MAHAVEER
INSTITUTE OF SCIENCE AND TECHNOLOGY
Vyasapuri, Bandlaguda, Post: Keshavgiri, Hyderabad- 500 005,
Telangana, India.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING (AI & ML)

SUBJECT: COMPUTER PROGRAMMING AND DATA STRUCTURES

by
SANGYAM SOUNDARYA
(Assistant Professor)

CPDS SYLLABUS

Module 1:

INTRODUCTION TO PROGRAMMING:

Introduction to components of a computer system: disks, primary and secondary memory, processor, operating system, compilers, creating, compiling and executing a program etc., Number systems.

Introduction to Algorithms: steps to solve logical and numerical problems. Representation of Algorithm, Flowchart/Pseudo code with examples, Program design and structured programming.

Introduction to C Programming Language: variables (with data types and space requirements), Syntax and Logical Errors in compilation, object and executable code, Operators, expressions and precedence, Expression evaluation, Storage classes (auto, extern, static and register), type conversion, The main method and command line arguments

Bitwise operations: Bitwise AND, OR, XOR and NOT operators Conditional Branching and Loops: Writing and evaluation of conditionals and consequent branching with if, if-else, switch-case, ternary operator, goto, Iteration with for, while, do- while loops.I/O: Simple input and output with scanf and printf, formatted I/O, Introduction to stdin, stdout and stderr.

Module 2:

ARRAYS, STRINGS, STRUCTURES AND PREPROCESSOR:

Arrays: one and two dimensional arrays, creating, accessing and manipulating elements of arrays.

Strings: Introduction to strings, handling strings as array of characters, basic string functions available in C (strlen, strcat, strcpy, strstr etc.), arrays of strings

Structures: Defining structures, initializing structures, unions, Array of structures.

Preprocessor: Commonly used Preprocessor commands like include, define, undef, If, ifdef, ifndef.

Module 3:

POINTERS AND FILE HANDLING IN C:

Pointers: Idea of pointers, defining pointers, Pointers to Arrays and Structures, Use of Pointers in self-referential structures, usage of self-referential structures in linked list (no implementation) Enumeration data type.

Files: Text and Binary files, Creating and Reading and writing text and binary files, appending data to existing files, Writing and reading structures using binary files, Random access using fseek, ftell and rewind functions.

Module 4:

FUNCTION AND DYNAMIC MEMORY ALLOCATION:

Functions: Designing structured programs, declaring a function, Signature of a function, Parameters and return type of a function, passing parameters to functions, call by value, passing arrays to functions, passing pointers to functions, idea of call by reference, Some C standard functions and libraries.

Recursion: Simple programs, such as Finding Factorial, Fibonacci series etc., Limitations of Recursive functions.

Dynamic memory allocation: Allocating and freeing memory, Allocating memory for arrays of different data types.

Module 5:

INTRODUCTION TO ALGORITHMS:

Basic searching algorithms (linear and binary search techniques),Basic sorting algorithms (Bubble, Insertion, Quick, Merge and Selection sort algorithms) Basic concept of order of complexity through the example programs

Text Books:

1. Ream Thareja, Programming in C, Oxford university press.
2. B.A. Forouzan and R.F. Gilberg, C Programming and Data Structures, Cengage Learning, (3^{rd}Edition).

Reference Books:

1. Brian W. Kernighan and Dennis M. Ritchie, The C Programming Language, Prentice Hall of India.
2. R.G. Dromey, How to solve it by Computer, Pearson (16th Impression)
3. Stephen G. Kochan, Programming in C, Fourth Edition, PearsonEducation.
4. Herbert Schildt, C: The Complete Reference, McGraw Hill, 4th Edition
5. Byron Gottfried, Schaum's Outline of Programming with C,McGraw-Hill

E - Resources:

1. https://fresh2refresh.com/c-programming/
2. https://www.studytonight.com/c/
3. https://beginnersbook.com/2014/01/c-tutorial-for-beginners-with-examples/
4. https://www.programiz.com/c-programming
5. http://www.gtucampus.com/uploads/studymaterials/Degree%20EngineeringSandipFundaments_of_C.pdf
6. http://cs.indstate.edu/~cbasavaraj/cs559/the_c_programming_language_2.pdf

Module 1

Pre-Requests are followings.

1. Enthusiasm
2. Logic Building Skills
3. Mathematics(Basics like prime number, factorial)

Introduction to computers:
Definition of terms

Computer : is an electronic device that operates (works) under the control of programs stored in its own memory unit.
A computer is an electronic machine that processes raw data to give information as output.

An electronic device that accepts data as input, and transforms it under the influence of a set of special instructions called Programs, to produce the desired output (referred to as Information).

Explanations;
A computer is described as an electronic device because; it is made up of electronic components and uses electric energy (such as electricity) to operate.

A computer has an internal memory, which stores data & instructions temporarily awaiting processing, and even holds the intermediate result (information) before it is communicated to the recipients through the Output devices.

It works on the data using the instructions issued, means that, the computer cannot do any useful job on its own. It can only work as per the set of instructions issued.

A computer will accept data in one form and produce it in another form. The data is normally held within the computer as it is being processed.

Program:
A computer Program is a set of related instructions written in the language of the computer & is used to make the computer perform a specific task (or, to direct the computer on what to do).

A set of related instructions which specify how the data is to be processed.
A set of instructions used to guide a computer through a process.

Data: Is a collection of raw facts, figures or instructions that do not have much meaning to the user.

Data may be in form of numbers, alphabets/letters or symbols, and can be processed to produce information.

TYPES OF DATA.
There are two types/forms of data:

a). **Digital (discrete) data:**
Digital data is discrete in nature. It must be represented in form of numbers, alphabets or symbols for it to be processed by a computer. Digital data is obtained by counting. E.g. 1, 2, 3 …

b). **Analogue (continuous) data:**
Analogue data is continuous in nature. It must be represented in physical nature in order to be processed by the computer. Analogue data is obtained by measurement. E.g. Pressure, Temperature, Humidity, Lengths or currents, etc. The output is in form of smooth graphs from which the data can be read.

Data Processing:
It is the process of collecting all items of data together & converting them into information.

Processing refers to the way the data is manipulated (or handled) to turn it into information.
The processing may involve calculation, comparison or any other logic to produce the required result. The processing of the data usually results in some meaningful information being produced.

Information: is the data which has been refined, summarized & manipulated in the way you want it, or into a more meaningful form for decision-making. The information must be accurate, timely, complete and relevant.

Characteristics / Features of a Computer.

Before 20th century, most information was processed manually or by use of simple machines. Today, millions of people are using computers in offices and at home to produce and store all types of information

The following are some of the attributes that make computers widely accepted & used in the day-to-day activities in our society:

1. Speed.
Computers operate at very high speeds, and can perform very many functions within a very short time.
They can perform a much complicated task much faster than a human being.
The speed of a computer is measured in Fractions of seconds.
Millisecond - a thousandth of a second (10-3)
Microsecond - a millionth of a second (10-6)
Nanosecond - a thousand millionth of a second (10-9)
Picosecond - a million millionth of a second (10-12)

The speed of a computer is usually linked to the technology used to build it.
a). 1st Generation computers (1940s & early 1950s).
• The computers were built using Vacuum tubes, and the speed was measured in Milliseconds. E.g., a computer could perform 5,000 additions & 300 multiplications per second.

b). 2nd Generation computers (1950s & early 1960s).
Were built using Transistors. Their operation speeds increased & were measured in Microseconds. E.g., a computer could perform 1 million additions per second.

c). Mid 1960s.
Integrated Circuit (IC), which combined a no. of transistors & diodes together on a silicon chip, was developed. The speed increased to tens of millions of operations per second.
d). In 1971, Intel Corporation produced a very small, single chip called a Microprocessor, which

could perform all the operations on the computer's processor. The chip contained about 1,600 transistors.
e). Today's microprocessors are very powerful, cheaper & more reliable due to the use of the Large Scale Integration (LSI) & Very Large scale Integration (VLSI) technologies, which combines hundreds of thousands of components onto a single chip.
The computer speeds are now measured in Nanoseconds & Picoseconds.

2. Accuracy:
Unlike human beings, computers are very accurate, i.e., they never make mistakes.
A computer can work for very long periods without going wrong. However, when an error occurs the computer has a number of in-built, self-checking features in their electronic components that can detect & correct such errors.
Usually errors are committed by the users entering the data to the computer, thus the saying Garbage in Garbage Out (GIGO).
This means that, if you enter incorrect data into the computer and have it processed, the computer will give you misleading information.

3. Reliability.
The computer can be relied upon to produce the correct answer if it is given the correct instructions & supplied with the correct data.
Therefore, if you want to add two numbers, but by mistake, give the computer a "Multiply" instruction, the computer will not know that you intended to "ADD"; it will multiply the numbers supplied.
Similarly, if you give it the ADD instruction, but make a mistake and enter an incorrect data; let say, 14 & 83 instead of 14 & 38; then the computer will produce the "wrong" answer 97 instead of 52.
However, note that, 97 is 'correct' based on the data supplied.
Therefore, the output produced by a computer is only as reliable as the instructions used & the data supplied.

4. Consistency:
Computers are usually consistent. This means that, given the same data & the same instructions, they will produce the same answer every time that particular process is repeated.

5. Storage:

A computer is capable of storing large amounts of data or instructions in a very small space.
A computer can store data & instructions for later use, and it can produce/ retrieve this data when required so that the user can make use of it.
Data stored in a computer can be protected from unauthorized individuals through the use of passwords.

6. Diligence:
Unlike human beings, a computer can work continuously without getting tired or bored. Even if it has to do a million calculations, it will do the last one with the same speed and accuracy as the first one.

7. Automation:
A computer is an automatic device. This is because, once given the instructions, it is guided by these instructions and can carry on its job automatically until it is complete.

It can also perform a variety of jobs as long as there is a well-defined procedure.

8. Versatile:
A computer can be used in different places to perform a large number of different jobs depending on the instructions fed to it.

9. Imposition of a formal approach to working methods:
Because a computer can only work with a strict set of instructions, it identifies and imposes rigid rules for dealing with the data it is given to process.

CLASSIFICATION OF COMPUTERS
Computer exist in different sizes, weight and shapes
The major ways in which computers are classified are
i) Classification according to Functionality
In this category, computers are categorized according to the way they process data and kind of data a computer can process.
Example of this data are,
a) Analog Data
b) Digital data

ii) Classification according to the purpose
iii) Classification according to physical size
iii) classification according to functionality

ACCORDING TO PHYSICAL SIZE
TYPES OF COMPUTER

Analog computer

Analog computer measures and answer the questions by the method of "HOW MUCH". The input data is not a number infect a physical quantity like temp, pressure, speed, velocity.

1. Signals are continuous of (0 to 10 V)
2. Accuracy 1% Approximately
3. High speed
4. Output is continuous

Time is wasted in transmission time

ANALOG COMPUTER

DIGITAL COMPUTERS

Digital computer counts and answers the questions by the method of "HOW Many". The input data is represented by a number. These are used for the logical and arithmetic operations.

1. Signals are two level of (0 V or 5 V)
2. Accuracy unlimited
3. low speed sequential as well as parallel processing
4. Output is continuous but obtain when computation is completed.

MICRO COMPUTERS

Micro computer are the smallest computer system. There size range from calculator to desktop size. Its CPU is microprocessor. It also known as Grand child Computer.

1. Application : - personal computer, Multi user system, offices.

MINI COMPUTERS

These are also small general purpose system. They are generally more powerful and most useful as compared to micro computer. Mini computer are also known as mid range computer or Child computer.

1. **Application :- Departmental systems, Network Servers, work group system.**

MAINFRAME COMPUTERS

Mainframe computers are those computers that offer faster processing and grater storage area. The word "main frame" comes from the metal frames. It is also known as Father computer.

1. Application – Host computer, Central data base server.

SUPER COMPUTERS

1. Super computer are those computer which are designed for scientific job like whether forecasting and artificial intelligence etc. They are fastest and expensive. A super computer contains a number of CPU which operates in parallel to make it faster. It also known as grandfather computer.
2. Application – weather forecasting, weapons research and development.

CLASSIFICATION OF DIGITAL COMPUTERS

1. Desktop
2. Workstation
3. Notebook
4. Tablet PC
5. Handheld computer
6. Smart Phone

WORKSTATION

TABLET PC

HANDHELD PC (PDA)

SMART PHONE

COMPUTER SYSTEM

COMPUTER SYSTEM

Definition: Is a collection of entities(hardware,software and liveware) that are designed to receive, process, manage and present information in a meaningful format.

COMPONENTS OF COMPUTER SYSTEM

1. **Computer hardware -** Are physical parts/ intangible parts of a computer. eg Input devices, output devices, central processing unit and storage devices
2. **Computer software** - also known as programs or applications. They are classified into two classes namely - system software and application software
3. **Liveware -** is the computer user. Also known as the human ware. The user commands the computer system to execute on instructions.

a) COMPUTER HARDWARE

Hardware refers to the physical, tangible computer equipment and devices, which provide support for major functions such as input, processing (internal storage, computation and control), output, secondary storage (for data and programs), and communication.

HARDWARE CATEGORIES (Functional Parts)

A computer system is a set of integrated devices that input, output, process, and store data and information. Computer systems are currently built around at least one digital processing device. There are five main hardware components in a computer system: Input, Processing, Storage, Output and Communication devices.

1. **INPUT DEVICES**

Are devices used for entering data or instructions to the central processing unit. Are classified according to the method they use to enter data.

a) KEYING DEVICES

Are devices used to enter data into the computer using a set of Keys eg Keyboard, key-to- storage and keypad.

i) The keyboard

Keyboard (similar to a typewriter) is the main input device of a computer . It contains three types of keys-- alphanumeric keys, special keys and function keys. **Alphanumeric keys** are used to type all alphabets, numbers and special symbols like $, %, @, A etc. **Special keys** such as <Shift>, <Ctrl>, <Alt>, <Home>, <Scroll Lock> etc. are used for special functions. **Function keys** such as <Fl>, <F2>, <F3> etc. are used to give special commands depending upon the software used e.g.F5 reloads a page of an internet browser. The function of each and every key can be well understood only after working on a PC. When any key is pressed, an electronic signal is produced. This signal is detected by a keyboard encoder that sends a binary code corresponding to the key pressed to the CPU. There are many types of keyboards but 101 keys keyboard is the most popular one.

How the keys are organized

The keys on your keyboard can be divided into several groups based on function:

1. **Typing (alphanumeric) keys.** These keys include the same letter, number, punctuation, and symbol keys found on a traditional typewriter.
2. **Special (Control) keys.** These keys are used alone or in combination with other keys to perform certain actions. The most frequently used control keys are CTRL, ALT, the Windows key, and ESC.

3. **Function keys.** The function keys are used to perform specific tasks. They are labelled as F1, F2, F3, and so on, up to F12. The functionality of these keys differs from program to program.
4. **Cursor Movement (Navigation) keys.** These keys are used for moving around in documents or WebPages and editing text. They include the arrow keys, HOME, END, PAGE UP, PAGE DOWN, DELETE, and INSERT and ARROW KEYS.
5. **Numeric keypad.** The numeric keypad is handy for entering numbers quickly. The keys are grouped together in a block like a conventional calculator or adding machine.

B. POINTING DEVICES

Are devices that enter data and instructions into the computer using a pointer that appears on the screen. The items to be entered are selected by either pointing to or clicking on them.e.g mice, joystick, touch sensitive screen, trackballs

i) THE MOUSE

A mouse is a small device used to point to and select items on your computer screen. Although mice come in many shapes, the typical mouse does look a bit like an actual mouse. It's small, oblong, and connected to the system unit by a long wire that resembles a tail and the connector which can either be PS/2 or USB. Some newer mice are wireless.

A mouse usually **has two buttons:** a primary button (usually the left button) and a secondary button. Many mice also have a wheel between the two buttons, which allows you to scroll smoothly through screens of information.

When you move the mouse with your hand, a pointer on your screen moves in the same direction. (The pointer's appearance might change depending on where it's positioned on your screen.) When you want to select an item, you point to the item and then click (press and release) the primary button. Pointing and clicking with your mouse is the main way to interact with your computer. There are several types of mice: Mechanical mouse, optical mouse, optical-mechanical mouse and laser mouse.

Basic parts

A mouse typically has two buttons: a primary button (usually the left button) and a secondary button (usually the right button). The primary button is the one you will use most often. Most mice also include a scroll wheel between the buttons to help you scroll through documents and WebPages more easily. On some mice, the scroll wheel can be pressed to act as a third button. Advanced mice might have additional buttons that can perform other functions.

Holding and moving the mouse

Place your mouse beside your keyboard on a clean, smooth surface, such as a mouse pad. Hold the mouse gently with your index finger resting on the primary button and you thumb resting on the side. To move the mouse, slide it slowly in any direction. Don't twist it—keep the front of the mouse aimed away from you. As you move the mouse, a pointer (see picture) on your screen moves in the same direction. If you run out of room to move your mouse on your desk or mouse pad, just pick up the mouse and bring it back closer to you.
Pointing to an object often reveals a descriptive message about it. The pointer can change depending

on what you're pointing at. For example, when you point to a link in your web browser, the pointer changes from an arrow to a hand with a pointing finger.

Most **mouse actions** combine pointing with pressing one of the mouse buttons. There are four basic ways to use your mouse buttons: clicking, double-clicking, right-clicking, and dragging.

Clicking (single-clicking)

To click an item, point to the item on the screen, and then press and release the primary button (usually the left button).

Clicking is most often used to select (mark) an item or open a menu. This is sometimes called single-clicking or left-clicking.

Double-clicking

To double-click an item, point to the item on the screen, and then click twice quickly. If the two clicks are spaced too far apart, they might be interpreted as two individual clicks rather than as one double-click.

Double-clicking is most often used to open items on your desktop. For example, you can start a program or open a folder by double-clicking its icon on the desktop.

Right-clicking

To right-click an item, point to the item on the screen, and then press and release the secondary button (usually the right button).

Right-clicking an item usually displays a list of things you can do with the item. For example, when you right-click the Recycle Bin on your desktop, Windows displays a menu allowing you to open it, empty it, delete it, or see its properties. If you are unsure of what to do with something, right-click it.

C) SCANNING DEVICES
Are devices that capture an object or a document directly from the source. They are classified according to the technology used to capture data e.g. Scanners and Document readers**.**
i) Scanners
Used to capture a source document and converts it into an electronic **f**orm.
Example are - FlatBed and HandHeld scanners**.**

ii) Document readers
Are documents that reads data directly from source document and convey them as input in the form of electronic signal. e
Types of Document Readers

i) Optical Mar Reader (OMR)
ii) Barcode readers

iii) Optical Character Readers

b) Magnetic Readers
Reads data using magnetic ink. it uses principle of magnetism to sense data which have been written using magnetized ink.

THE CENTRAL PROCESSING UNIT (C P U)

Is the brain or the heart of a computer. Is also known as processor and consist of three units namely -
i) Control Unit (C U)
ii) Arithmetic logic Unit (A L U)
iii) Main Memory unit (M M U**)**

The system unit is the core of a computer system. Usually it's a rectangular box placed on or underneath your desk. Inside this box are many electronic components that process data. The most important of these components is the central processing unit (CPU), or microprocessor, which acts as the "brain" of your computer. Another component is random access memory (RAM), which temporarily stores information that the CPU uses while the computer is on. The information stored in RAM is erased when the computer is turned off.

Almost every other part of your computer connects to the system unit using cables. The cables plug into specific ports (openings), typically on the back of the system unit. Hardware that is not part of the system unit is sometimes called a **peripheral device.** Peripheral devices can be **external** such as a mouse, keyboard, printer, monitor, external Zip drive or scanner or **internal**, such as a CD-ROM drive, CD-R drive or internal modem. Internal peripheral devices are often referred to as ***integrated peripherals*.** There are two types according to shape: **tower** and **desktop**.

Tower System Unit Desktop System Unit

A **motherboard** (**mainboard**, **system board**, **planar board** or **logic board**) is the main printed circuit board found in computers and other expandable systems. It holds many of the crucial electronic components of the system, such as the central processing unit (CPU) and memory, and provides connectors for other peripherals.

Motherboard

TYPES OF PROCESSORS
I) Complex Instruction Set Computers (CISC)
ii) Reduced Instruction Set Computers (RISC)

FUNCTIONS OF CENTRAL PROCESSING UNIT
- Process data
- Control sequence of operations within the computers
- It gives command to all parts of a computer
- It control the use of the main memory in storing of data and instructions
- it provides temporary storage (RAM) and permanent storage(ROM) of data

THE CONTROL UNIT
Is the center of operations for the computer system, it directs the activities of the computer system.
Functions of Control Unit

Disks.:

A **disk** refers to magnetic media, such as a floppy **disk**, the **disk** in your computer's hard drive, an external hard drive. **Disks** are always rewritable unless intentionally locked or write-protected. You can easily partition a **disk** into several smaller volumes, too.

Disk storage (also sometimes called **drive storage**) is a general category of storage mechanisms where data is recorded by various electronic, magnetic, optical, or mechanical changes to a surface layer of one or more rotating disks. A **disk drive** is a device implementing such a storage mechanism. Notable types are the hard disk drive (HDD) containing a non-removable disk, the floppy disk drive (FDD) and its removable floppy disk, and various optical disc drives (ODD) and associated optical disc media.

A disk drive is a randomly addressable and rewritable storage device.....However, in popular usage, it has come to relate mainly to hard disk drives (HDDs). Disk drives can either be housed internally within a **computer** or housed in a separate box that is external to the **computer**.

What's the difference between a "disc" and a "disk?"

They're pronounced the same, but, technically speaking, there is a distinct difference between a disc and a disk.

Discs

A disc refers to optical media, such as an audio CD, CD-ROM, DVD-ROM, DVD-RAM, or DVD-Video disc. Some discs are read-only (ROM), others allow you to burn content (write files) to the disc once (such as a CD-R or DVD-R, unless you do a multisession burn), and some can be erased and rewritten over many times (such as CD-RW, DVD-RW, and DVD-RAM discs).

All discs are removable, meaning when you unmount or eject the disc from your desktop or Finder, it physically comes out of your computer.

Disks

A disk refers to magnetic media, such as a floppy disk, the disk in your computer's hard drive, an external hard drive. Disks are always rewritable unless intentionally locked or write-protected. You can easily partition a disk into several smaller volumes, too.
Disks are usually sealed inside a metal or plastic casing (often, a disk and its enclosing mechanism are collectively known as a "hard drive").

Generally, a disk is a round plate on which data can be encoded. There are two basic types of disks: *magnetic disks* and *optical disks*.

Magnetic Disks

On magnetic disks, data is encoded as microscopic magnetized *needles* on the disk's surface. You can record and erase data on a magnetic disk any number of times, just as you can with a cassette tape.
Magnetic disks come in a number of different forms:

- **floppy disk :** A typical 5¼-inch floppy disk can hold 360K or 1.2MB (megabytes). 3½-inch floppies normally store720K, 1.2MB or 1.44MB of data. Floppy disks or obsolete today, and are found on older computer systems.
- **hard disk :**Hard disks can store anywhere from 20MB to more than 1-TB (terabyte). Hard disks are also from 10 to 100 times faster than floppy disks.
- **removable cartridge :** Removable cartridges are hard disks encased in a metal or plastic cartridge, so you can remove them just like a floppy disk. Removable cartridges are very fast, though usually not as fast as fixed hard disks.

Optical Disks

Optical disks record data by burning microscopic holes in the surface of the disk with a laser.
To read the disk, another laser beam shines on the disk and detects the holes by changes in the reflection pattern.
Optical disks come in three basic forms:

- **CD-ROM :** Most optical disks are read-only. When you purchase them, they are already filled with data. You can read the data from a CD-ROM, but you cannot modify, delete, or write new data.
- **WORM :** Stands for ***w**rite-**o**nce, **r**ead-**m**any.* WORM disks can be written on once and then read any number of times; however, you need a special WORM disk drive to write data onto a WORM disk.
- **erasable optical (EO):** EO disks can be read to, written to, and erased just like magnetic disks.

The machine that spins a disk is called a disk drive. Within each disk drive is one or more *heads* (often called *read/write heads*) that actually read and write data.
Accessing data from a disk is not as fast as accessing data from main memory, but disks are much cheaper. And unlike RAM, disks hold on to data even when the computer is turned off.
Consequently, disks are the storagemedium of choice for most types of data. Another storage medium is magnetic tape. But tapes are used only for backup and archiving because they are *sequential-access* devices (to access data in the middle of a tape, the tape drive must pass through all the preceding data).
A new disk, called a *blank disk,* has no data on it. Before you can store data on a blank disk, however, you must *format* it.

Primary and secondary memory

Difference between Primary Memory and Secondary Memory

Memory is the brain of the computer which stores data and information for storing and retrieving. Just like a human brain, memory is the storage space of the computer – like a physical device – that is capable of storing data or programs temporarily or permanently.

Memory is a fundamental component of the computer that is categorized into primary and secondary memory. Primary memory is the main memory of the computer which can be directly accessed by the central processing unit, whereas secondary memory refers to the external storage device which can be used to store data or information permanently. While both serve the same purpose; that is to store data or instructions for further processing by the CPU, they do it very differently. Let’s take a look at the two in detail.

What is Primary Memory?

Primary memory, also known as the main memory, is the area in a computer which stores data and information for fast access.

Semiconductor chips are the principle technology used for primary memory. It's a memory which is used to store frequently used programs which can be directly accessed by the processing unit for further processing. It's a volatile memory meaning the data is stored temporarily and is liable to change or lose in case of power failure.

In simple terms, data is intact as long as the computer is running and the moment it's off, data is lost. Every application on the computer first loads into the random access memory (RAM) which makes is faster to access. The term is more ambiguous, since it also refers to internal memory such as internal storage devices.

Secondary Memory

On the contrary, secondary memory is the external memory of the computer which can be used to store data and information on a long-term basis.

It's a non-volatile memory which means data stays intact even if the computer is turned off. Data cannot be directly processed by the processing unit in secondary memory; in fact, it is first transferred into the main memory and then it's transferred back to the processing unit.

Secondary memory refers to all external storage devices that are capable of storing high volumes of data such as hard drives, floppy disks, magnetic tapes, USB flash drives, CDs, DVDs, etc. It's generally slower than primary memory but can store substantial amount of data, in the range of gigabytes to terabytes.

Difference between Primary and Secondary Memory

1.	Basics of Primary and Secondary Memory

Memory plays a critical part in computers to store and retrieve data. Computer memory is categorized into primary and secondary memory. While primary memory is the main memory of the computer which is used to store data or information temporarily, whereas secondary memory refers to external storage devices that are used to store data or information permanently.

1.	Access of Primary and Secondary Memory

Primary memory holds only those data or instructions which the computer is currently processing allowing the processor to access running applications and services that are stored temporarily in a specific memory address. Secondary memory, on the other hand, is persistent in nature which means instructions are transferred to the main memory first and then re-routed to the central processing unit.

1. Data in Primary and Secondary Memory

In primary memory, data is directly accessed by the processing unit and it resides in the main memory until processing. Information and data are stored in semiconductor chips so they have a limited storage capacity. In secondary memory, information is stored in external storage devices and they cannot be directly accessed by the processing unit.

1. Nature of Primary and Secondary Memory

Primary memory is volatile in nature which means data or information stored in the main memory is temporarily which may lead to loss of data in case of power failure and it cannot be retained. On the contrary, secondary memory is non-volatile in nature which means information is stored permanently with no data loss in case of power failure. Data is intact unless the user erases it intentionally.

1. Devices for of Primary and Secondary Memory

Primary memory can also be referred to as RAM, short for Random Access Memory, because of the random selection of memory addresses. RAM holds data in a uniform manner and it can be lost when power fails. Secondary memory refers to external storage devices such as hard disk, optical disk, compact disk, flash drives, magnetic tapes, etc. They are high-storage devices with substantial storage capacities, in the range of gigabytes to terabytes.

1. Speed of Primary and Secondary Memory

In primary memory, applications and instructions are stored in the main memory which makes them relatively faster to access via data bus. Processor is able to retrieve data faster than it does with secondary memory, which acts more like a backup memory to store data in external storage devices.

Primary Memory vs. Secondary Memory: Comparison Chart

Summary of Primary Vs. Secondary Memory

Computer memory is categorized into primary memory and secondary memory, along with cache memory. Primary memory is the main memory or internal memory of the computer which is used to store frequently used data and instructions. It provides fast memory access because of its volatile nature which makes it easy to retrieve information directly from the main memory by the processing unit. Secondary memory, on the other hand, refers to external storage devices which are used to store substantial amount of data in hard drives, flash drives, CDs, DVDs, floppy disks, magnetic tapes, etc. Unlike primary memory, secondary memory is not directly accessed by the processor.

Cache memory:

Cache memory is a small-sized type of volatile computer memory that provides high-speed data access to a processor and stores frequently used computer programs, applications and data. It is the fastest memory in a computer, and is typically integrated onto the motherboard and directly embedded in the processor or main random access memory (RAM).

Cache memory provides faster data storage and access by storing instances of programs and data routinely accessed by the processor. Thus, when a processor requests data that already has an instance in the cache memory, it does not need to go to the main memory or the hard disk to fetch the data.

Cache memory can be primary or secondary cache memory, with primary cache memory directly integrated into (or closest to) the processor. In addition to hardware-based cache, cache memory also can be a disk cache, where a reserved portion on a disk stores and provides access to frequently accessed data/applications from the disk.

Processor

DEFINITION

processor (CPU)

A processor (CPU) is the logic circuitry that responds to and processes the basic instructions that drive a computer. The CPU is seen as the main and most crucial integrated circuitry (IC) chip in a computer, as it is responsible for interpreting most of computers commands. CPUs will perform most basic arithmetic, logic and I/O operations, as well as allocate commands for other chips and components running in a computer.

The term processor is used interchangeably with the term central processing unit (CPU), although strictly speaking, the CPU is not the only processor in a computer. The GPU (graphics processing unit) is the most notable example, but the hard drive and other devices within a computer also perform some processing independently. Nevertheless, the term processor is generally understood to mean the CPU.

Processors can be found in PCs, smartphones, tablets and other computers. The two main competitors in the processor market are Intel and AMD.

The basic elements of a processor

The basic elements of a processor include:

1. The arithmetic logic unit (ALU), which carries out arithmetic and logic operations on the operands in instructions.
2. The floating point unit (FPU), also known as a math coprocessor or numeric coprocessor, a specialized coprocessor that manipulates numbers more quickly than the basic microprocessor circuitry can.
3. Registers, which hold instructions and other data. Registers supply operands to the ALU and store the results of operations.
4. L1 and L2cache memory. Their inclusion in the CPU saves time compared to having to get data from random access memory (RAM).

CPU Operations

The four primary functions of a processor are fetch, decode, execute and write back.

1. Fetch- is the operation which receives instructions from program memory from a systems RAM.
2. Decode- is where the instruction is converted to understand which other parts of the CPU are needed to continue the operation. This is performed by the instruction decoder
3. Execute- is where the operation is performed. Each part of the CPU that is needed is activated to carry out the instructions.

Components and how CPUs work

The main components of a CPU are the ALU, registers and control unit. The basic functions of the ALU and register are labeled in the above “basic elements of a processor section.” The control unit is what operates the fetching and execution of instructions.

The processor in a personal computer or embedded in small devices is often called a microprocessor. That term means that the processor's elements are contained in a single IC chip. Some computers will operate using a multi-core processor—a chip containing more than one CPU. A CPU is typically a small device with pins on it facing down in a motherboard. CPUs can also be attached to a motherboard with a heat sink and a fan to dissipate heat.

Types

Most processors today are multi-core, which means that the IC contains two or more processors for enhanced performance, reduced power consumption and more efficient simultaneous processing of multiple tasks (see: parallel processing). Multi-core set-ups are similar to having multiple, separate processors installed in the same computer, but because the processors are actually plugged into the same socket, the connection between them is faster.

A processor, or "microprocessor," is a small chip that resides in computers and other electronic devices. Its basic job is to receive input and provide the appropriate output. While this may seem like a simple task, modern processors can handle trillions of calculations per second.
The central processor of a computer is also known as the CPU, or "central processing unit." This processor handles all the basic system instructions, such as processing mouse and keyboard input and running applications. Most desktop computers contain a CPU developed by either Intel or AMD, both of which use the x86 processor architecture. Mobile devices, such as laptops and tablets may use Intel and AMD CPUs, but can also use specific mobile processors developed by companies like ARM or Apple.
Modern CPUs often include multiple processing cores, which work together to process instructions. While these "cores" are contained in one physical unit, they are actually individual processors. In fact, if you view your computer's performance with a system monitoring utility like Windows Task Manager (Windows) or Activity Monitor (Mac OS X), you will see separate graphs for each processor. Processors that include two cores are called dual-core processors, while those with four cores are called quad-coreprocessors. Some high-end workstations contain multiple CPUs with multiple cores, allowing a single machine to have eight, twelve, or even more processing cores.
Besides the central processing unit, most desktop and laptop computers also include a GPU. This processor is specifically designed for rendering graphics that are output on a monitor. Desktop computers often have a video card that contains the GPU, while mobile devices usually contain a

graphics chip that is integrated into the motherboard. By using separate processors for system and graphics processing, computers are able to handle graphic-intensive applications more efficiently.

Operating System:

Introduction to Operating System (OS)
Definition
OPERATING SYSTEMS

1. Is a software that acts as an interface between the user, application software and the computer hardware

EXAPMLE OF OPERATING SYSTEM

1. Microsoft Disk operating system (Ms DOS)
2. Windows (98, 2000, XP, vista)
3. Linux
4. Unix

PARTS OF OPERATING SYSTEM

1. Shell – it is the outer part of an operating system and it is responsible of interacting with the operating system
2. Kernel – Responsible for managing and controlling computer resources such as the processor, main memory, storage devices, input devices, output devices and communication devices

RESOURCE UNER THE OPERATING SYSTEM CONTROL

1. The processor
2. Main memory
3. Input/Output Device
4. Secondary storage devices
5. Communication devices and ports

FUNCTIONS OF OPERATING SYSTEM

1. **Job scheduling**

– it is the process of the operating system to keep list of jobs currently being run by the computer and clocking them in and out of the processor.

1. **Interrupt handling**

1. It is a break from the normal sequential processing of instructions in a program

1. **Resource control and allocation**

1. It is situation where the processor gives a computer resources a unique number called interrupt number so that it can be able to recognize and prioritize it.

1. **Memory Management**

1. It is where the operating system constantly assigns main memory storage partitions to data an instructions

1. **Error handling**

1. It is a situation whereby an operating system alerts the user of errors that arises in ease of illegal operations, hardware or software failure.

1. **Input/output handling**

NOTE

What is interrupt request?

1. Is a unique number that is given to a resource for identification purposes

What is the importance of interrupt computer

1. To enable urgent tasks/ processes to b given the first priority during program execution

What is virtual memory

1. Part of the hard disk that acts as maim memory
2. Operating system organizes the main memory in blocks called page frames. The processes are divided into partitions that can fit in a page. The operatind system swaps these pages between the main memory and the hard disk. The part of the hard disk where these pages are held in the virtual memory.

What is a deadlock

1. Is where a particular task holds a resourcs and refuses to release it for othe tasks to use.

CLASSIFICATION OF OPERATING SYSTEM

They are classified into three ways/types:-

1. **According to the number of tasks handled concurrently**
2. Single task – one task is operated at a any given time
3. Multi-Task – More than one task is processed apparently simultaneously

1. **According to the number of users**
2. Single user – Single (One) user operates a computer at any given time
3. Multi user – More than one user can operate the computer at the same time

1. **Human computer interface / G U I**

1. Interaction between the computer and the user.

1. **Command line** – The user types the commands at the command prompt to activate them by pressing the enter key

Advantages

1. Fast in operation by experienced users
2. flexible
3. Use Less memory
4. Don't require expensive hardware

Disadvantages

- Hard to learn and understand

- Not user friendly

- If you mistype or forgot the syntax of writing the command, you cannot operate it

- Difficult to learn

- Difficult to move information from one application to another

- Difficult to design and produce printed reports

- Do not support multi-users and multi-task

1. **Menu driven** – The user is provided with a list of menu to choose from

Advantages

1. More user friendly than command line
2. More easier to learn and understand
3. Eliminates the problem of forgetting the syntax of command since commands are provided for you

Disadvantages

1. Slow to operate
2. Not flexible

1. **Graphical User Interface(GUI)** – The user interacts with the computer using icons and menus and select them using pointer

Advantages

- It is easy to learn and operate

- They make it easy to exchange information between application

- Reduce the user training time and cost due to their inform mode of operation

- It is more user friendly

- Supports multi user and multi task

Disadvantages

1. Cost of GUI supporting hardware is higher
2. Many objects on the GUI confuse new computer users
3. - Require faster processors that are more expensive

WINDOW as used in operating system

Is a rectangular object created on a screen by operating system to contain input or output data for a particular program

PROPERTIES OF A WINDOW

1. Title bar
2. Display/ working area
3. Horizontal and vertical scrolls bars
4. Menu bars
5. Status bar
6. Tool bar

TYPES OF MENUS

1. Pop up menu
2. Pop down menu
3. Sub menu(Cascading/ Side kick menu)

COMPUTER FILES

Are classifies into types :-

1. **System Files:** Are files that contain information that is critical to the operations of the computer

1. **Application Files**

Are files that holds programs or application files

An operating system (OS) is the program that, after being initially loaded into the computer by a boot program, manages all of the other application programs in a computer. The application programs make use of the operating system by making requests for services through a defined application program interface (API). In addition, users can interact directly with the operating system through a user interface such as a command line or a graphical user interface (GUI).

An operating system can perform the following services for applications:

1. In a multitasking operating system, where multiple programs can be running at the same time, the OS determines which applications should run in what order and how much time should be allowed for each application before giving another application a turn.
2. It manages the sharing of internal memory among multiple applications.

3. It handles input and output to and from attached hardware devices, such as hard disks, printers and dial-up ports.
4. It sends messages to each application or interactive user (or to a system operator) about the status of operation and any errors that may have occurred.
5. It can offload the management of batch jobs (for example, printing) so that the initiating application is freed from this work.
6. On computers that can provide parallel processing, an operating system can manage how to divide the program so that it runs on more than one processor at a time.

All major computer platforms (hardware and software) require and sometimes include an operating system, and operating systems must be developed with different features to meet the specific needs of various form factors.

Types of operating systems

A mobile OS allows smartphones, tablet PCs and other mobile devices to run applications and programs. Mobile operating systems include Apple iOS, Google Android, BlackBerry OS and Windows 10 Mobile.

An embedded operating system is specialized for use in the computers built into larger systems, such as cars, traffic lights, digital televisions, ATMs, airplane controls, point of sale (POS) terminals, digital cameras, GPS navigation systems, elevators, digital media receivers and smart meters.

A network operating system (NOS) is a computer operating system system that is designed primarily to support workstation, personal computer, and, in some instances, older terminals that are connected on a local area network (LAN).

A real-time operating system (RTOS) is an operating system that guarantees a certain capability within a specified time constraint. For example, an operating system might be designed to ensure that a certain object was available for a robot on an assembly line.

Examples of operating systems: Common desktop operating systems include:

1. Windows is Microsoft’s flagship operating system, the de facto standard for home and business computers. Introduced in 1985, the GUI-based OS has been released in many versions since then. The user-friendly Windows 95 was largely responsible for the rapid development of personal computing.
2. Mac OS is the operating system for Apple's Macintosh line of personal computers and workstations.
3. Unix is a multi-user operating system designed for flexibility and adaptability. Originally developed in the 1970s, Unix was one of the first operating systems to be written in C language.
4. Linux is a Unix-like operating system that was designed to provide personal computer users a free or very low-cost alternative. Linux has a reputation as a very efficient and fast-performing system.

What is an Operating System?

An operating system is a software which acts as an interface between the end user and computer hardware. Every computer must have at least one OS to run other programs. An application like Chrome, MS Word, Games, etc needs some environment in which it will run and perform its task.

The OS helps you to communicate with the computer without knowing how to speak the computer's language. It is **not** possible for the user to use any computer or mobile device without having an operating system.

History Of OS

1. Operating systems were first developed in the late 1950s to manage tape storage
2. The General Motors Research Lab implemented the first OS in the early 1950s for their IBM 701
3. In the mid-1960s, operating systems started to use disks
4. In the late 1960s, the first version of the Unix OS was developed
5. The first OS built by Microsoft was DOS. It was built in 1981 by purchasing the 86-DOS software from a Seattle company
6. The present-day popular OS Windows first came to existence in 1985 when a GUI was created and paired with MS-DOS.

Features of Operating System

Here is a list commonly found important features of an Operating System:

1. Protected and supervisor mode
2. Allows disk access and file systems Device drivers Networking Security
3. Program Execution
4. Memory management Virtual Memory Multitasking
5. Handling I/O operations
6. Manipulation of the file system
7. Error Detection and handling
8. Resource allocation
9. Information and Resource Protection

What is a Kernel?

10. The kernel is the central component of computer operating systems. The only job performed by the kernel is to manage the communication between the software and the hardware. A Kernel is at the nucleus of a computer. It makes the communication between the hardware and software possible. While the Kernel is the innermost part of an operating system, a shell is the outermost one.

Features of Kernel

1. Low-level scheduling of processes
2. Inter-process communication
3. Process synchronization
4. Context switching

Types of Kernels

There are many types of kernels that exists, but among them, the two most popular kernels are:

1. Monolithic: A monolithic kernel is a single code or block of the program. It provides all the required services offered by the operating system. It is a simplistic design which creates a distinct communication layer between the hardware and software.

2. Microkernels: Microkernel manages all system resources. In this type of kernel, services are implemented in different address space. The user services are stored in user address space, and kernel services are stored under kernel address space. So, it helps to reduce the size of both the kernel and operating system.

Functions of an Operating System

In an operating system software performs each of the function:

1. Process management:- Process management helps OS to create and delete processes. It also provides mechanisms for synchronization and communication among processes.

1. Memory management:- Memory management module performs the task of allocation and de-allocation of memory space to programs in need of this resources.

1. File management:- It manages all the file-related activities such as organization storage, retrieval, naming, sharing, and protection of files.

1. Device Management: Device management keeps tracks of all devices. This module also responsible for this task is known as the I/O controller. It also performs the task of allocation and de-allocation of the devices.

1. I/O System Management: One of the main objects of any OS is to hide the peculiarities of that hardware devices from the user.

1. Secondary-Storage Management: Systems have several levels of storage which includes primary storage, secondary storage, and cache storage. Instructions and data must be stored in primary storage or cache so that a running program can reference it.

1. Security:- Security module protects the data and information of a computer system against malware threat and authorized access.

1. Command interpretation: This module is interpreting commands given by the and acting system resources to process that commands.

1. Networking: A distributed system is a group of processors which do not share memory, hardware devices, or a clock. The processors communicate with one another through the network.

1. Job accounting: Keeping track of time & resource used by various job and users.

1. Communication management: Coordination and assignment of compilers, interpreters, and another software resource of the various users of the computer systems.

Types of Operating system

1. Batch Operating System
2. Multitasking/Time Sharing OS
3. Multiprocessing OS
4. Real Time OS
5. Distributed OS
6. Network OS
7. Mobile OS

Batch Operating System

Some computer processes are very lengthy and time-consuming. To speed the same process, a job with a similar type of needs are batched together and run as a group.

The user of a batch operating system never directly interacts with the computer. In this type of OS, every user prepares his or her job on an offline device like a punch card and submit it to the computer operator.

Multi-Tasking/Time-sharing Operating systems

Time-sharing operating system enables people located at a different terminal(shell) to use a single computer system at the same time. The processor time (CPU) which is shared among multiple users is termed as time sharing.

Real time OS

A real time operating system time interval to process and respond to inputs is very small. Examples: Military Software Systems, Space Software Systems.

Distributed Operating System

Distributed systems use many processors located in different machines to provide very fast computation to its users.

Network Operating System

Network Operating System runs on a server. It provides the capability to serve to manage data, user, groups, security, application, and other networking functions.

Mobile OS

Mobile operating systems are those OS which is especially that are designed to power smartphones, tablets, and wearables devices.

Some most famous mobile operating systems are Android and iOS, but others include BlackBerry, Web, and watchOS.

Firmware

Firmware is a software program or set of instructions programmed on a hardware device. It provides the necessary instructions for how the device communicates with the other computer hardware. But how can software be programmed onto hardware? Good question. Firmware is typically stored in the flash ROM of a hardware device. While ROM is "read-only memory," flash ROM can be erased and rewritten because it is actually a type of flash memory.
Firmware can be thought of as "semi-permanent" since it remains the same unless it is updated by a firmware updater. You may need to update the firmware of certain devices, such as hard drives and video cards in order for them to work with a new operating system. CD and DVD drive manufacturers often make firmware updates available that allow the drives to read faster media. Sometimes manufacturers release firmware updates that simply make their devices work more efficiently.

You can usually find firmware updates by going to the "Support" or "Downloads" area of a manufacturer's website. Keeping your firmware up-to-date is often not necessary, but it is still a good idea. Just make sure that once you start a firmware updater, you let the update finish, because most devices will not function if their firmware is not recognized.

Difference between Firmware and Operating System

Firmware	Operating System
Firmware is one kind of programming	OS provides functionality over and above

that is embedded on a chip in the device which controls that specific device.	that which is provided by the firmware.
Firmware is programs that been encoded by the manufacture of the IC or something and cannot be changed.	OS is a program that can be installed by the user and can be changed.
It is stored on non-volatile memory.	OS is stored on the hard drive.

Difference between 32-Bit vs. 64 Bit Operating System

Parameters	32. Bit	64. Bit
Architecture and Software	Allow 32 bit of data processing simultaneously	Allow 64 bit of data processing simultaneously
Compatibility	32-bit applications require 32-bit OS and CPUs.	64-bit applications require a 64-bit OS and CPU.
Systems Available	All versions of Windows 8, Windows 7, Windows Vista, and Windows XP, Linux, etc.	Windows XP Professional, Vista, 7, Mac OS X and Linux.
Memory Limits	32-bit systems are limited to 3.2 GB of RAM.	64-bit systems allow a maximum 17 Billion GB of RAM.

The advantage of using Operating System

1. Allows you to hide details of hardware by creating an abstraction
2. Easy to use with a GUI
3. Offers an environment in which a user may execute programs/applications
4. The operating system must make sure that the computer system convenient to use
5. Operating System acts as an intermediary among applications and the hardware components
6. It provides the computer system resources with easy to use format
7. Acts as an intermediator between all hardware's and software's of the system

Disadvantages of using Operating System

1. If any issue occurs in OS, you may lose all the contents which have been stored in your system

2. Operating system's software is quite expensive for small size organization which adds burden on them. Example Windows
3. It is never entirely secure as a threat can occur at any time

Summary

1. An operating system is a software which acts as an interface between the end user and computer hardware
2. Operating systems were first developed in the late 1950s to manage tape storage
3. The kernel is the central component of a computer operating systems. The only job performed by the kernel is to the manage the communication between the software and the hardware
4. Two most popular kernels are Monolithic and MicroKernels
5. Process, Device, File, I/O, Secondary-Storage, Memory management are various functions of an Operating System
6. Batch, Multitasking/Time Sharing, Multiprocessing, Real Time, Distributed, Network, Mobile are various types of Operating Systems

Compilers

A **compiler** is a special program that processes statements written in a particular programming language and turns them into machine language or "code" that a computer's processor uses. Typically, a programmer writes language statements in a language such as Pascal or C one line at a time using an editor.

A compiler is a special program that processes statements written in a particular programming language and turns them into machine language or "code" that a computer's processor uses. Typically, a programmer writes language statements in a language such as Pascal or C one line at a time using an *editor*. The file that is created contains what are called the *source statements*. The programmer then runs the appropriate language compiler, specifying the name of the file that contains the source statements.

When executing (running), the compiler first parses (or analyzes) all of the language statements syntactically one after the other and then, in one or more successive stages or "passes", builds the output code, making sure that statements that refer to other statements are referred to correctly in the final code. Traditionally, the output of the compilation has been called *object code* or sometimes an *object module* . (Note that the term "object" here is not related to object-oriented programming.) The object code is machine code that the processor can execute one instruction at a time.

The Java programming language, a language used in object-oriented programming, introduced the possibility of compiling output (called bytecode) that can run on any computer system platform for which a Java virtual machine or bytecode interpreter is provided to convert the bytecode into instructions that can be executed by the actual hardware processor. Using this

virtual machine, the bytecode can optionally be recompiled at the execution platform by a just-in-time compiler. (See also: Java compiler)

Traditionally in some operating systems, an additional step was required after compilation - that of resolving the relative location of instructions and data when more than one object module was to be run at the same time and they cross-referred to each other's instruction sequences or data. This process was sometimes called *linkage editing* and the output known as a *load module*.

A compiler works with what are sometimes called 3GL and higher-level languages. An assembler works on programs written using a processor's assembler language.

Compiling and executing the program:

C is one of the **basic languages fo a beginner**. It is invented by **Dennis Ritchie**. In this article, you are going to write your first program. You will be educated with the step-by-step procedure for compilation and execution of C program with hello world program.

Writing Hello World program is very easy and everyone can code it. But it is important to understand how C program runs and what are the purpose of each line of code in the program.

In this program, we are adding stdio.h file and using printf() statement to print the string message "hello World" on output console.

Writing Hello World program in C:

```
1    #include <stdio.h>
2    int main()
3    {
4      //print output string on console
5      printf("Hello, World!\n");
6      return 0;
7    }
```

This program runs all the operating systems including Windows, Linux, Mac OS. The only prerequisite as you should have GCC compiler installed on your system.

How does Hello World program run?

step by steps…

1. The first line of code #include<stdio.h> is a preprocessor. stdio.h is a header file which includes multiple functions defined in it such as scanf(), printf() functions. Adding this header file in our program, allow us to use functions defined in the header file.
2. Every C program execution starts with main() function. Your program may have multiple functions, the code inside the main() function block is executed first. Here, the return type of the main() function is int.
3. As like all other function, main() function block starts with a { and end with a } delimiter.
4. Write printf() statement by passing a string as an input. It will print the string message on the output console.
5. As we have defined the main with return int value, write return 0; It returns a SUCCESS message to the operating system.
6. \n in string message moves the cursor to a new line.

How "Hello, World!" program works?

1. The #include <stdio.h> is a preprocessor command. This command tells compiler to include the contents of stdio.h (standard input and output) file in the program.
   The stdio.h file contains functions such as scanf() and printf() to take input and display output respectively.
   If you use printf() function without writing #include <stdio.h>, the program will not be compiled.
2. The execution of a C program starts from the main() function.
3. The printf() is a library function to send formatted output to the screen. In this program, the printf() displays Hello, World! text on the screen.
4. The return 0; statement is the "Exit status" of the program. In simple terms, program ends with this statement.
5. The 0 return value of the main() function represents successful execution of program while the return value 1 represents the unsuccessful execution of program.

Pre-processor directive

#include is a pre-processor directive in 'C.'

#include <stdio.h>, stdio is the library where the function printf is defined. printf is used for generating output. Before using this function, we have to first include the required file, also known as a header file (.h).

You can also create your own functions, group them in header files and declare them at the top of the program to use them. To include a file in a program, use pre-processor directive

```
#include <file-name>.h
```

File-name is the name of a file in which the functions are stored. Pre-processor directives are always placed at the beginning of the program.

https://www.programmingsimplified.com/c-hello-world-program

Compilation and Execution of C Program

Follow the steps given below.

1. Save the program as helloWorld.c (with .c extension).
2. Open a command prompt.
3. Go to the current directory where the program is saved using. You can use cd command to change the current directory.
4. Compile the program with the following command.

```
gcc helloWorld.c
```

It will compile the program file and create an executable file.

(a.exe if you are running program on the window system. a.out for Linux system).

1. Running executable file.

For Windows,

```
a.exe
```

For Linux,

```
./a.out
```

The output of the Program after execution:

It will print “Hello, World!” on output console.

Hello, World!

Why do we use return 0?

The main() is the first function that runs. “return <vlaue>” returns value to the OS.

This value can be 0 or 1.

As a part of good practice, you can use EXIT_SUCCESS instead of 0
and EXIT_FAILUREinstead of 1.

EXIT_SUCCESS and EXIT_FAILURE are defined in the stdlib.h header file as follow…

```
#define EXIT_SUCCESS    0

#define EXIT_FAILURE    1
```

If you want to use EXIT_SUCCESS or EXIT_FAILURE, you have to add stdlib.h header file.

In our hello world program, we are using “return 0”. It means program runs successfully.

Where does the header file stdio.h file present in C?

The stdio.h header file is located in GCC directory at .../include/stdio.h
You can find the location of all the header files by running the following command.

```
gcc -H -fsyntax-only helloWorld.c
```

We usually use a compiler with a graphical user interface, to compile our C program. This can also be done by using cmd. The command prompt has a set of steps we need to perform in order to execute our program without using a GUI compiler. In this article we would be understanding how to compile C program in command prompt.

So let us get with this article on how to compile C progam in command prompt,

How to Compile C Program in Command Prompt?

STEP 1:

Run the command ‘gcc -v’ to check if you have a compiler installed. If not you need to download a gcc compiler and install it. You can search for cmd in your windows system to open the command prompt.

STEP 2:

Create a c program and store it in your system. I have written a program to find the Armstrong Number and store it in the directory user. We would be using following code.

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

```
3   {
4   int num, original, rem, sum = 0;
5   printf("Enter a three digit Number: ");
6   scanf("%d", &num);
7   original = num;
8   while (original != 0)
9   {
10  rem = original%10;
11  sum =sum + rem*rem*rem;
12  original =original/ 10;
13  }
14  if(sum == num)
15  printf("%d is an Armstrong number.",num);
16  else
17  printf("%d is not an Armstrong number.",num);
18  return 0;
19  }
```

STEP 3:

Change the working directory to where you have your C program. You can do that by using the command ‘cd’, which changes the directory. We need to pass the name of the directory in which the program is stored.

Example: >cd Desktop

Our program is already in the user directory so, we don’t need to change it.

STEP 4:

The next step is to compile the program. To do this we need to use the command gcc followed by the name of the program we are going to execute. In our case, we will use Armstrong.c.

After this, an executable file will be created in the directory that your c file exists in. Eg: Armstrong.exe

STEP 5:

In the next step, we can run the program. This is done by simply giving the name of the executable file without any extension. On giving this we will get an output. Here, our Armstrong code is executed and we got output for this code.

What is algorithm?

An algorithm is a procedure or step-by-step instruction for solving a problem. They form the foundation of writing a program.

For writing any programs, the following has to be known:

1. Input
2. Tasks to be preformed
3. Output expected

For any task, the instructions given to a friend is different from the instructions given to a computer. Let's look at the difference between them and know how to give instructions to a computer.

Instruction to a Friend:

Step 1: Go to the College Board.

Step 2: Search for my name on the board.

Step 3: Check the rank against my name.

Instructions to the Computer:

Step 1: Go to the College Board.

Step 2: Read the first name.

Step 3: If this is my name, see rank and come back.

Step 4: Read the next name.

Step 5: Repeat the above two steps until you find my name.

Breaking down the steps into smaller steps is called algorithm.

Example:

Considering the above example, let's break down the task into steps:

1. Input – Name

2. Tasks to be preformed – Search for my name
3. Output expected – My rank

It is a good practice to write down the algorithm first before attempting at writing a program. Each step of the algorithm will get converted to a line or a set of lines in the programming language.

More Examples for understanding the concept of Algorithms:

Example 1: Telling A Friend How To Boil Water In A Kettle.

Let's look at the 3 requirements for writing an algorithm for this task:

1. Input – Kettle, Water
2. Tasks to be preformed – Boiling water, fill up the kettle
3. Output expected – Boiled water

Instructions to a Friend:

Step 1: Fill the kettle with water.

Step 2: Place it on the stove and turn on the burner.

Step 3: Turn off the burner when the water starts boiling.

Instructions to the Computer:

Step 1: Put the kettle under the tap.

Step 2: Turn on the tap.

Step 3: Check if 90% of the kettle is filled.

Step 4: If not, repeat the above step.

Step 5: Turn off the tap.

Step 6: Place the kettle in the burner.

Step 7: Turn on the burner.

Step 8: Check if the water is 100 C

Step 9: If not, repeat the above step.

Step 10: Turn off the burner.

Example 2: Shopping on M.G Road

Let's break down the task of buying a red color shoe with a bow, from the Bata showroom. The following are the 3 requirements for writing an algorithm for this task:

1. Input – Name of the shop, name of the road, color of the shoe, size of the shoe.
2. Tasks to be preformed – Shopping.
3. Output expected – Shoe.

The algorithm of this task for a friend is fairly easy and simple. So, let's look at the algorithm for the computer.

Step 1: Walk to the first shop on M.G road.

Step 2: Check if this in the Bata Showroom.

Step 3: Enter the shop and ask for a red shoe size 5.

Step 4: Buy the shoe and come back.

Step 5: Go to the next shop, repeat step 2

Writing algorithms is a crucial step in programming. Take up more real life examples and try to write algorithms for them, and keep practising till the time writing algorithms becomes second nature to you.

Structured Programming in C:-

Structured programming is a programming technique in which a larger program is divided into smaller subprograms to make it easy to understand, easy to implement and makes the code reusable, etc. Structured programming enables code reusability. Code reusability is a method of writing code once and using it many times. Using structured programming technique, we write the code once and use it many times. Structured programming also makes the program easy to understand, improves the quality of the program, easy to implement and reduces time.

In C, the structured programming can be designed using functions concept. Using functions concept, we can divide the larger program into smaller subprograms and these subprograms are implemented individually. Every subprogram or function in C is executed individually.

http://btechsmartclass.com/c_programming/C-Functions.html

What is the Difference Between Pseudocode and Flowchart?

https://pediaa.com/what-is-the-difference-between-pseudocode-and-flowchart/

Algorithm: An algorithm is a step by step sequence of solving a given problem.

Here, time complexity refers to the time required to run an algorithm while space complexity is the amount of memory necessary for an algorithm. Moreover, it is important to select the best algorithm to solve it after analyzing the time complexity and space complexity. Thus, pseudocode and flowchart are two methods of representing an algorithm.

The main difference between Pseudocode and Flowchart is that pseudocode is an informal high-level description of an **algorithm** while flowchart is a pictorial representation of an algorithm.

What is a Pseudocode

A pseudocode is an informal way of writing a program. However, it is not a computer program. It only represents the algorithm of the program in natural language and mathematical notations. Besides, there is no particular programming language to write a pseudocode. Unlike in regular programming languages, there is no syntax to follow when writing a pseudocode. Furthermore, it is possible to use pseudocodes using simple English language statements.

A pseudocode to find the total of two numbers is as follows.

```
SumOfTwoNumbers()
Begin
Set sum =0;
Read: number 1, number 2;
Set sum = number1 + number 2;
Print sum;

End
```

A pseudocode to find the area of a triangle is as follows.

```
AreaofTrinagle()
Begin
Read: base, height;
Set area = 0.5 * base * height;
Print area;
End
```

So, after writing the pseudocode, we can write the actual program using that pseudocode. Moreover, as it represents the algorithm, we can implement it using any programming language.

What is a Flowchart

A flowchart represents an algorithm using a diagram. Furthermore, flowchart diagrams are commonly used in programming to find the steps to write a program.

A simple flowchart for marks calculation is as follows. In it, the oval shape denotes the start and end. And, the rhombus shape represents inputs and outputs. Entering marks is an input while displaying marks is an output. Further, the diamond shape symbol represents the decision selection. Depending on the decision, the correct output will be displayed.

In overall, a flow chart represents the sequence of steps to follow in order to solve the problem. Thus, the flowchart diagrams are easier to draw and understand.

Difference Between Pseudocode and Flowchart

Definition

Pseudocode is an informal high-level description of the operating principle of an algorithm while a flowchart is a diagrammatic representation that illustrates a solution model to a given problem. Thus, this is the main difference between Pseudocode and Flowchart.

Representation

Furthermore, a pseudocode is written in natural language and mathematical notations help to write pseudocode. However, a flowchart is written using various symbols. Hence, this is another difference between Pseudocode and Flowchart.

Conclusion

In brief, an algorithm is used to develop a computer program. Moreover, pseudocode and flowchart are two methods of representing an algorithm. The main difference between Pseudocode and Flowchart is that Pseudocode is an informal high-level description of an algorithm while flowchart is a pictorial representation of an algorithm.

Steps to solve logical and numerical problems:

Here's my process and some tips to tackling a sample problem that hopefully some of you may find helpful in your journey.

1. Read the problem at least three times (or however many makes you feel comfortable)

You can't solve a problem you don't understand. There is a difference between the problem and the problem you think you are solving. It's easy to start reading the first few lines in a problem and assume the rest of it because it's similar to something you've seen in the past. If you are making even a popular game like Hangman, be sure to read through any rules even if you've played it before. I once was asked to make a game like Hangman that I realized was "Evil Hangman" only after I read through the instructions (it was a trick!).

Sometimes I'll even try explaining the problem to a friend and see if her understanding of my explanation matches the problem I am tasked with. You don't want to find out halfway through that you misunderstood the problem. Taking extra time in the beginning is worth it. The better you understand the problem, the easier it will be to solve it.

Let's pretend we are creating a simple function selectEvenNumbers that will take in an array of numbers and return an array evenNumbers of only even numbers. If there are no even numbers, return the empty array evenNumbers.

```
function selectEvenNumbers() {
  // your code here
}
```

Here are some questions that run through my mind:

1. How can a computer tell what is an even number? Divide that number by 2 and see if its remainder is 0.
2. What am I passing into this function? An array
3. What will that array contain? One or more numbers
4. What are the data types of the elements in the array? Numbers
5. What is the goal of this function? What am I returning at the end of this function? The goal is to take all the even numbers and return them in an array. If there are no even numbers, return an empty array.

2. Work through the problem manually with at least three sets of sample data

Take out a piece of paper and work through the problem manually. Think of at least three sets of sample data you can use. Consider corner and edge cases as well.

Corner case: a problem or situation that occurs outside of normal operating parameters, specifically when multiple environmental variables or conditions are simultaneously at extreme levels, even though each parameter is within the specified range for that parameter.

Edge case: problem or situation that occurs only at an extreme (maximum or minimum) operating parameter

For example, below are some sets of sample data to use:
[1]
[1, 2]

[1, 2, 3, 4, 5, 6]
[-200.25]
[-800.1, 2000, 3.1, -1000.25, 42, 600]

When you are first starting out, it is easy to gloss over the steps.Because your brain may already be familiar with even numbers, you may just look at a sample set of data and pull out numbers like2 , 4 , 6 and so forth in the array without fully being aware of each and every step your brain is taking to solve it. If this is challenging, try using large sets of data as it will override your brain's ability to naturally solve the problem just by looking at it. That helps you work through the real algorithm.

Let's go through the first array [1]

1. Look at the only element in the array [1]
2. Decide if it is even. It is not
3. Notice that there are no more elements in this array
4. Determine there are no even numbers in this provided array
5. Return an empty array

Let's go through the array [1, 2]

1. Look at the first element in array [1, 2]
2. It is 1
3. Decide if it is even. It is not
4. Look at the next element in the array
5. It is 2
6. Decide if it is even. It is even
7. Make an array evenNumbers and add 2 to this array
8. Notice that there are no more elements in this array
9. Return the array evenNumbers which is [2]

I go through this a few more times. Notice how the steps I wrote down for [1] varies slightly from [1, 2]. That is why I try to go through a couple of different sets. I have some sets with just one element, some with floats instead of just integers, some with multiple digits in an element, and some with negatives just to be safe.

3. Simplify and optimize your steps

Look for patterns and see if there's anything you can generalize. See if you can reduce any steps or if you are repeating any steps.

1. Create a function selectEvenNumbers
2. Create a new empty array evenNumbers where I store even numbers, if any
3. Go through each element in the array [1, 2]
4. Find the first element
5. Decide if it is even by seeing if it is divisible by 2. If it is even, I add that to evenNumbers
6. Find the next element
7. Repeat step #4
8. Repeat step #5 and #4 until there are no more elements in this array
9. Return the array evenNumbers, regardless of whether it has anything in it

This approach may remind you of Mathematical Induction in that you:

1. Show it is true for n = 1, n = 2, ...
2. Suppose it is true for n = k
3. Prove it is true for n = k + 1

Example of pseudocode

4. Write pseudocode

Even after you've worked out general steps, writing out pseudocode that you can translate into code will help with defining the structure of your code and make coding a lot easier. Write pseudocode line by line. You can do this either on paper or as comments in your code editor. If

you're starting out and find blank screens to be daunting or distracting, I recommend doing it on paper.

Pseudocode generally does not actually have specific rules in particular but sometimes, I might end up including some syntax from a language just because I am familiar enough with an aspect of the programming language. Don't get caught up with the syntax. Focus on the logic and steps.

For our problem, there are many different ways to do this. For example, you can use filter but for the sake of keeping this example as easy to follow along as possible, we will use a basic for loop for now (but we will use filter later when we refactor our code).

Here is an example of pseudocode that has more words:
function selectEvenNumberscreate an array evenNumbers and set that equal to an empty arrayfor each element in that array
see if that element is even
if element is even (if there is a remainder when divided by 2)
add to that to the array evenNumbersreturn evenNumbers

Here is an example of pseudocode that has fewer words:
function selectEvenNumbersevenNumbers = []for i = 0 to i = length of evenNumbers
if (element % 2 === 0)
add to that to the array evenNumbersreturn evenNumbers

Either way is fine as long as you are writing it out line-by-line and understand the logic on each line.

Refer back to the problem to make sure you are on track.

5. Translate pseudocode into code and debug

When you have your pseudocode ready, translate each line into real code in the language you are working on. We will use JavaScript for this example.

If you wrote it out on paper, type this up as comments in your code editor. Then replace each line in your pseudocode.

Then I call the function and give it some sample sets of data we used earlier. I use them to see if my code returns the results I want. You can also write tests to check if the actual output is equal to the expected output.

```
selectEvenNumbers([1])
selectEvenNumbers([1, 2])
selectEvenNumbers([1, 2, 3, 4, 5, 6])
selectEvenNumbers([-200.25])
selectEvenNumbers([-800.1, 2000, 3.1, -1000.25, 42, 600])
```

I generally use console.log() after each variable or line or so. This helps me check if the values and code are behaving as expected before I move on. By doing this, I catch any issues before I get too far. Below is an example of what values I would check when I am first starting out. I do this throughout my code as I type it out.

```
function selectEvenNumbers(arrayofNumbers) {let evenNumbers = []
 console.log(evenNumbers) // I remove this after checking output
 console.log(arrayofNumbers) // I remove this after checking output}
```

After working though each line of my pseudocode, below is what we end up with. // is what the line was in pseudocode. Text that is bolded is the actual code in JavaScript.

```
// function selectEvenNumbers
function selectEvenNumbers(arrayofNumbers) {// evenNumbers = []
  let evenNumbers = []// for i = 0 to i = length of evenNumbers
  for (var i = 0; i < arrayofNumbers.length; i++) {// if (element % 2 === 0)
    if (arrayofNumbers[i] % 2 === 0) {// add to that to the array evenNumbers
      evenNumbers.push(arrayofNumbers[i])
    }
  }// return evenNumbers
  return evenNumbers
}
```

I get rid of the pseudocode to avoid confusion.

```
function selectEvenNumbers(arrayofNumbers) {
  let evenNumbers = []for (var i = 0; i < arrayofNumbers.length; i++) {
    if (arrayofNumbers[i] % 2 === 0) {
    evenNumbers.push(arrayofNumbers[i])
    }
  }return evenNumbers
}
```

Sometimes new developers will get hung up with the syntax that it becomes difficult to move forward. Remember that syntax will come more naturally over time and there is no shame in referencing material for the correct syntax later on when coding.

6. Simplify and optimize your code

You've probably noticed by now that simplifying and optimizing are recurring themes.
"Simplicity is prerequisite for reliability."
In this example, one way of optimizing it would be to filter out items from an array by returning a new array using filter. This way, we don't have to define another
variable evenNumbers because filter will return a new array with copies of elements that match the filter. This will not change the original array. We also don't need to use a for loop with this approach. filter will go through each item, return either true, to have that element in the array, or false to skip it.

```
function selectEvenNumbers(arrayofNumbers) {
  let evenNumbers = arrayofNumbers.filter(n => n % 2 === 0)
  return evenNumbers
}
```

Simplifying and optimizing your code may require you to iterate a few times, identifying ways to further simplify and optimize code.

Here are some questions to keep in mind:

1. What are your goals for simplifying and optimizing? The goals will depend on your team's style or your personal preference. Are you trying to condense the code as much as possible? Is the goal to make it the code more readable? If that's the case, you may prefer taking that extra line to define the variable or compute something rather than trying to define and compute all in one line.

2. How else can you make the code more readable?

3. Are there any more extra steps you can take out?

4. Are there any variables or functions you ended up not even needing or using?

5. Are you repeating some steps a lot? See if you can define in another function.

6. Are there better ways to handle edge cases?

"Programs must be written for people to read, and only incidentally for machines to execute."
— Gerald Jay Sussman and Hal Abelson, Authors of "Structure and Interpretation of Computer Programs"

7. Debug

This step really should be throughout the process. Debugging throughout will help you catch any syntax errors or gaps in logic sooner rather than later. Take advantage of your Integrated Development Environment (IDE) and debugger. When I encounter bugs, I trace the code line-by-line to see if there was anything that did not go as expected. Here are some techniques I use:

1. Check the console to see what the error message says. Sometimes it'll point out a line number I need to check. This gives me a rough idea of where to start, although the issue sometimes may not be at this line at all.

2. Comment out chunks or lines of code and output what I have so far to quickly see if the code is behaving how I expected. I can always uncomment the code as needed.

3. Use other sample data if there are scenarios I did not think of and see if the code will still work.

4. Save different versions of my file if I am trying out a completely different approach. I don't want to lose any of my work if I end up wanting to revert back to it!

"The most effective debugging tool is still careful thought, coupled with judiciously placed print statements."
— Brian W. Kernighan, Computer Science Professor at Princeton University

8. Write useful comments

You may not always remember what every single line meant a month later. And someone else working on your code may not know either. That's why it's important to write useful comments to avoid problems and save time later on if you need to come back to it.

Stay away from comments such as:

// This is an array. Iterate through it.

// This is a variable

I try to write brief, high-level comments that help me understand what's going on if it is not obvious. This comes in handy when I am working on more complex problems. It helps understand

what a particular function is doing and why. Through the use of clear variable names, function names, and comments, you (and others) should be able to understand:

1. What is this code for?
2. What is it doing?

9. Get feedback through code reviews

Get feedback from your teammates, professors, and other developers. Check out Stack Overflow. See how others tackled the problem and learn from them. There are sometimes several ways to approach a problem. Find out what they are and you'll get better and quicker at coming up with them yourself.
"No matter how slow you are writing clean code, you will always be slower if you make a mess." — Uncle Bob Martin, Software Engineer and Co-author of the Agile Manifesto

10. Practice, practice, practice

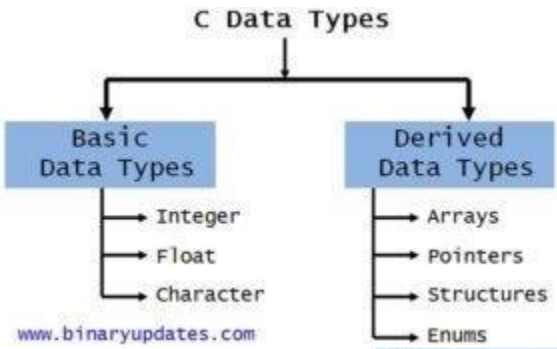
Even experienced developers are always practicing and learning. If you get helpful feedback, implement it. Redo a problem or do similar problems. Keep pushing yourself. With each problem you solve, the better a developer you become. Celebrate each success and be sure to remember how far you've come. Remember that programming, like with anything, comes easier and more naturally with time.

https://codeburst.io/10-steps-to-solving-a-programming-problem-8a32d1e96d74

Data types:
Let's learn about variables and data types in C Programming. We will first look at Variables in C; Variables are used to store the value during the execution of a program. The name itself means, the value of variable can be changed hence the name "Variable". The variables are stored in Main Memory i.e. RAM (size depending on the data type). In C Programming we always have to declare variable before we can use them. Note that the space is allocated to variable in memory during execution or run-time.
C is a strongly typed language. What this means it that, the type of a variable cannot be changed. Suppose we declared an integer type variable so we cannot store character or a decimal number in that variable.
There are set of rules to be followed while declaring variables and data types in C Programming:

1. The 1st letter should be alphabet.
2. Variables can be combination of alphabets and digits.
3. Underscore (_) is the only special character allowed.
4. Variables can be written in both Uppercase and Lowercase or combination of both.

5. Variables are Case Sensitive.
6. No Spaces allowed between Characters.
7. Variable name should not make use to the C Reserved Keywords.
8. Variable name should not start with a number.



Data Type	Description
Basic data types	int, char, float, double
Enumeration data type	Enum
Derived data types	pointer, array, structure, union
Void data type	Void

Variables and Data Types in C
Data Types in C Programming Language

EXAMPLE 1: Integer Variable in C

```
1 #include <stdio.h>
2
3 void main()
4 {
5     int i = 10;
6     printf("This is my integer: %d \n", i);
7 }
```

The *%d* is to tell **printf()** function to format the integer i as a decimal number. The output from this program would be This is my integer: 10.

EXAMPLE 2: Float Variable in C

```
1 #include <stdio.h>
2  void main()
```

```
3 {
4     float f = 3.1415;
5     printf("This is my float: %f \n", f);
6 }
```

The *%f* is to tell **printf()** function to format the float variable f as a decimal floating number. The output from this program would be This is my float: 3.1415. Now if we want to see only the first 2 numbers after the floating point, we would have to modify *printf()* function call to be as given below:

```
1  printf("This is my shorter float: %.2f \n", f);
```

After replacing *printf* function in a given example. The output from this program would be This is my float: 3.14.

EXAMPLE 3: Character Variable in C

```
1 #include <stdio.h>
2
3 void main()
4 {
5     char c;
6     c = 'b';
7     printf("This is my character: %c \n", c);
8 }
```

The *%c* is to tell **printf()** function to format the variable *"c"* as a character. The output from this program would be This is my character: b.

Data types in C Programming

All the data types defined by C are made up of units of memory called bytes. On most computer architectures a byte is made up of eight bits, each bit stores a one or a zero. These eight bits with two states give 256 combinations (2^8). So an integer which takes up four bytes can store a number between 0 to 4,294,967,295 (0 and 2^{32}). However, integer variables use the first bit to store whether the number is positive or negative so their value will be between -2,147,483,648 and + 2,147,483,647.

As we mentioned, there are eight basic data types defined in the C language. Five types for storing integers of varying sizes and three types for storing floating point values (values with a decimal point). C doesn't provide a basic data type for text. Text is made up of individual

characters and characters are represented by numbers. In the last example we used one of the integer types: int. This is the most commonly used type in the C language.

Data Type	Size	Value Range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 byte	-32,768 to 32,767
unsigned short	2 byte	0 to 65,535
long	4 byte	-2,147,483,648 to 2,147,483,647
unsigned long	4 byte	0 to 4,294,967,295

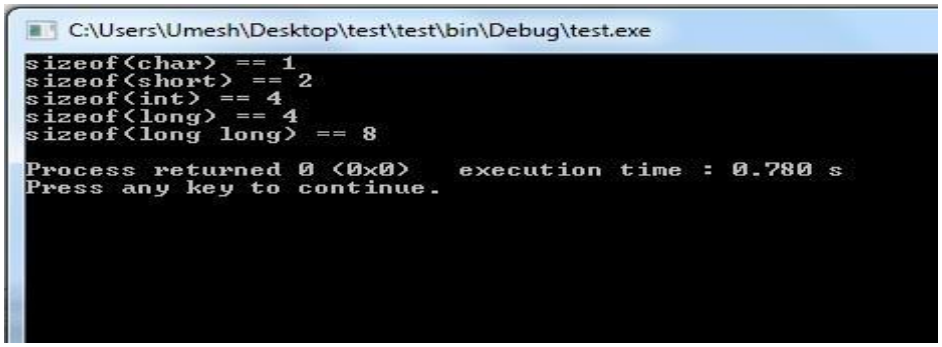
The **char** data type is usually one byte, it is so called because they are commonly used to store single characters. The size of the other types is dependent on machine. Most desktop machines are “32-bit”, this refers to the size of data that they are designed for processing. On “32-bit” machines the *int* data type takes up 4 bytes (232). The *short* is usually smaller, the *long* can be larger or the same size as an *int* and finally the *long long* is for handling very large numbers.
The type of variable you use generally doesn’t have a big impact on the speed or memory usage of your application. Unless you have a special need you can just use **int** variables. We will try to point out the few different cases like. A decade ago, most machines had 16-bit processors, this limited the size of **int**variables to 2 bytes. At the time, **short** variables were usually also 2 bytes and **long** would be 4 bytes. Nowadays, with 32-bit machines, the default type (**int**) is usually large enough to satisfy what used to require a variable of type **long**. The **long long** type was introduced more recently to handle very large numeric values.
EXAMPLE 3: Calculate Size of Data type on Your Machine

To find out the size of each data type on your machine compile and run this program. It uses one new language construct **sizeof()**. This tells us how many bytes a data type takes up.

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

```
2  int main()
3  {
4    printf("sizeof(char) == %d\n", sizeof(char));
5    printf("sizeof(short) == %d\n", sizeof(short));
6    printf("sizeof(int) == %d\n", sizeof(int));
7    printf("sizeof(long) == %d\n", sizeof(long));
8    printf("sizeof(long long) == %d\n", sizeof(long long));
9    return 0;
10 }
11
12
```

Theoutput from this program would be:



Syntax and Logical Errors in compilation:

Very programmer knows that debugging is a time-consuming nightmare, so it makes a good candidate for programmers to learn how to deal with. There are generally two types of errors: **syntax** errors and **logic** errors.

Syntax errors occur when a program does not conform to the grammar of a programming language, and the compiler cannot compile the source file. Logic errors occur when a program does not do what the programmer expects it to do.

Syntax errors are usually easy to fix because the compiler will tell you where the error occurs and you simply fix the syntax error. For example you may miss a semicolon or a curly bracket where it's supposed to be. Simply locate those errors and fix them.

C Programming Error Types – Runtime, Compile & Logical Errors

While writing c programs, errors also known as bugs in the world of programming may occur unwillingly which may prevent the program to compile and run correctly as per the expectation of the programmer.

Basically there are three types of errors in c programming:

1. Runtime Errors
2. Compile Errors
3. Logical Errors

C Runtime Errors

C runtime errors are those errors that occur during the execution of a c program and generally occur due to some illegal operation performed in the program.

Examples of some illegal operations that may produce runtime errors are:

1. Dividing a number by zero
2. Trying to open a file which is not created
3. Lack of free memory space

It should be noted that occurrence of these errors may stop program execution, thus to encounter this, a program should be written such that it is able to handle such unexpected errors and rather than terminating unexpectedly, it should be able to continue operating. This ability of the program is known as robustness and the code used to make a program robust is known as guard code as it guards program from terminating abruptly due to occurrence of execution errors.

Compile Errors

Compile errors are those errors that occur at the time of compilation of the program. C compile errors may be further classified as:

Syntax Errors

When the rules of the c programming language are not followed, the compiler will show syntax errors.

For example, consider the statement,

```
1    int a,b:
```

The above statement will produce syntax error as the statement is terminated with : rather than ;

Semantic Errors

Semantic errors are reported by the compiler when the statements written in the c program are not meaningful to the compiler.

For example, consider the statement,

```
1       b+c=a;
```

In the above statement we are trying to assign value of a in the value obtained by summation of b and c which has no meaning in c. The correct statement will be

```
1       a=b+c;
```

Logical Errors

Logical errors are the errors in the output of the program. The presence of logical errors leads to undesired or incorrect output and are caused due to error in the logic applied in the program to produce the desired output.

Also, logical errors could not be detected by the compiler, and thus, programmers has to check the entire coding of a c program line by line.

Object and executable code:-

What does *Object Code* mean?

Object code is produced when an interpreter or a compiler translates source code into recognizable and executable machine code.

Object code is usually produced by a compiler that reads some higher level computer language source instructions and translates them into equivalent machine language instructions.

{{{

Just as human beings understand native languages, computers understand machine language, which is made up of object code. Software applications are built in multiple programming languages with a standard objective: to execute processes via a machine.

A compiler translates source code into object code, which is stored in object files. Object files contain object code that includes instructions to be executed by the computer. It should be noted that object files may require some intermediate processing by the operating system (OS) before the instructions contained in them are actually executed by the hardware.

Object file examples include common object file format (COFF), COM files and ".exe" files.

}}}

https://www.quora.com/What-is-the-object-code-source-code-and-executable-code-in-C
Source code is the C program that you write in your editor and save with a ‘ .C ‘ extension. Which is **un-compiled** (when written for the first time or whenever a change is made in it and saved).

Object code is the **output of a compiler** after it processes the **source code**. The object code is usually a ***machine code***, also called a ***machine language***, which can be **understood directly by a specific** type of CPU (central processing unit), such as x86 (i.e., Intel-compatible) or PowerPC.

Executable (also called the Binary) is the **output of a linker** after it processes the **object code**. A machine code file can be immediately *executable* (i.e., runnable as a program), or it might require *linking* with other object code files (e.g. ***libraries***) to produce a complete executable program.

---

https://www.thecrazyprogrammer.com/2018/05/source-code-and-object-code.html

http://www2.hawaii.edu/~takebaya/ics111/process_of_programming/process_of_programming.html

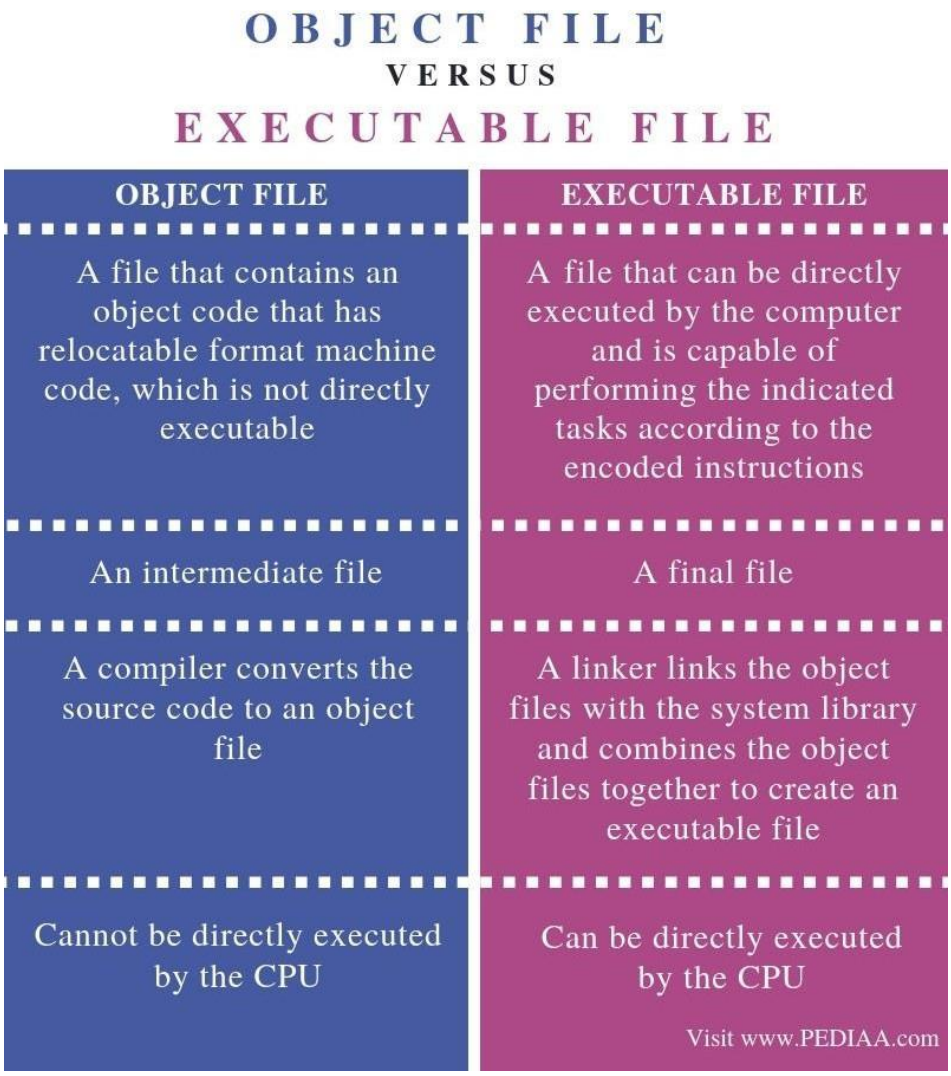
Linker takes the object files or run time libraries as input and combines them to produce executable) file.

Note:-

In UNIX/Linux, the executable file doesn't have extension whereas in Windows the executables for example may have .exe, .com and .dll.

https://pediaa.com/what-is-the-difference-between-object-file-and-executable-file/

The main difference between object file and executable file is that an object file is a file generated after compiling the source code while an executable file is a file generated after linking a set of object files together using a linker.



Operators Precedence in C

Operator precedence determines the grouping of terms in an expression and decides how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has a higher precedence than the addition operator.

For example, x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has a higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category	Operator	Associativity

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

Example

Try the following example to understand operator precedence in C –

```
#include <stdio.h>


main() {
   int a = 20;
   int b = 10;
   int c = 15;
   int d = 5;
   int e;
    e = (a + b) * c / d;      // ( 30 * 15 ) / 5
   printf("Value of (a + b) * c / d is : %d\n",  e );
   e = ((a + b) * c) / d;    // (30 * 15 ) / 5
   printf("Value of ((a + b) * c) / d is  : %d\n" ,  e );
   e = (a + b) * (c / d);  // (30) * (15/5)
   printf("Value of (a + b) * (c / d) is  : %d\n",  e );

   e = a + (b * c) / d;     //  20 + (150/5)
   printf("Value of a + (b * c) / d is  : %d\n" ,  e );
   return 0;
}
```

When you compile and execute the above program, it produces the following result −

```
Value of (a + b) * c / d is : 90
Value of ((a + b) * c) / d is  : 90
Value of (a + b) * (c / d) is  : 90
Value of a + (b * c) / d is  : 50
```

Expression evaluation

http://ecomputernotes.com/what-is-c/types-and-variables/what-is-expressions-type-of-expression

An expression is a combination of variables constants and operators written according to the syntax of C language. In C every expression evaluates to a value i.e., every expression results in some value of a certain type that can be assigned to a variable. Some examples of C expressions are shown in the table given below.

Algebraic Expression	C Expression
a x b – c	a * b – c
(m + n) (x + y)	(m + n) * (x + y)
(ab / c)	a * b / c
$3x^2$ +2x + 1	3*x*x+2*x+1
(x / y) + c	x / y + c

Evaluation of Expressions

Expressions are evaluated using an assignment statement of the form
Variable = expression;
Variable is any valid C variable name. When the statement is encountered, the expression is evaluated first and then replaces the previous value of the variable on the left hand side. All variables used in the expression must be assigned values before evaluation is attempted. Example of evaluation statements are
x = a * b – c
y = b / c * a
z = a – b / c + d;
The following program illustrates the effect of presence of parenthesis in expressions.

```
main ()
{
   float a, b, c x, y, z;
  a = 9;
        b = 12;
        c = 3;
        x = a – b / 3 + c * 2 – 1;
        y = a – b / (3 + c) * (2 – 1);
        z = a – ( b / (3 + c) * 2) – 1;
        printf ("x = %f",x);
        printf ("y = %f",y);
        printf ("z = %f",z);
}
```

output

x = 10.00
y = 7.00
z = 4.00

Precedence in Arithmetic Operators

An arithmetic expression without parenthesis will be evaluated from left to right using the rules of precedence of operators. There are two distinct priority levels of arithmetic operators in C.

*High priority * / %*
Low priority + -

Rules for evaluation of expression

• First parenthesized sub expression left to right are evaluated.

• If parenthesis is nested, the evaluation begins with the innermost sub expression.

• The precedence rule is applied in determining the order of application of operators in evaluating sub expressions.

• The associability rule is applied when two or more operators of the same precedence level appear in the sub expression.

• Arithmetic expressions are evaluated from left to right using the rules of precedence.

• When Parenthesis is used, the expressions within parenthesis assume highest priority.

Specifier Meaning

%c – Print a character
%d – Print a Integer
%i – Print a Integer
%e – Print float value in exponential form.
%f – Print float value
%g – Print using %e or %f whichever is smaller
%o – Print actual value
%s – Print a string
%x – Print a hexadecimal integer (Unsigned) using lower case a – F
%X – Print a hexadecimal integer (Unsigned) using upper case A – F
%a – Print a unsigned integer.
%p – Print a pointer value
%hx – hex short
%lo – octal long
%ld – long

Storage classes (auto, extern, static and register)

What is a Storage Class?

A storage class defines the scope (visibility) and life-time of variables and/or functions within a C Program.

A storage class represents the visibility and a location of a variable. It tells from what part of code we can access a variable. A storage class is used to describe the following things:

1. The variable scope.
2. The location where the variable will be stored.
3. The initialized value of a variable.
4. A lifetime of a variable.
5. Who can access a variable?

Thus a storage class is used to represent the information about a variable.

NOTE: A variable is not only associated with a data type, its value but also a storage class.

There are total four types of standard storage classes. The table below represents the storage classes in 'C'.

Storage class	**Purpose**
auto	It is a default storage class.
extern	It is a global variable.
static	It is a local variable which is capable of returning a value even when control is transferred to the function call.
register	It is a variable which is stored inside a Register.

Auto storage class

The variables defined using auto storage class are called as local variables. Auto stands for automatic storage class. A variable is in auto storage class by default if it is not explicitly specified.

The scope of an auto variable is limited with the particular block only. Once the control goes out of the block, the access is destroyed. This means only the block in which the auto variable is declared can access it.

A keyword auto is used to define an auto storage class. By default, an auto variable contains a garbage value.

Example, auto int age;

The program below defines a function with has two local variables

```
int add(void) {
   int a=13;
   auto int b=48;
return a+b;}
```

We take another program which shows the scope level "visibility level" for auto variables in each block code which are independently to each other:

```
#include <stdio.h>
int main( )
{
  auto int j = 1;
  {
    auto int j= 2;
    {
      auto int j = 3;
      printf ( " %d ", j);
    }
    printf ( "\t %d ",j);
  }
  printf( "%d\n", j);}
```

OUTPUT:

```
 3 2 1
```

Extern storage class

Extern stands for external storage class. Extern storage class is used when we have global functions or variables which are shared between two or more files.

Keyword extern is used to declaring a global variable or function in another file to provide the reference of variable or function which have been already defined in the original file.

The variables defined using an extern keyword are called as global variables. These variables are accessible throughout the program. Notice that the extern variable cannot be initialized it has already been defined in the original file

Example, extern void display();

First File: main.c

```
#include <stdio.h>

extern i;

main() {

  printf("value of the external integer is = %d\n", i);

  return 0;}
```

Second File: original.c

```
#include <stdio.h>

i=48;
```

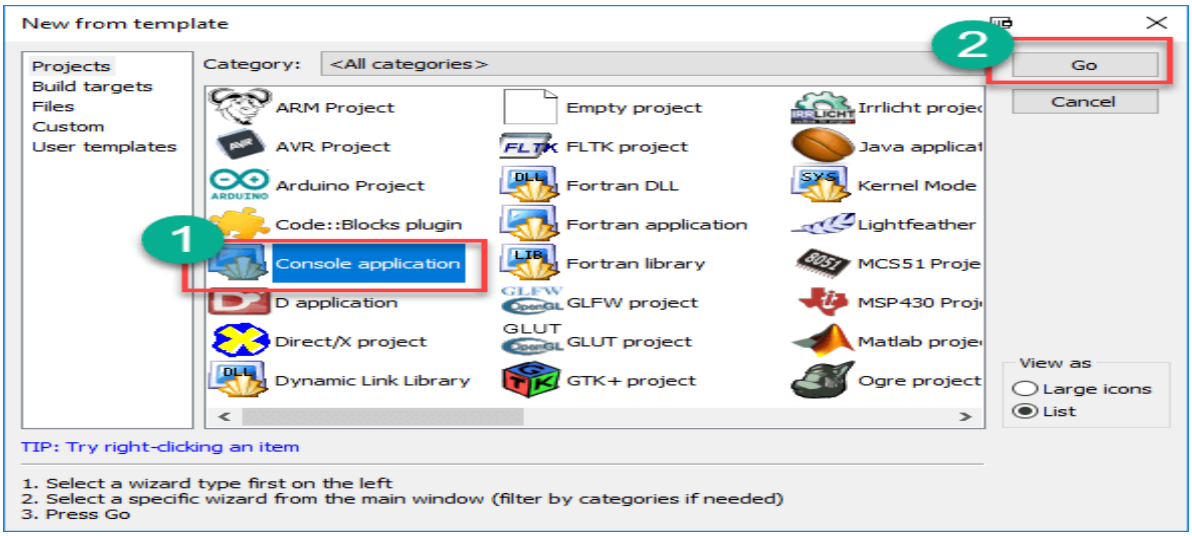
Result:

```
value of the external integer is = 48
```

In order to compile and run the above code, follow the below steps

Step 1) Create new project,

1. Select Console Application
2. Click Go



Step 2) Select C and click Next

Step 3) Click Next

Step 4) Enter details and click Next

Step 5) Click Finish

Step 6) Put the main code as shown in the previous program in the main.c file and save it

Step 7) Create a new C file [File -> new -> Empty File , save (as original.c) and add it to the current project by clicking "OK" in the dialogue box .

Step 8) Put and save the C code of the original.c file shown in the previous example without the main() function.

Step 9) Build and run your project. The result is shown in the next figure

Static storage class

The static variables are used within function/ file as local static variables. They can also be used as a global variable

1. Static local variable is a local variable that retains and stores its value between function calls or block and remains visible only to the function or block in which it is defined.
2. Static global variables are global variables visible only to the file in which it is declared.

Example: static int count = 10;

Keep in mind that static variable has a default initial value zero and is initialized only once in its lifetime.

```
#include <stdio.h> /* function declaration */
```

```
void next(void);
static int counter = 7; /* global variable */
main() {
 while(counter<10) {
      next();
      counter++;  }
return 0;}
void next( void ) {     /* function definition */
   static int iteration = 13; /* local static variable */
   iteration ++;
   printf("iteration=%d and counter= %d\n", iteration, counter);}
```

Result:

```
iteration=14 and counter= 7
iteration=15 and counter= 8
iteration=16 and counter= 9
```

Global variables are accessible throughout the file whereas static variables are accessible only to the particular part of a code.

The lifespan of a static variable is in the entire program code. A variable which is declared or initialized using static keyword always contains zero as a default value.

Register storage class

You can use the register storage class when you want to store local variables within functions or blocks in CPU registers instead of RAM to have quick access to these variables. For example, "counters" are a good candidate to be stored in the register.

```
Example: register int age;
```

The keyword register is used to declare a register storage class. The variables declared using register storage class has lifespan throughout the program.

It is similar to the auto storage class. The variable is limited to the particular block. The only difference is that the variables declared using register storage class are stored inside CPU registers instead of a memory. Register has faster access than that of the main memory.

The variables declared using register storage class has no default value. These variables are often declared at the beginning of a program.

```
#include <stdio.h> /* function declaration */

main() {

{register int  weight;

int *ptr=&weight ;/*it produces an error when the compilation occurs ,we cannot get a memory l
ocation when dealing with CPU register*/}

}
```

OUTPUT:

```
error: address of register variable 'weight' requested
```

The next table summarizes the principal features of each storage class which are commonly used in C programming

Storage Class	Declaration	Storage	Default Initial Value	Scope	Lifetime
auto	Inside a function/block	Memory	Unpredictable	Within the function/block	Within the function/block
register	Inside a function/block	CPU Registers	Garbage	Within the function/block	Within the function/block
extern	Outside all functions	Memory	Zero	Entire the file and other files where the variable is declared as extern	program runtime
Static (local)	Inside a function/block	Memory	Zero	Within the function/block	program runtime
Static (global)	Outside all functions	Memory	Zero	Global	program runtime

1. auto: This is the default storage class for all the variables declared inside a function or a block. Hence, the keyword auto is rarely used while writing programs in C language. Auto variables can be only accessed within the block/function they have been declared and not outside them (which defines their scope). Of course, these can be accessed within nested blocks within the parent block/function in which the auto variable was declared. However, they can be accessed outside their scope as well using the concept of pointers given here by pointing to the very exact memory location where the variables resides. They are assigned a garbage value by default whenever they are declared.

2. **extern**: Extern storage class simply tells us that the variable is defined elsewhere and not within the same block where it is used. Basically, the value is assigned to it in a different block and this can be overwritten/changed in a different block as well. So an extern variable is nothing but a global variable initialized with a legal value where it is declared in order to be used elsewhere. It can be accessed within any function/block. Also, a normal global variable can be made extern as well by placing the 'extern' keyword before its declaration/definition in any function/block. This basically signifies that we are not initializing a new variable but instead we are using/accessing the global variable only. The main purpose of using extern variables is that they can be accessed between two different files which are part of a large program. For more information on how extern variables work, have a look at this link.

1. **static**: This storage class is used to declare static variables which are popularly used while writing programs in C language. Static variables have a property of preserving their value even after they are out of their scope! Hence, static variables preserve the value of their last use in their scope. So we can say that they are initialized only once and exist till the termination of the program. Thus, no new memory is allocated because they are not re-declared. Their scope is local to the function to which they were defined. Global static variables can be accessed anywhere in the program. By default, they are assigned the value 0 by the compiler.

2. **register**: This storage class declares register variables which have the same functionality as that of the auto variables. The only difference is that the compiler tries to store these variables in the register of the microprocessor if a free register is available. This makes the use of register variables to be much faster than that of the variables stored in the memory during the runtime of the program. If a free register is not available, these are then stored in the memory only. Usually few variables which are to be accessed very frequently in a program are declared with the register keyword which improves the running time of the program. An important and interesting point to be noted here is that we cannot obtain the address of a register variable using pointers.

To specify the storage class for a variable, the following syntax is to be followed:

Syntax:

```
storage_class var_data_type var_name;
```

Functions follow the same syntax as given above for variables. Have a look at the following C example for further clarification:

```
// A C program to demonstrate different storage

// classes

#include <stdio.h>

  // declaring the variable which is to be made extern

// an intial value can also be initialized to x

int x;

  void autoStorageClass()

{

      printf("\nDemonstrating auto class\n\n");

      // declaring an auto variable (simply  writing "int a=32;" works as well)

    auto int a = 32;

      // printing the auto variable 'a'

    printf("Value of the variable 'a'"           " declared as auto: %d\n",           a);

    printf("-------------------------------------");

}

void registerStorageClass()

{

      printf("\nDemonstrating register class\n\n");      // declaring a register variable

    register char b = 'G';       // printing the register variable 'b'

    printf("Value of the variable 'b'"           " declared as register: %d\n",           b);

      printf("-------------------------------------");

}

void externStorageClass()

{

      printf("\nDemonstrating extern class\n\n");
```

```
    // telling the compiler that the variable      // z is an extern variable and has been
   // defined elsewhere (above the main     // function)
   extern int x;
    // printing the extern variables 'x'
   printf("Value of the variable 'x'"           " declared as extern: %d\n",         x);
    // value of extern variable x modified
   x = 2;
    // printing the modified values of      // extern variables 'x'
   printf("Modified value of the variable 'x'"           " declared as extern: %d\n",           x);
    printf("-----------------------------------");
}
 void staticStorageClass()
{
   int i = 0;
    printf("\nDemonstrating static class\n\n");       // using a static variable 'y'
   printf("Declaring 'y' as static inside the loop.\n"          "But this declaration will occur
only"          " once as 'y' is static.\n"          "If not, then every time the value of 'y' "
         "will be the declared value 5"          " as in the case of variable 'p'\n");
    printf("\nLoop started:\n");
    for (i = 1; i < 5; i++) {          // Declaring the static variable 'y'
      static int y = 5;          // Declare a non-static variable 'p'
      int p = 10;          // Incrementing the value of y and p by 1
      y++;        p++;          // printing value of y at each iteration
      printf("\nThe value of 'y', "  "declared as static, in %d "  "iteration is %d\n",  i, y);
        // printing value of p at each iteration
      printf("The value of non-static variable 'p', " "in %d iteration is %d\n",  i, p);
   }
```

```
    printf("\nLoop ended:\n");
    printf("-----------------------------------");
}
 int main()
{
   printf("A program to demonstrate"    " Storage Classes in C\n\n");
    // To demonstrate auto Storage Class
   autoStorageClass();       // To demonstrate register Storage Class
   registerStorageClass();       // To demonstrate extern Storage Class
   externStorageClass();       // To demonstrate static Storage Class
   staticStorageClass();       // exiting
   printf("\n\nStorage Classes demonstrated");
    return 0;
}
// This code is improved by RishabhPrabhu
```

Output:

A program to demonstrate Storage Classes in C

Demonstrating auto class

Value of the variable 'a' declared as auto: 32

Demonstrating register class

Value of the variable 'b' declared as register: 71

Demonstrating extern class

Value of the variable 'x' declared as extern: 0
Modified value of the variable 'x' declared as extern: 2

Demonstrating static class

Declaring 'y' as static inside the loop.
But this declaration will occur only once as 'y' is static.
If not, then every time the value of 'y' will be the declared value 5 as in the case of variable 'p'

Loop started:

The value of 'y', declared as static, in 1 iteration is 6
The value of non-static variable 'p', in 1 iteration is 11

The value of 'y', declared as static, in 2 iteration is 7
The value of non-static variable 'p', in 2 iteration is 11

The value of 'y', declared as static, in 3 iteration is 8
The value of non-static variable 'p', in 3 iteration is 11

The value of 'y', declared as static, in 4 iteration is 9
The value of non-static variable 'p', in 4 iteration is 11

Loop ended:

Storage Classes demonstrated

Type conversion:

https://www.geeksforgeeks.org/type-conversion-c/

A type cast is basically a conversion from one type to another. There are two types of type conversion:

1. **Implicit Type Conversion**

1. Also known as 'automatic type conversion'.
    1. Done by the compiler on its own, without any external trigger from the user.
    2. Generally takes place when in an expression more than one data type is present. In such condition type conversion (type promotion) takes place to avoid lose of data.
    3. All the data types of the variables are upgraded to the data type of the variable with largest data type.
    4.
    5. **bool -> char -> short int -> int ->**
    6. **unsigned int -> long -> unsigned ->**
    7. **long long -> float -> double -> long double**
    8. It is possible for implicit conversions to lose information, signs can be lost (when signed is implicitly converted to unsigned), and overflow can occur (when long long is implicitly converted to float).

    Example of Type Implicit Conversion:

filter_none
edit
play_arrow
brightness_4

```
// An example of implicit conversion
#include<stdio.h>
int main()
{
    int x = 10;    // integer x
```

```
    char y = 'a';  // character c

    // y implicitly converted to int. ASCII
    // value of 'a' is 97
    x = x + y;
        // x is implicitly converted to float
    float z = x + 1.0;
      printf("x = %d, z = %f", x, z);
    return 0;
}
```

Output:

```
x = 107, z = 108.000000
```

1. **Explicit Type Conversion**–

This process is also called type casting and it is user defined. Here the user can type cast the result to make it of a particular data type.

The syntax in C:

```
(type) expression
```

Type indicated the data type to which the final result is converted.

filter_none
edit
play_arrow
brightness_4

```
// C program to demonstrate explicit type casting
#include<stdio.h>

int main()
{
   double x = 1.2;

   // Explicit conversion from double to int
   int sum = (int)x + 1;

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

   return 0;
}
```

Output:

```
sum = 2
```

Advantages of Type Conversion

1. This is done to take advantage of certain features of type hierarchies or type representations.

2. It helps us to compute expressions containing variables of different data types.

The main method and command line arguments:-

{{{{{

The most important function of C is main() function. It is mostly defined with a return type of int and without parameters :

```
int main() { /* ... */ }
```

We can also give command-line arguments in C .Command-line arguments are given after the name of the program in command-line shell of Operating Systems.
To pass command line arguments, we typically define main() with two arguments : first argument is the number of command line arguments and second is list of command-line arguments.

```
int main(int argc, char *argv[]) { /* ... */ }
```

1. **argc (ARGument Count)** is int and stores number of command-line arguments passed by the user including the name of the program. So if we pass a value to a program, value of argc would be 2 (one for argument and one for program name)
2. The value of argc should be non negative.
3. **argv(ARGument Vector)** is array of character pointers listing all the arguments.
4. If argc is greater than zero,the array elements from argv[0] to argv[argc-1] will contain pointers to strings.
5. Argv[0] is the name of the program , After that till argv[argc-1] every element is command -line arguments.

Properties of Command Line Arguments:

1. They are passed to main() function.
2. They are parameters/arguments supplied to the program when it is invoked.
3. They are used to control program from outside instead of hard coding those values inside the code.
4. argv[0] holds the name of the program.
5. argv[1] points to the first command line argument and argv[n] points last argument.

Note : You pass all the command line arguments separated by a space, but if argument itself has a space then you can pass such arguments by putting them inside double quotes “” or single quotes ”.

Ex;- Helloworld arg1 arg2 “arg three”

```
// C program to illustrate
// command line arguments
```

```
#include<stdio.h>

int main(int argc,char* argv[])
{
   int counter;
   printf("Program Name Is: %s",argv[0]);
   if(argc==1)
      printf("\nNo Extra Command Line Argument Passed Other Than Program Name");

   if(argc>=2)
   {
      printf("\nNumber Of Arguments Passed: %d",argc);
      printf("\n----Following Are The Command Line Arguments Passed --- ");
      for(counter=0;counter<argc;counter++)
         printf("\nargv[%d]: %s",counter,argv[counter]);
   }
   return 0;
}
```

Output in different scenarios:

1. **Without argument:** When the above code is compiled and executed without passing any argument, it produces following output.

```
2.  $ ./a.out
3.  Program Name Is: ./a.out
4.  No Extra Command Line Argument Passed Other Than Program Name
```

2. **Three arguments :** When the above code is compiled and executed with a three arguments, it produces the following output.

```
5.  $ ./a.out First Second Third
6.  Program Name Is: ./a.out
7.  Number Of Arguments Passed: 4
8.  ----Following Are The Command Line Arguments Passed----
9.  argv[0]: ./a.out
10. argv[1]: First
11. argv[2]: Second
12. argv[3]: Third
```

3. **Single Argument :** When the above code is compiled and executed with a single argument separated by space but inside double quotes, it produces the following output.

```
13. $ ./a.out "First Second Third"
14. Program Name Is: ./a.out
```

```
15.  Number Of Arguments Passed: 2
16.  ----Following Are The Command Line Arguments Passed----
17.  argv[0]: ./a.out
18.  argv[1]: First Second Third
```

4. **Single argument in quotes separated by space :** When the above code is compiled and executed with a single argument separated by space but inside single quotes, it produces the following output.

```
$ ./a.out 'First Second Third'
Program Name Is: ./a.out
Number Of Arguments Passed: 2
----Following Are The Command Line Arguments Passed----
argv[0]: ./a.out
argv[1]: First Second Third
```

https://www.geeksforgeeks.org/find-largest-among-three-different-positive-numbers-using-command-line-argument/s

References:

http://www.cprogramming.com/tutorial/lesson14.html
http://c0x.coding-guidelines.com/5.1.2.2.1.html

}}}}}}}}}}

Command line argument is a parameter supplied to the program when it is invoked. Command line argument is an important concept in C programming. It is mostly used when you need to control your program from outside. Command line arguments are passed to the main() method.

Syntax:

```
int main(int argc, char *argv[])
```

Here argc counts the number of arguments on the command line and argv[ ] is a pointer array which holds pointers of type char which points to the arguments passed to the program.

Example for Command Line Argument

```
#include <stdio.h>
#include <conio.h>
int main(int argc, char *argv[])
```

```
{
   int i;
   if( argc >= 2 )
   {
      printf("The arguments supplied are:\n");
      for(i = 1; i < argc; i++)
      {
         printf("%s\t", argv[i]);
      }
   } else
   {
      printf("argument list is empty.\n");
   }
   return 0;}
```

Remember that argv[0] holds the name of the program and argv[1] points to the first command line argument and argv[n] gives the last argument. If no argument is supplied, argc will be 1.

Bitwise operations: Bitwise AND, OR, XOR and NOT operators Conditional Branching and Loops: Writing and evaluation of conditionals and consequent branching with if, ifelse, switch-case, ternary operator, goto, Iteration with for, while, do- while loops.I/O: Simple input and output with scanf and printf, formatted I/O, Introduction to stdin, stdout andstderr.

Bitwise operations

In arithmetic-logic unit (which is within the CPU), mathematical operations like: addition, subtraction, multiplication and division are done in bit-level. To perform bit-level operations in C programming, bitwise operators are used.

Operators	Meaning of operators
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR

Operators	Meaning of operators
~	Bitwise complement
<<	Shift left
>>	Shift right

Bitwise AND operator &

The output of bitwise AND is 1 if the corresponding bits of two operands is 1. If either bit of an operand is 0, the result of corresponding bit is evaluated to 0.

Let us suppose the bitwise AND operation of two integers 12 and 25.

```
12 = 00001100 (In Binary)

25 = 00011001 (In Binary)

Bit Operation of 12 and 25

  00001100

& 00011001

  00001000  = 8 (In decimal)
```

Example #1: Bitwise AND

```
1.    #include <stdio.h>
2.    int main()
3.    {
4.        int a = 12, b = 25;
5.        printf("Output = %d", a&b);
6.        return 0;
7.    }
```

Output

```
Output = 8
```

Bitwise OR operator |

The output of bitwise OR is 1 if at least one corresponding bit of two operands is 1. In C Programming, bitwise OR operator is denoted by |.

```
12 = 00001100 (In Binary)

25 = 00011001 (In Binary)

Bitwise OR Operation of 12 and 25

  00001100

| 00011001

  00011101  = 29 (In decimal)
```

Example #2: Bitwise OR

```
1.    #include <stdio.h>
2.    int main()
3.    {
4.        int a = 12, b = 25;
5.        printf("Output = %d", a|b);
6.        return 0;
7.    }
```

Output

```
Output = 29
```

Bitwise XOR (exclusive OR) operator ^

The result of bitwise XOR operator is 1 if the corresponding bits of two operands are opposite. It is denoted by ^.

```
12 = 00001100 (In Binary)

25 = 00011001 (In Binary)
```

```
Bitwise XOR Operation of 12 and 25

  00001100

^ 00011001

  00010101  = 21 (In decimal)
```

Example #3: Bitwise XOR

```
1.      #include <stdio.h>
2.      int main()
3.      {
4.          int a = 12, b = 25;
5.          printf("Output = %d", a^b);
6.          return 0;
7.      }
```

Output: Output = 21

Bitwise complement operator ~

Bitwise compliment operator is an unary operator (works on only one operand). It changes 1 to 0 and 0 to 1. It is denoted by ~.

```
35 = 00100011 (In Binary)

Bitwise complement Operation of 35

~ 00100011

  11011100  = 220 (In decimal)
```

Twist in bitwise complement operator in C Programming

The bitwise complement of 35 (~35) is -36 instead of 220, but why?

For any integer n, bitwise complement of n will be -(n+1). To understand this, you should have the knowledge of 2's complement.

2's Complement

Two's complement is an operation on binary numbers. The 2's complement of a number is equal to the complement of that number plus 1. For example:

Decimal	Binary	2's complement
0	00000000	-(11111111+1) = -00000000 = -0(decimal)
1	00000001	-(11111110+1) = -11111111 = -256(decimal)
12	00001100	-(11110011+1) = -11110100 = -244(decimal)
220	11011100	-(00100011+1) = -00100100 = -36(decimal)

Note: Overflow is ignored while computing 2's complement.

The bitwise complement of 35 is 220 (in decimal). The 2's complement of 220 is -36. Hence, the output is -36 instead of 220.

Bitwise complement of any number N is -(N+1). Here's how:

bitwise complement of N = ~N (represented in 2's complement form)

2'complement of ~N= -(~(~N)+1) = -(N+1)

Example #4: Bitwise complement

```
1.  #include <stdio.h>
2.  int main()
3.  {
4.      printf("Output = %d\n",~35);
5.      printf("Output = %d\n",~-12);
6.      return 0;
7.  }
```

Output

```
Output = -36
Output = 11
```

Shift Operators in C programming

There are two shift operators in C programming:

1. Right shift operator
2. Left shift operator.

Right Shift Operator

Right shift operator shifts all bits towards right by certain number of specified bits. It is denoted by >>.

```
212 = 11010100 (In binary)

212>>2 = 00110101 (In binary) [Right shift by two bits]

212>>7 = 00000001 (In binary)

212>>8 = 00000000

212>>0 = 11010100 (No Shift)
```

Left Shift Operator

Left shift operator shifts all bits towards left by certain number of specified bits. It is denoted by <<.

```
212 = 11010100 (In binary)

212<<1 = 110101000 (In binary) [Left shift by one bit]

212<<0 =11010100 (Shift by 0)
```

```
212<<4 = 110101000000 (In binary) =3392(In decimal)
```

Example #5: Shift Operators

```
1.      #include <stdio.h>
2.      int main()
3.      {
4.          int num=212, i;
5.          for (i=0; i<=2; ++i)
6.              printf("Right shift by %d: %d\n", i, num>>i);
7.           printf("\n");
8.           for (i=0; i<=2; ++i)
9.              printf("Left shift by %d: %d\n", i, num<<i);
10.          return 0;
11.     }
```

```
Right Shift by 0: 212
Right Shift by 1: 106
Right Shift by 2: 53

Left Shift by 0: 212
Left Shift by 1: 424
Left Shift by 2: 848
```

Conditional Branching and Loops:

C provides two sytles of flow control:

1. Branching
2. Looping

Branching is deciding what actions to take and looping is deciding how many times to take a certain action.

Branching:

Branching is so called because the program chooses to follow one branch or another.

if statement

This is the most simple form of the branching statements.

It takes an expression in parenthesis and an statement or block of statements. if the expression is true then the statement or block of statements gets executed otherwise these statements are skipped.

NOTE: Expression will be assumed to be true if its evaulated values is non-zero.

if statements take the following form:

Show Example

```
if (expression)
  statement;


or


if (expression)
  {
    Block of statements;
  }


or


if (expression)
  {
    Block of statements;
  }
else
  {
    Block of statements;
  }


or


if (expression)
```

```
 {
   Block of statements;
 }
else if(expression)
 {
   Block of statements;
 }
else
 {
   Block of statements;
 }
```

? : Operator

The ? : operator is just like an if ... else statement except that because it is an operator you can use it within expressions.

? : is a ternary operator in that it takes three values, this is the only ternary operator C has.

? : takes the following form:

Show Example

```
if condition is true ? then X return value : otherwise Y value;
```

switch statement:

The switch statement is much like a nested if .. else statement. Its mostly a matter of preference which you use, switch statement can be slightly more efficient and easier to read.

Show Example

```
switch( expression )
    {
      case constant-expression1:        statements1;
```

```
    [case constant-expression2:     statements2;]
    [case constant-expression3:     statements3;]
    [default : statements4;]
  }
```

Using break keyword:

If a condition is met in switch case then execution continues on into the next case clause also if it is not explicitly specified that the execution should exit the switch statement. This is achieved by using *break*keyword.

Try out given example Show Example

What is default condition:

If none of the listed conditions is met then default condition executed.

Looping

Loops provide a way to repeat commands and control how many times they are repeated. C provides a number of looping way.

The most basic loop in C is the while loop.A while statement is like a repeating if statement. Like an If statement, if the test condition is true: the statments get executed. The difference is that after the statements have been executed, the test condition is checked again. If it is still true the statements get executed again.This cycle repeats until the test condition evaluates to false.

Basic syntax of while loop is as follows:

Show Example

```
while ( expression )
{
  Single statement
  or
  Block of statements;
}
```

for loop

for loop is similar to while, it's just written differently. for statements are often used to proccess lists such a range of numbers:

Basic syntax of for loop is as follows:

Show Example

```
for( expression1; expression2; expression3)

{

  Single statement

  or

  Block of statements;

}
```

In the above syntax:

1. expression1 - Initialisese variables.
2. expression2 - Condtional expression, as long as this condition is true, loop will keep executing.
3. expression3 - expression3 is the modifier which may be simple increment of a variable.

do...while loop

do ... while is just like a while loop except that the test condition is checked at the end of the loop rather than the start. This has the effect that the content of the loop are always executed at least once.

Basic syntax of do...while loop is as follows:

Show Example

```
do

{

  Single statement

  or

  Block of statements;

}while(expression);
```

break and continue statements

C provides two commands to control how we loop:

1. break -- exit form loop or switch.
2. continue -- skip 1 iteration of loop.

Goto:

The goto statement is a jump statement which is sometimes also referred to as unconditional jump statement. The goto statement can be used to jump from anywhere to anywhere within a function.

A **goto** statement in C programming provides an unconditional jump from the 'goto' to a labeled statement in the same function.

NOTE − Use of **goto** statement is highly discouraged in any programming language because it makes difficult to trace the control flow of a program, making the program hard to understand and hard to modify. Any program that uses a goto can be rewritten to avoid them.

Syntax

The syntax for a **goto** statement in C is as follows −

```
goto label;
..
.
label: statement;
```

Here **label** can be any plain text except C keyword and it can be set anywhere in the C program above or below to **goto** statement.

Flow Diagram

ExampleLive Demo

```
#include <stdio.h>
int main () {
   /* local variable definition */
   int a = 10;
   /* do loop execution */
   LOOP:do {
      if( a == 15) {
         /* skip the iteration */
         a = a + 1;
         goto LOOP;
      }
      printf("value of a: %d\n", a);
      a++;
   }while( a < 20 );
```

```
   return 0;
}
```

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

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Ternary operator:

The ternary operator is used to execute code based on the result of a binary condition.

It takes in a binary condition as input, which makes it similar to an 'if-else' control flow block. It also, however, returns a value, behaving similar to a function.

Syntax

```
result =binaryCondition ? valueReturnedIfTrue:valueReturnedIfFalse;
```

Notes

The ternary cannot be used to execute code. It must be either returned in a function, or set equal to a variable with the same data type as the returned values.

Example

```
//Looking at the maximum example

int findMaximum(int a, int b){

   //if a > b, it returns a, if not it returns b

   return (a > b) ? a : b;

}
```

Simple input and output with scanf and printf,:

In C programming you can use **scanf**() and **printf**() predefined function to read and print data. The above program **scanf**() is used to take **input** from the user, and respectively **printf**() is used to display **output** result on the screen.

In C programming, printf() is one of the main output function. The function sends formatted output to the screen. For example,

Example 1: C Output

```
#include <stdio.h>
int main()
{
   // Displays the string inside quotations
   printf("C Programming");    return 0;
}
```

Output

```
C Programming
```

How does this program work?

1. All valid C programs must contain the main() function. The code execution begins from the start of the main() function.
2. The printf() is a library function to send formatted output to the screen. The function prints the string inside quotations.
3. To use printf() in our program, we need to include stdio.h header file using #inclue <stdio.h> statement.
4. The return 0; statement inside the main() function is the "Exit status" of the program. It's optional.

Example 2: Integer Output

```
#include <stdio.h>
int main()
{
   int testInteger = 5;    printf("Number = %d", testInteger);    return 0;
}
```

Output

```
Number = 5
```

We use %d format specifier to print int types. Here, the %d inside the quotations will be replaced by the value of testInteger.

Example 3: float and double Output

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

```
4.          float number1 = 13.5;    double number2 = 12.4;
5.          printf("number1 = %f\n", number1);
6.          printf("number2 = %lf", number2);
7.          return 0;
8.      }
```

Output

```
number1 = 13.500000
number2 = 12.400000
```

To print float, we use %f format specifier. Similarly, we use %lf to print double values.

Example 4: Print Characters

```
1.      #include <stdio.h>
2.      int main()
3.      {
4.          char chr = 'a';
5.          printf("character = %c.", chr);
6.          return 0;
7.      }
```

Output

```
character = a
```

Formatted I/O:

Concepts

1. I/O is essentially done one character (or byte) at a time
2. **stream** -- a sequence of characters flowing from one place to another
    1. *input stream*: data flows from input device (keyboard, file, etc) into memory
    2. *output stream*: data flows from memory to output device (monitor, file, printer, etc)
3. **Standard I/O streams** (with built-in meaning)
    1. stdin: standard input stream (default is keyboard)
    2. stdout: standard output stream (defaults to monitor)
    3. stderr: standard error stream
4. stdio.h -- contains basic I/O functions
    1. scanf: reads from standard input (stdin)
    2. printf: writes to standard output (stdout)
    3. There are other functions similar to printf and scanf that write to and read from other streams
    4. How to include, for C or C++ compiler
    5. #include <stdio.h> // for a C compiler
    6. #include <cstdio> // for a C++ compiler

5. *Formatted I/O* -- refers to the conversion of data to and from a stream of characters, for printing (or reading) in plain text format
    1. All text I/O we do is considered *formatted* I/O
    2. The other option is reading/writing direct binary information (common with file I/O, for example)

Printf:

1. The basic format of a **printf** function call is:
2. printf (*format_string*, *list_of_expressions*);

    where:

    1. *format_string* is the layout of what's being printed
    2. *list_of_expressions* is a comma-separated list of variables or expressions yielding results to be inserted into the output
3. To output string literals, just use one parameter on printf, the string itself
4. printf("Hello, world!\n");
   printf("Greetings, Earthling\n\n");

%d int (signed decimal integer)

%u unsigned decimal integer

%f floating point values (fixed notation) - float, double

%e floating point values (exponential notation)

%s string

%c character

Scanf:

1. To read data in from standard input (keyboard), we call the **scanf** function. The basic form of a call to scanf is:
2. scanf(*format_string*, *list_of_variable_addresses*);
    1. The format string is like that of printf
    2. But instead of expressions, we need space to store incoming data, hence the list of variable addresses
3. If **x** is a variable, then the expression **&x** means "address of x"
4. scanf example:
5. int month, day;
6.
7. printf("Please enter your birth month, followed by the day: ");
8. scanf("%d %d", &month, &day);

9. **Conversion Specifiers**
    1. Mostly the same as for output. Some small differences
    2. Use %f for type float, but use %lf for types double and long double
10. The data type read, the conversion specifier, and the variable used need to match in type
11. White space is skipped by default in consecutive *numeric* reads. But it is *not* skipped for character/string inputs.

Introduction to stdin, stdout andstderr.

When we say **Input**, it means to feed some data into a program. An input can be given in the form of a file or from the command line. C programming provides a set of built-in functions to read the given input and feed it to the program as per requirement.

When we say **Output**, it means to display some data on screen, printer, or in any file. C programming provides a set of built-in functions to output the data on the computer screen as well as to save it in text or binary files.

The Standard Files

C programming treats all the devices as files. So devices such as the display are addressed in the same way as files and the following three files are automatically opened when a program executes to provide access to the keyboard and screen.

Standard File	File Pointer	Device
Standard input	stdin	Keyboard
Standard output	stdout	Screen
Standard error	stderr	Your screen

The file pointers are the means to access the file for reading and writing purpose. This section explains how to read values from the screen and how to print the result on the screen.

The getchar() and putchar() Functions

The **int getchar(void)** function reads the next available character from the screen and returns it as an integer. This function reads only single character at a time. You can use this method in the loop in case you want to read more than one character from the screen.

The **int putchar(int c)** function puts the passed character on the screen and returns the same character. This function puts only single character at a time. You can use this method in the

loop in case you want to display more than one character on the screen. Check the following example −

```
#include <stdio.h>
int main( ) {
   int c;
   printf( "Enter a value :");
   c = getchar( );
   printf( "\nYou entered: ");
   putchar( c );
   return 0;
}
```

When the above code is compiled and executed, it waits for you to input some text. When you enter a text and press enter, then the program proceeds and reads only a single character and displays it as follows −

```
$./a.out
Enter a value : this is test
You entered: t
```

The gets() and puts() Functions

The **char *gets(char *s)** function reads a line from **stdin** into the buffer pointed to by **s** until either a terminating newline or EOF (End of File).

The **int puts(const char *s)** function writes the string 's' and 'a' trailing newline to **stdout**.

NOTE: Though it has been deprecated to use gets() function, Instead of using gets, you want to use fgets().

```
#include <stdio.h>
int main( ) {
   char str[100];
   printf( "Enter a value :");
   gets( str );
   printf( "\nYou entered: ");
   puts( str );
   return 0;
}
```

When the above code is compiled and executed, it waits for you to input some text. When you enter a text and press enter, then the program proceeds and reads the complete line till end, and displays it as follows −

```
$./a.out
Enter a value : this is test
You entered: this is test
```

The scanf() and printf() Functions

The **int scanf(const char *format, ...)** function reads the input from the standard input stream **stdin** and scans that input according to the **format** provided.

The **int printf(const char *format, ...)** function writes the output to the standard output stream **stdout** and produces the output according to the format provided.

The **format** can be a simple constant string, but you can specify %s, %d, %c, %f, etc., to print or read strings, integer, character or float respectively. There are many other formatting options available which can be used based on requirements. Let us now proceed with a simple example to understand the concepts better −

```
#include <stdio.h>
int main( ) {
   char str[100];
   int i;
   printf( "Enter a value :");
   scanf("%s %d", str, &i);
   printf( "\nYou entered: %s %d ", str, i);
   return 0;
}
```

When the above code is compiled and executed, it waits for you to input some text. When you enter a text and press enter, then program proceeds and reads the input and displays it as follows −

```
$./a.out
Enter a value : seven 7
You entered: seven 7
```

Here, it should be noted that scanf() expects input in the same format as you provided %s and %d, which means you have to provide valid inputs like "string integer". If you provide "string string" or "integer integer", then it will be assumed as wrong input. Secondly, while reading a string, scanf() stops reading as soon as it encounters a space, so "this is test" are three strings for scanf().

MODULE – II: ARRAYS, STRINGS, STRUCTURES AND PREPROCESSOR:

Arrays: one and two dimensional arrays, creating, accessing and manipulating elements of arrays.

Strings: Introduction to strings, handling strings as array of characters, basic string functions available in C (strlen, strcat, strcpy, strstr etc.), arrays of strings

Structures: Defining structures, initializing structures, unions, Array of structures

Preprocessor: Commonly used Preprocessor commands like include, define, undef, if, ifdef, ifndef.

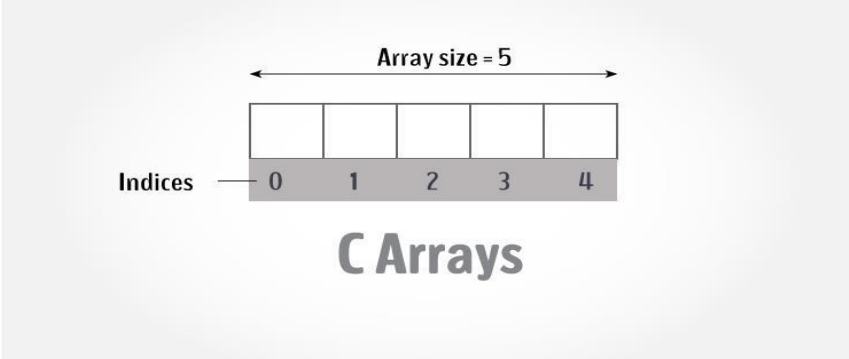
Arrays: one and two dimensional arrays

What is an Array?

ARRAY is a homogeneous collection of elements of same data types where the data types can be int, char, float etc….

A **one-dimensional array** (or **single dimension array**) is a type of linear **array**. Accessing its elements involves a **single** subscript which can either represent a row or column index. As an example consider the **C** declaration int anArrayName[10];

You will learn to declare, initialize and access array elements of an array with the help of examples.



An array is a variable that can store multiple values. For example, if you want to store 100 integers, you can create an array for it.

```
1. int data[100];
```

How to declare an array?

```
dataType arrayName[arraySize];
```

For example,

```
float mark[5];
```

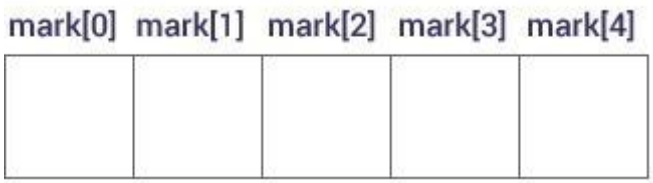
Here, we declared an array, `mark`, of floating-point type. And its size is 5. Meaning, it can hold 5 floating-point values.

It's important to note that, the size and type of an array cannot be changed once it is declared.

Access Array Elements

You can access elements of an array by indices.

Suppose you declared an array `mark` as above. The first element is `mark[0]`, the second element is `mark[1]` and so on.



Few keynotes:

- Arrays have 0 as the first index, not 1. In this example, `mark[0]` is the first element.
- If the size of an array is `n`, to access the last element, the `n-1` index is used. In this example, `mark[4]`
- Suppose the starting address of `mark[0]` is **2120d**. Then, the address of the `mark[1]` will be **2124d**. Similarly, the address of `mark[2]` will be **2128d** and so on.
  This is because the size of a `float` is 4 bytes.

How to initialize an array?

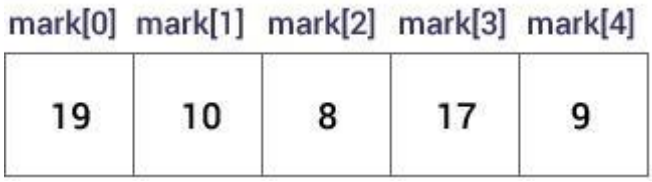
It is possible to initialize an array during declaration. For example,

```
int mark[5] = {19, 10, 8, 17, 9};
```

You can also initialize an array like this.

```
int mark[] = {19, 10, 8, 17, 9};
```

Here, we haven't specified the size. However, the compiler knows its size is 5 as we are initializing it with 5 elements.



Here,

mark[0] is equal to 19

mark[1] is equal to 10

mark[2] is equal to 8

mark[3] is equal to 17

mark[4] is equal to 9

Change Value of Array elements

```
int mark[5] = {19, 10, 8, 17, 9}

// make the value of the third element to -1
mark[2] = -1;

// make the value of the fifth element to 0
mark[4] = 0;
```

Input and Output Array Elements

Here's how you can take input from the user and store it in an array element.

```
// take input and store it in the 3rd element
scanf("%d", &mark[2]);

```

```
// take input and store it in the ith element
scanf("%d", &mark[i-1]);
```

```
// print the first element of the array
printf("%d", mark[0]);

// print the third element of the array
printf("%d", mark[2]);

// print ith element of the array
printf("%d", mark[i-1]);
```

Example 1: Array Input/Output

```
// Program to take 5 values from the user and store them in an array
// Print the elements stored in the array
#include <stdio.h>

int main() {
  int values[5];

  printf("Enter 5 integers: ");

  // taking input and storing it in an array
  for(int i = 0; i < 5; ++i) {
     scanf("%d", &values[i]);
  }

  printf("Displaying integers: ");

  // printing elements of an array
  for(int i = 0; i < 5; ++i) {
     printf("%d\n", values[i]);
  }
  return 0;
}
```

Output

```
Enter 5 integers: 1

-3
```

```
34

0

3

Displaying integers: 1

-3

34

0

3
```

Here, we have used a `for` loop to take 5 inputs from the user and store them in an array. Then, using another `for` loop, these elements are displayed on the screen.

Example 2: Calculate Average

```c
// Program to find the average of n numbers using arrays

#include <stdio.h>
int main()
{
    int marks[10], i, n, sum = 0, average;

    printf("Enter number of elements: ");
    scanf("%d", &n);

    for(i=0; i<n; ++i)
    {
        printf("Enter number%d: ",i+1);
        scanf("%d", &marks[i]);

        // adding integers entered by the user to the sum variable
        sum += marks[i];
    }

    average = sum/n;
    printf("Average = %d", average);
```

```
22.
23.     return 0;
24. }
```

Output

```
Enter n: 5

Enter number1: 45

Enter number2: 35

Enter number3: 38

Enter number4: 31

Enter number5: 49

Average = 39
```

Here, we have computed the average of `n` numbers entered by the user.

Access elements out of its bound!

Suppose you declared an array of 10 elements. Let's say,

```
int testArray[10];
```

You can access the array elements from `testArray[0]` to `testArray[9]`.

Now let's say if you try to access `testArray[12]`. The element is not available. This may cause unexpected output (undefined behavior). Sometimes you might get an error and some other time your program may run correctly.

Hence, you should never access elements of an array outside of its bound.

Multidimensional arrays

In this tutorial, you learned about arrays. These arrays are called one-dimensional arrays.

C programming language allows multidimensional arrays. Here is the general form of a multidimensional array declaration −

type name[size1][size2]...[sizeN];

For example, the following declaration creates a three dimensional integer array −

int threedim[5][10][4];

Two-dimensional Arrays

The simplest form of multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size [x][y], you would write something as follows −

type arrayName [x][y];

Where **type** can be any valid C data type and **arrayName** will be a valid C identifier. A two-dimensional array can be considered as a table which will have x number of rows and y number of columns. A two-dimensional array **a**, which contains three rows and four columns can be shown as follows −

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Thus, every element in the array **a** is identified by an element name of the form **a[i][j]**, where 'a' is the name of the array, and 'i' and 'j' are the subscripts that uniquely identify each element in 'a'.

Initializing Two-Dimensional Arrays

Multidimensional arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row has 4 columns.

```
int a[3][4] = {

  {0, 1, 2, 3} ,   /*  initializers for row indexed by 0 */

  {4, 5, 6, 7} ,   /*  initializers for row indexed by 1 */
```

```
  {8, 9, 10, 11}   /*  initializers for row indexed by 2 */

};
```

The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to the previous example −

int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};

Accessing Two-Dimensional Array Elements

An element in a two-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array. For example −

int val = a[2][3];

The above statement will take the 4th element from the 3rd row of the array. You can verify it in the above figure. Let us check the following program where we have used a nested loop to handle a two-dimensional array −

```
#include <stdio.h>


int main () {


  /* an array with 5 rows and 2 columns*/

  int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};

  int i, j;


  /* output each array element's value */

  for ( i = 0; i < 5; i++ ) {


    for ( j = 0; j < 2; j++ ) {
```

```
      printf("a[%d][%d] = %d\n", i,j, a[i][j] );

    }

  }


  return 0;

}
```

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

a[0][0]: 0

a[0][1]: 0

a[1][0]: 1

a[1][1]: 2

a[2][0]: 2

a[2][1]: 4

a[3][0]: 3

a[3][1]: 6

a[4][0]: 4

a[4][1]: 8

As explained above, you can have arrays with any number of dimensions, although it is likely that most of the arrays you create will be of one or two dimensions.

You can initialize a three-dimensional array in a similar way like a two-dimensional array. Here's an example,

```
1. int test[2][3][4] = {
2.    {{3, 4, 2, 3}, {0, -3, 9, 11}, {23, 12, 23, 2}},
3.    {{13, 4, 56, 3}, {5, 9, 3, 5}, {3, 1, 4, 9}}};
```

Example 1: Two-dimensional array to store and print values

```c
// C program to store temperature of two cities of a week and display it.
#include <stdio.h>
const int CITY = 2;
const int WEEK = 7;
int main()
{
  int temperature[CITY][WEEK];

  // Using nested loop to store values in a 2d array
  for (int i = 0; i < CITY; ++i)
  {
    for (int j = 0; j < WEEK; ++j)
    {
      printf("City %d, Day %d: ", i + 1, j + 1);
      scanf("%d", &temperature[i][j]);
    }
  }
 printf("\nDisplaying values: \n\n");

  // Using nested loop to display vlues of a 2d array
  for (int i = 0; i < CITY; ++i)
  {
    for (int j = 0; j < WEEK; ++j)
    {
      printf("City %d, Day %d = %d\n", i + 1, j + 1, temperature[i][j]);
    }
  }
  return 0;
}
```

Output

```
City 1, Day 1: 33

City 1, Day 2: 34

City 1, Day 3: 35

City 1, Day 4: 33

City 1, Day 5: 32

City 1, Day 6: 31
```

City 1, Day 7: 30

City 2, Day 1: 23

City 2, Day 2: 22

City 2, Day 3: 21

City 2, Day 4: 24

City 2, Day 5: 22

City 2, Day 6: 25

City 2, Day 7: 26

Displaying values:

City 1, Day 1 = 33

City 1, Day 2 = 34

City 1, Day 3 = 35

City 1, Day 4 = 33

City 1, Day 5 = 32

City 1, Day 6 = 31

City 1, Day 7 = 30

City 2, Day 1 = 23

City 2, Day 2 = 22

City 2, Day 3 = 21

City 2, Day 4 = 24

City 2, Day 5 = 22

City 2, Day 6 = 25

City 2, Day 7 = 26

Example 2: Sum of two matrices

```c
// C program to find the sum of two matrices of order 2*2

#include <stdio.h>
int main()
{
  float a[2][2], b[2][2], result[2][2];

  // Taking input using nested for loop
  printf("Enter elements of 1st matrix\n");
  for (int i = 0; i < 2; ++i)
    for (int j = 0; j < 2; ++j)
    {
      printf("Enter a%d%d: ", i + 1, j + 1);
      scanf("%f", &a[i][j]);
    }

  // Taking input using nested for loop
  printf("Enter elements of 2nd matrix\n");
  for (int i = 0; i < 2; ++i)
    for (int j = 0; j < 2; ++j)
    {
      printf("Enter b%d%d: ", i + 1, j + 1);
      scanf("%f", &b[i][j]);
    }

  // adding corresponding elements of two arrays
  for (int i = 0; i < 2; ++i)
    for (int j = 0; j < 2; ++j)
    {
      result[i][j] = a[i][j] + b[i][j];
    }

  // Displaying the sum
 printf("\nSum Of Matrix:");

  for (int i = 0; i < 2; ++i)
```

```
    for (int j = 0; j < 2; ++j)
    {
      printf("%.1f\t", result[i][j]);

      if (j == 1)
        printf("\n");
    }
  return 0;
}
```

Output

```
Enter elements of 1st matrix

Enter a11: 2;

Enter a12: 0.5;

Enter a21: -1.1;

Enter a22: 2;

Enter elements of 2nd matrix

Enter b11: 0.2;

Enter b12: 0;

Enter b21: 0.23;

Enter b22: 23;


Sum Of Matrix:

2.2     0.5

-0.9    25.0
```

Example 3: Three-dimensional array

```c
// C Program to store and print 12 values entered by the user

#include <stdio.h>
int main()
{
  int test[2][3][2];

  printf("Enter 12 values: \n");

  for (int i = 0; i < 2; ++i)
  {
    for (int j = 0; j < 3; ++j)
    {
      for (int k = 0; k < 2; ++k)
      {
        scanf("%d", &test[i][j][k]);
      }
    }
  }

  // Printing values with proper index.

  printf("\nDisplaying values:\n");
  for (int i = 0; i < 2; ++i)
  {
    for (int j = 0; j < 3; ++j)
    {
      for (int k = 0; k < 2; ++k)
      {
        printf("test[%d][%d][%d] = %d\n", i, j, k, test[i][j][k]);
      }
    }
  }

  return 0;
}
```

Output

```
Enter 12 values:

1
```

```
2

3

4

5

6

7

8

9

10

11

12


Displaying Values:

test[0][0][0] = 1

test[0][0][1] = 2

test[0][1][0] = 3

test[0][1][1] = 4

test[0][2][0] = 5

test[0][2][1] = 6

test[1][0][0] = 7

test[1][0][1] = 8

test[1][1][0] = 9

test[1][1][1] = 10
```

```
test[1][2][0] = 11

test[1][2][1] = 12
```

Accessing and manipulating elements of arrays:

An array is a group (or collection) of same data types. For example an int array holds the elements of int types while a float array holds the elements of float types.

Why we need Array in C Programming?

Consider a scenario where you need to find out the average of 100 integer numbers entered by user. In C, you have two ways to do this: 1) Define 100 variables with int data type and then perform 100 scanf() operations to store the entered values in the variables and then at last calculate the average of them. 2) Have a single integer array to store all the values, loop the array to store all the entered values in array and later calculate the average.
Which solution is better according to you? Obviously the second solution, it is convenient to store same data types in one single variable and later access them using array index.

How to declare Array in C

```
int num[35];  /* An integer array of 35 elements */

char ch[10];  /* An array of characters for 10 elements */
```

Similarly an array can be of any data type such as `double, float, short` etc.

How to access element of an array in C

You can use **array subscript** (or index) to access any element stored in array. Subscript starts with 0, which means `arr[0]` represents the first element in the array arr.

In general `arr[n-1]` can be used to access nth element of an array. where n is any integer number.

For example:

```
int mydata[20];
```

```
mydata[0] /* first element of array mydata*/

mydata[19] /* last (20th) element of array mydata*/
```

Example of Array In C programming to find out the average of 4 integers

```
#include <stdio.h>

int main()

{

   int avg = 0;

   int sum =0;

   int x=0;



   /* Array- declaration – length 4*/

   int num[4];



   /* We are using a for loop to traverse through the array

    * while storing the entered values in the array

    */

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

   {

      printf("Enter number %d \n", (x+1));

      scanf("%d", &num[x]);

   }

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

   {
```

```
        sum = sum+num[x];
    }


    avg = sum/4;
    printf("Average of entered number is: %d", avg);
    return 0;
}
```

Output:

```
Enter number 1
10
Enter number 2
10
Enter number 3
20
Enter number 4
40
Average of entered number is: 20
```

Lets discuss the important parts of the above program:

Input data into the array

Here we are **iterating the array** from 0 to 3 because the size of the array is 4. Inside the loop we are displaying a message to the user to enter the values. All the input values are stored in the corresponding array elements using scanf function.

```
for (x=0; x<4;x++)
```

```
{

   printf("Enter number %d \n", (x+1));

   scanf("%d", &num[x]);

}
```

Reading out data from an array

Suppose, if we want to display the elements of the array then we can use the for loop in C like this.

```
for (x=0; x<4;x++)

{

   printf("num[%d]\n", num[x]);

}
```

Various ways to initialize an array

In the above example, we have just declared the array and later we initialized it with the values input by user. However you can also initialize the array during declaration like this:

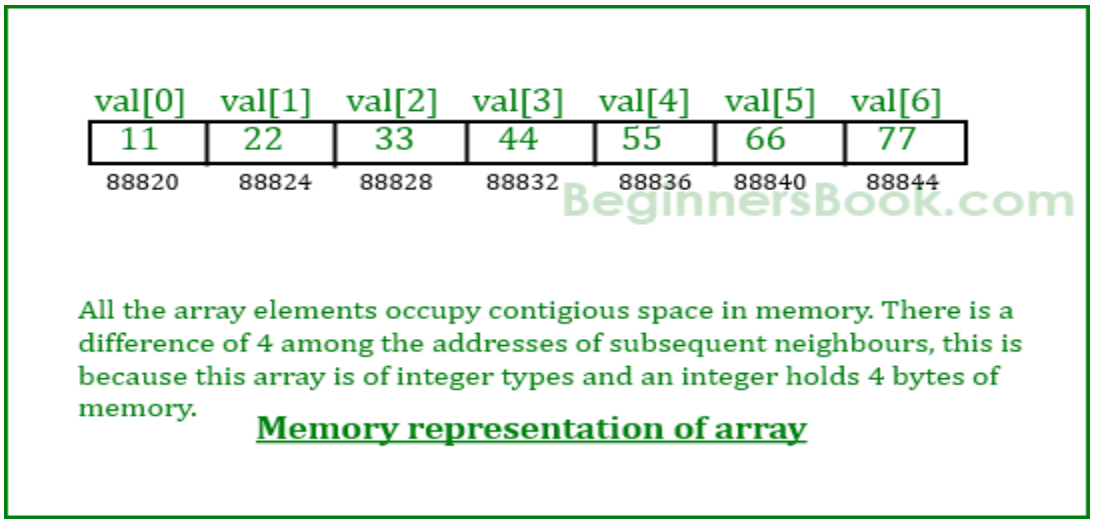
```
int arr[5] = {1, 2, 3, 4 ,5};
```

OR (both are same)

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

Un-initialized array always contain garbage values.

C Array – Memory representation



Arrays in Detail

Arrays are important to C and should need a lot more attention. The following important concepts related to array should be clear to a C programmer −

Sr.No.	Concept & Description
1	Multi-dimensional arrays: C supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array.
2	Passing arrays to functions ": You can pass to the function a pointer to an array by specifying the array's name without an index.
3	Return array from a function: C allows a function to return an array.
4	Pointer to an array : You can generate a pointer to the first element of an array by simply specifying the array name, without any index.

Vector:

An array (**vector**) is a common-place data type, used to hold and describe a collection of elements. These elements can be fetched at runtime by one or more indices (identifying keys). ... Using **C** as the **language** of implementation this post will guide you through building a simple **vector** data-structure.

http://www.how2lab.com/programming/c/array1.php

Basic array declaration and manipulation

An array is a data structure composed of a fixed number of components of the same type which are organized in a linear sequence. A component of an array is selected by assigning an integer value to its *index* (or *subscript*) which identifies the position of the component in the sequence. In C, arrays are intimately related to pointers. All sorts of array manipulations can also be performed by using pointers, which will be discussed later.

Array Declaration

The syntax of array declaration is as follows,

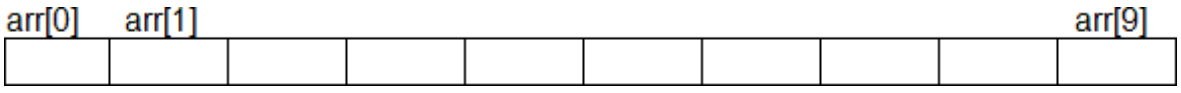
```
{<storage class>} data type <array_name>[expression]{[expression]};
```

In the above, curly braces indicate optional parts of the declaration.

Example:int arr[10];

The above declaration defines an array called *arr* of size 10, i.e., *arr* consists of 10 elements of same data type *int*. The elements occupy consecutive cells in the memory (each cell houses one element) and forms an ordered set of elements. Each element can be identified by its position in the array, and is also referred as the subscript of the array. The first element is at position 0 and n^{th} element can be found in the $(n-1)^{th}$ position.

The name of the array is *arr*, which contains the address of the first element (i.e. &arr[0]) of the array. However, an array name such as *arr* differs from an ordinary pointer variable (likeint *p;), because, it is static in nature and cannot point to a new memory location other than what it is pointing to by virtue of the declaration.



More examples of array declaration:

```
static float grade[10];

char name[20];
```

fflush(stdin):

fflush() is typically used for output stream only. Its purpose is to clear (or flush) the output buffer and move the buffered data to console (in case of stdout) or disk (in case of file output stream). Below is its syntax.

fflush(FILE *ostream);

ostream points to an output stream

or an update stream in which the

most recent operation was not input,

the fflush function causes any

unwritten data for that stream to

be delivered to the host environment

to be written to the file; otherwise,

the behavior is undefined.

Can we use it for input stream like stdin?
As per C standard, it is undefined behavior to use fflush(stdin). However some compilers like Microsoft visual studio allows it allow it. How is it used in these in these compilers? While taking an input string with spaces, the buffer does not get cleared for the next input and considers the previous input for the same. To solve this problem fflush(stdin) is. used to clear the stream/buffer.

// C program to illustrate situation

// where flush(stdin) is required only

// in certain compilers.

#include <stdio.h>

#include<stdlib.h>

int main()

{

```
    char str[20];
    int i;
    for (i=0; i<2; i++)
    {
        scanf("%[^\n]s", str);
        printf("%s\n", str);
        // fflush(stdin);
    }
    return 0;
}
```

Input:

```
geeks
geeksforgeeks
```

Output:

```
geeks
geeks
```

The code above takes only single input and gives the same result for the second input. Reason is because as the string is already stored in the buffer i.e. stream is not cleared yet as it was expecting string with spaces or new line. So, to handle this situation fflush(stdin) is used.

```
// C program to illustrate flush(stdin)
// This program works as expected only
// in certain compilers like Microsoft
// visual studio.
#include <stdio.h>
```

```
#include<stdlib.h>

int main()

{

    char str[20];

    int i;

    for (i = 0; i<2; i++)

    {

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

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


        // used to clear the buffer

        // and accept the next string

        fflush(stdin);

    }

    return 0;

}
```

Input:

```
geeks

geeksforgeeks
```

Output:

```
geeks

geeksforgeeks
```

Is it good to use fflush(stdin)?

Although using “fflush(stdin)” after “scanf()” statement also clears the input buffer in certain compilers, it is not recommended to use it as it is undefined behavior by language standard. In C and C++, we have different methods to clear the buffer.

Array Manipulation

The most convenient way of performing array manipulation is to use the *for repetitive construct* (for loop) for accessing elements of the array.

The following example illustrates the usage of an array in implementing addition of two vectors:

```
#include <stdio.h>

#define dimension 100

typedef int vector[dimension];



main()

{

        vector vect_1, vect_2, result_vect;

        int i,n;

        printf("Enter the Vector dimension: ");

        scanf("%d", &n); fflush(stdin);



        /* Accept values for the two arrays vect_1 & vect_2 */

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

        {

                printf("Give Vector element #%d: ", i+1);

                scanf("%d %d", &vect_1[i], &vect_2[i]); fflush(stdin);

                result_vect[i] = 0; //initialize elements of array result_vect to 0
```

```
        }


        /* Add vectors and create resultant vector */
        for(i=0; i < n; i++)
                result_vect[i] += (vect_1[i] + vect_2[i]);


        /* Display the resultant matrix */
        printf("\n\n");
        for(i = 0; i < n; i++)
        {
                printf("%d\t", result_vect[i]);
        }
        printf("\n");
}
```

How to initialize array elements during declaration?

Array initialization can be performed in the following way:

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

Note, however, that this definition-cum-initialization is permitted only in case of *external variable* definition. To include such initialization statement inside a function, storage class clause *static* has to be used as a prefix. See examples below.

Describe the output generated by each of the following programs.

```
#include <stdio.h>

main()

{

        int i;

        int a, sum = 0;


        //declaring & initializing within a function

        static int x[10] = {9,8,7,6,5,4,3,2,0};


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

                if((i % 2) == 0)

                        sum += x[i];

        printf("%d", sum);

}
#include <stdio.h>

#define  ROWS  3

#define  COLS  4


//external declaration & initialization

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


main()

{
```

```
        int i, j, max;
        for(i = 0; i < ROWS; ++i)
        {
                max = 9999;
                for(j = 0; j < COLS; ++j)
                        if(x[i][j] < max)
                                max = z[i][j];
                printf("%d", max);
        }
}
```

Exercises

Identify the errors in the following C program (if any) which initializes an array such that each of its ten elements is assigned with 0 value.

```
main()
{
        int num_arr[10], i = 0;
        ...
        ...
        for(; ++i < 10; num_arr[i] = 0);
}
```

Identify the array defined in each of the following statements. Indicate what values are assigned to the individual array elements.

```
char game[7] = {'C', 'R', 'I', 'C', 'K', 'E', 'T'};

char match[] = "Football";
```

Write an appropriate array definition for each of the following cases:

a. Define a one dimensional, integer array called **A** with 10 elements and initialize the array elements with **2, 5, 8, 11, ... , 29**.
b. Create a one dimensional, four element character array called **object** and assign the characters **'C', 'I', 'R', 'C', 'L' and 'E'** to the array elements.
c. Define a one dimensional, six element floating point array called **flt_const** having following initials values - **2.005, -3.05452, -1e-4, 340.179, 0.3e8, 0.023415**

Lab work

Write a C program that will accept a line of text as input, store it in an array, and then print it backwards. Assume that the length of the string cannot exceed 80 characters, and while entering the data, the string will be terminated by a carriage return. Test the program by entering a suitable message.

https://overiq.com/c-programming-101/array-of-strings-in-c/

What is an Array of Strings?

A string is a 1-D array of characters, so an array of strings is a 2-D array of characters.

Just like we can create a 2-D array of int, float etc; we can also create a 2-D array of character or array of strings. Here is how we can declare a 2-D array of characters.

```
1 char ch_arr[3][10] = {

2                   {'s', 'p', 'i', 'k', 'e', '\0'},

3                   {'t', 'o', 'm','\0'},
```

```
4                   {'j', 'e', 'r', 'r', 'y','\0'}

5               };
```

It is important to end each 1-D array by the null character, otherwise, it will be just an array of characters. We can't use them as strings.

Declaring an array of strings this way is rather tedious, that's why C provides an alternative syntax to achieve the same thing. This above initialization is equivalent to:

```
1 char ch_arr[3][10] = {

2                   "spike",

3                   "tom",

4                   "jerry"

5               };
```

The first subscript of the array i.e 3 denotes the number of strings in the array and the second subscript denotes the maximum length of the string. Recall the that in C, each character occupies 1 byte of data, so when the compiler sees the above statement it allocates 30 bytes (3*10) of memory.

We already know that the name of an array is a pointer to the 0th element of the array. Can you guess the type of ch_arr?

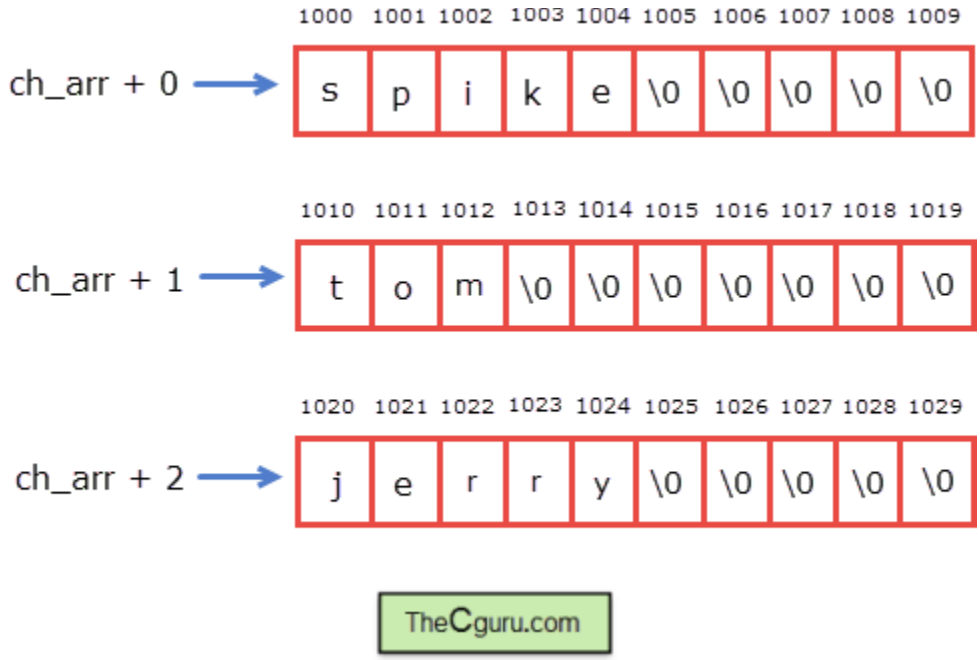
The ch_arr is a pointer to an array of 10 characters or int(*)[10].

Therefore, if ch_arr points to address 1000 then ch_arr + 1 will point to address 1010.

From here we can conclude that:

ch_arr + 0 points to the 0th string or 0th 1-D array.
ch_arr + 1 points to the 1st string or 1st 1-D array.
ch_arr + 2 points to the 2nd string or 2nd 1-D array.

In general, ch_arr + i points to the ith string or ith 1-D array.



We know that when we dereference a pointer to an array, we get the base address of the array. So, on dereferencing ch_arr + i we get the base address of the 0th 1-D array.

From this we can conclude that:

*(ch_arr + 0) + 0 points to the 0th character of 0th 1-D array (i.e s)
*(ch_arr + 0) + 1 points to the 1st character of 0th 1-D array (i.e p)
*(ch_arr + 1) + 2 points to the 2nd character of 1st 1-D array (i.e m)

In general, we can say that:

*(ch_arr + i) + j points to the jth character of ith 1-D array.

Note that the base type of *(ch_arr + i) + j is a pointer to char or (char*), while the base type of ch_arr + i is array of 10 characters or int(*)[10].

To get the element at jth position of ith 1-D array just dereference the whole expression*(ch_arr + i) + j.

1 *(*(ch_arr + i) + j)

We have learned in chapter Pointers and 2-D arrays that in a 2-D array the pointer notation is equivalent to subscript notation. So the above expression can be written as follows:

1 ch_arr[i][j]

The following program demonstrates how to print an array of strings.

```
 1  #include<stdio.h>
 2
 3  int main()
 4  {
 5      int i;
 6
 7      char ch_arr[3][10] = {
 8                              "spike",
 9                              "tom",
10                              "jerry"
11                          };
12
13      printf("1st way \n\n");
14
15      for(i = 0; i < 3; i++)
16      {
17          printf("string = %s \t address = %u\n", ch_arr + i, ch_arr + i);
18      }
19
20      // signal to operating system program ran fine
```

```
21      return 0;
22 }
```

Expected Output:

```
1 string = spike address = 2686736
2 string = tom address = 2686746
3 string = jerry address = 2686756
```

Some invalid operation on an Array of string

```
1 char ch_arr[3][10] = {
2                       {'s', 'p', 'i', 'k', 'e', '\0'},
3                       {'t', 'o', 'm','\0'},
4                       {'j', 'e', 'r', 'r', 'y','\0'}
5                   };
```

It allocates 30 bytes of memory. The compiler will do the same thing even if we don't initialize the elements of the array at the time of declaration.

We already know that the name of an array is a constant pointer so the following operations are invalid.

```
1 ch_arr[0] = "tyke"; // invalid
2 ch_arr[1] = "dragon"; // invalid
```

Here we are trying to assign a string literal (a pointer) to a constant pointer which is obviously not possible.

To assign a new string to ch_arr use the following methods.

```
1 strcpy(ch_arr[0], "type"); // valid
2 scanf(ch_arr[0], "type"); // valid
```

Let's conclude this chapter by creating another simple program.

This program asks the user to enter a username. If the username entered is one of the names in the master list then the user is allowed to calculate the factorial of a number. Otherwise, an error message is displayed.

```
 1  #include<stdio.h>
 2  #include<string.h>
 3  int factorial(int );
 4
 5  int main()
 6  {
 7      int i, found = 0, n;
 8
 9      char master_list[5][20] = {
10                                  "admin",
11                                  "tom",
12                                  "bob",
13                                  "tim",
14                                  "jim"
15                                }, name[10];
16
17      printf("Enter username: ");
18      gets(name);
19
20      for(i = 0; i < 5; i++)
21      {
```

```
22        if(strcmp(name, master_list[i]) == 0 )
23        {
24            found = 1;
25            break;
26        }
27    }
28
29    if(found==1)
30    {
31        printf("\nWelcome %s !\n", name);
32        printf("\nEnter a number to calculate the factorial: ");
33        scanf("%d", &n);
34        printf("Factorial of %d is %d", n, factorial(n));
35    }
36
37    else
38    {
39        printf("Error: You are not allowed to run this program.", name);
40    }
41
42    // signal to operating system program ran fine
43    return 0;
44 }
```

```
45
46 int factorial(int n)
47 {
48     if(n == 0)
49     {
50         return 1;
51     }
52
53     else
54     {
55         return n * factorial(n-1);
56     }
57 }
```

Expected Output:

1st run:

```
1 Enter username: admin
2
3 Welcome admin !
4
5 Enter a number to calculate the factorial: 4
6 Factorial of 4 is 24
```

Strings: Introduction to strings, handling strings as array of characters, basic string functions available in C (strlen, strcat, strcpy, strstr etc.), arrays of strings

Strings in C

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

Declaration of strings: Declaring a string is as simple as declaring a one dimensional array. Below is the basic syntax for declaring a string.

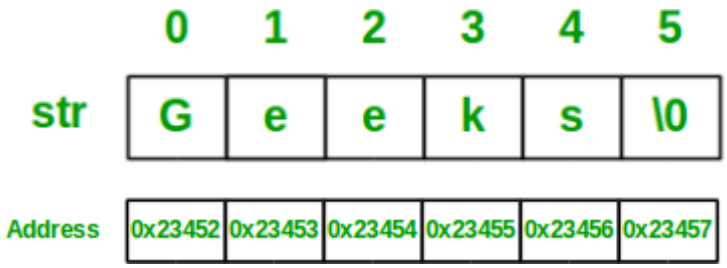
```
char str_name[size];
```

In the above syntax str_name is any name given to the string variable and size is used define the length of the string, i.e the number of characters strings will store. Please keep in mind that there is an extra terminating character which is the Null character ('\0') used to indicate termination of string which differs strings from normal character arrays.

Initializing a String: A string can be initialized in different ways. We will explain this with the help of an example. Below is an example to declare a string with name as str and initialize it with "GeeksforGeeks".

```
1. char str[] = "GeeksforGeeks";


2. char str[50] = "GeeksforGeeks";


3. char str[] = {'G','e','e','k','s','f','o','r','G','e','e','k','s','\0'};


4. char str[14] = {'G','e','e','k','s','f','o','r','G','e','e','k','s','\0'};
```

Below is the memory representation of a string "Geeks".



Let us now look at a sample program to get a clear understanding of declaring and initializing a string in C and also how to print a string.

filter_none

edit

play_arrow

brightness_4

```
// C program to illustrate strings


#include<stdio.h>


int main()

{

    // declare and initialize string

    char str[] = "Geeks";


    // print string

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

```
    return 0;

}
```

Output:

```
Geeks
```

We can see in the above program that strings can be printed using a normal printf statements just like we print any other variable. Unlike arrays we do not need to print a string, character by character. The C language does not provide an inbuilt data type for strings but it has an access specifier "**%s**" which can be used to directly print and read strings.

Below is a sample program to read a string from user:

filter_none

brightness_4

```
// C program to read strings


#include<stdio.h>


int main()

{

    // declaring string

    char str[50];


    // reading string

    scanf("%s",str);


    // print string
```

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


    return 0;

}
```

You can see in the above program that string can also be read using a single scanf statement. Also you might be thinking that why we have not used the '&' sign with string name 'str' in scanf statement! To understand this you will have to recall your knowledge of scanf. We know that the '&' sign is used to provide the address of the variable to the scanf() function to store the value read in memory. As str[] is a character array so using str without braces '[' and ']' will give the base address of this string. That's why we have not used '&' in this case as we are already providing the base address of the string to scanf.

Passing strings to function: As strings are character arrays, so we can pass strings to function in a same way we pass an array to a function. Below is a sample program to do this:

filter_none

edit

play_arrow

brightness_4

```
// C program to illustrate how to

// pass string to functions

#include<stdio.h>


void printStr(char str[])

{

    printf("String is : %s",str);

}
```

```
int main()

{

    // declare and initialize string

    char str[] = "GeeksforGeeks";


    // print string by passing string

    // to a different function

    printStr(str);


    return 0;

}
```

Output:

```
String is : GeeksforGeeks
```

Introduction to strings:

Strings in C. **Strings** are defined as an array of characters. The difference between a character array and a **string** is the **string** is terminated with a special character '\0'. Declaration of **strings**: Declaring a **string** is as simple as declaring a one dimensional array.

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

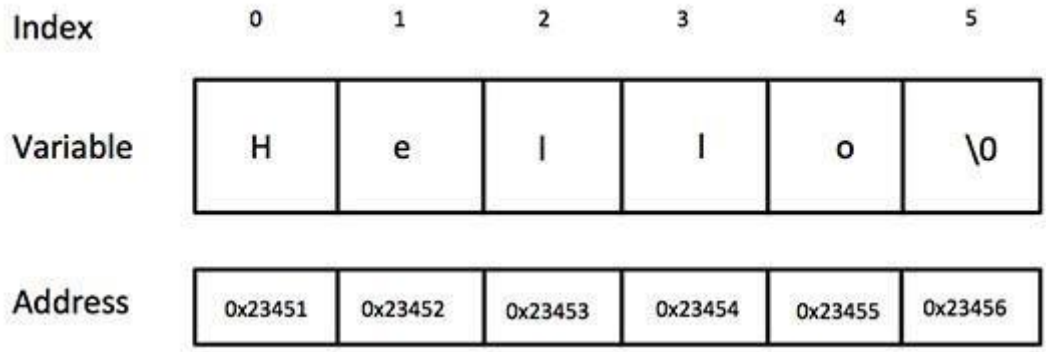
The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

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

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

char greeting[] = "Hello";

Following is the memory presentation of the above defined string in C/C++ −



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

```
#include <stdio.h>


int main () {


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

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

   return 0;

}
```

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

```
Greeting message: Hello
```

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

Sr.No.	Function & Purpose
1	**strcpy(s1, s2);**Copies string s2 into string s1.
2	**strcat(s1, s2);**Concatenates string s2 onto the end of string s1.
3	**strlen(s1);**Returns the length of string s1.
4	**strcmp(s1, s2);**Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	**strchr(s1, ch);**Returns a pointer to the first occurrence of character ch in string s1.
6	**strstr(s1, s2);**Returns a pointer to the first occurrence of string s2 in string s1.

The following example uses some of the above-mentioned functions −

```
#include <stdio.h>

#include <string.h>

int main () {

   char str1[12] = "Hello";

   char str2[12] = "World";

   char str3[12];

   int  len ;

   /* copy str1 into str3 */

   strcpy(str3, str1);

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

   /* concatenates str1 and str2 */

   strcat( str1, str2);

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

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

```
   len = strlen(str1);

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

   return 0;

}
```

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

```
strcpy( str3, str1) :  Hello

strcat( str1, str2):   HelloWorld

strlen(str1) :  10
```

Handling strings as array of characters:

String and Character Array

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

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

Declaring and Initializing a string variables

There are different ways to initialize a character array variable.

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

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

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

Some examples of illegal initialization of character array are,

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

```
char str[4];

str = "hell";   // Illegal
```

String Input and Output

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

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

```
#include<stdio.h>

#include<string.h>

void main()

{

    char str[20];

    printf("Enter a string");

    scanf("%[^\n]", &str);  //scanning the whole string, including the white spaces

    printf("%s", str);

}
```

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

```
char text[20];

gets(text);

printf("%s", text);
```

String Handling Functions

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

The following are the most commonly used string handling functions.

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

strcat() function

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

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

strlen() function

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

```
int j;

j = strlen("studytonight");

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

output:12

strcmp() function

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

```
int j;
```

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

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

Output: -1

strcpy() function

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

```
#include<stdio.h>

#include<string.h>

int main()

{

    char s1[50];

    char s2[50];

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

    strcpy(s2, s1);     //copies string s1 to string s2

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

        return(0);

}
```

Output:

StudyTonight

strrev() function

It is used to reverse the given string expression.

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

```
int main()
{
   char s1[50];
     printf("Enter your string: ");
   gets(s1);
   printf("\nYour reverse string is: %s",strrev(s1));
   return(0);
}
```

Output:

Enter your string: studytonight

Your reverse string is: thginotyduts

Arrays of strings:

https://www.codingame.com/playgrounds/14213/how-to-play-with-strings-in-c/array-of-c-string

If a C string is a one dimensional character array then what's an array of C string looks like? It's a two dimensional character array!

Here is how an array of C string can be initialized:

```
#define NUMBER_OF_STRING 4
#define MAX_STRING_SIZE 40

char arr[NUMBER_OF_STRING][MAX_STRING_SIZE] =
{ "array of c string",
```

```
  "is fun to use",

  "make sure to properly",

  "tell the array size"

};
```

Since all but the highest dimension can be omitted from an array declaration, the above declaration can be reduced to:

```
#define MAX_STRING_SIZE 40 char arr[][MAX_STRING_SIZE] = { "array of c string", "is fun to use", "make sure to properly", "tell the array size" };
```

But it is a good practice to specify size of both dimensions.

Now each arr[x] is a C string and each arr[x][y] is a character. You can use any function on arr[x] that works on string!

Following example prints all strings in a string array with their lengths.

```
#define NUMBER_OF_STRING 4

#define MAX_STRING_SIZE 40

char arr[NUMBER_OF_STRING][MAX_STRING_SIZE] = { "array of c string", "is fun to use", "make sure to properly", "tell the array size" };

for (int i = 0; i < NUMBER_OF_STRING; i++)

{

        printf("'%s' has length %d\n", arr[i], strlen(arr[i]));

}
```

Following example reverses all strings in a string array:

```
#include <stdio.h>

#include <string.h>

#define NUMBER_OF_STRING 4

#define MAX_STRING_SIZE 40
```

```
void print_array(const char arr[NUMBER_OF_STRING][MAX_STRING_SIZE])
{
	for (int i = 0; i < NUMBER_OF_STRING; i++)
	{
		printf("'%s' has length %d\n", arr[i], strlen(arr[i]));
	}
}
int main()
{
	char arr[NUMBER_OF_STRING][MAX_STRING_SIZE] =
	{ "array of c string",
	  "is fun to use",
	  "make sure to properly",
	  "tell the array size"
	};
	printf("Before reverse:\n");
	print_array(arr);
	for (int i = 0; i < NUMBER_OF_STRING; i++)
	{
		for (int j = 0, k = strlen(arr[i]) - 1; j < k; j++, k--)
		{
			char temp = arr[i][j];
			arr[i][j] = arr[i][k];
```

```
                arr[i][k] = temp;
            }
        }
        printf("\nAfter reverse:\n");
        print_array(arr);
        return 0;
}
```

Output:

Following example concatenates all strings in a string array with space between them into a single string:

```
#include <stdio.h>
#include <string.h>
#define NUMBER_OF_STRING 4
#define MAX_STRING_SIZE 40
#define DEST_SIZE 100
int main(){
        char arr[NUMBER_OF_STRING][MAX_STRING_SIZE] =
        { "array of c string",
          "is fun to use",
          "make sure to properly",
          "tell the array size"
        };
```

```
    char dest[DEST_SIZE] = "";
    for (int i = 0; i < NUMBER_OF_STRING; i++)
    {
        strcat(dest, arr[i]);
        if (i < NUMBER_OF_STRING - 1)
        {
            strcat(dest, " ");
        }
    }
    printf(dest);
    return 0;
}
```

Output:

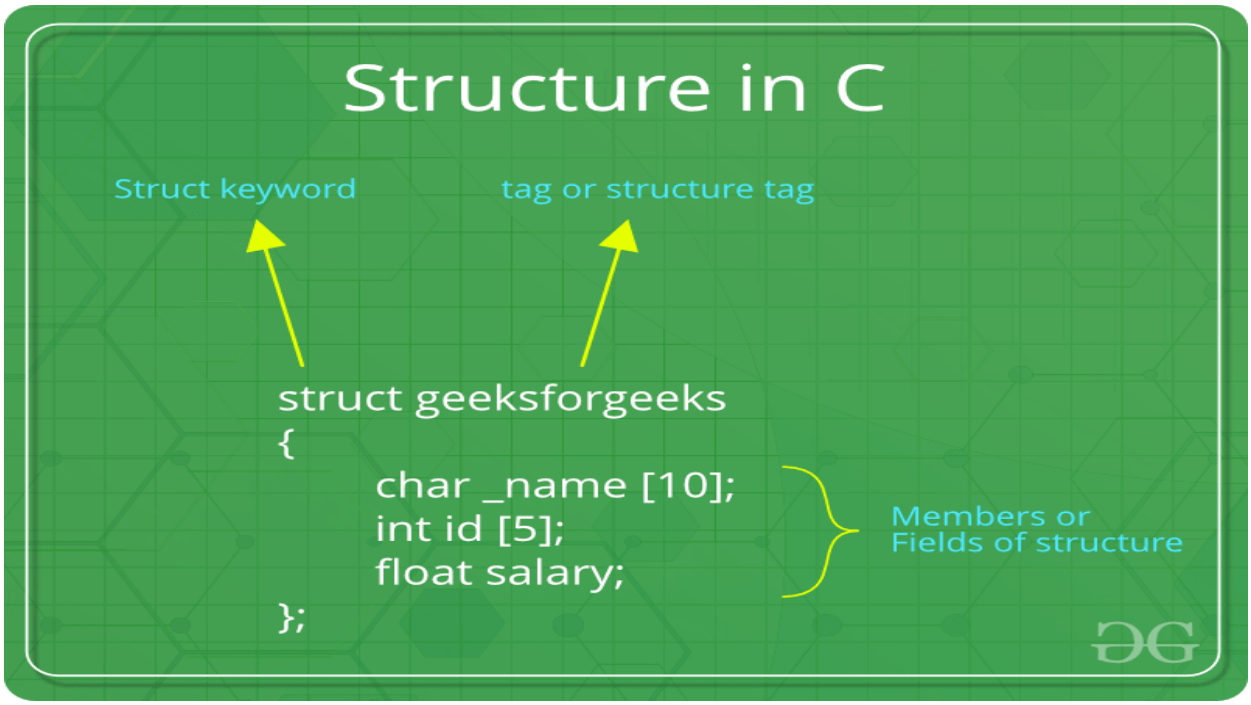
Structures: Defining structures, initializing structures, unions, Array of structures

Structures

Structure is a user-defined datatype in **C** language which allows us to combine data of different types together. **Structure** helps to construct a complex data type which is more meaningful. ... In **structure**, data is stored in form of records.

What is a structure?

A structure is a user defined data type in C/C++. A structure creates a data type that can be used to group items of possibly different types into a single type.



How to create a structure?
'struct' keyword is used to create a structure. Following is an example.

```
struct address

{

  char name[50];

  char street[100];

  char city[50];

  char state[20];

  int pin;

};
```

How to declare structure variables?
A structure variable can either be declared with structure declaration or as a separate declaration like basic types.

```
// A variable declaration with structure declaration.

struct Point

{

  int x, y;

} p1;  // The variable p1 is declared with 'Point'



// A variable declaration like basic data types

struct Point

{

  int x, y;

};


int main()

{

  struct Point p1;  // The variable p1 is declared like a normal variable

}
```

Note: In C++, the struct keyword is optional before in declaration of a variable. In C, it is mandatory.

How to initialize structure members?
Structure members **cannot be** initialized with declaration. For example the following C program fails in compilation.

```
struct Point
```

```
{
  int x = 0;  // COMPILER ERROR:  cannot initialize members here
  int y = 0;  // COMPILER ERROR:  cannot initialize members here
};
```

The reason for above error is simple, when a datatype is declared, no memory is allocated for it. Memory is allocated only when variables are created.

Structure members **can be** initialized using curly braces '{}'. For example, following is a valid initialization.

```
struct Point
{
  int x, y;
};

int main()
{
  // A valid initialization. member x gets value 0 and y
  // gets value 1.  The order of declaration is followed.
  struct Point p1 = {0, 1};
}
```

How to access structure elements?

Structure members are accessed using dot (.) operator.

```
#include<stdio.h>

struct Point
```

```
{
   int x, y;
};

int main()
{
   struct Point p1 = {0, 1};

   // Accesing members of point p1
   p1.x = 20;
   printf ("x = %d, y = %d", p1.x, p1.y);

   return 0;
}
```

Output:

```
x = 20, y = 1
```

What is designated Initialization?
Designated Initialization allows structure members to be initialized in any order. This feature has been added in C99 standard.

```
#include<stdio.h>
  struct Point
{
   int x, y, z;
```

```
};

 int main()

{

  // Examples of initializtion using designated initialization

  struct Point p1 = {.y = 0, .z = 1, .x = 2};

  struct Point p2 = {.x = 20};

    printf ("x = %d, y = %d, z = %d\n", p1.x, p1.y, p1.z);

  printf ("x = %d", p2.x);

  return 0;

}
```

Output:

```
x = 2, y = 0, z = 1

x = 20
```

This feature is not available in C++ and works only in C.

What is an array of structures?

Like other primitive data types, we can create an array of structures.

```
#include<stdio.h>


struct Point

{

  int x, y;

};
```

```
int main()

{

  // Create an array of structures

  struct Point arr[10];


  // Access array members

  arr[0].x = 10;

  arr[0].y = 20;


  printf("%d %d", arr[0].x, arr[0].y);

  return 0;

}
```

Output:

```
10 20
```

What is a structure pointer?

Like primitive types, we can have pointer to a structure. If we have a pointer to structure, members are accessed using arrow (->) operator.

```
#include<stdio.h>


struct Point

{

  int x, y;

};
```

```
int main()

{

  struct Point p1 = {1, 2};


  // p2 is a pointer to structure p1

  struct Point *p2 = &p1;


  // Accessing structure members using structure pointer

  printf("%d %d", p2->x, p2->y);

  return 0;

}
```

Output:

```
1 2
```

What is structure member alignment?
See https://www.geeksforgeeks.org/structure-member-alignment-padding-and-data-packing/

Limitations of C Structures

In C language, Structures provide a method for packing together data of different types. A Structure is a helpful tool to handle a group of logically related data items. However, C structures have some limitations.

 The C structure does not allow the struct data type to be treated like built-in data types:

 We cannot use operators like +,- etc. on Structure variables. For example, consider the following code:

```
struct number

{
```

```
    float x;
};
int main()
{
    struct number n1,n2,n3;
    n1.x=4;
    n2.x=3;
    n3=n1+n2;
    return 0;
}

/*Output:

prog.c: In function 'main':
prog.c:10:7: error:
invalid operands to binary + (have 'struct number' and 'struct number')
  n3=n1+n2;

*/
```

C Structures

Structure is a user-defined datatype in C language which allows us to combine data of different types together. Structure helps to construct a complex data type which is more meaningful. It is

somewhat similar to an Array, but an array holds data of similar type only. But structure on the other hand, can store data of any type, which is practical more useful.

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

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

Defining a structure

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

Syntax:

```
struct [structure_tag]

{

   //member variable 1

   //member variable 2

   //member variable 3

   ...

}[structure_variables];
```

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

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

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

Example of Structure

```
struct Student
```

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

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

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

Declaring Structure Variables

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

1) Declaring Structure variables separately

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

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

2) Declaring Structure variables with structure definition

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

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

Accessing Structure Members

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

For example:

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

struct Student
{
   char name[25];
```

```
    int age;
    char branch[10];
    //F for female and M for male
    char gender;
};


int main()
{
    struct Student s1;


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

```
    printf("Age of Student 1: %d\n", s1.age);


    return 0;
}
```

```
Name of Student 1: Viraaj

Age of Student 1: 18
```

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

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

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

Structure Initialization

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

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


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

or,

```
struct Patient p1;
```

```
p1.height = 180.75;     //initialization of each member separately

p1.weight = 73;

p1.age = 23;
```

Array of Structure

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

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

```
#include<stdio.h>



struct Employee

{

    char ename[10];

    int sal;

};



struct Employee emp[5];

int i, j;

void ask()

{

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

    {

        printf("\nEnter %dst Employee record:\n", i+1);

        printf("\nEmployee name:\t");
```

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

Nested Structures

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

Example:

```
struct Student
{
    char[30] name;
    int age;
    /* here Address is a structure */
```

```
    struct Address
    {
        char[50] locality;
        char[50] city;
        int pincode;
    }addr;
};
```

Structure as Function Arguments

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

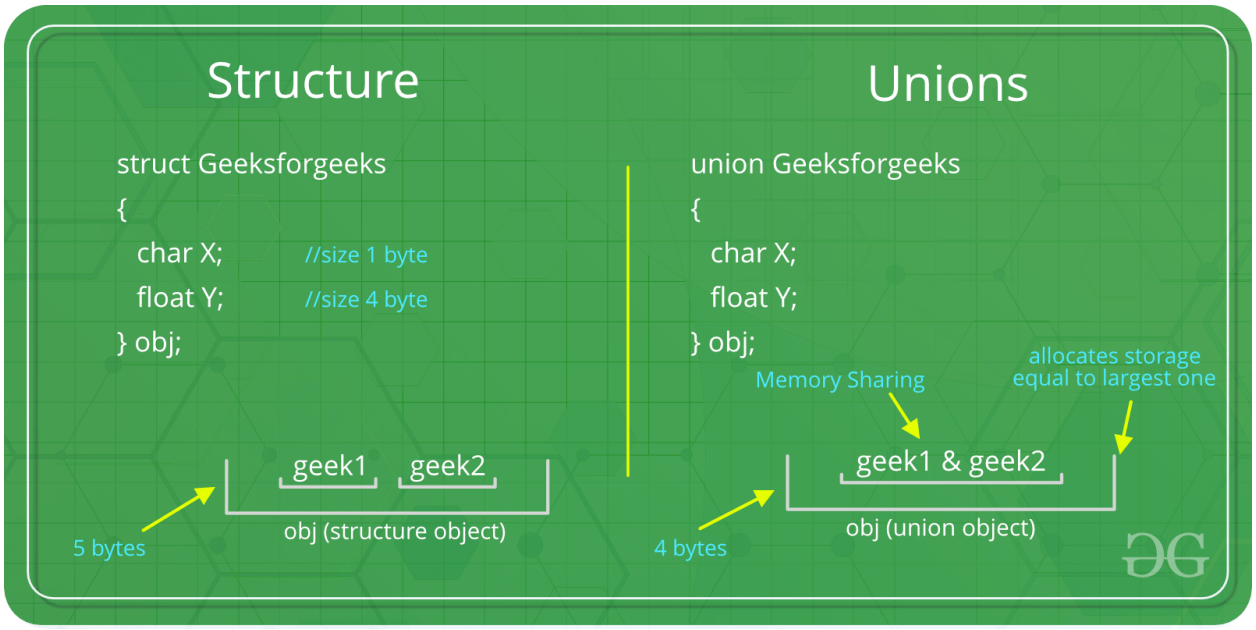
Example:

```
#include<stdio.h>


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


void show(struct Student st);


void main()
{
    struct Student std;
```

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

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

Union

Like Structures, union is a user defined data type. In union, all members share the same memory location.



For example in the following C program, both x and y share the same location. If we change x, we can see the changes being reflected in y.

```
#include <stdio.h>

  // Declaration of union is same as structures

union test {

    int x, y;

};

  int main()

{

    // A union variable t

    union test t;

      t.x = 2; // t.y also gets value 2

    printf("After making x = 2:\n x = %d, y = %d\n\n",
```

```
        t.x, t.y);

    t.y = 10; // t.x is also updated to 10

    printf("After making y = 10:\n x = %d, y = %d\n\n",

        t.x, t.y);

    return 0;

}
```

Output:

```
After making x = 2:

 x = 2, y = 2



After making y = 10:

 x = 10, y = 10
```

How is the size of union decided by compiler?
Size of a union is taken according the size of largest member in union.

```
#include <stdio.h>

  union test1 {

    int x;

    int y;

} Test1;

  union test2 {

  int x;

    char y;

} Test2;
```

```
 union test3 {
   int  arr[10];
   char y;
}     Test3;
 int main()
{
   printf("sizeof(test1) = %lu, sizeof(test2) = %lu, "
          "sizeof(test3) = %lu",
          sizeof(Test1),
          sizeof(Test2), sizeof(Test3));
   return 0;
}
```

Output:

```
sizeof(test1) = 4, sizeof(test2) = 4, sizeof(test3) = 40
```

Pointers to unions?
Like structures, we can have pointers to unions and can access members using the arrow operator (->). The following example demonstrates the same.

```
#include <stdio.h>


union test {
   int x;
   char y;
};
```

```
int main()

{

    union test p1;

    p1.x = 65;


    // p2 is a pointer to union p1

    union test* p2 = &p1;


    // Accessing union members using pointer

    printf("%d %c", p2->x, p2->y);

    return 0;

}
```

Output:

```
65 A
```

What are applications of union?

Unions can be useful in many situations where we want to use the same memory for two or more members. For example, suppose we want to implement a binary tree data structure where each leaf node has a double data value, while each internal node has pointers to two children, but no data. If we declare this as:

```
struct NODE {

    struct NODE* left;

    struct NODE* right;

    double data;

};
```

then every node requires 16 bytes, with half the bytes wasted for each type of node. On the other hand, if we declare a node as following, then we can save space.

```
struct NODE {
   bool is_leaf;
   union {
      struct
      {
         struct NODE* left;
         struct NODE* right;
      } internal;
      double data;
   } info;
};
```

A **union** is a special data type available in C that allows to store different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple-purpose.

Defining a Union

To define a union, you must use the **union** statement in the same way as you did while defining a structure. The union statement defines a new data type with more than one member for your program. The format of the union statement is as follows −

```
union [union tag] {
  member definition;
  member definition;
```

```
   ...

   member definition;

} [one or more union variables];
```

The **union tag** is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the union's definition, before the final semicolon, you can specify one or more union variables but it is optional. Here is the way you would define a union type named Data having three members i, f, and str −

```
union Data {

   int i;

   float f;

   char str[20];

} data;
```

Now, a variable of **Data** type can store an integer, a floating-point number, or a string of characters. It means a single variable, i.e., same memory location, can be used to store multiple types of data. You can use any built-in or user defined data types inside a union based on your requirement.

The memory occupied by a union will be large enough to hold the largest member of the union. For example, in the above example, Data type will occupy 20 bytes of memory space because this is the maximum space which can be occupied by a character string. The following example displays the total memory size occupied by the above union −

```
#include <stdio.h>

#include <string.h>



union Data {

   int i;

   float f;

   char str[20];

};
```

```
int main( ) {

   union Data data;

   printf( "Memory size occupied by data : %d\n", sizeof(data));

   return 0;
}
```

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

```
Memory size occupied by data : 20
```

Accessing Union Members

To access any member of a union, we use the **member access operator (.)**. The member access operator is coded as a period between the union variable name and the union member that we wish to access. You would use the keyword **union** to define variables of union type. The following example shows how to use unions in a program −

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

union Data {
   int i;
   float f;
   char str[20];
};
```

```
int main( ) {

   union Data data;

   data.i = 10;
   data.f = 220.5;
   strcpy( data.str, "C Programming");

   printf( "data.i : %d\n", data.i);
   printf( "data.f : %f\n", data.f);
   printf( "data.str : %s\n", data.str);

   return 0;
}
```

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

```
data.i : 1917853763
data.f : 4122360580327794860452759994368.000000
data.str : C Programming
```

Here, we can see that the values of **i** and **f** members of union got corrupted because the final value assigned to the variable has occupied the memory location and this is the reason that the value of **str** member is getting printed very well.

Now let's look into the same example once again where we will use one variable at a time which is the main purpose of having unions −

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

```
union Data {
   int i;
   float f;
   char str[20];
};

int main( ) {

   union Data data;

   data.i = 10;
   printf( "data.i : %d\n", data.i);

   data.f = 220.5;
   printf( "data.f : %f\n", data.f);

   strcpy( data.str, "C Programming");
   printf( "data.str : %s\n", data.str);

   return 0;
}
```

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

data.i : 10

data.f : 220.500000

data.str : C Programming

Here, all the members are getting printed very well because one member is being used at a time.

Array of structures:

Why use an array of structures?

Consider a case, where we need to store the data of 5 students. We can store it by using the structure as given below.

```
1.  #include<stdio.h>
2.  struct student
3.  {
4.      char name[20];
5.      int id;
6.      float marks;
7.  };
8.  void main()
9.  {
10.     struct student s1,s2,s3;
11.     int dummy;
12.     printf("Enter the name, id, and marks of student 1 ");
13.     scanf("%s %d %f",s1.name,&s1.id,&s1.marks);
14.     scanf("%c",&dummy);
15.     printf("Enter the name, id, and marks of student 2 ");
16.     scanf("%s %d %f",s2.name,&s2.id,&s2.marks);
17.     scanf("%c",&dummy);
18.     printf("Enter the name, id, and marks of student 3 ");
19.     scanf("%s %d %f",s3.name,&s3.id,&s3.marks);
20.     scanf("%c",&dummy);
21.     printf("Printing the details....\n");
22.     printf("%s %d %f\n",s1.name,s1.id,s1.marks);
23.     printf("%s %d %f\n",s2.name,s2.id,s2.marks);
24.     printf("%s %d %f\n",s3.name,s3.id,s3.marks);
25. }
```

Output

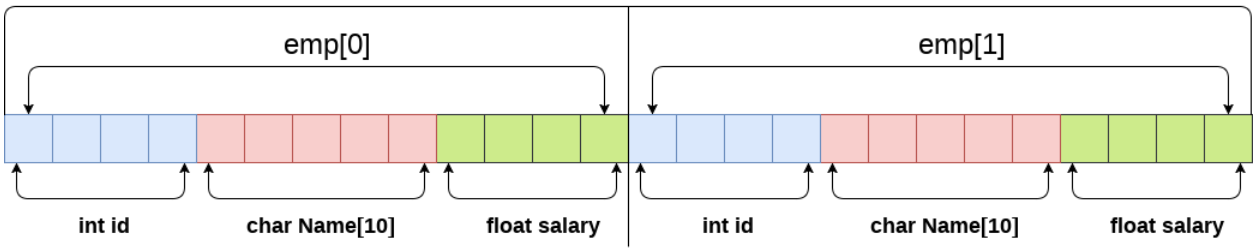
```
Enter the name, id, and marks of student 1 James 90 90

Enter the name, id, and marks of student 2 Adoms 90 90

Enter the name, id, and marks of student 3 Nick 90 90

Printing the details....

James 90 90.000000

Adoms 90 90.000000

Nick 90 90.000000
```

In the above program, we have stored data of 3 students in the structure. However, the complexity of the program will be increased if there are 20 students. In that case, we will have to declare 20 different structure variables and store them one by one. This will always be tough since we will have to declare a variable every time we add a student. Remembering the name of all the variables is also a very tricky task. However, c enables us to declare an array of structures by using which, we can avoid declaring the different structure variables; instead we can make a collection containing all the structures that store the information of different entities.

Array of Structures in C

An array of structres in C can be defined as the collection of multiple structures variables where each variable contains information about different entities. The array of structures in C are used to store information about multiple entities of different data types. The array of structures is also known as the collection of structures.

Array of structures




```
struct employee
{
    int id;
    char name[5];
    float salary;
};
struct employee emp[2];
```

sizeof (emp) = 4 + 5 + 4 = 13 bytes

sizeof (emp[2]) = 26 bytes

Let's see an example of an array of structures that stores information of 5 students and prints it.

```
1.  #include<stdio.h>
2.  #include <string.h>
3.  struct student{
4.  int rollno;
5.  char name[10];
6.  };
7.  int main(){
8.  int i;
9.  struct student st[5];
10. printf("Enter Records of 5 students");
11. for(i=0;i<5;i++){
12. printf("\nEnter Rollno:");
13. scanf("%d",&st[i].rollno);
14. printf("\nEnter Name:");
15. scanf("%s",&st[i].name);
16. }
17. printf("\nStudent Information List:");
18. for(i=0;i<5;i++){
19. printf("\nRollno:%d, Name:%s",st[i].rollno,st[i].name);
20. }
21.    return 0;
22. }
```

Output:

```
Enter Records of 5 students

Enter Rollno:1

Enter Name:Sonoo

Enter Rollno:2

Enter Name:Ratan

Enter Rollno:3

Enter Name:Vimal

Enter Rollno:4

Enter Name:James

Enter Rollno:5

Enter Name:Sarfraz


Student Information List:

Rollno:1, Name:Sonoo

Rollno:2, Name:Ratan

Rollno:3, Name:Vimal

Rollno:4, Name:James

Rollno:5, Name:Sarfraz
```

Nested Structure in C

C provides us the feature of nesting one structure within another structure by using which, complex data types are created. For example, we may need to store the address of an entity employee in a structure. The attribute address may also have the subparts as street number, city, state, and pin code. Hence, to store the address of the employee, we need to store the address of

the employee into a separate structure and nest the structure address into the structure employee. Consider the following program.

```
1. #include<stdio.h>
2. struct address
3. {
4.     char city[20];
5.     int pin;
6.     char phone[14];
7. };
8. struct employee
9. {
10.     char name[20];
11.     struct address add;
12. };
13. void main ()
14. {
15.     struct employee emp;
16.     printf("Enter employee information?\n");
17.     scanf("%s %s %d %s",emp.name,emp.add.city, &emp.add.pin, emp.add.phone);
18.     printf("Printing the employee information....\n");
19.     printf("name: %s\nCity: %s\nPincode: %d\nPhone: %s",emp.name,emp.add.city,emp.add.pin,
    emp.add.phone);
20. }
```

Output

```
Enter employee information?


Arun


Delhi


110001


1234567890
```

```
Printing the employee information....


name: Arun


City: Delhi


Pincode: 110001


Phone: 1234567890
```

The structure can be nested in the following ways.

1. By separate structure
2. By Embedded structure

1) Separate structure

Here, we create two structures, but the dependent structure should be used inside the main structure as a member. Consider the following example.

```
1.  struct Date
2.  {
3.     int dd;
4.     int mm;
5.     int yyyy;
6.  };
7.  struct Employee
8.  {
9.     int id;
10.    char name[20];
11.    struct Date doj;
12. }emp1;
```

As you can see, doj (date of joining) is the variable of type Date. Here doj is used as a member in Employee structure. In this way, we can use Date structure in many structures.

2) Embedded structure

The embedded structure enables us to declare the structure inside the structure. Hence, it requires less line of codes but it can not be used in multiple data structures. Consider the following example.

```
1. struct Employee
2. {
3.    int id;
4.    char name[20];
5.    struct Date
6.     {
7.       int dd;
8.       int mm;
9.       int yyyy;
10.    }doj;
11. }emp1;
```

Accessing Nested Structure

We can access the member of the nested structure by Outer_Structure.Nested_Structure.member as given below:

```
1. e1.doj.dd
2. e1.doj.mm
3. e1.doj.yyyy
```

C Nested Structure example

Let's see a simple example of the nested structure in C language.

```
1. #include <stdio.h>
2. #include <string.h>
3. struct Employee
4. {
5.    int id;
6.    char name[20];
7.    struct Date
8.     {
9.       int dd;
10.      int mm;
11.      int yyyy;
12.    }doj;
13. }e1;
```

```
14. int main( )
15. {
16.   //storing employee information
17.   e1.id=101;
18.   strcpy(e1.name, "Sonoo Jaiswal");//copying string into char array
19.   e1.doj.dd=10;
20.   e1.doj.mm=11;
21.   e1.doj.yyyy=2014;
22.
23.   //printing first employee information
24.   printf( "employee id : %d\n", e1.id);
25.   printf( "employee name : %s\n", e1.name);
26.   printf( "employee date of joining (dd/mm/yyyy) : %d/%d/%d\n", e1.doj.dd,e1.doj.mm,e1.doj.y
    yyy);
27.   return 0;
28. }
```

Output:

```
employee id : 101

employee name : Sonoo Jaiswal

employee date of joining (dd/mm/yyyy) : 10/11/2014
```

Passing structure to function

Just like other variables, a structure can also be passed to a function. We may pass the structure members into the function or pass the structure variable at once. Consider the following example to pass the structure variable employee to a function display() which is used to display the details of an employee.

```
1.  #include<stdio.h>
2.  struct address
3.  {
4.     char city[20];
5.     int pin;
6.     char phone[14];
7.  };
8.  struct employee
9.  {
10.    char name[20];
11.    struct address add;
12. };
```

```
13. void display(struct employee);
14. void main ()
15. {
16.     struct employee emp;
17.     printf("Enter employee information?\n");
18.     scanf("%s %s %d %s",emp.name,emp.add.city, &emp.add.pin, emp.add.phone);
19.     display(emp);
20. }
21. void display(struct employee emp)
22. {
23.   printf("Printing the details....\n");
24.   printf("%s %s %d %s",emp.name,emp.add.city,emp.add.pin,emp.add.phone);
25. }
```

Preprocessor: Commonly used Preprocessor commands like include, define, undef, if, ifdef, ifndef.

Preprocessor:

The **C Preprocessor** is not a part of the compiler, but is a separate step in the compilation process. In simple terms, a C Preprocessor is just a text substitution tool and it instructs the compiler to do required pre-processing before the actual compilation. We'll refer to the C Preprocessor as CPP.

All preprocessor commands begin with a hash symbol (#). It must be the first nonblank character, and for readability, a preprocessor directive should begin in the first column. The following section lists down all the important preprocessor directives −

Sr.No.	Directive & Description
1	**#define :** Substitutes a preprocessor macro.
2	**#include :** Inserts a particular header from another file.
3	**#undef :** Undefines a preprocessor macro.
4	**#ifdef :** Returns true if this macro is defined.
5	**#ifndef :** Returns true if this macro is not defined.

6	**#if :** Tests if a compile time condition is true.
7	**#else :** The alternative for #if.
8	**#elif :**#else and #if in one statement.
9	**#endif :** Ends preprocessor conditional.
10	**#error:** Prints error message on stderr.
11	**#pragma :** Issues special commands to the compiler, using a standardized method.

Preprocessors Examples

Analyze the following examples to understand various directives.

#define MAX_ARRAY_LENGTH 20

This directive tells the CPP to replace instances of MAX_ARRAY_LENGTH with 20. Use *#define* for constants to increase readability.

#include <stdio.h>

#include "myheader.h"

These directives tell the CPP to get stdio.h from **System Libraries** and add the text to the current source file. The next line tells CPP to get **myheader.h** from the local directory and add the content to the current source file.

#undef FILE_SIZE

#define FILE_SIZE 42

It tells the CPP to undefine existing FILE_SIZE and define it as 42.

#ifndef MESSAGE

  #define MESSAGE "You wish!"

#endif

It tells the CPP to define MESSAGE only if MESSAGE isn't already defined.

#ifdef DEBUG

```
   /* Your debugging statements here */

#endif
```

It tells the CPP to process the statements enclosed if DEBUG is defined. This is useful if you pass the *-DDEBUG* flag to the gcc compiler at the time of compilation. This will define DEBUG, so you can turn debugging on and off on the fly during compilation.

Predefined Macros

ANSI C defines a number of macros. Although each one is available for use in programming, the predefined macros should not be directly modified.

Sr.No.	Macro & Description
1	**__DATE__ :** The current date as a character literal in "MMM DD YYYY" format.
2	**__TIME__:** The current time as a character literal in "HH:MM:SS" format.
3	**__FILE__:** This contains the current filename as a string literal.
4	**__LINE__:** This contains the current line number as a decimal constant.
5	**__STDC__:** Defined as 1 when the compiler complies with the ANSI standard.

Let's try the following example −

```
#include <stdio.h>


int main() {


   printf("File :%s\n", __FILE__);

   printf("Date :%s\n", __DATE__);

   printf("Time :%s\n", __TIME__);

   printf("Line :%d\n", __LINE__);

   printf("ANSI :%d\n", __STDC__);
```

}

When the above code in a file **test.c** is compiled and executed, it produces the following result −

File :test.c

Date :Jun 2 2012

Time :03:36:24

Line :8

ANSI :1

Preprocessor Operators

The C preprocessor offers the following operators to help create macros −

The Macro Continuation (\) Operator

A macro is normally confined to a single line. The macro continuation operator (\) is used to continue a macro that is too long for a single line. For example −

```
#define  message_for(a, b)  \
   printf(#a " and " #b ": We love you!\n")
```

The Stringsize (#) Operator

The stringize or number-sign operator ('#'), when used within a macro definition, converts a macro parameter into a string constant. This operator may be used only in a macro having a specified argument or parameter list. For example −

```
#include <stdio.h>

#define  message_for(a, b)  \
   printf(#a " and " #b ": We love you!\n")

int main(void) {
```

```
   message_for(Carole, Debra);

   return 0;

}
```

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

```
Carole and Debra: We love you!
```

The Token Pasting (##) Operator

The token-pasting operator (##) within a macro definition combines two arguments. It permits two separate tokens in the macro definition to be joined into a single token. For example −

```
#include <stdio.h>


#define tokenpaster(n) printf ("token" #n " = %d", token##n)


int main(void) {

   int token34 = 40;

   tokenpaster(34);

   return 0;

}
```

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

```
token34 = 40
```

It happened so because this example results in the following actual output from the preprocessor −

```
printf ("token34 = %d", token34);
```

This example shows the concatenation of token##n into token34 and here we have used both **stringize** and **token-pasting**.

The Defined() Operator

The preprocessor **defined** operator is used in constant expressions to determine if an identifier is defined using #define. If the specified identifier is defined, the value is true (non-zero). If the symbol is not defined, the value is false (zero). The defined operator is specified as follows −

```
#include <stdio.h>


#if !defined (MESSAGE)
   #define MESSAGE "You wish!"
#endif


int main(void) {
   printf("Here is the message: %s\n", MESSAGE);
   return 0;
}
```

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

```
Here is the message: You wish!
```

Parameterized Macros

One of the powerful functions of the CPP is the ability to simulate functions using parameterized macros. For example, we might have some code to square a number as follows −

```
int square(int x) {
   return x * x;
}
```

We can rewrite above the code using a macro as follows −

```
#define square(x) ((x) * (x))
```

Macros with arguments must be defined using the **#define** directive before they can be used. The argument list is enclosed in parentheses and must immediately follow the macro name. Spaces are not allowed between the macro name and open parenthesis. For example −

```
#include <stdio.h>

#define MAX(x,y) ((x) > (y) ? (x) : (y))

int main(void) {
   printf("Max between 20 and 10 is %d\n", MAX(10, 20));
   return 0;
}
```

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

```
Max between 20 and 10 is 20
```

https://fresh2refresh.com/c-programming/c-preprocessor-directives/

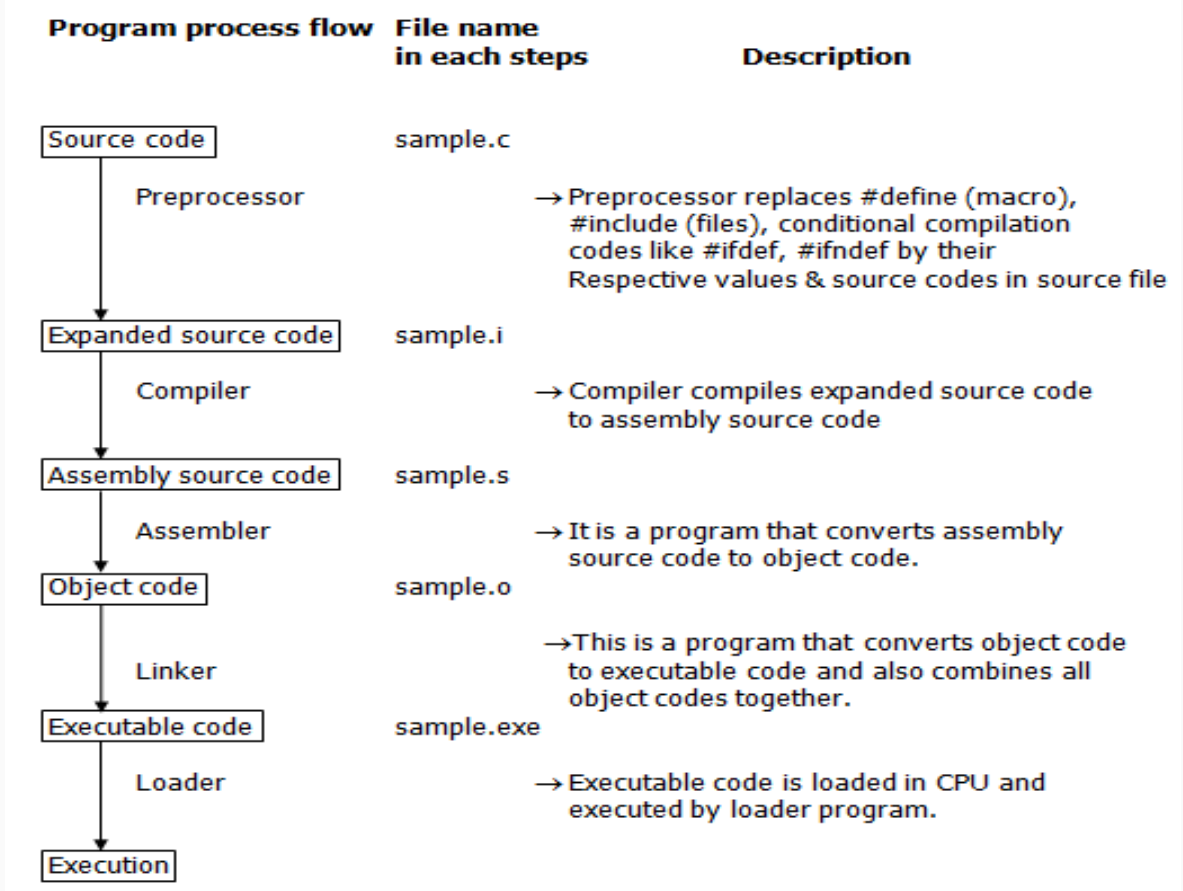
C PREPROCESSOR DIRECTIVES:

- Before a C program is compiled in a compiler, source code is processed by a program called preprocessor. This process is called preprocessing.
- Commands used in preprocessor are called preprocessor directives and they begin with "#" symbol.

Below is the list of preprocessor directives that C programming language offers.

	Syntax/Description
Preprocessor	
Macro	**Syntax:** #define This macro defines constant value and can be any of the basic data types.
Header file inclusion	**Syntax:** #include <file_name> The source code of the file “file_name” is included in the main program at the specified place.
Conditional compilation	**Syntax:** #ifdef, #endif, #if, #else, #ifndef Set of commands are included or excluded in source program before compilation with respect to the condition.
Other directives	**Syntax:** #undef, #pragma #undef is used to undefine a defined macro variable. #Pragma is used to call a function before and after main function in a C program.

A program in C language involves into different processes. Below diagram will help you to understand all the processes that a C program comes across.



There are 4 regions of memory which are created by a compiled C program. They are,

1. **First region** – This is the memory region which holds the executable code of the program.
2. **2nd region** – In this memory region, global variables are stored.
3. **3rd region** – stack
4. **4th region** – heap

DO YOU KNOW DIFFERENCE BETWEEN STACK & HEAP MEMORY IN C LANGUAGE?

Stack	**Heap**
Stack is a memory region where "local variables", "return addresses of function calls" and "arguments to functions" are hold while C program is executed.	Heap is a memory region which is used by dynamic memory allocation functions at run time.

CPU’s current state is saved in stack memory	Linked list is an example which uses heap memory.

DO YOU KNOW DIFFERENCE BETWEEN COMPILERS VS INTERPRETERS IN C LANGUAGE?

KEY POINTS TO REMEMBER:

1. Source program is converted into executable code through different processes like precompilation, compilation, assembling and linking.
2. Local variables uses stack memory.
3. Dynamic memory allocation functions use the heap memory.

EXAMPLE PROGRAM FOR #DEFINE, #INCLUDE PREPROCESSORS IN C LANGUAGE:

- #define – This macro defines constant value and can be any of the basic data types.
- #include <file_name> – The source code of the file “file_name” is included in the main C program where “#include <file_name>” is mentioned.

```
1  #include <stdio.h>
2
3  #define height 100
4  #define number 3.14
5  #define letter 'A'
6  #define letter_sequence "ABC"
7  #define backslash_char '\?'
8
9  void main()
10 {
11    printf("value of height    : %d \n", height );
12    printf("value of number : %f \n", number );
```

```
13    printf("value of letter : %c \n", letter );
14    printf("value of letter_sequence : %s \n", letter_sequence);
15    printf("value of backslash_char  : %c \n", backslash_char);
16
17 }
```

COMPILE & RUN

OUTPUT:

```
value of height : 100
value of number : 3.140000
value of letter : A
value of letter_sequence : ABC
value of backslash_char : ?
```

EXAMPLE PROGRAM FOR CONDITIONAL COMPILATION DIRECTIVES:

A) EXAMPLE PROGRAM FOR #IFDEF, #ELSE AND #ENDIF IN C:

- "#ifdef" directive checks whether particular macro is defined or not. If it is defined, "If" clause statements are included in source file.
- Otherwise, "else" clause statements are included in source file for compilation and execution.

```
1  #include <stdio.h>
2  #define RAJU 100
3
4  int main()
5  {
6     #ifdef RAJU
```

```
7    printf("RAJU is defined. So, this line will be added in " \
8          "this C file\n");
9    #else
10   printf("RAJU is not defined\n");
11   #endif
12   return 0;
13 }
```

COMPILE & RUN

OUTPUT:

```
RAJU is defined. So, this line will be added in this C file
```

B) EXAMPLE PROGRAM FOR #IFNDEF AND #ENDIF IN C:

- #ifndef exactly acts as reverse as #ifdef directive. If particular macro is not defined, “If” clause statements are included in source file.
- Otherwise, else clause statements are included in source file for compilation and execution.

```
1  #include <stdio.h>
2  #define RAJU 100
3  int main()
4  {
5    #ifndef SELVA
6    {
7      printf("SELVA is not defined. So, now we are going to " \
8            "define here\n");
9      #define SELVA 300
```

```
10    }
11    #else
12    printf("SELVA is already defined in the program”);
13
14    #endif
15    return 0;
16
17 }
```

COMPILE & RUN

OUTPUT:

```
SELVA is not defined. So, now we are going to define here
```

C) EXAMPLE PROGRAM FOR #IF, #ELSE AND #ENDIF IN C:

- “If” clause statement is included in source file if given condition is true.
- Otherwise, else clause statement is included in source file for compilation and execution.

```
1  #include <stdio.h>
2  #define a 100
3  int main()
4  {
5     #if (a==100)
6     printf("This line will be added in this C file since " \
7           "a \= 100\n");
8     #else
9     printf("This line will be added in this C file since " \
```

```
10          "a is not equal to 100\n");
11    #endif
12    return 0;
13 }
```

COMPILE & RUN

OUTPUT:

```
This line will be added in this C file since a = 100
```

EXAMPLE PROGRAM FOR UNDEF IN C LANGUAGE:

This directive undefines existing macro in the program.

```
 1  #include <stdio.h>
 2
 3  #define height 100
 4  void main()
 5  {
 6     printf("First defined value for height    : %d\n",height);
 7     #undef height          // undefining variable
 8     #define height 600     // redefining the same for new value
 9     printf("value of height after undef \& redefine:%d",height);
10 }
```

COMPILE & RUN

OUTPUT:

```
First defined value for height : 100
```

```
value of height after undef & redefine : 600
```

EXAMPLE PROGRAM FOR PRAGMA IN C LANGUAGE:

Pragma is used to call a function before and after main function in a C program.

```
 1  #include <stdio.h>
 2
 3  void function1( );
 4  void function2( );
 5
 6  #pragma startup function1
 7  #pragma exit function2
 8
 9  int main( )
10 {
11    printf ( "\n Now we are in main function" ) ;
12    return 0;
13 }
14
15 void function1( )
16 {
17    printf("\nFunction1 is called before main function call");
18 }
19
```

```
20 void function2( )
21 {
22    printf ( "\nFunction2 is called just before end of " \
23            "main function" ) ;"
24 }
```

COMPILE & RUN

OUTPUT:

```
Function1 is called before main function call

Now we are in main function

Function2 is called just before end of main function
```

MORE ON PRAGMA DIRECTIVE IN C LANGUAGE:

Pragma command	**Description**
#Pragma startup <function_name_1>	This directive executes function named “function_name_1” before
#Pragma exit <function_name_2>	This directive executes function named “function_name_2” just before termination of the program.
#pragma warn – rvl	If function doesn’t return a value, then warnings are suppressed by this directive while compiling.

#pragma warn – par	If function doesn’t use passed function parameter , then warnings are suppressed
#pragma warn – rch	If a non reachable code is written inside a program, such warnings are suppressed by this directive.

MODULE – III: POINTERS AND FILE HANDLING IN C:

Pointers: Idea of pointers, Defining pointers, Pointers to Arrays and Structures, Use of Pointers in self-referential structures, usage of self referential structures in linked list (no implementation) Enumeration data type

Files: Text and Binary files, Creating and Reading and writing text and binary files, Appending data to existing files, Writing and reading structures using binary files, Random access using fseek, ftell and rewind functions.

Pointers: Idea of pointers

- Pointers in C language is a variable that stores/points the address of another variable. A Pointer in C is used to allocate memory dynamically i.e. at run time. The pointer variable might be belonging to any of the data type such as int, float, char, double, short etc.
- Pointer Syntax : data_type *var_name; Example : int *p; char *p;
- Where, * is used to denote that "p" is pointer variable and not a normal variable.

KEY POINTS TO REMEMBER ABOUT POINTERS IN C:

- Normal variable stores the value whereas pointer variable stores the address of the variable.
- The content of the C pointer always be a whole number i.e. address.
- Always C pointer is initialized to null, i.e. int *p = null.
- The value of null pointer is 0.
- & symbol is used to get the address of the variable.
- * symbol is used to get the value of the variable that the pointer is pointing to.
- If a pointer in C is assigned to NULL, it means it is pointing to nothing.
- Two pointers can be subtracted to know how many elements are available between these two pointers.
- But, Pointer addition, multiplication, division are not allowed.
- The size of any pointer is 2 byte (for 16 bit compiler).

```
#include <stdio.h>
int main()
{
   int *ptr, q;
   q = 50;
   /* address of q is assigned to ptr */
   ptr = &q;
   /* display q's value using ptr variable */
   printf("%d", *ptr);
   return 0;
}
```

Output: 50

Defining pointers

Same as above

Pointers to Arrays and Structures

Create an array of structure variable

In the following example we are considering the student structure that we created in the previous tutorial and we are creating an array of student structure variable std of size 3 to hold details of three students.

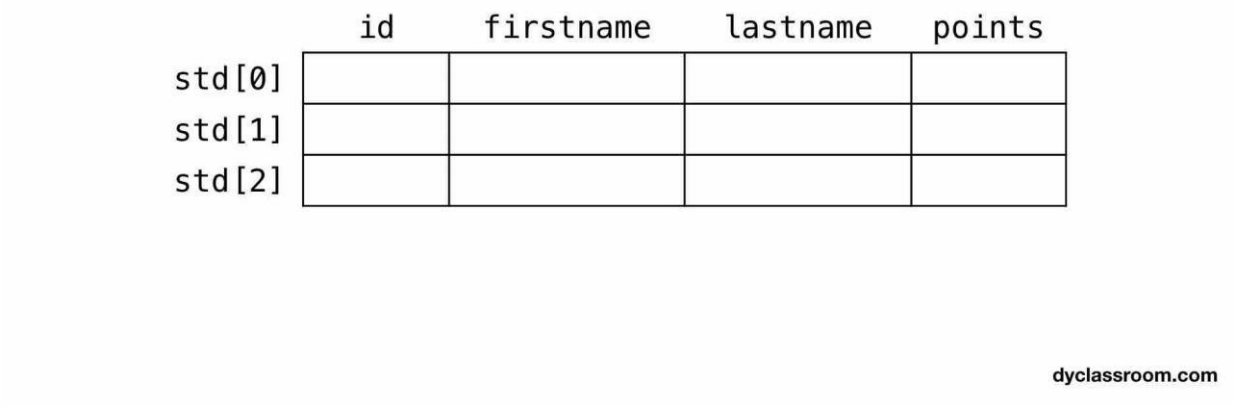
```
// student structure

struct student {

char id[15];

  char  firstname[64];

  char  lastname[64];

  float points;

};



// student structure variable

struct student std[3];
```

We can represent the std array variable as following.

```
struct student {
  char id[15];
  char firstname[64];
  char lastname[64];
  float points;
};
```

```
struct student std[3];
```

CLASSROOM



dyclassroom.com

Create pointer variable for structure

Now we will create a pointer variable that will hold the starting address of the student structure variable std.

```
// student structure pointer variable

struct student *ptr = NULL;



// assign std to ptr

ptr = std;
```

Note! std is an array variable and the name of the array variable points at the memory location so, we are assigning it to the structure pointer variable ptr.

Accessing each element of the structure array variable via pointer

For this we will first set the pointer variable ptr to point at the starting memory location of std variable. For this we write ptr = std;.

Then, we can increment the pointer variable using increment operator ptr++ to make the pointer point at the next element of the structure array variable i.e., from str[0] to str[1].

We will loop three times as there are three students. So, we will increment pointer variable twice. First increment will move pointer ptr from std[0] to std[1] and the second increment will move pointer ptr from std[1] to std[2].

To reset the pointer variable ptr to point at the starting memory location of structure variable std we write ptr = std;.

Complete code

```
#include <stdio.h>


int main(void) {


  // student structure

  struct student {

  char id[15];

    char  firstname[64];

    char  lastname[64];

    float points;

  };


  // student structure variable

  struct student std[3];


  // student structure pointer variable

  struct student *ptr = NULL;
```

```
  // other variables
  int i;

  // assign std to ptr
ptr = std;

  // get detail for user
  for (i = 0; i< 3; i++) {
printf("Enter detail of student #%d\n", (i + 1));
printf("Enter ID: ");
scanf("%s", ptr->id);
printf("Enter first name: ");
scanf("%s", ptr->firstname);
printf("Enter last name: ");
scanf("%s", ptr->lastname);
printf("Enter Points: ");
scanf("%f", &ptr->points);

    // update pointer to point at next element
    // of the array std
ptr++;
```

```
  }


  // reset pointer back to the starting
  // address of std array
ptr = std;


  for (i = 0; i< 3; i++) {
printf("\nDetail of student #%d\n", (i + 1));


    // display result via std variable
printf("\nResult via std\n");
printf("ID: %s\n", std[i].id);
printf("First Name: %s\n", std[i].firstname);
printf("Last Name: %s\n", std[i].lastname);
printf("Points: %f\n", std[i].points);


    // display result via ptr variable
printf("\nResult via ptr\n");
printf("ID: %s\n", ptr->id);
printf("First Name: %s\n", ptr->firstname);
printf("Last Name: %s\n", ptr->lastname);
printf("Points: %f\n", ptr->points);
```

```
    // update pointer to point at next element
    // of the array std
ptr++;
  }


  return 0;
}
```

Output:

```
Enter detail of student #1
Enter ID: s01
Enter first name: Yusuf
Enter last name: Shakeel
Enter Points: 8

Enter detail of student #2
Enter ID: s02
Enter first name: Jane
Enter last name: Doe
Enter Points: 9

Enter detail of student #3
```

```
Enter ID: s03

Enter first name: John

Enter last name: Doe

Enter Points: 6


Detail of student #1


Result via std

ID: s01

First Name: Yusuf

Last Name: Shakeel

Points: 8.000000


Result via ptr

ID: s01

First Name: Yusuf

Last Name: Shakeel

Points: 8.000000


Detail of student #2


Result via std
```

ID: s02

First Name: Jane

Last Name: Doe

Points: 9.000000

Result via ptr

ID: s02

First Name: Jane

Last Name: Doe

Points: 9.000000

Detail of student #3

Result via std

ID: s03

First Name: John

Last Name: Doe

Points: 6.000000

Result via ptr

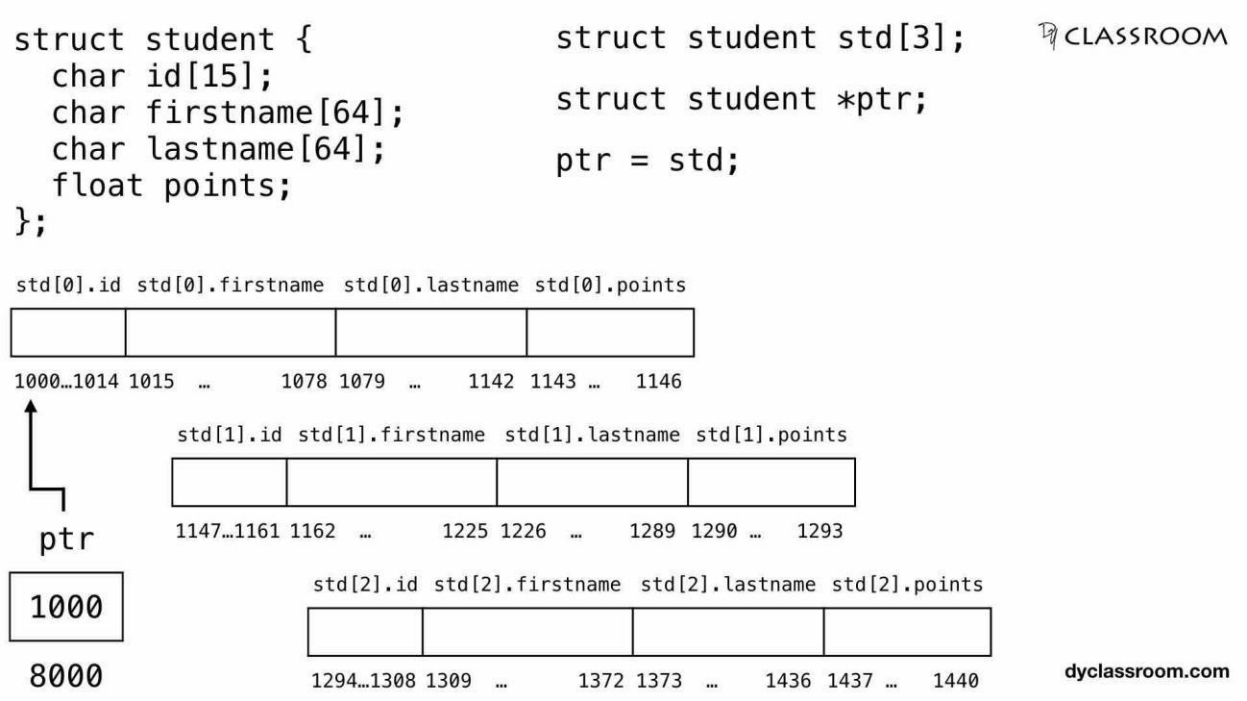
ID: s03

First Name: John

Last Name: Doe

Points: 6.000000

We can represent the std array variable in memory as follows.



Points to note!

Each student data takes 147 bytes of memory.

Member	Data Type	Size
id	char	15 bytes
firstname	char	64 bytes
lastname	char	64 bytes

points	float	4 bytes

And the array size is 3 so, total 147x3 i.e., 441 bytes is allocated to the std array variable.

The first element std[0] gets the memory location from 1000 to 1146.

The second element std[1] gets the memory location from 1147 to 1293.

And the third element std[2] gets the memory location from 1294 to 1440.

We start by first making the ptr pointer variable point at address 1000 which is the starting address of the first element std[0].

Then moving forward we increment the pointer ptr++ so, it points at the memory location 1147 i.e., the starting memory location of second element std[1].

Similarly, in the next run we point ptr at memory location 1294 i.e., starting location of third element std[2].

To access the members of the structure via pointer we use the -> arrow operator.

http://www.c4learn.com/c-programming/c-pointer-array-structure/

Pointer to Array of Structure

1. Pointer Variable can also Store the Address of the Structure Variable.
2. Pointer to **Array of Structure** stores the **Base address** of the Structure array.
3. Suppose

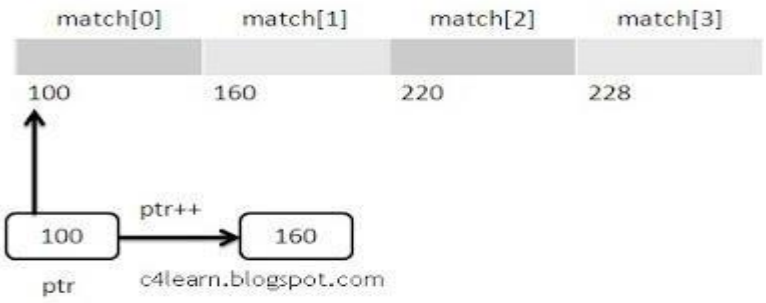
```
struct Cricket
{
char team1[20];
char team2[20];
char ground[18];
int result;
}match[4] = {
        {"IND","AUS","PUNE",1},
        {"IND","PAK","NAGPUR",1},
        {"IND","NZ","MUMBAI",0},
        {"IND","SA","DELHI",1}
```

```
};
```

4. Pointer Is Declared and initialized as —

```
struct Cricket *ptr = match
```

5. Here the address of match[0] is stored in pointer Variable ptr.



Live Example Program :

```
void main()
{
struct Cricket *ptr = match;   // By default match[0]
for(i=0;i<4;i++)
   {
printf("\nMatch : %d",i+1);
printf("\n%s Vs %s",ptr->team1,ptr->team2);
printf("\nPlace : %s",ptr->ground);

if(match[i].result == 1)
printf("nWinner : %s",ptr->team1);
else
printf("nWinner : %s",ptr->team1);
printf("\n");
```

```
    // Move Pointer to next structure element
ptr++;
    }
}
```

Output :

```
Match : 1
IND Vs AUS
Place : PUNE
Winner : IND
Match : 2
IND Vs PAK
Place : NAGPUR
Winner : IND

Match : 3
IND Vs NZ
Place : MUMBAI
Winner : NZ

Match : 4
IND Vs SA
Place : DELHI
Winner : IND
```

You can Write above Program :

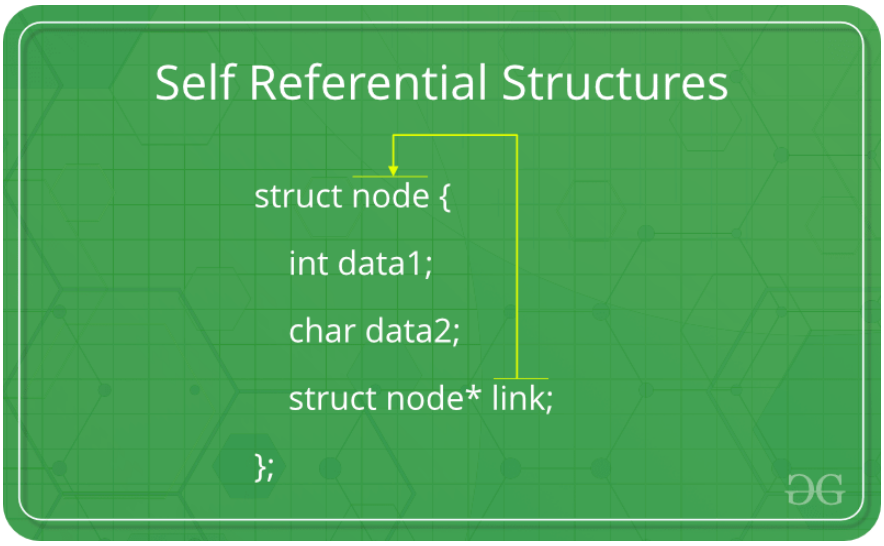
```
void main()
{
struct Cricket *ptr = match;   // By default match[0]
for(i=0;i<4;i++)
```

```
    {
printf("\nMatch : %d",i+1);
printf("\n%s Vs %s",ptr->team1,ptr->team2);
printf("\nPlace : %s",ptr->ground);

printf("\nWinner : %s",ptr->team1);
printf("\n");

    // Move Pointer to next structure element
ptr++;
    }
}
```

Use of Pointers in self-referential structures

Self Referential Structures

Self Referential structures are those structures that have one or more pointers which point to the same type of structure, as their member.



In other words, structures pointing to the same type of structures are self-referential in nature.

Example:

filter_none
brightness_4

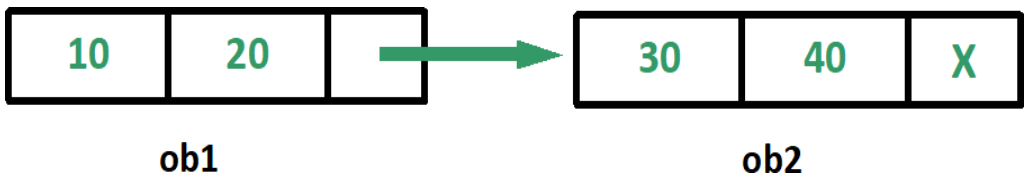
```
struct node {
    int data1;
    char data2;
    struct node* link;
};

int main()
{
    struct node ob;
    return 0;
}
```

In the above example ‘link’ is a pointer to a structure of type ‘node’. Hence, the structure ‘node’ is a self-referential structure with ‘link’ as the referencing pointer.
An important point to consider is that the pointer should be initialized properly before accessing, as by default it contains garbage value.

Types of Self Referential Structures

1. Self Referential Structure with Single Link
2. Self Referential Structure with Multiple Links

Self Referential Structure with Single Link: These structures can have only one self-pointer as their member. The following example will show us how to connect the objects of a self-referential structure with the single link and access the corresponding data members. The connection formed is shown in the following figure.



filter_none
edit
play_arrow
brightness_4

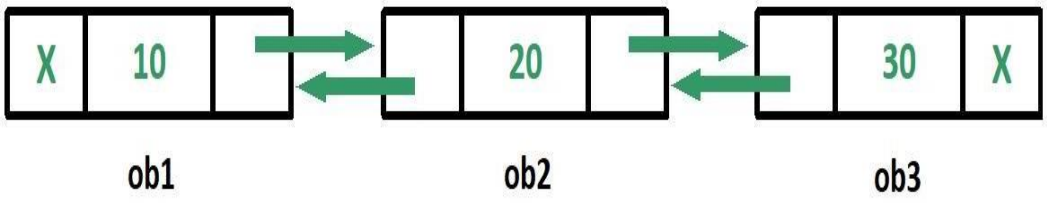
```
#include <stdio.h>

struct node {
    int data1;
    char data2;
    struct node* link;
};
```

```
int main()
{
   struct node ob1; // Node1

   // Intialization
   ob1.link = NULL;
   ob1.data1 = 10;
   ob1.data2 = 20;

   struct node ob2; // Node2

   // Initialization
   ob2.link = NULL;
   ob2.data1 = 30;
   ob2.data2 = 40;

   // Linking ob1 and ob2
   ob1.link = &ob2;

   // Accessing data members of  ob2 using ob1
   printf("%d", ob1.link->data1);
   printf("\n%d", ob1.link->data2);
   return 0;
}
```

Output:

```
30

40
```

Self Referential Structure with Multiple Links: Self referential structures with multiple links can have more than one self-pointers. Many complicated data structures can be easily constructed using these structures. Such structures can easily connect to more than one nodes at a time. The following example shows one such structure with more than one links.
The connections made in the above example can be understood using the following figure.



filter_none
edit
play_arrow
brightness_4

```
#include <stdio.h>

struct node {
    int data;
    struct node* prev_link;
    struct node* next_link;
};

int main()
{
    struct node ob1; // Node1

    // Intialization
    ob1.prev_link = NULL;
    ob1.next_link = NULL;
    ob1.data = 10;

    struct node ob2; // Node2

    // Intialization
    ob2.prev_link = NULL;
    ob2.next_link = NULL;
    ob2.data = 20;

    struct node ob3; // Node3

    // Intialization
    ob3.prev_link = NULL;
    ob3.next_link = NULL;
    ob3.data = 30;

    // Forward links
    ob1.next_link = &ob2;
    ob2.next_link = &ob3;

    // Backward links
    ob2.prev_link = &ob1;
    ob3.prev_link = &ob2;

    // Accessing  data of ob1, ob2 and ob3 by ob1
    printf("%d\t", ob1.data);
    printf("%d\t",  ob1.next_link->data);
```

```
    printf("%d\n",  ob1.next_link->next_link->data);

    // Accessing data of ob1, ob2 and ob3 by ob2
    printf("%d\t", ob2.prev_link->data);
    printf("%d\t", ob2.data);
    printf("%d\n",  ob2.next_link->data);

    // Accessing data of ob1, ob2 and ob3 by ob3
    printf("%d\t", ob3.prev_link->prev_link->data);
    printf("%d\t", ob3.prev_link->data);
    printf("%d", ob3.data);
    return 0;
}
```

Output:

```
10    20    30

10    20    30

10    20    30
```

In the above example we can see that ‘ob1’, ‘ob2’ and ‘ob3’ are three objects of the self referential structure ‘node’. And they are connected using their links in such a way that any of them can easily access each other’s data. This is the beauty of the self referential structures. The connections can be manipulated according to the requirements of the programmer.

Applications:
Self referential structures are very useful in creation of other complex data structures like:

- Linked Lists
- Stacks
- Queues
- Trees
- Graphs etc

https://www.geeksforgeeks.org/self-referential-structures/

usage of self referential structures in linked list (no implementation)

http://www.how2lab.com/programming/c/link-list1.php

Self Referential Data Structure in C - create a singly linked list

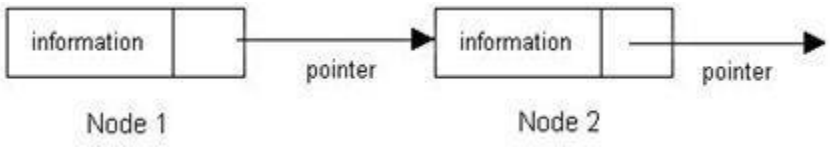
A self referential data structure is essentially a structure definition which includes at least one member that is a pointer to the structure of its own kind. A chain of such structures can thus be expressed as follows.

```
struct name {
                member 1;
                member 2;
                . . .
```

```
                struct name *pointer;
};
```

The above illustrated structure prototype describes one node that comprises of two logical segments. One of them stores data/information and the other one is a pointer indicating where the next component can be found. .Several such inter-connected nodes create a chain of structures.

The following figure depicts the composition of such a node. The figure is a simplified illustration of nodes that collectively form a chain of structures or linked list.



Such self-referential structures are very useful in applications that involve linked data structures, such as lists and trees. Unlike a *static data structure* such as array where the number of elements that can be inserted in the array is limited by the size of the array, a self-referential structure can dynamically be expanded or contracted. Operations like insertion or deletion of nodes in a self- referential structure involve simple and straight forward alteration of pointers.

Linear (Singly) Linked List

A linear linked list is a chain of structures where each node points to the next node to create a list. To keep track of the starting node's address a dedicated pointer (referred as *start pointer*) is used. The end of the list is indicated by a *NULL* pointer. In order to create a linked list of integers, we define each of its element (referred as *node*) using the following declaration.

```
struct node_type {
        int data;
        struct node_type *next;
};
struct node_type *start = NULL;
```

Note: The second member points to a node of same type.

A linear linked list illustration:

Example

Let us now develop a C program to manipulate linked lists. For this purpose we introduce a few basic functions, which can be used to create a list, displaying its contents, inserting into a list and deleting an existing element. We also introduce two functions *reverse* and *recReverse* for reversing the elements of the list.

When a list is created a pointer called *start* is used to indicate the beginning of the list. A function *createNode*, creates a node and returns a pointer to it. The function *insert* is used to insert a new node in an existing list provided the data is not already present in the list. If it is not present, we place the data in a manner so that the new element is appended at the end of the list.

```
#include <stdio.h>

struct node_type {
        int data;
        struct node_type *next;
};
typedef struct node_type list;

void showList();    //displays list contents
list *reverse();    //reverses the list
list *insert();
list *createNode();
list *delete();
list *find();

main()
{
        list *temp, *start = NULL; //start will point to first node of the list
        char c = 'y';
        int n;

        while(c != 'n' && c != 'N')
        {
                printf("\nEnter the data: ");
                scanf("%d",&n); getchar(); fflush(stdin);
                temp = createNode();
                temp->data = n;
                temp->next = NULL;
```

```
                if(!find(start,temp->data))
                        start = insert(start,temp);

                printf("\nDo you want to add new data in the list? (y/n): ");
                scanf("%c",&c); fflush(stdin);
        }
        printf("\nTHE LIST IS: "); showList(start); printf("\n\n");

        c = 'y';
        while(c != 'n' && c != 'N')
        {
                printf("\nEnter the data to be deleted: ");
                scanf("%d",&n); getchar(); fflush(stdin);
                if(find(start, n)) start = delete(start, n);

                printf("\nDo you want to delete another data from the list? (y/n):");
                scanf("%c", &c) ; fflush(stdin);
        }
        printf("\nTHE LIST AFTER DATA DELETION IS: "); showList(start); printf("\n\n");

        start = reverse(start);
        printf("\nTHE REVERSED LIST IS: "); showList(start); printf("\n\n");
}


/* Function to create a Node. Allocates memory for a new node. */
list *createNode()
{
        list *new;
        new = (list *)malloc(sizeof(list));
        return(new);
}


/* Recursive function to create and insert a new node at the end of the list */
list *insert(list *st, list *ndt)
{
        if(!st) return(ndt);
        st->next = insert(st->next, ndt);
        return(st);
}


/*
 Function to search a data item in the list and return the node address
 that matches data. In case no match found, returns NULL
```

```
*/
list *find(list *st, int dt)
{
	while(st)
		if(st->data == dt)
			return (st);
		else
			st = st->next;
	return(st);
}

void showList(list *temp)
{
	while(temp)
	{
		printf("%d ", temp->data);
		temp = temp->next;
	}
	printf("\n");
}


/* Function to reverse the list */
list *reverse(list *l)
{
	list *nxt, *temp;
	if(!l) return(l);
	else
	{
		nxt = l->next;
		l->next = NULL;
		while(nxt)
		{
			temp = nxt->next;
			nxt->next = l;
			l = nxt;
			nxt = temp;
		}
		return(l);
	}
}


/* Recursive function for deleting a node from the list */
list *delete(list *st, int n)
{
```

```
        list *tmp;
        if(!st) return(st);
        if(st->data == n)
        {
                tmp = st;
                st = st->next;
                free(tmp);
                return(st);
        }
        st->next = delete(st->next,n);
        return(st);
}
```

Exercises

1. Identify which one of the following declaration correctly defines a self referential data structure.

```
a) struct class_1 {
        char name[31];
        . . .
        struct class_2 *c;
  };

b) struct class_1 {
        char name[31];
        . . .
        struct class_1 *c;
  };

c) struct class_1 {
        char name[31];
        . . .
        struct class_1 c;
  };
```

2. Identify the errors (if any) in the following C programs.

a) The function *addList* inserts a new node (nw) at the beginning of the list pointed to by the start pointer.

```
addList(list *start, list *nu)
{
        if(start)
                start = nu;
```

```
        else
                nw->nxt = start;
}
```

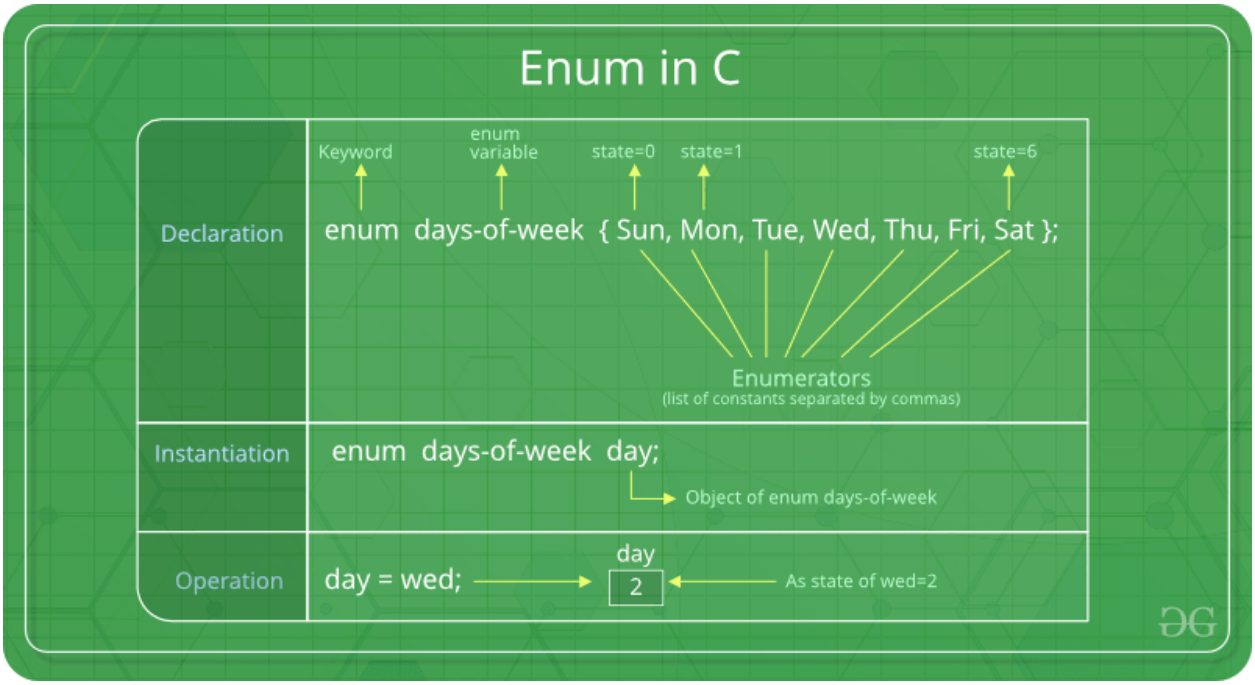
b) The function displayList (start pointer is passed as an argument) recursively displays all the nodes in the list.

```
displayList(list *st)
{
        if(!st) return;
        printf("%d ", st->data);
        displayList(st->nxt);
}
```

Enumeration data type

Enumeration (or enum) in C

Enumeration (or enum) is a user defined data type in C. It is mainly used to assign names to integral constants, the names make a program easy to read and maintain.




```
enum State {Working = 1, Failed = 0};
```

The keyword ‘enum’ is used to declare new enumeration types in C and C++. Following is an example of enum declaration.

```
// The name of enumeration is "flag" and the constant
// are the values of the flag. By default, the values
// of the constants are as follows:
// constant1 = 0, constant2 = 1, constant3 = 2 and
// so on.
enum flag{constant1, constant2, constant3,.....};
```

Variables of type enum can also be defined. They can be defined in two ways:

```
// In both of the below cases, "day" is
// defined as the variable of type week.


enum week{Mon, Tue, Wed};
enum week day;


// Or

enum week{Mon, Tue, Wed}day;
```

```
// An example program to demonstrate working
// of enum in C
#include<stdio.h>

enum week{Mon, Tue, Wed, Thur, Fri, Sat, Sun};

int main()
{
   enum week day;
   day = Wed;
   printf("%d",day);
   return 0;
}
```

Output:

```
2
```

In the above example, we declared “day” as the variable and the value of “Wed” is allocated to day, which is 2. So as a result, 2 is printed.

Another example of enumeration is:

```
// Another example program to demonstrate working
// of enum in C
#include<stdio.h>

enum year{Jan, Feb, Mar, Apr, May, Jun, Jul,
          Aug, Sep, Oct, Nov, Dec};

int main()
{
   int i;
   for (i=Jan; i<=Dec; i++)
      printf("%d ", i);

   return 0;
}
```

Output:

```
0 1 2 3 4 5 6 7 8 9 10 11
```

In this example, the for loop will run from i = 0 to i = 11, as initially the value of i is Jan which is 0 and the value of Dec is 11.

Interesting facts about initialization of enum.

1. Two enum names can have same value. For example, in the following C program both 'Failed' and 'Freezed' have same value 0.

```
#include <stdio.h>
enum State {Working = 1, Failed = 0, Freezed = 0};

int main()
{
   printf("%d, %d, %d", Working, Failed, Freezed);
   return 0;
}
```

Output:

```
1, 0, 0
```

2. If we do not explicitly assign values to enum names, the compiler by default assigns values starting from 0. For example, in the following C program, sunday gets value 0, monday gets 1, and so on.

```
#include <stdio.h>
enum day {sunday, monday, tuesday, wednesday, thursday, friday, saturday};

int main()
{
   enum day d = thursday;
   printf("The day number stored in d is %d", d);
   return 0;
}
```
Output:

```
The day number stored in d is 4
```

3. We can assign values to some name in any order. All unassigned names get value as value of previous name plus one.
filter_none
edit
play_arrow
brightness_4

```
#include <stdio.h>
enum day {sunday = 1, monday, tuesday = 5,
        wednesday, thursday = 10, friday, saturday};

int main()
{
   printf("%d %d %d %d %d %d %d", sunday, monday, tuesday,
         wednesday, thursday, friday, saturday);
   return 0;
}
```
Output:

```
1 2 5 6 10 11 12
```

4. The value assigned to enum names must be some integeral constant, i.e., the value must be in range from minimum possible integer value to maximum possible integer value.

5. All enum constants must be unique in their scope. For example, the following program fails in compilation.

```
enum state  {working, failed};
enum result {failed, passed};
```

```
int main()  { return 0; }
```
Output:

```
Compile Error: 'failed' has a previous declaration as 'state failed'
```

Exercise:
Predict the output of following C programs
Program 1:

```
#include <stdio.h>
enum day {sunday = 1, tuesday, wednesday, thursday, friday, saturday};

int main()
{
   enum day d = thursday;
   printf("The day number stored in d is %d", d);
   return 0;
}
```

Program 2:

```
#include <stdio.h>
enum State {WORKING = 0, FAILED, FREEZED};
enum State currState = 2;

enum State FindState() {
   return currState;
}

int main() {
  (FindState() == WORKING)? printf("WORKING"): printf("NOT WORKING");
  return 0;
}
```

Enum vs Macro
We can also use macros to define names constants. For example we can define ‘Working’ and ‘Failed’ using following macro.

```
#define Working 0
#define Failed 1
#define Freezed 2
```

There are multiple advantages of using enum over macro when many related named constants have integral values.
a) Enums follow scope rules.
b) Enum variables are automatically assigned values. Following is simpler

enum state {Working, Failed, Freezed};

Files: Text and Binary files, Creating and Reading and writing text and binary files,

Appending data to existing files, Writing and reading structures using binary files, Random access using fseek, ftell and rewind functions.

Files

A file is a container in computer storage devices used for storing data.

Why files are needed?

- When a program is terminated, the entire data is lost. Storing in a file will preserve your data even if the program terminates.
- If you have to enter a large number of data, it will take a lot of time to enter them all. However, if you have a file containing all the data, you can easily access the contents of the file using a few commands in C.
- You can easily move your data from one computer to another without any changes.

Types of Files

When dealing with files, there are two types of files you should know about:

1. Text files
2. Binary files

1. Text files

Text files are the normal **.txt** files. You can easily create text files using any simple text editors such as Notepad.

When you open those files, you'll see all the contents within the file as plain text. You can easily edit or delete the contents.

They take minimum effort to maintain, are easily readable, and provide the least security and takes bigger storage space.

2. Binary files

Binary files are mostly the **.bin** files in your computer.

Instead of storing data in plain text, they store it in the binary form (0's and 1's).

They can hold a higher amount of data, are not readable easily, and provides better security than text files.

File Operations

In C, you can perform four major operations on files, either text or binary:

1. Creating a new file
2. Opening an existing file
3. Closing a file
4. Reading from and writing information to a file

Working with files

When working with files, you need to declare a pointer of type file. This declaration is needed for communication between the file and the program.

```
1. FILE *fptr;
```

Opening a file - for creation and edit

Opening a file is performed using the `fopen()` function defined in the `stdio.h` header file.

The syntax for opening a file in standard I/O is:

```
1. ptr = fopen("fileopen","mode");
```

For example,

```
1. fopen("E:\\cprogram\\newprogram.txt","w");
2.
3. fopen("E:\\cprogram\\oldprogram.bin","rb");
```

- Let's suppose the file newprogram.txt doesn't exist in the location E:\cprogram. The first function creates a new file named newprogram.txt and opens it for writing as per the mode '**w**'.
  The writing mode allows you to create and edit (overwrite) the contents of the file.
- Now let's suppose the second binary file oldprogram.bin exists in the location E:\cprogram. The second function opens the existing file for reading in binary mode '**rb**'.
  The reading mode only allows you to read the file, you cannot write into the file.

Mode	Meaning of Mode	During Inexistence of file
R	Open for reading.	If the file does not exist, fopen() returns NULL.
Rb	Open for reading in binary mode.	If the file does not exist, fopen() returns NULL.
W	Open for writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
wb	Open for writing in binary mode.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
A	Open for append. Data is added to the end of the file.	If the file does not exist, it will be created.
Ab	Open for append in binary mode. Data is added to the end of the file.	If the file does not exist, it will be created.
r+	Open for both reading and writing.	If the file does not exist, fopen() returns NULL.
rb+	Open for both reading and writing in binary mode.	If the file does not exist, fopen() returns NULL.
w+	Open for both reading and writing.	If the file exists, its contents are

Mode	Meaning of Mode	During Inexistence of file
		overwritten. If the file does not exist, it will be created.
wb+	Open for both reading and writing in binary mode.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a+	Open for both reading and appending.	If the file does not exist, it will be created.
ab+	Open for both reading and appending in binary mode.	If the file does not exist, it will be created.

Opening Modes in Standard I/O

Closing a File

The file (both text and binary) should be closed after reading/writing.

Closing a file is performed using the fclose() function.

```
1. fclose(fptr);
```

Here, fptr is a file pointer associated with the file to be closed.

Reading and writing to a text file

For reading and writing to a text file, we use the functions fprintf() and fscanf().

They are just the file versions of printf() and scanf(). The only difference is that fprint() and fscanf() expects a pointer to the structure FILE.

Example 1: Write to a text file

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

```c
#include <stdlib.h>

int main()
{
  int num;
  FILE *fptr;

  // use appropriate location if you are using MacOS or Linux
  fptr = fopen("C:\\program.txt","w");

  if(fptr == NULL)
  {
     printf("Error!");
     exit(1);
  }

  printf("Enter num: ");
  scanf("%d",&num);

  fprintf(fptr,"%d",num);
  fclose(fptr);

  return 0;
}
```

This program takes a number from the user and stores in the file program.txt.
After you compile and run this program, you can see a text file program.txt created in C drive of your computer. When you open the file, you can see the integer you entered.

Example 2: Read from a text file

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

int main()
{
  int num;
  FILE *fptr;

  if ((fptr = fopen("C:\\program.txt","r")) == NULL){
     printf("Error! opening file");

     // Program exits if the file pointer returns NULL.
     exit(1);
  }

```

```
  fscanf(fptr,"%d", &num);

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

  return 0;
}
```

This program reads the integer present in the program.txt file and prints it onto the screen.
If you successfully created the file from **Example 1**, running this program will get you the integer you entered.
Other functions like fgetchar(), fputc() etc. can be used in a similar way.

Reading and writing to a binary file

Functions fread() and fwrite() are used for reading from and writing to a file on the disk respectively in case of binary files.

Writing to a binary file

To write into a binary file, you need to use the fwrite() function. The functions take four arguments:

1. address of data to be written in the disk
2. size of data to be written in the disk
3. number of such type of data
4. pointer to the file where you want to write.

```
fwrite(addressData, sizeData, numbersData, pointerToFile);
```

Example 3: Write to a binary file using fwrite()

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

struct threeNum
{
  int n1, n2, n3;
};

int main()
{
  int n;
  struct threeNum num;
  FILE *fptr;

  if ((fptr = fopen("C:\\program.bin","wb")) == NULL){
      printf("Error! opening file");

      // Program exits if the file pointer returns NULL.
```

```
19.       exit(1);
20.    }
21.
22.    for(n = 1; n < 5; ++n)
23.    {
24.      num.n1 = n;
25.      num.n2 = 5*n;
26.      num.n3 = 5*n + 1;
27.      fwrite(&num, sizeof(struct threeNum), 1, fptr);
28.    }
29.    fclose(fptr);
30.
31.    return 0;
32. }
```

In this program, we create a new file program.bin in the C drive.
We declare a structure threeNum with three numbers - n1, n2 and n3, and define it in the main function as num.
Now, inside the for loop, we store the value into the file using fwrite().
The first parameter takes the address of num and the second parameter takes the size of the structure threeNum.
Since we're only inserting one instance of num, the third parameter is 1. And, the last parameter *fptr points to the file we're storing the data.

Finally, we close the file.

Reading from a binary file

Function fread() also take 4 arguments similar to the fwrite() function as above.

```
1.  fread(addressData, sizeData, numbersData, pointerToFile);
```

Example 4: Read from a binary file using fread()

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.
4.  struct threeNum
5.  {
6.    int n1, n2, n3;
7.  };
8.
9.  int main()
10. {
11.   int n;
```

```
12.  struct threeNum num;
13.  FILE *fptr;
14.
15.  if ((fptr = fopen("C:\\program.bin","rb")) == NULL){
16.      printf("Error! opening file");
17.
18.      // Program exits if the file pointer returns NULL.
19.      exit(1);
20.  }
21.
22.  for(n = 1; n < 5; ++n)
23.  {
24.    fread(&num, sizeof(struct threeNum), 1, fptr);
25.    printf("n1: %d\tn2: %d\tn3: %d", num.n1, num.n2, num.n3);
26.  }
27.  fclose(fptr);
28.
29.  return 0;
30. }
```

In this program, you read the same file program.bin and loop through the records one by one.
In simple terms, you read one threeNum record of threeNum size from the file pointed by *fptr into the structure num.
You'll get the same records you inserted in **Example 3**.

Getting data using fseek()

If you have many records inside a file and need to access a record at a specific position, you need to loop through all the records before it to get the record.

This will waste a lot of memory and operation time. An easier way to get to the required data can be achieved using fseek().
As the name suggests, fseek() seeks the cursor to the given record in the file.

Syntax of fseek()

```
1.  fseek(FILE * stream, long int offset, int whence);
```

The first parameter stream is the pointer to the file. The second parameter is the position of the record to be found, and the third parameter specifies the location where the offset starts.

Whence	Meaning
SEEK_SET	Starts the offset from the beginning of the file.
SEEK_END	Starts the offset from the end of the file.
SEEK_CUR	Starts the offset from the current location of the cursor in the file.

Different whence in fseek()

Example 5: fseek()

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.
4.  struct threeNum
5.  {
6.    int n1, n2, n3;
7.  };
8.
9.  int main()
10. {
11.   int n;
12.   struct threeNum num;
13.   FILE *fptr;
14.
15.   if ((fptr = fopen("C:\\program.bin","rb")) == NULL){
16.       printf("Error! opening file");
17.
18.       // Program exits if the file pointer returns NULL.
19.       exit(1);
20.   }
21.
22.   // Moves the cursor to the end of the file
23.   fseek(fptr, -sizeof(struct threeNum), SEEK_END);
24.
25.   for(n = 1; n < 5; ++n)
26.   {
27.     fread(&num, sizeof(struct threeNum), 1, fptr);
```

```
28.     printf("n1: %d\tn2: %d\tn3: %d\n", num.n1, num.n2, num.n3);
29.     fseek(fptr, -2*sizeof(struct threeNum), SEEK_CUR);
30.   }
31.   fclose(fptr);
32.
33.   return 0;
34. }
```

This program will start reading the records from the file program.bin in the reverse order (last to first) and prints it.

https://www.programiz.com/c-programming/c-file-input-output

Appending data to existing files

C Program to Append the Content of File at the end of Another

This C Program appends the content of file at the end of another.
Here is source code of the C Program to append the content of file at the end of another. The C program is successfully compiled and run on a Linux system. The program output is also shown below.

```
1. /*
2.  * C Program to Append the Content of File at the end of Another
3.  */
4. #include <stdio.h>
5. #include <stdlib.h>
6.
7. main()
8. {
9.     FILE *fsring1, *fsring2, *ftemp;
10.    char ch, file1[20], file2[20], file3[20];
11.
12.    printf("Enter name of first file ");
13.    gets(file1);
14.    printf("Enter name of second file ");
15.    gets(file2);
16.    printf("Enter name to store merged file ");
17.    gets(file3);
18.    fsring1 = fopen(file1, "r");
19.    fsring2 = fopen(file2, "r");
20.    if (fsring1 == NULL || fsring2 == NULL)
21.    {
22.        perror("Error has occured");
```

```
23.         printf("Press any key to exit...\n");
24.         exit(EXIT_FAILURE);
25.     }
26.     ftemp = fopen(file3, "w");
27.     if (ftemp == NULL)
28.     {
29.         perror("Error has occures");
30.         printf("Press any key to exit...\n");
31.         exit(EXIT_FAILURE);
32.     }
33.     while ((ch = fgetc(fsring1)) != EOF)
34.         fputc(ch, ftemp);
35.     while ((ch = fgetc(fsring2) ) != EOF)
36.         fputc(ch, ftemp);
37.     printf("Two files merged  %s successfully.\n", file3);
38.     fclose(fsring1);
39.     fclose(fsring2);
40.     fclose(ftemp);
41.     return 0;
42. }
```

```
$ cc pgm47.c
$ a.out
Enter name of first file a.txt
Enter name of second file b.txt
Enter name to store merged file merge.txt
Two files merged merge.txt successfully.
```

Write a C program to read data from user and append data into a file. How to append data at end of a file in C programming. In this post I will explain append mode in file handling. I will cover how to append data into a file in C using append file mode.

Example

Source file content

I love programming.
Programming with files is fun.

String to append

Learning C programming at Codeforwin is simple and easy.

Output file content after append

I love programming.
Programming with files is fun.
Learning C programming at Codeforwin is simple and easy.

Required knowledge

Basic Input Output, Do while loop, Pointers, File Handling

In previous two posts, I explained to create a file and write data into file and how to read a file.
In this post we will continue further and learn to append data into a file.

How to append data into a file?

C programming supports different file open mode to perform different operations on file. To append data into a file you can use a file open mode.

Step by step descriptive logic to append data into a file.

- Input file path from user to append data, store it in some variable say filePath.
- Declare a FILE type pointer variable say, fPtr.
- Open file in a (append file) mode and store reference to fPtr using fPtr = fopen(filePath, "a");.
- Input data to append to file from user, store it to some variable say dataToAppend.
- Write data to append into file using fputs(dataToAppend, fPtr);.
- Finally close file to save all changes. Use fclose(fPtr);.

Program to append data into a file

```
/**
 * C program to append data to a file
 */

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

#define BUFFER_SIZE 1000


void readFile(FILE * fPtr);


int main()
```

```
{
   /* File pointer to hold reference of input file */
   FILE *fPtr;
   char filePath[100];

   char dataToAppend[BUFFER_SIZE];


   /* Input file path to remove empty lines from user */
   printf("Enter file path: ");
   scanf("%s", filePath);

   /*  Open all file in append mode. */
   fPtr = fopen(filePath, "a");


   /* fopen() return NULL if unable to open file in given mode. */
   if (fPtr == NULL)
   {
      /* Unable to open file hence exit */
      printf("\nUnable to open '%s' file.\n", filePath);
      printf("Please check whether file exists and you have write privilege.\n");
      exit(EXIT_FAILURE);
   }


   /* Input data to append from user */
   printf("\nEnter data to append: ");
   fflush(stdin);          // To clear extra white space characters in stdin
   fgets(dataToAppend, BUFFER_SIZE, stdin);


   /* Append data to file */
   fputs(dataToAppend, fPtr);


   /* Reopen file in read mode to print file contents */
   fPtr = freopen(filePath, "r", fPtr);

   /* Print file contents after appending string */
   printf("\nSuccessfully appended data to file. \n");
   printf("Changed file contents:\n\n");
   readFile(fPtr);


   /* Done with file, hence close file. */
```

```
    fclose(fPtr);

    return 0;
}



/**
 * Reads a file character by character
 * and prints on console.
 *
 * @fPtr    Pointer to FILE to read.
 */
void readFile(FILE * fPtr)
{
    char ch;

    do
    {
        ch = fgetc(fPtr);

        putchar(ch);

    } while (ch != EOF);
}
```

data/append.txt file contents before appending string.

```
I love programming.
Programming with files is fun.
```

Output

```
Enter file path: data\append.txt

Enter data to append: Learning C programming at Codeforwin is simple and easy.

Successfully appended data to file.
Changed file contents:

I love programming.
Programming with files is fun.
Learning C programming at Codeforwin is simple and easy.
```

Writing and reading structures using binary files

Read/Write structure to a file in C

Prerequisite: Structure in C
For writing in file, it is easy to write string or int to file using **fprintf** and **putc**, but you might have faced difficulty when writing contents of struct. **fwrite** and **fread** make task easier when you want to write and read blocks of data.

1. **fwrite :** Following is the declaration of fwrite function
2. **size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)**
3. **ptr** - This is pointer to array of elements to be written
4. **size** - This is the size in bytes of each element to be written
5. **nmemb** - This is the number of elements, each one with a size of size bytes
6. **stream** - This is the pointer to a FILE object that specifies an output stream

filter_none

edit
play_arrow
brightness_4

```
// C program for writing

// struct to file

#include <stdio.h>

#include <stdlib.h>

#include <string.h>


// a struct to read and write

struct person

{

   int id;

   char fname[20];

   char lname[20];

};
```

```
int main ()
{
   FILE *outfile;

   // open file for writing
   outfile = fopen ("person.dat", "w");
   if (outfile == NULL)
   {
      fprintf(stderr, "\nError opend file\n");
      exit (1);
   }

   struct person input1 = {1, "rohan", "sharma"};
   struct person input2 = {2, "mahendra", "dhoni"};

   // write struct to file
   fwrite (&input1, sizeof(struct person), 1, outfile);
   fwrite (&input2, sizeof(struct person), 1, outfile);

   if(fwrite != 0)
      printf("contents to file written successfully !\n");
   else
      printf("error writing file !\n");

   // close file
   fclose (outfile);
```

```
    return 0;
}
```

Output:

```
gcc demowrite.c
./a.out
contents to file written successfully!
```

7. **fread :** Following is the declaration of fread function
8. **size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)**
9. **ptr** - This is the pointer to a block of memory with a minimum size of size*nmemb bytes.
10. **size** - This is the size in bytes of each element to be read.
11. **nmemb** - This is the number of elements, each one with a size of size bytes.
    stream - This is the pointer to a FILE object that specifies an input stream.

filter_none

edit
play_arrow
brightness_4

```
// C program for reading
// struct from a file
#include  <stdio.h>
#include <stdlib.h>

// struct person with 3 fields
struct person
{
    int id;
    char fname[20];
    char lname[20];
};
```

```
// Driver program
int main ()
{
    FILE *infile;
    struct person input;

    // Open person.dat for reading
    infile = fopen ("person.dat", "r");
    if (infile == NULL)
    {
        fprintf(stderr, "\nError opening file\n");
        exit (1);
    }

    // read file contents till end of file
    while(fread(&input, sizeof(struct person), 1, infile))
        printf ("id = %d name = %s %s\n", input.id,
        input.fname, input.lname);

    // close file
    fclose (infile);

    return 0;
}
```

Output:

```
gcc demoread.c
```

```
./a.out
id = 1   name = rohan sharma
id = 2   name = mahendra dhoni
```

Reading and Writing from Binary Files

Binary files are very similar to arrays except for the fact that arrays are temporary storage in the memory but binary files are permanent storage in the disks. The most important difference between binary files and a text file is that in a binary file, you can *seek*, *write*, or *read* from any position inside the file and insert structures directly into the files. You must be wondering - why do we need binary files when we already know how to handle plaintexts and text files? Here are the **reasons why binary files are necessary**:

1. I/O operations are much faster with binary data.

Usually, large text files contain millions of numbers. It takes a lot of time to convert 32-bit integers to readable characters. This conversion is not required in the case of binary files as data can be directly stored in the form of bits.

2. Binary files are much smaller in size than text files.

For data that is in the form of images, audio or video, this is very important. Small size means less storage space and faster transmission. For example, a storage device can store a large amount of binary data as compared to data in character format.

3. Some data cannot be converted to character formats.

For example, Java compilers generate bytecodes after compilation.

Having said that, let's move on to handling I/O operations in a binary file in C. The **basic parameters that the read and write functions of binary files accept** are:

- the memory address of the value to be written or read
- the number of bytes to read per block
- the total number of blocks to read
- the file pointer

There are functions provided by C libraries to *seek*, *read*, and *write* to binary files. Let's explain this by reading and writing a structure called rec in a binary file. The structure is defined like this:

```
1.  /* The structure to be inserted in the binary file */
2.  struct record
3.  {
4.   int a,b,c;
```

```
5. };
```

The **fread() function** is used to read a specific number of bytes from the file. An example of fread() looks like this:

```
1. fread(&myRecord, sizeof(struct record), 1, ptr);
```

This statement reads 'a' bytes (in this case, it's the size of the structure) from the file into the memory address *&myRecord*. Here the number 1 denotes the number of blocks of 'a' bytes to be read. If we change it to 10, then it denotes 10 blocks of 'a' bytes will be read and stored into *&myRecord*. *ptr* is the pointer to the location of the file that is being read.

Now the **fwrite() function** is used to write to a binary file, like so:

```
1. fwrite(&myRecord, sizeof(struct record), 1, ptr);
```

In this example, the value inside the address *&myRecord* which is of the size of the structure *record* is written into the file with the help of the file pointer *ptr*.

Examples using fread() & frwrite()

Now that you know how to read and write binary files, let's discuss this with the help of examples, starting with an example of using fread().

```
1. #include<stdio.h>
2. /* Our structure */
3. struct record
4. {
5.   int a,b,c;
6. };
7. int main()
```

```
8.  {
9.   int count;
10.  FILE *ptr;
11.  struct record myRecord;
12.  ptr=fopen("test.bin","rb");
13.  if (!ptr)
14.  {
15.    printf("Unable to open file!");     return 1;
16.  }
17.  for ( count=1; count <= 10; count++)
18.  {
19.   fread(&myRecord,sizeof(struct record),1,ptr);   printf("%d\n",myRecord.a);
20.  }  fclose(ptr);
21.  return 0;
22. }
```

In this example, we tried to read each structure from the binary file using the fread() function. This reads the structure one byte at a time and stores it inside the address *myRecord*. Let's now look at an example using fwrite().

Random access using fseek, ftell and rewind functions.

fseek(), ftell() and rewind() in Files of Programming in C

fseek() - It is used to moves the reading control to different positions using fseek function. ftell() - It tells the byte location of current position in file pointer. rewind() - It moves the control to beginning of a file.

Program

```
1   #include
2   void main(){
3      FILE *fp;
4      int i;
5      clrscr();
6      fp = fopen("CHAR.txt","r");
7      for (i=1;i&lt;=10;i++){
8         printf("%c : %d\n",getc(fp),ftell(fp));
9         fseek(fp,ftell(fp),0);
10        if (i == 5)
11        rewind(fp);
12     }
13     fclose(fp);
14  }
```

Output

```
W : 0
e : 1
l : 2
c : 3
o : 4
W : 0
e : 1
l : 2
c : 3
o : 4
```

we will learn to randomly access file data in C programming language.

we will be discussing the fseek(), ftell() and rewind() functions to access data in a file.

The ftell function

The ftell() function tells us about the current position in the file (in bytes).

Syntax:

```
pos = ftell(fptr);
```

Where, fptr is a file pointer. pos holds the current position i.e., total bytes read (or written).

Example:

If a file has 10 bytes of data and if the ftell() function returns 4 then, it means that 4 bytes has already been read (or written).

The rewind function

We use the rewind() function to return back to the starting point in the file.

Syntax:

```
rewind(fptr);
```

Where, fptr is a file pointer.

The fseek function

We use the fseek() function to move the file position to a desired location.

Syntax:

```
fseek(fptr, offset, position);
```

Where, fptr is the file pointer. offset which is of type long, specifies the number of positions (in bytes) to move in the file from the location specified by the position.

The position can take the following values.

- 0 - The beginning of the file
- 1 - The current position in the file
- 2 - End of the file

Following are the list of operations we can perform using the `fseek()` function.

Operation	Description
`fseek(fptr, 0, 0)`	This will take us to the beginning of the file.
`fseek(fptr, 0, 2)`	This will take us to the end of the file.
`fseek(fptr, N, 0)`	This will take us to (N + 1)th bytes in the file.
`fseek(fptr, N, 1)`	This will take us N bytes forward from the current position in the file.
`fseek(fptr, -N, 1)`	This will take us N bytes backward from the current position in the file.
`fseek(fptr, -N, 2)`	This will take us N bytes backward from the end position in the file.

Write a program in C to save alphabet A to Z in file and then print the letters using fseek function

---

```
#include <stdio.h>


int main(void) {


  // integer variable

  int i;


  // character variable
```

```
char ch;
  // file pointer
FILE *fptr;
  // open file in write mode
fptr = fopen("char", "w");
if (fptr != NULL) {
  printf("File created successfully!\n");
}
else {
  printf("Failed to create the file.\n");
  // exit status for OS that an error occured
  return -1;
}
  // write data in file
for (ch = 'A'; ch <= 'Z'; ch++) {
  putc(ch, fptr);
}
  // close connection
fclose(fptr);
  // reference
printf("\nReference:\n");
for (i = 0; i < 26; i++) {
```

```
  printf("%d ", (i+1));
}
printf("\n");
for (i = 65; i <= 90; i++) {
  // print character
  printf("%c", (i));
      // manage space
  if (i - 65 >= 9) {
    printf("  ");
  }
  else {
    printf(" ");
  }
}
printf("\n\n");
  // open file for reading
fptr = fopen("char", "r");
  printf("Curr pos: %ld\n", ftell(fptr));
  // read 1st char in the file
fseek(fptr, 0, 0);
ch = getc(fptr);
printf("1st char: %c\n", ch);
```

```
  printf("Curr pos: %ld\n", ftell(fptr));
  // read 5th char in the file
fseek(fptr, 4, 0);
ch = getc(fptr);
printf("5th char: %c\n", ch);
  printf("Curr pos: %ld\n", ftell(fptr));
  // read 26th char in the file
fseek(fptr, 25, 0);
ch = getc(fptr);
printf("26th char: %c\n", ch);
  printf("Curr pos: %ld\n", ftell(fptr));
  // rewind
printf("rewind\n");
rewind(fptr);
  printf("Curr pos: %ld\n", ftell(fptr));
  // read 10th char in the file
fseek(fptr, 9, 0);
ch = getc(fptr);
printf("10th char: %c\n", ch);
  printf("Curr pos: %ld\n", ftell(fptr));
  // read 15th char in the file
fseek(fptr, 4, 1);    // move 4 bytes forward from current position
```

```
  ch = getc(fptr);
  printf("15th char: %c\n", ch);
    printf("Curr pos: %ld\n", ftell(fptr));
    // read 20th char in the file
  fseek(fptr, 4, 1);    // move 4 bytes forward from current position
  ch = getc(fptr);
  printf("20th char: %c\n", ch);
    printf("Curr pos: %ld\n", ftell(fptr));
    // close connection
  fclose(fptr);
    return 0;
}
```

Output:

File created successfully!

Reference:

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Curr pos: 0

1st char: A

Curr pos: 1

5th char: E

Curr pos: 5

26th char: Z

Curr pos: 26

rewind

Curr pos: 0

10th char: J

Curr pos: 10

15th char: O

Curr pos: 15

20th char: T

Curr pos: 20

MODULE – IV : FUNCTION AND DYNAMIC MEMORY ALLOCATION:

Functions: Designing structured programs, Declaring a function, Signature of a function, Parameters and return type of a function, passing parameters to functions, call by value, Passing arrays to functions, passing pointers to functions, idea of call by reference, Some C standard functions and libraries.

Recursion: Simple programs, such as Finding Factorial, Fibonacci series etc., Limitations of Recursive functions.

Dynamic memory allocation: Allocating and freeing memory, Allocating memory for arrays of different data types.

Functions:

In c, we can divide a large program into the basic building blocks known as function. The function contains the set of programming statements enclosed by {}. A function can be called multiple times to provide reusability and modularity to the C program. In other words, we can say that the collection of functions creates a program. The function is also known as *procedure* or *subroutine* in other programming languages.

Advantage of functions in C

There are the following advantages of C functions.

- By using functions, we can avoid rewriting same logic/code again and again in a program.
- We can call C functions any number of times in a program and from any place in a program.
- We can track a large C program easily when it is divided into multiple functions.
- Reusability is the main achievement of C functions.
- However, Function calling is always a overhead in a C program.

Function Aspects

There are three aspects of a C function.

- **Function declaration** A function must be declared globally in a c program to tell the compiler about the function name, function parameters, and return type.

- **Function call** Function can be called from anywhere in the program. The parameter list must not differ in function calling and function declaration. We must pass the same number of functions as it is declared in the function declaration.

- **Function definition** It contains the actual statements which are to be executed. It is the most important aspect to which the control comes when the function is called. Here, we must notice that only one value can be returned from the function.

SN	C function aspects	Syntax
1	Function declaration	return_type function_name (argument list);
2	Function call	function_name (argument_list)
3	Function definition	return_type function_name (argument list) {function body;}

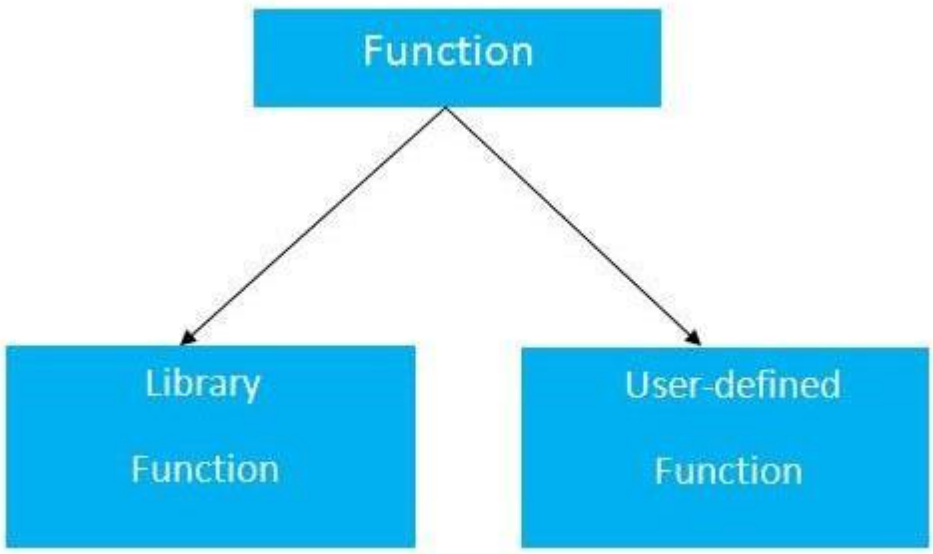
The syntax of creating function in c language is given below:

```
return_type function_name(data_type parameter...){
//code to be executed
}
```

Types of Functions

There are two types of functions in C programming:

1. **Library Functions**: are the functions which are declared in the C header files such as scanf(), printf(), gets(), puts(), ceil(), floor() etc.
2. **User-defined functions**: are the functions which are created by the C programmer, so that he/she can use it many times. It reduces the complexity of a big program and optimizes the code.



Return Value

A C function may or may not return a value from the function. If you don't have to return any value from the function, use void for the return type.

Let's see a simple example of C function that doesn't return any value from the function.

Example without return value:

```
1. void hello(){
2. printf("hello c");
3. }
```

If you want to return any value from the function, you need to use any data type such as int, long, char, etc. The return type depends on the value to be returned from the function.

Let's see a simple example of C function that returns int value from the function.

Example with return value:

```
1. int get(){
2. return 10;
3. }
```

In the above example, we have to return 10 as a value, so the return type is int. If you want to return floating-point value (e.g., 10.2, 3.1, 54.5, etc), you need to use float as the return type of the method.

```
1. float get(){
2. return 10.2;
3. }
```

Now, you need to call the function, to get the value of the function.

Different aspects of function calling

A function may or may not accept any argument. It may or may not return any value. Based on these facts, There are four different aspects of function calls.

- function without arguments and without return value
- function without arguments and with return value
- function with arguments and without return value
- function with arguments and with return value

Example for Function without argument and return value

Example 1

```
1. #include<stdio.h>
2. void printName();
3. void main ()
4. {
5.     printf("Hello ");
6.     printName();
7. }
8. void printName()
9. {
10.     printf("Javatpoint");
11. }
```

Output

```
Hello Javatpoint
```

Example 2

```
1. #include<stdio.h>
2. void sum();
3. void main()
4. {
5.     printf("\nGoing to calculate the sum of two numbers:");
6.     sum();
7. }
8. void sum()
9. {
10.     int a,b;
11.     printf("\nEnter two numbers");
12.     scanf("%d %d",&a,&b);
13.     printf("The sum is %d",a+b);
14. }
```

Output

```
Going to calculate the sum of two numbers:

Enter two numbers 10
24

The sum is 34
```

Example for Function without argument and with return value

Example 1

```
1. #include<stdio.h>
2. int sum();
3. void main()
4. {
5.     int result;
6.     printf("\nGoing to calculate the sum of two numbers:");
7.     result = sum();
8.     printf("%d",result);
9. }
10. int sum()
11. {
```

```
12.    int a,b;
13.    printf("\nEnter two numbers");
14.    scanf("%d %d",&a,&b);
15.    return a+b;
16. }
```

Output

```
Going to calculate the sum of two numbers:

Enter two numbers 10
24

The sum is 34
```

Example 2: program to calculate the area of the square

```
1.  #include<stdio.h>
2.  int sum();
3.  void main()
4.  {
5.     printf("Going to calculate the area of the square\n");
6.     float area = square();
7.     printf("The area of the square: %f\n",area);
8.  }
9.  int square()
10. {
11.    float side;
12.    printf("Enter the length of the side in meters: ");
13.    scanf("%f",&side);
14.    return side * side;
15. }
```

Output

```
Going to calculate the area of the square
Enter the length of the side in meters: 10
The area of the square: 100.000000
```

Example for Function with argument and without return value

Example 1

```
1. #include<stdio.h>
2. void sum(int, int);
3. void main()
4. {
5.     int a,b,result;
6.     printf("\nGoing to calculate the sum of two numbers:");
7.     printf("\nEnter two numbers:");
8.     scanf("%d %d",&a,&b);
9.     sum(a,b);
10. }
11. void sum(int a, int b)
12. {
13.     printf("\nThe sum is %d",a+b);
14. }
```

Output

```
Going to calculate the sum of two numbers:

Enter two numbers 10
24
The sum is 34
```

Example 2: program to calculate the average of five numbers.

```
1. #include<stdio.h>
2. void average(int, int, int, int, int);
3. void main()
4. {
5.     int a,b,c,d,e;
6.     printf("\nGoing to calculate the average of five numbers:");
7.     printf("\nEnter five numbers:");
8.     scanf("%d %d %d %d %d",&a,&b,&c,&d,&e);
9.     average(a,b,c,d,e);
10. }
11. void average(int a, int b, int c, int d, int e)
```

```
{
    float avg;
    avg = (a+b+c+d+e)/5;
    printf("The average of given five numbers : %f",avg);
}
```

Output

```
Going to calculate the average of five numbers:
Enter five numbers:10
20
30
40
50
The average of given five numbers : 30.000000
```

Example for Function with argument and with return value

Example 1

```
#include<stdio.h>
int sum(int, int);
void main()
{
    int a,b,result;
    printf("\nGoing to calculate the sum of two numbers:");
    printf("\nEnter two numbers:");
    scanf("%d %d",&a,&b);
    result = sum(a,b);
    printf("\nThe sum is : %d",result);
}
int sum(int a, int b)
{
    return a+b;
}
```

Output

```
Going to calculate the sum of two numbers:
Enter two numbers:10
20
```

```
The sum is : 30
```

Example 2: Program to check whether a number is even or odd

```
1.  #include<stdio.h>
2.  int even_odd(int);
3.  void main()
4.  {
5.   int n,flag=0;
6.   printf("\nGoing to check whether a number is even or odd");
7.   printf("\nEnter the number: ");
8.   scanf("%d",&n);
9.   flag = even_odd(n);
10.  if(flag == 0)
11.  {
12.     printf("\nThe number is odd");
13.  }
14.  else
15.  {
16.     printf("\nThe number is even");
17.  }
18. }
19. int even_odd(int n)
20. {
21.     if(n%2 == 0)
22.     {
23.         return 1;
24.     }
25.     else
26.     {
27.         return 0;
28.     }
29. }
```

Output

```
Going to check whether a number is even or odd
Enter the number: 100
```

```
The number is even
```

C Library Functions

Library functions are the inbuilt function in C that are grouped and placed at a common place called the library. Such functions are used to perform some specific operations. For example, printf is a library function used to print on the console. The library functions are created by the designers of compilers. All C standard library functions are defined inside the different header files saved with the extension **.h**. We need to include these header files in our program to make use of the library functions defined in such header files. For example, To use the library functions such as printf/scanf we need to include stdio.h in our program which is a header file that contains all the library functions regarding standard input/output.

The list of mostly used header files is given in the following table.

SN	Header file	Description
1	stdio.h	This is a standard input/output header file. It contains all the library functions regarding standard input/output.
2	conio.h	This is a console input/output header file.
3	string.h	It contains all string related library functions like gets(), puts(),etc.
4	stdlib.h	This header file contains all the general library functions like malloc(), calloc(), exit(), etc.
5	math.h	This header file contains all the math operations related functions like sqrt(), pow(), etc.
6	time.h	This header file contains all the time-related functions.
7	ctype.h	This header file contains all character handling functions.

8	stdarg.h	Variable argument functions are defined in this header file.
9	signal.h	All the signal handling functions are defined in this header file.
10	setjmp.h	This file contains all the jump functions.
11	locale.h	This file contains locale functions.
12	errno.h	This file contains error handling functions.
13	assert.h	This file contains diagnostics functions.

Designing structured programs:

Structured programming is a programming technique in which a larger program is divided into smaller subprograms to make it easy to understand, easy to implement and makes the code reusable, etc. Structured programming enables code reusability. **Code reusability** is a method of writing code once and using it many times. Using a structured programming technique, we write the code once and use it many times. Structured programming also makes the program easy to understand, improves the quality of the program, easy to implement and reduces time.

In C, the structured programming can be designed using **functions** concept. Using functions concept, we can divide the larger program into smaller subprograms and these subprograms are implemented individually. Every subprogram or function in C is executed individually.

Declaring a function:

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

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division is such that each function performs a specific task.

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

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

A function can also be referred as a method or a sub-routine or a procedure, etc.

Defining a Function

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

```
return_type function_name( parameter list ) {
   body of the function
}
```

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

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

Example

Given below is the source code for a function called **max()**. This function takes two parameters num1 and num2 and returns the maximum value between the two −

```
/* function returning the max between two numbers */
int max(int num1, int num2) {

   /* local variable declaration */
   int result;

   if (num1 > num2)
```

```
    result = num1;
  else
    result = num2;

  return result;
}
```

Function Declarations

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

A function declaration has the following parts −

```
return_type function_name( parameter list );
```

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

```
int max(int num1, int num2);
```

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

```
int max(int, int);
```

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

Calling a Function

While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.

When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.

To call a function, you simply need to pass the required parameters along with the function name, and if the function returns a value, then you can store the returned value. For example −

```
#include <stdio.h>

/* function declaration */
int max(int num1, int num2);

int main () {

  /* local variable definition */
```

```
   int a = 100;
   int b = 200;
   int ret;

   /* calling a function to get max value */
   ret = max(a, b);

   printf( "Max value is : %d\n", ret );

   return 0;
}

/* function returning the max between two numbers */
int max(int num1, int num2) {

   /* local variable declaration */
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

We have kept max() along with main() and compiled the source code. While running the final executable, it would produce the following result −

```
Max value is : 200
```

Function Arguments

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.

Formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways in which arguments can be passed to a function −

Sr.No.	Call Type & Description
1	Call by value This method copies the actual value of an argument into the formal parameter of the

	function. In this case, changes made to the parameter inside the function have no effect on the argument.
2	Call by reference This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

By default, C uses **call by value** to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function.

Signature of a function:

A **function signature** (or *type* signature, or *method* signature) defines input and output of functions or methods.
A signature can include:

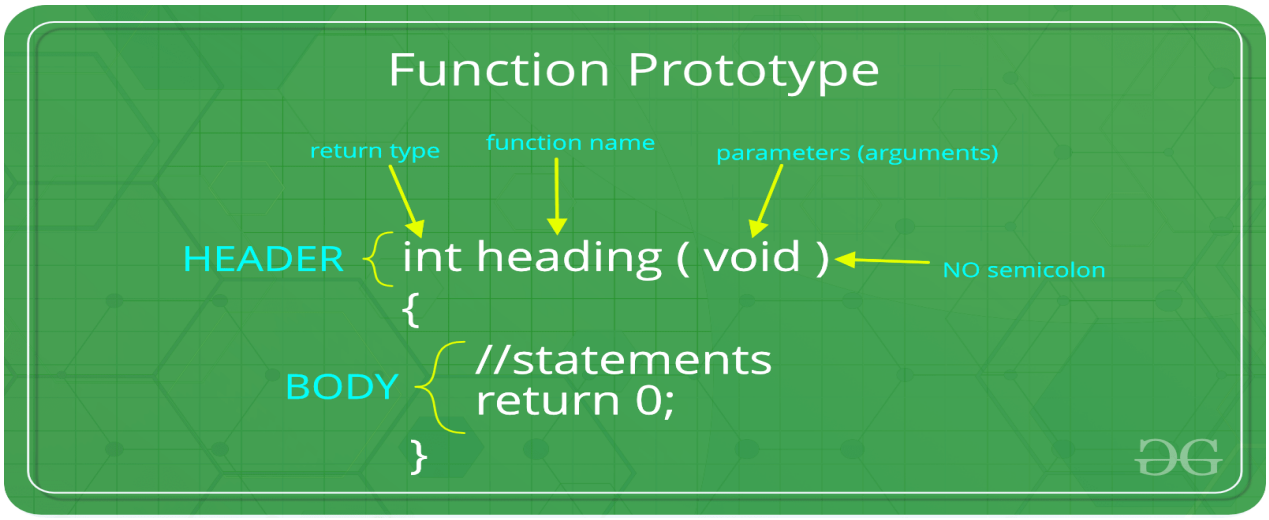
- parameters and their types
- a return value and type
- exceptions that might be thrown or passed back
- information about the availability of the method in an object-oriented program (such as the keywords public, static, or prototype).

- A function's **signature** includes the function's name and the number, order and type of its formal parameters.
- Two overloaded functions must not have the same signature.
- The return value is **not** part of a function's signature.
- These two functions have the same signature:

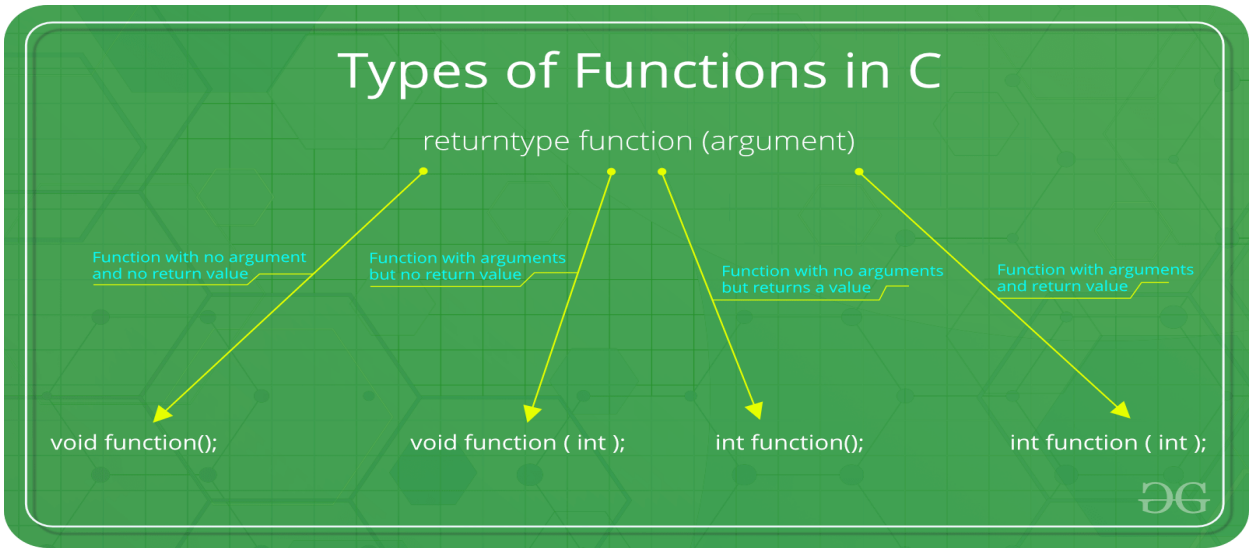
  - int Divide (int n, int m) ;
  - double Divide (int n, int m) ;

Parameters and return type of a function:

A function in C can be called either with arguments or without arguments. These function may or may not return values to the calling functions. All C functions can be called either with arguments or without arguments in a C program. Also, they may or may not return any values. Hence the function prototype of a function in C is as below:



There are following categories:



1. **Function with no argument and no return value :** When a function has no arguments, it does not receive any data from the calling function. Similarly when it does not return a value, the calling function does not receive any data from the called function.

Syntax :

Function declaration : void function();
Function call : function();
Function definition :

```
void function()
{
  statements;
}
```

// C code for function with no
// arguments and no return value

```
#include <stdio.h>
void value(void);
void main()
{
    value();
}
void value(void)
{
    int year = 1, period = 5, amount = 5000,
inrate = 0.12;
    float sum;
    sum = amount;
    while (year <= period) {
        sum = sum * (1 + inrate);
        year = year + 1;
    }
    printf(" The total amount is %f:", sum);
}
```

Output:

```
The total amount is 5000.000000
```

2. **Function with arguments but no return value :** When a function has arguments, it receive any data from the calling function but it returns no values.
   Syntax :

```
Function declaration : void function ( int );
Function call : function( x );
Function definition:
                    void function( int x )
                    {
                      statements;
                    }
```

```
// C code for function
// with argument but no return value

#include <stdio.h>

void function(int, int[], char[]);
int main()
{
    int a = 20;
    int ar[5] = { 10, 20, 30, 40, 50 };
    char str[30] = "geeksforgeeks";
    function(a, &ar[0], &str[0]);
    return 0;
}

void function(int a, int* ar, char* str)
{
    int i;
    printf("value of a is %d\n\n", a);
    for (i = 0; i < 5; i++) {
        printf("value of ar[%d] is %d\n", i, ar[i]);
    }
    printf("\nvalue of str is %s\n", str);
}
```

Output:

```
value of a is 20

value of ar[0] is 10

value of ar[1] is 20

value of ar[2] is 30

value of ar[3] is 40

value of ar[4] is 50
```

```
The given string is : geeksforgeeks
```

3. **Function with no arguments but returns a value :** There could be occasions where we may need to design functions that may not take any arguments but returns a value to the calling function. A example for this is getchar function it has no parameters but it returns an integer an integer type data that represents a character.

Syntax :

```
Function declaration : int function();
Function call : function();
Function definition :
        int function()
        {
            statements;
             return x;
         }
```

// C code for function with no arguments
// but have return value

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

int sum();
int main()
{
    int num;
    num = sum();
    printf("\nSum of two given values = %d",
num);
    return 0;
}

int sum()
{
    int a = 50, b = 80, sum;
    sum = sqrt(a) + sqrt(b);
    return sum;
}
```

Output:

```
Sum of two given values = 16
```

4. **Function with arguments and return value**

Syntax :

```
Function declaration : int function ( int );
Function call : function( x );
Function definition:
        int function( int x )
```

```
        {
          statements;
          return x;
        }
```

```
// C code for function with arguments
// and with return value

#include <stdio.h>
#include <string.h>
int function(int, int[]);
int main()
{
    int i, a = 20;
    int arr[5] = { 10, 20, 30, 40, 50 };
    a = function(a, &arr[0]);
    printf("value of a is %d\n", a);
    for (i = 0; i < 5; i++) {
        printf("value of arr[%d] is %d\n", i, arr[i]);
    }
    return 0;
}
int function(int a, int* arr)
{
    int i;
    a = a + 20;
    arr[0] = arr[0] + 50;
    arr[1] = arr[1] + 50;
    arr[2] = arr[2] + 50;
    arr[3] = arr[3] + 50;
    arr[4] = arr[4] + 50;
    return a;
}
```

Output:

```
value of a is 40

value of arr[0] is 60

value of arr[1] is 70

value of arr[2] is 80

value of arr[3] is 90

value of arr[4] is 100
```

Passing parameters to functions:

When a function is called, the calling function has to pass some values to the called functions.

There are two ways by which we can pass the parameters to the functions:

1. Call by value

- Here the values of the variables are passed by the calling function to the called function.
- If any value of the parameter in the called function has to be modified the change will be reflected only in the called function.
- This happens as all the changes are made on the copy of the variables and not on the actual ones.

Example: Call by value

```
#include <stdio.h>
int sum (int n);
void main()
{
    int a = 5;
    printf("\n The value of 'a' before the calling function is = %d", a);
    a = sum(a);
    printf("\n The value of 'a' after calling the function is = %d", a);
}
int sum (int n)
{
    n = n + 20;
    printf("\n Value of 'n' in the called function is = %d", n);
    return n;
}
```

Output:

The value of 'a' before the calling function is = 5

Value of 'n' in the called function is = 25

The value of 'a' after calling the function is = 25

2. Call by reference

- Here, the address of the variables are passed by the calling function to the called function.
- The address which is used inside the function is used to access the actual argument used in the call.
- If there are any changes made in the parameters, they affect the passed argument.
- For passing a value to the reference, the argument pointers are passed to the functions just like any other value.

Example: Call by reference

```
#include <stdio.h>
int sum (int *n);
void main()
{
    int a = 5;
    printf("\n The value of 'a' before the calling function is = %d", a);
    sum(&a);
    printf("\n The value of 'a' after calling the function is = %d", a);
}
int sum (int *n)
{
    *n = *n + 20;
    printf("\n value of 'n' in the called function is = %d", n);
}
```

Output:

The value of 'a' before the calling function is = 5

value of 'n' in the called function is = -1079041764

The value of 'a' after calling the function is = 25

Passing arrays to functions:

In C programming, you can pass en entire array to functions. Before we learn that, let's see how you can pass individual elements of an array to functions.

Passing individual array elements

Passing array elements to a function is similar to passing variables to a function.

Example 1: Passing an array

```
1.  #include <stdio.h>
2.  void display(int age1, int age2)
3.  {
4.     printf("%d\n", age1);
5.     printf("%d\n", age2);
6.  }
7.
8.  int main()
9.  {
10.    int ageArray[] = {2, 8, 4, 12};
11.
12.    // Passing second and third elements to display()
13.    display(ageArray[1], ageArray[2]);
14.    return 0;
15. }
```

Output

```
8


4
```

Example 2: Passing arrays to functions

```
// Program to calculate the sum of array elements by passing to a function

#include <stdio.h>
float calculateSum(float age[]);

int main() {
   float result, age[] = {23.4, 55, 22.6, 3, 40.5, 18};

   // age array is passed to calculateSum()
   result = calculateSum(age);
   printf("Result = %.2f", result);
   return 0;
}

float calculateSum(float age[]) {

 float sum = 0.0;

  for (int i = 0; i < 6; ++i) {
                    sum += age[i];
  }

  return sum;
}
```

Output

```
Result = 162.50
```

To pass an entire array to a function, only the name of the array is passed as an argument.

```
result =  calculateSum(age);
```

However, notice the use of [] in the function definition.

```
float calculateSum(float age[]) {
... ..
}
```

This informs the compiler that you are passing a one-dimensional array to the function.

Passing Multidimensional Arrays to a Function

Example 3: Passing two-dimensional arrays

```c
#include <stdio.h>
void displayNumbers(int num[2][2]);
int main()
{
    int num[2][2];
    printf("Enter 4 numbers:\n");
    for (int i = 0; i < 2; ++i)
        for (int j = 0; j < 2; ++j)
            scanf("%d", &num[i][j]);

    // passing multi-dimensional array to a function
    displayNumbers(num);
    return 0;
}

void displayNumbers(int num[2][2])
{
    printf("Displaying:\n");
    for (int i = 0; i < 2; ++i) {
        for (int j = 0; j < 2; ++j) {
            printf("%d\n", num[i][j]);
        }
    }
}
```

Output

```
Enter 4 numbers:
2
3
4
5
Displaying:
2
3
4
5
```

Passing pointers to functions:

C programming allows passing a pointer to a function. To do so, simply declare the function parameter as a pointer type.

Following is a simple example where we pass an unsigned long pointer to a function and change the value inside the function which reflects back in the calling function −

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

void getSeconds(unsigned long *par);

int main () {

   unsigned long sec;
   getSeconds( &sec );

   /* print the actual value */
   printf("Number of seconds: %ld\n", sec );

   return 0;
}

void getSeconds(unsigned long *par) {
   /* get the current number of seconds */
   *par = time( NULL );
   return;
}
```

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

Number of seconds :1294450468

The function, which can accept a pointer, can also accept an array as shown in the following example −

```
#include <stdio.h>

/* function declaration */
double getAverage(int *arr, int size);

int main () {

   /* an int array with 5 elements */
   int balance[5] = {1000, 2, 3, 17, 50};
   double avg;
```

```
   /* pass pointer to the array as an argument */
   avg = getAverage( balance, 5 ) ;

   /* output the returned value  */
   printf("Average value is: %f\n", avg );
   return 0;
}

double getAverage(int *arr, int size) {

   int  i, sum = 0;
   double avg;

   for (i = 0; i < size; ++i) {
      sum += arr[i];
   }

   avg = (double)sum / size;
   return avg;
}
```

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

Average value is: 214.40000

Just like any other argument, pointers can also be passed to a function as an argument. Lets take an example to understand how this is done.

Example: Passing Pointer to a Function in C Programming

In this example, we are passing a pointer to a function. When we pass a pointer as an argument instead of a variable then the address of the variable is passed instead of the value. So any change made by the function using the pointer is permanently made at the address of passed variable. This technique is known as call by reference in C.

Try this same program without pointer, you would find that the bonus amount will not reflect in the salary, this is because the change made by the function would be done to the local variables of the function. When we use pointers, the value is changed at the address of variable

```
#include <stdio.h>
void salaryhike(int  *var, int b)
{
   *var = *var+b;
```

```
}
int main()
{
   int salary=0, bonus=0;
   printf("Enter the employee current salary:");
   scanf("%d", &salary);
   printf("Enter bonus:");
   scanf("%d", &bonus);
   salaryhike(&salary, bonus);
   printf("Final salary: %d", salary);
   return 0;
}
```

Output:

```
Enter the employee current salary:10000
Enter bonus:2000
Final salary: 12000
```

Example 2: Swapping two numbers using Pointers

This is one of the most popular example that shows how to swap numbers using call by reference.

Try this program without pointers, you would see that the numbers are not swapped. The reason is same that we have seen above in the first example.

```
#include <stdio.h>
void swapnum(int *num1, int *num2)
{
  int tempnum;

  tempnum = *num1;
  *num1 = *num2;
  *num2 = tempnum;
}
int main( )
{
  int v1 = 11, v2 = 77 ;
  printf("Before swapping:");
  printf("\nValue of v1 is: %d", v1);
  printf("\nValue of v2 is: %d", v2);

  /*calling swap function*/
  swapnum( &v1, &v2 );

  printf("\nAfter swapping:");
```

```
   printf("\nValue of v1 is: %d", v1);
   printf("\nValue of v2 is: %d", v2);
}
```

Output:

```
Before swapping:
Value of v1 is: 11
Value of v2 is: 77
After swapping:
Value of v1 is: 77
Value of v2 is: 11
```

Idea of call by reference:

The **call by reference** method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. It means the changes made to the parameter affect the passed argument.

To pass a value by reference, argument pointers are passed to the functions just like any other value. So accordingly you need to declare the function parameters as pointer types as in the following function **swap()**, which exchanges the values of the two integer variables pointed to, by their arguments.

```
/* function definition to swap the values */
void swap(int *x, int *y) {

   int temp;
   temp = *x;    /* save the value at address x */
   *x = *y;      /* put y into x */
   *y = temp;    /* put temp into y */

   return;
}
```

Let us now call the function **swap()** by passing values by reference as in the following example –

```
#include <stdio.h>

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

int main () {
```

```
   /* local variable definition */
   int a = 100;
   int b = 200;

   printf("Before swap, value of a : %d\n", a );
   printf("Before swap, value of b : %d\n", b );

   /* calling a function to swap the values.
      * &a indicates pointer to a ie. address of variable a and
      * &b indicates pointer to b ie. address of variable b.
   */
   swap(&a, &b);

   printf("After swap, value of a : %d\n", a );
   printf("After swap, value of b : %d\n", b );

   return 0;
}
```

Let us put the above code in a single C file, compile and execute it, to produce the following result −

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :200
After swap, value of b :100
```

It shows that the change has reflected outside the function as well, unlike call by value where the changes do not reflect outside the function.

Some C standard functions and libraries:

Standard Library Functions in C – Use it in Smart Way & Stand Alone in Crowd

Have you ever wondered why Library Functions in C possess great importance in programming? Clear your confusion, because we are going to justify the importance of Library Functions in C through this tutorial and will cover all the important aspects related to it. These concepts will help you in a tremendous way to enhance your programming skills.

In this tutorial, we will discuss:

In order to acknowledge the potency of standard library functions, let us consider a situation where you want to simply display a statement without the use of standard output statements, like **printf() or puts()** or any other inbuilt display functions.
That would probably be a tedious task and an extensive knowledge of programming at the engineering level would be required to simply display a statement as your program output.

This is where the standard library functions in C come into play!

Before we move on it is required to be well acquainted with the skills of Functions in C

1. Standard Library Functions in C

Standard Library Functions are basically the inbuilt functions in the C compiler that makes things easy for the programmer.
As we have already discussed, every C program has at least one function, that is, the **main() function**. The main() function is also a standard library function in C since it is inbuilt and conveys a specific meaning to the C compiler.

2. Significance of Standard Library Functions in C

2.1 Usability

Standard library functions allow the programmer to use the pre-existing codes available in the C compiler without the need for the user to define his own code by deriving the logic to perform certain basic functions.

2.2 Flexibility

A wide variety of programs can be made by the programmer by making slight modifications while using the standard library functions in C.

2.3 User-friendly syntax

We have already discussed in Function in C tutorial that how easy it is to grasp and use the syntax of functions.

2.4 Optimization and Reliability

All the standard library functions in C have been tested multiple times in order to generate the optimal output with maximum efficiency making it reliable to use.

2.5 Time-saving

Instead of writing numerous lines of codes, these functions help the programmer to save time by simply using the pre-existing functions.

2.6 Portability

Standard library functions are available in the C compiler irrespective of the device you are working on. These functions connote the same meaning and hence serve the same purpose regardless of the operating system or programming environment.

3. Header Files in C

In order to access the standard library functions in C, certain header files need to be included before writing the body of the program.

Don't move further, if you are not familiar with the Header Files in C.
Here is a tabular representation of a list of header files associated with some of the standard library functions in C:

HEADER FILE	MEANING	ELUCIDATION
<stdio.h>	Standard input-output header	Used to perform input and output operations like scanf() and printf().
<string.h>	String header	Used to perform string manipulation operations like strlen and strcpy.

<conio.h>	Console input-output header	Used to perform console input and console output operations like clrscr() to clear the screen and getch() to get the character from the keyboard.
<stdlib.h>	Standard library header	Used to perform standard utility functions like dynamic memory allocation using functions such as malloc() and calloc().
<math.h>	Math header	Used to perform mathematical operations like sqrt() and pow() to obtain the square root and the power of a number respectively.
<ctype.h>	Character type header	Used to perform character type functions like isaplha() and isdigit() to find whether the given character is an alphabet or a digit. respectively.
<time.h>	Time header	Used to perform functions related to date and time like setdate() and getdate() to modify the system date and get the CPU time respectively.
<assert.h>	Assertion header	Used in program assertion functions like assert() to get an integer data type as a parameter which prints stderr only if the parameter passed is 0.
<locale.h>	Localization header	Used to perform localization functions like setlocale() and localeconv() to set locale and get locale conventions respectively.
<signal.h>	Signal header	Used to perform signal handling functions like signal() and raise() to install signal handler and to raise the signal in the program respectively.
<setjmp.h>	Jump header	Used to perform jump functions.
<stdarg.h>	Standard argument header	Used to perform standard argument functions like va_start and va_arg() to indicate the start of the variable-length argument list and to fetch the arguments from the variable-length argument list in the program respectively.

<errno.h>	Error handling header	Used to perform error handling operations like errno() to indicate errors in the program by initially assigning the value of this function to 0 and then later changing it to indicate errors.

Get a complete guide to learn Data types in C

Let us discuss some of the commonly used Standard library functions in C in detail:

3.1 <stdio.h>

This is the basic header file used in almost every program written in the C language.
It stands for standard input and standard output used to perform input-output functions, some of which are:

- **printf()**– Used to display output on the screen.
- **scanf()**– To take input from the user.
- **getchar()**– To return characters on the screen.
- **putchar()**– To display output as a single character on the screen.
- **fgets()**– To take a line as an input.
- **puts()**– To display a line as an output.
- **fopen()**– To open a file.
- **fclose()**– To close a file.

Here is a simple program in C that illustrates the use of <stdio.h> to use scanf() and printf() functions:

```
1.  #include<stdio.h> // Use of stdio.h header
2.  int main()
3.  {
4.
5.  char name[30];
6.  char line[30];
7.
8.  printf("Enter the name: "); // Use of printf() function
9.  scanf("%s", name); // Use of scanf() function
10. printf("The name is: %s\n", name);
11. return 0;
12. }
```

3.2 <string.h>

We have already discussed the various ***string manipulation functions in C*** in detail.

3.3 <stdlib.h>

Functions such as malloc(), calloc(), realloc() and free() can be used while dealing with dynamic memory allocation of variables.

It should be clear that these functions are used for dynamic memory allocation of variables, that is in contrast to arrays that allocate memory in a static (fixed) manner.

3.3.1 malloc()

malloc() stands for *memory allocation*. This function is responsible for reserving a specific block of memory and returns a null pointer during the execution of the program.

Syntax-

```
pointer_name = (cast_type * ) malloc (no_of_bytes * size_in_bytes_of_cast_type)
```

For instance,

```
pointer = ( float* ) malloc ( 100 * sizeof ( float ) );
```

```
#include <stdio.h>
#include <stdlib.h> // Use of stdlib header
int main()
{

printf("Welcome to DataFlair tutorials!\n\n");
int no_of_elements, iteration, *pointer, sum = 0;

printf("Enter number of elements: ");
scanf("%d", &no_of_elements);

pointer = (int*) malloc(no_of_elements * sizeof(int)); // Use of malloc() function

if(pointer == NULL)
{
printf("Sorry! Memory is not allocated.");
exit(0);
}
printf("Enter the elements: ");

/* Implementation of dynamic mememory allocation */

for(iteration = 0; iteration < no_of_elements; iteration++)
{
scanf("%d", pointer + iteration);
sum += *(pointer + iteration);
}

printf("The sum of the elements is = %d\n", sum);
return 0;
}
```

3.3.2 calloc()

calloc stands for contiguous allocation. It is similar to malloc in all respects except the fact that it initializes the memory to 0 and has the ability to allocate numerous blocks of memory before the execution of the program.

Syntax-

```
pointer_name = (cast_type*) calloc (no_of_bytes, size_of_cast_type);
```

For instance,

```
pointer = ( int *) calloc (50, sizeof ( int ) );
```

program:

```
#include <stdio.h>
#include <stdlib.h> // Use of stdlib header
int main()
{
printf("Welcome to DataFlair tutorials!\n\n");
int no_of_elements, iteration, *pointer, sum = 0;
printf("Enter number of elements: ");
scanf("%d", &no_of_elements);
pointer = (int*) calloc(no_of_elements, sizeof(int));
if(pointer == NULL)
{
printf("Sorry! Memory is not allocated.");
exit(0);
}
printf("Enter the elements: ");
/* Implementation of dynamic mememory allocation */
for(iteration = 0; iteration < no_of_elements; iteration++)
{
scanf("%d", pointer + iteration);
sum += *(pointer + iteration);
}
printf("The sum of the elements is = %d\n", sum);
return 0;
```

3.3.3 realloc()

realloc stands for reallocation. It is used to change the size of the previously allocated memory in case the previously allocated memory is insufficient to meet the required needs of the ***variable in C***.

Syntax-

```
1. pointer_name = realloc(pointer_name, new_size);
```

For instance,

```
1. If initially,
2. pointer = ( int* ) malloc( 20 * sizeof( int ) );
```

Then, using realloc

```
1. pointer = realloc(pointer, 24 * sizeof( int ) );
```

program:

```
1. #include <stdio.h>
2. #include <stdlib.h> // Use of stdlib header
3. int main()
4. {
5. printf("Welcome to DataFlair tutorials!\n\n");
6. int *pointer,*new_pointer, iteration;
7. pointer = (int *)malloc(2*sizeof(int));
8. *pointer = 5;
9. *(pointer + 1) = 10;
10. new_pointer = (int *)realloc(pointer, 3*sizeof(int));
11. *(new_pointer + 2) = 15;
12. printf("The elements are: ");
13. for(iteration = 0; iteration < 3; iteration++)
14. printf("%d\n ", *(new_pointer + iteration));
15. return 0;
16. }
```

3.3.4 free()

free is responsible to free the dynamically allocated memory done by malloc(), calloc() or realloc() to the system.

Syntax-

```
1. free( pointer_name );
```

For instance,

```
1. free( pointer );
```

Program:

```
1. #include <stdio.h>
2. #include <stdlib.h> // Use of stdlib header
3. int main()
4. {
```

```
printf("Welcome to DataFlair tutorials!\n\n");
int no_of_elements, iteration, *pointer, sum = 0;
printf("Enter number of elements: ");
scanf("%d", &no_of_elements);
pointer = (int*) malloc(no_of_elements * sizeof(int)); // Use of malloc() function
if(pointer == NULL)
{
printf("Sorry! Memory is not allocated.");
exit(0);
}
printf("Enter the elements: ");
/* Implementation of dynamic memmory allocation */
for(iteration = 0; iteration < no_of_elements; iteration++)
{
scanf("%d", pointer + iteration);
sum += *(pointer + iteration);
}
printf("The sum of the elements is = %d\n", sum);
free(pointer); // Use of free() function
return 0;
```

3.4 <math.h>

The math header is of great significance as it is used to perform various mathematical operations such as:

- **sqrt()** – This function is used to find the square root of a number
- **pow()** – This function is used to find the power raised to that number.
- **fabs()** – This function is used to find the absolute value of a number.
- **log()** – This function is used to find the logarithm of a number.
- **sin()** – This function is used to find the sine value of a number.
- **cos()** – This function is used to find the cosine value of a number.
- **tan()** – This function is used to find the tangent value of a number.

```
Int main()
{
printf("Welcome to DataFlair tutorials!\n\n");
double number=5, square_root;
int base = 6, power = 3, power_result;
int integer = -7, integer_result;
square_root = sqrt(number);
printf("The square root of %lf is: %lf\n", number, square_root);
power_result = pow(base,power);
printf("%d raised to the power %d is: %d\n", base, power, power_result);
```

```
10. integer_result = fabs(integer);
11. printf("The absolute value of %d is: %d\n", integer, integer_result);
12. return 0;
13. }
```

3.5 <ctype.h>

This function is popularly used when it comes to character handling.

Some of the functions associated with <ctype.h> are:

- **isalpha()** – Used to check if the character is an alphabet or not.
- **isdigit()** – Used to check if the character is a digit or not.
- **isalnum()** – Used to check if the character is alphanumeric or not.
- **isupper()** – Used to check if the character is in uppercase or not
- **islower()** – Used to check if the character is in lowercase or not.
- **toupper()** – Used to convert the character into uppercase.
- **tolower()** – Used to convert the character into lowercase.
- **iscntrl()** – Used to check if the character is a control character or not.
- **isgraph()** – Used to check if the character is a graphic character or not.
- **isprint()** – Used to check if the character is a printable character or not
- **ispunct()** – Used to check if the character is a punctuation mark or not.
- **isspace() –** Used to check if the character is a white-space character or

3.6 <conio.h>

It is used to perform console input and console output operations like **clrscr()** to clear the screen and **getch()** to get the character from the keyboard.

https://data-flair.training/blogs/standard-library-functions-in-c/

Recursion: such as simple program, finding factorial, Fibonacci series, etc, Limitations of recursive functions:

Recursion:

Recursion is the process which comes into existence when a function calls a copy of itself to work on a smaller problem. Any function which calls itself is called recursive function, and such function calls are called recursive calls. Recursion involves several numbers of recursive calls. However, it is important to impose a termination condition of recursion. Recursion code is shorter than iterative code however it is difficult to understand.

Recursion cannot be applied to all the problem, but it is more useful for the tasks that can be defined in terms of similar subtasks. For Example, recursion may be applied to sorting, searching, and traversal problems.

Generally, iterative solutions are more efficient than recursion since function call is always overhead. Any problem that can be solved recursively, can also be solved iteratively. However, some problems are best suited to be solved by the recursion, for example, tower of Hanoi, Fibonacci series, factorial finding, etc.

In the following example, recursion is used to calculate the factorial of a number.

```
1. #include <stdio.h>
2. int fact (int);
3. int main()
4. {
5.     int n,f;
6.     printf("Enter the number whose factorial you want to calculate?");
7.     scanf("%d",&n);
8.     f = fact(n);
9.     printf("factorial = %d",f);
10. }
11. int fact(int n)
12. {
13.     if (n==0)
14.     {
15.         return 0;
16.     }
17.     else if ( n == 1)
```

```
18.    {
19.        return 1;
20.    }
21.    else
22.    {
23.        return n*fact(n-1);
24.    }
25. }
```

Output

```
Enter the number whose factorial you want to calculate?5
factorial = 120
```

We can understand the above program of the recursive method call by the figure given below:

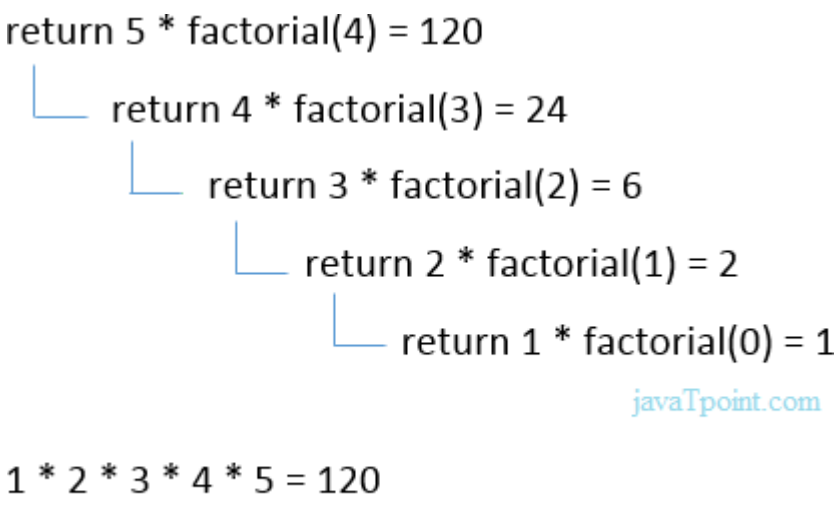


Fig: Recursion

Recursive Function

A recursive function performs the tasks by dividing it into the subtasks. There is a termination condition defined in the function which is satisfied by some specific subtask. After this, the recursion stops and the final result is returned from the function.

The case at which the function doesn't recur is called the base case whereas the instances where the function keeps calling itself to perform a subtask, is called the recursive case. All the recursive functions can be written using this format.

Pseudocode for writing any recursive function is given below.

```
1. if (test_for_base)
2. {
3.     return some_value;
4. }
5. else if (test_for_another_base)
6. {
7.     return some_another_value;
8. }
9. else
10. {
11.     // Statements;
12.     recursive call;
13. }
```

Example of recursion in C

Let's see an example to find the nth term of the Fibonacci series.

```
1. #include<stdio.h>
2. int fibonacci(int);
3. void main ()
4. {
5.     int n,f;
6.     printf("Enter the value of n?");
7.     scanf("%d",&n);
8.     f = fibonacci(n);
9.     printf("%d",f);
10. }
11. int fibonacci (int n)
12. {
```

```
13.    if (n==0)
14.    {
15.    return 0;
16.    }
17.    else if (n == 1)
18.    {
19.        return 1;
20.    }
21.    else
22.    {
23.        return  fibonacci(n-1)+fibonacci(n-2);
24.    }
25. }
```

Output

```
Enter the value of n?12
144
```

Memory allocation of Recursive method

Each recursive call creates a new copy of that method in the memory. Once some data is returned by the method, the copy is removed from the memory. Since all the variables and other stuff declared inside function get stored in the stack, therefore a separate stack is maintained at each recursive call. Once the value is returned from the corresponding function, the stack gets destroyed. Recursion involves so much complexity in resolving and tracking the values at each recursive call. Therefore we need to maintain the stack and track the values of the variables defined in the stack.

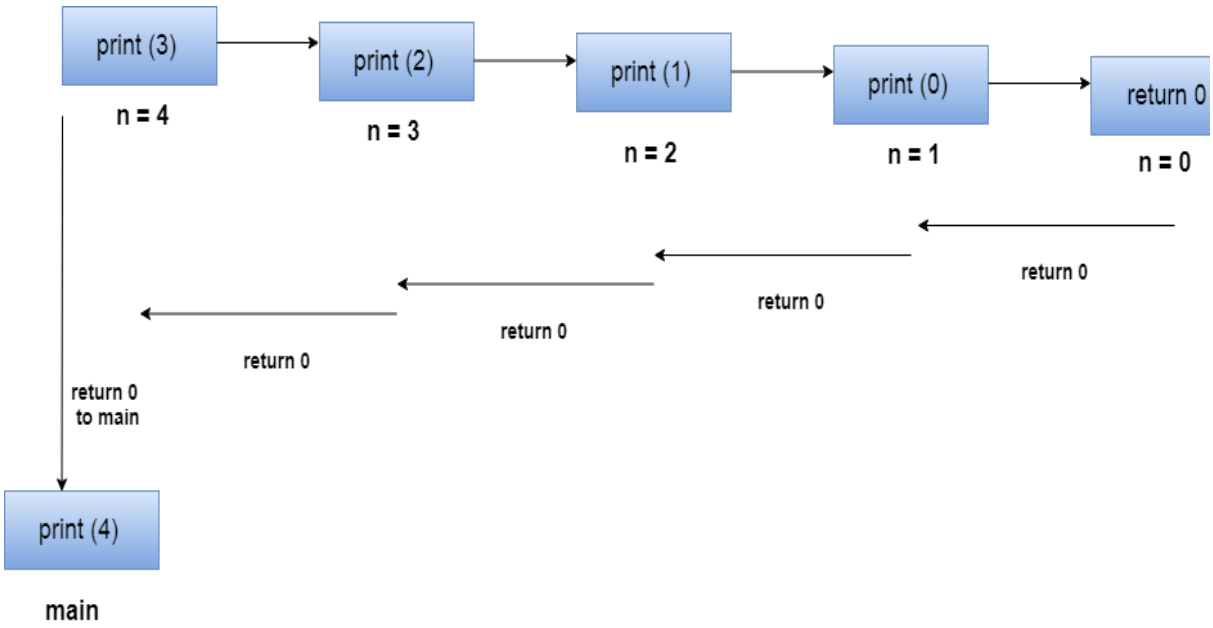
Let us consider the following example to understand the memory allocation of the recursive functions.

```
1.  int display (int n)
2.  {
3.      if(n == 0)
4.          return 0; // terminating condition
```

```
5.      else
6.      {
7.          printf("%d",n);
8.          return display(n-1); // recursive call
9.      }
10. }
```

Explanation

Let us examine this recursive function for n = 4. First, all the stacks are maintained which prints the corresponding value of n until n becomes 0, Once the termination condition is reached, the stacks get destroyed one by one by returning 0 to its calling stack. Consider the following image for more information regarding the stack trace for the recursive functions.



Stack tracing for recursive function call

Limitations of recursive function:

When a function calls itself from its body is called Recursion.

Advantages

- Reduce unnecessary calling of function.
- Through Recursion one can Solve problems in easy way while its iterative solution is very big and complex.

Disdvantages

- Recursive solution is always logical and it is very difficult to trace.(debug and understand).
- In recursive 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.
- Recursion takes a lot of stack space, usually not considerable when the program is small and running on a PC.
- Recursion uses more processor time.

```
#include<stdio.h>

long Factorial(int);

void main()

{

int num;

long fact;

printf("\n\tEnter any positive number : ");

scanf("%d",&num);

fact = Factorial(num);

printf("\n\tThe Factorial of %d is %ld",num,fact);

}

long Factorial(int num)

{

if (num == 1)

        return 1;
```

```
        else
                return num*Factorial(num-1);
         }
```

Output :

```
        Enter any positive number :
        The Factorial of 10 is 3628800
```

DISADVANTAGES OF RECURSION:

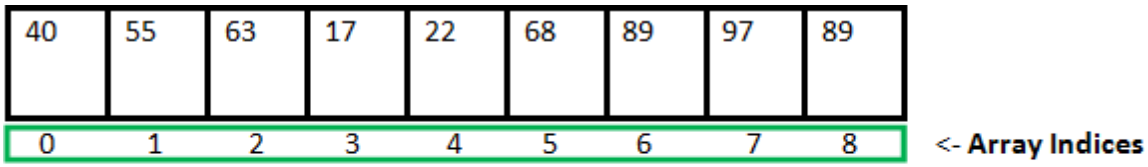
Fairly slower than its iterative solution.
For each step we make a **recursive** call to a **function**. ...
May cause stack-overflow if the **recursion** goes too deep to solve the problem.
Difficult to debug and trace the values with each step of **recursion**.

Dynamic memory allocation: Allocating and freeing memory, Allocating memory for arrays of different data types.

Dynamic memory allocation:

Dynamic Memory Allocation in C using malloc(), calloc(), free() and realloc()

Since C is a structured language, it has some fixed rules for programming. One of it includes changing the size of an array. An array is collection of items stored at continuous memory locations.



Array Length = 9
First Index = 0
Last Index = 8

As it can be seen that the length (size) of the array above made is 9. But what if there is a requirement to change this length (size). For Example,

- If there is a situation where only 5 elements are needed to be entered in this array. In this case, the remaining 4 indices are just wasting memory in this array. So there is a requirement to lessen the length (size) of the array from 9 to 5.
- Take another situation. In this, there is an array of 9 elements with all 9 indices filled. But there is a need to enter 3 more elements in this array. In this case 3 indices more are required. So the length (size) of the array needs to be changed from 9 to 12.

This procedure is referred to as **Dynamic Memory Allocation in C**.
Therefore, C **Dynamic Memory Allocation** can be defined as a procedure in which the size of a data structure (like Array) is changed during the runtime.
C provides some functions to achieve these tasks. There are 4 library functions provided by C defined under **<stdlib.h>** header file to facilitate dynamic memory allocation in C programming. They are:

1. malloc()
2. calloc()
3. free()
4. realloc()

Let's look at each of them in greater detail.

1. C malloc() method

"malloc" or **"memory allocation"** method in C is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form.

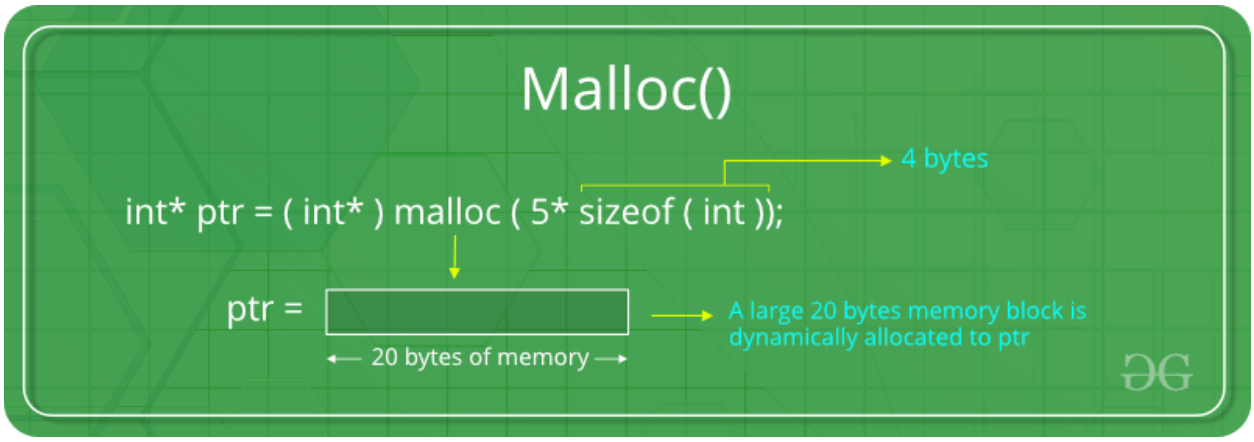
Syntax:

```
ptr = (cast-type*) malloc(byte-size)
```

For Example:

ptr = (int*) malloc(100 * sizeof(int));

Since the size of int is 4 bytes, this statement will allocate 400 bytes of memory. And, the pointer ptr holds the address of the first byte in the allocated memory.



1.
2. If space is insufficient, allocation fails and returns a NULL pointer.

Example:

```
#include  <stdio.h>

#include <stdlib.h>

int main()

{

   // This pointer will hold the

   // base address of the block created

   int* ptr;

   int n, i, sum = 0;

    // Get the number of elements for the array

   n = 5;

   printf("Enter number of elements: %d\n", n);

    // Dynamically allocate memory using malloc()

   ptr = (int*)malloc(n * sizeof(int));
```

```
    // Check if the memory has been successfully
    // allocated by malloc or not
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {
          // Memory has been successfully allocated
        printf("Memory successfully allocated using malloc.\n");
          // Get the elements of the array
        for (i = 0; i < n; ++i) {
            ptr[i] = i + 1;
        }
          // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }
    }
      return 0;
}
```

Output:

```
Enter number of elements: 5
Memory successfully allocated using malloc.
The elements of the array are: 1, 2, 3, 4, 5,
```

3. C calloc() method

“calloc” or **“contiguous allocation”** method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. It initializes each block with a default value ‘0’.

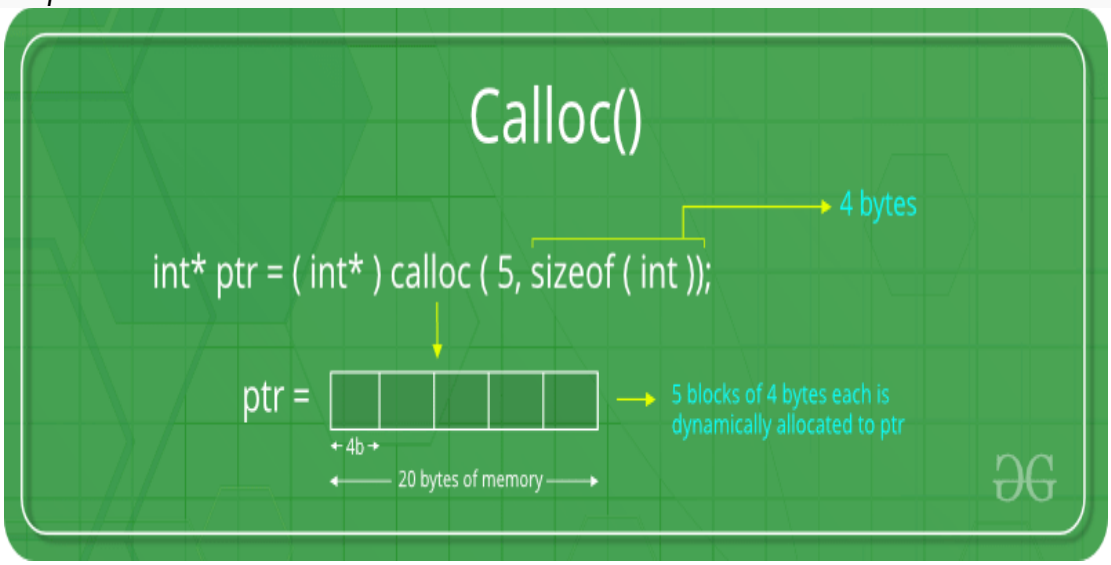
Syntax:

```
ptr = (cast-type*)calloc(n, element-size);
```

For Example:

ptr = (float*) calloc(25, sizeof(float));

This statement allocates contiguous space in memory for 25 elements each with the size of the float.



1. If space is insufficient, allocation fails and returns a NULL pointer.

Example:

```
#include  <stdio.h>

#include <stdlib.h>

int main()

{

    // This pointer will hold the

  // base address of the block created

  int* ptr;
```

```
int n, i, sum = 0;
  // Get the number of elements for the array
n = 5;
printf("Enter number of elements: %d\n", n);
  // Dynamically allocate memory using calloc()
ptr = (int*)calloc(n, sizeof(int));
  // Check if the memory has been successfully
// allocated by malloc or not
if (ptr == NULL) {
    printf("Memory not allocated.\n");
    exit(0);
}
else {
      // Memory has been successfully allocated
    printf("Memory successfully allocated using calloc.\n");
      // Get the elements of the array
    for (i = 0; i < n; ++i) {
        ptr[i] = i + 1;
    }
      // Print the elements of the array
    printf("The elements of the array are: ");
    for (i = 0; i < n; ++i) {
        printf("%d, ", ptr[i]);
    }
}
```

```
    return 0;
}
```

Output:

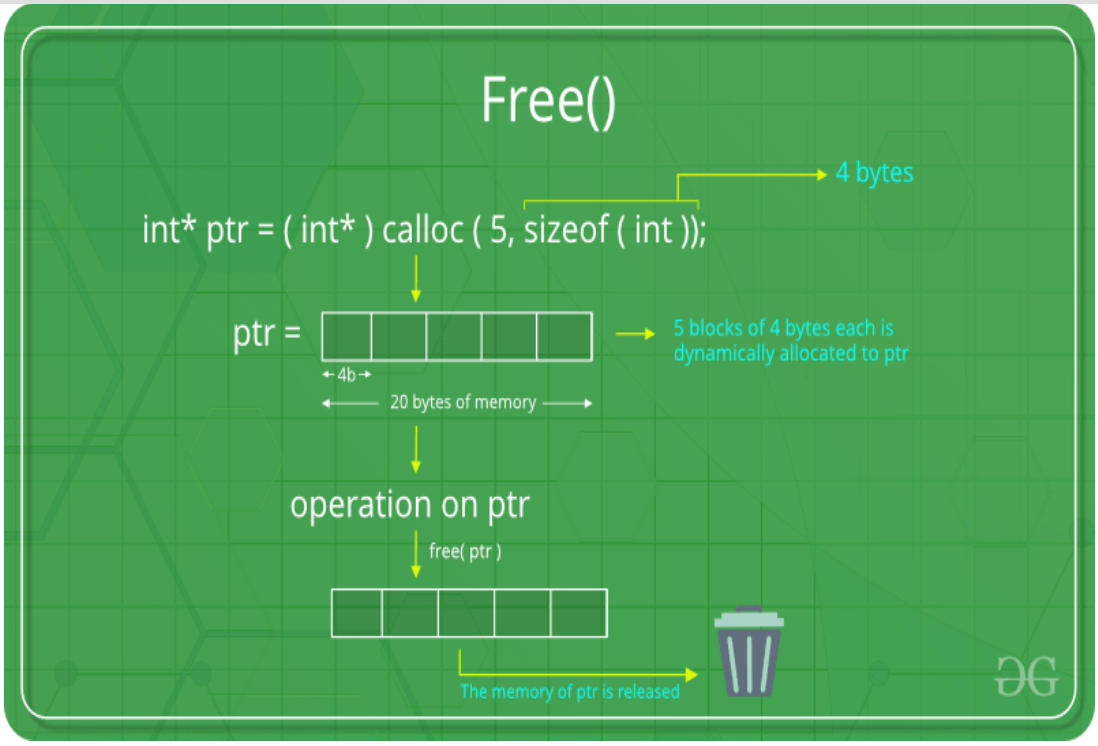
```
Enter number of elements: 5
Memory successfully allocated using calloc.
The elements of the array are: 1, 2, 3, 4, 5,
```

2. C free() method

"free" method in C is used to dynamically **de-allocate** the memory. The memory allocated using functions malloc() and calloc() is not de-allocated on their own. Hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.

Syntax:

```
free(ptr);
```



Example:

```
#include <stdio.h>

#include <stdlib.h>
```

```
 int main()
{
    // This pointer will hold the
   // base address of the block created
   int *ptr, *ptr1;
   int n, i, sum = 0;
    // Get the number of elements for the array
   n = 5;
   printf("Enter number of elements: %d\n", n);
    // Dynamically allocate memory using malloc()
   ptr = (int*)malloc(n * sizeof(int));

   // Dynamically allocate memory using calloc()
   ptr1 = (int*)calloc(n, sizeof(int));
    // Check if the memory has been successfully
   // allocated by malloc or not
   if (ptr == NULL || ptr1 == NULL) {
      printf("Memory not allocated.\n");
      exit(0);
   }
   else {
       // Memory has been successfully allocated
      printf("Memory successfully allocated using malloc.\n");
        // Free the memory
      free(ptr);
```

```
        printf("Malloc Memory successfully freed.\n");
          // Memory has been successfully allocated
        printf("\nMemory successfully allocated using calloc.\n");
          // Free the memory
        free(ptr1);
        printf("Calloc Memory successfully freed.\n");
    }
     return 0;
}
```

Output:

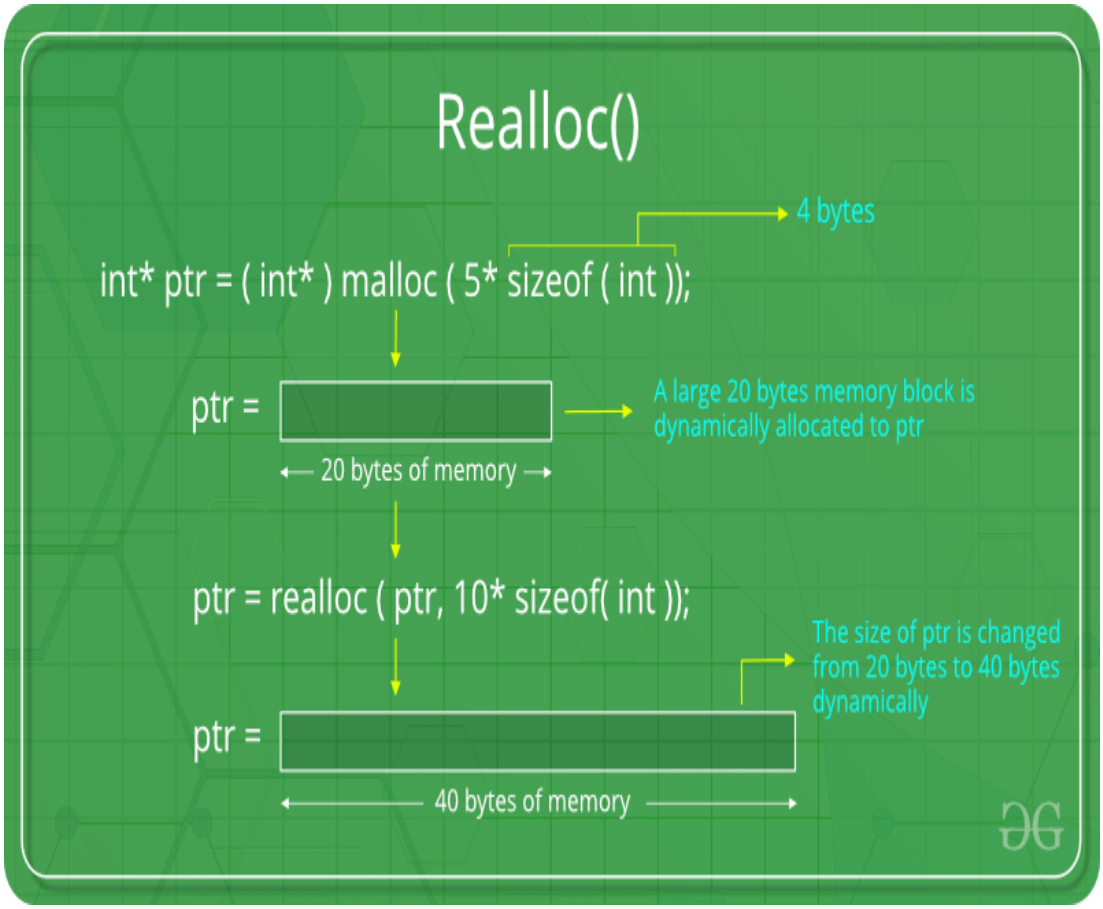
```
Enter number of elements: 5
Memory successfully allocated using malloc.
Malloc Memory successfully freed.


Memory successfully allocated using calloc.
Calloc Memory successfully freed.
```

3. C realloc() method

“realloc” or **“re-allocation”** method in C is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to **dynamically re-allocate memory**.

Syntax:

```
ptr = realloc(ptr, newSize);


where ptr is reallocated with new size 'newSize'.
```



If space is insufficient, allocation fails and returns a NULL pointer.

Example:

```
#include  <stdio.h>

#include <stdlib.h>

int main()

{

    // This pointer will hold the

   // base address of the block created

   int* ptr;

   int n, i, sum = 0;

    // Get the number of elements for the array

   n = 5;
```

```
printf("Enter number of elements: %d\n", n);
  // Dynamically allocate memory using calloc()
ptr = (int*)calloc(n, sizeof(int));
  // Check if the memory has been successfully
// allocated by malloc or not
if (ptr == NULL) {
    printf("Memory not allocated.\n");
    exit(0);
}
else {
      // Memory has been successfully allocated
    printf("Memory successfully allocated using calloc.\n");
      // Get the elements of the array
    for (i = 0; i < n; ++i) {
        ptr[i] = i + 1;
    }
      // Print the elements of the array
    printf("The elements of the array are: ");
    for (i = 0; i < n; ++i) {
        printf("%d, ", ptr[i]);
    }
      // Get the new size for the array
    n = 10;
    printf("\n\nEnter the new size of the array: %d\n", n);
      // Dynamically re-allocate memory using realloc()
```

```
        ptr = realloc(ptr, n * sizeof(int));
          // Memory has been successfully allocated
        printf("Memory successfully re-allocated using realloc.\n");
          // Get the new elements of the array
        for (i = 5; i < n; ++i) {
            ptr[i] = i + 1;
        }
          // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }
          free(ptr);
    }
     return 0;
}
```

Output:

```
Enter number of elements: 5
Memory successfully allocated using calloc.
The elements of the array are: 1, 2, 3, 4, 5,

Enter the new size of the array: 10
Memory successfully re-allocated using realloc.
The elements of the array are: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
```

https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/

One:1

Consider the following program, where are i, j and k are stored in memory?

```
int i;
int main()
{
   int j;
   int *k = (int *) malloc (sizeof(int));
}
```

(A) i, j and *k are stored in stack segment
(B) i and j are stored in stack segment. *k is stored on heap.
(C) i is stored in BSS part of data segment, j is stored in stack segment. *k is stored on heap.
(D) j is stored in BSS part of data segment, i is stored in stack segment. *k is stored on heap.

Answer: (C)
Explanation: i is global variable and it is uninitialized so it is stored on BSS part of Data Segment
j is local in main() so it is stored in stack frame

*k is dynamically allocated so it is stored on Heap Segment.

What is the problem with following code?

```
#include<stdio.h>
int main()
{
   int *p = (int *)malloc(sizeof(int));

   p = NULL;

   free(p);
}
```

(A) Compiler Error: free can't be applied on NULL pointer
(B) Memory Leak
(C) Dangling Pointer
(D) The program may crash as free() is called for NULL pointer.

Answer: (B)

Explanation: free() can be called for NULL pointer, so no problem with free function call. The problem is memory leak, p is allocated some memory which is not freed, but the pointer is assigned as NULL. The correct sequence should be following:

```
free(p);
p = NULL;
```

Which of the following is true?
(A) "ptr = calloc(m, n)" is equivalent to following
ptr = malloc(m * n);

(B) "ptr = calloc(m, n)" is equivalent to following
ptr = malloc(m * n);
memset(ptr, 0, m * n);

(C) "ptr = calloc(m, n)" is equivalent to following
ptr = malloc(m);
memset(ptr, 0, m);

(D) "ptr = calloc(m, n)" is equivalent to following
ptr = malloc(n);
memset(ptr, 0, n);

Answer: (B)

Explanation: See calloc() versus malloc() for details.

The most appropriate matching for the following pairs (GATE CS 2000)

X: m=malloc(5); m= NULL; 1: using dangling pointers
Y: free(n); n->value=5; 2: using uninitialized pointers
Z: char *p; *p = 'a'; 3. lost memory is:
(A) X—1 Y—3 Z-2
(B) (X—2 Y—1 Z-3
(C) X—3 Y—2 Z-1
(D) X—3 Y—1 Z-2

Answer: (D)

Explanation: X -> A pointer is assigned to NULL without freeing memory so a clear example of memory leak
Y -> Trying to retrieve value after freeing it so dangling pointer.
Z -> Using uninitialized pointers

What is the return type of malloc() or calloc()
(A) void *
(B) Pointer of allocated memory type
(C) void **
(D) int *

Answer: (A)

Explanation: malloc() and calloc() return void *.
We may get warning in C if we don't type cast the return type to appropriate pointer.

Which of the following is/are true
(A) calloc() allocates the memory and also initializes the allocates memory to zero, while memory allocated using malloc() has random data.
(B) malloc() and memset() can be used to get the same effect as calloc().
(C) calloc() takes two arguments, but malloc takes only 1 argument.
(D) Both malloc() and calloc() return 'void *' pointer.
(E) All of the above

Answer: (E)

Explanation: All of the given options are true.

Consider the following three C functions :

filter_none
edit
play_arrow
brightness_4

[PI] int * g (void)
{

```
  int x= 10;
  return (&x);
}

[P2] int * g (void)
{
  int * px;
  *px= 10;
  return px;
}

[P3] int *g (void)
{
  int *px;
  px = (int *) malloc (sizeof(int));
  *px= 10;
  return px;
}
```

Which of the above three functions are likely to cause problems with pointers? (GATE 2001)
(A) Only P3
(B) Only P1 and P3
(C) Only P1 and P2
(D) P1, P2 and P3

Answer: (C)

Explanation: In P1, pointer variable x is a local variable to g(), and g() returns pointer to this variable. x may vanish after g() has returned as x exists on stack. So, &x may become invalid.
In P2, pointer variable px is being assigned a value without allocating memory to it.
P3 works perfectly fine. Memory is allocated to pointer variable px using malloc(). So, px exists on heap, it's existence will remain in memory even after return of g() as it is on heap.

Output?

filter_none
edit
play_arrow
brightness_4

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

void fun(int *a)
{
```

```
    a = (int*)malloc(sizeof(int));
}

int main()
{
    int *p;
    fun(p);
    *p = 6;
    printf("%d\n",*p);
    return(0);
}
```
(A) May not work
(B) Works and prints 6

Answer: (A)

Explanation: The program is not valid. Try replacing “int *p;” with “int *p = NULL;” and it will try to dereference a null pointer.
This is because fun() makes a copy of the pointer, so when malloc() is called, it is setting the copied pointer to the memory location, not p. p is pointing to random memory before and after the call to fun(), and when you dereference it, it will crash.
If you want to add memory to a pointer from a function, you need to pass the address of the pointer (ie. double pointer).

Allocating and freeing memory:

“**free**” method in **C** is used to dynamically de-**allocate** the **memory**. The **memory allocated** using functions malloc() and calloc() is not de-**allocated** on their own. Hence the **free**() method is used, whenever the dynamic **memory allocation** takes place. It helps to reduce wastage of **memory** by **freeing** it.

Allocating memory for arrays of different data types:

https://www.tutorialcup.com/cprogramming/array-memory-allocation.htm

Array Memory Allocation in C Programming

We have already discussed that whenever an array is declared in the program, contiguous memory to it elements are allocated. Initial address of the array – address of the first element of the array is called base address of the array. Each element will occupy the memory space required to accommodate the values for its type, i.e.; depending on elements datatype, 1, 4 or 8 bytes of memory is allocated for each elements. Next successive memory address is allocated to the next element in the array. This process of allocating memory goes on till the number of element in the array gets over.

One Dimensional Array

Below diagram shows how memory is allocated to an integer array of N elements. Its base address – address of its first element is 10000. Since it is an integer array, each of its element will occupy 4 bytes of space. Hence first element occupies memory from 10000 to 10003. Second element of the array occupies immediate next memory address in the memory, i.e.; 10004 which requires another 4 bytes of space. Hence it occupies from 10004 to 10007. In this way all the N elements of the array occupies the memory space.

If the array is a character array, then its elements will occupy 1 byte of memory each. If it is a float array then its elements will occupy 8 bytes of memory each. But this is not the total size or memory allocated for the array. They are the sizes of individual elements in the array. If we need to know the total size of the array, then we need to multiply the number of elements with the size of individual element.

i.e.; **Total memory allocated to an Array = Number of elements * size of one element**

Total memory allocated to an Integer Array of N elements = Number of elements * size of one element

= N * 4 bytes
= 10 * 4 bytes = **40 Bytes**, where N = 10
= 500 * 4 bytes = **2000 Bytes**, where N = 500

Total memory allocated to an character Array of N elements= Number of elements * size of one element

= N * 1 Byte
= 10 * 1 Byte = **10 Bytes**, where N = 10
= 500 * 1 Byte = **500 Bytes**, where N=500

This is how memory is allocated for the single dimensional array.

Multidimensional Array

In the case of multidimensional array, we have elements in the form of rows and columns. Here also memories allocated to the array are contiguous. But the elements assigned to the memory location depend on the two different methods:

Row Major Order

Let us consider a two dimensional array to explain how row major order way of storing elements works. In the case of 2D array, its elements are considered as rows and columns of a table. When we represent an array as intArr[i][j], the first index of it represents the row elements and the next index represents the column elements of each row. When we store the array elements in row major order, first we will store the elements of first row followed by second row and so on. Hence in the memory we can find the elements of first row followed by second row and so on. In memory there will not be any separation between the rows. We have to code in such a way that we have to count the number of elements in each row depending on its column index. But in memory all the rows and their columns will be contiguous. Below diagram will illustrate the same for a 2D array of size 3X3 i.e.; 3 rows and 3 columns.

Array indexes always start from 0. Hence the first element of the 2D array is at intArr[0][0]. This is the first row-first column element. Since it is an integer array, it occupies 4 bytes of space. Next memory space is occupied by the second element of the first row, i.e.; intArr [0][1] – first row-second column element. This continues till all the first row elements are occupied in the memory. Next it picks the second row elements and is placed in the same way as first row. This goes on till all the elements of the array are occupies the memory like below. This is how it is placed in the memory. But seeing the memory address or the value stored in the memory we cannot predict which is the first row or second row or so.

Total size/ memory occupied by 2D array is calculated as

Total memory allocated to 2D Array = Number of elements * size of one element
= Number of Rows * Number of Columns * Size of one element

Total memory allocated to an Integer Array of size MXN = Number of elements * size of one element

=M Rows* N Columns * 4 Bytes
= 10*10 * 4 bytes = **400 Bytes**, where M =N = 10
= 500*5 *4 bytes= **10000 Bytes**, where M=500 and N= 5

Total memory allocated to an character Array of N elements= Number of elements * size of one element

= M Rows* N Columns * 1 Byte
= 10*10 * 1 Byte = **100 Bytes**, where N = 10
= 500*5 * 1 Byte = **2500 Bytes**, where M=500 and N= 5

Column Major Order

This is the opposite method of row major order of storing the elements in the memory. In this method all the first column elements are stored first, followed by second column elements and so on.

Total size/ memory occupied by 2D array is calculated as in the same way as above.

Total memory allocated to 2D Array = Number of elements * size of one element
= Number of Rows * Number of Columns * Size of one element

Total memory allocated to an Integer Array of size MXN = Number of elements * size of one element

=M Rows* N Columns * 4 Bytes
= 10*10 * 4 bytes = **400 Bytes**, where M =N = 10
= 500*5 *4 bytes= **10000 Bytes**, where M=500 and N= 5

Total memory allocated to an character Array of N elements= Number of elements * size of one element

= M Rows* N Columns * 1 Byte
= 10*10 * 1 Byte = 100 Bytes, where N = 10
= 500*5 * 1 Byte = 2500 Bytes, where M=500 and N= 5

If an array is 3D or multidimensional array, then the method of allocating memory is either row major or column major order. Whichever is the method, memory allocated for the whole array is contiguous and its elements will occupy them in the order we choose – row major or column major. The total size of the array is the **total number of elements * size of one element**.

UNIT – V : INTRODUCTION TO ALGORITHMS:

Basic searching algorithms (linear and binary search techniques),Basic sorting algorithms (Bubble, Insertion, Quick, Merge and Selection sort algorithms) Basic concept of order of complexity through the example programs

---

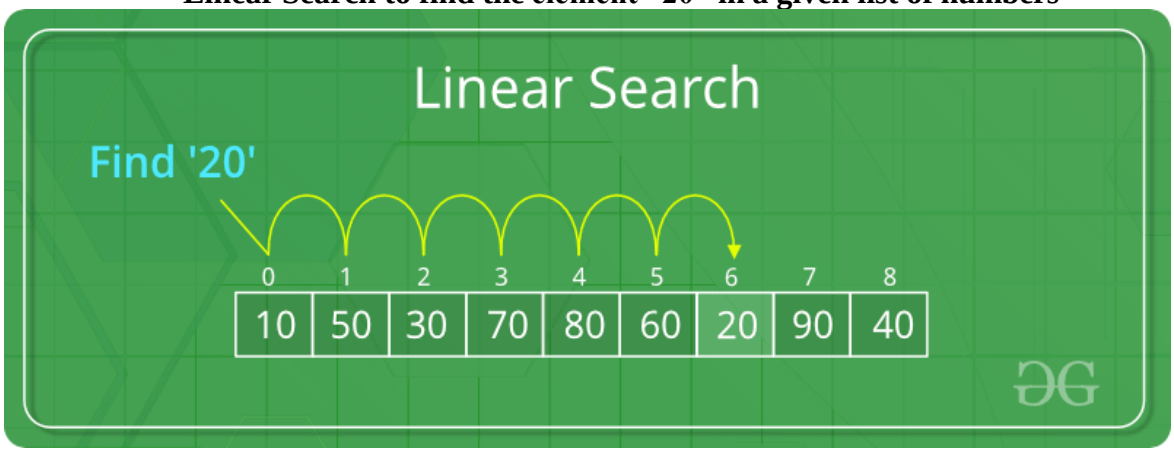
Basic searching algorithms:

Searching Algorithms are designed to check for an element or retrieve an element from any data structure where it is stored. Based on the type of search operation, these algorithms are generally classified into two categories:

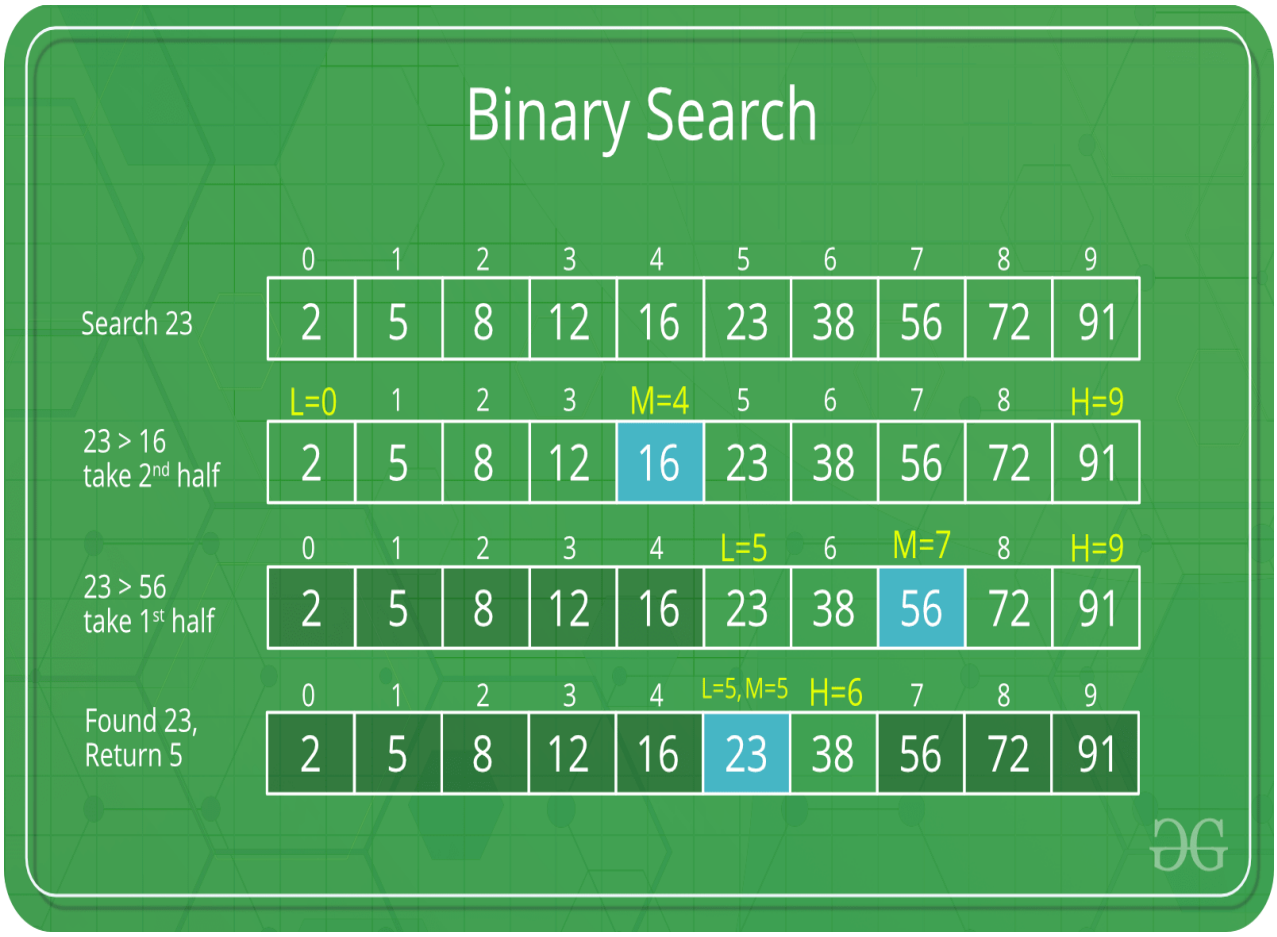
1. **Sequential Search**: In this, the list or array is traversed sequentially and every element is checked. For example: Linear Search.
2. **Interval Search**: These algorithms are specifically designed for searching in sorted data-structures. These type of searching algorithms are much more efficient than Linear Search as they repeatedly target the center of the search structure and divide the search space in half. For Example: Binary Search.

Linear Search to find the element “20” in a given list of numbers

Linear Search to find the element “20” in a given list of numbers



Binary Search to find the element “23” in a given list of numbers



Linear Search

Problem: Given an array arr[] of n elements, write a function to search a given element x in arr[].

Examples :

Input : arr[] = {10, 20, 80, 30, 60, 50,
110, 100, 130, 170}
x = 110;

Output : 6

Element x is present at index 6

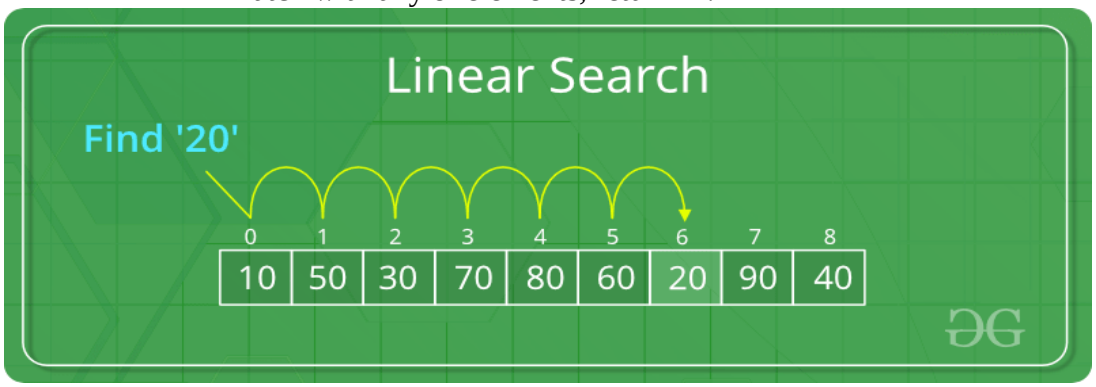
Input : arr[] = {10, 20, 80, 30, 60, 50,
110, 100, 130, 170}
x = 175;

Output : -1

Element x is not present in arr[].

A simple approach is to do **linear search**, i.e

- Start from the leftmost element of arr[] and one by one compare x with each element of arr[]
- If x matches with an element, return the index.
- If x doesn't match with any of elements, return -1.




```
// C++ code to linearly search x in arr[]. If x
// is present then return its location, otherwise
// return -1

#include <stdio.h>

int search(int arr[], int n, int x)
{
    int i;
    for (i = 0; i < n; i++)
        if (arr[i] == x)
            return i;
    return -1;
}

int main(void)
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int x = 10;
    int n = sizeof(arr) / sizeof(arr[0]);
    int result = search(arr, n, x);
    (result == -1) ? printf("Element is not present in array")
                   : printf("Element is present at index %d",
                            result);
```

```
    return 0;
}
```

Output:

```
Element is present at index 3
```

The time complexity of above algorithm is O(n).

Linear search is rarely used practically because other search algorithms such as the binary search algorithm and hash tables allow significantly faster searching comparison to Linear search.

Binary Search

Given a sorted array arr[] of n elements, write a function to search a given element x in arr[].

A simple approach is to do **linear search.**The time complexity of above algorithm is O(n). Another approach to perform the same task is using Binary Search.
Binary Search: Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

Example :

The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(Log n).

We basically ignore half of the elements just after one comparison.

1. Compare x with the middle element.
2. If x matches with middle element, we return the mid index.

3. Else If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So we recur for right half.
4. Else (x is smaller) recur for the left half.

Recursive implementation of Binary Search

```
// C program to implement recursive Binary Search
#include <stdio.h>

// A recursive binary search function. It returns
// location of x in given array arr[l..r] is present,
// otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
   if (r >= l) {
      int mid = l + (r - l) / 2;

      // If the element is present at the middle
      // itself
      if (arr[mid] == x)
         return mid;

      // If element is smaller than mid, then
      // it can only be present in left subarray
      if (arr[mid] > x)
         return binarySearch(arr, l, mid - 1, x);

      // Else the element can only be present
      // in right subarray
      return binarySearch(arr, mid + 1, r, x);
   }

   // We reach here when element is not
   // present in array
   return -1;
}

int main(void)
{
   int arr[] = { 2, 3, 4, 10, 40 };
   int n = sizeof(arr) / sizeof(arr[0]);
   int x = 10;
   int result = binarySearch(arr, 0, n - 1, x);
   (result == -1) ? printf("Element is not present in array")
                  : printf("Element is present at index %d",
                           result);
```

```
    return 0;
}
```

Output :

```
Element is present at index 3
```

Iterative implementation of Binary Search

```
// C program to implement iterative Binary Search
#include <stdio.h>

// A iterative binary search function. It returns
// location of x in given array arr[l..r] if present,
// otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
    while (l <= r) {
        int m = l + (r - l) / 2;

        // Check if x is present at mid
        if (arr[m] == x)
            return m;

        // If x greater, ignore left half
        if (arr[m] < x)
            l = m + 1;

        // If x is smaller, ignore right half
        else
            r = m - 1;
    }

    // if we reach here, then element was
    // not present
    return -1;
}

int main(void)
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int n = sizeof(arr) / sizeof(arr[0]);
    int x = 10;
    int result = binarySearch(arr, 0, n - 1, x);
    (result == -1) ? printf("Element is not present in array")
                   : printf("Element is present at "
```

```
                "index %d",
                result);
    return 0;
}
```

Output :

```
Element is present at index 3
```

Time Complexity:

The time complexity of Binary Search can be written as

$T(n) = T(n/2) + c$

The above recurrence can be solved either using Recurrence T ree method or Master method. It falls in case II of Master Method and solution of the recurrence is .

Auxiliary Space: O(1) in case of iterative implementation. In case of recursive implementation, O(Logn) recursion call stack space.

Basic sorting algorithms (Bubble, Insertion, Quick, Merge and Selection sort algorithms)

Basic sorting algorithms:

Bubble sort algorithm:
Bubble Sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

Example:
First Pass:
(**5** **1** 4 2 8) –> (**1** **5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
(1 **5** **4** 2 8) –> (1 **4** **5** 2 8), Swap since 5 > 4
(1 4 **5** **2** 8) –> (1 4 **2** **5** 8), Swap since 5 > 2
(1 4 2 **5** **8**) –> (1 4 2 **5** **8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.
Second Pass:
(**1** **4** 2 5 8) –> (**1** **4** 2 5 8)
(1 **4** **2** 5 8) –> (1 **2** **4** 5 8), Swap since 4 > 2
(1 2 **4** **5** 8) –> (1 2 **4** **5** 8)
(1 2 4 **5** **8**) –> (1 2 4 **5** **8**)
Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.
Third Pass:
(**1** **2** 4 5 8) –> (**1** **2** 4 5 8)

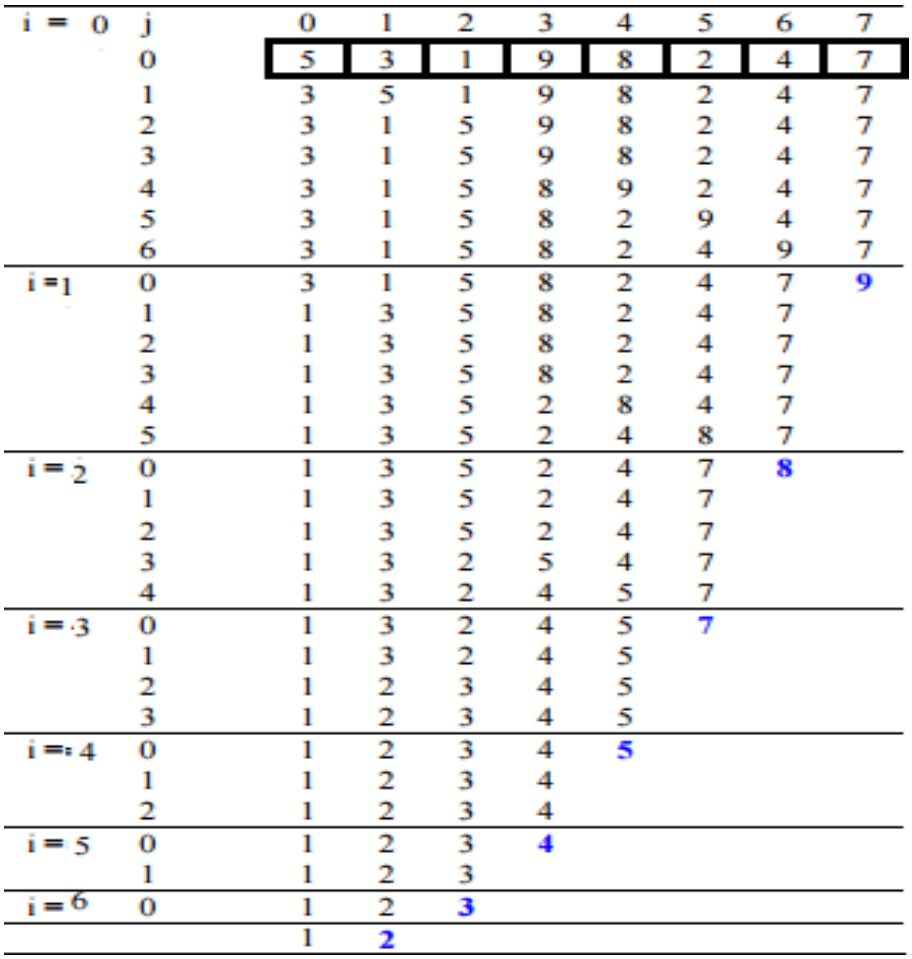
(1 **2 4** 5 8) –> (1 **2 4** 5 8)
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)

```
// C++ program for implementation of Bubble sort
#include <bits/stdc++.h>
using namespace std;

void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

// A function to implement bubble sort
void bubbleSort(int arr[], int n)
{
    int i, j;
    for (i = 0; i < n-1; i++)

    // Last i elements are already in place
    for (j = 0; j < n-i-1; j++)
        if (arr[j] > arr[j+1])
            swap(&arr[j], &arr[j+1]);
}

/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver code
int main()
{
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr)/sizeof(arr[0]);
    bubbleSort(arr, n);
    cout<<"Sorted array: \n";
    printArray(arr, n);
    return 0;
}
```

Output:

```
Sorted array:

11 12 22 25 34 64 90
```

<!—**-Illustration :**



		0	1	2	3	4	5	6	7
i = 0	j								
	0	5	3	1	9	8	2	4	7
	1	3	5	1	9	8	2	4	7
	2	3	1	5	9	8	2	4	7
	3	3	1	5	9	8	2	4	7
	4	3	1	5	8	9	2	4	7
	5	3	1	5	8	2	9	4	7
	6	3	1	5	8	2	4	9	7
i =1	0	3	1	5	8	2	4	7	9
	1	1	3	5	8	2	4	7	
	2	1	3	5	8	2	4	7	
	3	1	3	5	8	2	4	7	
	4	1	3	5	2	8	4	7	
	5	1	3	5	2	4	8	7	
i = 2	0	1	3	5	2	4	7	8	
	1	1	3	5	2	4	7		
	2	1	3	5	2	4	7		
	3	1	3	2	5	4	7		
	4	1	3	2	4	5	7		
i = 3	0	1	3	2	4	5	7		
	1	1	3	2	4	5			
	2	1	2	3	4	5			
	3	1	2	3	4	5			
i = 4	0	1	2	3	4	5			
	1	1	2	3	4				
	2	1	2	3	4				
i = 5	0	1	2	3	4				
	1	1	2	3					
i = 6	0	1	2	3					
		1	2						

Insertion sort:

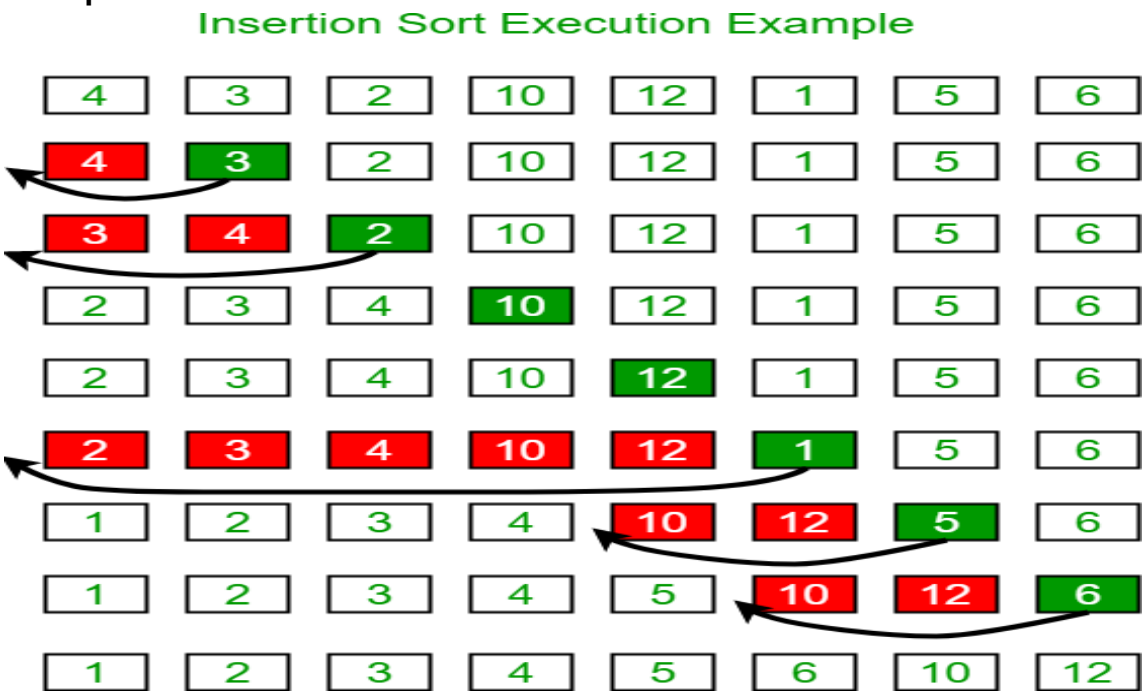
Insertion Sort

Insertion sort is a simple sorting algorithm that works the way we sort playing cards in our hands.

Algorithm

```
// Sort an arr[] of size n
insertionSort(arr, n)
```

Loop from i = 1 to n-1.
……a) Pick element arr[i] and insert it into sorted sequence arr[0…i-1]
Example:



Another Example:
12, 11, 13, 5, 6
Let us loop for i = 1 (second element of the array) to 4 (last element of the array)

i = 1. Since 11 is smaller than 12, move 12 and insert 11 before 12
11, 12, 13, 5, 6
i = 2. 13 will remain at its position as all elements in A[0..I-1] are smaller than 13
11, 12, 13, 5, 6
i = 3. 5 will move to the beginning and all other elements from 11 to 13 will move one position ahead of their current position.
5, 11, 12, 13, 6
i = 4. 6 will move to position after 5, and elements from 11 to 13 will move one position ahead of their current position.
5, 6, 11, 12, 13

```
// C program for insertion sort
#include <math.h>
#include <stdio.h>

/* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;

        /* Move elements of arr[0..i-1], that are
```

```
          greater than key, to one position ahead
          of their current position */
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

// A utility function to print an array of size n
void printArray(int arr[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

/* Driver program to test insertion sort */
int main()
{
    int arr[] = { 12, 11, 13, 5, 6 };
    int n = sizeof(arr) / sizeof(arr[0]);

    insertionSort(arr, n);
    printArray(arr, n);

    return 0;
}
```

Output:

```
5 6 11 12 13
```

Time Complexity: O(n*2)

Auxiliary Space: O(1)

Boundary Cases: Insertion sort takes maximum time to sort if elements are sorted in reverse order. And it takes minimum time (Order of n) when elements are already sorted.

Algorithmic Paradigm: Incremental Approach

Sorting In Place: Yes

Stable: Yes

Online: Yes

Uses: Insertion sort is used when number of elements is small. It can also be useful when input array is almost sorted, only few elements are misplaced in complete big array.

What is Binary Insertion Sort?
We can use binary search to reduce the number of comparisons in normal insertion sort. Binary Insertion Sort uses binary search to find the proper location to insert the selected item at each iteration. In normal insertion, sorting takes O(i) (at ith iteration) in worst case. We can reduce it to O(logi) by using binary search. The algorithm, as a whole, still has a running worst case running time of O(n2) because of the series of swaps required for each insertion. Refer this for implementation.

How to implement Insertion Sort for Linked List?
Below is simple insertion sort algorithm for linked list.

```
1) Create an empty sorted (or result) list

2) Traverse the given list, do following for every node.

......a) Insert current node in sorted way in sorted or result list.

3) Change head of given linked list to head of sorted (or result) list.
```

Quick sort:

Like Merge Sort, QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

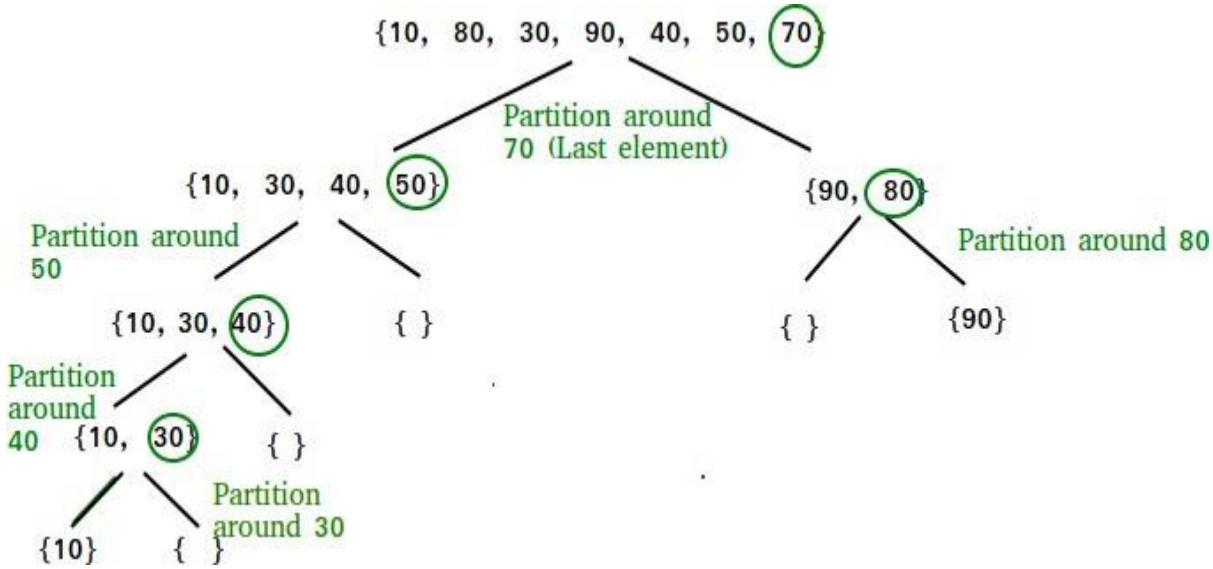
1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

Pseudo Code for recursive QuickSort function :

```
/* low  --> Starting index,  high  --> Ending index */

quickSort(arr[], low, high)

{

    if (low < high)

    {

        /* pi is partitioning index, arr[pi] is now

           at right place */

        pi = partition(arr, low, high);
```

```
        quickSort(arr, low, pi - 1); // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```



Partition Algorithm

There can be many ways to do partition, following pseudo code adopts the method given in CLRS book. The logic is simple, we start from the leftmost element and keep track of index of smaller (or equal to) elements as i. While traversing, if we find a smaller element, we swap current element with arr[i]. Otherwise we ignore current element.

```
/* low --> Starting index, high --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1); // Before pi
        quickSort(arr, pi + 1, high); // After pi
```

```
    }
}
```

Pseudo code for partition()

```
/* This function takes last element as pivot, places
   the pivot element at its correct position in sorted
   array, and places all smaller (smaller than pivot)
   to left of pivot and all greater elements to right
   of pivot */
partition (arr[], low, high)
{
    // pivot (Element to be placed at right position)
    pivot = arr[high];

    i = (low - 1)  // Index of smaller element

    for (j = low; j <= high- 1; j++)
    {
        // If current element is smaller than the pivot
        if (arr[j] < pivot)
        {
            i++;    // increment index of smaller element
            swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
}
```

Illustration of partition() :

```
arr[] = {10, 80, 30, 90, 40, 50, 70}
Indexes:  0   1   2   3   4   5   6
```

```
low = 0, high = 6, pivot = arr[h] = 70

Initialize index of smaller element, i = -1

Traverse elements from j = low to high-1
j = 0 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 0
arr[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j
                                     // are same

j = 1 : Since arr[j] > pivot, do nothing
// No change in i and arr[]

j = 2 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 1
arr[] = {10, 30, 80, 90, 40, 50, 70} // We swap 80 and 30

j = 3 : Since arr[j] > pivot, do nothing
// No change in i and arr[]

j = 4 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 2
arr[] = {10, 30, 40, 90, 80, 50, 70} // 80 and 40 Swapped
j = 5 : Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]
i = 3
arr[] = {10, 30, 40, 50, 80, 90, 70} // 90 and 50 Swapped

We come out of loop because j is now equal to high-1.
Finally we place pivot at correct position by swapping
arr[i+1] and arr[high] (or pivot)
arr[] = {10, 30, 40, 50, 70, 90, 80} // 80 and 70 Swapped

Now 70 is at its correct place. All elements smaller than
70 are before it and all elements greater than 70 are after
it.
```

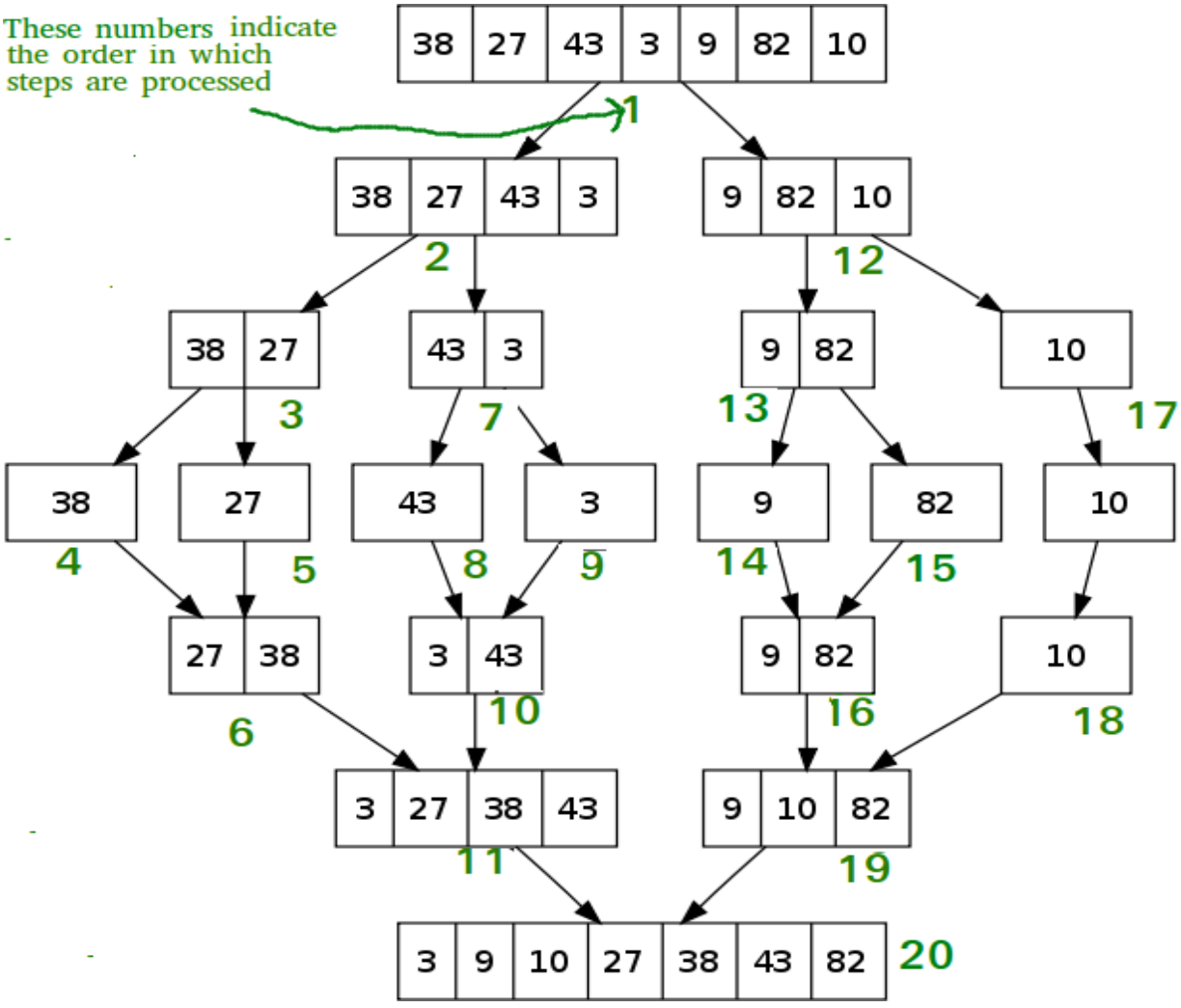
Merge sort:

Like QuickSort, Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. **The merge() function** is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one. See following C implementation for details.

```
MergeSort(arr[], l,  r)
If r > l
```

```
1. Find the middle point to divide the array into two halves:
        middle m = (l+r)/2
2. Call mergeSort for first half:
        Call mergeSort(arr, l, m)
3. Call mergeSort for second half:
        Call mergeSort(arr, m+1, r)
4. Merge the two halves sorted in step 2 and 3:
        Call merge(arr, l, m, r)
```

The following diagram from wikipedia shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}. If we take a closer look at the diagram, we can see that the array is recursively divided in two halves till the size becomes 1. Once the size becomes 1, the merge processes comes into action and starts merging arrays back till the complete array is merged.




```
/* C program for Merge Sort */
#include<stdlib.h>
```

```
#include<stdio.h>

// Merges two subarrays of arr[].
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
void merge(int arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 =  r - m;

    /* create temp arrays */
    int L[n1], R[n2];

    /* Copy data to temp arrays L[] and R[] */
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1+ j];

    /* Merge the temp arrays back into arr[l..r]*/
    i = 0; // Initial index of first subarray
    j = 0; // Initial index of second subarray
    k = l; // Initial index of merged subarray
    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    /* Copy the remaining elements of L[], if there
       are any */
    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }

    /* Copy the remaining elements of R[], if there
       are any */
    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
```

```
    }
}

/* l is for left index and r is right index of the
   sub-array of arr to be sorted */
void mergeSort(int arr[], int l, int r)
{
    if (l < r)
    {
        // Same as (l+r)/2, but avoids overflow for
        // large l and h
        int m = l+(r-l)/2;

        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m+1, r);

        merge(arr, l, m, r);
    }
}

/* UTILITY FUNCTIONS */
/* Function to print an array */
void printArray(int A[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", A[i]);
    printf("\n");
}

/* Driver program to test above functions */
int main()
{
    int arr[] = {12, 11, 13, 5, 6, 7};
    int arr_size = sizeof(arr)/sizeof(arr[0]);

    printf("Given array is \n");
    printArray(arr, arr_size);

    mergeSort(arr, 0, arr_size - 1);

    printf("\nSorted array is \n");
    printArray(arr, arr_size);
    return 0;
}
```

Output:

```
Given array is

12 11 13 5 6 7
```

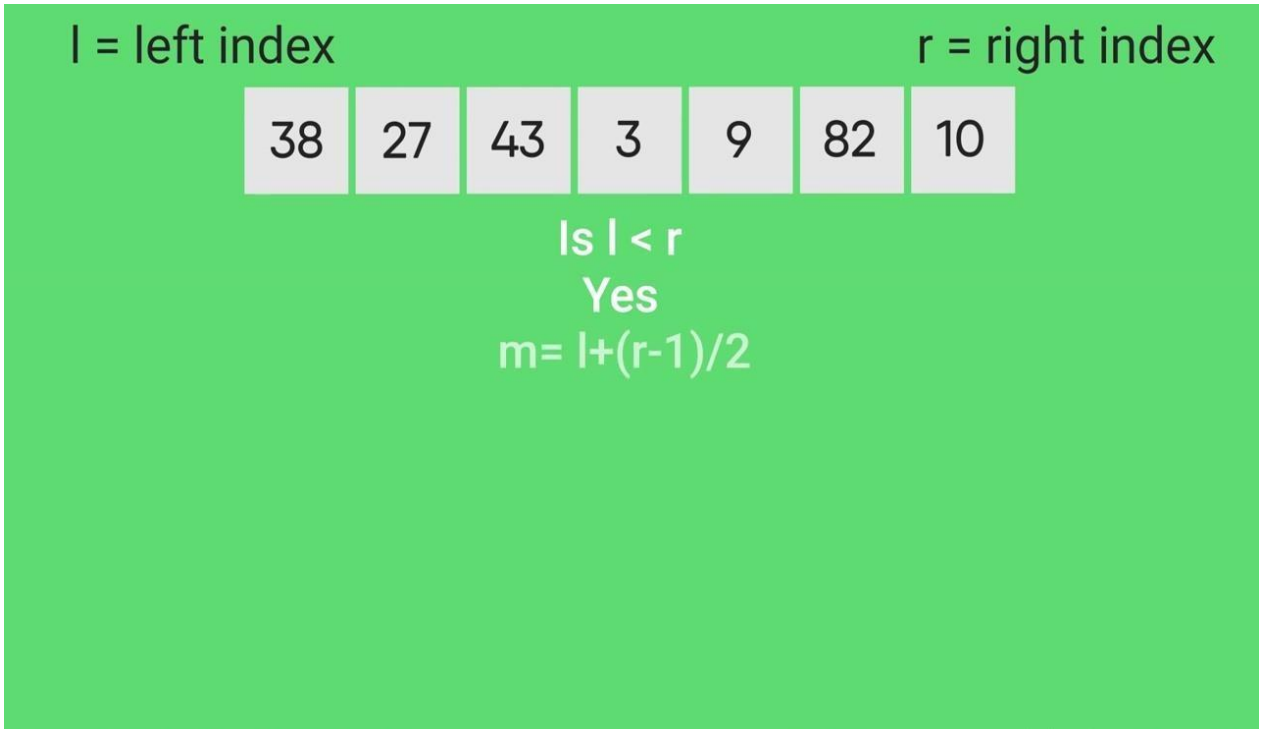
```
Sorted array is

5 6 7 11 12 13
```

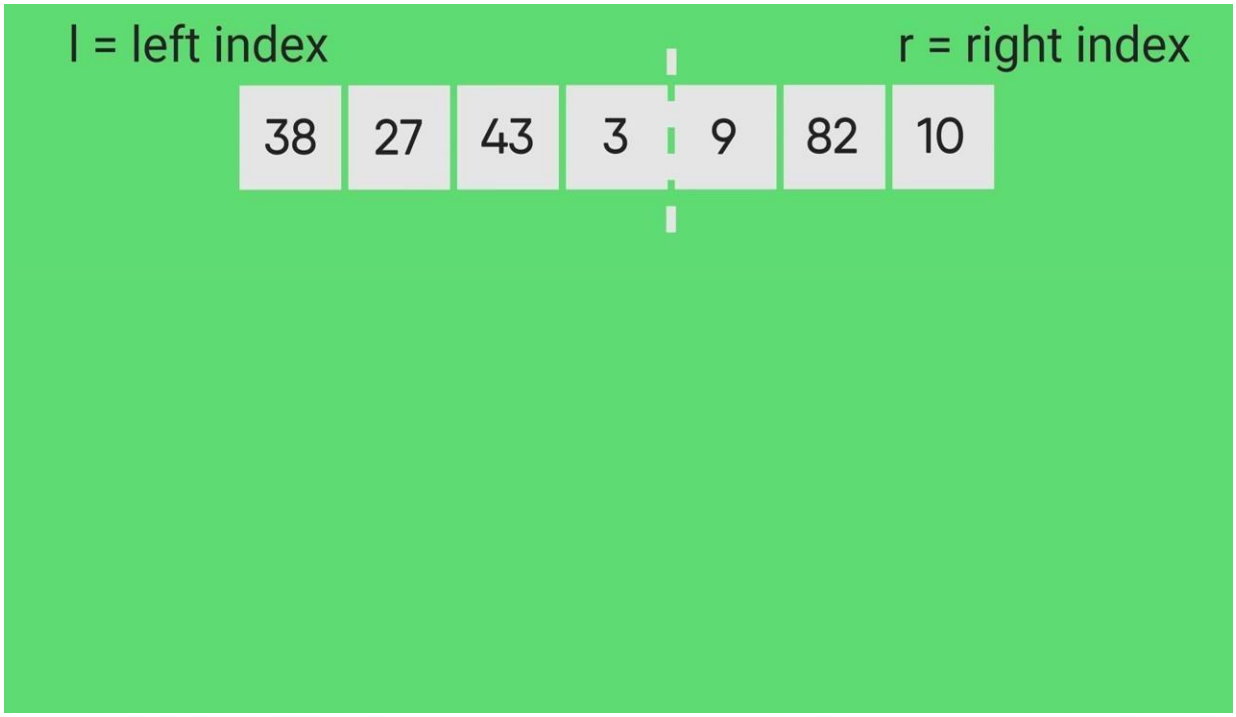
Time Complexity: Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

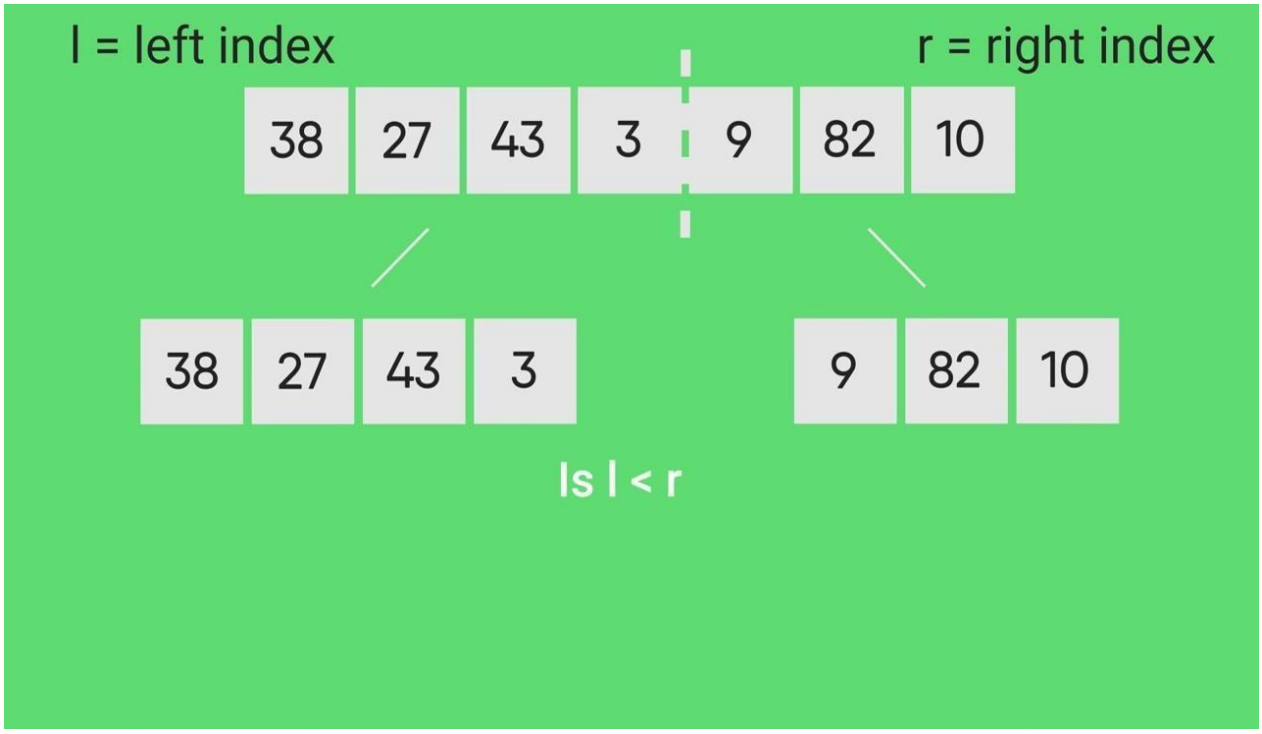
T(n) = 2T(n/2) +
The above recurrence can be solved either using Recurrence Tree method or Master method. It

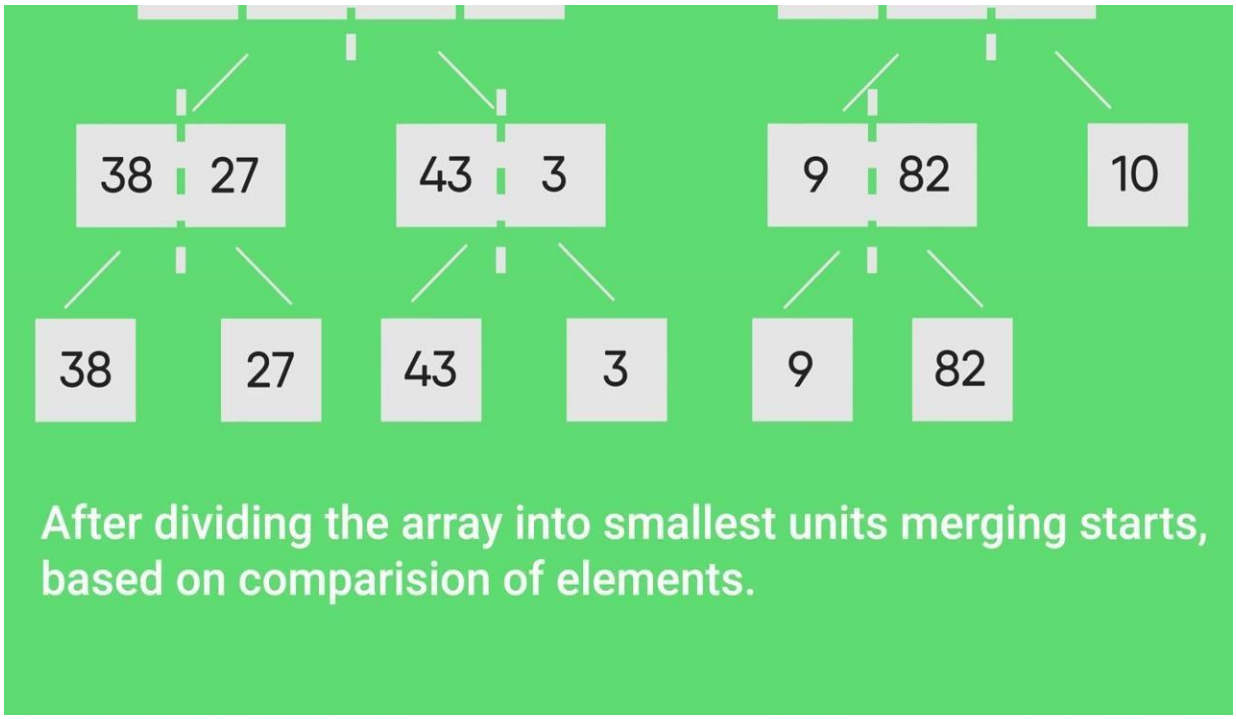
falls in case II of Master Method and solution of the recurrence is .
Time complexity of Merge Sort is in all 3 cases (worst, average and best) as merge sort always divides the array into two halves and take linear time to merge two halves.
Auxiliary Space: O(n)
Algorithmic Paradigm: Divide and Conquer
Sorting In Place: No in a typical implementation
Stable: Yes
Applications of Merge Sort

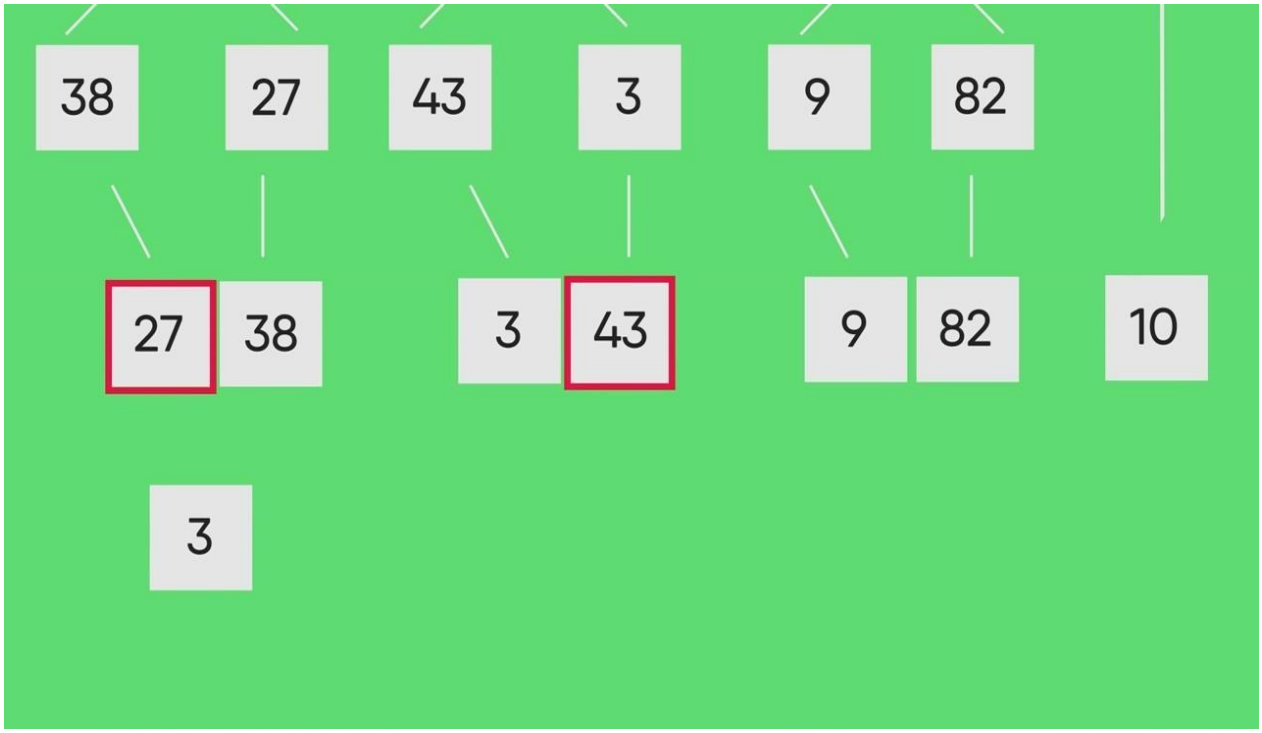
1. Merge Sort is useful for sorting linked lists in O(nLogn) time.In the case of linked lists, the case is different mainly due to the difference in memory allocation of arrays and linked lists. Unlike arrays, linked list nodes may not be adjacent in memory. Unlike an array, in the linked list, we can insert items in the middle in O(1) extra space and O(1) time. Therefore merge operation of merge sort can be implemented without extra space for linked lists.
   In arrays, we can do random access as elements are contiguous in memory. Let us say we have an integer (4-byte) array A and let the address of A[0] be x then to access A[i], we can directly access the memory at (x + i*4). Unlike arrays, we can not do random access in the linked list. Quick Sort requires a lot of this kind of access. In linked list to access i'th index, we have to travel each and every node from the head to i'th node as we don't have a continuous block of memory. Therefore, the overhead increases for quicksort. Merge sort accesses data sequentially and the need of random access is low.
2. Inversion Count Problem
3. Used in External Sorting

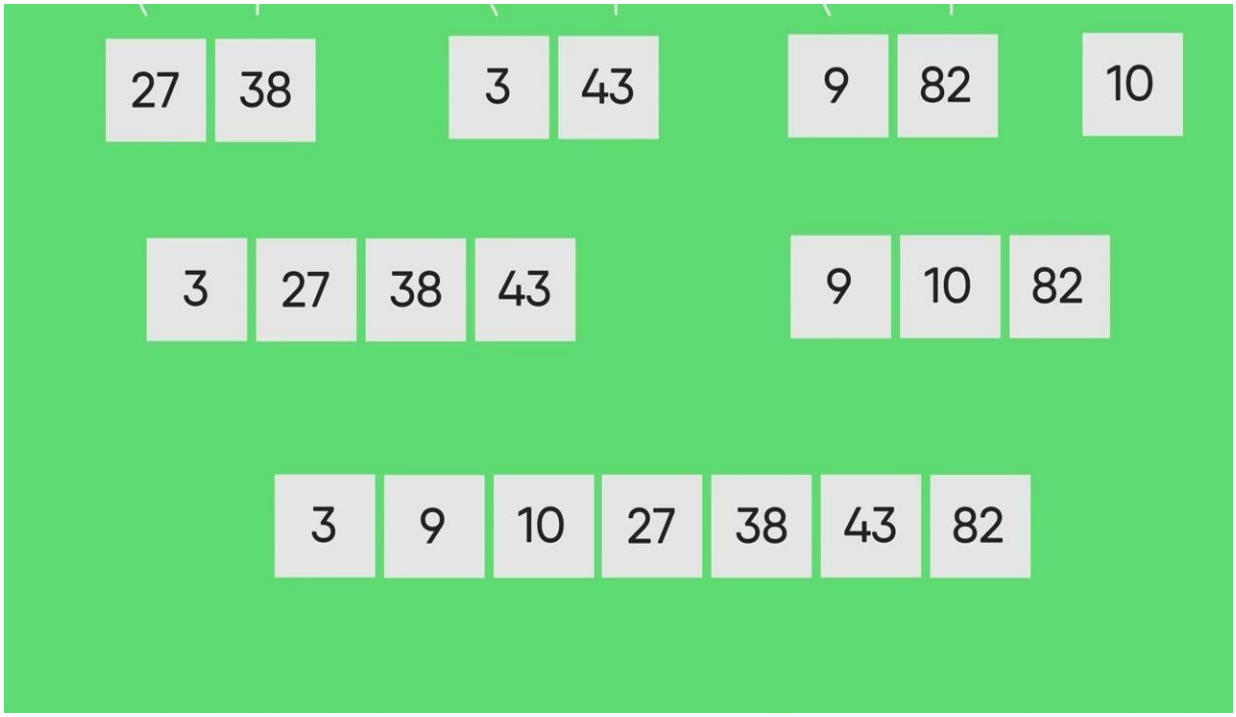

l = left index
r = right index
38
27
43
3
9
82
10
Is l < r
Yes
m= l+(r-1)/2


l = left index
r = right index
38
27
43
3
9
82
10


l = left index
r = right index
38 27 43 3 9 82 10
38 27 43 3
9 82 10
Is l < r


38 27
43 3
9 82
10
38
27
43
3
9
82
After dividing the array into smallest units merging starts, based on comparision of elements.


38
27
43
3
9
82
27
38
3
43
9
82
10
3


27
38
3
43
9
82
10
3
27
38
43
9
10
82
3
9
10
27
38
43
82

Selection sort:

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

1) The subarray which is already sorted.
2) Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

Following example explains the above steps:

arr[] = 64 25 12 22 11

// Find the minimum element in arr[0...4]

// and place it at beginning

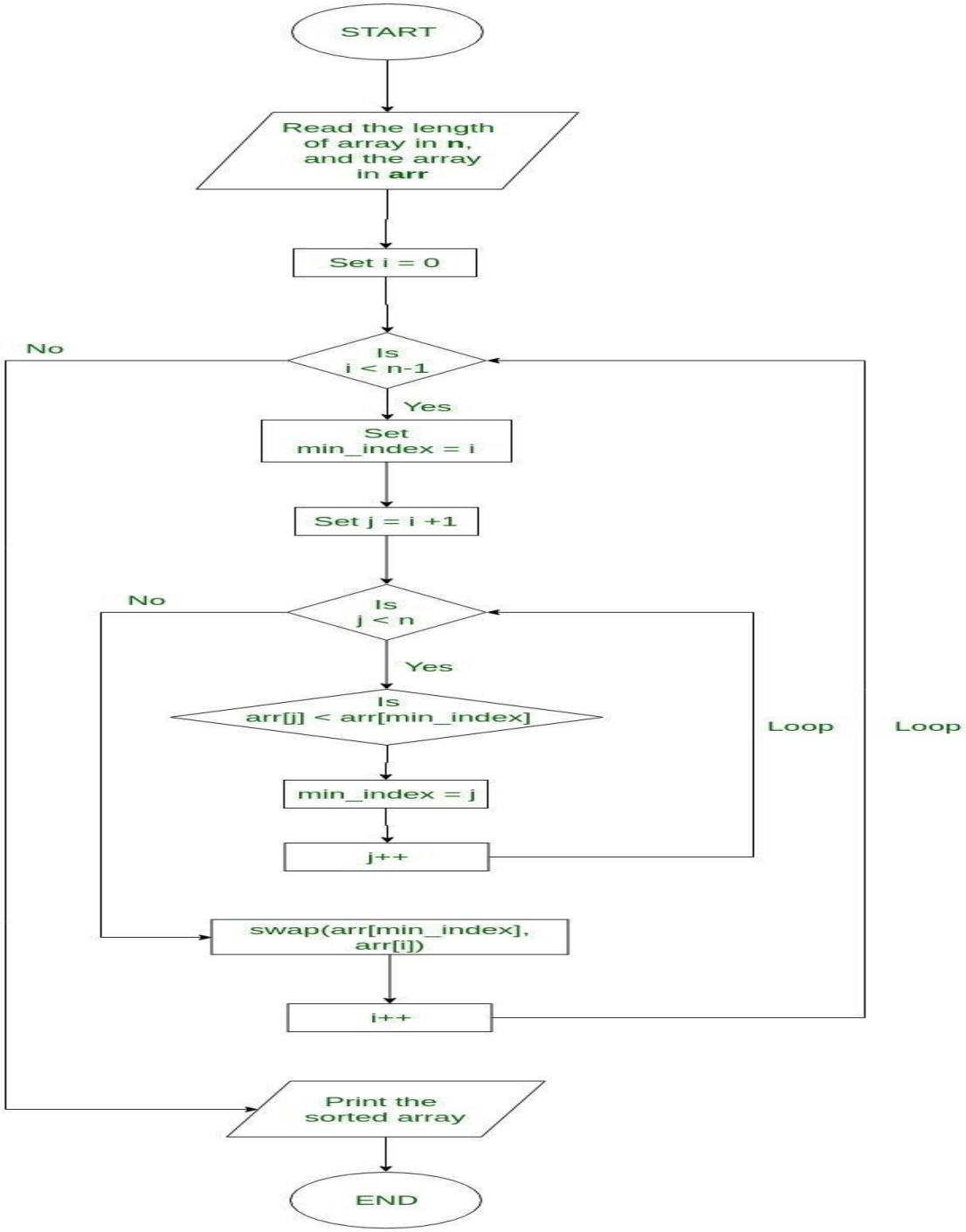
11 25 12 22 64

// Find the minimum element in arr[1...4]
// and place it at beginning of arr[1...4]
11 **12** 25 22 64

// Find the minimum element in arr[2...4]
// and place it at beginning of arr[2...4]
11 12 **22** 25 64

// Find the minimum element in arr[3...4]
// and place it at beginning of arr[3...4]
11 12 22 **25** 64

Flowchart of the Selection Sort:



Flowchart for Selection Sort

```
// C++ program for implementation of selection sort
#include <bits/stdc++.h>
using namespace std;

void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

void selectionSort(int arr[], int n)
{
    int i, j, min_idx;

    // One by one move boundary of unsorted subarray
    for (i = 0; i < n-1; i++)
    {
        // Find the minimum element in unsorted array
        min_idx = i;
        for (j = i+1; j < n; j++)
        if (arr[j] < arr[min_idx])
            min_idx = j;

        // Swap the found minimum element with the first element
        swap(&arr[min_idx], &arr[i]);
    }
}

/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver program to test above functions
int main()
{
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr)/sizeof(arr[0]);
    selectionSort(arr, n);
    cout << "Sorted array: \n";
    printArray(arr, n);
    return 0;
}
```

Output:

```
Sorted array:

11 12 22 25 64
```

Time Complexity: $O(n^2)$ as there are two nested loops.
Auxiliary Space: O(1)
The good thing about selection sort is it never makes more than O(n) swaps and can be useful when memory write is a costly operation.

Basic concept of order of complexity through the example programs

Understanding Time Complexity with Simple Examples

A lot of students get confused while understanding the concept of time-complexity, but in this article, we will explain it with a very simple example:

Imagine a classroom of 100 students in which you gave your pen to one person. Now, you want that pen. Here are some ways to find the pen and what the O order is.

$O(n^2)$: You go and ask the first person of the class, if he has the pen. Also, you ask this person about other 99 people in the classroom if they have that pen and so on, This is what we call $O(n^2)$.
O(n): Going and asking each student individually is O(N).
O(log n): Now I divide the class into two groups, then ask: "Is it on the left side, or the right side of the classroom?" Then I take that group and divide it into two and ask again, and so on. Repeat the process till you are left with one student who has your pen. This is what you mean by O(log n).
I might need to do the $O(n^2)$ search if only one student knows on which student the pen is hidden. I'd use the O(n) if one student had the pen and only they knew it. I'd use the O(log n) search if all the students knew, but would only tell me if I guessed the right side.
NOTE :
We are interested in rate of growth of time with respect to the inputs taken during the program execution .

Another Example

Time Complexity of algorithm/code is **not** equal to the actual time required to execute a particular code but the number of times a statement executes. We can prove this by using time command. For example, Write code in C/C++ or any other language to find maximum between N numbers, where N varies from 10, 100,

1000, 10000. And compile that code on Linux based operating system (Fedora or Ubuntu) with below command:

```
gcc program.c – o program

run it with time ./program
```

You will get surprising results i.e. for N = 10 you may get 0.5ms time and for N = 10, 000 you may get 0.2 ms time. Also, you will get different timings on the different machine. So, we can say that actual time requires to execute code is machine dependent (whether you are using pentium1 or pentiun5) and also it considers network load if your machine is in LAN/WAN. Even you will not get the same timings on the same machine for the same code, the reason behind that the current network load.

Now, the question arises if time complexity is not the actual time require executing the code then what is it?
The answer is : Instead of measuring actual time required in executing each statement in the code, we consider how many times each statement execute.
For example:

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

Output:

```
Hello World
```

In above code “Hello World!!!” print only once on a screen. So, time complexity is constant: O(1) i.e. every time constant amount of time require to execute code, no matter which operating system or which machine configurations you are using.
Now consider another code:

filter_none

edit

play_arrow

brightness_4

```
#include <stdio.h>

void main()

{

    int i, n = 8;
```

```
    for (i = 1; i <= n; i++) {
        printf("Hello Word !!!");
    }
}
```

Output:

```
Hello Word !!!Hello Word !!!Hello Word !!!Hello Word !!!
Hello Word !!!Hello Word !!!Hello Word !!!Hello Word !!!
```

In above code “Hello World!!!” will print N times. So, time complexity of above code is O(N).

Source : Reddit

ADDITIONAL INFORMATION :

For example:
Let us consider a model machine which has the following specifications:
–Single processor
–32 bit
–Sequential execution
–1 unit time for arithmetic and logical operations
–1 unit time for assignment and return statements

1.Sum of 2 numbers :

filter_none

edit

play_arrow

brightness_4

```
Pseudocode:

Sum(a,b){

return a+b  //Takes 2 unit of time(constant) one for arithmetic operation and one for retur

}
```

Tsum= 2 = C =O(1)

2.Sum of all elements of a list :

```
Pseudocode:
```

```
list_Sum(A,n){//A->array and n->number of elements in the array
total =0            // cost=1  no of times=1
for i=0 to n-1      // cost=2  no of times=n+1 (+1 for the end false condition)
sum = sum + A[i]    // cost=2  no of times=n
return sum          // cost=1  no of times=1
}
```

Tsum=1 + 2 * (n+1) + 2 * n + 1 = 4n + 1 =C1 * n + C2 = O(n)

Definition - What does *Time Complexity* mean?

Time complexity is a concept in computer science that deals with the quantification of the amount of time taken by a set of code or algorithm to process or run as a function of the amount of input.

In other words, time complexity is essentially efficiency, or how long a program function takes to process a given input.