1.1. History of Computing and Computers

Definition of a Computer

Before 1935, a computer was a person who performed arithmetic calculations.
Between 1935 and 1945 the definition referred to a machine, rather than a person.
The modern machine definition is based on von Neumann's concepts: a device that accepts
input, processes data, stores data, and produces output.
We have gone from the vacuum tube to the transistor, to the microchip. Then the microchip
started talking to the modem. Now we exchange text, sound, photos and movies in a digital
environment.

Computing milestones and machine evolution:

14th C. - Abacus - an instrument for performing calculations by sliding counters along rods or in grooves.
17th C. - Slide rule - a manual device used for calculation that consists in its simple form
of a ruler and a movable middle piece which are graduated with similar logarithmic scales.
1642 – Pascaline -- a mechanical calculator built by Blaise Pascal, a 17th century mathematician, for whom the Pascal computer programming language was named .
1804 - Jacquard loom - a loom programmed with punched cards invented by Joseph Marie Jacquard
1850 - Difference Engine , Analytical Engine - Charles Babbage and Ada Byron. Babbage's description, in 1837, of the Analytical Engine, a hand cranked, mechanical digital computer anticipated virtually every aspect of present-day computers. It wasn't until
over a 100 years later that another all purpose computer was conceived.
1939 -1942 - Atanasoff Berry Computer - built at Iowa State by Prof. John V. Atanasof and graduate student Cliford Berry. Represented several "firsts" in computing, including a
binary system of arithmetic, parallel processing, regenerative memory, separation of memory and computing functions, and more. Weighed 750 lbs. and had a memory storage
of 3,000 bits (0.4K). Recorded numbers by scorching marks into cards as it worked through
a problem.
1940s - Colossus - a vacuum tube computing machine which broke Hitler's codes during
WW II. It was instrumental in helping Alan Turing break the German's codes during WW II to
turn the tide of the war. In the summer of 1939, a small group of scholars became code breakers, working at Bletchley Part in England. This group of pioneering code breakers helped shorten the war and changed the course of history.
1946 - ENIAC - World's first electronic, large scale, general-purpose computer, built by

Mauchly and Eckert, and activated at the University of Pennsylvania in 1946. ENIAC recreated on a modern computer chip. The ENIAC is a 30 ton machine that measured 50 x
30 feet. It contained 19,000 vacuum tubes, 6000 switches, and could add 5,000 numbers in
a second, a remarkable accomplishment at the time. A re-programmable machine, the ENIAC performed initial calculations for the H-bomb. It was also used to prepare artillery shell trajectory tables and perform other military and scientific calculations.
Since there was no software to reprogram the computer, people had to rewire it to get it to
perform diferent functions. The human programmers had to read wiring diagrams and know what each switch did. J. Presper Eckert, Jr. and John W. Mauchly drew on Alansof's
work to create the ENIAC, the Electronic Numerical Integrator and Computer.
1951-1959 - vacuum tube based technology. Vacuum Tubes are electronic devices, consisting of a glass or steel vacuum envelope and two or more electrodes between which
electrons can move freely. First commercial computers used vacuum tubes: Univac, IBM
701.
1950s -1960s - UNIVAC - "punch card technology" The first commercially successful computer, introduced in 1951 by Remington Rand. Over 40 systems were sold. Its memory
was made of mercury filled acoustic delay lines that held 1,000 12 digit numbers. It used magnetic tapes that stored 1MB of data at a density of 128 cpi. UNIVAC became synonymous with computer (for a while).
1960-1968 - transistor based technology. The transistor, invented in 1948, by Dr. John
Bardeen, Dr. Walter Brattain, and Dr. William Shockley . It almost completely replaced the
vacuum tube because of its reduced cost, weight, and power consumption and its higher
reliability. The transistor is made to alter its state from a starting condition of conductivity (switched 'on', full current flow) to a final condition of insulation (switched 'of', no current flow).
1969 - The Internet, originally the ARPAnet (Advanced Research Projects Agency network), began as a military computer network.
1969-1977 - integrated circuits (IC) based technology. The first integrated circuit was demonstrated by Texas Instruments inventor, Jack Kilby, in 1958. It was 7/16" wide and
contained two transistors. Examples of early integrated circuit technology: Intel 4004, Dec
pdp 8, CRAY 1. Now circuits may contain hundreds of thousands of transistors on a small
piece of material, which revolutionized computing.

1976 - CRAY 1 - The world's first electronic digital computer, developed in 1946. A 75MHz,
64-bit machine with a peak speed of 160 megaflops, (one million floating point operations
per second) the world's fastest processor at that time.
1976 - Apples/MACs - The Apple was designed by Steve Wozniak and Steve Jobs. Apple
was the first to have a "window" type graphical interface and the computer mouse. Like modern computers, early Apples had a peripheral keyboard and mouse, and had a floppy
drive that held 3.5" disks. The Macintosh replaced the Apple.
1978 to 1986 - large scale integration (LSI); Alto - early workstation with mouse and Apple. The PC and clone market begins to expand. This begins first mass market of desktop computers.
1990 - Tim Berners-Lee invented the networked hypertext system called the World Wide
Web.
1992 - Bill Gates' Microsoft Corp. released Windows 3.1, an operating system that made IBM and IBM-compatible PCs more user-friendly by integrating a graphical user interface into the software. In replacing the old Windows command-line system, however,
Microsoft created a program similar to the Macintosh operating system. Apple sued for copyright infringement, but Microsoft prevailed.
Windows 3.1 went to Win 95, then Win 98, Windows XP, Vista, Windows 7/8. There are other
OSs, of course, but Windows is the dominant OS today. MACs, by Apple, still have a faithful
following. Linux has a faithful following.
1996 - Personal Digital Assistants (such as the Palm Pilot became available to consumers. They can do numeric calculations, play games and music and download information from the Internet.

Pioneer computer scientists

Charles Babbage (1792-1871) - Diference Engine, Analytical Engine. Ada Byron, daughter of the poet, Lord Byron, worked with him. His description, in 1837, of the Analytical Engine, a mechanical digital computer anticipated virtually every aspect of present-day computers.
Alan Turing -- 1912-1954. British Code breaker. Worked on the Colossus (code breaking
machine, precursor to the computer) and the ACE (Automatic Computing Engine). Noted for
many brilliant ideas, Turing is perhaps best remembered for the concepts of the Turing Test
for Artificial Intelligence and the Turing Machine, an abstract model for modeling computer
operations. The Turing Test is the "acid test" of true artificial intelligence, as defined by the

English scientist Alan Turing. In the 1940s, he said "a machine has artificial intelligence when there is no discernible diference between the conversation generated by the machine and that of an intelligent person." Turing was instrumental in breaking the German
enigma code during WWII with his Bombe computing machine. The Enigma is a machine
used by the Germans to create encrypted messages.

J. von Neumann -- (1903-1957). A child prodigy in mathematics, authored landmark paper explaining how programs could be stored as data. (Unlike ENIAC, which had to be rewired
to be re-programmed.). Virtually all computers today, from toys to supercomputers costing millions of dollars, are variations on the computer architecture that John von Neumann created on the foundation of the work of Alan Turing's work in the 1940s. It included three components used by most computers today: a CPU; a slow-to-access storage area, like a hard drive; and secondary fast-access memory (RAM). The machine
stored instructions as binary values (creating the stored program concept) and executed instructions sequentially - the processor fetched instructions one at a time and processed
them. The instruction is analyzed, data is processed, the next instruction is analyzed, etc.

Today "***von Neumann architecture***" often refers to the sequential nature of computers based on this model.

John V. Atanasoff -- (1904 - 1995) - one of the contenders, along with Konrad Zuse and
H. Edward Roberts and others, as the inventor of the first computer. The limited-function vacuum-tube device had limited capabilities and did not have a central. It was not programmable, but could solve diferential equations using binary arithmetic.

J. Presper Eckert, Jr. and John W. Mauchly completed the first programmed general purpose electronic digital computer in 1946. They drew on Alansof's work to create the ENIAC, the Electronic Numerical Integrator and Computer. In 1973 a patent lawsuit resulted
in John V. Atanasof's being legally declared as the inventor. Though Atanasof got legal status for his achievement, many historians still give credit to J. Presper Eckert, Jr., and John
W. Mauchly the founding fathers of the modern computer. Eckert and Mauchly formed the
first computer company in 1946. Eckert received 87 patents.

They introduced the first modern binary computer with the Binary Automatic Computer (BINAC), which stored information on magnetic tape rather than punched cards. Their UNIVAC I ,was built for the U.S. Census Bureau.

Konrad Zuse-- (1910-1995) a German who, during WW II, designed mechanical and electromechanical computers. Zuse's Z1, his contender for the first freely programmable computer, contained all the basic components of a modern computer (control unit, memory, micro sequences, etc.). Zuse, because of the scarcity of material during WW II,

used discarded video film as punch cards. Like a modern computer, it was adaptable for diferent purposes and used on/of switch relays, a binary system of 1s and 0s (on = 1, of = 0). Completed in 1938, it was destroyed in the bombardment of Berlin in WW II, along with the construction plans. In 1986, Zuse reconstructed the Z1.

Early Counting

The start of the modern science that we call "Computer Science" can be traced back to a
long ago age where man still dwell-ed in caves or in the forest, and lived in groups for protection and survival from the harsher elements on the Earth. Many of these groups possessed some primitive form of animistic religion; they worshiped the sun, the moon, the
trees, or sacred animals. Within the tribal group was one individual to whom fell the responsibility for the tribe's spiritual welfare. It was he or she who decided when to hold both the secret and public religious ceremonies, and interceded with the spirits on behalf of
the tribe. In order to correctly hold the ceremonies to ensure good harvest in the fall and fertility in the spring, the shamans needed to be able to count the days or to track the seasons. **From the shamanistic tradition, man developed the first primitive counting mechanisms -- counting notches on sticks or marks on walls.**
From the caves and the forests, man slowly evolved and built structures such as Stonehenge. Stonehenge, which lies 13km north of Salisbury, England, is believed to have
been an ancient form of calendar designed to capture the light from the summer solstice in
a specific fashion. The solstices have long been special days for various religious groups
and cults. Archaeologists and anthropologists today are not quite certain how the structure,
believed to have been built about 2800 B.C., came to be erected since the technology required to join together the giant stones and raise them upright seems to be beyond the
technological level of the Britons at the time. It is widely believed that the enormous edifice
of stone may have been erected by the Druids. Regardless of the identity of the builders, it
remains today a monument to **man's intense desire to count and to track the occurrences of the physical world** around him.

Abacus

Meanwhile in Asia, the Chinese were becoming very involved in commerce with the Japanese, Indians, and Koreans. Businessmen needed a way to tally accounts and bills.
Somehow, out of this need, the abacus was born. The abacus is the first true precursor to
the adding machines and computers which would follow. It worked somewhat like this:
The value assigned to each pebble (or bead, shell, or stick) is determined not by its shape but by its position: one pebble on a particular line or one

bead on a particular wire has the value of 1; two together have the value of 2. A pebble on the next line, however, might have the value of 10, and a pebble on the third line would have the value of 100. Therefore, three properly placed pebbles--two with values of 1 and one with the value of 10--could signify 12, and the addition of a fourth pebble with the value of 100 could signify 112, using a place-value notation system with multiples of 10.

Thus, the abacus works on the principle of place-value notation: the location of the bead determines its value. In this way, relatively few beads are required to depict large numbers.

The beads are counted, or given numerical values, by shifting them in one direction. The values are erased (freeing the counters for reuse) by shifting the beads in the other direction. An abacus is really a memory aid for the user making mental calculations, as opposed to the true mechanical calculating machines which were still to come.

Forefathers of Computing

For over a thousand years after the Chinese invented the abacus, not much progress was made to automate counting and mathematics. The Greeks came up with numerous mathematical formulas and theorems, but all of the newly discovered math had to be worked out by hand. A mathematician was often a person who sat in the back room of an establishment with several others and they worked on the same problem. The redundant personnel working on the same problem were there to ensure the correctness of the answer. It could take weeks or months of laborious work by hand to verify the correctness of a proposed theorem. Most of the tables of integrals, logarithms, and trigonometric values were worked out this way, their accuracy unchecked until machines could generate the tables in far less time and with more accuracy than a team of humans could ever hope to achieve.

Blaise Pascal

Blaise Pascal, noted mathematician, thinker, and scientist, built the first mechanical adding machine in 1642 based on a design described by Hero of Alexandria (2AD) to add up the distance a carriage traveled. The basic principle of his calculator is still used today in water meters and modern-day odometers. Instead of having a carriage wheel turn the gear, he made each ten-teeth wheel accessible to be turned directly by a person's hand (later inventors added keys and a crank), with the result that when the wheels were turned in the

proper sequences, a series of numbers was entered and a cumulative sum was obtained.
The gear train supplied a mechanical answer equal to the answer that is obtained by using
arithmetic.
This first mechanical calculator, called the **Pascaline**, had several disadvantages. Although
it did ofer a substantial improvement over manual calculations, only Pascal himself could
repair the device and it cost more than the people it replaced! In addition, the first signs of
techno phobia emerged with mathematicians fearing the loss of their jobs due to progress.

Charles Babbage

While Tomas of Colmar was developing the first successful commercial calculator, Charles
Babbage realized as early as 1812 that many long computations consisted of operations
that were regularly repeated. He theorized that it must be possible to design a calculating
machine which could do these operations automatically. He produced a prototype of this
"diference engine" by 1822 and with the help of the British government started work on
the full machine in 1823. It was intended to be steam-powered; fully automatic, even to the
printing of the resulting tables; and commanded by a fixed instruction program.
In 1833, Babbage ceased working on the diference engine because he had a better idea.
His new idea was to build an "analytical engine." The analytical engine was a real parallel
decimal computer which would operate on words of 50 decimals and was able to store
1000 such numbers. The machine would include a number of built-in operations such as
conditional control, which allowed the instructions for the machine to be executed in a
specific order rather than in numerical order. The instructions for the machine were to be
stored on punched cards, similar to those used on a Jacquard loom.

Herman Hollerith

A step toward automated computation was the introduction of punched cards, which were
first successfully used in connection with computing in 1890 by Herman Hollerith working
for the U.S. Census Bureau. He developed a device which could automatically read census
information which had been punched onto card. Surprisingly, he did not get the idea from
the work of Babbage, but rather from watching a train conductor punch tickets. As a result

of his invention, reading errors were consequently greatly reduced, work flow was increased, and, more important, stacks of punched cards could be used as an accessible
memory store of almost unlimited capacity; furthermore, diferent problems could be stored on diferent batches of cards and worked on as needed. Hollerith's tabulator became
so successful that he started his own firm to market the device; this company eventually became International Business Machines (IBM).

Konrad Zuse

Hollerith's machine though had limitations. It was strictly limited to tabulation. The punched cards could not be used to direct more complex computations. In 1941, Konrad
Zuse, a German who had developed a number of calculating machines, released the first
programmable computer designed to solve complex engineering equations. The machine,
called the Z3, was controlled by perforated strips of discarded movie film. As well as being
controllable by these celluloid strips, it was also the first machine to work on the binary system, as opposed to the more familiar decimal system.

Binary Representation

The binary system is composed of 0s and 1s. A punch card with its two states--a hole or no hole-- was admirably suited to representing things in binary. If a hole was read by the card reader, it was considered to be a 1. If no hole was present in a column, a zero was appended to the current number. The total number of possible numbers can be calculated by putting 2 to the power of the number of bits in the binary number. A bit is simply a single occurrence of a binary number--a 0 or a 1. Thus, if you had a possible binary number of 6 bits, 64 diferent numbers could be generated. (2^(n-1)) .

Binary representation was going to prove important in the future design of computers which took advantage of a multitude of two-state devices such card readers, electric circuits which could be on or of, and vacuum tubes.

Howard Aiken

By the late 1930s punched-card machine techniques had become so well established and
reliable that Howard Aiken, in collaboration with engineers at IBM, undertook construction
of a large automatic digital computer based on standard IBM electromechanical parts. Aiken's machine, called the Harvard Mark I, handled 23-decimal-place numbers (words) and
could perform all four arithmetic operations; moreover, it had special built-in programs, or
subroutines, to handle logarithms and trigonometric functions. The Mark I was originally controlled from pre-punched paper tape without provision for reversal, so that automatic "transfer of control" instructions could not be programmed. Output was by card punch and

electric typewriter. Although the Mark I used IBM rotating counter wheels as key components in addition to electromagnetic relays, the machine was classified as a relay computer. It was slow, requiring 3 to 5 seconds for a multiplication, but it was fully automatic and could complete long computations without human intervention. The Harvard
Mark I was the first of a series of computers designed and built under Aiken's direction.

Alan Turing

The Turing machine, reads in the symbols from the tape one at a time. What we would like
the machine to do is to give us an output of 1 anytime it has read at least 3 ones in a row
of of the tape. When there are not at least three ones, then it should output a 0. The reading and outputting can go on infinitely.
Turing's purpose was not to invent a computer, but rather to describe problems which are
logically possible to solve. His hypothetical machine, however, foreshadowed certain characteristics of modern computers that would follow. For example, the endless tape could
be seen as a form of general purpose internal memory for the machine in that the machine
was able to read, write, and erase it--just like modern day RAM.

John W. Mauchly and J. Presper Eckert

Back in America, with the success of Aiken's Harvard Mark-I as the first major American development in the computing race, work was proceeding on the next great breakthrough
by the Americans. Their second contribution was the development of the giant **ENIAC (Electrical Numerical Integrator and Computer)** machine by John W. Mauchly and J. Presper Eckert at the University of Pennsylvania. ENIAC used a word of 10 decimal digits
instead of binary ones like previous automated calculators/computers. ENIAC also was the
first machine to use more than 2,000 vacuum tubes, using nearly 18,000 vacuum tubes. Storage of all those vacuum tubes and the machinery required to keep the cool took up over 167 square meters (1800 square feet) of floor space. Nonetheless, it had punchedcard
input and output and arithmetically had 1 multiplier, 1 divider-square rooter, and 20 adders employing decimal "ring counters," which served as adders and also as quickaccess
(0.0002 seconds) read-write register storage.
The executable instructions composing a program were embodied in the separate units of
ENIAC, which were plugged together to form a route through the machine for the flow of computations. These connections had to be redone for each diferent problem, together with presetting function tables and switches. This **"wire-your-own" instruction** technique

was inconvenient, and only with some license could ENIAC be considered programmable; it was, however, efficient in handling the particular programs for which it had been designed.

ENIAC is generally acknowledged to be the first successful high-speed electronic digital computer (EDC) and was productively used from 1946 to 1955. As controversy developed in 1971, however, over the patentability of ENIAC's basic digital concepts, the claim being made that another U.S. physicist, John V. Atanasof, had already used the same ideas in a simpler vacuum-tube device he built in the 1930s while at Iowa State College. In 1973, the court found in favor of the company using Atanasof claim and Atanasof received the acclaim he rightly deserved.

John von Neumann

In 1945, mathematician John von Neumann undertook a study of computation that demonstrated that a computer could have a simple, fixed structure, yet be able to execute any kind of computation given properly programmed control without the need for hardware modification. Von Neumann contributed a new understanding of how practical fast computers should be organized and built; these ideas, often referred to as the storedprogram technique, became fundamental for future generations of high-speed digital computers and were universally adopted. The primary advance was the provision of a special type of machine instruction called conditional control transfer--which permitted the program sequence to be interrupted and re-initiated at any point, similar to the system suggested by Babbage for his analytical engine--and by storing all instruction programs together with data in the same memory unit, so that, when desired, instructions could be arithmetically modified in the same way as data. Thus, data was the same as program. As a result of these techniques and several others, computing and programming became faster, more flexible, and more efficient, with the instructions in subroutines performing far more computational work. Frequently used subroutines did not have to be reprogrammed for each new problem but could be kept intact in "libraries" and read into memory when needed. Thus, much of a given program could be assembled from the subroutine library.

The all-purpose computer memory became the assembly place in which parts of a long computation were stored, worked on piece wise, and assembled to form the final results. As soon as the advantages of these techniques became clear, the techniques became standard practice.

Jack St. Clair Kilby

In 1958, Jack St. Clair Kilby of Texas Instruments manufactured the first integrated circuit or chip. A chip is really a collection of tiny transistors which are connected together when the transistor is manufactured. Thus, the need for soldering together large numbers of transistors was practically nullified; now only connections were needed to other electronic components. In addition to saving space, the speed of the machine was now increased since there was a diminished distance that the electrons had to follow.

For More Detailed Information visit [*http://www.computerhistory.org/timeline/*]

1.2. History of Programming Languages

Ever since the invention of Charles Babbage's diference engine in 1822, computers have required a means of instructing them to perform a specific task. This means is known as a programming language. Computer languages were first composed of a series of steps to wire a particular program; these morphed into a series of steps keyed into the computer and then executed; later these languages acquired advanced features such as logical branching and object orientation. The computer languages of the last fifty years have come in two stages, the first major languages and the second major languages, which are in use today.

In the beginning, Charles Babbage's diference engine could only be made to execute tasks by changing the gears which executed the calculations. Thus, the earliest form of a computer language was physical motion. Eventually, physical motion was replaced by electrical signals when the US Government built the ENIAC in 1942. It followed many of the same principles of Babbage's engine and hence, could only be "programmed" by presetting switches and rewiring the entire system for each new "program" or calculation.

This process proved to be very tedious.

In 1945, John Von Neumann was working at the Institute for Advanced Study. He developed two important concepts that directly afected the path of computer programming languages.

The first was known as "shared-program technique". This technique stated that the actual computer hardware should be simple and not need to be hand-wired for each program. Instead, complex instructions should be used to control the simple hardware, allowing it to be reprogrammed much faster.

The second concept was also extremely important to the development of programming

languages. Von Neumann called it “conditional control transfer”. This idea gave rise to the notion of subroutines, or small blocks of code that could be jumped to in any order, instead of a single set of chronologically ordered steps for the computer to take. The second part of the idea stated that computer code should be able to branch based on logical statements such as IF (expression) THEN, and looped such as with a FOR statement. “Conditional control transfer” gave rise to the idea of “libraries,” which are blocks of code that can be reused over and over.

Short Code

In 1949, a few years after Von Neumann’s work, the language Short Code appeared. It was the first computer language for electronic devices and it required the programmer to change its statements into 0’s and 1’s by hand. Still, it was the first step towards the complex languages of today. In 1951, Grace Hopper wrote the first compiler, A-0. A compiler is a program that turns the language’s statements into 0’s and 1’s for the computer to understand. This lead to faster programming, as the programmer no longer had to do the work by hand.

FORTRAN

In 1957, the first of the major languages appeared in the form of FORTRAN. Its name stands for FORmula TRANslating system. The language was designed at IBM for scientific computing. The components were very simple, and provided the programmer with lowlevel access to the computers innards. Today, this language would be considered restrictive as it only included IF, DO, and GOTO statements, but at the time, these commands were a big step forward. The basic types of data in use today got their start in FORTRAN, these included logical variables (TRUE or FALSE), and integer, real, and double-precision numbers.

COBOL

Though FORTAN was good at handling numbers, it was not so good at handling input and output, which mattered most to business computing. Business computing started to take of in 1959, and because of this, COBOL was developed. It was designed from the ground up as the language for businessmen. Its only data types were numbers and strings of text.

It also allowed for these to be grouped into arrays and records, so that data could be tracked and organized better. It is interesting to note that a COBOL program is built in a way similar to an essay, with four or five major sections that build into an elegant whole. COBOL statements also have a very English-like grammar, making it quite easy to learn. All of these features were designed to make it easier for the average business to learn and

adopt it.

LISP

In 1958, John McCarthy of MIT created the LISt Processing (or LISP) language. It was designed for Artificial Intelligence (AI) research. Because it was designed for a specialized field, the original release of LISP had a unique syntax: essentially none. Programmers wrote code in parse trees, which are usually a compiler-generated intermediary between higher syntax (such as in C or Java) and lower-level code. Another obvious diference between this language (in original form) and other languages is that the basic and only type of data is the list; in the mid-1960's, LISP acquired other data types. A LISP list is denoted by a sequence of items enclosed by parentheses. LISP programs themselves are written as a set of lists, so that LISP has the unique ability to modify itself, and hence grow on its own.

ALGOL

The Algol language was created by a committee for scientific use in 1958. It's major contribution is being the root of the tree that has led to such languages as Pascal, C, C++, and Java. It was also the first language with a formal grammar, known as Backus-Naar Form or BNF. Though Algol implemented some novel concepts, such as recursive calling of functions, the next version of the language, Algol 68, became bloated and difficult to use.

This lead to the adoption of smaller and more compact languages, such as Pascal.

PASCAL

Pascal was begun in 1968 by Niklaus Wirth. Its development was mainly out of necessity for a good teaching tool. In the beginning, the language designers had no hopes for it to enjoy widespread adoption. Instead, they concentrated on developing good tools for teaching such as a debugger and editing system and support for common early microprocessor machines which were in use in teaching institutions.

Pascal was designed in a very orderly approach, it combined many of the best features of the languages in use at the time, COBOL, FORTRAN, and ALGOL. The combination of features, input/output and solid mathematical features, made it a highly successful language. Pascal also improved the "pointer" data type, a very powerful feature of any language that implements it. It also added a CASE statement, that allowed instructions to to branch like a tree.

Pascal also helped the development of dynamic variables, which could be created while a program was being run, through the NEW and DISPOSE commands. However, Pascal did not

implement dynamic arrays, or groups of variables, which proved to be needed and led to its downfall. Wirth later created a successor to Pascal, Modula-2, but by the time it appeared, C was gaining popularity and users at a rapid pace.

C Language

C was developed in 1972 by Dennis Ritchie while working at Bell Labs in New Jersey. The transition in usage from the first major languages to the major languages of today occurred with the transition between Pascal and C. Its direct ancestors are B and BCPL, but its similarities to Pascal are quite obvious. All of the features of Pascal, including the new ones such as the CASE statement are available in C. C uses pointers extensively and was built to be fast and powerful at the expense of being hard to read. But because it fixed most of the mistakes Pascal had, it won over former-Pascal users quite rapidly.

Ritchie developed C for the new Unix system being created at the same time. Because of this, C and Unix go hand in hand. Unix gives C such advanced features as dynamic variables, multitasking, interrupt handling, forking, and strong, low-level, input-output. Because of this, C is very commonly used to program operating systems such as Unix, Windows, the MacOS, and Linux.

C++ Language

In the late 1970's and early 1980's, a new programming method was being developed. It was known as Object Oriented Programming, or OOP. Objects are pieces of data that can be packaged and manipulated by the programmer. Bjarne Stroustroup liked this method and developed extensions to C known as "C With Classes." This set of extensions developed into the full-featured language C++, which was released in 1983.

C++ was designed to organize the raw power of C using OOP, but maintain the speed of C and be able to run on many diferent types of computers. C++ is most often used in simulations, such as games. C++ provides an elegant way to track and manipulate hundreds of instances of people in elevators, or armies filled with diferent types of soldiers.

Other Languages

Besides these, now we have better languages such as Java, Perl, Ruby and Python.

1.3. Block Diagram of a Computer

A computer can process data, pictures, sound and graphics. They can solve highly complicated problems quickly and accurately.

Input Unit:

Computers need to receive data and instruction in order to solve any problem. Therefore we need to input the data and instructions into the computers. The input unit consists of one or more input devices. Keyboard is the one of the most commonly used input device.

Other commonly used input devices are the mouse, floppy disk drive, magnetic tape, etc.

All the input devices perform the following functions:

- Accept the data and instructions from the outside world.
- Convert it to a form that the computer can understand.
- Supply the converted data to the computer system for further processing.

Storage Unit:

The storage unit of the computer holds data and instructions that are entered through the input unit, before they are processed. It preserves the intermediate and final results before these are sent to the output devices. It also saves the data for the later use. The various storage devices of a computer system are divided into two categories.

1. **Primary Storage:** Stores and provides very fast. This memory is generally used to hold the program being currently executed in the computer, the data being received from the input unit, the intermediate and final results of the program. The primary memory is temporary in nature. The data is lost, when the computer is switched of. In order to store the data permanently, the data has to be transferred to the secondary memory. The cost of the primary storage is more compared to the secondary storage. Therefore most computers have limited primary storage capacity.
2. **Secondary Storage:** Secondary storage is used like an archive. It stores several programs, documents, data bases etc. The programs that you run on the computer are first transferred to the primary memory before it is actually run. Whenever the results are saved, again they get stored in the secondary memory. The secondary memory is slower and cheaper than the primary memory. Some of the commonly used secondary memory devices are Hard disk, CD, etc.

Output Unit:

The output unit of a computer provides the information and results of a computation to outside world. Printers, Visual Display Unit (VDU) are the commonly used output devices.

Other commonly used output devices are floppy disk drive, hard disk drive, and magnetic tape drive.

Arithmetic and Logical Unit:

All calculations are performed in the Arithmetic Logic Unit (ALU) of the computer. It also does comparison and takes decision. The ALU can perform basic operations such as addition, subtraction, multiplication, division, etc and does logic operations viz, >, <, =, etc. Whenever calculations are required, the control unit transfers the data from storage unit to ALU once the computations are done, the results are transferred to the storage unit

by the control unit and then it is send to the output unit for displaying results.

Control Unit:

It controls all other units in the computer. The control unit instructs the input unit, where to store the data after receiving it from the user. It controls the flow of data and instructions from the storage unit to ALU. It also controls the flow of results from the ALU to the storage unit. The control unit is generally referred as the central nervous system of the computer that control and synchronizes its working.

Central Processing Unit:

The Control Unit and ALU of the computer are together known as the Central Processing Unit (CPU). The CPU is like brain performs the following functions:

- It performs all calculations.
- It takes all decisions.
- It controls all units of the computer.

Other internal Units

Registers: It is a special temporary storage location within the CPU. Registers quickly, accept, store and transfer data and instructions that are being used immediately (main memory hold data that will be used shortly, secondary storage holds data that will be used later). To execute an instruction, the control unit of the CPU retrieves it from main memory and places it onto a register. The typical operations that take place in the processing of instruction are part of the instruction cycle or execution cycle. The instruction cycle refers to the retrieval of the instruction from main memory and its sub sequence at decoding. The process of alerting the circuits in CPU to perform the specified operation. The time it takes to go through the instruction cycle is referred to as instruction time.

Bus: The term Bus refers to an electrical pathway through which bits are transmitted between the various computer components. Depending on the design of the system, several types of buses may be present. The most important one is the data bus, which carries the data through out the central processing unit. The wider the data bus, the more data it can carry at one time and thus the greater the processing speed of the computer. Ex: Intel 8088 processor uses a data bus of 8 bits wide. Some super computers contain buses that are 128 bits wide.

Random Access Memory (RAM): The main memory of the computer is called as RAM.

The name derives from the fact that data can be stored in and retrieved at random, from anywhere in the electronic main memory chips in approximately the same amount of time, no matter where the data is. Main memory is in an electronic or volatile state. When the

computer is of, main memory is empty, when it is on it is capable of receiving and holding
a copy of the software instructions, and data necessary for processing.
Because the main memory is a volatile form of storage that depends on electric power can
go of during processing, users save their work frequently on to non volatile secondary storage devices such as diskettes or hard disk.
The main memory is used for the following purposes:
1. Storage of the copy of the main software program that controls the general operation of the computer. This copy is loaded on to the main memory when the computer is turned on, and it stays there as long as the computer is on.
2. Temporary storage of a copy of application program instruction, to be received by CPU for interpretation and processing or execution.
3. Temporary storage of data that has been input from the key board, until instructions call for the data to be transferred in to CPU for processing.
4. Temporary storage of data, which is required for further processing or transferred as output to output devices such as screen, a printer, a disk storage device.

1.4. Generations of Computer

The history of computer development is often referred to in reference to the diferent generations of computing devices. Each of the five generations of computers is characterized by a major technological development that fundamentally changed the way
computers operate, resulting in increasingly smaller, cheaper, more powerful and more efficient and reliable computing devices. The five generations of computers and the technology developments that have led to the current devices that we use today. Our journey starts in 1940 with vacuum tube circuitry and goes to the present day -- and beyond -- with artificial intelligence.

First Generation (1940-1956) - Vacuum Tubes

The first computers used vacuum tubes for circuitry and magnetic drums for memory, and
were often enormous, taking up entire rooms. They were very expensive to operate and in
addition to using a great deal of electricity, generated a lot of heat, which was often the cause of malfunctions.
First generation computers relied on machine language, the lowest-level programming language understood by computers, to perform operations, and they could only solve one
problem at a time. Input was based on punched cards and paper tape, and output was displayed on printouts.
The UNIVAC and ENIAC computers are examples of first-generation computing devices. The
UNIVAC was the first commercial computer delivered to a business client, the U.S. Census
Bureau in 1951.

Second Generation (1956-1963) - Transistors

Transistors replaced vacuum tubes and ushered in the second generation of computers. The transistor was invented in 1947 but did not see widespread use in computers until the late 1950s. The transistor was far superior to the vacuum tube, allowing computers to become smaller, faster, cheaper, more energy-efficient and more reliable than their firstgeneration predecessors. Though the transistor still generated a great deal of heat that subjected the computer to damage, it was a vast improvement over the vacuum tube. Second-generation computers still relied on punched cards for input and printouts for output.

Second-generation computers moved from cryptic binary machine language to symbolic, or assembly, languages, which allowed programmers to specify instructions in words. Highlevel programming languages were also being developed at this time, such as early versions of COBOL and FORTRAN. These were also the first computers that stored their instructions in their memory, which moved from a magnetic drum to magnetic core technology.

The first computers of this generation were developed for the atomic energy industry.

Third Generation (1964-1971) - Integrated Circuits

The development of the integrated circuit was the hallmark of the third generation of computers. Transistors were miniaturized and placed on silicon chips, called semiconductors, which drastically increased the speed and efficiency of computers. Instead of punched cards and printouts, users interacted with third generation computers through keyboards and monitors and interfaced with an operating system, which allowed the device to run many diferent applications at one time with a central program that monitored the memory. Computers for the first time became accessible to a mass audience because they were smaller and cheaper than their predecessors.

Fourth Generation (1971-Present) - Microprocessors

The microprocessor brought the fourth generation of computers, as thousands of integrated circuits were built onto a single silicon chip. What in the first generation filled an entire room could now fit in the palm of the hand. The Intel 4004 chip, developed in 1971, located all the components of the computer—from the central processing unit and memory to input/output controls—on a single chip.

In 1981 IBM introduced its first computer for the home user, and in 1984 Apple introduced the Macintosh. Microprocessors also moved out of the realm of desktop computers and into many areas of life as more and more everyday products began to use microprocessors.

As these small computers became more powerful, they could be linked together to form networks, which eventually led to the development of the Internet. Fourth generation computers also saw the development of GUIs, the mouse and hand held devices.

Fifth Generation (Present and Beyond) - Artificial Intelligence

Fifth generation computing devices, based on artificial intelligence, are still in development,
though there are some applications, such as voice recognition, that are being used today.
The use of parallel processing and superconductors is helping to make artificial intelligence
a reality. Quantum computation and molecular and nanotechnology will radically change the face of computers in years to come. The goal of fifth-generation computing is to develop devices that respond to natural language input and are capable of learning and self-organization.

1.5. Types of Computer

Classify by principle of operation:

- Analog Computer
- Digital Computer
- Hybrid Computer

Classify by size:

- Micro Computer
- Mini Computer
- Mainframe Computer
- Super Computer

Classify by function:

- Servers
- Workstations
- Information Appliances
- Embedded Computers

Analog Computer

Analog Computer is a computing device that works on continuous range of values. The results given by the analog computers will only be approximate since they deal with quantities that vary continuously. It generally deals with physical variables such as voltage,
pressure, temperature, speed, etc.

Digital Computer

On the other hand a digital computer operates on digital data such as numbers. It uses binary number system in which there are only two digits 0 and 1. Each one is called a bit.
The digital computer is designed using digital circuits in which there are two levels for an
input or output signal. These two levels are known as logic 0 and logic 1. Digital Computers
can give more accurate and faster results.

Digital computer is well suited for solving complex problems in engineering and technology.
Hence digital computers have an increasing use in the field of design, research and data
processing.
Based on the purpose, Digital computers can be further classified as,
General Purpose Computers
Special Purpose Computers
Special purpose computer is one that is built for a specific application. General purpose computers are used for any type of applications. They can store diferent programs and do
the jobs as per the instructions specified on those programs. Most of the computers that we
see today, are general purpose computers.

Hybrid Computer

A hybrid computer combines the desirable features of analog and digital computers. It is mostly used for automatic operations of complicated physical processes and machines. Now-a-days analog-to-digital and digital-to-analog converters are used for transforming the
data into suitable form for either type of computation.
For example, in hospital's ICU, analog devices might measure the patients temperature, blood pressure and other vital signs. These measurements which are in analog might then
be converted into numbers and supplied to digital components in the system. These components are used to monitor the patient's vital sign and send signals if any abnormal
readings are detected. Hybrid computers are mainly used for specialized tasks.

Super Computers

Super computers are the best in terms of processing capacity and also the most expensive
ones. These computers can process billions of instructions per second. Normally, they will
be used for applications which require intensive numerical computations such as stock analysis, weather forecasting etc.
Other uses of supercomputers are scientific simulations, (animated) graphics, fluid dynamic
calculations, nuclear energy research, electronic design, and analysis of geological data (e.g. in petrochemical prospecting). Perhaps the best known super computer manufacturer
is Cray Research. Some of the "traditional" companies which produce super computers are
Cray, IBM and Hewlett-Packard. Currently, China's Tianhe-2 is the fastest super computer in
the world.

Mainframe Computers

Mainframe computers can also process data at very high speeds i.e., hundreds of million
instructions per second and they are also quite expensive. Normally, they are used in banking, airlines and railways etc for their applications.

Mini Computers

Mini computers are lower to mainframe computers in terms of speed and storage capacity.
They are also less expensive than mainframe computers. Some of the features of mainframes will not be available in mini computers. Hence, their performance also will be
less than that of mainframes.

Micro Computers

The invention of microprocessor (single chip CPU) gave birth to the much cheaper micro
computers. They are further classified into :

- Desktop Computers
- Laptop Computers
- Handheld Computers (PDAs, smart phones, tablets, etc)

More Info: [http://www.cs.cmu.edu/~fgandon/lecture/uk1999/computers_types/]

1.6. Types of Software

Various types of computer software are used to simplify the operations and applications of
computer programs. Computer software enables the computer system to perform in accordance with the given tasks.
Basically, there are only 2 main types of Software.

- System Software
- Application Software

Besides these, software can also be categorized depending on the nature of use. There are
2 categories based on that. They are:

- Ready-made (of-the-shelf) Software
- Tailored (custom) Software

Also, we can categorize software under another broad heading as:

- Open Source Software
- Closed Source (Proprietary) Software

System Software

System software is the most commonly used variety types of software. System software ofers a protective shield to all software applications. It also provides support to the physical components of computers. System software coordinates all external devices of computer system like printer, keyboard, displays etc.
System software sits directly on top of your computer's hardware components (also referred to as its bare metal). It includes the range of software you would install to your system that enables it to function. This includes the operating system, drivers for your hardware devices, linkers and debuggers. Systems software can also be used for managing

computer resources. Systems software is designed to be used by the computer system itself, not human users.

Application Software

Application software is used for commercial purpose. The application software is widely used in educational, business and medical fields. Computer games are the most popular
forms of application software. Industrial automation, databases, business software and medical software prove to be of great help in the respective fields. Educational software is
widely used in educational institutes across the globe.
Applications software is designed to be used by end-users. Applications software, in essence, sits on top of system software, as it is unable to run without the operating system
and other utilities. Applications software includes things like database programs, word processors and spreadsheets, e-mail applications, computer games, graphics programs and
such. Generally, people will refer to applications software as software.

Ready-made (off-the-shelf) Software

These types of software are made and distributed by large enterprise companies. They are
made with a general purpose requirement in mind. People using these kind of software may
need to adjust in few features and usability.

Advantages:

• Lower up-front cost
• Contains many features, often more than you need
• Support is often included or can be added with a maintenance contract
• Upgrades may be provided for free or at reduced cost
• If it's software-as-a-service (SaaS) there is no hardware or software to install

Disadvantages:

• Slow to adapt or change to industry needs
• Your feature request may get ignored if it doesn't benefit the larger customer base
• May require you to change your process to fit the software
• Higher customization fees (proprietary software vendors often charge ridiculous hourly fees unless they provide an open API)

Tailored (custom) Software

These types of software are made with a specific customer in mind. They are given the opportunity to interact when in need to make any changes or adding any new features.

Advantages:

• You can start with the minimum necessary requirements and add on later
• Can be tailored to your exact business needs and processes
• Changes can be made quickly

Disadvantages:

• Very high initial cost
• All changes and feature requests will be billable

• May incur additional costs ramping up new developers

Open Source Software

These types of software provide have their source code available publicly. People can download, modify and re-distribute the modified versions of the software. They are maintained and contributed by thousands, if not millions, of people all over the world. They release very frequently and a number of features are added by diferent developers if they wish to do so. They release the bug fixes and other releases very quickly. They are generally free of cost as well, but not all of the open source software are free of cost. Examples of some of the most popular open source software are Linux Kernel, Firefox Web Browser, Apache Web Server, LibreOffice Suite, etc.

Closed Source (Proprietary) Software

These types of software do not provide their source code to the general public. The developers work in a closed group and do not release the source code publicly. They are generally controlled by a particular enterprise company. These software generally charge money for license or subscription. But, there can be closed source software that are free of cost, they are called Freewares. The updates and bug fixes in these types of software are generally late as compared to open source software. People have to wait for the bug to be fixed by the company and release the updated version of the software.
Examples of some of the most popular closed source (proprietary) software are Microsoft Windows, Microsoft Office Suite, Internet Explorer, Skype, etc.

1.7. Types of Programming Languages

Fundamentally, languages can be broken down into two types: **imperative languages** in which you instruct the computer how to do a task, and **declarative languages** in which you tell the computer what to do. Declarative languages can further be broken down into **functional languages**, in which a program is constructed by composing functions, and l**ogic programming languages**, in which a program is constructed through a set of logical connections. Imperative languages read more like a list of steps for solving a problem, kind of like a recipe. Imperative languages include C, C++, and Java; functional languages include Haskell; logic programming languages include Prolog.
Imperative languages are sometimes broken into two subgroups: **procedural languages** like C, and **object-oriented languages** like C++ and Java. Object-oriented languages are

a bit orthogonal to the groupings, though, as there are object-oriented functional languages
(OCaml and Scala being examples).
You can also group **languages by typing: static and dynamic**. Statically-typed languages are ones in which typing is checked (and usually enforced) prior to running the
program (typically during a compile phase); dynamically-typed languages defer type checking to runtime. C, C++, and Java are statically-typed languages; Python, Ruby, JavaScript, and Objective-C are dynamically-typed languages.
You can also **group languages by their typing discipline**: weak typing, which supports
implicit type conversions, and strong typing, which prohibits implicit type conversions. The
lines between the two are a bit blurry: according to some definitions, C is a weakly-typed
languages, while others consider it to be strongly-typed. Typing discipline isn't really a useful way to group languages, anyway.
Diferent languages have diferent purposes, so it makes sense to talk about diferent kinds, or types, of languages. Some types are:
• Machine languages — interpreted directly in hardware
• Assembly languages — thin wrappers over a corresponding machine language
• High-level languages — anything machine-independent
• System languages — designed for writing low-level tasks, like memory and process management
• Scripting languages — generally extremely high-level and powerful
• Visual languages — non-text based
NOTE: These types are not mutually exclusive: Perl is both high-level and scripting; C is considered both high-level and system.

Machine Code

Most computers work by executing stored programs in a fetch-execute cycle. Machine code
generally features :
• Registers to store values and intermediate results
• Very low-level machine instructions (add, sub, div, sqrt)
• Labels and conditional jumps to express control flow
• A lack of memory management support — programmers do that themselves
• Machine code is usually written in hex.

Assembly Language

An assembly language is basically just a simplistic encoding of machine code into something more readable. It does add labeled storage locations and jump targets and subroutine starting addresses, but not much more.

High-Level Languages

A high-level language gets away from all the constraints of a particular machine. HLLs have
features such as:

• Names for almost everything: variables, types, subroutines, constants, modules
• Complex expressions (e.g. 2 * (y^5) >= 88 && sqrt(4.8) / 2 % 3 == 9)
• Control structures (conditionals, switches, loops)
• Composite types (arrays, structs)
• Type declarations
• Type checking
• Easy ways to manage global, local and heap storage
• Subroutines with their own private scope
• Abstract data types, modules, packages, classes
• Exceptions

System Languages

System programming languages difer from application programming languages in that they are more concerned with managing a computer system rather than solving general problems in health care, game playing, or finance. System languages deal with:
• Memory management
• Process management
• Data transfer
• Caches
• Device drivers
• Operating systems

Scripting Languages

Scripting languages are used for wiring together systems and applications at a very high level. They are almost always extremely expressive (they do a lot with very little code) and
usually dynamic (the compiler does little, the run-time system does almost everything).

1.8. Traditional (Structured) Programming Concepts

Structured Programming is a method of planning programs that avoids the branching category of control structures. It is a technique for organizing and coding computer programs in which a hierarchy of modules is used, each having a single entry and a single
exit point, and in which control is passed downward through the structure without unconditional branches to higher levels of the structure. Three types of control flow are used: sequential, test or selection, and iteration.
Traditional programming techniques focus on structures and separate functions that perform operations on those structures. Object-oriented programming treats the properties
of a structure and possible operations on a structure as a single unit, called an object. Thus the principal diference is the transitivity of the action: in traditional programming, the
program performs operations on data, while in object-oriented programming, the program
instructs objects to perform actions that in turn perform operations on data. OOP therefore
introduces an additional level of abstraction over traditional programming techniques.
Principles of Structured Programming:

• Make flow of control as easily understood as possible. (Emphasis on use of Control Structures)
• Build your program from top-down. Decompose the problem into smaller and smaller pieces. (Top down programming)
• Avoid repeating the same code, can fix code in one place.

Top-down programming

Top-down programming refers to a style of programming where an application is constructed starting with a high-level description of what it is supposed to do, and breaking
the specification down into simpler and simpler pieces, until a level has been reached that
corresponds to the primitives of the programming language to be used.

Disadvantages of top-down programming

Top-down programming complicates testing. Noting executable exists until the very late in
the development, so in order to test what has been done so far, one must write stubs .
Furthermore, top-down programming tends to generate modules that are very specific to
the application that is being written, thus not very reusable.
But the main disadvantage of top-down programming is that all decisions made from the start of the project depend directly or indirectly on the high-level specification of the application. It is a well-known fact that this specification tends to change over time. When
that happens, there is a great risk that large parts of the application need to be rewritten.

How does top-down programming work?

Top-down programming tends to generate modules that are based on functionality, usually
in the form of functions or procedures. Typically, the high-level specification of the system
states functionality. This high-level description is then refined to be a sequence or a loop of
simpler functions or procedures, that are then themselves refined, etc.
In this style of programming, there is a great risk that implementation details of many data
structures have to be shared between modules, and thus globally exposed. This in turn makes it tempting for other modules to use these implementation details, thereby creating
unwanted dependencies between diferent parts of the application.

Bottom-up programming

Bottom-up programming refers to a style of programming where an application is constructed starting with existing primitives of the programming language, and constructing gradually more and more complicated features, until the all of the application
has been written.

Advantages of bottom-up programming

Testing is simplified since no stubs are needed. While it might be necessary to write test functions, these are simpler to write than stubs, and sometimes not necessary at all, in particular if one uses an interactive programming environment such as Common Lisp or GDB.

Pieces of programs written bottom-up tend to be more general, and thus more reusable, than pieces of programs written top-down. In fact, one can argue that the purpose bottomup programming is to create an application-specific language . Such a language is suitable for implementing an entire class of applications, not only the one that is to be written. This fact greatly simplifies maintenance, in particular adding new features to the application. It also makes it possible to delay the final decision concerning the exact functionality of the application. Being able to delay this decision makes it less likely that the client has changed his or her mind between the establishment of the specifications of the application and its implementation.

How does bottom-up programming work?

In a language such as C or Java, bottom-up programming takes the form of constructing abstract data types from primitives of the language or from existing abstract data types. In Common Lisp, in addition to constructing abstract data types, it is common to build functions bottom-up from simpler functions, and to use macros to construct new special forms from simpler ones.

2.1. Problem Analysis

The Seven Steps of Problem Analysis

• **Read the case thoroughly:** To understand fully what is happening in a case, it is necessary to read the case carefully and thoroughly. You may want to read the case rather quickly the first time to get an overview of the industry, the company, the people, and the situation. Read the case again more slowly, making notes as you go.
• **Define the central issue:** Many cases will involve several issues or problems. Identify the most important problems and separate them from the more trivial issues. After identifying what appears to be a major underlying issue, examine related problems in the functional areas (for example, marketing, finance, personnel, and so on). Functional area problems may help you identify deep-rooted problems that are the responsibility of top management.
• **Define the firm's goals:** Inconsistencies between a firm's goals and its performance may further highlight the problems discovered in step 2. At the very least, identifying the firm's goals will provide a guide for the remaining analysis.
• **Identify the constraints to the problem:** The constraints may limit the solutions available to the firm. Typical constraints include limited finances, lack of additional production capacity, personnel limitations, strong competitors, relationships with suppliers and customers, and so on. Constraints have to be considered when suggesting a solution.
• **Identify all the relevant alternatives:** The list should all the relevant alternatives that could solve the problem(s) that were identified in step 2. Use your creativity in coming up with alternative solutions. Even when solutions are suggested in the case, you may be able to suggest better solutions.
• **Select the best alternative:** Evaluate each alternative in light of the available information. If you have carefully taken the proceeding five steps, a good solution to the case should be apparent. Resist the temptation to jump to this step early in the case analysis. You will probably miss important facts, misunderstand the problem, or skip what may be the best alternative solution. You will also need to explain the logic you used to choose one alternative and reject the others.
• **Develop an implementation plan:** The final step in the analysis is to develop a plan for effective implementation of your decision. Lack of an implementation plan even for a very good decision can lead to disaster for a firm and for you. Don't overlook this step. Your teacher will surely ask you or someone in the class to explain how to implement the decision.

2.2. Requirements Analysis

The members of a software development team must have a clear understanding of what the software product must do.
The first step is to perform a thorough analysis of the client’s current situation, careful to define the situation as precisely as possible.
This analysis may require examination of a current manual system being operated, or may
need an appraisal of some computerized system to be performed. Once a clear picture of

the current situation is obtained, then the question of “What must the new product be able
to do?” may be answered.
During the requirements phase the developer must :-
• Attempt to identify problems in the current system
• Avoid blindly taking statements about the client’s wants e.g. a wish list
• Attempt to determine the real needs of the client
• Recognize the client is not always conscious of all the needs
• Overcome any lack of computer-literacy on the part of the client i.e. bridge the technical divide between developer and client
• Correctly interpret client’s requests even if not stated in the best way possible
Requirements analysis begins with the requirements team meeting with members of the client organization. The initial meeting may be used to plan subsequent interviews or techniques for soliciting the relevant information from the client’s organization.
Several techniques can be employed to elicit requirements:
• Interviews
• Questionnaires
• Client Documents
• Scenarios
• Rapid Prototypes

Interviews

Both the structured and unstructured interview may be planned. In the case of the structured interview a list of specific closed ended questions are posed and the answers recorded. E.g. “How long does it take to perform activity X” or “How many people are in the
marketing dept?” or “How much was spent on the current system last month?”
These type of targeted questions seek information the interviewer deems relevant in finding the true needs of the client.
In the case of the unstructured interview open-ended questions are asked which allow the
interviewee to outline broad areas or express views/opinions/convictions that may be hard
to quantify. E.g. “Explain why the current product is unsatisfactory?” or “What are the best
features of the current system?” or “What would be the most effective way to accomplish
task X?”
At the end of the interview process the interviewer prepares a written report outlining the results of the interview. Those interviewed should be given copies and allowed to add/clarify statements made.

Questionnaires

A well-structured questionnaire is a useful tool for gathering information. In the case of large departments/organizations it may be impractical to conduct numerous interviews. Unlike the interview process that is interactive in nature, there is no way to pose new questions (follow-ups) based on the answers given to previous questions. This means a

skillful and methodical interviewer will obtain better information than that obtained using a
well-developed and well-worded questionnaire.

Client Documents

Another way to obtain information about a client's operations is to examine the forms that
are currently used. E.g. A purchase order may carry fields such as stock no., product code,
quantity, etc. which shed light on the work flow in the organization.
Other documents such as job descriptions, internal reports, operating procedures etc. can
also be very helpful.

Observations

Use of video cameras can be used to monitor what actually happens on the job in the office
environment. The presence of cameras in the workplace may represent a privacy issue so
careful consideration of all factors and the risks involved should be taken before embarking
on this path. Legal issues may also be involved.

Scenarios

A scenario is a possible way in which a user can utilize the target product to accomplish some specific objective.
The developer may give a scenario of what output will be given by the product-to-be for a
given input. The client representative then indicates what modifications will be needed for
the scenario to correspond to what happens in practice.
Scenarios attempt to illustrate the behavior of a future product in a way that the user can
easily understand. This can result in new requirements being discovered by the developer.

Rapid Prototypes

A rapid prototype can be built to exhibit key functionality of the product-to-be. The client/users then will experiment with the prototype. The development team watches the experimentation and makes notes. Users can also indicate where they feel changes are necessary. The rapid prototype is changed several times until both developers and users
are satisfied that the rapid prototype currently embodies the key needs of the clients.
The rapid prototype then becomes a basis for creating the specifications document.
**** Key Point*** – Rapid prototype must be built quickly and be capable of easy on-the-spot modification. This means being written in a 4GL or interpreted language (Such as Prolog,
lisp, or java)

2.3. Feasibility Study

A feasibility study is carried out to select the best system that meets performance

requirements.
The main aim of the feasibility study activity is to determine whether it would be financially
and technically feasible to develop the product. The feasibility study activity involves the analysis of the problem and collection of all relevant information relating to the product such as the different data items which would be input to the system, the processing required to be carried out on these data, the output data required to be produced by the system as well as various constraints on the behavior of the system.

Technical Feasibility

This is concerned with specifying equipment and software that will successfully satisfy the
user requirement. The technical needs of the system may vary considerably, but might include :

- The facility to produce outputs in a given time.
- Response time under certain conditions.
- Ability to process a certain volume of transaction at a particular speed.
- Facility to communicate data to distant locations.

In examining technical feasibility, configuration of the system is given more importance than the actual make of hardware. The configuration should give the complete picture about the system's requirements:
How many workstations are required, how these units are interconnected so that they could
operate and communicate smoothly.
What speeds of input and output should be achieved at particular quality of printing.

Economic Feasibility

Economic analysis is the most frequently used technique for evaluating the effectiveness of
a proposed system. More commonly known as Cost / Benefit analysis, the procedure is to
determine the benefits and savings that are expected from a proposed system and compare them with costs. If benefits outweigh costs, a decision is taken to design and implement the system. Otherwise, further justification or alternative in the proposed system will have to be made if it is to have a chance of being approved. This is an outgoing
effort that improves in accuracy at each phase of the system life cycle.

Operational Feasibility

This is mainly related to human organizational and political aspects. The points to be considered are:

- What changes will be brought with the system?
- What organizational structure are disturbed?
- What new skills will be required? Do the existing staff members have these skills? If not, can they be trained in due course of time?

This feasibility study is carried out by a small group of people who are familiar with information system technique and are skilled in system analysis and design process. Proposed projects are beneficial only if they can be turned into information system that will

meet the operating requirements of the organization. This test of feasibility asks if the system will work when it is developed and installed.

2.4. Algorithm

Algorithm is a step-by-step description of how to solve a particular problem. An algorithm
provides step by step description of various methods to solve a problem. It is an effective
procedure for solving a problem in a defined number of steps. Algorithm maintains sequences of computer instructions required to solve a problem in such a way that if the instructions are executed in specified sequence.

The Desirable Features of an Algorithm are:

1. Each step of the algorithm should be simple.
2. It should be unambiguous in the sense that the logic should be clear.
3. It should be effective.
4. It must end in finite number of steps.
5. It should be an efficient as possible.
6. One or more instructions should not be repeated infinitely.
7. Desirable result must be obtained on the algorithm termination.

Example for developing an algorithm is steps of program design. Let us consider an example of an algorithm for making a tea.

Step 1: Start
Step2: Put water in Kettle
Step 3: Plug the Kettle into switch.
Step 4: If the water in the kettle is not boil, then go to step 3.
Step 5: Switch off the kettle
Step 6: Pour water from the kettle into the teapot.
Step 7: Stop

The algorithm shows the following three features:

Sequences (Process): Sequences means that each step or process in the algorithm is executed in the specified order. Each process must be in proper place previous steps must
be executed before any other next steps.

Decision (Selection): In some cases we have to make decision to do something. If the output of the decision is true, one thing is done otherwise other control should execute. The
outcomes of decision either true or false, there is not state in between them. For eg. If the
number is less than zero, then the number is negative.

Repetition (Iteration or loop): Repetition can be implemented using control statements.

2.5. Flowchart

The flowchart is graphical representation of an algorithm using standard symbols. In other
words, flowchart is a pictorial representation of an algorithm that used different geometric

pictures for different instruction. The flowcharts play vital role in programming to solve problem and they are quite helpful in understanding the logic of complicated and lengthy problems. Once the flowchart is drawn it becomes easy to write the program in any high level language. Hence the flowchart is better documentation of the complex program. Flowchart is also a very effective analytical tool. With the help of a flowchart, programmer
can quickly show a series of alternative approaches to solve problems.

Symbols Used in Flowchart

The various flowchart symbols suggested by ANSI are as follows:

Terminal Symbol: It is used to indicate a point at which the flowchart begins or ends. The
words START & STOP are written within the terminal symbol. It is represented by geometric
"oval" shape.

Processing Symbol: This symbol represents some operations on data. It is represented by
geometric "rectangle" shape.

I/O Symbol: It is used to represent the logical positioning of input/output operation. It is represented by geometric "parallelogram" shape.

Decision Symbol: This symbol represents a logical operation showing a decision point in a
program. It is represented by geometric "rhombus" shape. The two main components of a
decision symbol are:

• A question that defines the logical operation.
• The result of the decision (yes, no)

Connector Symbol: It is used to indicate a junction at whom the flowchart comes from a
part of the flowchart on another page. It is represented by geometric "circle" shape.

Flow Symbol: A flow symbol is an arrow that shows the flow of program logic in a flowchart. It is represented by "arrow headed line" shape.

Advantages

• **Better Communication:** Flowcharts are a better way of communications it quickly provides logic, ideas and detailed descriptions of computer operations.
• **Effective Analysis:** Flowchart provides a clear overview of the entire problem and its algorithm for solutions. IT shows all major elements and their relationship among components.
• **Proper Documentation:** The flowchart provides a permanent programming logic. It documents the steps followed in an algorithm. A clearly comprehensive flowchart is an indispensable part of documentations for each program.
• **Efficient Coding:** Flowcharts shows all major parts of a program. Programmer can easily instruct the computer in any platform. The flowchart specifies the steps to be coded and help to prevent errors. Thus flowchart is blue print of system analysis and program development phase.
• **Easy in Debugging:** Flowchart helps easy debugging and maintenance of operation in program.

• **Better Understanding:** A flowchart is a pictorial representation of a program. Hence it is easier for a programmer to explain the logic of a program through flowchart.
• **Easy to Convert:** Easy to convert flowchart to programming language .

Limitations

• **Complex Logic:** A flowchart becomes complex when the program logic is quite complicated.
• **Difficulty in alternation and modifications:** if alterations are required the flowchart may need to redrawn completely.
• Very time consuming and laborious job .
• Redrawing a flowchart is a tedious task .
• How much to include in flowchart is unclear.

2.6. Coding, Execution, Debugging and Testing

Coding

The coding is process of transforming program logic design into a computer language format. This stage translates program design into computer instructions using some programming language like C. The coding is the act of transforming operation into program
statement. The knowledge of computer programming language is necessary to write coding. The code written using programming language is also known as source code. During coding of program, programmer should eliminate all syntax and format errors form
the program and all logic errors are detected and resolved during process.

Compilation and Execution

The process of changing high level language into machine level language is known as compilation. It is done by special software, known as compiler. Compilation process tests
the program whether it contains errors or not. If syntax errors are present, compiler cannot
compile the code. Once compilation is completed then the program is linked with other object program needed for execution. Thereby resulting in binary program and then program is loaded in the memory for the purpose of execution. During execution program
may ask user for inputs and generates outputs after processing the inputs.

Debugging and Testing

Debugging is the recovery and correction of programming errors. Even after taking full care
in program design and coding, some errors may remain in the program become the programmer might never about case. These errors may appear during compilation or execution of program. When the errors are appeared the debug is necessary. Testing ensure
that program perform correctly the required task. Thus programming testing and debugging are very closer.

Program Documentation

Program documentation is description of program and its logic written to support

understanding the program structure. Documentation of program helps to use and extend
the program future. A program may be difficult to understand even to the programmer who
wrote the code after same day. If a program coded by one person is to maintained, there
will be more difficult to understand it. Proper documentation is necessary which will be useful and efficient in debugging, testing, maintained and redesign process.

Historical Development of C Language

During 60's there were a number of programming languages developed but almost all were
used for specific purpose only. For example FORTAN (Formula Translation) developed by IBM
was used for engineering and scientific applications. COBOL (Common Business Oriented
Language) developed in 1959 was used for commercial applications especially for business
purpose. Due to those specific purpose languages the programmer had to learn more. i.e.
one language was not for sufficient for every field.
Therefore computer scientist started to think for common language for all possible applications. As a result (ALGOL) Algorithm Language was developed by European and American, but ALGOL never really become popular because it seems to general purpose
programming language only.
To eliminate this problem CPL (Combined Programming language) was developed by the
mathematical Laboratory at Cambridge University of London. This collaborative effort was
responsible for combined in the name of the language. It was heavily influence by ALGOL.
The new language CPL is developed with mixture of FORTAN and COBOL but this is not
become more popular.
The next efforts the Basic CPL is designed by Martin Richard of the University of Cambridge
in 1966, was much simpler language primarily as a system programming language, particularly for writing compiler. At the same time the language B was developed at Bell Labs. It was mostly the work of Ken Thompson with contributions for Dennis Ritchie in 1969. But like BCPL was not become much popular one. At 1972 C was written and designed by Dennis Ritchie at AT and Bell Lab for the use of UNIX operating system by inheriting the features of B and BCP language add more features. The origin of C is closely
to the development of UNIX operating system. It was named "C" because many of its features were derived from an earlier language "B". C is a general purpose, structure,

procedural imperative language. Some of the most common C compilers are Turbo C/C++
IDE, Borland C/C++, and Microsoft visual C++ etc.

Importance/Advantages of C Programming Language

Robust Language: C is robust language because which rich set is of built in functions and
operators can be used to write any complex program. C compiler combines the capabilities
of an Assembly language with the features of high level language.

Efficient and Fast: Programs written in C are efficient and fast due to its variety of data types and powerful operators.

Highly Portable: C is highly portable; meaning that C program written for one computer can run on another computer with little or no modifications of source code.

Structure Language: C is structured language as it has a fixed structure. C program can
be divided into numbers of block or modules. Thus, the proper collection of modules would
make complete program.

Extensible: C programs may contain a number of user defined functions. We can add our
own user defined functions to the C library if required.

Middle Level Language: C is middle level language because it combines the best part of
high level language with low language. It is well suited for writing both user and machine oriented program.

Rich System Library: There are large number of built in functions, keywords and operators in C system library supports.

Execution of C Program

1. Writing the Source Code

Computer instructions are written in a text editor to perform certain jobs. These instructions written using the syntax of C is known as the source code of C program. The source code can be written using any text editor such as Notepad, Turbo C. However it should be saved with ".C" extension. For e.g. "first.c" is valid file name.

2. Compiling and Linking the Program

The computer instructions written in the form of source code are translated into executable code with the help of compiler that is suitable for program execution. The translation is done by a special program called compiler that processor statements written in programming language and converts them into a machine level language. So compiling means creating an executable file for particular platform. The compiler first analyzes the entire language statements syntactically one after another and then build the output code called object code. The object codes are machine code that the processors can process or execute one instruction at a time.

During compilation linking process takes place. Linking is the process of putting translated program and other objects from system library such as reading inputs, producing output and computing mathematical functions. To create an executable program the object program must be linked to system library subprogram.

3. Execution of Program

While execution of program the loader loads executable object code into the computer memory that executes the instruction. During execution, the program may require for some data to be entered through the keyboard.

Basic Structure of C Program

The structure of C program implies composition of a program it describes main components
to write in C program, how are they organized. The following table shows the parts are included in the structure of C program.

- Documentation Section
- Linking Section
- Definition Section
- Global Declaration
- main() function
- Sub program()

1. Documentation Section

This section contains a set of comments lines giving the name of program, the designer may write algorithm, methods used and other detail information related to the program. This will be useful in further for users and development teams. Documentation acts as a communication medium between members of development team working in the same project. It helps while debugging and testing the program.

/* This program display natural numbers from 1 to 10 */

Note: /----- */ denotes comments in C Whatever the text written within comment if just ignore by the compiler.*

2. Linking Section

This section provides instruction to the compiler to link functions with program from the system library.

#include<stdio.h>

#include<conio.h>

It links system function library I/O that handles printf() and scanf()functions. They are preprocessors which provide the designer platform to design program.

3. Definition Sections

In this section all symbolic constants are defined.

#define PI 3.1416

4. Global Declaration

The variables which are used in more than one function/block are called global variables.

Those variables are defined in this section. This section also defined all the user defined functions.

5. Main Functions

Every C program must have one main function through which program starts execution. The main function has deceleration and execution parts. Deceleration part declares all the
variables used in the program execution part. In this section some sort of calculation and
other functionality are to be kept. e.g.

int i,k; /*deceleration of integer variable to store integer value*/
i=90,k=4*i; /*execution part and assignment parts.*/

6. Subprogram

This section contains all the user defined functions that are called in the main function.

Example C Program

```
/*C program display "Hello World"*/
#include<stdio.h>
#include<conio.h>
void main()
{
clrscr();
printf("Hello World.");
getch();
}
```

3.1. Character Sets, Keywords & Data Types

Character Sets

The set of characters that used to form words, numbers, and expression in C is called character set. The combination of these characters form words, numbers and expressions
in C. The characters in C are grouped into the following four categories:

- **Letters or Alphabets:** Upper Case / Lower Cases (A-Z, a-z)
- **Digits:** 0 to 9
- **Special Characters:** (,,;?,””,\,/-,>,^&,%,*,/=,+!,{()}>,<,[~,! etc)
- **White Space:** (Blank space, Horizontal tab, Carriage return, New line, Form feed)

Keywords

Keywords are predefined words for C program. All keywords have fixed meaning and these
meanings cannot be changed. They serve as basic building blocks for program statements. They are as: double, int, void, struct, typedef, char, default, do, goto, if else, return, for, while, switch etc. These words are used for pre-defined purpose and cannot be
used as identifiers to declare variable name. Thus the keywords are also called reserved
words.

Data Types

There are various data types for example 10 and 100.4 are data of different items. The data 10 is integer number where as 100.4 is fractional number. There are other varieties of
data type supported by C, each may be represented differently within computers memory.
The varieties of data type of C supports are:

- Primary data type
- User-defined data type and
- Derived data types

The primary data types and derived are discussed in this section. The user defined data types such as arrays, functions, structures and pointers are discussed in next chapters.
There are five primary data types they are:

- Integer (int)
- Floating point type (float)
- Double-precision floating type (double)
- Character Type (char) and
- Void Type (void)

Integer Type

The integers are whole numbers, non fractional numbers. Generally integer requires 16 bytes (2 bit) of storage area. C has further three types of integer they are int, short, and long in all three data types there are two sign and unsigned forms. The appropriate data types are used according to our requirements.

Signed Integer(signed int) Unsigned integer(unsigned int)

It represents both positive and negative

integers.
It represents only positive integers
The data type qualifiers is signed int or int
Variable are defined as signed int a, b;
The data qualifier is Unsigned int or
unsigned. Variables are defined as unsigned
int a,b;
By default all int are signed Unsigned int have to declared explicitly
It reserves 16 bits(2 bytes) in memory It reserves 16 bits(2 bytes)in memory
Int represents values of range -32768 to
+32767 (2^{-15} **to** 2^{15} **-1)**
It represents integer values of range 0 to -2
^{16} -1 (0 to 65535)
Its conversion character is %d Its conversion character is %u

Signed short Integer(signed short int)

Unsigned Short integer(unsigned short int)

It represents both positive and negative
integers.
It represents only positive integers
The data type qualifiers is signed int or
short int
Variable are defined as signed short int a,
b;
The data qualifier is unsigned short int or
unsigned. Variables are defined as unsigned
short int a,b;
By default all short int are signed. Unsigned short int have to declared explicitly
It reserves 16 bits(2 bytes) in memory It reserves 16 bits(2 bytes)in memory
Int represents values of range -0 to 255(8
bits)
It represents integer values of range (-127 to
128) bits
Its conversion character is %d or %i Its conversion character is %u

Signed long Integer(signed long int) Unsigned Long integer(unsigned long int)

It represents both positive and negative
integers.
It represents only positive integers
The data type qualifiers is signed long int or
long int . Variable are defined as signed long
int a, b;
The data qualifier is unsigned long int or
unsigned. Variables are defined as unsigned
long int a,b;

By default all short int are signed. Unsigned long int have to declared explicitly
It reserves 32 bits(4 bytes) in memory It reserves 32bits(4 bytes)in memory
Int represents values of range -2147483648
to +2147483647 (-2^{31} to $+2^{31}-1$)
It represents integer values of range
0 to $+2^{32}-1$ (0 to 4294967295)
Its conversion character is %ld Its conversion character is %lu

Floating Point Types

Floating point types are factional numbers. They are defined in C with keyword float. Floating numbers reserve 32 bytes (4) bytes of storage with 6 digits of precision.

Float

- It reserves 4 bytes in memory
- It represents fractional numbers of range 3.4* 10-38 to -3.4 1+38
- The data type qualifier is float variable declaration as float a;
- Its conversion character is % f.

Double Precision Floating Point Type

When accuracy provided by a float number is not sufficient, the type double can be used
to define the number. A double data type number used 64 bits (8 bytes) giving a precision
of 14 digits. There are double expression numbers. To extent the precision further long double can be used which use 80 bits (10bytes) giving 18 digits of precision.

Character Type

A single character can be defined as a character type data. Characters are stored in 8 bits
(1 byte). The qualifier is char. The qualifier signed and unsigned may be used with char. The unsigned char has values between 0 to 255. The sighed char has values form -128 to
127. The conversion character for this type is % c.

Void Data Type

The void type has no return values. This is usually used to specify a type of function when
it does not return any value to the calling function.

User Defined Data Types

C supports type definition which allows defining an identifier that would represent and existing data type. The typedef statement is used to give new name to an existing data type. It allows user to define new data types that are equal to existing data types. It takes
the general form

- ***typedef int integer;***

Here integer symbolizes int data type. Now we can declare int variable a as integer a instead of int a.

- ***typedef float decimal;***

Here float f is equivalent to decimal f. Thus typedef statement is used to alias existing data types as convenient.

3.2. Constants and Variables

Constant

Constant is quantity that does not change during the execution of program. C supports constant can be divided into different category. They are:

Integer Constants

Integer constant refers to a sequence of digits with not decimal point either positive or negative. If no sign precedes an integer constant, it is assumed to be positive. No commas
or blank spaces are allowed within the integer constant. Example 345,456,-8765.
There are three type of integer namely decimal, octal and hexadecimal.
Decimal Integer Constant: The decimal integer constant are the set of digits 0 to 9 proceeded by an optional – or + signs. Here decimal does not imply fractional numbers. It
represent numbers having base of 10.
Octal Integer Constants: An octal integer constant consists of any combination of digits
for the set form 0 to 7, with leading 0. Valid example are 074, 0676.
Hexadecimal Integer Constants: A hexadecimal integer constant consists of any combination of digits form the set of 0 to 9 and ABCDEF with leading) 0X examples are 0x345, 0x123, 0xaA.

Real Constants

Real constants are often called floating point constants. They can be written into two forms- fractional form and exponential form. Real constants are two types.
Fractional Constant: Fractional form of constants must have at least one digit and a decimal point. It can either be positive or negative but default sing is positive. Commas or
blank space are not allowed within a real constant. For example: 23.56, 45.9,-67.5003 are
in fractional form.
Exponential Constant: In exponential representation, the real constant is represented in two parts mantissa and exponents. The digit before is called mantissa and after is called
exponent. The mantissa part may be positive or negative sign but default is positive. Similarly, the exponents must have at least one digit which can either positive or negative
for eg. +3.2e5, 9.1e6.

Character Constant

A character constant is single character alphabet digit or special symbol enclosed within single ‘ ‘ marks. For example ‘A’ is a valid character. The maximum length of a character
constant can be only one character. It must enclose by single code for example: ‘a’,’g’.

String Constant

A string constant is a sequence of characters enclosed in double quotes. It may contain letters numbers or special characters or blank space. However it does not have an equivalent ASCII value. For example “Hello good morning”. This must be enclosed by

double coated(“”).

Variable

A variable is a symbolic name which is used to store data items. Variables are defined in a
computer program to represent an item of data input by the user, any intermediate calculations results. Unlike constant the value of variable can change during the execution
of a program. The same variable can store different values at a different portion of program.
Variable name may constant of letters, digits, or underscore characters. Since a variable is
an identifier, the rules for naming variables are similar to those of identifiers.
Some valid examples data type are: firstname, num1,x2, email_id etc can be used
Some invalid examples are: data type, 1nepal, (total), first name, %male. etc.

Need for Variable Declaration

The declaration of variable associates the variable with a specific data type. This means each variable must be declared or defined as what type it is. By doing this the compiler will allocate a memory space in the computer as required by data type.
Thus variables are assigned with specific data type according to the values to be stored. If
we have integer values to store, we need to define a variable of type “int”. It gives the name of variable and its types should have matched properly, when we declared it. It allocates appropriate memory space according to the data types declared in front of it.

Rules for Variable Declaration

• The variable should be stored with only letters or underscore.(itn i; int_p;)
• The variable name should not be any keyword.(int auto,int int, int do;)
• The variable name is case sensitive characters.(if you declared a variable x then while accessing it do not use X: these two are different representation)
• No two variables of the same name are allowed.(int i; int i are not allowed in same program)
• It must starts form letters/alphabets.(int a1, char p2 not but int 1a are not allowed)

Preprocessor Directives

Preprocessor directive is a collection of special statements that are executed in the beginning of compilation process. They are written in source program before the main program. Preprocessor directives follows special syntax rules that are different form normal C syntax. They all begins with the symbol # (Hash) and do not require semicolon at end.

#include<stdio.h> /*used for standard input and output*/
#include<conio.h> /* used for console input and output*/
#define PI 3.1416 /*defining symbol*/
#define TRUE 1 /*used for defining TRUE as 1*/
#define FALSE 0 /*used for defining FALSE as 0*/

These statements are called preprocessor directives as they preprocessed source program before compilation of any source code in the program. The other codes in the program are compiled sequentially line –by line.

Escape Sequence

An escape sequence is non-printing characters used in C. It is character combination of backslash followed by latter or by combination of digits. Escape sequences are typically used to specify actions such as carriage returns and tab movements on terminals and printers. They are also used to provide literal represents of characters that have special meanings such as double quotation mark (“”).

```
#include<stdio.h>
#include<conio.h>
void main(){
printf (“Hello\n I am testing escape sequence”);
getch(); }
```

Generally, any text written within double quotes (“”) in printf () function will be displayed on the screen. There are some more escape sequences .

\a Alert purpose by an audible alert or beep sound
\b To delete one character to the left of current cursor position
\f Form feet to feed a blank page when the user prints the
document
\n Line feed next line
\r Carriage return move cursor to the next line during typing
\t Horizontal tabs to move cursor to next tab stop position
\0 For null characters

Symbolic Constant

A symbolic constant is a name that is used in place of sequence of characters. The character may represent numeric, a character constant or a string constant. When a program is compiled, each occurrence of symbolic constant is replaced by its corresponding character sequences.

3.3. Operators

Operator

An operator is a symbol that operates some sort of calculation. Operators are used in programming to perform certain mathematical or logical manipulations. For example, in a
simple expression 9+67 the symbol “+” is called an operator which operates on two items
9 and 67 are operands. The data items that operate through operator are called operands.
Here 9 and 67 are operands. According to the number of operands required for an calculation, we can classify operators into unary, binary and ternary.

Unary Operators: The operator which requires only one operand is known as unary operators. For example: ++ increment – decrement, a++,--a,++a, Here only one variable is used.

Binary Operators: The operators which requires two operand are known as binary operators for example + - * / >, < , a+b, a/b etc are binary operator.

Ternary Operators: The operators that require three operands are known as ternary operators for example “?:c” is conditional operator.

C operators can be classified into a number of categories

• Arithmetic operators
• Relational operators
• Logical operators
• Assignment operators
• Increment and decrement operator
• Conditional operators
• Conditional operators
• Bit wise operators
• Special operators

Arithmetic Operators

An arithmetic operator performs arithmetic operations. There are five arithmetic operators
in C.

Operator Meaning Example Output int A=20, B=6, C=6;

+ Addition A+B 26
- Subtraction A-B 14
* Multiplication A*B 120
/ Division A/B 3
% Modulo A%C 2(gives remainder)

The +,-,*, / all work in the same way in usual mathematics. The modulo operator (%) calculates remainder after division of one integer operand by another integer operand. So
the remainder of 20 divided 6 will give remainder 2 but 8%2=0 gives zero as remainder.

Integer Arithmetic

When the operands in a single arithmetic expression such as a+b are integer, the expression is called an integer arithmetic expression. Integer arithmetic always yields an
integer value. That is why the operation 20/6 in the above example yields 3. The factional
positions of actual results (3.333333) are truncated.

Division Rules

int/int=int float/float=float
int/float=float float/int=float

During integer decision if both operands are of same sign, the results truncated towards zero.
4/7=0. However if one of the operands is negative then the direction of truncated is machine dependent. This is -4/7 may be 0 or 1 depending upon the machine. Similarly during modulo, the sign of the result is always the sign of the first operand. 20% 3=2, -20%3=-2, the modulo division operation cannot used on floating point data.

Relational Operator

Relational operators are used to compare two similar operands, and depending on their relation, take some actions. The relational operators compare their left hand side operand
with the right hand side operand by using lesser than, greater than, lesser than or equal to
relations. The value of relational expression is either 1 or 0 (if conditions is false 0 will

produce where as if condition is true result become 1).

Operator Meaning Example Output(int a=20,b=6)

< Less Than A<b 0

> Greater than a>b 1

<= Less than or equal A<=b 0

>= Greater than or
equal
a>=b 1

== Equal to A==b 0

!= Not equal to A!=b 1

Logical Operator

Logical operators are used to compare or evaluate logical and relational expressions. The
operands of these operators must produce either 1 or 0. The whole result produce by these operators is also either true or false. There are three logical operators used in C programming.

(&&) Logical AND (||) Logical OR (!)Logical NOT

A B A&&B A||B !A !B

0 0 0 0 1 0

0 1 0 1 1 0

1 0 0 1 0 1

1 1 1 1 0 1

Assignment Operator

Assignment operators are used to assign the result of an expression to a variable. The common assignment operator is =. There are other various shorthand's assignments they
are:

a+=b // a=a+b

a-=b // a=a-b

a/=b // a=a/b

a%=b // a=a%b

• If two operands in assignment expressions are different data type, then the value of expression on the right will automatically be converted to the type of identifier on left.

• A floating-point value may be truncated if assignment to an integer.

• A double precision value may be rounded if assignment to floating point value.

• An integer quantity may be altered if assignment to a shorter integer variable.

Increment and Decrement Operators

The increment operator is to increase the variable of an operand by 1, and the decrement
operator is used to decrease the value of an operand by 1. They take only one operand, so
called unary operator.

The increment/decrement operators are prefix or postfix in prefix notation, the value of variable is first incremented or decremented and then the value is assigned to the variable.

Similarly, the postfixes, first the previous value of variable is used and then the value is incremented or decremented.

Conditional Operator

The conditional operator that takes three operands so it is also called ternary operator. The syntax of this operator is ***Expression1? Expression2: Expression 3***.
Here, expression 1 is evaluated first, if Expression 1 is true, the value of Expression 2 is assigned to the large variable. Else Expression 1 is false then value of Expression3 is assigned to the large variable.

Comma Operator

The comma operator can be used to link related expressions together. Comma linked lists
of expressions are evaluated from left to right and the value of the rightmost expression is
the value of the combined expression.
N2=(n1=50, n2=10,n1+n2)
This will first assign the value 50 to n1, then again 10 is assign to n2 variables. The finally
the sum is calculated then assigned to N2 variable (n1+n2 ie 60). Since comma operator
has lower precedence of all operators.

Sizeof Operator

The sizeof operator is used with an operand to return the number of bytes it occupies. It is
a compile time operand. The operand may be a constant, value or data type qualifier as following. The syntax sizeof(argument). Here sizeof operator gives its argument size.

Operator Precedence

The precedence is used to determine how an expression is evaluated and which order it will start to evaluation among various operators used. There are distinct levels of precedence and an operator may belong to one of these levels. The operators at the higher level of precedence are evaluated first. The operators of same precedence are evaluated either form left to right or right to left depending to the level . This is known as associative property of an operator.

Precedence Operator Description Associative

1 (Highest) ()[] Function call Left to right
2 +,-,+
+,--,!,~,*,&,
sizeof, type
Unary,increment/decrement,
address, pointer, type cast
Right to left
3 *,/,% Multiplication/ division/modulo l-R
4 +,- Addition/division L-R
5 <<,<=<>,>= Relational L-R
6 ==,!= Relational L-R
7 &,|<&&,||?: Logical L-R
8 =,*=,/=,+= Assignment Right to Left

9(Lowest) , Comma L-R

Type Conversion in Expression

C allows mixing of constants and variables of different types in single expression. In such
situation, one type must be converted to another before evaluation can be done. This kind
of conversion is called type conversion. The common type conversion are two types implicitly or explicitly.

Implicit Type Conversion

When there are constants and variables of different types in an expression, C automatically converts any intermediate values to the proper type. This conversion assures that the expressions can be evaluated without losing any digits inside. This automatically convert typed is known as implicit type conversion.
During evaluation if operands are of different types, the lower type is automatically converted into the higher type before the operation proceeds. The conversion takes place
according to rule called the conversion hierarchy.
The final result of an expression is converted to the type of variable on the left of the assignment signs before assignment the value to it. However the following changes are introduced before the final assignment.

Float to int causes truncation of the fractional part
Double to float causes rounded of digits
Long int to int causes dropping of the excess higher order bits.

Explicit Type Conversion

Instance where the type can be converted forcefully by the user himself is called explicit type conversion or casting value. The general rule to cast is (type name) expression.
N1=(int)9.50 9.50 is converted to integer by truncation
N2=(int)16.9/(int)6.5 Evaluation is done 16/6, resulting 2
N3=(double)sum/n Sum is calculate to double and division is done
in floating mode
N4= (int) (a+b) The results is converted then assign
N3=sin((double)x) X is converted to double and the sin value is
calculated and assign

3.4. Input and Output Operations/Management

Formatting scanf() and printf() functions

A program without any input or output has not meaning; generally a program reads input
data form keyboard or other files. The program then processes input data and result is displayed on the screen or monitor. Let’s consider program to create mark sheet of student then program must take marks of five subjects then after getting some information the percentage and division will be calculated. Finally the required results is shown in the screen.
C has number of input and output functions, when a program needs data. We take the data through input functions and results to output devices through output functions. As keyboard is standard input devices, the input functions used to read data form keyboard

are called standard input function.
The standard input functions which are scanf(), getchar(), getche(), getch(), gets(). Similarly output functions are printf(), putchar(), putch(), puts (). The standard library <stdio.h>provides functions for input and output. The instruction include<stdio.h> tells the compiler to search <stdio.h>from the standard i/o directory and place its contents in the program. The content of header files become part of source code. The input and output factions are two types: Formatted functions and Unformatted functions.

Formatted Function

Formatted functions allow to give input/ read form the keyboard and output displayed on screen to can be formatted according to user requirements. The input function scanf() and
printf() are fall under this category. While displaying certain data on screen we can specify
the number of digits after point, number of spaces before the data, and position where output is to be displayed are handled through there functions.

Formatted Input/Scanf()

Formatted input refers to an input data that has been arranged in a particular format for example 50,30.4, name. This line contains three types of data and must be read according
to its format. The first part should move into variable int, the second must be variable in float and the third into char. This is possible in C by using scanf() function.
The built in function scanf() can be used to enter input data to the computer form the standard input devices. The functions can be used to enter any conditional or numerical values, single characters or string. In other word, it is used for runtime assignments of variables.
Syntax: scanf("control string",args,arg2,args);
The address of location where the data is stored signifies in control string. It represents pointer that indicates the address of data items within computer memory (&). The control
string consists of individual groups of data format with one group for each input data items. Each data format must begin with a percentage sign (%).

Formatted Output/Printf()

The formatted output refers to the output of data that has been arranged in particular format. The printf() is a built in function which is used to output data from the computer output devices on screen. The printf statements can be used to control the alignment and
spacing.
Syntax: printf ("control string", args,args,args);
The control string consists of four type items
• Characters that will be printed on the screen
• Format specification that define output format for display of each items
• Escape sequences that define output format for display
• Any combination of characters, format specification and escape sequences
The control string has as form:
printf (" %[flag][field][.precision] conversion character");
Flags [Optional]: The flag affects the appearance of the output. They must be placed

just after percentage sign. -+,0, o, x or blank space are symbol . – indicates the left justified. + indicates +ve or –ve numbers. A blank space will precede each positive signed
numerical data. o it refers octal similarly x refers hexadecimal numbers.
Output of Integer Numbers: %wd is used to display in a desired width of integer values. Where w is the minimum field width for the output and d specifies that the value printed is an integer.
Output of Single Character: A single character can be designed in a desired position using format %wc.
Precision [Optional]: The operation of precision field depends on data type of conversion. It must start with period (.).
Output of Real Numbers: The precision can further be extended using field width of form w.pf where w is field width and p is precision. The conversion character is %f.
Output String: The format specifies for output string is form of %w. p s where w is width
for display and p instruct that only the first p character of string are displayed, it count white space too.

Unformatted Functions

Unformatted function do not allow user to read or display data in desired format. These library functions are available in the header files they are: getchar(), putchar(), gets(), puts(), getche(), getch() are unformatted functions.
getchar() and putchar(): The getchar function reads character form standard input devices. It takes the following format char p=getchar();. Similarly putchar () function display a character to standard out devices. It takes flowing format putchar (character variable);
getch(), getche() and putch(): The function getche() and getch() reads single character of instant of time, it is type without waiting for enter key to be hit. The difference between them is that getch() reads the character typed without echoing it on the screen, while getche () reads the character and echoes display it onto the screen. getch() format is char variable=getch(); the function putch() prints a character onto the screen. It has the form putch (char variable name);

4.1. Control Statements

Introduction

The statements which alter flow of execution of the program are known as control statements. In the absence of control statements, instructions are executed in same order
in the program. Such instructions are called sequential statements. Sometimes we have to
perform certain task depending on outcome of logical test. Similarly sometimes it is necessary to perform repeated action or skip some statements. For these cases we need
control statements. They control flow of program. So that it is not necessary to execute statements in the same order in which they appear in program.

Any program logic, no matter how complex, could be expressed by using only the following
three simple control structures:

a. Sequence Logic
b. Selection Logic and
c. Iteration (or Looping) Logic

Sequence Logic: Sequence logic is used to perform instruction to be followed after another in sequential order. The logic of flow the instruction from top to bottom. In this type
of instruction the instruction starts top most instruction execute first then sequential statements will execute in top down hierarchy.

4.2. Selection Logic/Decision Making Statements

Selection is a special kind of branching, in which one group of statements is selected from
several alternatives groups, i.e., it allows alternative actions to be taken according to conditions that exist at particular stage in program execution. The conditions are normally
in the form of expressions that are evaluated test results (true or false).

Selection logic are following types:

a. if …
b. if … … else
c. nested if
d. switch statement
e. goto

Since decision making statements control the flow of execution, they also fall under the category of control statements.

4.2.1. If… statement

The if statement is a powerful decision making statement is used to control flow of execution of the program. It is basically a two way decision statements and it is within expression. The if evaluates the expression first then if the value of expression is true, it executes the statements within its block. Otherwise it skips statements within its block and
continues form first statement outside the if block.

if (test expression)
{
statement block;
} statement x;

The statement block can be a single statement or group of statements. If the test expression is true the statements block within if is executed; otherwise the statement block
will be skipped. The execution will jump to the statement x are executed in sequence order.
The brace {} can be omitted if there is only one statement following the true condition of if statement.

4.2.2. if .. else Statement

The if.. else statement is an extension of simple if statement. It is used when there are two
possible actions are to be evaluated. When a condition is true it executes statements inside
if condition. Else the statement inside else is executed.
If the test condition is true, then true block statement, immediately following if statements
are executed, otherwise, false block statement are executed. In either case true or false block will be executed, not both at the same time. This is best suit in only two choices or alternative.

4.2.3. If... else if... else Statement

The if... else if... else statement is used when there are more than two possible actions depending upon outcome of test condition. When specific action is satisfied the specific action only is executed, no other action will be executed. In such situation if... elseif... elseif.. else... structure is used. If the elseif statement is satisfied then else if condition is executed else last else statement is executed.
The condition is evaluated from top to button. As soon as true condition is found, the statements associated within it are executed and the control is transferred to statement X,
at last the others are skipping the structure. When all conditions become false then final else containing default statement will be executed.

4.2.4. Nested if... else Statement

If an entire if..else construct is written under either the body of an if statement and the body of an else statement, then such type of construction is called nested if…. else statement. In other words nested if else is such type of construction in which there is an if..else statement within another if.. else statement. One if...else is nested into another if...
else statement in nested form. In nested form the condition of inner statement is evaluated
only if the condition of outer if is satisfied. Otherwise it is skipped and the else part of outer
if is evaluated.

Dangling else Problem: One of the most common problem of nested if... else statement
is dangling else. This occurs when a matching else is not available of an if statement. The
solution to this is to always match an else to the most recent unmatched if in the current block.

4.3. Loops

4.3.1. Loop Operation

Loop may be defined as block of statements which are repeatedly executed for a certain number of times or until a particular condition is satisfied. When an identical task is to be
performed for a numbers of times, then loop is used. The loop is executed repeatedly until
an expression is true. When expression become false, the loop terminates and then control
is passed to the another statements followed by the loop. A loop consists of two segments,
one is known as the control statement and other is body of loop. The control statement in
loop decides whether the body is to be executed or not.

Types of loops:

- For Loop
- While loop
- Do... While loop

4.3.2. for Loop

The for loop is useful to execute a statement for a number of times. When number of repetition action is required we use for loop is more efficient and reliable. Thus, this loop is
also known as a determinant of definite data.

Syntax:

for (counter initialization; test condition; increment or decrement)
{
statements; or body of loop;
}

Counter Initialization: The counter variable is initialized using assignment statements such as i=0,i=10 and count=0. Here the variable i, j, count are known as counter control variable whose value controls the loop execution. This expression or statement is executed
only once prior to the statements within the for loop.

Test condition: The value of counter variable is tested using test condition. If the condition is true, the body of the loop is executed, otherwise the loop is terminated and execution continues with the statements that immediately follows the loop structure.

Update Expression/ Increment or Decrement: When body of loop is executed, the control is transferred back to statement after evaluating last statement in the body of loop.

The counter variable is updated either increment or decrement and then only it is to be tested with that conditions and the same process is repeated until the condition become false.
The different components in for loop are executed in direction sequence as above.
At first the counter is initialized to some value. Then counter variable is tested with test condition. If test is true, body of loop is executed. After finishing body, the counter variable
is incremented or decremented and then again update counter variable is tested with the
condition. The same process repeated as long as the condition is true. If the result with test
condition becomes false the control passes outside the loop.

4.3.3. while Loop

Syntax:

while(test condition)
{
body of loop ;
}

The test condition is evaluated and if the condition is true then body of the loop is executed. After execution of data once, test condition is again evaluated and if it is true, the body is executed one again. This process of repeated execution of body continues until
test condition finally becomes false and control is transferred out of loop. On exit, the program continues with the statement immediately after the body of loop.

4.3.4. do while Loop

Syntax:

do
{
statement or body of loop;
}
while (condition);

In do while loop body of loop is executed first without testing condition. At the end of loop,
test condition in the while statement is evaluated. If condition is true, program continues to
evaluate the body of t he loop once again. This process continues as long as condition is
true. When condition becomes false, the loop is terminated and then control goes to the statements that appears immediately after the while statement. Since the test condition is
evaluated at bottom of the loop do while loop construct provides an exit controlled, or bottom of the loop and therefore body of loop is always executed at least once.

4.3.5. Nested Loop

When the body part of the loop contains another loop then the inner loop is said to be

nested within the outer loop. Since a loop may contain other loops within its body, there is
no limit of the number of loops that can be nested. In the case of nested loop, for each value of outer loop, inner loop is completely executed. Thus inner loop operates fast and
outer loop operates slowly.

While Do while

While loop is entry control loop i.e. test
condition is evaluated first and body of loop
is executed only if test is true

Do while loop is exit control loop i.e. the body of
the loop is executed first without checking
condition and at the end of body of loop, the
condition is evaluated.

The body of the loop may not be executed at
all if the condition is not satisfied at the very
first attempt.

The body of loop is always executed at least
once

4.4. break, continue and goto Statement

4.4.1. break Statement

The brake statement terminates the execution of loop and controls are transferred to the statement immediately following the loop. Generally the loop is terminated when its test condition becomes false. But if we have to terminate the loop instantly without testing loop
termination condition, the statement is useful. The syntax: break;

Break statement is also used in switch statement which causes a transfer of control out of
the entire switch statement. To the first statement following the switch statement to the first following the stitch statement. The switch statement will be discussed later.

4.4.2. continue Statement

The continue statement is used to bypass expression of current process through a loop. The
loop does not terminate when a continue statement is encountered. Instead of termination,
it skipped some portion some part of program and the computation proceeds directly to the
next stage through the loop. syntax is: continue;

4.4.3. goto Statement

The goto statement is used to alter normal sequence of program execution by unconditionally transferring control to some other part of program. The go to statement transfer control to the labeled statement somewhere in current function. The go to statement has the following syntax : goto label; Here label is an identifier used to the target statement to which control would be transferred. The target statement must be labeled using syntax: label statements:

Generally use of go to statement is avoided as it makes program illegible. This statement is
used in unique statements like:
• Branching around statements or groups of statements under certain condition.
• Jumping to end of a loop under certain condition, thus passing remainder of loop during current pass.
• Jumping completely out of loop under certain conditions, Termination executions of a loop.

4.5. switch Statement

When there are number cases to be handled, switch statement is another way of representing this multi way selection. A switch statement allows user to choose a statement among several alternatives. The switch statement is useful when a variable is to
be compared with different constants and in case it is equal to a constant a set of statements are to be executed. The constants in case statement may be either char or int
type only.

Syntax:

switch (variable name)
{
case statements:
break;
case statements:
break;
default:
statements;
}

5.1. Introduction to Arrays

An array is a group of related data items that share common name. In other word an array is
a data structure that store number of data items as a single entity (Object). The individual
data items are called elements and all of them must have same data type.
Array is used when multiple data items that have common type and format. Therefore an
array is a collection of individual data elements that is ordered (0, 1, 2, 3….), fixed size of
homogeneous elements.
Suppose we have to require 10 numbers of integer type and we have to sort ascending. If
we have no array we must have declared 10 integer of elements like int a1, b1, c1, c2, c3 up
to 10 times of int type to store these data .When defining different variables to be stored for different numbers of times are very tedious jobs. In such case, we use array to store multiple numbers of same type to be stored at single time. The individual elements are separated by array name followed one or more subscript or index enclosed in square bracket. The individual data item can be character, integer, floating numbers. However they
must be the same type and same storage class. We may used auto, register, static keyword
just before deceleration variable of array type.

5.1.1. One Dimensional Array

There are several forms of an array in C, they are one dimensional and multi dimensional
array. In one dimensional array, there is a single subscript or index whose value refers to
individual array element which ranges from 0 to (n-1) where n is size of array e.g. int A [5] is
a declaration of one dimensional array of int type. Its elements can be illustrated as:

1st element	2nd element	3rd element	4th element	5th element
A[0]	A[1]	A[2]	A[3]	A[4]
2001	2002	2003	2004	2005

The elements of an integer array A [5] are stored in continuous memory locations. It is assured that the starting memory location may be 2001. As each integer element requires 2
bytes, subsequent elements appears after gap of 2 locations. (Here int A[5] occupy 10 bytes
memory area).

5.2. Declaration of Arrays

5.2.1. Declaration of 1 D Array

Syntax: storage class data type array name [size];

• Storage class refers to the storage class of array. It may be auto, static and register. It is optional.

• Data type declares of array variable hold. It may be int, float, and char. The array elements are all values of same data type. If int is used, this means that array store all data items of integer types.

• Array name is name of array. It is user defined name for array. The name of array may be any name valid for name of variable deceleration.

• Size of the array is the number of elements in the array. The size is mentioned within [] bracket. The size must be in int, constant like 10, 5. Or constant expression likes symbolic constant SIZE. #define SIZE 20, the array can be defined as int a [SIZE]. The size of an array must be specified (i.e. should not be blank) except in array initialization. Example: int num [5];

This statement tells to the compiler that num is an array of int type; num can store five integers' values. The compiler reserves 2 bytes of memory of each integer array element.

Thus total memory size allocated by an integer is 10 bytes. It is individual elements are recognized by num [0], num [1], num [2], num [3], num [4]. The integer value within square

bracket [] is called index of array. Index of an array always starts form 0 to size of array minus one.

Similarly, char name [10]; Here name is a character variable of size 10, it can store 10 characters.

float salary [20]; here salary is variable name having float data type array of size is 20. It can

store 20 factorial values.

5.3. Initialization of Arrays

5.3.1. Initialization of 1 D array

If array elements are not given any specific values, they are supposed to contain garbage

values. The values can be assigned to element at time of array declaration, which is called

array initialization. Since an array his multiple elements, braces are used to denote the entire array and commas are used separate the individual values assigned to the elements

in the array initialization statements.

Syntax: storage class data types array name [size] = {value1, value2, value3};

Where value1 is first elements value 2 is second element and value3 is last element. This

array size is 2.

The Array Initialization May be Three Types:

1. int a [5] = {1, 2, 3, 4, 5};

Here "a" is integer type array which holds 5 elements. Their values are assigned as 1,2,3,4,5(a[0]=1,a[1]=2,, a[2]=3,a[3]=4 ,a[4]=5).The array elements are stored sequentially in memory location.

2. int b [] = {1, 2, 3};
Here size of array is not given; the compiler automatically sets its size accordingly to number of values present in braces. The size of array is 2 in this array initialization as three values 1,2,3 are assigned at time of initialization (a[0]=1,a[1]=2,a[2]=3).
3. int a [10] = {1, 2, 3, 4, 5};
In this example the size of array has set 10 but only five elements are assigned at the time of initialization. In this situation all individual elements that are not assigned explicitly contains 0 as initialization values, as
a[0]=1,a[1]=2,a[2]=3,a[3]=4,a[4]=5,a[5]=0,a[6]=0,a[7]=0,a[8]=0,a[9]=0.

5.4. Accessing Arrays Elements

The single operation that could not include entire array is not permitted in C. Thus if num
and list are two similar array (data type and dimension), assignment and comparison operation of those two arrays must be carried out on the basic of element by element operation. All elements of an array can neither be set at once or one array is assigned to
another.
e.g. int a [5], b [5]; then a=0; b=a; //wrong if (a<b) {…} //wrong
The particular array elements in an array are accessed by specifying the name of array, followed by sequence bracket enclosing an integer, which is called array index. The array
index indicates particular elements of array which we want to access. The number of index
starts form zero and ends to number one less than the size of array.
For example: float marks [5];
Then array elements are marks[0], marks[1], marks[2], marks[3] & marks[4]. We can assign
float value to these individual elements directly or a loop can be used to input and output
from the elements of array location.
Asking for user values in loop for each elements:

```
for(int i=0;i<5;i++)
{
printf("Enter the value for marks[%d]:",i)
scanf("%f", &i);
}
```

5.5. Characteristics of Array

The deceleration int a[5] is nothing but creation of 5 variables of integer type in memory
.
Instead of declaring five variables for five values, the programmer can define them in array
of any order.
• All elements of an array share same name and they are distinguished form one another with the help of array index.
• The element number or array index in the array plays an important role for calling each element.

• Any particular element of an array can be modified separately without disturbing other elements.
• The single operations, which involve entire array, are not permitted in C. Thus if num and list are two similar array(same data type dimensions and size), then assignment operations comparison operators etc, involving these tow arrays must be carried out on comparison through element by element basic.
• Any element of an array can be assigned to another ordinary variable or array variable. int b, a[10], c=9; then b=a[2];a[5]=a[3];a[6]=c; are valid conditions.

5.6. Multidimensional Arrays

Multidimensional array are those which have more than one dimension. Multidimensional
arrays are defined as same manner as one dimensional array, except that a separate pair of
square brackets is required for each subscript or dimensional or index.
Thus two dimensional array will require two pairs of square brackets, similarly, the three dimensional arrays will require three pairs of square brackets and so on.
The two dimensional array is also called matrix. Multidimensional array can be defined as:

Syntax: storage class data type array name [dim1] [dim2];

Here dim1 and dim2 are positive value integer expression that indices the numbers of array
elements associated with each sub-script. An m*n two dimensional array can be tabular form of value having m rows and columns. For int p [3][3] ;

col1 col2 col3
row1 p[0][0] p[0][1] p[0][2]
row2 p[1][0] p[1][1] p[1][2]
row3 p[2][0] p[2][1] p[2][2]

5.6.1. Declaration of Two Dimensional Arrays

Like one dimensional array two dimensional arrays must also be declared before using it.

Syntax: storage class data type array name [row size][column size]

For int matrix [2][3]; It contains 2 rows and 3 columns it contains 6 elements.

5.6.2. Initialization of 2 D Array

Like one dimensional array multidimensional array can be initialized at the time of array defend or declaration. A few example of 2-D array initialization are:

int marks[2][3]={2,4,6},{8,10,12};

is equivalent to

• ***mark[0][0]=2***
• ***marks[0][1]= 4***
• ***mark[0][2]=6***
• ***marks[1][0]= 8***
• ***mark[1][1]=10***
• ***marks[1][2]= 12***

&

int marks[2][3]={2,4,6,8,10,12};

The first dimension may be empty like this.

int mark3[][3]= {2,4,6,8,10,12};
int p[][3]={2,4,6,8,10};
p[0][0]=2,p[0][1]=4,p[0][2]=6, p[1][0]=8,p[1][1]=10,p[1][2]=0
Note: if value provided while initializing are less than an array, zero is assigned to remaining elements.
All above example are valid while initializing array, it is necessary to mention second dimensional column where as first dimension row is optional. This following are invalid initialization:
int marks[3][]={2,4,6,10,12};//wrong
int marks3[][]={2,4,6,8,10};//error

5.6.3. Accessing 2d Array Elements

In 2D array first dimension specifies number of rows and second specifies columns. Each
row contains elements of many columns. 2D array contains multiple rows.
Thus 2D array is an array of 1D array. As each size row will contain elements of many columns, the columns index ranges from 0 to size-1 inside every row in one dimensional representation. So columns index changes faster than row index as one dimensional representation of array inside computer is traversed.
2D array is traversed row by row (every columns elements in first row are traversed first and
then columns elements of second row are traversed and so on. The nested loop is used to
traverse the 2D array.
Let's consider following 2D array marks, size of 4*3 matrix having 4 row and 3 columns.
35 10 11
34 90 76
13 8 5
76 4 1
i.e.: 35 10 11 34 90 76 13 8 5 76 4 1
To access a particular elements of 2D array we have to specify array name followed by two
square brackets with row and columns number inside it.
For example: marks[0][0]=35, marks[0][1]=10

5.7. Strings

Strings are array of characters. They are arranged one after another in memory. Thus character array is also known as string.
Each character within string will be stored within each element of the array. Strings are used by “%c” to manipulate text such as word and sentences. A string is always terminated
by a null character “\0”. The terminating null character is important because it is only way
string handling functions can known where string ends. Normally each character is stored in
one byte, and successive characters of the string are stored in successive bytes.
Initialization of Strings:
char name[]={‘R’,’A’,’M’};

Then string name is initialized to “RAM”. This technique is also valid but offers special way
to initialize the sting as:

char name[]={“RAM”};

The characters of string are enclosed within the double quotes.

Arrays of Strings:

String is array of characters. Thus an array of string is 2D array of characters. This is explained with the help of following example:

/* program to read name of 5 different person using array of strings & display them. */

```
#include<stdio.h>
#include<conio.h>
void main()
{
char name[5][10]; int i;
printf("enter name of 5 students");
for(i=0;i<5;i++)
scanf("%s",name[i]);
printf("The enter name are");
for(i=0;i<5;i++)
printf("\nThe%d name is%s",i,name[i] );
getch();
}
```

5.7. String Handling Functions

The library built in functions strlen (), strcpy (), strcat (), strcmp () strrev () etc are used for
string manipulation in C. These functions are defined in header file string.h

5.7.1. strlen ()

The strlen () function returns integer which denotes length of string passed. The length of
string is number of characters presented in it, excluding terminating null character.

Syntax: integer variable=strlen (string);

WAP to find the length of Input from user using function.

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
char name[10];
int len;
clrscr();
printf("\n Enter Your Name");
gets(name);
len=strlen(name);
printf("\n the number of character in our yname is%d",len);
getch();
}
```

5.7.2. strcpy()

The strcpy () function copies one string to another. The function accepts two strings as parameters. The first string copies the string of second string character by character into

first one up to and including the null character of second string.

Syntax: strcpy (destination string, sources string);

WAP to copy one string to another

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
char name[]="Hello World";
char s[25];
clrscr();
strcpy(s,name) ;
printf("\n the Value s=%s\t name =%s",s,name);
getch();
}
```

5.7.3. strcat ()

This function concatenates two string i.e it appends one string at the end of another. This
function accepts two strings as parameter and stores the contents of the second string at
end of the first string.

Syntax: strcat (string1, string2);

WAP to concatenate two strings

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
char fname[20]="Hello ", lname[10]="World";
clrscr();
strcat(fname,lname) ;
printf("\n full name is%s",fname);
getch();
}
```

5.7.4. strcmp()

This function compares two strings to find out whether they are same or different. This function is useful for searching string as arranged in a dictionary order. This function accepts two strings as parameters and returns an integer whose value is ,

- Less than 0 if first string is less than second string
- Equal to 0 if both are same
- Greater than 0 if the first string is greater than second

Two strings are compared character by character until there is a mismatch or end of strings.

When two characters in two strings differ, the string which has the character with a higher
ASCII values is displayed.

Syntax: strcmp(string1, string2);

WAP to illustrate to use of strcmp () function

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

```
void main()
{
char fname[20],lname[10];
int diff;
printf("enter first name"); gets(fname);
printf("Enter second naame"); gets(lname);
diff=strcmp(fname,lname);
if(diff<0)
printf("%s is greater than %s by value of %d",fname,lname,diff);
else if(diff<0)
printf("%s is less than %s by value of %d",lname,fname,diff);
else
printf("%s is same length %s ",fname,lname);
getch();
}
```

5.7.5. strrev()

This function is used to reverse all characters in string except null character at the end of
string. The reverse of string “abc” is “cba”.

Syntax: strrev(str1);

WAP to illustrate strrev function

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
char name1[15]="Hello World", name2[15];
strcpy(name2,name1);
strrev(name1);
printf("The reversed string of original string %s is%s",name2,name1);
getch();
}
```

6. FUNCTION

6.1. What is Function

A function is defined as a self contained block of statements that performs particular task/jobs. This is a
logical unit composed of a number of statements grouped into a single unit. It is defined as a section of
a program performing a specific task. The function main is always present in each program which is
executed first and other function are optional.
The separate program segment is called functional that is called main program when ever required. For
example to calculate prime numbers we should have to calculate each number generated form loop
should have to test to be prime or not, in such case we have to repeat same logic of finding prime
numbers up to defined times. Instead of such case we have to make general function which calculates
factorial of a given number and this function is called function which calculates prime of a given
numbers by passing them to the function to be tested. Therefore function is a special code block that
has some functionality . it is called inside main methods when it required.

6.1.1. Advantage of Functions:

Manageability: Using function for specific job, it is easier to write programs and keep track of what
they are doing. It makes programs significantly easier to understand and maintain by breaking them
into easily manageable sub units. The code for each job or activity is placed in individual functions
such that testing and debugging because easy and efficient. Thus use of functions in a program helps to
manage the code.
Code Re-usability: A single function can be used multiple times by a single program form different
places or different programs. It avoid re writing same code again that recuce large efforts and reuse
code of computer designer. Thus it helps to reuse code once written and tested. The C standard library
is an example of functions being reused.
Avoid Redundant: While using function same function is called whenever needed inside main
program. Thus particular job or activity is to be accessed repeatedly form several different place within
or outside the program. The repeated activity can be placed within a single function, which can then be

accessed whenever it is needed. If the function is not used in such situation the code must have to
redesign for same activity is to be rewritten the same instruction again and again.
Logical Clarity: When single program is decomposed into various well-defined functions, the main
program consists of a series of function calls rather than rewrite lines of code so that size of main
program seems smaller and becomes logical clarity understandable.
Easy to Divide the Work Among Different Programmers: Different programmers working at large
project can divide the working by writing different functions easily.

6.2. User Defined Function and Library Functions

Library Functions (Built in Functions): There are some functions which are already written
compiled and placed in C library. They are not required to be re-written by a developer. The function
name its return type and their arguments number and types have been already defined. We can use these
functions whenever we are required them. For example printf (), scanf (), sqrt(), getch() are library
functions generally used.
User Defined Function: These are functions which are defined by user at time of writing a program.
The user has choice to choose its name, return type, arguments and their types. In case of above
example the function factorial () is user defined function. The job of each user-defined function is as
defined by user. A complex program can be divided into a number of user defined functions.
About main () Function: The function main is a user defined function except that name of function is
defined or fixed by the language. The return type arguments and body of function are defined by
programmer as required. This function is executed first, when the program starts execution.
Function Definition: The collection of program statement that describes specific task to be done by
function is called a function definition. It consists of function header which defines functions name its
return type and arguments list and body of function which are the block of code enclosed in
parenthesis.
Syntax:
return type function name (data type variable name, data type variable name)
{
….;

….;
}
The first line of function definition is known as function declaration or header. This is followed by
function which is composed of statement that makes up function, delimited by braces.
float areaofcircle(float radius){ return(3.14*radius*radius);}
void display(int a){ printf(“%d”,a);}
The return type is optional the default is int type. The arguments variable 1, variable 2, variable 3
presents in function definition are called arguments or parameters because they represent name of data
items that are transferred into function from calling function/program. There are local variable for
function.
Function Declaration/Prototype: The function declaration or prototype is model or blueprint of
function. If function are used before they are defined, then function declaration or prototype necessary
which provides following information to computer.
• Name of function and type of value returned by function
• The number and type of arguments
• When a user defined function is defined before used
Syntax:
return type function name (arguments, arguments);
Here return type specifies data types of value returned by function. A function can return value of any
data type. If there is not return value, the keyword void is used. The function deceleration and in header
function must use the same function name, numbers of arguments and its type and returned types.
Note: It is important to note that the function prototype has semicolon at the end and to specify the
same name of formal arguments is not mandatory but type is mandatory.
Return Statement : It is statement that is executive just before function completes its job and control
is transferred back to the calling function. The job of return statement is to hand over some value given
form where the call was called. The return statement serves mainly two purposes. They are:
• It immediately transfers control back to the calling program after executed after the return
statement i.e. no statement within function body is executed after return statement.
• It returns the value to calling function.
Syntax:
return (expression);

When expression must evaluated to value of type specified in function header from the return value.
The expression can be any desired expression as long as it ends up with value of required type. The
value of expression is returned is omitted, return statement simply cause control to revert back to
calling portion of return statement each containing a different expression, but a function can return only
one value to calling portion of the program via return statement.
Function may or may not return any value. If a function doesn't return a value type in the function
definition and deceleration is specified as void. Otherwise return type is specified as a valid data type.
A return statement at end is optional for function without return value.
Accessing Function: A function can be called or accessed by specifying its name, followed by a list of
arguments enclosed in parentheses must separate by comma. For example, function add () with two
arguments is called by add (int a, int b) to add two numbers. If function call does not required any
argument it has empty pair of parenthesis must followed the name of function. The arguments
appearing in function call are referred as actual argument. In function call there will be one argument
for each formal argument. The value of each argument is transferred into function and assignment to
corresponding formal argument. If functions a value returned value can be assigned to a variable of
type same as return type of function. When a function is called the program controls is passed to
function. Once the function completes its task, program control is passed back to the calling function.
The general form of function call statement is:
• If function has parenthesis but it does not return value
• If function has no arguments and it does not return value
• If function has parameters and it returns value
• If function has do not parameters but returns value
The following points to be taken into consideration wish function calls statement are used:
• The function name type & number of variable listed in function statements must that of function deceleration statement and the header of function definition.
• The names of the variables in the function deceleration function call statement and that in the
header of definition may be different.
• By default arguments are always passed by value in C function call. This means that local

copies of the values of arguments are passed function.
• Arguments presents in the form of expression are evaluated and converted to the type of formal
parameters at the beginning body of the function.
• A variable may be assigned value returned by function after it is executed using function call
statement, provided return type where the function is not void type.

Function Parameters: Function parameters are means for communication between calling and called
functions. They can be classified into formal parameters and actual parameters. The formal parameters
are the parameters given in the function declaration and function definition, the actual arguments or
permanents are specified in the function are called. In above example a and b are actual arguments
which are used in calling unction while parameters of x and y are formal arguments which are defined
in function definition. The name arguments or parameters which must be same in function definition.
The name of formal and actual arguments need not be same but data type and numbers of them must be
matched.

6.2.1 Category of Function According to Return Value and Arguments

According to the arguments and return type present in function definition and design can categorized
function in three type:
• Function with no arguments and no return value
• Function with arguments and no return type
• The function with arguments and return values

Function with on Arguments and No Return Values: When a function has no arguments it does not
receive any data from calling function similarly it does not return a value. The function do not receive
any data from the calling function. Thus in such function there is not data transfer between the calling
function and called function.
This type of function is defined as: **void function name () { }**
They keyword void means the function does not return any values there is not arguments within the
parenthesis which implies function has no argument and it does not receive any data for the called
function.

WAP to illustrate function with no arguments and no return value

```
#include<stdio.h>
#include<conio.h>
void addition()
{
```

```
int a,b,sum;
printf("Enter two numbers"); scanf("%d%d",&a,&b);
sum=a+b;
printf("the sum is%d",sum);
}
void main()
{
clrscr();
addition();
getch();
}
```

Function with Arguments but has Arguments but No Return Value: The function has argument and reveries the data from calling function. The function completes task and does not return any values to calling function.

Such type of functions are defined as: **void function (arguments) { }**

WAP to illustrate the functions with arguments but no return type

```
#include<stdio.h>
#include<conio.h>
void addition(int a,int b)
{
int sum;
sum=a+b;
printf("the sum is%d",sum);
}
void main()
{
int a,b;
clrscr();
printf("Enter two numbers");
scanf("%d%d",&a,&b);
addition(a,b);
getch();
}
```

Function with Arguments and Return Value: The function of this category has arguments and reveries data from calling function. After completing its task, it returns results to calling function through return statement. Thus there is data transfer between called function and calling function using return values and arguments.

These types of functions are defined as: **return_types function_name (arguments, ..., ...) { body}**

WAP to illustrate the function with arguments and return values.

```
#include<stdio.h>
#include<conio.h>
int addition(int a,int b)
{
int sum;
sum=a+b;
return sum;
```

```
}
void main()
{
int a,b;
clrscr();
printf("Enter two numbers"); scanf("%d%d",&a,&b);
printf("the sum is%d",addition(a,b));
getch();
}
```

6.2.2. Different Type of Function Calls

The arguments in function can be passed in two ways:

• pass arguments by value

• pass arguments by address or references for pointer

Function Call by Value (Pass Arguments by Value)

When values of actual arguments are passed to function as argument it is known as function call by value. In this call, value of each actual argument is copied into corresponding formal arguments of the function definition. The content of arguments int calling functions are not altered even if they are changed into called function.

WAP to illustrate function call by value

```
#include<stdio.h>
#include<conio.h>
void swap(int,int);
void main()
{
int a,b;
clrscr();
a=90;b=89;
printf("\nbefroe swap a and b are %d and %d",a,b);
swap(a,b);
printf("\nafter swap are %d and %d",a,b);
getch();
}
void swap(int x,int y)
{
int temp;
temp=x;
x=y;
y=temp;
printf ("\nthe value within function %d and %d",x,y);
}
```

In this program the value a and b are copy to void swap(int x , int y) then calculates inside function only, when again original value s are not changed in the main program so this process is similar to pass argument by value.

Function Call by Reference (Pass Arguments by Address)

In this type of function call, address of variable or arguments is passed to the function as arguments

instead of actual value of variable.
Pointer: A pointer is a variable that store a memory address of a variable. Pointer can be any name that
is legal for other variable and it is declared in same fashion like other variable but it always proceeded
by*(asterisk) operator. Thus pointer variables are defined as: int *s; float *b; char*c;
Here int *a, int *b, int *c is pointer variables which stores address of another integer, float and
character variables.
Thus the following are valid:
• a=&p;
• b=&y;
• c=&y;

WAP to swap the values of two variables using call by reference

```
#include<stdio.h>
#include<conio.h>
void swap(int*,int*);
void main()
{
int a,b;
clrscr();
a=99;
b=9;
printf("\n before swap a and b are %d and %d",a,b);
swap(&a,&b);
printf("\n after swap are %d and %d",a,b);
getch();
}
void swap(int *x,int *y)
{
int temp;
temp=*x;
*x=*y;
*y=temp;
}
```

6.3. Recursive Functions

Recursion is a technique for defining a problem in terms of one or more smaller sub unit of same
problem. The solution of problem is built on result from the simpler versions. The recursive function
is one that calls itself directly or indirectly to solve a smaller version of its task until a final call which
does not require a self call. Thus function is called recursive function if it calls to itself. Recursion is a
process by which a function calls itself repeatedly until some specified condition will be satisfied. Thus
process is used for repetitive calculation in which each action is stated in term of previous result. Many
iterative or repetitive problems can be written in this form.
• To solve problem using recursion methods two conditions must be satisfied therefore

• Problems could be written or defined in term of its previous result.
• Problems statement must include a stopping condition. We must have an if statement somewhere to force the function to return without the recursive call being executed otherwise
the function will never return.

WAP to find the factorial of number using recursive method

```
#include<stdio.h>
#include<conio.h>
int factorial(int n)
{
if(n==1)
return(1);
else
return(n*factorial(n-1));
}
void main()
{
int num; clrscr();
printf("Enter a number"); scanf("%d",&num);
printf("the factorial is%d", factorial (num));
getch();
}
```

Recursive Iteration

A function is called for definition of same function to do repeated task

Loop is used to do repetitive action.

Recursion is the top down approach to problems solving it divides the problem int pieces.

Iteration is like a bottom up approach; it begins with what is known and form this if constructs the solution step by step procedure.

In recursion a function calls to itself until some condition will be satisfied.

Iteration a function does not call to itself

Problem could be written or defined in term of its previous result to solve a problem.

It is not necessary to define a problem in terms of its previous result to solve in iteration

All problems cannot be solved using recursion All problems cannot be solved using iteration

6.4. Concept of Local, Global, Static & Register Variables

6.4.1. Local /Automatic Variables

The automatic variables are always declared within a function/block. They are local to the particular
function/block in which they are declared. As local variables are defined within body of function/block,
other functions cannot access the variables. The compiler shows errors in case other function cannot

access the variables. The local variables are created when function is called and destroyed
automatically when the function exit. The keyword auto may be used for storage class specification
while deceleration of variable. A variable declared inside a function without storage class specification
auto is by default, an automatic local variable.
Default initial value of local variable is unpredictable value which is often called garbage value. The
scope of it is local to the block in which variable is defined. Again its life is till the control remains
within in which the variable is declared. The system supply the garbage value to it.
Initial Value: The garbage value assigned to a variable if no value is assigned to local variable.
Scope: Scope of variable can be defined as region open which variables are visible.
Lifetime: The period of variable availability function on time duration which memory is associated
with as variable. Its life time as available only inside function block only.

WAP to illustrate local variable.

```
#include<stdio.h>
#include<conio.h>
int fact(int n)
{
int f=1; int i;
for(i=1;i<=n;i++)
f*=i;
return (f);
}
void main()
{
int num=5;
clrscr();
printf("the factorial of 5 is%d",fact(num));
getch();
}
```

6.4.2. Global / External Variable

Variable that are both are available and active thorough entire program are known as global variable.
They are also known external variable. The external variable or global variable are declared outside the
block of functions. Unlike local variables, global variables can be accessed by any functions in the
program. The default initial variables of variables are zero. The scope of variable is within whole
program. The life times are as long as the program's execution.
Initial Value: ZERO
Scope: Throughout the program.
Lifetime: As long as program's execution.

WAP to illustrate global variables

```
#include<stdio.h>
#include<conio.h>
void fun();
int a=10;
void main()
{
printf("\t%d",a); fun();
printf("\t%d",a);
getch();
}
void fun()
{
a=20;
printf("\t%d",a++);
}
```

6.4.3. Static Variable

This is another class of local variable. A static variable can only be accessed from function in which it
was declared, like a local variable. The static variable is not destroyed on exit from function
termination instead its value is preserved and becomes available again when the function is next called.
Static variables are declared as local variables, but declaration is preceded by word static. Static
variable can be initialized as normal; the initialization is done once only when program starts. The
default initial value for this type of variables is zero (if user doesn't initialize it). The scope is similar
local variable is only available to block in which variable is defined. But life time is global

Initial Value: ZERO.
Scope: Similar to Local Variables.
Lifetime: Similar to Global Variables.

WAP to illustrate static variable.

```
#include<stdio.h>
#include<conio.h>
void staticc(void);
void main()
{
staticc();
staticc();
staticc();
staticc();
getch();
}
void staticc(void)
{
static int i=1;
printf("\n The value is %d",i);
i++;
}
```

Output:
The value is 1

```
The value is 2
The value is 3
The value is 4
--------------------------------------------------------------------------
#include<stdio.h>
#include<conio.h>
void staticc();
void main()
{
staticc();
staticc();
staticc();
getch();
}
void staticc()
{
int i=1;
printf("\nthe value of %d",i); i++;
}
Output
The value of 1
The value of 1
The value of 1
The value of 1
```

6.4.4. Register Variable

Register variables are special type of automatic variables. Automatic variables are allocated storage in memory of computer however for most computers accessing data in memory is considerably slower than processing in the CPU. Those computers often having small amount of storage within CPU itself.

Where data can be stored and accessed quickly. Theses storage calls are called register. Normally the computer determines what is to be stored in register of CPU at what times .However C language provides the storage class register so that the programmer can suggest to the compiler that particular automatic variables should be allocated to CPU register. Thus register variable provide a certain control over efficient of program execution is faster accessed. Variables which are used repeatedly and have good access times are critical may be declared to register.

Register variables behaves way just like automatic variables. The storage block and storage is freed when block is exited. The scope of register variable is local to block where they are declared. Rule for initialization for register variables are same as for automatic variables.

6.4.5. Difference between Local, Global and Static variables

Local Global Static

The variables are declared

within function /block. the
scope is only within the
function in which they are
defined
Floral variables are defined outside
the function so that their scope is
through the program
Static variables are
special case of local
variables thus static
variables are also
defined inside the
functions or block
The initial value is
unpredictable or garbage
value
The initial value is zero The initial value is
zero
The lifetime is till the control
remains within the block or
function in which the variable
is defined. The variable is
destroyed when function
returns to the calling
functions or block ends
The lifetime is till programs
execution does not come to an end
variables are created when n
program starts and destroyed when
program ends
The value persists
between different
function calls. Thus
lifetime is same as
global variables
The keyword “auto” is used The keywords “extern” is used The keyword
“static” is used.

6.5. Passing Pointer to a Function

A pointer can be passed to a function as an argument. Passing a pointer means passing address of
variable instead of value to a function. As address is passed in this case, this mechanism is also known
as call by address or call by reference. When pointer is passed to a function the formal argument of the
function must be compatible with the passing pointer i.e. if integer is being passed, the formal

argument in function must be pointer of type. As address of variable is passed in mechanism, if value in
passed to address is changed within function definition, the value of actual variable also changed.

WAP to illustrate the use of passing pointer to a function

```
#include<stdio.h>
#include<conio.h>
void add(int *m)
{
*m=*m+10;
}
void main()
{
int marks;
clrscr();
printf("Enter actual marks\t");
scanf("%d",&marks);
add(&marks);
printf("\n the grace marks is\t%d",marks);
getch();
}
```

6.6. Passing Arrays to Function

Like ordinary variables it is possible to pass value of an array elements i.e. entire array argument to a
function. To pass entire array to a function array name must appear by itself, without brackets or
subscript, an actual argument in function call statement. The corresponding formal argument in
function definition is written in same manner. When declaring a one dimensional array as a formal
argument, the array name is written with a pair of empty square brackets. The size of array is not
specified within pair of empty square brackets. The size of array is not specified within the formal
argument deceleration.

Syntax:

function_name (array name); //for function call.
return_type function_name (data_type array_name) //for function prototype;
return_type function_name (data_type* pointer_variable);

When an array is passed to function the value of array elements are not passed to the function. Rather
the array name is interpreted as the address of element. This address is assigned to the corresponding
formal argument when the function is called. The formal argument therefore becomes a pointer to first
array element. Thus array is passed using call by reference.

7. POINTER

Memory is important area where data/information are stored. As computers have primary memory also known as RAM (Random Access Memory). RAM holds currently running program along with data variables. All variables used in a program reside in memory when the program is executed. Computer memory RAM is divided into various of small units locations. Each location is represented by some unique number known as memory address.

Each memory location is represented by some unique number which is known as byte. A char data is one byte in size and hence needs one memory location, similarly integer data is two bytes in size so it requires two memory locations. The smaller unit of memory address is byte(0 or 1).

Each byte in memory is accessed with unique address. An address is an integer labeling bytes in memory. The integer is always positive values that range from zero to some positive integer constant corresponding to location in memory. Thus a computer having 1 GB has 1024*1024*1024 i.e. 1073741824 bytes. The memory address 65524 represents bytes in memory and it can store data of one byte.

Each variable in C program is assigned space in memory. When a variable is declared, it tells computer to hold in memory in appropriate size variable. According to type of variable declared, the required memory locations are reserved. int a=30; float b; char c. here a reserve 2 bytes similarly char c holds 1 bytes.

WAP to display memory location reserved by variables

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a; a=20;
printf("\n the address of a is:\t%u",&a);
printf("\n the value of a is: \t%d",a);
getch();
}
```

Here &a denote address of variable. The variables takes two bytes data in memory. As it's of type is int. It takes two memory area locations, although it takes a block of two bytes memory consisting two different addresses, this memory is accessed by first address. The first address is only shown which known as base address. Again control string %u is used to display address of variables as value of address is always positive number.

WAP to illustrate address reserved by different data types

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a=10,b=20;float c=10.5; char d='M';
clrscr();
printf("\n The base address of a\t%u",&a);
printf("\n The base address of b\t%u",&b);
printf("\n The base address of c\t%u",&c);
printf("\n The base address of d\t%u",&d);
getch();
}
```

Output:

The base address of a 1045064
The base address of b 1045060
The base address of c 1045056
The base address of d 1045055

7.1. Introduction to Pointer

A pointer is a variable that contains a memory address of another variable. Normally a pointer variable is declared some way like other variables in C. So that it will work only with
address of another variable. Just as integer variable can hold only integer, a character variable can hold only character type, each pointer variables can point only to one specific
type (int ,float, char).
Pointer can have name that is legal for other variable and it is declared in some fashion like
other variable but while declaration we preceded by * (asterisk/indirection operator) in front of variable name.
For Eg:

- ***int num;*** Here num is normal variable of integer it can store only integer value.
- ***int *p;*** It signifies p is pointer variable it can store address of integer variable.
- The address of float variable cannot be stored in p. Eg: ***int *p; float num; p=&num*** is not allowed

7.1.1. Pointer Declaration

Pointer variable can be declared as:
data type *variable name;
For Eg:
int *s; char *p;
The first s is an integer pointer and it tells compiler that holds address of any int variables.
In same way char pointer that store address of any char variable.

WAP to demonstrate pointer variables

```
#include<stdio.h>
#include<conio.h>
void main()
{
```

```
int v=10,*p; //*p is pointer variable
p=&v; //the address of variable v is assigned to p
printf("\n address of v is\t%u",&v); // display address of v directly
printf("\n the address v is\t%u",p); // display address of v indirectly
printf("\n value of v is\t%d",v); // display value directly
printf("\n value of address v is \t%d",*p); //display value of p indirectly
printf("\n address of p=%u",&p); // display address of *p
getch();
}
```

7.1.2. Indirection or De-reference Operator

The * operator is used inform of a variable, is called pointer or indirection or de-reference operator. Normal variable provides direct access to their own values where as a pointer provides indirect access to variables. The indirection operator (*) is used in two distinct ways within pointers deceleration. When the pointer is de-referenced in form of variable the indirection operator indicates the values at the memory location stored in pointer. The compiler knows automatically which operator to call based on the context of is used.

7.1.3. Address Operator

Ampersand (&) operator is known as address operator thus “&a” denotes address of normal to pointer variable.

7.1.4. Initialization Pointers

Address of some variable can be assigned to a pointer variable at the time of declaration of pointer variable. ***int num; int * ptr=#*** These two statements are equivalent to following statements. ***int num; int *p;p=#***

7.1.5. Bad Pointer

When a pointer variable is first declared it does not have a valid address, the pointer is uninitialized or bad. The de-reference operation on a bad pointer is a serious run time error.

Each pointer must have be assigned a valid address before it can support de-reference operators. Before that the pointer is bad and must not be used.

In fact every pointer contains garbage value before assignment of some valid address. Correct code override the garbage value with a correct reference to an address and thereafter the pointer work fine. There is nothing automatic that gives a pointer valid address.

7.1.6. Void Pointer

A void pointer is special type of pointer. It can point to any data type from an integer float or string of characters. Using void pointer the pointer data can be referenced directly. Type casting or assignment must be used to change void pointer to concrete data type to which we can refer.

WAP to illustrate the void pointer

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a=40; double b=4.5;
void *v;
clrscr();
v=&a;
printf("\n a=%d",*(int*)v);/* not simple*/
v=&b;
printf("\nb=%lf",*((double*)v));
getch();
}
```

7.1.7. Null Pointer

A null pointer is a special pointer value that points nowhere or nothing.
To test a pointer for “NULL” before inspect value pointed to code such as the following can
be used.

If(ptr!=NULL) printf(“value =%d”,*ptr);

7.2. Pointer to Pointer (Double Pointer)

C allows use of pointer that point to other pointer and theses in turn, point to data for pointer to do that we only need to add asterisk for each level of reference.

```
int a=30;
int *p;
int **q;
p=&a;
q=&p;
```

**q is double pointer it requires double pointing to refer to variable address.
Double pointer variable requires two label of indirection while assigning the value we use
double address to q variable. As both *p and **q display 30 if they are pointing to the same
address.

7.2.1. Arrays of Pointers

An array of pointers can be declared as

data type *pointer (size); int *p [5];

This declares an array of 5 elements each of which points to an integer, the first pointer is
called p [0];
Initially these pointers are uninitialized and they can be used as int a=10,b=100,c=1000;
p[0]=&a; p[1]=&a; p[2]=&c;

7.2.2. Relationship Between 1d Array and Pointer

Array name by itself is an address of pointer. It points to address of first element.
Thus if x is one dimensional array then address of first array element can be expressed as
either &x [0] or simple x [0]. In general the address of the array element i+1 can be expressed as either &x+i. In the expression x+i, x represents array name whose elements

may be integers, characters, float, etc, and I represents integer quantity.
Thus x+i specifies address that is a certain number of memory calls beyond the address of
first element. Thus x[i] and *(x+i) both represent same content of that address.

WAP to display elements with their address using name as pointer.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int x[5]={20,40,6,8,100},k;
clrscr();
printf("\n aray elements\t elements \t address");
for(k=0;k<5;k++)
printf("\ns[%d]\t%d\t%u",k,*(x+k),x+k);
getch();
}
```

Similarly,

```
#include<stdio.h>
#include<conio.h>
void main()
{
int x[5]={20,40,6,8,100},k;
clrscr();
printf("\n aray elements\t elements\t address");
for(k=0;k<5;k++)
printf("\ns[%d]\t%d\t%u",k,x[k],&x[k]);
getch();
}
```

From the above it can be concluded that in case of array &x[k] is same as x+k and x[k] is
same as *(x+k).

7.2.3. One Dimensional Array

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a[5]={1,2,3,4,5};
int i;
for(i=0;i<5;i++)
{
printf("add is\t%d\tand \t value \t%d\n",&a[i],a[i]);/*for Normal
printf("add is\t%d\tand \t value \t%d\n",(a+i),*(a+i));/*for pointer type
printf("\n");
}
getch();
}
```

7.2.4. Pointer and 2d Arrays

Multidimensional array can also be represented with an equivalent pointer single dimensional array. A two dimensional array is actual a collecting of one dimensional arrays,
each indicating a row. It is stored in memory of row form.

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

```
#include<conio.h>
void main()
{
int a[3][3]={{1,2,3},{4,5,6},{7,8,9}};
int *b[3][3],i,j;
clrscr();
for(i=0;i<3;i++)
{
for(j=0;j<3;j++)
{
b[i][j]=&a[i][j];
printf("The value is%d and addressed\n",*b[i][j],&*b[i][j]);
}
}
getch();
}
```

It is possible to access two-dimensional array elements using pointers same way in one dimensional array. Each row of the two dimensional array is treated as a one dimensional
array. Each row of two dimensional array as a pointer to a group of contiguous one dimensional array.

Syntax: ***data type (*pt variable) [size2]; instated data type array [size1] [size2];***

Suppose x is two dimensional integer array having 4 row and 5 columns. We can declare x as
int (*x) [4];

Rather than int x [4] [5]; it can be concluded that x pointer to first row.

Here in first declaration x is defined to be a pointer to a group of continuous one dimensional element integer array. The x points to first 5 elements array which is actual first row of two dimensional arrays. Similarly x+1 points to the second row 5 elements which
is second row of the two dimensional array.

It can be illustrated as:

*x ([0][0]) *x+1 ([0][1]) *x+2 ([0][2]) *x+3 ([0][3]) *x+4 ([0][4])
*(x+1) ([1][0]) *(x+1)+1 ([1]
[1])
*(x+1)+2 ([1]
[2])
*(x+1)+3 ([1]
[3])
*(x+1)+4 ([1]
[4])
*(x+2) ([2][0]) *(x+2)+2 ([2]
[1])
*(x+2)+2 ([2]
[2])
*(x+2)+3 ([2]
[3])
*(x+2)+4 ([2]
[4])

*(x+3) ([3][0]) *(x+3)+3 ([3]
[1])
*(x+3)+2 ([3]
[2])
*(x+3)+3 ([3]
[3])
*(x+3)+4 ([3]
[4])

Two Dimensional Array

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a[3][3]={{1,2,3},{4,5,6},{7,8,9}};
int i,j;
for(i=0;i<3;i++)
{
for(j=0;j<3;j++)
{
printf("add is\t%d\tand \t value \t%d\n",&a[i][j],a[i][j]);/*for normal
printf("add is\t%d\tand \t value \t%d\n",*((a+i)+j),*(*(a+i)+j));/*for
pointer
printf("\n");
}
}
getch();
}
```

Illustrate 2d array represents in memory.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int p[2][3]={{1,2,3},{4,5,6}};
clrscr();
printf("\n%u and%u",p,*(p+1));
printf("\n*p=%u\t*(p+1)%u",*p,*(p+1));
printf("\n(*(p+0)+1)%u\t*(*(p+1)+1)%u",*(p+0)+1,*(p+1)+1);
printf("\n(*(p+0)+1)%u\t*(*(p+1)+1)%u",*(*(p+0)+1),*(*(p+1)+1));
getch();
}
```

WAP to add to add two m*n matrices using pointer.

```
#include<stdio.h>
#include<conio.h>
#define m 2
#define n 3
void main()
{
int(*a)[n],(*b)[n],*(sum)[n],i,j;
clrscr();
printf("firtst matrix");
for(i=0;i<m;i++)
for(j=0;j<n;j++)
scanf("%d",*(a+i)+j);
printf("second matrix");
```

```
for(i=0;i<m;i++)
for(j=0;j<n;j++)
scanf("%d",*(b+i)+j);
printf("sum is");
for(i=0;i<m;i++)
{
for(j=0;j<n;j++)
{
*(*(sum+i)+j)=*(*(a+i)+j)+ *(*(b+i)+j);
printf("\t%d",*(*sum+i)+j);
}
printf("\n");
}
getch();
}
```

WAP to multiply matrices of order m*n and p*q using pointer

```
#include<stdio.h>
#include<conio.h>
#define m 2
#define n 3
#define p 3
#define q 2
void main()
{
int(*matrix1)[n],(*matrix2)[q],pr[m][q],i,j,k;
int sum=0;
clrscr();
printf("firtst matrix");
for(i=0;i<m;i++) for(j=0;j<n;j++)
scanf("%d",(*(matrix1+i)+j));
printf("second matrix");
for(i=0;i<p;i++) for(j=0;j<q;j++) scanf("%d",*(matrix2+i)+j);
for(i=0;i<m;i++) for(j=0;j<p;j++) { for(k=0;k<n;k++)
sum+=*(*(matrix1+i)+k)**(*(matrix2+k)+j);
*(*(pr+i)+j)=sum;
sum=0;}
printf("\nproduct are\n:");
for(i=0;i<m;i++){ for(j=0;j<q;j++)
printf("\t%d",(*(*pr+i)+j));
printf("\n"); }
getch(); }
```

7.5. Pointer Operations

To illustrate operations let us consider following declaration of ordinary variable and pointer variables.

int a, b; float c; int *p1,*p2; float*f;

1. A pointer variable can be assigned the address of ordinary variable. For example p1=&a; p2=&b, f=&c;
2. Content of one pointer can be assigned to other pointer provide they point to same data type. p1=p2; where both of them are pointer type as above.
3. Integer data can be added to pointer variable, p1+2;
4. One pointer can be subtracted from other pointer they must point same elements of array

```
void main(){
```

int a[]={1,2,3,4,5},*pf,*p1;
p1=a; pf=a+2; printf(“%d”,pf-p1);
}
5. The two pointer variables cannot be compared both pointers point to objects of the same data type If(p1>p2){}//invalid
6. There is no sense in ascending an integer to pointer variable.
p1=100;p2=200; //invalid
7. Two pointer variables cannot be multiplied together.p1*p2;
8. A pointer variable cannot be multiplied by constant. p1*3;
9. A null value can be assigned to a pointer variable.(p1=NULL)

String and Pointer

As strings are arrays and arrays are closed connected with pointers, we can say that string and pointers are closely related for example: char name [5] =”Shyam”;
As string variable name is an array of 5 characters. It is a pointer to first character of string and can be used to access and manipulate characters of the string. When a pointer to char is pointed in format of string, it will start to print pointer character and then successive characters until end of string is reached.

Dynamic Memory Allocation (DMA)

The process of allocating and freeing memory at run time is known as Dynamic Memory Allocation. This reserves the memory required by the program and returns this valuable resources to the system then reserved space is utilized for variable dynamically.
Through arrays can be used for data storage, they are of fixed size. The programmer must know size of array or data in advance while writing programs. A variable cannot be used to define size of array while declaring an array. In most situations it is not possible to know the size for memory required until run time. For example the size of the array used to store marks of students is fixed. Suppose the size of array used to stored name in character is 100. The program wont work if the number of students is more than 100. If number of students is less than 100 say 10. The 10 memory locations are used and rest 90 locations are reserved but not used, this is wastage of memory occurs. In these situations DMA will very be useful.
Since an array name is actual pointer to first element within the array, it is possible to define array as pointer variable rather than general array. While defining conventional array system reserve fixed block of memory at the beginning of program design which is inefficient but this does not occur if the array is represented in terms of a pointer variable.

The use of a pointer variable to represent an array requires some type of initial memory assignment before array elements are processed. This is known as Dynamic Memory Allocation. At execution time a program the compiler request more memory to be free memory. Thus DMA refers allocating and freeing memory at execution time or run time for
better memory management.
There are four library functions malloc() calloc(),free() and realloc() for memory useful management. These functions are defined with header file stdlib.h and alloc.h.
malloc () : It allocates requested size of bytes and returns a pointer to first byte of the allocated space.
Syntax:
ptr=data type* malloc (size);
Here ptr is a pointer type data. The malloc () returns a pointer to an area of memory with size specified in size.
eg: s= (int*) malloc (100*size of (int));
A memory space equivalent to 100 times the size of a integer bytes is reserved and the address of the first bytes of allocated memory
calloc () : The function calloc () provides access to the C memory heap which is available for
dynamics allocation of variable sized block of memory. Unlike malloc (), function calloc ()
accepts two arguments no of block and size of each block. The first parameter specifies the
number of items to allocate and size of each block specifies size of each items. The function
calloc () allocates multiple blocks of storage each of same size and then sets all bytes to zero. The difference between them is that calloc initializes all bytes in the allocated block to
zero. Thus it is normally used for requesting memory space at run time for storing derived
data types such as and structure. s= (int*) calloc (5, 10*size of (int));
In above statement allocates continuous space for 5 blocks, each of size 20 bytes. We can
store 5 arrays each of 10 elements of integer types.
free (): This built-in function free up previously allocated space by calloc, malloc and realloc
functions. The memory dynamically allocated is not return to the system until the programmer returns the memories explicitly. This can be done using free() function. This
function is used to release the space when it is not required/reserved. free (ptr); Here ptr
is pointer to a memory block which must already created.
realloc (): This function is used to modify size of previously allocated memory space. sometimes previously allocated memory is not sufficient, we need additional space and sometimes the allocated memory is much larger than necessary , in both situations we should change the memory size already allocated with the help of function realloc().

This function allocates the new space size to the pointer variable ptr and returns a pointer
to the first byte of the new memory block of memory fails to allocate it return null.

WAP to illustrate the use of realloc() function

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
void main()
{
char*name;
clrscr();
name=(char*) malloc(11);
strcpy(name,"Happy Singh");
printf("\nname=%s",name);
name=(char*) realloc(name,23);
strcpy(name,"Mr. Happy Singh Sardar");
printf("\nname=%s",name);
getch();
}
```

Application of Pointer

There are number of applications of pointer

1. Pointer is widely used in dynamic memory allocation. The functions used for allocation of memory at run time that return the address of allocated memory.
2. Pointer can be used to pass information back and forth between a function and its reference point. Pointer provides a way to return multiple values form arguments to function by reference address.
3. Pointers provide an alternative way to access individual array elements. They used to manipulate arrays more easily by using pointer nested of using array themselves.
4. They increase execution speed as they refer particular address directly.
5. They are used to create complex data structure such as linked list, trees, and graphs.

Passing Pointer to Function

A pointer can be passed to function as an argument. Passing pointer means passing address
variable instead of value. As address is passed in this mechanism is also known as call by
reference. When pointer is passed to function while calling, formal argument of function must be compatible with the passing pointer. If integer pointer is being passed formal argument in function must be pointer of type. If value in the passed address is changed within function definition, the value of actual variable also changed.

WAP to illustrarate user passing pointer to function

```
#include<stdio.h>
#include<conio.h>
void mul(int *m)
{
*m=*m+10;
}
void main()
{
int marks;
clrscr();
```

```
printf("the actual marks is");
scanf("%d", &marks);
mul(&marks);
printf("\n the numbers are %d", marks);
getch();
}
```

8. Structure and Union

A structure is a collection of variable under a single name. The variables may be of different
type and each has a name which is used to select it form structure. The variables are called
member of structure. A structure is a convenient way of grouping several pieces of related
information together. A structure combine several pieces of related information together.
A structure is new name of heterogeneous data type or user defined data type. It may use
other structures arrays or pointers as some of its members in a single unit.
An array can be used only to represent a group of data items that must be same type. Such
as int, float. However if we want to represents collection of data items of different type
using same name. Array cannot do so. In this situation, structure is used, which is method
for packing data of different items. For example name roll, fees marks are related
information of student. To store information about a student we would require to store the
name of student which is array of characters, roll of student which is int, fees of student is
in float similarly marks of student may be float and character of students may be m or f.
there attributes of students are grouped together into single entity, student. Here student
is known as structure which organized different data items in more meaningful way.

8.1. Defining a Structure

Syntax:

```
struct structure_name
{
data type variable;
date type variable;
……;
};
```

Once structure name is declared as new data type then, variables of that type can be declared as strut student name variable.

Example:

```
struct student
{
char name [];
int roll;
float marks;
};
```

Here, student is structure name and its members are name, roll marks. The student is new
data type and variable of structure student can be declared as: **struct student st;**
Here structure st is variable of type student. The multiple variables are declared as std2, std3, std3.

Each variables of structure has its own copy of member variables. The member variables are
accessed using dot (.) operator. For example std1.name std1.roll, std1.marks.

8.2. Structure Initialization

Like standard data type, the members of a structure variable can be initialized. The values
to be initialized must appear in order as in the definition of structure within brace and separated by comma. C does not allow initialization of individual structure member within
its definition.

Syntax:

struct structure_name= {variable, variable, variable, variable};

Example:

```
struct student
{
char name [];
int roll;
float marks ;
};
```

The variable of this type can be declared as: **strut student st= {“Yagya”, 100, 34.54};** or
is equivalent to **strut student st; st.name=”Ram”, st. Marks=34.54;st.roll=45;**

8.3. Processing and Accessing Member of Structure

The member of structure is usually processed individually, as separate entity. Therefore we
must be able to access individual structure members. A structure member can be accessed
by using period (.) operator.

Syntax:

structure variable. member

Example:

std.name,std.id,std.marks

8.3.1. Precedence of Dot Operator (.)

The dot operator is member of the highest precedence group and associates form left to right. Since it is an operator of highest precedence, dot operator will take first precede over
various arithmetic, relational and logical assignment and unary operator. Thus ++st .marks is
equivalent to ++(st. marks), implying that the dot operators acts first and then unary operator.

8.3.2. Structure Elements

The elements of structure are always stored in contiguous memory locations. A structure
variable reverses number of bytes equal to sum of bytes needs to each of its members. For
example int following structure students variable st takes 7 bytes in members are its

member's variable(roll, 2 bytes marks needs 4 bytes and remarks need 1 byte.)

8.3. Array of Structure

Like array of int, float or char type there may be array of structure. In this case, the array will have individual structure as its elements. In our previous structure example. If we want to keep record of 50 students, we have to make 50 structure variables like std1, std2. But this technique is not good. At this situation we can use array of structure to store records of 50 students. Similarly to declaring structure variable an array of structure can be declared in two ways:

```
struct employee
{
char name [];
int id;
float salary ;
} emp [10];
struct employee emp[10];
```

8.3.1. Initialization Array of Structure

We can initialize array of structure in same way as a single structure. This is illustrated as:

```
struct book
{
char name [30];
int page;
float price;
} ;
struct book b[5]={“”,45,45.3,””,30,50.90,””,45,80.89,””,230,809.89,””,67,68.34};
```

struct book[5] dimension means it reserve five location for each name, int page and float price.

8.4. Structure within Another Structure/Nested Structure

One structure can be nested within another structure in C. In other words the individual members of structure can be nested other on structure as well. Let us consider structure date which has member day, month and year. The date structure can be nested within another structure say person. Structure date is member of another structure person. This can be done as follows.

```
struct employee
{
char name[];
int id;
float salary;
struct
{
int day;
```

```
int month;
int year
}date;
}emp[10];
```
This can be nested within another structure as its member.
Here, structure person contain member date which is itself a structure with there members.
The members within structure date are accessed as big structure variable as
p.date.day,
p.date.month.

8.5. Pointer to Structure

Pointer can be used with structure to store address of structure type variable; we can define a structure type pointer variable as normal way. Let us consider a structure book that has members name, page and price. It can be declared as:

```
struct book
{
char name[45];
int page;
float price;
};
struct book b; // this is normal variable;
struct book *bp; // here bp is pointer variable of structure book.
```

To use structure members through pointer memory must be allocated for a structure by using function call malloc() or by adding declaration and assignment as given bp=&b;
An individual structure member can be accessed in terms of its corresponding pointer variable by writing variable -> member. Here -> is called arrow operator and there must be
structure on left side of this operator.

```
b.name or bp->name or (*bp).name
b.page or bp->page or (*bp).page
b.price or bp->price or (*bp).price
```

8.5.1. Function and Structure

Structure variable can also be passed to a function we may either pass individual structure
elements or the entire structure.

8.5.2. Passing Structure Members to Functions

A structure member can be treated just as normal variable of structure type. For example,
integer structure variable can be treated just as integer. Thus structure members can be passed to function like ordinary variables.

8.5.3. Passing Whole Structure to Functions

It is possible to send entire structures functions as arguments in the functions call structure
variable is treated as any ordinary variable.

8.5.4. Passing Structure Pointer to Functions

In this case the structure pointer is passed to function. If the pointer to a structure is passed

as an argument to a function, then any changes that are made in the function are visible in
the caller.

8.5.5. Passing Array of Structure to Function

Passing an array of structure to function involves same syntax and properties as passing
array to a function. The passing is done using a pointer array structure in function if, any changes made in function to structure to the structure also visible in caller.

8.6. Union

Union are similar to structure its syntax and use. It also contains members whose individual
data types may differ from one another (heterogeneous data type).
The distinction is that all members within union share the same storage area of computer
memory where as each member within a structure is assigned its own unique storage. Thus
unions are used to save memory space.
Since same memory is shared by all members. One variable can reside into another memory
at a time. When another variable is set into memory of previous involving multiple members
whose value need to be assigned to all members at the same time.
Therefore although union may contain many members of different types it can handled only
one big member at a time. The compiler allocates a piece of storage that is large enough to
hold largest variable type in union.

```
union student
{
int roll;
float marks ;
};
```

here union student allocates float data type within it int resides on.

9. File Handling

The input /output function printf() scanf() getchar(),puthar() are known as console oriented i/o
functions which always use keyboard is input device and monitor are out devices . While using these
library functions the entire data is lost when either program is terminated or the computer is turned off.
Again it becomes bored and time consuming to handle large volume of data through keyboard. It takes
a lot of time to enter entire data. These problems of data files in which data can be stored memory than
read/write whenever necessary, without destroying data.
A file is place on disk where a group of related data are stored. The data file allows us to store
information permanently and to access and alter these information when necessary. Programming
language C has various library functions for creating and processing data files. Mainly, there are two
types of data files one is stream oriented high-level and low level data files. In high-level data files
functions do their own buffer management where as the programmer should do it explicitly in the case
of system oriented level files.
The standard data files are again subdivided into text files and binary files. The test files consists
consecutive characters by characters be inter operated as individual data item. The binary file
origination data into blocks containing continues byte of information. For each binary and text files,
there are a number of Formatted and Unformatted library functions.

9.1. Opening and Closing a Files

Before a program write to a file or read form a file, the program must be open. Opening a file
establishes a link between program and operating system. This provides the operating system the name
of the file and mode in which file is to be opened. While working with high level data file we need
buffer are where information is stored temporarily in course of transferring data between computer
memory to data file. The buffer area is established as: FILE * ptr variable;
Here, FILE is special structure declared in header stdio.h . Each file we opened as its own file is special
structure that contains information about file being used. Such as its current size, its location in

memory etc. The ptr variable is a pointer type. That stores beginning address and buffer area allocated
after a file opened. Thus pointer contains all the information about file and its use and communication
link between system and the program. A data file is opened as:

ptr variable=open (filename, file mode);

The associate file name with buffer area and specifics how data files will be open in specified mode.

The function open () returns a pointer to beginning of buffer area associated with file. A file NULL
value is returned if file cannot be opened due to some reasons. After opening a file we can processes
data files as required. Finally data file must be closed. The closing a file ensures that all outstanding
information associated with file is flushed out form buffer and links to file are broken. It also prevent
any accidental misuse of file. The file is closed using other library functions fclose() as:

fclose(ptr variable);

9.1.1 File Opening Modes

The file opening mode specifies way in which a file should be opened. In other words, it specifies the
purpose of openings a file.

a) "r" read mode: This opens existing files for reading mode only. It searches specified file. If file
exists it loads it into memory and sets up a pointer to first character in it. If file does not exist, it returns
NULL. The possible operation reading from the file only.

b) "w" write mode: This mode of file opens a file for writing only. It searches specified file, If file
already exist, the contents are overwritten. If file does not exist new file is crated. It returns NULL, if it
is unable to open file in write mode. The possible operation is only writing to file.

c) "a" append mode: It opens an existing file for appending. It searches specified file, If specified
exists it, loads into memory and setup a pointer that points last character in it. If file does not exist, new
file is created. It returns NULL. if unable to open file. The possible operation is adding new content at
end of file.

d) "r+" read+write mode: It opens existing file for reading and writing. It searches specified file. If
file exists it loads into memory and a pointer first character. It will returns NULl, if the file does not
pointer exist. The possible operations are reading existing contents writing contents, modifying existing
contents can be done in this mode.

e) “w+” read+write mode: It opens file for reading and writing. If specified file exists. It contents are
destroyed. If file does not exist t new file is created. It returns NULL. If unable to open file. The
possible operations are writing new contents, reading them back and modifying contents of the file.
f) “a+” append and read mode: It opens an existing file for both reading and appending. A file will
be created it specified file does not exist. The possible operations are reading existing contents
appending new contents to end of file but it cannot modify existing content.

9.2. Library Functions for Reading File

String Input /Output:
fgets(): This function is used to read from file. **Syntax:** ***fgets (string,int, fptr variable)***
fputs (): It is used to write a string to the file. **Syntax:** ***fputs(string, file ptr variable);***
Character Input/Output:
Using these functions, data can be read form file or written to file one character at a time.
fgetc(): This function is used to read a character form a file.
Syntax: ***char vari= fgetc(file variable);***
fputc(): This function is used to write a character form a file.
Syntax: ***fputc(char or char variable, fle variable);***
Formatted I/O:
These functions are used to read/write number, character or string form file in format as user
requirement.
fprint(): This function is formatted output function, which is used to write some integer, float,
char or string to specific file.
Syntax: ***fprint(file variable, “control string” list variable);***
fscanf(): This function is formatted input function which is used to read some integer, float,
char or string form a file.
Syntax: ***fscanf(file variable," control string”, &list variables);***
End of File (EOF):
The EOF represents End of File and determines whether the file associated with a file handle has
reached end of file or not . This unique integer is sent to program by operating system and is defined in
a header file stdio.h . While creating a file, operating system transmits EOF signal when it finds last
character of file. In text mode, a special character 1A hex decimal is inserted after the last character in
the file. This character is used to indicate EOF. If character last encountered an any pint in file, the read

function will return EOF signal to program. An attempt to read after EOF might either case the program
to terminate with errors or result in a infinite loop situations. Thus the last point of file is detected using
EOF while reading data form file. Without this mark we cannot detect last character at the file such that
is difficultly to find up to what time character is to be read while reading data from the file.

Binary Data Files:

The binary data files organize data into block containing continuous bytes of information. A binary file
is file of any length that holds bytes with variables in range 0 to 0xff (0 to 225).These bytes have other
meaning like value of 13 means, carriage return, 10 means, line feed, 26 means, end of file. In modern
terms we recognized binary file as stream of bytes file opening mode of text file is appended by a
character.

1. "r" is replaced by rb
2. "w" is replaced by wb
3. "a" is replaced by ab
4. "r+" is replaced by r+b
5. "w "+ is replaced by w+b
6. "a+" is replaced by a+b;
7. fptr=fopen(“text, ‘’wb”);

The default mode is text mode. Thus when simple r is written as mode of opening, this is assumed as
text mode. We can write "rb" in place of r and file text mode of file. This syntax of all above mode in
the case of binary mode is similar to the previous.

Difference Between Binary Mode and Text Mode

There are mainly three difference between binary and text mode, they are as follows:

1. In text mode, a special character EOF whose ASCII is 1A hex (26 in decimal) is inserted after last
character in file. The binary mode files keep track of end from numbers present in directory entry of
file ranges from 0 to 255. In this mode numbers of characters presents in directory entry of the file its
mode. In this mode, numbers are stored in binary format. Therefore a number stored in memory
happens to be 1A hex.(bytes have other means). Therefore a number stored in memory may be 1A but
in text mode it represents the end of file.
2. In text mode, numbers are stored assuring of character where as in binary format they are stored the

same way as hey are stored in computer main memory. The number 1234 occupies 2 bytes but it
occupies 4 bytes in text mode because 1 character for one bytes in binary mode. If occupies 2 bytes.
The solution of file opens to fread() function and replaced fprintf() and fscanf() functions.
3. The end of line single character, the new line charcter is marked by '\n' with ASCII value 10 in
decimal whereas in dos e end of line is marked by two characters, the carriage return CR with ASCII
value 13 in decimal and line feed (LF) with ASCII value 10 in decimal which is same as of c's new
line character. In text mode a new line character is automatically converted into carriage return
combination before being written to the disk of file. Like the carriage return line feed combination on
the disk is converted into a new line when the file is read by C program. However these conversions
don't take place in this case of binary character. Similarly & while reading the combination of CR/LF
will be treated as two separate character as r and n character.

Record Input/Output

It is clear that character i/o and string i/o permits reading and writing of character data only formatted
i/o permits reading and writing of characters and number. However the numbers are stored as
sequential of character not in memory as result they take a lot of disk space in addition these methods
provides no direct way to read write complex data types such array and structures. Array and structure
can be handled by reading and writing each array element one at time but this approach is very
inefficient.
The solution of these problems is record i/o also known as block i/o record i/o writes number to file in
binary format so that integer are stored in 2 bytes are stored in 4 bytes with single precisions floating
point numbers are stored in 4 bytes the same format used to store number data in memory record i/o
permits reading and writing of data once. This process is not limited to single character or string of the
few values in fact arrays structure array of structure can be read and rewritten as single unit.
The function fwrite() is used for record output while fread() is used for record input.

Syntax: ***fwrite (&ptr,size of array, number of structure array);***
fread(&ptr, sixe of array, number of structure array, fptr);

DIRECT /RANDOM Access

The reading and writing in all previous programs was in sequential. While reading data form a file, data
items are read from beginning of file in sequence until end of file. Similarly while writing data to a file
data items are placed one after the other in sequence. We can access a particular data items place in any
location without starting beginning. Such type of access to a data items is called direct or random
access.
For random access in a file, knowledge of file pointer is necessary. A file pointer to particular bytes in a
file. While opening a file in write mode, the file pointer is at beginning of file. Every time when we
write to a file, file pointer moves to the end of data items so that writing can conscious from that point
while opening a file in read mode file pointer is at the beginning file. After reading data items the file
pointer moves to the beginning of next data item, which can subsequently are read. However if file is io
end in aped mode, then file pointer will be positioned at end of existing file, so that new data items can
be written form there onward. Therefore we can read any data items form a file or write a file randomly
if we would be successful to move this file pointer as our requirement. Important functions used in
random access area as:

fseek(): It sets the file pointer associate with a stream file handle to a new position. This function is
used to move he file pointer to different positions.

Syntax: ***fseek (fpt, offset, mode);***

rewind (): One way of positioning the file pointer to the beginning of the file to close the file and them
reopen it again . This same stack can accomplished without closing ah file using rewind() function .
This function positions hat file pointer in the beginning of the file. The use of functions rewind() is
equivalent or using fssek()fptr,0,seek-*set*)

ftell(): This function determines current location of file pointer . It returns integer values.

Syntax: ***ftell(fptr)***

Memory Leak

A memory leak, in computer programming, occurs when a computer program consumes memory but is unable to release it back to the operating system.
A memory leak can diminish the performance of the computer by reducing the amount of
available memory. Eventually, in the worst case, too much of the available memory may become allocated and all or part of the system or device stops working correctly, the application fails, or the system slows down unacceptably due to thrashing.
Memory leaks may not be serious or even detectable by normal means. In modern operating systems, normal memory used by an application is released when the application
terminates. This means that a memory leak in a program that only runs for a short time may
not be noticed and is rarely serious.

Example of Memory Leak in C

```
#include <stdlib.h>
void function_which_allocates(void)
{
/* allocate an array of 45 floats */
float *a = malloc(sizeof(float) * 45);
/* additional code making use of 'a' */
/* return to main, having forgotten to free the memory we malloc'd */
}
int main(void)
{
function_which_allocates();
/* the pointer 'a' no longer exists, and therefore cannot be freed,
but the memory is still allocated. a leak has occurred. */
}
```

We have to use the free() function to release the memory referenced by the pointer *a to
avoid any memory leak.

Macros v/s Functions

Macros are just substitution patterns applied to your code. They can be used almost anywhere in your code because they are replaced with their expansions before any compilation starts.
Functions, on the other hand, are actual block of code whose body is directly injected into
the call site.

Some points to note:

• In C, macro invocations do not perform type checking, or even check that arguments are well-formed, whereas function calls usually do.
• In C, a macro cannot use the return keyword with the same meaning as a function would do (it would make the function that asked the expansion terminate, rather than the macro). In other words, a macro cannot return anything which is not the result of the last expression invoked inside it.
• Since C macros use mere textual substitution, this may result in unintended sideeffects

and inefficiency due to re-evaluation of arguments and order of operations.
• Compiler errors within macros are often difficult to understand, because they refer to the expanded code, rather than the code the programmer typed. Thus, debugging information for inline code is usually more helpful than that of macro-expanded code.
• Many constructs are awkward or impossible to express using macros, or use a significantly different syntax.
• Many compilers can also inline expand some recursive functions; recursive macros are typically illegal.

Pseudocode

Pseudocode is an informal high-level description of the operating principle of a computer
program or other algorithm. It uses the structural conventions of a programming language,
but is intended for human reading rather than machine reading. Pseudocode typically omits
details that are not essential for human understanding of the algorithm, such as variable declarations, system-specific code and some functions.
The purpose of using pseudocode is that it is easier for people to understand than conventional programming language code, and that it is an efficient and environmentindependent
description of the key principles of an algorithm. It is commonly used in
textbooks and scientific publications that are documenting various algorithms, and also in
planning of computer program development, for sketching out the structure of the program before the actual coding takes place.
No standard for pseudocode syntax exists, as a program in pseudocode is not an executable
program. Pseudocode resembles, but should not be confused with skeleton programs, including dummy code, which can be compiled without errors. Flowcharts can be thought of
as a graphical alternative to pseudocode, but are more spacious on paper.

C Style Pseudocode

```
void function fizzbuzz
For (i = 1; i<=100; i++) {
set print_number to true;
If i is divisible by 3
print "Fizz";
set print_number to false;
If i is divisible by 5
print "Buzz";
set print_number to false;
If print_number, print i;
print a newline;
}
```

Difference between Algorithm and Pseudo-code

Algorithm --> A precise rule (or set of rules) specifying how to solve some problem

Algorithm means a method to solve the given problem.
Pseudo-Code --> Statements outlining the operation of a computer program, written in something
similar to computer language but in a more understandable format
Pseudo-code is a way to describe the algorithm in order to transform the algorithm into real source
code. You can write a Pseudo-code in any way if you like. But mostly, we use widely accepted symbols
to write it.
In short Pseudocode does not have any Rules its some kind of note making that helps us to tackle the
program. Algorithm have some rules and logic.