**PROGRAMMING FOR PROBLEM SOLVING USING C**

**LECTURE NOTES**

****

**Department of Computer Science & Engineering**

**Government College of Engineering Keonjhar**

**758001**

**UNIT–I:** Introduction to components of a computer system (disks, memory, processor, where a program is stored and executed, operating system, compilers etc.)

Idea of Algorithm: steps to solve logical and numerical problems. Representation of Algorithm: Flowchart/Pseudocode with examples.

From algorithms to programs; source code, variables (with data types) variables and memory locations, Syntax and logical errors in compilation, object and executable code

**Introduction to components of a computer system:**

Being a modern-day kid you must have used, seen, or read about computers. This is because they are an integral part of our everyday existence. Be it school, banks, shops, railway stations, hospital or your own home, computers are present everywhere, making our work easier and faster for us. As they are such integral parts of our lives, we must know what they are and how they function. Let us start with defining the term computer formally.

The literal meaning of computer is a device that can calculate. However, modern computers can do a lot more than calculate. **Computer** is an electronic device that receives input, stores or processes the input as per user instructions and provides output in desired format.

## Input-Process-Output Model:

Computer input is called **data** and the output obtained after processing it, based on user’s instructions is called **information**. Raw facts and figures which can be processed using arithmetic and logical operations to obtain information are called **data**.



**The processes that can be applied to data are of two types** −

* **Arithmetic operations** − Examples include calculations like addition, subtraction, differentials, square root, etc.
* **Logical operations** − Examples include comparison operations like greater than, less than, equal to, opposite, etc.

***The corresponding figure for an actual computer looks something like this*:**



**The basic parts of a computer are as follows**:

* **Input Unit** − Devices like keyboard and mouse that are used to input data and instructions to the computer are called input unit.
* **Output Unit** − Devices like printer and visual display unit that are used to provide information to the user in desired format are called output unit.
* **Control Unit** − As the name suggests, this unit controls all the functions of the computer. All devices or parts of computer interact through the control unit.
* **Arithmetic Logic Unit** − This is the brain of the computer where all arithmetic operations and logical operations take place.
* **Memory** − All input data, instructions and data interim to the processes are stored in the memory. Memory is of two types – **primary memory** and **secondary memory**. Primary memory resides within the CPU whereas secondary memory is external to it.

Control unit, arithmetic logic unit and memory are together called the **central processing unit** or **CPU**. Computer devices like keyboard, mouse, printer, etc. that we can see and touch are the **hardware** components of a computer. The set of instructions or programs that make the computer function using these hardware parts are called **software**. We cannot see or touch software. Both hardware and software are necessary for working of a computer.

## Characteristics of Computer

To understand why computers are such an important part of our lives, let us look at some of its characteristics −

* **Speed** − Typically, a computer can carry out 3-4 million instructions per second.
* **Accuracy** − Computers exhibit a very high degree of accuracy. Errors that may occur are usually due to inaccurate data, wrong instructions or bug in chips – all human errors.
* **Reliability** − Computers can carry out same type of work repeatedly without throwing up errors due to tiredness or boredom, which are very common among humans.
* **Versatility** − Computers can carry out a wide range of work from data entry and ticket booking to complex mathematical calculations and continuous astronomical observations. If you can input the necessary data with correct instructions, computer will do the processing.
* **Storage Capacity** − Computers can store a very large amount of data at a fraction of cost of traditional storage of files. Also, data is safe from normal wear and tear associated with paper.

## Advantages of Using Computer

Now that we know the characteristics of computers, we can see the advantages that computers offer:

* Computers can do the same task repetitively with same accuracy.
* Computers do not get tired or bored.
* Computers can take up routine tasks while releasing human resource for more intelligent functions.

## Disadvantages of Using Computer

Despite so many advantages, computers have some disadvantages of their own:

* Computers have no intelligence; they follow the instructions blindly without considering the outcome.
* Regular electric supply is necessary to make computers work, which could prove difficult everywhere especially in developing nations.

## Booting

Starting a computer or a computer-embedded device is called **booting**. Booting takes place in two steps:

* Switching on power supply
* Loading operating system into computer’s main memory
* Keeping all applications in a state of readiness in case needed by the user

The first program or set of instructions that run when the computer is switched on is called **BIOS** or **Basic Input Output System**. BIOS is a **firmware**, i.e. a piece of software permanently programmed into the hardware.

If a system is already running but needs to be restarted, it is called **rebooting**. Rebooting may be required if a software or hardware has been installed or system is unusually slow.

There are two types of booting:

* **Cold Booting** − When the system is started by switching on the power supply it is called cold booting. The next step in cold booting is loading of BIOS.
* **Warm Booting** − When the system is already running and needs to be restarted or rebooted, it is called warm booting. Warm booting is faster than cold booting because BIOS is not reloaded.

**Basics of Computer - Software Concepts**

As you know, the hardware devices need user instructions to function. A set of instructions that achieve a single outcome are called program or procedure. Many programs functioning together to do a task make a **software**.

For example, a word-processing software enables the user to create, edit and save documents. A web browser enables the user to view and share web pages and multimedia files. There are two categories of software −

* System Software
* Application Software
* Utility Software

Let us discuss them in detail.



**System Software**

Software required to run the hardware parts of the computer and other application software are called **system software**. System software acts as **interface** between hardware and user applications. An interface is needed because hardware devices or machines and humans speak in different languages.

Machines understand only binary language i.e. 0 (absence of electric signal) and 1 (presence of electric signal) while humans speak in English, French, German, Tamil, Hindi and many other languages. English is the pre-dominant language of interacting with computers. Software is required to convert all human instructions into machine understandable instructions. And this is exactly what system software does.

Based on its function, system software is of four types:

* Operating System
* Language Processor
* Device Drivers

**Operating System**

System software that is responsible for functioning of all hardware parts and their interoperability to carry out tasks successfully is called **operating system (OS)**. OS is the first software to be loaded into computer memory when the computer is switched on and this is called **booting**. OS manages a computer’s basic functions like storing data in memory, retrieving files from storage devices, scheduling tasks based on priority, etc.

**Language Processor**

As discussed earlier, an important function of system software is to convert all user instructions into machine understandable language. When we talk of human machine interactions, languages are of three types:

* **Machine-level language**: This language is nothing but a string of 0s and 1s that the machines can understand. It is completely machine dependent.
* **Assembly-level language**: This language introduces a layer of abstraction by defining **mnemonics**. **Mnemonics** are English like words or symbols used to denote a long string of 0s and 1s. For example, the word “READ” can be defined to mean that computer has to retrieve data from the memory. The complete **instruction** will also tell the memory address. Assembly level language is **machine dependent**.
* **High level language**: This language uses English like statements and is completely independent of machines. Programs written using high level languages are easy to create, read and understand.

Program written in high level programming languages like Java, C++, etc. is called **source code**. Set of instructions in machine readable form is called **object code** or **machine code**. **System software** that converts source code to object code is called **language processor**. There are three types of language interpreters:

* **Assembler** − Converts assembly level program into machine level program.
* **Interpreter** − Converts high level programs into machine level program line by line.
* **Compiler** − Converts high level programs into machine level programs at one go rather than line by line.

**Device Drivers**

System software that controls and monitors functioning of a specific device on computer is called **device driver**. Each device like printer, scanner, microphone, speaker, etc. that needs to be attached externally to the system has a specific driver associated with it. When you attach a new device, you need to install its driver so that the OS knows how it needs to be managed.

**Application Software**

A software that performs a single task and nothing else is called **application software**. Application software is very specialized in their function and approach to solving a problem. So spreadsheet software can only do operations with numbers and nothing else. Hospital management software will manage hospital activities and nothing else. Here are some commonly used application software:

* Word processing
* Spreadsheet
* Presentation
* Database management
* Multimedia tools

**Utility Software**

Application software that assists system software in doing their work is called **utility software**. Thus utility software is actually a cross between system software and application software. Examples of utility software include −

* Antivirus software
* Disk management tools
* File management tools
* Compression tools
* Backup tools

**Basics of Computers - System S/W**

As you know, system software acts as an interface for the underlying hardware system. Here we will discuss some important system software in detail.



**Operating System**

**Operating system (OS)** is the lifeline of computer. You connect all the basic devices like CPU, monitor, keyboard and mouse; plug in the power supply and switch it on thinking you have everything in place. But the computer will not start or come to life unless it has an operating system installed in it because OS:

* Keeps all hardware parts in a state of readiness to follow user instructions
* Co-ordinates between different devices
* Schedules multiple tasks as per priority
* Allocates resource to each task
* Enables computer to access network
* Enables users to access and use application software

Besides initial booting, these are some of the functions of an operating system:

* Managing computer resources like hardware, software, shared resources, etc.
* Allocating resources
* Prevent error during software use
* Control improper use of computer

One of the earliest operating systems was **MS-DOS,** developed by Microsoft for IBM PC. It was a **Command Line Interface (CLI)** OS that revolutionized the PC market. DOS was difficult to use because of its interface. The users needed to remember instructions to do their tasks. To make computers more accessible and user-friendly, Microsoft developed **Graphical User Interface (GUI)** based OS called **Windows**, which transformed the way people used computers.

**Assembler**

Assembler is system software that converts assembly level programs to machine level code.



These are the advantages provided by assembly level programming:

* Increases efficiency of the programmer as remembering mnemonics is easier
* Productivity increases as number of errors decreases and hence debugging time
* Programmer has access to hardware resources and hence has flexibility in writing programs customized to the specific compute

**Interpreter**

The major advantage of assembly level language was its ability to optimize memory usage and hardware utilization. However, with technological advancements computers had more memory and better hardware components. So ease of writing programs became more important than optimizing memory and other hardware resources.

In addition, a need was felt to take programming out of a handful of trained scientists and computer programmers, so that computers could be used in more areas. This led to development of high level languages that were easy to understand due to resemblance of commands to English language.

The system software used to translate high level language source code into machine level language object code line by line is called an **interpreter**. An interpreter takes each line of code and converts it into machine code and stores it into the object file.

The **advantage** of using an interpreter is that they are very easy to write and they do not require a large memory space. However, there is a major disadvantage in using interpreters, i.e., interpreted programs take a long time in executing. To overcome this **disadvantage**, especially for large programs, **compilers** were developed.

**Compiler**

System software that store the complete program, scan it, translate the complete program into object code and then creates an executable code is called a compiler. On the face of it compilers compare unfavorably with interpreters because they:

* are more complex than interpreters
* need more memory space
* take more time in compiling source code

However, compiled programs execute very fast on computers. The following image shows the step-by-step process of how a source code is transformed into an executable code:



***These are the steps in compiling source code into executable code:***

* **Pre-processing** − In this stage pre-processor instructions, typically used by languages like C and C++ are interpreted, i.e. converted to assembly level language.
* **Lexical analysis** − Here all instructions are converted to **lexical units** like constants, variables, arithmetic symbols, etc.
* **Parsing** − Here all instructions are checked to see if they conform to **grammar rules** of the language. If there are errors, compiler will ask you to fix them before you can proceed.
* **Compiling** − At this stage the source code is converted into **object code**.
* **Linking** − If there are any links to external files or libraries, addresses of their executable will be added to the program. Also, if the code needs to be rearranged for actual execution, they will be rearranged. The final output is the **executable code** that is ready to be executed.

Difference between Compiler and Interpreter:

|  |  |
| --- | --- |
| **Compiler** | **Interpreter** |
| Scans the entire program before translating it into machine code. | Translates and executes the program line by line. |
| Converts the entire program to machine code and executes program only when all the syntax errors are removed. | The interpreter executes one line at a time, after checking and correcting its syntax errors and then converting it to machine code. |
| Slow in debugging or removal of mistakes from a program. | Good for fast debugging. |
| Program execution time is less. | Program execution time is more. |

## Algorithm:

## An algorithm is procedure consisting of a finite set of unambiguous rules (instructions) which specify a finite sequence of operations that provides the solution to a problem, or to a specific class of problems for any allowable set of input quantities (if there are inputs). In other word, an algorithm is a step-by-step procedure to solve a given problem.

Algorithm has the following characteristics:

* **Input**: An algorithm may or may not require input
* **Output:** Each algorithm is expected to produce at least one result
* **Definiteness**: Each instruction must be clear and unambiguous.
* **Finiteness**: If the instructions of an algorithm are executed, the algorithm should terminate after finite number of steps

The algorithm and flowchart include following three types of control structures.

1. **Sequence**: In the sequence structure, statements are placed one after the other and the execution takes place starting from up to down.
2. **Branching (Selection):** In branch control, there is a condition and according to a condition, a decision of either TRUE or FALSE is achieved. In the case of TRUE, one of the two branches is explored; but in the case of FALSE condition, the other alternative is taken. Generally, the ‘IF-THEN’ is used to represent branch control.
3. **Loop (Repetition):** The Loop or Repetition allows a statement(s) to be executed repeatedly based on certain loop condition e.g. WHILE, FOR loops.

# Advantages of algorithm:

* It is a step-wise representation of a solution to a given problem, which makes it easy to understand.
* An algorithm uses a definite procedure.
* It is not dependent on any programming language, so it is easy to understand for anyone even without programming knowledge.
* Every step in an algorithm has its own logical sequence so it is easy to debu

# HOW TO WRITE ALGORITHMS

Step 1: **Define your algorithms input**: Many algorithms take in data to be processed, e.g. to calculate the area of rectangle input may be the rectangle height and rectangle width.

Step 2: **Define the variables**: Algorithm's variables allow you to use it for more than one place. We can define two variables for rectangle height and rectangle width as HEIGHT and WIDTH (or H & W). We should use meaningful variable name e.g. instead of using H & W use HEIGHT and WIDTH as variable name.

Step 3: **Outline the algorithm's operations:** Use input variable for computation purpose,

e.g. to find area of rectangle multiply the HEIGHT and WIDTH variable and store the value in new variable (say) AREA. An algorithm's operations can take the form of multiple steps and even branch, depending on the value of the input variables.

Step 4: Output **the results of your algorithm's operations**: In case of area of rectangle output will be the value stored in variable AREA. if the input variables described a rectangle with a HEIGHT of 2 and a WIDTH of 3, the algorithm would output the value of 6.

An example of an algorithm people use would be a recipe to make a cake.

"4 extra large eggs, beaten

1&1/2 C. stock

1/2 teaspoon salt 1 scallion, minced

1 C. small shrimp or lobster flakes

1 t. soy sauce

 1 Tablespoon oil

1 Mix all the ingredients, except the oil, in a deep bowl.

1. Mix all the ingredients, except the oil, in a deep bowl.
2. Put 1" water in wide pot, then place deep bowl of batter inside.
3. Cover pot tightly and steam 15 min.
4. Heat oil very hot and pour over custard.
5. Steam 5 more min.
6. Serve 4 people"

## This breaks down 'Making Chinese egg custard' into smaller steps. To make the product one still needs to know how to execute each of the steps in the procedure and understand all of the terms.

# FLOWCHART:

The first design of flowchart goes back to 1945 which was designed by John Von Neumann. Unlike an algorithm, Flowchart uses different symbols to design a solution to a problem. It is another commonly used programming tool. By looking at a Flowchart one can understand the operations and sequence of operations performed in a system. Flowchart is often considered as a blueprint of a design used for solving a specific problem.

## Advantages of flowchart:

* + Flowchart is an excellent way of communicating the logic of a program.
	+ Easy and efficient to analyze problem using flowchart.
	+ During program development cycle, the flowchart plays the role of a blueprint, which makes program development process easier.
	+ After successful development of a program, it needs continuous timely maintenance during the course of its operation. The flowchart makes program or system maintenance easier.
	+ It is easy to convert the flowchart into any programming language code.

# General Rules for flowcharting

* + 1. All boxes of the flowchart are connected with Arrows. (Not lines)
		2. Flowchart symbols have an entry point on the top of the symbol with no other entry points. The exit point for all flowchart symbols is on the bottom except for the Decision symbol.
		3. The Decision symbol has two exit points; these can be on the sides or the bottom and one side.

Generally a flowchart will flow from top to bottom. However, an upward flow can be shown as long as it does not exceed 3 symbols.

* + 1. Connectors are used to connect breaks in the flowchart. Examples are:
			- From one page to another page.
			- From the bottom of the page to the top of the same page.
			- An upward flow of more than 3 symbols
		2. Subroutines and Interrupt programs have their own and independent flowcharts.
		3. All flow charts start with a Terminal or Predefined Process (for interrupt programs or subroutines) symbol.
		4. All flowcharts end with a terminal or a contentious loop.

Flowcharting uses symbols that have been in use for a number of years to represent the type of operations and/or processes being performed. The standardised format provides a common method for people to visualise problems together in the same manner. The use of standardised symbols makes the flow charts easier to interpret; however, standardising symbols is not as important as the sequence of activities that make up the process.

**Flowcharting Tips**

* Chart the process the way it is really occurring. Do not document the way a written process or a manager thinks the process happens.
* People typically modify existing processes to enable a more efficient process. If the desired or theoretical process is charted, problems with the existing process will not be recognised and no improvements can be made.

Note all circumstances actually dealt with.

* Test the flow chart by trying to follow the chart to perform the process charted. If there is a problem performing the operation as charted, note any differences and modify the chart to correct. A better approach would be to have someone unfamiliar with the process try to follow the flow chart and note questions or problems found.

Include mental steps in the process such as decisions. These steps are sometimes left out because of familiarity with the process; however, represent sources of problems due to a possible lack of information used to make the decision can be inadequate or incorrect if performed by a different person.

**Flowchart** is diagrammatic /Graphical representation of sequence of steps to solve a problem. To draw a flowchart following standard symbols are use,

|  |  |  |
| --- | --- | --- |
| **Symbol Name** | **Symbol** | **Function** |
| Oval |  | Used to represent start and end of flowchart |
| Parallelogram |  | Used for input and output operation |
| Rectangle |  | Processing: Used for arithmetic operations and data-manipulations |
| Diamond |  | Decision making. Used to represent the operation in which there are two/three alternatives, true and false etc |
| Arrows |   | Flow line Used to indicate the flow of logic by connecting symbols |
| Circle |  | Page Connector |
|  |  | Off Page Connector |
|  |  | Predefined Process/Function Used to represent a group of statements performing one processing task. |
|  |  | Preprocessor |
|  | |----------------------- ||-------------- | Comments |

# PSEUDOCODE

**Pseudo code** is one of the tools that can be used to write a preliminary plan that can be developed into a computer program. **Pseudo code** is a generic way of describing an algorithm without use of any specific programming language syntax. It is, as the name suggests, *pseudo* code —it cannot be executed on a real computer, but it models and resembles real programming code, and is written at roughly the same level of detail.

Pseudo code, by nature, exists in various forms; although most borrow syntax from popular programming languages (like **C**, **Lisp**, or **FORTRAN**). Natural language is used whenever details are unimportant or distracting.

Computer science textbooks often use pseudo code in their examples so that all programmers can understand them, even if they do not all know the same programming languages. Since pseudo code style varies from author to author, there is usually an accompanying introduction explaining the syntax used.

In the algorithm design, the steps of the algorithm are written in free English text and, although brevity is desired, they may be as long as needed to describe the particular operation. The steps of an algorithm are said to be written in pseudo code.

Many languages, such as Pascal, have a syntax that is almost identical to pseudo code and hence make the transition from design to coding extremely easy.

The following section deals with the control structures (control constructs) Sequence, Selection and Iteration or Repetition.

# CONTROL STRUCTURES OR LOGICAL STRUCTURES

The key to better algorithm design and thus to programming lies in limiting the control structure to only three constructs. These are illustrated below:

# The sequence structure

The first type of control structures is called the sequence structure. This structure is the most elementary structure. The sequence structure is a case where the steps in an algorithm are constructed in such a way that, no condition step is required. The sequence structure is the logical equivalent of a straight line.

For example, suppose you are required to design an algorithm for finding the average of six numbers, and the sum of the numbers is given. The pseudocode will be as follows

*Start*

*Get the sum Average = sum / 6 Output the average Stop*

**Example 3:** This is the pseudo-code required to input three numbers from the keyboard and output the result.

*Use variables: sum, number1, number2, number3 of type integer Accept number1, number2, number3*

*Sum = number1 + number2 + number3 Print sum*

*End program*

**Example 4:** The following pseudo-code describes an algorithm which will accept two numbers from the keyboard and calculate the sum and product displaying the answer on the monitor screen.

*Use variables sum, product, number1, number2 of type real display “Input two numbers”*

*accept number1, number2 sum = number1 + number2 print “The sum is “, sum*

*product = number1 \* number2 print “The Product is “, product end program*

# Decision Structure or Selection Structure

The decision structure or mostly commonly known as a selection structure, is case where in the algorithm, one has to make a choice of two alternatives by making decision depending on a given condition.

Selection structures are also called caseselection structures when there are two or more alternatives to choose from.

# In pseudo code form we get

*If condition is true Then do task A*

*else*

*Do Task-B*

In this example, the condition is evaluated, if the condition is true Task-A is evaluated and if it is false, then Task-B is executed.

**If condition is true then**

*Do Task-A*

In this case, if condition is false, nothing happens. Otherwise Task-A is executed. The selection requires the following

* Choose alternative actions as a result of testing a logical condition
* Produce code to test a sequence of logical tests

## Making Choices

There are many occasions where a program is required to take alternative actions. For example, there are occasions where we need to take action according to the user choice. All computer languages provide a means of selection. Usually it is in the form of **If** statement and our pseudo-code is no exception to this.

We will use the if statement together with logical operators to test for true or false as shown below.

***If*** *a = b*

*print “a = b”*

The action is only taken when the test is true.

The logical operators used in our pseudo-code are

= is equal to

> is greater than

< is less than

>= is greater than or equal

<= is less than or equal

<> is not equal to

**Example 5:** The following shows how the selection control structure is used in a program where a user chooses the options for multiplying the numbers or adding them or subtracting.

*Use variables: choice, of the type character*

*ans, number1, number2, of type integer display “choose one of the following”*

*display “m for multiply” display “a for add” display “s for subtract” accept choice*

*display “input two numbers you want to use” accept number1, number2*

*if choice = m then ans = number1 \* number2 if choice = a then ans = number1 + number2 if choice = s then ans = number1 - number2 display ans*

## Compound Logical Operators

There are many occasions when we need to extend the conditions that are to be tested. Often there are conditions to be linked.

In everyday language we say things like ***If I had the time and the money I would go on holiday***. The ***and*** means that ***both conditions must be true*** before we take an action. We might also say ***I am happy to go to the theatre or the cinema***. The logical link this time is ***or.*** Conditions in **if** statements are linked in the same way. Conditions linked with **and** only result in an action when all conditions are true. For example, if a >b and a > c then display “a is the largest”. Conditions linked with an **or** lead to an action when either or both are true.

**Example 6:** The program is to input a examination mark and test it for the award of a grade. The mark is a whole number between 1 and 100. Grades are awarded according to the following criteria:

>= 80 Distinction

>= 60 Merit

>= 40 Pass

< 40 fail The pseudo-code is

*Use variables: mark of type integer If mark >= 80 display “distinction”*

*If mark >= 60 and mark < 80 display “merit” If mark >= 40 and mark < 60 display “pass” If mark < 40 display “fail”*

An **if** statement on its own is often not the best way of solving problems. A more elegant set of conditions can be created by adding an **else** statement to the if statement. The **else** statement is used to deal with situations as shown in the following examples.

**Example 7:** A person is paid at top for category 1 work otherwise pay is at normal rate.

*If the work is category 1 pay-rate is top*

*Else*

*pay-rate is normal*

The else statement provides a neat way of dealing with alternative condition. In pseudo- code we write

*If work = cat1 then p-rate: = top Else p-rate = normal*

Or

*If work = cat1 then p-rate: = top*

*Else*

*p-rate = normal*

The following example illustrate the use of if … else statements in implementing double alternative conditions.

*If salary < 50000 then Tax = 0*

*Else*

*If salary > 50000 AND salary < 100000 then Tax = 50000 \* 0.05*

*Else*

*Tax = 100000 \* 0.30*

**Some examples of Algorithm & flow charts:**

|  |
| --- |
| Example 1: Algorithm & Flowchart to find the sum of two numbers |
| **Algorithm**Step-1 StartStep-2 Input first numbers say A Step-3 Input second number say B Step-4 SUM = A + BStep-5 Display SUM Step-6 Stop |  |
|  |  |
| OR **Algorithm**Step-1 StartStep-2 Input two numbers say A & B Step-3 SUM = A + BStep-4 Display SUM Step-5 Stop |  |

|  |
| --- |
| **Example 2:Algorithm & Flowchart to find Simple Interest**  |
| P : Principle Amount N : Time in YearsR : % Annual Rate of Interest SI : Simple Interest**Algorithm**Step-1 StartStep-2 Input value of P, N, R Step-3 SI = (P x N x R)/100.0Step-4 Display SI F Step-6 Stop |
| **Example 3:Algorithm & Flowchart to find Compound Interest** |
| P : Principle Amount N : Time in YearsR : % Annual Rate of Interest CI : Compound Interest**Algorithm**Step-1 StartStep-2 Input value of P, N, R C Step-3 CI = P(1+R/100)N - PStep-4 Display CIStep-6 Stop |

****



**INTRODUCTION TO C**

* The C programming language is a structure oriented programming language, developed at Bell Laboratories in 1972 by Dennis Ritchie.
* C programming language features were derived from an earlier language called “B” (Basic Combined Programming Language – BCPL)
* C language was invented for implementing UNIX operating system.
* In 1978, Dennis Ritchie and Brian Kernighan published the first edition “The C Programming Language” and is commonly known as K&R C.
* In 1983, the American National Standards Institute (ANSI) established a committee to

provide a modern, comprehensive definition of C. The resulting definition, the ANSI standard, or “ANSI C”, was completed late 1988.

* Many of C’s ideas & principles were derived from the earlier language B, thereby naming this new language “C”.



Taxonomy of C Language

**Middle level language**

|  |  |  |
| --- | --- | --- |
| ***High Level*** |  ***Middle Level*** | ***Low Level*** |
| High level languages provide almost everything that the programmer might need to do as already built into the language | Middle level languages don’t provide all the built-in functions found in high level languages, but provides all building blocks that we need to produce the result we want | Low level languages provides nothing other than access to the machines basic instruction set |
| Examples: Java, Python | C, C++ | Assembler |

###### **CHARECTERESTICS OF A C PROGRAM**

* Small size – has only 32 keywords
* Extensive use of function calls- enables the end user to add their own functions to the C library.
* Supports loose typing – a character can be treated as an integer & vice versa.
* Structured language
* Low level (Bit Wise) programming readily available
* Pointer implementation - extensive use of pointers for memory, array, structures and functions.
* It has high-level constructs.
* It can handle low-level activities.
* It produces efficient programs.
* It can be compiled on a variety of computers.

|  |  |  |
| --- | --- | --- |
| ***Structure oriented*** | ***Object oriented*** | ***Non structure*** |
| In this type of language, largeprograms are divided into small programs called functions | In this type of language,programs are divided into objects | There is no specificstructure for programming this language |
| Prime focus is on functions andprocedures that operate on the data | Prime focus is in the data that isbeing operated and not on the functions or procedures | N/A |
| Data moves freely around thesystems from one function to another | Data is hidden and cannot be accessed by external functions | N/A |
| Program structure follows “TopDown Approach” | Program structure follows“Bottom UP Approach” | N/A |
| Examples:C, Pascal, ALGOL and Modula-2 | C++, JAVA and C# (C sharp) | BASIC, COBOL,FORTRAN |

###### **USES**

The C programming language is used for developing system applications that forms a major portion of operating systems such as Windows, UNIX and Linux. Below are some examples of C being used:

* Database systems
* Graphics packages
* Word processors
* Spreadsheets
* Operating system development
* Compilers and Assemblers
* Network drivers
* Interpreters

###### **STRUCTURE OF A C PROGRAM**

The structure of a C program is a protocol (rules) to the programmer, which he has to follow while writing a C program. The general basic structure of C program is shown in the figure below.

Based on this structure, we can sketch a C program.

**Example:**

/\* This program accepts a number & displays it to the user\*/

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

*{ int number;*

*printf( "Please enter a number: " ); scanf( "%d", &number );*

*printf( "You entered %d", number ); return 0;}*

Stepwise explanation:

***#include***

* The part of the compiler which actually gets your program from the source file is called the preprocessor.

 *#include <stdio.h>*

* #include is a pre-processor directive. It is not really part of our program, but instead it is an instruction to the compiler to make it do something. It tells the C compiler to include the contents of a file (in this case the system file called stdio.h).
* The compiler knows it is a system file, and therefore must be looked for in a special place, by the fact that the filename is enclosed in <> characters

***<stdio.h>***

* *stdio.h* is the name of the standard library definition file for all STanDard Input and Output functions.
* Your program will almost certainly want to send information to the screen and read things from the keyboard, and stdio.h is the name of the file in which the functions that we want to use are defined.
* The function we want to use is called printf. The actual code of printf will be tied in later by the linker.
* The ".h" portion of the filename is the language extension, which denotes an include file.

***void***

* This literally means that this means nothing. In this case, it is referring to the function whose name follows.
* Void tells to C compiler that a given entity has no meaning, and produces no error.

 ***main:***

* In this particular example, the only function in the program is called *main*.
* A C program is typically made up of large number of functions. Each of these is given a name by the programmer and they refer to each other as the program runs.
* C regards the name main as a special case and will run this function first i.e. the program execution starts from *main*.

***(void):***

* This is a pair of brackets enclosing the keyword *void*.
* It tells the compiler that the function main has no parameters.
* A parameter to a function gives the function something to work on.

***{ (Brace):***

* This is a brace (or curly bracket). As the name implies, braces come in packs of two - for every open brace there must be a matching close one.
* Braces allow us to group pieces of program together, often called a block.
* A block can contain the declaration of variable used within it, followed by a sequence of program statements.
* In this case the braces enclose the working parts of the function main.

***; (semicolon)***

* The semicolon marks the end of the list of variable names, and also the end of that declaration statement.
* All statements in C programs are separated by ";" (semicolon) characters.
* The ";" character is actually very important. It tells the compiler where a given statement ends.
* If the compiler does not find one of these characters where it expects to see one, then it will produce an error.

***scanf***

* In other programming languages, the printing and reading functions are a part of the language.
* In C this is not the case; instead they are defined as standard functions which are part of the language specification, but are not a part of the language itself.
* The standard input/output library contains a number of functions for formatted data transfer; the two we are going to use are scanf (scan formatted) and printf (print formatted).

***printf***

* The printf function is the opposite of scanf.
* It takes text and values from within the program and sends it out onto the screen.
* Just like scanf, it is common to all versions of C and just like scanf, it is described in the system file stdio.h.
* The first parameter to a printf is the format string, which contains text, value descriptions and formatting instructions.

###### **FILES USED IN A C PROGRAM**

* ***Source File*-** This file contains the source code of the program. The file extension of any c file is **.c**. The file contains C source code that defines the *main* function & maybe other functions.
* ***Header File*-** A header file is a file with extension **.h** which contains the C function declarations and macro definitions and to be shared between several source files.
* ***Object File-*** An object file is a file containing object cod[e,](http://en.wikipedia.org/wiki/Object_code) with an extension **.o**, meaning relocatable format machine cod[e](http://en.wikipedia.org/wiki/Machine_code) that is usually not directly executable. Object files are produced by an [assembler, compiler,](http://en.wikipedia.org/wiki/Assembler_%28computer_programming%29) or other [language translator,](http://en.wikipedia.org/wiki/Translator_%28computing%29) and used as input to the [linker,](http://en.wikipedia.org/wiki/Linker_%28computing%29) which in turn typically generates an [executable](http://en.wikipedia.org/wiki/Executable) or [library](http://en.wikipedia.org/wiki/Library_%28computing%29) by combining parts of object files.
* ***Executable File-*** The binary executable file is generated by the linker. The linker links the various object files to produce a binary file that can be directly executed.

**COMPLIATION & EXECUTION OF A C PROGRAM:**



**ELEMENTS OF C**

Every language has some basic elements & grammatical rules. Before starting with programming, we should be acquainted with the basic elements that build the language.

**Character Set**

Communicating with a computer involves speaking the language the computer understands. In C, various characters have been given to communicate.

Character set in C consists of;

|  |  |
| --- | --- |
| **Types** | **Character Set** |
| Lower case | a-z |
| Upper case | A-Z |
| Digits | 0-9 |
| Special Character | !@#$%^&\* |
| White space | Tab or new lines or space |

###### **Keywords**

Keywords are the words whose meaning has already been explained to the C compiler. The keywords cannot be used as variable names because if we do so we are trying to assign a new meaning to the keyword, which is not allowed by the computer.

There are only 32 keywords available in C. Below figure gives a list of these keywords for your ready reference.

###### **Identifier**

In the programming language C, an identifier is a combination of alphanumeric characters, the first being a letter of the alphabet or an underline, and the remaining being any letter of the alphabet, any numeric digit, or the underline.

Two rules must be kept in mind when naming identifiers.

1. The case of alphabetic characters is significant. Using "INDEX" for a variable is not the same as using "index" and neither of them is the same as using "InDeX" for a variable. All three refer to different variables.
2. As C is defined, up to 32 significant characters can be used and will be considered significant by most compilers. If more than 32 are used, they will be ignored by the compiler.

###### **Data Type**

In the C programming language, data types refer to a domain of allowed values & the operations that can be performed on those values. The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted. There are 4 fundamental data types in C, which are- *char, int, float &, double. Char* is used to store any single character; *int* is used to store any integer value, *float* is used to store any single precision floating point number & *double* is used to store any double precision floating point number. We can use 2 qualifiers with these basic types to get more types.

There are 2 types of qualifiers-

Sign qualifier- signed & unsigned Size qualifier- short & long

The data types in C can be classified as follows:

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

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Storage size** | **Value range** | **Precision** |
| float | 4 bytes | 1.2E-38 to 3.4E+38 | 6 decimal places |
| double | 8 bytes | 2.3E-308 to 1.7E+308 | 15 decimal places |
| long double | 10 bytes | 3.4E-4932 to 1.1E+4932 | 19 decimal places |

###### **Constants**

A constant is an entity that doesn’t change whereas a variable is an entity that may change. C constants can be divided into two major categories:



Here our only focus is on primary constant. For constructing these different types of constants certain rules have been laid down.

***Rules for Constructing Integer Constants:***

An integer constant must have at least one digit.

1. It must not have a decimal point.
2. It can be either positive or negative.
3. If no sign precedes an integer constant it is assumed to be positive.
4. No commas or blanks are allowed within an integer constant.
5. The allowable range for integer constants is-32768to 32767.

 Ex.: 426, +782,-8000, -7605

***Rules for Constructing Real Constants:***

Real constants are often called Floating Point constants. The real constants could be written in two forms—Fractional form and Exponential form.

***Rules for constructing real constants expressed in fractional form:***

1. A real constant must have at least one digit.
2. It must have a decimal point.
3. It could be either positive or negative.
4. Default sign is positive.
5. No commas or blanks are allowed within a real constant. Ex. +325.34, 426.0, -32.76, -48.5792

***Rules for constructing real constants expressed in exponential form:***

1. The mantissa part and the exponential part should be separated by a letter e.
2. The mantissa part may have a positive or negative sign.
3. Default sign of mantissa part is positive.
4. The exponent must have at least one digit, which must be a positive or negative integer. Default sign is positive.
5. Range of real constants expressed in exponential form is -3.4e38 to 3.4e38. Ex. +3.2e-5, 4.1e8, -0.2e+3, -3.2e-5

***Rules for Constructing Character Constants:***

1. A character constant is a single alphabet, a single digit or a single special symbol enclosed within single inverted commas.
2. The maximum length of a character constant can be 1 character.

Ex.: ‘M’, ‘6’, ‘+’

**VARIABLES**

Variables are names that are used to store values. It can take different values but one at a time. A data type is associated with each variable & it decides what values the variable can take. When you decide your program needs another variable, you simply declare (or define) a new variable and C makes sure you get it. You declare all C variables at the top of whatever blocks of code need them. Variable declaration requires that you inform C of the variable's name and data type. Syntax – data type variable name;

*Eg:*

*int page\_no;*

*char grade;*

*float salary*

*long y;*

###### **Declaring Variables:**

There are two places where you can declare a variable:

* + After the opening brace of a block of code (usually at the top of a function)
	+ Before a function name (such as before main() in the program) Consider various examples:

Suppose you had to keep track of a person's first, middle, and last initials. Because an initial is obviously a character, it would be prudent to declare three character variables to hold the three initials. In C, you could do that with the following statement:

1. *main()*

*{*

*char first, middle, last;*

*// Rest of program follows*

*}*

1. *main()*

*{ char first; char middle; char last;*

*// Rest of program follows*

*}*

###### **Initialization of Variables**

When a variable is declared, it contains undefined value commonly known as garbage value. If we want we can assign some initial value to the variables during the declaration itself. This is called *initialization of the variable.*

Eg- int pageno=10;

char grade=’A’;

float salary= 20000.50;

###### **Expressions**

An expression consists of a combination of operators, operands, variables & function calls. An expression can be arithmetic, logical or relational. Here are some expressions:

a+b *–* arithmetic operation a>b- relational operation a

== b - logical operation func (a,b) – function call

4+21

a\*(b + c/d)/20 q = 5\*2 x =

++q % 3

q > 3

As you can see, the operands can be constants, variables, or combinations of the two. Some expressions are combinations of smaller expressions, called sub expressions. For example, c/d is a sub expression of the sixth example.

An important property of C is that every C expression has a value. To find the value, you perform the operations in the order dictated by operator precedence.

###### **Statements**

Statements are the primary building blocks of a program. A program is a series of statements with some necessary punctuation. A statement is a complete instruction to the computer. In C, statements are indicated by a semicolon at the end. Therefore

*legs = 4*

is just an expression (which could be part of a larger expression), but

*legs = 4;*

is a statement.

What makes a complete instruction? First, C considers any expression to be a statement if you append a semicolon. (These are called expression statements.) Therefore, C won't object to lines such as the following:

*8;*

*3 + 4;*

However, these statements do nothing for your program and can't really be considered sensible statements. More typically, statements change values and call functions:

*x = 25;*

*++x;*

*y = sqrt(x);*

Although a statement (or, at least, a sensible statement) is a complete instruction, not all complete instructions are statements. Consider the following statement:

*x = 6 + (y = 5);*

In it, the sub expression y = 5 is a complete instruction, but it is only part of the statement. Because a complete instruction is not necessarily a statement, a semicolon is needed to identify instructions that truly are statements.

###### **Compound Statements (Blocks)**

A compound statement is two or more statements grouped together by enclosing them in braces; it is also called a block. The following while statement contains an example:

*while (years < 100)*

*{*

*wisdom = wisdom \* 1.05; printf("%d %d\n", years, wisdom); years = years + 1;*

*}*

If any variable is declared inside the block then it can be declared only at the beginning of the block. The variables that are declared inside a block can be used only within the block.

**Module-2**

**1. Arithmetic Expression**

An **arithmetic expression** is composed of operators and operands. Operators act on operands to yield a result. Commonly used **arithmetic** operators are +, -, \*, / and %. ... In the last **expression**, parentheses are used to perform a certain operation first. This is because in **C**, operators follow a precedence rule.

**2. Operators**

An operator is a symbol that operates on a value or a variable. For example: + is an operator to perform addition.C has a wide range of operators to perform various operations.

**a) Arithmetic Operators**

An arithmetic operator performs mathematical operations such as addition, subtraction, multiplication, division etc on numerical values (constants and variables).

Operator Meaning of Operator

+ Addition or unary plus

- Subtraction or unary minus

\* Multiplication

/ Division

% remainder after division (modulo division)

**Example 1: Arithmetic Operators**

#include <stdio.h>

int main()

{

int a = 9,b = 4, c;

c = a+b;

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

c = a-b;

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

c = a\*b;

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

c = a/b;

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

c = a%b;

printf("Remainder when a divided by b = %d \n",c);

return 0;

}

**Output**

a+b = 13

a-b = 5

a\*b = 36

a/b = 2

Remainder when a divided by b=1

The operators +, - and \* computes addition, subtraction, and multiplication respectively as you might have expected.

In normal calculation, 9/4 = 2.25. However, the output is 2 in the program.

It is because both the variables a and b are integers. Hence, the output is also an integer. The compiler neglects the term after the decimal point and shows answer 2 instead of 2.25.

The modulo operator % computes the remainder. When a=9 is divided by b=4, the remainder is 1. The % operator can only be used with integers.

Suppose a = 5.0, b = 2.0, c = 5 and d = 2. Then in C programming,

// Either one of the operands is a floating-point number

a/b = 2.5

a/d = 2.5

c/b = 2.5

// Both operands are integers

c/d = 2

**b) Increment and Decrement Operators**

C programming has two operators increment ++ and decrement -- to change the value of an operand (constant or variable) by 1.Increment ++ increases the value by 1 whereas decrement -- decreases the value by 1. These two operators are unary operators, meaning they only operate on a single operand.

**Example 2: Increment and Decrement Operators.**

#include <stdio.h>

int main()

{

int a = 10, b = 100;

 float c = 10.5, d = 100.5;

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

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

printf("++c = %f \n", ++c);

printf("--d = %f \n", --d);

return 0;

}

**Output**

++a = 11

--b = 99

++c = 11.500000

++d = 99.500000

Here, the operators ++ and -- are used as prefixes. These two operators can also be used as postfixes like a++ and a--. Visit this page to learn more about how increment and decrement operators work when used as postfix.

**c) Assignment Operators**

An assignment operator is used for assigning a value to a variable. The most common assignment operator is = .

Operator Example Same as

= a = b a = b

+= a += b a = a+b

-= a -= b a = a-b

\*= a \*= b a = a\*b

/= a /= b a = a/b

%= a %= b a = a%b

**Example 3: Assignment Operators.**

#include <stdio.h>

int main()

{

int a = 5, c;

 c = a;

printf("c = %d\n", c);

 c += a;

printf("c = %d\n", c);

c -= a;

printf("c = %d\n", c);

c \*= a;

printf("c = %d\n", c);

c /= a;

printf("c = %d\n", c);

c %= a;

printf("c = %d\n", c);

 return 0;

}

Output

c = 5

c = 10

c = 5

c = 25

c = 5

c = 0

**d) Relational Operators**

A relational operator checks the relationship between two operands. If the relation is true, it returns 1; if the relation is false, it returns value 0.Relational operators are used in decision making and loops.

Operator Meaning of Operator Example

== Equal to 5 == 3 is evaluated to 0

> Greater than 5 > 3 is evaluated to 1

< Less than 5 < 3 is evaluated to 0

!= Not equal to 5 != 3 is evaluated to 1

>= Greater than or equal to 5 >= 3 is evaluated to 1

<= Less than or equal to 5 <= 3 is evaluated to 0

**Example 4: Relational Operators**

#include <stdio.h>

int main()

{

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

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

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

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

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

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

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

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

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

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

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

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

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

 return 0;

}

Output

5 == 5 is 1

5 == 10 is 0

5 > 5 is 0

5 > 10 is 0

5 < 5 is 0

5 < 10 is 1

5 != 5 is 0

5 != 10 is 1

5 >= 5 is 1

5 >= 10 is 0

5 <= 5 is 1

5 <= 10 is 1

**e) Logical Operators**

An expression containing logical operator returns either 0 or 1 depending upon whether expression results true or false. Logical operators are commonly used in decision making in C programming.

Operator Meaning Example

&& Logical AND. True only if all operands are true,If c = 5 and d = 2 then, expression ((c==5) && (d>5)) equals to 0.

|| Logical OR. True only if either one operand is true, If c = 5 and d = 2 then, expression ((c==5) || (d>5)) equals to 1.

! Logical NOT. True only if the operand is 0, If c = 5 then, expression!(c==5) equals to 0.

**Example 5: Logical Operators**

#include <stdio.h>

int main()

{

 int a = 5, b = 5, c = 10, result;

 result = (a == b) && (c > b);

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

 result = (a == b) && (c < b);

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

 result = (a == b) || (c < b);

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

 result = (a != b) || (c < b);

 printf("(a != b) || (c < b) is %d \n", result);

 result = !(a != b);

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

 result = !(a == b);

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

 return 0;

}

Output

(a == b) && (c > b) is 1

(a == b) && (c < b) is 0

(a == b) || (c < b) is 1

(a != b) || (c < b) is 0

!(a != b) is 1

!(a == b) is 0

Explanation of logical operator program

(a == b) && (c > 5) evaluates to 1 because both operands (a == b) and (c > b) is 1 (true).

(a == b) && (c < b) evaluates to 0 because operand (c < b) is 0 (false).

(a == b) || (c < b) evaluates to 1 because (a = b) is 1 (true).

(a != b) || (c < b) evaluates to 0 because both operand (a != b) and (c < b) are 0 (false).

!(a != b) evaluates to 1 because operand (a != b) is 0 (false). Hence, !(a != b) is 1 (true).

!(a == b) evaluates to 0 because (a == b) is 1 (true). Hence, !(a == b) is 0 (false).

**f) Bitwise Operators**

During computation, mathematical operations like: addition, subtraction, multiplication, division, etc are converted to bit-level which makes processing faster and saves power.Bitwise operators are used in C programming to perform bit-level operations.

Operators Meaning of operators

& Bitwise AND

| Bitwise OR

^ Bitwise exclusive OR

~ Bitwise complement

<< Shift left

>> Shift right

**g) Other Operators**

**Comma Operator:** Comma operators are used to link related expressions together. For example:

int a, c = 5, d;

**The sizeof operator:** The sizeof is a unary operator that returns the size of data (constants, variables, array, structure, etc).

**Example 6: sizeof Operator**

#include <stdio.h>

int main()

{

 int a;

 float b;

 double c;

 char d;

 printf("Size of int=%lu bytes\n",sizeof(a));

 printf("Size of float=%lu bytes\n",sizeof(b));

 printf("Size of double=%lu bytes\n",sizeof(c));

 printf("Size of char=%lu byte\n",sizeof(d));

 return 0;

}

Output

Size of int = 4 bytes

Size of float = 4 bytes

Size of double = 8 bytes

Size of char = 1 byte

**3. Precedence**

This rule of **priority** of operators is called operator **precedence**. In **C**, **precedence** of arithmetic operators( \*, %, /, +, -) is higher than relational operators(==, != , >, <, >=, <=) and **precedence** of relational operator is higher than logical operators(&&, || and !).

**4. Conditional branching**

Conditional statements help you to make a decision based on certain conditions. These conditions are specified by a set of conditional statements having Boolean expressions which are evaluated to a Boolean value true or false. There are following types of conditional statements in C.

If statement, If-Else statement, Nested If-else statement, If-Else If ladder, Switch statement

**a) If statement**

The single if statement in C language is used to execute the code if a condition is true. It is also called one-way selection statement.

Syntax

if(expression)

{

 //code to be executed

}

If the expression is evaluated to nonzero (true) then if block statement(s) are executed. If the expression is evaluated to zero (false) then Control passes to the next statement following it.

**If Statement Example**

#include<stdio.h>

#include<conio.h>

void main()

{

int num=0;

printf("enter the number");

scanf("%d",&num);

if(n%2==0)

{

printf("%d number in even",num);

}

getch();

}

**b) If-else statement**

The if-else statement in C language is used to execute the code if condition is true or false. It is also called two-way selection statement.

Syntax

if(expression)

{

 //Statements

}

else

{

 //Statements

}

If the expression is evaluated to nonzero (true) then if block statement(s) are executed. If the expression is evaluated to zero (false) then else block statement(s) are executed.

**If else Statement Example**

#include<stdio.h>

#include<conio.h>

void main()

{

int num=0;

printf("enter the number");

scanf("%d",&num);

if(n%2==0)

{

printf("%d number in even", num);

}

else

{

printf("%d number in odd",num);

}

getch();

}

**c) Nested If-else statement**

The nested if...else statement is used when a program requires more than one test expression. It is also called a multi-way selection statement. When a series of the decision are involved in a statement, we use if else statement in nested form.

Syntax

if( expression )

{

 if( expression1 )

 {

 statement-block1;

 }

 else

 {

 statement-block 2;

 }

}

else

{

 statement-block 3;

}

**Example**

#include<stdio.h>

#include<conio.h>

void main( )

{

 int a,b,c;

 clrscr();

 printf("Please Enter 3 number");

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

 if(a>b)

 {

 if(a>c)

 {

 printf("a is greatest");

 }

 else

 {

 printf("c is greatest");

 }

 }

 else

 {

 if(b>c)

 {

 printf("b is greatest");

 }

 else

 {

 printf("c is greatest");

 }

 }

getch();

}

**d) If else If ladder**

The if-else-if statement is used to execute one code from multiple conditions. It is also called multipath decision statement. It is a chain of if..else statements in which each if statement is associated with else if statement and last would be an else statement.

Syntax

if(condition1)

{

 //statements

}

else if(condition2)

{

 //statements

}

else if(condition3)

{

 //statements

}

else

{

 //statements

}

 **If else If ladder Example**

#include<stdio.h>

#include<conio.h>

void main( )

{

 int a;

 printf("enter a number");

 scanf("%d",&a);

 if( a%5==0 && a%8==0)

 {

 printf("divisible by both 5 and 8");

 }

 else if( a%8==0 )

 {

 printf("divisible by 8");

 }

 else if(a%5==0)

 {

 printf("divisible by 5");

 }

 else

 {

 printf("divisible by none");

 }

getch();

}

**f) Switch Statement**

switch statement acts as a substitute for a long if-else-if ladder that is used to test a list of cases. A switch statement contains one or more case labels which are tested against the switch expression. When the expression match to a case then the associated statements with that case would be executed.

Syntax

Switch (expression)

{

 case value1:

 //Statements

 break;

 case value 2:

 //Statements

 break;

 case value 3:

 //Statements

 case value n:

 //Statements

 break;

 Default:

 //Statements

}

**Switch statement Example**

#include<stdio.h>

#include<conio.h>

void main( )

{

 char grade = 'B';

 if (grade == 'A')

 {

 printf("Excellent!");

 }

 else if (grade == 'B')

 {

 printf("Well done");

 }

 else if (grade == 'D')

 {

 printf("You passed");

 }

 else if (grade == 'F')

 {

 printf("Better try again");

 }

 else

 {

 printf("You Failed!");

 }

 }

getch();

}

Note

The switch statement must be an integral type.

Case labels must be constants.

Case labels must be unique.

Case labels must end with a colon.

The break statement transfers the control out of switch statement.

The break statement is optional.

**5) Iteration and loop**

A **Loop** executes the sequence of statements many times until the stated condition becomes false. A loop consists of two parts, a body of a loop and a control statement. The control statement is a combination of some conditions that direct the body of the loop to execute until the specified condition becomes false. The purpose of the loop is to repeat the same code a number of times.

**5.1) Types of Loops**

Depending upon the position of a control statement in a program, a loop is classified into two types:

1. Entry controlled loop

2. Exit controlled loop

In an entry controlled loop, a condition is checked before executing the body of a loop. It is also called as a pre-checking loop.

In an exit controlled loop, a condition is checked after executing the body of a loop. It is also called as a post-checking loop.

**a) While Loop**

A while loop is the most straightforward looping structure. The basic format of while loop is as follows:

while (condition)

 {

 statements;

}

It is an entry-controlled loop. In while loop, a condition is evaluated before processing a body of the loop. If a condition is true then and only then the body of a loop is executed. After the body of a loop is executed then control again goes back at the beginning, and the condition is checked if it is true, the same process is executed until the condition becomes false. Once the condition becomes false, the control goes out of the loop.

After exiting the loop, the control goes to the statements which are immediately after the loop. The body of a loop can contain more than one statement. If it contains only one statement, then the curly braces are not compulsory. It is a good practice though to use the curly braces even we have a single statement in the body.

In while loop, if the condition is not true, then the body of a loop will not be executed, not even once. It is different in do while loop which we will see shortly.

**Example while loop**

#include<stdio.h>

#include<conio.h>

int main()

{

int num=1; //initializing the variable

while(num<=10) //while loop with condition

{

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

num++; //incrementing operation

}

return 0;

}

Output:

1

2

3

4

5

6

7

8

9

10

**b) Do-While loop**

A do-while loop is similar to the while loop except that the condition is always executed after the body of a loop. It is also called an exit-controlled loop.

The basic format of while loop is as follows:

 do

{

 statements

} while (expression);

As we saw in a while loop, the body is executed if and only if the condition is true. In some cases, we have to execute a body of the loop at least once even if the condition is false. This type of operation can be achieved by using a do-while loop.

In the do-while loop, the body of a loop is always executed at least once. After the body is executed, then it checks the condition. If the condition is true, then it will again execute the body of a loop otherwise control is transferred out of the loop.

Similar to the while loop, once the control goes out of the loop the statements which are immediately after the loop is executed.

The critical difference between the while and do-while loop is that in while loop the while is written at the beginning. In do-while loop, the while condition is written at the end and terminates with a semi-colon (;)

**Example of do while loop**

#include<stdio.h>

#include<conio.h>

int main()

{

int num=1; //initializing the variable

do //do-while loop

{

printf("%d\n",2\*num);

num++; //incrementing operation

}while(num<=10);

return 0;

}

Output:

2

4

6

8

10

12

14

16

18

20

**c) For loop**

A for loop is a more efficient loop structure in 'C' programming. The general structure of for loop is as follows:

for (initial value; condition; increment or decrement )

{

Statements;

}

The initial value of the for loop is performed only once. The condition is a Boolean expression that tests and compares the counter to a fixed value after each iteration, stopping the for loop when false is returned.The increment/decrement increases (or decreases) the counter by a set value.

**Example of for loop**

#include<stdio.h>

int main()

{

int number;

for(number=1;number<=10;number++) //for loop to print 1-10 numbers

{

printf("%d\n",number); //to print the number

}

return 0;

}

Output:

1

2

3

4

5

6

7

8

9

10

**6. Array**

 An array is defined as finite ordered collection of homogenous data, stored in contiguous memory locations. Here the words, finite means data range must be defined, ordered means data must be stored in continuous memory addresses, and homogenous means data must be of similar data type.

Example where arrays are used, to store list of Employee or Student names, to store marks of students,

Or to store list of numbers or characters etc.

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

**6.1 Declaring an Array**

Like any other variable, arrays must be declared before they are used.

General form of array declaration is:

Data-type variable-name [size];

**/\* Example of array declaration \*/**

int arr[10];

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

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

**6.2 Initialization of an Array**

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

**Compile time Array initialization**

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

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

/\* Here are a few examples \*/

int marks[4]={ 67, 87, 56, 77 }; // integer array initialization

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

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

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

#include<stdio.h>

void main()

{

 int i;

 int arr[] = {2, 3, 4}; // Compile time array initialization

 for(i = 0 ; i < 3 ; i++)

 {

 printf("%d\t",arr[i]);

 }

}

Output

2 3 4

**Runtime Array initialization**

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

#include<stdio.h>

void main()

{

 int arr[4];

 int i, j;

printf("Enter array element");

for(i = 0; i < 4; i++)

{

 scanf("%d", &arr[i]); //Run time array initialization

 }

 for(j = 0; j < 4; j++)

 {

 printf("%d\n", arr[j]);

 }

}

**6.3 Two dimensional Arrays**

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

**6.3.1 Two-dimensional arrays declaration**

Data-type array-name[row-size][column-size]

/\* Example \*/

int a[3][4];

**6.3.2 Two dimensional array initialization**

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

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

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

**Runtime initialization of a two dimensional Array**

#include<stdio.h>

void main()

{

int arr[3][4];

int i, j, k;

 printf("Enter array element");

 for(i = 0; i < 3;i++)

 {

 for(j = 0; j < 4; j++)

 {

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

 }

 }

 for(i = 0; i < 3; i++)

 {

 for(j = 0; j < 4; j++)

 {

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

 }

 }

}

**7.1 Character arrays and string**

Strings are actually one-dimensional array of characters terminated by a null character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a null.

The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

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

If you follow the rule of array initialization then you can write the above statement as follows −

char greeting[] = "Hello";

Actually, you do not place the null character at the end of a string constant. The C compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print the above mentioned string −

#include <stdio.h>

int main ()

 {

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

printf("Greeting message: %s\n", greeting );

return 0;

}

When the above code is compiled and executed, it produces the following result -

Greeting message: Hello

C supports a wide range of functions that manipulate null-terminated strings −

Serial No. Function & Purpose

1. strcpy(s1, s2);

Copies string s2 into string s1.

1. strcat(s1, s2);

 Concatenates string s2 onto the end of string s1.

1. strlen(s1);

 Returns the length of string s1.

1. strcmp(s1, s2);

 Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.

1. strchr(s1, ch);

 Returns a pointer to the first occurrence of character ch in string s1.

1. strstr(s1, s2);

 Returns a pointer to the first occurrence of string s2 in string s1.

The following example uses some of the above-mentioned functions

#include <stdio.h>

#include <string.h>

int main ()

{

 char str1[12] = "Hello";

 char str2[12] = "World";

 char str3[12];

 int len ;

 /\* copy str1 into str3 \*/

 strcpy(str3, str1);

 printf("strcpy( str3, str1) : %s\n", str3 );

 /\* concatenates str1 and str2 \*/

 strcat( str1, str2);

 printf("strcat( str1, str2): %s\n", str1 );

 /\* total lenghth of str1 after concatenation \*/

 len = strlen(str1);

 printf("strlen(str1) : %d\n", len );

 return 0;

}

When the above code is compiled and executed, it produces the following result −

strcpy( str3, str1) : Hello

strcat( str1, str2): HelloWorld

strlen(str1) : 10

<https://www.programiz.com/c-programming/c-operators>

<https://www.guru99.com/c-loop-statement.html>

<https://www.studytonight.com/c/arrays-in-c.php>

Module-3

Function in C Language

A function in C language is a block of statement that performs a specifc task.

Why we need function ?

Suppose you are building a programme and you need to perform same task more than once. In such a case you can avoid writing the same code again and again by using function tecnique,that is create a function to perform that task , and just call it every time you need to perform that task.

Some of you may say that I am very good in typing and I can write the code again and again, and I do not need to use the function.

Let me know

WHY I NEED TO USE FUNCTION ?

Function is used because of the following reasons.

1. Improve the readability of code.

2. Improve the reusaility of the code, same function can be used in any programe rather that writing the same code from scratch.

3. Debugging of the code would be easier if you use function, as error are easy to be traced.

4. Reduce the size of the code, duplicate set of statement are replaced by function call.

Before we proceed to know the design structure of function we must know that some function are already designed by the c developer and we are provided with a well defined step (procedure ) to design our own function, from this point of view we have two type of function

1.Predefined standard Library Functions-:Thease are the function which already have a defination in header files,so we just call them whenever there is a need to use them(Note-:we need to include the header file at the beginning in which it is definded and generally thease files are stored with .h extention.)

Example-:puts(),gets(),printf(),scanf() etc.

2.User Defined functions-:The function the we create in a programme are know as user defined functions

Now we know how to use predefined function but the questin is

How to design our own function (user defined function)?

Let’s start with the syntax-:

return\_type function\_name (argument list)

{

set of statemet(specific block of code to be repeated)

}



**return type** : It can be of any data type sucn as int,double,char,void,short etc.

**function name**-:A meaning full name is given such as fact to find the factorial,square to find the square of a numer,total\_salary to calculate total salary etc.(Note-: Identifier naming rule are to be followed while providing a function name ).

**argument list**-: contains variable name along with their data type. Thease arguments are kind of input for the function.for example a function which is used to add two integer variable , will be having two integer argument and can be written as int x,int y.

Note-:when we donot want to provide any input to function we can use keyword void as argument.

**Block of code-:**Set of C statements, which will be executed whenever a call will be made to the funciton.

**Let’s take an example-:**Suppose you want to create a function to add two integer variables. A meaning full name can be given as function such as **add**,**sum**,**addition.**Since we are planing to add two integer values the result will be integer hence the return type of the function should be integer data type (int).

We want to input two integer value to the function hence our argument list will be two integer type.we refer two integer parameters

So we can design basic structure that is-: int add(int x,int y)

This is called function prototype of function signature.

Note-: do not put semicolon since complete design structure is to be written next.



***Example 1:Creating a user defined function add()***

#include <stdio.h>

int add(int x, int y)

{

int sum;

sum=x+y; //arguments are used here

return (sum); // function return tpe is integer so we are returning an integer value.That is sum .

} // End of function design

int main()

{

Int var1,var2,result;

printf(“\n Enter the first numer :”);

scanf(“%d”,&var1);

printf (“\n Enter the second number: “);

scanf(“%d”,&var2);

//we wil call the function here

result=add(var1,var2);

printf(“\n The sum of two no is= %d”,result);

return (0);

}//End of main functon

Note-: since the funciton return an integer value we must have one variable to hold the value hence it is written result = add(var1,var2);

***Example 1:Creating a user defined function that doesn’t return anything***

#include <stdio.h>

void add(int num1, int num2)

{

int sum;

sum=num1+num2; //arguments are used here

printf(“\n The sum of two no is= %d”,sum);

} // End of function design

int main()

{

Int var1,var2;

printf(“\n Enter the first numer :”);

scanf(“%d”,&var1);

printf (“\n Enter the second number: “);

scanf(“%d”,&var2);

//we wil call the function here

addition(var1,var2);

}//End of main functon

Note-: since the funciton does not return value we can directly call the function add(var1,var2);

Few more points regarding function in C

1. Main() is a first function in which we usually write our programme.
2. Each c programme must have at least one function called main()
3. There is no limit on number of function we can design. C programme can have any number of functions.
4. A function which call itself is called Recursive function.
5. A function design has three parts such as **prototype of function, function** **call, and function defination.**
6. Prototype declaration is not required if we define the function before beginning of the main function as it will be automatically detected by compiler ,shown in the above example.
7. Function prototype provides three incormation to the compiler and they are

Name of functon ,return type of the function and no of argument it will accept.

1. Input to a function is called parameter passing or argument list .
2. We can also pass structure , address , array as input to function.
3. The variable used inside the main function are called actual argument and the variable used inside the function is called formal argument .

Types of function call-:Depends on how we want to use the formal and actual argument function call can be devided into two categories.

1. Call by value method-: in the call by value methods the actual arguments are copied to the formal arguments , hence any operation performed by function on arguments doesn’t affect actual parameter.
2. Call by reference method-: unlike call by value, in this method, address of actual arguments or parameter is passed to the formal parameters, which means any operation performed on formal parameters affects the value of actual parameters.



**Actual parameters:** The parameters that appear in function calls.
**Formal parameters:** The parameters that appear in function declarations.



Order in which argument passed to a function-:When function call is encountered there are two possibilities in which arguments are passed to function

1. Argument might be passed from left to right .
2. Argument might be passed from right to left.

C language follows the second technique that is right to left.

If variable passed to a function are independent , order in which it is passed is not important however in some function call the order of passing argumet become important consideration

For example-:

Int a=1;

Printf(“%d%d%d”,a,++a,a++);

It seems the out put will be 1 2 3 . Surprisingly the out put is 3 3 1.

This is because arguments are passed to the function from right to left.

1. 1 is passed throuth the expression a++ (a is increamented to 2)
2. 2 is passed through the expression ++a( a is increamented to 3)
3. 3 is the latest value and it is passed to expression a(a is now 3)

Thus we get the output 3 3 1.

**Advance Features of Function-:**

Following are the advance features of function which will be discussed

1. Function Declaration and Prototypes.
2. Calling function by value or by reference
3. Recursion
4. Pointer to function

1.Function Declaration and Prototypes.

Any C functon by default returns an integer value.(compiler assumes).if we desire that a function should return a value other than an int , then it is necessary to explicitly mention in te return type .

Example-:

#include<stdio.h>

square (float x)

{

float y;

y=x\*x;

return(y);

}

main()

{

float a,b;

printf(“\n Enter any number-: ”);

scanf(“%f”,&a);

b=square(a);

printf(“\n Square of %f is=%f”,a,b);

return(0);

}

The output of this programme is after three executions is as shown below.

Enter any number-: 4

Square of 4 is = 16.000000

Enter any number -: 1.5

Square of 1.5 is 2.000000

Enter any number 2.5

Square of 2.5 is 6.000000

The first out put is correct but the next two outputs is wrong.this is because any C function by default always returns an integer value and though the square is calculated correctly inside the function but it is not able to return the float value .

To overcome this we have to specify the return type and the correct code will be

#include <stdio.h>

main()

{

float a,b;

float square (float x); //prototype declaration

printf(“\n Enter any number-: ”);

scanf(“%f”,&a);

b=square(a); // function call

printf(“\n Square of %f is=%f”,a,b);

return(0);

}

float square (float x) //function defination

 {

float y;

y=x\*x;

return(y);

 }

Now you will get the excepted answer.that is square of 1.5 is 2.25 and so on.

So prototype declaration informs the compiler about function signature that is the return type , function name and no of argument as well as their type.

Sometime we do not want the function should return any value . in this case the return type must be specified as void .

Example-:

void display()

{

printf(“ this function do not return any value”);

}

Main()

{

display();

return(0);

}

In the above declaration function display(), do not returns any value hence we have to specify the return type as void.

Remember the above way of writing function is also know as call by value Since we are passing value through variable.

Then question is can we pass the address of a variable to a function and the answer is yes we can. But before that you must have knowledge about pointer and address concept .

Here is the introduction about pointer-:

When we declare the following line

Int i=3;

The declaration tells the C compiler to

1. Reserve space in memory to hold the integer value
2. Associate the name I with the memory location.
3. Store the value 3 at this location.

Name of variable location in memory stored value

 i -------------------------🡪65524---------------🡪3

we can think that computer has selected memory location 65524 as a place to store the value .note that location is allocated by compiler and second time it may be different and it depends upon availability of free spae.

We can also print the address of variable location by the following programme

#include<stdio.h>

void main()

{

Int i=3;

printf(“The address of variable i is = %u”,&i);

printf(“The value stored at this location is=%d”,i);

return(0);

}

Note-:%u is format specifier for printing an unsigned integer.

The pointer operator available in C is ‘\*’ is also called indirection operator. And the value at address operator \*(&).

*Here is the example*-:

#include<stdio.h>

Main()

{

Int i=3;

Printf(“\n Address of i=%u”,&i);

Printf(“\nValue of i=%d”,i);

Printf(“\n Value of i=%d”,\*(&i));

}

The out put of the above programme

Address of i=65524

Value of i=3

Value of i=3

the above programme demonstrate the way we can indirectly access the content of memory location by refering to the address of the variable. C language provide this technique through Pointer .We can define a pointer as a variable which can store the address of another variable

For example

Int \*ptr; here ptr is a pointer variable

And when we write int j=5 and ptr=&j;

Pointer ptr stors the address of j variable not the content.

Here is programme example to understand this

#include<stdio.h>

Main()

{

Int i=3;

Int \*ptr;

ptr=&i;

printf (“\n Address of i=%u”,&i);

printf (“\n value of i=%d”,i)

printf (“\n Address of ptr=%u”,&ptr);

printf (“\n value of ptr=%u”,ptr);

printf (“\n value of i trhough ptr =%d”,\*ptr);

return(0);

}

Out put of the above programme is

 Address of i=65524

value of i=3

Address of ptr=65522

value of ptr=65524

value of i trhough ptr =3

From the above example it is clear that we can access the content of variable indirectly through pointer and that is possible by accessing address of that variable.

Note-:please refer more example about pointer ,double pointer etc

**Back to Function Call**

**Pass by reference-:** in this case we pass the address of variable to a function. Which means the argument of function or parameter of function will be able to store the address of the variable.

Here is a programme example to swap the content of two variable using pass by reference technique.

#include <stdio.h>

main()

{

Int a=10,b=20;

void swap(int \*p, int \*q);

printf(“\n before swap the value of a=%d and b=%d”,a,b);

swap(&a,&b);

printf(“\n after swap the value of a=%d and b=%d”,a,b);

return (0);

}

swap(int \*p,int \*q)

{

Int t;

t=\*p;

\*p=\*q;

\*q=t;

}

In the above programme it is clear that we are passing address of variable a and b to the function swap and indirectly the content of variable is swapped in side the function through pointer . since the content are changed through the address

The original value of a and b variable are swapped

The out put of the above programe is

before swap the value of a=10 and b=20

after swap the value of a=20 and b=10

the outcome of the above discussion is

1. If we want te value of an actual argument should not get changed in the function being called , we will use pass by value or call by value technique
2. If we want the value of an actual argumet should get chaged in the function being called, we will use pass by reference or call by reference technique.
3. Function always return only one value hence in case we want more than one value to be returned then we use pass by reference or call by reference tecnique.

**Recursion-:**
A function is called recursive if a statement within the body of the function calls the same function.That is a technique of defining a function in terms of its own defination.

 Recursive algorithm and programme has following advantage

1. Elegant
2. Simple and prove to be correct
3. Easy to implement

The only disadvantages is we have to design **base case** carefully to stop the execution or otherwise it will be a infinite loop.

Example of a recursive algorithm

Recursive algorithm(input)

{

If small enough number(input)

Compute and return the value

Else

Break the input into input-1,input-2… and

Soloution-1=Recursive (input -1)

Soluution -2=Recursive(input-2)

.

.

.

Return solution.

}

Example -: write a function to compute sum of number from 1 to N .

|  |  |
| --- | --- |
| Simple Function Design | Recursive Function Design |
| Int sum(int n){int s=0;for(i=0;i<n;i++)s+=ireturn (i);} | int sum(int n){int s;if(n=0)return (0);elses=n+sum(n-1);return(s);} |

**Two more programme example**

#include<stdio.h>

{

Int a,fact;

Int factorial(int x);

printf(“\n Enter any number”);

scanf(“%d”,&a);

fact=factorial(a);

printf(“\nThe facutorial value is=%d”,fact);

return(0);

}

int factorial(int x)

{

int f;

if (x==1)

return(1);

else

f=x\*factroial(x-1)

return(f);

}

Let us understand this recursive factorial function thoroughly. In the first run when the number entered through **scanf( )** is 1, let us see what action does **rec( )** take. The value of **a** (i.e. 1) is copied into **x**. Since **x** turns out to be 1 the condition **if ( x == 1 )** is satisfied and hence 1 (which indeed is the value of 1 factorial) is returned through the **return** statement.

When the number entered through **scanf( )** is 2, the **( x == 1 )** test fails, so we reach the statement,

f = x \* rec ( x - 1 ) ;

And here is where we meet recursion. How do we handle the expression **x \* rec ( x - 1 )**? We multiply **x** by **rec ( x - 1 )**. Since the current value of **x** is 2, it is same as saying that we must calculate the value (2 \* rec ( 1 )). We know that the value returned by **rec ( 1 )** is 1, so the expression reduces to (2 \* 1), or simply 2. Thus the statement,

x \* rec ( x - 1 ) ;

evaluates to 2, which is stored in the variable **f**, and is returned to **main( )**, where it is printed as

Factorial value is = 2

**Another example of recursion**.

#include <stdio.h>

int fibonacci(int i)

{

 if(i == 0) {

 return 0;

 }

 if(i == 1) {

 return 1;

 }

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

}

int main() {

 int i;

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

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

 }

 return 0;

}

Note-: from the above two example it is clear the we must have a terminateing condtion in the recursive function call and it must be designed carefully otherwise it will be a infinite call to function leeding to infinate execution of programme.

**Quick Sort (A Recursive Technique)**

It is the sorting technique developed by C.A.R Hoare, having very good average among all the sorting techniques.

Quick sort algorithm first selects a value (**pivot element** ) that is to be used as split-point from the list of given numbers.

Searching and swaping teching are followed to arrange the Elements so that, all the numbers smaller than the split-point are brought to one side of the list and the rest on the other. This operation is called splitting.

From this it is understood that the list is devided into two sub list , sublist 1 which is left to the pivot element and sublist 2 which is right from te pivot element.Thease sublist are again individually sorted using Quick sort algorithm which again execute the spilting algorithm.The process is recursively done until all elements are arranged in order

This algorithm is also know as devide and conquer algorithm.

Now we will understand it by one example.

Let’s define an array having 10 elements as shown in fig .

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Array Index | a[0] | a[1] | a[2] | a[3] | a[4] | a[5] | a[6] | a[7] | a[8] |
| Content  | 40 | 90 | 60 | 5 | 13 | 10 | 20 | 45 | 50 |

 n-1

we will select first element as pivot element and place its address in a variable.

p=a[start]; //initial value of start is 0, and we want to store the starting address of the array in variable p

We will find the biggest element than the pivot element from begining using “i” and smallest element from the last using “j” and interchange them

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Array Index | a[0] | a[1] | a[2] | a[3] | a[4] | a[5] | a[6] | a[7] | a[8] |
| Content  | 40 | 90 | 60 | 5 | 13 | 10 | 20 | 45 | 50 |

 i--------------> 🡨---------------j

Analyze the following code for the above diagram.

i=start;

j=end;

 while(a[i]<=p) /\* continues until get biggest element than pivot element\*/

  i++;

 while(a[j]>=p) /\* continues until get biggest element than pivot element\*/

  j++;

 if(i<j)

  temp=a[i],a[i]=a[j],a[j]=temp;

 /\* inter changing a[i] and a[j] \*/

a[start]=a[j];

a[j]=p;

return j;  /\* returning splitting position \*/

after executing the code the diagram will be look like followng

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Array Index | a[0] | a[1] | a[2] | a[3] | a[4] | a[5] | a[6] | a[7] | a[8] |
| Content  | 40 | 20 | 60 | 5 | 13 | 10 | 90 | 45 | 50 |

 i-------------🡪 j🡨------------------------

We follow the same procedure as long as i<j

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Array Index | a[0] | a[1] | a[2] | a[3] | a[4] | a[5] | a[6] | a[7] | a[8] |
| Content  | 40 | 20 | 60 | 5 | 13 | 10 | 90 | 45 | 50 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Array Index | a[0] | a[1] | a[2] | a[3] | a[4] | a[5] | a[6] | a[7] | a[8] |
| Content  | 40 | 20 | 10 | 5 | 13 | 60 | 90 | 45 | 50 |

As i<j is false, the current process is stopped and a[start] and a[j] are interchanged.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Array Index | a[0] | a[1] | a[2] | a[3] | a[4] | a[5] | a[6] | a[7] | a[8] |
| Content  | 40 | 20 | 10 | 5 | 13 | 60 | 90 | 45 | 50 |

 i-------------------------------------------------🡪

 j🡨------------------

we exchange 40 with 13

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Array Index | a[0] | a[1] | a[2] | a[3] | a[4] | a[5] | a[6] | a[7] | a[8] |
| Content  | 13 | 20 | 10 | 5 | 40 | 60 | 90 | 45 | 50 |

It is now clear that all elements greater than 40 is at the right hand side and all elements less than 40 is at left hand side creating sublist 1 and sublist 2

The above sub list can be recursive solove to produce the sorted list of elements and the recursive function code is as follows

void qsort(int start,int end)

{

 int s;

 if(start>=end)

   return;

 s=split(start,end);

 qsort(start,s-1);         /\* takes left sub array to split point \*/

 qsort(s+1,end);         /\* takes right sub array to split point \*/

}



The fomplete programme for quick sort .

#include<stdio.h>

int a[50];

void qsort(int,int);

int split(int,int);

int main()

{

 int n,i;

 printf("How many elements?");

 scanf("%d",&n);

 printf("Enter %d elements:\n",n);

 for(i=0;i<n;i++)

  scanf("%d",&a[i]);

 qsort(0,n-1);

 printf("The resultant array:\n");

 for(i=0;i<n;i++)

   printf("%5d",a[i]);

 return 0;

}

void qsort(int start,int end)

{

 int s;

 if(start>=end)

  return;

 s=split(start,end);

 qsort(start,s-1);

 qsort(s+1,end);

}

int split(int start,int end)

{

 int p=a[start];

 int i=start,j=end,temp;

 while(i<j)

 {

   while(a[i]<=p)

     i++;

   while(a[j]>p)

     j--;

   if(i<j)

     temp=a[i],a[i]=a[j],a[j]=temp;

 }

 a[start]=a[j];

 a[j]=p;

 return j;

}

Merge Sort-:

Like quickSort , Merge Sort is Divide and conquer algorithm..It divides input array in two halves , calls itself for the two halves and then merges the two sorted halves . The merge() function is used to merging two halves.

The merge(arr,l,m,r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub array into one .See following C impleentation.

mergeSort(arr[],l,r)

if(r>1)

m=(l+r)/2 /\*find middle point to divide the array into two halves . VARIABLE ‘m’ stores the location.\*/

mergeSort(arr,l,m);/\* call merge sort for the first half\*/

mergeSort(arr,m+1,r);/\*call merge sort for second half\*/

mergeSort(arr,l,m,r);/\* call merg sort to two sorted array in ebove two call and sort it

The process is dipicted in the figure below.

.



/\* C program for Merge Sort \*/

#include<stdlib.h>

#include<stdio.h>

// Merges two subarrays of arr[].

// First subarray is arr[l..m]

// Second subarray is arr[m+1..r]

void merge(int arr[], int l, int m, int r)

{

    int i, j, k;

    int n1 = m - l + 1;

    int n2 =  r - m;

    /\* create temp arrays \*/

    int L[n1], R[n2];

    /\* Copy data to temp arrays L[] and R[] \*/

    for (i = 0; i < n1; i++)

        L[i] = arr[l + i];

    for (j = 0; j < n2; j++)

        R[j] = arr[m + 1+ j];

    /\* Merge the temp arrays back into arr[l..r]\*/

    i = 0; // Initial index of first subarray

    j = 0; // Initial index of second subarray

    k = l; // Initial index of merged subarray

    while (i < n1 && j < n2)

    {

        if (L[i] <= R[j])

        {

            arr[k] = L[i];

            i++;

        }

        else

        {

            arr[k] = R[j];

            j++;

        }

        k++;

    }

    /\* Copy the remaining elements of L[], if there

       are any \*/

    while (i < n1)

    {

        arr[k] = L[i];

        i++;

        k++;

    }

    /\* Copy the remaining elements of R[], if there

       are any \*/

    while (j < n2)

    {

        arr[k] = R[j];

        j++;

        k++;

    }

}

/\* l is for left index and r is right index of the

   sub-array of arr to be sorted \*/

void mergeSort(int arr[], int l, int r)

{

    if (l < r)

    {

        // Same as (l+r)/2, but avoids overflow for

        // large l and h

        int m = l+(r-l)/2;

        // Sort first and second halves

        mergeSort(arr, l, m);

        mergeSort(arr, m+1, r);

        merge(arr, l, m, r);

    }

}

/\* UTILITY FUNCTIONS \*/

/\* Function to print an array \*/

void printArray(int A[], int size)

{

    int i;

    for (i=0; i < size; i++)

        printf("%d ", A[i]);

    printf("\n");

}

/\* Driver program to test above functions \*/

int main()

{

    int arr[] = {12, 11, 13, 5, 6, 7};

    int arr\_size = sizeof(arr)/sizeof(arr[0]);

    printf("Given array is \n");

    printArray(arr, arr\_size);

    mergeSort(arr, 0, arr\_size - 1);

    printf("\nSorted array is \n");

    printArray(arr, arr\_size);

    return 0;

}

**Ackermann Function.**

* It is recursive function used to compute the value
* It takes two arguments as input.
* Each argumet must be integer and non negative.

It can be defined as

Ack(m,n)={n+1 if m=0

 {Ack(m-1,1) if n=0

 {Ack(m-1,Ack(m,n-1))

The above defination can be written in simplified procedure

Ack(m,n)

{

If(m==0) /\* condition 1\*/

Ans=n+1;

Elseif(n==0) /\* condition 2\*/

Ans=Ack(m-1,1);

Else /\* condition 3\*/

Ans=Ack(m-1,Ack(m,n-1))

Let’s take Few example to understand the output.

Ack(0,5), here m=0 and n=5

So the ans=n+1 =6

Example-2

Ack(1,2), here m=1 and n=2

So following step are compute recrsively

Ack(1,2)=Ack(1-1,Ack(1,2-1))//based on condition 3

Ack(1,2)=Ack(0,Ack(1,1))=Ack(0,Ack(1-1,Ack(1,1-1)) //based on condition 3

Ack(1,2)=Ack(0,Ack(1,1))=Ack(0,Ack(0,Ack(1,0))//based on condition 2

Ack(1,2)=Ack(0,Ack(1,1))=Ack(0,Ack(0,Ack(1,0))= Ack(0,Ack(0,Ack(1-1,1)))

Ack(1,2)=Ack(0,Ack(1,1))=Ack(0,Ack(0,Ack(1,0))= Ack(0,Ack(0,Ack(0,1))) based on condition 1

Ack(0,1) =2 and we will replace the value

Ack(1,2)= Ack(0,Ack(0,2)))

Ack(0,2)=3 and we replace the value

 Ack(1,2)=Ack(0,3)

Ack(0,3)=4

So finally we get the output

Ack(1,2)=4.

We can Design The function Easily

 Int ack(int m,int n)

{

Int ans;

If(m==0)

ans=n+1;

elseif(n==0)

ans=ack(m-1,1);

else

ans=ack(m-1,ack(m,n-1));

return(ans);

}

**Write the complete programme.**