

Computer Science

Paper 1

Tejas Shah



Operating System	1
Scope of syllabus:.....	1
Introduction	1
Popular Operating Systems.....	1
Classification of Operating Systems.....	3
Components and Functions of an Operating System	4
Information Management	4
Process Management	9
Memory Management.....	11
Security and Protection	15
Data Structure	17
Scope of syllabus:.....	17
Introduction	17
Array.....	17
Record.....	21
Linked List	21
Tree	23
HTML.....	25
Scope of syllabus:.....	25
Introduction	25
Basic structure of a HTML document	25
Working with Text.....	26
Working with Links.....	30
Working with Lists.....	30
Working with Tables	31
Working with Images	33
C++	34
Scope of syllabus:.....	34
Introduction	34
First C++ program	34
C++ Basics.....	35
Control Structures.....	38
Functions.....	40
Arrays.....	42
Pointers.....	44
Strings	46
Object Oriented Programming	46
Classes and Objects.....	48
Constructors and Destructors.....	51
Operator Overloading.....	53
Type Conversion	54
Inheritance.....	56
Polymorphism	58
Working with Files	59
Important Questions	62
Operating Systems.....	62
Data Structures	62
HTML.....	62
C++	63

Operating System

Probable Marks: 22 marks

Scope of syllabus:

- What is an Operating System?
- Services in OS.
- Overview of Windows 98, Windows NT and Linux operating systems.
- Concepts related to Information Management - File system, Device driver and Terminal I/O.
- Concepts related to Process Management - Process, Concepts of multiprogramming, Context switching, Process states, Priority and Multitasking.
- Concepts related to Memory Management - Memory map of single user computer system, Memory partitioning, Paging, Segmentation and Virtual memory.
- Basics of GUI - GUI features like window, task list, drag, resize, minimize, maximize and close.
- Access and Security aspects of OS - Security threats, Computer Viruses, Computer Worms, Prevention mechanisms.

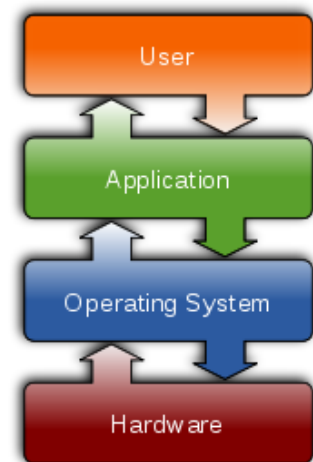
Introduction

What is an Operating System?

An operating system is a software program that manages the computer hardware and acts as an interface between user of a computer and the computer hardware. The purpose of the operating system is to provide an environment in which the user may execute programs.

At the simplest level, an operating system does two things:

1. It manages the hardware and software resources of the system. In a desktop computer, these resources include such things as the processor, memory, disk space and more (In a cell phone, they include the keypad, the screen, the address book, the phone dialer, the battery and the network connection). Thus the OS act as a resource allocator.
2. It provides a stable, consistent way for applications to deal with the hardware without having to know all the details of the hardware. This relieves application programs from having to manage these details and makes it easier to write applications.



The figure shows the relation of an Operating System with respect to the components of a computer system.

Popular Operating Systems

Windows Operating System

Windows 98 and Windows XP

MS-DOS was the first operating system developed by Microsoft for the IBM personal computer. The first version of MS-DOS 1.0 was released in August 1981. Later on Microsoft released newer versions of MS-DOS. Windows 3.0 was introduced as a graphical user interface built for MS-DOS. Over the years Microsoft has released Windows 95 and Windows 98 operating systems. Each version of the Windows operating system has added new features and made it easier for people to use the computer.

Features of Windows 98 and Windows XP

- Easy to use - Better GUI, Ability to install new hardware without restarting computer, Plug n Play support for new hardware makes it easy to install new hardware.
- Fast - Programs run faster than on previous versions of Windows.
- Web Integration - Easily connect to the internet, Comes with Internet Explorer to browse the internet and Outlook Express to check email.
- Support for multimedia - Supports playback of audio/video CDs and DVD.

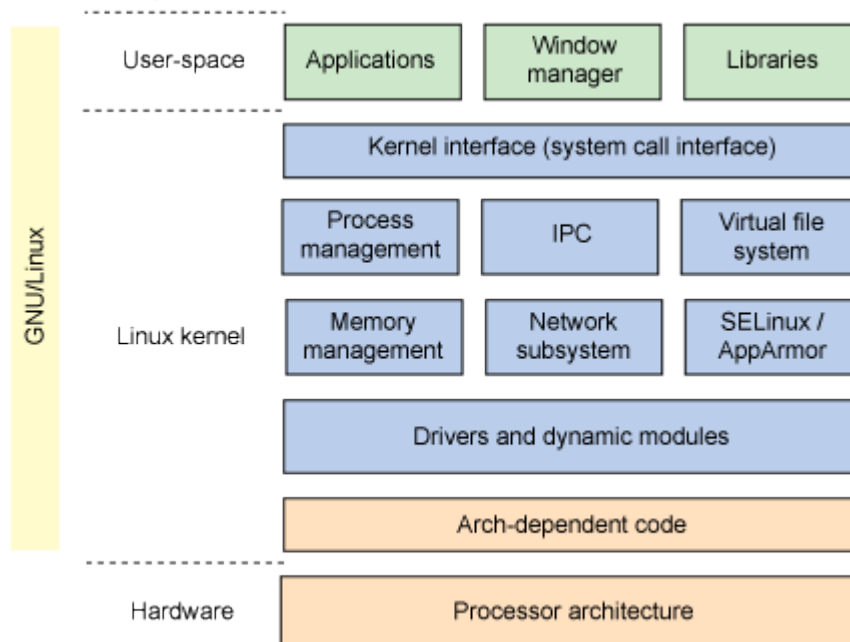
Windows NT, Windows 2000, Windows 2003 and Windows 2008

Windows NT is a family of operating systems produced by Microsoft, the first version of which was released in July 1993. It was originally designed to be a powerful high-level-language-based, processor-independent, multiprocessing, multiuser operating system with features comparable to UNIX. It was intended to complement consumer versions of Windows that were based on MS-DOS. NT was the first fully 32-bit version of Windows. Windows 2000, Windows XP, Windows Server 2003, Windows Vista, Windows Home Server, Windows Server 2008 and Windows 7 and Windows 8 are based on Windows NT.

Linux Operating System

Linux is a generic term referring to Unix-like computer operating systems based on the Linux kernel. Their development is one of the most prominent examples of free and open source software collaboration. The name “Linux” comes from the Linux kernel (a computer program that manages input/output requests from software and translates them into data processing instructions for the CPU) originally written in 1991 by Linus Torvalds. Linux is a multiuser, multitasking operating system supported on a variety of hardware platforms. It supports a wide base of applications and is available free of cost. Popular Linux distributions are Red Hat Linux, Ubuntu, SUSE Linux and Debian.

Components of Linux Operating System



Linux consists of two levels. At the top is the user, or application, space. This is where the user applications are executed. Below the user space is the kernel space. Here, the Linux kernel exists.

The Linux kernel can be divided into three levels. At the top is the system call interface (or shared system libraries), which implements the basic functions such as read and write. Below the system call interface is the kernel code, which can be more accurately defined as the architecture-independent kernel code. This code is common to all of the processor architectures supported by Linux. Typical components of the architecture-independent kernel are interrupt handlers to service interrupt requests, a scheduler to share processor time among multiple processes, a memory management system to manage process address spaces, and system services such as networking and inter-process communication (IPC). Below this is the architecture-dependent code (also called Board Support Package or Loadable Kernel modules). This code serves as the processor and platform-specific code for the given architecture and enables Linux to run on a vast variety of hardware platforms.

The kernel executes in an elevated system state compared to normal user applications. This includes a protected memory space and full access to the hardware (manipulate memory mappings, set timers, define interrupt vectors, access restricted memory, or halt the processor). This system state and memory space is collectively referred to as kernel-space. User applications execute in user-space. They see a subset of the machine's available resources and are unable to perform certain system functions or directly access hardware. Applications running on the system

communicate with the kernel via system calls. An application typically calls functions in a library for example, the C library that in turn rely on the system call interface to instruct the kernel to carry out tasks on their behalf.

System Calls

A program often needs to read a file, write to some device, or maybe even run another program. All these require operating system intervention. The interface between the operating system and user programs is the set of system calls (open, close, read, fork, execve, etc). Making a system call uses the trap mechanism to switch to a well-defined point in the kernel, running in kernel mode.

To execute a system call usually involves storing the parameters on a specially created stack, storing the number of the system call (each function has a corresponding number), and then issuing a software interrupt instruction to switch control to the operating system operating in kernel mode.

For example, a call to the getpid system call (get process ID) on Intel/Linux systems puts the number 20 into register eax (20 happens to be the number corresponding to the getpid system call) and then executes INT 0x80, which generates a trap.

As with function calls, the kernel needs to be careful to save all user registers and restore everything before it returns. All this ugliness is hidden from the programmer with a set of library routines that correspond to each of the system calls. The library routines save the parameters, issue the trap, and copy the results back onto the user's stack so the system call looks exactly like a regular function call.

Classification of Operating Systems

Operating systems are classified based on various parameters like the number of users that can simultaneously use the system, the number of processes (or tasks that the operating system can perform at a time), number of threads that the operating system can handle etc. Most current operating systems are multi-user, multi-process and multi-threaded.

Single-User and Multi-User OS

Based on number of users that can use the operating system simultaneously operating systems are classified into single-user and multi-user operating systems.

- In a **single-user** operating system only one user can be logged on to the computer at a given point in time. Windows XP, Windows 7 and 8 are examples of single-user operating systems. As against this a multi-user operating system allows many different users to take advantage of the computer's resources simultaneously.
- A **multi-user** operating system must make sure that the requirements of the various users are balanced, and that each of the programs they are using has sufficient and separate resources so that a problem with one user doesn't affect the entire community of users. Windows Server, UNIX and Linux are examples of multi-user operating systems.

Single-Task and Multi-Task OS

On basis of number of tasks the computer can handle at a time the operating systems can be classified into single-task or multi-tasking (also referred to as multi-programming) operating system.

- A **single-task** the name implies, this operating system is designed to manage the computer so that one user can effectively do one thing at a time. The Palm OS for Palm handheld computers is a good example of a modern single-user, single-task operating system.
- **Multi-tasking** operating systems are most commonly used by people on their desktop and laptop computers today. Microsoft's Windows and Apple's Mac OS platforms are both examples of operating systems that will let a single user have several programs in operation at the same time. For example, it's entirely possible for a Windows user to be writing a note in a word processor while downloading a file from the Internet while printing the text of an e-mail message. This is made possible either by using multiple CPUs or timesharing or a mix of both.

Timesharing and Real-time OS

- **Timesharing operating system** – A time sharing operating system uses different algorithms to share the CPU time with more than one process. This allows a computer with only one CPU to give the illusion that it is running more than one program at the same time.
- **Real-time operating system (RTOS)** – Real-time operating systems are used to control machinery, scientific instruments and industrial systems. A very important part of an RTOS is managing the resources of the computer so that a particular operation executes in precisely the same amount of time, every time it occurs.

Components and Functions of an Operating System

The major components of an OS are –

- Information Management
 - File System Management
 - I/O System Management
- Process Management
- Memory Management
- Application Programming Interface
- User Interface

Information Management: Information management refers to the set of services used for storing, retrieving, modifying or removing information on various devices. It consists of two sub-elements –

- File Management
- I/O System Management

Process Management: Every program running on a computer, be it background services or applications, is a process. One CPU can run one process at a time. Process management component is responsible for distributing CPU time as well as other resources across different process running on the computer.

Memory Management: An operating system's memory manager coordinates the use of these various types of memory (processor registers, CPU cache, RAM and disk storage) by tracking which one is available, which is to be allocated or unallocated and how to move data between them.

Application Programming Interface: The Application Programming Interface (APIs) provides a consistent way for applications to deal with the hardware without having to know all the details of the hardware. This relieves application programs from having to manage these details and makes it easier to write applications.

User Interface: User interface provides a link between the user and the computer, which is one of the main functions of an OS. This can be in the form of providing the user with the convenience of giving text commands or else giving the comfort of a graphical interface to interact with the OS.

Information Management

Information management refers to the set of services used for storing, retrieving, modifying or removing information on various devices. It consists of two sub-elements – I/O System Management and File Management.

I/O System Management

Each I/O device has its own characteristics, requiring careful programming. A special program is written for each I/O device. This program is called a device driver. The OS ensures that the correct driver for every device connected to the computer is loaded and coordinates between the driver and the application program wanting to interact with the hardware. In this way the I/O system hides the peculiarities of specific hardware device from the user. The I/O system management is divided in to two parts:

- The Physical Input Output Control System implements device level I/O. The physical IOCS determines the order in which I/O operations should be performed to achieve high device throughput. It implements features like disk scheduler and disk cache.

- The Logical Input Output Control System is responsible for efficient organization and access of data on I/O devices. It provides basic capabilities for file definition, choice of data organization and access methods. It implements features like buffering, blocking of file data and file cache.

File Management

A file is a collection of related information defined by its creator. The file management component manages the organization of information in terms of directories (folders) and files and allocation and de-allocating the sectors to various files.

File System

A file system is a method for storing and organizing computer files and the data they contain to make it easy to find and access them. File systems may use a data storage device such as a hard disk or CD-ROM and involve maintaining the physical location of the files.

It consists of the following major components –

- Disk Management
- Naming
- Protection
- Reliability

Disk Management: Most file systems make use of an underlying data storage device that offers access to an array of fixed-size blocks, sometimes called sectors, generally a power of 2 in size (512 bytes or 1, 2, or 4 KB are most common). The file system software is responsible for organizing these sectors into files and folders, and keeping track of which sectors belong to which file and which are not being used. Most file systems address data in fixed-sized units called “clusters” or “blocks” which contain a certain number of disk sectors (usually 1-64). This is the smallest logical amount of disk space that can be allocated to hold a file.

Naming: It is difficult for users to referring files by blocks and sectors. Naming provides functionality to assign names to files and directories thus making it easy to refer to files and directories.

Protection: Protection component is responsible for assigning and enforcing security of files. To provide security to files the OS maintains an access control list which contains information of the permissions given to different users.

Reliability: Reliability protects the loss of information due to system crashes.

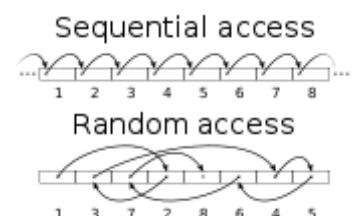
File Operations

- **Create:** Locate empty space on the storage create an entry for the file in the directory.
- **Write:** Add information to a file.
- **Read:** Read information from a file.
- **Rewind:** Set the read/write pointer to the beginning of the file.
- **Delete:** Release the space being used by the file and delete the entry from the directory.

File Access Methods

When we want to use the information stored in files we need to load it from physical storage to computer memory. Two common methods to access files are –

- Sequential Access
- Random (Direct) Access



Sequential access refers to reading or writing data records in sequential order, i.e.

one record after the other. To read record 10, for example, you would first need to read records 1 through 9. This differs from random access, in which you can read and write records in any order. Sequential-access is faster if one always access records in the same order. Random-access is faster if one needs to read or write records in a random order.

Devices can also be classified as sequential access or random access. For example, a tape drive is a sequential-access device because to get to point 'q' on the tape, the drive needs to pass through points 'a' through 'p'. A disk drive, on

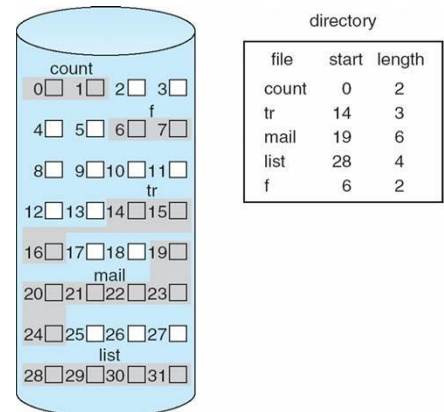
the other hand, is a random-access device because the drive can access any point on the disk without passing through all intervening points.

File Allocation Methods

One main problem in file management is how to allocate space for files so that disk space is utilized effectively and files can be accessed quickly. Three major methods of allocating disk space are contiguous, linked, and indexed. Each method has its advantages and disadvantages.

Contiguous Allocation

The contiguous allocation method requires each file to occupy a set of continuous address on the disk. If the file is n blocks long, and starts at location b , then it occupies blocks $b, b+1, b+2, \dots, b+n-1$. The directory entry for each file indicates the address of the starting block and the length of the area allocated for this file. Accessing a file which has been contiguously allocated is easy. The files system remembers the disk address of the last block and when necessary ready the next block. Accessing block $b+1$ after block b normally requires no head movement except from the last sector of one cylinder to the first sector of the next cylinder, which is only one track. Thus, the number of disk seeks required for accessing contiguous allocated files is minimal.



The difficulty with contiguous allocation is finding space for a new file. If the file to be created is n blocks long, then the OS must search for n free contiguous blocks.

Common strategies used to select a free hole from the set of available holes.

- First-fit: This allocates the first block that is big enough. We can stop searching as soon as we find a large enough free block. This strategy reduces time required find an empty block.
- Best-fit: This allocates the smallest block that is big enough. The entire list must be searched unless the list is kept ordered by size. This strategy reduces wastage on memory space.
- Worst-fit: This allocates the largest block. Again, the entire block list must be searched.

Drawbacks of contiguous allocation

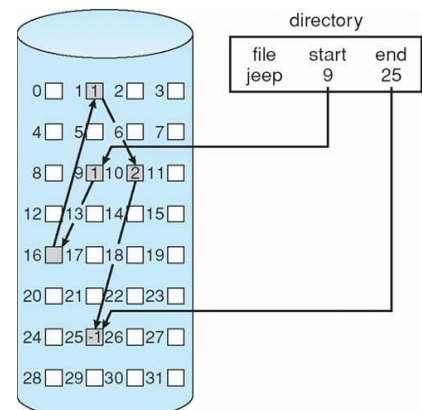
Contiguous allocation suffers from external fragmentation. As files are allocated and deleted, the free disk space is broken into little pieces. External fragmentation exists when enough total disk space exists to satisfy a request, but this space not contiguous; storage is fragmented into a large number of small holes. To tackle the problem of external fragmentation a process called as compaction is used. In this process, all the files are stored to an external storage and copied back to the storage such that all the free space is consolidated at the end of the storage.

Another problem with contiguous allocation is determining how much disk space is needed for a file. When the file is created, the total amount of space it will need must be known and allocated.

Linked allocation

In linked allocation, each file is a linked list of disk blocks. The directory entry contains a pointer to the first and (optionally the last) block of the file. For example, a file of 5 blocks which starts at block 4, might continue at block 7, then block 16, block 10, and finally block 27. Each block contains a pointer to the next block and the last block contains a NULL pointer.

With linked allocation, each directory entry has a pointer to the first disk block of the file. A write to a file finds the first free block and writes to that block. This new block is then linked to the end of the file. To read a file, the pointers are just followed from block to block.



Advantages

- There is no external fragmentation with linked allocation. Any free block can be used to satisfy a request.

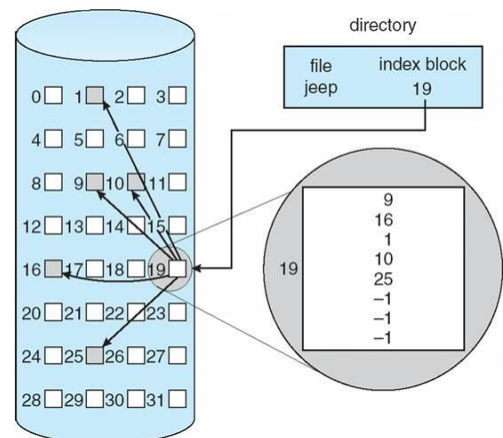
- There is no need to declare the size of a file when that file is created. A file can continue to grow as long as there are free blocks.

Drawbacks

- It is inefficient to support direct-access (i.e. randomly start reading from a point within a file). It is effective only for sequential-access files. To find the i^{th} block of a file, it must start at the beginning of that file and follow the pointers until the i^{th} block is reached.
- Another severe problem is reliability. A bug in OS or disk hardware failure might result in pointers being lost and damaged. This could lead to picking up a wrong pointer and linking it to a free block or into another file.

Indexed allocation

The indexed allocation method is the solution to the problem of both contiguous and linked allocation. This is done by bringing all the pointers together into one location called the index block. In indexed allocation, each file has its own index block, which is an array of addresses. The i^{th} entry in the index block points to the i^{th} sector of the file. The directory contains the address of the index block of a file. To read the i^{th} sector of the file, the pointer in the i^{th} index block entry is read to find the desired sector. To increase reliability two copies of the index can be maintained at different locations of the disk.



Advantages

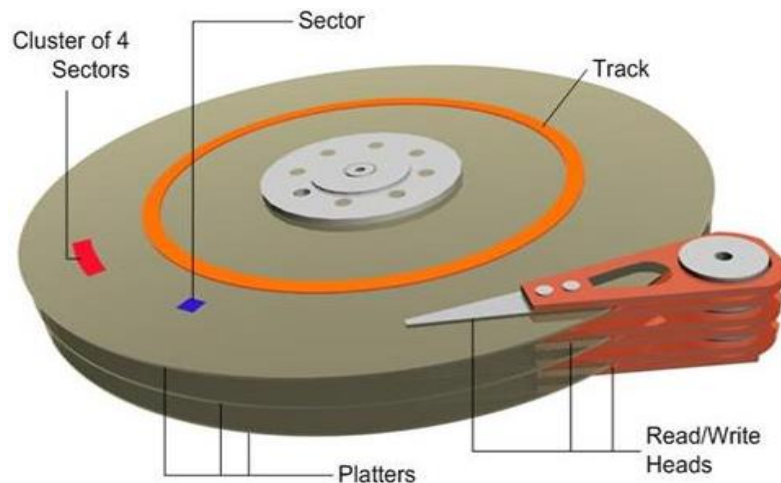
- Indexed allocation supports direct access, without suffering from external fragmentation. Any free block anywhere on the disk may satisfy a request for more space.

Drawbacks

- The index block will occupy some space and thus is an overhead of the method.

Hard Disk Internals

Construction



Hard drives house platters, one or more thin, circular disks, on the surfaces of which are the electronic media that store information.

Each side of each platter has thousands of tracks. The set of tracks with the same diameter from all platter surfaces comprises a cylinder. (For modern drives, the cylinder concept is no longer relevant, and tracks are no longer arranged in concentric circles, yet it is useful to understand the origin of the terms.) Each platter surface has a dedicated read/write head. Tracks are divided into sectors. A sector is the minimum chunk of data that can be read or written to a hard drive. Historically, sector size has been fixed at 512 bytes. Newer drives may offer 1 KB, 2 KB, or 4 KB sectors.

Cluster is a file allocation unit. Cluster size is determined when the partition is formatted by the operating system. For example, if the sectors of a hard drive are 512 bytes, a 4 KB cluster has 8 sectors, and a 64 KB cluster has 128 sectors.

Performance characteristics

The access time or response time of a rotating drive is a measure of the time it takes before the drive can actually transfer data. The factors that control this time on a rotating drive are mostly related to the mechanical nature of the rotating disks and moving heads. It is composed of a few independently measurable elements that are added together to get a single value when evaluating the performance of a storage device. These include:

1. **Seek time:** the seek time measures the time it takes the head assembly on the actuator arm to travel to the track of the disk where the data will be read or written.
2. **Rotational latency:** Rotational latency (sometimes called rotational delay or just latency) is the delay waiting for the rotation of the disk to bring the required disk sector under the read-write head. It depends on the rotational speed of a disk (or spindle motor), measured in rpm (revolutions per minute).
3. **Settle time:** The settle time is the time it takes the heads to settle on the target track and stop vibrating so they do not read or write off track. This time is usually very small, typically less than 0.1 ms.
4. **Transmission time:** It is the time required to activate the read/write head for appropriate surface and read or write the data.

Terminal I/O

Terminal hardware can be considered to be divided into two parts: the keyboard, which is used as an input medium and the video screen which is used as an output medium. These days, if one uses light pens and similar devices, the screen can be used as an input medium also.

The terminal can be a dumb terminal or an intelligent terminal. The dumb terminal has a microprocessor in it on which can run some rudimentary software. It also can have a very limited memory. The dumb terminal is responsible for the basic input and output of characters. Even then, it is called 'dumb' because it does no processing on the input characters. As against this, the intelligent terminal can also carry out some processing (e.g. validation) on the input. This requires a more powerful hardware and software for it.

Memory mapped character oriented alphanumeric terminals

These terminals have a video RAM as shown in the figure. This video RAM is basically the memory that the terminal hardware itself has. The figure shows that the video RAM in our example has 2000 data bytes (0 to 1999) preceded by 2000 attribute bytes (0 to 1999). There is therefore, one attribute byte for each data byte. A typical alphanumeric screen (monochrome IBM-PC) can display 25 lines, each consisting of 80 characters, i.e. $25 \times 80 = 2000$ characters bytes.

Attribute Byte 0	Data Byte 0	Attribute Byte 1	Data Byte 1		

At any time all the 2000 characters stored in the video RAM are displayed on the screen by the video controller using display electronics. Therefore, to have a specific character appear on the screen at a specific position, all one needs to do is to move the ASCII code for that character to the video RAM at the corresponding position with appropriate coordinates. The rest is actually handled by the video controller using display electronics.

The attribute byte tells the video controller how the character is to be displayed. It signifies whether the corresponding data character which is stored next to it in the video RAM is to be displayed bold, underlined, blinking or in reverse video etc. All this information is codified in the 8 bits of the attribute byte. Therefore, when a command is given to a word processor to display a specific character in bold, the word processor instructs the terminal driver to set up the attribute byte for that character appropriately in the video RAM after moving the actual data byte also in the video RAM. The display electronics consults the attribute byte which, in essence, is an instruction to the display electronics to display that character in a specific way.

For the monochrome IBM-PC display, only one attribute (i.e. 8 bits) is sufficient to specify how that character is to be displayed. For bit oriented color graphics terminals, one may require as many as 24 or 32 bits for each byte or even for each bit if a very fine distinction in colors and intensities is needed. This increases the video RAM capacity requirement. It also complicates the video controller as well as the display electronics. But then one gets finer color pictures.

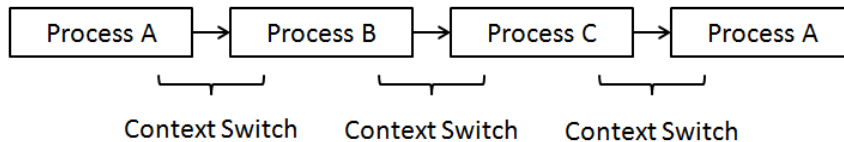
This terminal is called a memory mapped terminal because the video RAM is treated as part of the main memory only. Therefore, for moving any data in or out of the video RAM, ordinary load/store instructions are sufficient. You do not need specific I/O instructions to do this. This simplifies things but then it reduces the memory locations available for other purposes.

When a character is keyed in, the electronics in the keyboard generates an 8 bit ASCII code from the keyboard. This character is stored temporarily in the memory of the terminal itself. Every key depression causes an interrupt to the CPU. The ISR for that terminal picks up that character and moves it into the buffers maintained by the operating system for that terminal. From this buffer the character is sent to the video RAM if the character is also to be displayed (i.e. echoed). Normally, the operating system has one buffer for each terminal. When the user finishes keying in the data, i.e. he keys in the carriage return or the new line, etc., the data stored in the operating system buffer for that terminal is flushed out to the I/O area of the application program which wants that data and to which the terminal is connected (e.g. the data entry program).

Process Management

A process is an instance of a computer program that is being sequentially executed. Every program running on a computer, be it background services or applications, is a process. One CPU can run one process at a time. Process management is an operating system's way of dealing with running multiple processes on a single CPU. Since most computers contain one processor with one core, multitasking is done by simply switching processes quickly (known as context switching). Process management involves computing and distributing CPU time as well as other resources.

Context Switching



Steps involved in context switch:

The state of the first process must be saved somehow, so that, when the scheduler gets back to the execution of the first process, it can restore this state and continue. This is accomplished as follows:

1. All the data required to define the state of the process - all the registers that the process may be using, especially the program counter, plus any other operating system specific data that may be necessary is stored in one data structure, called a switchframe or a process control block (PCB).
2. The PCB for the first process is created and saved.
3. The OS then loads the PCB and context of the second process. In doing so, the program counter from the PCB is loaded, and thus execution can continue in the new process.

In a context switch new processes are chosen from a queue or queues. Process and thread priority can influence which process continues execution, with processes of the highest priority checked first for ready threads to execute.

Important Definitions

Time Slice: The period of time for which a process is allowed to run uninterrupted in a pre-emptive multitasking operating system is called as time slice. Context switch happens at the end of each time slice.

Degree of multiprogramming: The number of process that can be run simultaneously on a computer is known as degree of multiprogramming.

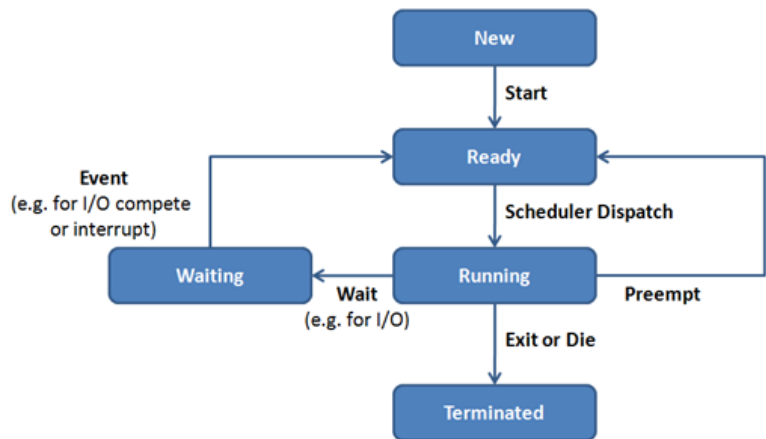
Process States

Created or New

When a process is first created, it occupies the “created” or “new” state. In this state, the process awaits admission to the “ready” state. This admission will be approved or delayed by process scheduler. Typically in most desktop computer systems, this admission will be approved automatically, however for real time operating systems this admission may be delayed.

Ready

A “ready” or “waiting” process has been loaded into main memory and is awaiting execution on a CPU. There may be many “ready” processes at any one point of the systems execution - for example, in a one processor system, only one process can be executing at any one time, and all other “concurrently executing” processes will be waiting for execution.



A ready queue is used in computer scheduling. Modern computers are capable of running many different programs or processes at the same time. However, the CPU is only capable of handling one process at a time. Processes that are ready for the CPU are kept in a queue for “ready” processes. Other processes that are waiting for an event to occur, such as loading information from a hard drive or waiting on an internet connection, are not in the ready queue.

Running or Executing

A “running”, “executing” or “active” process is a process which is currently executing on a CPU. From this state the process may exceed its allocated time slice and be context switched back to “ready” state by the operating system or it may indicate that it has finished and can be terminated or it may block on some needed resource (such as an input/output resource) and be moved to a “blocked” state.

Blocked or Sleeping

Should a process “block” on a resource (such as a file or a device), it will be removed from the CPU (as a blocked process cannot continue execution) and will be in the blocked state. The process will remain “blocked” until its resource becomes available, which can unfortunately lead to deadlock. From the blocked state, the operating system may notify the process of the availability of the resource it is blocking on. Once the operating system is aware that a process is no longer blocking, the process is again “ready” and can from there be dispatched to its “running” state, and from there the process may make use of its newly available resource.

Terminated

A process may be terminated, either from the “running” state by completing its execution or by explicitly being killed. In either of these cases, the process moves to the “terminated” state.

Process Scheduling

Scheduling is a key concept in computer multitasking and multiprocessing operating system design. It refers to the way processes are assigned priorities in a priority queue. This assignment is carried out by software known as a scheduler.

Scheduling Objectives

The scheduler is concerned mainly with –

- CPU utilization – The CPU must be kept as busy as possible.
- Throughput – The number of processes that are completed per time unit. The aim of the scheduler is to maximize the throughput.
- Turnaround time – Total amount of time required to execute a particular process. The scheduler aims to minimize the turnaround time.

- **Waiting time** – The amount of time a process has been waiting in the ready queue. The scheduler tries to minimize the waiting time.
- **Response time** – The time interval between when a request was submitted until the first response is produced. The scheduler tries to reduce the response time.

Scheduling Algorithms

- **Non-preemptive scheduling:** In non-preemptive scheduling, a job is completed before making another scheduling decision. Since the OS waits till a job is completed, one poorly designed program can cause the whole system to hang. Non-preemptive scheduling is generally used in real time operating systems.
- **Preemptive scheduling:** In preemptive scheduling a scheduling decision can be made while the current job is executing. The OS does not wait for a job to complete to schedule another job but provides each process with a fixed amount of processor time (also known as time slice). It is more reliable and makes the OS more responsive.

Process Priority

The concept of priority is important as many processes are competing with each other for the CPU time and memory. The priority can be external or internal.

- **External priority** is specified by the user at the time of initiating a process. This priority can be changed during the time of execution. If the user does not assign a priority then the OS assigns a default priority.
- **Internal priority** is based on the calculation of the current state of the process. This can be based on the estimated time it would take for a process to complete. Based on this, the OS can decide which process it should execute first.
- **Purchased priority** is used in data centers. In this case each user pays for the time used and priority. Higher priority processes are charged a premium.

Process management system calls

System calls associated with process management are:

- Create a new process.
- Suspend or block a process.
- Resume a blocked process.
- Terminate a process.
- Change the priority of a running process.
- Delay a process.

Memory Management

The memory management modules of the operating system are responsible for keeping a track of free and allocated memory locations and making memory available to new process. This is done by partitioning, segmentation, paging and virtual memory.

Partitioning

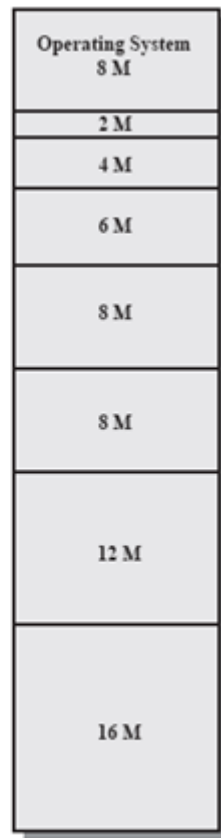
Memory partitioning means dividing the memory into various sections. These sections are called partitions. Operating systems use memory partitioning to allow for multiprogramming and multitasking. OS can implement either fixed or variable partitions. Only one process can be loaded in one partition.

Fixed partitioning

In fixed partitioning the physical memory is split into partitions (equal or variable sized) to which a process may be assigned. Figure shows the two alternatives for fixed partitioning. Once a partition is created its size cannot be changed and hence it is known as fixed partitioning.



(a) Equal-size partitions



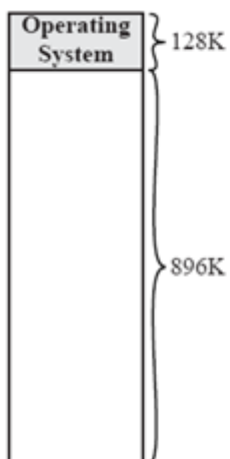
(b) Unequal-size partitions

Drawbacks

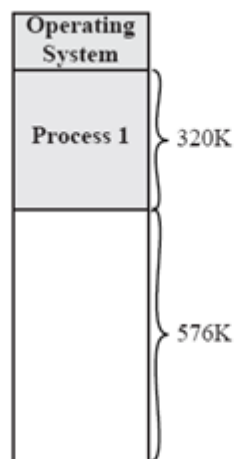
- Program might require more memory than the biggest available partition.
- Internal fragmentation occurs because each partition might not be fully utilized and hence leaving free space in the partition which cannot be used by other programs.

Variable partitioning

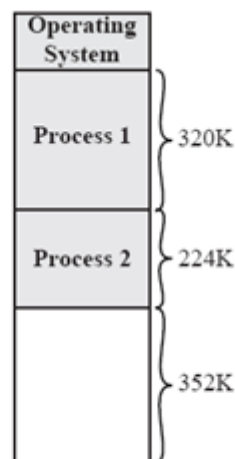
Variable partitioning address the problem of internal fragmentation found in fixed partitioning by allowing partition size to be changed even after creation. Figure below shows the implementation of variable partitions.



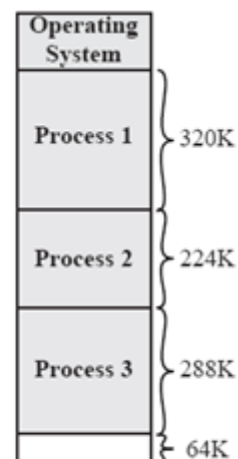
(a)



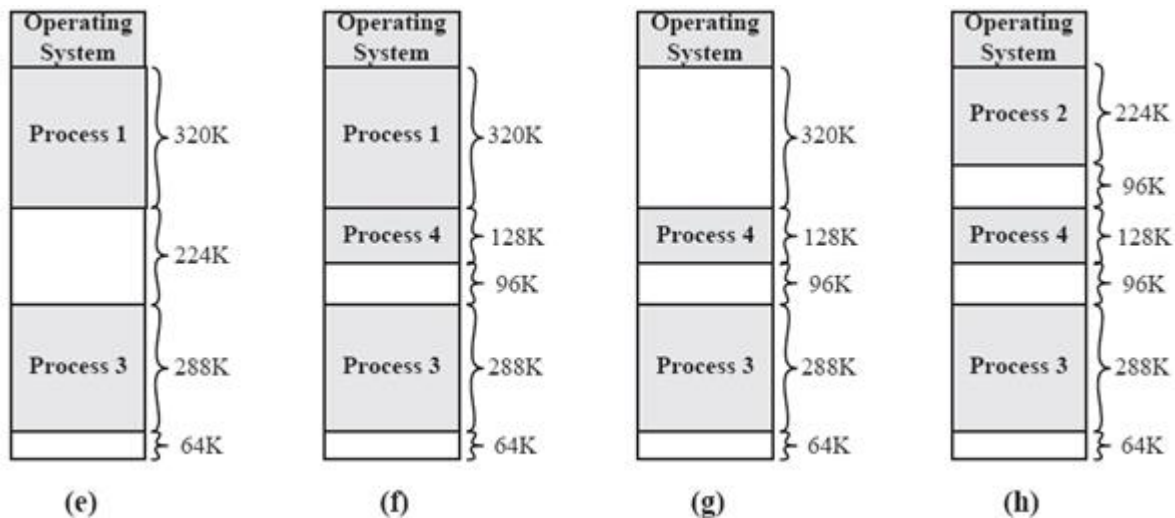
(b)



(c)



(d)

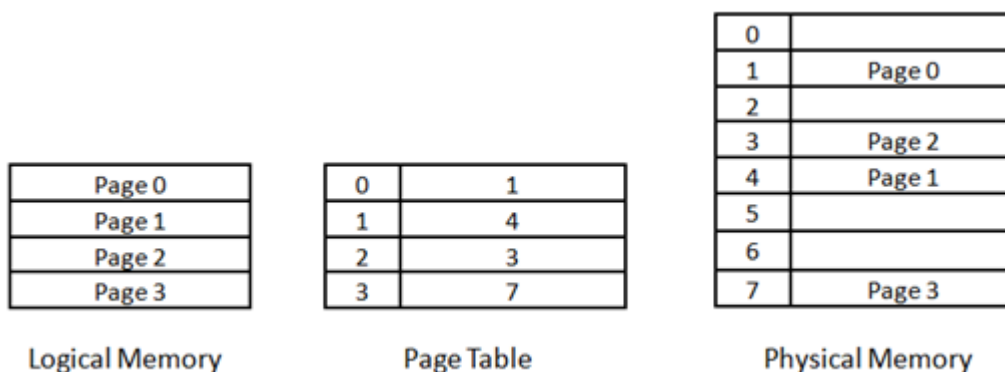


Drawbacks

Variable partitions lead to external fragmentation – blocks of empty space which cannot be allocated to programs. So even if the total available free memory is greater than the requirement of the program since it is not available in one block the program cannot be loaded in memory. This problem is addressed by compaction and paging. Compacting is equivalent to defragmenting the memory to consolidate all free space in one continuous block.

Paging

In paging the physical memory is divided into equal size page frames and the program is divided into equal sized pages (typically 4096 bytes or 4 Kb). The size of the page frame and the page is the same so any page can exactly fit in a page frame and hence can be assigned to any available page frame.



When a program is executed its pages are loaded in to the available memory and the page table is used to translate from user pages to memory frames. If the program requires more memory than that is available then only part of the program is loaded in the memory. When the program tries to access pages that are not currently mapped to physical memory (RAM) a page fault occurs. In response to a page faulty the operating system takes control and handles the page fault, in a manner invisible to the program. The operating system does the following steps –

1. Determine the location of the data in auxiliary storage.
2. Obtain an empty page frame in RAM to use as a container for the data.
3. Load the requested data into the available page frame.
4. Update the Page Table to show the new data.
5. Return control to the program, transparently retrying the instruction that caused the page fault.

Important definitions

Working set: At any time, a process has a number of pages in the physical memory. These set of pages is known as working set.

Dirty page: Page that has been modified in memory for writing to disk, are marked “dirty” and is called a dirty page. A dirty page has to be flushed to disk before it can be freed. When a file write occurs, the page backing the particular block is looked up. If it is already found in cache, the write is done to that page in memory. Otherwise, the page(s) are fetched from disk and requested modifications are done and the page(s) are marked as dirty.

Page replacement policy: When all the available page frames are occupied and a page faulty occurs, the operating system needs to overwrite an existing page in the memory. The operating system chooses this page by using the page replacement policy.

Locality of reference: Locality of reference provides a basis to forecast which page is most likely to be referenced in the future based on various parameters. This helps in identifying which pages should be preloaded and which pages should be unloaded from the physical memory. There are two types of locality of reference:

- **Spatial:** Based on the physical location of the page in memory. Generally pages around a page that was used would belong to the same program and hence would be required and should not be removed from the memory.
- **Temporal:** Based on time when the page was last accessed. While removing pages from the memory it is preferred to remove the page which has not been used for a long time.

Demand paging: In a system that uses demand paging, the operating system copies a disk page into physical memory only if an attempt is made to access it (i.e., if a page fault occurs). It follows that a process begins execution with none of its pages in physical memory, and many page faults will occur until most of a process's working set of pages is located in physical memory. Advantages include faster startup and less use of memory as only pages required are loaded in the memory. Disadvantages include increased number of page faults and more complex page replacement algorithms.

Virtual Memory

Virtual memory is a memory management technique that is implemented using both hardware and software. It maps memory addresses used by a program, called virtual addresses, into physical addresses in computer memory. Main storage as seen by a process or task appears as a contiguous address space. The operating system manages virtual address spaces and the assignment of real memory to virtual memory. Address translation hardware in the CPU, often referred to as a memory management unit or MMU, automatically translates virtual addresses to physical addresses. Software within the operating system may extend these capabilities to provide a virtual address space that can exceed the capacity of real memory and thus reference more memory than is physically present in the computer.

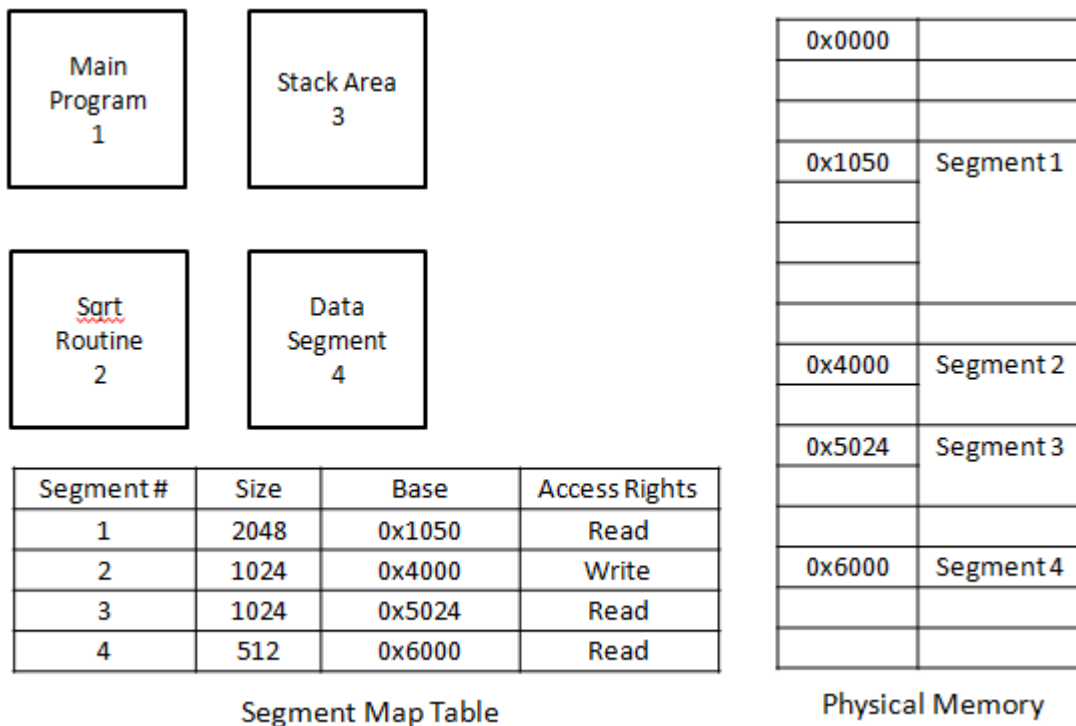
The primary benefits of virtual memory include freeing applications from having to manage a shared memory space, increased security due to memory isolation, and being able to conceptually use more memory than might be physically available, using the technique of paging.

Systems that use virtual memory make programming of large applications easier and use physical memory (e.g. RAM) more efficiently.

Segmentation

Memory segmentation is one of the most common ways to achieve memory protection. In a computer system using segmentation, an instruction operand that refers to a memory location includes a value that identifies a segment and an offset within that segment. A segment has a set of permissions, and a length, associated with it. If the currently running process is allowed by the permissions to make the type of reference to memory that it is attempting to make, and the offset within the segment is within the range specified by the length of the segment, the reference is permitted; otherwise, a hardware exception is raised.

Code, Data and Stack segments are the most common segments created for any program executing on 80186 and above microprocessors. This isolates the three sections of the program and thus increasing security.



Memory map of a single user operating system

Free memory
User programs Program A Program B Program C Program D
Command interpreter COMMAND.COM
Resident programs and Device drivers
DOS kernel (IO.SYS and MSDOS.SYS)
BIOS data
Interrupt vector table (IVT)

The memory map for single user operating system like MS-DOS is shown in the table above. It consists of the following areas -

1. The interrupt vector table is located in the low address area and contains the addresses of interrupt handlers.
2. BIOS is nothing but Basic Input/Output System. BIOS is a collection of programs that is stored in the ROM. It can also contain information about the system, such as the type of hardware present.
3. The DOS kernel area contains programs from IO.SYS and MSDOS.SYS files. IO.SYS enables the DOS programs to use the keyboard, the display, printer and other I/O devices. This block also contains the buffer area for disk input/output operations and the area for file system control tables - FCB's (File Control Block). Resident programs or drivers which control the hardware are located at the top of this area.
4. The next section of memory contains the resident part of the command interpreter (COMMAND.COM). This DOS component is always stored in RAM and is the last DOS component in the low-addressed memory. The COMMAND.COM processes the DOS commands as they are typed from the keyboard. The section above this is reserved for user programs.
5. The space above the end of the resident part of the command processor is not occupied by DOS routines and can be used by applications. This area is referred to as user program space and is available for user programs.

Security and Protection

Viruses, Worms and Trojan Horses

Viruses, worms and Trojan Horses are all malicious programs that can cause damage to your computer.

Virus

A computer virus attaches itself to a program or file enabling it to spread from one computer to another, leaving infections as it travels. Almost all viruses are attached to an executable file, which means the virus may exist in a computer but it actually cannot infect the computer unless the user runs or opens the malicious program. Viruses cannot be spread without a human action (such as running an infected program).

People continue the spread of a computer virus, mostly unknowingly, by sharing infecting files or sending e-mails with viruses as attachments in the e-mail.

Viruses can be classified as under:

- Boot sector virus: Virus that infects the boot sector of a hard disk.
- Memory resident virus: Virus that stays in memory after it executes and after its host program is terminated. In contrast, non-memory-resident viruses only are activated when an infected application runs.
- Command processor virus: Virus that attaches itself to the command processor module of DOS.
- Macro virus: Virus that infects macros in Microsoft Office documents.
- General purpose virus.

Worms

Worms spread from computer to computer, but unlike a virus, it has the capability to travel without any human action. The biggest danger with a worm is its capability to replicate itself. Due to the copying nature of a worm and its capability to travel across networks the end result in most cases is that the worm consumes too much system memory (or network bandwidth), causing web servers, network servers and individual computers to stop responding.

Trojan horse

A Trojan horse is full of as much trickery as the mythological Trojan horse it was named after. The Trojan horse, at first glance will appear to be useful software but will actually do damage once installed or run on your computer. Those on the receiving end of a Trojan horse are usually tricked into opening them because they appear to be receiving legitimate software or files from a legitimate source. Unlike viruses and worms, Trojans do not reproduce by infecting other files nor do they self-replicate.

Combating Viruses, Worms and Trojan horse

Prevention mechanisms

- Use genuine software. There is a high probability that pirated software might have been tampered with and might contain a virus or Trojan horse.
- Keep the operating system and other software up-to-date. This ensures that any vulnerability found software is fixed and worms cannot make use of them.
- Install anti-virus software on the system and ensure that the updates are downloaded frequently to ensure the software has the latest fixes for new viruses, worms, and Trojan horses.
- In a computer is connected to a network a firewall should be installed on the computer. A firewall is a system that prevents unauthorized use and access to your computer.

Virus detection and removal

Viruses reproduce by infecting “host applications,” meaning that they copy a portion of executable code into an existing program. So to ensure that they work as planned; viruses are programmed to not infect the same file multiple times. To do so, they include a series of bytes in the infected application to check if has already been infected: This is called a virus signature. Antivirus programs rely on this signature, which is unique to each virus, in order to detect them. This method is called signature scanning. It is the oldest method used by antivirus software. This method is only reliable if the antivirus program's virus database is up-to-date and includes signatures for all known viruses. However, this method cannot detect viruses which have not been archived by the publishers of the antivirus software.

Some antivirus programs use an integrity checker to tell if the folders have been changed. The integrity checker builds a database containing information on the executable files on the system (date modified, file size, and possibly a checksum). When an executable file's characteristics change, the antivirus program warns the machine's user.

The heuristic method involves analyzing the behavior of applications in order to detect activity similar to that of a known virus. This kind of antivirus program can therefore detect viruses even when the antivirus database has not been updated. On the other hand, they are prone to triggering false alarms.

Data Structure

Probable Marks: 15 marks

Scope of syllabus:

- Introduction to data structures
- Data structure operations
- Control structures
- Arrays - Introduction, Common operations (insert, delete, search and sort) and Representing arrays in memory
- Linked List - Introduction, Types of linked list and Representing linked list in memory
- Trees and Binary Trees - Introduction, Definition of terms (node, leaf, parent, root, depth etc.), Binary tree and Representing trees in memory

Introduction

A data structure in computer science is a way of storing data in a computer so that it can be used efficiently.

Common terms used in data structures

- **Entity:** An entity is something that has certain attributes or properties which may be assigned values. Values can be numeric or non-numeric. Example - an employee of an organization with attributes name and age.
- **Information:** Information means meaningful data.
- **Field:** Field is a single elementary unit of information representing an attribute of an entity. Example - the name or the age of an employee.
- **Record:** A collection of field values of a given entity is known as a record. Example - Information about one employee.
- **File:** A collection of records of entities is called a file. Example - Information about all the employees of the company.

Data Structure Operations

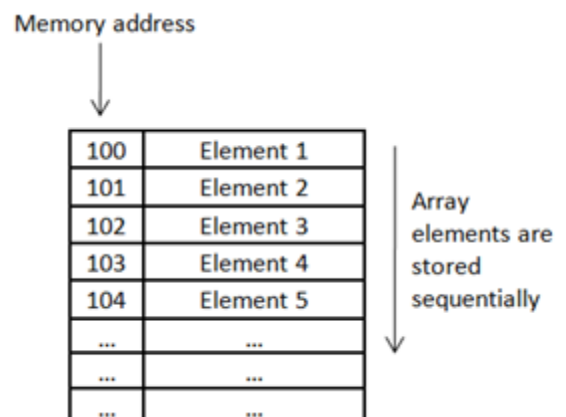
- **Traversing:** Traversing means accessing each record only once so that it can be processed.
- **Searching:** Searching means finding the location of one or more records which meet the criteria from a given set of records.
- **Inserting:** Inserting means adding a new record to the structure.
- **Deleting:** Deleting means removing the record from the structure.
- **Sorting:** Sorting means arranging the records in some logical order. Example - arranging names alphabetically or arranging numbers in ascending order.
- **Merging:** Merging means combining the records in to different files into a single file.

Array

A linear array is a list of finite number (n) of homogenous data elements (i.e. the data elements are of the same type). The elements of the array are referenced respectively by an index set consisting of ' n ' consecutive numbers. The elements of the array are stored in successive memory locations. The number ' n ' of elements is called the length or size of the array.

Representation of linear arrays in memory

The elements of linear array are stored in successive memory cells. The number of memory cells required depends on the type of data elements. Computer keeps track of the address of the first element of the array. This is known as the base address of the array. Using this base address the computer can calculate the address of any element.



Traversing linear arrays

Let LB be the lower bound of the array (starting index) and UB the upper bound of the array (upper index)

1. Initialize counter C to the starting index (LB).
2. If $C \leq UB$ go to step 3, else go to step 6.
3. Read the C^{th} element of the array.
4. Increment the counter by 1 (Set $C = C + 1$).
5. Go to step 2.
6. Exit.

Inserting and deleting in arrays

Inserting and deleting elements at the end of the array is easy. To insert anywhere else, we need to shift the elements occurring after that location down to create space for the new element. Similarly if we delete an element in anywhere in the array we need to move all the elements coming after that element up.

Insert an element in an array

1	14
2	50
3	73
4	9
5	24
6	3
7	92
8	-3

Original Array

1	14
2	50
3	73
4	
5	9
6	24
7	3
8	92
9	-3

Elements shifted to create place

1	14
2	50
3	73
4	32
5	9
6	24
7	3
8	92
9	-3

Array after Insertion

Algorithm to insert an element

Let LB be the lower bound of the array AR, UB the upper bound and P be the position where we want to add an element.

1. Initialize counter $C = UB$.
2. If $C \geq P$ go to step 3, else go to step 6
3. Copy the value of the element stored in location C to location $C+1$ ($AR[C+1] = AR[C]$).
4. Decrement counter by 1 (Set $C = C - 1$).
5. Go to step 2.
6. Store the new element in the array at location P.
7. Exit.

Delete an element from an array

1	14
2	50
3	73
4	9
5	24
6	3
7	92
8	-3

Original Array

1	14
2	50
3	73
4	9
5	
6	3
7	92
8	-3

5th Element deleted – leaving an empty location

1	14
2	50
3	73
4	9
5	3
6	92
7	-3
8	

Array after Deletion

Algorithm to delete an element

Let LB be the lower bound of the array AR, UB the upper bound and P be the position where we want to delete an element from.

1. Initialize counter $C = P$.
2. If $C \leq UB$ go to step 3 else go to step 6.
3. Copy the value of the element stored in location $C+1$ to location C ($AR[C] = AR[C + 1]$).
4. Increment counter by 1 (Set $C = C + 1$).
5. Go to step 2.
6. Set the value of element stored at the UB to NULL ($AR[UB] = \text{NULL}$).
7. Exit.

Sorting arrays

Bubble Sort algorithm

Bubble sort is a simple sorting algorithm. It works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items of the list and swapping them if they are in the wrong order. This passing through the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm gets its name from the way smaller elements “bubble” to the top of the list.

In the worst case bubble sort requires n^2 comparisons, where n is the number of items being sorted. Hence the time required for bubble sort algorithm to sort an array is proportional to n^2 .

Let LB be the lower bound of the array AR and UB the upper bound.

1. Initialize counter $I = LB$
2. Set value of counter $J = 1$
3. If $I < (UB - 1)$ go to step 4 else go to step 10.
4. If $J < (UB - I)$ go to step 5 else go to step 8.
5. Compare the value of element at J^{th} and $(J+1)^{\text{th}}$ locations. If value of element at J^{th} location $>$ that at $(J+1)^{\text{th}}$ location then swap the elements.
6. Increment value of J by 1
7. Go to step 4.
8. Increment step I by 1
9. Go to step 3.
10. Exit

Step-by-step example

Let us take the array of numbers “5 1 4 2 8”, and sort the array from lowest number to greatest number using bubble sort algorithm. In each step, elements written in bold are being compared.

First Pass:

(5 1 4 2 8) \rightarrow (**1** 5 4 2 8), Here, algorithm compares the first two elements, and swaps them.

(1 5 4 2 8) \rightarrow (1 4 5 2 8), Swap since $5 > 4$

(1 4 5 2 8) \rightarrow (1 4 2 5 8), Swap since $5 > 2$

(1 4 2 5 8) \rightarrow (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) \rightarrow (1 4 2 5 8)

(1 4 2 5 8) \rightarrow (1 2 4 5 8)

(1 2 4 5 8) \rightarrow (1 2 4 5 8)

(1 2 4 5 8) \rightarrow (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) \rightarrow (1 2 4 5 8)

(1 2 4 5 8) \rightarrow (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)

Finally, the array is sorted, and the algorithm can terminate.

Searching in arrays

Linear Search

Linear search algorithm is one of the simplest algorithms to search for an element. In linear search the array is traversed and the element is compared with each element of the array. In the worst case scenario that number that we are searching for could be at the end of the array and thus require n comparisons (where n is the length of the array) before the number is found or before we can confidently say that the number is not part of the array.

Let LB be the lower bound of the array AR, UB the upper bound of the array and E the element to be found.

1. Initialize counter $C = LB$.
2. If $C \leq UB$ go to step 3, else go to step 5.
3. Compare the C^{th} element of the array with E. If the values match then display the number was found at the C^{th} location and go to step 5.
4. Increment C by 1 (Set $C = C + 1$).
5. Go to step 2.
6. Exit.

Binary Search

Binary search is an example of divide and rule algorithm. In this we use the knowledge that the array is sorted to decrease the number of comparisons that we need to do before we find the number or can confidently say that the number does not exist in the array. In the worst case scenario binary search requires $\log_2 n$ comparisons.

Let LB be the lower bound of the array AR sorted in ascending order, UB the upper bound of the array and E the element to be found.

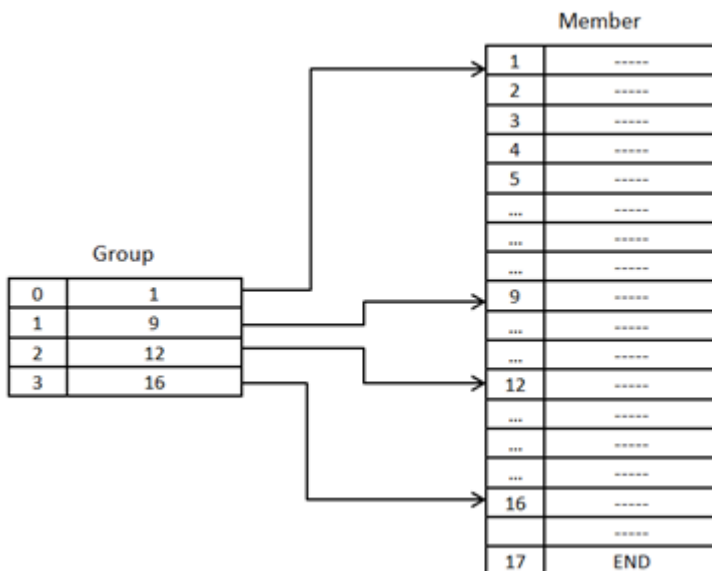
Consider two variables BEG and END. BEG will stand for the beginning (lower bound) and END for the end (upper bound) for the current range of locations that we are working with in the program.

1. Set $BEG = LB$ and $END = UB$.
2. While BEG is not equal to END repeat step 3 else go to step 4.
3. Compare the number stored at the middle of the current range of locations $((BEG + END)/2)$ to E.
 - a. If the values match then display the number was found at $((BEG + END)/2)^{\text{th}}$ and go to step 4.
 - b. If number stored at the middle of the current range of locations $((BEG + END)/2)$ is greater than the element E then the number can be in the first half of the current range hence set $END = ((BEG + END)/2)$
 - c. If number stored at the middle of the current range of locations $((BEG + END)/2)$ is less than the element E then the number can be in the second half of the current range hence set $BEG = ((BEG + END)/2)$
4. Exit

Pointer Arrays

A pointer is an object whose value refers directly to (or “points to”) another value stored elsewhere in the computer memory using its address. An array is called a pointer array if each element of the array is a pointer. We can use pointer arrays to point to other arrays.

Example – Suppose we have 4 groups. Each group consists of list of members. This list is to be stored in memory then the most efficient method is to form 2 arrays. One is a list of all members one after the other and another pointer array group containing the starting location of different groups.



Record

Record is a collection of fields of an entity. A record may contain non homogenous data (i.e. data items in a record may have different data types).

There are two ways one can store records in memory –

1. Using one linear array each to store values for each field. The figure below shows three arrays being used to store the name, age and city of a person. As it can be seen in the figure the nth element of each array represents on field for the nth record.

Name		Age	City
1	Abhishek	14	Mumbai
2	Ravi	50	Delhi
3	Jatin	25	Mumbai
4	Sanjay	32	Chennai

2. In case the space taken by each field is known then we can store all the values one after another and use the information about the size of each field to segregate these values in the program. C/C++ uses this method to store data of type Structures and Unions.

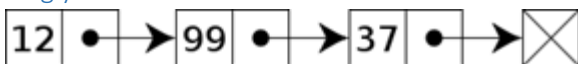
Linked List

A linked list consists of a sequence of nodes, each containing data fields and one or two references (“links”) pointing to the next and/or previous nodes. The main advantage of a linked list over a conventional array is that the order of the linked items may be different from the order that the data items are stored in memory or on disk, allowing the list of items to be traversed in a different order. A linked list is a self-referential data type because it contains a pointer or link to another data of the same type.

Types of Linked List

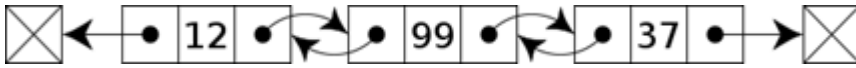
Several different types of linked list exist: singly-linked lists, doubly-linked lists, and circularly-linked lists.

Singly-linked list



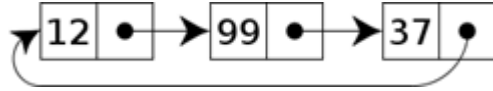
The simplest kind of linked list is a singly-linked list, which has one link per node. This link points to the next node in the list, or to a null value or empty list if it is the final node.

Double-linked list



A more sophisticated kind of linked list is a doubly-linked list or two-way linked list. Each node has two links: one points to the previous node, or points to a null value; and one points to the next, or points to a null value.

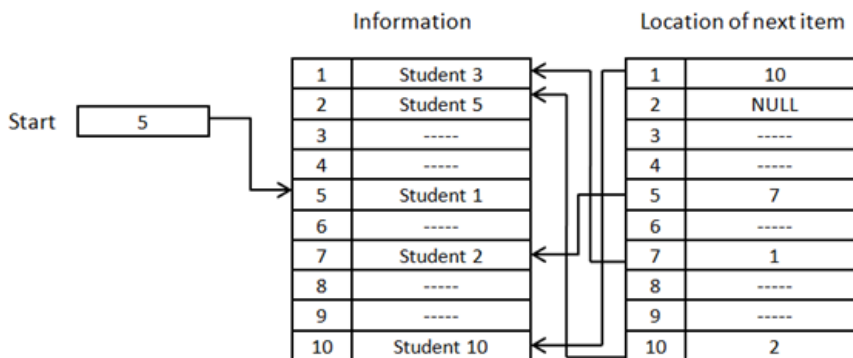
Circularly-linked list



In a circularly-linked list, the first and final nodes are linked together. This can be done for both singly and doubly linked lists. To traverse a circular linked list, one can begin at any node and follow the list in either direction till one returns to the original node. Thus, a circularly-linked list can be said to have no beginning or end.

Storing linked list in memory

A singly-linked list can be represented in memory using two linear arrays. One array contains the information stored in node and the second the address (location) of the next node.

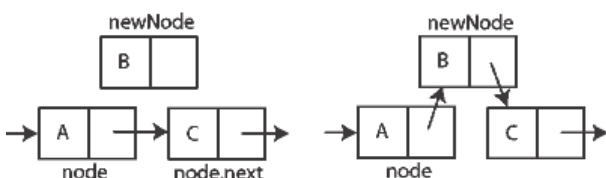


In the figure above, the first element of the linked list is stored in location 5. The array having the information contains the value "Student 1" at the 5 location while the address array contains the value 7 which indicates that the next item of in the linked list is stored at memory location 7. The address array for the last item ("Student 5") points to NULL.

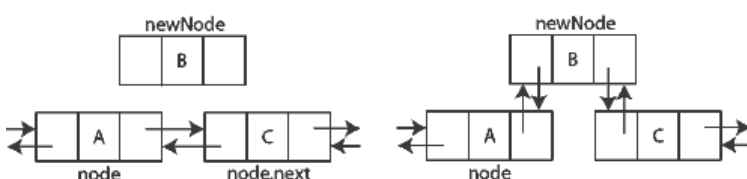
Inserting a new node in a linked list

To insert a node in the linked list, the pointer value of the previous node is assigned to the pointer of the new node to be inserted and then the pointer of the previous node is updated to point to the node being inserted.

Insert a new node in a singly-linked list:

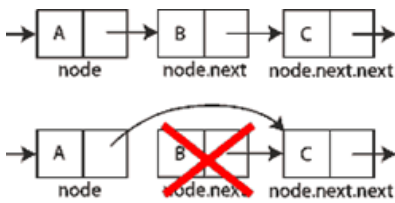


Insert a new node in a doubly-linked list:



Deleting a node from a linked list

To delete a node from the linked list, one needs to copy the value of that node's pointer to that of the previous node and set the current node to null.

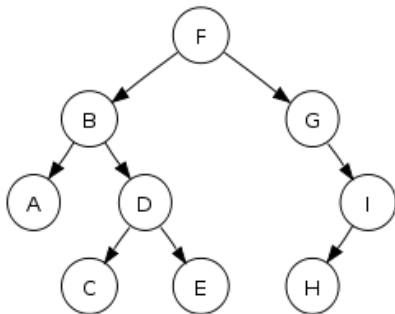


Traversing a linked list

Traversal of a singly-linked list is simple, beginning at the first node and following each next link until we come to the end. In a double-linked list you can traverse in both directions till you reach the end.

Tree

A tree is a nonlinear data structure. It is used to represent data containing a hierarchical relationship between elements.



Terminology

A node may contain a value or a condition. Each node in a tree has zero or more child nodes, which are below it in the tree (by convention, trees grow down, not up as they do in nature). A node that has a child is called the child's parent node (or ancestor node, or superior). A node has at most one parent. Nodes that share the same parent are called as siblings.

The height of a node is the length of the longest downward path to a leaf from that node. A tree with only 1 node (the root) has a height of zero. The height of the root node is the height of the tree. The depth of a node is the length of the path from the root to the node (i.e., its root path). The root node is at depth zero.

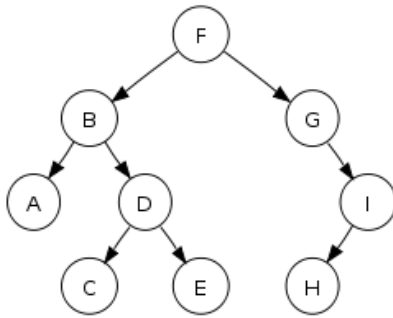
The topmost node in a tree is called the root node. Being the topmost node, the root node will not have parents. It is the node at which operations on the tree commonly begin. All other nodes can be reached from it by following edges or links. Every node in a tree can be seen as the root node of the sub tree rooted at that node.

Nodes at the bottommost level of the tree are called leaf nodes. Since they are at the bottommost level, they do not have any children. A node with no children is called a leaf node. They are also referred to as terminal node.

An internal node or inner node is any node of a tree that has child nodes and is thus not a leaf node. An intermediate node between the root and the leaf nodes is called an internal node.

A sub tree is a portion of a tree data structure that can be viewed as a complete tree in itself. Any node in a tree T, together with all the nodes below it, comprise a sub tree of T.

Binary Trees



A binary tree is a tree data structure in which each node has at most two children. Typically the child nodes are called left and right. Since each node of a binary tree can have at most 2 child nodes, there can be a maximum of $2^d - 1$ nodes where d is the depth of the tree with root having the depth as 1.

Representing binary trees in memory

Binary trees can be represented in memory either by linked representation or in a single array using sequential representation.

Linked representation

Linked representation uses three parallel arrays, INFO, LEFT and RIGHT and a pointer variable ROOT. Each node N of T will correspond to a location K such that –

- INFO[K] contains the data at node N
- LEFT[K] contains the location of left child node N
- RIGHT[K] contains the location of right child node N
- ROOT will contain the location of root R of T

	INFO	LEFT	RIGHT
1	A	NULL	NULL
2	B	1	10
3	H	NULL	NULL
4	G	NULL	6
5	F	2	7
6	I	3	NULL
7	G	3	4
8	C	NULL	NULL
9	E	NULL	NULL
10	D	8	9

Linked representation

	TREE
1	F
2	B
3	G
4	A
5	D
6	NULL
7	I
8	NULL
9	NULL
10	C
11	E
12	NULL
13	NULL
14	H
15	NULL

Sequential representation

Sequential representation

In sequential representation only one linear array is used. The root R is stored at TREE[1]. If the node occupies TREE[K], then its left child is stored at in TREE[K*2] and right child is stored in TREE[K*2 + 1].

HTML

Probable Marks: 15 marks

Scope of syllabus:

- Introduction to HTML - Advantages and disadvantages
- Study of tags - <html>, <head>, <title>, <body>, <p>,
, , , <pre>, <marquee>
- Font styles - , <i>, <u>, <big>, <small>, <sub>, <sup>,
- Working with images - embedding image in a web page, tag and attributes
- Working with tables - <table>, <caption>, <tr>, <td>, <th>
- VBScript - for... next, if... then, msgbox, dim and set

Introduction

HTML stands for Hypertext Markup Language. Hypertext is text, displayed on a computer, with references (hyperlinks) to other text that the reader can immediately follow, usually by a mouse click or key press sequence. A markup language is a set of annotations (symbols) to text that describe how it is to be structured, laid out, or formatted. Each symbol used for markup in HTML is a command that tells the browser how the text following it must be displayed. HTML, thus provides a means to describe the structure of text-based information in a document – by denoting certain text as links, headings, paragraphs, lists, and so on – and to supplement that text with interactive forms, embedded images, and other objects. HTML is written in the form of tags, surrounded by angle brackets. It is commonly used to create web pages. HTML was originally developed by Tim Berners-Lee.

Advantages of HTML

- A HTML document can be created and viewed on any hardware (Mainframe or Desktop) or software platform (Windows or Linux).
- For creating a HTML document, one requires only a text editor (like notepad or WordPad). No special software is required.
- For viewing a HTML document, only a browser is required. Many browsers are freely available for different platforms.
- It is easy to find error in HTML.
- HTML is free. One does not require buying any licenses to use HTML.
- Learning HTML is easier than any Programming Language.

Disadvantages / Limitations of HTML

- HTML is not a programming language. As a result no calculations can be done using HTML. To implement any kind of programming language HTML has to depend of other languages like vbscript and JavaScript.
- HTML can be used to create only static web pages. To create interactive and dynamic web pages we need to use other technologies like ASP.
- Since HTML is not a standard but a set of recommendations, different web browsers implement it differently. Hence the same page might look or behave differently in different web browsers.
- Unlike programming languages, it is not possible to prevent other people from seeing and copying the HTML code for your web page.
- Since HTML pages reside on a web server, every time anyone wants to open a HTML page one need to request the page from the web server which takes some time. Also in case the internet is not working then the web page is not available.

Basic structure of a HTML document

Elements, Tags and Attributes

HTML documents are text files made up of HTML elements. HTML elements are defined using HTML tags. The element's name appears in the start tag (written <element_name>) and the end tag (written </element_name>). Start and end tags are also called as opening and closing tags. Some HTML element types allow authors to omit end tags and are called as unpaired tags. HTML tags are not case sensitive. Elements may have associated properties, called attributes, which may have values. Attribute/value pairs appear before the final ">" of an element's start tag.

Any number of (legal) attribute value pairs, separated by spaces, may appear in an element's start tag. They may appear in any order. Attributes names are not case sensitive.

Basic structure of a HTML document

A simple HTML document can be defined as below and makes use of the HTML, HEAD, TITLE and the BODY elements.

```
<html>
<head>
<title>My first HTML document</title>
</head>
<body>
<p>Hello world!</p>
</body>
</html>
```

HTML Element

<HTML> tag declares that the text that follows defines an HTML web page that can be viewed in a web browser. The closing tag </HTML> ends the page.

HEAD Element

All data in the header section of an HTML document is considered “meta-data”, meaning “data about data”. The HEAD element contains information about the current document, such as its title, keywords that may be useful to search engines, and other data that is not considered document content. Browsers do not generally render elements that appear in the HEAD as content.

TITLE Element

The text appearing in the TITLE element is displayed in the title bar of the browser. This text is used to identify the contents of the web page. The text should be descriptive as it is frequently used by searching engines to name your web page.

BODY Element

The actual contents of the web page that will be displayed in the web browser are contained within the body element. The body section starts with <BODY> tag and ends with the </BODY> tag.

The body tag has several optional attributes –

Attribute	Acceptable Values	Description
bgcolor	16 defined color codes like red, blue, green, yellow, black, white etc. Color value using RGB code e.g. #DDEEFF.	Set the background color of the web page.
background	Address of the file to use as background picture. e.g. “bgpicture.png”	Set the picture that will be used as the background of the web page.
text	Same as bgcolor	Set the color of the text on the web page.
link	Same as bgcolor	Set the color of a hyperlink on the web page.
vlink	Same as bgcolor	Set the color of a visited hyperlink on the web page.
alink	Same as bgcolor	Set the color of an active (selected) hyperlink on the web page.

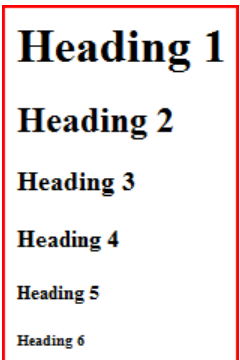
Working with Text

Headings

- A heading element briefly describes the topic of the section it introduces.
- There are six levels of headings in HTML with H1 as the most important and H6 as the least. The <h1>, <h2>, <h3>... <h6> tags are used to represent the heading elements.
- Web browsers usually render more important headings in larger fonts than less important ones.
- HTML headings can be used in any order.
- It is not necessary to start with <h1> as the first heading.

Attribute	Acceptable Values	Description
align	left right center justify	Set the alignment of text

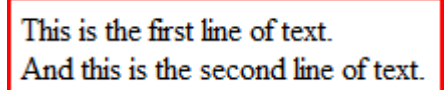
Example:

Code: <pre><h1>Heading 1</h1> <h2>Heading 2</h2> <h3>Heading 3</h3> <h4>Heading 4</h4> <h5>Heading 5</h5> <h6>Heading 6</h6></pre>	Output: 
--	---

Line break: BR Element

- The
 tag forcibly breaks (ends) the current line of text.
- It is an unpaired tag.

Example:

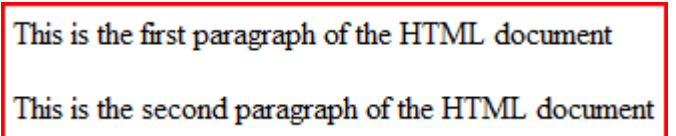
Code: This is the first line of text. And this is the second line of text.	Output: 
--	--

Paragraph: P Element

- The P element represents a paragraph.
- The start tag <P> is required but the end tag </P> is optional.

Attribute	Acceptable Values	Description
align	left right center justify	Set the alignment of text

Example:

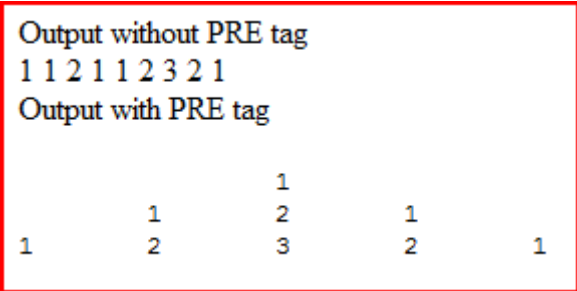
Code: <pre><p>This is the first paragraph.</p> <p>And this is the second paragraph.</p></pre>	Output: 
---	--

Preformatted Text: PRE Element

The PRE element tells web browsers that the enclosed text is “preformatted”. When handling preformatted text, browsers:

- May leave white space intact.
- May disable automatic word wrap.

Example:

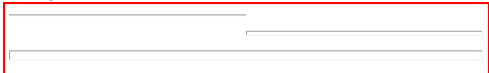
Code: Output without PRE tag <pre> 1 1 2 1 1 2 3 2 1 Output with PRE tag <pre> 1 1 2 1 1 2 3 2 1 </pre></pre>	Output: 
---	--

HR Element

- A web page can be divided into separate sections by using horizontal rule, <hr> tag.
- It is also called as horizontal line.
- This tag is mostly used for decorative purposes.

Attribute	Acceptable Values	Description
align	left right center justify	Set the alignment of the horizontal line.
size	number (in pixels)	Set the thickness of the line in pixels.
width	number (in percentage or pixels)	Set the length of the line.

Example:

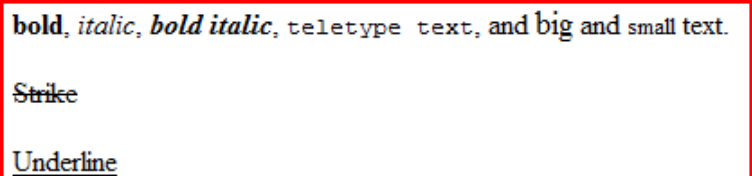
Code: <pre><hr width="50%" align="left" /> <hr size="5" width="50%" align="right"/> <hr size="10" width="100%" align="center"/></pre>	Output: 
---	--

Font style Elements: B, BIG, BLINK, I, SMALL, STRIKE, SUB, SUP, TT and U

These are also known as physical style elements.

Tag	Meaning	Display Style
	Bold contents	bold
<big>	Increased font size	bigger text
<blink>	Alternating fore- and background colors	blinking text
<i>	Italic contents	<i>italic</i>
<small>	Decreased font size	smaller text
<s>, <strike>	Strike-through text	strike
<sub>	Subscripted text	_{subscript}
<sup>	Superscripted text	^{superscript}
<u>	Underlined contents	<u>underlined</u>

Example:

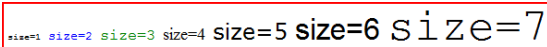
Code: <pre><p>bold, <i>italic</i>, <i>bold italic</i>, <tt>teletype text</tt>, and <big>big</big> and <small>small</small> text. <s>Strike</s> <u>Underline</u></p></pre>	Output: 
--	--

Font Element

The font tag is used to format the size, typeface and color of the enclosed text.

Attribute	Acceptable Values	Description
face	comma-separated list of font names e.g. "Arial, Verdana, Times New Roman".	Defines the fonts which the browser must search for in order of preference.
size	An integer between 1 and 7. A relative increase in font size. The value "+1" means one size larger. The value "-3" means three sizes smaller.	Set the size of the enclosed text.
color	16 defined color codes like red, blue, green, yellow or color value using RGB code e.g. #DDEEFF.	Set the color of the enclosed text.

Example:

Code: <pre>size=1 size=2 size=3 size=4 size=5 size=6 size=7</pre>	Output: 
--	--

Marquee Element

The marquee tag is a non-standard HTML markup element type which causes text to scroll up, down, left or right. The tag was first introduced in early versions of Microsoft's Internet Explorer.

Attribute	Acceptable Values	Description
align	left right center	Set the alignment of the text.
behavior	scroll (scrolls the text from right-to-left, and restarts at the right side of the marquee when it has reached the left side) slide alternate (Text bounces from the left side of the box to the right side)	Set the behavior of the marquee.
bgcolor	16 defined color codes like red, blue, green, yellow, black, white etc. Color value using RGB code e.g. #DDEEFF.	Set the background color.
direction	left right	Set the direction of marquee. Default is right to left.
Height	number (pixels)	Sets the height of the marquee.
Width	number (pixels)	Sets the width of the marquee.
Loop	number	Sets the number of times marquee must loop its text.
Scrollamount	number (pixels)	Defines the amount by which the marquee moves between frames.
scrolldelay	number (milliseconds)	Defines the amount of time between frames in milliseconds. Helps to control the speed of the marquee.

Example:

Code: <pre><marquee behavior="alternate">This text will bounce from LTR</marquee> <marquee direction="right">This text will scroll from LTR.</marquee> <marquee height="20px">The height of this marquee is 20 pixels.</marquee> <marquee loop="2">Loops twice before it stops playing.</marquee></pre>

```
<marquee scrollamount="10">Text will move ten pixels per 'frame'</marquee>
<marquee scrolldelay="1000">Very slow</marquee>
```

Working with Links

A link is a connection from one Web resource to another. Although a simple concept, the link has been one of the primary forces driving the success of the Web.

- A link has two ends -- called anchors -- and a destination. The link starts at the “source” anchor and points to the “destination” anchor, which may be any Web resource (e.g., an image, a video clip, a sound bite, a program, an HTML document, an element within an HTML document, etc.).
- To incorporate links in an HTML document the <A> tag along with a value of its href (hyperlink reference) attribute is used. The text between the opening and closing anchor tag will appear as a hyperlink and clicking it will open the page indicated by the href attribute.

Example:

Code: Click here to go to the home page of html.	Output: Click here to go to the <u>home</u> <u>page</u> of html.
---	---

Working with Lists

HTML offers three mechanisms for specifying lists of information. All lists must contain one or more list elements. Lists may contain:

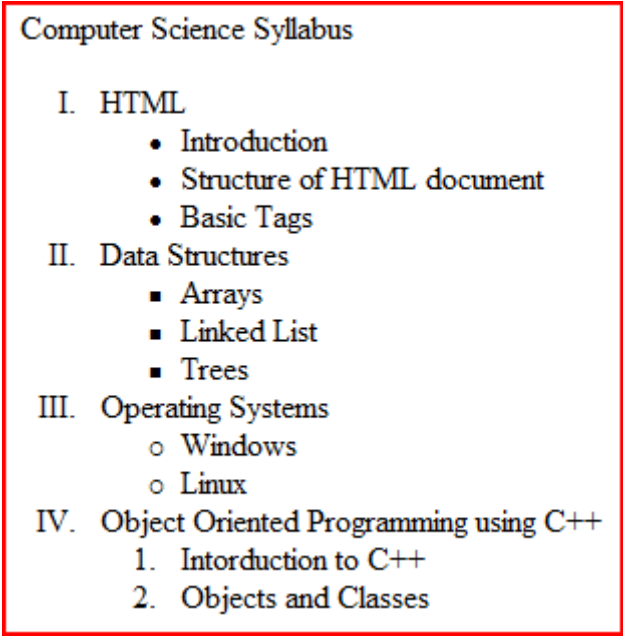
- Unordered information – unordered lists.
- Ordered information – ordered lists.
- Definitions – definition lists.

Ordered lists are numbered while unordered lists are bulleted. Definition lists consist of term followed by its definition.

Both ordered and unordered lists require start and end tags (or) as well as use of a special element (li) to indicated the beginning of each list item.

Attribute	Acceptable Values	Description
type	For ordered list – <ul style="list-style-type: none"> • 1 for Arabic numbers (default) (1, 2, 3 ...) • A for uppercase alphabets (A, B, C ...) • a for lowercase alphabets (a, b, c ...) • I for uppercase roman numerals (I, II, III, IV...) • i for lowercase roman numerals (i, ii, iii ...) For unordered list – <ul style="list-style-type: none"> • circle • square • disc 	Specifies the type of numbering or bulleting to use for ordered and unordered lists respectively.
start	number (for ol list only)	Specifies the starting number of the first item in an ordered list. The default starting number is “1”.
value	number (for li tag only)	Sets the number of the current list item.

Example:

Code: <pre> Computer Science Syllabus <ol type = I> HTML <ul type = disc> Introduction Structure of HTML document Basic Tags Data Structures <ul type = square> Arrays Linked list Trees Operating Systems <ul type = circle> Windows Linux Object Oriented Programming using C++ Intorduction to C++ Objects and Classes </pre>	Output: 
---	--

Working with Tables

The HTML table model allows authors to arrange into rows and columns of cells.

- Each table may have an associated caption (see the CAPTION element) that provides a short description of the table's purpose.
- A longer description may also be provided (via the summary attribute) for the benefit of people using speech or Braille-based browsers.
- Table cells may either contain “header” information (see the TH element) or “data” (see the TD element).
- Cells may span multiple rows and columns.

Attribute	Acceptable Values	Description
border	number (in pixels)	Specifies the width of frame surrounding the table.
bgcolor	16 defined color codes like red, blue, green, yellow, black, white etc. Color value using RGB code e.g. #DDEEFF.	Specifies the background color of the table.
cellspacing	number (in pixels)	Set the space between adjacent cells.
cellpadding	number (in pixels)	Set the space between the cell border and cell data.
width	number (in percentage or pixels)	Sets the width of the table.
summary	text	A long description of the table. Not rendered by the web browser.
align	left right center	Sets the alignment of the table in the HTML page.

Table Captions: The CAPTION element

When present, the CAPTION element's text should describe the nature of the table. The CAPTION element is only permitted immediately after the TABLE start tag. A TABLE element may only contain one CAPTION element.

Table rows: The TR element

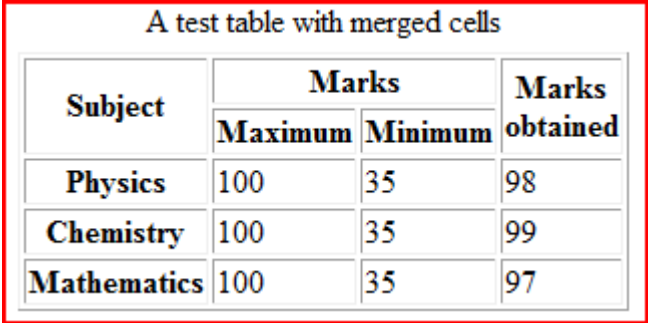
The TR element acts as a container for a row of table cells. The end tag may be omitted.

Table cells: The TH and TD elements

Table cells may contain two types of information: header information and data. This distinction enables browsers to render header and data cells distinctly, even in the absence of style sheets. The TH element defines a cell that contains header information. The TD element defines a cell that contains data.

Attribute	Acceptable Values	Description
bgcolor	16 defined color codes like red, blue, green, yellow, black, white etc. Color value using RGB code e.g. #DDEEFF.	Specifies the background color of the table.
colspan	number	Set the number of columns the cell should span. Used to merge cells.
rowspan	number	Set the number of rows the cell should span. Used to merge cells.
width	number (in percentage or pixels) or *	Sets the width of the table.
height	number (in percentage or pixels) or *	Sets the height of the table.
align	left right center	Sets the alignment of the text within the cell.
valign	top bottom middle	Sets the vertical alignment of the text within the cell.

Example:

Code: <pre> <table border="1" summary="this table marks obtained by a student"> <caption>A test table with merged cells</caption> <tr> <th rowspan="2">Subject</th> <th colspan="2">Marks</th> <th rowspan="2">Marks obtained</th> </tr> <tr> <th>Maximum</th> <th>Minimum</th> </tr> <tr> <th>Physics</th> <td>100</td> <td>35</td> <td>98</td> </tr> <tr> <th>Chemistry</th> <td>100</td> <td>35</td> <td>99</td> </tr> <tr> <th>Mathematics</th> <td>100</td> <td>35</td> <td>97</td> </tr> </table> </pre>	Output: 
--	--

Working with Images

Images included in a web page are sometimes referred to as inline images because the images are inserted within a line of the body text.


Popular format for images are

- .jpeg or .jpg (Joint Photographic Experts Group)
- .bmp (Bitmap)
- .png (Portable Network Graphics)
- .gif (Graphics Interchange Format)

Images can be added to a HTML page using the tag.

Attribute	Acceptable Values	Description
src	URI to image file	Set the source (path) to the image that must be shown in the web page.
align	left right top middle bottom	Set the horizontal and vertical alignment of the image.
alt	text	Alternate text to be displayed in case browser cannot render the image. This text is also displayed when mouse hovers over the image.
height	number (in percentage or pixels)	Sets the height of image. Can be used to resize the image.
width	number (in percentage or pixels)	Sets the width of image. Can be used to resize the image.
border	number (in pixels)	Determines whether or not to display and the thickness of the border of the image.

Example:

Code: <pre> </pre>	Output: 
--	---

C++

Probable Marks: 38

Scope of syllabus:

- Review of C++
- Arrays, Pointers, References and Strings
- Principle of Object Oriented Programming (OOPs)
- Classes and Objects
- Constructors and Destructors
- Operator overloading and type conversions
- Inheritance
- Virtual Functions and Polymorphism
- Working with files

Introduction

C++ (pronounced “See plus plus”) is a general-purpose programming language. It is regarded as a middle-level language, as it comprises a combination of both high-level and low-level language features. It was developed by Bjarne Stroustrup starting in 1979 at Bell Labs as an enhancement to the C programming language and originally named “C with Classes”. In 1983, the name of the language was changed from C with Classes to C++. (The ++ is C language operator).

The first commercial release of the C++ language was in October of 1985. After years of development, the C++ programming language standard was ratified in 1998.

C++ is widely used in the software industry, and remains one of the most popular languages ever created. Some of its application domains include systems software, application software, device drivers, embedded software, high-performance server and client applications, and entertainment software such as video games.

Advantages of C++

- Object-oriented programming - The possibility to design a program around objects allows the programmer to design applications from a point of view more like a communication between objects rather than on a structured sequence of code. This allows a greater reusability of code in a more logical and productive way.
- Portability - The same C++ code can be compiled in almost any type of computer and operating system without making any changes. C++ is the most used and ported programming language in the world.
- Brevity - Code written in C++ is very short in comparison with other languages, since the use of special characters is preferred to key words, saving some effort to the programmer.
- Modular programming - An application's body in C++ can be made up of several source code files that are compiled separately and then linked together. It is not necessary to recompile the complete application when making a single change but only the file that contains it. In addition, this characteristic allows to link C++ code with code produced in other languages, such as Assembler or C.
- C Compatibility - C++ is backwards compatible with the C language. Any code written in C can easily be included in a C++ program without making any change.
- Speed - The resulting code from a C++ compilation is very efficient, due to its duality as high-level and low-level language and the reduced size of the language itself.

First C++ program

Hello World is one of the simplest programs that can be written in the C++ language.

```
//My first program in C++ - Hello World
#include <iostream>
using namespace std;
int main()
{
    cout << "Hello World";
    return 0;
}
```


//My first program in C++ - Hello World

The first line of the program is a comment line. Every line that starts with two slash signs (//) are considered comments and will have no effect on the behavior or outcome of the program. (Words between /* and */ will also be considered as comments (old style comments)). Comments are used to explain difficult sections of the program.

#include <iostream>

Lines beginning with a pound sign (#) are used by the compilers pre-processor. In this case the directive #include tells the pre-processor to include the iostream standard file. This file iostream includes the declarations of the basic standard input/output library in C++.

using namespace std;

All the elements of the standard C++ library are declared within what is called a namespace. In this case the namespace with the name std. This line declares that the program will make use of the functionality offered in the namespace std. This line of code is used very frequent in C++ programs that use the standard library.

int main()

Every program must have a main() function. The main function is the point where all C++ programs start their execution.

```
{}
```

The two curly brackets (one in the beginning and one at the end) are used to indicate the beginning and the end of the function main. (Also called the body of a function). Everything contained within these curly brackets is what the function does when it is called and executed.

cout << "Hello World";

This line is a C++ statement. A statement is a simple expression that can produce an effect. This case the statement prints "Hello World" to the screen. cout represents the standard output stream in C++. It is declared in the iostream standard file within the std namespace. The words Hello World have to be between "", but the "" will not be printed on the screen. They indicate that the sentence begins and where it will end. The semicolon (;) is used to mark the end of the statement. The semicolon must be placed behind all statements in C++ programs.

return 0;

The return statement causes the main function to finish. Since the main function was defined as "int main()" the program expects that the main function will return an int (integer). Generally a zero indicates that everything went ok and a one indicates that something has gone wrong.

C++ Basics**Tokens**

A token is the smallest element of a C++ program that is meaningful to the compiler. C++ compiler identifies the following tokens -

- Keywords
- Identifiers
- Operators
- Literals

Keywords

- Keywords are predefined reserved identifiers that have special meanings.
- They cannot be used as identifiers in your program.
- C++ defines 48 keywords
- Examples include – int, double, float, for, if, void, return etc...

Identifiers

Identifier refers to the names of variables, function, arrays, classes, unions etc.

Rules for defining an identifier –

- Can contain only alphabets, numbers and underscore
- Identifier name cannot contain special characters
- Identifier name cannot contain spaces
- Cannot start with a digit
- Is case sensitive (upper and lower case are treated distinctly)
- Cannot be a keyword
- Can be of any length

Operators

Operators specify an evaluation to be performed on one of the following -

- One operand (unary operand)
- Two operands (binary operand)
- Three operands (ternary operand)

Commonly used operators:

- assignment: =
- compound assignment: += -= *= /= %=
- arithmetic: + - * / %
- increment, decrement: ++ --
- bitwise operators:
 - & | ~ and, or, and complement
 - << >> left and right bit shifting
- comparison: < > <= >= != ==
- logical conditions: && || ! and , or and not

Insertion (<<) and Extraction (>>) Operator

Insertion or Put Operator <<: It inserts (or sends) the contents of the variable on its right to the object on its left. The following statement will display the value of the variable `studentName` on the screen.

```
cout << studentName
```

Extraction Get Operator >>: It extracts (or takes) the value from the keyboard and assigns it to the variable to its right. The following statement causes the program to wait for the user to input the value of the variable `studentName`.

```
cin >> studentName
```

Scope Resolution (::) Operator

When a local variable has the same name as a global variable, all references to the variable name made within the scope of the local variable refer to the local variable. The local variable takes the precedence over the global variable. In such cases, the scope resolution operator is used to access the global variable.

Data Types

C++ has three types of data – built-in or fundamental, user defined and derived.

Built-In (Native or Fundamental)

Built-in data types are predefined data types in C++ language.

- int, long, short, char (signed and unsigned): Default is signed
 - C++ guarantees a char is one byte in size
 - Sizes of other types are platform dependent
- float, double (floating point division)

- More expensive in space and time
- bool type
 - Logic type, takes on values true, false
- void

The table below summarizes different built-in data types, their size and range of data that they can hold.

Name	Description	Size	Range
char	Character or small integer.	1 byte	signed: -128 to 127 unsigned: 0 to 255
short int (short)	Short Integer.	2 bytes	signed: -32768 to 32767 unsigned: 0 to 65535
int	Integer.	4 bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
long int (long)	Long integer.	4 bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
bool	Boolean value. It can take one of two values: true or false.	1 byte	true or false
float	Floating point number.	4 bytes	3.4e +/- 38 (7 digits)
double	Double precision floating point number.	8 bytes	1.7e +/- 308 (15 digits)
long double	Long double precision floating point number.	8 bytes	1.7e +/- 308 (15 digits)

long, short, signed and unsigned are known as data type modifiers. These modifiers are used to change the size (and hence the range of data that can be stored) of the basic data type.

User Defined

User defined data types allow a programmer to extend the functionality of C++ and define data types which make it easier to solve the problem. In most cases user defined data types define a collection of various built-in data types.

- Union, Structures and Classes
- Enumeration

Derived Type

Derived data types are not unique data types but are derived from one of the fundamental or user defined type. For example an array of integers denotes a collection of number of integers stored in a particular format in memory but does not define a unique data type.

- Array - collection of objects.
- Pointer - points to the location where an object is stored in memory.
- Function - returns an object (the data type of a function is the return type of the function).

Variables

Variable represents a storage location in the computer's memory. Each variable needs an identifier that distinguishes it from the others. Most common way that a variable obtains a value is by means of assignment

```
variable = expression or value;
```

A variable cannot be used without declaring it. To declare a variable use the following syntax

```
type variable;
```

Type is a standard or user defined data type (int, double, float etc...).

Reference Variable

A reference variable provides an alternate name (alias) to a previously defined variable. A reference variable is created as follows.

```
type &reference_variable_name = variable_name
```

Example:

```
float total = 100;
float &sum = total;
```

In the above example, both sum and total refer to the same data object in memory. Changing the value of one variable will change the value of the other variable.

Constants

Constants are expressions with fixed value. Constants can be of three types -

1. Literals are used to express particular values within the source code of a program
 - a. Integer Numerals
 - b. Floating-Point Numerals
 - c. Characters and Strings
 - d. Boolean Values

```
1 | 100    //decimal interger
2 | 0x4b   //hexadecimal interger
3 | 3.1415 //floating point numeral
4 | "Hello World" //string constant
```

2. Defined constants - Users can define their own constants

```
1 | #define PI 3.141592;
2 | #define newline "\n";
```

3. Declared constants - Define constants of a particular type

```
1 | const int arraysize = 100;
```

Control Structures

Conditional Statements

If condition

The 'if condition' is used to check for a particular condition. If the condition evaluates as true a course of action is followed and a statement or a set of statements are executed.

Syntax:

```
if (condition)
{
    statements
}
if (condition)
{
    statements
}
else
{
    statements
}
```

Example:

```
if (x % 2 == 0)
{
    cout << "x is an even number";
}
else
{
    cout << "x is an odd number";
}
```

Switch Statement

C++ provides a multiple-branch selection statement known as switch. This selection statement successively tests the value of an expression against a list of integer or character constants. When a match is found, the statements associated with that constant are executed.

```
switch (variable)
{
    case constant1:
        group of statements 1
        break;
    case constant2:
        group of statements 2
        break;
    .
    .
    default:
        default group of statements
}
```

Example:

```
switch (x)
{
    case 1:
        cout << "value of x is 1";
        break;
    case 2:
        cout << "value of x is 2";
        break;
    case 3:
        cout << "value of x is 3";
        break;
    default:
        cout << "value of x is not known";
}
```

Loops

For loop

The syntax of a for loop is -

```
for (initialization; condition; increase)
{
    group of statements;
}
```

The for loop repeats a statement or group of statements while condition remains true. This is similar to a while loop but in addition, the for loop provides specific locations to contain an initialization statement and an increase statement. This loop is specially designed to perform a repetitive action with a counter which is initialized and increased on each iteration.

It works in the following way:

1. initialization is executed. The value of the counter is initialized to the starting value. This is executed only once.
2. condition is checked. If it is true the loop continues, otherwise the loop ends and statements are skipped (not executed).
3. statements are executed.
4. finally, the counter is incremented or decremented and the loop gets back to step 2.

Example:

```
//code to print the first 10 numbers
for (int i = 1; i <= 10; i++) {
    cout << "i" << endl; }
```

While loop

The syntax of a while loop is -

```
while (condition)
{
    group of statements;
}
```

The while loop is an entry controlled loop. In a while loop the group of statements is executed only when the condition is true.

Example:

```
//code to print the first 10 numbers
int i = 1;
while (i <= 10) {
    cout << "i" << endl;
    i = i + 1;
}
```

Do while loop

The syntax of a do while loop is -

```
do
{
    group of statements;
}
while (condition);
```

In a do-while loop, the group of statements are executed and then the condition is checked. Unlike the while loop, the do-while loop is executed at least once.

Example:

```
//code to print the first 10 numbers
int i = 1;
do {
    cout << "i" << endl;
    i = i + 1;
}
while (i <= 10);
```

Functions

Functions allow developers to modularize their program. Benefits of using functions include –

- Divide and conquer (construct a program from smaller pieces or components).
- Software reusability
 - Use existing functions as building blocks for new programs.
 - Abstraction - hide internal details (library functions).
- Avoid code repetition.

Syntax for defining any function is –

```
return_type  function_name( list_of_parameters )
{
    declarations and statements
}
```

Where the functions are declared in your program does not matter, as long as a functions name is known to the compiler before it is called.

Functions can accept parameters and can return a result. The return_type indicates the data type of the result. If a function does not return any value then the return type of that function is set as void.

Passing parameters to functions

There are two ways to pass parameters (information to a function) –

Call by value

In call by value the copy of variable is passed to function. Changes to the value in function do not affect original value. It is used when the function does not need to modify variable. Thus it avoids accidental changes to the value of the variable.

Call by reference

In call by reference the pointer to the original variable is passed to the function. This is accomplished by using the ‘&’ operator to pass the address of the variable being passed. Any change to the value in the function will change the value of the original variable.

Example:

```
#include <iostream>
using namespace std;

void passByValue(int x)
{
    x = x + 1;
}
void passByReference(int &x)
{
    x = x + 1;
}
void main()
{
    int a = 10;
    cout << "a = " << a << endl; //Output : 10
    /*Copy of variable is sent in pass by value. Original value is not changed*/
    passByValue(a);
    cout << "a = " << a << endl; //Output : 10
    //A pointer to the original variable is sent in pass by reference.
    //Any changes in the function will change the original value.
    passByReference(a);
    cout << "a = " << a << endl; //Output : 11
}
```

Inline Function

An inline function is one for which the compiler copies the code from the function definition directly into the code of the calling function rather than creating a separate set of instructions in memory. Instead of transferring control to and from the function code segment, a modified copy of the function body may be substituted directly for the function call. In this way, the performance overhead of a function call is avoided.

A function is declared inline by using the inline function specifier or by defining a member function within a class or structure definition.

Example:

```
//program to find the maximum of two numbers using inline function.
#include
using namespace std;
inline void maxOfTwoNumbers (int a, int b)
{
    if (a > b) {
        cout << "a is greater";
    }
    else {
        cout << "b is greater";
    }
}
void main ()
{
```

```

int x, y;
cin >> x;
cin >> y;
maxOfTwoNumbers(x, y);
}

```

Function Overloading

C++ allows more than one function to have the same name as long as they have a different signature. The signature of a function is list of parameters (type and number of parameters). As long as the parameter list is different the compiler is able to distinguish between the functions even if they have the same name. This is known as function overloading.

Example:

```

int max (int x, int y);
int max (int x, int y, int z);
float max (float x, float y);
double max (float x, float y);

```

The above example shows 3 overloaded functions. The 3 functions have the same name (max) but take in different types of data or different number of parameters. When the function max is called, based on the type and number of parameters being passed, the compiler will call the appropriate function.

Default Arguments

C++ allows programmers to assign default value to the parameters being passed to the function. If a default value is defined and no value is passed while calling the function, the function uses the default value.

```

float interest(float principal, float time, float rate = 0.05) //rate has default value of 5%
{
    return principal * time * rate;
}

```

Example:

The above function can be called in the following ways –

```

interest (1000, 1, 0.10) //as value for rate is passed, the function will use rate = 0.10
interest (1000, 2) //no value for rate is passed, the default value of 0.05 will be used

```

An important thing to note about default values is that a parameter can have a default value only if all the parameters appearing on its right have a default value. The following example shows incorrect use of default arguments.

```

float interest(float principal, float time = 1, float rate) //illegal as rate does not have
default value
{
    return principal * time * rate;
}

```

Arrays

An array lets you declare and work with a collection of objects of the same type. The objects in an array are stored in consecutive memory locations in the computer's memory. Each object (element) can be accessed, by using square brackets, with the element number inside. All arrays start at element zero and will go to n-1.

Declaring an array

An array declaration requires 3 things -

1. Type of objects that will be stored in the array.
2. Name of the array.
3. Size of the array or in other words then number of elements that will be stored in the array.

Syntax to declare an array shown below -

```

type array_name[size];

```


Array initialization

C++ provides the facility to initialize an array at the time of array declaration. The syntax to initialize an array at declaration is as under -

```
type array_name[size] = { list_of_value_seperated_by_commas };
```

Example:

```
int daysOfMonth[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

Character arrays can also be initialized as under –

```
char subject[] = "Computer Science";
```

Multi-dimensional arrays

Multi-dimensional arrays have more than one dimension. Syntax for declaring a n-dimensional arrays is -

```
type array_name[size1][size2]....[sizen]
```

The common multi-dimensional array is a two dimensional array. A two dimensional array is similar to a table which can store data in rows and columns.

Arrays and Loops

Loops can be used to manipulate elements of an array.

Example:

```
//reading and writing to arrays using loops
#include
using namespace std;

int main()
{
    int a[4];
    int i;
    for ( i = 0; i < 4; i++ ) {
        a[i] = i; }
    for ( i = 0; i < 4; i++ ) {
        cout << a[i] << endl; }
    return 0;
}
```

Arrays and Functions

Individual array elements are passed to a called function in the same way as scalar variables. However it is not possible to pass a complete array by value to a function. Arrays can be passed to a function by reference only. As a result the calling function receives access to that actual array and not a copy of the array.

Example:

```
//passing array as input to a function
#include
using namespace std;
void array_function(int[] a){
    for (int i = 0; i < 5; i++ ) {
        cout << a[i] << endl;
    }
}
void main()
{
    int numbers[] = {1, 2, 3, 4, 5};
    array_function(numbers);
}
```

Pointers

A pointer stores a memory address. Since pointers store memory address, they can be used to point to the location in memory where a normal variable is stored. Because pointers contain addresses to other variables they are also called address variables.

Declaring pointers

Pointers are declared like any other variables; the only difference being that an asterisk (*) is put before the variable name.

```
type *variable_name;
```

Example:

```
#include <iostream>
using namespace std;

void main()
{
    int x;
    int *ptr;

    x = 5;
    ptr = &x;
    cout << "Address of x = " << &x << endl;    //Address of x = 004EF798
    cout << "Value at x = " << x << endl;        //Value at x = 5
    cout << "Address of ptr = " << &ptr << endl; //Address of ptr = 004EF78C
    cout << "Value at ptr = " << ptr << endl;    //Value at ptr = 004EF798
    cout << "Value at pointed location = " << *ptr;
}
```

In the example above, x is an integer variable and ptr is an integer pointer which points to x. x is stored at 004EF798 and ptr is stored at 004EF78C. x stores the value 5 while ptr stores the address of x i.e. 004EF798.

Advantages of using pointers

Some of the advantages of pointers include:

- It allows to pass and return structured variables like arrays, strings, functions and objects to and from a function.
- Pointers support dynamic memory allocation and de-allocation. This helps in design memory efficient programs.
- Variables can be swapped without using any extra memory by just swapping their pointers.
- Pointers help to implement powerful data structures like linked lists and trees.
- Pointers also help to implement certain features of OOP like abstract classes and virtual functions.

Reference and De-reference Operator

The ampersand sign (&) is called the reference operator. The reference operator is used to get the “address of” a variable. The line: ptr = &x; instructs the computer to store the address of the variable x in the pointer ptr.

The asterisk sign (*) is called the dereference operator. The dereference operator is used to get the “value pointed by” a pointer. The line: cout << *ptr; instructs the computer to print the value stored at the address pointed by ptr. A pointer must have a value before it is dereferenced.

Pointer Arithmetic

Addition and subtraction of pointers is slightly different than addition and subtraction of normal variables. Addition and subtraction of pointers changes the value of the pointer by [(size of data type pointed to) * (the value being added or subtracted)]. In the example below, when 1 is added to the ptr (an integer pointer) the value of the pointer increases by 4 (an integer in C++ requires 4 bytes of memory).

Example:

```
#include <iostream>
using namespace std;

void main()
{
    int x = 5;
    int *ptr;
    ptr = &x;
    cout << "Value at ptr = " << ptr << endl;           //Value at ptr = 004EF770
    ptr_int = ptr_int + 1;
    cout << "Value at ptr = " << ptr << endl;           //Value at ptr = 004EF774
}
```

Pointers and Arrays

The name of an array is also a pointer to the starting address of the array and subsequent elements are stored sequentially one after another in an array. Pointers can therefore be used to read arrays as shown in the code below where p is a pointer which points to the starting address of an array a. Pointer arithmetic and dereference operator is then used to read the elements stored in an array.

```
#include <iostream>
using namespace std;

void main()
{
    const int size = 10;
    int a[size] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

    int* p = a;

    cout << "Printing the array using pointers..." << endl;
    for (int i = 0; i < size; i++)
    {
        cout << *p+i;
        if (i != size - 1) cout << ", ";
    }
    cout << endl;
}
```

Dynamic memory allocation

New and Delete operators

There are cases when static allocation (at the start of a program or function) of memory often requires too much memory to be allocated and thus wasting precious memory. The **new**, **new []**, **delete** and **delete []** operators can be used to allocate and free variable sized blocks of memory for classes, structures and arrays while a program is running. new and delete are unary operators. In the example below the new [] operator is used to allocate 5 bytes of memory to store characters and then the delete [] operator is used to free up that memory.

The new and delete operators are used to allocate and de-allocate one element at a time while the new [] and delete [] operators are used to allocate and de-allocate an array of elements.

Example:

```
#include <iostream>
using namespace std;

void main()
{
    char *str;
    str = new char[5]; // allocates 5 bytes
    strcpy(str, "test");
    cout << str;
    delete [] str;
    str = NULL;
}
```

Memory leaks

When some memory which was dynamically assigned is not returned to the computer using the delete operator it results in a memory leak. Memory leaks make the system unstable and result in less usable memory than is physically available in the computer.

Example:

```
void somefunction()
{
    int A1[10];
    int *A2 = new int [10];
    ...
    G(A1);
    G(A2);
}
```

In the above example when scope is exited memory from automatic array A1 is freed but not memory pointed by A2. This results in a memory leak equivalent to size of 10 integers. Adding delete [] A2; before exiting the function will fix the memory leak.

Strings

There are two ways to work with strings in C++ -

1. C style programming - each string is implemented as a char array terminated by NUL or '\n'.
2. C++ style programming - making use of the string class defined in namespace std.

C++ also defines various string manipulation functions that allows programmers to work with strings. Commonly used functions include -

- strcpy - used to copy strings
- strlen - get the length of a string
- strcmp - compare two strings
- strcat - concatenate string (join strings)

Object Oriented Programming

Object-oriented programming (OOP) is a programming paradigm that uses “objects” – data structures consisting of data fields (variables) and methods (functions) together with their interactions – to design applications and computer programs.

An object-oriented program may be viewed as a collection of interacting objects, as opposed to the conventional model, in which a program is seen as a list of tasks (subroutines) to perform. In OOP, each object is capable of receiving messages, processing data, and sending messages to other objects and can be viewed as an independent ‘machine’ with a distinct role or responsibility. OOP follows a bottoms-up programming approach as against the top down approach followed by traditional programming.

Objects

Real-world objects share two characteristics: They all have state and behavior. Dogs have state (name, color, breed, hungry) and behavior (barking, fetching, wagging tail). Bicycles also have state (current gear, current pedal cadence, current speed) and behavior (changing gear, changing pedal cadence, applying brakes).

Software objects are conceptually similar to real-world objects: they too consist of state and related behavior. An object stores its state in fields (variables in some programming languages) and exposes its behavior through methods (functions in some programming languages). Methods operate on an object's internal state and serve as the primary mechanism for object-to-object communication. Hiding internal state and requiring all interaction to be performed through an object's methods is known as data encapsulation — a fundamental principle of object-oriented programming.

Bundling code into individual software objects provides a number of benefits, including -

- **Modularity:** The source code for an object can be written and maintained independently of the source code for other objects. Once created, an object can be easily passed around inside the system.
- **Information-hiding:** By interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world.
- **Code re-use:** If an object already exists (perhaps written by another software developer), it can re-used in a program.
- **Pluggability and debugging ease:** If a particular object turns out to be problematic, you can simply remove it from your application and plug in a different object as its replacement. This is analogous to fixing mechanical problems in the real world. If a bolt breaks, you replace it, not the entire machine.

Classes

In the real world, there are many individual objects all of the same kind. There may be thousands of other bicycles in existence, all of the same make and model. Each bicycle was built from the same set of blueprints and therefore contains the same components. In object-oriented terms, we say that a particular bicycle is an instance of the class of objects known as bicycles. A class is the blueprint from which individual objects are created.

Inheritance

Different kinds of objects often have a certain amount in common with each other. A Maruti, BMW, and Honda, for example, all share the characteristics of cars (efficiency, current speed and current gear). Yet each also defines additional features that make them different (safety and security features like airbags, ABS etc.)

Object-oriented programming allows classes to inherit commonly used state and behavior from other classes. In this example, car now becomes the superclass of Maruti, BMW and Honda.

Data Abstraction

The importance of abstraction is derived from its ability to hide irrelevant details. Abstraction is essential in the construction of programs. It places the emphasis on what an object does rather than how it is represented in memory or how it works.

Functions are responsible for manipulating the data. The public interface completely defines how to use this object. Programs that want to manipulate an object only have to be concerned about which messages this object understands. They do not have to worry about how these tasks are achieved nor the internal structure of the object. The hiding of internal details makes an object abstract, and the technique is normally known as data abstraction.

Data Encapsulation

There is a very close attachment between data items and functions. Generally in in OOP, access to data is only possible through functions. Thus functions control what you can and cannot do with the data. In other words data is encapsulated by the functions. In addition access to functions can be restricted by using access specifiers thus allowing a developer to control who can access a particular function and hence the data exposed by that function.

Polymorphism

In object-oriented programming, polymorphism (from the Greek meaning “having multiple forms”) is the characteristic of being able to assign a different meaning to the same term in different contexts. In OOPS this allows an entity such as an operator, a variable, a function, or an object to have more than one form. Polymorphism is a mechanism that allows you to implement a function in different ways. Function overloading and overloaded constructors are examples of polymorphism.

Benefits of Object Oriented Programming

- Analyzing user requirements - software objects model real world objects, so the complexity is reduced and the program structure is very clear.
- Constructing software -
 - modularity: each object forms a separate entity whose internal workings are decoupled from other parts of the system.
 - modifiability: it is easy to make minor changes in the data representation or the procedures in an OO program. Changes inside a class do not affect any other part of a program, since the only public interface that the external world has to a class is through the use of methods.
 - extensibility: adding new features or responding to changing operating environments can be solved by introducing a few new objects and modifying some existing ones.
 - maintainability: objects can be maintained separately, making locating and fixing problems easier.
 - re-usability: objects can be reused in different programs.

Classes and Objects

In the real world, there are many individual objects all of the same kind. There may be thousands of other bicycles in existence, all of the same make and model. Each bicycle was built from the same set of blueprints and therefore contains the same components. In object-oriented terms, we say that a particular bicycle is an instance of the class of objects known as bicycles. A class is the blueprint from which individual objects are created.

A class can not only hold data variables, but can also hold functions. These variables and functions are members of a class. The variables are called data members and functions are called member functions.

Defining a class

The keyword “class” is used to define a class in C++. Syntax for defining a class is as follows -

```
class class_name
{
    access_specifier:
        member1;
    access_specifier:
        member2;
};
```

The access_specifier can be one of the following keywords: private, public or protected. With these specifiers the access right of the members that will follow are set.

- private members of a class are only accessible by other private members of the same class.
- protected members are accessible by members of the same class. protected members are also accessible by members of derived classes.
- public members are accessible from anywhere where the object is used.

Note: all members of a class (declared with the class keyword) have private access by default, unless another specifier overrules it.

Creating objects

Objects of any class are created like variables of built-in data types. Just the class name is used instead of the built-in data type while defining the type of variable.

```
class_name variable_name;
```

Example:

```
#include <iostream>
using namespace std;

const float PI = 3.1415;

class Circle
{
private:
    float radius;
public:
    void setRadius(float r)
    {
        radius = r;
    }
    float getCircumference()
    {
        return 2 * PI * radius;
    }
};

void main()
{
    Circle c;
    cout << "Enter the radius of the circle: ";
    int radius;
    cin >> radius;
    c.setRadius(radius);
    cout << "Circumference = " << c.getCircumference() << endl;
}
```

Defining Member Functions

Functions can be defined either inside the class definition as shown in the example above or outside the class definition as show in the example below. When functions are defined outside the class, the function prototype is included in the class definition and the scope resolution operator is used to associate the function with the class.

Example:

```
#include <iostream>
using namespace std;
const float PI = 3.1415;
class Circle{
private:
    float radius;
public:
    void setRadius(float r);
    float getCircumference();
};
void Circle::setRadius(float r){
    radius = r;
}
float Circle::getCircumference(){
    return 2 * PI * radius;
}

void main(){
    Circle c;
    cout << "Enter the radius of the circle: ";
    int radius;
    cin >> radius;
    c.setRadius(radius);
    cout << "Circumference = " << c.getCircumference() << endl;
}
```

The only difference between defining a class member function completely within its class and to include only the prototype (later on its definition), is that in the first case the function will automatically be considered an inline member function by the compiler, while in the second it will be a normal (not inline) class member function, which in fact supposes no difference in behavior.

Friend Function

A C++ friend functions are special functions which can access the private members of a class. If we want to declare an external function as friend of a class, thus allowing this function to have access to the private and protected members of this class, we do it by declaring a prototype of this external function within the class, and preceding it with the keyword friend as shown in the example below.

Example:

```
#include <iostream>
using namespace std;

class Complex
{
private:
    float real;
    float img;
public:
    friend Complex add(Complex c1, Complex c2);
    Complex(float r = 0, float i = 0)
    {
        real = r;
        img = i;
    }
};

Complex add(Complex c1, Complex c2)
{
    Complex result;
    result.real = c1.real + c2.real;
    result.img = c1.img + c2.img;
    return result;
}

void main()
{
    Complex c1(5,5);
    Complex c2(-4,-4);
    Complex c3 = add(c1,c2);
}
```

Friend functions have the following properties:

- Friend of the class can be member of some other class.
- Friend of one class can be friend of another class or all the classes in one program, such a friend is known as GLOBAL FRIEND.
- Friends are non-members hence do not get “this” pointer.
- Friends, can be friend of more than one class, hence they can be used for message passing between the classes.
- Friend can be declared anywhere (in public, protected or private section) in the class.

Static class members

Static means something maintaining state either forever or up to some point. In C++, static members get their life active as soon as the execution of the program starts, no matter, whether they are created in local, global or class scope.

Scope and life of static members –

- They are allocated as soon as the execution of program starts, regardless of their scope.
- They are de-allocated at the end of the program execution.
- Since they have lifetime, during the whole of the program execution, they retain the values in them.
- They are usable in their scope only though their life is for the whole of the program execution.
- On creation, they are initialized with ZERO.(other locally created variables shall get some undefined (garbage) value in them).
- Static members are created only once as soon as the execution of the program starts and the same is shared among all the objects
- Static members are constrained by the access modifiers (public, private, protected).

Declaring and using static members –

- Static members of the class must be re-declared outside the class (globally).
- Static data member can be used directly in static member functions only; otherwise they are used using Scope Resolution Operator.
- Static functions can only use static data members or local variables; they cannot use non static data members of the class.

Example:

```
#include <iostream>
using namespace std;
class Rectangle
{
public:
    int length;
    Rectangle()
    {
        length = 0;
        count++;
    }
    ~Rectangle()
    {
        count--;
    }
    static void displayCount(); //static member function

private:
    static int count; //static data member
};
int Rectangle::count;
void Rectangle::displayCount()
{
    cout << "There are " << count << " rectangles" << endl;
}

void main()
{
    Rectangle rectangles[5];
    Rectangle::displayCount();
}
```

Constructors and Destructors

Classes can have complicated internal structures, so object initialization and clean-up of a class is much more complicated than for any other data structures. Constructors and destructors are special member functions of classes that are used to construct and destroy class objects. The construction can be for example: initialization for objects or memory allocation. The destruction may involve de-allocation of memory or other clean-up for objects.

Constructors and destructors are declared within a class declaration (as like any other member function). A constructor or a destructor can be defined in-line or external to the class declaration. We may declare some default

arguments when we make a constructor. Generally constructors are defined as public so that objects can be created from any function within the program.

There are some restrictions that apply to constructors and destructors:

- The constructor and destructor must have the same name as the class. The only difference being that the destructor is preceded with a tilde sign (~).
- While parameters can be passed to a constructor, no parameters can be passed to a destructor.
- Destructors cannot be inherited.
- Constructors and destructors cannot have a return type (not even void).
- Pointers and references cannot be used on constructors and destructor's (It is not possible to get their address)
- Constructors and destructors cannot be declared static, const or volatile.
- Constructors cannot be declared with the keyword virtual.
- The same access rules apply to constructors and destructors as with any other member function.

```
#include <iostream>
using namespace std;

class Square{
public:
    float length;
    Square()
    {
        length = 0.0;
        cout << "A new square with 0 length has been created.";
    }

    Square (float l)
    {
        length = l;
        cout << "A new square with length = " << length << "was created" << endl;
    }

    ~Square()
    {
        cout << "Square with length " << length << " was destroyed " << endl;
    }
};

void main() {
    Square s1;
    Square s2(5);
}
```

Default Constructor

Constructors are called automatically by the compiler when defining class objects. The destructor is called when a class object goes out of scope.

If the programmer does not declare any constructors destructor in a class definition, the compiler will create a default constructor with no arguments. Similarly the compiler will also create a default destructor if not destructor has been specified explicitly.

Overloaded Constructors (or Parameterized Constructors)

Like any other function, a constructor can also be overloaded with more than one function that have the same name but different types or number of parameters. For overloaded functions the compiler will call the one whose parameters match the arguments used in the function call. In the case of constructors, which are automatically called when an object is created, the one executed is the one that matches the arguments passed on the object declaration. In the above example, when Square s1 was created the empty constructor was called, while when Square s2 was created, the overloaded constructor which takes one parameter (side of the square) is called.

Copy Constructor

To create an object from an existing object requires a special type of constructor, called a copy constructor. It allocates memory but copies the data (value of all variables in the class) from another object of the same class. It requires one parameter - a reference to an existing object. Like a default constructor, one is supplied by the compiler if there is a need for one and you haven't supplied it.

Copy constructors is automatically called when -

1. an object is returned by value
2. an object is passed (to a function) by value as an argument
3. the programmer uses the assignment (=) operator

Example:

```
#include <iostream>
using namespace std;
class Circle{
public:
    float radius;
    Circle(float r = 0.0f){
        radius = r;
    }

    Circle (Circle& original){
        radius = original.radius;
        cout << "copy constructor is called" << endl;
    }
};
void dummyMethod(Circle c){
    cout << "some text";
}
void main(){
    Circle c1(5.0);
    Circle c2 = c1;
    c2.radius = 10;
    dummyMethod(c2);
}
```

Operator Overloading

Operator overloading is a concept of overloading of existing operators, so that they can be used in customized ways. The C++ language uses the keyword “operator” for overloading of operators.

Syntax:

```
return_type operator op (list_of_parameters)
{
    statements
    ...
}
```

Operator overloading is normally applied on “class” data types, as these are user defined types and operators normally do not work with them. By applying overloading of operators, we can make them (operators) work for user defined types.

Example:

```
#include <iostream>
using namespace std;

class Complex{
private:
    float real;
    float img;
public:
    Complex operator+(Complex c2);
}
```

```

Complex(float r = 0, float i = 0){
    real = r;
    img = i;
}
void display(){
    cout << "Complex number is: " << real << "+" << img << "i" << endl;
}
};
Complex Complex :: operator+(Complex c2){
    Complex result;
    result.real = real + c2.real;
    result.img = img + c2.img;
    return result;
}

void main(){
    Complex c1(5,5);
    Complex c2(-4,-4);
    Complex c3 = c1 + c2;
    c3.display();
}

```

Rules for overloading operators

1. You cannot define new operators, such as **. Only existing operators can be overloaded.
2. You cannot redefine the meaning of operators when applied to built-in data types.
3. Overloaded operators must either be a non-static class member function or a global function. A global function that needs access to private or protected class members must be declared as a friend of that class. A global function must take at least one argument that is of user defined type.
4. Unary operators declared as member functions take no arguments; if declared as global functions, they take one argument.
5. Binary operators declared as member functions take one argument; if declared as global functions, they take two arguments. Binary operators must explicitly return a value.
6. If an operator can be used as either a unary or a binary operator (&, *, +, and -), you can overload each use separately.
7. There are some operators that cannot be overloaded like size of operator(sizeof), membership operator(.), pointer to member operator(*), scope resolution operator(::) and conditional operators(?:).
8. We cannot use "friend" functions to overload certain operators. However, member function can be used to overload them. Friend Functions cannot be used with assignment operator(=), function call operator(()), subscripting operator([]), class member access operator(->) etc.
9. Overloaded operators cannot have default arguments.
10. Overloaded operators follow the syntax rules of the original operators. They cannot be overridden.
11. All overloaded operators except assignment (operator=) are inherited by derived classes.

Type Conversion

Implicit type conversion is performed by the compiler in expressions with mixed data types. There are three types of data conversions in C++:

1. Conversion from built-in type to class type.
2. Conversion from class to built-in type.
3. Conversion from one class type to another class type.

When user defined types are used in expressions containing mixed data types, conversions must be defined by the class developer using either a conversion operator, or an appropriate constructor.

Conversion Constructor

A constructor that can be called with a single argument is used for conversions from the type of the argument to the class type. Such a constructor is called a conversion constructor.

In the example below, Polar coordinate is converted to Rectangular coordinate by passing it as a parameter to the constructor of a Rectangular class.

```
#include <iostream>
using namespace std;
const float PI = 3.1415;
class Polar{
public:
    float r;
    float theta;
    Polar(float r1 = 0, float theta1 = 0)
    {
        r = r1;
        theta = theta1;
    }
};
class Rectangular{
public:
    float x;
    float y;
    Rectangular(float x1 = 0, float y1 = 0)
    {
        x = x1;
        y = y1;
    }
    Rectangular(Polar p1)
    {
        x = p1.r * cos(p1.theta*PI/180);
        y = p1.r * sin(p1.theta*PI/180);
    }
    void display()
    {
        cout << "Coordinates are: " << x << "," << y << endl;
    }
};

void main(){
    Polar polar(2,60);
    Rectangular rec(polar);
    polar.display();
}
```

Conversion Operator

A conversion operator is a special member function that specifies the implicit conversion of a class object to another type.

In the example below, Polar coordinate is converted to Rectangular coordinate by use of a conversion operator defined in the Polar class.

```
#include <iostream>
using namespace std;
const float PI = 3.1415f;
class Rectangular
{
public:
    float x;
    float y;
    Rectangular(float x1 = 0, float y1 = 0)
    {
        x = x1;
        y = y1;
    }

    void display()
    {
        cout << "Coordinates are: " << x << "," << y << endl;
    }
};
```

```

    }
};
class Polar
{
public:
    float r;
    float theta;
    Polar(float r1 = 0, float theta1 = 0)
    {
        r = r1;
        theta = theta1;
    }

    operator Rectangular()
    {
        float x = r * cos(theta*PI/180);
        float y = r * sin(theta*PI/180);
        return Rectangular(x,y);
    }
};

void main()
{
    Polar p(1,30);
    Rectangular r = p;
    r.display();
}

```

Inheritance

Inheritance is a concept of linking two or more classes with each other in a hierarchical manner so that their properties and functions can be shared. (One class will extend to another class.) This leads to the biggest advantage of re-usability of the members and avoids redundancy.

Syntax:

```

class derived_class_name: access_specifier base_class_name
{
    base_class_definition;
}

```

A derived class inherits every member of a base class except:

- its constructor and its destructor
- its friends
- its operator=() members

Where derived_class_name is the name of the derived class and base_class_name is the name of the class on which it is based. This access specifier (public, private or protected) limits the most accessible level for the members inherited from the base class: The members with a more accessible level are inherited with this level instead, while the members with an equal or more restrictive access level keep their restrictive level in the derived class.

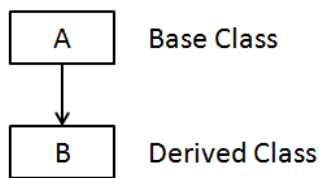
We can use the following chart for seeing the accessibility of the members in the base class and derived class.

		Inheritance Mode		
		public	protected	private
Members in base class	Public	public	protected	private
	Protected	protected	protected	private
	Private	-	-	-
		Members in derived class		

Types of inheritance

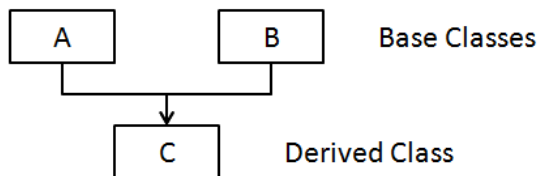
Single inheritance

A derived class inherits from only one base class.



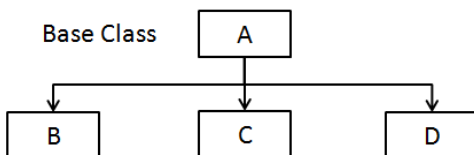
Multiple inheritance

A derived class inherits from multiple base classes.



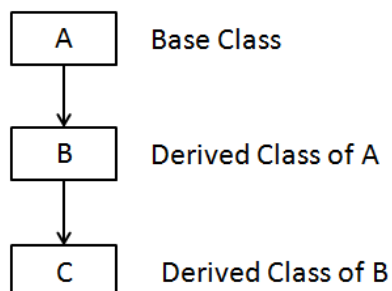
Hierarchical inheritance

Many derived classes inherit from a single base class.



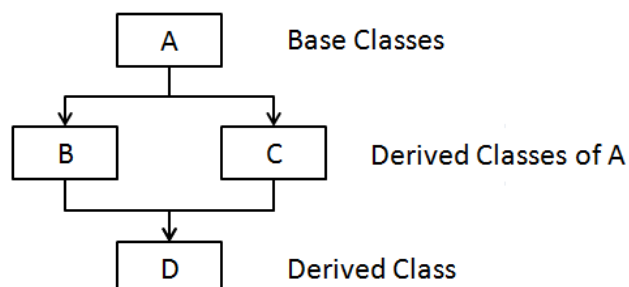
Multilevel inheritance

A derived class is inherited from a class which itself has been inherited from another base class.



Hybrid inheritance

A derived class inherits from multiple base classes which have inherited from the same base class.



Virtual Base Class

Because a class can be an indirect base class to a derived class more than once (example hybrid inheritance), C++ provides a way to optimize the way such base classes work. Virtual base classes offer a way to save space and avoid ambiguities in class hierarchies that use multiple inheritance.

Each nonvirtual object contains a copy of the data members defined in the base class. This duplication wastes space and requires you to specify which copy of the base class members you want whenever you access them.

When a base class is specified as a virtual base, it can act as an indirect base more than once without duplication of its data members. A single copy of its data members is shared by all the base classes that use it as a virtual base.

When declaring a virtual base class, the **virtual** keyword appears in the base lists of the derived classes.

Example:

```
class A{
};

class B: virtual public A{
};

class C: virtual public A{
};

class D: public B, public C{
};
```

Virtual Functions

A virtual function is a member function that you expect to be redefined in derived classes. When a derived class object is referred using a pointer or a reference to the base class, a virtual function for that object can be called and the derived class's version of the function can be executed.

Virtual functions ensure that the correct function is called for an object, regardless of the expression used to make the function call.

Suppose a base class contains a function declared as virtual and a derived class defines the same function. The function from the derived class is invoked for objects of the derived class, even if it is called using a pointer or reference to the base class.

Abstract Base Class

Abstract classes act as expressions of general concepts from which more specific classes can be derived. They are classes that cannot be instantiated (one cannot create objects of an abstract class), and are frequently either partially implemented, or not at all implemented.

A class that contains at least one pure virtual function is considered an abstract class. Classes derived from the abstract class must implement the pure virtual function or they, too, are abstract classes.

Example:

```
class Shape
{
public:
    virtual int area() = 0;
};
```

Polymorphism

In object-oriented programming, polymorphism (from the Greek meaning “having multiple forms”) is the characteristic of being able to assign a different meaning or usage to something in different contexts – specifically, to allow an entity such as an operator, a variable, a function, or an object to have more than one form. Polymorphism is a mechanism that allows you to implement a function in different ways. Function overloading and overloaded constructors are examples of polymorphism.

Different types of polymorphism supported in C++ are -

1. Static polymorphism (compile time)
 - a. Using overloaded functions (and constructors)
 - b. Using overloaded operators
2. Dynamic polymorphism (run time)
 - a. Using virtual functions

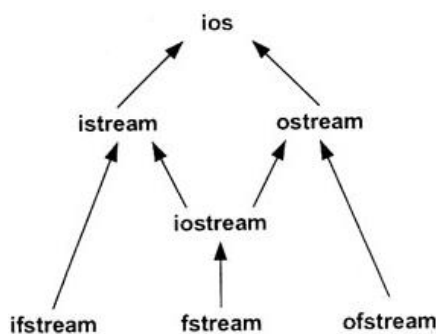
In static polymorphism, the object is bound to its method calls at compile time. This method is also known as early binding. In dynamic polymorphism, the binding happens at run time. When decision as to which object's method should be called while using virtual function does not happen till the call is made. This is also known as late binding.

Working with Files

C++ provides the following classes to perform output and input of characters to/from files:

- `ofstream`: Stream class to write on files
- `ifstream`: Stream class to read from files
- `fstream`: Stream class to both read and write from/to files.

These classes are derived directly or indirectly from the classes `istream`, and `ostream`. To use `ofstream` and `ifstream`, `fstream` has to be included in the program.



The beauty of the C++ method of handling files rests in the simplicity of the actual functions used in basic input and output operations. Because C++ supports overloading operators, it is possible to use `<<` and `>>` in front of the instance of the class as if it were `cout` or `cin`. In fact, file streams can be used exactly the same as `cout` and `cin` after they are opened.

Open a file

The first operation generally performed on an object of one of these classes is to associate it to a real file. This procedure is known as to open a file. An open file is represented within a program by a stream object and any input or output operation performed on this stream object will be applied to the physical file associated to it.

In order to open a file with a stream object we use its member function `open()` -

```
open (filename, mode);
```

The default mode for opening a file with `ofstream`'s constructor is to create it if it does not exist, or delete everything in it if something does exist in it. If necessary, a second argument that specifies how the file should be handled can be passed while opening the file. These parameters are called file mode indicator.

Mode	Description
<code>ios::app</code>	All output operations are performed at the end of the file, appending the content to the current content of the file. This flag can only be used in streams open for output-only operations.
<code>ios::ate</code>	Set the initial position at the end of the file. If this flag is not set to any value, the initial position is the beginning of the file.
<code>ios::in</code>	Open for input operations.
<code>ios::out</code>	Open for output operations.
<code>ios::trunc</code>	If the file opened for output operations already existed before, its previous content is deleted and replaced by the new one.
<code>ios::nocreate</code>	Do not create a new file if it does not exist.
<code>ios::noreplace</code>	Do not replace (overwrite) an existing file.
<code>ios::binary</code>	Open file in binary mode.

All these flags can be combined using the bitwise operator `OR (|)`. For example, if we want to open the file `example.bin` in binary mode to add data we could do it by the following call to member function `open()` -

```
ofstream myfile;
myfile.open ("example.bin", ios::out | ios::app | ios::binary);
```

Each one of the `open()` member functions of the classes `ofstream`, `ifstream` and `fstream` has a default mode that is used if the file is opened without a second argument -

class	default mode parameter
<code>ofstream</code>	<code>ios::out</code>
<code>ifstream</code>	<code>ios::in</code>

fstream	ios::in ios::out
---------	--------------------

Writing to a text file

The following example shows the use of the << operator to write to a text file called “example.txt” -

```
#include <iostream>
#include <fstream>
using namespace std;

int main () {
    ofstream myfile ("example.txt");
    if (myfile.is_open())
    {
        myfile << "This is a line.\n";
        myfile << "This is another line.\n";
        myfile.close();
    }
    else cout << "Unable to open file";
    return 0;
}
```

Reading from a text file

The following example shows reading a line from a text file called “example.txt” -

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main () {
    string line;
    ifstream myfile ("example.txt");
    while (! myfile.eof() ) {
        getline (myfile,line);
        cout << line << endl;
    }
    myfile.close();
    return 0;
}
```

Closing a file

A call to the stream's member function close() is used to close a file. This member function takes no parameters, and what it does is to flush the associated buffers and close the file -

```
Myfile.close();
```

Programmers need not use the close command as it will automatically be called when the program terminates. However it is useful if the file needs to be closed before the program ends.

Detecting End of File

The eof() function can be used to check for end of file. This function return a non-zero value if the end of file condition (EOF) is encountered and zero otherwise.

File Pointers

get and put stream pointers

All i/o streams objects have, at least, one internal stream pointer -

- ifstream, like istream, has a pointer known as the get pointer that points to the element to be read in the next input operation.

- ofstream, like ostream, has a pointer known as the put pointer that points to the location where the next element has to be written.
- Finally, fstream, inherits both, the get and the put pointers, from istream (which is itself derived from both istream and ostream).

These internal stream pointers that point to the reading or writing locations within a stream can be manipulated using the tellg(), tellp(), seekg() and seekp() member functions.

[tellg\(\) and tellp\(\)](#)

These two member functions have no parameters and return a value of the member type pos_type, which is an integer data type representing the current position of the get stream pointer (in the case of tellg) or the put stream pointer (in the case of tellp).

[seekg\(\) and seekp\(\)](#)

These functions allow us to change the position of the get and put stream pointers. seekg() and seekp() have two overloaded versions. The first version moves the pointers to a particular position in the file and the second version moves the pointer by the offset in the direction (forward or backwards) indicated by the direction parameter.

```
seekg ( position );  
seekp ( position );  
seekg ( offset, direction );  
seekp ( offset, direction );
```

Important Questions

Operating Systems

1. What is an operating system? What the key services provided by an operating system?
2. What are the features of Windows 98, NT and Linux operating systems?
3. What are the components of the Linux operating system?
4. What are the different types of operating systems? Give examples of each type.
5. Explain the terms – tracks & sectors, seek time, transmission time, latency or rotational delay.
6. What are the different file allocation methods?
7. Explain the different file system operations.
8. Explain the different process states.
9. Explain context switching.
10. Explain process scheduling? What are the objectives of scheduling? What do you understand by preemptive and non-preemptive scheduling?
11. Explain the following terms – turnaround time, wait time, terminal response time, event response time.
12. Explain the following terms – external priority, internal priority, purchased priority, time slice.
13. What are the functions performed by memory management.
14. Explain the memory map of a single user operating system.
15. Explain fixed and variable partitioning.
16. What is internal and external fragmentation?
17. What is paging? Explain in detail.
18. Write a short note on memory segmentation and virtual memory.
19. Explain the steps involved in allocating a partition in case of fixed partition memory management.
20. Explain the following terms – locality of reference, page fault, working set, page replacement policy, dirty page.
21. What is GUI? Explain any three essential components of GUI.
22. Explain in brief the following components of Windows – Program Manager, File Manager and Control Panel.
23. Explain in brief the function of the menu bar, scroll bar and title bar of a graphic user application.
24. Write a note on security with respect to operating system – confidentiality, integrity and availability.
25. What are computer worms, and viruses? What is the difference between the two? Discuss detection, prevention and removal mechanisms.
26. What are the different ways in which a virus can infect programs?
27. What are system calls?
28. Explain the use of VDU. Why is it called a memory mapped terminal?

Data Structures

1. Explain in brief any six data structure operations.
2. Explain with flowcharts the following control structures – sequence logic, selection logic and iteration logic.
3. What is an array? How are arrays stored in memory? Give the algorithm to traverse elements in an array.
4. Explain bubble sort algorithm with suitable example.
5. Compare linear search and binary search.
6. Explain binary search algorithm.
7. What is a record? How is it different from an array?
8. What are linked lists? Explain with suitable diagram linked list with six nodes.
9. How are linked list stored in memory?
10. What is a binary tree? Define – root node, height of tree, depth of node. What is relation between the depth of a tree and the number of elements?
11. How are binary trees represented in memory?
12. Draw the binary tree structure for given equations.

HTML

1. What is HTML? Give its advantages and disadvantage
2. Important tags in HTML – marquee, sub, hr, pre, body, ul
3. Programming - Working with tables and lists (cellpadding, cellspacing, rowspan, colspan, type)

C++

1. Compare traditional and object oriented programming.
2. Write a note on data types in C++.
3. What are pointers? What are the advantages of using pointers?
4. Explain the use of scope resolution operator.
5. What are functions? How do you define a function in C++? Explain call by value and call by reference.
6. Explain how a memory address of a variable can be accessed in C++.
7. What is Object Oriented Programming? Give its features / advantages.
8. What is a class? How do you define a class in C++?
9. Explain the following – classes, objects, inheritance, data abstraction, encapsulation and polymorphism.
10. How do you define a member function inside and outside of a class definition?
11. What are friend functions? What are the features of a friend function?
12. What are constructors and destructors? What are the rules for defining them?
13. What is operator overloading? Explain difference between operator overloading of member function and friend function.
14. State any eight rules for overloading operators.
15. Explain the three types of data conversions in C++.
16. What is inheritance? Explain the difference types of inheritance.
17. What is polymorphism? Explain runtime and compile time polymorphism.
18. State the rules for defining virtual functions.
19. Describe the various classes available for file operations.
20. What are different file modes?

Programs

1. Read 10 numbers to an array and calculate their sum and average.
2. Write a program to find the first occurrence of a number in a given array.
3. Write a program to read an array from the user and find the greatest number in the array.
4. Print the first n numbers of the Fibonacci series.
5. Write a C++ program to count the number of words in a given text.
6. Write a program to calculate the factorial of a number.
7. Write a program to sort numbers in ascending / descending order.
8. Write a power function that calculates the value of x^p . Function prototype should be `double power(double x, int p)`.
9. Write a program to print the values of an array using pointers.
10. Write a program to calculate the GCD of two numbers. Write methods to read value and calculate the GCD.
11. Write a program to check if a given number is a prime number.
12. Write a program to calculate the volume and surface area of a sphere. The function prototype is given as `void ComputeSphere (float &s, float &v, float r)`.
13. Implement a class circle with a function area to calculate its area.
14. Write a program with `ComputeTriangle()` to calculate the area of a triangle given its three sides.
15. Write a program to replace space with -.
16. Write a program to count the occurrence of 'J' in the given string.
17. Write a program to right align (justify) lines of text.
18. Write a program to calculate the permutation of two numbers. Read the values of n and r from the user.
19. Write a program to print all prime numbers less than n (where n is a natural number specified by the user).
20. Implement a class temperature to convert degree Fahrenheit to degree Centigrade.