* for e.g.,

#include <stdio.h>

int main ()

{

char ch;

printf ("Enter the character: ");

ch= getchar ();

printf ("Character entered is: %c" ch);

return 0;

}

Output:

Enter the character: cat

character entered is: C

1. putchar ()

* putchar () function is the function of standard output (stdout)
* It displays only a single character at a time.
* The character which is passed to this function is passed as a parameter.
* The return type of this function is int and it return the ASCII value of the character which is passed to it.
* This function is declared in stdio.h (header file)
* And the function displays the character after clicking enter.

Syntax: putchar (variable\_name);

for example:

#include <stdio.h>

int main ()

{

char ch;

printf ("Enter any character: ");

ch=getchar (); //Reads a characters

putchar (ch); //Displays a character

return 0;

}

Output:

Enter the character: car

C

1. getche ()

* The getche () function is used to take characters from standard input Keyboards and after taking input it immediately prints the output, on output screen. we don't need to hit enter to give command for output.
* This function is declared in conio.h header file.
* Syntax: getche ();

(OR)

Variable\_name = getche ();

for e.g.:

#include <conio.h>

#include <stdio.h>

int main ()

{

printf ("Enter any character: ");

getche ();

return 0;

}

Output:

Enter any character: C

1. getch ()

* getch function reads a single character from the keyboard by the user but doesn't display that character on the screen and immediately returned without pressing enter key.
* This function is declared in conio.h (header file)
* getch () is also used for hold the screen.

Syntax: getch ();

(OR)

Variable\_name= getch ();

* for e.g.,

#include <stdio.h>

#include <conio.h>

int main ()

{

printf ("Enter the character: ");

getch ();

return 0;

}

Output:

Enter the character:

* This input is usually used for system security like for password.

2)String I/O functions

* This function performs Input/output of one string at a time.
* for string input, the function is gets ()
* for string output, the function is puts ()

1. gets ()

* The gets () function is used for the input.
* The gets () function reads a group of characters or strings from Keyboard by the user and these characters get stored, a character array.
* This function allows us to write space and separated texts.
* This function is declared in stdio.h header file
* The return type of this function is string.
* The function takes the input of characters/strings till enter is clicked.

Syntax:

char str\_ Variable\_name [length of string]; //Declare a char type variable

gets (str\_ Variable\_name);

* for e.g.,

#include <stdio.h>

int main ()

{

char name [20];

printf ("Enter the name:");

gets (name);

printf ("You entered: %s", name);

return 0;

}

Output:

Enter the name: ABCDEF

You entered: ABCDEF

1. puts ()

* puts () function is used to display a group of characters or strings which is already stored in a character array.
* This function is declared in stdio.h header file.
* Its return type is integer.
* The function displays the string after clicking enter.

Syntax: puts (variable\_name);

for e.g.:

#include <stdio.h>

int main ()

{

char name [50];

printf ("Enter name:");

gets (name);

puts(name);

return 0;

}

Output:

Enter name: ABCDEF

ABCDEF

2) Formatted I/O function

* The formatted I/O functions are used to take various inputs from the user and display multiple outputs to the user.
* These types of I/O functions can help to display the output to the user in different formats using the format specifier.
* We can format these functions according to our needs.
* The various format specifiers used with these functions are as follows

|  |  |
| --- | --- |
| Format specifiers | Type |
| %d | Integer |
| %c | Character |
| %f | float |
| %s | String |
| %lf | double |

* we have scanf () function for input
* for output, the function is printf ().

1. scanf () function.

* To make the program generally the program itself should ask the user to supply the values through the keyboard during execution. This can be achieved using a function called scanf ().
* The scanf () method reads the input from Standard input (stdin).
* scanf () reads the input and scans that input according to the format provided.
* The scanf () function is used for reading or taking any value from the Keyboard by the user, These values can be of any data type like integer, float, character, string and many more.
* This function is declared in stdio.h header file.
* In scanf () function we use & (address of operator) used to store the variable value on the memory location of that variable.

Syntax: scanf ("format specifier", list of addresses of variables);

* Note that input date must be supplied strictly according to the specified format specifier otherwise results can be very different.

for e.g.,

#include <stdio.h>

int main ()

{

Int a;

float b;

char c;

printf ("Enter the integer a, b and character c: ");

scanf ("%d %f %c", &a, &b, &c);

printf ("%d %f %c", a, b, c);

return 0;

}

Output: Enter the integer a, b and character c: 2 4 g

2 4.000000 g

1. printf () function

* All output to screen is achieved using readymade library functions. One such function is printf (). printf () outputs the values to the screen.
* Printf () can not only print values of variables, it can also print the result of an expression.
* The printf () function writes the output to the standard output
* The printf () prints the value passed as the parameter to it.
* The printf () function used in C to display any type of values like float, integer, character etc.
* The printf () function is declared in stdio.h header file

Syntax: printf ("format specifier", list of Variables);

* The printf () function produces the output according to the format provided.

Unit 3

**Control Statement and Modular Programming**

**Conditional Branching and Loops**

Control Statement

* A computer program is a set of instructions for a computer.
* These instructions, also known as statements, which are executed. Sequentially.
* In practice, it is often required to change the order of execution of Statements or to repeat a group of statements for Known no. Of times or until certain specified conditions are met
* The C language provide facilities for controlling the order of execution of statements which are known as control statements.
* The control statements are divided into three categories:

1. Decision Making / Conditional statements
2. Looping statement
3. Jump control / Unconditional statements.

1) Conditional statements / Decision Making statements.

* Many a times, we want a set of instructions to be executed in one situation, and an entirely different set of instructions to be executed in another situation.
* This kind of situation is dealt in C programs using a decision control instruction.
* The conditional statements are also called as selection statements.
* Conditional statements are statements that test a particular condition Before running the code.
* These statements allow the execution of selective statements based on the outcome of a certain decision.
* The various type of conditional statements supported by 'C' are as follows

1. If statement
2. If else statement
3. Nested if statement
4. Switch statement
5. Else if ladder.

1)If statement

* The if statement evaluates the given test expression inside the Parenthesis ()
* The expression may represent a relations expression, a logical expression, a numeric variable or constant.
* The expression evaluates to a zero (false) or non-zero (True) value.
* If the test expression is evaluated to true, then statements Inside the body of if are executed.
* If the test expression is evaluated to false, then statements inside the body of it are not executed.
* The condition in if can be a simple expression or compound expression.

Syntax

if (this condition is true)

{

execute this block of statements;

}

* Note that opening and closing Braces are required only when code after if statement occupies multiple lines.
* Note that if is in lowercase letters. Uppercase If will generate an error.

2) if else statement

* if else statement is a combination of both if and else statements.
* Using if statement we execute one group of statements if the expression evaluates to true and
* using else statement, we execute another group of statements if the expression evaluates to false.
* The if statement executes a statement, when the specified expression evaluates to a non-zero value. It does nothing when it evaluates to a zero value.
* Where as in if else statement there are two types of blocks: true and false.
* If the condition given in if evaluates to a non-zero value or evaluated to true the statements inside the body of if are executed and the else part is bypassed.
* If the condition given in it evaluates to a zero or evaluated to false the statement inside the body of else are executed and the if part is by passed.
* The group of statements after the if up to and not including the else is called an ‘if block’. Similarly, the statements after the else form the ‘else block’.
* Note: the else is written exactly below the if.

Syntax

If (this condition is true)

{

execute this block of statements;

}

else

{

execute this block of statements;

}

3) Nested if statement

* It is perfectly all right if we write an entire if-else construct within either the body of the if statement or the body of an else statement. This is called ‘nesting’ of ifs.
* Nested if statements refer to it statements that are contained within Other if statements.
* If statement can be contained within another if statement
* The inner statement will be executed if expression of outer if statement evaluates to non-zero value.

Syntax

If (Condition 1)

{

If (Condition 2)

{

Block of statements;

}

else {

Block of statements;

}

}

else

{

Block of statements;

}

4) if-else ladder (if-else if ladder)

* The if else statement executes two different codes depending upon whether the test expression is true or false. Sometimes, a choice has to be made from more than 2 possibilities.
* The If else ladder allows you to check between multiple test expressions and execute different statements.
* The expressions are evaluated in order, and if any expression is true then the statement associated with the block is executed, and terminate whole loop.
* If none of the specified expressions are satisfied then the last else part or default case is executed.

Syntax

If (condition 1)

Statements 1;

else if (condition 2)

Statements 2;

else if (condition 3)

Statements 3;

-

-

else if (condition n)

Statements n;

else

Statements;

5) Switch statement

* In real life we are often faced with situations where we are required to make a choice between a number of alternatives rather than only one or two.
* In C programming the choice we are asked to make is more complicated than merely selecting between two alternatives. C provides a special control statement that allows us to handle such cases effectively.
* The control statement that allows us to make a decision from the number of choices is called a switch, or more correctly a switch-case default, since these three keywords go together to make up the control statement.
* The switch statement provides an alternative to else if statement.
* A case expression can be repeatedly used in a switch statement.
* Switch statement works by testing desired case among all the given Cases
* When the case is found the block of statement associated with that Case is executed.
* The use of break statement in every case is used to quit the switch Statement after a particular case is matched. Thus, only one case gets executed because break takes control out of the loop.
* If The break statement is not used, then all the statements following the matched cave will get executed.
* If the value of expression does not match any of the case, control goes to the default keyword. And execute the statements.
* If there is no default keyword, the whole switch statement Simply terminates when there is no match.

Syntax

switch (integer expression)

{

case constant 1:

Statement 1;

break;

case constant 2:

Statement 2;

break;

default:

Statements;

}

* The integer expression following the keyword switch is any C expression that will yield an integer value. It could be an integer constant like 1, 2 or 3, or an expression that evaluates to an integer.
* The keyword case is followed by an integer or a character constant. Each constant in each case must be different from all the others.
* When we run a switch case statement, First, the integer expression following the keyword switch is evaluated.
* The value it gives is then matched, one by one, against the constant values that follow the case statements. When a match is found, the program executes the statements following that case,
* If no match is found with any of the case statements, only the statements following the default are executed.

Note:

* there is no need for a break statement after the default, since the control comes out of the switch anyway.
* Even if there are multiple statements to be executed in each case there is no need to enclose them within a pair of braces.
* Every statement in a switch must belong to some case or the other. If a statement doesn’t belong to any case the compiler won’t report an error. However, the statement would never get executed.
* If we have no default case, then the program simply falls through the entire switch and continues with the next instruction (if any,) that follows the closing brace of switch.
* The disadvantage of switch is that one cannot have a case in a switch which looks like:

case i <= 20:

* There are some things that you simply cannot do with a switch. These are:

1. A float expression cannot be tested using a switch
2. Cases can never have variable expressions (for example it is wrong to say case a +3 :)
3. Multiple cases cannot use same expressions. Thus, the following switch is illegal:

switch (a)

{

case 3:

...

case 1 + 2:

...

}

* When cases are more Switch works faster than an equivalent if-else ladder. This is because the compiler generates a jump table for a switch during compilation. But in cases of less than 4 or 5 cases if else ladder works faster.

2) Looping / Iterative Statements

* In General cases we need to perform an action over and over, often with variations in the details each time. The mechanism, which meets this need, is the loop.
* The versatility of the computer lies in its ability to perform a set of instructions repeatedly. This involves repeating some portion of the program either a specified number of times or until a particular condition is being satisfied. This repetitive operation is done through loop control Instruction.
* Looping statements are used for running a particular set of code for any number of times
* Loops are used to repeat a block of code until a specified. Condition is met.
* The various iterative statements & supported by 'c' are as follows

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

* Three things loop contain

1. Setting a loop counter to an initial value. (Initialization of variable)
2. Testing the loop counter to determine whether its value has reached the number of repetitions desired. (Testing the condition)
3. Increasing the value of loop counter each time the program segment within the loop has been executed. (Incrementation/decrementation)

1) while loop

* It is often the case in programming that you want to do something a fixed number of times. The while loop is suited for such cases.
* The while statement used when it is not known in advance that how many times a Statement or statement-block will be executed.
* The main function of the while statement is to repeat a particular set of statements. For a fixed number of times till condition is met
* In the while statement the initialization of looping variables, the testing of a condition and incrementation of loop variable occurs at different place in the program.
* There must be a statement prior to while statement that initializes the expression and, in the statement block, there must be Statement that modifies (i.e., Increment /decrement) the expression.

Syntax

initialise loop counter;

while (test loop counter using a condition)

{

Statements;

Modifications loop counter;

}

* The while loop contains expression or test condition
* The statement is executed repeatedly till the expression evaluate to a non-Zero value
* firstly, the value of variable (say i) is set to an initial value 1. When the while statement is executed for the first time, the condition is tested.
* If the condition is satisfied and then the body of the loop is executed for the first time.
* Then the value of I variable gets incremented/decremented
* Upon reaching the closing brace of while, control is sent back to the while statement.
* Again, the test is performed to check whether the new value of i satisfy the condition
* If the value of i satisfy the condition, the statements within the braces of while are executed again. The body of the while loop continues to get executed till the condition being tested remains true. when the condition becomes false, the control passes to the first statement that follows the body of the while loop.
* NOTE

1. The variable 'i' is many a times called either a ‘loop counter’ or an ‘index variable’.
2. Other than condition in place of the condition there can be any other valid expression. So long as the expression evaluates to a non-zero value the statements within the loop would get executed.
3. The condition being tested may use relational or logical operators
4. The statements within the loop may be a single and a block of statements
5. As a rule, the while must test a condition that will eventually become false otherwise the loop would be executed forever, indefinitely.
6. It is not necessary ta loop counter must only be an int. It can also be a float. Even floating-point loop counters can be decremented.
7. Once again, the increment and decrement could be by any value, not necessarily 1.
8. if we use i++ ' ++' operator is called a post incrementation operator. With this, incrementation of i happens after its usage or comparison.
9. if we use ++i '++' operator is called a pre incrementation operator. With this, incrementation of i happens before its usage or comparison.

2) for loop

* The for statement is used for problems where the number of times a statement or statement -block will be executed is known in advance
* For statement can be treated as a compact form of while statement.
* The for allows us to specify three things about a loop in a single line:

1. Initialization of variable

2. Testing the condition

3. Incrementation/decrementation.

Syntax

for (initialise counter; test counter; increment counter)

{

Statements;

}

* When the for statement is executed for the first time, the value of variable (say i) is set to an initial value 1.
* Then the condition is tested. If the condition is satisfied and the body of the loop is executed for the first time.
* Upon reaching the closing brace of for, control is sent back to the for statement, where the value of i variable gets incremented/decremented
* Again, the test is performed to check whether the new value of i satisfy the condition. If the value of i satisfy the condition, the statements within the braces of for are executed again.
* The body of the for loop continues to get executed till the condition being tested remains true. when the condition becomes false, the control passes to the first statement that follows the body of the for loop.
* In the initialization and update expression parts can have more than one statement separated by a comma.

Note:

* Neither the initialisation, nor the incrementation is done in the for statement, but still the two semicolons are necessary.
* The way if statements can be nested, similarly two or more whiles and for's can also be nested. And a for loop can occur within a while loop, or a while within a for.
* The initialisation expression of the for loop can contain more than one statement separated by a comma.

For example,

for (i = 1, j = 2; j <= 10; j++)

* Multiple statements can also be used in the incrementation expression of for loop;

i.e., you can increment (or decrement) two or more variables at the same time.

* However, only one expression is allowed in the test expression. This expression may contain several conditions linked together using logical operators.

3) do while loop.

* The do-while statement is also used when it is not known in advance that how many time a statement will be executed.
* The do-while bop is similar to the while statement, but the only difference between the two is that the testing condition is present at the end of do while statement; whereas the while statement in which It is present at the beginning.
* The body of do while loop is executed at least once only then, the test expression is evaluated.

i.e., the body of loop is executed once though the condition is false.

Syntax

do

{

//statements;

} while (test expression);

* The do while loop executes at least once i.e., the first iteration runs without checking the condition. The condition is checked only after the first iteration has been executed.
* If the condition is true, then body of loop is executed again. The Statements are executed repeatedly till the exp evaluates to non-zero value
* When condition is false the loop is terminated.
* There is a minor difference between the working of while and do-while loops.

|  |  |
| --- | --- |
| While loop | Do-while loop |
| 1.While loop is an entry loop because firstly, the condition is checked then the loop’s body is executed. | 1.do while loop is an exit control loop because in this, first of all, the body of the loop is executed then the condition is checked. |
| 2.The statement of while loop may not be executed at all if condition is false. | 2.The statement of the do while loop must be executed at least once |
| 3.in while loop, at the end of the condition there is no semicolon. | 3. in do while loop at the end of the condition there is a semicolon. |
| 4.in a while loop, the number of executions depends on the condition defined in the while block. | 4.in a do while, irrespective of the condition mentioned, a minimum of 1 execution occurs. |
| 5.syntax  while (condition)  {  Block of statements;  }  Statement; | 5.Syntax  do  {  Statements;  }  while(condition); |

Note The Odd Loop

* when it is not known beforehand how many times the statements in the loop are to be executed. This situation can be programmed as shown below:

Example.

#include<stdio.h>

main ()

{

char another = 'y';

int num;

for (; another == 'y' ;)

{

printf ("Enter a number ");

scanf ("%d", &num);

printf ("square of %d is %d", num, num \* num);

printf ("\n Want to enter another number y/n ");

scanf (" %c", &another);

}

}

* Here loop would keep getting executed till the user continues to answer y. The moment he answers n, the loop terminates.

3) Jumping / Unconditional statement

* The jumping statements used to transfer the control from one part of program to another specified part in the program.
* The various Jumping statements supported by "c" are as follows.
  + 1. break Statement
    2. continue statement
    3. go to statement
    4. return statement

1) break statement

* We often come across situations where we want to jump out of a loop instantly, without waiting to get back to the conditional test. The keyword break allows us to do this.
* The break statement terminates the execution of the loop and the Control is transferred to the statement immediately following the loop.
* The break statement is used to terminate the loop and transfer the control outside the loop.
* It can be used within for, while, do-while or switch statement.
* Break is usually associated with an if.

Syntax

break;

2) continue statement

* In some programming situations we want to take the control to the beginning of the loop, bypassing the statements inside the loop, which have not yet been executed. The keyword continue allows us to do this.
* Continue statement is used to pass the control for next repetition of loop, by terminating the current iteration of loop.
* The Continue statement breaks one iteration in loop, if a Specified condition occurs, and continues with the next iteration. In the loop.
* The continue statement transfers the control to the beginning of the next iteration of the loop bypassing the statements which are not yet executed.
* A continue usually associated with an If

Syntax

continue:

3)go to statement

* In a difficult programming situation, it seems so easy to use a go-to to take the control where you want.
* The big problem with go-to is that when we do use them, we can never be sure how we got to a certain point in our code. They Disturb the flow of control. So as far as possible skip them.
* The only programming situation in favour of using go-to is when we want to take the control out of the loop that is contained in several other loops.
* The usage of the go-to keyword should be avoided as it usually violets the normal flow of execution.
* A go to statement provides an unconditional jump from the go to statement to a labelled statement in the same function.
* The go to statement allows us to transfer control of the program to the specified label.
* Go to statement uses the identifier label, in order to locate the particular statement, where the control is to be transferred.
* The label is a valid variable name followed by a colon and put Immediately before the statement where the control needs to be Jumped unconditionally
* When go to statement is encountered the control of program jumps to label and starts executing the code.

Syntax

go to label:

-

-

Label: Statement;

(OR)

Label:

-

-

go to label;

Statement;

4) return statement

* The return statement returns the flow of the execution to the function from where it is called.
* The execution of return statement immediately transfers control from the function back to the calling function.
* The return is used to return the value from called function back to the calling function.
* There is no restriction on the number of return statements that may be present in a function. Also, the return statement need not always be present at the end of the called function.
* Note that a function can return only one value at a time.
* If we want that a called function should not return any value, we must use void return type.
* For non-void function a return value must be returned.

Syntax

return (expression);

Or

return (variable /value);

Functions

* Computer Cannot handle all the tasks by itself. Instead, it requests, other program like entitles called "functions' in C to get its tasks done.
* The best way to handle large and complex problems is to split the problem into small problems, called Subproblems. That can be handled easily.
* The process of subdividing a problem into manageable parts top-down design. This process is also called decomposition.
* "The top-down design is done using function. A function in c is an independent module that will be called to do a designated task?"
* A function is a self-contained block of statements that perform a designated task of some kind." Every c program can be thought of as a collection of these functions.”
* Main is the one & important function that is used in every c program.

Definition: A function is a block of code which only runs when it is called.

Syntax

main ()

{

Function call; //function name ();

}

Function definition //function name ()

{

Statements;

}

Note:

* main () is a function and through it we will be calling the other function to do specific task.
* What do we mean when we say that main () ‘calls’ the function?
* We mean that the control passes to the function. The activity of main () is temporarily suspended; it falls asleep while the function wakes up and goes to work. When the function runs out of statements to execute, the control returns to main (), which comes to life again and begins executing its code at the exact point where it left off.
* Thus, main () becomes the ‘calling’ function, whereas and function called becomes the ‘called’ function.

Why Use Functions

* Why write separate functions at all? Why not squeeze the entire logic into one function, main ()? Two reasons:
  1. Writing functions avoids rewriting the same code over and over.
  + Suppose you have a section of code in your program that calculates area of a triangle. If later in the program you want to calculate the area of a different triangle, you won’t like it if you are required to write the same instructions all over again. Instead, you would prefer to jump to a ‘section of code’ that calculates area and then jump back to the place from where you left off. This section of code is nothing but a function.
  1. Using functions, it becomes easier to write programs and keep track of what they are doing.
  + If the operation of a program can be divided into separate activities, and each activity placed in a different function, then each could be written and checked more or less independently.
  + Separating the code into modular functions also makes the program easier to design and understand
* Note
  + Don’t try to cram the entire logic in one function. It is a very bad style of programming. Instead, break a program into small units and write functions for each of these isolated subdivisions

Advantages of Functions

The advantages of using functions are:

* Increases program readability: separating the code into smaller parts or modules makes the program easter to understand
* Avoid repetition of codes: writing functions avoids rewriting the same code over and over
* Reduces chances of error: Using functions it becomes easier to write programs and keep track of what they are doing
* Reusability of code: using function we define the code Once and use it many times it saves time and memory.
* Modifying a program becomes easier by using function.

Function Calling/Invoking / Accessing

* A function in c is an independent module that will be called to do a designated task
* A called function receives control from a calling function. When the called function completes its task, it returns control to the Calling function.
* It may or may not return a value to calling function.
* The main function is also called by the operating system; main in turn calls other functions. Those in turn call other functions.
* When main runs out of function calls and its execution, Control returns to operating system.
* Therefore, every function in a program must be called directly or indirectly by main () function. In other words, the main () function drives other functions.
* Note, that Each function in a program is called in the sequence Specified by the function calls in main ().
* A function gets called when the function name is followed by a semicolon.
* A function can be called any number of times.
* The order in which the functions are defined in a program and the order in which they get called need not necessarily be same. However, it is advisable to define the functions in the same order in which they are called. This makes the program easier to understand.

Syntax

function\_name (parameters); //actual arguments

Example: big = largest (a, b, c);

Function Declaration

* Like every other data item, function also need to be declared.
* The function declaration, which needs to be done before the function call, gives the whole picture of the function that needs to be defined later

Syntax

return\_type function\_name (actual argument list);

E.g.: int largest (int a, int b, int c);

* The declaration mentions the name of the function, the return type, and the type and order of formal arguments. A semicolon follows the function declaration.
* If the function has no argument, then we write void in the parentheses. If the function has more arguments, each is separated by comma.
  + A function declaration statement is also known as prototype of the function or as function prototype"
* Any C function by default returns an int value whenever a call is made to a function. If we desire that function should return a value other than int, then it is necessary to explicitly mention in the calling function as well as in the called function.

Function definition

Function definition consists of two parts:

Function header and function body

Function header

Syntax: returntype function name (formal argument list)

* Note that A semicolon is not used at the end of the function definition
* The formal argument list declares the variables that will receive the data from the calling function.

Function body

* Function body contains the local declarations and the statements that define the task to be accomplished by the function.
* The body starts with local declarations that specify the variable needed by the function. After the local declarations, the function statements followed by the return statement are written
* If the return type of function is void, it can be written without return statement.

Function definition Syntax

//function header

Returntype function Name (formal argument list)

{ // function body

// local declarations

- - -

// Statements

}

* Note: A function can be called from other function, but a function cannot be defined in another function

Passing Values between Functions

* + To communicate between the ‘calling’ and the ‘called’ functions. We use the mechanism to convey information to the function is the ‘argument’.
  + We have unknowingly used the arguments in the printf () and scanf () functions;
  + the format string and the list of variables used inside the parentheses in these functions are arguments. The arguments are sometimes also called parameters.
  + The variables in function call are called ‘actual arguments’, whereas the variables in function definition are called ‘formal arguments.
  + Any number of arguments can be passed to a function being called.
  + the type, order and number of the actual and formal arguments must always be same.
  + Instead of using different variable names in function definition, we can use the same variable names. But the compiler would still treat them as different variables since they are in different functions.
  + There are two methods of declaring the formal arguments.
  + For e.g.,

sum (x, y, z)

int x, y, z;

(Or)

sum (int x, int y, int z)

Return Statement

The return statement serves two purposes:

* Execution of return statement immediately transfers control from the function back to the calling function.
* Whatever is following the return statement is returned as a value to the calling function.

The Syntax of return statement is

Return; (or) return (exp);

E.g.: return big;

* Where expression can be a constant, variable or expression.
* The return statement need not be at the end of the function. It can be used anywhere in the function. As soon as it is executed, the control will return to the calling function. A function can contain any number of return statements.
* There is key limitation of return statement - it can return only one value. If you want your function to return two or more Values to the calling function, you need another mechanism.

Note

* There is no restriction on the number of return statements that may be present in a function. Also, the return statement need not always be present at the end of the called function.
* If the value of a formal argument is changed in the called function, the corresponding change does not take place in the calling function.
* the scope of a variable is local to the function in which it is defined.

Calling Convention

* Calling convention indicates the order in which arguments are passed to a function when a function call is encountered.
* There are two possibilities here:

(a)Arguments might be passed from left to right.

(b)Arguments might be passed from right to left.

* C language follows the second order.
* Consider the following function call:

fun (a, b, c, d);

* In this call it doesn’t matter whether the arguments are passed from left to right or from right to left.
* However, in some function call the order of passing arguments becomes an important consideration.
* For example:

int a = 1;

printf ("%d %d %d", a, ++a, a++);

* It appears that this printf () would output 3 3 1
* First 1 is passed through the expression a++ and then a is incremented to 2. Then result of ++a is passed. That is, a is incremented to 3 and then passed. Finally, latest value of a, i.e., 3, is passed. Thus, in right to left order 1, 3, 3 get passed.

Function Calls

* There are two types of function calls
  + 1. call by value and
    2. call by reference.
* Arguments can generally be passed to functions in one of the two ways:

1. sending the values of the arguments i.e., call by value
2. sending the addresses of the arguments i.e., Call by reference.

1.Call by Value

* In the first method the ‘value’ of each of the actual arguments in the calling function is copied into corresponding formal arguments of the called function.
* With this method the changes made to the formal arguments in the called function have no effect on the values of actual arguments in the calling function
* So, whenever we called a function and pass the ‘values’ of variables to the called function. Such function calls are called ‘calls by value’.
* By this what we mean is, on calling a function we are passing values of variables to it.
* The actual arguments can be constants, variables or expressions.
* For example,

a = swap (x, y); // function call

void swap (int a, int b) // function definition.

* Note that values of variables remain unchanged even after exchanging the values of variable
* Example (Call by value)

/\* program to illustrate the passing of arguments by value \*/

#include <stdio.h>

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

int main ()

{

int x= 10, y = 20;

printf ("Before swapping”);

printf ("\n x = %d \t y = %d", x, y);

swap (x, y); //function call

printf (“\n After swapping----values in main function”);

printf ("\n x = %d \t y = %d", x, y);

return 0;

}

void swap (int x, int y) //function definition

{

int temp;

temp = x;

x = y;

y = temp;

printf ("\n After swapping---values in swap function”);

printf ("\n x = %d \t y = %d", x, y);

}

* The output of the above program would be:

Before swapping

x = 20 y = 10

After swapping---values in swap function

x = 10 y = 20

After swapping----values in main function

x = 20 y = 10

2.Call by reference

* In the second method (call by reference) the addresses of actual arguments in the calling function are copied into formal arguments of the called function.
* This means that when the addresses of actual arguments are substituted in corresponding formal arguments using these addresses, we would have an access to the actual arguments and hence we would be able to manipulate them
* Note in this method program manages to exchange the values of variables using their addresses stored in variables
* Therefore, any changes made to the formal argument will have immediate effed on actual arguments.
* The actual arguments can only be variables and the formal arguments are declared as pointers.
* For e.g., swap (&x, &y); //function call
* void swap (int \*a, int \*b) // function definition.
* Example (Call by reference)

/\*program to illustrate the passing of arguments by references\*/

#include <stdio.h>

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

int main ()

{

int x= 10, y = 20;

printf ("Before swapping”);

printf ("\n x = %d \t y = %d", x, y);

swap (&x, & y); //function call

printf (“\n After swapping----values in main function”);

printf ("\n x = %d \t y = %d", x, y);

return 0;

}

void swap (int \*x, int \*y) //function definition

{

int temp;

temp = \*x;

\*x = \*y;

\*y = temp;

printf ("\n After swapping---values in swap function”);

printf ("\n x = %d \t y = %d", \*x, \* y);

}

* The output of the above program would be:

Before swapping

x = 20 y = 10

After swapping---values in swap function

x = 10 y = 20

After swapping----values in main function

x = 10 y = 20

* کو

Note:

* From the programs that we discussed here we can draw the following conclusions:
* If we want that the value of an actual argument should not get changed in the function being called, pass the actual argument by value.
* If we want that the value of an actual argument should get changed in the function being called, pass the actual argument by reference.
* If a function is to be made to return more than one value at a time then return these values indirectly by using a call by reference.

Difference between Call by Value and call by Reference

|  |  |
| --- | --- |
| Call by value | Call by Reference |
| 1. Actual arguments can be constants, variables or expressions. | 1. Actual arguments can only be Variables. |
| 1. Formal arguments are ordinary variables. | 1. Formal arguments are pointer variables. |
| 1. Values of actual argument are Substituted in formal arguments. | 1. Addresses of the actual arguments are substituted in formal arguments |
| 1. It requires more memory | 1. It requires less memory |
| 1. Any change made to formal arguments will have no effect on actual arguments, since the function will only be using the local copy of arguments. | 1. Any change made to formal arguments will have immediate effect on actual arguments, since the function will be working on actual arguments through address. |
| 1. function Prototype:   void swap (int a, int b); | 1. function Prototype:   void swap (int \*a, int \*b); |
| 1. function definition:   void swap (int a, int b).  {  int temp;  temp = a;  a=b;  b=temp;  } | 1. function definition:   void swap (int \*a, int \*b).  {  int temp;  temp = \*a;  \*a= \*b;  \*b=temp;  } |
| 1. function call   swap (x, y); | 1. function call   swap (&x, &y); |

Recursion

* In c, it is possible for the functions to call themselves.
* A function is called "recursive" if a statement within the body of function calls the same function.
* Sometimes it is also called circular definition, therefore recursion is the process of defining something in terms of itself.
* The recursive function is the function that calls itself with argument smaller than its original argument.
* Note that while writing recursive function you must have an if statement having base condition some-where in the recursive function to force the function to return without recursive call being executed.
* Otherwise, you will fall in an indefinite loop.

Advantages of recursive function

1. It makes code easier to write
2. Solves naturally recursive problems

Disadvantages of recursive function

1. Uses more memory
2. Recursion can be slow
3. Difficult to analyse code

For Example

/\*Following is the recursive version of the function to calculate the factorial value\*/

#include<stdio.h>

int rec (int num);

int main ()

{

int num, fact;

printf ("\n Enter any number ");

scanf ("%d", &num);

fact = rec (num);

printf ("Factorial value = %d", fact);

return 0;

}

int rec (int num)

{

if (num == 1)

return (1);

else

return (num\* rec (num - 1));

}

* And here is the output of the program

Enter any number 5

Factorial value = 120

* Explanation

rec (5) returns (5 times rec (4),

which returns (4 times rec (3),

which returns (3 times rec (2),

which returns (2 times rec (1),

which returns (1)))))

Passing Array as Argument to function

* Instead of passing values of all elements of an array to a function, only address of the array is passed.
* Since the name of an array is the address of the first element of the array, only the name of the array is passed.

For example

#include <stdio.h>

/\* function declaration \*/

void fun (int x [], int m);

void main ()

{

int x [10], m;

/\* other local declarations/

/\* function call \*/

fun (x, m);

/\* other statements \*/

}

/\* function definition \*/

void fun (int x [], int m).

{

/\* local declarations/

/\* other statement\*/

}

Pointers as Function Arguments

* We know that the arguments to a function could be passed using either call by value or call by reference mechanism.
* when we use call by reference mechanism, the formal arguments in function definition are declared as pointers and at execution time, the addresses of the actual arguments are copied into formal arguments.
* For example

#include <stdio.h>

/\*function declaration\*/

void swap (int \*x, int \*y);

void main ()

{

int x, y:

/\* other local declarations /

/\* function call/

swap (&x, &y);

/\* other statements \*/

}

/\* function definition \*/

void swap (int \*x, int \*y)

{

/\*local declarations\*/

/\* other statement\*/

}

Passing String to a function

A String can be passed as an argument to a function using subscripted notations for arrays.

In a function definition, the formal parameter is declared as an array of characters.

For example

#include <stdio.h>

/\*function declaration \*/

void fun (char str []);

void main ()

{

char str [] ="Sample string”;

fun (str); /\* function call/

}

فى

/\* function definition \*/

void fun (char str[])

{

/\* other statements \*/

}

Pointers

* Each memory location in computer's memory is assigned a unique number called location number or address.
* Each data operand is stored in a data cell and the system associates each of the variables with these addresses.
* In order to access the data operand, either we use variable name or address of the memory cell.
* Since addresses are simply positive integer numbers, they can also be assigned to some variables and stored in memory, like any other variable.
* Such a variable that holds an address in known as a pointer.

Definition: pointer is a variable that contain address of another variable in memory.

* Note: pointer is also a variable, so its value is also stored in memory in another address.

Accessing Address of a Variable

* To determine the address of variable we use address of operator (&) using & operator we will assign address of variable to the pointer.
* Note

1. The address of operator can only be used with variables or an array element.
2. pointer variable can be initialized only with the use of address of operator or by assigning the value of another Compatible Pointer variable.

Declaring and Initializing Pointers

Declaring a pointer variable

* like other variables, the pointer variables are to be declared first in the declaration block to tell the compiler that to which kind of values these variables will be pointing.
* The Syntax for declaring pointer variable is follows:

datatype \* pointer\_name;

for e.g., int \* ptr1;

* *The declaration int \* ptr1 does not mean that ptr1 is going to contain an integer value. What it means is, ptr1 is going to contain the address of an integer value.*
* *This declaration tells the compiler that ptr1 will be used to store the address of an integer value. In other words, ptr1 points to an integer.*
* Where datatype is a predefined or user defined data types. and indicates that pointer will point to the variables of that Specific datatype, and character ‘\*’ indicates that variable is a pointer variable.
* When we declare a variable, computer allocates memory of appropriate size according to the data type of a variable.
* *Pointers are variables that contain addresses, and since addresses are always whole numbers, pointers would always contain whole numbers.*

Initialization of pointer variable

* Pointer Initialization is a process of assigning address of a variable to pointer variable.
* To determine the address of variable the address operator ‘&’ is used.

Syntax

Pointer\_name = &Variable;

e.g., ptr =&a;

* Once a pointer has been declared, it must be initialized prior to its use. It is important to note here that like other variables, a pointer variable will take a garbage value, which can be an address of any storage location.
* Therefore, if not properly initialized, a pointer may point to a location in memory including those locations where operating system is running, and your system may hang-up.
* Such un-initialized pointers are sometimes referred to as dangling / wild pointers.

Accessing a Variable through its Pointer.

* Once a pointer has been assigned the address of a variable. We access the value of the variable using the pointer. and this can do using indirection operator (\*).
* When indirection operator (\*) used with pointer variable, it refers to variable being pointed to.
* You can get the value of variable the pointer points to by using the ‘\*’ operator.

Pointer Arithmetic

* The following operations are permitted on pointers

1. Addition of a number to a pointer variable.

For e.g., Suppose p is a pointer variable pointing to an element of integer type, then the statement

p++; or ++p; or p+=1.

* Increments the value p by a factor of 2, so that it points to next location that holds another value of integer type.

1. Subtraction of a number from a pointer variable

For e.g., Suppose p is a pointer variable pointing to an element of integer type, then the statement

p--; or –p; or p-=I;

* decrements the value of pay a factor of 2, so that it now points to the location preceding the current location.

1. Subtraction of one pointer variable from another

* One pointer variable can be subtracted from another provided both point to the same datatype.
* The difference of the two indicates the number of bytes separating the corresponding elements.

1. Pointer variables can be compared with one another, provided they are of same data type.

* The following arithmetic operations are not permitted on pointers

1. Addition of two pointer variables
2. multiplication of a pointer variable by a number
3. Division of a pointer variable by a number.

Pointer to a Pointer

* *The pointer can also contain another pointer’s address*
* We can have a variable that hold an address of a variable that in turn holds an address of another variable. This type of variable is known as pointer to a pointer. it is also called double indirection.
* This concept can be further extended to triple level, fourth level and so on.
* A pointer to a pointer will be declared as follows

Syntax

datatype \*\* pointer\_name;

for e.g., int \*\*ptr;

* Value pointed to by a pointer to another pointer will be accessed as follows

Syntax

\*\*pointer\_name;

e.g., \*\*ptr;

Pointer and arrays.

* When we declare an array, the compiler allocates a block of memory of appropriate size so that all the elements of an array can be stored in contiguous memory locations.
* The address of the first memory location of allocated block is known as the base address, and is assigned to the array name.
* Thus, array name behaves as a pointer variable.

Pointers and one-dimensional Arrays

* The array name holds the address of the first element of the array i.e., base address of the array.
* By adding index of the element to this base address the other elements of the array can be accessed

For e.g., \*(a+0), \*(a+1), \* (a +2), - -

e.g.: int arr [10] = {12, 15, 20, 17};

ptr=arr;

for (int sum=0, i=0; i<10; i++)

sum = sum+ \* (ptr + i);

Pointers and two-dimensional Arrays

* The array name holds the address of the first element of the array.
* The elements of the array, irrespective of that whether it is one dimensional, two dimensional or three dimensional, are stored in a continuous chain one after the other.
* Suppose we want to access the jth element of the ith row. Then if a[i] is the address of the ith row, the address of jth element of ith row will be (a[i]+j).
* Thus, the element can be accessed as \* (a[i]+j).
* Since a[i] in subscript notation is equivalent to \*(a+ i).
* Thus, the following expression refer to element aij

\* (\* (a + i) + j)

Pointers and strings

* The more practical and efficient way of string a list of string is using array of pointers to strings.
* The following declaration is e.g., of pointers and strings

e.g.: char \*name [4] = {"Ram", "Inder Mohan", "Amanpreet", "Rajan"}

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

UNIT-IV

User Defined Datatypes & Pointers

Arrays

* The C language provides a capability that enables the user to design a set of similar data types, called array.
* Definition: An array is a collection of homogeneous data elements described by a single name.
  + In other words, an array is a collective name given to a group of elements of same data type.
  + Each individual element of an array is referenced by a subscripted variable.
* The array Is a derived datatype that can store values of Characters, Integers and floats a in the continuous memory locations.
* Note that the elements in an array share the same name but at different index number.
* *Each member in the array is referred to by its position in the array*
* *In C the counting of elements begins with 0 and not with 1.*

One dimensional Array

* If single subscript is required to refer to an element of an array, The array is known as one dimensional array or linear array.
* In one dimensional Array the data are organized linearly only in one direction.
* One dimensional array is a collective name given to a group of elements of same datatype and whose individual elements can be referenced using only one subscript.

Array declaration and definition

* *To begin with, like other variables an array needs to be declared so that the compiler will know what kind of an array and how large an array we want.*
* An array must be declared and defined before it can be used.
* Array declaration and definition tell the compiler the name of the array, the type of its elements and the size or number of elements in the array.
* Note that the size of the array is a constant and must have a value at compilation time.

Syntax for declaration of one-dimensional array

data-type array\_Name [array\_size];

* + - For e.g., int marks [10];
    - Here, datatype-specifies the type of variables
    - Array name -Specifies the name of variable
    - Areay size - Specifies the size of an array
    - Brackets []- tells the compiler that it is an array.

*Array Elements in Memory*

* *Consider the following array declaration:*
* *int arr [8];*
* *What happens in memory when we make this declaration?*
* *16 bytes get immediately reserved in memory, 2 bytes each for the 8 integers and since the array is not being initialized, all eight values present in it would be garbage values.*
* Storage class of array is assumed to be auto. If the storage class is declared to be static then all the array elements would have a default initial value as zero.
* Whatever be the initial values, all the array elements would always be present in contiguous memory locations,

Accessing Elements of an Array

* The Single index value is used to access individual element in a one-dimensional array.
* The index value must be an integral value or an expression that evaluates to an integral value.
* The array's name is a reference for the address to the first Byte of the block of memory allocated for the array.
* The address of the first byte for the array is known as base address of the array.
* Note that whenever we use the array's name, we actually refer to the first byte of the array.
* Note: the first Index value of first element of array is Zero and the last index value is n-1.

Storing Values in Array

* + Declaration only reserve space for the elements of the array. No values are Stored in the array at time of declaration
  + To store value in the array We must follow one of the three methods.

1. Initialization
2. Inputting values
3. Assigning values.

1.Initialization

* The elements of the arrays can also be initialized for each element in the array we provide a value.
* To Initialize an array, we provide value which are enclosed in braces, and if there is more than one, they are separated by comma.
* The following are possible ways of initializing one dimensional array.

e.g., int b [10] = {12,0,14, -4, 7, 8, 10, 11, 9, 15};

(or)

int b [] = {12,0,14, -4, 7, 8, 10, 11, 9, 15};

In this case since size of array is not specifies, the Compiler Counts the number of elements in the initialization list and fixes that as the array size.

* *Note the following points :*
  + - * *Till the array elements are not given any specific values, they are supposed to contain garbage values.*
      * *If the array is initialised where it is declared, mentioning the dimension of the array is optional*

2.Inputting Values

* + The second way to store values in the array, is to read the values from the keyboard or a data file.
  + This method of inputting values is done using a loop.
  + The following is a way to input the values into one dimensional array

for (i=0; i<n; i++)

{

scanf ("%d", &a[i]);

}

3.Assigning Values

* The third way to store values in the array is to assign the values to the individual elements using the assignment operator.
* The following is the way to assign the value to any element.

a [3] = 45;

* Note- Note that one array cannot be copied to another using assignment.

Outputting Values of an array

* To output values of elements of an array, we use the loop as follows

for (i=0; i<n; i++)

{

printf ("%d", a[i]);

}

printf (" \n");

Two-Dimensional Array

* The two-dimensional array can be defined as an array which stores data in tabular form
* The two-dimensional array is also called a matrix.
* We can also consider two dimensional as an array of arrays.
* *Two-dimensional array is nothing but a collection of a number of one-dimensional arrays placed one below the other*

Declaration and Definition of 2-D array

* like one-dimensional array, two-dimensional array must be declared before using.
* Declaration tells the compiler the name of the array, the type of its elements and the size of each dimension.

Syntax for declaration of 2-D array

datatype array\_Name [Size of Dimension 1 (row)] [Size of Dimension 2(Column)];

e.g., int table [4] [4];

* The declaration statement allocates a contiguous memory block.
* Elements of two-dimensional array are stored row-wise i.e., in the allocated contiguous block of memory, first elements of first row are stored, then elements of second row, then elements of third row and so on.

Accessing Elements of 2-D array

* We use two indexes to access individual element in a two-dimensional array.
* The first index is used to denote the row and the second index to denote the column.

Storing Values in 2-D array

* There are three different ways to store values in the array

1. Initialization
2. Inputting values
3. Assigning values.

1.Initialization

* There are two-different ways to initialize 2-D array
* The first way is as follows

e.g.: int table [4][4] = {0, 0, 0, 0, 1,1, 1, 1, 2, 2, 2, 2, 3, 3, 3,3};

* In above initialization, we initialize all elements row wise separating each value with commas.
* The second way and the best recommended way is to use the nest braces to initialize values.

e.g.,

int table [4] [4] = {{0,0,0,0},

{1,1,1,1},

{2,2,2,2},

{3,3,3,3}

}

* In above initialization we initialize each row are one-dimensional array.
* Note that we use Commas between the elements in the Columns and also commas between the rows.
* Note.
* It is important to remember that while initializing a 2-D array it is necessary to mention the second (column) dimension, whereas the first dimension (row) is optional.
* Thus, the declarations,

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

* are perfectly acceptable, whereas,

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

* would never work.
* Note: Memory doesn’t contain rows and columns. In memory whether it is a one-dimensional or a two-dimensional array the array elements are stored in one continuous chain.

2.Inputting values

* The second way to read the values from the keyboard or a data file.
* This method of inputting values & done using nested loops.

e.g., The Following is way to input values in two-dimensional array

for (i=0; i<m; i++)

{

for (j = 0; j < n; j++)

{

scanf ("%d", &a[i][j]);

}

}

3.Assigning Values

* The third way is to assign the values to the individual elements using the assignment operator.

e.g., a [1][2] = 40;

(or)

Using loop

for (i=0; i<m; i++)

{

for (j=0; j < n; j++)

{

b[i][j] = a[i][j];

}

* Note: To initialize the whole aways to zeros we need to only specify the first value

e.g., a [2][3] = {0};

Outputting Values.

* To output values of elements of two-dimensional array.
* We use nested loops as follows:

for (i=0; i<m; i++) // controls the row.

{

for (j=0; j<n; j++) // controls the columns.

{

printf ("%d", a[i][j]);

}

printf("\n");

}

*Strings*

* *The way a group of integers can be stored in an integer array, similarly a group of characters can be stored in a character array. Character arrays are many a time also called strings.*
* *Character arrays or strings are used by programming languages to manipulate text such as words and sentences.*
* *A string constant is a one-dimensional array of characters terminated by a null (‘\0’).*

*For example, char name [] = {‘H', 'E', 'L', 'L','\0'};*

* *A string is a collection of individual characters stored at contiguous memory location, in which each character is stored using 1byte in the memory and the last character is always ‘\0’.‘\0’ is called null character*
* *The string is a sequence of characters enclosed in double quotes (" ")*

*For e.g., "Hello!", "You are Welcome";*

* *Note that if single (‘ ‘) or double (" ") quotes, are part of the String, they must be used with the escape sequence.*
* *Note: "A" is a string while ’A’ is a character*

*Need of String Delimiter*

* *A string is a variable length array of characters that is terminated by delimiter or null character.*
* *Since, strings are of varying length, there is a need to identity the end of data.*
* *If the data is of fixed length, then we don't need string rather it can easily be stored in an array.*
* *But, if the data is not fixed i.e., varying length, then we need some way to determine the end of the data. So null character is used to mark the end of the string data.*
* *Therefore, the terminating null (‘\0’) is important, as it is the only way the functions that work with a string can know where the string ends. In fact, a string not terminated by a ‘\0’ is not really a string, but merely a collection of characters.*

*Declaring String Variable*

* *The following is the syntax to declare a string variable*

*Syntax: char string\_name [dimension];*

*For e.g., char str [20];*

* *This statement declares string variable str of length 20 but maximum characters that can be stored are 19 as last character is always the null character*
* *Note: The maximum characters that can be stored in string (n-1) because the last character is always the null character.*

*Initializing string Variable*

* *String variable can also be initialized during declaration*
* *There are too ways in which we can initialize a string*
* *Approach one*

*For e.g., char str [] = {'y', `o', `u', ` `, `a', 'r', `e`, ’ ’, 'W', `e', 'l', 'c', 'o', 'm', 'e', '\0'};*

*In this initialization the size of array is not specified. Here even size is not specified the C compiler consider size of string as number of character constants specified, including the null character.*

* *Approach two*

*C provides a short and efficient approach for initializing strings*

*For e.g., char str [] ="you are welcome";*

*Note:*  *in this declaration ‘\0’ is not necessary. C inserts the null character automatically.*

*Input/Output of Strings.*

* *The C language provides library functions for input and output of string data.*

*Input of a string*

* *Single word string can be entered using the following statement*

*scanf ("%s", str);*

* *Multiple word strings can be entered using the following statement*

*scanf ("%[^\n] s", str);*

*Here we use the expression to take the complete line including spaces as a string input in C.*

*Note: While entering the string using scanf () we must be cautious about the length of the string should not exceed the dimension of the character array. This is because the C compiler doesn’t perform bounds checking on character arrays. Hence, if you carelessly exceed the bounds there is always a danger of overwriting something important.*

* *The other way is, for string input the function gets () is used.*

*gets () = This function accepts a string from the standard input device. The length of the string is limited by the declaration of the string variable.*

* *The input string may consist of multiple words. Typing enter key completes the input.*

*e.g., gets (str);*

*Output of a string*

* + *We can output string using following statement*

*printf ("%s", str);*

* + *The other way is, for string output the function puts () is used*

*puts (): This function outputs a string constant or a string variable to the standard output device.*

*Note: puts( ) can display only one string at a time*

*String Manipulation Functions*

* *The C compiler provides a large number of library functions for manipulating strings.*

|  |  |
| --- | --- |
| *String function* | *Operation performed* |
| *Strlen* | *Finds length of a string* |
| *Strcat* | *Appends the characters of second argument at the end of the first-string variable.* |
| *Strcpy* | *Copies the character of Source string variable to destination string variable.* |
| *Strcmp* | *Compares two strings (Comparison is case sensitive).* |
| *Strlwr* | *Converts a string to lower case* |
| *Strrev* | *Reverses the string. (Only supported by turbo c)* |
| *Strupr* | *Converts a string to uppercase.* |

1. *The strlen () function*
   * *This function takes one argument that can be a string constant or a variable.*
   * *This function counts the number of character present in the string.*
   * *Note that null character “\0” is not a part of the character.*

*e.g., len = strlen (str);*

1. *The strcat () function*

* *This function takes two arguments, of which first is the string variable and second can be a string variable.*
* *It appends the characters of the second string at the end of the first argument.*
* *Note there must be sufficient space available to store too strings*

*e.g., strcat (str1, str2);*

1. *The strcpy () function*

* *This function copies the contents of one string into another.*
* *This function takes two arguments; of which first is a string variable and second can be string variable where the copy of first string to be stored.*

*e.g., strcpy (str1, str2);*

* *On supplying the base addresses, strcpy () goes on copying the characters in source string into the target string till it doesn't encounter the end of source string (‘\0’).*
* *It is our responsibility to see to it that the target string’s dimension is big enough to hold the string being copied into it.*
* *Thus, a string gets copied into another, piece-meal, character by character.*

1. *The strcmp () function*

* *This function takes two arguments of which both can be string variables.*
* *This function compares two strings to find out whether they are same or different. The two strings are compared character by character until there is a mismatch or end of one of the strings is reached, whichever occurs first.*
* *If the two strings are identical, strcmp () returns a value zero.*
* *If they’re not, it returns the numeric difference between the ASCII values of the first nonmatching pairs of characters.*

|  |  |
| --- | --- |
| *Value return* | *Meaning* |
| *Less than zero* | *str1 is less than str2*  *i.e., str1 comes first str2 in dictionary order.* |
| *Zero* | *str1 is equal to str2* |
| *Greater than zero* | *str1 is greater than str2*  *i.e., str1 comes after str2 in dictionary order.* |

*2-D Array of Strings*

* *We can use two-dimensional array of character if we want to handle list of strings. Because string variable can store only one string value at a time.*
* *In two dimensional arrays, the first-row stores string, the second row stores the second string and so on.*
* *The order of the subscripts in the array declaration is important.*
* *The first subscript gives the number of names in the array, while the second subscript gives the length of each item in the array.*
* *for e.g.:*

*char str [4] [31] = {" Ram Parkash",*

*"Mohan Singh",*

*” Amanpreet”,*

*"Rajan"};*

* *To input the value, we can even use loop with library function*
* *For e.g.:*

*for (i=0; i<n; i++)*

*{*

*printf ("Enter string: ",i+1);*

*gets (str [i]);*

*}*

* *But this scheme of representing list of strings can cause memory wastage. If we could not use the array of specified size completely.*

*Structures*

* *We deal with entities that are collections of things, each thing having its own attributes, just as the entity we call a ‘book’ is a collection of things such as title, author, call number, publisher, number of pages, date of publication, etc.*
* *For example, suppose you want to store data about a book. You might want to store its name (a string), its price (a float) and number of pages in it (an int). If data about say 3 such books is to be stored, then we can follow two approaches:*
  1. *Construct individual arrays, one for storing names, another for storing prices and still another for storing number of pages.*
  2. *Use a structure variable.*
* *C provides a data type called ‘structure’. A structure gathers together, different atoms of information that comprise a given entity.*
* *A structure contains a number of data types grouped together. These data types may or may not be of the same type.*
* *The structure is a user defined data type, which a user creates as per their requirements.*
* *A structure is a collection of related data items that belong to different datatypes.*
* *These data items occupy contiguous memory locations as same elements of an array are stored in contiguous memory locations.*

*Defining a Structure*

* *There are two ways to define a structure*
  1. *Tagged structure*
  2. *Type -defined structure*

1. *Tagged structure Definition*
   * *The Syntax for declaring a tagged structure is*

*struct Structure name*

*{*

*structure element 1;*

*structure element 2;*

*structure element 3;*

*------*

*};*

* + *For e.g.,*

*Struct STUDENT*

*{*

*int roll No;*

*char name [20];*

*char fname [20];*

*char grade;*

*};*

* + *The tagged structure starts with the keyword "Struct"*
  + *The next element in the definition is tag [structure name]*
  + *The tag is used to declare variable, arguments of the function and return types of the functions.*

*Type - defined Structure*

*Definition*

* *The Syntax for declaring a type-defined structure is*

*typedef struct*

*{*

*structure element 1;*

*structure element 2;*

*structure element 3;*

*-----*

*} structure name;*

* *For e.g.,*

*typedef struct*

*{*

*int roll no;*

*char name [20];*

*char grade;*

*} STUDENT;*

* *The typedef is used before the struct keyword and the structure name is written after the closing brace and before semicolon.*

*Declaring Structure Variables*

* *Once a structure is defined, we can declare variables using these declarations.*
* *Note: Usually the definition of structure is placed in global area in the program, so that it is visible to all the functions in the program.*

*Tagged structure declaration*

*Syntax: Struct Structure\_name Structure-variable;*

*For e.g., struct STUDENT astudent;*

*Type-defined structure declaration*

*Syntax: Structure name structure\_variable;*

*For e.g.: STUDENT astudent;*

* *The type defined structure is easier because you don't need to use the struct keyword for every declaration of variable.*
* Note the following points while declaring a structure type:

1. The closing brace in the structure type declaration must be followed by a semicolon.
2. It is important to understand that a structure type declaration does not tell the compiler to reserve any space in memory. All a structure declaration does is, it defines the ‘form’ of the structure.
3. Usually, structure type declaration appears at the top of the source code file, before any variables or functions are defined. In very large programs they are usually put in a separate header file, and the file is included (using the preprocessor directive #include) in whichever program we want to use this structure type.

*Initializing Structures*

* *like arrays and other variables, structure variables can also be initialized where they are declared.*
* *The format used us quite similar to that used to initialize arrays.*

*For e.g.: STUDENT astudent = {1000, "Surbhi",5.6};*

* *Note: Note that if a structure variable is partially initialized. The remaining numeric fields will be initialized to zero and the string fields will be initialized to null.*

*Accessing structure Elements.*

* *In arrays we can access individual elements of an array using a subscript.*
* *For structures individual structure elements of a structure variable can be accessed using the dot operator (`.')*
* *This dot operator is also referred as membership operator.*

*Syntax for accessing a structure element*

*Structure variable. Structure element*

*For e.g.: astudent. rollNo*

*Assigning Of Structures*

* *The structure can be assigned / copied into another structure of the same type using the assignment operator.*
* *All the values in the structure are assigned to corresponding structure element.*

*For e.g., astudent = bstudent;*

* *Note: It is unique feature of structures as compared to arrays. Simple assignment statement cannot be used for arrays.*
* *To assign one array to another, we have to use a loop to assign [copy] element-by element.*

*Reading / Writing Structures*

* *We can read into and write data from a structure element in same manner as we do with simple variables.*

*For e.g., To read input and print output*

*printf ("\n Enter roll number of student: ");*

*scanf ("%d", &astudent. rollNo);*

*(or)*

*printf (" \n Employee's Code: %d", aemployee. code);*

*Array of Structures*

* *We can declare array of structure variable*
* *In an array of structures all elements of array are stored in adjacent memory locations.*
* *For example, an array of structures to store the data of student will be declared as*
* *For e.g., STUDENT Student [50];*

*Accessing Elements of Array of structures*

* *Individual elements of a structure in an array of structures are accessed by referring to structure variable name, followed by index, followed by direct selection operator, and ending with the structure element desired.*

*For e.g., student [i].name;*

*Initializing Array of structures*

* *An array of structures, can also be initialized. Since each element of array structures is a different structure.. We need to include elements of each structure in a separate set of braces as shown*

*e.g., STUDENT Students [50] = {{1000, "Monika", {56,76,85,69}, "A"},*

*{1001, "Ram", {50,66,70,60}, `B'},*

*-*

*-*

*{1049, "Raju", {65,62,68,69}, `B'}*

*};*

*Nested Structures*

* *A nested structure is a structure within structure. One structure can be declared inside another structure in the same way structure members are declared inside a structure.*

*Accessing Elements of Nested structures*

* *Individual elements of a nested structure are accessed by referring to outer structure variable name, followed by direct selection operator (.) Followed by inner structure variable name, followed by direct selection operator, and ending with the desired member of the structure.*
* *E.g., aEmployee, date of Joining.Year;*
* *Syntax: Outer structure variable name. Inner structure variable name. member of the structure;*

*Initialization of Nested structures.*

* *Initialization of nested structures follows the roles of simple structures.*
* *Each structure must be initialized individual separately before preceding to the next element.*
* *Each structure is enclosed in a set of braces.*

*e.g., Struct EMPLOYEE aemployee = {1256,” Rakesh Bahl”, {12,06, 1962},"Finance", 40000}};*

*Structure and functions*

* *Structure can be passed by value as well as by reference*
* *passing structure by reference*

*#include <stdio.h>*

*typedef Struct {*

*int day;*

*int month;*

*int years;*

*} DATE;*

*/\* function declaration \*/*

*void printdate (DATE \*aDate);*

*int main ()*

*{*

*DATE aDate = {10, 6, 2008};*

*/\* function call/*

*printdate (&adate);*

*/\* statements \*/*

*}*

*/\* function definition\* /*

*void printdate (DATE \*aDate)*

*{*

*/\* statements \*/*

*}*

* *Passing structure by value*

*#include <stdio.h>*

*/\* Structure definition \*/*

*typedef struct*

*{*

*int day;*

*int month;*

*int years*

*} DATE;*

*/\*function declaration \*/*

*void printdate (DATE adate);*

*void main ()*

*{*

*DATE adate = {10, 6, 2008};*

*int x;*

*/\*function call\*/*

*printdate (adate);*

*/\* function definition \*/*

*void printdate (DATE adate)*

*{*

*/\*local declarations\*/*

*/\*other statements \*/*

*}*

*Memory Allocation*

* *For every program to be executed or data to be processed, some memory must be allocated for them.*
* *The memory allocation can be done in two ways.*
  1. *Statically*
  2. *Dynamically.*

1. *Static Memory Allocation*

* *In this memory is allocated during compilation of program (during compile time).*
* *The memory requirements are determined prior to execution of program and the compiler then can determine the memory requirement for data as well as the instruction and there after your program neither can acquire more memory nor can release the free the memory, if not required.*
* *In such a situation, memory allocation is known as static memory allocation.*

1. *Dynamic Memory Allocation.*

* *In many situations when the exact memory requirements for the data may not be known in advance or the data requirements may vary from on program execution to another program execution.*
* *Therefore, when the amount of memory is not known before hand for a particular data item, them it is allocated at the execution time.*
* *In such a situation memory allocation is known as dynamic memory allocation.*
* *The process of allocating memory a runtime is known as dynamic memory allocation.*

*Heap / Free Store*

* *The unallocated memory is known as heap or free store.*
* *Therefore, whenever the memory of required amount by the program it is taken from the free store and when the previously allocated memory is not required further, it is returned back to the free store.*

*Dynamic Memory Management*

* *C language provides library function called dynamic memory management functions, to allocate and deallocated memory at execution time i.e., dynamically*
* *By allocation, we mean that your program can obtain as much memory as required by your program even during execution of your program.*
* *By deallocation, we mean that the memory acquired dynamically can be released at any time during your program execution.*
* *Note that, the memory allocated dynamically, must be de-allocated" Before your program finishes its execution. Otherwise, even if your program terminates memory allocated dynamically is never released. Automatically, and from operating system may run out of memory.*
* *The following are the dynamic memory Management functions and are defined in stdlib.h.*

|  |  |
| --- | --- |
| *Function name* | *Description* |
| *malloc* | *Allocates memory from heap* |
| *calloc* | *Allocates memory from heap and initializes the allocated memory to zeroes* |
| *realloc* | *Readjusts the existing block and copies the contents to new location.* |
| *free* | *Deallocate a block allocated by malloc, calloc and realloc functions"* |
| *coreleft* | *Returns a measure of unused memory (Supported by Turbo c only)* |

1. *malloc () function*

* *malloc () function takes one argument that specifies the size of the block in bytes.*
* *The function allocates the memory of required size from heap and returns a pointer to the allocated memory on success*
* *A NULL pointer (o) in case of failure.*
* *The pointer returned is of type void which need to be casted into a pointer of desired type.*
* *The memory allocated is left uninitialized i.e., all locations get the garbage value.*
* *Syntax for use of malloc () function*

*datatype \*ptr*

*ptr = (datatype \*) malloc (n\* size\_of\_data\_bytes);*

* *e.g., int \*ptr;*
* *ptr = (int \*) malloc (n\* sizeof (int));*

1. *calloc () function*

* *calloc () function takes too arguments that specify the no. Of items for which memory is to be allocated and the data type of the item.*
* *The function allocates the memory of required number of elements from heap and returns a pointer to the allocated memory on success*
* *A NULL pointer (o) in case of failure*
* *The pointer returned is also of type void which need to be Casted into a pointer of desired type*
* *The memory allocated is initialized with 0. i.e., all locations get the 0 value.*
* *The Syntax for use of callac () function*
* *datatype \*ptr*
* *Ptr = (datatype \*) calloc (n, datatype);*
* *For e.g., int \*ptr*
* *ptr = (int \*) calloc (10, size of (int));*

1. *realloc () function*

* *The realloc () function allocates a block of memory (which can be of larger or smaller in size than the original and copies the contents of the old block to the new block of memory.*
* *realloc () function first allocates the new block of required memory, Copies the contents of the existing block, reallocates the exiting block and at the end returns the address of the new block.*
* *The realloc () function takes two arguments - first argument is the pointer to the original block, and the second argument is the size of the new block.*
* *It returns NULL pointer (o) in case of failure.*
* *The syntax for use of realloc () function*

*datatype \*ptr1 \*ptr2*

*ptr2 = (datatype \*) realloc (ptr1, size\_jn\_bytes);*

*For e.g., int \*ptr1\_\*ptr2;*

*ptr2 = (int \*) realloc (ptr1, 20\* sizeof(int));*

1. *free () function*

* *The free () function releases/deallocates a dynamically allocated block of memory.*
* *The syntax for use of free () function*

*free (ptr);*

* *It takes one argument that specifies the pointer to the allocated block.*
* *Note that only the allocated block is deallocated, the pointer variable is not deleted.*

*UNIT-V*

*File Handling, Pre-Processor directives and Basic Algorithms*

*File Handing*

*Introduction*

* *When the volume of data is very large it is very difficult to accept the input data from the keyboard at the time of execution.*
* *If the data is entered in computer's memory and stored in a disk file, a program can read it directly from the disk and that too at very high speed.*
* *It is more efficient to store the output of a program in a disk file, which can subsequently be read by another program.*
* *The C program can read and write files in a variety of ways.*
* *Depending on the way a file is opened for processing, a file can be classified as a text file or a binary file.*

*Declaring the file pointer*

* *Syntax:*

*FILE \*filepointer;*

*Difference between Text file and Binary file*

|  |  |
| --- | --- |
| *Text file* | *Binary file* |
| *Data is stored using human readable characters.* | *Data is stored in the same format as it is stored in memory.* |
| *Each line of data ends with a newline character.* | *There are no lines or newline character* |
| *There is a special character called End of file (EOF) marker at the end of the file.* | *There is an end of tile marker at the end of the file, which is usually Implementation dependent* |

*STEPS IN PROCESSING A FILE*

* *The processing of file involves following steps* 
  + 1. *opening a file - using fopen ()*
    2. *Reading from or writing on to a file*
    3. *closing the file- using fclose ()*

1. *Opening a File*
   * *First step in file processing is to Open the file in appropriate mode, using fopen () function.*
   * *If you want to read from a file in your program, the file must be opened in read/input mode.*
   * *If you want to write the output of your program to a file, the file must be opened in the write/output mode.*
   * *There are many such modes in file processing some of the file opening modes are as follows.*

|  |  |
| --- | --- |
| *Mode* | *Purpose* |
| *r* | *Open a file for reading, The file must already exist.* |
| *w* | *Open a file for writing. If the file already exists, its contents will be discarded. If it does not exist, it will be created.* |
| *a* | *Open a file for extending. Data will be added to the end of the existing file, if file already exists. If it does not exist it will be created.* |
| *r+* | *Open a file for both reading and writing. The file must already exist.* |
| *w+* | *Open a file for both reading and writing. If it does not exist, it will be created.* |
| *a+* | *Open a file for both reading and appending. If the file does not exist, it will be created* |

*Declaration to open a file*

*Syntax:*

*fptr = fopen (" file name ", " opening modes");*

*For e.g., fptr = fopen ("test.date", "w");*

* *This declaration request the OS to open a file named test.date in textmode and prepare it for writing*

*Trouble with opening a file*

* *The request for opening a file may be granted or rejected.*
* *In case of rejection, the program will not run and will terminate abnormally.*
* *There are many reasons for the rejection of program request some of them are as follows:*
  + 1. *Trying to open a file for reading that does not exist.*
    2. *Trying to open a file for writing, but sufficient space is not available on the disk.*
* *Note: It is important for the program to check whether a file has been opened successfully before trying to read from or write onto the file.*
* *If the file cannot be opened fopen () function return a NULL value.*
* *Example code to check whether file opened successfully or not*

*fptr = fopen ("test.data", "w");*

*if (fptr == NULL)*

*{*

*printf ("\n Can't open file \n");*

*return 1;*

*}*

1. *Reading from or writing onto a file*

*Once a file is opened successfully, data can be read from a file or written to a file in variety of ways.*

*The following are ways of reading data from a file and writing data to a file*

*Character I/O functions – fgetc (), fputc ()*

*String I/O functions – fgets (), fputs ()*

*File I/O function*

*formatted I/O functions – fscanf (), fprintf ()*

*Record I/O functions – fread (), fwrite ()*

1. *Character I/O functions*

* *Once the program is opened, then data can be read from a file or written to a file.*
* *Using character, I/O functions, data can be read or written one character at a time.*
* *For reading from a file or for character input, the function is fgetc ().*
* *The Syntax of the function that reads and returns one character at a time is*

*character variable = fgetc (file pointer);*

* *e.g., ch= fgetc (ptr);*
* *for writing to a file or for character output, the function is fputc ()*
* *The syntax of function that writes one character at a time is*

*fputc (character variable, file pointer);*

* *for e.g., fputc (ch, fptr);*
* *fgetc () function returns the character and read from the file or end of file (EOF) if it has reached the end of the file.*

1. *String I/O functions*

* *Using String I/O functions, data can be read or written in the form of string of characters.*
* *For reading from a file or for string input, the function is fgets ().*
* *The syntax of the function that reads strings from a file is*

*Fgets (String [], max length of input, file pointer);*

*For e.g., fgets (str, n, fptr);*

* *The fgets () function returns a NULL value when it reads End of file (EOF).*
* *For writing to a file or for string output, the function is fputs ().*
* *The Syntax of the function that writes a string of characters at a time is*

*fputs (string [], file pointer);*

*For e.g., fputs. (str. fptr);*

1. *Formatted I/O functions*

* *Formatted I/O functions are used to take various inputs from the file and display multiple output to the file.*
* *This type of functions can help to display the outputs in different formats using format Specifiers.*
* *We can format the function according to our needs.*
* *For Reading from a file at the function is fscanf ().*
* *The Syntax of the function that reads formatted data to a file is*

*fscanf (file pointer, "format-string", list of address);*

*For e.g., fscanf (fptr, “%s, %d, %f", name, & code, & height);*

* *For writing to a file, the function is fprintf ()*
* *The Syntax of the function that writes formatted data to a file is*

*fprintf (filepointer, "format-string", list of variables);*

*For e.g., fprintf (fptr, "%s, %d, %f", name, code, height);*

1. *Closing a file*

* *This is the final step in file processing. Once a program has finished with reading and writing of the files, each file must be closed.*
* *This task is carried out by using a library function fclose ().*
* *The Syntax of fclose function is as follows.*

*fclose (file pointer);*

*e.g., fclose (fptr);*

* *Closing a file explicitly has two effects. These are*
* *Any data remaining in the (storage area) butter is written to the file.*
* *Frees the communication area used by the particular file. So that they are available for other files.*

*These areas include the file structure and the buffer itself.*

*File status functions*

* *The C language provides following three functions to handle file stater queries:*

1. *eof ()- to test end of file*
2. *ferror () - to test error.*
3. *clearerr () - to clear error.*
4. *Test End of File: feof () function*
   * *The feof () function is used to check if the end of the file has been reached*
   * *If the file pointer is at the end i.e., all data have been read, the function returns 1.*
   * *If the end of the file is not reached. It returns 0.*
   * *Syntax:*

*int feof (FILE \*Stream);*

* + *Stream means file pointer for currently opened file*

1. *Test Error: ferror () function*
   * *The ferror () function is used to test the error status of file.*
   * *The ferror () function returns 1 if an error has occurred after a file operation.*
   * *If no error has occurred the file ferror () returns value 0.*
   * *Syntax:*

*int ferror (FILE \*stream);*

1. *Clear Error: Clearerr () function*

* *When an error occurs, the subsequent calls to the ferror function return 1, until the error status of the file is reset.*
* *Syntax:*

*void clearerr (FILE \*stream);*

*File Positioning functions.*

*The C language provides following three functions to handle file positioning operations:*

1. *rewind (): - To set the file pointer to the beginning the file.*
2. *ftell (): - to know the current position of the file pointer in the file.*
3. *fseek (): - To change the position of the file pointer in the file.*
4. *Rewind file: rewind () function.*

* *One way of positioning the file pointer to the beginning of the file is to close the file and then re-open it again.*
* *We can accomplish same task using rewind is function. This function position, the file pointer in the beginning of the file.*
* *Syntax:*

*void rewind (FILE \* Stream);*

1. *Current Location: ftell () function*

* *To find the current location of the file pointer within the file we use ftell () function.*
* *The ftell () function returns a long integer.*
* *When file pointer is at the beginning of the file, the ftell () function returns 0.*
* *If the file pointer is at the second byte, the function returns value 1.*
* *Syntax:*

*long int ftell (FILE \*stream);*

1. *Set Position: fseek () function.*

* *In order to read a data item from anywhere in a file, we have to move the file pointer to the beginning of that data item.*
* *To accomplish this task, we use fseek () function.*
* *Syntax:*

*int fseek (FILE \*stream, long offset, int wherefrom);*

* *fseek () function takes three arguments,*
* *first argument-- is a file pointer*
* *Second argument-- is a variable of type long integer that specifies the number of bytes by which file pointer is to moved*
* *Third argument – specifies from which position offset is measured.*
* *Values of where from are*

*0 - Beginning of the file*

*1-Current position of the file*

*2-End of the file.*

*Preprocessor Directives*

* *The C Source code may include instructions for the compiler.*
* *These instructions are known as preprocessor commands or directives.*
* *The preprocessor will process directives that are inserted into the C source code.*
* *These directives allow additional actions to be taken on the C Source code before it is compiled into object code.*
* *Each directive begins with character "#"*
* *The C compiler is made up of two functional parts*

1. *a preprocessor*
2. *a translator.*

* *The preprocessor processes all the directives and as a result produces an intermediate code known as translation unit, which is then translated by the translator to machine code and place it in object file.*
* *The preprocessor offers several features called preprocessor directives. Each of these preprocessor directives begin with a #symbol.*
* *The directives can be placed anywhere in a program but are most often placed at the beginning of a program, before the first function definition. We would learn the following preprocessor directives here:*

1. *Macro expansion*
2. *File inclusion*
3. *Conditional Compilation*
4. *Miscellaneous directives*

* *The following are some of the preprocessor directives that can be used in source code.*
  1. *#include*
  2. *#define*
  3. *#undef*
  4. *#if*
  5. *#ifndef*
  6. *#elif*
  7. *#else*
  8. *#error.*

1. *#include directive*

* *The #include directive, also known as file inclusion.*
* *This instructs the preprocessor to copy the contents of specified file into the program.*
* *The files to be included are generally header files.*
* *Syntax*

*#include <filename.h> /\* for standard directory\*/*

*#include "filename.h" /\* for user directory\*/*

* *The first directs the preprocessor to include the header file from the standard directory.*
* *The second directs the preprocessor to include the specified header file from user directory.*

1. *#define directive*

* *The #define directive, is used to define an identifier and a character string that is substituted for the identifier each time it is encountered in the source file.*
* *The identifier is called a macro name. The replacement process is called macro-substitution.*
* *Syntax*

*#define identifier string*

*For e.g., #define MAX 100*

* *In other words, a piece of code in a program that is given some name is macros. Whenever this name is encountered by the compiler, the compiler replaces the name with the actual piece of code.*

1. *#undef directive*

* *The #undef directive is used to remove a previous definition of the macro-name that follows it.*
* *Syntax*

*#undef macro-name*

1. *#If directive*

* *If the constant expression following #if is true, the code between it is compiled; otherwise, the code is ignored by the compiler*
* *The #endif is used to mark the end of a #if block*
* *Syntax:*

*#if constant \_ expression*

*statement(s)*

*#endif*

* *For e.g.,*

*#if Max >99*

*printf (" Compiled");*

*#endif*

1. *#else directive*

* *The #else directive works the same way as the else statement.*
* *It provides an alternative to the #if directive*
* *Syntax:*

*#if constant \_expression*

*statements(s)*

*#else*

*statements(s)*

*#endif*

1. *#elif directive*

* *The #elif directive means else if and is used to establish an if-else-if ladder for compilation options*
* *The #elif is followed by a constant expression.*
* *If the expression is true, that block of code is compiled and no other #elif expressions are tested, otherwise, the next in the series is checked.*
* *Syntax:*

*#if expression-1*

*statement*

*#elif expression-2*

*statement*

*#elif expression-3*

*statement*

*#end if*

1. *#ifndef directive*

* *The directive #ifndef means if defined.*
* *If the macro- name has been previously defined in a #define statement, the statement sequence between the #ifndef and #endif is compiled*
* *Syntax*

*#ifndef macro\_name*

*statement*

*#endif*

1. *#ifdef directive*

* *The directive #ifdef means if defined.*
* *If the macro\_name has been previously defined in a #define statement, the statement sequence between the #ifdef and #endif is compiled*
* *Syntax*

*#ifdef macro\_name*

*statement*

*#endif*

1. *#error directive*

* *The #error directive forces the C compiler to stop compiling when it is encountered. It is primary used for debugging*
* *Syntax*

*#error error\_message*

*Searching and Sorting Algorithms*

*Searching Algorithms*

* *Searching is the process of finding the location of given element in an array.*
* *The search is said to be successful if the given element is found*
* *i.e., the element does exist in the array; otherwise, unsuccessful.*
* *There are two approaches to search operation.*

1. *Linear search*
2. *Binary Search*

* *Note:*

1. *If the elements are in random order, then one have to use linear search technique.*
2. *If the array elements are sorted, then it is preferable to use binary search technique.*
3. *Linear Search*

* *When the array elements are in random order, the only way to search for given element item is to compare item with each element of an array one by one.*
* *This method, which traverses array sequentially to locate items called linear search or sequential search.*
* */\*Example code to demonstrate linear search\*/*

*for (i=0; i<n; i++)*

*{*

*if (item==a[i])*

*{*

*flag=1;*

*break;*

*}*

*}*

*if (flag == 1)*

*printf ("\n Element Found at index: %d ", i);*

*else*

*printf ("\n Element not found ...");*

1. *Binary Search*

* *Suppose the elements of the array are sorted in ascending/descending order (number) or dictionary order (if strings). Then the best searching of algorithm, is binary search.*
* *In this algorithm.*
  + *we first divide the array into two halves and find the middle index*
  + *Then compare the middle element of the array with the item.*

1. *If the item is found at middle element, the process is terminated.*
2. *If the item is not found at middle element, then we choose which half will be used as the next search.*

* *If the item is smaller than the middle element, then left side is used for next search*
* *If the item is larger than the middle element, then right side is used for next search.*
* *This process is continued until the item is found or the total array element are searched.*

*/\* Example code to search demonstrate binary search \*/*

*flag = 0;*

*beg = 0;*

*end=n-1;*

*while (beg<end)*

*{*

*mid = (beg+ end)/2;*

*if (item == a [mid])*

*{*

*flag = 1;*

*break;*

*}*

*if (item < a[mid])*

*{*

*end = mid-1;*

*}*

*else*

*{*

*beg =mid+1;*

*}*

*}*

*if (flag==1)*

*printf ("\n Element found at index: %d", mid);*

*else*

*printf (" Element not found “);*

*Sorting Algorithms*

* *Sorting is the process of arranging the element in some logical order.*
* *This logical order may be ascending or descending (in case of numeric values) or dictionary order (in case of string values).*
* *There are three approaches to sorting operation.*
  1. *Bubble Sort*
  2. *Insertion sort*
  3. *Selection Sort*

1. *Bubble Sort*

* *Bubble sort is a Sorting algorithm that compares two adjacent elements and swaps them until they are in the intended order.*
* *This algorithm repeatedly compares the current element with the one after it, and swap their values if needed.*

*Working of bubble sort*

* *Suppose we are trying to sort the element in ascending order*
* *Starting for the first index, compare the fast and the second elements.*
* *If the first element is greater than the second element, they are swapped*
* *Now compare the second and third elements. Then swap them if they are not in order.*
* *The above process goes on until the last element.*
* *After each iteration. The largest element among the unsorted elements is placed at the end.*
* *In each iteration, the comparison takes place up to the last unsorted element*
* *The array is sorted when all the unsorted elements are placed at their correct positions.*

*/\*Example program to demonstrate the use of Bubble Sort\*/*

*for (i=0; i<n-1; i++)*

*{*

*for (j=0; j<n-i-1; j++)*

*{*

*if (a [j] > a [j +1])*

*{*

*temp = a[j];*

*a [j] = a[j+1];*

*a[j+1] = temp;*

*}*

*}*

*}*

1. *Insertion Sort*

*Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration.*

*This algorithm builds the final sorted array by comparison of one item at a time.*

*Working of Insertion Sort*

* *Suppose we need to sort the array in ascending under.*
* *The first element in the array is assumed to be sorted.*
* *Take the second element and store it separately in temporary variable*
* *Then Compare temp variable with the first element.*

*If the first element is greater than temp, then first element is placed in place of second element and temp variable is placed in place of first element.*

*Now, the first two elements are sorted.*

* *Take the third element and store it in temp variable*
* *Then compare it with the 2nd element*

*if temp is smaller, then place 2nd element in 3rd place and then check if temp variable is smaller than 1st element if yes place 1st element in 2nd place and place and place temp in 1st place.*

* *Similarly, place every unsorted element at its correct position*

*/\*example code to demonstrate the use of insertion sort \*/*

*for (k=1; k<n; k++)*

*{*

*temp = a[k];*

*j=k-1;*

*while ((temp < a[j]) && (j >=0))*

*{*

*a[[j+1] = a [ j];*

*j--;*

*}*

*a[j+1] = temp;*

*}*

1. *Selection Sort*

* *Selection sort is a sorting algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list.*

*Working of selection sort*

* *Suppose we need to sort array in ascending order.*
* *Set the first element as minimum*
* *Compare minimum with the second element.*

*If the 2nd element is smaller than minimum, assign the second element as minimum.*

* *Compare minimum with the third element*

*Again, if the third element is smaller, then assign minimum to the third element*

* *The process goes on until last element.*
* *after each iteration minimum is placed in front of unsorted list.*
* *for each iteration, indexing starts from the first unsorted element.*

*/\*example code to demonstrate use of selection sort \*/*

*for (i=0; i<n-1; i++)*

*{*

*min=a[i];*

*loc = I;*

*for (j = i + 1; j<n; j++)*

*{*

*if (a(j) <min)*

*{*

*min=a [j];*

*loc = j;*

*}*

*if (loc! = i)*

*{*

*temp =a[i];*

*a[i]= a [loc];*

*}*

*}*

-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Command line arguments

Introduction

Till now no arguments were passed in the main function. But the c programming language provide provision to add parameters or argument inside the main function to reduce the length of the code.

These arguments are called command line arguments in c.

Command line arguments are the arguments which the user gives from the operating system's command line.

Command line arguments helps us to control the program us from the outside/console

Definition

The Command line arguments are arguments that we pass on to the main () function at the Command prompt.

In other words, Command line arguments are arguments that are Specified after the name Of the Program in the system's command line (or prompt), and these argument values are passed on to program during execution.

Arguments

In order to implement Command line arguments, generally 2 parameters are passed into the main function:

The first argument is argc i.e., the number of command line arguments

This argument is of type integer

The value of this argument is equal to the number of strings to which argv points.

This argument argc is set number of strings given on the command line, including the programme name.

The second argument is \* argv [] i.e., an array of pointers to strings. Which holds individual arguments.

Argument argv [0] holds the name of the program

Argument argv [1] holds for address of first argument

Argument argv [2] holds address of second argument

And so on.

Note: whenever we pass arguments to main (), we must Check whether the correct number of arguments have been Passed on to main () or not.