

## INTERNET APPLICATION AND EMAIL:

### INTERNET

The Internet is a global system of interconnected computer networks that use the standard Internet protocol suite (TCP/IP) to link several billion devices worldwide. It is a network of networks that consists of millions of private, public, academic, business and government networks of local to global scope, Linked by broad array of electronic, wireless and optical networking technologies. The Internet carries an extensive range of information resources and services, such as the inter-linked hypertext documents and applications of the World Wide Web (www), the infrastructure to support email, and peer-to-peer networks for file sharing and telephony.

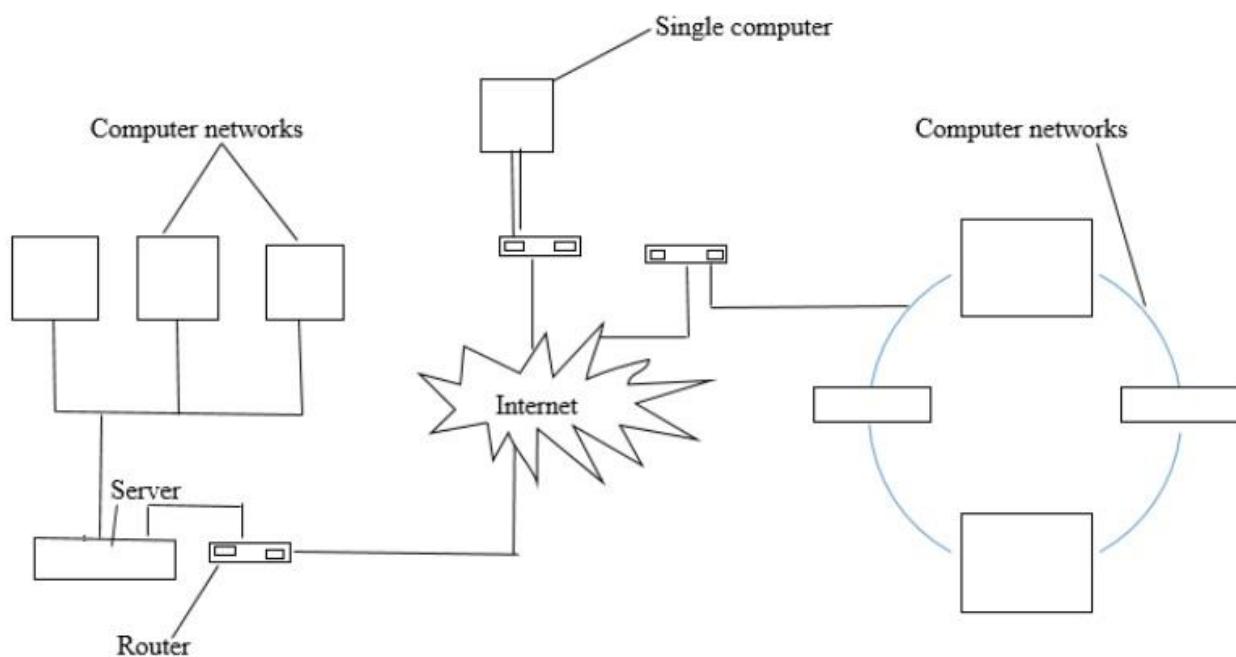


Fig. Logical view of the internet

### Description of internet

Computers within an office or building can be connected together using transmission media such as cables to communicate with one another. Interconnection of computer is referred to as networking.

The term Internet refer to as the global interconnection of computer or networks for the purpose of communication and resource sharing. Internet can be broken down into two words, inter and net which implies that there is an interconnection of networks. Internet enables millions of computers from different organizations and people to communicate and share resources globally.

## HISTORY OF THE INTERNET

### Outline

- Origins of the Internet
- Internet timeline from 1970s until today
- The Internet today

### Origin of the Internet

- In 1966, Roberts went to DARPA and started development of ARPANET
- ARPANET grew to become the Internet we know today

### Internet Timeline: 1970s

- 1971: 15 nodes (23 hosts) all at US locations
- 1972: Ray Tomlinson modifies email program for use on ARPANET; it becomes a big hit
- 1973: First international connections to ARPANET(England and Norway)
- 1976: Elizabeth II sends email
- 1979: USENET (Newsgroups) established
- 1979: Scott Fahlman suggests of emoticons,

:-( and :-(

### Internet Timeline: 1980s

- 1983: ARPANET split into ARPANET(civilian) and MILNRT(military)
- 1983: Berkley UNIX Desktop OS includes IP networking software
- 1984: Domain Name System (DNS) Introduced
- 1985: File Transfer Protocol(FTP) Introduced
- 1987: Number of hosts exceeds 10,000
- 1988: Morris worm
- 1989: 100,000 hosts

### Internet Timeline: 1990s

- 1990: ARPANET ceases to exist
- 1990: Tim Berners-Lee (CERN) invent HTTP and implements server and browser
- 1992: Jeans Armour Polly : surfing the internet
- 1993: NCSA Mosaic takes the internet by storm
- 1994: Pizza Hut takes orders online
- 1995: Sun launches java (including applets)
- 1995: Netscape introduces LiveScript (javascript)

- 1996: Beginning of Browser Wars (Netscape vs Microsoft)

### **Internet Connectivity Requirements:**

The first step to accessing and using the Internet is to get connected. This section seeks to discuss what is required to make internet accessible.

#### **Data terminal equipments**

Data terminal equipments are devices used to process, host and transmit data on a network. Examples are computer mobile phones and personal digital assistants (PDAS)

#### **Transmission Media**

Transmission media are physical or wireless pathways used to transmit data and information from one point to another. Example of transmission media are the telephone lines, Radio waves, microwaves and satellites.

#### **Telecommunication lines.**

A computer is connected to the internet using a telephone line and has to dial a remote computer in a device known as a modem.

Modem change data from analog to digital for the computer to understand it through a process called demodulation.

#### **Satellite Transmission**

Intercontinental transfer of data is achieved by having satellite base stations transmitting the data through a wireless up link channel to the satellite. The satellite then sends the data signal to the base station on another continent where it is picked up and sent to telephone exchanges for transfer to the destination computer

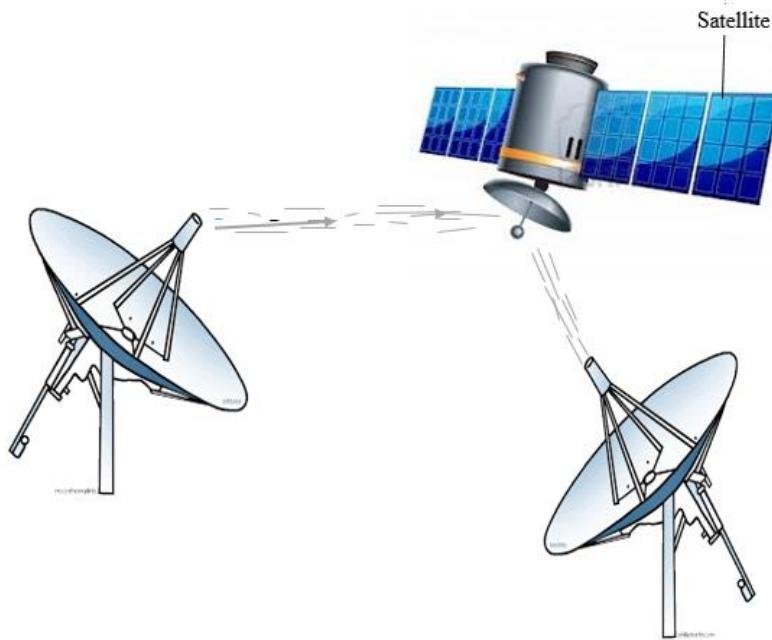


Fig. Internet connectivity infrastructure.

### Internet Timeline: 2000s

- 2000: A massive DoS attack is launched against Yahoo, Amazon and eBay
- 2000: Code Red worm and Sircam virus infect thousands of web servers and email accounts
- 2002: Weblogs (blogs) become hip
- 2003: Year of viruses and worms: Slammer, Sobig. F, and Blaster

### Internet Timeline: Today

- Most widely-used Internet protocol: HTTP
- Most popular web browsers (Nov.2006):
  - Internet Explorer: 59%
  - Firefox: 30%
  - Mozilla, Netscape, Opera: 5%
- Highest bandwidth consumption: peer-to-peer file sharing protocols

### Summary

- Internet started out as US DoD project (ARPANET)

- Major Internet protocols and formats
  - Email(1973)
  - Newsgroups(1979)
  - File Transfer protocol(1985)
  - Archie, Gopher(1990s)
  - Hypertext Transfer Protocol(HTTP-1990,1993)
  - Hypertext Markup Language(HTML-1993)
  - Real Simple Syndication(RSS- 1999)
- Major additions to HTTP and HTML
  - Java(1995), JavaScript(1995), Flash and Shockwave(1996)

## The Hypertext Transfer Protocol

### Outline

- History of HTTP
- The HTTP Protocol
  - HTTP Requests
  - HTTP replies
- Summary

### HTTP Protocol Summary

- A typical HTTP transaction:
  - Client (browser) open connection to server
  - Client sends request to server
  - Server processes request
  - Server replies to client

- Client and server close connection
- Variants
- Connection stays open
- A proxy sits between client and server

## The HTTP request

- Requests includes a header and optional body, separated by a blank line

GET/~morin/index.html HTTP/1.1

Host: localhost: 3128

User-Agent: Mozilla/5.0(X11; U; Linus i686(x86\_64); en-US; rv: 1.8.0.9)

Accept:

Text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,Accept-Language:en-us,en;q=0.5

Accept-Encoding:gzip, deflate

Accept-Charset: ISO-8859-1,utf-8;q=0.7\*:q=0.7

Keep-Alive:300

Connection:keep-alive

## Main HTTP Request Types

- GET
  - Get (retrieve) a document
- HEAD
  - Same as GET, but only retrieve header
- PUT
  - Put information on the server
- POST

- Send information to the server
  - OPTIONS
- Get information about the server

## The HTTP Response

- Request includes a header and optional body

HTTP/1.1 200 OK

Date:Wed ,03 jan 2007 12:10:27 GMT

Server:Apache/2.0.54(Fedora)

Last-Modified: Tue,02 jan 2007 16:01:12 GMT

ETag:"548268-1b2d-d6ce0200"

Accept-Ranges:bytes

Content-Length :6957

Connection: close

Content-Type: text/html; charset=UTF-8  
"http://www.w3.org/TR/html4/loose.dtd">

.....

## Common Response Codes

- 200 OK
- Success
  - 404 Not Found
- The specified document does not exist
  - 403 Forbidden

- The specific document exists, but cannot be accessed.
  - 301 & 302 Document Moved
- Document is at the new(specified) location
  - 4\*\* Error
- Any 4\*\* message indicates an error has occurred.

## **Summary**

- The basic HTTP transaction is simple request from client followed by a reply from the server
- Both requests and replies have header followed by an optional body
- 200 indicates a success
- 4xx indicates an error
- We will see more complicated transactions and header field later on

## **Browsers**

A browser is a program that lets the user surf or browse the internet. Some examples include Microsoft Internet explorer, Mozilla Firefox and Netscape navigator.

## **E- Mail software**

Email client on the other hand is a software that enables the user to receive, compose and send e-mails. Example are Microsoft outlook Express, Eudora, Yahoo mail, Gmail etc

## **Electronic mail and fax**

An electronic mail simply referred to as e-mail is a message transmitted electronically over the internet. All you need to receive and send emails is to open an e-mail account.

## **Using electronic mail**

To create, receive and send electronic mail (e-mail), a special e-mail program such as Microsoft outlook Express, yahoo mail, Gmail and Eudora are required. To launch an email program from start menu Task bar.

## **Checking for Mails in Yahoo**

To use Yahoo mail, you first sign up for an email account. Yahoo assign each user a unique user name referred to as an ID and a password.

## To Open an email in Yahoo mail

- 1 Sign in using your yahoo ID and password
- 2 Click check mail button then inbox.
- 3 In the inbox list, check the subject of the mail to read
- 4 Read the mail. Open an attachment if any.

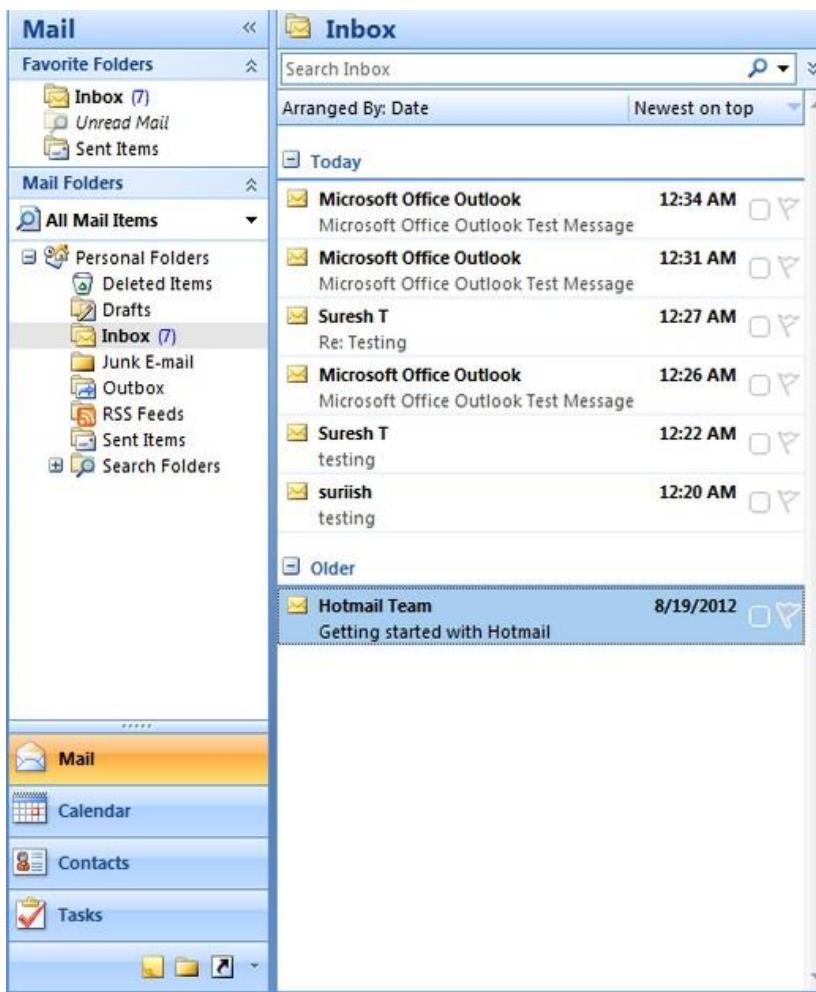


Fig show Yahoo mail inbox.

## To Open an e-mail in outlook Express

### Getting started with Hotmail

Hotmail Team [member\_services@live.com]

Click here to download pictures. To help protect your privacy, Outlook prevented automatic download of this picture from the Internet.  
Hotmail

Sent: Sun 8/19/2012 2:25 PM  
To: suriish@hotmail.com

Right-click here to download pictures. To help protect your privacy, Outlook prevented automatic download of this picture from the Internet.  
Hotmail

Right-click here to download pictures. To help protect your privacy, Outlook prevented automatic download of this picture from the Internet.  
Hotmail

Welcome to Hotmail. The efficient way to...

Here are some tips to get you started:

Bring in your contacts over to Hotmail (that'll make it easier to find them).  
Read messages from your other email accounts right here in one place.  
Make your inbox feel like home, choose a theme.

### Go beyond the basics

Take your email on the road. Go mobile.  
Chat with your Messenger friends from your inbox or calendar.

Microsoft respects your privacy. To learn more, please visit [www.microsoft.com/privacy](#).  
Microsoft Corporation, One Microsoft Way, Redmond, WA 98052-6399.  
Microsoft Corporation. All rights reserved.

1 Launch outlook Express from the task bar.

2 On the Folders list ,click inbox

3 Read the mail. Open an attachment if any

The screenshot shows the Microsoft Outlook Express interface. The top menu bar includes File, Edit, View, Go, Tools, Actions, and Help. Below the menu is a toolbar with New, Reply, Reply to All, Forward, Send/Receive (which is highlighted with a red box), and Search address books. The left sidebar displays Mail, Favorite Folders (Inbox, Unread Mail, Sent Items), Mail Folders (All Mail Items, Personal Folders like Deleted Items, Drafts, Inbox, Junk E-mail, Outbox, RSS Feeds, Sent Items, Search Folders). The main pane shows the 'Inbox' folder with messages arranged by date (Newest on top). The messages are:

- Microsoft Office Outlook 12:34 AM Microsoft Office Outlook Test Message
- Microsoft Office Outlook 12:31 AM Microsoft Office Outlook Test Message
- Suresh T 12:27 AM Re: Testing
- Microsoft Office Outlook 12:26 AM Microsoft Office Outlook Test Message
- Suresh T 12:22 AM testing
- suriish 12:20 AM testing
- Hotmail Team 8/19/2012 Getting started with Hotmail

To the right of the inbox, there is a preview pane titled 'Getting started with Hotmail' from the Hotmail Team. It includes a note about download protection, a welcome message, and tips for starting. Below the inbox, there is a sidebar with links for Mail, Calendar, Contacts, Tasks, and a bottom status bar showing '7 Items'.

### Email Address Format:

Just like the normal postal address, an email address determines the destination of the email sent.

A typical email address would look like this:

chemwer@yahoo.com

1. chemwer is the user name and is usually coined by the user during email account registration.
2. @ is the symbol for “at” which separates the user name from the rest of the address
3. Yahoo.com is the name of the host computer in the network
4. The period “.” Is read as dot and is used to separate different parts of the e-mail address
5. Com identifies the type of organization offering a particular services

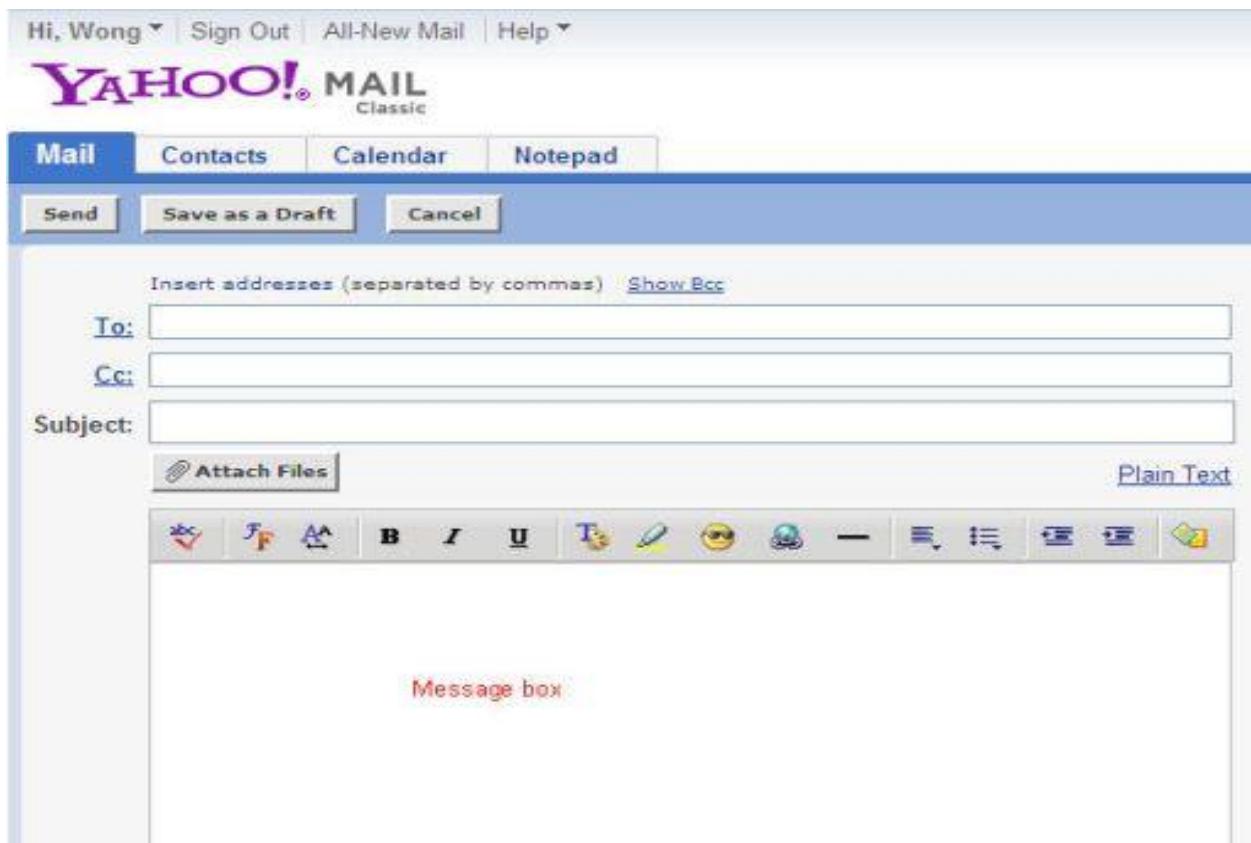
and is called the domain ,meaning it is a commercial institution other commonly used domains include

<u>Domain</u>	<u>Type</u>
.edu	Education Institution
.gov	Government Institution
.org	Non-profit making organization
.mil	Military organization
.al	An academic Institution

Composing and Sending email.

To compose an email in either Yahoo mail or outlook Express:

1. Click the compose button
2. Type the recipient address or get it from the address book.
3. Type in the subject of the message
4. Type in the message in the message
5. Click the send button

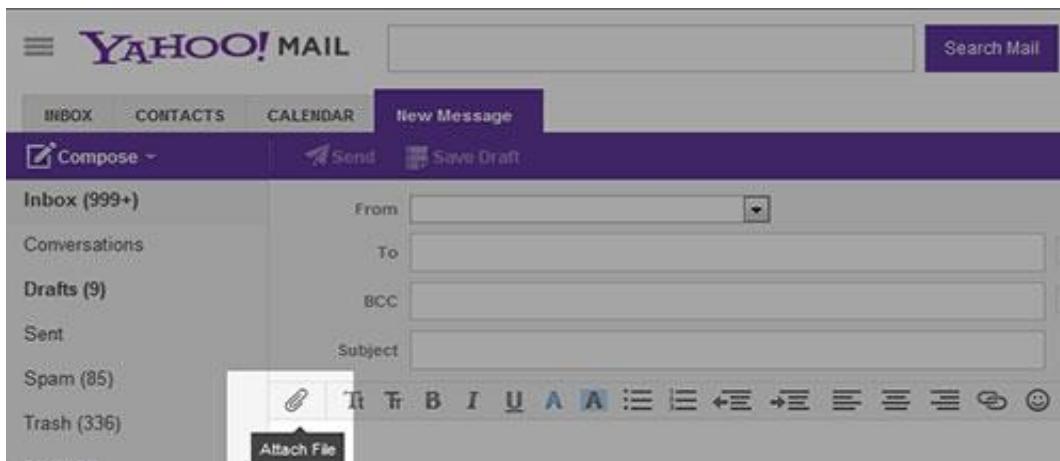


## Attaching files into e-mail

Just the way you would attach a document such as a resume ordinary mail, You can as well attach a file to an e-mail. A file may contain pictures, videos or text:

To attach a file to an email:

1. Click the attachment button on the toolbox
2. In the Dialog box that appear, select the file (s) to attach
3. Click Attach button to attach the files



**MICROSOFT**

**OUT**

**LOOK**

## Introduction to Microsoft Outlook

Microsoft Outlook is a comprehensive tool that enables you to manage your email, calendar, contacts, tasks and more all in one place.

You probably already know that you can use Microsoft Outlook to manage your email, but did you know that Outlook also includes tools to help you manage your schedule, tasks, contacts, and notes? One of the major advantages to using Outlook is its integrated tool set, yet many people aren't aware of all of Outlook's capabilities. Take some time to get acquainted with some of the useful, integrated tools Outlook offers.

## The Power of Integration

One of the major selling points of Outlook is its seamless integration between Outlook's various tools, as well as other Microsoft products. For instance, let's say you receive an email and you would like to follow-up on it at a later date. You can simply flag the email for follow-up; this will automatically add the email to your Task list. You can also tie Outlook into other Microsoft products like Microsoft SharePoint (Microsoft's premiere collaboration suite). For example, let's say that you have a remote team that is using SharePoint for online collaboration. You can easily sync Outlook tasks lists and calendars with SharePoint; i.e., if team members add tasks/events to SharePoint, these items will appear on in Outlook for you.

## Outlook Basic Features

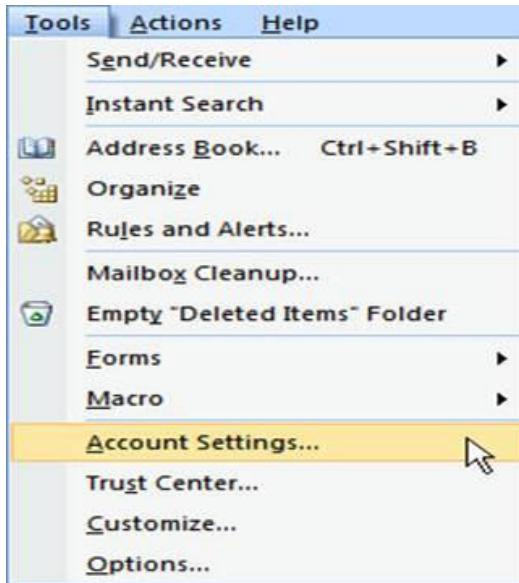
### 1. Email Management

The most common use for Outlook is email management. Whether you are interested in the basics of sending/receiving mail, or want to figure out how to create filters and/or rules to help you automatically organize your email, be sure to check out this guide's wide range of email management topics.

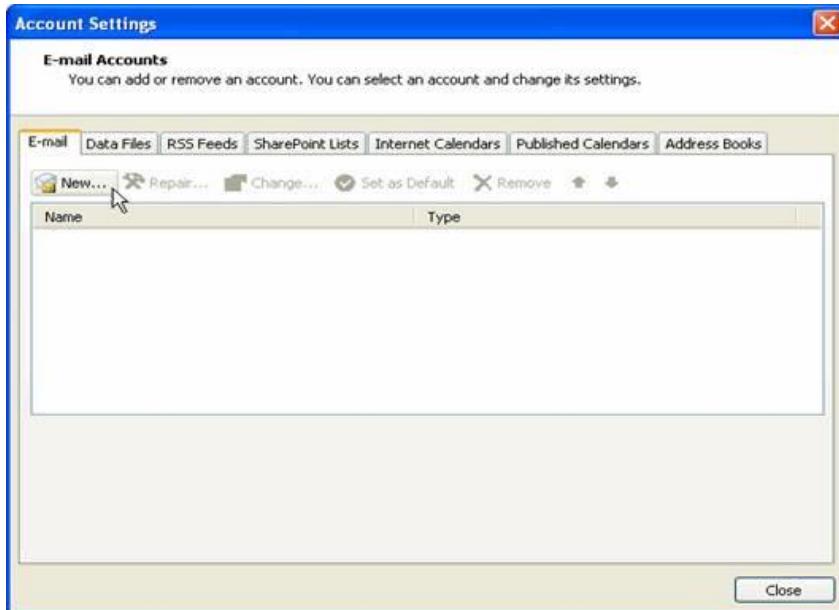
## How to Set Up Outlook 2007 for Windows to Send and Receive Email (Wizard)

- **Operating System(s):** Windows XP/ Window 07
- **Application:** Microsoft Outlook
- **Application Version(s):** 2007

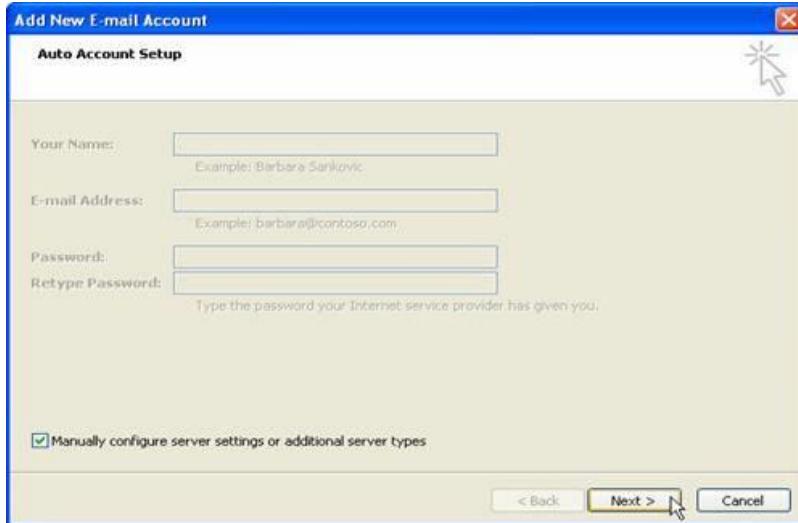
1. Open Outlook. Select **Account Settings...** from the **Tools** menu.



1. On the E-mail tab, click **New**.



2. Select "Manually configure server settings or additional server types" and click **Next >**.



3. Select **Internet E-mail** and click **Next >**.



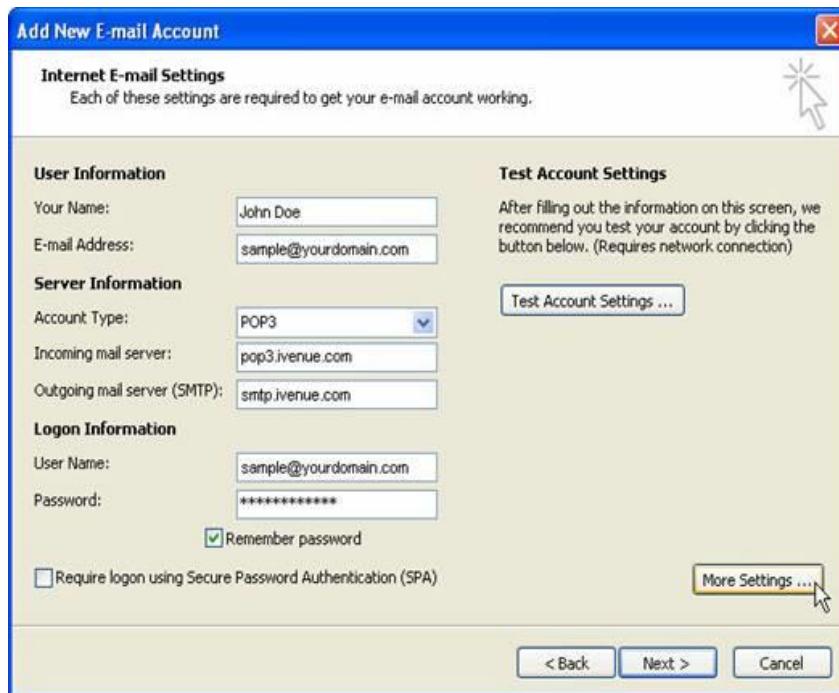
4. Enter the following information for E-mail Accounts.

- A. **Your Name:** Enter the name you wish recipients to see when they receive your message.
- B. **Email Address:** This is the address that your contacts' email program will reply to your messages. This is also the address that will get recorded in your contacts' address book if they add you as a contact.
- C. **Account Type:** POP3
- D. **Incoming mail server:** Enter pop3.yahoo.com
- E. **Outgoing mail server (SMTP):** Enter smtp.yahoo.com

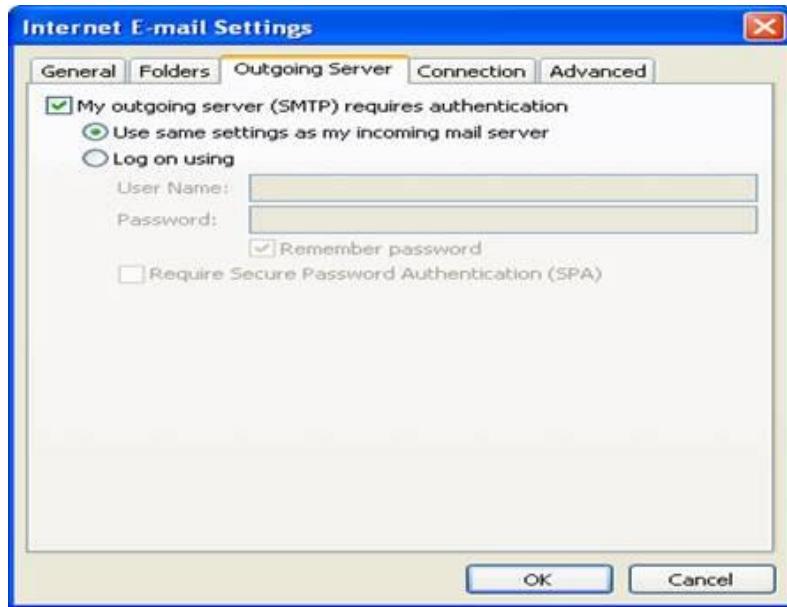
F. **User Name:** Enter your full e-mail address

G. **Password:** If you wish for Outlook to save your password, check the box labeled **Remember Password** and enter your password in the text field.

H. Click **More Settings...**



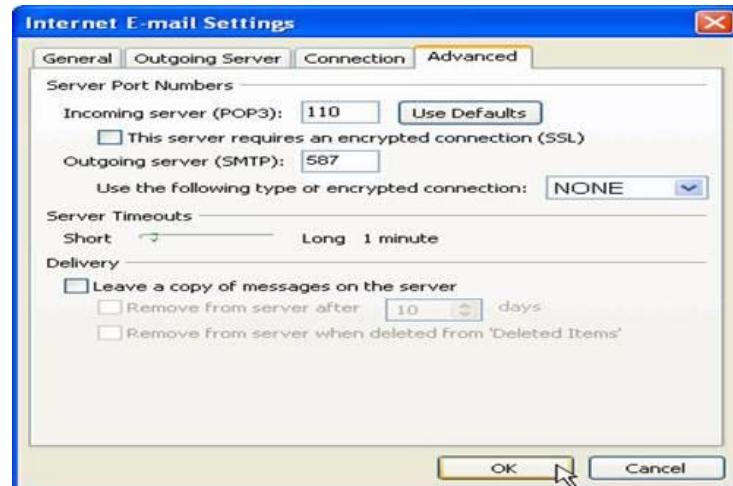
5. Click on the Outgoing Server tab, and check the box labeled **My outgoing server (SMTP) requires authentication**. Then choose to **Use same settings as my incoming mail server**



6. Click on the Advanced tab.

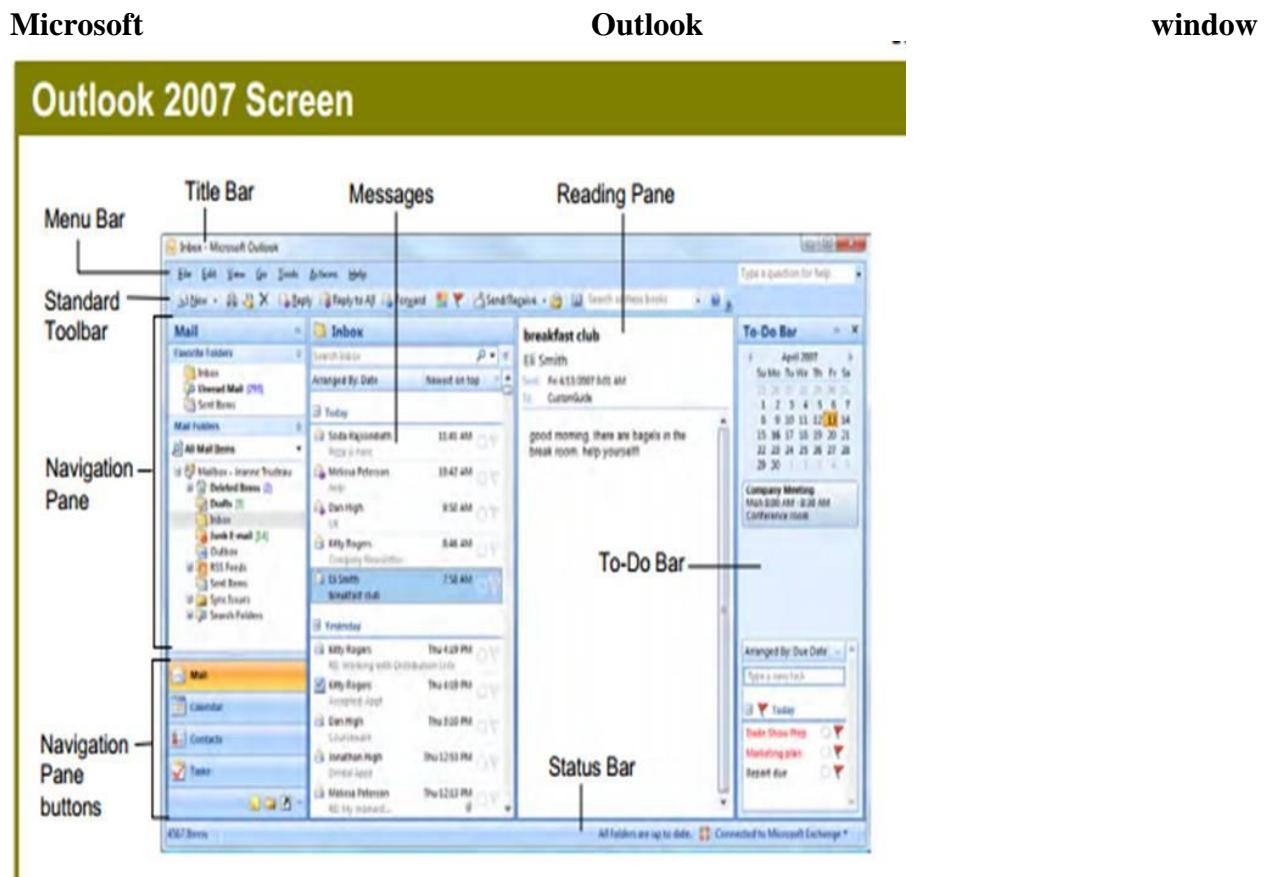
-Under **Incoming Server (POP3)**, the port number should be set to **110**.

-Under **Outgoing Server (SMTP)**, the port number should be set to **587**.



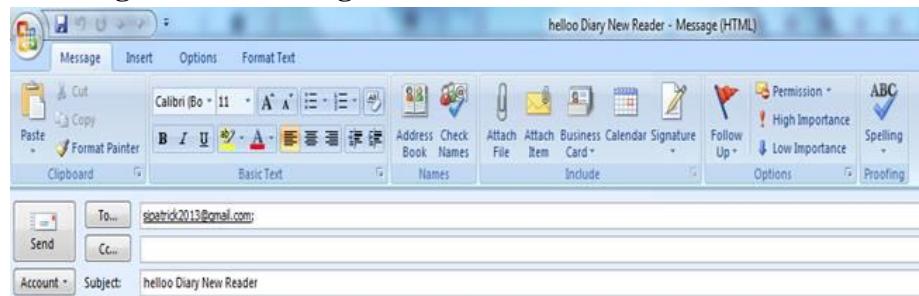
7. Click **OK**

8. Click **Next**. Click **Finish**.



## EMAIL MANAGEMENT

### Creating and Addressing Emails



### Composing a New Email

- To compose a new email, click the **New** button on the Standard Toolbar. A blank 'Message' window appears.
- In the field next to the '**To...**' box, type in the e-mail address of the person you wish to send mail to; or, click the '**To...**' button to search for contacts in the UNC Directory.
- Clicking '**To**' adds a selected person as the primary recipient of your e-mail, and '**Cc**' (Carbon copy), adds a selected person as a secondary recipient.



- Enter the subject of the email in the **Subject** box, then type the body of your message in the text box below.



## Addressing Email

- To add a contact from the UNC Directory, click the 'To...' button. In the Select Names window that pops up, enter the name of the person you wish to send mail to in the Search box in the format of **Last name, First name** (eg: Doe, John). Make sure the Address Book is set to "**Global Address List**".



- Alternatively, you can type the name of the person you are searching for in the **To...** field and then click on the **Check Names** button (or **Alt + K**) to find that person's email address.

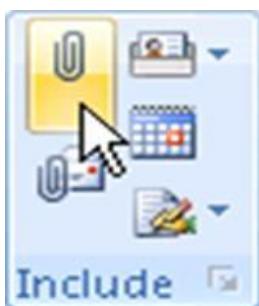
## Adding a BCC (Blind Carbon Copy) Field



- **Bcc** is short for Blind carbon copy. If you add a recipient's name to this box in a mail message, a copy of the message is sent to that recipient, and the recipient's name is not visible to other recipients of the message. If the Bcc box isn't visible when you create a new message, you can add it. The option to send a Bcc is under the Options tab in the Ribbon of the Compose Mail screen. Go to **Options** > **Show Bcc**. The field will appear under Cc...

## Adding Attachments

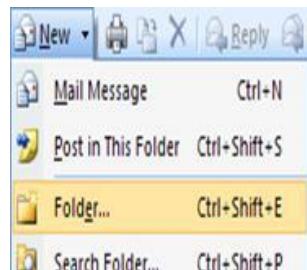
### Procedures for Adding Attachments



- To add an attachment to an e-mail message, click **Attach File** from the **Include** group on the Ribbon (note: you must be on the default Message tab of the ribbon in order to see this). In the Insert File dialog box, browse to and select the file that you want to attach, and then click **Insert**.
- An alternative way of adding an attachment is to click on the **Insert** tab on the Ribbon and select **Attach File** (other attachment types are also available).
- You can attach multiple files simultaneously by selecting the files and dragging them from a folder on your computer to an open message in Outlook.
- **Note:** The maximum size for an email attachment within the School of Medicine is **50MB**. If you wish to email a larger attachment we suggest you use the [File Upload Tool](#).
- If you wish to remove an added attachment simply click on it once in the **Attached** field to highlight it and then hit the delete key on your keyboard.

## Organizing Your Mail

### Using Folders to Organize Your Mail



As your Inbox grows in the number of messages it holds, it will become necessary to organize your email messages. By default you have several folders which are used to organize the items you have in your mailbox. Folders such as the Inbox, Deleted Items, Drafts, Sent Items, etc. Outlook also gives you the ability to create custom folders.

Outlook Mail contains default folders such as Inbox, Deleted Items, Drafts, Sent Items, etc to help you organize your email. Outlook also gives you the ability to create custom folders. As your Inbox grows in the number of messages it holds, it will become necessary to create custom folders and organize your email messages.

- To create a new folder, click the **drop-down arrow** next to the 'New Mail Message' button on the Standard Toolbar. From there, select 'Folder...', then choose the location of this new folder. (Recommended location: 'Inbox' if you might need to access this folder from the web; 'Archive Folders' if not).

### **Using Rules to Organize Your Mail**

Rules are an automated way of organizing your email. Once you set a rule, Outlook will automatically perform certain actions when it receives email that matches the criteria you've specified. Messages with a certain subject line or sender can be rerouted to folders for organizational purposes. Advanced information about customizing rules can be found at the University of Wisconsin-Eau Claire's site, Outlook.

### **Personalizing Mail with Signatures**

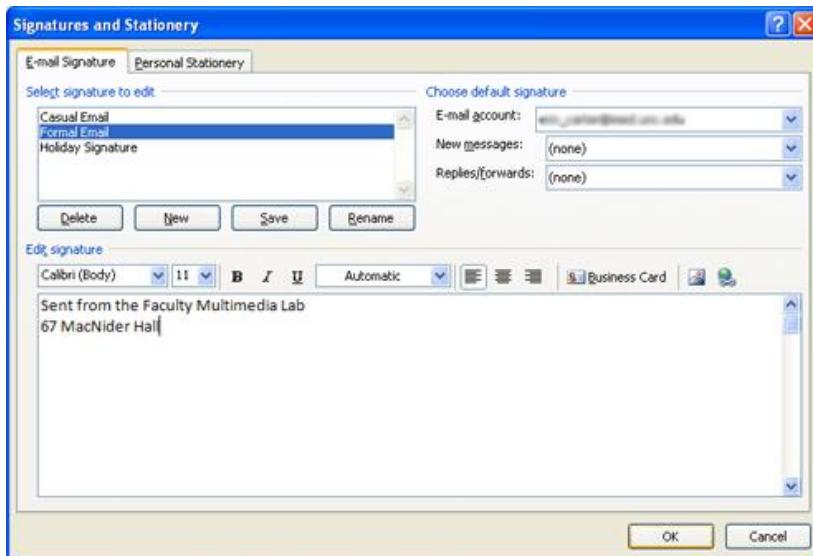
A signature is a block of text appended at the bottom of an e-mail message often containing the sender's name, address, phone number, disclaimer or other contact information. You can use Outlook's Signature feature to add a personal signature to all of your messages so you don't have to repeatedly type the same information in all of your emails.

You can create as many signatures as you'd like. You can also configure Outlook to automatically add a signature to outgoing messages, or you can manually add the signature whenever it's needed. You can even create custom signatures for different types of audiences.

### **Create a Signature**

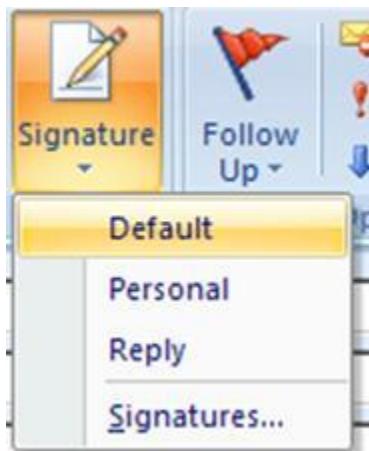
1. Click **Tools**, then **Options**. This will open the Options dialog box.
2. Click on the **Mail Format** tab, then click on the **Signature** button, about 3/4 of the way down the dialog box. This will open the Signatures and Stationery dialog box.
3. Type a name for the signature, and then click **OK**.
4. In the Edit signature box, type the text that you want to include in the signature. To change fonts or font sizes, add bold or italics etc, simply select the text and then use the formatting buttons and drop down boxes above the text area.
5. Define which signature to use in the **New Messages** and the **Replies/Forwards** fields.

6. Click **OK** when you are finished.



## Insert a Signature Manually

In the Ribbon of a new message, click the **Signature** button, and then select the signature you wish to use.



## Remove an Automatic Signature from a Message

In the body of the message, select the signature, and then press the **Delete** key on your keyboard.

### Email Quota and Keeping it in Check

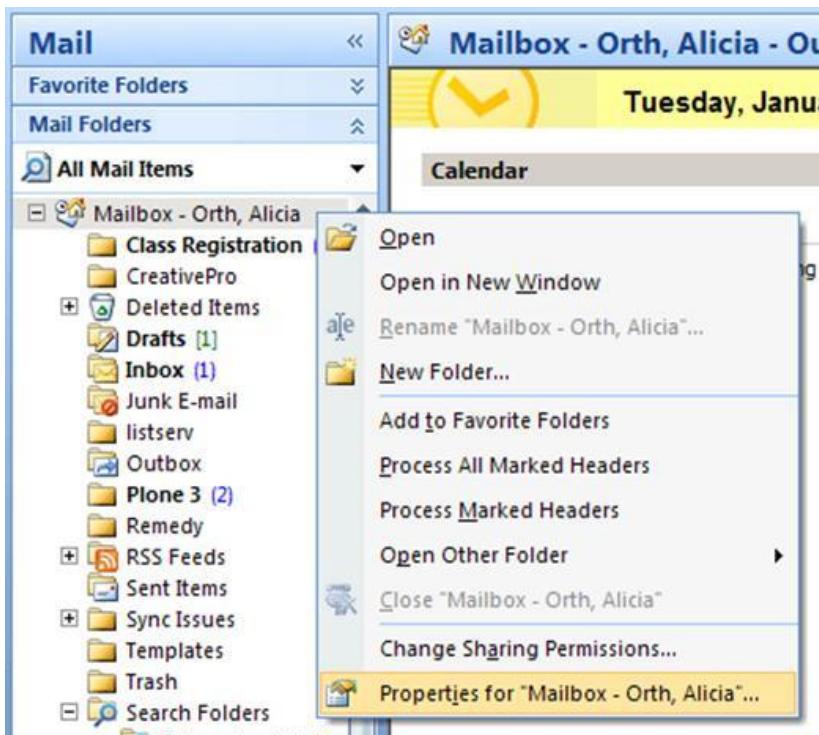
Storage on the mail server is limited to **2GB (beginning at the end of October 2010)** per user. **Once that quota has been exceeded, you will no longer be able to send or receive email.**

Be sure to check the amount of space you are using periodically to avoid interruptions in your email service.

## Checking Mail Quota from Outlook 2007

### Option One

1. In the **Mail Folder List** locate the entry **Mailbox - Your Name** and right-click on it.
2. Select **Properties for Mailbox-YourName**



3. The Outlook Today window will display.
4. Select the **General** tab and click on the **Folder Size...** button.
5. Click on the **Server Data** tab. This will display the total disk usage in a folder-by-folder breakdown.

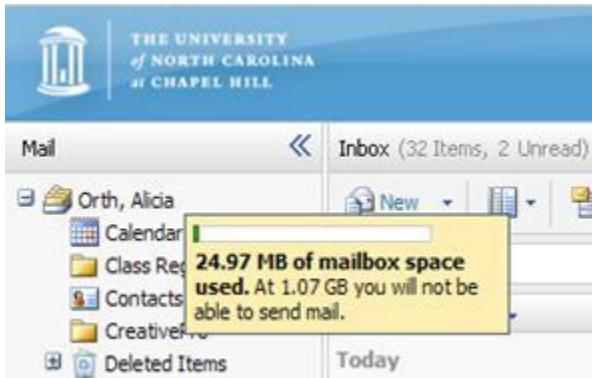
### Option Two

1. Click on the **Tools** menu
2. Select **Mailbox Cleanup**
3. Click on the **View Mailbox Size** button
4. Click on the **Server Data** tab

## Checking Mail Quota From a Web Browser

1. Using Internet Explorer 8 (or later), navigate to [outlook.unc.edu](http://outlook.unc.edu) and log in using your Onyen. **Note:** this will not work in other browsers.
2. Hover your mouse over your name in the top left of the Mail pane.

3. After a second or two, a popup window will display your disk usage.



### Freeing Up Space

To ensure that you receive all your email messages, it is important that you regularly clean out your Email. A way to keep your quota smaller than the size limit is to delete messages and/or move messages from the server to locally stored files.

**Tip:** Use the Large Mail Search Folder to easily find and delete large email messages. Click here to learn more about [Search Folders](#).

### Recovering Deleted Emails

If you are looking to recover a message deleted from the Exchange server you should first look in the **Deleted Items** folder for that message. When you delete an email message from any folder other than the Deleted Items folder that message is sent to the Deleted Items folder. Any message deleted from the Deleted Items folder is considered permanently deleted but can be restored within 14 days of its deletion following this procedure:

1. From the Outlook **Mail** panel, select the mailbox from which the email was deleted (for example, your Inbox).
2. Go to the **Tools** menu and select **Recover Deleted Items...**
3. The pop-up window will display a list of all items deleted from the selected folder within the past 14 days.
4. Select the item(s) you wish to restore. You can shift-click or control-click to select multiple items, or use the Select All button .
5. Click the **Recover Selected Items** button . The selected emails will be returned to their original folder.
6. Caution: the **Purge Selected Items** button  will permanently remove any selected messages from Outlook.

**Note:** This only works for messages deleted from the Exchange server and not for messages deleted from a local folder.

### Server Files vs. Personal/Local Files

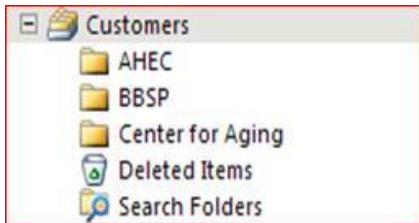
Initially all incoming mail (as well as contacts, appointments, tasks, notes, and journal entries) is delivered to and stored on the Exchange Server. You however, have the option of creating and moving your messages to a Personal Storage folder (.pst). One of the biggest benefits of a Personal Storage folder is that it is not subject to the 2GB (beginning at the end of October 2010) mailbox size limits that are set on the server.

#### Server Files

1. **Storage Location:** on the server
2. **Size Limit:** 2GB (beginning at the end of October 2010)
3. **Accessibility:** can be accessed from your work machine (through Microsoft Outlook) or accessed anywhere via the internet (log on [tooutlook.unc.edu](http://tooutlook.unc.edu)).
4. Messages that you'll want continuous access to should be kept here.

#### Personal / Local Files

1. **Storage Location:** your work computer
2. **Size Limit:** the available space of your hard drive
3. **Accessibility:** can only be accessed when at your work machine
4. This is generally a good place for messages that you no longer need to act upon but want to keep for future reference.



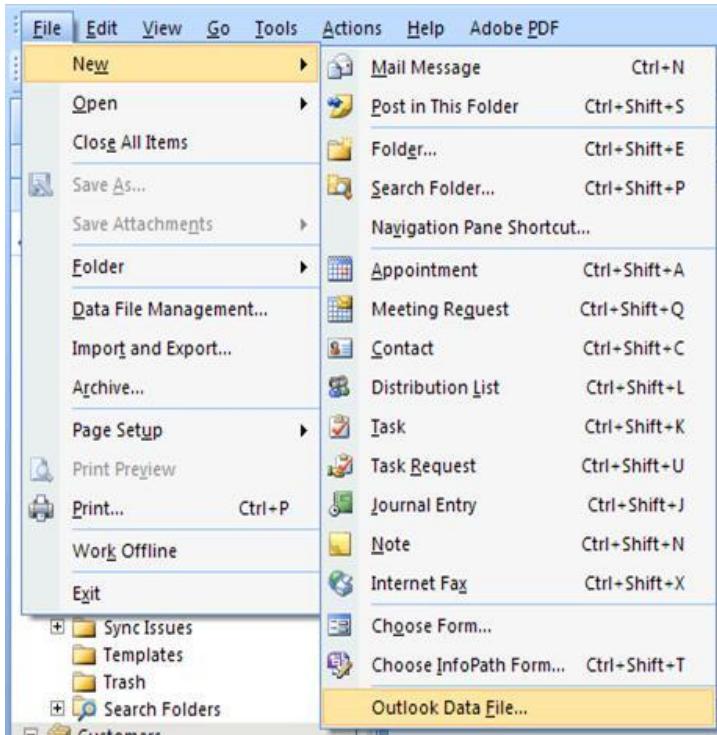
**Example:** as a graphic designer working at the School of Medicine, I work with a lot of different customers. Once I complete a job for a customer I no longer need to keep the email correspondence but I do just in case I need to refer back to it (for billing purposes or whatever). Since I rarely have to reference these old files I created a Personal Storage Folder entitled 'Customers' and then created a separate folder under that for each customer. It's an easy way of keeping and organizing needed messages without taking up my allotted server space.

### How to Create a Personal Storage Folder

1. On the File menu, point to **New**, and then click **Outlook Data File**.
2. Ensure **Office Outlook Personal Folders File (.pst)** is highlighted, and then click **OK**.
3. In the **File** name box, type a name for the file, and then click **OK**.
4. In the **Name** box, type a display name for the .pst folder, and then click **OK**.

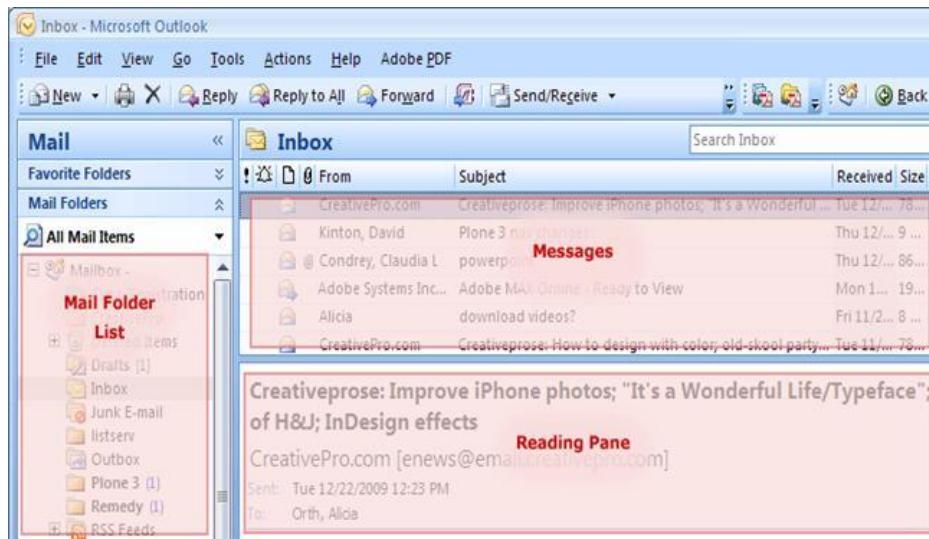
**Note:** you can set a password for the folder but it is critical that you remember it as it cannot retrieve.

- Now you can drag and drop messages from your server folders to the new folder. Press CTRL while dragging to copy items instead of moving them.



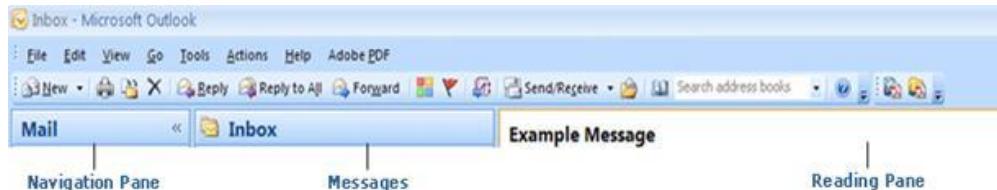
## Navigating the Inbox The Inbox: Panes & Bars

Outlook enables you to customize the way mail is presented by simply choosing **View** (from the top menu bar) and then **Current View**. In the example below, the current view is the standard "Messages" view. We've also chosen to position the "Reading Pane" underneath messages (see "Positioning the Reading Pane" below for more information). Although your Outlook screen may not look exactly like the screen below, the basic concepts are still the same.



- The **Mail Folder List** (far left) allows you to navigate through your various mail folders and view their contents.
- Your **Messages** (center top) displays the name of the selected folder (in this example it is the Inbox), and a list of messages it contains.
- The **Reading Pane** (center bottom) shows the text of a selected email message.

### Positioning the Reading Pane



In Outlook the Reading pane can either be displayed to the right of the Messages pane or under it. If you wish to change this display setting, go under View > Reading Pane. Select 'Bottom' or 'Right.'

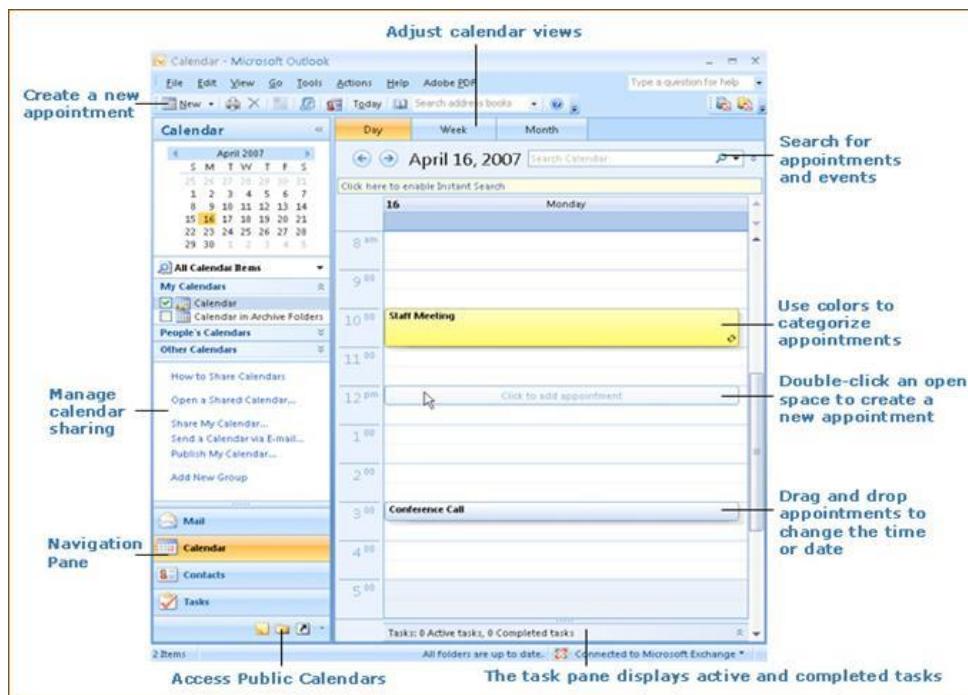
### Change the Size of Text in the Reading Pane

If you are experiencing problems reading a message and wish to enlarge the font size for visibility purposes you can make the text or font in the Reading pane larger or smaller by using the scroll wheel on your mouse.

Click in the Reading Pane, press CTRL, and roll the scroll wheel. Rolling the wheel away from you makes the text bigger, rolling it towards you makes the text smaller.

## OUTLOOK CALENDAR

With the Outlook 2007 Calendar feature, you can organize your schedule and integrate communications, tasks and calendar items in one location. Click on the Calendar section of the Navigation Pane to display your calendar.

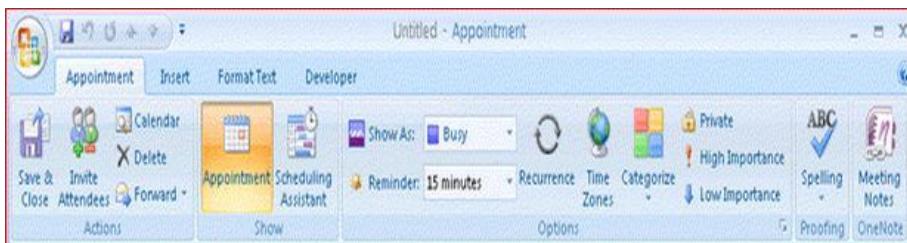


## CREATING AN APPOINTMENT IN OUTLOOK CALENDAR

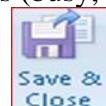
Create one-time or recurring appointments using Outlook 2007.

### To create an appointment:

1. Click the **New** button to open the **Appointment** dialog window. Alternatively, you can double-click on the date in the calendar where you want to set an appointment.
2. Enter a **subject** and **location** (if applicable) for your appointment and select the **date** and **time**. For an all day events, you can select **All day event**.
3. In the Appointment dialog window you can use the options listed in the **Ribbon** to customize your appointment:



- **Appointment Status (Show As):** Choose between Free, Tentative, Busy or Out of Office.
- **Reminder:** Choose a time for a appointment reminder.
- **Recurrence:** Click this button to show options for recurring events.
- **Categorize:** Keep your appointments organized by using Categories.
- **Private:** Keep your appointment details private. If you select this option others will only see "Private Appointment" during the specified time and a color bar along the left of the appointment to denote the appointment status (busy, out of office, etc.).



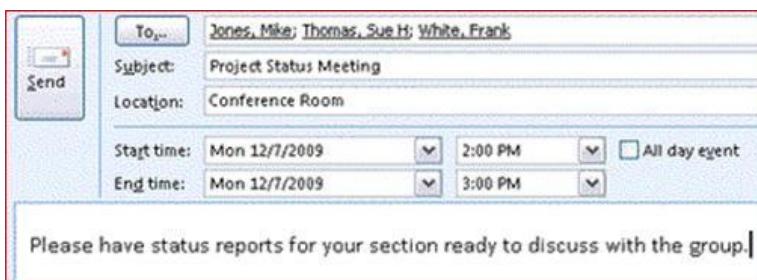
4. To set your appointment, click **Save & Close**.

## USING THE SCHEDULING ASSISTANT

*Outlook 2007 has a scheduling assistant feature that allows you to schedule a meeting with your colleagues.*

### Sending Meeting Requests

1. In the **Calendar** section, click the drop-down arrow next to the **New** button and select **Meeting Request** to open the **Meeting Request** dialog window.



2. Enter the names or e-mail addresses of your invitees in the **to...** field (separated by a semicolon). If recipients are listed in your **Address Book**, you can click the **to...** button and select them



**Tip:** Entering names manually will list each person as a **required attendee**. Alternatively, click the **To...** button, select a name in the **Name** list and click **Required** or **Optional**.

3. Enter a subject, location and date/time for your meeting.
4. In the body of the message, you can enter a message to your recipients or additional meeting information.
5. Click **Send** to send your invitation.

### Using the Scheduling Assistant

The **Scheduling Assistant** will help you determine when all participants are available for a meeting.

1. In the **Meeting Request** dialog window, click on the **Scheduling Assistant** button (found in the **Meeting** tab in the **Show** section).
2. This will show the invited attendees, as well as when they are busy, out of the office or available. Click a row under the **All Attendees** column to add additional participants.
3. Click on a block of time when all participants are free. You can adjust the green and red start/end times by clicking and dragging them left or right.



**Tip:** You can also click on one of the Suggested Times on the right panel.

Appointment color indicators are as follows:

1. **Blue:** The time is marked as **busy**.
2. **Blue and White Striped:** The marked time is scheduled with **tentative** appointments.
3. **Purple:** The time is marked as **out of office**.
4. **Black and White Striped:** Outlook has **no information** for the time period marked. This most likely means that the attendee is not an Outlook/Exchange user within the UNC system.
4. Click on the **Appointment** button on the **Ribbon** to return to your appointment window (found in the **Meeting** tab in the **Show** section).
5. Click **Send** to send your invitation.

## RESPONDING TO MEETING REQUESTS

When a **Meeting Request** is sent, an E-mail message is sent to all invited attendees. In the message, they have the option to **accept**, **tentatively accept**, **decline** or **propose a new time** for the meeting. The attendee can click on **Calendar...** to show a copy of his or her calendar with the meeting scheduled.

✓ Accept | ? Tentative | X Decline | ⏲ Propose New Time | 🗓 Calendar...

**Web Site Discussion**

Doe, John

Please respond.

**Required:** Doe, Jane

**When:** Tuesday, January 05, 2010 3:00 PM-4:00 PM.

**Location:** 72 MacNider

**Description:**

### Proposing New Meeting Times

As a response to a Meeting Request, attendees can propose new times for the meeting. If you receive a response proposing a new meeting time, you can either **accept the proposal** or **view all proposals** to open the scheduling page and reschedule the meeting according to everyone's

proposals. After a new time is selected, be sure to send an update to notify all attendees of the new time.

### Checking Attendee Status

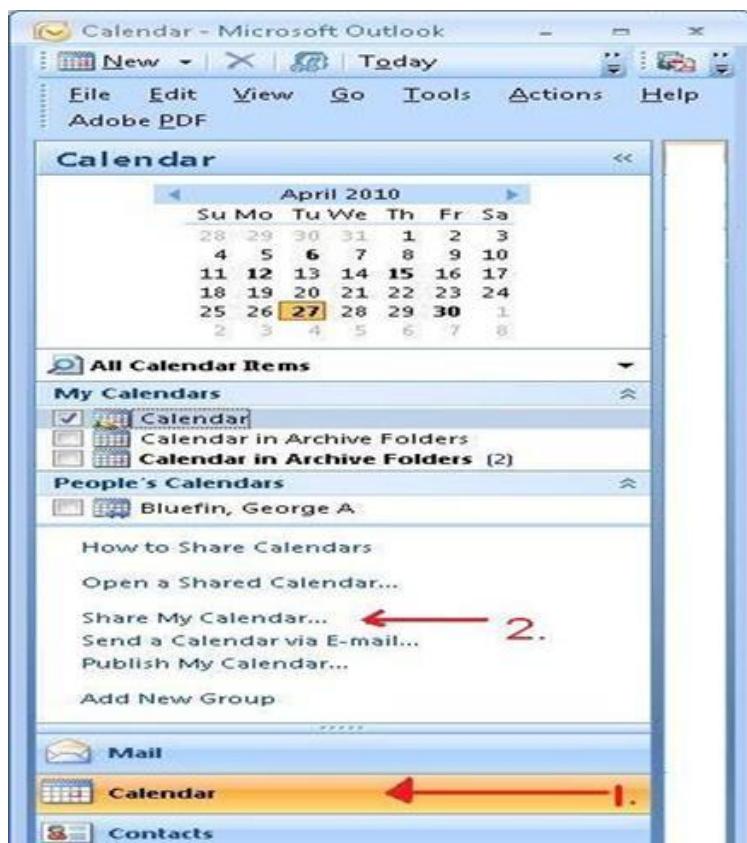
As the meeting organizer you will receive E-mail responses from invited attendees as they respond to your request. Alternatively, you can also check the status of each attendee using the **Tracking** button, shown in the **Show** group on the **Meeting** tab (this button only shows in the toolbar after a **Meeting Request** has been sent). The **Tracking** button shows each attendee, whether their attendance is required or optional and the status of their response. Only the meeting organizer can see this information.

The following responses to this meeting have been received:

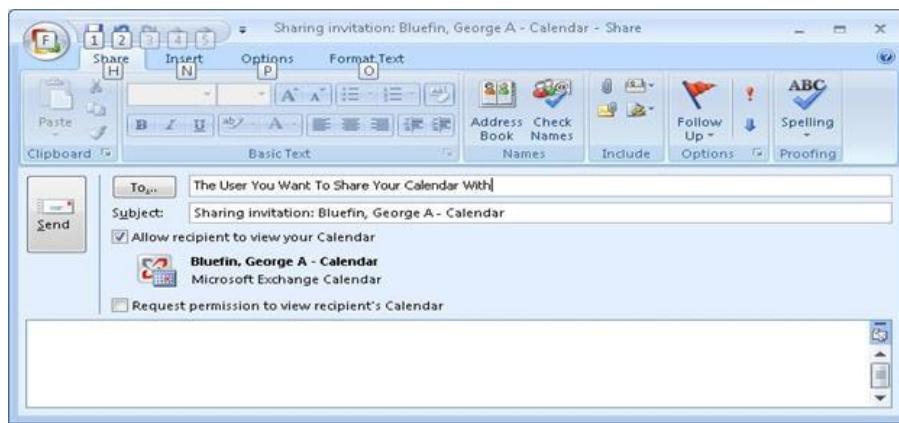
Name	Attendance	Response
Doe, John	Meeting Organizer	None
Doe, Jane	Required Attendee	Tentative
Doe, Jim	Optional Attendee	Declined

## HOW TO SHARE YOUR CALENDAR

1. First Open Outlook 2007.
2. Select the calendar button from the left pane. Then click Share My Calendar...



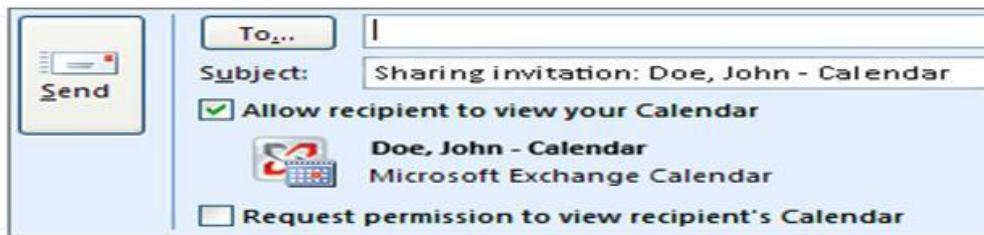
3. A dialog box appears that will send an email to the person you wish to share your calendar with. You can also choose to ask that person for view permission to their calendar by selecting the "Request permission to view recipient's Calendar" check box.



4. Click Send and you're all done.

For people not in your organization, you can also send an email with the contents of your calendar.

## **TO SEND A CALENDAR VIA E-MAIL:**



## Procedures

1. In the **Calendar** section, under **My Calendars**, click **Send a Calendar via E-mail...**
2. Specify the **dates** that you want to include in your e-mail.
3. Choose the **amount of detail** that you want to include (Availability only, Limited details, and Full details).
4. Click **Show >>** to show more advanced options
5. Click **OK** to return to the Message Composition window.
6. Add **recipients**, a custom **subject**, and an additional note in the body of your message and click **Send** to send your calendar.

## Viewing Public Calendars

You can easily view Public Calendars that available to your Exchange community through the **Public Folders link**.



### To view Public Calendars (Folders):

1. Click on the Folder List icon in the Navigation Pane.
2. This will display all your Email folders. At the bottom of the list, click on Public Folders.
3. Within this list is where you will find link to Public Calendars that are made available to your group.



Tip: Right-click on a Public Calendar and choose Add to Favorites... to have easy access to frequently viewed calendars (via the Favorites folder).

## Server Files vs. Personal/Local Files

Initially all incoming mail (as well as contacts, appointments, tasks, notes, and journal entries) is delivered to and stored on the Exchange Server. You however, have the option of creating and moving your messages to a Personal Storage folder (.pst). One of the biggest benefits of a Personal Storage folder is that it is not subject to the 2GB (beginning at the end of October 2010) mailbox size limit that is set on the server.

### Server Files

1. **Storage Location:** on the server
2. **Size Limit:** 2GB (beginning at the end of October 2010)
3. **Accessibility:** can be accessed from your work machine (through Microsoft Outlook) or accessed anywhere via the internet (log on to [outlook.unc.edu](http://outlook.unc.edu)).
4. Messages that you'll want continuous access to should be kept here.

### Personal / Local Files

1. **Storage Location:** your work computer
2. **Size Limit:** the available space of your hard drive
3. **Accessibility:** can only be accessed when at your work machine
4. This is generally a good place for messages that you no longer need to act upon but want to keep for future reference.

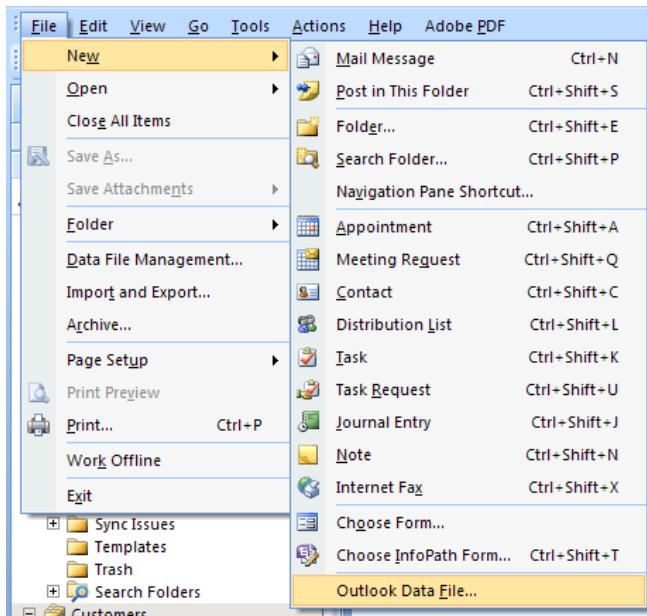


5. **Example:** as a graphic designer working at the School of Medicine, I work with a lot of different customers. Once I complete a job for a customer I no longer need to keep the email correspondence but I do just in case I need to refer back to it (for billing purposes or whatever). Since I rarely have to reference these old files I created a Personal Storage Folder entitled 'Customers' and then created a separate folder under that for each customer. It's an easy way of keeping and organizing needed messages without taking up my allotted server space.

## How to Create a Personal Storage Folder

1. On the File menu, point to **New**, and then click **Outlook Data File**.
2. Ensure **Office Outlook Personal Folders File (.pst)** is highlighted, and then click **OK**.
3. In the **File** name box, type a name for the file, and then click **OK**.
4. In the **Name** box, type a display name for the .pst folder, and then click **OK**.

- **Note:** you can set a password for the folder but it is critical that you remember it as it cannot retrieve.
5. Now you can drag and drop messages from your server folders to the new folder. Press CTRL while dragging to copy items instead of moving them.



## Auto Archive

To learn more about Auto-Archiving, see [Archiving: Manual Archiving vs. Auto-Archiving under Advanced Topics](#).

Auto Archive is turned on by default. However, you can change its default settings:

1. On the Menu Bar, go to **Tools > Options > Other**.
2. Select '**Auto Archive...**'
3. Make sure '**Run Auto Archive every X days**' is selected.
4. From here, you can change options for:
  - How often you'd like to Auto Archive (Recommended setting: Once every 7-14 days).
  - When old mail is archived (Recommended setting: Clean out after 6 months).
  - **Where to store archived mail** (Recommended setting: Put an 'Archive' folder somewhere easy to remember, such as 'My Documents').
5. Click '**Apply these settings to all folders now,**' to begin an Auto Archive using these settings. Click '**OK**' to save these settings.



## Searching Email

### How to Search Your Mail

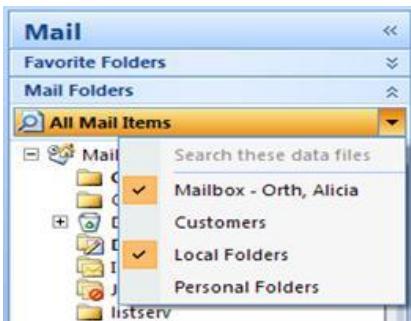
- Click on/highlight the folder you wish to search.
- In the Search box, type your search text and hit the **Enter** key on your keyboard.

From	Subject	Received	Size	Categories
Peter St... [cms]	Meeting Minutes for 7/16/09	Mon 7/20/09	14 KB	
Ben Ayc...	Re: [cms] Meeting Minutes for 7/16/09	Mon 7/20/09	7 KB	
Billy Hyl...	[webmasters] August Meeting ~ unc.edu Redesign	Tue 7/21/09	15 KB	
OIS We...	Tasks - Wireless Mice Evaluation Prepara...	Mon 8/3/09	19 KB	
Billy Hyl...	[webmasters] Reminder: August Meeting - UNC R...	Mon 8/3/09	13 KB	

- Messages that contain the text that you typed are displayed in the Search Results pane with the search text highlighted.
- You can use the logical operators **AND**, **NOT**, **OR**, **<**, **>**, **=**, and so forth to refine your search. [Examples of useful searches](#) can be found on the Microsoft web site.
- To narrow your search, type more text in the search box.
- To widen your search, click **Try searching again in All Mail Items** at the end of the search results. Alternatively, in the Navigation Pane, under Mail Folders, click **All Mail Items** or press **CTRL+ALT+A**.

**Note:** searching "All Mail Items" is based on the folders that are checked for this search option. To view these options, click on the down arrow located to the right of **All Mail Items**. For example, you will initially need to place a check mark next to **Local Folders**

to include this mail in the search results. Click here for more information on [Server Files vs. Personal/Local Folders](#).

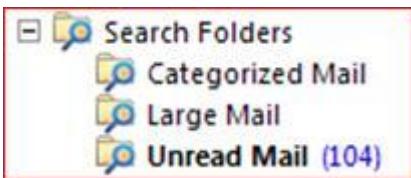


## Search Folders

The Outlook 2007 search folders are a useful feature for finding and organizing messages. A search folder isn't really a folder but rather a special view that functions much like a separate folder. In effect, a search folder is a saved search. You specify conditions for the folder, such as all messages from a specific sender or all messages received in the last day, and Outlook 2007 displays messages that meet the specified conditions in that search folder view.

Although the messages seem to exist in the search folder, they continue to reside in their respective folders. For example, a search folder might show all messages in the Inbox and Sent Items folders that were sent by John Doe. Even though these messages appear in the John Doe search folder (for example), they are actually still located in the Inbox and Sent Items folders.

Outlook 2007 includes three search folders by default, which you can use as is or customize to suit your needs:



- Categorized Mail** - This search folder shows all messages that have categories assigned to them.
- Large Mail** - This search folder shows all messages that are 100 KB or larger.
- Unread Mail** - This search folder shows all messages that are marked as unread.

## Create a New Search Folder

To create a new search folder, right-click the **Search Folders** folder, and then choose **New Search Folder** to open the New Search Folder dialog box.

The New Search Folder dialog box provides several predefined search folders which you can easily customize and create. If none of the predefined search folders are what you want then scroll to the bottom of the **Select A Search Folder** list and select **Create A Custom Search Folder**. Click

the **Choose** button to open the Custom Search Folder dialog box to specify your criteria for the search folder, a search folder name, and which sub-folders to include.



### Edit an Existing Search Folder

To customize an existing search folder, right-click the folder, and then choose **Customize This Search Folder** to open the Customize dialog box.

To change which folders are included in the search folder, click **Browse** in the Customize dialog box to open the Select Folder(s) dialog box. Place a check mark next to each folder that you want to include, or select **Mailbox - Your Name** to include all folders in the search. Place a check mark in the **Search Sub-folders** option to include all sub-folders for a selected folder. When you have finished selecting folders, click **OK**, and then click **OK** again to close the Customize dialog box.

### Create and Manage a Group/Distribution List

*Do you find yourself emailing the same group of people over and over again? If so, a distribution list may be a good solution for you. Distribution lists contain email addresses for multiple people. Unlike a Lister, a distribution list is your own personal list of people that you manage via Outlook.*

#### Create a group/distribution list:

1. From the Navigation pane, click "Contacts"
2. Select New > Distribution List
3. In the Name text box, enter a name for the list.
4. To add members from the UNC directory (or your local contacts), select "Select Members" to open the address book.
  - Enter the member's name and add it to the list by either double-clicking the name or selecting it and pressing the "members" button.

- Save and close
5. To add members outside the UNC directory, select "Add New"
  6. Enter the person's name and email address.
  7. Press OK

## Email a group/distribution list:

1. From the Navigation pane, click "Contacts"
2. Select the distribution list you would like to email
3. Select New > Message to Contact

## KEYBOARD SHORTCUTS

*Once you've gotten the hang of Outlook, you can utilize keyboard shortcuts to make your every day tasks more efficient.*

Outlook makes it easy to figure out keyboard shortcuts by listing them with their associated items in the top menu bar. For instance, if you select File (from the top menu bar), then New, you'll see several new items you can create as well as their associated keyboard shortcuts.

## Common Keyboard Shortcuts

Pane Navigation	Email Shortcuts	Calendar Shortcuts
<ul style="list-style-type: none"> <li>• Go to Mail: Ctrl+1</li> <li>• Go to Calendar: Ctrl+2</li> <li>• Go to Contact: Ctrl+3</li> <li>• Go to Tasks: Ctrl+4</li> </ul>	<ul style="list-style-type: none"> <li>• New Message: Ctrl+N</li> <li>• Check Name: Ctrl+K</li> <li>• Move to Folder: Ctrl+Shift+V</li> <li>• Reply: Ctrl+R</li> <li>• Reply to All: Ctrl+Shift+R</li> <li>• Forward: Ctrl+F</li> <li>• Follow Up: Insert key</li> <li>• Send/Receive: F9</li> <li>• Address Book: Ctrl+Shift+B</li> </ul>	<ul style="list-style-type: none"> <li>• New Appointment: Ctrl+N</li> <li>• Meeting Request: Ctrl+Shift+Q</li> <li>• Send Meeting Request: Ctrl+Enter</li> <li>• Accept Meeting: Alt+C</li> <li>• Tentative Meeting: Alt+N</li> <li>• Decline Meeting: Alt+D</li> <li>• Save and Close: Alt+S</li> </ul>
<b>Pane Independent Functions</b> <ul style="list-style-type: none"> <li>• Copy: Ctrl+C</li> <li>• Cut: Ctrl+X</li> <li>• Paste: Ctrl+V</li> <li>• Delete: Ctrl+D</li> <li>• Print: Ctrl+P</li> <li>• Help: F1</li> <li>• Undo: Ctrl+Z</li> </ul>		

## Mail Merge

The automatic addition of names and addresses from a database to letters and envelopes in order to facilitate sending mail, especially advertising, to many addresses.

### **Modern Usage of Mail Merge**

Now used generically, the term "mail merge" is a process to create personalized letters and per-addressed envelopes or mailing labels mass mailings from a form letter – a word processing document which contains fixed text, which will be the same in each output document, and variables, which act as placeholders that are replaced by text from the data source.

The data source is typically a spreadsheet or a database which has a field or column for each variable in the template. When the mail merge is run, the word processing system creates an output document for each row in the database, using the fixed text exactly as it appears in the template, but substituting the data variables in the template with the values from the matching columns.

**Mail merging is done in following simple steps:**

1. Creating a Main document.
2. Creating a Data Source.
3. Adding the merge fields into main document.
4. Merging the data with the main document.

## **PROCEDURES FOR MAIL MERGE**

### **Mail Merge Checklist: Create These *before* You Begin**

1. The simplest way to complete a Mail Merge is to **begin with a contact list you've created in an Excel spreadsheet** (with *clearly-labeled* columns). Column labels typically include First Name, Last Name, Courtesy Title, Address Line 1, City, State, Email Address, etc.
  - a. **If you're creating Email messages**, you **must** include email addresses in their own column on your spreadsheet. You must also be using the same version of Outlook & Word (for example, Outlook 2010 & Word 2007 will not work together).
  - b. Instead of an Excel spreadsheet, you can also use Outlook contacts, if you have appropriately-formatted entries for each person you'd like to contact.
  - c. If you don't have either of these, you can create one during the Mail Merge; however, for simplicity, preparing a contact list is recommended.

1. **A form letter** (if you're creating a letter or an email message). It's a good idea to go ahead & mark places in the letter where you'll need to insert contact information, as you go along. This doesn't need to be fancy or perfect; it's just a time-saver. It can be as simple as underlining some text that says address here.

- a. **If you don't have a form letter**, you can create one during the Mail Merge, either from a template or from scratch. We still recommend creating one before the Mail Merge, for simplicity.
- b. If you are creating labels, envelopes, or directories, you won't need a form letter at all.

## Mail Merge Wizard

1. **If you already have a form letter**, open it in Word 2010. If not, open Word & continue to #2.
2. Select Mailings > Start Mail Merge > Step-by-Step Mail Merge Wizard...
  - a. **Step 1 of 6: Select document type** – Choose the type of end product you want to create. Options include Letters, Email messages, Envelopes, Labels, and Directories.
  - b. **Step 2 of 6: Select starting document** – If you have a document prepared & open (recommended), select 'Use the current document.'
    - i. If you're creating a letter now, you can either type it out now, or select 'Start from a template.' If you have a document prepared, but it is not yet open in Word, choose 'Start from existing document.'
    - ii. These options will differ if you're creating envelopes, labels, & directories. Contact the Multimedia Lab if you need help with these options, as they can get very specific, depending on which envelopes & labels you're using.
  - c. **Step 3 of 6: Select recipients** – Choose the source you will be importing contact information from. If you've created a spreadsheet (recommended), choose 'Use an existing list,' & browse to the Excel file.
    - i. Alternately, you can 'Select from Outlook contacts,' or 'Type a new list.' If you select 'Type a new list,' the wizard will guide you through the creation of a spreadsheet-style contact list.
  - d. **Step 4 of 6: Write your letter** – This is where you insert all the contact information fields to be merged. To insert a field, click any location in the document where you need to insert a block of contact information, & choose that type of information from the options to the right. If a column of information you

entered in your spreadsheet doesn't fit any of the categories listed, select **More items...**, then choose the column name from your spreadsheet that contains the desired information.

- e. **Step 5 of 6: Preview your letters** – Scroll through each individual's copy of the letter to proofread.
  - i. If you see you need to make changes to someone's contact information, choose **Edit recipient list...**, select your Excel file under Data Source, then click **Edit....** Make any necessary changes, then click OK when you're finished.
- f. **Step 6 of 6: Complete the Merge** – Here, you can print your merged documents, personalize individual letters, or send out email messages.
  - i. **If you'd like to double-check your email messages before they send**, don't complete the merge quite yet. Instead, go to Outlook & select File > Work Offline. Then, come back to Word; select **Electronic Mail...**, enter your Subject line, & hit OK. Double-check your messages in your Outlook outbox. Once you're sure they're correct, select File > Work Offline again, to uncheck it.

One of the main reasons people use computers is to communicate and share information.

- **E-mail** software is used to create, send, receive, forward, store, print, and delete e-mail (electronic mail).
- **A Web browser** is a software application used to access and view Web pages.
- **A chat client** is software that allows you to connect to a chat room, which permits users to chat via the computer.
- **A newsreader** is a software program used to participate in a newsgroup, which is an online area on the Web where users conduct written discussion about a particular subject.
- **An instant messenger** is a software program installed to use instant messaging (IM), a real-time communications service that notifies you when one or more people are online and then allows you to exchange messages or files.
- **Groupware** is a software application that helps groups of people on a network work together and share information.

A video conference is a meeting between two or more geographically separated people who use a network or the Internet to transmit audio and video data.

---

## DATA PRESENTATION

### 1 Information Representation

Computers are information processing tools that give meaning to raw data. Data should be represented in a form that permits efficient storage and ease of manipulation (although these two goals might be conflicting at times). In data communications, we are also concerned with issues such as security, error detection and error correction.

Data are usually represented in some form of code for compactness and uniformity. For instance, a 'yes' response may be represented by 'Y', and a 'no' by 'N'; the four seasons by 'S' (spring), 'M' (summer), 'A' (autumn) and 'W' (winter). We can see that the choices are rather arbitrary. In real-life, codes are used for identification purposes, such as the Singapore NRIC number and the NUS student's matriculation number.

Computers are also number crunching devices. Numeric values are represented in a form that eases arithmetic computations and preserves accuracy as much as possible.

The atomic unit of data is the *bit* (*binary digit*), stemming from the fact that the elementary storage units in a computer are electronic switches where each switch holds one of the two states: *on* (usually represented by 1), or *off* (0), as shown below.



**Two states of a switch.**

Figure 1

Ultimately, all data, numeric or non-numeric, are represented in sequence of bits of zeroes and ones. Eight bits constitute a *byte* (a *nibble* is half a byte, or four bits, but this is rarely used these days), and a *word*, which is a unit for data storage and transfer, is usually in multiples of byte, depending on the width of the system bus. Computers nowadays typically use 32-bit or 64-bit words.

A sequence of bits allows for a range of representations. Storage units can be grouped together to provide larger range of representations. For convenience, we shall refer to these

representations as values, though they may represent non-numeric data. For example, the four seasons could be represented by two switches, to cover a range of four values 00, 01, 10 and 11, as shown below.

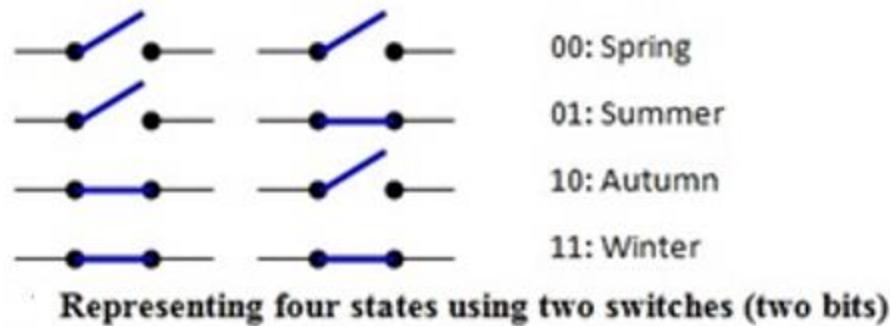


Figure 2

In general,  $N$  bits can represent up to  $2^N$  distinct values. Conversely, to represent a range of  $M$  values, the number of bits required is  $\lceil \log_2 M \rceil$ .

1 bit	→ represents up to 2 values (0, 1)
2 bits	→ represents up to 4 values (00, 01, 10, 11)
3 bits	→ represents up to 8 values (000, 001, 010, 011, 100, 101, 110, 111)
4 bits	→ represents up to 16 values (0000, 0001, 0010, ..., 1110, 1111)
32 values	→ requires 5 bits
40 values	→ requires 6 bits
64 values	→ requires 6 bits
100 values	→ requires 7 bits
1024 values	→ requires 10 bits

## 2 Positional Numerals

Different numeral systems have emerged in civilizations over the history. They can be roughly categorized as follows:

- Non-positional numeral systems
- Relative-position numeral systems
- Positional numeral systems

Non-positional systems are position-independent and include those of the Egyptian and Greek. In the Greek numeral system, more than 27 symbols are used, such as **a** for 1, **I** for 30, and **f** for 500, and so 531 would be written as **fla**. The Egyptians drew different symbols for powers of ten, for example, **I** for 1, **n** for 10, and **s** for 100. Hence, 531 would be depicted as .

The Roman numeral system includes different symbols for some basic values: **I** (1), **V** (5), **X** (10), **C** (50), and **M** (100), but employs complicated rules on relative positioning of symbols to represent other values. For instance, **IV** is 4 but **VI** is 6.

The systems described above are not well suited for computations. The problem is overcome by the elegant positional notation, where each position carries an implicit *weight*. The familiar Arabic decimal numeral is one such system.

In the decimal numeral system, the positional weights are powers of ten. In general, a decimal number  $(a_n a_{n-1} \dots a_0. f_1 f_2 \dots f_m)$  has the value

$$(a_n \cdot 10^n) + (a_{n-1} \cdot 10^{n-1}) + \dots + (a_0 \cdot 10^0) + (f_1 \cdot 10^{-1}) + (f_2 \cdot 10^{-2}) + \dots + (f_m \cdot 10^{-m})$$

$$\text{For example, } 28.75 = (2 \cdot 10^1) + (8 \cdot 10^0) + (7 \cdot 10^{-1}) + (5 \cdot 10^{-2})$$

### 3 Bases of Number Systems

The *base* or *radix* of a number system is the number of digits present. The decimal numeral system has a base or radix of 10, where the set of 10 symbols (digits) is {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. The weights are in powers of ten.

Number systems on other bases are defined likewise. For instance, a base-four number system consists of the set of four symbols {0, 1, 2, 3} with weights in powers of four. Hence, a base-four number 123.1, also written as  $(123.1)_4$  for clarity, has the value  $(1 \cdot 4^2) + (2 \cdot 4^1) + (3 \cdot 4^0) + (1 \cdot 4^{-1})$  or  $16 + 8 + 3 + 0.25$  or 27.25 in decimal, or  $(27.25)_{10}$ . Listing base-four integers in increasing value yields 0, 1, 2, 3, 10, 11, 12, 13, 20, 21, 22, 23, 30, 31, 32, 33, 100, 101, 102, 103, 110, ... .

In general, a base- $b$  number  $(a_n a_{n-1} \dots a_0. f_1 f_2 \dots f_m)_b$  has the value

$$(a_n \cdot b^n) + (a_{n-1} \cdot b^{n-1}) + \dots + (a_0 \cdot b^0) + (f_1 \cdot b^{-1}) + (f_2 \cdot b^{-2}) + \dots + (f_m \cdot b^{-m})$$

The point that separates the integer part and fraction part is known as the *radix point*. The weights are in powers of  $b$ . The above forms the basis for conversion of a base- $b$  number to decimal.

We may also adopt the computationally more efficient *Hornets rule* to convert a base- $b$  integer to decimal. This method simply factors out the powers of  $b$ , by rewriting  $(a_n \cdot b^n) + (a_{n-1} \cdot b^{n-1}) + \dots + (a_0 \cdot b^0)$  as  $((a_n \cdot b) + a_{n-1}) \cdot b + \dots + a_0$ . For example,  $(123)_4 = ((1 \cdot 4) + 2) \cdot 4 + 3 = 27$ . For fractions, we may adopt the same method by replacing multiplication with division.

Special names are given to number systems on certain bases. If you think that the base is a small value, you are wrong. The Babylonians used a sexagesimal system with base 60!

### Bases of Positional Numeral Systems

Table 1:-

Base	Name	Base	Name
2	Binary	9	Nonary
3	Ternary	10	Decimal (or Denary)
4	Quaternary	12	Duodecimal
5	Quinary	16	Hexadecimal
6	Senary	20	Vigesimal
7	Septimal	60	sexagesimal
8	Octal (or Octonary)		

For a base  $b$  that is less than ten, the symbols used are the first  $b$  Arabic symbols: 0, 1, 2, ...,  $b-1$ . For bases that exceed ten, more symbols beyond the ten Arabic symbols must be introduced. The hexadecimal system uses the additional symbols A, B, C, D, E and F to represent 10, 11, 12, 13, 14 and 15 respectively.

The binary number system is of particular interest to us as the two symbols 0 and 1 correspond to the two bits that represent the states of a switch. The octal and hexadecimal number systems are also frequently encountered in computing.

*Examples:* Convert the following numbers into their decimal equivalent.

$$(1101.101)_2 = (1 \cdot 2^3) + (1 \cdot 2^2) + (1 \cdot 2^0) + (1 \cdot 2^{-1}) + (1 \cdot 2^{-3})$$

$$= 8 + 4 + 1 + 0.5 + 0.125 = 13.625$$

$$(572.6)_8 = (5 \cdot 8^2) + (7 \cdot 8^1) + (2 \cdot 8^0) + (6 \cdot 8^{-1})$$

$$= 320 + 56 + 2 + 0.75 = 378.75$$

$$(2A.8)_{16} = (2 \cdot 16^1) + (10 \cdot 16^0) + (8 \cdot 16^{-1})$$

$$= 32 + 10 + 0.5 = 42.5$$

$$(341.24)_5 = (3 \cdot 5^2) + (4 \cdot 5^1) + (1 \cdot 5^0) + (2 \cdot 5^{-1}) + (4 \cdot 5^{-2})$$

$$= 75 + 20 + 1 + 0.4 + 0.16 = 96.56$$

#### 4. Decimal to Binary Conversion

One way to convert a decimal number to its binary equivalent is the *sum-of-weights* method. Here, we determine the set of weights (which are in powers of two) whose sum is the number in question.

*Examples:* Convert decimal numbers to their binary equivalent using sum-of-weights.

$$(9)_{10} = 8 + 1 = 2^3 + 2^0 = (1001)_2$$

$$(18)_{10} = 16 + 2 = 2^4 + 2^1 = (10010)_2$$

$$(58)_{10} = 32 + 16 + 8 + 2 = 2^5 + 2^4 + 2^3 + 2^1 = (111010)_2$$

$$(0.625)_{10} = 0.5 + 0.125 = 2^{-1} + 2^{-3} = (0.101)_2$$

We observe that multiplying a number  $x$  by  $2^n$  is equivalent to shifting left the binary representation of  $x$  by  $n$  positions, with zeroes appended on the right. For example, the value  $(5)_{10}$  is represented as  $(101)_2$ , and  $(40)_{10}$  (which is  $5 \cdot 2^3$ ) is  $(101000)_2$ . Similarly, dividing a number  $x$  by  $2^n$  is equivalent to shifting right its binary representation by  $n$  positions. This may help in quicker conversion.

The second method deals with the integral portion and the fractional portion separately, as follows:-

- Integral part: *Repeated division-by-two*
- Fractional part: *Repeated multiplication-by-two*
- Repeated Division-by-Two

To convert a decimal integer to binary, we divide the number successively by two until the quotient becomes zero. The remainders form the answer, with the first remainder serving as the *least significant bit* (LSB) and the last remainder the *most significant bit* (MSB).

The example below shows how  $(43)_{10}$  is converted into its binary equivalent.

2	43	
2	21 rem 1	← LSB
2	10 rem 1	
2	5 rem 0	
2	2 rem 1	
2	1 rem 0	
	0 rem 1	← MSB

$(43)_{10} = (101011)_2$

- Repeated Multiplication-by-Two

To convert a decimal fraction to binary, we multiply the number successively by two, removing the *carry* in each step, until the fractional product is zero or until the desired number of bits is collected. The carries form the answer, with the first carry serving as the MSB and the last as the LSB.

The example below shows how  $(0.3125)_{10}$  is converted into its binary equivalent.

		Carry
0.3125 × 2	= 0.625	0 ← MSB
0.625 × 2	= 1.25	1
0.25 × 2	= 0.5	0
0.5 × 2	= 1.0	1 ← LSB

$(0.3125)_{10} = (0.0101)_2$

## 5. Conversion Between Bases

- Decimal to Base-*R* Conversion

We may extend the repeated-division and repeated-multiplication techniques to convert a decimal value into its equivalent in base  $R$ :

- Integral part: *Repeated division-by-R*
  - Fractional part: *Repeated multiplication-by-R*
  - Truncation and Rounding

In the course of converting values – especially fractional values – between bases, there might be instances when the values are to be corrected within a specific number of places. This may be done by *truncation* or *rounding*.

In truncation, we simply chop a portion off from the fraction. Table 2-2 shows the truncation of the value  $(0.12593)_{10}$  with respect to the number of decimal places desired.

## Truncation versus Rounding

<i>Number of decimal places</i>	<i>Truncated value</i>	<i>Rounded value</i>
4	$(0.1259)_{10}$	$(0.1259)_{10}$
3	$(0.125)_{10}$	$(0.126)_{10}$
2	$(0.12)_{10}$	$(0.13)_{10}$
1	$(0.1)_{10}$	$(0.1)_{10}$

In rounding, we need to examine the leading digit of the portion we intend to remove. We divide the digits of the numeral system into two equal-sized sets, where the first set contains the ‘lighter’ digits (example: 0, 1, 2, 3, 4 in the decimal system), and the second set the ‘heavier’ digits (5, 6, 7, 8, 9). If that digit being examined belongs to the former set, we remove the unwanted portion; if it belongs to the latter set, we ‘promote’ (round up) the last digit in the retained portion to its next higher value (propagating the promotion if necessary). Table 2 above shows the rounding of the value  $(0.12593)_{10}$  with respect to the specified number of decimal places.

For bases that are odd numbers, there exists a ‘middle’ digit that may be classified either as a ‘light’ digit or a ‘heavy’ digit. For example,  $(6.143)_7$ , may be rounded to two places either as  $(6.14)_7$  or  $(6.15)_7$ .

- Conversion between Binary and Octal/Hexadecimal

There exist simple techniques for conversion between binary numbers and octal (or hexadecimal) numbers. They are given below.

- Binary  $\circledast$  Octal: Partition (from the radix point outwards) in groups of 3; each group of 3 bits corresponds to an equivalent octal digit.
- Octal  $\circledast$  Binary: Expand each octal digit into an equivalent group of 3 bits.
- Binary  $\circledast$  Hexadecimal: Partition (from the radix point outwards) in groups of 4; each group of 4 bits corresponds to an equivalent hexadecimal digit.
- Hexadecimal  $\circledast$  Binary: Expand each hexadecimal digit into an equivalent group of 4 bits.

*Examples:* Conversion between binary and octal/hexadecimal.

$$(10\ 111\ 011\ 001.\ 101\ 11)_2 = (2731.56)_8$$

$$(2731.56)_8 = (010\ 111\ 011\ 001.\ 101\ 110)_2$$

$$(101\ 1101\ 1001.\ 1011\ 1)_2 = (5D9.B8)_{16}$$

$$(5D9.B8)_{16} = (0101\ 1101\ 1001.\ 1011\ 1000)_2$$

Octal and hexadecimal notations are commonly encountered in computing literature as they provide more compact writing compared to binary notation, and they can be easily converted into binary form.

On closer observation, it is not difficult to uncover the underlying principle behind these techniques. (Hint: 8 is  $2^3$ , and 16 is  $2^4$ .) With this, we can design similar techniques to convert between related bases, such as between binary and quaternary (base 4), and between ternary (base 3) and notary (base 9).

### General Conversion

Apart from the techniques for conversion between related bases such as those discussed above, for general conversion between two bases, the approach is to convert the given value first into decimal, followed by converting the decimal value into the target base.

## 6. Arithmetic Operations on Binary Numbers

Arithmetic operations on binary numbers are similar to those on decimal.

The example below shows the multiplication of two binary numbers. The multiplicand (11001) is multiplied with every bit of the multiplier (10101), and the partial products are then added.

$$\begin{array}{r}
 1\ 1\ 0\ 0\ 1 \\
 \times 1\ 0\ 1\ 0\ 1 \\
 \hline
 1\ 1\ 0\ 0\ 1 \\
 1\ 1\ 0\ 0\ 1 \\
 + 1\ 1\ 0\ 0\ 1 \\
 \hline
 1\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 0\ 1
 \end{array}$$

Binary addition is performed in the same manner as in decimal. The examples below compare addition in the two systems.

Binary	Decimal
1 1 0 1 1	6 4 8
+ 1 0 0 1 1	+ 5 9 7
<hr/>	<hr/>
1 0 1 1 1 0	1 2 4 5

The addition is performed column by column, from right to left. At each column three bits are added: one bit from each of the two numbers, and a *carry-in* bit. The column addition generates the *sum* bit, and a *carry-out* bit that is propagated to the next column to its left as the carry-in. Table 3 shows the eight possible outcomes of a column addition.

Bit addition table  
Table 3:-

0 + 0 + 0 = 0 0
0 + 0 + 1 = 0 1
0 + 1 + 0 = 0 1
0 + 1 + 1 = 1 0
1 + 0 + 0 = 0 1
1 + 0 + 1 = 1 0
1 + 1 + 0 = 1 0
1 + 1 + 1 = 1 1

*Carry-in*    *Sum*  
*Carry-out*

Subtraction is performed in a similar fashion.

Binary	Decimal
1 0 0 1 0	8 2 3
- 0 1 1 0 1	- 3 9 7
—————	—————
0 0 1 0 1	4 2 6

Like addition, the subtraction is performed column by column. However, unlike traditional method where we might progressively ‘borrow’ from a few columns before returning to the column we are currently working on, we adopt the ‘borrow-in’ and ‘borrow-out’ concept that allows the information to be propagated one column at a time. Table 4 shows the eight possible outcomes of a column subtraction. A column subtraction generates a *difference* bit and a *borrow-out* bit that becomes the *borrow-in* of the next column to its left.

Bit

subtraction

Table

table

4:-

0 0 - 0 = 0 0
0 0 - 1 = 1 1
0 1 - 0 = 0 1
0 1 - 1 = 0 0
1 0 - 0 = 1 1
1 0 - 1 = 1 0
1 1 - 0 = 0 0
1 1 - 1 = 1 1

*Borrow-in*    *Difference*  
*Borrow-out*

## 7. Negative Numbers

Till now, we have only considered *unsigned* numbers, which are non-negative values. We shall now consider a few schemes to represent *signed* integers (positive and negative values). The four common representations for signed binary numbers are:

- Sign-and-magnitude
- Excess
- 1's complement
- 2's complement
- Sign-and-Magnitude

The *sign-and-magnitude* representation employs a prefix bit to indicate the *sign* of the number, followed by the *magnitude* field. A positive value has a sign bit of 0 while a negative value a sign bit of 1. Figure 3 shows how the value  $-75$  is represented in an 8-bit sign-and-magnitude scheme. We may write the value as  $(11001011)_{sm}$ .



Figure 3 An 8-bit- Sign-and-Magnitude representation of -75

An 8-bit sign-and-magnitude representation allows values between  $-127$  (represented as  $(11111111)_{sm}$ ) and  $+127$  (represented as  $(01111111)_{sm}$ ) to be represented. There are two representations for zero:  $(00000000)_{sm}$  and  $(10000000)_{sm}$ . In general, an  $n$ -bit sign-and-magnitude representation can cover this range of values  $[-(2^{n-1} - 1), 2^{n-1} - 1]$ .

To negate a value, we invert the sign bit.

### Excess

The *excess* system (also known as *biased* system) is another popular system in use for negative numbers. To represent a value, a fixed bias is added to this value, and the binary representation of the result is the desired representation. For example, assuming a 5-bit excess system, if the bias chosen is 16, then it is called the excess-16 system. In the excess-16 system, the value  $-16$  is represented as  $(00000)_{ex16}$  (since  $-16 + 16 = 0$ ), and the value 3

would be represented as  $(10011)_{\text{ex}16}$  (since  $3 + 16 = 19$ ). Since the excess system is just a simple translation of the binary system,  $(00000)_{\text{ex}16}$  would represent the most negative value (which is  $-16$ ), and  $(11111)_{\text{ex}16}$  would represent the largest positive value (which is  $15$ ).

The bias is usually chosen so that the range of values represented has a balanced number of positive and negative values. Hence we pick  $16$  as the bias for a 5-bit excess system, resulting in the range of  $[-16, 15]$ . (Sometimes we pick  $15$  as the bias, which results in the range  $[-15, 16]$ .) For a 4-bit system, a reasonable bias would be  $8$  (or sometimes  $7$ ). In general, for  $n$  bits, the bias is usually  $2^{n-1}$ .

Table 5 shows the 4-bit excess-8 system.

The 4-bit Excess-8 system  
Table 5

Value	Excess-8	Value	Excess-8
-8	0000	0	1000
-7	0001	1	1001
-6	0010	2	1010
-5	0011	3	1011
-4	0100	4	1100
-3	0101	5	1101
-2	0110	6	1110
-1	0111	7	1111

### 1's Complement

For positive values, the *ones' complement* representation is identical to the binary representation. For instance, the value  $75$  is represented as  $(01001011)_2$  in an 8-bit binary system, as well as  $(01001011)_{1s}$  in an 8-bit 1's complement system. What about negative values?

Given a number  $X$  which can be expressed as an  $n$ -bit binary number, its negated value,  $-X$ , can be obtained in 1's complement form by this formula:

$$-X = 2^n - X - 1$$

For example, 75 is represented as  $(01001011)_2$  in an 8-bit binary system, and hence  $-75$  is represented as  $(10110100)_{1s}$  in the 8-bit 1's complement system. (Note that  $2^8 - 75 - 1 = 180$  whose binary form is  $10110100$ .)

We observe that we can easily derive  $-X$  from  $X$  in 1's complement by inverting all the bits in the binary representation of  $X$ . Note also that the first bit serves very much like a sign bit, indicating that the value is positive (negative) if the first bit is 0 (1). However, the remaining bits do not constitute the magnitude of the number, particularly for negative numbers.

An 8-bit 1's complement representation allows values between  $-127$  (represented as  $(10000000)_{1s}$ ) and  $+127$  (represented as  $(01111111)_{1s}$ ) to be represented. There are two representations for zero:  $(00000000)_{1s}$  and  $(11111111)_{1s}$ . In general, an  $n$ -bit 1's complement representation has a range  $[-(2^{n-1} - 1), 2^{n-1} - 1]$ .

To negate a value, we invert all the bits. For example, in an 8-bit 1's complement scheme, the value 14 is represented as  $(00001110)_{1s}$ , therefore  $-14$  is represented as  $(11110001)_{1s}$ .

## 2's Complement

The *two's complement* system share some similarities with the ones' complement system. For positive values, it is the same as the binary representation. The first bit also indicates the sign of the value (0 for positive, 1 for negative).

Given a number  $X$  which can be expressed as an  $n$ -bit binary number, its negated value,  $-X$ , can be obtained in 2's complement form by this formula:

$$-X = 2^n - X$$

For example, 75 is represented as  $(01001011)_2$  in an 8-bit binary system, and hence  $-75$  is represented as  $(10110101)_{2s}$  in the 8-bit 2's complement system. (Note that  $2^8 - 75 = 181$  whose binary form is  $10110101$ .)

Again, we observe that we can easily derive  $-X$  from  $X$  in 2's complement by inverting all the bits in the binary representation of  $X$ , and then adding one to it.

An 8-bit 2's complement representation allows values between  $-128$  (represented as  $(10000000)_{2s}$ ) and  $+127$  (represented as  $(01111111)_{2s}$ ) to be represented, and hence it has a

range that is one larger than that of the 1's complement representation. This is due to the fact that there is only one unique representation for zero:  $(00000000)_{2s}$ . In general, an  $n$ -bit 2's complement representation has a range  $[-2^{n-1}, 2^{n-1} - 1]$ .

To negate a value, we invert all the bits and plus 1. For example, in an 8-bit 2's complement scheme, the value 14 is represented as  $(00001110)_{2s}$ , therefore  $-14$  is represented as  $(11110010)_{2s}$ .

### Comparisons of Sign-and-Magnitude and Complements

Table 6 compares the sign-and-magnitude, 1's complement and 2's complement schemes of a 4-bit signed number system.

Sign-and-Magnitudes, 1's complement and 2's complement  
Table 6:-

Value	Sign-and-Magnitude		1's Comp.	2's Comp.	Value	Sign-and-Magnitude		1's Comp.	2's Comp.
	Comp.	Comp.	Comp.	Comp.		Comp.	Comp.	Comp.	Comp.
+7	0111		0111	0111	-0	1000		1111	-
+6	0110		0110	0110	-1	1001		1110	1111
+5	0101		0101	0101	-2	1010		1101	1110
+4	0100		0100	0100	-3	1011		1100	1101
+3	0011		0011	0011	-4	1100		1011	1100
+2	0010		0010	0010	-5	1101		1010	1011
+1	0001		0001	0001	-6	1110		1001	1010
+0	0000		0000	0000	-7	1111		1000	1001
					-8	-	-	1000	

### Diminished Radix Complement and Radix Complement

The complement systems are not restricted to the binary number system. In general, there are two complement systems associated with a base- $R$  number system.

- Diminished Radix (or  $(R-1)$ 's) Complement
- Radix (or  $R$ 's) Complement

Given an  $n$ -digit base- $R$  number  $X$ , its negated value,  $-X$ , can be obtained in  $(R-1)$ 's complement form by this formula:

$$-X = R^n - X - 1$$

For example,  $(-22)_{10}$  is represented as  $(77)_{9s}$  in the 2-digit 9's complement system, and  $(-3042)_5$  is represented as  $(1402)_{4s}$  in the 4-digit 4's complement system.

Given an  $n$ -digit base- $R$  number  $X$ , its negated value,  $-X$ , can be obtained in  $R$ 's complement form by this formula:

$$-X = R^n - X - 1$$

For example,  $(-22)_{10}$  is represented as  $(78)_{10s}$  in the 2-digit 10's complement system, and  $(-3042)_5$  is represented as  $(1403)_{5s}$  in the 4-digit 5's complement system.

## 8. Addition and Subtraction in Complements

We shall discuss how addition of binary numbers is performed under the complement schemes.

- Addition in 2's Complement System

The following is the algorithm for  $A + B$  in the 2's complement system:

1. Perform binary addition on the two numbers  $A$  and  $B$ .
2. Discard the carry-out of the MSB.
3. Check for *overflow*: an overflow occurs if the carry-in and carry-out of the MSB column are different, or if  $A$  and  $B$  have the same sign but the result has an opposite sign.

The following are some examples of addition on 4-bit 2's complement numbers. The decimal values are shown for verification.

3	0	0	1	1
+ 4	+ 0	1	0	0

4	0	1	0	0
+ -7	+ 1	0	0	1

6	0	1	1	0
+ -3	+ 1	1	0	1

-2	1	1	1	0
+ -6	+ 1	0	1	0

There is no overflow in the above examples, as the results are all within the value range of values  $[-8, 7]$  for the 4-bit 2's complement system. The following two examples, however, illustrate the cases where overflow occurs. Neither 11 nor  $-9$  falls within the valid range of values.

5	0 1 0 1
$+ \quad 6$	$+ \quad 0 1 1 0$

−6	1 0 1 0
$+ \quad -3$	$+ \quad 1 1 0 1$

Overflow is detected when two positive values are added to produce a negative value, or adding two negative values results in a positive value.

### Addition in 1's Complement System

The following is the algorithm for  $A + B$  in the 1's complement system:

1. Perform binary addition on the two numbers  $A$  and  $B$ .
2. Add the carry-out of the MSB to the result.
3. Check for *overflow*: an overflow occurs if  $A$  and  $B$  have the same sign but the result has an opposite sign.

The following are some examples of addition on 4-bit 1's complement numbers. Again, the decimal values are shown for verification.

3	0 0 1 1
$+ \quad 4$	$+ \quad 0 1 0 0$

4	0 1 0 0
$+ \quad -7$	$+ \quad 1 0 0 0$

−2	1 1 0 1
$+ \quad -4$	$+ \quad 1 0 1 1$
$\hline$	$\hline$
−6	1 0 0 0

−3	1 1 0 0
$+ \quad -6$	$+ \quad 1 0 0 1$
$\hline$	$\hline$
−9	1 0 1 0

In the last example above, an overflow occurs as the value  $-9$  is out of range, and is detected by the sign of the result being opposite to the sign of the two values.

### Subtraction in Complement Systems

In hardware implementation, subtraction is more complex than addition. Therefore, the subtraction operation is usually carried out by a combination of complement and addition. In essence, we convert the operation  $A - B$  into  $A + (-B)$ . As we have seen, negating  $B$  to obtain  $-B$  is a simple matter in the complement systems.

For example, to perform  $(5)_{10} - (3)_{10}$  in the 4-bit 2's complement binary system, we get  $(0101)_2 - (0011)_2$ , or  $(0101)_2 + (1101)_2$ . Applying the addition algorithm described earlier, the result is  $(0010)_2$ , or  $(2)_{10}$ .

## 9 .Fixed-Point Representation

In representing real numbers in the computer, the radix point is not explicitly stored. One strategy is to assume a position of the radix point, which divides the field into two portions, one for the integral part and the other the fractional part. This is known as *fixed-point representation*. For example, Figure 4 shows the value  $(21.75)_{10}$  or  $(010101.11)_2$ , in an 8-bit sign-and-magnitude scheme with two bits allocated to the fraction part.

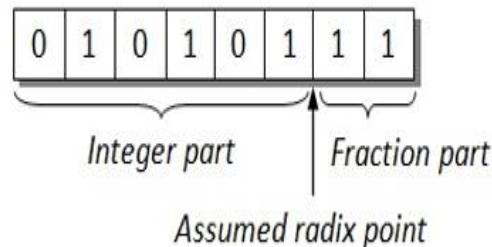


Figure 4 An 8-bit Sign - and - Magnitude Fixed - Point Representation.

It is apparent that the number of bits allocated to the representation determines the maximum number of values (up to  $2^n$  values for an  $n$ -bit representation), the width of the integral portion determines the range of values, and the width of the fractional portion determines the *precision* of the values. It is the finite size of the fractional portion that results in imprecise representation of certain numeric values, and computational error due to truncation or rounding. Due to the discrete nature of computer, real numbers are only approximated in their computer representation.

## 10 . Floating-Point Representation

The simple fixed-point representation allows for a very limited range of values that can be represented. A more versatile scheme, the *floating-point representation*, overcomes this weakness. The radix point in this format is 'floating', as we adopt the concept of *scientific*

notation. Some examples of values written in the scientific notation are shown below.

*Examples:*

Decimal values expressed in scientific notation.

$$0.123 \cdot 10^{30} = 123,000,000,000,000,000,000,000,000,000$$

$$0.5 \cdot 10^{-17} = 0.000\,000\,000\,000\,000\,005$$

$$-98.765 \cdot 10^{-7} = -0.000\,009\,876\,5$$

Binary values expressed in scientific notation.

$$101.101 \cdot 2^{-7} = 0.0000101101$$

$$110.011 \cdot 2^8 = 11001100000$$

The scientific notation comprises the *mantissa* (also called *significant*), the *base* (or *radix*) and the *exponent*. Figure 5 shows the three components of a decimal number in scientific notation.

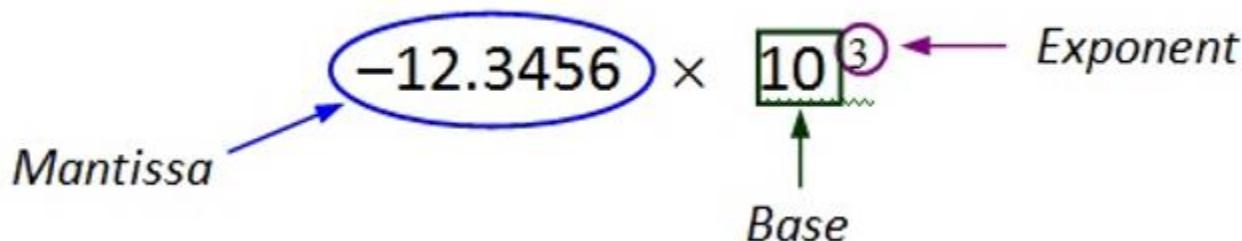


Figure 5 Scientific Notation.

These representations:  $-12.3456 \cdot 10^3$ ,  $-123456 \cdot 10^{-1}$ , and  $-0.00123456 \cdot 10^7$  all represent the same value  $-12345.6$ . To achieve consistency and storage efficiency, we adopt a form where the mantissa is said to be *normalized*. A normalized mantissa is one in which the digit immediately after the radix point is non-zero. Therefore, the normalized form of our example would be  $-0.123456 \cdot 10^5$ .

The scientific notation is implemented as floating-point representation. The sign-and-magnitude format is usually chosen to represent the mantissa, and the base (radix) is assumed to be two, since numbers are represented in binary in computers. The floating-

point format hence consists of three fields: a sign bit, a mantissa field, and an exponent field (the relative positions of the mantissa field and exponent field, as well as their widths, may vary from system to system), as shown in Figure 6.

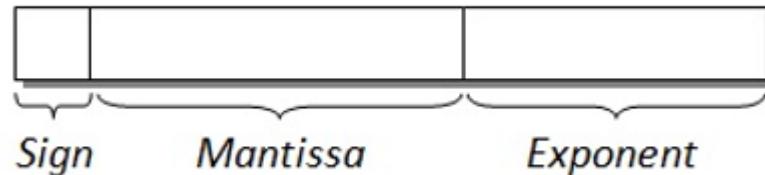


Figure 6 A Floating-Point Representation

The mantissa is usually normalized, so the first bit in the mantissa field must be a 1. Indeed, some systems further save on this bit, and store the mantissa only from the second bit onwards. The exponent field may be implemented in different schemes, such as sign-and-magnitude, 1's complement, 2's complement, or excess. More bits for mantissa offers better precision, while a bigger exponent field provides a larger range of values. There is hence a trade-off between precision and range in determining the widths of these two fields.

The examples below assume a floating-point representation with 1-bit sign, 5-bit normalized mantissa, and 4-bit exponent for the value  $(0.1875)_{10}$ , under the various schemes for exponent.

*Examples:*

Convert the given value to binary and express it in scientific notation with normalized mantissa:

$$(0.1875)_{10} = (0.0011)_2 = 0.11 \times 2^{-2}$$

If exponent is in 1's complement format:

0	11000	1101
---	-------	------

If exponent is in 2's complement format :

0	11000	1110
---	-------	------

If exponent is in excess-8 format:

0	11000	0110
---	-------	------

## Multiplication on Floating-Point Numbers

The steps for multiplying two numbers  $A$  and  $B$  in floating-point representation are:

1. Multiply the mantissas of  $A$  and  $B$ .
2. Add the exponents of  $A$  and  $B$ .

### 3. Normalize if necessary.

For example: if  $A = (0.11)_2 \cdot 2^4$  and  $B = (0.101)_2 \cdot 2^{-1}$ , then  $A \cdot B = (0.11 \cdot 0.101) \cdot 2^{4-1} = (0.01111)_2 \cdot 2^3 = (0.1111)_2 \cdot 2^2$ . This illustrates the multiplication of two decimal values 12 and  $\frac{5}{16}$  to obtain 3.75.

## Addition on Floating-Point Numbers

The steps for adding two numbers  $A$  and  $B$  in floating-point representation are:

1. Equalize the exponents of  $A$  and  $B$ .
2. Add the mantissas of  $A$  and  $B$ .
3. Normalize if necessary.

For example: if  $A = (0.11)_2 \cdot 2^4$  and  $B = (0.101)_2 \cdot 2^{-1}$ , then we convert  $A$  to  $(11000)_2 \cdot 2^{-1}$ , so  $A + B = (11000 + 0.101) \cdot 2^{-1} = (11000.101)_2 \cdot 2^{-1} = (0.11000101)_2 \cdot 2^4$ . This illustrates the addition of two decimal values 12 and  $\frac{5}{16}$  to obtain 12.3125.

## IEEE Floating Point Representation

The IEEE has devised the IEEE Standard 754 floating point representation for representing real numbers on computers, adopted widely in Intel-based computers, Mackintoshes and most UNIX machines. It provides a 32-bit format for single-precision values, and a 64-bit format for double-precision values. The double-precision format contains a mantissa field that is more than twice as long as the mantissa field of the single-precision format, permitting greater accuracy.

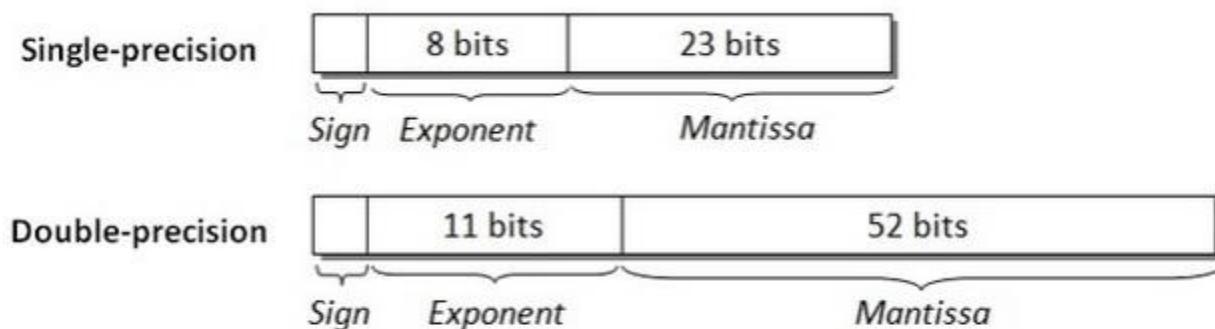


Figure 7 The IEEE Standard 754 Floating-Point Representation.

The mantissa field assumes an implicit leading bit of 1, and the exponent field adopts the excess system with a bias value of 127 for the single-precision format, and a bias of 1023 for the double-precision format. Some representations are reserved for special values such as zero, infinity, NAN (not-a-number), denormalised values.

## 11

## Codes

Our preceding discussion has been on the various representations for numeric values with the objective of easing arithmetic operations. However, we recognize that we are too accustomed to the decimal number system, whereas the ‘natural’ choice for computer representation is the binary number system, and the conversion between the two systems can be costly.

If arithmetic computations are not our concern, we may devise some coding schemes to represent decimal numbers. These schemes support quick conversion between the code and the value it represents, with the objective for data communications.

We shall discuss a few popular coding schemes such as BCD, Excess-3, 84-2-1, 2421, and the Binary codes. These codes are used to represent the ten decimal digits.

BCD	(Binary	Coded	Decimal)
The <i>BCD</i> (Binary Coded Decimal) represents each decimal digit using a 4-bit code with weights 8, 4, 2 and 1. Hence it is also known as the <i>8421 code</i> .			

For example, the decimal number 294 would be represented as 0010 1001 0100 in BCD, with a 4-bit BCD code for each decimal digit in the number. (For negative values, the negative sign ‘–’ is to be coded separately and is not within the scope of BCD coding.)

Some codes are unused: 1010, 1011, ..., 1111. These codes are considered as errors.

The BCD code offers the advantage of ease of conversion to its decimal equivalent. Performing arithmetic operations on BCD codes however, are more complicated and best avoided. BCD codes are useful for interfaces such as keypad inputs.

- Excess-3 Code

The *Excess-3 code* uses a bias value of three. Hence the codes for the ten digits 0, 1, ..., 9 are 0011, 0100, ..., 1100 respectively. The decimal number 294 would be represented as 0101 1100 0111

- 84-2-1 Code

The *84-2-1 code* uses the weights of 8, 4, -2 and -1 in the coding. The decimal number 294 would be represented as 0110 1111 0100.

- 2421 Code

The *2421 code* uses the weights of 2, 4, 2 and 1 in the coding. According to the weights, certain digits may have alternative codes. For instance, the digit 3 could be represented as 0011 or 1001. However, we pick the former in the standard 2421 coding, so that the codes for the first five digits 0 – 4 begin with 0, whereas the codes for the last five digits 5 – 9 begin with 1. The decimal number 294 would be represented as 0010 1111 0100.

- Biquinary Code

The *Biquinary code* (*bi*=two, *quinary*=five) uses seven bits. The first two bits are either 01 (for representations of the first five digits) or 10 (for representations of the last five digits), and the following five bits consist of a single 1. The Biquinary code has error-detecting capability. The decimal number 294 would be represented as 0100100 1010000 0110000.

#### Comparison of Decimal Codes

#### BCD, Excess-3, 84-2-1, 2421 and Biquinary Codes

Table 7

Decimal digit	BCD 8421	Excess-3	84-2-1	2421	Biquinary 5043210
0	0000	0011	0000	0000	0100001
1	0001	0100	0111	0001	0100010
2	0010	0101	0110	0010	0100100
3	0011	0110	0101	0011	0101000
4	0100	0111	0100	0100	0110000
5	0101	1000	1011	1011	1000001
6	0110	1001	1010	1100	1000010
7	0111	1010	1001	1101	1000100
8	1000	1011	1000	1110	1001000
9	1001	1100	1111	1111	1010000

Table 7 compares the common decimal codes. Note that the codes for the first five digits all begin with 0, while the codes for the last five digits begin with 1. Other codes exist, such as the 5211 code.

The following are some definitions:

- A *weighted code* is one where each bit position has an associated weight. The BCD, 84-2-1, 2421 and Biquinary codes are all weighted codes.
- A *self-complementing* (or *reflective*) code is one in which the codes for complementary digits are also complementary to each other. For instance, the Excess-3 code for digit 2 is 0101, while the code for digit 7 (the complement of 2) is 1010 (the complement of 0101). The Excess-3, 84-2-1 and 2421 codes are self-complementing.
- A *sequential code* is one in which each succeeding code value is one binary value greater than its preceding code value. The BCD and Excess-3 codes are sequential.
  - ASCII Code

Apart from numerical values, computing devices also handle textual data comprising characters. The character set includes letters in the alphabet ('A' ... 'Z', 'a' ... 'z'), digits ('0' ... '9'), special symbols ('+', '\$', '.', ';', '@', '\*', etc.) and non-printable characters (control-A, backspace, control-G, etc.).

The *ASCII (American Standard Code for Information Interchange)* character set is a commonly used standardised code that uses 7 bits, plus a parity bit for error detection. Table 8 shows the ASCII character set. Every character has a unique ASCII value. The ASCII value of character 'A', for example, is  $(1000001)_2$  or  $(65)_{10}$ .

The	ASCII	Character	Set
-----	-------	-----------	-----

Table 8 :-

	MSBs							
LSBs	000	001	010	011	100	101	110	111
<b>0000</b>	NUL	DLE	SP	0	@	P	`	p
<b>0001</b>	SOH	DC <sub>1</sub>	!	1	A	Q	a	q
<b>0010</b>	STX	DC <sub>2</sub>	"	2	B	R	b	r
<b>0011</b>	ETX	DC <sub>3</sub>	#	3	C	S	c	s
<b>0100</b>	EOT	DC <sub>4</sub>	\$	4	D	T	d	t
<b>0101</b>	ENQ	NAK	%	5	E	U	e	u

<b>0110</b>	ACK	SYN	&	6	F	V	f	v
<b>0111</b>	BEL	ETB	'	7	G	W	g	w
<b>1000</b>	BS	CAN	(	8	H	X	h	x
<b>1001</b>	HT	EM	)	9	I	Y	i	y
<b>1010</b>	LF	SUB	*	:	J	Z	j	z
<b>1011</b>	VT	ESC	+	;	K	[	k	{
<b>1100</b>	FF	FS	,	<	L	\	l	
<b>1101</b>	CR	GS	-	=	M	]	m	}
<b>1110</b>	O	RS	.	>	N	^	n	~
<b>1111</b>	SI	US	/	?	O	_	o	DEL

- Parity Bit

Errors may occur during data transmission. If we could detect the error, we could request for re-transmission. Better still, if the error could be corrected automatically, it would save the re-transmission, but this requires a more complex mechanism.

The Biquinary code uses 7 bits, 3 more bits than the usual 4-bit codes such as BCD, to provide error-detection capability. If a supposedly Biquinary code received shows 0101001, we detect the error instantly.

For the detection of single-bit error – a bit is inverted during the transmission – a simple error-detection scheme called the *parity bit* scheme can be used. An additional parity bit is attached (usually appended) to the data. The parity bit scheme comes in two flavours: the *odd parity* and the *even parity*.

In the odd parity scheme, the additional parity bit is chosen so that the total number of 1's in the data, including the parity bit, is an odd number. Likewise, for even parity scheme, the total number of 1's in the data and parity must be even.

If we use the odd parity scheme on the ASCII character set, then the character 'A', having the ASCII value 1000001, would be appended with the parity bit 1 to become 10000011.

The parity bit scheme detects only single-bit error. If two bits are inverted during data transmission, it would go undetected under this scheme.

The parity bit scheme may be extended to a block of data. For example, in Figure 8, a block of five 4-bit words is shown on the left. On the right, it is shown how a parity bit is

appended to each of the five words, and an additional parity word is added where each bit in that parity word is the parity bit for its associated column. Odd parity scheme is assumed in this example.

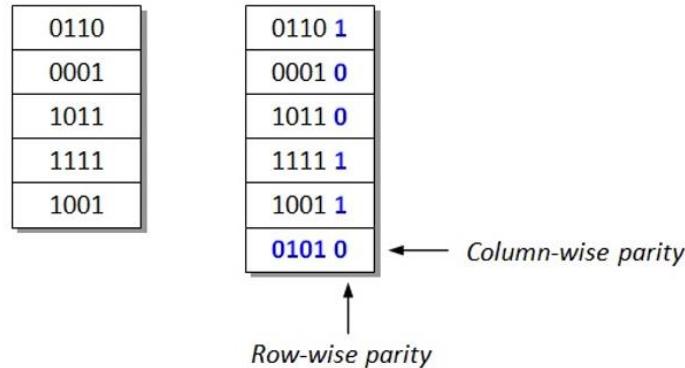


Figure 8 Applying parity bits on a block of data

The parity bit scheme has limited ability on error correction. More complex schemes, such as the *Hamming code*, are needed to do error correction.

## 12 Gray Code

We shall conclude this chapter with a special code – the *Gray code*. The Gray code is neither a decimal code, nor is it an arithmetic code created for the purpose of computation.

The essential feature of a Gray code is that there is only a *single bit change* from one code value to the next. The sequence is circular. This feature makes the Gray code suitable for error detection applications.

With the above feature in mind, the following two sequences are examples of 2-bit Gray codes:

Sequence #1: 00 → 01 → 11 → 10 → (back to 00)

Sequence #2: 00 → 10 → 11 → 01 → (back to 00)

Other sequences can be derived from the above two sequences, by choosing one of the code values as the first code value in the cyclic sequence. Among all the valid Gray code sequences, we adopt the first sequence above as the standard 2-bit Gray code sequence.

Table 9 shows the standard 4-bit Gray code sequence. The binary sequence is shown alongside the Gray code sequence for comparison.

The	Standard	4-bit	Gray	Code
-----	----------	-------	------	------

Table 9:-

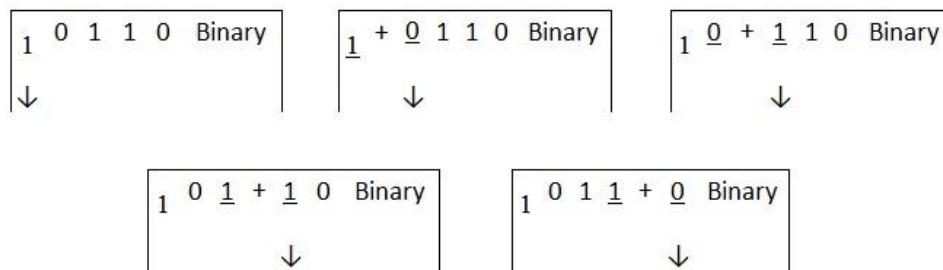
Decimal	Binary	Gray code	Decimal	Binary	Gray code
0	0000	0000	8	1000	1100
1	0001	0001	9	1001	1101
2	0010	0011	10	1010	1111
3	0011	0010	11	1011	1110
4	0100	0110	12	1100	1010
5	0101	0111	13	1101	1011
6	0110	0101	14	1110	1001
7	0111	0100	15	1111	1000

- Binary-to-Gray Conversion

The algorithm to convert a binary value to its corresponding standard Gray code value is as follows:

1. Retain the MSB.
2. From left to right, add each adjacent pair of binary code bits to get the next Gray code bit, discarding the carry.

The following example shows the conversion of binary number  $(10110)_2$  to its corresponding standard Gray code value,  $(11101)_{\text{Gray}}$ .



- Gray-to-Binary Conversion

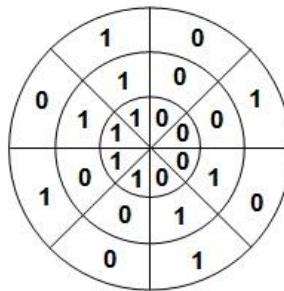
The algorithm to convert a standard Gray code value to its corresponding binary value is as follows:

1. Retain the MSB.
2. From left to right, add each binary code bit generated to the Gray code bit in the next position, discarding the carry.

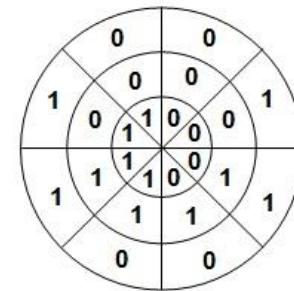
The following example shows the conversion of the standard Gray code value  $(11011)_{\text{Gray}}$  to its corresponding binary value,  $(10010)_2$ .

$\begin{array}{ccccc} 1 & 1 & 0 & 1 & 1 \\ \downarrow & & & & \end{array}$ Gray	$\begin{array}{ccccc} 1 & \underline{1} & 0 & 1 & 1 \\ + \downarrow & & & & \end{array}$ Gray	$\begin{array}{ccccc} 1 & 1 & \underline{0} & 1 & 1 \\ + \downarrow & & & & \end{array}$ Gray
$\begin{array}{ccccc} 1 & 1 & 0 & \underline{1} & 1 \\ + \downarrow & & & & \end{array}$ Gray	$\begin{array}{ccccc} 1 & 1 & 0 & 1 & \underline{1} \\ + \downarrow & & & & \end{array}$ Gray	

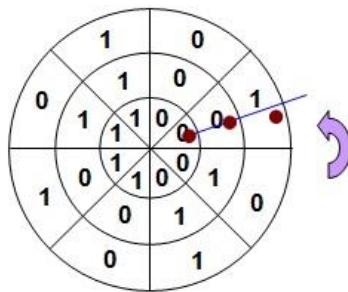
Let us consider a disk whose surface is divided into eight sectors. Each sector is identified by a 3-bit code. Figure 9a shows the binary coding on the sectors, and Figure 9b the Gray coding.



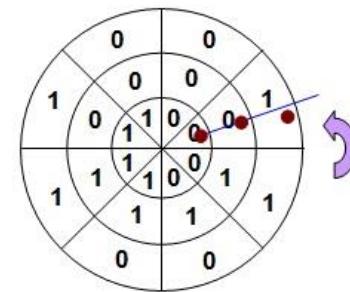
(a) Binary coded sectors.



(b) Gray coded sectors.



(c) Binary coded:  $001 \rightarrow 000 \rightarrow 010$ .



(d) Gray coded:  $001 \rightarrow 011 \rightarrow 010 \rightarrow 000$ .

Fig:9a:Show Binary coding

Fig:9b: Show Gray coding

With misaligned sensors, on a disk with binary coded sectors, Figure9c shows an instance where the system registers the transition from sector 001 to 000 to 010 as the disk rotates, where in fact the correct transition should be sector 001 to 010. The spurious sector 000 registered is due to the binary coded sequence, as one of the misaligned sectors crosses into the next sector before the other sensors. Figure 9d shows that a Gray coded sequence would eliminate such error.

### Quick Review Questions

2-1. Convert the binary number 1011011 to its decimal equivalent.

- a. 5                  b. 63                  c. 91                  d. 92                  e. 139

2-2. What is the weight of the digit ‘3’ in the base-7 number 12345?

- a. 3                  b. 7                  c. 14                  d. 21                  e. 49

2-3. Which of the following has the largest value?

- a.  $(110)_{10}$       b.  $(10011011)_2$     c.                  d.  $(1111)_5$                   e.                  f.  $(222)_8$

2-4. If  $(321)_4 = (57)_{10}$ , what is the decimal equivalent of  $(321000000)_4$ ?

- a.  $57 \cdot 10^4$       b.  $57 \cdot 10^6$       c.  $57 \cdot 4^4$       d.  $57 \cdot 4^6$       e.  $57^4$

2-5. Convert each of the following decimal numbers to binary (base two) with at most eight digits in the fractional part, rounded to eight places.

- a. 2000                  b. 0.875                  c. 0.07                  d. 12.345

2-6. Convert each of the decimal numbers in Question 2-5 above to septimal (base seven) with at most six digits in the fractional part, rounded to six places.

2-7. Convert each of the decimal numbers in Question 2-5 above to octal (base eight) with at most four digits in the fractional part, rounded to four places.

2-8. Convert each of the decimal numbers in Question 2-5 above to hexadecimal (base sixteen) with at most two digits in the fractional part, rounded to two places.

2-9. Which of the following octal values is equivalent to the binary number  $(110001)_2$ ?

- a.  $(15)_8$       b.  $(31)_8$       c.  $(33)_8$       d.  $(49)_8$       e.  $(61)_8$

2-10. Convert the binary number  $(1001101)_2$  to

- a. quaternary    b. octal    c. decimal    d. hexadecimal

2-11. What is  $(1011)_2 \times (101)_2$ ?

- a.  $(10000)_2$     b.  $(110111)_2$     c.  $(111111)_2$     d.  $(111011)_2$     e.  $(101101)_2$

2-12. Perform the following operations on binary numbers.

a.  $(10111110)_2 + (10001101)_2$

b.  $(11010010)_2 - (01101101)_2$

c.  $(11100101)_2 - (00101110)_2$

2-13. In a 6-bit 2's complement binary number system, what is the decimal value represented by  $(100100)_{2s}$ ?

- a. -4    b. 36    c. -36    d. -27    e. -28

2-14. In a 6-bit 1's complement binary number system, what is the decimal value represented by  $(010100)_{1s}$ ?

- a. -11    b. 43    c. -43    d. 20    e. -20

2-15. What is the range of values that can be represented in a 5-bit 2's complement binary system?

- a. 0 to 31    b. -8 to 7    c. -8 to 8    d. -15 to 15    e. -16 to 15

2-16. In a 4-bit 2's complement scheme, what is the result of this operation:  $(1011)_{2s} + (1001)_{2s}$ ?

- a. 4    b. 5    c. 20    d. -12    e. overflow

2-17. Assuming a 6-bit 2's complement system, perform the following subtraction operations by converting it into addition operations:

a.  $(011010)_{2s} - (010000)_{2s}$

b.  $(011010)_{2s} - (001101)_{2s}$

c.  $(000011)_{2s} - (010000)_{2s}$

2-18. Assuming a 6-bit 1's complement system, perform the following subtraction operations by converting it into addition operations:

a.  $(011111)_{1s} - (010101)_{1s}$

b.  $(001010)_{1s} - (101101)_{1s}$

c.  $(100000)_{1s} - (010011)_{1s}$

2-19. Which of the following values cannot be represented accurately in the 8-bit sign-and-magnitude fixed-point number format shown in Figure 2.4?

- a. 4                  b.  $-29.5$                   c. 20.2                  d.  $-3.75$                   e. 12.25

2-20. What does 1 110 1001 represent in this floating-point number scheme: 1-bit sign, 3-bit normalized mantissa, followed by 4-bit 2's complement exponent?

- a.  $0.125 \times 2^9$     b.  $-0.125 \times 2^9$     c.  $-0.75 \times 2^{-1}$     d.  $-0.75 \times 2^{-6}$     e.  $-0.75 \times 2^{-7}$

2-21. How to represent  $(246)_{10}$  in the following system/code?

- a. 10-bit binary    b. BCD                  c. Excess-3    d. 2421 code    e. 84-2-1 code

2-22. The decimal number 573 is represented as 1111 0110 1011 in an unknown self-complementing code. Find the code for the decimal number 642.

2-23. Convert  $(101011)_2$  to its corresponding Gray code value.

- a.  $(101011)_{\text{Gray}}$     b.  $(010100)_{\text{Gray}}$     c.  $(110010)_{\text{Gray}}$     d.  $(111110)_{\text{Gray}}$     e.  $(43)_{\text{Gray}}$

2-24. Convert  $(101011)_{\text{Gray}}$  to its corresponding binary value.

- a.  $(101011)_2$     b.  $(010100)_2$     c.  $(110010)_2$     d.  $(111110)_2$     e.  $(010101)_2$

### Answers to Quick Review Questions

- 2-1. (c)    2-2. (e)    2-3. (c)    2-4. (d)

2-5. (a)  $(2000)_{10} = (11111010000)_2$       (b)  $(0.875)_{10} = (0.111)_2$

(c)  $(0.07)_{10} = (0.00010010)_2$       (d)  $(12.345)_{10} = (1100.01011000)_2$

2-6. (a)  $(2000)_{10} = (5555)_7$       (b)  $(0.875)_{10} = (0.606061)_7$

(c)  $(0.07)_{10} = (0.033003)_7$  or  $(0.033004)_7$

(d)  $(12.345)_{10} = (15.226223)_7$

2-7. (a)  $(2000)_{10} = (3720)_8$       (b)  $(0.875)_{10} = (0.7)_8$

(c)  $(0.07)_{10} = (0.0437)_8$       (d)  $(12.345)_{10} = (14.2605)_8$

2-8. (a)  $(2000)_{10} = (7D0)_{16}$       (b)  $(0.875)_{10} = (0.E)_{16}$

(c)  $(0.07)_{10} = (0.12)_{16}$       (d)  $(12.345)_{10} = (C.58)_{16}$

2-9. (e)

2-10. (a)  $(1031)_4$       (b)  $(115)_8$       (c)  $(77)_{10}$       (d)  $(4D)_{16}$

2-11. (b)

2-12. (a)  $(101001011)_2$       (b)  $(01100101)_2$       (c)  $(10110111)_2$

2-13. (e)      2-14. (d)      2-15. (e)      2-16. (e)

2-17. (a)  $(001010)_{2s} = (10)_{10}$       (b)  $(001101)_{2s} = (13)_{10}$       (c)  $(110011)_{2s} = -(13)_{10}$

2-18. (a)  $(001010)_{1s} = (10)_{10}$       (b)  $(011100)_{1s} = (28)_{10}$       (c) overflow

2-19. (c)      2-20. (e)

2-21. (a)  $(0011110110)_2$       (b)  $(0010\ 0100\ 0110)_{BCD}$       (c)  $(0101\ 0111\ 1001)_{\text{Excess-3}}$

(d)  $(0010\ 0100\ 1100)_{2421}$  (e)  $(0110\ 0100\ 1010)_{84-2-1}$

2-22.  $642 = 0100\ 0000\ 1001$

2-23. (d)      2-24 .(c)

## Exercises

2-25. How is subtraction carried out for binary numbers represented in the sign-and-magnitude form?

2-26. According to the fixed-point number format in Figure 2.4, what are the largest and smallest positive values that can be represented? What are the largest and smallest negative values?

2-27. Find out how the special values are represented in the IEEE Standard 754 floating-point representation.

2-28. Perform the following number system conversions:

- |  |                             |                       |
|--|-----------------------------|-----------------------|
| (a) $1101011_2 = ?_{16}$                     | (d) $10100.1101_2 = ?_{10}$ | (g) $125_{10} = ?_2$  |
| (b) $67.24_8 = ?_2$                          | (e) $7156_8 = ?_{10}$       | (h) $1435_{10} = ?_8$ |
| (c) DEAD.BEEF <sub>16</sub> = ? <sub>8</sub> | (f) $15C.38_{16} = ?_{10}$  |                       |

2-29. Sometimes we need to round a value to a specified number of places. Perform the following rounding, providing an answer that is most accurate.

- (a) The answer for question 2-28b, correct to two places.
- (b) The answer for question 2-28b, correct to one place.
- (c) The answer for question 2-28c, correct to three places.
- (d) The answer for question 2-28c, correct to two places.

2-30. Add the following pairs of unsigned binary numbers (unsigned numbers are non-negative values), showing all carries:

$$(a) \begin{array}{r} 110101 \\ + 11001 \\ \hline \end{array}$$

$$(b) \begin{array}{r} 1110010 \\ + 1101101 \\ \hline \end{array}$$

2-31. Write the 8-bit sign-and-magnitude, 1's-complement, and 2's-complement representations for each of these decimal numbers:

+18, +115, +79, -49, -3, -100.

2-32. The diminished radix complement of base-6 numbers is called the 5's complement.

- (a) Obtain the 5's complement of  $(543210)_6$ .
- (b) Design a 3-bit self-complementing code to represent each of the six digits of the base-6 number system.

2-33. Convert the following binary numbers into standard Gray codes:

- (a)  $01011_2$
- (b)  $101101_2$
- (c)  $10101111_2$

2-34. Convert the following standard Gray codes in to binary numbers:

- (a)  $01011_{\text{Gray}}$
- (b)  $101101_{\text{Gray}}$
- (c)  $10101111_{\text{Gray}}$

2-35. Represent the decimal 3951 in the following coding schemes:

- (a) BCD code
- (b) Excess-3 code
- (c) 2421 code
- (d) 84-2-1 code
- (e) ASCII (ASCII code for character '0' is  $48_{10}$ , or  $0110000_2$ .)

2-36. Complete the following sequence so that the result is a 3-bit Gray code sequence.

011  $\circledast$  ?  $\circledast$  ?  $\circledast$  100  $\circledast$  101  $\circledast$  001  $\circledast$  ?  $\circledast$  ?

## Boolean Algebra Logic and Logic Gates

### Basic Theorems and Properties of Boolean Algebra:

#### Duality

- We simply interchange OR and AND operators and replace 1's by 0's and 0's by 1's. Also, the duality principle states that if two Boolean expressions are equal, then their duals are also equal. Ex:  $X + 0 = X$  &  $X \cdot 1 = X$

#### Basic Theorem

- Boolean Addition:

$$x + 0 = 0 + x = x \quad (0 \text{ is called the additive identity})$$

$$x + 1 = 1 \quad (X = AB + C \text{ è } AB + C + 1 = 1)$$

- Boolean Multiplication

$$x \cdot 0 = 0$$

$$x \cdot 1 = 1 \cdot x = x \quad (1 \text{ is called the multiplicative identity})$$

- Idempotent Laws

$$x + x = x$$

$$x \cdot x = x$$

- Involution Law

$$(x')' = x$$

- Laws of Complement

$$x + x' = 1$$

$$x \cdot x' = 0$$

- Commutative Laws

$$x + y = y + x$$

$$x \cdot y = y \cdot x$$

- Associative Laws

$$(x + y) + z = x + (y + z) = x + y + z$$

$$(x \cdot y) \cdot z = x(y \cdot z) = x \cdot y \cdot z$$

- Distributive Law

$$x \cdot (y + z) = (x \cdot y) + (x \cdot z)$$

$$x + (y \cdot z) = (x + y) \cdot (x + z)$$

- De Morgan's Laws

$$(x + y)' = x' y'$$

$$(x \cdot y)' = x' + y'$$

- Consensus Law

$$x \cdot y + x' \cdot z + y \cdot z = x \cdot y + x' \cdot z$$

The term  $yz$  is referred to as the "consensus term". A consensus term is a redundant term and it can be eliminated. Given a pair of terms for which a variable appears in one term and the complement of that variable in another term, the consensus term is formed by multiplying the two original terms together, leaving out the selected variables and its complement.

- A field is a set of elements, together with two binary operators.
- The set of real numbers together with the binary operators  $+$  and  $\cdot$  form the field of real numbers.
- The field of real numbers is the basis for arithmetic and ordinary algebra. The operators and postulates have the following meanings:

The binary operator  $+$  defines addition.

The additive identity is 0.

The additive inverse defines subtraction.

The binary operator  $\cdot$  defines multiplication.

The multiplicative identity is 1.

The multiplicative inverse of  $a = 1/a$  defines division, i.e.  $a \cdot 1/a = 1$ .

The only distributive law applicable is that of  $\cdot$  over  $+$ :

$$a \cdot (b + c) = (a \cdot b) + (a \cdot c)$$

- A two-valued Boolean Algebra is defined on a set of 2 elements,  $B = \{0, 1\}$ , with rules for the 2 binary operators + and  $\cdot$ .

X	Y	$X \cdot Y$
0	0	0
0	1	0
1	0	0
1	1	1

X	Y	$X + Y$	X	$X'$
0	0	0	0	1
0	1	1	1	0
1	0	1	1	0
1	1	1	1	0

X	Y	Z	$Y + Z$	$X \cdot (Y + Z)$	$X \cdot Y$	$X \cdot Z$	$(X \cdot Y) + (X \cdot Z)$
0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	1	1	0	0	0	0
1	0	0	0	0	0	0	0
1	0	1	1	1	0	1	1
1	1	0	1	1	1	0	1
1	1	1	1	1	1	1	1

Truth table to verify the Distributive Law

A) $X \ Y$			$X + Y$	$X' + Y'$	B )			$X$	$Y$	$X'$	$Y'$	$X' \cdot Y'$
0	0	0		1				0	0	1	1	1
0	1	1		0				0	1	1	0	0
1	0	1		0				1	0	0	1	0
1	1	1		0				1	1	0	0	0

## Logic Gates

NAME	GRAPHIC SYMBOL	ALGEBRAIC FUNCTION	TRUTH TABLE															
OR		$F = x + y$	<table border="1"> <thead> <tr> <th>x</th><th>y</th><th>F</th></tr> </thead> <tbody> <tr> <td>0</td><td>0</td><td>0</td></tr> <tr> <td>0</td><td>1</td><td>1</td></tr> <tr> <td>1</td><td>0</td><td>1</td></tr> <tr> <td>1</td><td>1</td><td>1</td></tr> </tbody> </table>	x	y	F	0	0	0	0	1	1	1	0	1	1	1	1
x	y	F																
0	0	0																
0	1	1																
1	0	1																
1	1	1																
AND		$F = xy$	<table border="1"> <thead> <tr> <th>x</th><th>y</th><th>F</th></tr> </thead> <tbody> <tr> <td>0</td><td>0</td><td>0</td></tr> <tr> <td>0</td><td>1</td><td>0</td></tr> <tr> <td>1</td><td>0</td><td>0</td></tr> <tr> <td>1</td><td>1</td><td>1</td></tr> </tbody> </table>	x	y	F	0	0	0	0	1	0	1	0	0	1	1	1
x	y	F																
0	0	0																
0	1	0																
1	0	0																
1	1	1																
BUFFER		$F = x$	<table border="1"> <thead> <tr> <th>x</th><th>F</th></tr> </thead> <tbody> <tr> <td>0</td><td>0</td></tr> <tr> <td>1</td><td>1</td></tr> </tbody> </table>	x	F	0	0	1	1									
x	F																	
0	0																	
1	1																	
INVERTER		$F = x'$	<table border="1"> <thead> <tr> <th>x</th><th>F</th></tr> </thead> <tbody> <tr> <td>0</td><td>1</td></tr> <tr> <td>1</td><td>0</td></tr> </tbody> </table>	x	F	0	1	1	0									
x	F																	
0	1																	
1	0																	
EXCLUSIVE OR XOR		$F = xy' + x'y$ $x \oplus y$	<table border="1"> <thead> <tr> <th>x</th><th>y</th><th>F</th></tr> </thead> <tbody> <tr> <td>0</td><td>0</td><td>0</td></tr> <tr> <td>0</td><td>1</td><td>1</td></tr> <tr> <td>1</td><td>0</td><td>1</td></tr> <tr> <td>1</td><td>1</td><td>0</td></tr> </tbody> </table>	x	y	F	0	0	0	0	1	1	1	0	1	1	1	0
x	y	F																
0	0	0																
0	1	1																
1	0	1																
1	1	0																
NOR		$F = (xy)'$	<table border="1"> <thead> <tr> <th>x</th><th>y</th><th>F</th></tr> </thead> <tbody> <tr> <td>0</td><td>0</td><td>1</td></tr> <tr> <td>0</td><td>1</td><td>0</td></tr> <tr> <td>1</td><td>0</td><td>1</td></tr> <tr> <td>1</td><td>1</td><td>0</td></tr> </tbody> </table>	x	y	F	0	0	1	0	1	0	1	0	1	1	1	0
x	y	F																
0	0	1																
0	1	0																
1	0	1																
1	1	0																
EXCLUSIVE NOR or EQUIVALENCE		$F = xy + x'y'$ or $= x \odot y$	<table border="1"> <thead> <tr> <th>x</th><th>y</th><th>F</th></tr> </thead> <tbody> <tr> <td>0</td><td>0</td><td>1</td></tr> <tr> <td>0</td><td>1</td><td>0</td></tr> <tr> <td>1</td><td>0</td><td>0</td></tr> <tr> <td>1</td><td>1</td><td>1</td></tr> </tbody> </table>	x	y	F	0	0	1	0	1	0	1	0	0	1	1	1
x	y	F																
0	0	1																
0	1	0																
1	0	0																
1	1	1																
NOR		$F = (x + y)'$	<table border="1"> <thead> <tr> <th>x</th><th>y</th><th>F</th></tr> </thead> <tbody> <tr> <td>0</td><td>0</td><td>1</td></tr> <tr> <td>0</td><td>1</td><td>0</td></tr> <tr> <td>1</td><td>0</td><td>0</td></tr> <tr> <td>1</td><td>1</td><td>0</td></tr> </tbody> </table>	x	y	F	0	0	1	0	1	0	1	0	0	1	1	0
x	y	F																
0	0	1																
0	1	0																
1	0	0																
1	1	0																

## Operator Precedence

- The operator precedence for evaluating Boolean expressions is (1) (), (2) NOT, (3) AND, and (4) OR. Ex:  $(x + y)'$ .

## Venn Diagram

- See Fig. 2 – 1, 2 – 2 Page 44.

## Boolean Functions

- A Boolean function is an expression formed with binary variables, the 2 two binary operators **OR** and **AND**, and unary operator **NOT**, parentheses, and an equal sign. For a given value of the variables, the function can be either **0** or **1**.
- Ex:  $F_1 = xyz'$ . The function is equal to 1 if  $x = 1$  and  $y = 1$   $z' = 1$ ; otherwise  $F_1 = 0$ .
- To represent a function in a truth table, we a list of  $2^n$  combinations of **1**'s and **0**'s of  $n$  binary variables and a column to show the combination for which the function = **0**, **1**.

Table 2

Truth tables for  $F_1 = xyz'$ ,  $F_2 = x + y'z$ ,  $F_3 = x'y'z + x'yz + xy'$ , and  $F_4 = xy' + x'z$

$x$	$y$	$z$	$F_1$	$F_2$	$F_3$	$F_4$
0	0	0	0	0	0	0
0	0	1	0	1	1	1
0	1	0	0	0	0	0
0	1	1	0	0	1	1
1	0	0	0	1	1	1
1	0	1	0	1	1	1
1	1	0	1	1	0	0
1	1	1	0	1	0	0

Two functions of  $n$  binary values are said to be equal if they have the same value for all possible  $2^n$  combination of the  $n$  variables. è  $F_3$  and  $F_4$

- A Boolean function may be transformed from an algebraic expression into a logic diagram composed of AND, OR, and NOT gates. Fig. 4.
- Since  $F_3$  and  $F_4$  equal Boolean functions, it is more economical to implement the  $F_4$  form than the  $F_3$  form.

### Algebraic Manipulation

- A *literal* is a primed or unprinted variable. When a Boolean function is implemented with logic gates. Each literal in the function designates an input to a gate, and each term is implemented with a gate.
- The number of literals in a Boolean function can be minimized by algebraic manipulations.
- There are no specific rules to follow that will guarantee the final answer.
- The only method available is a cut-and-try procedure employing postulates, basic theorems, and other manipulation method that becomes familiar with use.

**Ex:**

Simplify the following Boolean functions to a minimum number of literals.

1.  $x + x' y = (x + x') (x + y) = 1 (x + y) = x + y$
2.  $x (x' + y) = (x x') + (x y) = 0 + xy = xy$
3.  $x' y' z + x' y z + x y' = x' z (y + y') + x y' = x' z + x y'$
4.  $x y + x' z + y z = x y + x' z + y z (x + x')$   
 $= x y + x' z + x y z + x' y z$   
 $= x y (1 + z) + x' z (1 + y)$   
 $= x y + x' z$

5.  $(x + y) (x' + z) (y + z) = (x + y) (x' + z)$  by duality from function 4.

$$(x + y) (x' + z) (y + z) = (\cancel{x} \cancel{x'} + x \cdot z + x' \cdot z + y \cdot z) (y + z)$$

0

$$\begin{aligned}
 &= (x \cdot y \cdot z + x \cdot \cancel{z} \cdot \cancel{z} + x' \cdot \cancel{y} \cdot \cancel{y} + x' \cdot y \cdot z + y \cdot \cancel{z} \cdot \cancel{z} + y \cdot \cancel{z} \cdot \cancel{z}) \\
 &\quad z \qquad \qquad \qquad y \qquad \qquad \qquad z \qquad \qquad z \\
 &= (x \cdot y \cdot z + x \cdot z + x' \cdot y + x' \cdot y \cdot z + y \cdot z) \\
 &= x \cdot z (1+y) + x' \cdot y (1+z) + y \cdot z \\
 &= \quad \quad \quad 1 \qquad \qquad \qquad 1 \\
 &= x \cdot z + x' \cdot y + y \cdot z \\
 &= \cancel{x} \cdot \cancel{x'} + x \cdot z + x' \cdot y + y \cdot z \\
 &= 0 \\
 &= (x + y) (x' + z)
 \end{aligned}$$

- Functions 1 and 2 are the duals of each other and use dual expressions in corresponding steps.

### ***Complement of a function***

- The complement of a function  $F$  is  $F'$  and is obtained from an interchange of 0's for 1's and 1's for 0's in the values of  $F$ .
- The complement of a function may be derived algebraically through De Morgan's
- $(A + B + C + D + \dots + F)' = A' B' C' D' \dots F'$
- $(A B C D \dots F)' = A' + B' + C' + D' + \dots + F'$
- The generalized form of De Morgan's theorem states that the complement of a function is obtained by interchanging AND and OR operates and complementing each literal.
- $F_1 = x' y z' + x' y' z$

$$F'_1 = (x + y' + z) (x + y + z')$$

$$F_2 = x (y' z' + y z)$$

$$F'_2 = x' + (y + z) (y' + z')$$

## Representations of A Function

- A function can be specified or represented in any of the following ways:
  - o A truth table
  - o A circuit
  - o A Boolean expression
    - SOP (Sum Of Products)
    - POS (Product of Sums)
    - Canonical SOP
    - Canonical POS

## Canonical And Standard Forms

### Min terms and Max terms

- A binary variable may appear in normal form ( $x$ ) or in complement form ( $x'$ )
- Ex: Two binary variables  $x$  and  $y$  combined with an AND gate. There are four combinations:  $x'y'$ ,  $x'y$ ,  $xy'$ , &  $xy$ .
- Each of these four AND terms represents one of the distinct areas in the Venn diagram and is called a *min term* or a *standard product*.
- Similarly,  $n$  variables can be combined to form  $2^n$  min terms. The  $2^n$  different min terms may be determined by a method similar to the table shown below.
- Each min term is obtained from an **AND** term of the  $n$  variables, with each variable being primed if the corresponding bit of the binary number is a 0 and unprimed if a 1.
- In a similar fashion,  $n$  variables can be combined to form  $2^n$  *max terms* or *standard sums*. The  $2^n$  different max terms may be determined by a method similar to the table shown below.
- Each max term is obtained from an **OR** term of the  $n$  variables, with each variable being primed if the corresponding bit of the binary number is a 1 and unprimed if a 0.
- Each max-term is the complement of its corresponding min term, and vice versa.

- A Boolean function may be expressed algebraically from a given truth table by forming a min term for each combination of the variables that produces a 1 in the function and then taking the **OR** of those terms.

Minterms			Maxterms			
x	y	z	Term	Designation	Term	Designation
0	0	0	$x'y'z'$	$m_0$	$x + y + z$	$M_0$
0	0	1	$x'y'z$	$m_1$	$x + y + z'$	$M_1$
0	1	0	$x'y z'$	$m_2$	$x + y' + z$	$M_2$
0	1	1	$x'y z$	$m_3$	$x + y' + z'$	$M_3$
1	0	0	$xy'z'$	$m_4$	$x' + y + z$	$M_4$
1	0	1	$xy'z$	$m_5$	$x' + y + z'$	$M_5$
1	1	0	$xy z'$	$m_6$	$x' + y' + z$	$M_6$
1	1	1	$xy z$	$m_7$	$x' + y' + z'$	$M_7$

- Ex: the function in the following table is determined by expressing the combination of 001, 100, and 111 as  $x'y'z$ ,  $xy'z'$ , and  $xyz$ . Since each one of these minterms results in  $f_1 =$ , we should have

$$f_1 = x'y'z + xy'z' + xyz = m_1 + m_4 + m_7$$

- Similarly, we can verify that

$$f_2 = x'yz + xy'z + xyz' + xyz = m_3 + m_5 + m_6 + m_7$$

- The complement of  $f_1$  is “may be read from the truth table by forming a min term for each combination that produces a 0 in the function and then ORing those terms”:

$$f'_1 = x'y'z' + x'yz' + x'yz + xy'z + xyz'$$

### Functions of Three Variables

<i>x</i>	<i>y</i>	<i>z</i>	<i>Function f<sub>1</sub></i>	<i>Function f<sub>2</sub></i>
0	0	0	0	0
0	0	1	1	0
0	1	0	0	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

- If we take the complement of  $f'_1$ , we obtain the function  $f_1$ :

$$\begin{aligned}f_1 &= (x + y + z)(x + y' + z)(x + y' + z')(x' + y + z')(x' + y' + z) \\&= M_0 \cdot M_2 \cdot M_3 \cdot M_5 \cdot M_6\end{aligned}$$

- Similarly, it is possible to read the expression for  $f_2$  from the table:

$$\begin{aligned}f_2 &= (x + y + z)(x + y + z')(x + y' + z)(x' + y + z')(x' + y + z) \\&= M_0 \cdot M_1 \cdot M_2 \cdot M_4\end{aligned}$$

- Boolean functions expressed as a sum of min terms or products of max terms are said to be in *canonical form*

Sum			of	Min	terms
A	B	C	F		
0	0	0	0		
0	0	1	1		
0	1	0	0		
0	1	1	0		
1	0	0	1		
1	0	1	1		
1	1	0	1		
1	1	1	1		

$$F(A, B, C) = \Sigma(1, 4, 5, 6, 7)$$

Ex: Express the function  $F = A + B'C$  in a sum of min terms. The function has 2 variables. Term A is missing 2 variables:

$$A = A(B + B') = AB + AB' \quad \text{Now } A \text{ is missing 1 variable}$$

$$A = AB(C + C') + AB'(C + C')$$

$$A = ABC + ABC' + AB'C + AB'C'$$

$B'C$  is missing one variable:

$$B'C = B'C(A + A') = AB'C + A'B'C \quad \text{Now we combine terms}$$

$$F = ABC + ABC' + AB'C + AB'C' + AB'C + A'B'C$$

$$= A'B'C + AB'C' + AB'C + ABC' + ABC$$

$$= m_1 + m_4 + m_5 + m_6 + m_7$$

$$F(A, B, C) = \Sigma(1, 4, 5, 6, 7)$$

Product of Max terms

Truth Table for  $F = xy + x'z$

x	y	z	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

The sum of min terms is

$$F(x, y, z) = \Sigma(1, 3, 6, 7)$$

The sum of max terms is

$$F(x, y, z) = \prod(0, 2, 4, 5)$$

Ex: Express the Boolean function  $F = xy + x'z$  in a product of max terms. Convert the function into OR terms using the distributive law:

$$F = xy + x'y = (xy + x')(xy + z)$$

$$= (\cancel{x+x'}) (y + x') (x + z) (y + z)$$

$$= (x' + y) (x + z) (y + z)$$

The function has 3 variables: x, y, & z. Each OR term is missing one variable:

$$x' + y = x' + y + zz' = (x' + y + z) (\cancel{x' + y + z'})$$

$$x + z = x + z + yy' = (\cancel{x + y + z}) (x + y' + z)$$

$$y + z = y + z + xx' = (\cancel{x + y + z}) (\cancel{x' + y + z'})$$

Combine the terms and remove the terms that appear more than once:

$$\begin{aligned} F &= (x + y + z) (x + y' + z) (x' + y + z) (x' + y + z') \\ &= M_0 + M_2 + M_4 + M_5 \end{aligned}$$

The function is expressed as follows:  $F(x, y, z) = \prod (0, 2, 4, 5)$

### Conversion between Canonical Forms

- Ex: The complement of

$$F(A, B, C) = \Sigma (1, 4, 5, 6, 7) \text{ is}$$

$$F'(A, B, C) = \Sigma (0, 2, 3) = m_0 + m_2 + m_3$$

- If we apply the complement of  $F'$  using De Morgan's theorem, we obtain  $F$  in a different form:

$$F = (m_0 + m_2 + m_3)' = m'_0 \cdot m'_2 \cdot m'_3 = M_0 + M_2 + M_3 = \prod (0, 2, 3)$$

- The last conversion follows from the definition of min terms and max terms. It is clear that the following relation holds true:  $m'_j = M_j$  è the max term with subscript  $j$  is a complement of the min term with the same subscript  $j$ , and vice versa.
- To convert from one canonical form to another, interchange the symbol  $\Sigma$  and  $\prod$  and list those numbers missing from the original form.

### Integrated Circuits “IC”

- It's a small silicon semiconductor, called a chip, containing the electronic components for the digital gates. The gates are interconnected inside the chip to form the required circuit.

### Levels of Integration

- *Small-scale Integration (SSI)*: contains several independent gates in a single package. The number of gates is usually fewer than 10.

- *Medium-scale Integration (MSI)*: have a complexity between 10 and 100 gates in a single package. They perform specific digital operations such as decoders, adders, and multiplexers.
  - *Large-scale Integration (LSI)*: contains between 100 and 1000s gates in a single package. The number of gates is usually fewer than 10. They include processors, memory chips, and programmable logic devices.
  - *Very Large-scale Integration (VLSI)*: contains thousands of gates in a single package. Examples are large memory arrays and complex microcomputer chips.
- 

## C-PROGRAM

### THE ORIGIN OF C++

The C programming language was developed at AT&T for the purpose of writing the operating system for the PDP-11 series of computers which ultimately became the UNIX operating system. C was developed with the primary goal of operating efficiency. **Bjarne Stroustrup**, also of AT&T, developed C++ in order to add object oriented constructs to the C language. Because object oriented technology was new at the time and all existing implementations of object oriented languages were very slow and inefficient, the primary goal of C++ was to maintain the efficiency of C.

C++ can be viewed as a traditional procedural language with some additional constructs. Beginning with C, some constructs are added for object oriented programming and some for improved procedural syntax. A well written C++ program will reflect elements of both object oriented programming style and classic procedural programming. C++ is actually an extendible language since we can define new types in such a way that they act just like the predefined types which are part of the standard language. C++ is designed for large scale software development.

### C++ syntax

By definition, *syntax* is the rules governing the formation of statements in a programming language. In other words, it is the specific language (code) you use to turn algorithms into programs and specific rules you have to follow when programming with C++. As you will find out, if you make one small mistake when writing code, the compiler will notify you of the syntax error when you try to run it. C++, like all languages, is very picky when it comes to syntax. Let's look at a simple example program written in C++.

For now, notice how each executable statement in the program is ended with a semi-colon. Every executable statement must always be terminated with a semi-colon, which signals that it is the end of the statement. Any code placed after // will not affect the code; // is used for comments and documentation. Also notice that text is always enclosed in parentheses. Let's dig deep into the code to find out why you need to use certain code statements.

- `cout` - *console output* - It is simply letting the compiler know that something is being sent into the output stream (normally the screen)
- `<<` - *insertion operator* - cout and the insertion operator go hand and hand; you must use it whenever you issue a cout command
- `cin` - *console input* - It is simply letting the compiler know that something is being sent into the input stream (normally into the memory of a computer)
- `>>` - *extraction operator* - cin and the extraction operator go hand and hand; you must use it whenever you issue a cin command
- `endl` - *end line* - basically the same as pressing the return key in a text editor; returns down to the next line

## Lexical Conventions

Introduces the fundamental elements of a C++ program as they are meaningful to the compiler are used to construct statements, definitions, declarations, and so on, which are used to construct complete programs.

The fundamental elements of a C++ program are called “lexical elements” or “tokens” to construct statements, definitions, declarations, and so on, which are used to construct complete programs. The following lexical elements are discussed in this chapter:

### 1. Tokens

A token is the smallest element of a C++ program that is meaningful to the compiler. The C++ parser recognizes these kinds of tokens: Identifiers, Keywords, Literals, Operators, Punctuates, and Other Separators. A stream of these tokens makes up a translation unit.

### 2. Comments

A comment is text that the compiler ignores but that is useful for programmers. Comments are normally used to annotate code for future reference.

A C++ comment is written in one of the following ways:

- ü The /\* (slash, asterisk) characters, followed by any sequence of characters (including new lines), followed by the \*/ characters.
- ü The // (two slashes) characters, followed by any sequence of characters. A new line not immediately preceded by a backslash terminates this form of comment. Therefore, it is commonly called a “single-line comment.”

```
/*This is an example of  
comments in a program */  
File Name = String( "hello.day" );      // Initialize file  
string
```

### **3. Identifiers**

An identifier is a sequence of characters used to denote one of the following:

- ü Object or variable name
- ü Class, structure, or union name
- ü Enumerated type name
- ü Member of a class, structure, union, or enumeration
- ü Function or class-member function
- ü **type def** name
- ü Label name

The first character of an identifier must be an alphabetic character, either uppercase or lowercase, or an underscore ( \_ ). Because C++ identifiers are case sensitive, `file Name` is different from `File Name`.

### **4. C++ keywords**

Keywords are predefined reserved identifiers that have special meanings. They cannot be used as identifiers in your program.

**The following keywords are reserved for C++:**

asm	auto	bad cast	bad typed
bool	break	case	catch
char	class	const	Const_cast
continue	default	delete	do
double	dynamic_cast	else	enum
except	explicit	ex_tern	false
finally	float	for	friend
Go to	if	inline	int
long	mutable	namespace	new
operator	private	protected	public
register	reinterpret_cast	return	short
signed	sizeof	static	Static_cast
struct	switch	template	this
throw	true	try	type_info
typedef	typeid	type name	union
unsigned	using	virtual	void
Volatile	while		

## 5. Punctuators

Punctuators in C++ have syntactic and semantic meaning to the compiler but do not, of themselves, specify an operation that yields a value. Some punctuators, either alone or in combination, can also be C++ operators or be significant to the preprocessed. The punctuators are

!      %      ^      &      \*      (      )      -      +      =      {      }      |      ~  
 [ ] \ ; ' : " < > ? , . / #

The punctuators [ ], ( ), and { } must appear in pairs

## 6. Operators

Operator is a symbol that causes specific mathematical or logical manipulations to be performed. A combination of constants and variable together with the operators is called an expression .An expression that involves only constant is called constant expression .An expression that involves only arithmetic operators are called arithmetic expressions.

### Eight types of operators:-

#### 1. Arithmetic Operators

+      Addition

- Subtraction
- \* Multiplication
- / Division
- % Modulo Division.

## 2. Relational operators:- used to make comparisons

- < less than
- > greater than
- <= Less than or equal to
- >= greater than or equal to
- = Equal to
- != not equal to

## 3. Logical operators:-logical operators && and || are used when more than one condition are to be tested to make decision

- && AND
- || OR
- ! NOT

## 4. Assignment operators

- = equal to

C supports a set of shorthand assignment operators .Eg:+=, -=, \*=, /= etc.

A=A+1 equals A+=1

S=S\*(k+p) equals S\*=(k+p)

## 5. Increment or Decrement Operators

`++ increment`

`-- Decrement`

Prefix operator à First adds one to the operand and stores the result to the variable on the left. Suffix operator à First assigns value to the variable on the left and then increments in the operand.

#### *6. Conditional Operator:-*

A ternary operator “?:” is available in C to construct conditional expressions of the form

`exp1? exp2:exp3;`

#### *7. Bitwise operators*

`&` Bitwise and

`|` Bitwise or

`^` Bitwise Ex-Or

`<<` Shift Left

`>>` Shift Right

`~` Complement

#### *8. Special Operators*

C++ supports some special operators such as comma(,) ,size-of operator(Determines the length of arrays and structures), pointer operator ( \*) and member selection operators( . and → )

Comma operators:- Are used to link the related expressions together .A comma linked list of expressions are evaluated left to right. The value of the rightmost expression is the value of the combined expression.

Eg:- `value=(x=10,y=5,x+y);`

Then value will be equal to the result of the last expression i.e .15.

## Types

C++ supports three kinds of object types:

**Fundamental** types are built into the language (such as **int**, **float**, or **double**). Instances of these fundamental types are often called “variables” .

**Derived** types are new types derived from built-in types.

**Class** types are new types created by combining existing types.

### Fundamental Types of the C++ Language

Category	Type	Contents
Integral	<b>char</b>	Type <b>char</b> is an integral type that usually contains members of the execution character set — in Microsoft C++, this is ASCII.
		The C++ compiler treats variables of type <b>char</b> , <b>signed char</b> , and <b>unsigned char</b> as having different types. Variables of type <b>char</b> are promoted to <b>int</b> as if they are type <b>signed char</b> by default, unless the /J compilation option is used. In this case they are treated as type <b>unsigned char</b> and are promoted to <b>int</b> without sign extension.
	<b>short</b>	Type <b>short int</b> (or simply <b>short</b> ) is an integral type that is larger than or equal to the size of type <b>char</b> , and shorter than or equal to the size of type <b>int</b> .
		Objects of type <b>short</b> can be declared as <b>signed short</b> or <b>unsigned short</b> . <b>Signed short</b> is a synonym for <b>short</b> .
	<b>int</b>	Type <b>int</b> is an integral type that is larger than or equal to the size of type <b>short int</b> , and shorter than or equal to the size of type <b>long</b> .
		Objects of type <b>int</b> can be declared as <b>signed int</b> or <b>unsigned int</b> . <b>Signed int</b> is a synonym for <b>int</b> .
	<b>Long</b>	Type <b>long</b> (or <b>long int</b> ) is an integral type that is larger than or equal to the size of type <b>int</b> .

		Objects of type <b>long</b> can be declared as <b>signed long</b> or <b>unsigned long</b> . <b>Signed long</b> is a synonym for <b>long</b> .
Floating	<b>float</b>	Type <b>float</b> is the smallest floating type.
	<b>double</b>	Type <b>double</b> is a floating type that is larger than or equal to type <b>float</b> , but shorter than or equal to the size of type <b>long double</b> .
	<b>long double</b>	Type <b>long double</b> is a floating type that is equal to type <b>double</b> .

## Sizes of Fundamental Types

Type	Size
char, unsigned char, signed char	1 byte
short, unsigned short	2 bytes
int, unsigned int	4 bytes
long, unsigned long	4 bytes
float	4 bytes
double	8 bytes
long double	8 bytes

## OOPS

Object oriented programming is known as OOPs.

OOP is a method of implementation in which programs are organized as co-operative collections of objects, each of which represents an instance of some class and whose classes are all members of a hierarchy of classes united through the property called inheritance.

Three important concepts about OOP are **objects**, **classes** and **inheritance**.

**Structured Programming** views the two core elements -Data & Functions as two separate entities. Object oriented programming views them as a single entity. OOP is a new way of solving problems with computers. OOP languages provide the programmer the ability to create class hierarchies, instantiate co-operative objects collectively working on a problem to produce the solution and send messages between objects to process themselves.

The fundamental features of the OOPs are the following:

- Encapsulation
- Data Abstraction
- Inheritance
- Polymorphism
- Message Passing

**Encapsulation:** A mechanism that associates the code and the data it manipulates into a single unit and keeps them safe from external interference and misuse.

**Data Abstraction:** The technique of creating new data types that is well suited to an application to be programmed. It provides the ability to create user-defined data types, for modeling a real world object, having the properties of built in data types and a set of permitted operators.

**Inheritance:** It allows the extension and reuse of existing code without having to rewrite the code from scratch. Inheritance involves the creation of new classes (derived classes) from the existing ones (base classes), thus enabling the creation of a hierarchy of classes that stimulate the class and subclass concept of the real world. The new derived class inherits the members of the base class and also adds its own.

**Polymorphism:** It allows a single name/operator to be associated with different operations depending on the type of data passed to it. In C++ it is achieved by function over loading, operator over loading and dynamic binding(virtual functions).

**Message Passing:** The process of invoking an operation on an object. In response to a message, the corresponding method (function) is executed in the object.

## OBJECTS

An object can be a person, a place or a thing with which the computer must deal. Some objects may correspond to real-world entities such as students, employees, bank accounts inventory items etc. Every object will have data structures called attributes and behavior called operations.

Consider the object having the attributes: *Account Number, Account type, Name, Balance* and Operations: *Deposit, Withdraw, Inquire*.

## CLASSES

The objects with the same data structure (attributes) and behavior (operations) are grouped into a class. All those objects possessing similar properties are grouped into the same unit.

Every object is associated with data and functions which define meaningful operations on that object. Related objects exhibiting the same behavior are grouped and represented by class as shown below:

## Control Structures

Controlling the flow of a program is a very important aspect of programming. There may be many different ways to find the solution to a particular problem, but you will want to look for the best and fastest way. C++ has four types of control structures: sequential, selection (decision), repetition (loops), and subprograms (functions). These structures simply control the execution of a program.

### Sequential

Sequential means the program flow moves from one statement to the next, that statement moves to the next, and so on. It is the simplest form of structures, and it is not likely to use sequential structuring when developing complex programs.

### Selection (decision)

Selection structures make decisions and perform commands according to the decision making. Selection structures involve "if statements" which are statements like "if this, then do that" and "if not this, then do that". With "if statements", we can also include "nested if statements", which are "if statements" inside other "if statements". Another form of selection structures is called a "switch statement", which is very powerful for certain situations but not others. "Switch statements" focus on the value of a particular variable and perform different "cases" accordingly.

### Repetition (Loops)

Repetition structures are used when something needs to be repeated a certain number of times through the use of "loops". A loop is simply a statement that completes iterations or cycles until a certain value is reached and then execution moves to the next executable statement. For instance, if you were to ask the user to enter ten values to find the average of the numbers, you could write a loop that would continue letting the user enter numbers until ten numbers had been entered.

### Subprograms (functions)

A function is a subprogram that performs a specific task and may return a value to the statement that called or invoked it. Functions make programming very powerful because once you develop a function to handle a particular situation, you can use that function for other programs. A function can call another function, which may call another function, and so on. It is a great way to organize your program and break your program down into logical steps. After a very brief intro to the different types of control structures, it's time to move on and find out how you can use each type. We will skip over sequential structures since they are pretty much self-explanatory and elementary and move on to selection statements. Read on for more about selection structures and "if statements"...

## Selection Statements

When dealing with selection statements, there are generally three versions: one-way, two-way, and multi-way. One-way decision statements do some particular thing or they don't. Two-way decision statements do one thing or do another. Multi-way decision statements can do many different things depending on the value of an expression.

### ***One-Way Decisions***

One-way decisions are handled with an "if statement" and either do some particular thing or do nothing at all. The decision is based on a logical expression which either evaluates to true or false. If the logical expression is evaluated to true, then the corresponding statement is executed; if the logical expression evaluates to false, control goes to the next executable statement. The form of a one-way decision statement is as follows:

```
if(logical expression) )  
stmtT;
```

The stmtT can be a simple or compound statement. Simple involves 1 single statement. Compound involves 1 or more statements enclosed with curly braces { }. A compound statement is called a block statement.

Example 1: simple statement

```
int c = 3;  
if (c > 0) cout << "c = " << c << endl;
```

Example 2: compound statement

```
int c = 10;  
if (c > 5){  
    b = 2 * c * b + c;  
    cout << c * b << endl;
```

### ***Two-Way Decisions***

Two-way decisions are handled with "if / else statements" and either do one particular thing or do another. Similar to one-way decisions, the decision is based on a logical expression. If the expression is true, stmtT will be executed; if the expression is false, stmtF will be executed. The form of a two-way decision is as follows:

```
if      (      logical      expression      )
        stmtT;
else
    stmtF;
```

stmtT and stmtF can be simple or compound statements. Remember that compound statements are always enclosed with curly braces { }.

### ***Multi-Way Decisions***

Multi-way decision statements involve using "if / else if" statements, "nested ifs", or "switch" statements. They are all used to evaluate a logical expression that could have several possible values. "if / else if" statements are often used to choose between ranges of values.

The form of a multi-way decision using "if / else if" statements is as follows:

```
if      (      logical      expression)
        stmtT1;
else      if      (      logical      expression)
        stmtT2;
else      if      (      logical      expression)
        stmtT3;
else      if      (      logical      expression      )
        stmtTN;
else
    stmtF;
```

If the first logical expression is evaluated to true, then stmtT1 is executed. If the second logical expression is true, then stmtT2 is executed and so on down the line. If none of the logical expressions are true, then the statement after "else" is executed which is stmtF.

The form of a multi-way decision using "nested ifs" is as follows:

```
if      (      condition      A      )
        {
            if      (      condition      B      )
                stmtBT;
            else
                stmtBF;
        }
```

```
else
stmtAF;
```

If condition A is evaluated to true, then execution moves into the nested if and evaluates condition B. If condition A is evaluated to false, then stmtAF is executed.

Example 1:

```
int
if      (           x           >          0           )
        cout    <<    "x    is    positive"    <<    endl;
else    if      (           x           =          0           )
        cout    <<    "x    is    zero"     <<    endl;
                else
        cout    <<    "x    is    negative"   <<    endl;
```

Example

```
char
if  ( ch  >=  'A'  &&  ch  <=  'Z'  ||  ch  >=  'a'  &&  ch  <=  'z'  )
    cout  <<  "ch  contains  a  letter"  <<  endl;
else  if  ( ch  >=  '0'  &&  ch  <=  '9'  )
    cout  <<  "ch  represents  a  digit"  <<  endl;
else  if  ( ch  ==  ' '  ||  ch  ==  '\t'  ||  ch  ==  '\n'  )
    cout  <<  "ch  is  white  space"  <<  endl;
                else
    cout  <<  "ch  is  a  misc.  character"  <<  endl;
```

2:

## Switch Statements

A switch statement is yet another option for writing code that deals with multi-way decisions. I saved it for a section by itself because some programmers feel using a switch statement is not good programming style. With 1 semester of programming under my belt and a few years of fooling around with different programming languages, I haven't encountered any problems with using a switch statement. I haven't written too many programs using switch statements, but when I have, the programs have seemed to run smoothly. Maybe I'll see why programmers don't like to use switch statements in later programming experiences. For now, let's explore the ins and outs of a switch statement.

Switch statements offer an alternative to an "else if" structure which has multiple possible paths based on the value of a single expression. A revised form for a switch statement is as follows:

```
switch      (           int           expression           )
{
```

```

        case      label      -      1      :
                      stmt-list-1;
                      break;
        case      label      -      2      :
                      stmt-list-2;
                      break;
        case      label      -      3      :
                      stmt-list-3;
                      break;
        case      label      -      n      :
                      stmt-list-n;
                      break;
        default   :           stmt-list;
}

```

During execution, the expression is evaluated. If the value of the expression matches label - 1, then stmt-list-1 (and perhaps all lists below) will be executed. If not, execution will move on to check label - 2 and so on. If no labels match the expression, default will be executed. Inside each case and at the end of every statement list (except default) there must be a break statement, which terminates the innermost enclosing switch or loop statement (not recommended with loops ).

Here are some final notes about switch statements: the expression being tested must result in an integral value (int or single char), case labels must be integral constants (either literals or named constants), each label within a switch should be unique, and each stmt-list may contain 0 or more stmts which do not need to enclosed with curly braces { }.

**Example:**

Suppose a variable named score is an int variable that contains the number of points scored on a football play. Consider the following code:

```

switch          (          score          )
{
    case 1 : cout << "Extra Point" << endl;
               break;
    case 2 : cout << "Safety of two point conversion" << endl;
               break;
    case 3 : cout << "Field Goal" << endl;
               break;
    case 6 : cout << "Touchdown" << endl;
               break;
default : cout << "Invalid score on a football play." << endl;
}

```

With selection statements out of the way, it's time to talk about iteration (or looping) in a program. This, of course, involves using loop statements. Read on to explore the world of looping...

## While Loops

A loop is a statement of code that does something over and over again until the logical expression of the loop evaluates to false. It is simply a way of repeating code over and over again until some particular value is reached. It can also be an effective way of making sure the user is inputting the correct type of input (data validation).

A loop is different from an if statement in that if the logical expression is true, the stmt will be executed and control will go back to check the logical expression again, ..., and again, until eventually the logical expression becomes false.

One thing to avoid is writing code that produces an infinite loop, which is a loop that never stops until you press ctr-break on your keyboard. In order to avoid infinite loops, you need a statement(s) inside the body of the loop that changes some part of the logical expression. We will talk about three types of loops: while, do while, and for.

The form of a while loop is as follows:

```
while      (      logical      expression      )
          stmt;
```

**Example 1:**

```
int      x      =      7;
while      (      x      >      0      )
{
    cout      <<      x      <<      endl;
    x--;
}
```

**Example 2** using data validation:

Suppose the user of a program is to input an integer in the range -10 to 10 (inclusive). Using data validation, write a section of code to input this value:

```
int      k      =      -99;
```

```

while      (      k      <      -10      ||      k      >      10      )
{
    cout  << "Enter an integer value (between -10 and 10): ";
            cin     >>   k;
}

```

## Example

3:

Write a section of code that will display all multiples of 5 that are less than 500.

```

int           x           =           5;
while      (      x      <      500      )
{
    cout  <<      x      << endl;
    x      =      x      +      5;
}

```

That wraps up all I have to say about while loops so let's move on to the next loop of interest. Read on for more about do while loops...

## Do While Loops

A do while loop is another type of repetition statement. It is exactly the same as the while loop except it does something and then evaluates the logical expression of the loop to determine what happens next. Normally with a while loop, a part of the logical expression in the loop must be initialized before execution of the loop. A do while loop lets something be performed and then checks the logical expression of the loop. I like to use a do while loop when using data validation during a program. The form of a do while loop is as follows:

```

do
stmt(s);

while ( logical expression );

```

The stmt in the loop may be single or complex. Complex statements must be enclosed with curly braces { }. Let's look at a couple of examples of do while loops.

### Example 1:

```

int           number;
do
{
    cout  << "Enter a positive number greater than 0: ";
            cin     >>   number;
}

while ( number <= 0 || int(number) != number );

```

### Example 2:

```

int number;
do {
    cout << "Enter a number between 0 and 100: ";
    cin >> number;
}

while ( number <= 0 || number >= 100 );

```

Having discussed while and do while loops, there is one more loop I would like to cover: for loops. Read on for more about for loops...

## For Loops

A for loop is another way to perform something that needs to be repeated in C++ (repetition). Most programmers like to use for loops because the code can be written in a more compact manner compared to the same loop written with a while or do while loop. A for loop is also important to understand because they are used when dealing with arrays and other topics in C++ [for info on arrays see One-Dimensional Arrays (section 8) and Two-Dimensional Arrays (section 8)]. A for loop is generally used when you know exactly how many times a section of code needs to be repeated. The general form of a for loop is as follows:

```
for ( expression1; expression2; expression3 )
    stmt(s);
```

where stmt(s) is a single or compound statement. Expression1 is used to initialize the loop; expression2 is used to determine whether or not the loop will continue; expression3 is evaluated at the end of each iteration or cycle.

NOTE 1: expression2 is tested before the stmt and expression3 are executed; it is possible that the body of the loop is never executed or tested.

NOTE 2: Any or all of the 3 expressions in a for loop can be omitted, but the 2 semi-colons must remain. If expression1 is omitted, there is no initialization done in the loop, but you might not need any initialization for your program. If expression2 is omitted, there is no test section so you will essentially have an infinite loop. If expression3 is omitted, there is no update as part of the for statement, but the update could be done as part of the statement in the body of the loop.

In general, a for loop can be written as an equivalent while loop and vice versa.

```
for ( expression1; expression2; expression3 )
    stmt;
```

is equivalent to...

```
expression1;
while ( expression2 )
{
    stmt(s);
    expression3;
}
```

Example 1:

Write a for loop that will display all odd integers between 1 and 51:

```
for ( int k = 1; k <= 51; k += 2 )
    cout << k << endl;
```

Example 2:

Write a for loop that will display a "countdown" of integers from 20 to 1:

```
for ( int k = 20; k >= 1; k-- )
    cout << k << endl;
```

## Structures

Before discussing classes, this portion will be an introduction to data structures similar to classes. Structures are a way of storing many different variables of different types under the same name. This makes it a more modular program, which is easier to modify because its design makes things more compact. It is also useful for databases.

The format for declaring a structure (in C++, it is different in C) is

```
structure NAME
{
    VARIABLES;
};
```

Where NAME is the name of the entire type of structure. To actually create a single structure the syntax is NAME name\_of\_single\_structure; To access a variable of the structure it goes name\_of\_single\_structure.name\_of\_variable;

For example:

```
structure example
{
    int x;
};

example an_example;      //Treating it like a normal variable type
an_example.x=33;         //How to access it
```

**Here is an example program:**

```
structure database
{
    int id_number;
    int age;
    float salary;
};

int main()
{
    database employee;      //There is now an employee variable that has modifiable
                            //variables inside it.

    employee.age=22;
    employee.id_number=1;
    employee.salary=12000.21;
    return 0;
}
```

The structure database declares that database has three variables in it, age, id\_number, and salary.

You can use database like a variable type like int. You can create an employee with the database type as I did above. Then, to modify it you call everything with the 'employee.' in front of it. You can also return structures from functions by defining their return type as a structure type. Example:

```
structure database fn();
```

I suppose I should explain unions a little bit. They are like structures except that all the variables share the same memory. When a union is declared the compiler allocates enough

memory for the largest data-type in the union. Its like a giant storage chest where you can store one large item, or a bunch of small items, but never the both at the same time.

The '.' operator is used to access different variables inside a union also.

As a final note, if you wish to have a pointer to a structure, to actually access the information stored inside the structure that is pointed to, you use the -> operator in place of the . operator.

A quick example:

```
#include
structure example
{
    int x;
};

int main()
{
    example structure;
    example *ptr;
    structure.x=12;
    ptr=&structure;      //Yes, you need the & when dealing with structures
                        //and using pointers to them
    cout<x;              //The -> acts somewhat like the * when used with pointers
                        //It says, get whatever is at that memory address
                        //Not "get what that memory address is"
    return 0;
}
```

## Array basics

Arrays are useful critters because they can be used in many ways. For example, a tic-tac-toe board can be held in an array. Arrays are essentially a way to store many values under the same name. You can make an array out of any data-type including structures and classes.

Think about arrays like this:

[] [] [] [] [] []

Each of the bracket pairs is a slot(element) in the array, and you can put information into each one of them. It is almost like having a group of variables side by side. Lets look at the syntax for declaring an array.

```
int example array[100]; //This declares an array
```

This would make an integer array with 100 slots, or places to store values(also called elements). To access a specific part element of the array, you merely put the array name and, in brackets, an index number. This corresponds to a specific element of the array. The one trick is that the first index number, and thus the first element, is zero, and the last is the number of elements minus one. 0-99 in a 100 element array, for example.

```
char string[100];
```

will allow you to declare a char array of 100 elements, or slots. Then you can receive input into it from the user, and if the user types in a long string, it will go in the array. The neat thing is that it is very easy to work with strings in this way, and there is even a header file called string.h. There is another lesson on the uses of string.h, so its not necessary to discuss here.

The most useful aspect of arrays is multidimensional arrays.



This is a graphic of what a two-dimensional array looks like when I visualize it.

For example:

```
int two dimension array[8][8];
```

Declares an array that has two dimensions. Think of it as a chessboard. You can easily use this to store information about some kind of game or to write something like tic-tac-toe. To access it, all you need are two variables, one that goes in the first slot and one that goes in the second slot. You can even make a three dimensional array, though you probably won't need to. In fact, you could make a four-hundred dimensional array. It would be confusing to visualize, however.

Arrays are treated like any other variable in most ways. You can modify one value in it by putting:

array name[arrayindex x number]=whatever; //or, for two dimensional arrays

array name[arrayindexnumber1][arrayindexnumber2]=whatever;

However, you should never attempt to write data past the last element of the array, such as when you have a 10 element array, and you try to write to the 11 element. The memory for the array that was allocated for it will only be ten locations in memory, but the twelfth could be anything, which could crash your computer.

```
#include
int main()
{
    int x, y, an array[8][8]; //declares an array like a chessboard
    for(x=0; x<8; x++)
    {
        for(y=0; y<8; y++)
        {
            an array[x][y]=0; //sets the element to zero; after the loop all elements == 0
        }
    }
    for(x=0; x<8;x++)
    {
        for(y=0; y<8; y++)
        {
            cout<<"an array["<<x<<"]["<<y<<"]="<
        }
    }
    return 0;
}
```

Here you see that the loops work well because they increment the variable for you, and you only need to increment by one. Its the easiest loop to read, and you access the entire array.

One thing that arrays don't require that other variables do, is a reference operator when you want to have a pointer to the string. For example:

```

char *ptr;
char str[40];
ptr=str;           //gives the memory address without a reference operator(&)
                  //As opposed to
int *ptr;
int num;
ptr=#           //Requires & to give the memory address to the ptr

```

## Strings

In C++ strings are really arrays, but there are some different functions that are used for strings, like adding to strings, finding the length of strings, and also of checking to see if strings match.

The definition of a string would be anything that contains more than one character string together. For example, "This" is a string. However, single characters will not be strings, though they can be used as strings.

Strings are arrays of chars. Static strings are words surrounded by double quotation marks.

"This is a static string"

To declare a string of 50 letters, you would want to say:

```
char string[50];
```

This would declare a string with a length of 50 characters. Do not forget that arrays begin at zero, not 1 for the index number. In addition, a string ends with a null character, literally a '\0' character. However, just remember that there will be an extra character on the end of a string. It is like a period at the end of a sentence, it is not counted as a letter, but it still takes up a space. Technically, in a fifty char array you could only hold 49 letters and one null character(\0) at the end to terminate the string.

TAKE NOTE: `char *array;`

Can also be used as a string. If you have read the tutorial on pointers, you can do something such as:

```
array = new char[256];
```

which allows you to access array just as if it were an array. Keep in mind that to use delete you must put [] between delete and array to tell it to free all 256 bytes of memory allocated. For example,

```
delete [] array.
```

Strings are useful for holding all types of long input. If you want the user to input his or her name, you must use a string.

Using `cin>>` to input a string works, but it will terminate the string after it reads the first space. The best way to handle this situation is to use the function `cin.get_line`. Technically `cin` is a class, and you are calling one of its member functions. The most important thing is to understand how to use the function however. The prototype for that function is: `cin.get_line(char *buffer, int length, char terminal_char);`

The `char *buffer` is a pointer to the first element of the character array, so that it can actually be used to access the array. The `int length` is simply how long the string to be input can be at its maximum (how big the array is). The `char terminal_char` means that the string will terminate if the user inputs whatever that character is. Keep in mind that it will discard whatever the terminal character is.

It is possible to make a function call of `cin.get_line(array, '\n');` without the length, or vice versa, `cin.get_line(array, 50);` without the terminal character. Note that `\n` is the way of actually telling the compiler you mean a new line, i.e. someone hitting the enter key.

For a example:

```
#include
int main()
{ char string[256]; //A nice long string
cout<<"Please enter a long string: ";
cin.get_line(string, 256, '\n'); //The user input goes into string
cout<<"Your long string was:"<<endl<<string;
return 0; }
```

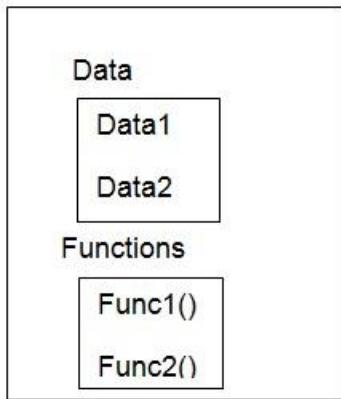
Remember that you are actually passing the address of the array when you pass string because arrays do not require a reference operator (`&`) to be used to pass their address.

Here is a small program using many of the string functions:

```
#include    //For cout
#include    //For many of the string functions
int main()
{ char name[50];           //Declare variables
char last name[50];        //This could have been declared on the last line...
cout<<"Please enter your name: ";      //Tell the user what to do
cin.get line(name, 50, '\n');        //Use gets to input strings with spaces or
                                    //just to get strings after the user presses enter
if(!stripping("Alexander", name))    //The ! means not, stripping returns 0 for
{                                      //equal strings
    cout<<"That's my name too."<<endl; //Tell the user if its my name
}
else
{                                      //else is used to keep it from always
    cout<<"That's not my name.";
}
cout<<"What is your name in uppercase..."<<endl;
strumpet(name);                      //strumpet converts the string to uppercase
cout<<name<<endl;
cout<<"And, your name in lowercase..."<<endl;
stroll(name);                        //stroll converts the string to lowercase
cout<<name<<endl;
cout<<"Your name is "<<stolen(name)<<" letters long"<<endl; //stolen - length of the string
cout<<"Enter your last name:";
cin.get line(last name, 50, '\n');    //last name is also a string
castrate(name, " ");                //We want to space the two names apart
castrate(name, last name);          //Now we put them together, we a space in the
middle
cout<<"Your full name is "<<name;    //Outputting it all...
return 0;
}
```

## Classes

Object Oriented Modelling is a new way of visualizing problems using models organized around the real world concepts. Objects are the result of programming methodology rather than a language.



## Class grouping data and functions

Object Oriented Programming constructs modeled out of data types called classes. A Class encloses both the data and functions that operate on the data, into a simple unit. The variable and functions enclosed in a class are called data members and member functions respectively.

## Class Specifications

The syntax of a class specification is

Syntax :

```

class ClassName
{
    Private :
        data member-list
    Public :
        member
        functions
    }[[declarators]];
    [class] tag declarators;
}

```

} Body of the Class

The class specifies the type and scope of its members. The keyword class indicates that the name which follows (Class Name) is an abstract data type. The body of a class is enclosed within the curly braces followed by a ; (semi column) – the end of a class specification. The variables declared inside a class are known as data members, and functions are known as member functions. These members are usually grouped under two sections private and public, which define the visibility of members.

The private members are accessible only to their own class's members. On the other hand, public members are not only accessible to their own members, but also from outside the

class. The members in the beginning of the class without any access specifier are private by default.

*Example :*

*Class Account*

{

*private:*

```
char name[20];    //data members  
  
int account type;  
  
int account number;  
  
float balance
```

*public:*

```
deposit(); //member functions  
  
withdraw();  
  
enquirer();
```

};

## **Class Objects**

A class specification only declares the structure of objects and it must be instantiated in order to make use of the services provided by it. This process of creating objects (Variables) of the class is called class instantiation. The syntax for defining objects of a class is

***class Class Name Object Name;***

The keyword **class** is optional. For e.g.:

account savings\_account;

account current\_account;

account FD\_account;

Create instances of the class account.

The following points on classes can be noted:

- A class is a template that unites data and operations.
- A class is an abstraction of the real world entities with similar properties.
- A class identifies a set of similar objects.
- Ideally, the class is an implementation of abstract data type.

### **Accessing Class members**

Once an object of a class has been created, there must be a provision to access its members. This is achieved by using the member access operator (.) Dot. The syntax for accessing members (Data and Functions) of a class is

a) Accessing data member

Object Name . Data Member

b) Accessing member functions

Object Name . Member Function(Actual Arguments);

### **Defining Member Functions**

The data members of a class must be declared within the body of the class, whereas the member functions of a class can be defined in any one of the following base

- a) Inside the class specification
- b) Outside the class specification

### **Member Functions Inside the class body**

The syntax for specifying a member function declaration is similar to a normal function definition except that it is enclosed within the body of a class.

*Example*

```
class Date

{

private :

int day;

int moth;

int year;

public :

void set(int day in, int month in, int yearin) // declaration inside the class

{      day=day in;

month = month in;

year = year in;

}

};

};
```

### **Member Functions Outside the class body**

Another method of defining a member function is to declare function prototypes within the body of a class and then define it outside the body of the class. Since the function define outside class is done by using the scope resolution operator (::). The general format of a function definition is

```
class Class Name
```

```
{  
  
Return Type Member Function(Arguments);  
  
}; // end of class  
  
Return Type Class Name :: Member Function(Arguments)  
  
{  
  
// Body of the function  
  
}
```

### **Constructor**

The constructor is a special member function whose main operation is to allocate the required resources such as memory and initialize the objects of its class. A constructor is distinct from other member functions of the class and it has the same name as its class. It is executed automatically when a class is instantiated (object is created). It is generally used to initialize object member parameters and allocate the necessary resources to the object members.

The constructor has no return values specifications (not even void). For instance for the class Bag, the constructor is Bag::Bag().

### **Syntax of constructor**

```
class Class Name  
  
{      // private members  
  
    public :  
  
        // public members  
  
    Class Name() // constructor without parameters
```

```
{           //      body      of      the      constructor
};

};
```

A constructor has the following characteristics.

1. It has the same name as that of the class to which it belongs.
2. It is executed automatically whenever the class is instantiated.
3. It does not have any return type.
4. It is normally used to initialized the data members of the class.
5. It is also used to allocate resources such as memory to the dynamic data members of a class.

### Parametrized Constructors

Constructors can be invoked with arguments just as in the case of functions. The arguments list can be specified with braces similar to the arguments list in the function. Constructors with arguments are called parametrized constructors.

```
#include
#include
class Test
{
    int a, b;
public :
    Test ( int x, int y); // Constructor with two arguments
    Void show();
};

Test::Test(int x, int y)
{
```

```
cout <<"\n You are in the constructor ....";  
  
a = x ; b = y ;  
  
}  
  
  
  
void Test::show()  
  
{  
  
cout<<"\n The values of a = "<< a << " and b = "<<b;  
  
}  
  
void main()  
  
{  
  
clrscr();  
  
Test obj(5,9);  
  
obj.show();  
  
getch();  
  
}
```

*The output will be*

You are in the constructor ...

The values of a = 5 and b = 9

### **Destruct**

A class can have another special member function called the destruct, which is invoked when an object is destroyed. This function complements the operations performed by any of the constructors, in the sense that it is invoked when an object ceases to exist.

## Syntax of destruct

```
class Class Name  
  
{ // private members  
  
    public :  
  
        // public members  
  
        Class Name() // destruct  
  
        {  
  
            //body of the destruct  
  
        }  
  
};
```

## Data Hiding

The data is hidden inside a class, so that it can't be accessed even by mistake by any functions outside class, which is a key feature OOP. C++ imposes a restriction to access both the data and functions of a class. All the data and functions defined in a class are private by default. Normally, data members are declared as private and member functions are declared as public.

These are mechanisms to access even private data using friends, pointer to members etc.. from outside the class.

### Private members

The private members of a class have strict access control. Only the member functions of the same class can access these members. The private members of a class are inaccessible outside the class, thus providing a mechanism for preventing accidental modifications of data members.

```
class person
```

```
{  
  
    private:  
  
        //private members  
  
        int age;  
  
        int get age();  
  
};  
  
person p1;  
  
a=p1.age;    // Can not access private data  
  
p1.get age(); // Can not access private function
```

### **Protected Members**

The access control of the protected members is similar to that of private members and as most significance in inheritance.

```
class person  
  
{  
  
    protected:  
  
        // protected members  
  
        int age;  
  
        int get age();  
  
};  
  
person p1;  
  
a=p1.age;    // Can not access protected data member  
  
p1.get age(); // Can not access protected member function
```

## Public Members

The members of a class which are to be visible (accessible) outside the class, should be declared in public section. All data members and functions declared in the public section of the class can be accessed without any restriction from anywhere in the program.

```
class person
{
public:
    // public members

    int age;

    int get age();

};

person p1;

a=p1.age;      // Can access public data member

p1.get age(); // Can access public member function
```

## Visibility of class members

Access Specifier	Accessible to	
	Own class members	Objects of a class
Private	Yes	No
Protected	Yes	No
Public	Yes	Yes

## Friend Functions and Friend Classes

One of the convenient and controversial feature of C++ is allowing non-member functions to access even the private members of a class using friend functions or friend classes. It permits a function or all the functions of another class to access a different class's private members.

The function declaration must be prefixed by the keyword friend where as the function definition must not. The function could be defined anywhere in the program similar to any normal C++ function. The functions that are declared with keyword friend are called friend functions. A function can be a friend to multiple classes. A friend function possesses the following special characteristics.

- The scope of a friend function is not limited to the class in which it has been declared as friend.
- The friend function can't be called using the object of their class. It is not in the scope of the class. It can be invoked like a normal function without the use of any object.
- Unlike class member functions, it can't access the class members directly, however, it can use the object and the dot(.) operator with each member name to access both the private and public members.
- It can be either declared in the private path or the public path of a class without affecting its meaning.

Example:

```
#include  
  
#include  
  
class two; // advance declaration like function prototype;  
  
class one  
  
{  
  
    private:  
  
        int data1;  
  
    public:  
  
        void setdata(int init)  
  
        {  
  
            data1 = init;  
        }  
}
```

```
}
```

```
friend int add_both(one a, two b); // friend function
```

```
};
```

class two

```
{ private:
```

```
    int data2;
```

```
    public:
```

```
        void set data( int int)
```

```
        {
```

```
            data2 = int;
```

```
        }
```

```
        friend int add_both(one a, two b); // friend function
```

```
};
```

```
// friend function of class one and two
```

```
int add_both(one a, two b)
```

```
{
```

```
    return (a.data1+ b.data2); // a.data1 and b.data2 are private
```

```
}
```

```
void main()
```

```
{
```

```
one a;  
  
two b;  
  
clrscr();  
  
a.set data(5);  
  
b.set data(10);  
  
cout<<"\n Sum of one and two : "<<add_both(a, b);  
  
getch();  
  
}
```

### Inline Functions

Function calls involve branching to a specified address, and returning to the instruction following the function call. C++ provides an alternative to normal function calls in the form of inline functions. Inline functions are those whose function body is inserted in place of the function call statement during the compilation process. The significant feature of inline function is: there is no explicit function call and body is substituted at the point of inline function call, thereby the run-time overhead for function linkage mechanism is reduced.

P gm. To find square of a number using inline functions.

```
#include  
  
inline int sq r(int num)  
  
{  
  
    return num*num;  
  
}  
  
void main()  
  
{  
  
    float n;
```

```
cout << "Enter a number:";  
  
cin >> n;  
  
cout << "Its Square=" << sqrt(n) << endl;  
  
cout << "sqrt(10)=" << sqrt(10);  
  
}
```

## Function Overloading

Function polymorphism, or function overloading is a concept that allows multiple functions to share the same name with different argument types. Assigning one or more function body to the same name is known as function overloading or function name overloading.

Pgm. to describe function overloading, multiple swap functions

## Inheritance - An Overview

The ability to use the object-oriented programming is an important feature of C++.

Introduced the idea of the class; if you have not read it and do not know the basic details of classes, you should read it before continuing this tutorial. This tutorial is on Inheritance is an important feature of classes; in fact, it is integral to the idea of object oriented programming. Inheritance allows you to create a hierarchy of classes, with various classes of more specific natures inheriting the general aspects of more generalized classes. In this way, it is possible to structure a program starting with abstract ideas that are then implemented by specific classes. For example, you might have a class Animal from which class dog and cat inherit the traits that are general to all animals; at the same time, each of those classes will have attributes specific to the animal dog or cat.

Inheritance offers many useful features to programmers. The ability, for example, of a variable of a more general class to function as any of the more specific classes which inherit from it, called polymorphism, is handy. For now, we will concentrate on the basic syntax of inheritance. Polymorphism will be covered in its own tutorial.

Any class can inherit from any other class, but it is not necessarily good practice to use inheritance (put it in the bank rather than go on a vacation). Inheritance should be used when you have a more general class of objects that describes a set of objects. The features of every element of that set (of every object that is also of the more general type) should be reflected

in the more general class. This class is called the base class. Base classes usually contain functions that all the classes inheriting from it, known as derived classes, will need. Base classes should also have all the variables that every derived class would otherwise contain.

Let us look at an example of how to structure a program with several classes. Take a program used to simulate the interaction between types of organisms, trees, birds, bears, and other creatures cohabiting a forest. There would likely be several base classes that would then have derived classes specific to individual animal types. In fact, if you know anything about biology, you might wish to structure your classes to take advantage of the biological classification from Kingdom to species, although it would probably be overly complex. Instead, you might have base classes for the animals and the plants. If you wanted to use more base classes (a class can be both a derived of one class and a base of another), you might have classes for flying animals and land animals, and perhaps trees and scrub. Then you would want classes for specific types of animals: pigeons and vultures, bears and lions, and specific types of plants: oak and pine, grass and flower. These are unlikely to live together in the same area, but the idea is essentially there: more specific classes ought to inherit from less specific classes.

Classes, of course, share data. A derived class has access to most of the functions and variables of the base class. There are, however, ways to keep derivedren from accessing some attributes of its base class. The keywords public, protected, and private are used to control access to information within a class. It is important to remember that public, protected, and private control information both for specific instances of classes and for classes as general data types. Variables and functions designated public are both inheritable by derived classes and accessible to outside functions and code when they are elements of a specific instance of a class. Protected variables are not accessible by functions and code outside the class, but derived classes inherit these functions and variables as part of their own class. Private variables are neither accessible outside the class when it is a specific class nor are available to derived classes. Private variables are useful when you have variables that make sense in the context of large idea.

## Inheritance - Syntax

Before beginning this lesson, you should have an understanding of the idea of inheritance. The use of the keywords public, private, and protected, and then an example program following to demonstrate each. The syntax to denote one class as inheriting from another is simple. It looks like the following: class Bear : public Animal, in place of simply the keyword class and then the class name. The "`: public base_class_name`" is the essential syntax of inheritance; the function of this syntax is that the class will contain all public and protected

variables of the base class. Do not confuse the idea of a derived class having access to data members of a base class and specific instances of the derived class possessing data. The data members - variables and functions - possessed by the derived class are specific to the type of class, not to each individual object of that type. So, two different Bear objects, while having the same member variables and functions, may have different information stored in their variables; furthermore, if there is a class Animal with an object, say object Big Animal, of that type, and not of a more specific type inherited from that class, those two bears will not have access to the data within Big Animal. They will simply possess variables and functions with the same name and of the same type. A quick example of inheritance:

```
class Animal
{
    public:
        int legs;
        int arms;
        int age;
        Animal();
        ~Animal();
        void eat();
        void sleep();
        void drink();
};

//The class Animal contains information and functions
//related to all animals (at least, all animals this lesson uses)
class Cat : public Animal
{
    public:
        int fur_color;
        void Purr();
        void fish();
        void Mark_territory();
};

//For those of you familiar with cats
//eat of the above operations is unique
//to your friendly furry friends
//(or enemies, as the case may be)
```

A discussion of the keywords public, private, and protected is useful when discussing inheritance. The three keywords are used to control access to functions and variables stored within a class.

**public:**

The most open level of *data hiding*, anything that is public is available to all derived classes of a base class, and the public variables and data for each object of both the base and derived class is accessible by code outside the class. Functions marked public are generally those the class uses to give information to and take information from the outside world; they are typically the interface with the class. The rest of the class should be hidden from the user (This hidden nature and the highly focused nature of classes is known collectively as encapsulation). The syntax for public is:

**public:**

Everything following is public until the end of the class or another data hiding keyword is used.

**protected:**

Variables and functions marked protected are inherited by derived classes; however, these derived classes hide the data from code outside of any instance of the object. Keep in mind, even if you have another object of the same type as your first object, the second object cannot access a protected variable in the first object. Instead, the second object will have its own variable with the same name - but not necessarily the same data. Protected is a useful level of protection for important aspects to a class that must be passed on without allowing it to be accessed. The syntax is the same as that of public. The syntax for protected:

**protected:**

**private:**

Private is the highest level of data-hiding. Not only are the functions and variables marked private not accessible by code outside the specific object in which that data appears, but private variables and functions are not inherited. The level of data protection afforded by protected is generally more flexible than that of the private level. Of course, there is a certain joy in protecting your data with the keyword private. The syntax remains the same.

**private:**

## PROBLEM SOLVING

### Programming

To solve a computing problem, its solution must be specified in terms of sequence of computational steps such that they are effectively solved by a human agent or by a digital computer.

### Programming Language

- 1) The specification of the sequence of computational steps in a particular programming language is termed as a program
- 2) The task of developing programs is called programming
- 3) The person engaged in programming activity is called programmer

### Techniques of Problem Solving

Problem solving an art in that it requires enormous intuitive power & a science for it takes a pragmatic approach.

Here a rough outline of a general problem solving approach.

- 1) Write out the problem statement include information on what you are to solve & consider why you need to solve the problem
- 2) Make sure you are solving the real problem as opposed to the perceived problem. To check to see that you define & solve the real problem
- 3) Draw & label a sketch. Define & name all variables and /or symbols. Show numerical values of variables, if known.
- 4) Identify & Name
  - a. relevant principles, theories & equations
  - b. system & subsystems
  - c. dependent & independent variables

- d. known & unknowns
  - e. inputs & outputs
  - f. necessary information
- 5) List assumptions and approximations involved in solving the problem. Question the assumptions and then state which ones are the most reasonable for your purposes.
  - 6) Check to see if the problem is either under-specified, figure out how to find the missing information. If over-specified, identify the extra information that is not needed.
  - 7) Relate problem to similar problem or experience
  - 8) Use an algorithm
  - 9) Evaluate and examine and evaluate the answer to see it makes sense.

## **Introduction to C Programming**

C is a general-purpose computer programming language developed in 1972 by Dennis Ritchie at the Bell Telephone Laboratories for use with the Unix operating system. C is a structured programming language, which means that it allows you to develop programs using well-defined control structures (you will learn about control structures in the articles to come), and provides modularity (breaking the task into multiple sub tasks that are simple enough to understand and to reuse). C is often called a middle-level language because it combines the best elements of low-level or machine language with high-level languages.

### ***Where is C useful?***

C's ability to communicate directly with hardware makes it a powerful choice for system programmers. In fact, popular operating systems such as Unix and Linux are written entirely in C. Additionally, even compilers and interpreters for other languages such as FORTRAN, Pascal, and BASIC are written in C. However, C's scope is not just limited to developing system programs. It is also used to develop any kind of application, including complex business ones. The following is a partial list of areas where C language is used:

- Embedded Systems
- Systems Programming
- Artificial Intelligence

- Industrial Automation
- Computer Graphics
- Space Research

### **Why you should learn C?**

You should learn C because:

- C is simple.
- There are only 32 *keywords* so C is very easy to master. Keywords are words that have special meaning in C language.
- C programs run faster than programs written in most other languages.
- C enables easy communication with computer hardware making it easy to write system programs such as *compilers* and *interpreters*.

### **WHY WE NEED DATA AND A PROGRAM**

Any computer program has two entities to consider, the data, and the program. They are highly dependent on one another and careful planning of both will lead to a well planned and well written program. Unfortunately, it is not possible to study either completely without a good working knowledge of the other. For that reason, this tutorial will jump back and forth between teaching methods of program writing and methods of data definition. Simply follow along and you will have a good understanding of both. Keep in mind that, even though it seems expedient to sometimes jump right into coding the program, time spent planning the data structures will be well spent and the quality of the final program will reflect the original planning

#### ***How to run a simple c program***

1. Copy Turbo c/c++ in computer
2. Open c:\tc\bin\tc.exe
3. A window appears
4. Select File->new to open a new file

5. Type the following program on editor

```
#include  
  
void main()  
  
{  
  
    print f("hello");  
  
}
```

6. compile the program by pressing ALT+F9

7. Run the program by pressing CTRL +F9

**Note:**

1. C is case sensitive
2. Always terminate statements with semicolon.
3. A program starts with main()

### **Explanation of program**

#include is known as compiler directive. A compiler directive is a command to compiler to translate the program in a certain way. These statement are not converted into machine language but only perform some other task.

main() is a function which the starting point for compiler to start compilation. So a function must contain a main() function.

### **DETECTION AND CORRECTION OF ERRORS**

Syntactic errors and execution errors usually result in the generation of error messages when compiling or executing a program. Error of this type is usually quite easy to find and correct. There are some logical errors that can be very difficult to detect. Since the output resulting from a logically incorrect program may appear to be error free. Logical errors are often hard to find, so in order to find and correct errors of this type is known as logical debugging. To detect errors test a new program with data that will give a known answer. If the correct results are not obtained then the program obviously contains errors even if the correct results are obtained.

**Computer Applications:** However you cannot be sure that the program is error free, since some errors cause incorrect result only under certain circumstances. Therefore a new program should receive thorough testing before it is considered to be debugged. Once it has been established that a program contains a logical error, some ingenuity may be required to find the error. Error detection should always begin with a thorough review of each logical group of statements within the program. If the error cannot be found, it sometimes helps to set the program aside for a while. If an error cannot be located simply by inspection, the program should be modified to print out certain intermediate results and then be rerun. This technique is referred to as tracing. The source of error will often become evident once these intermediate calculations have been carefully examined. The greater the amount of intermediate output, the more likely the chances of pointing the source of errors. Sometimes an error simply cannot be located. Some C compilers include a debugger, which is a special program that facilitates the detection of errors in C programs. In particular a debugger allows the execution of a source program to be suspended at designated places, called break points, revealing the values assigned to the program variables and array elements at the time execution stops. Some debuggers also allow a program to execute continuously until some specified error condition has occurred. By examining the values assigned to the variables at the break points, it is easier to determine when and where an error originates.

## **Linear Programming**

Linear program is a method for straightforward programming in a sequential manner. This type of programming does not involve any decision making. General model of these linear programs is:

1. Read a data value
2. Computer an intermediate result
3. Use the intermediate result to computer the desired answer
4. Print the answer

## 5. Stop

### Structured Programming

Structured programming (sometimes known as modular programming) is a subset of procedural programming that enforces a logical structure on the program being written to make it more efficient and easier to understand and modify. Certain languages such as Ada, Pascal, and BASE are designed with features that encourage or enforce a logical program structure.

Structured programming frequently employs a top-down design model, in which developers map out the overall program structure into separate subsections. A defined function or set of similar functions is coded in a separate module or sub module, which means that code can be loaded into memory more efficiently and that modules can be reused in other programs. After a module has been tested individually, it is then integrated with other modules into the overall program structure.

### Advantages of Structured Programming

#### 1. **Easy to write:**

Modular design increases the programmer's productivity by allowing them to look at the big picture first and focus on details later. Several Programmers can work on a single, large program, each working on a different module. Studies show structured programs take less time to write than standard programs. Procedures written for one program can be reused in other programs requiring the same task. A procedure that can be used in many programs is said to be reusable.

#### 2. **Easy to debug:**

Since each procedure is specialized to perform just one task, a procedure can be checked individually. Older unstructured programs consist of a sequence of instructions that are not grouped for specific tasks. The logic of such programs is cluttered with details and therefore difficult to follow.

#### 3. **Easy to Understand:**

The relationship between the procedures shows the modular design of the program. Meaningful procedure names and clear documentation identify the task performed by each module. Meaningful variable names help the programmer identify the purpose of each variable.

#### 4. **Easy to Change:**

Since a correctly written structured program is self-documenting, it can be easily understood by another programmer.

## Structured Programming Constructs

It uses only three constructs -

- Sequence (statements, blocks)
- Selection (if, switch)
- Iteration (loops like while and for)

### ***Sequence***

- Any valid expression terminated by a semicolon is a statement.
- Statements may be grouped together by surrounding them with a pair of curly braces.
- Such a group is syntactically equivalent to one statement and can be inserted wherever
- One statement is legal.

### ***Selection***

The selection constructs allow us to follow different paths in different situations. We may also think of them as enabling us to express decisions.

The main selection construct is:

*if (expression)*

*statement1*

*else*

*statement2*

*statement1* is executed if and only if *expression* evaluates to some non-zero number. If *expression* evaluates to 0, *statement1* is not executed. In that case, *statement2* is executed.

If and else are independent constructs, in that if can occur without else (but not the reverse) Any else is paired with the most recent else-less if, unless curly braces enforce a different scheme. Note that only curly braces, not parentheses, must be used to enforce the pairing. Parentheses

## ***Iteration***

Looping is a way by which we can execute any some set of statements more than one times continuously .In C there are mainly three types of loops are used :

- while Loop
- do while Loop
- For Loop

The control structures are easy to use because of the following reasons:

- 1) They are easy to recognize
  - 2) They are simple to deal with as they have just one entry and one exit point
  - 3) They are free of the complications of any particular programming language
- 

## **WEBSITE DEVELOPMENT**

### **Modular Design of Programs**

One of the key concepts in the application of programming is the design of a program as a set of units referred to as blocks or modules. A style that breaks large computer programs into smaller elements called modules. Each module performs a single task; often a task that needs to be performed multiple times during the running of a program. Each module also stands alone with defined input and output. Since modules are able to be reused they can be designed to be used for multiple programs. By debugging each module and only including it when it performs its defined task, larger programs are easier to debug because large sections of the code have already been evaluated for errors. That usually means errors will be in the logic that calls the various modules.

Languages like Modula-2 were designed for use with modular programming. Modular programming has generally evolved into object-oriented programming.

***Programs can be logically separated into the following functional modules:***

- 1) Initialization
- 2) Input

- 3) Input Data Validation
- 4) Processing
- 5) Output
- 6) Error Handling
- 7) Closing procedure

***Basic attributes of modular programming:***

- 1) Input
- 2) Output
- 3) Function
- 4) Mechanism
- 5) Internal data

***Control Relationship between modules:***

The structure charts show the interrelationships of modules by arranging them at different levels and connecting modules in those levels by arrows. An arrow between two modules means the program control is passed from one module to the other at execution time. The first module is said to call or invoke the lower level modules. There are three rules for controlling the relationship between modules.

- 1) There is only one module at the top of the structure. This is called the root or boss module.
- 2) The root passes control down the structure chart to the lower level modules. However, control is always returned to the invoking module and a finished module should always terminate at the root.
- 3) There can be more than one control relationship between two modules on the structure chart, thus, if module A invokes module B, then B cannot invoke module A.

***Communication between modules:***

- 1) **Data:** Shown by an arrow with empty circle at its tail.
- 2) **Control :** Shown by a filled-in circle at the end of the tail of arrow

## **Module Design Requirements**

A hierarchical or module structure should prevent many advantages in management, developing, testing and maintenance. However, such advantages will occur only if modules fulfill the following requirements.

a) **Coupling:** In computer science, coupling is considered to be the degree to which each program module relies on other modules, and is also the term used to describe connecting two or more systems. Coupling is broken down into loose coupling, tight coupling, and decoupled. Coupling is also used to describe software as well as systems. Also called dependency

## **Types of Programming Language**

### **Low Level Language**

First-generation language is the lowest level computer\_language. Information is conveyed to the computer by the\_programmer

as binary instructions. Binary instructions are the equivalent of the on/off signals used by computers to carry out operations. The language consists of zeros and ones. In the 1940s and 1950s, computers were programmed by scientists sitting before control panels equipped with toggle switches so that they could input instructions as strings of zeros and ones.

Advantages

- Fast and efficient
- Machine oriented
- No translation required

Disadvantages

- Not portable
- Not programmer friendly

### **Assembly Language**

Assembly or assembler language was the second generation of computer language. By the late 1950s, this language had become popular. Assembly language consists of letters of the alphabet. This makes programming much easier than trying to program a series of zeros and

ones. As an added programming assist, assembly language makes use of mnemonics, or memory aids, which are easier for the human programmer to recall than are numerical codes.

### **Assembler**

An assembler is a program that takes basic computer instructions and converts them into a pattern of bits that the computer's processor can use to perform its basic operations. Some people call these instructions assembler language and others use the term *assembly language*. In other words An **assembler** is a computer program for translating assembly language — essentially, a mnemonic representation of machine language — into object code. A **cross assembler** (see cross compiler) produces code for one processor, but runs on another.

As well as translating assembly instruction mnemonics into op codes, assemblers provide the ability to use symbolic names for memory locations (saving tedious calculations and manually updating addresses when a program is slightly modified), and macro facilities for performing textual substitution — typically used to encode common short sequences of instructions to run inline instead of in a subroutine.

### **High Level Language**

The introduction of the compiler in 1952 spurred the development of third-generation computer languages. These languages enable a programmer to create program files using commands that are similar to spoken English. Third-level computer languages have become the major means of communication between the digital computer and its user. By 1957, the International Business Machine Corporation (IBM) had created a language called FORTRAN (Formulas Translate). This language was designed for scientific work involving complicated mathematical formulas. It became the first high-level programming language (or "source code") to be used by many computer users.

Within the next few years, refinements gave rise to ALGOL (Algorithmic Language) and COBOL (Common Business Oriented Language). COBOL is noteworthy because it improved the record keeping and data management ability of businesses, which stimulated business expansion.

#### Advantages

- Portable or *machine independent*
- Programmer-friendly

#### Disadvantages

-Not as efficient as low-level languages

-Need to be translated

Examples : C, C++, Java, FORTRAN, Visual Basic, and Delphi.

## **Interpreter**

An **interpreter** is a computer program that executes other programs. This is in contrast to a compiler which does not execute its input program (the source code) but translates it into executable machine code (also called object code) which is output to a file for later execution. It may be possible to execute the same source code either directly by an interpreter or by compiling it and then executing the machine code produced.

It takes longer to run a program under an interpreter than to run the compiled code but it can take less time to interpret it than the total required to compile and run it. This is especially important when prototyping and testing code when an edit-interpret-debug cycle can often be much shorter than an edit-compile-run-debug cycle.

Interpreting code is slower than running the compiled code because the interpreter must analyses each statement in the program each time it is executed and then perform the desired action whereas the compiled code just performs the action. This run-time analysis is known as "interpretive overhead". Access to variables is also slower in an interpreter because the mapping of identifiers to storage locations must be done repeatedly at run-time rather than at compile time.

## **COMPILER**

A program that translates source code into object code. The compiler derives its name from the way it works, looking at the entire piece of source code and collecting and reorganizing the instructions. Thus, a compiler differs from an interpreter, which analyzes and executes each line of source code in succession, without looking at the entire program. The advantage of interpreters is that they can execute a program immediately. Compilers require some time before an executable program emerges. However, programs produced by compilers run much faster than the same programs executed by an interpreter.

Every high-level programming language (except strictly interpretive languages) comes with a compiler. In effect, the compiler is the language, because it defines which instructions are acceptable.

## **Fourth Generation Language**

Fourth-generation languages attempt to make communicating with computers as much like the processes of thinking and talking to other people as possible. The problem is that the computer still only understands zeros and ones, so a compiler and interpreter must still convert the source code into the machine code that the computer can understand. Fourth-generation languages typically consist of English-like words and phrases. When they are implemented on microcomputers, some of these languages include graphic devices such as icons and onscreen push buttons for use during programming and when running the resulting application.

Many fourth-generation languages use Structured Query Language (SQL) as the basis for operations. SQL was developed at IBM to develop information stored in relational databases. Examples of fourth-generation languages include PROLOG, an Artificial Intelligence language

## UNIT-2

### FEATURES OF ‘C’

#### C-Language keywords

auto	break	case	char	const
continue	default	do	double	else
enum	extern	float	for	goto
if	int	long	register	return
short	signed	sizeof	static	struct
switch	typedef	union	unsigned	void
volatile	while			

### Data Types

A C language programmer has to tell the system before-hand, the type of numbers or characters he is using in his program. These are data types. There are many data types in C language. A C programmer has to use appropriate data type as per his requirement. C language data types can be broadly classified as

- Primary data type
- Derived data type
- User defined data type

#### Primary data type

All C Compilers accept the following fundamental data types

1.	Integer	int
2.	Character	char
3.	Floating Point	float
4.	Double precision floating point	double
5.	Void	void

The size and range of each data type is given in the table below

DATA TYPE	RANGE OF VALUES
char	-128 to 127

Int	-32768 to +32767
float	3.4 e-38 to 3.4 e+38
double	1.7 e-308 to 1.7 e+308

### **Integer Type:**

Integers are whole numbers with a machine dependent range of values. A good programming language as to support the programmer by giving a control on a range of numbers and storage space. C has 3 classes of integer storage namely short int, int and long int. All of these data types have signed and unsigned forms. A short int requires half the space than normal integer values. Unsigned numbers are always positive and consume all the bits for the magnitude of the number. The long and unsigned integers are used to declare a longer range of values.

### **Floating Point Types:**

Floating point number represents a real number with 6 digits precision. Floating point numbers are denoted by the keyword float. When the accuracy of the floating point number is insufficient, we can use the double to define the number. The double is same as float but with longer precision. To extend the precision further we can use long double which consumes 80 bits of memory space.

### **Void Type:**

Using void data type, we can specify the type of a function. It is a good practice to avoid functions that does not return any values to the calling function.

### **Character Type:**

A single character can be defined as a defined as a character type of data. Characters are usually stored in 8 bits of internal storage. The qualifier signed or unsigned can be explicitly applied to char. While unsigned characters have values between 0 and 255, signed characters have values from -128 to 127.

### **Size and Range of Data Types on 16 bit machine.**

TYPE	SIZE (Bits)	Range
Char or Signed Char	8	-128 to 127

Unsigned Char	8	0 to 255
Int or Signed int	16	-32768 to 32767
Unsigned int	16	0 to 65535
Short int or Signed short int	8	-128 to 127
Unsigned short int	8	0 to 255
Long int or signed long int	32	-2147483648 to 2147483647
Unsigned long int	32	0 to 4294967295
Float	32	3.4 e-38 to 3.4 e+38
Double	64	1.7e-308 to 1.7e+308
Long Double	80	3.4 e-4932 to 3.4 e+4932

### Variable:

Variable is a name of memory location where we can store any data. It can store only single data (Latest data) at a time. In C, a variable must be declared before it can be used. Variables can be declared at the start of any block of code, but most are found at the start of each function.

A declaration begins with the type, followed by the name of one or more variables. For example,

```
Data Type Name_of_Variable_Name;
```

```
int a,b,c;
```

### Variable Names

Every variable has a name and a value. The name identifies the variable, the value stores data. There is a limitation on what these names can be. Every variable name in C must start with a letter; the rest of the name can consist of letters, numbers and underscore characters. C recognizes upper and lower case characters as being different. Finally, you cannot use any of C's keywords like main, while, switch etc as variable names.

Examples of legal variable names include:

x	result	out_file	best-yet
x1	x2	out_file	best_yet
power	impetus	gamma	hi_score

It is conventional to avoid the use of capital letters in variable names. These are used for names of constants. Some old implementations of C only use the first 8 characters of a variable name.

### **Local Variables**

Local variables are declared within the body of a function, and can only be used within that function only.

Syntax:

```
Void main( ){
```

```
    int a,b,c;
```

```
}
```

```
Void fun1()
```

```
{
```

```
    int x,y,z;
```

```
}
```

Here a,b,c are the local variable of void main() function and it can't be used within fun1() Function. And x, y and z are local variable of fun1().

## **Global Variable**

A global variable declaration looks normal, but is located outside any of the program's functions. This is usually done at the beginning of the program file, but after preprocessed directives. The variable is not declared again in the body of the functions which access it.

Syntax:

```
#include  
  
int a,b,c;  
  
void main()
```

```
{  
  
}
```

```
Void fun1()  
  
{  
  
}
```

Here a,b,c are global variable .and these variable cab be accessed (used) within a whole program.

## **Constants**

C constant is usually just the written version of a number. For example 1, 0, 5.73, 12.5e9. We can specify our constants in octal or hexadecimal, or force them to be treated as long integers.

- Octal constants are written with a leading zero - 015.
- Hexadecimal constants are written with a leading 0x - 0x1ae.
- Long constants are written with a trailing L - 890L.

Character constants are usually just the character enclosed in single quotes; 'a', 'b', 'c'. Some characters can't be represented in this way, so we use a 2 character sequence.

```
'\n'    newline
'\t'    tab
'\\'    backslash
'\'    single quote
'\0'    null (used automatically to terminate character strings).
```

In addition, a required bit pattern can be specified using its octal equivalent.

'\044' produces bit pattern 00100100.

Character constants are rarely used, since string constants are more convenient. A string constant is surrounded by double quotes e.g. "Brian and Dennis". The string is actually stored as an array of characters. The null character '\0' is automatically placed at the end of such a string to act as a string terminator.

Constant is a special types of variable which can not be changed at the time of execution.  
Syntax:

```
const int a=20;
```

## C Language Operator Precedence Chart

Operator precedence describes the order in which C reads expressions. For example, the expression `a=4+b*2` contains two operations, an addition and a multiplication. Does the C compiler evaluate `4+b` first, then multiply the result by 2, or does it evaluate `b*2` first, then add 4 to the result? The operator precedence chart contains the answers. Operators higher in the chart have a higher precedence, meaning that the C compiler evaluates them first. Operators on the same line in the chart have the same precedence, and the "Associatively" column on the right gives their evaluation order.

Operator Precedence Chart		
Operator Type	Operator	Associativity
Primary Expression Operators	( ) [ ] . -> <i>expr++</i> <i>expr--</i>	left-to-right

Unary Operators	<code>* &amp; + - ! ~ ++expr --expr (typecast) sizeof()</code>	right-to-left
Binary Operators	<code>*</code> / %	left-to-right
	<code>+ -</code>	
	<code>&gt;&gt; &lt;&lt;</code>	
	<code>&lt; &gt; &lt;= &gt;=</code>	
	<code>== !=</code>	
	<code>&amp;</code>	
	<code>^</code>	
	<code> </code>	
	<code>&amp;&amp;</code>	
	<code>  </code>	
Ternary Operator	<code>? :</code>	right-to-left
Assignment Operators	<code>= += -= *= /= %= &gt;&gt;= &lt;&lt;= &amp;= ^=  =</code>	right-to-left

Comma	,	left-to-right
-------	---	---------------

## **Operators Introduction**

An operator is a symbol which helps the user to command the computer to do a certain mathematical or logical manipulations. Operators are used in C language program to operate on data and variables. C has a rich set of operators which can be classified as

- |                      |                    |
|----------------------|--------------------|
| 1.Arithmetic         | operators          |
| 2.Relational         | Operators          |
| 3.Logical            | Operators          |
| 4.Assignment         | Operators          |
| 5.Increments         | and      Decrement |
| 6.Conditional        | Operators          |
| 7.Bitwise            | Operators          |
| 8. Special Operators | Operators          |

### **1. Arithmetic Operators**

All the basic arithmetic operations can be carried out in C. All the operators have almost the same meaning as in other languages. Both unary and binary operations are available in C language. Unary operations operate on a single operand, therefore the number 5 when operated by unary – will have the value –5.

#### **Arithmetic Operators**

Operator	Meaning
+	Addition or Unary Plus
–	Subtraction or Unary Minus
*	Multiplication
/	Division
%	Modulus Operator

Examples of arithmetic operators are

x	+	y
x	-	y
-x	+	y
a	*	b
-a * b		c

etc.,

here a, b, c, x, y are known as operands. The modulus operator is a special operator in C language which evaluates the remainder of the operands after division.

## Example

```
#include           //include          header          file          stdio.h
void  main()    //tell   the   compiler   the   start   of   the   program
{
int num1, num2, sum, sub, mul, div, mod; //declaration of variables
scanf ("%d %d", &num1, &num2); //inputs the operands

sum = num1+num2; //addition of numbers and storing in sum.
printf("\n Thu sum is = %d", sum); //display the output

sub = num1-num2; //subtraction of numbers and storing in sub.
printf("\n Thu difference is = %d", sub); //display the output

mul = num1*num2; //multiplication of numbers and storing in mul.
printf("\n Thu product is = %d", mul); //display the output

div = num1/num2; //division of numbers and storing in div.
printf("\n Thu division is = %d", div); //display the output

mod = num1%num2; //modulus of numbers and storing in mod.
printf("\n Thu modulus is = %d", mod); //display the output
```

}

## Integer Arithmetic

When an arithmetic operation is performed on two whole numbers or integers than such an operation is called as integer arithmetic. It always gives an integer as the result. Let  $x=27$  and  $y = 5$  be 2 integer numbers. Then the integer operation leads to the following results.

x	+	y	=	32
x	-	y	=	22
x	*	y	=	115
x	%	y	=	2
x	/	y	=	5

In integer division the fractional part is truncated.

## Floating point arithmetic

When an arithmetic operation is preformed on two real numbers or fraction numbers such an operation is called floating point arithmetic. The floating point results can be truncated according to the properties requirement. The remainder operator is not applicable for floating point arithmetic operands.

Let	x	=	14.0	and	y	=	4.0	then
x	+		y		=		18.0	
x	-		y		=		10.0	
x	*		y		=		56.0	
$x / y = 3.50$								

## Mixed mode arithmetic

When one of the operand is real and other is an integer and if the arithmetic operation is carried out on these 2 operands then it is called as mixed mode arithmetic. If any one operand is of real type then the result will always be real thus  $15/10.0 = 1.5$

## 2. Relational Operators

Often it is required to compare the relationship between operands and bring out a decision and program accordingly. This is when the relational operator come into picture. C supports the following relational operators.

<b>Operator</b>	<b>Meaning</b>
<	is less than
<=	is less than or equal to
>	is greater than
>=	is greater than or equal to
==	is equal to

It is required to compare the marks of 2 students, salary of 2 persons, we can compare them using relational operators.

A simple relational expression contains only one relational operator and takes the following form.

expl relational operator exp2

Where `exp1` and `exp2` are expressions, which may be simple constants, variables or combination of them. Given below is a list of examples of relational expressions and evaluated values.

6.5	<=	25	TRUE	
-65	>	0	FALSE	
10	<	7	+ 5	TRUE

Relational expressions are used in decision making statements of C language such as if, while and for statements to decide the course of action of a running program.

### 3. Logical Operators

C has the following logical operators, they compare or evaluate logical and relational expressions.

Operator	Meaning
&&	Logical AND
	Logical OR
!	Logical NOT

## Logical AND (&&)

This operator is used to evaluate 2 conditions or expressions with relational operators simultaneously. If both the expressions to the left and to the right of the logical operator is true then the whole compound expression is true.

### **Example**

a > b && x = 10

The expression to the left is **a > b** and that on the right is **x == 10** the whole expression is true only if both expressions are true i.e., if a is greater than b and x is equal to 10.

## Logical OR (||)

The logical OR is used to combine 2 expressions or the condition evaluates to true if any one of the 2 expressions is true.

### **Example**

a < m || a < n

The expression evaluates to true if any one of them is true or if both of them are true. It evaluates to true if a is less than either m or n and when a is less than both m and n.

## Logical NOT (!)

The logical not operator takes single expression and evaluates to true if the expression is false and evaluates to false if the expression is true. In other words it just reverses the value of the expression.

### **For example**

$!(x \geq y)$  the NOT expression evaluates to true only if the value of x is neither greater than or equal to y

## 4. Assignment Operators

The Assignment Operator evaluates an expression on the right of the expression and substitutes it to the value or variable on the left of the expression.

### **Example**

$x = a + b$

Here the value of  $a + b$  is evaluated and substituted to the variable  $x$ .

In addition, C has a set of shorthand assignment operators of the form.

var                    oper                    =                    exp;

Here var is a variable, exp is an expression and oper is a C binary arithmetic operator. The operator  $oper =$  is known as shorthand assignment operator

### Example

$x += 1$  is same as  $x = x + 1$

The commonly used shorthand assignment operators are as follows  
Shorthand assignment operators .

Statement with simple assignment operator	Statement with shorthand operator
$a = a + 1$	$a += 1$
$a = a - 1$	$a -= 1$
$a = a * (n+1)$	$a *= (n+1)$
$a = a / (n+1)$	$a /= (n+1)$
$a = a \% b$	$a \%= b$

### Example for using shorthand assignment operator

```
#define N 100 //creates a variable N with constant value 100
#define A 2 //creates a variable A with constant value 2

main() //start of the program
{
int a; //variable a declaration
a = A; //assigns value 2 to a

while (a < N) //while value of a is less than N
{ //evaluate or do the following}
```

```
printf("%d\n",a); //print the current value of a
a *= a; //shorthand form of a = a * a
} //end of the loop
}//end of the program
```

## Output

2  
4  
16

### **5. Increment and Decrement Operators**

The increment and decrement operators are one of the unary operators which are very useful in C language. They are extensively used in for and while loops. The syntax of the operators is given below

- |    |          |           |        |
|----|----------|-----------|--------|
| 1. | ++       | variable  | name   |
| 2. |          | variable  | name++ |
| 3. | -        | -variable | name   |
| 4. | variable | name-     | -      |

The increment operator ++ adds the value 1 to the current value of operand and the decrement operator -- subtracts the value 1 from the current value of operand. ++variable name and variable name++ mean the same thing when they form statements independently, they behave differently when they are used in expression on the right hand side of an assignment statement.

#### **Consider the following .**

m = 5;  
y = ++m; (prefix)

In this case the value of **y** and **m** would be 6  
Suppose if we rewrite the above statement as

m =  
y = m++; (post fix)

Then the value of **y** will be 5 and that of **m** will be 6. A prefix operator first adds 1 to the operand and then the result is assigned to the variable on the left. On the other hand, a postfix operator first assigns the value to the variable on the left and then increments the operand.

## **6. Conditional or Ternary Operator**

The conditional operator consists of 2 symbols the question mark (?) and the colon (:). The syntax for a ternary operator is as follows .

exp1                      ?                      exp2                      :                      exp3

The ternary operator works as follows

exp1 is evaluated first. If the expression is true then exp2 is evaluated & its value becomes the value of the expression. If exp1 is false, exp3 is evaluated and its value becomes the value of the expression. Note that only one of the expression is evaluated.

<b>For</b>	<b>example</b>
a	= 10;
b	= 15;
x = (a > b) ? a : b	

Here x will be assigned to the value of b. The condition follows that the expression is false therefore b is assigned to x.

```
/*
 * Example : to find the maximum value using conditional operator)
#include
void main() //start of the program
{
int i,j,larger; //declaration of variables
printf ("Input 2 integers : "); //ask the user to input 2 numbers
scanf("%d %d",&i, &j); //take the number from standard input and store it
larger = i > j ? i : j; //evaluation using ternary operator
printf("The largest of two numbers is %d \n", larger); // print the largest number
} // end of the program
*/
```

## Output

Input	2	integers	:	34	45
-------	---	----------	---	----	----

The largest of two numbers is 45

## 7. Bitwise Operators

C has a distinction of supporting special operators known as bitwise operators for manipulation data at bit level. A bitwise operator operates on each bit of data. Those operators are used for testing, complementing or shifting bits to the right or left. Bitwise operators may not be applied to a float or double.

Operator	Meaning
&	Bitwise AND
	Bitwise OR
^	Bitwise Exclusive
<<	Shift left
>>	Shift right

## 8. Special Operators

C supports some special operators of interest such as comma operator, size of operator, pointer operators(& and \*) and member selection operators(. and ->). The size of and the comma operators are discussed here. The remaining operators are discussed in forthcoming chapters.

### The Comma Operator

The comma operator can be used to link related expressions together. A comma-linked list of expressions are evaluated left to right and value of right most expression is the value of the combined expression.

#### For example the statement

value = (x = 10, y = 5, x + y);

First assigns 10 to **x** and 5 to **y** and finally assigns 15 to **value**. Since comma has the lowest precedence in operators the parenthesis is necessary. Some examples of comma operator are

In for loops:

```
for (n=1, m=10, n <=m; n++,m++)
```

In while loops

```
While (c=get char(), c != '10')
```

Exchanging values.

```
t = x, x = y, y = t;
```

### **The size of Operator**

The operator size of gives the size of the data type or variable in terms of bytes occupied in the memory. The operand may be a variable, a constant or a data type qualifier.

#### **Example**

```
m           =           size           of           (sum);
n           =           size           of           (long          int);
k = size of (235L);
```

The size of operator is normally used to determine the lengths of arrays and structures when their sizes are not known to the programmer. It is also used to allocate memory space dynamically to variables during the execution of the program.

Example program that employs different kinds of operators. The results of their evaluation are also shown in comparison .

main()	//start	of	program
{			
int a, b, c, d; //declaration	of	variables	
a = 15; b = 10; c = ++a-b; //assign values to variables			
printf ("a = %d, b = %d, c = %d\n", a,b,c); //print the values			
d=b++	+		a;
printf ("a = %d, b = %d, d = %d\n", a,b,d);			

```

printf ("a / b = %d\n", a / b);
printf ("a %% b = %d\n", a % b);
printf ("a *= b = %d\n", a *= b);
printf ("%d\n", (c > d) ? 1 : 0 );
printf ("%d\n", (c < d) ? 1 : 0 );
}

```

Notice the way the increment operator `++` works when used in an expression. In the statement `c = ++a - b;` new value `a = 16` is used thus giving value 6 to C. That is a is incremented by 1 before using in expression.

However in the statement `d = b++ + a;` The old value `b = 10` is used in the expression. Here b is incremented after it is used in the expression.

We can print the character `%` by placing it immediately after another `%` character in the control string. This is illustrated by the statement.

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

This program also illustrates that the expression

`c > d ? 1 : 0`

Assumes the value 0 when `c` is less than `d` and 1 when `c` is greater than `d`.

## Type conversions in expressions

### **Implicit type conversion**

C permits mixing of constants and variables of different types in an expression. C automatically converts any intermediate values to the proper type so that the expression can be evaluated without losing any significance. This automatic type conversion is known as implicit type conversion.

During evaluation it adheres to very strict rules and type conversion. If the operands are of different types the lower type is automatically converted to the higher type before the operation proceeds. The result is of higher type.

The following rules apply during evaluating expressions

All short and char are automatically converted to int then

1. If one operand is long double, the other will be converted to long double and result will be long double.
2. If one operand is double, the other will be converted to double and result will be double.
3. If one operand is float, the other will be converted to float and result will be float.
4. If one of the operand is unsigned long int, the other will be converted into unsigned long int and result will be unsigned long int.
5. If one operand is long int and other is unsigned int then .
  - a. If unsigned int can be converted to long int, then unsigned int operand will be converted as such and the result will be long int.
  - b. Else Both operands will be converted to unsigned long int and the result will be unsigned long int.
6. If one of the operand is long int, the other will be converted to long int and the result will be long int.
7. If one operand is unsigned int the other will be converted to unsigned int and the result will be unsigned int.

### **Explicit Conversion**

Many times there may arise a situation where we want to force a type conversion in a way that is different from automatic conversion.

Consider for example the calculation of number of female and male students in a class

$$\text{Ratio} = \frac{\text{female\_students}}{\text{male\_students}}$$

Since if female\_students and male\_students are declared as integers, the decimal part will be rounded off and its ratio will represent a wrong figure. This problem can be solved by converting locally one of the variables to the floating point as shown below.

$$\text{Ratio} = (\text{float}) \text{female\_students} / \text{male\_students}$$

The operator float converts the female\_students to floating point for the purpose of evaluation of the expression. Then using the rule of automatic conversion, the division is performed by floating point mode, thus retaining the fractional part of the result. The process of such a local conversion is known as explicit conversion or casting a value. The general form is

(type\_name) expression

## Specifier Meaning

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

## Input and Output

Input and output are covered in some detail. C allows quite precise control of these. This section discusses input and output from keyboard and screen.

The same mechanisms can be used to read or write data from and to files. It is also possible to treat character strings in a similar way, constructing or analysis them and storing results in variables. These variants of the basic input and output commands are discussed in the next section

- [The Standard Input Output File](#)
- [Character Input / Output](#)
  - [getchar](#)
  - [putchar](#)
- [Formatted Input / Output](#)
  - [printf](#)
  - [scanf](#)
- [Whole Lines of Input and Output](#)
  - [gets](#)
  - [puts](#)

This offers more structured output than put char. Its arguments are, in order; a control string, which controls what gets printed, followed by a list of values to be substituted for entries in the control string

Example: int a,b;

printf(" a = %d,b=%d",a,b);.

Control String Entry	What Gets Printed
%d	A Decimal Integer
%f	A Floating Point Value
%c	A Character
%s	A Character String

It is also possible to insert numbers into the control string to control field widths for values to be displayed. For example %6d would print a decimal value in a field 6 spaces wide, %8.2f would print a real value in a field 8 spaces wide with room to show 2 decimal places. Display is left justified by default, but can be right justified by putting a - before the format information, for example %-6d, a decimal integer right justified in a 6 space field

### **scanf**

scanf allows formatted reading of data from the keyboard. Like printf it has a control string, followed by the list of items to be read. However scanf wants to know the address of the items to be read, since it is a function which will change that value. Therefore the names of variables are preceded by the & sign. Character strings are an exception to this. Since a string is already a character pointer, we give the names of string variables unmodified by a leading &.

Control string entries which match values to be read are preceded by the percentage sign in a similar way to their printf equivalents.

Example: int a,b;

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

### **getchar**

getchar returns the next character of keyboard input as an int. If there is an error then EOF (end of file) is returned instead. It is therefore usual to compare this value against EOF before using it. If the return value is stored in a char, it will never be equal to EOF, so error conditions will not be handled correctly.

As an example, here is a program to count the number of characters read until an EOF is encountered. EOF can be generated by typing Control - d.

```
#include
main()
{    int ch, i = 0;
    while((ch = getchar()) != EOF)
        i++;
    printf("%d\n", i);
}
```

### **putchar**

putchar puts its character argument on the standard output (usually the screen).

The following example program converts any typed input into capital letters. To do this it applies the function to\_upper from the character conversion library c type .h to each character in turn.

```
#include /* For definition of to-upper */
#include /* For definition of getchar, putchar, EOF */
main()
{    char ch;
while((ch = getchar()) != EOF)
    putchar (to-upper(ch));
}
```

### **gets**

gets reads a whole line of input into a string until a new line or EOF is encountered. It is critical to ensure that the string is large enough to hold any expected input lines.

When all input is finished, NULL as defined in studio is returned.

```
#include

main()
{    char ch[20];
    gets(x);
    puts(x);
}
```

### **puts**

puts writes a string to the output, and follows it with a new line character.

Example: Program which uses gets and puts to double space typed input.

```
#include
main()
{   char line[256]; /* Define string sufficiently large to store
a line of input */
    while(gets(line) != NULL) /* Read line */
    {   puts(line); /* Print line */
        printf("\n"); /* Print blank line */
    }
}
```

Note that putchar, printf and puts can be freely used together

### **Expression Statements**

Most of the statements in a C program are *expression statements*. An expression statement is simply an expression followed by a semicolon. The lines

```
i = 0;
i = i + 1;
```

and

```
printf("Hello, world!\n");
```

are all expression statements. (In some languages, such as Pascal, the semicolon separates statements, such that the last statement is not followed by a semicolon. In C, however, the semicolon is a statement terminator; all simple statements are followed by semicolons. The semicolon is also used for a few other things in C; we've already seen that it terminates declarations, too.

## **UNIT – 3**

### **Branching and looping**

### **CONTROL FLOW STATEMENT**

#### **IF- Statement:**

It is the basic form where the if statement evaluate a test condition and direct program execution depending on the result of that evaluation.

Syntax:

```
If (Expression)
```

```
{
```

```
Statement 1;
```

```
Statement 2;
```

```
}
```

Where a statement may consist of a single statement, a compound statement or nothing as an empty statement. The Expression also referred so as test condition must be enclosed in parenthesis, which cause the expression to be evaluated first, if it evaluate to true (a non zero value), then the statement associated with it will be executed otherwise ignored and the control will pass to the next statement.

Example:

```
if (a>b)
```

```
{
```

```
printf ("a is larger than b");
```

```
}
```

### **IF-ELSE Statement:**

An `if` statement may also optionally contain a second statement, the ```else clause`,'' which is to be executed if the condition is not met. Here is an example:

```
if(n > 0)
    average = sum / n;
else {
    printf("can't compute average\n");
    average = 0;
```

}

### **NESTED-IF- Statement:**

It's also possible to nest one `if` statement inside another. (For that matter, it's in general possible to nest any kind of statement or control flow construct within another.) For example, here is a little piece of code which decides roughly which quadrant of the compass you're walking into, based on an `x` value which is positive if you're walking east, and a `y` value which is positive if you're walking north:

```
if(x > 0)
{
    if(y > 0)
        printf("Northeast.\n");
    else printf("Southeast.\n");
}
else {
    if(y > 0)
        printf("Northwest.\n");
    else printf("Southwest.\n");
}

/* Illuminates nested if else and multiple arguments to the scanf function. */
#include
main()
{
    int invalid_operator = 0;
    char operator;
    float number1, number2, result;
    printf("Enter two numbers and an operator in the format\n");
    printf(" number1 operator number2\n");
    scanf("%f %c %f", &number1, &operator, &number2);
    if(operator == '*')
        result = number1 * number2;
    else if(operator == '/')
        result = number1 / number2;
    else if(operator == '+')
        result = number1 + number2;
    else if(operator == '-')
        result = number1 - number2;
```

```

else
    invalid_operator = 1;
if( invalid_operator != 1 )
    printf("%f %c %f is %f\n", number1, operator, number2, result );
else
printf("Invalid operator.\n");
}

```

### **Sample Program Output**

Enter two numbers and an operator in the format

```

number1 operator number2
23.2 + 12
23.2 + 12 is 35.2

```

### **Switch Case**

This is another form of the multi way decision. It is well structured, but can only be used in certain cases where;

- Only one variable is tested, all branches must depend on the value of that variable. The variable must be an integral type. (int, long, short or char).
- Each possible value of the variable can control a single branch. A final, catch all, default branch may optionally be used to trap all unspecified cases.

Hopefully an example will clarify things. This is a function which converts an integer into a vague description. It is useful where we are only concerned in measuring a quantity when it is quite small.

```

estimate(number)
int number;
/* Estimate a number as none, one, two, several, many */
{
    switch(number) {
        case 0 :
            printf("None\n");
            break;
        case 1 :
            printf("One\n");
            break;
    }
}

```

```

        case 2 :
            printf("Two\n");
            break;
        case 3 :
        case 4 :
        case 5 :
            printf("Several\n");
            break;
    default :
        printf("Many\n");
        break;
    }
}

```

Each interesting case is listed with a corresponding action. The break statement prevents any further statements from being executed by leaving the switch. Since case 3 and case 4 have no following break, they continue on allowing the same action for several values of number.

Both if and switch constructs allow the programmer to make a selection from a number of possible actions.

## Loops

Looping is a way by which we can execute any some set of statements more than one times continuously .In c there are mainly three types of loops are use :

- while Loop
- do while Loop
- For Loop

## While Loop

Loops generally consist of two parts: one or more *control expressions* which (not surprisingly) control the execution of the loop, and the *body*, which is the statement or set of statements which is executed over and over.

The general syntax of a `while` loop is

## Initialization

```
while( expression )
{
    Statement1
    Statement2
    Statement3
}
```

The most basic *loop* in C is the `while` loop. A `while` loop has one control expression, and executes as long as that expression is true. This example repeatedly doubles the number 2 (2, 4, 8, 16, ...) and prints the resulting numbers as long as they are less than 1000:

```
int x = 2;
while(x < 1000)
{
    printf("%d\n", x);
    x = x * 2;
}
```

(Once again, we've used braces `{ }` to enclose the group of statements which are to be executed together as the body of the loop.)

## For Loop

Our second loop, which we've seen at least one example of already, is the `for` loop. The general syntax of a `while` loop is

```
for( Initialization;expression;Increments/decrements )
{
    Statement1
    Statement2
    Statement3
}
```

The first one we saw was:

```
for (i = 0; i < 10; i = i + 1)
    printf ("i is %d\n", i);
```

(Here we see that the `for` loop has three control expressions. As always, the *statement* can be a brace-enclosed block.)

### **Do while Loop**

This is very similar to the while loop except that the test occurs at the end of the loop body. This guarantees that the loop is executed at least once before continuing. Such a setup is frequently used where data is to be read. The test then verifies the data, and loops back to read again if it was unacceptable.

```
do
{
    printf("Enter 1 for yes, 0 for no :");
    scanf("%d", &input_value);
} while (input_value != 1 && input_value != 0)
```

### **The break Statement**

We have already met break in the discussion of the switch statement. It is used to exit from a loop or a switch, control passing to the first statement beyond the loop or a switch.

With loops, break can be used to force an early exit from the loop, or to implement a loop with a test to exit in the middle of the loop body. A break within a loop should always be protected within an if statement which provides the test to control the exit condition.

### **The continue Statement**

This is similar to break but is encountered less frequently. It only works within loops where its effect is to force an immediate jump to the loop control statement.

- In a while loop, jump to the test statement.
- In a do while loop, jump to the test statement.
- In a for loop, jump to the test, and perform the iteration.

Like a break, continue should be protected by an if statement. You are unlikely to use it very often.

*Take the following example:*

*int i;*

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

```
if (i==5)
continue;
printf("%d",i);
if (i==8)
break;
}
```

This code will print 1 to 8 except 5.

*Continue means, whatever code that follows the continue statement WITHIN the loop code block will not be executed and the program will go to the next iteration, in this case, when the program reaches i=5 it checks the condition in the if statement and executes 'continue', everything after continue, which are the printf statement, the next if statement, will not be executed.*

*Break statement will just stop execution of the loop and go to the next statement after the loop if any. In this case when i=8 the program will jump out of the loop. Meaning, it wont continue till i=9, 10.*

### **Comment:**

- o The compiler is "line oriented", and parses your program in a line-by-line fashion.

- o There are two kinds of comments: single-line and multi-line comments.

- o The single-line comment is indicated by "//"

This means everything after the first occurrence of("//", UP TO THE END OF CURRENT LINE, is ignored.

- o The multi-line comment is indicated by the pair "/\*" and "\*/".

This means that everything between these two sequences will be ignored. This may ignore any number of lines.

Here is a variant of our first program:

```
/* This is a variant of my first program.  
 * It is not much, I admit.  
 */  
int main() {  
    printf("Hello World!\n"); // that is all?  
    return(0);  
}
```

## **UNIT – 4**

### **ARRAY AND STRING**

Arrays are widely used data type in ‘C’ language. It is a collection of elements of similar data type. These similar elements could be of all integers, all floats or all characters. An array of character is called as string whereas an array of integer or float is simply called as an array. So array may be defined as a group of elements that share a common name and that are defined by position or index. The elements of an arrays are stored in sequential order in memory.

There are mainly two types of Arrays are used:

- One dimensional Array
- Multidimensional Array

#### ***One dimensional Array***

So far, we've been declaring simple variables: the declaration

```
int i;
```

declares a single variable, named `i`, of type `int`. It is also possible to declare an *array* of several elements. The declaration

```
int a[10];
```

declares an array, named `a`, consisting of ten elements, each of type `int`. Simply speaking, an array is a variable that can hold more than one value. You specify which of the several values

you're referring to at any given time by using a numeric *subscript*. (Arrays in programming are similar to vectors or matrices in mathematics.) We can represent the array `a` above with

`a:` 

a picture like this: `[0] [1] [2] [3] [4] [5] [6] [7] [8] [9]`

In C, arrays are *zero-based*: the ten elements of a 10-element array are numbered from 0 to 9. The subscript which specifies a single element of an array is simply an integer expression in square brackets. The first element of the array is `a[0]`, the second element is `a[1]`, etc. You can use these ``array subscript expressions'' anywhere you can use the name of a simple variable, for example:

```
a[0] = 10;
a[1] = 20;
a[2] = a[0] + a[1];
```

Notice that the sub scripted array references (i.e. expressions such as `a[0]` and `a[1]`) can appear on either side of the assignment operator. it is possible to initialize some or all elements of an array when the array is defined. The syntax looks like this:

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

The list of values, enclosed in braces {}, separated by commas, provides the initial values for successive elements of the array.

The subscript does not have to be a constant like 0 or 1; it can be any integral expression. For example, it's common to loop over all elements of an array:

```
int i;
for(i = 0; i < 10; i = i + 1)
    a[i] = 0;
```

This loop sets all ten elements of the array `a` to 0.

Arrays are a real convenience for many problems, but there is not a lot that C will do with them for you automatically. In particular, you can neither set all elements of an array at once nor assign one array to another; both of the assignments

```
a = 0; /* WRONG */
```

and

```
int b[10];
b = a; /* WRONG */
```

are illegal.

To set all of the elements of an array to some value, you must do so one by one, as in the loop example above. To copy the contents of one array to another, you must again do so one by one:

```
int b[10];
for(i = 0; i < 10; i = i + 1)
    b[i] = a[i];
```

Remember that for an array declared

```
int a[10];
```

there is no element `a[10]`; the topmost element is `a[9]`. This is one reason that zero-based loops are also common in C. Note that the `for` loop

```
for(i = 0; i < 10; i = i + 1)
    ...
```

does just what you want in this case: it starts at 0, the number 10 suggests (correctly) that it goes through 10 iterations, but the less-than comparison means that the last trip through the loop has `i` set to 9. (The comparison `i <= 9` would also work, but it would be less clear and therefore poorer style.)

### **Multidimensional Array**

The declaration of an array of arrays looks like this:

```
int a2[5][7];
```

You have to read complicated declarations like these ``inside out.'' What this one says is that `a2` is an array of 5 something's, and that each of the something's is an array of 7 `ints`. More briefly, ```a2` is an array of 5 arrays of 7 `ints`,'' or, ```a2` is an array of array of `int`.'' In the declaration of `a2`, the brackets closest to the identifier `a2` tell you what `a2` first and foremost is. That's how you know it's an array of 5 arrays of size 7, not the other way around. You can think of `a2` as having 5 ``rows'' and 7 ``columns,'' although this interpretation is not mandatory. (You could also treat the ``first'' or inner subscript as ``x'' and the second as ``y.'')

Unless you're doing something fancy, all you have to worry about is that the subscripts when you access the array match those that you used when you declared it, as in the examples below.)

To illustrate the use of multidimensional arrays, we might fill in the elements of the above array `a2` using this piece of code:

```
int i, j;
for(i = 0; i < 5; i = i + 1)
{
    for(j = 0; j < 7; j = j + 1)
        a2[i][j] = 10 * i + j;
}
```

This pair of nested loops sets `a[1][2]` to 12, `a[4][1]` to 41, etc. Since the first dimension of `a2` is 5, the first subscripting index variable, `i`, runs from 0 to 4. Similarly, the second subscript varies from 0 to 6.

We could print `a2` out (in a two-dimensional way, suggesting its structure) with a similar pair of nested loops:

```
for (i = 0; i < 5; i = i + 1)
{
    for (j = 0; j < 7; j = j + 1)
        printf ("%d\t", a2[i][j]);
    printf ("\n");
}
```

(The character `\t` in the `printf` string is the tab character.)

Just to see more clearly what's going on, we could make the ``row'' and ``column'' subscripts explicit by printing them, too:

```
for(j = 0; j < 7; j = j + 1)
    printf("\t% d:", j);
printf ("\n");
for(i = 0; i < 5; i = i + 1)
{
    printf("% d:", i);
    for(j = 0; j < 7; j = j + 1)
```

```

        printf("\t%d", a2[i][j]);
        printf("\n");
    }
}

```

This last fragment would print

	0:	1:	2:	3:	4:	5:	6:
0:	0	1	2	3	4	5	6
1:	10	11	12	13	14	15	16
2:	20	21	22	23	24	25	26
3:	30	31	32	33	34	35	36
4:	40	41	42	43	44	45	46

## **STRING**

String are the combination of number of characters these are used to store any word in any variable or constant. A string is an array of character. It is internally represented in system by using ASCII value. Every single character can have its own ASCII value in the system. A character string is stored in one array of character type.

e.g. “Ram” contains ASCII value per location, when we are using strings and then these strings are always terminated by character ‘\0’. We use conversion specifies %s to set any string we can have any string as follows:-

char NM [25].

When we store any value in nm variable then it can hold only 24 character because at the end of the string one character is consumed automatically by ‘\0’.

### **#include**

There are some common inbuilt functions to manipulation on string in string.h file. these are as follows:

- 1. *strlen*** - string length
- 2. *strcpy*** - string copy
- 3. *strcmp*** - string compare

**4. *syrups* - string upper**

**5. *strlwr* - string lower**

**6. *strcat* - string concatenate**

## **UNIT- 5**

### **Function**

A function is a ``black box'' that we've locked part of our program into. The idea behind a function is that it *compartmentalizes* part of the program, and in particular, that the code within the function has some useful properties:

1. It performs some well-defined task, which will be useful to other parts of the program.
2. It might be useful to other programs as well; that is, we might be able to reuse it (and without having to rewrite it).
3. The rest of the program doesn't have to know the details of how the function is implemented. This can make the rest of the program easier to think about.
4. The function performs its task *well*. It may be written to do a little more than is required by the first program that calls it, with the anticipation that the calling program (or some other program) may later need the extra functionality or improved performance. (It's important that a finished function do its job well, otherwise there might be a reluctance to call it, and it therefore might not achieve the goal of reusability.)
5. By placing the code to perform the useful task into a function, and simply calling the function in the other parts of the program where the task must be performed, the rest of the program becomes clearer: rather than having some large, complicated, difficult-to-understand piece of code repeated wherever the task is being performed, we have a single simple function call, and the name of the function reminds us which task is being performed.
6. Since the rest of the program doesn't have to know the details of how the function is implemented, the rest of the program doesn't care if the function is reimplemented later, in some different way (as long as it continues to perform its same task, of course).

course!). This means that one part of the program can be rewritten, to improve performance or add a new feature (or simply to fix a bug), without having to rewrite the rest of the program.

Functions are probably the most important weapon in our battle against software complexity. You'll want to learn when it's appropriate to break processing out into functions (and also when it's not), and *how* to set up function interfaces to best achieve the qualities mentioned above: re-usability, information hiding, clarity, and maintainability.

So what defines a function? It has a *name* that you call it by, and a list of zero or more *arguments* or *parameters* that you hand to it for it to act on or to direct its work; it has a *body* containing the actual instructions (statements) for carrying out the task the function is supposed to perform; and it may give you back a *return value*, of a particular type.

Here is a very simple function, which accepts one argument, multiplies it by 2, and hands that value back:

```
int multbytwo(int x)
{
    int retrieval;
    retrieval = x * 2;
    return retrieval;
}
```

On the first line we see the return type of the function (`int`), the name of the function (`multbytwo`), and a list of the function's arguments, enclosed in parentheses. Each argument has both a name and a type; `multbytwo` accepts one argument, of type `int`, named `x`. The name `x` is arbitrary, and is used only within the definition of `multbytwo`. The caller of this function only needs to know that a single argument of type `int` is expected; the caller does not need to know what name the function will use internally to refer to that argument. (In particular, the caller does not have to pass the value of a variable named `x`.)

Next we see, surrounded by the familiar braces, the body of the function itself. This function consists of one declaration (of a local variable `retrieval`) and two statements. The first statement is a conventional expression statement, which computes and assigns a value to `retrieval`, and the second statement is a `return` statement, which causes the function to return to its caller, and also specifies the value which the function returns to its caller.

The `return` statement can return the value of any expression, so we don't really need the local `retrieval` variable; the function could be collapsed to-

```
int multbytwo(int x)
```

```
{  
    return x * 2;  
}
```

How do we call a function? We've been doing so informally since day one, but now we have a chance to call one that we've written, in full detail. Here is a tiny skeletal program to call `multby2`:

```
#include  
external int multbytwo(int);  
int main()  
{  
    int i, j;  
    i = 3;  
    j = multbytwo(i);  
    printf("%d\n", j);  
    return 0;  
}
```

This looks much like our other test programs, with the exception of the new line

```
external int multbytwo(int);
```

This is an *external function prototype declaration*. It is an external declaration, in that it declares something which is defined somewhere else. (We've already seen the defining instance of the function `multbytwo`, but maybe the compiler hasn't seen it yet.) The function prototype declaration contains the three pieces of information about the function that a caller needs to know: the function's name, return type, and argument type(s). Since we don't care what name the `multbytwo` function will use to refer to its first argument, we don't need to mention it. (On the other hand, if a function takes several arguments, giving them names in the prototype may make it easier to remember which is which, so names may optionally be used in function prototype declarations.) Finally, to remind us that this is an external declaration and not a defining instance, the prototype is preceded by the keyword `external`.

The presence of the function prototype declaration lets the compiler know that we intend to call this function, `multbytwo`. The information in the prototype lets the compiler generate the correct code for calling the function, and also enables the compiler to check up on our code

(by making sure, for example, that we pass the correct number of arguments to each function we call).

Down in the body of `main`, the action of the function call should be obvious: the line

```
j = multbytwo(i);
```

calls `multbytwo`, passing it the value of `i` as its argument. When `multbytwo` returns, the return value is assigned to the variable `j`. (Notice that the value of `main`'s local variable `i` will become the value of `multbytwo`'s parameter `x`; this is absolutely not a problem, and is a normal sort of affair.)

This example is written out in ``longhand," to make each step equivalent. The variable `i` isn't really needed, since we could just as well call

```
j = multbytwo(3);
```

And the variable `j` isn't really needed, either, since we could just as well call

```
printf("%d\n", multbytwo(3));
```

Here, the call to `multbytwo` is a sub expression which serves as the second argument to `printf`. The value returned by `multbytwo` is passed immediately to `printf`. (Here, as in general, we see the flexibility and generality of expressions in C. An argument passed to a function may be an arbitrarily complex sub expression, and a function call is itself an expression which may be embedded as a sub expression within arbitrarily complicated surrounding expressions.)

We should say a little more about the mechanism by which an argument is passed down from a caller into a function. Formally, C is *call by value*, which means that a function receives *copies* of the values of its arguments. We can illustrate this with an example. Suppose, in our implementation of `multbytwo`, we had gotten rid of the unnecessary `retrieval` variable like this:

```
int multbytwo(int x)
{
    x = x * 2;
    return x;
}
```

### **Recursive Functions**

A recursive function is one which calls itself. This is another complicated idea which you are unlikely to meet frequently. We shall provide some examples to illustrate recursive functions.

Recursive functions are useful in evaluating certain types of mathematical function. You may also encounter certain dynamic data structures such as linked lists or binary trees. Recursion is a very useful way of creating and accessing these structures.

Here is a recursive version of the Fibonacci function. We saw a non recursive version of this earlier.

```
int fib(int num)
/* Fibonacci value of a number */
{
    switch(num) {
        case 0:
            return(0);
            break;
        case 1:
            return(1);
            break;
        default: /* Including recursive calls */
            return(fib(num - 1) + fib(num - 2));
            break;
    }
}
```

We met another function earlier called power. Here is an alternative recursive version.

```
double power(double val, unsigned pow)
{
    if(pow == 0) /* pow(x, 0) returns 1 */
        return(1.0);
    else
        return(power(val, pow - 1) * val);
}
```

Notice that each of these definitions incorporate a test. Where an input value gives a trivial result, it is returned directly; otherwise the function calls itself, passing a changed version of the input values. Care must be taken to define functions which will not call themselves indefinitely, otherwise your program will never finish.

The definition of fib is interesting, because it calls itself twice when recursion is used. Consider the effect on program performance of such a function calculating the Fibonacci function of a moderate size number.

Input Value	Number of times fib is called
0	1
1	1
2	3
3	5
4	9
5	15
6	25
7	41
8	67
9	109
10	177

If such a function is to be called many times, it is likely to have an adverse effect on program performance.

Don't be frightened by the apparent complexity of recursion. Recursive functions are sometimes the simplest answer to a calculation. However there is always an alternative non-recursive solution available too. This will normally involve the use of a loop, and may lack the elegance of the recursive solution.

## **UNIT – 6**

### **Pointer**

a pointer is a variable that points to or references a memory location in which data is stored. In the computer, each memory cell has an address that can be used to access that location so a pointer variable points to a memory location we can access and change the contents of this memory location via the pointer.

#### **Pointer declaration:**

A pointer is a variable that contains the memory location of another variable in which data is stored. Using pointer, you start by specifying the type of data stored in the location. The asterisk helps to tell the compiler that you are creating a pointer variable. Finally you have to give the name of the variable. The syntax is as shown below.

```
type * variable name
```

#### **The following example illustrate the declaration of pointer variable :**

```
int *ptr;  
float *string;
```

#### **Address operator:**

Once we declare a pointer variable then we must point it to something we can do this by assigning to the pointer the address of the variable you want to point as in the following example:

```
ptr=#
```

The above code tells that the address where num is stores into the variable ptr. The variable ptr has the value 21260, if num is stored in memory 21260 address then

**The following program illustrate the pointer declaration :**

```
/* A program to illustrate pointer declaration*/

main()
{
int
int
sum=45;
ptr=&ptr;
printf      ("\\n      Sum      is      %d\\n",      sum);
printf      ("\\n      The      sum      pointer      is      %d",      ptr);
}
```

**Pointer expressions & pointer arithmetic:**

In expressions, like other variables pointer variables can be used. For example if p1 and p2 are properly initialized and declared pointers, then the following statements are valid.

```
y= *p1**p2;
sum=sum+*p1;
z=           5*           -           *p2/p1;
*p2= *p2 + 10;
```

C allows us to subtract integers to or add integers from pointers as well as to subtract one pointer from the other. We can also use short hand operators with pointers p1+=; sum+=\*p2; etc., By using relational operators, we can also compare pointers like the expressions such as p1 >p2 , p1==p2 and p1!=p2 are allowed.

**The following program illustrate the pointer expression and pointer arithmetic :**

```
/*Program to illustrate the pointer expression and pointer arithmetic*/
#include<stdio.h>
main()
{
    int ptr1,ptr2;
    a,b,x,y,z;
    a=30;b=6;
    ptr1=&a;
    ptr2=&b;
    x=*(ptr1+6);
    y=6-*ptr1;
    printf("\nAddress of a = %u",ptr1);
    printf("\nAddress of b = %u",ptr2);
    printf("\na=%d,b=%d",a,b);
    printf("\nx=%d,y=%d",x,y);
    ptr1=ptr1+70;
    ptr2=ptr2;
    printf("\na=%d,b=%d",a,b);
}
```

### **Pointers and function:**

In a function declaration, the pointer are very much used . Sometimes, only with a pointer a complex function can be easily represented and success. In a function definition, the usage of the pointers may be classified into two groups.

- 1. Call by reference**
- 2. Call by value.**

### Call by value:

We have seen that there will be a link established between the formal and actual parameters when a function is invoked. As soon as temporary storage is created where the value of actual parameters is stored. The formal parameters picks up its value from storage area the mechanism of data transfer between formal and actual parameters allows the actual parameters mechanism of data transfer is referred as call by value. The corresponding formal parameter always represents a local variable in the called function. The current value of the corresponding actual parameter becomes the initial value of formal parameter. In the body of the actual parameter, the value of formal parameter may be changed. In the body of the subprogram, the value of formal parameter may be changed by assignment or input statements. This will not change the value of the actual parameters.

```
/* Include< stdio.h >

void main()
{
int x,y;
x=20;
y=30;
printf("\n Value of a and b before function call =%d %d",a,b);
fncn(x,y);
printf("\n Value of a and b after function call =%d %d",a,b);
}

fncn(p,q)
int p,q;
{
p=p+p;
q=q+q;
}
```

### Call by Reference:

The address should be pointers, when we pass address to a function the parameters receiving. By using pointers, the process of calling a function to pass the address of the variable is known as call by reference. The function which is called by reference can change the value of the variable used in the call.

```
/* example of call by reference*/

/*
Include<stdio.h>
main()
{
int x,y;
x=20;
y=30;
printf("\n Value of a and b before function call =%d %d",a,b);
fncn(&x,&y); printf("\n Value of a and b after function call =%d %d",a,b);
}
fncn(p,q)
int p,q;
{
*p=*p+*p;
*q=*q+*q;
}
```

### Pointer to arrays:

an array is actually very much similar like pointer. We can declare as int \*a is an address, because a[0] the arrays first element as a[0] and \*a is also an address the form of declaration is also equivalent. The difference is pointer can appear on the left of the assignment operator and it is a is a variable that is value. The array name cannot appear as the left side of assignment operator and is constant.

```
/* A program to display the contents of array using pointer*/
main()
{
int a[100];
int i,j,n;
```

```

printf("\nEnter          the      elements      of      the      array\n");
scanf(%d,&n);
printf("Enter          the      array      elements");
for(l=0;l<
scanf(%d,&a[l]);
printf("Array          element      are");
for(ptr=a,ptr<
printf("Value      of      a[%d]=%d      stored      at      address      %u",j+=,*ptr,ptr);
}

```

### Pointers and structures :

We know the name of an array stands for address of its zeros element the same concept applies for names of arrays of structures. Suppose item is an array variable of the struct type. Consider the following declaration:

```

structure                                products
{
char                                         name[30];
int                                         manufacture;
float                                        net;
item[2],*ptr;

```

## UNIT - 7

### STRUCTURES

#### What is a Structure?

- Structure is a method of packing the data of different types.
- When we require using a collection of different data items of different data types in that situation we can use a structure.
- A structure is used as a method of handling a group of related data items of different data types.

A structure is a collection of variables under a single name. These variables can be of different types, and each has a name which is used to select it from the structure. A structure is a convenient way of grouping several pieces of related information together.

A structure can be defined as a new named type, thus extending the number of available types. It can use other structures, arrays or pointers as some of its members, though this can get complicated unless you are careful.

### **Defining a Structure**

A structure type is usually defined near to the start of a file using a `typedef` statement. `typedef` defines and names a new type, allowing its use throughout the program. `typedefs` usually occur just after the `#define` and `#include` statements in a file.

Here is an example structure definition.

```
type def structure {
    char name[64];
    char course[128];
    int age;
    int year;
} student;
```

This defines a new type `student` variables of type `student` can be declared as follows.

```
student st_rec;
```

Notice how similar this is to declaring an `int` or `float`.

The variable name is `st_rec`, it has members called `name`, `course`, `age` and `year`.

### **Accessing Members of a Structure**

Each member of a structure can be used just like a normal variable, but its name will be a bit longer. To return to the examples above, member name of structure `st_rec` will behave just like a normal array of `char`, however we refer to it by the name .

```
st_rec.name
```

Here the dot is an operator which selects a member from a structure.

Where we have a pointer to a structure we could deference the pointer and then use dot as a member selector. This method is a little clumsy to type. Since selecting a member from a structure pointer happens frequently, it has its own operator `->` which acts as follows. Assume that `st_ptr` is a pointer to a structure of type `student` We would refer to the `name` member as.

```
st_ptr -> name
```

```
/* Example program for using a structure*/
#include<stdio.h>
void main()
{
    int id_no;
    char name[20];
    char address[20];
    char combination[3];
    int age;
    struct student;
    information;
    printf("Enter the student information");
    printf("Now Enter the student id_no");
    scanf("%d",&new_id_no);
    printf("Enter the student name");
    scanf("%s",&new_student.name);
    printf("Enter the student address");
    scanf("%s",&new_student.address);
    printf("Enter the student combination");
    scanf("%s",&new_student.combination);
    printf("Enter the student age");
    scanf("%d",&new_student.age);
    printf("Student information\n");
    printf("student id_number=%d\n",new_id_no);
    printf("student name=%s\n",new_student.name);
    printf("student Address=%s\n",new_student.address);
    printf("student combination=%s\n",new_student.combination);
    printf("student age=%d\n",new_student.age);
}
```

### **Arrays of structure:**

It is possible to define a array of structures for example if we are maintaining information of all the students in the college and if 100 students are studying in the college. We need to use an array than single variables. We can define an array of structures as shown in the following example:

```
structure information
{
```

```

int id_no;
char name[20];
char address[20];
char combination[3];
int age;
}
student[100];

```

An array of structures can be assigned initial values just as any other array can. Remember that each element is a structure that must be assigned corresponding initial values as illustrated below.

```

#include< stdio.h >
{
    structure info
{
    int id_no;
    char name[20];
    char address[20];
    char combination[3];
    int age;
}
    structure info std[100];
int l,n;
printf("Enter the number of students");
scanf("%d",&n);
printf(" Enter Id_no,name address combination age\b");
for(l=0;l<n;l++)
scanf("%d%s%s%s%d",&std[l].id_no,std[l].name,std[l].address,std[l].combination,&std[l].age);
printf("\n Student information");
for (l=0;l<n;l++)
printf("%d%s%s%s%d\b",
",std[l].id_no,std[l].name,std[l].address,std[l].combination,std[l].age);
}

```

### Structure within a structure:

A structure may be defined as a member of another structure. In such structures the declaration of the embedded structure must appear before the declarations of other

structures.

```

        structure           date
{
int                                day;
int                                month;
int                               year;
};

        structure          student
{
int                                id_no;
char                               name[20];
char                               address[20];
char                               combination[3];
int                                age;
structure                         date      def;
structure                         date      doa;
}oldstudent, newstudent;

```

the structure student contains another structure date as one of its members.

## **UNIT- 8**

### **Union:**

Unions like structure contain members whose individual data types may differ from one another. However the members that compose a union all share the same storage area within the computer's memory whereas each member within a structure is assigned its own unique storage area. Thus unions are used to observe memory. They are useful for applications involving multiple members. Where values need not be assigned to all the members at any one time. Like structures union can be declared using the keyword union as follows:

```

union           item
{
int             m;
float           p;
char            c;
}
code;

```

this declares a variable code of type union item. The union contains three members each with a different data type. However we can use only one of them at a time. This is because if only

one location is allocated for union variable irrespective of size. The compiler allocates a piece of storage that is large enough to access a union member we can use the same syntax that we use to access structure members. That is

code.m  
code.p  
code.c

are all valid member variables. During accessing we should make sure that we are accessing the member whose value is currently stored.

For example a statement such as -

```
code.m=456;  
code.p=456.78;  
printf("%d",code.m);
```

Would produce erroneous result..

## Enum declarations

There are two kinds of `enum` type declarations. One kind creates a named type, as in

```
enum MyEnumType { ALPHA, BETA, GAMMA };
```

If you give an `enum` type a name, you can use that type for variables, function arguments and return values, and so on:

```
enum MyEnumType x; /* legal in both C and C++ */  
MyEnumType y; // legal only in C++
```

The other kind creates an unnamed type. This is used when you want names for constants but don't plan to use the type to declare variables, function arguments, etc. For example, you can write

```
enum { HOMER, MARGE, BART, LISA, MAGGIE };
```

## Values of enum constants

If you don't specify values for `enum` constants, the values start at zero and increase by one with each move down the list. For example, given

```
enum MyEnumType { ALPHA, BETA, GAMMA };
```

`ALPHA` has a value of 0, `BETA` has a value of 1, and `GAMMA` has a value of 2.

If you want, you may provide explicit values for `enum` constants, as in `enum Foo Size { SMALL = 10, MEDIUM = 100, LARGE = 1000 };`

There is an implicit conversion from any `enum` type to `int`. Suppose this type exists:

```
enum MyEnumType { ALPHA, BETA, GAMMA };
```

Then the following lines are legal:

```
int i = BETA; // give i a value of 1  
int j = 3 + GAMMA; // give j a value of 5
```

On the other hand, there is *not* an implicit conversion from `int` to an `enum` type:

```
MyEnumType x = 2; // should NOT be allowed by compiler  
MyEnumType y = 123; // should NOT be allowed by compiler
```

Note that it doesn't matter whether the `int` matches one of the constants of the `enum` type; the type conversion is always illegal

## **TypeDefs**

A type def in C is a declaration. Its purpose is to create new types from existing types; whereas a variable declaration creates new memory locations. Since a type def is a declaration, it can be intermingled with variable declarations, although common practice would be to state `typedefs` first, then variable declarations. A nice programming convention is to capitalize the first letter of a user-defined type to distinguish it from the built-in types, which all have lower-case names. Also, `typedefs` are usually global declarations.

Example: Use a Type def To Create A Synonym for a Type Name

```
type def int Integer; //Integer can now be used in place of int
```

```
int a,b,c,d; //4 variables of type int
```

```
Integer e,f,g,h; //the same thing
```

In general, a type def should never be used to assign a different name to a built-in type name; it just confuses the reader. Usually, a type def associates a type name with a more complicated type specification, such as an array. A type def should always be used in situations where the same type definition is used more than once for the same purpose. For example, a vector of 20 elements might represent different aspects of a scientific measurement.

Example: Use a Type def To Create A Synonym for an Array Type

```
type def int Vector[20]; //20 integers
```

```
Vector a,b;
```

```
int a[20], b[20]; //the same thing, but a type def is preferred
```

## TypeDefs for Enumerated Types

Every type has constants. For the "int" type, the constants are 1,2,3,4,5; for "char", 'a','b','c'. When a type has constants that have names, like the colors of the rainbow, that type is called an **enumerated type**. Use an enumerated type for computer representation of common objects that have names like Colors, Playing Cards, Animals, Birds, Fish etc. Enumerated type constants (since they are names) make a program easy to read and understand.

We know that all names in a computer usually are associated with a number. Thus, all of the names (RED, BLUE, GREEN) for an enumerated type are "encoded" with numbers. In eC, if you define an enumerated type, like Color, you cannot add it to an integer; it is not type compatible. In standard C++, anything goes. Also, in eC an enumerated type must always be declared in a **typedef** before use (in fact, all new types must be declared before use).

Example: Use a **Typedef** To Create An Enumerated Type

```
typedef enum {RED, BLUE, GREEN} Color;
```

```
Color a,b;
```

```
a = RED; //NOT ALLOWED in eC
```

```
if ((a == BLUE) || (a==b)) cout<<"great";
```

Notice that an enumerated type is a code that associates symbols and numbers. The char type can be thought of as an enumeration of character codes. The default code for an enumerated type assigns the first name to the value 0 (RED), second name 1 (BLUE), third 2 (GREEN) etc. The user can, however, override any, or all, of the default codes by specifying alternative values.

*linked list* is a chain of structure or records called nodes. Each node has at least two members, one of which points to the next item or node in the list! These are defined as *Single Linked Lists* because they only point to the next item, and not the previous. Those that do point to both are called *Doubly Linked Lists* or *Circular Linked Lists*. Please note that there is a distinct difference between Double Linked lists and Circular Linked lists. I won't go into any depth on

it because it doesn't concern this tutorial. According to this definition, we could have our record hold **anything** we wanted! The only drawback is that each record must be an instance of the same structure. This means that we couldn't have a record with a char pointing to another structure holding a short, a char array, and a long. Again, they have to be instances of the same structure for this to work. Another cool aspect is that each structure can be located anywhere in memory, each node doesn't have to be linear in memory!

a **linked list** is data structure that consists of a sequence of data records such that in each record there is a field that contains a reference (i.e., a *link*) to the next record in the sequence. A linked list whose nodes contain two fields: an integer value and a link to the next node.

Linked lists are among the simplest and most common data structures, and are used to implement many important abstract data structures, such as stacks, queues, hash tables, symbolic expressions, skip lists, and many more.

The principal benefit 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. For that reason, linked lists allow insertion and removal of nodes at any point in the list, with a constant number of operations.

On the other hand, linked lists by themselves do not allow random access to the data, or any form of efficient indexing. Thus, many basic operations — such as obtaining the last node of the list, or finding a node that contains a given data, or locating the place where a new node should be inserted — may require scanning most of the list elements.

### **What Linked Lists Look Like**

An array allocates memory for all its elements lumped together as one block of memory.

In contrast, a linked list allocates space for each element separately in its own block of memory called a "linked list element" or "node". The list gets its overall structure by using pointers to connect all its nodes together like the links in a chain.

Each node contains two fields: a "data" field to store whatever element type the list holds for its client, and a "next" field which is a pointer used to link one node to the next node.

Each node is allocated in the heap with a call to `malloc()`, so the node memory continues to exist until it is explicitly deallocated with a call to `free()`. The front of the list is a 5

pointer to the first node. Here is what a list containing the numbers 1, 2, and 3 might look

like...

Build One,Two,Three()

This drawing shows the list built in memory by the function BuildOneTwoThree() (the full source code for this function is below). The beginning of the linked list is stored in a "head" pointer which points to the first node. The first node contains a pointer to the second node. The second node contains a pointer to the third node, ... and so on. The last node in the list has its .next field set to NULL to mark the end of the list. Code can access any node in the list by starting at the head and following the .next pointers. Operations towards the front of the list are fast while operations which access node farther down the list take longer the further they are from the front. This "linear" cost to access a node is fundamentally more costly than the constant time [ ] access provided by arrays. In this respect, linked lists are definitely less efficient than arrays.

Drawings such as above are important for thinking about pointer code, so most of the examples in this article will associate code with its memory drawing to emphasize the habit. In this case the head pointer is an ordinary local pointer variable, so it is drawn separately on the left to show that it is in the stack. The list nodes are drawn on the right to show that they are allocated in the heap.

### **The Empty List — NULL**

The above is a list pointed to by head is described as being of "length three" since it is made of three nodes with the .next field of the last node set to NULL. There needs to be some representation of the empty list — the list with zero nodes. The most common representation chosen for the empty list is a NULL head pointer. The empty list case is the one common weird "boundary case" for linked list code. All of the code presented in this article works correctly for the empty list case, but that was not without some effort.

When working on linked list code, it's a good habit to remember to check the empty list case to verify that it works too. Sometimes the empty list case works the same as all the cases, but sometimes it requires some special case code. No matter what, it's a good case to at least think about.

### **Linked List Types: Node and Pointer**

Before writing the code to build the above list, we need two data types...

- *Node* The type for the nodes which will make up the body of the list.

These are allocated in the heap. Each node contains a single client data element and a pointer to the next node in the list. Type: struct node

```
struct node {
```

```
    int data;
```

```
    structure node* next;
```

```
};
```

- *Node Pointer* The type for pointers to nodes. This will be the type of the head pointer and the .next fields inside each node. In C and C++, no separate type declaration is required since the pointer type is just the node type followed by a '\*'. Type: structure node\*

### **BuildOneTwoThree() Function**

Here is simple function which uses pointer operations to build the list {1, 2, 3}. The memory drawing above corresponds to the state of memory at the end of this function. This function demonstrates how calls to malloc() and pointer assignments (=) work to build a pointer structure in the heap.

```
/*
```

Build the list {1, 2, 3} in the heap and store its head pointer in a local stack variable.

Returns the head pointer to the caller.

```
*/
```

```
struct node* BuildOneTwoThree() {
```

```
struct node* head = NULL;  
  
struct node* second = NULL;  
  
struct node* third = NULL;  
  
head = malloc(sizeof(struct node)); // allocate 3 nodes in the heap  
  
second = malloc(sizeof(struct node));  
  
third = malloc(sizeof(struct node));  
  
head->data = 1; // setup first node  
  
head->next = second; // note: pointer assignment rule  
  
second->data = 2; // setup second node  
  
second->next = third;  
  
third->data = 3; // setup third link  
  
third->next = NULL;  
  
// At this point, the linked list referenced by "head"  
// matches the list in the drawing.  
  
return head;  
  
}
```

### **Basic concepts and nomenclature**

Each record of a linked list is often called an **element** or **node**.

The field of each node that contains address of the next node is usually called the **next link** or **next pointer**. The remaining fields may be called the **data**, **information**, **value**, or **payload** fields.

The **head** of a list is its first node, and the **tail** is the list minus that node (or a pointer thereto). In Lisp and some derived languages, the tail may be called the **CDR** (pronounced *could-R*) of the list, while the payload of the head node may be called the

### Linear and circular lists

In the last node of a list, the link field often contains a **null** reference, a special value that is interpreted by programs as meaning "there is no such node". A less common convention is to make it point to the first node of the list; in that case the list is said to be **circular** or **circularly linked**; otherwise it is said to be **open** or **linear**.

### Simply-, doubly-, and multiply-linked lists

In a **doubly-linked list**, each node contains, besides the next-node link, a second link field pointing to the *previous* node in the sequence. The two links may be called **forward(s)** and **backwards**. Linked lists that lack such pointers are said to be **simply linked**, or **simple linked lists**.

A doubly-linked list whose nodes contain three fields: an integer value, the link forward to the next node, and the link backward to the previous node

The technique known as **XOR-linking** allows a doubly-linked list to be implemented using a single link field in each node. However, this technique requires the ability to do bit operations on addresses, and therefore may not be available in some high-level languages.

In a **multiply-linked list**, each node contains two or more link fields, each field being used to connect the same set of data records in a different order (e.g., by name, by department, by date of birth, etc.). (While doubly-linked lists can be seen as special cases of multiply-linked list, the fact that the two orders are opposite to each other leads to simpler and more efficient algorithms, so they are usually treated as a separate case.)

### Linked lists vs. arrays

	Array	Linked list
Indexing	$\Theta(1)$	$\Theta(n)$
Time complexity of indexing operation		
Space complexity of storage		

Inserting / Deleting at end	$\Theta(1)$	$\Theta(1)$ or $\Theta(n)$
Inserting / Deleting in middle (with iterator)	$\Theta(n)$	$\Theta(1)$
<u>Persistent</u>	No	Singly yes
<u>Locality</u>	Great	Poor

Linked lists have several advantages over arrays. Insertion of an element at a specific point of a list is a constant-time operation, whereas insertion in an array may require moving half of the elements, or more. While one can "delete" an element from an array in constant time by somehow marking its slot as "vacant", an algorithm that iterates over the elements may have to skip a large number of vacant slots.

Moreover, arbitrarily many elements may be inserted into a linked list, limited only by the total memory available; while an array will eventually fill up, and then have to be resized — an expensive operation, that may not even be possible if memory is fragmented. Similarly, an array from which many elements are removed may have to be resized in order to avoid wasting too much space.

On the other hand, arrays allow random access, while linked lists allow only sequential access to elements. Singly-linked lists, in fact, can only be traversed in one direction. This makes linked lists unsuitable for applications where it's useful to look up an element by its index quickly, such as heap sort. Sequential access on arrays is also faster than on linked lists on many machines, because they have greater locality of reference and thus profit more from processor caching.

Another disadvantage of linked lists is the extra storage needed for references, which often makes them impractical for lists of small data items such as characters or Boolean values. It can also be slow, and with a naive allocator, wasteful, to allocate memory separately for each new element, a problem generally solved using memory pools.

Some hybrid solutions try to combine the advantages of the two representations. Unrolled linked lists store several elements in each list node, increasing cache performance while decreasing memory overhead for references. CDR coding does both these as well, by replacing references with the actual data referenced, which extends off the end of the referencing record.

A good example that highlights the pros and cons of using arrays vs. linked lists is by implementing a program that resolves the Josephus problem. The Josephus problem is an election method that works by having a group of people stand in a circle. Starting at a predetermined person, you count around the circle  $n$  times. Once you reach the  $n$ th person, take them out of the circle and have the members close the circle. Then count around the circle the same  $n$  times and repeat the process, until only one person is left. That person wins the election. This shows the strengths and weaknesses of a linked list vs. an array, because if you view the people as connected nodes in a circular linked list then it shows how easily the linked list is able to delete nodes (as it only has to rearrange the links to the different nodes). However, the linked list will be poor at finding the next person to remove and will need to recurse through the list until it finds that person. An array, on the other hand, will be poor at deleting nodes (or elements) as it cannot remove one node without individually shifting all the elements up the list by one. However, it is exceptionally easy to find the  $n$ th person in the circle by directly referencing them by their position in the array.

The list ranking problem concerns the efficient conversion of a linked list representation into an array. Although trivial for a conventional computer, solving this problem by a parallel algorithm is complicated and has been the subject of much research.

### **Simply-linked linear lists vs. other lists**

While doubly-linked and/or circular lists have advantages over simply-linked linear lists, linear lists offer some advantages that make them preferable in some situations.

For one thing, a simply-linked linear list is a recursive data structure, because it contains a pointer to a *smaller* object of the same type. For that reason, many operations on simply-linked linear lists (such as merging two lists, or enumerating the elements in reverse order) often have very simple recursive algorithms, much simpler than any solution using iterative commands. While one can adapt those recursive solutions for doubly-linked and circularly-linked lists, the procedures generally need extra arguments and more complicated base cases.

Linear simply-linked lists also allow tail-sharing, the use of a common final portion of sub-list as the terminal portion of two different lists. In particular, if a new node is added at the beginning of a list, the former list remains available as the tail of the new one — a simple example of a persistent data structure. Again, this is not true with the other variants: a node may never belong to two different circular or doubly-linked lists.

In particular, end-sentinel nodes can be shared among simply-linked non-circular lists. One may even use the same end-sentinel node for *every* such list. In Lisp, for example, every proper

list ends with a link to a special node, denoted by `nil` or `()`, whose `CAR` and `CDR` links point to itself. Thus a Lisp procedure can safely take the `CAR` or `CDR` of *any* list.

Indeed, the advantages of the fancy variants are often limited to the complexity of the algorithms, not in their efficiency. A circular list, in particular, can usually be emulated by a linear list together with two variables that point to the first and last nodes, at no extra cost.

### **Doubly-linked vs. singly-linked**

Double-linked lists require more space per node (unless one uses xor-linking), and their elementary operations are more expensive; but they are often easier to manipulate because they allow sequential access to the list in both directions. In particular, one can insert or delete a node in a constant number of operations given only that node's address. To do the same in a singly-linked list, one must have the *previous* node's address. Some algorithms require access in both directions. On the other hand, they do not allow tail-sharing, and cannot be used as persistent data structures.

### **Circularly-linked vs. linearly-linked**

A circularly linked list may be a natural option to represent arrays that are naturally circular, e.g. for the corners of a polygon, for a pool of buffers that are used and released in FIFO order, or for a set of processes that should be time-shared in round-robin order. In these applications, a pointer to any node serves as a handle to the whole list.

With a circular list, a pointer to the last node gives easy access also to the first node, by following one link. Thus, in applications that require access to both ends of the list (e.g., in the implementation of a queue), a circular structure allows one to handle the structure by a single pointer, instead of two.

A circular list can be split into two circular lists, in constant time, by giving the addresses of the last node of each piece. The operation consists in swapping the contents of the link fields of those two nodes. Applying the same operation to any two nodes nodes in two distinct lists joins the two list into one. This property greatly simplifies some algorithms and data structures, such as the quad-edge and face-edge.

The simplest representation for an empty circular list (when such thing makes sense) has no nodes, and is represented by a null pointer. With this choice, many algorithms have to test for this special case, and handle it separately. By contrast, the use of null to denote an empty linear list is more natural and often creates fewer special cases.

## Using sentinel nodes

Sentinel node may simplify certain list operations, by ensuring that the next and/or previous nodes exist for every element, and that even empty lists have at least one node. One may also use a sentinel node at the end of the list, with an appropriate data field, to eliminate some end-of-list tests. For example, when scanning the list looking for a node with a given value  $x$ , setting the sentinel's data field to  $x$  makes it unnecessary to test for end-of-list inside the loop. Another example is the merging two sorted lists: if their sentinels have data fields set to  $+\infty$ , the choice of the next output node does not need special handling for empty lists.

However, sentinel nodes use up extra space (especially in applications that use many short lists), and they may complicate other operations (such as the creation of a new empty list).

However, if the circular list is used merely to simulate a linear list, one may avoid some of this complexity by adding a single sentinel node to every list, between the last and the first data nodes. With this convention, an empty list consists of the sentinel node alone, pointing to itself via the next-node link. The list handle should then be a pointer to the last data node, before the sentinel, if the list is not empty; or to the sentinel itself, if the list is empty.

The same trick can be used to simplify the handling of a doubly-linked linear list, by turning it into a circular doubly-linked list with a single sentinel node. However, in this case, the handle should be a single pointer to the dummy node itself.

## Linked list operations

When manipulating linked lists in-place, care must be taken to not use values that you have invalidated in previous assignments. This makes algorithms for inserting or deleting linked list nodes somewhat subtle. This section gives pseudo code for adding or removing nodes from singly, doubly, and circularly linked lists in-place. Throughout we will use *null* to refer to an end-of-list marker or sentinel, which may be implemented in a number of ways.

### Linearly-linked lists

#### Singly-linked lists

Our node data structure will have two fields. We also keep a variable *first Node* which always points to the first node in the list, or is *null* for an empty list.

```
record Node {  
    data // The data being stored in the node  
    next // A reference to the next node, null for last node
```

```

}

record List {
    Node first Node //points to first node of list; null for empty 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:

```

node := list.first Node
while node not null {
    (do something with node.data)
    node := node.next
}

```

The following code inserts a node after an existing node in a singly linked list. The diagram shows how it works. Inserting a node before an existing one cannot be done; instead, you have to locate it while keeping track of the previous node.

```

function insert After(Node node, Node new Node) { //insert new Node after node

    newNode.next := node.next
    node.next := newNode
}

```

Inserting at the beginning of the list requires a separate function. This requires updating *first Node*.

```

function insert Beginning(List list, Node newNode) { //insert node before current first
node
    newNode.next := list.first Node
    list.first Node := newNode
}

```

Similarly, we have functions for removing the node *after* a given node, and for removing a node from the beginning of the list. The diagram demonstrates the former. To find and remove a particular node, one must again keep track of the previous element.

```

function remove After(node node) { //remove node past this one

    obsolete Node := node.next
}

```

```

node.next := node.next.next
destroy obsolete Node
}
function remove Beginning(List list) { // remove first node
    obsolete Node := list.first Node
    list.first Node := list.first Node.next      // point past deleted node
    destroy obsolete Node
}

```

Notice that `remove Beginning()` sets `list.first Node` to `null` when removing the last node in the list.

Since we can't iterate backwards, efficient "insertBefore" or "removeBefore" operations are not possible.

Appending one linked list to another can be inefficient unless a reference to the tail is kept as part of the `List` structure, because we must traverse the entire first list in order to find the tail, and then append the second list to this. Thus, if two linearly-linked lists are each of length  $n$ , list appending has asymptotic time complexity of  $O(n)$ . In the Lisp family of languages, list appending is provided by the append procedure.

Many of the special cases of linked list operations can be eliminated by including a dummy element at the front of the list. This ensures that there are no special cases for the beginning of the list and renders both `insertBeginning()` and `removeBeginning()` unnecessary. In this case, the first useful data in the list will be found at `list.firstNode.next`.

### Circularly-linked list

In a circularly linked list, all nodes are linked in a continuous circle, without using `null`. For lists with a front and a back (such as a queue), one stores a reference to the last node in the list. The `next` node after the last node is the first node. Elements can be added to the back of the list and removed from the front in constant time.

Circularly-linked lists can be either singly or doubly linked.

Both types of circularly-linked lists benefit from the ability to traverse the full list beginning at any given node. This often allows us to avoid storing `firstNode` and `lastNode`, although if the list may be empty we need a special representation for the empty list, such as a `lastNode` variable which points to some node in the list or is `null` if it's empty; we use such a `lastNode`

here. This representation significantly simplifies adding and removing nodes with a non-empty list, but empty lists are then a special case.

### **circularly-linked lists**

Assuming that *someNode* is some node in a non-empty circular simply-linked list, this code iterates through that list starting with *someNode*:

```
function iterate(someNode)
if someNode ≠ null
    node := someNode
    do
        do something with node.value
        node := node.next
    while node ≠ someNode
```

Notice that the test "**while** node ≠ someNode" must be at the end of the loop. If it were replaced by the test "" at the beginning of the loop, the procedure would fail whenever the list had only one node.

This function inserts a node "newNode" into a circular linked list after a given node "node". If "node" is null, it assumes that the list is empty.

```
function insertAfter(Node node, Node newNode)
    if node = null
        newNode.next := newNode
    else
        newNode.next := node.next
        node.next := newNode
```

Suppose that "L" is a variable pointing to the last node of a circular linked list (or null if the list is empty). To append "newNode" to the *end* of the list, one may do

```
insertAfter(L, newNode)
L = newNode
```

To insert "newNode" at the *beginning* of the list, one may do

```
insertAfter(L, newNode)
if L = null
```

L = newNode

### Linked lists using arrays of nodes

Languages that do not support any type of reference can still create links by replacing pointers with array indices. The approach is to keep an array of records, where each record has integer fields indicating the index of the next (and possibly previous) node in the array. Not all nodes in the array need be used. If records are not supported as well, parallel arrays can often be used instead.

As an example, consider the following linked list record that uses arrays instead of pointers:

```
record Entry {
    integer next; // index of next entry in array
    integer prev; // previous entry (if double-linked)
    string name;
    real balance;
}
```

By creating an array of these structures, and an integer variable to store the index of the first element, a linked list can be built:

```
integer listHead;
Entry Records[1000];
```

Links between elements are formed by placing the array index of the next (or previous) cell into the Next or Prev field within a given element. For example:

Index	Next	Prev	Name	Balance
0	1	4	Jones, John	123.45
1	-1	0	Smith, Joseph	234.56
2 (listHead)	4	-1	Adams, Adam	0.00
3			Ignore, Ignatius	999.99
4	0	2	Another, Anita	876.54
5				
6				
7				

In the above example, `ListHead` would be set to 2, the location of the first entry in the list. Notice that entry 3 and 5 through 7 are not part of the list. These cells are available for any additions to the list. By creating a `ListFree` integer variable, a free list could be created to keep track of what cells are available. If all entries are in use, the size of the array would have to be increased or some elements would have to be deleted before new entries could be stored in the list.

The following code would traverse the list and display names and account balance:

```
i := listHead;
while i >= 0 { // loop through the list
    print i, Records[i].name, Records[i].balance // print entry
    i = Records[i].next;
}
```

When faced with a choice, the advantages of this approach include:

- The linked list is relocatable, meaning it can be moved about in memory at will, and it can also be quickly and directly serialized for storage on disk or transfer over a network.
- Especially for a small list, array indexes can occupy significantly less space than a full pointer on many architectures.
- Locality of reference can be improved by keeping the nodes together in memory and by periodically rearranging them, although this can also be done in a general store.
- Naïve dynamic memory allocators can produce an excessive amount of overhead storage for each node allocated; almost no allocation overhead is incurred per node in this approach.
- Seizing an entry from a pre-allocated array is faster than using dynamic memory allocation for each node, since dynamic memory allocation typically requires a search for a free memory block of the desired size.

This approach has one main disadvantage, however: it creates and manages a private memory space for its nodes. This leads to the following issues:

- It increase complexity of the implementation.
- Growing a large array when it is full may be difficult or impossible, whereas finding space for a new linked list node in a large, general memory pool may be easier.
- Adding elements to a dynamic array will occasionally (when it is full) unexpectedly take linear ( $O(n)$ ) instead of constant time (although it's still an amortized constant).
- Using a general memory pool leaves more memory for other data if the list is smaller than expected or if many nodes are freed.

For these reasons, this approach is mainly used for languages that do not support dynamic memory allocation. These disadvantages are also mitigated if the maximum size of the list is known at the time the array is created.

### Language support

Many programming languages such as Lisp and Scheme have singly linked lists built in. In many functional languages, these lists are constructed from nodes, each called a cons or cons cell. The cons has two fields: the car, a reference to the data for that node, and the cdr, a reference to the next node. Although cons cells can be used to build other data structures, this is their primary purpose.

In languages that support Abstract data types or templates, linked list ADTs or templates are available for building linked lists. In other languages, linked lists are typically built using references together with records. Here is a complete example in C:

```
#include /* for printf */
#include /* for malloc */

struct node
{
    int data;
    struct node *next; /* pointer to next element in list */
};

struct node *list_add(struct node **p, int i)
{
    struct node *n = malloc(sizeof(struct node));
    if (n == NULL)
        return NULL;

    n->next = *p; /* the previous element (*p) now becomes the "next" element */
    *p = n; /* add new empty element to the front (head) of the list */
    n->data = i;

    return *p;
}
void list_remove(struct node **p) /* remove head */
```

```

{
    if (*p != NULL)
    {
        struct node *n = *p;
        *p = (*p)->next;
        free(n);
    }
}

struct node **list_search(struct node **n, int i)
{
    while (*n != NULL)
    {
        if ((*n)->data == i)
        {
            return n;
        }
        n = &(*n)->next;
    }
    return NULL;
}
void list_print(struct node *n)
{
    if (n == NULL)
    {
        printf("list is empty\n");
    }
    while (n != NULL)
    {
        printf("print %p %p %d\n", n, n->next, n->data);
        n = n->next;
    }
}

int main(void)
{
    struct node *n = NULL;
    list_add(&n, 0); /* list: 0 */
    list_add(&n, 1); /* list: 1 0 */
}

```

```
list_add(&n, 2); /* list: 2 1 0 */
list_add(&n, 3); /* list: 3 2 1 0 */
list_add(&n, 4); /* list: 4 3 2 1 0 */
list_print(n);
list_remove(&n);           /* remove first (4) */
list_remove(&n->next);    /* remove new second (2) */
list_remove(list_search(&n, 1)); /* remove cell containing 1 (first) */
list_remove(&n->next);    /* remove second to last node (0) */
list_remove(&n);           /* remove last (3) */
list_print(n);
return 0;
```

## **FILE MANAGEMENT**

### **What is a File?**

Abstractly, a file is a collection of bytes stored on a secondary storage device, which is generally a disk of some kind. The collection of bytes may be interpreted, for example, as characters, words, lines, paragraphs and pages from a textual document; fields and records belonging to a database; or pixels from a graphical image. The meaning attached to a particular file is determined entirely by the data structures and operations used by a program to process the file. It is conceivable (and it sometimes happens) that a graphics file will be read and displayed by a program designed to process textual data. The result is that no meaningful output occurs (probably) and this is to be expected. A file is simply a machine decipherable storage media where programs and data are stored for machine usage.

Essentially there are two kinds of files that programmers deal with text files and binary files. These two classes of files will be discussed in the following sections.

### **ASCII Text files**

A text file can be a stream of characters that a computer can process sequentially. It is not only processed sequentially but only in forward direction. For this reason a text file is usually opened for only one kind of operation (reading, writing, or appending) at any given time.

Similarly, since text files only process characters, they can only read or write data one character at a time. (In C Programming Language, Functions are provided that deal with lines of text, but these still essentially process data one character at a time.) A text stream in C is a special kind of file. Depending on the requirements of the operating system, newline characters may be converted to or from carriage-return/linefeed combinations depending on whether data is being written to, or read from, the file. Other character conversions may also occur to satisfy the storage requirements of the operating system. These translations occur transparently and they occur because the programmer has signalled the intention to process a text file.

### **Binary files**

A binary file is no different to a text file. It is a collection of bytes. In C Programming Language a byte and a character are equivalent. Hence a binary file is also referred to as a character stream, but there are two essential differences.

1. No special processing of the data occurs and each byte of data is transferred to or from the disk unprocessed.
2. C Programming Language places no constructs on the file, and it may be read from, or written to, in any manner chosen by the programmer.

Binary files can be either processed sequentially or, depending on the needs of the application, they can be processed using random access techniques. In C Programming Language, processing a file using random access techniques involves moving the current file position to an appropriate place in the file before reading or writing data. This indicates a second characteristic of binary files â€“ they are generally processed using read and write operations simultaneously.

For example, a database file will be created and processed as a binary file. A record update operation will involve locating the appropriate record, reading the record into memory, modifying it in some way, and finally writing the record back to disk at its appropriate location in the file. These kinds of operations are common to many binary files, but are rarely found in applications that process text files.

### **Creating a file and output some data**

In order to create files we have to learn about File I/O i.e. how to write data into a file and how to read data from a file. We will start this section with an example of writing data to a file. We begin as before with the include statement for stdio.h, then define some variables for use in the example including a rather strange looking new type.

```
/* Program to create a file and write some data the file */
#include
#include
main()
{
    FILE *fp;
    char stuff[25];
    int index;
    fp = fopen("TENLINES.TXT","w"); /* open for writing */
    strcpy(stuff,"This is an example line.");
    for (index = 1; index <= 10; index++)
        fprintf(fp,"%s Line number %d\n", stuff, index);
    fclose(fp); /* close the file before ending program */
}
```

The type FILE is used for a file variable and is defined in the stdio.h file. It is used to define a file pointer for use in file operations. Before we can write to a file, we must open it. What this really means is that we must tell the system that we want to write to a file and what the file name is. We do this with the fopen() function illustrated in the first line of the program. The file pointer, fp in our case, points to the file and two arguments are required in the parentheses, the file name first, followed by the file type.

The file name is any valid DOS file name, and can be expressed in upper or lower case letters, or even mixed if you so desire. It is enclosed in double quotes. For this example we have chosen the name TENLINES.TXT. This file should not exist on your disk at this time. If you have a file with this name, you should change its name or move it because when we execute this program, its contents will be erased. If you don't have a file by this name, that is good because we will create one and put some data into it. You are permitted to include a directory with the file name. The directory must, of course, be a valid directory otherwise an error will occur. Also, because of the way C handles literal strings, the directory separation character \ must be written twice. For example, if the file is to be stored in the \PROJECTS sub directory then the file name should be entered as “\\PROJECTS\\TENLINES.TXT”. The second parameter is the file attribute and can be any of three letters, r, w, or a, and must be lower case.

### **Reading (r)**

When an r is used, the file is opened for reading, a w is used to indicate a file to be used for writing, and an indicates that you desire to append additional data to the data already in an existing file. Most C compilers have other file attributes available; check your Reference Manual for details. Using the r indicates that the file is assumed to be a text file. Opening a file for reading requires that the file already exist. If it does not exist, the file pointer will be set to NULL and can be checked by the program.

Here is a small program that reads a file and display its contents on screen. /\* Program to display the contents of a file on screen \*/

```
#include
void main()
{
    FILE *fopen(), *fp;
    int c;
    fp = fopen("prog.c", "r");
    c = getc(fp) ;
    while (c!= EOF)
```

```
{
    putchar(c);
    c = getc(fp);
}
fclose(fp);
}
```

## Writing (w)

When a file is opened for writing, it will be created if it does not already exist and it will be reset if it does, resulting in the deletion of any data already there. Using the w indicates that the file is assumed to be a text file.

Here is the program to create a file and write some data into the file.

```
#include
int main()
{
FILE *fp;
file = fopen("file.txt","w");
/*Create a file and add text*/
fprintf(fp,"%s","This is just an example :)");
/*writes data to the file*/
fclose(fp); /*done!*/
return 0;
}
```

## Appending (a):

When a file is opened for appending, it will be created if it does not already exist and it will be initially empty. If it does exist, the data input point will be positioned at the end of the present data so that any new data will be added to any data that already exists in the file. Using the a indicates that the file is assumed to be a text file.

Here is a program that will add text to a file which already exists and there is some text in the file.

```
#include
int main()
{
FILE *fp
```

```

file = fopen("file.txt","a");
fprintf(fp,"%s","This is just an example :)");
fclose(fp);
return 0;
}

```

### **Outputting to the file**

The job of actually outputting to the file is nearly identical to the outputting we have already done to the standard output device. The only real differences are the new function names and the addition of the file pointer as one of the function arguments. In the example program, fprintf replaces our familiar printf function name, and the file pointer defined earlier is the first argument within the parentheses. The remainder of the statement looks like, and in fact is identical to, the printf statement.

### **Closing a file**

To close a file you simply use the function fclose with the file pointer in the parentheses. Actually, in this simple program, it is not necessary to close the file because the system will close all open files before returning to DOS, but it is good programming practice for you to close all files in spite of the fact that they will be closed automatically, because that would act as a reminder to you of what files are open at the end of each program.

You can open a file for writing, close it, and reopen it for reading, then close it, and open it again for appending, etc. Each time you open it, you could use the same file pointer, or you could use a different one. The file pointer is simply a tool that you use to point to a file and you decide what file it will point to. Compile and run this program. When you run it, you will not get any output to the monitor because it doesn't generate any. After running it, look at your directory for a file named TENLINES.TXT and type it; that is where your output will be. Compare the output with that specified in the program; they should agree! Do not erase the file named TENLINES.TXT yet; we will use it in some of the other examples in this section.

### **Reading from a text file**

Now for our first program that reads from a file. This program begins with the familiar include, some data definitions, and the file opening statement which should require no explanation except for the fact that an r is used here because we want to read it.

#include

```

main( )
{
    FILE *fp;
    char c;
    funny = fopen("TENLINES.TXT", "r");
    if (fp == NULL)
        printf("File doesn't exist\n");
    else {
        do {
            c = getc(fp); /* get one character from the file
            */
            putchar(c); /* display it on the monitor
            */
        } while (c != EOF); /* repeat until EOF (end of file)
        */
    }
    fclose(fp);
}

```

In this program we check to see that the file exists, and if it does, we execute the main body of the program. If it doesn't, we print a message and quit. If the file does not exist, the system will set the pointer equal to NULL which we can test. The main body of the program is one do while loop in which a single character is read from the file and output to the monitor until an EOF (end of file) is detected from the input file. The file is then closed and the program is terminated. At this point, we have the potential for one of the most common and most perplexing problems of programming in C. The variable returned from the getc function is a character, so we can use a char variable for this purpose. There is a problem that could develop here if we happened to use an unsigned char however, because C usually returns a minus one for an EOF - which an unsigned char type variable is not capable of containing. An unsigned char type variable can only have the values of zero to 255, so it will return a 255 for a minus one in C. This is a very frustrating problem to try to find. The program can never find the EOF and will therefore never terminate the loop. This is easy to prevent: always have a char or int type variable for use in returning an EOF. There is another problem with this program but we will worry about it when we get to the next program and solve it with the one following that.

After you compile and run this program and are satisfied with the results, it would be a good exercise to change the name of TENLINES.TXT and run the program again to see that the

NULL test actually works as stated. Be sure to change the name back because we are still not finished with TENLINES.TXT.

## UNIT 11

### C - PREPROCESSOR

#### Overview

The C preprocessor, often known as *cpp*, is a *macro processor* that is used automatically by the C compiler to transform your program before compilation. It is called a macro processor because it allows you to define *macros*, which are brief abbreviations for longer constructs.

The C preprocessor is intended to be used only with C, C++, and Objective-C source code. In the past, it has been abused as a general text processor. It will choke on input which does not obey C's lexical rules. For example, apostrophes will be interpreted as the beginning of character constants, and cause errors. Also, you cannot rely on it preserving characteristics of the input which are not significant to C-family languages. If a Makefile is preprocessed, all the hard tabs will be removed, and the Makefile will not work.

Having said that, you can often get away with using *cpp* on things which are not C. Other Algol-ish programming languages are often safe (Pascal, Ada, etc.) So is assembly, with caution. *-traditional-cpp* mode preserves more white space, and is otherwise more permissive. Many of the problems can be avoided by writing C or C++ style comments instead of native language comments, and keeping macros simple

#### Include Syntax

Both user and system header files are included using the preprocessing directive `#include'. It has two variants:

```
#include <file>
```

This variant is used for system header files. It searches for a file named *file* in a standard list of system directories. You can prepend directories to this list with the *-I* option (see [Invocation](#)).

```
#include "file"
```

This variant is used for header files of your own program. It searches for a file named *file* first in the directory containing the current file, then in the quote directories and then the same directories used for *<file>*. You can prepend directories to the list of quote directories with the *-iquote* option.

The argument of `#include', whether delimited with quote marks or angle brackets, behaves like a string constant in that comments are not recognized, and macro names are not expanded. Thus, `#include <x/*y>` specifies inclusion of a system header file named `x/*y`.

However, if backslashes occur within *file*, they are considered ordinary text characters, not escape characters. None of the character escape sequences appropriate to string constants in C are processed. Thus, `#include "x\n\\y"` specifies a filename containing three backslashes. (Some systems interpret `\' as a pathname separator. All of these also interpret `/' the same way. It is most portable to use only `').)

It is an error if there is anything (other than comments) on the line after the file name.

### **Object-like**

### **Macros**

An *object-like macro* is a simple identifier which will be replaced by a code fragment. It is called **object-like** because it looks like a data object in code that uses it. They are most commonly used to give symbolic names to numeric constants.

You create macros with the `#define' directive. `#define' is followed by the name of the macro and then the token sequence it should be an abbreviation for, which is variously referred to as the macro's *body*, *expansion* or *replacement list*. For example,

```
#define BUFFER_SIZE 1024
```

defines a macro named `BUFFER_SIZE` as an abbreviation for the token `1024`. If somewhere after this `#define' directive there comes a C statement of the form .

```
foo = (char *) malloc (BUFFER_SIZE);
```

then the C preprocessor will recognize and *expand* the macro `BUFFER_SIZE`. The C compiler will see the same tokens as it would if you had written .

```
foo = (char *) malloc (1024);
```

By convention, macro names are written in uppercase. Programs are easier to read when it is possible to tell at a glance which names are macros.

The macro's body ends at the end of the `#define' line. You may continue the definition onto multiple lines, if necessary, using backslash-newline. When the macro is expanded, however, it will all come out on one line. For example,

```
#define NUMBERS 1, \
```

```

2, \
3
int x[] = { NUMBERS };
==> int x[] = { 1, 2, 3 };

```

The most common visible consequence of this is surprising line numbers in error messages.

There is no restriction on what can go in a macro body provided it decomposes into valid preprocessing tokens. Parentheses need not balance, and the body need not resemble valid C code. (If it does not, you may get error messages from the C compiler when you use the macro.)

The C preprocessor scans your program sequentially. Macro definitions take effect at the place you write them. Therefore, the following input to the C preprocessor

```

foo = X;
#define X 4
bar = X;

```

produces

```

foo = X;
bar = 4;

```

When the preprocessor expands a macro name, the macro's expansion replaces the macro invocation, then the expansion is examined for more macros to expand. For example,

```

#define TABLESIZE BUFSIZE
#define BUFSIZE 1024
TABLESIZE
==> BUFSIZE
==> 1024

```

`TABLESIZE` is expanded first to produce `BUFSIZE`, then that macro is expanded to produce the final result, `1024`.

Notice that `BUFSIZE` was not defined when `TABLESIZE` was defined. The `#define' for `TABLESIZE` uses exactly the expansion you specify—in this case, `BUFSIZE`—and does not check to see whether it too contains macro names. Only when you *use* `TABLESIZE` is the result of its expansion scanned for more macro names.

This makes a difference if you change the definition of `BUFSIZE` at some point in the source file. `TABLESIZE`, defined as shown, will always expand using the definition of `BUFSIZE` that is currently in effect:

```
#define BUFSIZE 1020
#define TABLESIZE BUFSIZE
#undef BUFSIZE
#define BUFSIZE 37
```

## Conditional Syntax

A conditional in the C preprocessor begins with a *conditional directive*: `#if`, `#ifdef` or `#ifndef`.

- Ifdef
- If
- Defined
- Else
- Elif

### Ifdef

The simplest sort of conditional is

```
#ifdef MACRO
controlled text
#endif /* MACRO */
```

This block is called a *conditional group*. *controlled text* will be included in the output of the preprocessor if and only if `MACRO` is defined. We say that the conditional *succeeds* if `MACRO` is defined, *fails* if it is not.

The *controlled text* inside of a conditional can include preprocessing directives. They are executed only if the conditional succeeds. You can nest conditional groups inside other conditional groups, but they must be completely nested. In other words, `#endif` always matches the nearest `#ifdef` (or `#ifndef`, or `#if`). Also, you cannot start a conditional group in one file and end it in another.

Even if a conditional fails, the *controlled text* inside it is still run through initial transformations and tokenization. Therefore, it must all be lexically valid C. Normally the only way this matters

is that all comments and string literals inside a failing conditional group must still be properly ended.

The comment following the `#endif` is not required, but it is a good practice if there is a lot of *controlled text*, because it helps people match the `#endif` to the corresponding `#ifdef`. Older programs sometimes put *MACRO* directly after the `#endif` without enclosing it in a comment. This is invalid code according to the C standard. CPP accepts it with a warning. It never affects which `#ifndef` the `#endif` matches.

Sometimes you wish to use some code if a macro is *not* defined. You can do this by writing `#ifndef` instead of `#ifdef`. One common use of `#ifndef` is to include code only the first time a header file is included. See [Once-Only Headers](#).

## **If**

The `#if` directive allows you to test the value of an arithmetic expression, rather than the mere existence of one macro. Its syntax is

```
#if expression
controlled text
#endif /* expression */
```

*expression* is a C expression of integer type, subject to stringent restrictions. It may contain

- Integer constants.
- Character constants, which are interpreted as they would be in normal code.
- Arithmetic operators for addition, subtraction, multiplication, division, bitwise operations, shifts, comparisons, and logical operations (`&&` and `||`). The latter two obey the usual short-circuiting rules of standard C.
- Macros. All macros in the expression are expanded before actual computation of the expression's value begins.
- Uses of the `defined` operator, which lets you check whether macros are defined in the middle of an `#if`.
- Identifiers that are not macros, which are all considered to be the number zero. This allows you to write `#if MACRO` instead of `#ifdef MACRO`, if you know that MACRO, when defined, will always have a nonzero value. Function-like macros used without their function call parentheses are also treated as zero.

## **Defined**

The special operator `defined` is used in `#if` and `#elif` expressions to test whether a certain name is defined as a macro. `defined name` and `defined (name)` are both expressions whose value is 1 if `name` is defined as a macro at the current point in the program, and 0 otherwise. Thus, `#if defined MACRO` is precisely equivalent to `#ifdef MACRO`.

`defined` is useful when you wish to test more than one macro for existence at once. For example,

```
#if defined (__vax__) || defined (__ns16000__)
```

would succeed if either of the names `__vax__` or `__ns16000__` is defined as a macro.

Conditionals written like this:

```
#if defined BUFSIZE && BUFSIZE >= 1024
```

can generally be simplified to just `#if BUFSIZE >= 1024`, since if `BUFSIZE` is not defined, it will be interpreted as having the value zero.

If the `defined` operator appears as a result of a macro expansion, the C standard says the behavior is undefined. GNU cpp treats it as a genuine `defined` operator and evaluates it normally. It will warn wherever your code uses this feature if you use the command-line option `-pedantic`, since other compilers may handle it differently.

## **Else**

The `#else' directive can be added to a conditional to provide alternative text to be used if the condition fails. This is what it looks like:

```
#if expression
  text-if-true
#else /* Not expression */
  text-if-false
#endif /* Not expression */
```

If `expression` is nonzero, the `text-if-true` is included and the `text-if-false` is skipped. If `expression` is zero, the opposite happens.

You can use `#else' with `#ifdef' and `#ifndef', too.

## Elif

One common case of nested conditionals is used to check for more than two possible alternatives. For example, you might have

```
#if X == 1  
...  
#else /* X != 1 */  
#if X == 2  
...  
#else /* X != 2 */  
...  
#endif /* X != 2 */  
#endif /* X != 1 */
```

Another conditional directive, `#elif', allows this to be abbreviated as follows:

```
#if X == 1  
...  
#elif X == 2  
...  
#else /* X != 2 and X != 1 */  
...  
#endif /* X != 2 and X != 1 */
```

`#elif' stands for “else if”. Like `#else', it goes in the middle of a conditional group and subdivides it; it does not require a matching `#endif' of its own. Like `#if', the `#elif' directive includes an expression to be tested. The text following the `#elif' is processed only if the original `#if'-condition failed and the `#elif' condition succeeds.

More than one `#elif' can go in the same conditional group. Then the text after each `#elif' is processed only if the `#elif' condition succeeds after the original `#if' and all previous `#elif' directives within it have failed.

`#else' is allowed after any number of `#elif' directives, but `#elif' may not follow `#else'.

---

## SYSTEM DEVELOPMENT

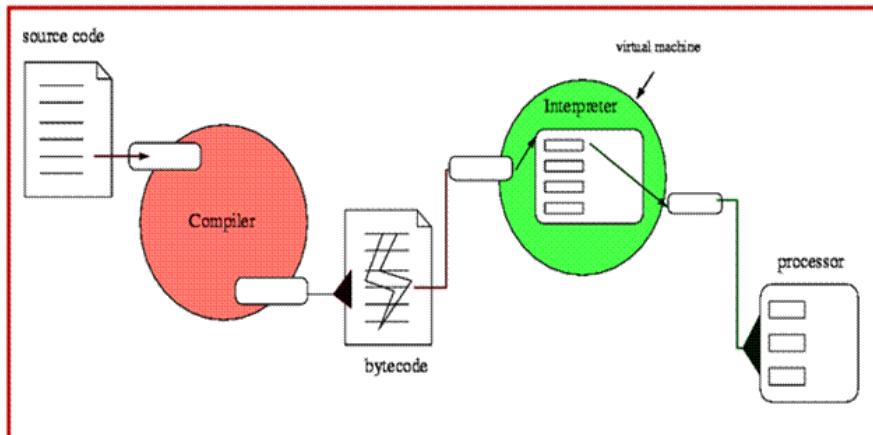


fig:show source code processing mechanism

## Introduction

A Programming language is a notational system for describing tasks/computations in a machine and .human readable form.

Most computer languages are designed to facilitate certain operations and not others

: Numerical computation, or text manipulation, or I/O.

More broadly, a computer language typically embodies a particular programming paradigm

### Characteristics of a programming language:

Every language has syntax and semantics:

- ❑ **Syntax:** The syntax of a program is the form of its declarations, expressions, statements and program units.
- ❑ **Semantic:** The semantic of a program is concerned with the meaning of its program.

## Programming Languages

Languages are used to communicate between different entities. Computer language makes it possible to talk to the computers and ask the computer to perform specific work. Computer language produces programs which are executed by CPU, and then CPU instructs all the other parts of computers to perform work accordingly. Computers only understand programs in their own machine language. Machine language is the language of 0's and 1's. It is difficult to write program in machine language.

Example	of	computer	Languages:
(i)	High	-	language
(ii)	Machine language		

### High-Level Languages (HLLs)

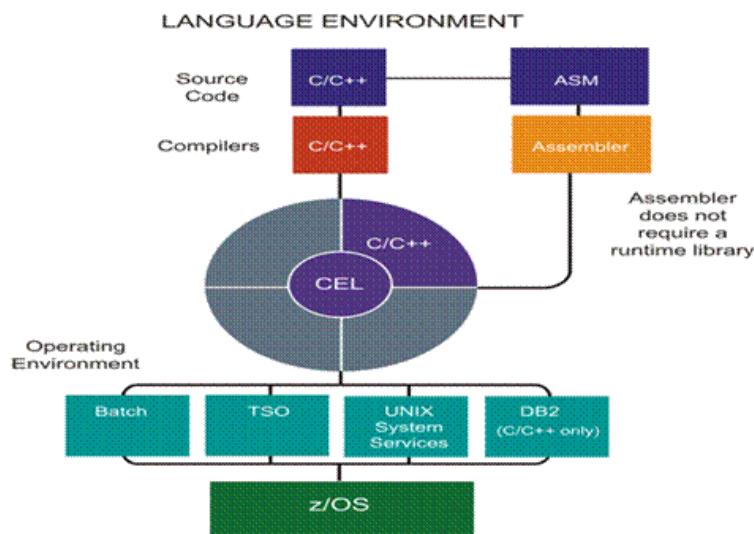
A high level language is one which hide details on how a computer operates in favor of making more abstract, human way at instructing it to perform tasks. HLLs are programming languages that look like natural language text.

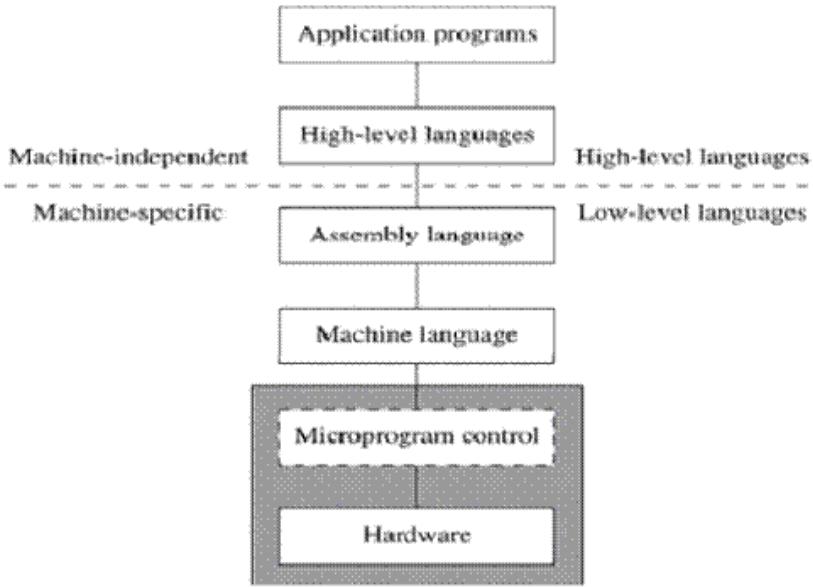
#### ADVANTAGES :

- (1) They make programming easier and more abstract
- (2) HLLs programs are machine independent, they can run on different hardware platforms i.e. Different computer with different instruction sets and a compiler is a program which does this.

### Machine language:

sometimes referred to as machine code or object code. Machine language is a collection of binary digits or bits that the computer reads and interprets. Machine language is the only language a computer is capable of understanding.





## Mapping Between HLL and Machine Language

Translating HLL programs to machine language programs is not a one-to-one mapping. A HLL instruction (usually called a statement) will be translated to one or more machine language instructions. The number of mapped machine instructions depends on the efficiency of the compiler in producing optimized machine language programs from the HLL programs. A machine language program produced by a compiler or an assembler. Usually, machine language programs produced by compilers are not efficient (i.e. they contain many unnecessary instructions that increase processing and slow down execution).

## Assembly language

Assembly language is the most basic programming language available for any processor. With assembly language, a programmer works only with operations implemented directly on the physical CPU. Assembly language lacks high-level conveniences such as variables and functions, and it is not portable between various families of processors. Nevertheless, assembly language is the most powerful computer programming language available, and it gives programmers the insight required to write effective code in high-level languages.

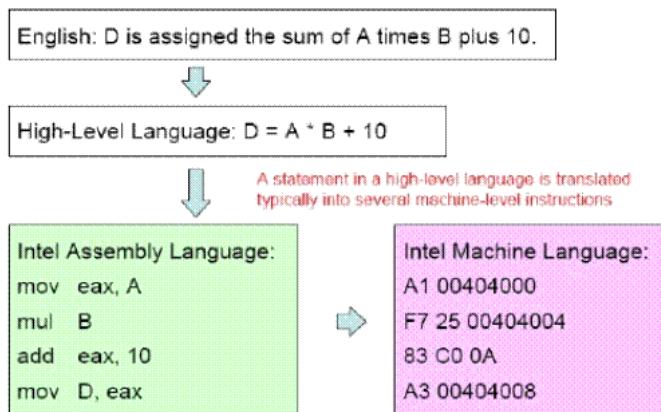
Machine code for displaying \$ sign on lower right corner of screen.

**10111000, 00000000, 10111000, 10001110, 11011000, 11000110, 00000110, 10011110,  
00001111, 00100100, 11001101, 00011111**

The program above, written in assembly language, looks like this:

## MOV AX, 47104MOV DS, AXMOV [3998], 36INT 32

*When an assembler reads this sample program, it converts each line of code into one CPU-level instruction.*

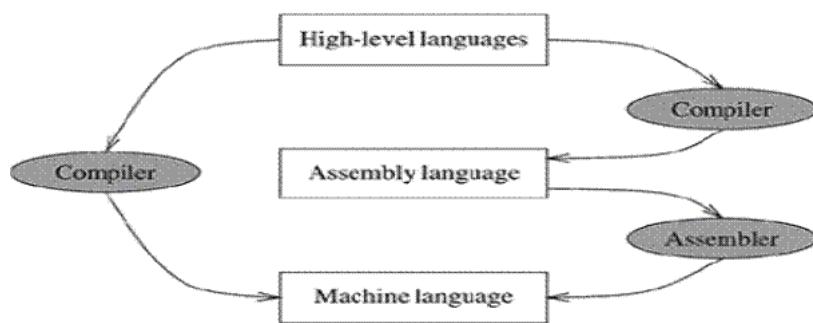


### ***Advantages of learning Assembly language***

1. Very useful for making efficient and fast running programs.
2. Very useful for making small programs for embedded system applications.
3. Easy to access hardware in assembly language
4. Writing compact code.

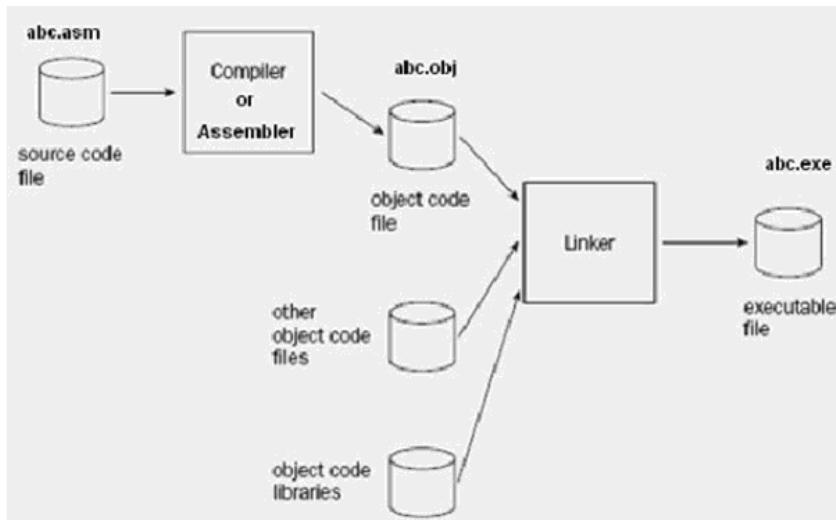
### ***The Compiler***

*Compiler is a program that translates the high level programs to machine code either directly or via assembler*



### ***The Assembler***

The program that translates from assembly language to machine language is called an assembler. It allows the programmer to specify the memory locations for his data and programs and symbolically refer to them in his assembly code. It will translate these symbolic addresses to actual (physical) addresses in the produced machine code.



## The Linker

*This is the program that is used to link together separately assembled/compiled programs into a single executable code. The linker program is used to create executable code which finally runs on the CPU.*

## Debugger and Monitor

*These are the tools that allow the assembly programmers to:*

1. *Display and alter the contents of memory and registers while running their code.*
2. *Perform disassembly of their machine code (show the assembly language equivalent).*
3. *Permit them to run their programs stop (or halt) them, run them step-by-step or insert break points. Break points: Positions in the program that if are encountered during run time, the program will be halted so the programmer can examine the memory and registers contents and determine what went wrong.*

## Compilers and Interpreters

All high level language (HLL) programs need to be translated into machine code (aka 1's and 0's/binary). There are two forms of translators to make this happen; Compilers and Interpreters.

**Compilers** – These turn the whole HLL program into machine code in one go. This results in a file (object code) that can then be run on any computer system (is very portable) and runs very quickly (as it does not need to be translated again). Unfortunately if you have made a mistake (called a syntax error) in your program then it won't run at all.

**Interpreters** - These turn the HLL program into binary each time you try and run it. An interpreter will go through each line of your program each time it translates it. This means that it is slow to translate your program.

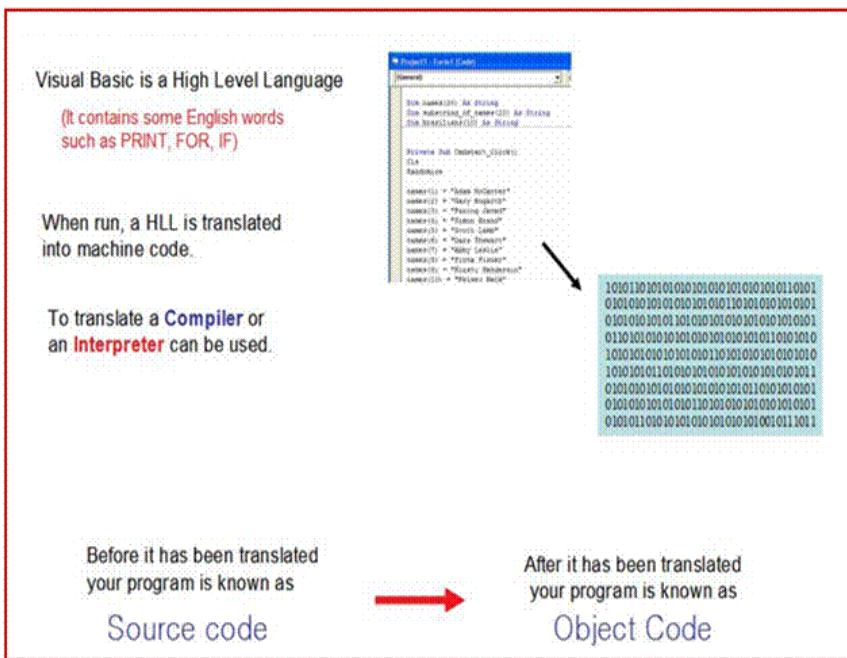


fig: show translation of program in HLL to machine language

## 1. Software Paradigms

### Introduction

"Paradigm" (a Greek word meaning example) is commonly used to refer to a category of entities that share a common characteristic.

We can distinguish between three different kinds of Software Paradigms:

- **Programming Paradigm** is a model of how programmers communicate and calculate to computers
- **Software Design Paradigm** is a model for implementing a group of applications sharing common properties
- **Software Development Paradigm** is often referred to as Software Engineering, may be seen as a management model for implementing big software projects using engineering principles.

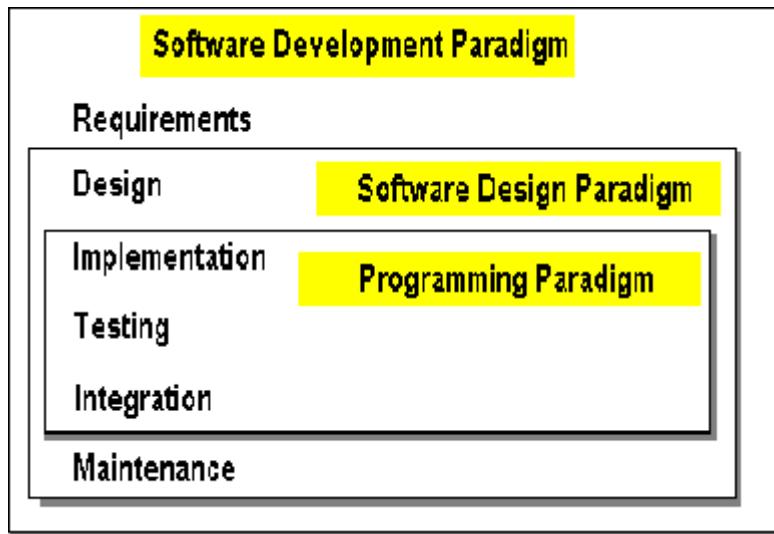


Fig: show software development paradigm

## Programming Paradigm

A Programming Paradigm is a model for a class of Programming Languages that share a set of common characteristics.

A **programming language** is a system of signs used to communicate a task/algorithm to a computer, causing the task to be performed. The task to be performed is called a *computation*, which follows absolutely precise and unambiguous rules.

### Components to any language

(i ) The **language paradigm** is a general principles that are used by a programmer to communicate a task/algorithm to a computer.

(ii) The ***syntax*** of the language is a way of specifying what is legal in the phrase structure of the language; knowing the syntax is analogous to knowing how to spell and form sentences in a natural language like English. However, this doesn't tell us anything about what the sentences mean.

(iii) The third component is ***semantics***, or meaning, of a program in that language. Ultimately, without semantics, a programming language is just a collection of meaningless phrases; hence, the semantics is the crucial part of a language.

There have been a large number of programming languages. Back in the 60's there were over 700 of them – most were academic, special purpose, or developed by an organization for their own needs.

Fortunately, there are just four major programming language paradigms:

- *Imperative (Procedural) Paradigm* (FORTRAN, C, Ada, etc.)
- *Object-Oriented Paradigm* (Smalltalk, Java, C++)
- *Logic Paradigm* (Prolog)
- *Functional Paradigm* (Lisp, ML, Haskell)

Generally, a selected Programming Paradigm defines main property of a software developed by means of a programming language supporting the paradigm.

- Scalability/modifiability
- Inerrability/reusability
- Portability
- Performance
- Reliability
- Ease of creation

## Software Design Paradigm

Software Design Paradigm embody the results of people's ideas on how to construct programs, combine them into large software systems and formal mechanisms for how those ideas should be expressed.

Thus, we can say that a Software Design Paradigm is a model for a class of problems that share a set of common characteristics.

Software design paradigms can be sub-divided as:

- Design Patterns
- Components
- Software Architecture
- Frameworks

It should be especially noted that a particular Programming Paradigm essentially defines software design paradigms. For example, we can speak about Object-Oriented design patterns, procedural components (modules), functional software architecture, etc.

Design Patterns:

A design pattern is a proven solution for a general design problem. It consists of communicating 'objects' that are customized to solve the problem in a particular context.

Patterns have their origin in object-oriented programming where they began as collections of objects organized to solve a problem. There isn't any fundamental relationship between patterns and objects; it just happens they began there. Patterns may have arisen because objects seem so elemental, but the problems we were trying to solve with them were so complex.

- Architectural Patterns: An architectural pattern expresses a fundamental structural organization or schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.
- Idioms: An idiom is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language.

## Components:

A component is a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces, typically represents the physical packaging of otherwise logical elements, such as classes, interfaces, and collaborations.

Software components are binary units of independent production, acquisition, and deployment that interact to form a functioning program.

A component must be compatible and interoperate with a whole range of other components.

Examples of components: "Window", "Push Button", "Text Editor", etc.

## Software Architecture:

Software architecture is the structure of the components of the solution. A particular software architecture decomposes a problem into smaller pieces and attempts to find a solution (Component) for each piece. We can also say that an architecture defines a software system components, their integration and interoperability:

- Integration means the pieces fit together well.
- Interoperation means that they work together effectively to produce an answer.

There are many software architectures. Choosing the right one can be a difficult problem in itself.

## Frameworks:

A *software framework* is a reusable mini-architecture that provides the generic structure and behavior for a family of software abstractions, along with a context of metaphors which specifies their collaboration and use within a given domain.

Frameworks can be seen as an intermediate level between components and a software architecture.

Example: Suppose architecture of a WBT system reuse such components as "Text Editing Input object" and "Push buttons". A software framework may define an "HTML Editor" which can be further reused for building the architecture.

## Software Engineering and Software Paradigms

The term "software engineering" was coined in about 1969 to mean "the establishment and use of sound engineering principles in order to economically obtain software that is reliable and works efficiently on real machines".

This view opposed uniqueness and "magic" of programming in an effort to move the development of software from "magic" (which only a select few can do) to "art" (which the talented can do) to "science" (which supposedly anyone can do!). There have been numerous definitions given for software engineering (including that above and below).

Software Engineering is not a discipline; it is an aspiration, as yet unarchived. Many approaches have been proposed including reusable components, formal methods, structured methods and architectural studies. These approaches chiefly emphasize the engineering product; the solution rather than the problem it solves.

Software Development current situation:

- People developing systems were consistently wrong in their estimates of time, effort, and costs
- Reliability and maintainability were difficult to achieve
- Delivered systems frequently did not work

i.e. 1979 study of a small number of government projects showed that:

- \* 2% worked
- \* 3% could work after some corrections
- \* 45% delivered but never successfully used
- \* 20% used but extensively reworked or abandoned
- \* 30% paid and undelivered
- Fixing bugs in delivered software produced more bugs
- Increase in size of software systems
  - NASA
  - Star Wars Defense Initiative
  - Social Security Administration
  - financial transaction systems

- Changes in the ratio of hardware to software costs
  - early 60's - 80% hardware costs
  - middle 60's - 40-50% software costs
  - today - less than 20% hardware costs
- Increasingly important role of maintenance
  - Fixing errors, modification, adding options
  - Cost is often twice that of developing the software
- Advances in hardware (lower costs)
- Advances in software techniques (e.g., users interaction)
- Increased demands for software
  - Medicine, Manufacturing, Entertainment, Publishing
- Demand for larger and more complex software systems
  - Airplanes (crashes), NASA (aborted space shuttle launches),
  - "ghost" trains, runaway missiles,
  - ATM machines (have you had your card "swallowed"?), life-support systems, car systems, etc.
  - US National security and day-to-day operations are highly dependent on computerized systems.

Manufacturing software can be characterized by a series of steps ranging from concept exploration to final retirement; this series of steps is generally referred to as a *software lifecycle*.

Steps or phases in a software lifecycle fall generally into these categories:

- Requirements (Relative Cost 2%)
- Specification (analysis) (Relative Cost 5%)
- Design (Relative Cost 6%)
- Implementation (Relative Cost 5%)

- Testing (Relative Cost 7%)
- Integration (Relative Cost 8%)
- Maintenance (Relative Cost 67%)
- Retirement

**Software engineering employs a variety of methods, tools, and paradigms.**

Paradigms refer to particular approaches or philosophies for designing, building and maintaining software. Different paradigms each have their own advantages and disadvantages which make one more appropriate in a given situation than perhaps another (!).

A method (also referred to as a technique) is heavily depended on a selected paradigm and may be seen as a procedure for producing some result. Methods generally involve some formal notation and process(es).

**Tools are automated systems implementing a particular method.**

Thus, the following phases are heavily affected by selected software paradigms

- Design
- Implementation
- Integration
- Maintenance

The software development cycle involves the activities in the production of a software system. Generally the software development cycle can be divided into the following phases:

- Requirements analysis and specification
- Design
  - Preliminary design
  - Detailed design
- Implementation

- Component Implementation
- Component Integration
- System Documenting
- Testing
  - Unit testing
  - Integration testing
  - System testing
- Installation and Acceptance Testing
- Maintenance
  - Bug Reporting and Fixing
  - Change requirements and software upgrading

Software life cycles that will be briefly reviewed include:

- Build and Fix model
- Waterfall and Modified Waterfall models
- Rapid Prototyping
- Boehm's spiral model

### **Build and Fix model**

This works OK for small, simple systems, but is completely unsatisfactory for software systems of any size. It has been shown empirically that the cost of changing a software product is relatively small if the change is made at the requirements or design phases but grows large at later phases.

The cost of this process model is actually far greater than the cost of a properly specified and designed project. Maintenance can also be problematic in a software system developed under this scenario.

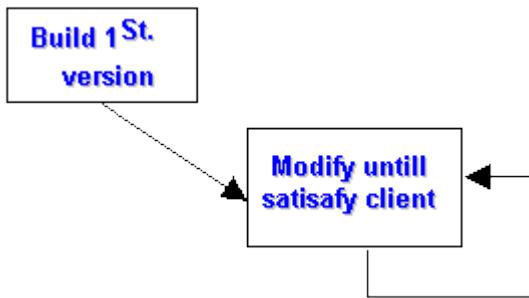


Figure: Build and Fix model

## Waterfall and Modified Waterfall models

### Waterfall Model

Offered a means of making the development process more structured, expresses the interaction between subsequent phases.

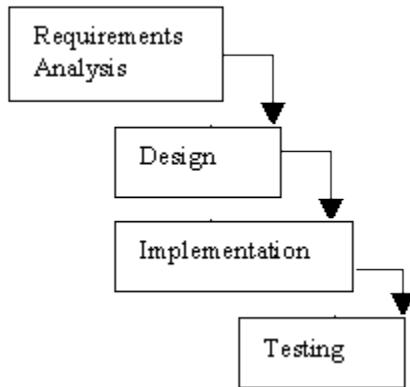


Figure: Waterfall model

Each phase cascades into the next phase. In the original waterfall model, a strict sequentially was at least implied. This meant that one phase had to be completed before the next phase was begun.

It also did not provide for feedback between phases or for updating/re-definition of earlier phases. Implies that there are definite breaks between phases, i.e., that each phase has a strict, non-overlapping start and finish and is carried out sequentially.

Critical point is that no phase is complete until the documentation and/or other products associated with that phase are completed.

## Modified Waterfall Model

Needed to provide for overlap and feedback between phases. Rather than being a simple linear model, it needed to be an iterative model. To facilitate the completion of the goals, milestones, and tasks, it is normal to freeze parts of the development after a certain point in the iteration. Verification and validation are added. Verification checks that the system is correct (building the system right). Validation checks that the system meets the users desires (building the right system).

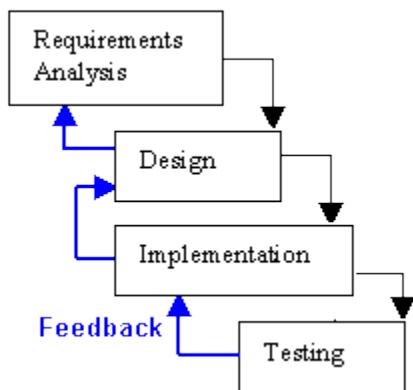


Figure: Modified Waterfall model

The waterfall model (and modified waterfall model) is inflexible in the partitioning of the project into distinct phases. However, they generally reflect engineering practice.

Considerable emphasis must be placed on discerning users' needs and requirements prior to the system being built. The identification of users' requirements as early as possible, and the agreement between user and developer with respect to those requirements, often is the deciding factor in the success or failure of a software project. These requirements are documented in the requirements specification, which is used to verify whether subsequent phases are complying with the requirements. Unfortunately specifying users' requirements is very much an art, and as such is extremely difficult. Validation feedback can be used to prevent the appearance of a strong divergence between the system under development and the users' expectations for the delivered system.

Unfortunately, the waterfall life cycle (and the modified waterfall life cycle) are inadequate for realistic validation activities. They are exclusively document driven models. The resulting design reality is that only 50% of the design effort occurs during the actual design phase with 1/3 of the design effort occurring during the coding activity! This is topped by the fact that over 16% of the design effort occurs after the system is supposed to be completed! In general

the behavior of many individuals in this type of process is opportunistic. The boundaries of phases are indiscriminately crossed with deadlines being somewhat arbitrary.

## Rapid Prototyping

Prototyping also referred to as evolutionary development, prototyping aims to enhance the accuracy of the designer's perception of the user's requirements. Prototyping is based on the idea of developing an initial implementation for user feedback, and then refining this prototype through many versions until a satisfactory system emerges. The specification, development and validation activities are carried out concurrently with rapid feedback across the activities. Generally, prototyping is characterized by the use of very high-level languages, which probably will not be used in the final software implementation but which allow rapid development, and the development of a system with less functionality with respect to quality attributes such as robustness, speed, etc.

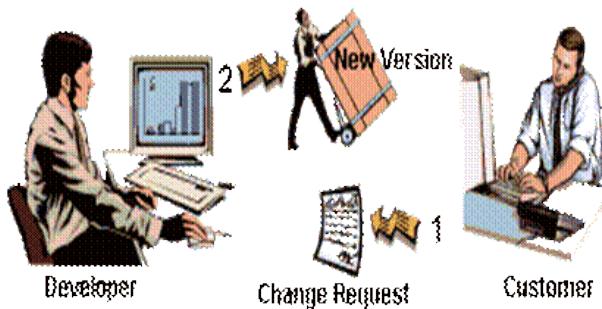


Figure: Rapid Prototyping model

Prototyping allows the clarification of users requirements through, particularly, the early development of the user interface. The user can then try out the system, albeit a (sub) system of what will be the final product. This allows the user to provide feedback before a large investment has been made in the development of the wrong system.

There are two types of prototypes:

- Exploratory programming: Objective is to work with the user to explore their requirements and deliver a final system. Starts with the parts of the system which are understood, and then evolves as the user proposes new features.
- Throw-away prototyping: Objective is to understand the users' requirements and develop a better requirements definition for the system. Concentrates on poorly understood components.

## Boehm's Spiral Model

Need an improved software life cycle model which can subsume all the generic models discussed so far. Must also satisfy the requirements of management.

Boehm proposed a spiral model where each round of the spiral

- a) identifies the sub problem which has the highest risk associated with it
- b) finds a solution for that problem.

## Imperative (Procedural) Programming Paradigm

Any imperative program consists of

- *Declarative statements* which gives a name to a value. A named value is called a variable. Thus, declarative statements create variables. In procedural languages it is common for the same variable to keep changing value as the program runs.
- *Imperative statements* which assign new values to variables
- *Program flow control statements* which define order in which imperative statements are evaluated.

Example:

```
var factorial = 1; /*Declarative statement*/  
  
var argument = 5;  
  
var counter = 1;  
  
while (counter <= argument) /*Program flow statement*/  
  
{  
  
    factorial = factorial*counter; /*Imperative statement*/  
  
    counter++;  
  
}
```

## Variables and Types

Different variables in a program may have different types. For example, a language may treat a two bytes as a string of characters and as a number. Dividing a string ‘20’ by number ‘2’ may not be possible. A language like this has at least two types - one for strings and one for numbers.

Example:

```
var PersonName = new String(); /*variable type “string”*/
```

```
var PersonSalary = new Integer(); /*variable type “integer”*/
```

Types can be weak or strong. Strong type means that at any point in the program, when it is running, the type of a particular chunk of data (i.e. variable) is known. Weak type means that imperative operators may change a variable type.

Example:

```
var PersonName; /*variable of a weak type”*/
```

```
PersonName = 0; /*PersonName is an “integer”*/
```

```
PersonName = ‘Nick’; /*PersonName is a “string”*/
```

Obviously, languages supporting weak variable types need sophisticated rules for type conversions.

Example:

```
var PersonName; /*variable of a weak type”*/
```

```
PersonName = 0; /*PersonName is an “integer”*/
```

```
PersonName = PersonName + ‘Nick’ + 0; /*PersonName is a string “0Nick0”*/
```

To support weak typing, values are boxed together with information about their type - value and type are then passed around the program together.

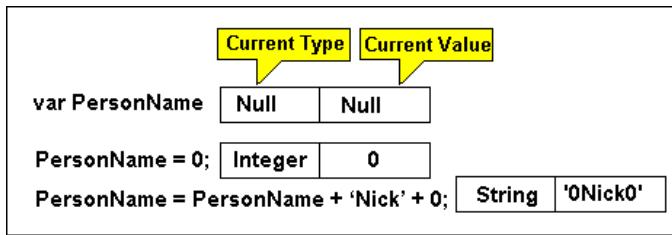


Fig . show how to support weak typing

## Functions (Procedures)

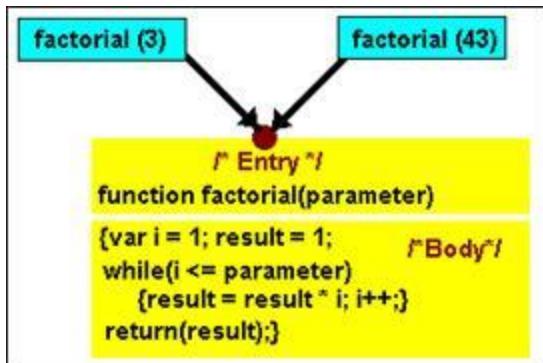
Programmers have dreamed/attempted of building systems from a library of reusable software components bound together with a little new code.

Imperative (Procedural) Programming Paradigm is essentially based on concept of so-called “Functions” also known as “Modules”, “Procedures” or “Subroutines”.

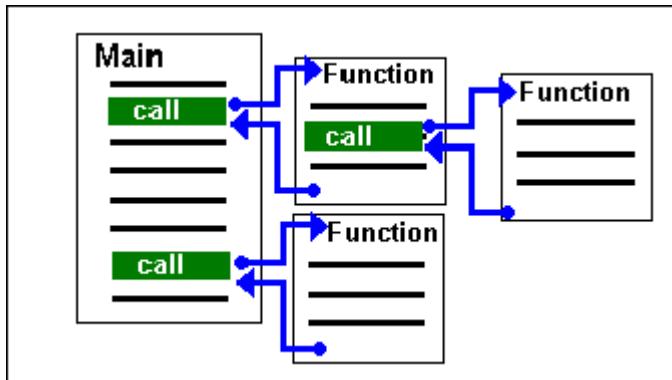
A function is a section of code that is parceled off from the main program and hidden behind an interface:

```
function factorial(parameter)
{
    var i = 1;
    var result = 1;
    while(i <= parameter)
    {
        result = result * i;
        i++;
    }
    return(result);
}
```

- The code within the function performs a particular activity, here generating a factorial value
- The idea of parceling the code off into a subroutine is to provide a single point of entry. Anyone wanting a new factorial value has only to call the “factorial” function with the appropriate parameters.



Here's what the conventional application based on the Imperative (Procedural) Programming Paradigm looks like:



- Main procedure determines the control flow for the application
- Functions are called to perform certain tasks or specific logic
- The main and sub procedures that comprise the implementation are structured as a hierarchy of tasks.
- The source for the implementation is compiled and linked with any additional executable modules to produce the application

## Data Exchange between Functions (Procedures)

When a software system functionality is decomposed into a number of functional modules, data exchange/flow becomes a key issue. Imperative (Procedural) Programming Paradigm extends the concept of variables to be used as such data exchange mechanism.

Thus, each procedure may have a number of special variables called parameters. The parameters are just named place-holders which will be replaced with particular values (or references to existing values) of arguments when the procedure is called.

Example:

```
function main()
{
    var argument = 25;
    var result = factorial(argument)
    /* Note, the imperative operator replaces the "parameter" place holder with a
    current value of the variable "argument"*/
}

function factorial(parameter)
{
    var i = 1;
    var result = 1;
    while(i <= parameter)
    {
        result = result * i;
        i++;
    }
}
```

```
    return(result);  
  
}
```

### **Passing arguments to a function**

There might be two different techniques for such replacement which are known as: passing an argument value and passing an argument reference.

In case of passing a value, a current argument value is duplicated as a value for new parameter variable dynamically created for the procedure. In this case, variables used as arguments for calling sub-routines cannot be modified by imperative operators inside of the sub-routines.

In case of passing a reference, the sub-routine gets control (i.e. reference) to a current value of the argument variable. In this case, variables used as arguments for calling sub-routines can be modified by imperative operators inside of the sub-routines.

Thus, types of variables defined as parameters of a function should be equivalent to (or at least compatible with) types of variables (constants) used as arguments.

### **Polymorphic Languages**

When strong static typing is enforced it can be difficult to write generic algorithms - functions that can act on a range of different types. Polymorphism allows "any" to be included in the type system. For example, the types of a list of items are unimportant if we only want to know the length of the list, so a function can have a type that indicates that it takes lists of "any" type and returns an integer. Moreover, polymorphism allows combining functions implemented by means of different programming languages supporting potentially different types of variables.

Pragmatically speaking, polymorphic languages allow to define new types as hidden functions which should be automatically applied to values of such "user-defined type" to convert it to values of a "standard" language type.

### **Variable Scope**

Normally, variables that are defined within a function are created each time the function is used and destroyed again when the function ends. The value that the function returns is not destroyed, but it is not possible to assign a value to the variable inside the function definition from outside.

Example:

```
function one()

{

    var dynamicLocalVariable = 25;

    two();

    /* at this point just one variable “dynamicLocalVariable” exists */

    alert(dynamicLocalVariable);

    /* this operator displays the current value “25” */

}

function two()

{

    var dynamicLocalVariable = 55;

    /* at this point two variables “dynamicLocalVariable” exists */

    alert(dynamicLocalVariable);

    /* this operator displays the current value “55” */

}
```

Such variables are called *dynamic local variables*. There may be also so-called *static local variables*. Static local variables that are defined within a function, are created only once when the function is used for a first time. The value of such variable is not destroyed and can be reused when the function is called again.

Example:

```
function one()

{
```

```
var x = two();  
  
alert(x);  
  
/* this operator displays the current value “10” */  
  
x = two();  
  
alert(x);  
  
/* this operator displays the current value “20” */  
  
}  
  
function two()  
  
{  
  
    var static staticLocalVariable = 0;  
  
    staticLocalVariable = staticLocalVariable + 10;  
  
    return(staticLocalVariable);  
  
}
```

Note that function “two” returns different values for one and the same set of arguments. Such functions are called reactive functions. Generally, testing and maintenance of projects having many reactive functions becomes a very difficult task. For practical reasons many software projects do use some static data.

Note, it is still not possible to assign a value to the local static variable inside a function from outside.

There may be also so-called *static global variables*. Static global variables that are defined within any function, are created only once when the whole software system is initiated. The value of such variable is never destroyed and can be reused by imperative operators inside any function.

Example:

```
function one()
{
    var global globalLocalVariable = 0;
    two();
    alert(globalLocalVariable);
    /* this operator displays the current value “10” */
    two();
    alert(globalLocalVariable);
    /* this operator displays the current value “20” */
}

function two()
{
    globalLocalVariable = globalLocalVariable + 10;
}
```

Here, the function “two” also demonstrates a “reactive” behavior. Maintaining and testing of projects heavily based on global variables becomes even more difficult than in case of local static variables. Nevertheless, for practical reasons many software development paradigms do use such global static variables.

## Software Design Methodology (Procedural Paradigm)

### Benefits of the Paradigm:

*Re-usability:* anyone that needs a particular functionality can use an appropriate module, without having to code the algorithm from scratch.

*Specialization:* one person can concentrate on writing a best possible module (function) for a particular task while others look after other areas.

*Upgradability:* if a programmer comes up with a better way to implement a module then he/she simply replace the code within the function. Provided the interface remains the same - in other words the module name and the order and type of each parameter are unchanged - then no changes should be necessary in the rest of the application.

However procedural modules have serious limitations:

- For a start, there is nothing to stop another programmer from meddling with the code within a module, perhaps to better adapt it to the needs of a particular application.
- There is also nothing to stop the code within the function making use of global variables, thus negating the benefits of a single interface providing a single point of entry.

Obviously, the paradigm is best suited for the *waterfall model* of software development.

## Design

A particular software system is viewed in terms of its modules and data flowing between them starting with a high-level view.

In this case, software design methodology can be categorized as a Top-down modular design (functional design viewpoint).

### The basic design concepts include:

- Modularity
  - Modules are used to describe a functional decomposition of the system
  - A module is a unit containing:
    - executable statements

- data structures
- other modules

A module:

- has a name
- can be separately compiled
- can be used in a program or by other modules

• System design generally determines what goes into a module

Cohesive

- Single clearly defined function
- Description of when and how used
- Loosely Coupled Modules (Modules implement functionality, but not parts of other modules)

Black Boxes (information hiding)

- each module is a black box
- each module has a set of known inputs and a set of predictable outputs
- inner workings of module are unknown to user
- can be reusable

Preliminary and Detailed Design specify the modules to carry out the functions in the DataFlow Diagrams (DFD).

**Preliminary design deals mainly with Structure Charts:**

Hierarchical tree structure

- Modules - rectangle boxes
- calling relationships are shown with arrows
- arrows are labeled with the data flowing between modules

Module Design

- Title
- Module ID - from structure charts
- Purpose
- Method - algorithm
- Usage - who calls it
- External references - other modules called
- Calling sequence - parameter descriptions
- Input assertion
- Output assertion
- Local variables
- Author(s)
- Remarks

### Preliminary Design Document

- Cover Page
- Table of Contents
- Design Description
- Software Structure Charts
- Data Dictionary
- Module Designs
- Module Headers
- Major Data Structures Design
- Design Reviews (Examination of all or part of the software design to find design anomalies )

### Overview of Detailed Design

- select an algorithm for each module

- refine the data structures
- produce detailed design document

## Implementation

Coding (for each Module)

- Source Code
- Documentation

Integration

- Decide what order the modules will be assembled
- Assemble and test integration of modules
- After final assembly perform system test
- Note, coding and testing are often done in parallel

## Testing

Types of testing

- Unit testing
- Integration testing
- Acceptance testing

As it was mentioned above, the paradigm is best suited for the *waterfall model* of software development. Implementing change requirements and especially rapid prototyping are weak points of the programming paradigm.

---