

DATA STRUCTURE AND ALGORITHMS

Data Structure

Computer Science is the study of data, its representation and transformation by Computer. For every data object, we consider the class of operations to be performed and then the way to represent the object so that these operations may be efficiently carried out. We require two techniques for this:

- Devise alternative forms of data representation
- Analyses the algorithm which operates on the structure.

These are several terms involved above which we need to know carefully before we proceed. These include data structure, data type and data representation.

A data type is a term which refers to the kinds of data that variables may hold. With every programming language there is a set of built-in data types. This means that the language allows variables to name data of that type and provides a set of operations which meaningfully manipulates these variables. Some data types are easy to provide because they are built-in into the computer's machine language instruction set, such as integer, character etc. Other data types require considerably more efficient to implement. In some languages, these are features which allow one to construct combinations of the built-in types (like structures in 'C'). However, it is necessary to have such mechanism to create the new complex data types which are not provided by the programming language. The new type also must be meaningful for manipulations. Such meaningful data types are referred as abstract data type.

ABSTRACT DATA TYPE:

An abstract data type can be assumed as a mathematical model with a collection of operations defined on that model i.e. an ADT is a new data type derived or created from basic or built in data type based on a particular logical or mathematical model.

For Example:

Set of integers consisting of different numbers may be an ADT. A set is a combination of more than one integer, but the operations on set is a generalized operation of different integers such as union, intersection, product, and difference. Above ADT or set encapsulates different mathematical operations and generalizes operations on ADT.

Basic Properties of ADT are: -

- (i) Encapsulation and
- (ii) Generalization

Let us consider the following example:

Struct student

```
{  
  
    int rno;  
  
    char name[21],branch[11]  
  
    int marks;  
  
};
```

The above structure can be used to collect or retrieve the information of a student. The structure can be called as ADT if all the operations on student can be performed using the structure.

DATA STRUCTURE:

An implementation of abstract data type is data structure i.e. a mathematical or logical model of a particular organization of data is called data structure.

Thus, a data structure is the portion of memory allotted for a model, in which the required data can be arranged in a proper fashion.

TYPES:-

A data structure can be broadly classified into

- (i) Primitive data structure
- (ii) Non-primitive data structure

(i) Primitive data structure

The data structures, typically those data structure that are directly operated upon by machine level instructions i.e. the fundamental data types such as int, float, double in case of 'c' are known as primitive data structures.

(ii) Non-primitive data structure

The data structures, which are not primitive are called non-primitive data structures.

There are two types of-primitive data structures.

(a) Linear Data Structures:-

A list, which shows the relationship of adjacency between elements, is said to be linear data structure. The most, simplest linear data structure is a 1-D array, but because of its deficiency, list is frequently used for different kinds of data.

(b) Non-linear data structure:-

A list, which doesn't show the relationship of adjacency between elements, is said to be non-linear data structure.

Linear Data Structure:

A list is an ordered list, which consists of different data items connected by means of a link or pointer. This type of list is also called a linked list. A linked list may be a single list or double linked list.

- Single linked list: - A single linked list is used to traverse among the nodes in one direction.
- Double linked list: - A double linked list is used to traverse among the nodes in both the directions.

A linked list is normally used to represent any data used in word-processing applications, also applied in different DBMS packages.

A list has two subsets. They are: -

- Stack: - It is also called as last-in-first-out (LIFO) system. It is a linear list in which insertion and deletion take place only at one end. It is used to evaluate different expressions.
- Queue: - It is also called as first-in-first-out (FIFO) system. It is a linear list in which insertion takes place at once end and deletion takes place at other end. It is generally used to schedule a job in operating systems and networks.

Non-linear data structure:-

The frequently used non-linear data structures are

- (a) Trees : - It maintains hierarchical relationship between various elements
- (b) Graphs : - It maintains random relationship or point-to-point relationship between various elements.

OPERATION ON DATA STRUCTURES: -

The four major operations performed on data structures are:

- (i) Insertion : - Insertion means adding new details or new node into the data structure.
- (ii) Deletion : - Deletion means removing a node from the data structure.
- (iii) Traversal : - Traversing means accessing each node exactly once so that the nodes of a data structure can be processed. Traversing is also called as visiting.
- (iv) Searching : - Searching means finding the location of node for a given key value.

Apart from the four operations mentioned above, there are two more operations occasionally performed on data structures. They are:

- (a) Sorting : - Sorting means arranging the data in a particular order.
- (b) Merging : - Merging means joining two lists.

REPRESENTATION OF DATA STRUCTURES:-

Any data structure can be represented in two ways. They are: -

- (i) Sequential representation
- ((Linked representation

- (i) Sequential representation: - A sequential representation maintains the data in continuous memory locations which takes less time to retrieve the data but leads to time complexity during insertion and deletion operations. Because of sequential nature, the elements of the list must be freed, when we want to insert a new element or new data at a particular position of the list. To acquire free space in the list, one must shift the data of the list towards the right side from the position where the data has to be inserted. Thus, the time taken by CPU to shift the data

will be much higher than the insertion operation and will lead to complexity in the algorithm. Similarly, while deleting an item from the list, one must shift the data items towards the left side of the list, which may waste CPU time.

Drawback of Sequential representation: -

The major drawback of sequential representation is taking much time for insertion and deletion operations unnecessarily and increasing the complexity of algorithm.

- (ii) *Linked Representation:* - Linked representation maintains the list by means of a link between the adjacent elements which need not be stored in continuous memory locations. During insertion and deletion operations, links will be created or removed between which takes less time when compared to the corresponding operations of sequential representation.

Because of the advantages mentioned above, generally, linked representation is preferred for any data structure.

Algorithm Analysis:

An algorithm is a finite set of instructions that, if followed, accomplishes a particular task. In addition, all algorithms must satisfy the following criteria.

1. Input
2. Output
3. Definiteness
4. Finiteness
5. Effectiveness

The criteria 1 & 2 require that an algorithm produces one or more outputs & have zero or more input. According to criteria 3, each operation must be definite such that it must be perfectly clear what should be done. According to the 4th criteria algorithm should terminate after a finite no. of operations. According to 5th criteria, every instruction must be very basic so that it can be carried out by a person using only pencil & paper.

There may be many algorithms devised for an application and we must analyse and validate the algorithms to judge the suitable one.

To judge an algorithm the most important factors is to have a direct relationship to the performance of the algorithm. These have to do with their computing time & storage requirements (referred as Time complexity & Space complexity).

Space Complexity:

The space complexity of an algorithm is the amount of memory it needs to run.

Time Complexity:

The time taken by a program is the sum of the compiled time & the run time. The time complexity of an algorithm is given by the number of steps taken by the algorithm to compute the function it was written for.

[NOTE: example on how to calculate computing time will be given later]

STORAGE STRUCTURE FOR ARRAYS

Array is set of homogenous data items represented in contiguous memory locations using a common name and sequence of indices starting from 0. Array is a simplest data structure that makes use of computed address to locate its elements. An array size is fixed and therefore requires a fixed number of memory locations.

Suppose A is an array of n elements and the starting address is given then the location and element I will be

$$\text{LOC}(A_i) = \text{Base address of A} + (i - 1) * W$$

Where W is the width of each element.

A multidimensional array can be represented by an equivalent one-dimensional array. A two dimensional array consisting of number of rows and columns is a combination of more than 1 one-dimensional array. A 2 dimensional array is referred in two different ways. Considering row as major order or column as major order any array may be used to refer the elements.

If we consider the row as major order then the elements are referred row by row whose addressing function may be

$$\text{LOC}(A_{rc}) = \text{Base address of A} + [(r-1) * N + (c-1)] * W$$

Where r and c are subscripts. N is number of columns per row. W is the width of each element.

If we consider the column as major order then the elements are referred column by column.

Sparse Matrices

Matrices with relatively high proportion of zero or null entries are called sparse matrices.

When matrices are sparse, then much space and computing time could be saved if the non-zero entries were stored explicitly i.e. ignoring the zero entries the processing time and space can be minimized in sparse matrices.

$$\begin{pmatrix} 0 & 0 & 0 & 24 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 9 & 0 & 0 \\ 0 & 0 & 0 & 0 & 18 & 0 & 0 \\ 0 & 0 & 0 & 0 & 8 & 0 & 0 \end{pmatrix}$$

In the above matrix we have 6 rows and 7 columns. There are 5 nonzero entries out of 42 entries. It requires an alternate form to represent the matrix without considering the null entries.

The alternate data structure that we consider to represent a sparse matrix is a triplet. The triplet is a two dimensional array having t+1 rows and 3 columns. Where, t is total number of nonzero entries.

The first row of the triplet contains number of rows, columns and nonzero entries available in the matrix in its 1st, 2nd and 3rd column respectively. Second row onwards it contains the row subscript, column subscript and the value of the nonzero entry in its 1st, 2nd and 3rd column respectively.

Let us represent the above matrix in the following triplet of 6 rows and 3 columns

6	7	5
1	4	24
2	6	5
4	5	9
5	5	18
6	5	8

The above triplet contains only non-zero details by reducing the space for null entries.

[Follow the algorithms taught in class]

Stacks

A stack is a linear data structure in which an element may be inserted or deleted only at one end called the top end of the stack i.e. the elements are removed from a stack in the reverse order of that in which they were inserted into the stack.

A stack follows the principle of last-in-first-out (LIFO) system. According to the stack terminology, PUSH and POP are two terms used for insert and delete operations.

Representation of Stacks

A stack may be represented by means of a one way list or a linear array. Unless, otherwise stated, each of the stacks will be maintained by a linear array STACK, A variable TOP contains the location of the top element of the stack. A variable N gives the maximum number elements that can be held by the stack. The condition where TOP is NULL, indicate that the stack is empty. The condition where TOP is N, will indicate that the stack is full.

[Follow the push and pop algorithms discussed in class]

Application of Stacks

There are two important applications of stacks.

a) Recursion

b) Arithmetic Expression

Recursion

Recursion is an important facility in many programming languages. There are many problems whose algorithmic description is best described in a recursive manner.

A function is called recursive if the function definition refers to itself or does refer to another function which in turn refers back to the same function. In-order for the definition not to be circular, it must have the following properties:

- (i) There must be certain arguments called base values, for which the function does not refer to itself.
- (ii) Each time the function does refer to itself, the argument of the function must be closer to a base value.

A recursive function with those two properties is said to be well defined.

Let us consider the factorial of a number and its algorithm described recursively:

We know that $N! = N * (N-1)!$

$(N-1)! = (N-1) * (N-2)!$ and so on up to 1.

FACT(N)

1. if $N=1$

 return 1

2. else

 return $N * \text{FACT}(N-1)$

3. end

Let N be 5.

Then according to the definition FACT(5) will call FACT(4), FACT(4) will call FACT(3), FACT(3) will call FACT(2), FACT(2) will call FACT(1). Then the execution will return back by finishing the execution of FACT(1), then FACT(2) and so on up to FACT(5) as described below.

1) $5! = 5 * 4!$

2) $4! = 4 * 3!$

3) $3! = 3 * 2!$

4) $2! = 2 * 1!$

5) $1! = 1$

6) $2! = 2 * 1 = 2$

7) $3! = 3 * 2 = 6$

8) $4! = 4 * 6 = 24$

9) $5! = 5 * 24 = 120$

From above example it is clear that every sub function contain parameters and local variables. The parameters are the arguments which receive values from objects in the calling program and which transmit values back to the calling program. The sub-function must also keep track of the return address in the calling program. This return address is essential since control must be transferred back to its proper place in the calling program. After completion of the sub-function when the control is transferred back to its calling program, the local values and returning address is no longer needed. Suppose our sub-program is a recursive one, when it call itself, then current values must be saved, since they will be used again when the program is reactivated.

Thus, in recursive process a data structure is required to handle the data of ongoing called function and the function which is called at last must be processed first. i.e the data accessed last must be processed first i.e Last in first out principle. So, a stack may be suitable data structure that follows LIFO to implement recursion.

Arithmetic Expression

This section deals with the mechanical evaluation or compilation of infix expression. The stack is found to be more efficient to evaluate an infix arithmetical expression by first converting to a suffix or postfixes expression and then evaluating the suffix expression. This approach will eliminate the repeated scanning of an infix expressions in order to obtain its value.

A normal arithmetic expression is normally called as infix expression. E.g. $A+B$

A Polish mathematician found a way to represent the same expression called Polish notation or prefix expression by keeping operators as prefix. E.g. $+AB$

We use the reverse way of the above expression for our evaluation. The representation is called Reverse Polish Notation (RPN) or postfixes expression. E.g. $AB+$

The arithmetic expression evaluation is performed in two phases, they are

☞ Conversion of infix to postfixes expression

☞ Evaluation of postfixes expression

[Follow the algorithms and examples discussed in class]

Queue

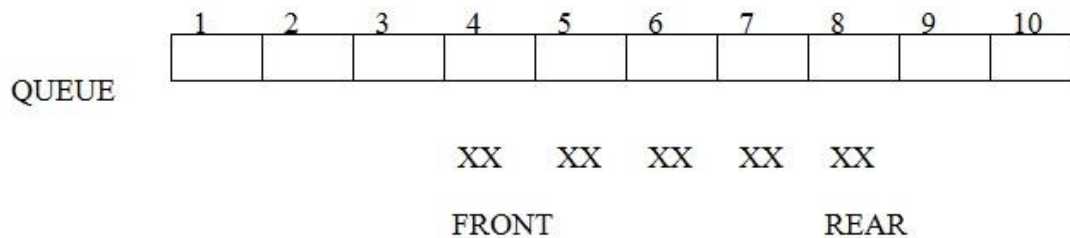
Queue is a linear data structure in which insertion can take place at only one end called rear end and deletion can take place at other end called top end. The front and rear are two terms used to represent the two ends of the list when it is implemented as queue. Queue is also called First In First Out (FIFO) system since the first element in queue will be the first element out of the queue.

Like stacks, queues may be represented in various ways, usually by means of one way list or linear arrays. Generally, they are maintained in linear array QUEUE. Two pointers FRONT and REAR are used to represent front and last element respectively. N may be the size of the linear array. The condition when FRONT is NULL indicates that the queue is empty. The condition when REAR is N indicates overflow.

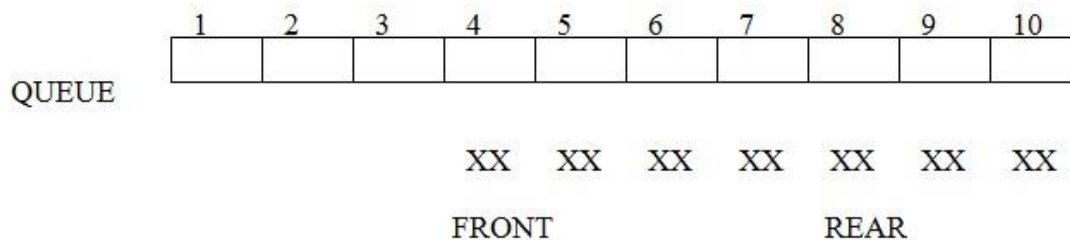
[Follow the algorithms for insert and delete discussed in class]

Circular Queue

The linear arrangement of the queue always considers the elements in forward direction. In the above two algorithms, we had seen that, the pointers front and rear are always incremented as and when we delete or insert element respectively. Suppose in a queue of 10 elements front points to 4th element and rear points to 8th element as follows.



When we insert two more elements then the array will become



Later, when we try to insert some elements, then according to the logic when REAR is N then it encounters an overflow situation. But there are some elements are left blank at the beginning part of the array. To utilize those left over spaces more efficiently, a circular fashion is implemented in queue representation. The circular fashion of queue reassigns the rear pointer with 1 if it reaches N and beginning elements are free and the process is continued for deletion also. Such queues are called Circular Queue.

[Follow the CQINSERT and C DELETE algorithm]

Types of QUEUE

There are two types of Queue

📁 Priority Queue

📁 Double Ended Queue

Priority Queue

A priority queue is a collection of elements such that each element has been assigned a priority value such that the order in which elements are deleted and processed comes from the following rules

1. An element of higher priority is processed before any element of lower priority.
2. Two elements with the same priority are processed according to the order in which they were added to the queue.

There are various ways of maintaining a priority queue in memory. One is using one way list. The sequential representation is never preferred for priority queue. We use linked Queue for priority Queue.

Double Ended Queue

A Double Ended Queue is in short called as Dequeue (pronounced as Deck or dequeue). A dequeue is a linear queue in which insertion and deletion can take place at either ends but not in the middle.

There are two types of Dequeue.

1. Input restricted Dequeue
2. Output restricted Dequeue

A Dequeue which allows insertion at only at one end of the list but allows deletion at both the ends of the list is called Input restricted Dequeue.

A Dequeue which allows deletion at only at one end of the list but allows insertion at both the ends of the list is called Output restricted Dequeue.

Garbage Collection and Compaction

Garbage collection is the process of collecting all unused nodes and returning them to available space. This process is carried out in two phases:

In first phase, known as marking phase, all nodes in use are marked.

In second phase, all unmarked nodes are returned to the available space list.

The second phase is trivial when all nodes are of a fixed size. In this case, the second phase requires only the examination of each node to see whether or not it has been marked. In this situation it is only the first or marking phase that is of any interest in designing algorithm. When variable size nodes are in use, it is desirable to compact memory so that all free nodes form a contiguous block of memory. In this case, the second phase is referred to as memory compaction. Compaction of disk space to reduce average retrieval time is desirable even for fixed size.

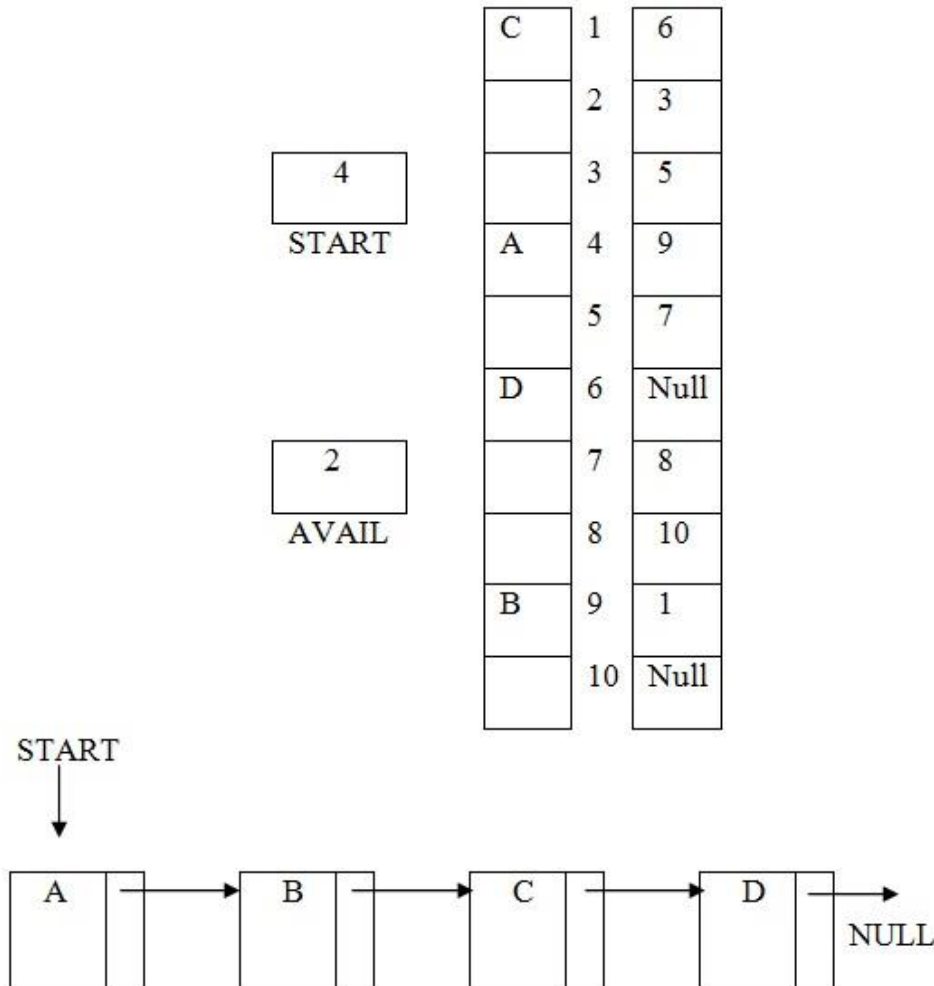
Linked List (One way List)

We understand that the sequential representation of the ordered list is expensive while inserting or deleting arbitrary elements stored at fixed distance in a fixed memory.

The linked representation reduces the expense because the elements are not stored at fixed distance and they are represented randomly and the operations such as insertion and deletion are required change in link rather than movement of data.

A linked list is a linked representation of the ordered list. It is a linear collection of data elements termed as nodes whose linear order is given by means of link or pointer. Every node consists of two parts. The first part is called INFO, contains information of the data and second part is called LINK, contains the address of the next node in the list. A variable called START, always points to the first node of the list and the link part of the last node always contains null value. A null value in the START variable denotes that the list is empty.

INFO LINK



Along with the linked list in the memory, a special list is maintained which consists of list of unused memory cells or unused nodes. This list is called list of available space or availability list or list of free storage or free storage list or free pool. A variable AVAIL is used to store the starting address of the availability list.

Sometimes, during insertion, there may not be available space for inserting a data into a data structure, then the situation is called **OVERFLOW**. Programmers generally handle the situation by checking whether AVAIL is NULL or not.

The situation where one wants to delete data from a data structure that is empty is called **UNDERFLOW**. The situation is encountered when START is NULL.

Header Linked List :

A header linked list is a linked list, which always contains a special node called the header node at the beginning of the list. The header node contains the overall information of the list, which is frequently required for many operations and useful while looking for such information. There are two kinds of header lists.

- a) A *grounded header list* is a header list where the last node contains the null pointer.
- b) A *circular header list* is a header list where the last node points back to the header node.

Circular Linked List :

A linked list is called circular if the last node contains the address of first node or header list. The advantage of circular linked list is it requires minimum time to traverse the nodes which are already traversed, without moving to starting node.

Linked Stack :

The problems with array-based stacks are that the size must be determined at compile time. Instead, let's use a linked list, with the stack pointer pointing to the top element, let fresh be the new node. To push a new element on the stack, we must do:

fresh->next = top;

top = fresh;

To pop an item from a linked stack, we just have to reverse the operation.

p = top;

top = top->next;

Linked Queues

Queues in arrays were ugly because we need wrap around for circular queues. Linked lists make it easier. We need two pointers to represent our queue - one to the *rear* for *enqueue* operations, and one to the *front* for *dequeue* operations.

Note that because both operations move forward through the list, no back pointers are necessary!

Application of Linked List:

1) Polynomial Manipulation:

A polynomial has multiple terms with same information such as coefficient and powers. Each term of a polynomial is treated as a node of a list and normally a linked list used to represent a polynomial. The implementation of polynomial addition is the only operation that is discussed many place. Multiplication of polynomials can be obtained by performing repeated additions.

Each polynomial is stored in decreasing order of by term according to the criteria of that polynomial. i.e. The term whose powers are more are stored at first node and the least power term is stored at last. This ordering of polynomials makes the addition of polynomials easy. In fact two polynomials can be added or multiplied by scanning each of their terms only once.

[Follow the algorithms discussed in class]

2) Linked Dictionary:

An important part of any compiler is the construction and maintenance of a dictionary containing names and their associated values. Such dictionary is also called Symbol Table. There may be several symbols corresponding to variable names, labels, literals, etc.

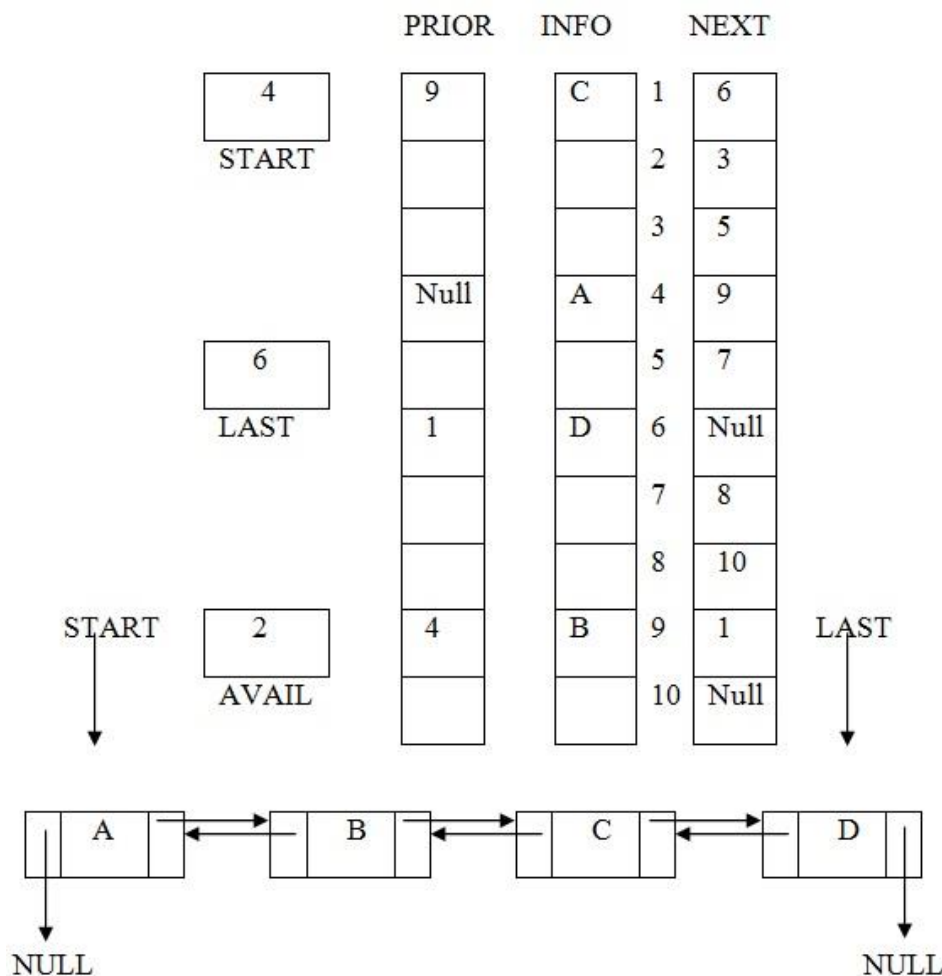
The constraints, which must be considered in the design of the symbol tables, are processing time and memory space. There are many phases associated with the construction of symbol tables. The main phases are building and referencing.

It is very easy to construct a very fast symbol table system, provided that a large section of memory is available. In such case a unique address is assigned to each name. The most straightforward method of accessing a symbol table is linear search technique. This method involves arranging the symbols sequentially in memory via an array or by using a simple linked list. An insertion can be easily handled by adding new element to the end of the list. When it is desired to access a particular symbol, the table is searched sequentially from its beginning until it is found. It will take $n/2$ comparisons to find a particular symbol. The insertion mechanism is fast but the referencing is extremely slow. The referencing will be fast if we use binary search technique. To implement a binary search on symbol table a tree representation is used.

Double Linked List (Two way List)

Since the single linked list contains only one single pointer that points to the next of the linked list, there is only one way traversal. So, the reverse direction is not possible, in the single linked list.

For bi-directional movement, a two-way list or double linked list is considered. IT is a linear collection of data elements, called nodes. Where each node is divided into three parts: INFO, PRIOR, and NEXT. The INFO part contains the information of the node and PRIOR and NEXT are the pointers refers to predecessor node and successor node address respectively. The list also has two pointers: START and LAST. START points to the first node of the list where as LAST points to last node of the list. The PRIOR part of the first node and NEXT part of the last node contains always null value.



Difference between single linked list and double linked list :

The Single linked list has only one advantage, that it can traverse a list in one direction. That means one cannot get the address of its predecessor node. i.e. When we look for any previous information of the list during operations then one has to traverse again from the start node of the one way list. Which uses an extra pointer and additional searching time. But in case double linked list we can have the address of the next as well as previous node. So, while we look for previous node address, we can obtain through prior part of the two-way list which need not require extra pointer or takes less time than that of the single linked list. So apart from the bi-directional movement facility, the two-way list also saves the time and space during traversal operation.

TREES

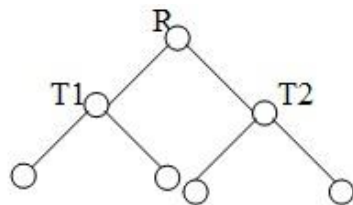
A tree is a nonlinear data structure and is generally defined as a nonempty finite set of elements, called nodes such that:

1. T contains a distinguished node called *root* of the tree.
2. The remaining elements of tree form an ordered collection of zero or more disjoint subsets called sub tree.

BINARY TREE:

A binary tree is defined as a finite set of elements, called nodes, such that:

- 1) Tree is empty (called the null tree or empty tree) or
- 2) Tree contains a distinguished node called root node, and the remaining nodes form an ordered pair of disjoint binary trees



In the above tree R is the root node and T1 and T2 are called sub-trees. T1 and T2 are left and right successor of R. The node R is called parent node and T1 and T2 are called children. All lower level nodes are called *descendants* and upper level nodes are called *ancestors* of their descendants. The line drawn between parent and child is called an *edge* or *arc* where as the line(s) between and ancestor and descendant is called *path*. A node without any

children is called a *terminal or leaf node* and all others are called *non-terminal or non-leaf node*. A path ending with a leaf is called a branch.

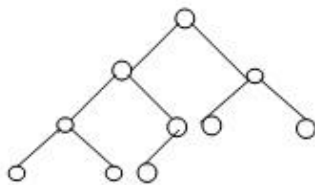
Each node in a binary tree is assigned a level number, as follows. The root node is assigned the level number 0, and every other node is assigned a level number, which are 1 more than the level number of its parent. The nodes of same level number are said to belong to same *generation*. Nodes of same parent are called *siblings*.

The *depth or height* of a tree is the maximum level number of the tree or maximum number of nodes in a branch of a tree.

Two trees are said to be *similar* if they have the same structure and are said to be *copies* if they are similar and if they have same contents at corresponding nodes.

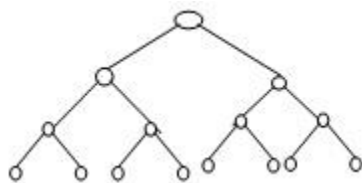
Complete Binary Tree:

A binary tree is said to be complete if all its level except possibly the last, have maximum number of possible nodes, and if all the nodes at the last level appear as far left as possible.



Full binary tree:

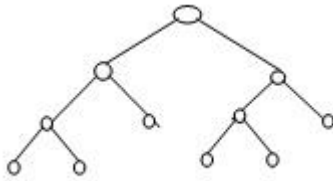
A binary tree said to be full if all its level have maximum number of possible node.



Extended Binary Tree (Strictly Binary Tree or 2-tree):

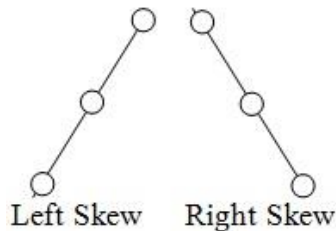
A binary tree is said to be Extended binary tree if each node has either 0 or 2 children. In this case the leaf nodes are called

external nodes and the node with two children are called internal nodes.



Skewed Tree:

A tree is called Skew if all the nodes of a tree are attached to one side only. i.e A left skew will not have any right children in its each node and right skew will not have any left child in its each node.



Binary Search Trees:

A tree is called binary search tree if each node of the tree has following properties.

The value at a node is greater than every value in the left sub-tree and is less than every value in the right sub-tree.

Heap:

A binary tree is also called a heap and there are two types of heap. They are Max Heap and Min Heap. A heap is called maximum heap if value of a node is greater than or equal to each of its descendant node. A heap is called minimum heap if value of a node is less than or equal to each of its descendant node.

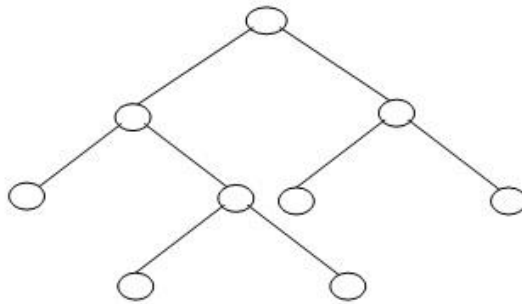
Representation of Binary Search Tree:

Sequential Representation:

The sequential representation of tree stores data in an array as per the following rules:

1. The root node is stored in 1st position.
2. Every left and right child of a parent node at location k will be stored in $(2 \times K)^{\text{th}}$ position and $(2 \times K + 1)^{\text{th}}$ position respectively.

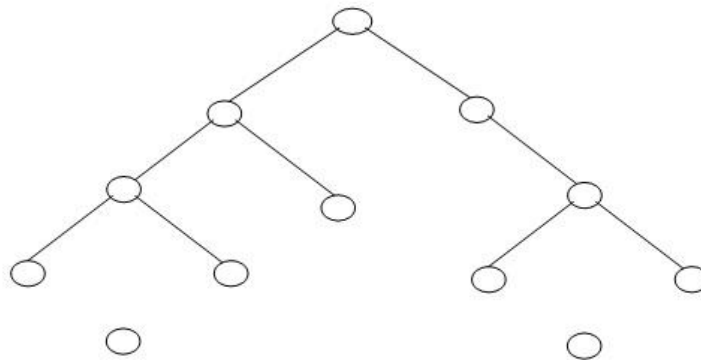
The following example shows the representation of binary tree in an array.



F	B	H	A	D	G	I			C	E					
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

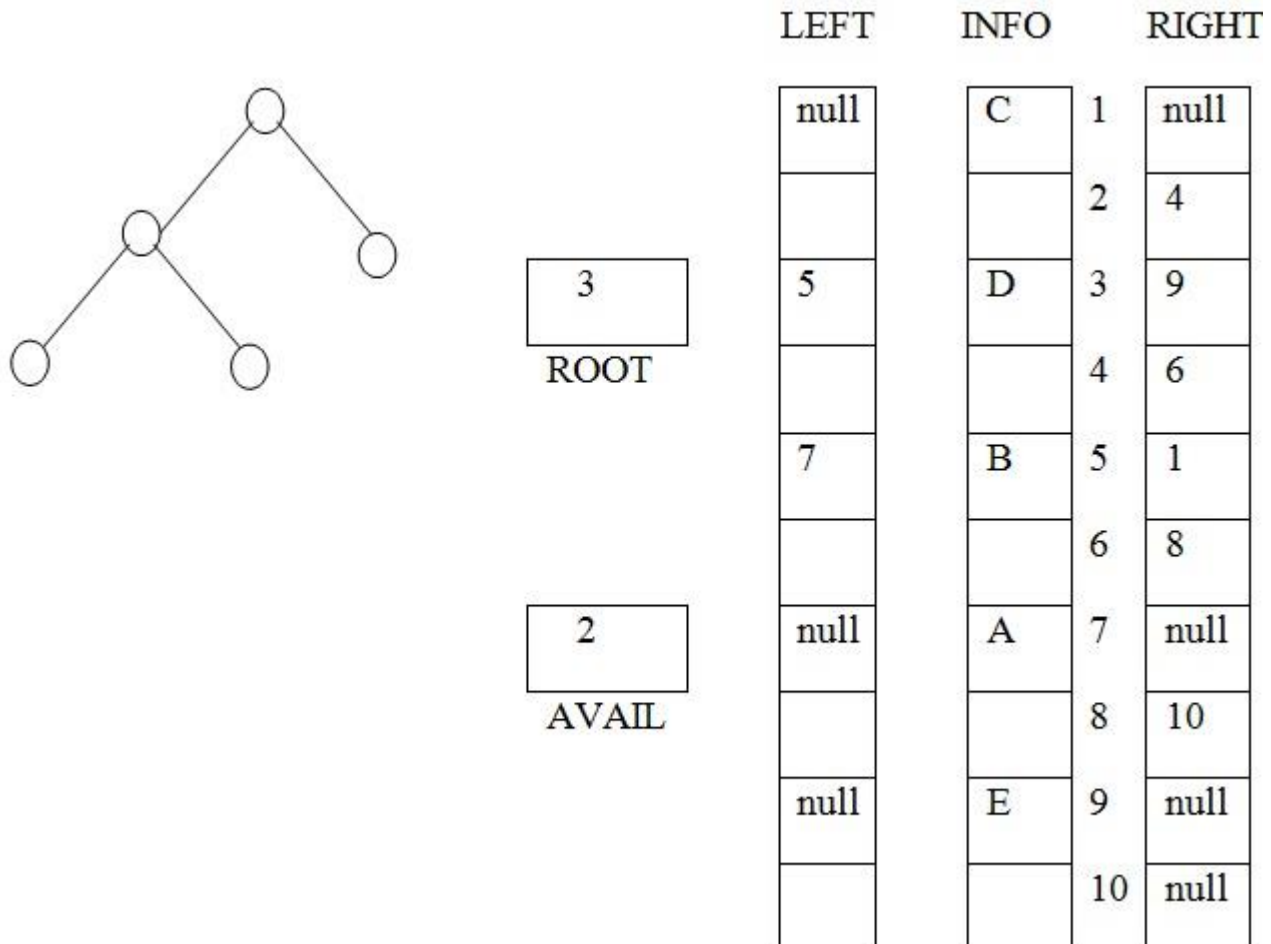
Suppose an array is representing a tree then its tree representation will be drawn using the same rule and an example is shown bellow.

A	B	C	D	E		F	G	H					I	J	
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16



Linked Representation of Tree:

The Linked representations of tree, maintains three parallel arrays. An INFO array contains the data of each node, LEFT array contains the location of left child and RIGHT array contains location of right child. A ROOT pointer points to the root node of the tree.



Header Nodes:

When a binary tree is maintained in memory by means of a linked representation. Sometimes an extra, special node, called a header node, is added to the beginning of the tree. When this extra node is used, the tree pointer variable, which is called HEAD, will point to the header node, and the left pointer of the header node will point to the root. Generally, the header node of any tree contains the general or overall information of the tree, which is required frequently in the operation of the tree. For example, number of employees present in employee tree, number nodes present in a tree, cumulative values of all the nodes etc. so that while accessing such information the nodes of the tree need not be traversed.

Threads and Threaded Binary Tree:

Approximately half of the entries in the pointer fields Left and right of any binary tree contains null elements. Replacing the null entries by some other type of information may more efficiently use this space. Specifically, we will replace certain null entries by special pointers, which point to nodes higher in the tree. These special pointers are called threads and the tree is called threaded binary tree.

There are many ways to thread a binary tree, but each threading will correspond to a particular traversal of tree. Unless otherwise stated, threading will correspond to in-order traversal.

There are two types of threading:

📁 One way threading

📁 Two way threading

In one way threading, either left pointer or right pointer will be used for threading. When left pointer is used to point the predecessor node of the tree according to in-order traversal, then the threading is called left-in threading. When a right pointer is used to point the successor node according to in-order traversal, then the threading is called right-in threading.

In two way threading both left and right pointers are used to point predecessor and successor nodes of the tree according to in-order traversal.

Height Balanced Tree(AVL Tree):

Adelson-Velskii and Land in 1962 introduced a binary tree structure that is balanced with respect to heights of the sub-trees. As a result of the balanced nature of this type of tree, dynamic retrievals can be performed in less time. At the same time an identifier may be inserted and deleted in that tree in less time.

Definition: A tree is called height balanced. If the tree is nonempty binary tree T with T_L and T_R as its left and right sub-trees, then tree is called height balanced iff

- a) $h_L - h_R$ is -1 , or 0 , or 1 where h_L and h_R are heights of left sub-tree T_L and right sub-trees T_R respectively.
- b) T_L and T_R are height balanced.

Generally, while inserting or deleting an identifier in tree we balance the tree. Balancing of a tree is carried out using essentially two kind of rotation left rotation and right rotation. When a tree at a node has Balance Factor less -1 then the tree at that node is considered to be right heavy. To balance the right heavy tree the tree at that node rotated towards left. Similarly, if the tree is at a node is having Balance Factor more than 1 will be considered as left heavy. To balance the left heavy tree, the tree is rotated towards right at that node.

Application of Binary Tree:

1.Symbol Table Construction:

The notion of symbol table arises frequently in computer science while building compilers, loaders, linkers, assemblers etc. A symbol table is a set of name-value pairs. Associated with each name in the table is an attribute, a collection of attributes, or some directions about what further processing is needed. One of the criteria that a symbol table routine must meet is that the table searching must be performed efficiently. This requirement originates in the compilation phase while handling many lexemes and tokens of the program. The three required operation of the symbol table are:

- a) Insertion of new entry
- b) Deletion of existing entry
- c) Looking up information of an existing entry.

Each of above operation requires searching.

Generally, a tree is used to construct a symbol table because

- a) if the symbol table entries as encountered are uniformly distributed according to lexicographic order, then table searching becomes approximately equivalent to a binary search, as long as the tree is maintained in lexicographic order.
- b) A binary tree is easily maintained in lexicographic order.

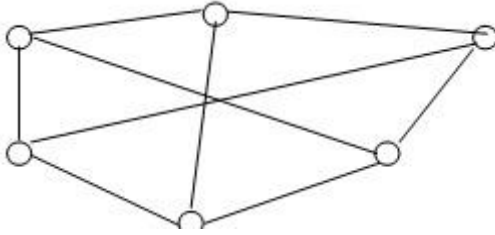
2)Manipulation of the Arithmetic Expressions:

We observed that the formulas in Reverse polish notation are very useful in the compilation process. There is a close relationship between binary trees and formulas in prefix or suffix notations. Let us write the infix formula as a binary tree where a node has an operator as a value and where the left and right sub-trees are the left and right operands of that operator. The leaves of the tree are the variables and constants of the expression. We represent the expression in binary tree due to similarities of infix to in order and postfixes to positor traversal of tree. The tree used for expression is called parse tree.

[Follow the conversion process taught in class room]

Graphs

A Graph is a nonlinear data structure, which is having point to point relationship among the nodes. Each node of the graph is called as a vertex and link or line drawn between them is called and edge.



Mathematically, A graph 'G' consists of two sets V and 'E' such that $G = \{V, E\}$

Where V is finite nonempty set of vertices or nodes. $V(G)$ represents set of vertices.

And E is a set of edges. $E(G)$ represents set of Edges.

According to above example, $V(G) = \{1, 2, 3, 4, 5, 6\}$

$$E(G) = \{(1, 2), (2, 1), (1, 4), (4, 1), \dots\}$$

Suppose edge $e = \{u, v\}$, then the nodes u and v are called end points of the edge e. The node u is called source node and node v is called destination node, the nodes u and v are called *adjacent nodes*. The line drawn between to adjacent nodes is called an edge.

If an edge is having direction, then the source node is called *adjacent to* the destination and destination node is *adjacent from* source.

Path: A path is a sequence of consecutive edges between a source and a destination through different nodes. A path, said to be *closed* if source is equal to destination. The path is said to be *simple* if all nodes are distinct.

Cycle: A cycle is closed path with length 3 or more. A cycle of length k is called a k-cycle.

Loop : If an edge is having identical end points, then the edge is called a loop.

Degree/order: A degree of a node is the number of edges containing that node. The number edges pointing towards the node are called *in-degree/in-order*. The number edges pointing away from the node are called *out-degree/out-order*.

A graph in which the edges are having direction is called **directed graph or digraph**, otherwise the graph is called **undirected graph**.

Isolated node: If degree of a node is zero i.e. if the node is not having any edges, then the node is called isolated node.

Complete Graph : A graph is called complete if all the nodes of the graph are adjacent to each other. A complete graph with n nodes will have $n*(n-1)/2$ edges.

Weighted Graph : A graph is said to be weighted if each edge in the graph is assigned a non-negative numerical value called the weight or cost of the edge. If an edge does not have any weight then the weight is considered as 1.

Multigraph : If a graph has two parallel path to an edge or multiple edges along with a loop is said to be multigraph.

Representation of Graph:

A graph may be represented in two ways. They are

Sequential Representation and Linked Representation.

Sequential Representation:

The graph is represented in sequential memory using two matrices. They are

Adjacency Matrix and Path Matrix

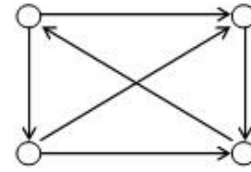
Adjacency Matrix:

Suppose G is a graph with n nodes and the nodes of G are being ordered and are called $v_1, v_2, v_3, \dots, v_n$ then the adjacency matrix $A=(a_{ij})$ of the graph G is defined as

$$a_{ij} = \begin{cases} 1 & \text{if } v_i \text{ is adjacent to } v_j \\ 0, & \text{otherwise} \end{cases}$$

The adjacency matrix with 1's and 0's is also called bit matrix.

$$A = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$



Path Matrix or Reach-ability Matrix :

Suppose G is a graph with n nodes and the nodes of G are being ordered and are called $v_1, v_2, v_3, \dots, v_n$ then the Path matrix $P=(p_{ij})$ of the graph G is defined as

$$p_{ij} = \begin{cases} 1, & \text{if there is a path between } v_i \text{ and } v_j \\ 0, & \text{otherwise} \end{cases}$$

Adjacency matrix A is a path of length 1. Similarly, $A^2, A^3, A^4, \dots, A^n$ are the path matrix of length 2, 3, 4, ..., n respectively. Then before calculating path matrix the matrix B_n will be calculated to find P .

$$\text{Where } B_n = A^2 + A^3 + A^4 + \dots + A^n$$

All non-zero elements of B_n are replaced with 1 to form path matrix P .

Linked Representation:

The Matrix representation of graph does not keep track of the information related to the nodes. Hence a linked representation is used to represent a graph called adjacency structure. The adjacency structure of the graph maintains two lists called node list and edge list.

Node List: Each node in the node list will correspond to a node in the graph and will have three fields. They are the information of the node called INFO, Pointer to the next node of the list called NEXT, a pointer to the edge list called ADJ.

Edge List: Each element of the edge list will correspond to an edge of the graph and will have two fields. They are DEST contains the address of the destination node and LINK contains the address of the next node of the edge list.

[Follow the algorithm taught in class for node and edge insertion and deletion]

Graph Traversal:

Traversing a graph means visiting all the vertices in a graph exactly one. It is of two types:

Breadth First Traversal and Depth First Traversal.

Breadth First Traversal:

The traversal starts at a node v , after marking the node the traversal visits all incident edges to node v after marking the nodes and then moving to an adjacent node and repeating the process. The traversal continues until all unmarked nodes in the graph have been visited.

A queue is maintained in the technique to maintain the list of incident edges and marked nodes. It is more appropriate for a digraph.

Depth First Traversal:

A DEPTH FIRST SEARCH OF AN ARBITRARY GRAPH CAN BE USED TO PERFORM A TRAVERSAL OF A GENERAL GRAPH. THE TECHNIQUE PICKS UP A NODE AND MARKS IT. AN UNMARKED ADJACENT NODE TO PREVIOUS NODE IS THEN SELECTED AND MARKED, BECOMES THE NEW START NODE, POSSIBLY LEAVING THE PREVIOUS NODE WITH UNEXPLORED EDGES FOR THE PRESENT. THE TRAVERSAL CONTINUED RECURSIVELY, UNTIL ALL UNMARKED NODES OF THE CURRENT PATH ARE VISITED. THE PROCESS IS CONTINUED FOR ALL THE PATHS OF THE GRAPH.

HASHING

Hashing is a searching technique which is key to address transformation technique. The normal linear and binary search technique, searches for a key via sequence of comparisons. Hashing differs from this in that the address or location of an identifier X , is obtained by computing some arithmetic function, f of X , $f(x)$ gives the address of X in the table. This address will be referred to as the hash or home address of X . Depending on the address yielded by the function the data are stored in sequential memory location, called hash table.

Hash Table:

The memory available to maintain the symbol table is assumed to be sequential. This memory is referred to as the hash table HT. The hash table is partitioned into b buckets, $HT(0)$, $HT(1)$, ..., $HT(b-1)$. Each bucket is divided into S slots and each slot is capable of holding a records. Thus, a bucket is said to consist of s slots, each slot being large enough to hold 1 record. Usually $s=1$ and each bucket can hold exactly 1 record. A hashing function,

$f(x)$, is used perform an identifier transformation on X . $f(x)$ maps the set possible identifier on to the integers 0 through $b - 1$.

The ratio n/T is the **identifier density**, while $n/(s*b)$ is the loading density or loading factor.

Where n is the number of identifiers ,

b is number of buckets,

T is total number of possible identifiers

s is number of slots.

HASHING FUNCTION-

A hashing function, f , transforms an identifier X into a bucket address in the hash table .As mentioned earlier the desired properties of such a function are that it be easily computable and that it minimize the number of collisions.

Since many programs use several identifiers with the same first letter, we would like the function to depend upon all the characters in the identifiers in addition, we would like the hash function to be such that it does not result in a biased use of the hash table for random inputs.

Several kinds of uniform hash functions are in use.

1 . Division 2. Mid-square 3 .Folding 4. Digit Analysis

Only division method is used frequently and is most preferred one.

Division Method:

This is the most common method used for hash function. The function is used to find a number may be prime or it is number of buckets. Then the number will be used to divide the key by it. The remainder is the hash address for that key. For example let us consider a hash table of 10 buckets and try to find the address of following values.

34, 56, 89, 432, 87, 651

the home address of 34 will be $34\%10 = 4$

The home address of 56 will be $56\%10 = 6$

And so on for others as mentioned in the table

	KEY	INFO
0		
1	651	XX
2	432	XX
3		
4	34	XX
5		
6	56	XX
7	87	XX
8		
9	89	XX

Some times two different keys may yield same hash address. The there will be collision between the keys. There are few techniques for resolving the collision.

Collision Resolution Technique:

When there is a collision, then a random rehashing function is used to resolve the collision. The efficiency of collision resolution procedure is measured by the average number of probes(key comparisons) needed to find the location of the record with the given key.

Normally the collision is resolved by dividing the each bucket into multiple slots. So, that the keys of same address can be kept in different slots of same bucket. There are two different ways to resolve the collision.

Open Addressing and Chaining.

The open addressing is uses a sequential representation for hash table like two dimensional or three dimensional array. The chaining concept uses a linked representation for each bucket and each bucket is linked with linked list maintaining the slots of that bucket.

Permutations

The permutation problem is as follows: Given a list of items, list all the possible orderings of those items.

We typically list permutations of letters. For example, here are all the permutations of CAT:

CAT

CTA

ACT

ATC

TAC

TCA

There are several different permutation algorithms, but since recursion an emphasis of the course, a recursive algorithm to solve this problem will be presented. (Feel free to come up with an iterative algorithm on your own.)

The idea is as follows:

In order to list all the permutations of CAT, we can split our work into three groups of permutations:

1) Permutations that start with C.

2) Permutations that start with A.

3) Permutations that start with T.

The other nice thing to note is that when we list all permutations that start with C, they are nothing but strings that are formed by attaching C to the front of ALL permutations of "AT". This is nothing but another permutation problem!!!

Number of recursive calls

Often times, when recursion is taught, a rule of thumb given is, "recursive functions don't have loops." Unfortunately, this rule of thumb is exactly that; it's not always true. An exception to it is the permutation algorithm.

The problem is that the number of recursive calls is variable. In the example on the previous page, 3 recursive calls were needed. But what if we were permuting the letters in the word, "COMPUTER"? Then 8 recursive calls (1 for each possible starting letter) would be needed.

In essence, we see the need for a loop in the algorithm:

for (each possible starting letter)

list all permutations that start with that letter

What is the terminating condition?

Permuting either 0 or 1 element. In the code that will be presented in this lecture, the terminating condition will be when 0 elements are being permuted. (This can be done in exactly one way.)

Use of an extra parameter

As we have seen, recursive functions often take in an extra parameter as compared to their iterative counterparts. For the permutation algorithm, this is also the case. In the recursive characterization of the problem, we have to specify one more piece of information in order for the chain of recursive calls to work. Here is what our function prototype, pre-conditions and post-conditions will look like:

// Pre-condition: str is a valid C String, and k is non-negative

// and less than or equal to the length of str.

// Post-condition: All of the permutations of str with the first k

```
// characters fixed in their original positions  
  
// are printed. Namely, if n is the length of str,  
  
// then (n-k)! permutations are printed.  
  
void Recursive Permute(char str[], int k);
```

Utilizing this characterization, the terminating condition is when k is equal to the length of the string str, since this means that all the letters in str are fixed. If this is the case, we just want to print out that one permutation.

Otherwise, we want a for loop that tries each character in index k. It'll look like this:

```
for (j=k; j<strlen(str); j++) {  
  
    Exchange Characters(str, k, j);  
  
    Recursive Permute(str, k+1);  
  
    Exchange Characters(str, j, k);  
  
}
```

where Exchange Characters swaps the two characters in str with the given indexes passed in as the last two parameters. The whole function is included on the next page.

```
void Recursive Permute(char str[], int k) {  
  
    int j;  
  
    // Base-case: All fixed, so print str.  
  
    if (k == strlen(str))  
        printf("%s\n", str);  
  
    else {  
  
        // Try each letter in spot j.
```

```
for (j=k; j<strlen(str); j++) {  
    // Place next letter in spot k.  
    Exchange Characters(str, k, j);  
    // Print all with spot k fixed.  
    Recursive Permute(str, k+1);  
    // Put the old char back.  
    Exchange Characters(str, j, k);  
}  
}  
}
```

Iterative Permutation Algorithm - Background

Another algorithm that cycles through permutations goes through each of them in lexicographical ordering. Roughly speaking, lexicographical ordering is the same as alphabetical ordering. To determine which of two permutations should appear first in a lexicographical ordering, start comparing individual items from the left until you hit a difference. The permutation that should come first is the one with the item that comes earlier when comparing the two different items from the two different permutations.

For example, when comparing permutations of ACT, we find that CAT comes before CTA, because A comes before C. For a numerical example, the permutation 4,6,2,8,3,7,5,1 comes before 4,6,2,8,5,1,3,7, since 3 is smaller than 5 at the spot of the first discrepancy.

Given this definition for comparing two permutations of a set of items, a complete natural ordering is imposed on all the permutations on the list. For example, for the letters A, C, and T, the natural ordering of the permutations, using this definition is as follows:

ACT

ATC

CAT

CTA

TAC

TCA

Similarly, the ordering of the permutations of 1, 2, 3 and 4 are as follows:

1234	2134	3124	4123
1243	2143	3142	4132
1324	2314	3214	4213
1342	2341	3241	4231
1423	2413	3412	4312
1432	2431	3421	4321

In order to come up with an algorithm that iterates through all of the permutations of a set of items in this order, we need to have a successor function. Namely, we need a function that advances an array storing one permutation to the following permutation.

Once we write this successor function, we simply need to start with the first permutation (in this case, 1234 or ACT), and call the successor function the correct number of times. Since there are $n!$ (read, “ n factorial”) orderings of n items, we must call the successor function $n!-1$ times.

Note: We can derive the total number of permutations of n distinct objects as follows:

For the first object, we have n choices.

For the second object, we have $n-1$ choices (all but what we chose for the first object.)

For the third object, we have $n-2$ choices, etc.

Since each of these choices is independent of the rest, to calculate the number of different permutations, we simply need to multiply each of these numbers:

$$n \times (n-1) \times (n-2) \times \dots \times 1$$

This product is so common in mathematics, that it has a special name (factorial) and symbol (!). Symbolically, we have:

$$n \times (n-1) \times (n-2) \times \dots \times 1 = n!$$

Next Permutation Function

Let's examine an example of finding the next permutation of

4,6,2,8,3,7,5,1

and utilize it to come up with a general algorithm.

We know that the fewer items we change on the right the better. The reason is that if we change the 4 to a 5, for example, then we are definitely missing other permutations that might start with 4 that come after the current one. In essence, our goal is to maximize the number of items, starting from the right, that stay fixed.

A real quick inspection will reveal that we can keep 4, 6, 2, and 8 fixed.

The reason is that 3, 7, 5, and 1 can be rearranged to form a "higher" permutation.

BUT, notice that 3 CAN NOT be fixed because it is IMPOSSIBLE to rearrange

7, 5, 1

to create a higher permutation. (This is the highest one since all the values are in descending order.)

Thus, the key to our successor algorithm is to determine the first item, from the left, that has to be changed.

Simply put, all of the items after this item have to be arranged in descending order.

So, here is step #1 of the algorithm:

Scan from the right side of the permutation, going backwards, continue scanning until you find the first pair of values in ascending order. The first value in this pair is the item that will be switched out.

Thus, if our input was 4,6,2,8,3,7,5,1,

We note that (5,1) is descending.

We note that (7,5) is descending.

But, we find that (3,7) is ascending.

Now that we've identified this value, our key is to determine WHICH value to switch into its place.

First, we know that all the values to its left will stay fixed, so we are NOT switching with any of these values. (In our example, that means we won't switch 3 with 4, 6, 2 or 8.)

With this analysis, we have determined the following about the next permutation of 4,6,2,8,3,7,5,1:

- 1) The values, 4, 6, 2, and 8 will be fixed.
- 2) The value 3 must be changed.
- 3) It must be changed to 1, 5, or 7

Next, we know we must switch it with a higher value, otherwise our permutation won't be a higher one. This reduces our list of possible values in our example to 7 and 5. (More generally, these are all the items that appear AFTER our designated item and are larger than it.)

Now, of the possibilities left, we MUST switch it with the lowest value left (5,7 in our example). The reason for this is that any permutation that starts with 5 precedes any permutation that starts with 7. In general, we want the lowest permutation possible of the ones left, and we can achieve this by minimizing our next choice.

Thus, we know that our permutation must start:

4, 6, 2, 8, 5

In doing so, we have exchanged the 3 for the 5, so our array currently looks like this:

4, 6, 2, 8, 5, 7, 3, 1

Now, we can state step #2 of the algorithm:

Determine the smallest value larger than the value to be exchanged in the permutation that comes AFTER the value to be exchanged, and swap these two values.

In our example, 3 was the value that needed to be exchanged and 5 was the smallest value listed after 3 that was also larger than 3.

Now, that we have made that change, we would like to “minimize” the rest of the permutation. For any permutation, we can minimize it by listing the items in ascending order. Currently, however, our items are listed in descending order:

4, 6, 2, 8, 5, 7, 3, 1

In a nutshell, we must simply take all the values that come after our original swapped location in the array, and reverse these values to obtain:

4, 6, 2, 8, 5, 1, 3, 7.

This is the next permutation.

Now, let's look at the steps of the algorithm all together:

Iterative Permutation Algorithm

1. Scan from the right side of the permutation, going backwards, continue scanning until you find the first pair of values in ascending order. The first value in this pair is the item that will be switched out.
2. Scan to the right of the item identified in step 1, looking for the smallest item that is greater than the item identified in step 1. Swap these two items.
3. Reverse the part of the permutation that starts from the original location first identified in the array and ends at the end of the array.

A function that implements this algorithm is located on the next page.

```
void next Perm(int perm[], int length) {  
  
    // Find the spot that needs to change.  
  
    int i = length-1;  
  
    while (i>0 && perm[i] < perm[i-1]) i--;  
  
    i--; // Advance to swap location.  
  
    // So last perm doesn't cause a problem.  
  
    if (i == -1) return;  
  
    // Find the spot with which to swap.  
  
    int j=length-1;  
  
    while (j>i && perm[j]<perm[i]) j--;  
  
    // Swap it.  
  
    int temp = perm[i];  
  
    perm[i] = perm[j];  
  
    perm[j] = temp;  
  
    // reverse from index i+1 to length-1.  
  
    int k,m;  
  
    for (k=i+1,m=length-1; k<m; k++,m--) {  
  
        temp = perm[k];  
  
        perm[k] = perm[m];  
  
        perm[m] = temp;  
  
    }  
}
```

}

SORT

DEFINITION

INTERNAL

SORT

SORT

STRAIGHT

EXOTIC SORTS

VS

DATA

ALGORITHMS

EXTERNAL

BEHAVIOR

STRUCTURES

SORTS

DEFINITION

The process of sorting, or creating a linear ordering of a list of objects, is one of the most fundamental of all operations.

- The objects to be sorted are assumed to be records, one field of which is the sort key.
- The sort problem is to arrange the objects so that the keys form a monotonic (non-decreasing or non-increasing) sequence.

INTERNAL SORTS

An internal sort is one in which the list to be sorted resides in the "internal" memory of the system. In an external sort the data resides on a peripheral device, such as a disk.

We will only deal with internal sorts.

SORT BEHAVIOR



- A sorting technique is said to have natural behavior if it executes faster when the list of objects is already partially or fully ordered.
- It is said to exhibit stable behavior if objects with equal keys are not swapped during the sorting process.

Many sort algorithms exploit these behaviors in order to increase performance.

SORT DATA STRUCTURES

Our default data structure will be an array of integers. However, the only absolute requirement for the data structure is that it be capable of ordering.

Some of the more exotic sorts require an array but the basic sorts will all work with a linked list.

WHICH SORTS?

The sorts of primary interest are:

- **Shell** sort (257)
- **Heap** sort (260)
- **Quick** sort (269)

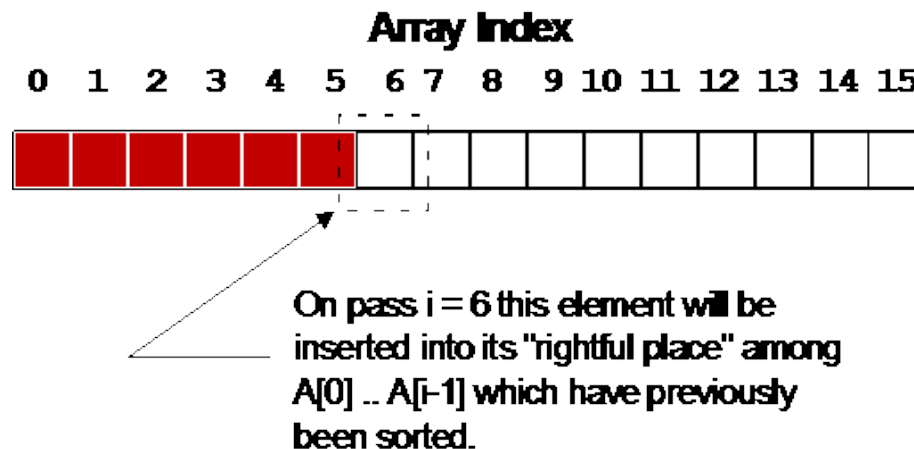
We will first review the "generic" versions of these sorts first.

- The **insertion** sort (254)
- The **selection** sort
- The **exchange** sort

STRAIGHT SORTS

- Straight-forward (or generic) implementations of the basic sort algorithms.
- They typically require a time which is on the order of $O(n^2)$.
- They are well-suited for demonstrating the algorithms.
- They require little code and generally are often faster for small values of n .

INSERTION SORT



The insertion sort works by starting at one end of the list “inserting” each element, p , in its proper place of the first $p+1$ elements.

INSERTION SORT ALGORITHM

for $i := 2$ **to** $array_size$ **do**

begin

$temp := a[i];$

$a[0] := temp;$

```

j := i - 1;

while temp < a [j] do

    begin

        a [j + 1] := a [j];

        j := j - 1;

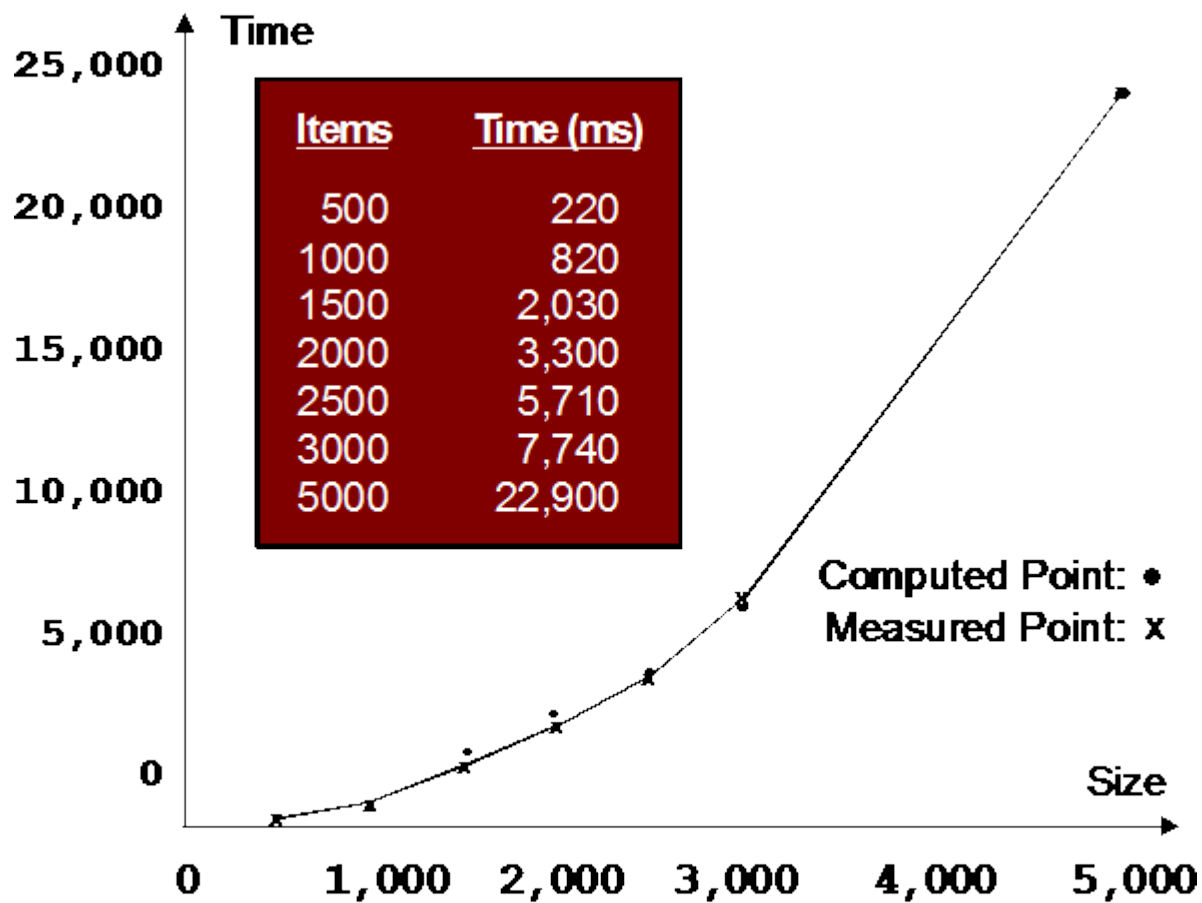
    end;

a [j + 1] := temp

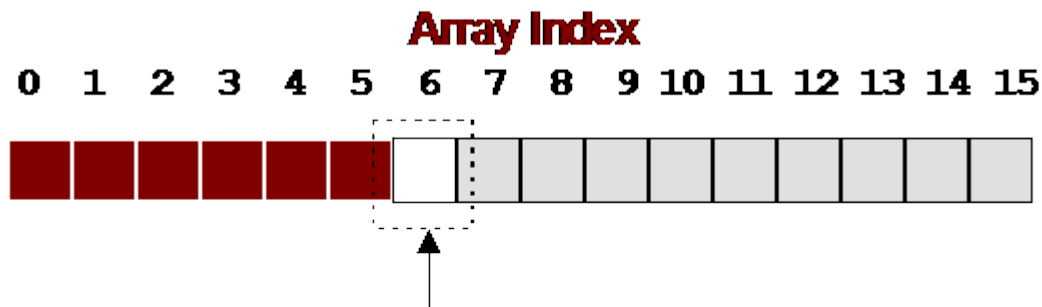
end;

```

INSERTION SORT BEHAVIOR



SELECTION SORT



On pass p the element that is the largest (smallest) of $A[p + 1] \dots A[n]$ will be swapped with element p . Since the previous $p - 1$ elements have been sorted, the largest (smallest) p items now occupy locations $A[0] \dots A[p]$.

In the selection sort we “select” the largest (or smallest) element of the unsorted data on each pass.

SELECTION SORT ALGORITHM

```

for  $i := 1$  to  $array\_size - 1$  do

    begin

         $temp := a[i];$ 

         $k := i;$ 

        for  $j := i + 1$  to  $array\_size$  do

            if  $a[j] < temp$  then

                begin

                     $k := j;$ 

                     $temp := a[j]$ 

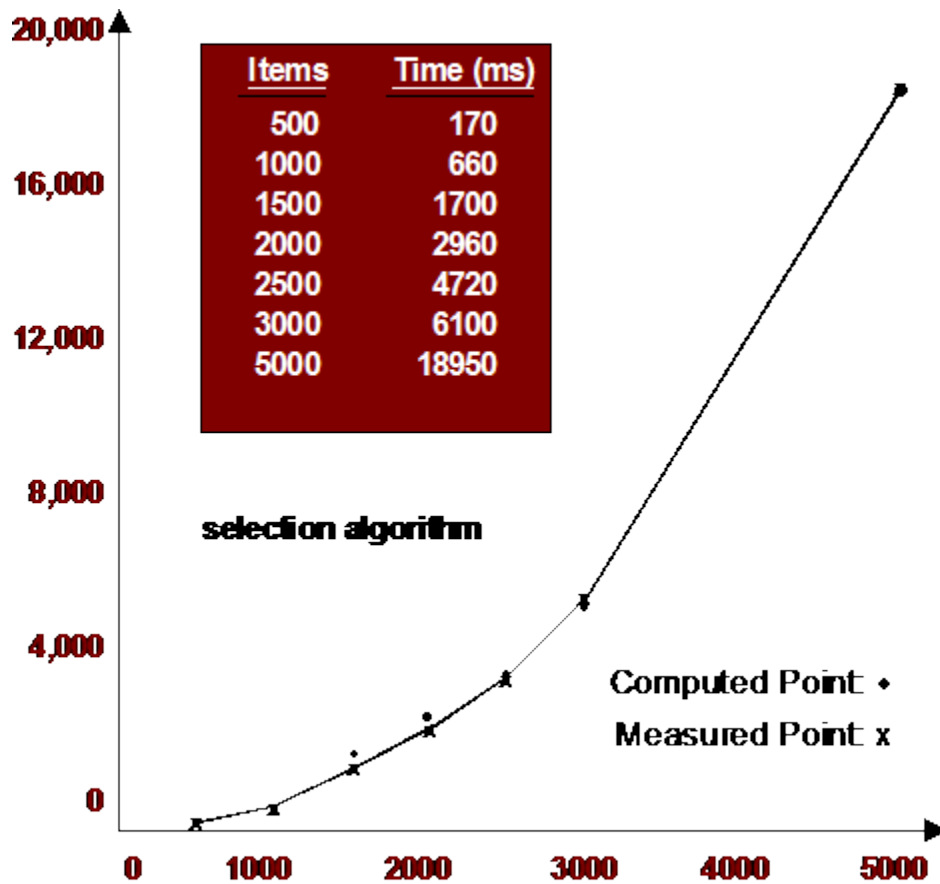
                end;
    
```

$a[k] := a[i];$

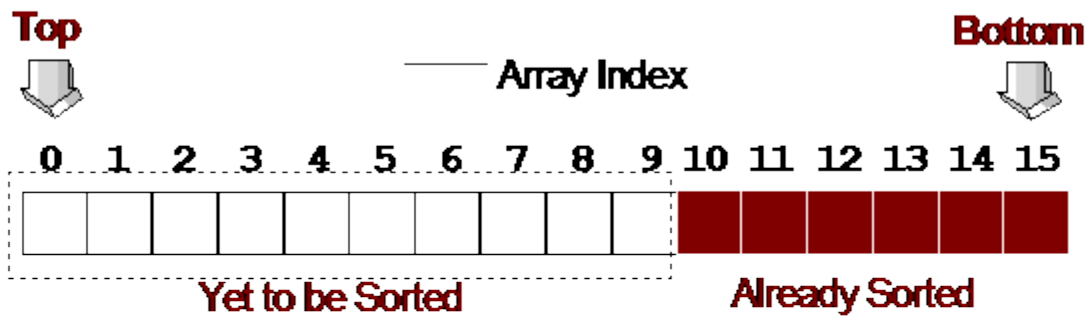
$a[i] := temp$

end;

SELECTION SORT BEHAVIOR



EXCHANGE SORT



On each pass i , each pair of items $A[0] \dots A[n+i]$ is compared and the largest (or smallest) placed in the lower location such that at the end of the pass, the smallest (or largest) i items are in locations $A[n - i + 1] \dots A[n]$. ($i = 6$ is shown).

The exchange sort swaps overlapping pairs so that the largest (smallest) item is at the end of the list each pass.

EXCHANGE SORT ALGORITHM

```

for  $i := 2$  to  $array\_size$  do

    begin

        for  $j := array\_size$  down to  $i$  do

            if  $a[j - 1] > a[j]$  then

                begin

                     $temp := a[j - 1];$ 

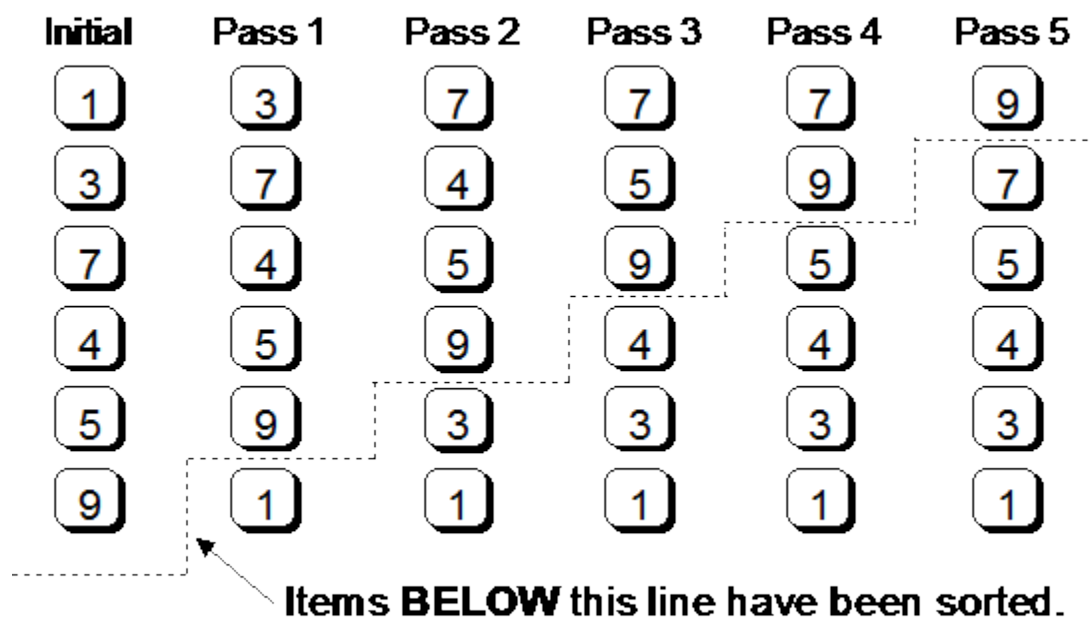
                     $a[j] := temp$ 

                end

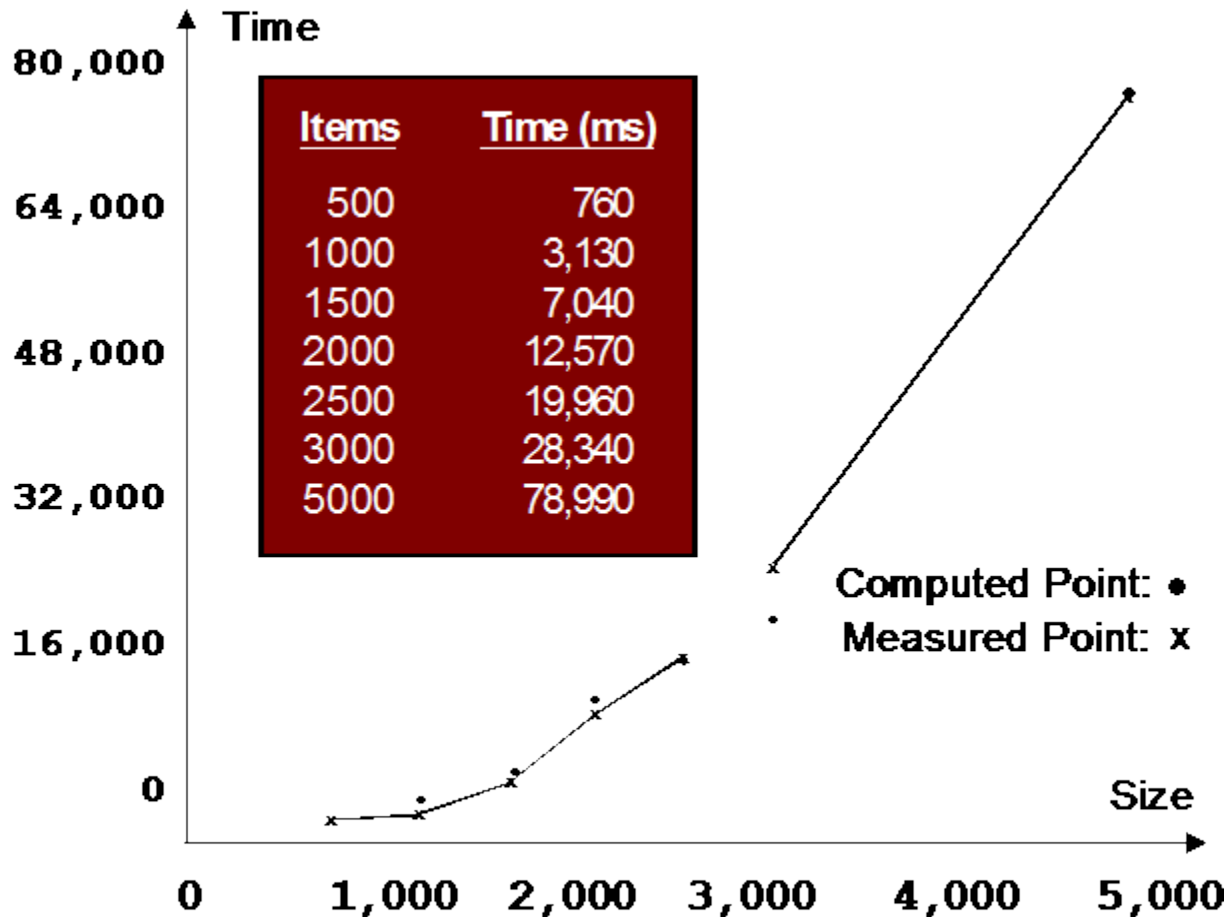
            end;
    
```

EXCHANGE SORT OPERATION

This sort is sometimes called the "bubble sort" because the largest (smallest) items seem to "bubble" up from the bottom as the sort proceeds.



EXCHANGE SORT BEHAVIOR



MODIFIED EXCHANGE SORT

The exchange sort seems to go blindly on, making comparisons even when the data is already ordered, suggesting the following modification:

```
i := 1;
```

```
while (i <= max_item_count) and (swap_flag) do
```

```
begin
```

```
    swap_flag := False;
```

```
    i := i + 1;
```

```
for j := max_item_count down to i do
```

```
    if a [j-1] > a [j] then
```

```
        begin
```

```
            swap_flag := True;
```

```
            temp := a [j-1];
```

```
            a [j-1] := a [j];
```

```
            a [j] := temp
```

```
        end
```

```
    end;
```

THE EXOTIC SORTS

The exotic sorts are refinements of the straight sorts. We shall discuss the following:

- insertion sort ifž shell sort
- selection sort ifž heap sort
- exchange sort ifž quick sort

These sorts all attempt to reduce either the number of comparisons or moves or both by making assumptions about the ordering of the data.

SHELL SORT

The shell sort (named after D. L. Shell) is a refinement of the insertion sort. It is also known as sorting by "diminishing increment". The basis of the shell sort is that it is quicker to make several passes over the array, sorting a subset each time.

2I-SORT TECHNIQUE



For the first pass all items which are n (n must be a power of 2) positions apart are grouped and sorted separately.



For the next pass the items $n/2$ positions apart are sorted until the last pass sorts adjacent items.

SELECTION OF THE INCREMENTS

The Shell sort works for any selection of increments. What is not obvious is that it works better if the increments are not chosen as powers of 2. A number of studies have proposed schemes for choosing the increments. Those given in the following example are by Knuth.

The shell sort is not readily analyzed (many claim it isn't that easy to understand either!) so we will have to depend upon measured performance.

SHELL SORT

h : **array** [Increment Range] of Integer;

begin

$h[1] := 9; h[2] := 5; h[3] := 3; h[4] := 1;$

for $pass := 1$ **to** Num Passes **do**

BEGIN

$k := h[pass]; s := -k;$

for $i := k + 1$ **to** Dictionary Size **do**

begin

$temp := dictionary[i]; j := i - k;$

if $s = 0$ **then**

$s := -k;$

$s := s + 1;$

$dictionary[s] := temp;$

while $temp < dictionary[j]$ **do**

begin

$dictionary[j+k] := dictionary[j]; j := j - k$

end;

$dictionary[j+k] := temp;$

end

END

HEAP SORT

- A refinement of the selection sort.
- Sorting by straight selection is based on the repeated selection of the least key among n items, then among the remaining $n-1$ items, etc.
- Finding the least key among n items requires $n-1$ comparisons,
- The selection sort can be improved by retaining from each scan more information than just the identification of the least item.

USE OF THE HEAP

The heap sort works by first arranging the data into a heap.

A heap is a tree such that the root is greater than or equal to the largest of its children.

Since the largest element is the root, it is removed and placed in the sorted list. Then, the remaining tree is readjusted to be a heap. This process continues until all items have been processed.

HEAP OPERATION

- Assuming the data stored in an array, the parent of node i is stored at $i/2$; the left child of node i is stored at $2i$; and the right child is stored at index $2i + 1$. The array is initially all heap area.
- After each pass, the heap area shrinks by one and the sorted area grows by one.

HEAP SORT PERFORMANCE

For large n , Heap sort is very efficient. The larger n becomes, the better Heapsort performs since it takes $O(n \log n)$ in both the worst and best case. Generally, Heapsort seems to "like" initial sequences in which the items are more less sorted in inverse order, and therefore it displays unnatural behavior.

QUICK SORT

- A refinement of the exchange sort.
- Based upon the fact that exchanges are best performed over large distances.
- Consider the case where the data is reverse ordered. In this case we can sort n items in $n/2$ exchanges by first exchanging the left and rightmost items and then working in from both sides. While this is only possible where the ordering of the data is known, it points out the possibilities.

PARTITIONING

- Pick an item at random (obviously using some scheme) and call it x .

- Scan the array from the left until an item $a_i > x$ is found then scan from the right until an item $a_j < x$ is found.
- Now exchange the two items and continue this process until the two scans meet somewhere in the middle of the array, resulting in an array which is partitioned into a left part with keys less than x and a right part with keys greater than x .

PARTITIONING

The partitioning process may be stated as follows:

BEGIN

$i := 1; j := n;$

< select an item x >

repeat

while $a[i] < x$ **do**

$i := i + 1;$

while $a[j] > x$ **do**

$j := j - 1;$

if $i \leq j$ **then**

begin

$temp := a[i]; \quad a[i] := a[j]; \quad a[j] := temp;$

$i := i + 1; \quad j := j - 1;$

end;

until $i > j;$

end;

COMPLETION OF THE SORT

Now we complete the task of sorting the array by applying the same process to both partitions until every partition consists of only one item. This suggests the use of recursion although a stack can be used.

QUICK SORT ALGORITHM

begin

$i := l; j := r; temp1 := a[(l+r)/2];$

repeat

while $a[i] < temp1$ **do**

$i := i + 1;$

while $temp1 < a[j]$ **do**

$j := j - 1;$

if $i \leq j$ **then**

BEGIN

$temp2 := a[i]; \quad a[i] := a[j]; \quad a[j] := temp2;$

$i := i + 1; j := j - 1$

end

until $i > j;$

if $l < j$ **then**

Quick Sort (l, j);

if $i < r$ **then**

Quick Sort (i, r)

end;

COLLECTING SORTS METRICS DATA

Since most of the sort metrics are dependent on the data being sorted, it is useful to be able to collect this data real-time. An example (a straight insertion) of a sort program with measurement probes inserted is shown below. Since the statements used to collect the data consume processing time, this version of the sort does not return an accurate value of CPU time.

DATA COLLECTION EXAMPLE

for $i := 2$ to $array_size$ **do**

begin

$temp := dictionary[i];$

$moves := moves + 1;$ $dictionary[0] := temp;$

$moves := moves + 1;$ $j := i - 1;$

while $temp < dictionary[j]$ **do**

begin

$compares := compares + 1;$

$dictionary[j+1] := dictionary[j];$

$moves := moves + 1;$

$j := j - 1;$

end;

$dictionary[j + 1] := temp;$

$moves := moves + 1;$

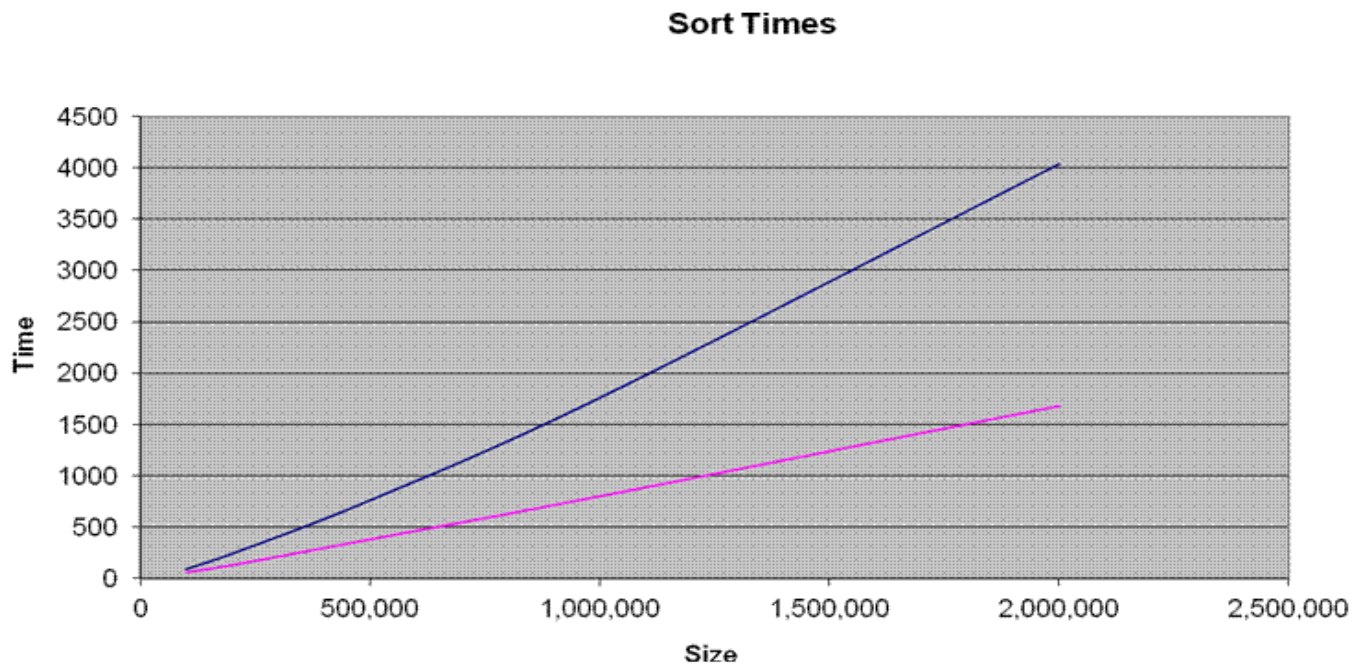
end;

PERFORMANCE – GENERIC SORTS

Size	Insertion	Selection	Exchange	Bubble
5,000	60	170	410	410
5,000	160	200	360	381
5,000	0	191	190	0
10,000	370	791	1,750	1,780
10,000	0	801	781	0
10,000	871	891	1,520	1,640
20,000	1,750	3,260	7,040	6,990
20,000	0	3,270	3,150	0
20,000	4,030	3,730	6,340	68
40,000	7,550	13,400	28,810	28,150
40,000	0	1,402	12,930	0
40,000	16,620	15,020	25,300	26,600
100,000	47,280	82,820	181,790	175,630
100,000	0	82,610	87,770	0
100,000	94,120	90,230	162,930	187,570

s ⇒ sorted data, r ⇒ reverse sorted data

PERFORMANCE – HEAP AND QUICK SORTS



DATA COMMUNICATION AND NETWORKING

INTRODUCTION

- 📖 Communications has extended our uses for the microcomputer.
- 📖 Communication systems are the electronic systems that transmit data over communications lines from one location to another.
- 📖 You can set up a network in your home or apartment using existing telephone lines.
- 📖 Competent end users need to understand the concept of connectivity, the impact of the wireless revolution, and the elements of a communications system.
- 📖 They must also understand the basics of communications channels, connection devices, data transmission, networks, network architectures, and network types.

COMMUNICATIONS

- Computer communications is the process of sharing data, programs, and information between two or more computers.

CONNECTIVITY

- Connectivity is a concept related to using computer networks to link people and resources
- You can use telephone lines to link to nearly any computer in the world.

THE WIRELESS REVOLUTION

- The single most dramatic change in connectivity and communications in the past five years has been the widespread use of mobile or wireless telephones.
- In 2002, it was estimated that there are over 600 million mobile telephones in use worldwide, and by 2004, almost 1.5 billion (source: Newsweek, June 7, 2004 pg 51 Next Frontiers: Your Next Computer).
- This wireless technology allows individuals to stay connected with one another from almost anywhere at any time.
- Originally developed for voice, the wireless revolution can transmit nearly any kind of information

COMMUNICATIONS SYSTEMS

- Communications systems have four basic elements

SENDING AND RECEIVING DEVICES

- Often a computer or specialized communications device.

COMMUNICATION CHANNEL (AKA TRANSMISSION MEDIUM)

- The actual connection that carries the message
- Can be a physical wire, or a wireless connection

CONNECTION DEVICE (AKA COMMUNICATIONS DEVICE)

- Act as an interface between the sending and receiving devices. They convert outgoing messages into a digital format, and back again at the receiving (incoming) end.

DATA TRANSMISSION SPECIFICATIONS

- 📖 The rules and procedures that coordinate the sending and receiving devices by precisely defining how the message will be sent across the communication channel.

PHYSICAL CONNECTIONS

TELEPHONE LINES

- 📖 Typically use twisted pair cables, copper wires covered with an insulating jacket
- 📖 Relatively inexpensive way to connect devices
- 📖 Now being phased out by more technically advanced and reliable media

COAXIAL

CABLE

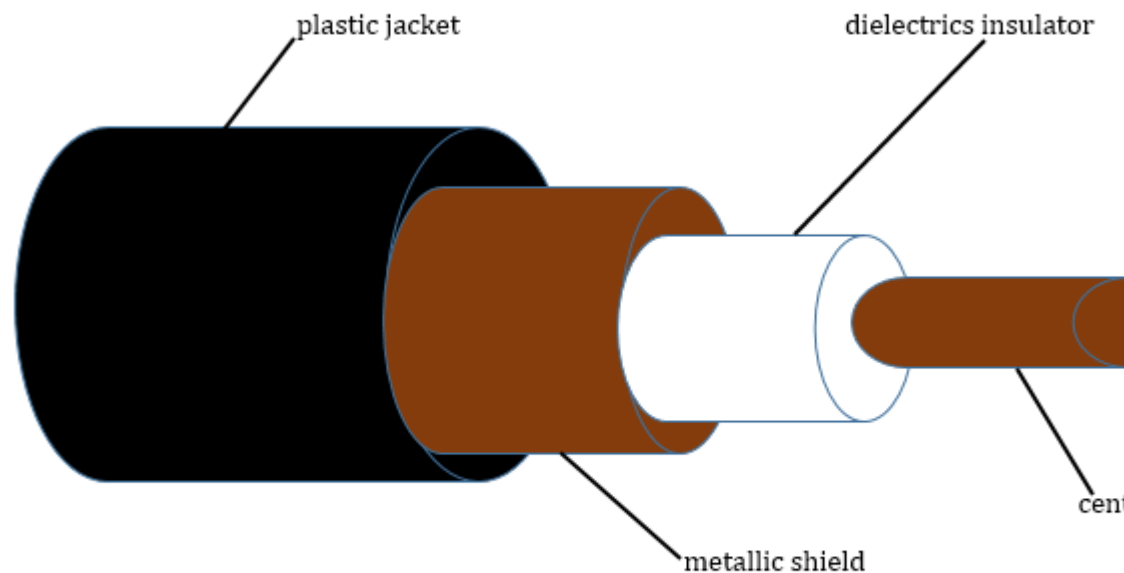


Fig: Show Coaxial Cable

- 📖 A high frequency transmission cable, can be used to replace multiple lines of twisted pair cable with one single, solid copper core.
- 📖 Can carry 80 times the capacity of one twisted pair cable

UNSHIELDED TWISTED-PAIR

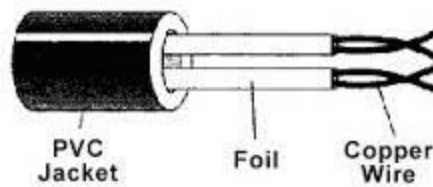


Fig:Un shield Twisted- Pair (UTP)



Fig:Shielded Twisted -Pair (STP)

FIBER-OPTIC

CABLE

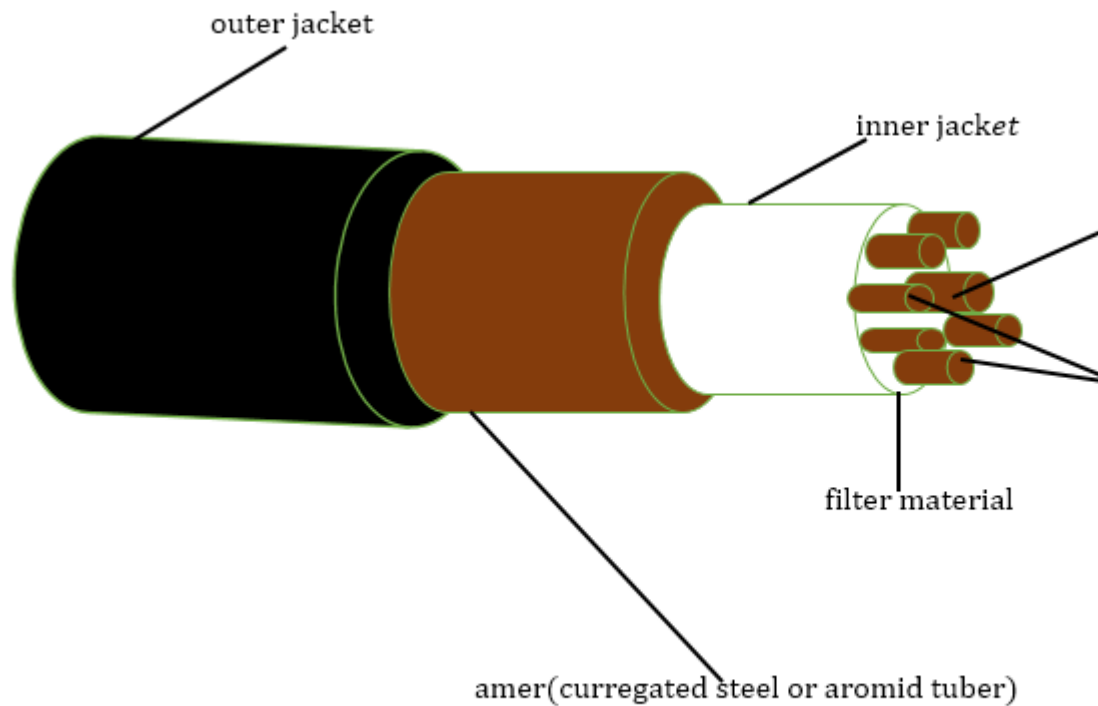


Fig: Show Fiber Cable

- 📖 Transmit data as a pulse of light through tiny tubes of glass
- 📖 Has over 26,000 times the capacity as one twisted pair cable

- 📁 Fiber optic cables are rapidly replacing twisted pair telephone wires

WIRELESS CONNECTIONS

INFRARED

- 📁 Use infrared light waves to communicate
- 📁 Known as a “line of sight” communication medium
- 📁 Commonly used to transmit data from a PDA to a desktop PC

BROADCAST

RADIO

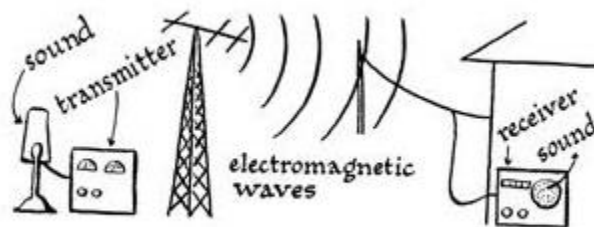


Fig: Show Radio Broadcasting

- 📁 Uses special sending and receiving towers called “transceivers”
- 📁 The transceiver sends and receives many signals from different wireless devices
- 📁 Cellular telephones communicate using this technology
- 📁 WiFi (Wireless Fidelity aka 802.11 technologies) are used to build wireless local area networks

MICROWAVE

- 📁 Uses high frequency radio waves
- 📁 Line of sight medium
- 📁 Transmit data over relatively short distances (within 10-20 miles) due to curvature of the earth
- 📁 Microwave signals are sometimes repeated at microwave stations with microwave dishes

- Bluetooth is a short-range wireless communication standard that uses microwaves to transmit data over very short distances (less than 33 feet). This may become popular for connecting peripheral devices to computers

SATELLITE



Fig: Show satellite

- Uses satellites orbiting up to 22,000 miles above the earth to send large volumes of data
- Up link is sending data to a satellite
- Down link is receiving data from a satellite
- GPS (Global Positioning Systems) use satellite data to pinpoint locations nearly anywhere on the earth. They are used for both military and commercial navigation systems

CONNECTION DEVICES

- A great deal of communication takes place over telephone lines
- Since telephone was used for voice, the technology typically used analog signals to transmit calls
- Computer use digital signals

- 📎 To connect computers via telephone lines, a system was needed to transmit data from digital to analog to digital again. Modems were created to do this.

MODEMS

- 📎 Modem is an acronym meaning “modulator – demodulator”
- 📎 Modulation converts a digital signal to an analog signal
- 📎 Demodulation converts an analog signal back to digital
- 📎 Speed at which modems communicate is measured in bits per second (bps).
- 📎 Typically modem speeds are 33.6 and 56 kbps (kilo bits per second)

EXTERNAL MODEM

- 📎 Modem circuitry housed in a separate case
- 📎 Connected to computer using a serial port, and to telephone using a phone wire and an RJ-11 jack

INTERNAL MODEM

- 📎 Modem circuitry is housed inside the computer
- 📎 Connects to telephone wall jack using a phone wire

PC CARD MODEM

- 📎 Serves as an “external modem” for a laptop
- 📎 Credit card sized expansion board to connect a laptop computer to a telephone line

WIRELESS MODEM

- 📎 Can be an external, internal, or PC Card modem, but rather than connecting to the telephone system using a wire, it connects via wireless technology (e.g. a cellular phone connection)

TYPES OF CONNECTIONS

- 📎 Standard telephone lines and modems are called dial up services
- 📎 Large organizations use higher speed connections such as T1, T2, T3, and T4 lines.
- 📎 These support all digital communications, so they don’t use modems but do require special equipment.

- 📎 They tend to be expensive, but can transmit data at high speeds, e.g. 1.5 Mbps (1,500 kbps) almost 26 times faster than standard dial up service

DIGITAL SUBSCRIBER LINE (DSL)

- 📎 A high speed Internet service offered by phone companies

CABLE MODEMS

- 📎 A high speed Internet service offered by Cable TV companies

SATELLITE/AIR CONNECTION SERVICES

- 📎 Another competitor for high speed Internet services, often offered in areas where Cable or DSL is not available

DATA TRANSMISSION PG

- 📎 Several factors affect how data is transmitted across a communication medium, including:

BANDWIDTH

VOICE BAND (AKA VOICE GRADE OR LOW BANDWIDTH)

- 📎 Standard telephone connections
- 📎 Typical speed is 56 kbps
- 📎 Low cost, but lower speed

MEDIUM BAND

- 📎 Bandwidth used in special leased lines to connect minicomputers and main frames as well as transmitting data over long distances
- 📎 Typically used by businesses and not individuals

BROADBAND

- 📎 Used for high-capacity transmissions
- 📎 Microcomputers with DSL, cable, or satellite connections use this
- 📎 Speeds are typically 1.5 Mbps, but can go higher

PROTOCOLS

- 📎 Protocols are the rules for exchanging data across a network

- 📖 A standard for the Internet is the TCP/IP protocol – Transmission Control Protocol / Internet Protocol
- 📖 Essential features of TCP/IP is for identifying the sending and receiving devices, and reformatting the data so it can be sent via the Internet

IDENTIFICATION

- 📖 Every computer on the Internet has an IP address (Internet Protocol address). This is a numeric address such as 198.45.19.151
- 📖 A Domain Name Server (DNS) converts a text based address, e.g. <http://www.McGraw-Hill.com> into the IP address 198.45.19.151

REFORMATTING

- 📖 Information sent or transmitted across the Internet usually travels through numerous interconnected networks.
- 📖 The data is broken into a series of “packets” and sent separately over the Internet.
- 📖 At the receiving end, these packets are reassembled into the correct order, and the transmission is complete

NETWORKS

- 📖 A computer network is a communication system that connects two or more computers so that they can exchange information and share resources.

TERMS USED

NODE

- 📖 Any device connected to a computer; a printer, a PC, storage device, etc.

CLIENT

- 📖 A node that requests and uses resources available from other nodes

SERVER

- 📖 A node that shares resources with other nodes

- ☞ Dedicated servers include application servers, communication servers, database servers, file servers, printer servers or web servers

HUB / SWITCH

- ☞ The center or central node for other nodes
- ☞ It may be a server or a central connection point

NETWORK INTERFACE CARD (NIC)

- ☞ An adapter card for connecting a node to a network

NETWORK OPERATING SYSTEM (NOS)

- ☞ Controls and coordinates the activities of all computers and other devices on a network

DISTRIBUTED PROCESSING

- ☞ A system in which computing power is located and shared at different locations

HOST COMPUTER

- ☞ A large, centralized computer, usually a minicomputer or a main frame

NETWORK MANAGER

- ☞ A computer specialist, also known as a network administrator, responsible for maintaining the network operations

Networking Basics

Basic Network Structure

A network consists of 2 or more computers connected together, and they can communicate and share resources (e.g. information)

Why Networking?

Sharing information — data communication

Sharing hardware or software - print document

Centralize administration and support - Internet-based, so everyone can access the same administrative or support application from their PCs

How many kinds of Networks?

Depending on one's perspective, we can classify networks in different ways

Based on transmission media: Wired (UTP, coaxial cables, fiber-optic cables) and Wireless

Based on network size: LAN and WAN (and MAN)

Based on management method: Peer-to-peer and Client/Server

Based on topology (connectivity): Bus, Star, Ring ...

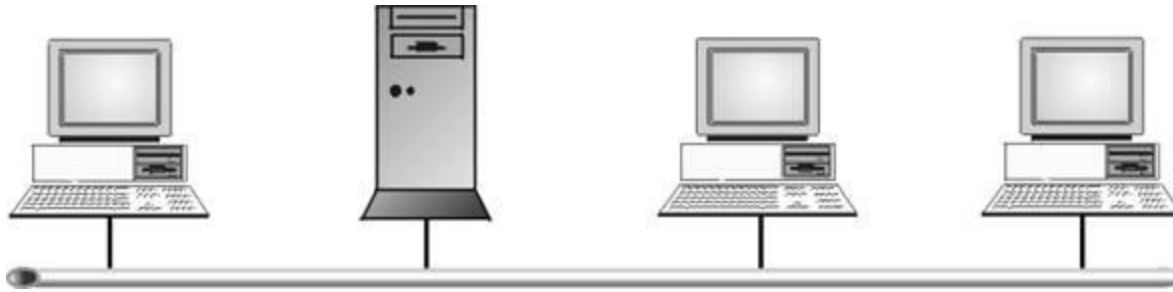
Topologies

There are different topologies that make up computer networks. Topology is the physical layout of computers, cables, and other components on a network. Many networks are a combination of the various topologies that we will look at:

- Bus
- Star
- Mesh
- Ring

Bus Topologies

A bus topology uses one cable to connect multiple computers. The cable is also called a trunk, a backbone, and a segment. Most of the times, as seen in Figure below, T-connectors are used to connect to the cabled segment. They are called T-connectors because they are shaped like the letter T. You will commonly see coaxial cable used in bus topologies.



In a bus topology, all computers are connected on one linear cable.

Another key component of a bus topology is the need for termination. To prevent packets from bouncing up and down the cable, devices called *terminators* must be attached to both ends of the cable. A terminator absorbs an electronic signal and clears the cable so that other computers can send packets on the network. If there is no termination, the entire network fails.

Only one computer at a time can transmit a packet on a bus topology. Computers in a bus topology listen to all traffic on the network but accept only the packets that are addressed to them. Broadcast packets are an exception because all computers on the network accept them. When a computer sends out a packet, it travels in both directions from the computer. This means that the network is occupied until the destination computer accepts the packet. The number of computers on a bus topology network has a major influence on the performance of the network. A bus is a passive topology. The computers on a bus topology only listen or send data. They do not take data and send it on or regenerate it. So if one computer on the network fails, the network is still up.

ADVANTAGES

One advantage of a bus topology is cost. The bus topology uses less cable than the star topology or the mesh topology. Another advantage is the ease of installation. With the bus topology, you simply connect the workstation to the cable segment, or backbone. You need only the amount of cable to connect the workstations you have. The ease of working with a bus topology and the minimum amount of cable make this the most economical choice for a network topology. If a computer fails, the network stays up.

DISADVANTAGES

The main disadvantage of the bus topology is the difficulty of troubleshooting. When the network goes down, usually it is from a break in the cable segment. With a large network this can be tough to isolate. Figure 1-2 shows a cable break between computers on a bus topology, which would take the entire network down. Another disadvantage of a bus topology is that the heavier the traffic, the slower the network.

Scalability is an important consideration with the dynamic world of networking. Being able to make changes easily within the size and layout of your network can be important in future productivity or downtime. The bus topology is not very scalable.

Star Topologies

In a star topology, all computers are connected through one central hub or switch, as illustrated in Figure below. This is a very common network scenario.

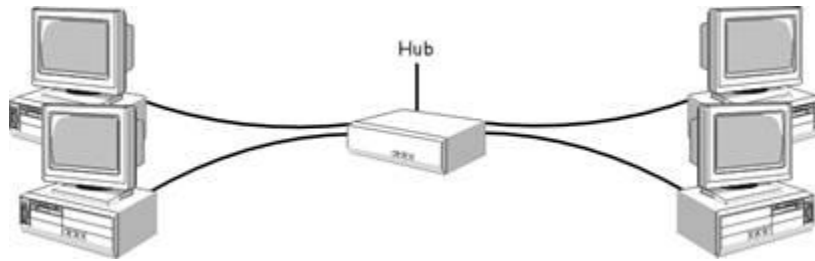


Fig: Computer in a star topology are all connected to a central hub

A star topology actually comes from the days of the mainframe system. The mainframe system had a centralized point where the terminals connected.

Advantages

One advantage of a star topology is the centralization of cabling. With a hub, if one link fails, the remaining workstations are not affected like they are with other topologies.

Centralizing network components can make an administrator's life much easier in the long run. Centralized management and monitoring of network traffic can be vital to network success. With this type of configuration, it is also easy to add or change configurations with all the connections coming to a central point.

Disadvantages

On the flip side to this is the fact that if the hub fails, the entire network, or a good portion of the network, comes down. This is, of course, an easier fix than trying to find a break in a cable in a bus topology.

Another disadvantage of a star topology is cost: to connect each workstation to a centralized hub, you have to use much more cable than you do in a bus topology.

MESH TOPOLOGIES

A mesh topology is not very common in computer networking, but you will have to know it for the exam. The mesh topology is more commonly seen with something like the national phone network. With the mesh topology, every workstation has a connection to every other component of the network.

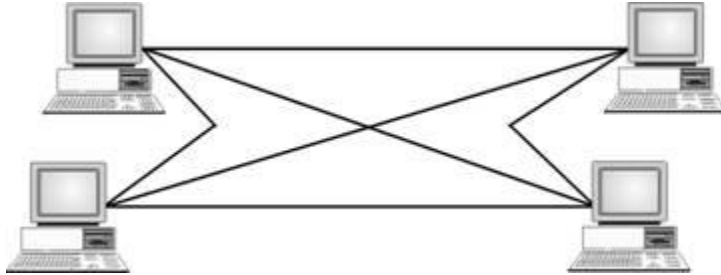


Fig: Computers in a mesh topology are all connected to every other component of the network

ADVANTAGES

The biggest advantage of a mesh topology is fault tolerance. If there is a break in a cable segment, traffic can be rerouted. This fault tolerance means that the network going down due to a cable fault is almost impossible. (I stress *almost* because with a network, no matter how many connections you have, it can crash.)

DISADVANTAGES

A mesh topology is very hard to administer and manage because of the numerous connections. Another disadvantage is cost. With a large network, the amount of cable needed to connect and the interfaces on the workstations would be very expensive.

RING TOPOLOGIES

In a ring topology, all computers are connected with a cable that loops around. As shown in Figure, the ring topology is a circle that has no start and no end. Terminators are not necessary in a ring topology. Signals travel in one direction on a ring while they are passed from one computer to the next. Each computer checks the packet for its destination and passes it on as a repeater would. If one of the computers fails, the entire ring network goes down.

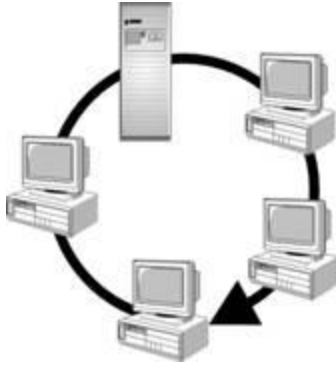


Fig: Signals travel in one direction on a ring topology

ADVANTAGES

The nice thing about a ring topology is that each computer has equal access to communicate on the network. (With bus and star topologies, only one workstation can communicate on the network at a time.) The ring topology provides good performance for each workstation. This means that busier computers who send out a lot of information do not inhibit other computers from communicating. Another advantage of the ring topology is that signal degeneration is low.

DISADVANTAGES

The biggest problem with a ring topology is that if one computer fails or the cable link is broken the entire network could go down. With newer technology this isn't always the case. The concept of a ring topology is that the ring isn't broken and the signal hops from workstation to workstation, connection to connection.

Isolating a problem can be difficult in some configurations also. (With newer technologies a workstation or server will beacon if it notices a break in the ring.) Another disadvantage is that if you make a cabling change to the network or a workstation change, such as a move, the brief disconnection can interrupt or bring down the entire network.

Network Operating Systems

We will focus on the four most widely used network operating systems available:

- Microsoft Windows Server
- Novels Net-Ware
- UNIX

- Linux

Network operating systems can operate in two fashions. In a *peer-to-peer* environment, each workstation on the network is equally responsible for managing resources. Each individual workstation can share its resources with other systems on the network. Configuring this type of network can be challenging, because nothing is centralized.

In a *client/server* environment, on the other hand, there is a centralized approach to the network operating system. If, as an administrator, you identify one machine as the network server, you can centralize network resource sharing. Clients then access the server.

Review

What is a network?

What are the advantages of using a network?

What are the different types of networks?

What is a network topology?

What are the four types of network topologies?

Write about the advantages and disadvantages of each topology?

What are the most widely used network operating systems?

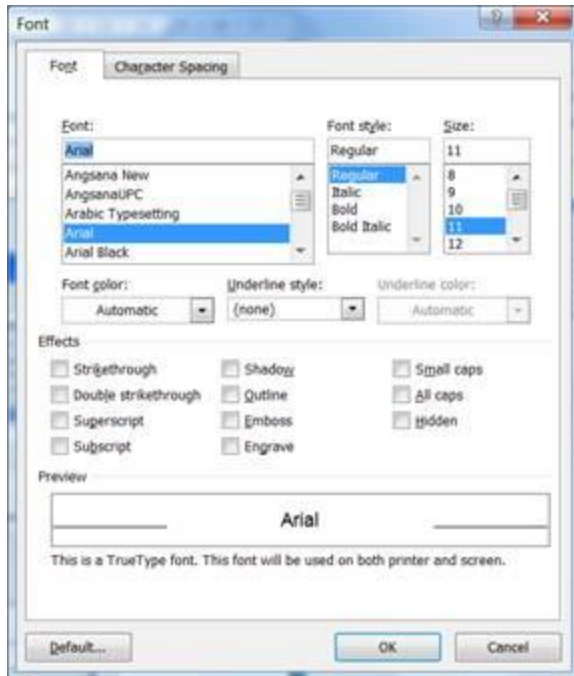
AN INTRODUCTION TO VISUAL PROGRAMMING USING VB.Net

Contents

1. Windows Environments

2. Programming Languages

3. VB.NET



This is an example of a form and all such dialogue boxes are examples of forms. Most Microsoft products use Multi-Document Interface (MDI) and Visual Programming Languages have been created to allow programmers to rapidly design and implement such programs. The form templates are provided upon which controls such as textboxes, drop-down lists etc can be placed. Such facilities allow for the rapid implementation of windows programs leaving the programmer to concentrate on the logic and functionality of the program

2. PROGRAMMING LANGUAGES

Since the early beginnings of the modern digital computer, many and various programming languages have been created. Each one has been created for a specific task or to improve the effectiveness and efficiency of the code. For example, one of the earlier languages, FORTRAN (Formulate Translation) was written for mathematicians and scientists to use. Because of its simplicity of use PASCAL has been adopted by many schools and colleges to teach the basics of programming. COBOL (Common Business Oriented Language) was, for many years the choice of language in business applications. Many of the earlier languages could be described as being **Procedural** languages which are based on the concept of the

Procedural Call. When the program is executed (run) the program will run through a sequence of procedures, sub procedures and functions until it has completed its task.

Later programming paradigms (models) include:

- **Event Driven** – a language based on events that happen during the execution of a program i.e. a mouse click, entering data in a textbox or temporal events i.e. events that occur at various times etc. Events are controlled by:
 - o **Event Handlers** - code that is called when an event is invoked
 - o **Event loops** -that are provided by the language to check if an event has occurred
 - o **Trigger functions**– select the event handler according to which event has occurred
- **Object Oriented** – This type of language is fast becoming the industry standard in software engineering. It is based on the concept of **Classes** and **Objects** that can be created from the classes. Programs created using OO techniques are usually cheaper, easier to maintain and can usually take less time to develop
- **Visual** – languages that use an **IDE** (Integrated Development Environment) whereby developers have access to pre-defined objects which can be dragged from a toolbox and placed on **Forms**.

3. VB.NET

Visual Basic.NET is a combination of all the above languages. It has a visual interface (IDE), is driven by events and is fully object oriented.

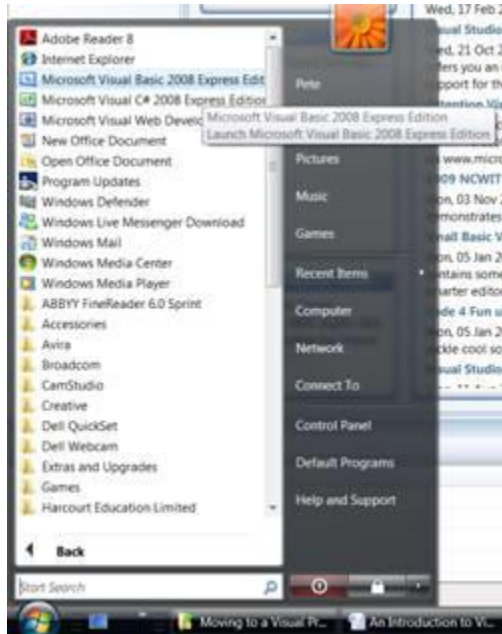
CREATING A MDI APPLICATION

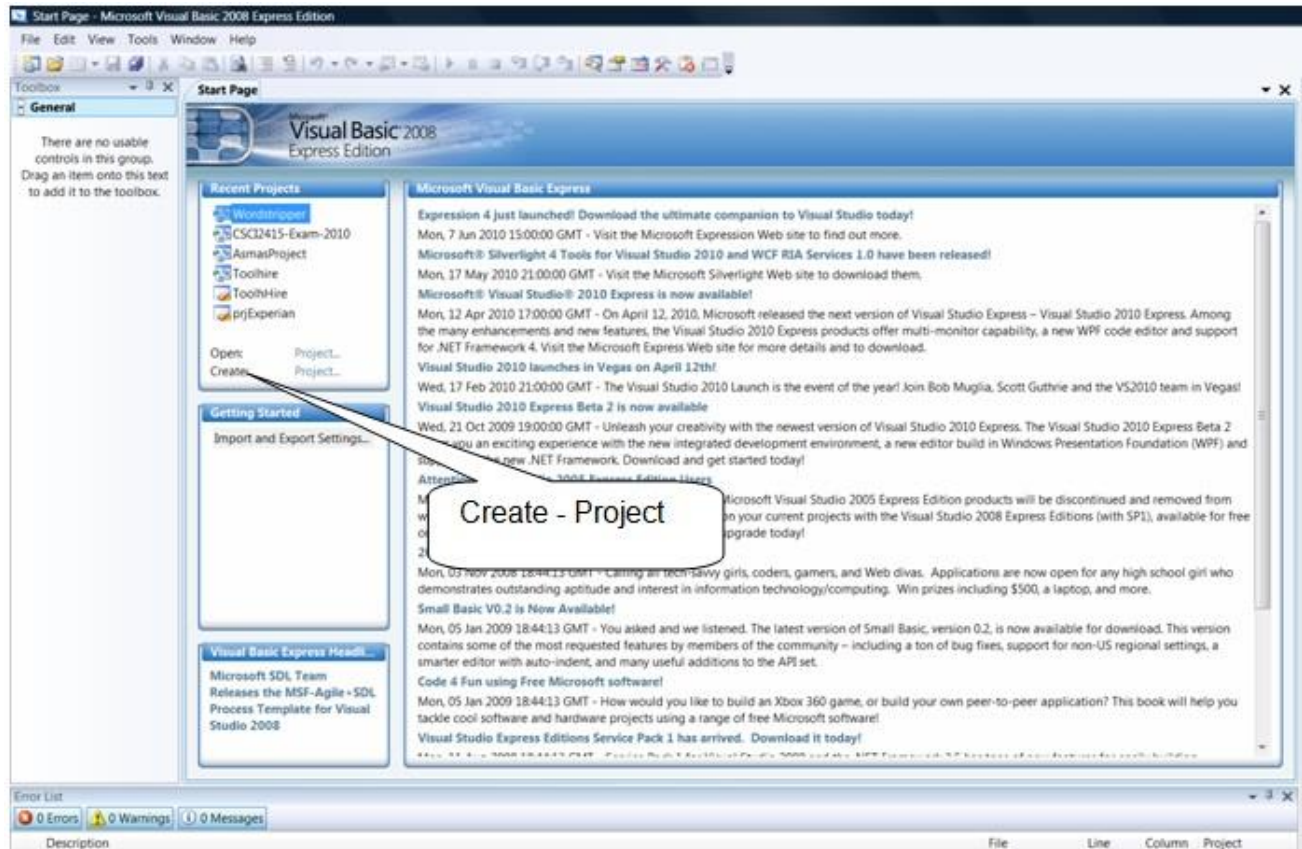
As a way of introducing the VB.Net IDE we will create a MDI that will be used to save all of the exercise programs that you implement during the topic.

- Software Design and Development
- Event Driven Programming
- Object Oriented Programming

STARTING A NEW PROJECT

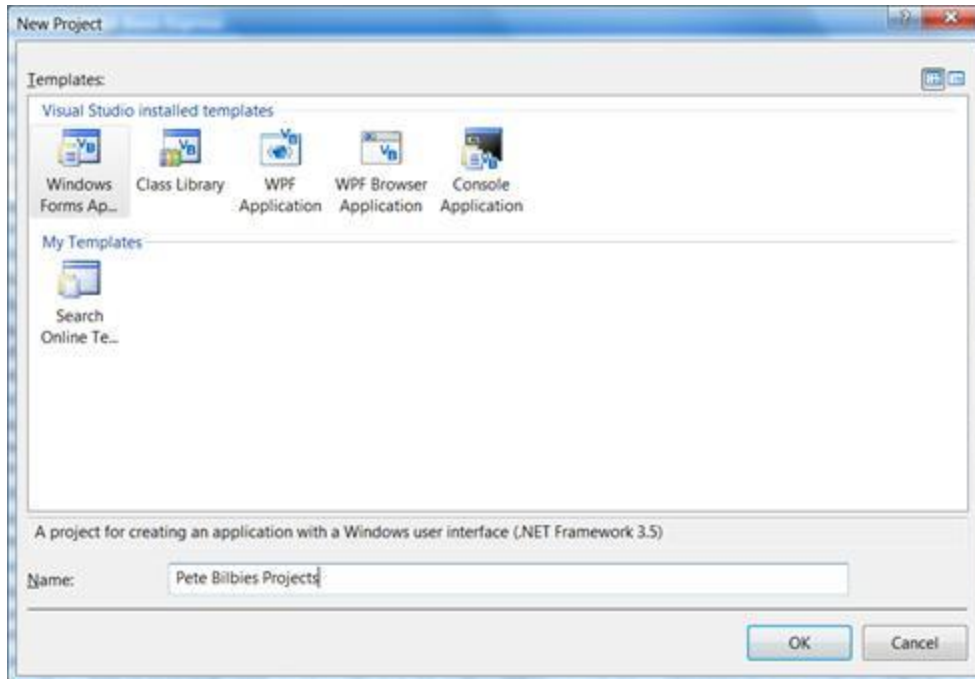
Click on **Programs->Microsoft Visual Basic 2008 Express Edition** to open the start page of VB.Net



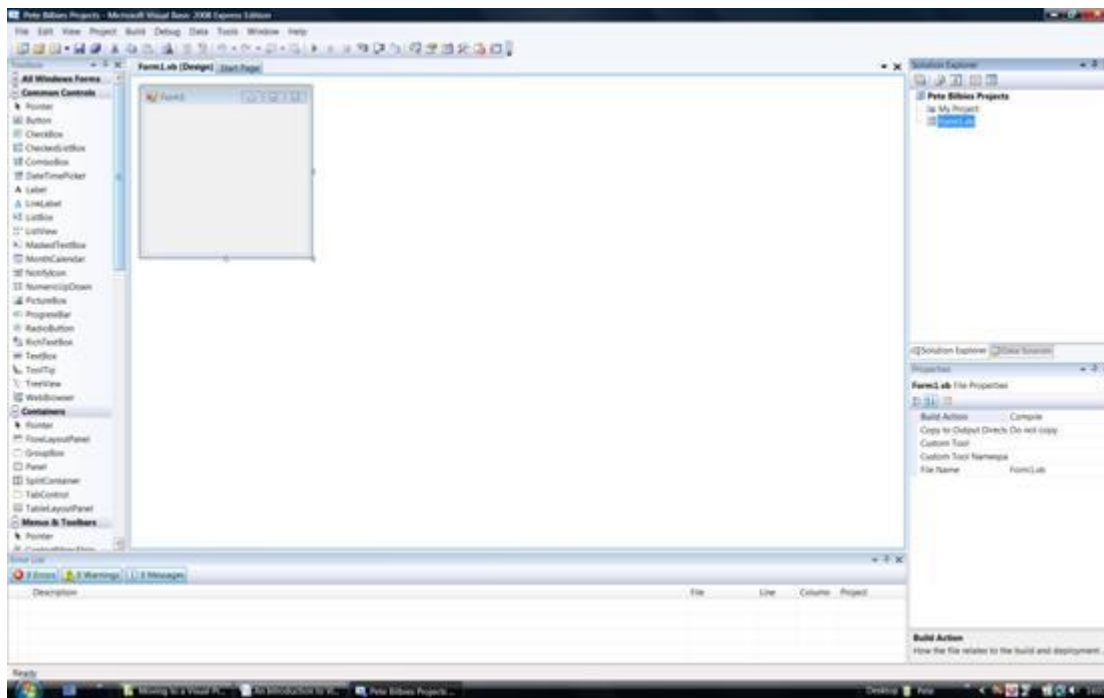


Click on **Create – Project** – see below:

In the following window make sure that the **Windows Form Application** is highlighted and add a name in the **Name** textbox. I suggest using your own name i.e. Fred Bloggs Project. Click OK

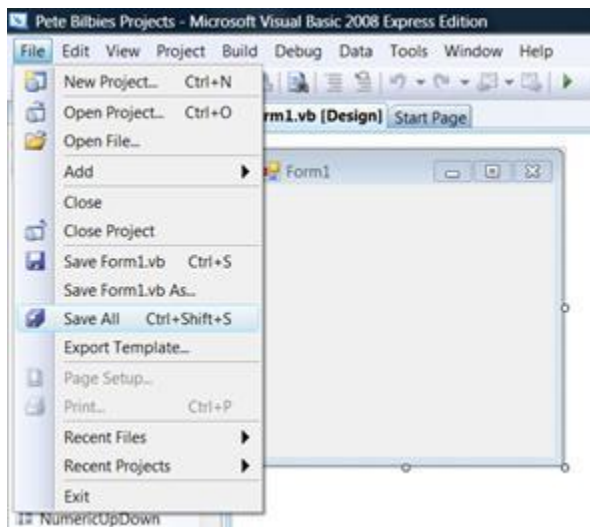


The IDE (Integrated Development Environment) select should now be displayed – see below:

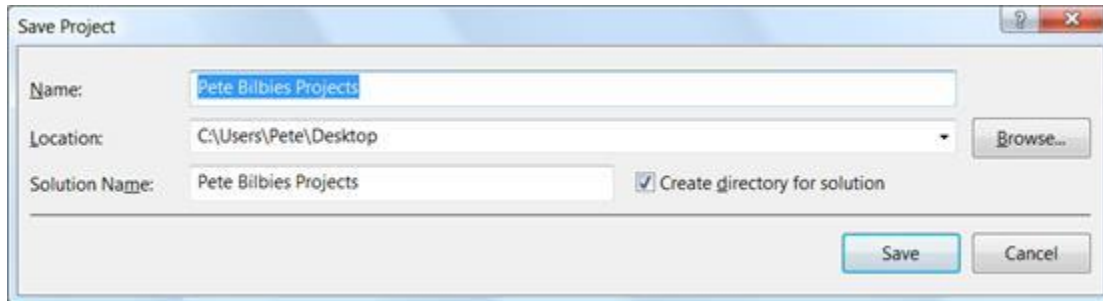


SAVING THE PROJECT

Before we continue it is best to save the project. Select **File->Save All** from the main menu – see below:

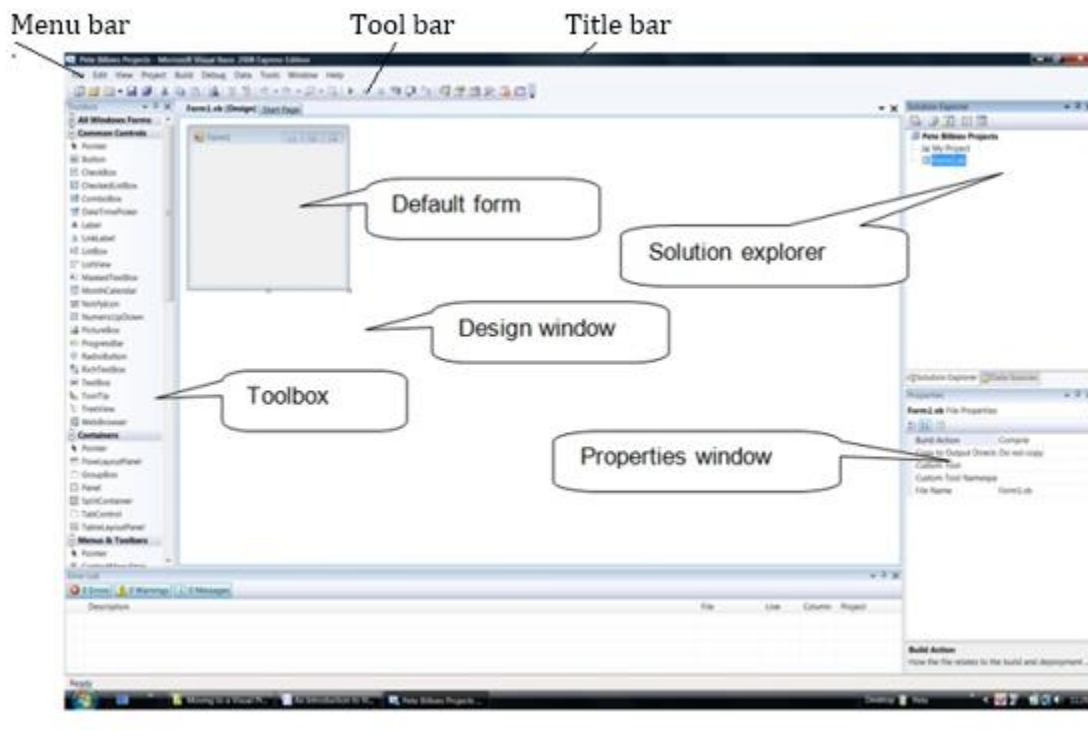


In the next window that is displayed select a suitable place to save the project (I'm saving mine to the desktop) and click **Save** – see below:



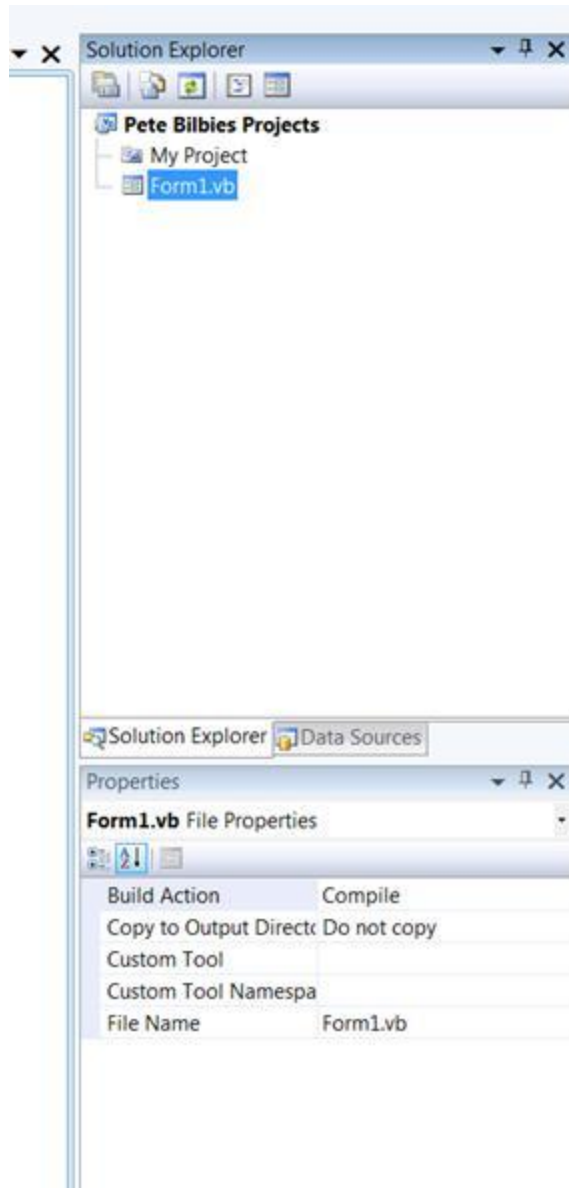
THE ELEMENTS THAT MAKE UP THE IDE

As you can see from the screen shot above there are a number of different windows in the default IDE which are explained below. There are many other windows that can be added but these will be explained in later sessions

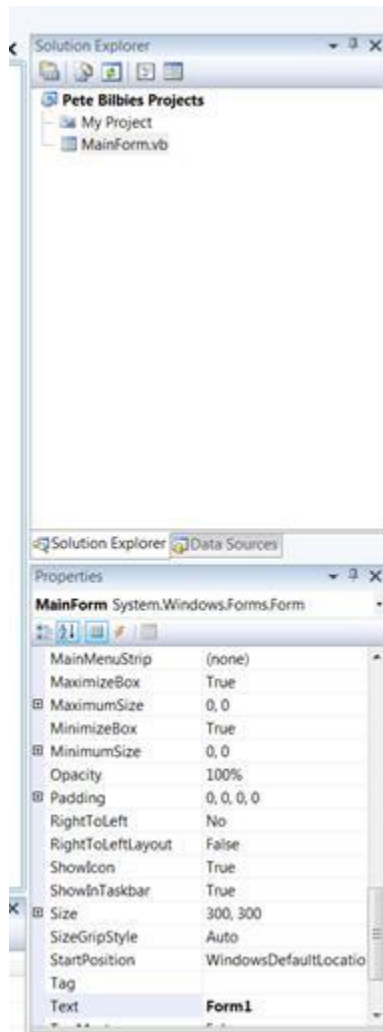


- **Design Window** – this is window that contains the forms for the program. All new projects have a default form. Many other forms can be added to the design.
- **Solution Explorer** – this contains the main components of the system i.e. Forms and the assemblies and references needed to create a basic program. As you add more functionality to the project the components will be added to the explorer
- **Properties Window** – All components of a program have properties that can be set either through this window or programmatically. If you click once on the Form icon in the Solution Explorer a set of properties for the form are displayed in the Properties Window – see above.
- **Toolbox** – This contains a list of all the objects that can be placed on a form –more about this later

The first step in creating the MDI program is to name the default form. To do this, **single click** on the form to select it. Do not double click on the form as this will take you to the **Code Window**. In the **Solution Explorer** single click on the Form1 icon and a set of properties for the form are displayed in the **Properties** widow below:



Change **The File Name** property of the default form (**Form1.vb**) to **Main Form.vb**. Do not have any spaces and make sure that you retain the **.vb** extension. If you now click on the form itself you will see that the name of the form in the **Solution Explorer** has been change and another set of properties is displayed in the **Properties Window** – see below.

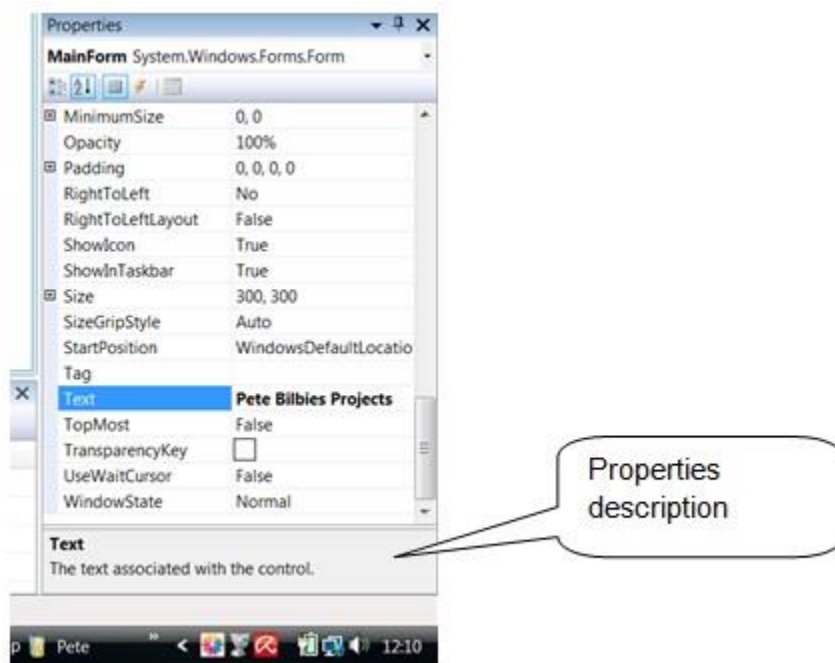


The properties now shown in the **Project** window are the main properties for the currently highlighted **control** (in this case Main Form). We use the name **Control** to describe any form or object from the **Toolbox** that is placed on the form. When a control is added to the form the controls properties are displayed or when it is highlighted. When a property is clicked an explanation of the property is displayed in the window directly below the properties window. Notice that the **(Name)** property has been changed to the name we

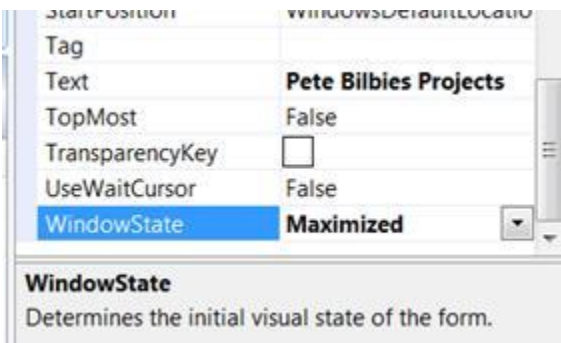
added earlier i.e. **Main Form**. The **(Name)** property is the identification for the control that is used to access the control when we are writing the code.

Other properties we need to change to make the form an MDI form are:

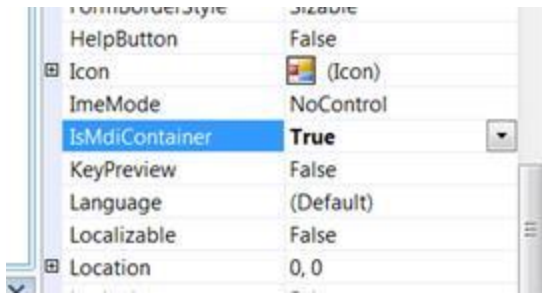
- **Text** – change to a suitable name – see below. This text will be displayed in the forms header. It can contain spaces



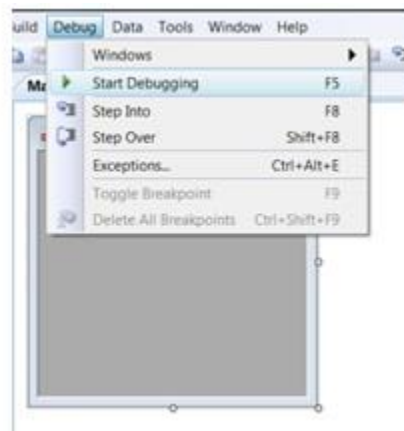
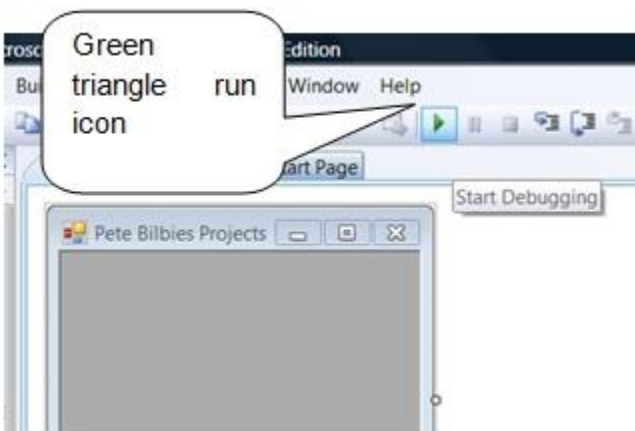
- **Window State** – set to Maximized by clicking on the drop-down list



IsMDIContainer – set this to **True**. Notice that when this property is set to true the background of the form changes to a darker shade



Each project can only contain one MDI form. It is going to be the form that is displayed when the program is executed. Setting the form to **Maximized** will make the form fill the screen. We can now execute (run) the program we have just created by either pressing **F5**, clicking on the green triangle in the main menu or by selecting **Debug->Start Debugging** from the top menu – see below:

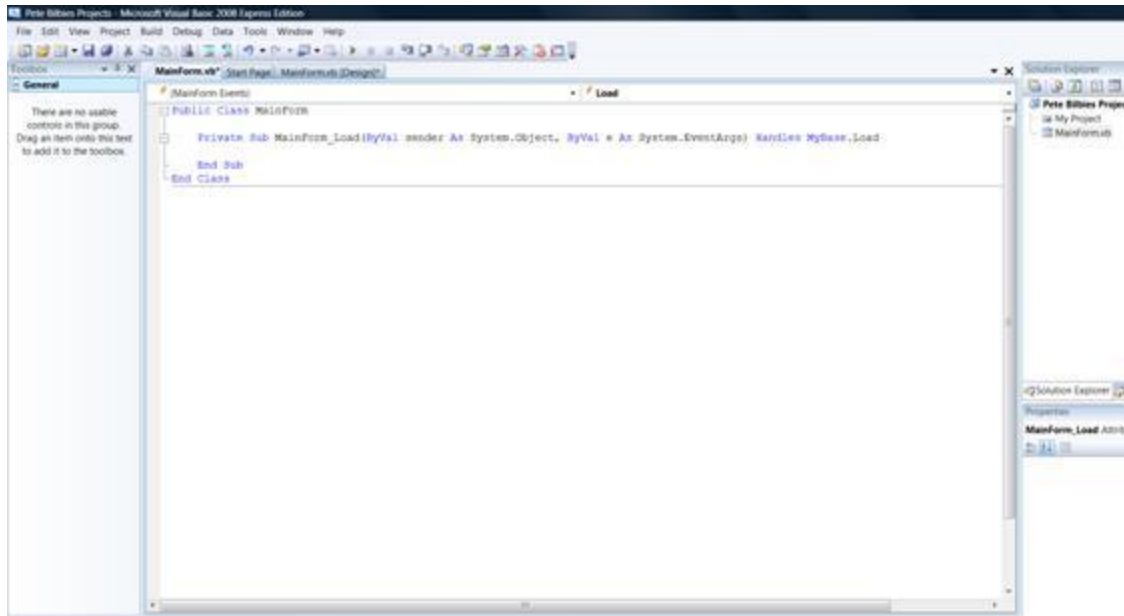


The MDI form should now be displayed maximized with the title shown in the top left-hand corner – see below:



You have just created your first VB.Net program. However, it does not do very much. Close the program by clicking on the X in the top right-hand corner of the form. This will return you to the IDE ready for the next stage of the process

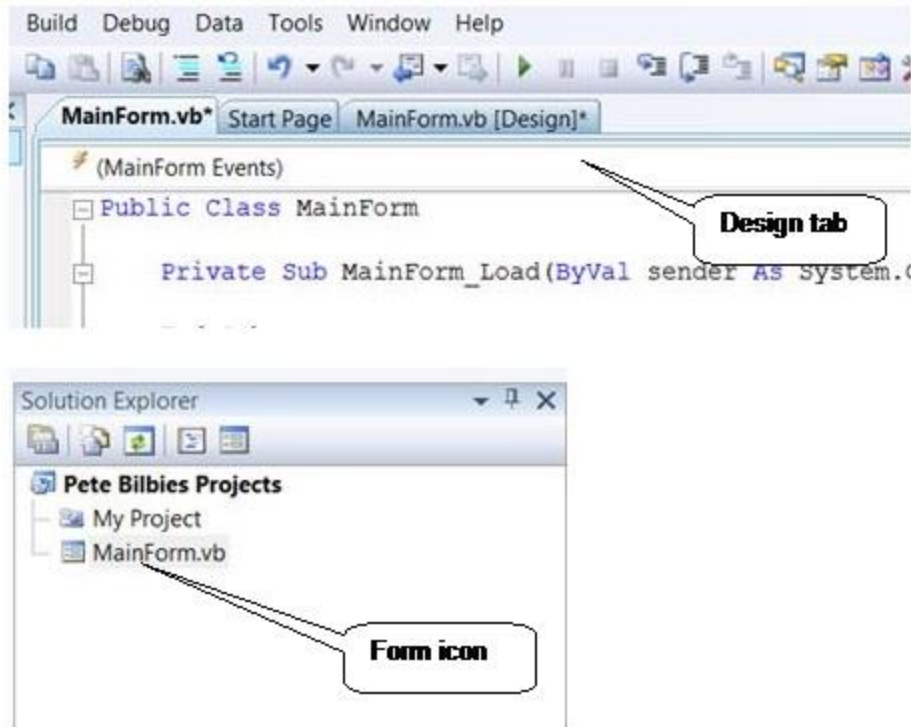
ACCESSING THE CODE WINDOW



Up to

now we have not written a single line of code as thus far the IDE has provided all the functionality we need. To access the code window of the form, double click on the form. The code window, similar to the one shown below will be displayed:

Notice that the **Toolbox** that was displayed when the IDE was in **Design Mode** i.e. when the form was displayed is now not accessible. To return to design mode either click on the **Design** tab at the top of the code window or click on the forms icon in the **Solution Explorer** – see below:



We will return to the code window later. What is needed before we add any code is to provide some form of navigation to different parts of the application program. Most of the applications created in a visual language consist of multiple forms that are accessed through a navigation system. VB.Net has controls that can be used for this purpose which can be accessed from the **Toolbox**.

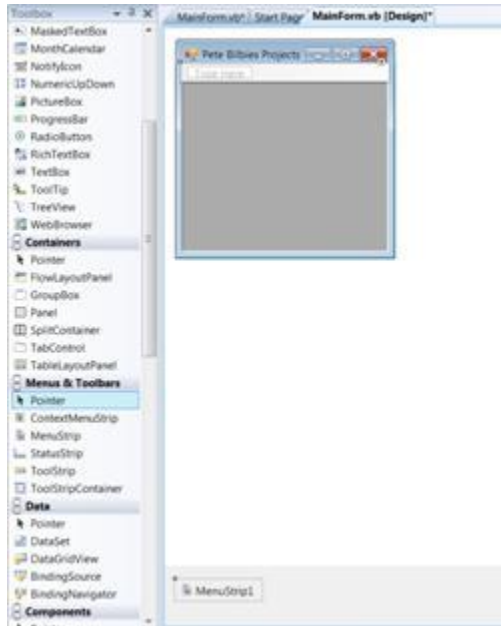
THE TOOLBOX

The **Toolbox** contains a list of all the controls i.e. textboxes, labels, buttons etc that can be added to a form. There are a number of different tabs to choose from which include:

- **Data** – controls used for accessing data sources i.e. databases
- **Components** – contains .NET components that monitor events within the operating system and within the Visual Studio Development environment
- **Printing** – controls used for printing
- **Menus and Toolbars** – controls for adding menus
- **Containers** – controls to group together other controls

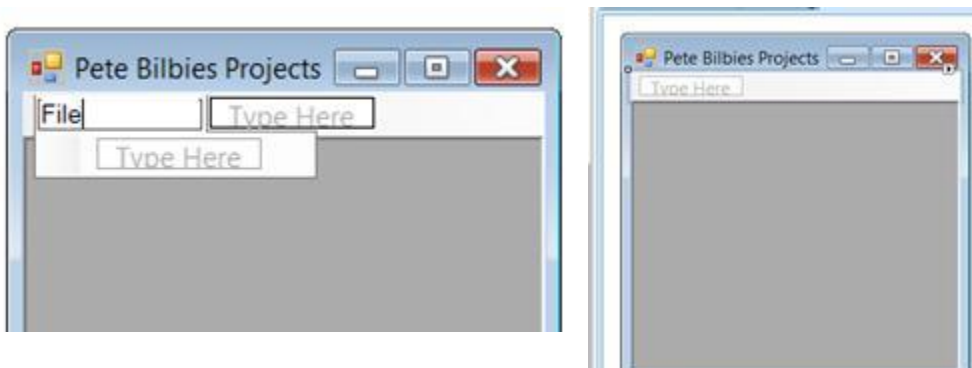
ADDING A MENU

In the Toolbox look for **Menus and Toolbars** and double click or click and drag the **Menu Strip** control onto the form. The form will then look as shown below:



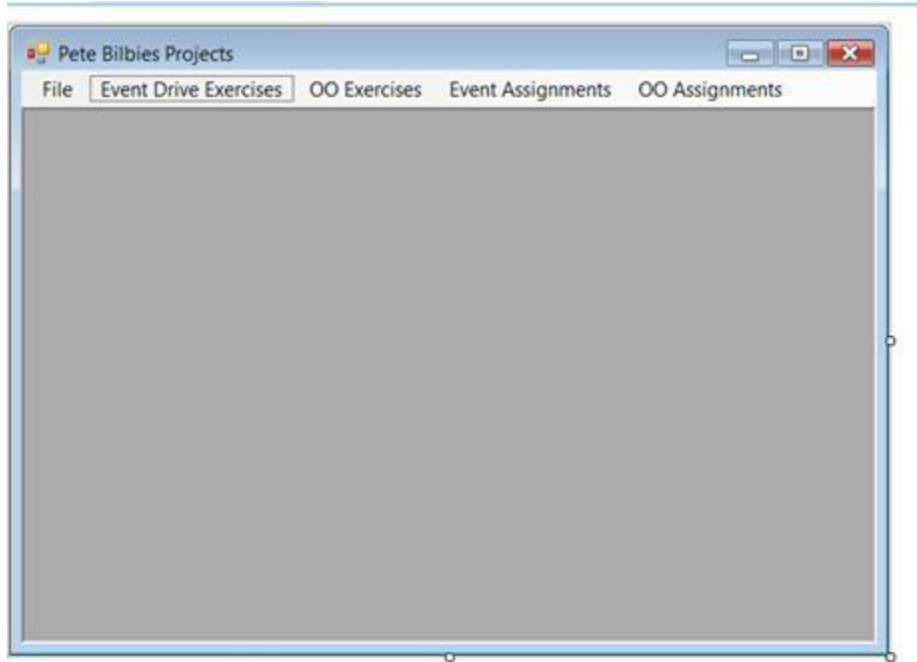
Notice that, not only has a Menu Strip been added to the form, but a menu strip icon has been added to the area directly below the Design window.

Click on the Menu Strip and where it says 'Type Here' add the word **File** – see below:



Notice that

when you add the text two more 'Type Here' labels appear - one to the right-hand side and one below. The one to the right-hand side is a top level menu item and the one below is a sub-menu of **File**. Add the text **Exit** to the one below and add four more top level menu items named **Event Driven Exercises**, **OO Exercises**, **Event Assignments** and **OO Assignments**. The form should now look as below:



If we execute the program and click on the menu items nothing happens. We need to add some code. To demonstrate this we will use the sub-menu **Exit**. Click on **File** and double-click on **Exit**. This will open the code window and the cursor will be flashing between the



Private
Sub and
the **End**
Sub

declaration of the **Event Handler** of the **Exit** menu item – see below:

Do not worry too much about any other code that is shown – that will be discussed later. For now we will concentrate the Exit Event Handler. Notice that the name has been changed from **Exit** to **Exit Tool Strip Menu_Click**. This defines the Event as being the **Click Event** of the menu item. More on Events later.

Add the following code - Application.Exit()- to the Click Event code:

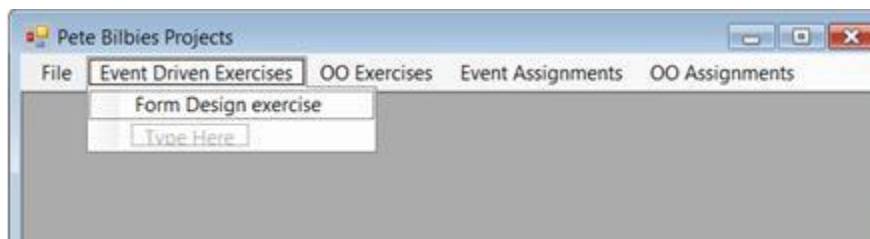
```
Private Sub Exit Tool Strip Menu Item_Click(...  
  
Application.Exit()  
  
End Sub
```

Run the program and click the **Exit** menu item and the program will close.

You have just written your first piece of VB code.

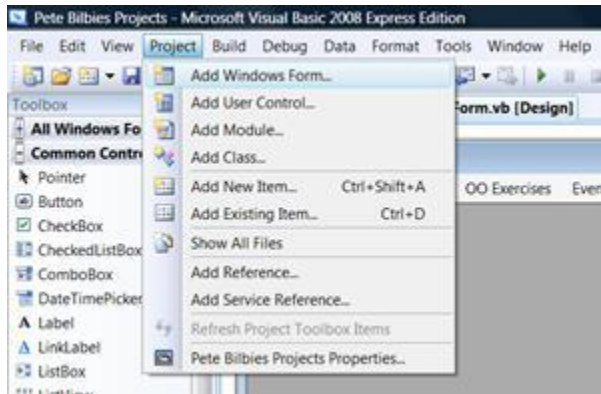
What's needed now is to add other forms to the program. Each form will contain solutions to exercises or solutions to assignment problems

The first class exercise we are going to complete is the designing of a form to which we will add various controls. It will be place in the **Event Driven Exercises** menu as a sub-menu item. We will name it **Form design exercise** – see below:

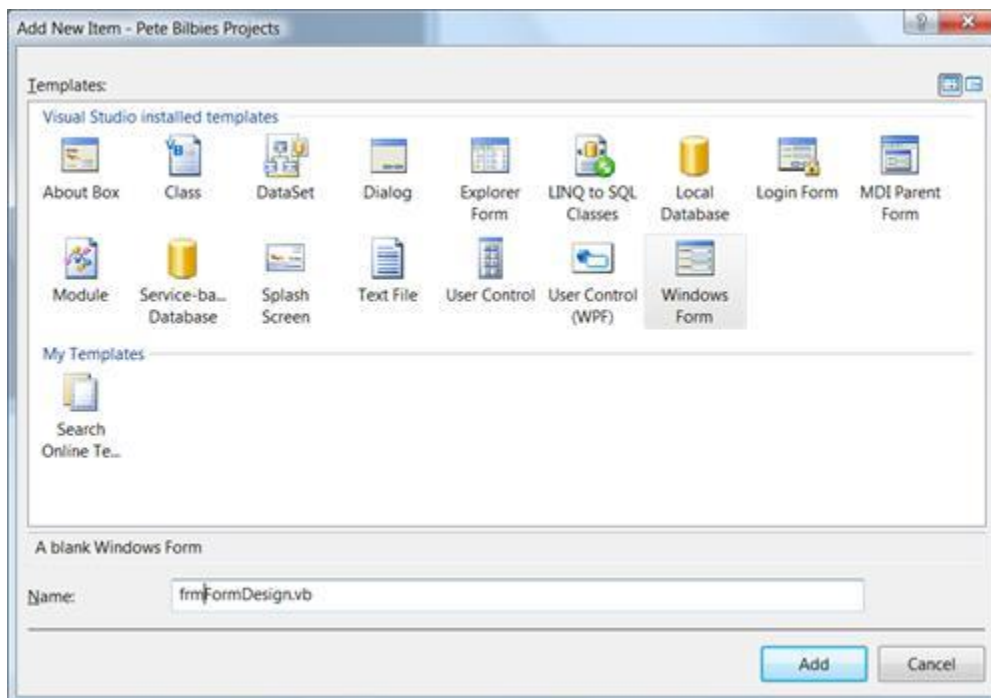


ADDING OTHER FORMS

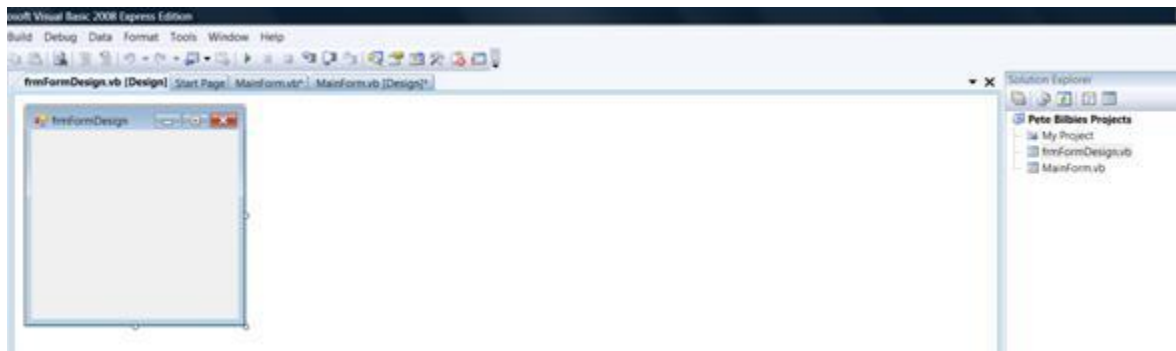
To add other forms to the application select **Project->Add Windows Form** from the main IDE menu – see below



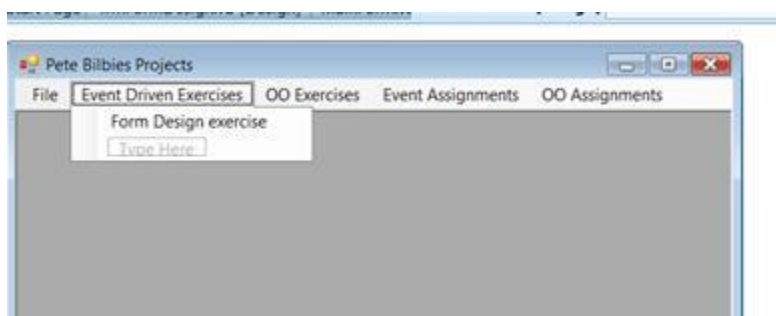
In the next window make sure Windows Form is highlighted and give the form a suitable name i.e. frm Form Design and click **Add**. Notice that I have used frm as a prefix for the name. When naming controls we use certain naming conventions so that we can identify the type of control we are using. There is a list of naming conventions for common controls in the appendices of this lesson



The new form will be displayed and If you look at the Solution Explorer you will see the extra form in the list – see below



We are now ready to program the menu item that will allow access to the new form. Click on the Main Form.vb[Design] tab to open the Main Form and Double click on Form Design Exercise menu item to access the code window – see below:



The Click Event Handler for the menu is shown below:

```

Public Class MainForm
    Private Sub ExitToolStripMenuItem_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
        Application.Exit()
    End Sub

    Private Sub MainForm_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.
    End Sub

    Private Sub FormDesignExerciseToolStripMenuItem_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
    End Sub
End Class

```

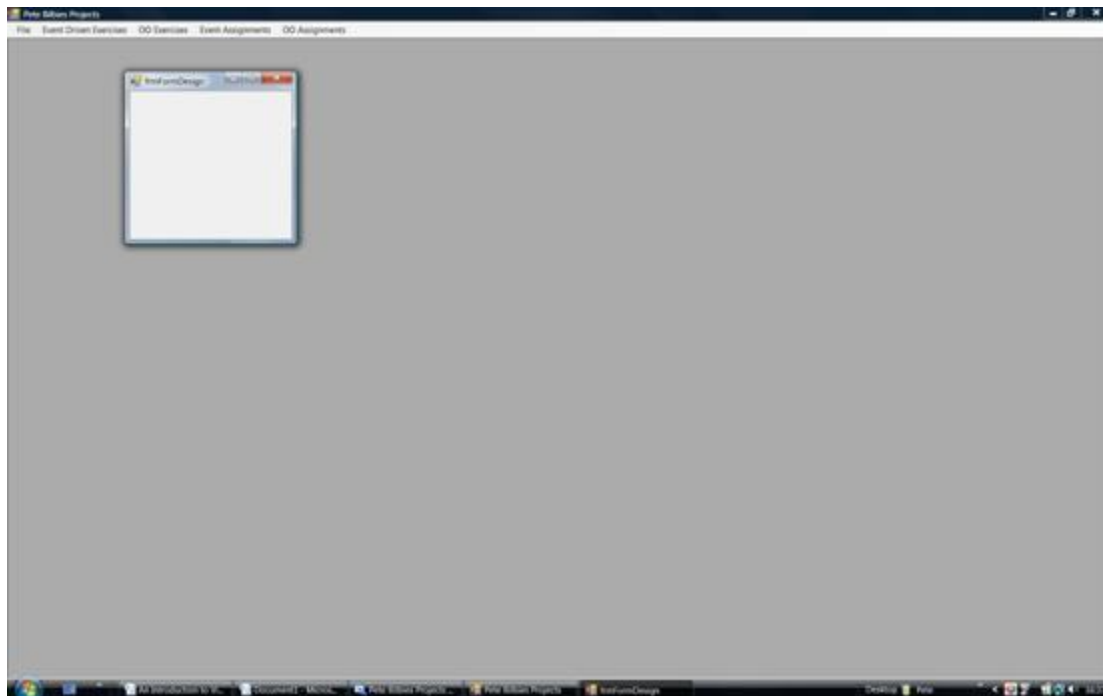
The minimum code needed to open the form is shown below:

```

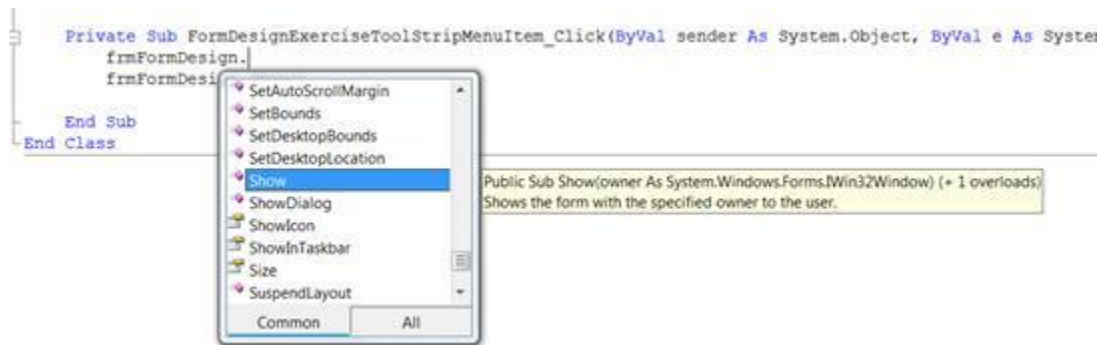
Private Sub Form Design Exercise Tool Strip Menu Item_Click(...)
    frm Form Design.Show()
End Sub

```

Execute the program and click on the menu item you have just coded. If you have followed the instructions the result should be as shown below:

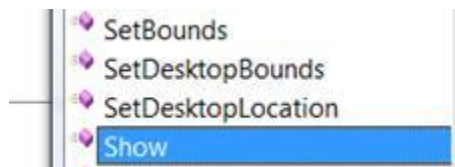


However, if you minimize the main form the other form does not minimize with it. To do this you have to 'tie' the new form to the main for in the code by making the new form a **Child** form of the **Parent** (main) form.

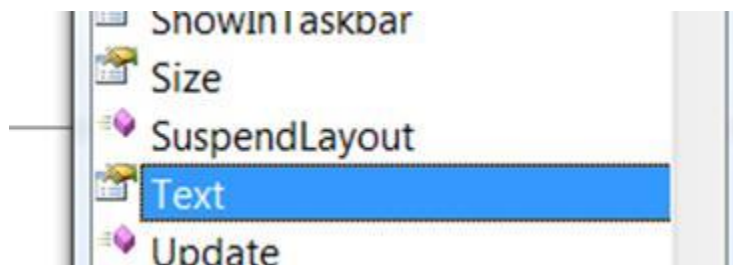


Each control has a list of methods (functions and procedures) and Properties which can be accessed using **dot** notation. To access these type the name of the control i.e. frm Form Design and add a full stop. The list will then be displayed – see below:

The methods are denoted by a flying purple box:



The properties are denoted by a hand holding a letter:



In the examples above **Show** is a method and **Text** is a property. We will discuss these concepts in greater detail in later sessions. The **Show** method will display the named form. A property of the form is **MDI Parent** i.e. `frmFormDesign.MdiParent`

This property has assigned to it the name of the form that is to be its MDI form in this example the **Main Form**. In VB.Net the = sign can be referred to as an **assignment operator**. When you are adding code to a form, if you want to refer to the form you can use the reserved word **Me**. Therefore the following statement assigns the **Main Form** to be the **MDI Parent** of the **frm Form Design** form:

```
frm Form Design.MdiParent = Me
```

Add the previous line of code to the Click Event of the menu item i.e.

```
Private Sub Form Design Exercise Tool Strip Menu Item_Click(...
```

```
    frm FormDesign.MdiParent = Me
```

```
    frm Form Design.Show()
```

```
End Sub
```

Execute the program and when you minimize the Main Form the frmFormDesign form will minimize with it

Summary

We have covered many concepts in this session i.e.

- Creating a new project
- Using the Solution Explorer
- Using the Properties window
- Using the toolbox
- Creating an MDI (parent) form
- Creating a menu with sub-menus
- Adding new forms to a MDI application (child forms)
- Properties and Methods

- Naming conventions for controls
- Coding Events

APPENDIX

PREFIXES AND NAMING CONVENTIONS

When we add controls to a form we must always remember to name the control correctly and to do this we use a convention of prefixes for different controls. The prefix for a menu item is menu i.e. menu File. Using this convention for all controls makes it easier to read code and identify particular controls, however, we must also give the controls meaningful names as well i.e. menu File, mnuExercise2 etc. The prefix is always shown in lower case except for forms which always have a upper case first letter to the prefix i.e. Frm My Form. Another tip to aid readability of the code is to capitalise the first letter of every word used in a name i.e. text Students Address – a textbox for inputting a student's address (spaces are not allowed). I always use this convention for variable and procedure names (more on this subject later)

A table of the most common control prefixes is shown below:

Control	Prefix
Label	lbl
Textbox	txt
Command Button	btn
Form	Frm
Combo Box	cbo
Check box	chk
List box	lst
Radio button	rad
Menu item	mnu
Timer	tmr

VISUAL PROGRAMMING

Visual Basic Coding Standards (VBCS)

WHY HAVE CODING CONVENTIONS

Visual Basic design and coding conventions are important for a number of reasons:

- Software is read many more times than it is written, often by someone other than its developer.
- Coding conventions increase the readability of the software, permitting software engineers to understand and maintain software more quickly and correctly.
- From 70% to 90% of the cost of software is in maintenance rather than development.
- The original developer will rarely maintain any software for its entire life.
- Software is a critical part of our products; it should be just as well engineered, packaged, and documented as our hardware.
- Providing standard coding conventions permits the developer to concentrate on more important issues such as design.

DEFINITIONS

A *convention* is one of the following:

- A *standard* (S) is a mandatory constraint on the content or format of something (e.g., code, documentation). You must follow standards unless you obtain a waiver from the Process Team and you document the deviation from the standard.
- A *guideline* (G) is an optional constraint on the content or format of something (e.g., code, documentation). You should follow guidelines in most cases. You must document any deviation from the guideline.
- A *recommendation* (R) offers advice. It is less constraining than either a standard or guideline. You may follow or ignore recommendations as appropriate. You need not document any deviations from recommendations.

Visual Basic (VB for short) refers to any programming environment, which uses a Visual Basic-like language. The standards in this document apply to these other environments, as well as Visual Basic itself:

- Visual Basic for Applications (VB A), used in Microsoft Office products
- Visual Basic Script (VBScript), used in Active Server Pages and in MS Outlook
- LotusScript, used in Lotus Notes.

A *module* is a block of code (declarations and procedures) which VB stores in a single file. Forms, Classes, UserDocuments, and Code Modules will all be referred to as modules in this document. In LotusScript, each object (Agent, View, Form) may be considered a module. In VBScript, each file may be considered a module.

Procedure refers to a Sub, Function, or Property Let, Get, or Set function.

USAGE GUIDELINES

The following sections provide standards and guidelines for the general usage of Visual Basic. These standards ensure consistent usage of various Visual Basic features.

OPTION STATEMENTS

- S) Use Option Explicit. Option Explicit should be included at the beginning of each module to enforce explicit declarations for all variables. In the VB environment, the option to “Require Variable Declaration” should be set. This tells VB to include Option Explicit in all new modules.

Rationale: When variables can be instantiated on the fly, typographical errors in variable names are not caught by the compiler and can cause run-time errors.

- S) Do not use Option Base. The default lower bound of 0 (zero) should be used for arrays; do not use Option Base to change it. If a different lower bound is needed for an array, explicitly define both the lower and upper bound of the array in the Dim or Re Dim statement.

Rationale: Most programmers will assume that the lower bound of arrays is 0. A change in the lower bound will not be apparent to someone in the middle of reading the code. But this will not be apparent to someone

Example:

‘* Array from 0 to 10, using default lower bound.
Dim a Words(10) as String

‘* Array from 1 to 12, both bounds explicit.

Dim a Months(1-12) as String

DECLARATIONS

The following conventions help you build good abstractions that maximize cohesion:

S) Use narrow scoping. Use private variables and constants whenever feasible. Local (procedure-level) variables are preferred, followed by module-level, with global as a last resort.

Rationale: Increases cohesion.

S) Declare module level variables in this order:

1. Public API Declarations
2. Public Constants
3. Public Types
4. Public Variables
5. Private API Declarations
6. Private Constants
7. Private Types
8. Private Variables

Rationale: A consistent declaration order makes the module header easier to scan. The prescribed order provides clear separation between Public and Private declarations, and suggests an appropriate dependency between Constants, Types, and Variables.

S) Declare scope explicitly. Use Private and Public, instead of Dim, for variables declared outside of procedures.

Rationale: VB has different defaults for each type of code module (form, class, etc.). Explicitly declaring the scope of variables reduces confusion.

R) Declare variables and constants at the beginning of their applicable scoping construct (module or procedure). VB allows you to declare variables anywhere within a module or procedure, but they should be placed at the top.

Rationale: This makes declarations easy to view, and avoids cluttering up the flow of executable code. Usually the rationale for putting the declarations in the middle of the executable code is that it declares variables near the point where they are used. But this is only important if procedures are allowed to grow beyond a manageable size. If a block of code is independent enough to require its own variables, it belongs in a separate procedure.

S) Use explicit types. Each variable should be declared with an explicit type, to prevent VB from creating all variables in the Variant type. If Variants are needed, they should be declared explicitly As Variant.

Rationale: Variants should be avoided because they are less efficient than defined types, and because their use prevents VB from catching errors through type-checking.

Exceptions: The VBScript environment only supports Variant types, and does not allow you to declare a variable as an explicit type. In VBScript, comments and naming conventions should be used to inform the reader as to the intended use of the variable.

T) Use narrow typing for objects. Don't use an Object type when the true type of the object is known. For example, if a variable contains a reference to a Text Box, the variable should be declared As Text Box, not As Object or As Control

Rationale: Narrow typing makes the true purpose of the variable more obvious. It also allows VB to catch type mismatch errors at compile time, and execute more quickly at run time through early binding.

ERROR HANDLING

The following conventions help you to handle errors effectively:

R) Include error handling in every event procedure. Each event procedure should have an On Error statement as its first executable statement.

Rationale: In VB, unhandled errors result in termination of the program. Since all code is executed as the result of an event, event procedures represent the last chance to handle errors, which bubble up from other procedures.

R) Include error handling in every procedure where errors are likely to occur. This includes any procedure that performs I/O, database functions, operating system calls, COM operations, or invokes methods on other objects. Errors should be handled in the same procedure where they occur.

Rationale: Localized error handling increases cohesion, and allows you to provide error handling specific to the types of errors that are likely to occur. The farther that a project-specific error bubbles up, the less likely that it will make sense to the procedure that must handle it.

- R) Use common error trapping label names. Use the same label name (for example, “Err-trap”) in all procedures

Rationale: This improves the consistency and readability of the code.

Example:

```
Sub cmdOK_Click
    On Error Go to Err-trap

    '* ... More code here ...

Exit Sub
```

Err-trap:

```
MsgBox “Error “ & Err.Number & “: “ & Err.Description & “ in “ & Err.Source,
vb Exclamation, App.Title
```

```
End Sub
```

TESTING

- R) **Use Debug.Print Statements.** Print the value of variables to help confirm intermediate results and confirm that code is executing as expected. Debug.Print statements are ignored when the code is compiled

Rationale: Makes code verification and debugging easier.

SOURCE CODE FORMATTING

The following standards provide a consistent look and feel to the Visual Basic code and improve its design.

CODE ORGANIZATION

The following conventions will help to ensure that the code is organized into easy-to-understand pieces.

R) Keep modules to a manageable size. Each module should be less than about 1000 lines, and contain a set of procedures that are related to each other. If a module is becoming larger than 1000 lines, consider splitting it based on some functional criteria.

Rationale: Small, cohesive modules are more manageable and easier to understand. They also reduce sharing conflicts in version control tools.

R) Avoid putting general code in Form modules. Form modules (and other modules with a visual interface) should contain only the event procedures for the form. Procedures called by the event procedures should be placed in other modules unless they deal directly with many of the controls on the form itself.

Rationale: Form modules tend to grow large through their event procedures and visual interface definitions. Moving other code into other modules will help to keep Form modules small and focused on their main purpose, which is interacting with the user.

Because Forms consist of both binary and textual components, they are not handled as well by version control tools. Keeping code in other modules minimizes conflicts.

R) Keep procedures to a manageable size. Each procedure should be concise enough that its entire purpose can be easily expressed and understood. One rule of thumb is that the well-commented procedure should fit on a printed page. A more relevant rule when editing the code is that it fit on one screen in the development environment.

Rationale: Small procedures are easier to understand and verify.

Exception: A single control structure (for example, a Select structure) may be unavoidably longer than the recommended procedure length. In this case, the control structure should be placed in its own procedure, separate from other code, so as not to obscure the function of the code around it.

INDENTATION

G) Use the standard indentation provided by Visual Basic tabs. The VB environment provides tools that make it easy to indent blocks of code by tab stops.

Rationale: Indentation improves readability. Using the standard tab-stop indentation improves maintainability, allowing the indent level of blocks of code to be easily changed.

Exception: LotusScript indents automatically, and does not support tabs.

G) Use four spaces for each indent level. The default tab spacing of in VB is four spaces, and this should be maintained.

Rationale: Four spaces is enough to make the indentation level obvious, without taking up too much horizontal space.

Exception: LotusScript indents automatically

SPACES

R) Apply spaces liberally.

Rationale: Spaces break up the code, making it easier to read.

Example 1:

```
void some Method (TypeX aTypeX, TypeY aTypeY) throws Some Exception;
```

MAXIMUM LINE LENGTH

G) Avoid lines longer than 80 characters.

Rationale: Longer lines are more difficult to read. Also, many terminals and tools do not handle longer lines well.

G) Break up long procedure declarations on to multiple lines with each parameter aligned.

Rationale: This makes the code much easier to read and makes it less likely that maintenance will introduce new bugs.

Example:

```
Function Collect Stats(sFileName As String, _  
    iStartLine As Long, _  
    iEndLine As Long) _  
    As Long
```

G) Limit method and object reference cascades to three levels.

Rationale: Method cascades can be hard to understand and can be the source of synchronization and exception handling problems. It slower to execute statements using multiple cascades. Visual Basic must traverse the object hierarchy from the top for each statement, so it is more efficient to assign a variable to a lower-level object if it will be used for multiple statements.

Acceptable Examples:

```
returnValue = anObject.method1().method2()
```

```
returnValue = anObject.method1().method2().method3()
```

```
oSelection = Word.ActiveDocument.Selection  
i Start= oSelection.Start
```

Not Acceptable Example:

```
returnValue = anObject.method1().method2().method3().method4()
```

```
Word.ActiveDocument.Selection.Font.Bold = True
```

BLANK LINES

S) Put a blank line between logical sections of a procedure.

Rationale: This improves readability.

Example:

```
Sub Process Data()
```

```
    Check Precondition
```

```
    Do Foo
```

```
    DoFam
```

```
    Check Post condition
```

End Sub

Comment: If the method is long or violates other standards, it should be factored based on the logical sections described above.

COMMENTS

The following Visual Basic comment standards provide a consistent approach to documenting code that will also improve the quality of the associated java documentation.

COMMENT FORMATTING

- R) Begin comments with '*' (an apostrophe, an asterisk, and a space). All VB comments start with an apostrophe and end with a line break. This convention adds an asterisk and space before the actual comment text.

Rationale: When viewing VB code, the apostrophe alone does distinguish the comment well from the code. The asterisk and space improves legibility by setting the comments off more clearly.

MODULE COMMENTS

- S) Write a comment block at the top of each module. The comment block will include the name of the module, a description its purpose, its authors, and its revision history. It should also contain version and copyright information, if appropriate. Each entry in the revision history will include the date that the change was made, the author who made the change, the reason that the change was made, and a list of the procedures affected.

Rationale: Module-level comments help future programmers to understand and respect the purpose of the module, and to consider the potential impacts before making revisions.

PROCEDURE COMMENTS

- S) Begin each procedure with a comment block. The comment block shall immediately follow the declaration of the procedure, and contain a description of the procedure, including the reason why it is needed. Following that should be a list of the pre- and post conditions necessary for its operation, a description of its parameters and return value, and possible errors which may occur.

Rationale: Well-commented procedures make the code easier to understand.

Example:

Function Check Spelling(text Comments As VBA.Text Box) As Boolean

- ‘* Run the spell checker on the textbox. This prevents
- ‘* spelling errors from being stored and displayed to the
- ‘* customers.

- ‘* Preconditions: Textbox has been filled
- ‘* Post conditions: The data in the textbox may change if
- ‘* spelling corrections have occurred

- ‘* Parameters:
- ‘* text Comments – Text Box containing free-form text
- ‘* Return Value:
- ‘* Boolean indicating whether spelling errors were found.

DECLARATION COMMENTS

G) Describe the abstraction of all declared constants, types, and variables, the meanings of which are not obvious from the name. This includes both module- and procedure-level declarations. Also, describe any constraints of the variables.

Rationale: This will make the code easier to understand and maintain.

Example:

```
Dim previous As Long          ‘* Previous cursor position
                               ‘* (0..MAXPOS)
```

GENERAL COMMENTS

G) **Do not comment obvious code.** If the code is not obvious, then first see if you can rewrite the code so that the use of comments will be unnecessary.

Rationale: This will make the code easier to maintain.

Bad Example:

```
Dim i Connection Count As Long ‘* number of connections
```


G) Where practical, line up trailing comments.

Rationale: This will make the comments easier to see, compare, and maintain.

Example:

If Not Data Ready() Then '* precondition fails

 Error 30005, "Data not available"

Else '* normal case

 Process Data

End If

S) Use proper spelling, punctuation, and spelling in comments.

Rationale: Poorly written comments are distracting and can obscure the intended meaning.

MODULE NAMES

S) Module names should be meaningful words or phrases that describe the abstraction of the module. Use nouns for object abstractions and verbs for functional abstractions. Module names should be without spaces, in mixed case, with the first letter uppercase, and the first letter of each subsequent word capitalized. The file name of the module should match the internal name of the module

Rationale: Consistency with standard Visual Basic naming conventions. Ease of deployment.

Examples:

Show Matches.frm

Registry.bas

Send Msg.bas

Customer.cls

PROCEDURE NAMES

GENERAL PROCEDURE NAMES

- S) Procedure names should be meaningful verbs or verb phrases that describe the functional abstraction of the procedure. The first word should be a verb. Procedure names should be without spaces, in mixed case, with the first letter uppercase, and the first letter of each subsequent word capitalized.

Rationale: Consistency with standard Visual Basic naming conventions. Encourages names which allow the reader to visualize the purpose of the procedure

Example: Open Data File

BOOLEAN FUNCTION NAMES

- S) Functions that return the results of a test of a Boolean condition T should be named IsT, HasT, or CanT.

Rationale: Consistency with standard Visual Basic naming conventions. Encourages readable code when the function is used in a condition.

Examples:

Connection.Is Available()

Router.Has Pending Request()

If Is Broken() Then Exit Sub

CONVERSION METHOD NAMES

- S) Name methods that convert an object to a particular format F ToF.

Rationale: Consistency with standard Visual Basic naming conventions.

Example: to String

STATE MODIFYING METHOD NAMES

- S) Name methods that modify the state of an object to state S BeS.

Rationale: Consistent naming.

Examples:

Be Available()

Be Unavailable()

VARIABLE AND PARAMETER NAMES

Variable and parameter names should be meaningful noun or noun phrases, without spaces, with the first letter lowercase and the first letter of any subsequent words capitalized. The first few letters of the variable name define the type of the variable. The remainder of the name should describes the role that the variable plays.

ATOMIC TYPES

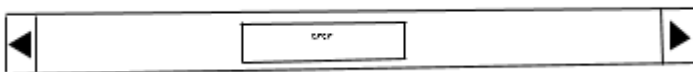
S) For atomic types, the first letter of the name defines the type, as follows:

Prefix	Types	Example
i	Integer or Long	i Word Count
f	Float (Single or Double)	f Radius
s	String	s First Name
b	Boolean	b Quiet
c	Currency	cCurrentBal
d	Date	dStartTime

Rationale: Consistency with standard Visual Basic and Windows API naming conventions. Allows easy identification of the type and role of a variable, without adding a lot of naming overhead for frequently used types.

OBJECT AND COMPLEX TYPES

COMBO BOX



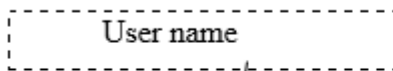
This is a scrow box which is used for top/ down scrowing and right/left scrowing.

CHECK BOX

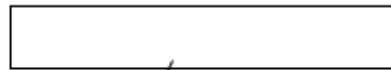
This is the box which allow you to select what you need



TEXT BOX / LABEL



Label box



Text box

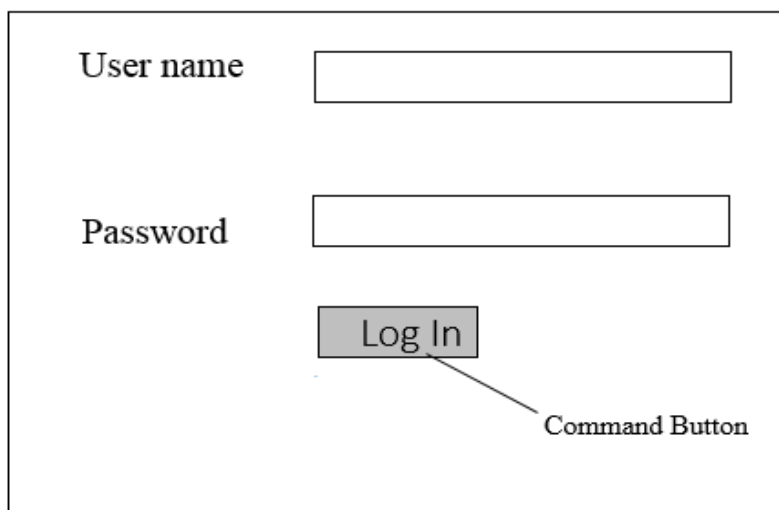
Text box allow user to write a text inside it and you can delete it.

Label is a place where developer is write a text he/ she need to appear at the form.

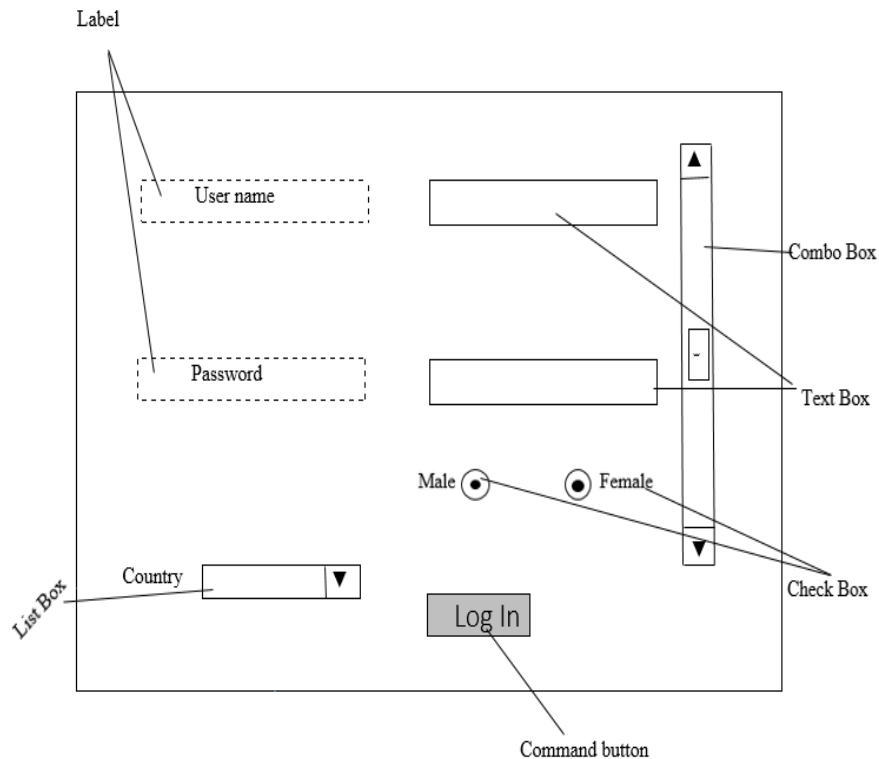
Command Button

Command Button is one which is used to send data to the database when user click it

Command button respond when user click on it



FORM WITH VISUAL BASIC



S) For objects and complex types, and three-letter prefix defines the type, as follows:

Prefix	Type	Example
frm	Form	frm Crustiness
txt	Text Box	text Last Name
lst	List Box	lstServiceTypes
cmd	Command Button	cmd Cancel
cbo	ComboBox	cbo Category
pic	PictureBox	pic Logo
chk	CheckBox	check Primary Address
opt	Option Button	opt GenderFemale
tmr	Timer	timer Elapsed
lbl	Label	lbl copyright
tbr	Toolbar	tbr Editing
ctl	Control (if type is not known)	ctl Sortable List
col	Collection	col Form Fields

obj	Object (if type is not known)	obj Parent
var	Variant (if type is not known)	var Next Field

Many other types are possible which are not on this list, including user-defined types and objects. For these types, and appropriate three- or four-letter prefix convention should be established and documented for the project.

(Note that the frm prefix should be used only for a variable in the code which holds a form reference, and not the VB class and file which defines the form. Form modules should be named as per the section “Module Names”, above.)

Rationale: Consistency with standard Visual Basic and Windows API naming conventions. Allows easy identification of the type and role of a variable.

CONSTANT NAMES

S) The names of constants should be meaningful noun or noun phrases with all letters capitalized and underscores between words.

Rationale: Consistency with standard Visual Basic and Windows API naming conventions. This convention emphasizes and differentiates constants from variables.

Example: MAXIMUM_NUMBER_OF_FILES

ARRAY NAMES

S) Array names should begin with a lower case “a”, followed by the prefix which describes the type of the array members, as described above, and followed finally by the descriptive name of the variable.

Note that Variants must often be used to hold arrays when they are passed as return values from functions. In this case, the Variant should be named according to the array naming convention.

Rationale: This identifies the variable as an array, and indicates its contents as well.

Example:

Dim as User Names(10) As String

```
Dim afVertices() As Single
```

```
Dim as Words As Variant  
as Words = Split(sLine, vb Tab)
```

GLOBAL VARIABLE NAMES:

- S) Names for global variables should follow the above conventions, but with a “g_” prefix before the name. Likewise, private module-level variables should have a “m_” before the name.

Rationale: This eases identification of global and module-level variables. While this convention is somewhat cumbersome, it appropriately discourages use of global and module-level variables.

Example:

```
Public g_sCurrentUser As String
```

```
Private m_aiTroubleCodes As Integer
```

MISCELLANEOUS

- S) **Do not use abbreviations or acronyms** unless they are extremely popular abbreviations in the project. The prefix standards described above or established by the project are examples of popular abbreviations, which are exempted.

Rationale: Abbreviations and acronyms make the code harder to understand.

Example:

sOutstandingBalance instead of sCurOutBal

- S) **Document cases where the client ignores the return value.**

Rationale: This is typically an error; if not, make the intent clear.

- S) **Use plural for collections and singular for individual objects.**

Rationale: This makes the names easier to understand.

Examples:

Dim colOpenForms As Collection

Dim asApprovedNames() As String

AN INTRODUCTION OF COMPUTER SECURITY AND PRIVACY

Hacking:

- Hacking is the process of gaining unauthorized access to a computer system for fun and challenge of it.



Hackers:

- Hackers are people who gain unauthorized access to a computer for fun and challenge of it.

WHAT IS COMPUTER SECURITY?



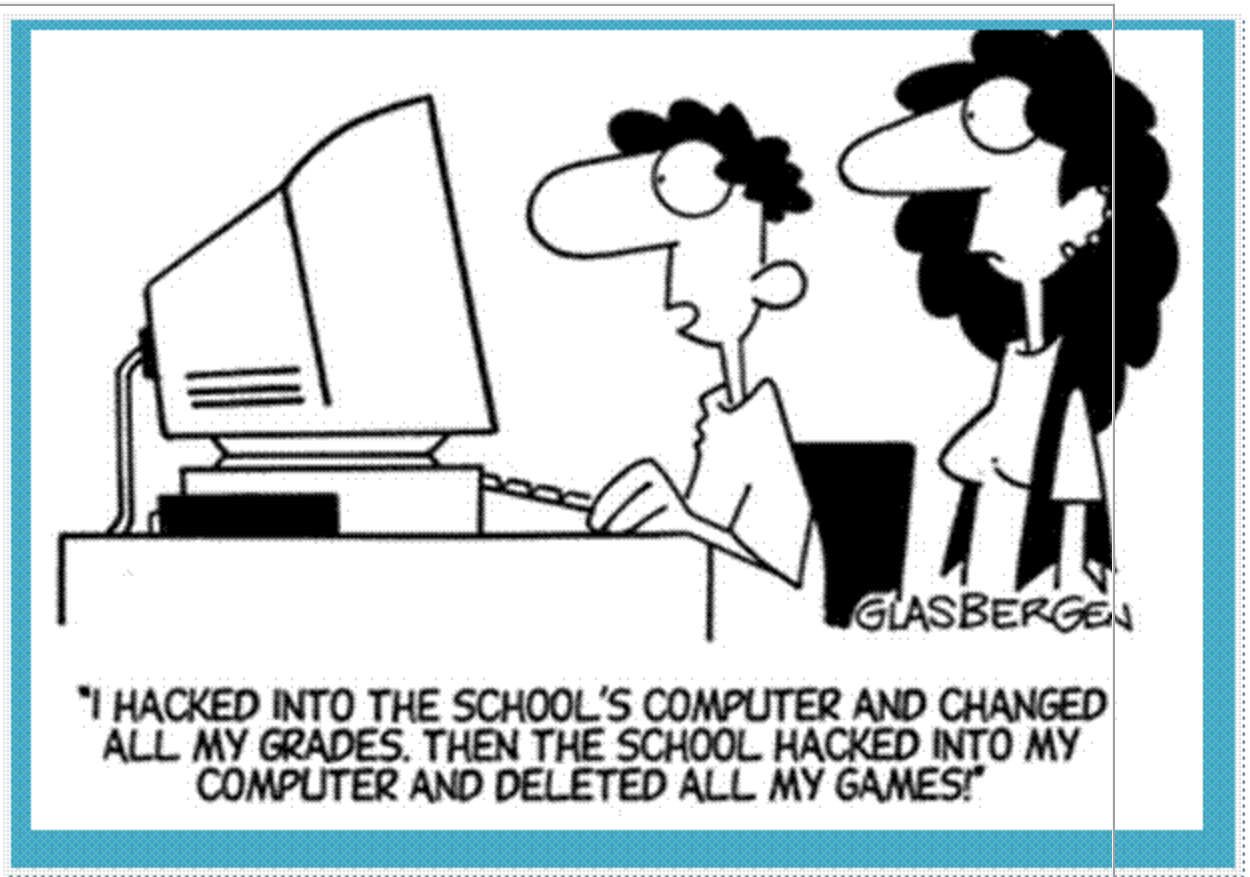
Is the field of computer science that analyzes the security properties of computer systems, OR IS the protection of resources (including data and programs) from accidental or malicious modification, destruction, or disclosure.

WHY IS IT IMPORTANT?

- 📁 Information is power and money
- 📁 Computer systems manage information and provide mission-critical support for business, government, and financial institutions

WHY IS HACKING SO BAD?





-Hacker can view your private information.

-Hacker can edit your data.

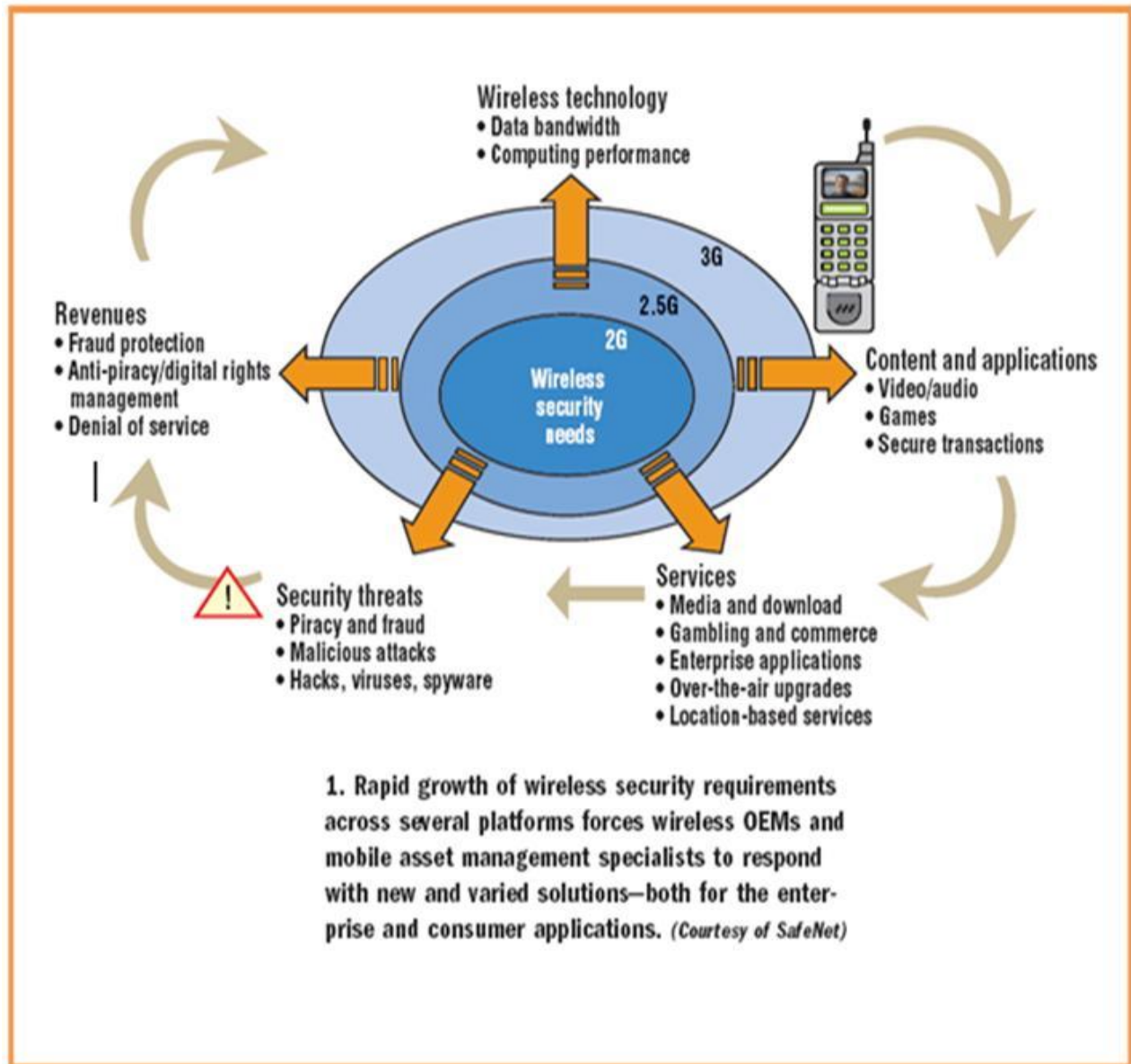
-Hacker can misuse other personal information.

-Can damage systems e.g. E- Bank system.

-Can steal money through network.

Computers are everywhere

- 📖 Computer systems constantly grow in complexity (and size)
- 📖 Today's networks are very heterogeneous, and critical components are often connected (maybe in indirect ways) to non-critical, poorly managed computer systems
- 📖 People make mistakes in both the development and the deployment of computer systems



Home Users Increase Vulnerabilities

Today most homes are connected, particularly with the advent of DSL and cable modems

Most home users:

- 📖 Are unaware of vulnerabilities
- 📖 Don't use firewalls
- 📖 Think they have nothing to hide or don't care if others get their data

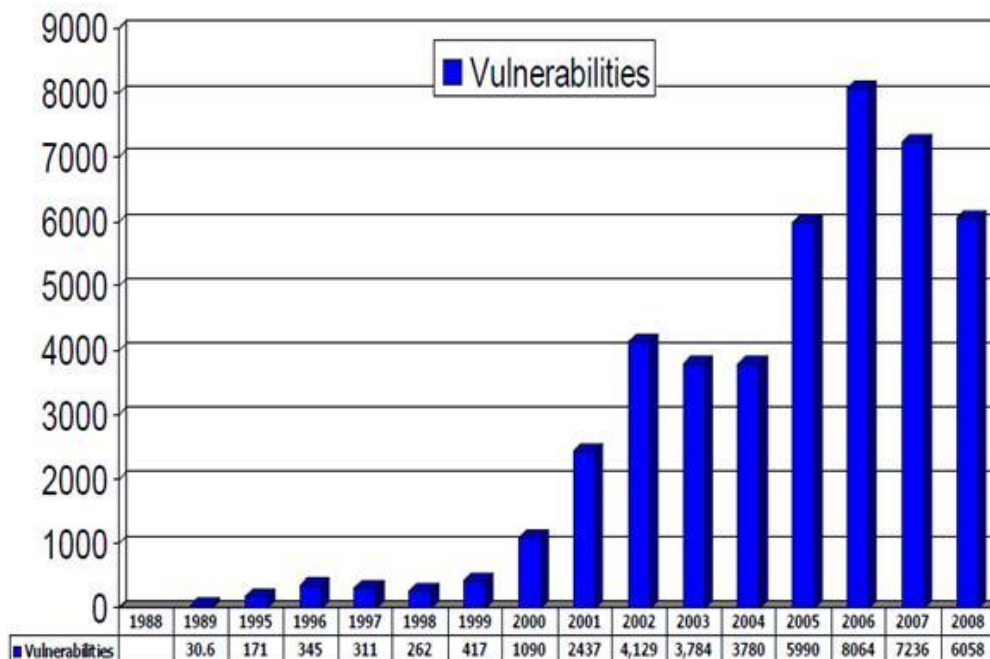
- ☞ Don't realize their systems can serve as jump off points for other attacks (zombies or bots)

Computer security is reactive

- ☞ usually reacting to latest attack
- ☞ offense is easier than defense

Security is expensive both in dollars and in time .There is not now, and never will be, a system with perfect security.

Security Vulnerabilities



SECURITY INCIDENTS

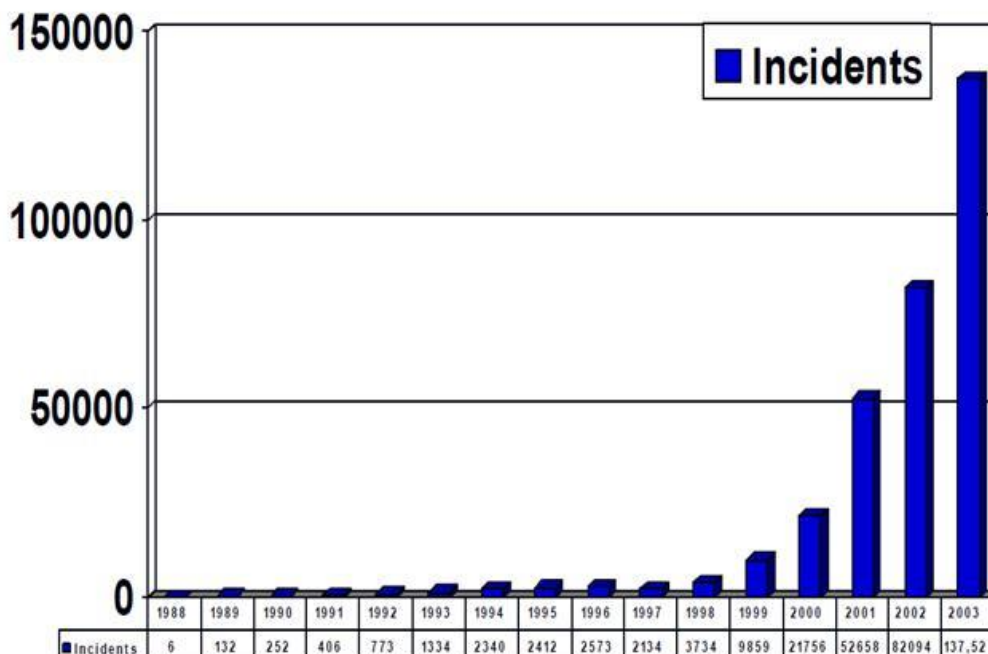
Who are the attackers?

- ☞ **Script kiddies** download malicious software from hacker web sites
- ☞ **Hackers** trying to prove to their peers that they can compromise a specific system

- 📖 **Insiders** are legitimate system users who access data that they have no rights to access
- 📖 **Organizational level attackers** use the full resources of the organization to attack

After September 11, 2001 the idea of nation State level cyber attacks being carried out by Terrorists became a big concern more recently, most attacks are financially motivated. There is a complete cyber underground economy

Security Incidents

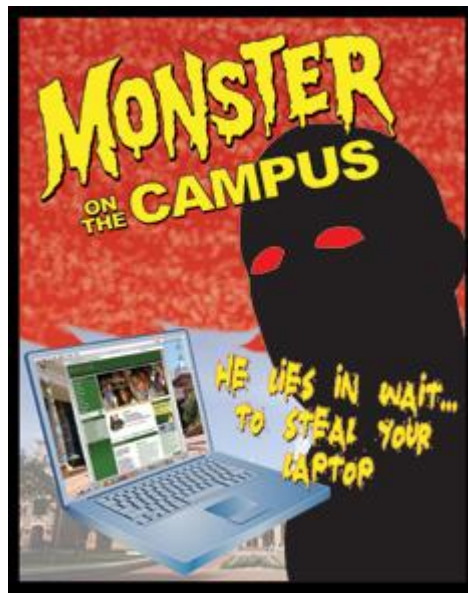


Source: CERT

Types of computer security

1. Physical Security/Hardware security
2. Network security/Software security
3. Data security

1. **Physical Security:** is the protection a hardware, data, networks from physical harm like thief. Hardware security refers to practices regarding how physical devices and computer hardware are handled and overseen to protect them from physical harm like theft.



The physical server mainframes that often house various networks and Internet websites can be damaged, resulting in loss of data, or they could be physically attacked in an effort to steal information directly from the system through data transfer between devices.

Many companies and individuals should also be aware of considerations regarding different types of computer security and physical theft. As computer technology improves, memory and data storage devices have become increasingly smaller. This means that someone can steal a single computer tower or laptop from a business or a person's home and potentially obtain vast amounts of data and information that may be private. Small data storage devices, such as thumb drives, should also be protected, as someone may carelessly forget such a device at a public computer terminal and create a very real opportunity for data loss.

Lock up the server room

Even before you lock down the servers, in fact, before you even turn them on for the first time, you should ensure that there are good locks on the server room door. Of course, the best lock in the world does no good if it isn't used, so you also need policies requiring that those doors be locked any time the room is unoccupied, and the policies should set out who has the key or key code to get in.

The server room is the heart of your physical network, and someone with physical access to the servers, switches, routers, cables and other devices in that room can do enormous damage.

Set up surveillance

Locking the door to the server room is a good first step, but someone could break in, or someone who has authorized access could misuse that authority. You need a way to know who goes in and out and when. A log book for signing in and out is the most elemental way to accomplish this, but it has a lot of drawbacks. A person with malicious intent is likely to just bypass it.

A better solution than the log book is an authentication system incorporated into the locking devices, so that a smart card, token, or biometric scan is required to unlock the doors, and a record is made of the identity of each person who enters.

A video surveillance camera, placed in a location that makes it difficult to tamper with or disable (or even to find) but gives a good view of persons entering and leaving should supplement the log book or electronic access system. Surveillance cams can monitor continuously, or they can use motion detection technology to record only when someone is moving about. They can even be set up to send e-mail or cell phone notification if motion is detected when it shouldn't be (such as after hours).



Make sure the most vulnerable devices are in that locked room

Remember, it's not just the servers you have to worry about. A hacker can plug a laptop into a hub and use sniffer software to capture data traveling across the network. Make sure that as many of your network devices as possible are in that locked room, or if they need to be in a different area, in a locked closet elsewhere in the building.

Use rack mounts servers

Rack mount servers not only take up less server room real estate; they are also easier to secure. Although smaller and arguably lighter than (some) tower systems, they can easily be locked into closed racks that, once loaded with several servers, can then be bolted to the floor, making the entire package almost impossible to move, much less to steal.

Don't forget the workstations

Hackers can use any unsecured computer that's connected to the network to access or delete information that's important to your business. Workstations at unoccupied desks or in empty offices (such as those used by employees who are on vacation or

have left the company and not yet been replaced) or at locations easily accessible to outsiders, such as the front receptionist's desk, are particularly vulnerable.

Disconnect and/or remove computers that aren't being used and/or lock the doors of empty offices, including those that are temporarily empty while an employee is at lunch or out sick. Equip computers that must remain in open areas, sometimes out of view of employees, with smart card or biometric readers so that it's more difficult for unauthorized persons to log on.

Keep intruders from opening the case

Both servers and workstations should be protected from thieves who can open the case and grab the hard drive. It's much easier to make off with a hard disk in your pocket than to carry a full tower off the premises. Many computers come with case locks to prevent opening the case without a key.

You can get locking kits from a variety of sources for very low cost, such as the one at Innovative Security Products.

Protect the portables

Laptops and handheld computers pose special physical security risks. A thief can easily steal the entire computer, including any data stored on its disk as well as network log on passwords that may be saved. If employees use laptops at their desks, they should take them with them when they leave or secure them to a permanent fixture with a cable lock, such as the one at PC Guardian.

Handhelds can be locked in a drawer or safe or just slipped into a pocket and carried on your person when you leave the area. Motion sensing alarms such as the one at Security Kit.com are also available to alert you if your portable is moved.

For portables that contain sensitive information, full disk encryption, biometric readers, and software that "phones home" if the stolen laptop connects to the Internet can supplement physical precautions.

Pack up the backups

Backing up important data is an essential element in disaster recovery, but don't forget that the information on those backup tapes, disks, or discs can be stolen and used by someone outside the company. Many IT administrators keep the backups next to the server in the server room. They should be locked in a drawer or safe at the very least.

Ideally, a set of backups should be kept off site, and you must take care to ensure that they are secured in that offsite location.

Don't overlook the fact that some workers may back up their work on floppy disks, USB keys, or external hard disks. If this practice is allowed or encouraged, be sure to have policies requiring that the backups be locked up at all times.

Disable the drives

If you don't want employees copying company information to removable media, you can disable or remove floppy drives, USB ports, and other means of connecting external drives. Simply disconnecting the cables may not deter technically savvy workers. Some organizations go so far as to fill ports with glue or other substances to permanently prevent their use, although there are software mechanisms that disallow it. Disk locks, such as the one at Security Kit.com, can be inserted into floppy drives on those computers that still have them to lock out other diskettes.

Protect your printers


You might not think about printers posing a security risk, but many of today's printers store document contents in their own on-board memories. If a hacker steals the printer and accesses that memory, he or she may be able to make copies of recently printed documents. Printers like servers and workstations that store important information, should be located in secure locations and bolted down so nobody can walk off with them.

Also think about the physical security of documents that workers print out, especially extra copies or copies that don't print perfectly and may be just abandoned at the printer or thrown intact into the trash can where they can be retrieved. It's best to implement a policy of immediately shredding any unwanted printed documents, even those that don't contain confidential information. This establishes a habit and frees the end user of the responsibility for determining whether a document should be shredded.

Summary


Remember that network security starts at the physical level. All the firewalls in the world won't stop an intruder who is able to gain physical access to your network and computers, so lock up as well as lock down.


2. Network security/Software security

 **Enable Encryption** Use 128-bit of encryption or higher. There are two different types of encryption WEP and WPA. WEP is weak and can be cracked easily within few minutes with software's available online. WPA is strong and uses TKIP encryption while WPA2 uses AES which is stronger than WPA.

 **Enable MAC Address filtering :**

Address will only allow specific devices to access the network. You can disable or permit certain MAC address to access the network.

 **Disable Remote Log in:** Remote log in can give anyone access to router setting remotely. This can be worst if an attacker tries to brute force router access and you're still using the default username and passwords. By default it's disabled on every router. Enable it only if you're updating your router remotely and disable it after the update is done.

 **Change SSID:** Change the default SSID name of your network. The SSID is the identifier name which identifies your network, so you can connect to it. Using the default SSID name will know that the router was setup by a novice and the attacker will try to brute force. This will make it worst, if you're using a default password.




 **Set password for router** Change the default password of the router. Use a random long password that cannot be easily guessed like include small letter, caps letter, numbers and special characters. That makes it Georgian College.



Fig: Show dialog box for setting password to route.


 **Secure and managing the network server.** - Install the server software on a dedicated host or on a virtual machine to test the software before installing it on the main server. - Create a logical partition for server data and remove software which are not required like gopher or FTP. And install server content on separate drive. - Configure the server to listen only on TCP and UDP ports. - Set an upload limit on the server, if your organization needs to upload files. Ensure that there is some software which scans the uploaded file before uploading on the server.


Configure the max number of connection. You might not want your server to get a DOS attack. - Check the server logs regularly and check if there was any intrusion or any suspicious activity. - Protect the log files, so if an attacker attacks the server, the attacker cannot get access to the log files to alter the data. –

 **Back up your server regularly,** so if any worst condition arises, you can always restore your server.

3. Data security:

Refers to ways in which attacks can be launched on data streams and software, without physical interaction of different devices or hardware.

 **Access control:** Access to confidential data must be provided on a least-privilege basis. No person or system should be given access to the data unless required by business process. In such cases where access is required, permission to use the data must be granted by the Data Steward

 **Sharing:** Protected data may be shared among the among University employees according to well-defined business process approved by the Data

Steward. It may be released publicly only according to well-defined business processes, and with the permission of the Data Steward.

- 📁 **Retention:** Confidential data should only be stored for as long as is necessary to accomplish the documented business process.
- 📁 **Incident Notification:** If there is a potential security incident that may place protected data at risk of unauthorized access, Its Technology Security Services must be notified:
- 📁 **Transit encryption:** Restricted data must be encrypted during transmission
- 📁 **Storage encryption:** Restricted data must be encrypted using strong, public cryptographic algorithms and reasonable key lengths given current computer processing capabilities. Keys must be stored securely, and access to them provided on a least-privilege basis (see ISO 11568 for recommendations on securing keys).

TYPES OF COMPUTER THREATS

- *Password Guessing*
- *Spotting*
- *Browsing*
- *Leakage*
- *Inference*
- *Tampering*
- *Accidental destruction*
- *Masquerading*
- *Denial of services*

Password Guessing

- i. Exhaustive search for passwords
- ii. Lists of commonly used passwords
- iii. Distributed default passwords
- iv. Password cracking programs readily available on the Internet.

📁 **Spoofing** :Duping a user into believing that he is talking to the system and revealing information

(E.g. password).

📁 **Browsing** :After an intruder has gained access to a system he may peruse any files that are available for reading and glean useful information for further penetrations

📁 Often done by legitimate users

📁 **Denial of Service** :Prevention of authorized access to computer resources or the delaying of time-critical operations

📁 **Masquerading**: Gaining access to the system under another user's account.

📁 **Leakage**: Transmission of data to an unauthorized user from a process that is allowed to access the data.

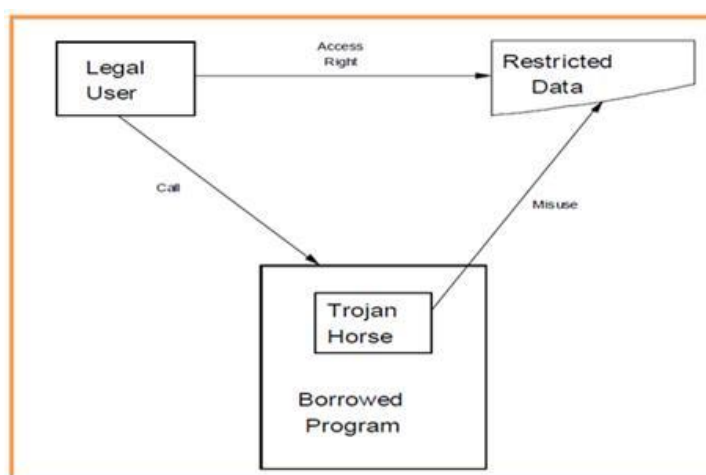
📁 **Tampering**: Making unauthorized changes to the value of information
Accidental Data Destruction Unintentional modification of information.


📁 **Trojan Horse** :A program that does more than it is supposed to do

📁 More sophisticated threat


📁 A text editor that sets all of your files to be publicly readable in addition to performing editing functions

📁 Every unverified program is suspect (especially games)



 **Trap Door** :A system modification installed by a penetration that opens the system on command

 May be introduced by a system developer

 Bogus system engineering change notice

 **Virus**: A program that can infect other programs by modifying them to include a possibly evolved copy of itself.


Examples of viruses

- i. Amiga Virus
 - Resident on boot block
- ii. IBM Christmas Virus
- iii. WWW Pages Containing Applets
- iv. MIME-encoded Mail
- v. Code Red Worm

TERMINOLOGIES

Confidentiality - Keeping data and resources hidden or protected from unauthorized disclosure

Integrity - Ensures that the data and programs are modified or destroyed only in a specified and authorized way

 Data integrity (integrity)

. Origin integrity (authentication)

Availability - ensures that the resources of the system will be usable whenever they are needed by an authorized user.

Browsing: Searching through main and secondary memory for residue information

Leakage: Transmission of data to an unauthorized user from a process that is allowed to access the data
Inference: Deducing confidential data about an individual by correlating unrelated statistics about groups of individuals

COMPUTER SECURITY THREATS

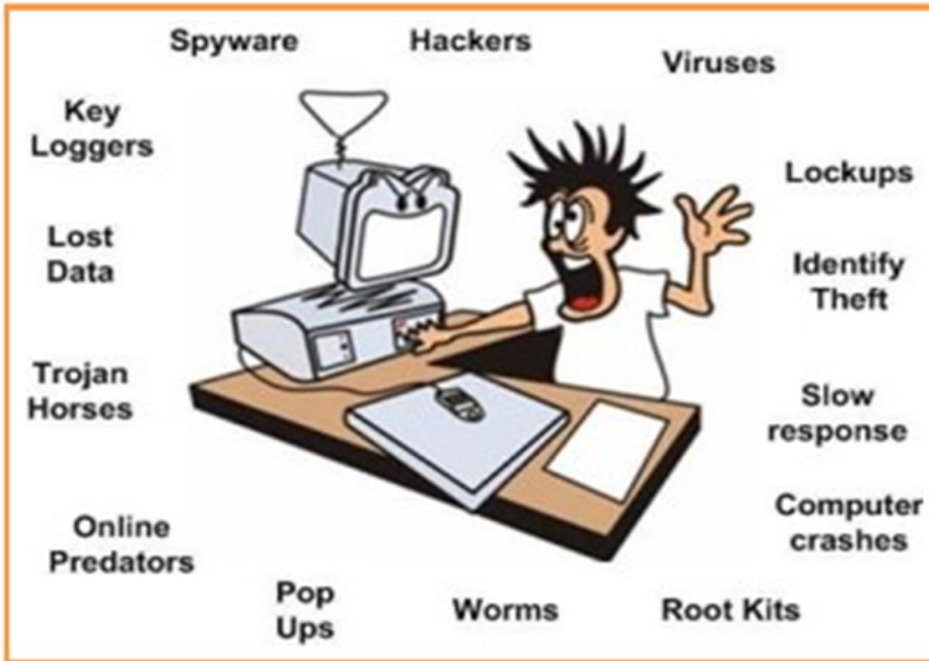


Fig:Show various computer security threats.

COMPUTER SECURITY TERMINOLOGIES

Threat: Is a potential violation of security.

Attacks: Are those actions which could cause a threat to occur.

Attackers: Are those who execute an attack Confidentiality, integrity, availability are enforced to counter the threats to the security of a system and foil attacks.

Vulnerability: Is a flaw in a system that allows a policy to be violated

Exploit: Is the act of exercising vulnerability also used to refer to an actual program, binary or Script that automates an attack.

Exposure: Is an information leak that may assist an attacker.

Disclosure :(unauthorized access to information).

Deception: (acceptance of false data).

Disruption: (interruption or prevention of correct operation).

Usurpation: (unauthorized control of some part of the system).

Security Policy

A security policy is a statement of what is and what is not allowed It defines the concept of "security" for a computer system.

Simple Example Policy

Policy disallows cheating – Includes copying homework, with or without permission.

ACCESS CONTROL

A means of limiting a user's access to only those entities that the policy determines should be accessed

TYPES OF ACCESS CONTROL

✓ **Discretionary Access Control (DAC)**

The owner specifies to the system what other users can access his files

(Access is at the user's discretion).

✓ **Mandatory Access Control (MAC)**

The system determines whether a user can access a file based on the fixed security attributes of the user and of the file.

Goals of Security: Given a security policy's definition of secure and insecure states, the corresponding security mechanisms can perform different functions.

METHOD, TOOL, OR PROCEDURE TO ENFORCE A SECURITY POLICY.

• **Prevention**

- Prevent attackers from violating security policy
- Example: use of passwords

• **Detection**

- Detect attackers' violations of security policy
- Example: use of logging of sensitive operations

• **Recovery**

- Stop attack, assess, and repair damage
- Example: use of check pointing and virtualization
- Continue to function correctly even if attack succeeds

APPROACHES TO SECURITY

1. Procedural

2. Functions and Mechanism

3. Assurance

1. **Procedural Approaches:** Prescribe appropriate behavior for a user interacting with the system

- Periods processing
- Guidelines for managing passwords
- Appropriate handling of removable storage devices
- Electronic voting systems
- Periods Processing: Split the day into periods and run different classification jobs in each period

2. **Functions and Mechanisms:** Enforce security policy

Examples are the 3As

- **Authentication:** assures that a particular user is who he/she claims to be
- **Access control:** a means of limiting a user's access to only those entities that the policy determines should be accessed
- **Audit:** a form of transaction record keeping. *The data collected is called an audit log*

3. Assurance

Assurance is a measure of how well the system meets its requirements –in other words, how much one can trust the system to do what it is supposed to do assurance is derived by analyzing the specification, design, and implementation of system.

Assurance Techniques

- ✓ Penetration analysis
- ✓ Covert channel analysis
- ✓ Formal verification

1. Penetration Analysis


Uses a collection of known flaws, generalizes the flaws, and tries to apply them to the system being analyzed


- Penetration team known as "Tiger Team"
- Demonstrates the presence not the absence of protection failures

2. Covert Channels

Covert channel – Uses entities not normally viewed as a data object to transfer information

Two Types of Covert Channels

 **Storage channels** – The sender alters the value of a data item and the receiver detects and interprets the altered value to receive information covertly

 **Timing channels** – The sender modulates the amount of time required for the receiver to perform a task or detect a change in an attribute, and the receiver interprets the delay or lack of delay to receive information covertly.

3. Formal verification: Is the act of proving or disproving the correctness of intended algorithms underlying a system with respect to a certain formal specification or property, using formal methods of mathematics.

PRIVACY

Privacy - ensures that only the information that an individual wishes to disclose is disclosed.

INTERNET PRIVACY

The ability to control what information one reveals about oneself over the Internet, and to control who can access that information.

Internet Privacy

These concerns include whether email can be stored or read by third parties without consent, or whether third parties can track the web sites someone has visited.

•Another concern is whether web sites which are visited collect, store, and possibly share personally identifiable information about users.

SOFTWARE SECURITY:

The following are general ways to help protect your computer against software potential security threats:

Firewall. A firewall can help to protect your computer by preventing hackers or malicious software from gaining access to it.

Virus protection. Antivirus software can help to protect your computer against viruses, worms and other security threats.

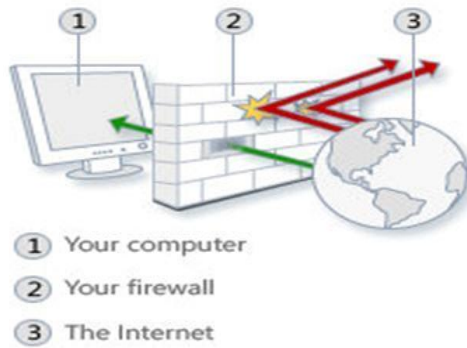
Spyware and other malware protection. Antispyware software can help to protect your computer from spyware and other potentially unwanted software.

Windows Update. Windows can routinely check for updates for your computer and install them automatically.

1. Use a firewall

A firewall is software or hardware that checks information coming from the Internet or a network and then either turns it away or allows it to pass through to your computer, depending on your firewall settings. In this way, a firewall can help prevent hackers and malicious software from gaining access to your computer.

Windows Firewall is built into Windows and is turned on automatically.



How a firewall works

If you run a program such as an instant messaging program or a multiplayer network game that needs to receive information from the Internet or a network, the firewall asks if you want to block or unblock (allow) the connection. If you choose to unblock the connection, Windows Firewall creates an exception so that the firewall won't bother you when that program needs to receive information in the future.

2. USE VIRUS PROTECTION/ ANTIVIRUS

Viruses, worms, and Trojan horses are programs created by hackers that use the Internet to infect vulnerable computers. Viruses and worms can replicate themselves from computer to computer, while Trojan horses enter a computer by hiding inside an apparently legitimate program, such as a screen saver. Destructive viruses, worms, and Trojan horses can erase information from your hard disk or completely disable your computer. Others don't cause direct damage, but worsen your computer's performance and stability.

Antivirus programs scan e-mail and other files on your computer for viruses, worms, and Trojan horses. If one is found, the antivirus program either **quarantines** (isolates) it or deletes it entirely before it damages your computer and files.

Windows does not have a built-in antivirus program, but your computer manufacturer might have installed one. If not, there are many antivirus programs available. **Microsoft** offers Microsoft Security Essentials, a free antivirus program you can download from the Microsoft Security Essentials website. You can also go to the **Windows 7 security software provider's** website to find a third-party antivirus program.

Because new viruses are identified every day, it's important to use an antivirus program with an automatic update capability. When the program is updated, it adds new viruses to its list of viruses to check for, helping to protect your computer from new attacks. If the list of

viruses is out of date, your computer is vulnerable to new threats. Updates usually require an annual subscription fee. Keep the subscription current to receive regular updates.

WARNING: IF YOU DON'T USE ANTIVIRUS SOFTWARE, YOU EXPOSE YOUR COMPUTER TO DAMAGE FROM MALICIOUS SOFTWARE. YOU ALSO RUN THE RISK OF SPREADING VIRUSES TO OTHER COMPUTERS.

3. USE SPYWARE PROTECTION

Spyware is software that can display advertisements, collect information about you, or change settings on your computer, generally without appropriately obtaining your consent. For example, spyware can install unwanted toolbars, links, or favorites in your web browser, change your default home page, or display pop-up ads frequently. Some spyware displays no symptoms that you can detect, but it secretly collects sensitive information, such as the websites you visit or the text you type. Most spyware is installed through free software that you download, but in some cases simply visiting a website results in a spyware infection.

To help protect your computer from spyware, use an anti spyware program. This version of Windows has a built-in anti spyware program called Windows Defender, which is turned on by default. Windows Defender alerts you when spyware tries to install itself on your computer. It also can scan your computer for existing spyware and then remove it.


Because new spyware appears every day, Windows Defender must be regularly updated to detect and guard against the latest spyware threats. Windows Defender is updated as needed whenever you update Windows. For the highest level of protection, set Windows to install updates automatically

4. UPDATE WINDOWS AUTOMATICALLY

Microsoft regularly offers important updates to Windows that can help protect your computer against new viruses and other security threats. To ensure that you receive these updates as quickly as possible, turn on automatic updating. That way, you don't have to worry that critical fixes for Windows might be missing from your computer.

Updates are downloaded behind the scenes when you're connected to the Internet. The updates are installed at 3:00 A.M. unless you specify a different time. If you turn off your computer before then, you can install updates before shutting down. Otherwise, Windows will install them the next time you start your computer.

5. TO TURN ON AUTOMATIC UPDATING

- A. Open Windows Update by clicking the **Start** button . In the search box, type **Update**, and then, in the list of results, click **Windows Update**.

- B. Click **Change settings**.
- C. Make sure **Install updates automatically (recommended)** is selected.
- D. Windows will install important updates for your computer as they become available. Important updates provide significant benefits, such as improved security and reliability.
- E. Under **Recommended updates**, make sure they **Give me recommended updates the same way I receive important updates** check box is selected, and then click **OK**.
- F. Recommended updates can address non-critical problems and help enhance your computing experience. 🌐 If you're prompted for an administrator password or confirmation, type the password or provide confirmation.

6. CREATE USER ACCOUNTS

When you log on to your computer, Windows grants you a certain level of rights and privileges depending on what kind of user account you have. There are three different types of user accounts: standard, administrator, and guest.

Although an administrator account provides complete control over a computer, using a standard account can help make your computer more secure. That way, if other people (or hackers) gain access to your computer while you're logged on, they can't tamper with the computer's security settings or change other user accounts. You can check your account type after you log on by doing the following:

4. Backup Your Data

To save yourself heartache you should regularly backup the data on the computer. A full system backup is recommended but with large hard drives that are available today this is not always practical. For large hard drives it is recommended that you at the least backup the files that you create (ie letters, documents, spreadsheets, accounting package data files, email etc). For steps on how to backup your computer please see our article on "Computer System Backups".

5. Defragment your Hard Drive

As you start to create and delete files and applications on your computer the hard drive will become fragmented. This means that the data is split into chunks and stored in different areas of the hard disk. The more fragmented your drive is, the less efficient your computer operates. Defragmentation consolidates the separate chunks, frees up disk space and speeds up your computer.

To perform this task, first open up **My Computer**, select the drive you wish to defragment and press the right mouse button. Select **Properties** and then the **Tools** tab, press the button to start the Defragmenter program.

What is Defragmentation?

As you start to create and delete files and applications on your computer the hard drive will become fragmented. This means that your data is split into chunks and stored in different areas of the hard disk. The more fragmented your drive is, the less efficient your computer operates. Defragmentation consolidates the separate chunks, frees up disk space and speeds up your computer. In simple terms imagine your hard drive as a bunch of little containers (with empty space being empty containers). An unfragmented hard drive would have all of the containers for programs or files next to each other like this:



Now what happens when you delete a program or file? An empty space will result and your hard drive

will look like this:



Next time a file is written to the drive it will be placed in the first empty space. If the file is larger than the empty space it will be split up with part being written to the empty space and the next part be written to the next free space and so on.



Eventually files will be split up into multiple sections, causing files to be located in various containers throughout the entire drive. The result is a hard drive looking something like this:



To defragment is to take all of the fragmented pieces and put them back in order. Windows comes with a defragmenting program. You can also get third party software such as Norton Utilities which comes with a defragmenting program.

How To Defrag Your Hard drive?

The following steps list how to defragment your hard drive. You should defragment your hard drive every few months, if you often add and/or remove programs and files you should defragment more often.

Note:

1. You should stop using your computer while you are defragmenting the drive. It is possible to continue to use your computer but your computer will operate slower and the defragmentation process restarts if the contents of the hard drive changes.
2. You should stop all programs running including your screen saver.
3. The defragmentation program can take a long time to run. It is recommended that you run this program overnight or when you do not need to use the computer for a few hours.

Windows 95/98/98SE/Millennium

To start the Disk Defragmenter program in MS Windows 95/98/98SE or Me, use the following steps:

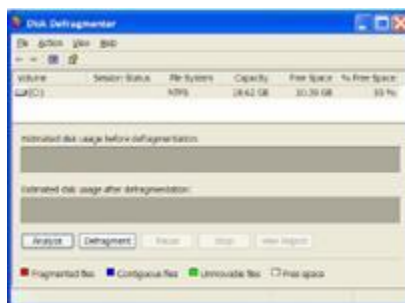
1. Click on the **Start button**, then move to the **Programs**, select **Accessories**, then **System Tools**, and then click on the **Disk Defragmenter**.

2. Select the drive you wish to defragment (usually Drive c), click **OK**, and then click **Yes**.
Wait for the defragmentation tool to finish (this may take a long time).



Windows 2000/XP

1. To open Disk Defragmenter, click **Start**, point to **All Programs**, point to **Accessories** then **System Tools**, and click **Disk Defragmenter**.
2. Click the drive you want to defragment, click the **Defragment** button and then wait for the defragmentation tool to finish (this may take a long time).
3. If you have any problems running the derangement you may need to be logged on as an administrator or a member of the Administrators group.



Windows NT

Windows NT does not have a built in defragmentation tool. Third party defragmentation tools are available.

6. Hard Disk Error Checking

As all your data is kept on the computers hard drive, it is essential that you regular check the drive for errors. To perform this task, first open up **My Computer**, select the drive you wish to check and press the right mouse button. Select **Properties** and then the **Tools** tab, press the button to start the Error Checking program.

7. Virus Checker

With the amount of computer viruses around these days it is absolutely essential that you have a virus checker (such as Norton Antivirus) installed on your machine. Having a virus checker installed on your system is not enough though as it needs to be kept up to date and your computer regularly scanned for viruses. As the various virus checking programs work in a variety of different ways consult the software manual or get a expert to assist you with updating the program and performing regular disk scans.

8. Uninstall Software

As you start to use the computer more and more you will begin to install new programs and applications. A lot of times you will use the program a few times and then either forget about it or realise it doesn't do what you wanted. The application will then just stay on your machine taking up space and using up resources. It is recommended to regularly check the software installed on your machine and uninstall and programs you no longer use. To uninstall a program in Windows 9x, 2000 & XP first open up your **My Computer** icon and then open up the **Control Panel** icon. An **Add or Remove** program icon will then be visible, start this program and you will be presented with a list of programs installed on your machine. Scroll through this list and remove any programs you no longer use. If you are unsure of a program leave it and ask a computer professional for advice.

9. Clean Out Your Recycle Bin

When you delete a file or email it doesn't necessarily delete the file from your hard drive, instead it places it in the "Recycle Bin" or "Deleted Items" folder for email in Outlook or Outlook Express. This is done so that you can easily recover files or emails you may have accidentally deleted. Over time though the Recycle Bin and Deleted Items folder can become large and the contents need to be purged.

Emptying the Recycle Bin

First check the recycle bin and to see if it doesn't contain any required files. To see what is in the recycle bin, double click on it to open it up and scroll through the list of files to see what it contains. If you are certain that you do not need any of the files select **File** and then **Empty Recycle Bin**. *If you are unsure of the files that are in the recycle bin seek assistance before emptying the bin.*

Emptying Your Deleted Email

First check the "Deleted Items" folder to see if it doesn't contain any required emails. To see what the folder contains start Outlook or Outlook Express and click on the "Deleted Items" folder and scroll through the list of emails to see what it contains. If you are certain that you do not need any of the emails, right click over the top of the "Deleted Items" folder and select Empty Deleted Items. *If you are unsure of the emails that are in the "Deleted Items Folder" seek assistance before emptying the folder.*

10. Operating System Reinstall

Over time you will install and uninstall various software, apply software and hardware patches, add and remove various bits and pieces of hardware and update system drivers. This will cause your computer system to gather old software programs, drivers and other system files that all contribute to your system running slow, behaving erratically and / or crashing. To remedy this situation contact a professional who can reload your operating system, applications and reset your computer software back to how it was when you first bought your machine.

General Computing Tips

How to Shutdown Your Computer Properly

Always shutdown your computer and any applications you have open properly. Only use the reset button if your computer locks up and you have no other choice. To shut down to **Start-->Shutdown-->** and choose the option you need; **Reset, Shutdown, or Log Off**.

The Golden Rule: When all else fails, REBOOT.

If your computer is not responding to the above an application has gone awry and you will need to close off the offending application. To do this press **Ctrl+Alt+Del** (all at the same time). A **Close Program** or **Task Manager** dialog box will appear. Select the task you want to

end (or the ones that say they are "not responding") and select **End Task**. If this doesn't work you can reboot the machine

in MS Window 9x by simply pressing Ctrl+Alt+Del twice in succession.

Unknown Emails

Be very, very cautious of emails especially those that contain attachments from unknown senders. Even if you know the sender and you are not expecting an email from them or are unsure what the attachment is, be careful. Use an antivirus program, keep it updated and set it to automatically scan your incoming and outgoing email. If you are unsure get a professional to help you.

Unknown File Downloads

Don't go around and haphazardly download every program you can. Be sure you know what you are downloading and from where. Not all downloads are bad, but you never know what may show up and some programs may contain a virus or a Trojan horse. Know exactly what you're downloading and installing to your hard drive beforehand. If you are unsure stop and get a professional to help you.

Incompatible Hardware and Software

Before purchasing new software or hardware be sure that the product you are buying is compatible with your system. If in doubt write down the specifications of your machine and ask the salesman.

Random Deletion of Files

Do not delete files or applications when you are not sure of what they belong to. You may delete that one file that runs your favorite software or delete that important finance information. If you are unsure, leave the file alone and ask for assistance.

Windows Update

Windows Update is an online extension to MS Windows 98, Me, 2000 and XP (Windows Update is not available in MS Windows 95) that helps you to keep your computer up-to-date. By using Windows Update you can choose the updates to install for your computer's operating system, software, and hardware. As new content is added regularly to the site you can always download the most recent updates and fixes to protect your computer and keep it running smoothly. Updates that are critical to the operation of your computer are

considered a "Critical Update," and are automatically selected for installation during the scan. These updates are available to help resolve known issues and to protect your computer from known security vulnerabilities. Critical updates should always be downloaded.

Starting Windows Update

MS Windows XP / 2000

In Windows XP Home Edition, you must be logged on as a computer administrator to install components or modify Automatic Updates settings. In Windows XP Professional, you must be logged on as an administrator or a member of the Administrators group. If your computer is connected to a network, network policy settings might also prevent you from completing this procedure.

- To start Windows update first connect to the Internet and then open the website:

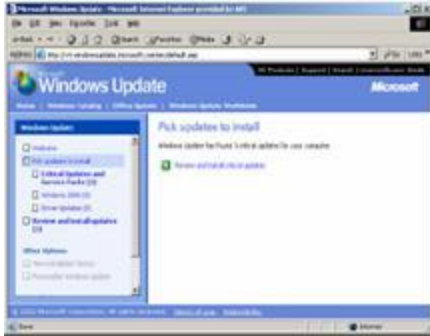
<http://windowsupdate.microsoft.com/>

in Internet Explorer (Note: You need Microsoft Internet Explorer v5 or higher, Netscape or other browser do not work).

- Once the Windows Update page has loaded, click **Scan for updates**, your system will then be scanned (this may take a while). Click **Yes** when prompted by any "Security Warnings" to install any required software or device drivers.



When the scan has completed any Critical Updates will be selected, press Review and Install Critical Updates to download and install them. Otherwise scroll through the list on the left hand side and select any Windows XP / 2000 and/or Driver Updates you wish to install.



MS Windows 98 / Me

First connect to the Internet and then open the website:<http://windowsupdate.microsoft.com/> in Internet Explorer (Note: You need Microsoft Internet Explorer v5 or higher, Netscape or other browser do not work). Once the Windows Update page has loaded, click **Product Updates**, your system will then be scanned (this may take a while, if any Security Warning appear click Yes). When the scan has finished, if there are any critical updates available click **Download** otherwise scroll through the list and select the updates you to download.



Computer Viruses

What is a virus?

A computer virus is a computer program designed to replicate and spread on its own, preferably without a user's knowledge. Computer viruses spread by attaching themselves to things such as a computer program, word processing or spreadsheet document, disks or email. Email is currently the most prolific way viruses spread. When an infected file is executed the virus starts. Depending upon the type of virus it can stay dormant, waiting to for an event to happen (such as a specific date) or become active straight away. When the virus becomes active it can do a number of things such as:

- Delete and / or rename files on your computer.
- Send an email to one person or many people.

The email can contain a predefined message, a copy of one of your emails or even a file on your computer

- Cause your computer not to even switch on or work (by modifying or erasing your BIOS).
- Display a certain message.
- Allow people to hack into your computer.

Computer viruses are increasing at an unprecedented rate. In 1986, there was one known computer virus; three years later, that number had increased to six and by 1990, the total had jumped to 80. By November of that year, viruses were being discovered at the rate of one per week. Today, between 10 and 15 new viruses appear every day. In fact, from December 1998 to October 1999, the total virus count jumped from 20,500 to 42,000. (Update: In March 2001 the antivirus software product, Norton Antivirus protected against 48,755 viruses as of October 2002 this figure jumped to 62,181.)

How do viruses spread?

Computer viruses spread whenever infected diskettes or files are exchanged. This can happen by people exchanging computer disks or viewing email attachments. People often do not know that the file they are sending you is infected so it pays to be wary of any program and email you open.

Why do people write viruses?

There are many reasons why people write computer viruses, some reasons are:

- The thrill of it. Just like an arsonist or a vandal, people want to see what kind of destruction they can create and if they can get away with it.
- To show off to people and prove to their peers how smart they are.
- To exploit a weakness in a product.

You do not have to be particularly smart to write a virus as there are many “virus writing kits”

available on the Internet. Using these kits a person can easily change one or more features and

send their creation out onto the Internet via email or spread the virus via other means.

How can you protect yourself from viruses?

- 1) Have an Antivirus program installed on your computer
- 2) Regularly scan the computer for viruses (at least once a week).
- 3) Keep your virus definition files (ie the list of viruses the antivirus knows about and can fix) up to date (at least once a month, more times if possible).
- 4) Keep updated in regards to the latest security patches for your operating systems and applications.
- 5) Do not open email attachments that are from people you don't know especially ones that end in ".vbs", ".exe", ".pif", ".bat" or ".scr". Even be wary with email from people you do know and trust. If you are not expecting an email from the person and it contains an attachment contact the person before you open the file.

ONLINE virus scan:

Trend micro

http://housecall.trendmicro.com/housecall/start_corp.asp

Computer Association

<http://www3.ca.com/securityadvisor/virusinfo/scan.aspx>

F Secure

<http://support.f-secure.com/enu/home/ols.shtml>

Downloadable – For DOS

http://www.f-prot.com/download/corporate/download_fpdos.html

Web Site Blocker

We-Blocker - http://weblocker.fameleads.com/get_weblocker.asp

Spy Ware, AdWare Scanner

Ad-aware

<http://www.lavasoft.de/support/download/>

Port Scanner

Sygate

<http://scan.sygate.com/>

How do I utilize F DISK?

The program that DOS supplies for setting up hard disk partitions is called F DISK, which is believed to stand for "fixed disk", an older term for hard disk. F DISK is used only for DOS (FAT) partitioning, and allows the user to perform the following functions:

Create Partitions:

FDISK allows you to create a primary DOS partition or logical DOS volumes. To create a logical DOS volume you must first create an extended DOS partition, since the logical are contained within the extended partition.

Set Active Partition:?

You can use FDISK to set the primary partition on your boot disk active, allowing it to boot. It's strange that FDISK doesn't do this automatically when you create the boot primary partition (since there can only be one primary DOS partition anyway), but you must do this manually in many cases. (At least FDISK warns you when no disk is set active, via a message at the bottom of the screen.)

Delete Partitions:?

FDISK will let you delete partitions as well. This is the only way to change the size of a partition in FDISK. You have to delete the old one and create a new one with the new size. If you want to change the size of the primary DOS partition using FDISK you must delete every FAT partition on the disk and start over... annoying, but necessary

Display Partition Information:?

The last option that FDISK gives is to display the partition information for the system. It will first show the primary and extended partitions and then ask you if you want to see the logical drives within the extended partition. In fact, if you want to see this information, you can just do "FDISK /STATUS" from the DOS command line. This will show you the partition information without taking you into FDISK, and therefore, you run no risk of accidentally doing something? you'll wish you hadn't.? Which in reality is always a good thing!

Some important points that you should keep in mind when using FDISK:

Be very Careful:

1. With just a few keystrokes, FDISK can wipe out part or all of your hard disk. Generally speaking, don't use FDISK unless you need to, and make sure you understand what you are doing before you begin.
2. Run It From DOS: Windows 95 allows you to run FDISK direct from the graphical user interface, and even while other applications are open and running. Since FDISK alters critical disk structures at a very low level, running it while files are open and other applications are using the disk is asking for trouble. To be safe, always exit to DOS ("Restart the computer in MS-DOS mode") before using FDISK (except for using "FDISK /STATUS", will work safely from within a DOS box in Windows 95/98, remember, you're not changing anything). FAT32 Support: The version of FDISK that comes with Windows 95 OEM SR2 supports the creation of partitions that use the FAT32 enhanced file system for larger volumes. Some clever person at Microsoft decided not to call it FAT32 however within this program. Instead, when you run FDISK on a system that has Windows 95 OEM SR2 installed, and a hard disk over 512 MB (the minimum for using FAT32), you will receive a message asking you if you want to "enable large disk support". If you answer "Y" then any new partitions created in that session will be FAT32 partitions. Note: It is often useful to include FDISK as one of the programs on a bootable floppy. This way you can use it when setting up new hard disks. Considering how important it is, FDISK is a rather primitive program. It works, but it's cryptic and hard to use. Anything you can do in FDISK you can do more flexibly and easily using a third-party program like Partition Magic. FDISK will not allow you to select or change cluster sizes resize partitions, move partitions, etc. Disk's primary advantage is, of course, that it is free (well, built-in anyway).

Windows NT uses a program called Disk Administrator to handle disk setup tasks. In essence, this is an enhanced version of FDISK that allows you not only to manipulate

partitions, but also access some of NT's unique disk management features like NTFS. I added this because not everyone uses the same operating systems. And things are handled slightly different, but if they didn't, everyone would just be using the same thing. That would just be boring.

Before proceeding with installing an operating system (DOS/Windows 9x), the drive must first be partitioned and formatted. A drive/partition will not be assigned a drive letter and can not be formatted until partitioned using FDISK or a similar utility. If the new drive is a second drive, partitioning and formatting can be done within Windows. Go to the DOS Prompt and run FDISK to set up partitions. A second drive can be formatted within Windows by right clicking the drive in My Computer or Windows Explorer and selecting Format. If the new drive is to be the boot drive, then a startup diskette will need to be used (Windows Me requires the distribution CD to build a bootable hard drive). The startup diskette should be included with the operating system or one can be made from within Windows. A Startup disk can be made by going to Start | Settings | Control Panel | Add/Remove Programs | Startup Disk tab. Place the startup diskette into the A: drive and reboot the computer.

FDISK

After configuring the new drive as master or slave and configuring system BIOS for the new drive, boot the system with the startup disk in drive A: At the prompt, type FDISK and .

After the FDISK utility starts, the first screen (if using FAT32 version) should ask if you want to use Large Drive support. If your drive is over 512 MB and you want to make partitions over 2 GB, answer Yes. The next screen should be a menu with either 4 or 5 numbered selections.

FDISK Options

Current Fixed Disk Drive: 1

Choose one of the following options

1. Create DOS Partition or Logical DOS drive
2. Set Active Partition
3. Delete partition or Logical DOS drive
4. Display partition information

5. Change current fixed disk drive (Only if more than one drive is present)

Enter Choice []

Option 1 is used to create a Primary DOS partition or an Extended partition and set the partition(s) size. Use Option 2 to set the boot partition as Active (only one partition can be set Active using FDISK). Option 3 is used to delete partitions (Primary DOS, Extended, Non-DOS). Option 4 displays all settings for each partition. Option 5 lets you select which Fixed Disk to partition (maximum of 4 fixed disks with FDISK).

After all configurations are made, use the ESC key to exit the program. Before the drive can be Formatted, the computer will need to be re-booted in order for the new partitions to be given a Drive Letter.

Steps for partitioning a hard drive using FDISK

1. If you're in Windows, open a DOS window.
2. From the C:\ prompt, type "FDISK" and press ENTER.
3. If you're changing the partitions on an existing disk, choose option 4 from the FDISK menu to display existing partition information.
4. If all of the space on the drive is already partitioned, you will need to use FDISK menu option 3 to remove existing partitions before creating new ones.
5. For a new drive from which you will boot your PC, you must first create a Primary DOS Partition. Choose option 1 from the FDISK main menu and Select option 1 from the Create menu to create a Primary DOS Partition.
6. If you only want to have one partition on the drive, type Y when prompted to make one large partition. If making multiple partitions, type N.
7. Enter the size for the partition if you selected N in step 6.
8. To create an extended (non-bootable) DOS partition, choose option 1 from the main FDISK menu, then option 2 from the Create menu.
9. Press enter to use all remaining available space for the partition.

10. Create logical drives on the extended partition by entering the desired size(s) in MB or percent of disk space.
11. Continue until all available space is assigned to logical drives.
12. If you will be booting from this disk, choose option 2 from the FDISK menu and enter 1 to make the primary DOS partition ACTIVE.
13. Press the ESC key to exit FDISK. If running from Windows 9x, you must manually reboot your PC at this point. You must format all partitions before they can be used.

Tips

1. If you are using a 16 bit OS (Windows 3.x or DOS), do not use the maximum available size for your Primary DOS Partition on a drive larger than 2GB or you will be unable to use the rest of the space.
2. Before repartitioning an existing drive, be sure to make a good backup of all of the data on the drive. FDISK will destroy all existing data on the drive.
3. There are utilities (e.g., Partition Magic) that can re-partition existing drives without destroying data or minimizing data loss.

No Fixed Disk present

This means FDISK is unable to find your hard disk drive. Insure that the hard drive is properly setup in CMOS. If the drive is setup properly in CMOS, double check all cable connections on the hard disk drive and the System board. Also make sure all ATA devices are configured as Master or slave.

Drive Lettering

Drive letter assignment is dynamic, meaning DOS/Windows hands out drive letters in sequence each time the computer boots. Booting to a DOS/Windows floppy and running FDISK /STATUS is a quick way to see how the operating system has assigned hard drive letters. DOS/Windows (FAT File Structure) is limited to four Primary DOS partitions and can only assign one Primary DOS partition per physical hard drive. Using Extended Partitions and Logical Drives, you are only limited by the alphabet for Drive Letters.

Every device in your computer has a priority. Drive letters A: and B: are reserved for the

Floppy drives. Hard Drives begin with Drive letter C:. Primary DOS partitions have priority over Extended DOS partitions. Therefore, the bootable floppy drive is A:, the non-bootable floppy drive (if present) is B:, the first Active (bootable) primary partition on the first hard drive is C:. Any logical drives or Extended DOS partitions would become D:, E:, F:, etc. (Some networking software reserves Drive Letters starting with F:).

If a second hard drive is in the system, the second Primary DOS partition would be D: and then logical drives or Extended DOS partitions on the first drive would then become E:, F:, G:, respectively (Primary DOS has priority over Extended DOS) followed by any logical drives or Extended DOS partitions on the second drive.

After four Primary DOS partitions have been used, the next priority is for devices loaded by an external BIOS such as SCSI devices. The last priority is for devices controlled by "block mode" drivers loaded from the CONFIG.SYN and/or AUTOEXEC.BAT; most CD-ROM fall in this category and their drive letters are assigned when the MSCDEX.EXE loads. RAM drives, Parallel Port drives, and Double Spaced or Stacked drives also fall into this category. If the CD-ROM device driver is loaded first, it gets the next available drive letter.

When you add a new device, it will get the next available letter following the above mentioned priority assignment and displace any device using that letter. The displaced device will pick up the next available drive letter, and all associated drivers for the displaced device may need to be reconfigured, a common occurrence when adding a second hard drive to a system with a CD-ROM.

FORMAT

After a drive is partitioned with FDISK, each partition must be Formatted to make it useable by the Operating System. After re-booting to the A:> prompt, type FORMAT x: (x= drive letter) and . If this is to be the boot drive, use FORMAT C: /S and to format the drive and make it bootable by transferring the system files (Windows Me requires the distribution CD to make a drive bootable). The FORMAT command will ask a couple of times if you are sure you want to continue with the operation with the warning that Formatting will destroy all data on the drive. If sure, select Yes. The Format command should start the process and show a percent of drive formatted.

If this is a second drive and you are using Windows 95/98, double click the My Computer icon on your Windows desktop. Right click on the first partition of your new drive and choose Format. If this will become the new boot disk, be sure to check the box at the bottom that says "Copy system files." Choose full format. Give the drive a label of up to 11 letters and/or numbers if you like. Click the Start button to begin formatting. You must

repeat these steps for each partition you created on the new drive.

SYS

This command can be used to transfer a fresh copy of system files to a drive that has been partitioned and Formatted (Windows Me will not allow use of SYS command to transfer the system). This will not destroy any data on the drive, but will simply replace the current system files with the ones from the source disk. This is a good way to refresh a drive that has data on it but will no longer boot. Use: SYS x: (x=Drive letter to be refreshed).

An alternate method of partitioning and formatting is included with the DiscWizard/DiscWizard Starter Edition utility.

FORMAT

Type: External (1.0 and later)

Syntax:

FORMAT d:[/1][/4][/8][/F:(size)] [/N:(sectors)] [/T:(tracks)] [/B|/S][/C]
[/V:(label)] [/Q][/U][/V]

Purpose: Formats a disk to accept DOS files.

DISCUSSION

Formats the disk in the specified drive to accept DOS files, analyzing the entire disk for defects.

Initializes the directory and file allocation tables. Can be used to format both diskettes and fixed disks

NOTE:

In some earlier versions of DOS, the drive designation letter was optional. If you are using one of these versions, you can format a diskette or a FIXED DISK if you enter FORMAT while working in that drive.

For more information about the FORMAT command, see Chapter 2, **Using Common DOS Commands**, in the downloadable book **DOS the Easy Way.**

/1 - Format for single-sided use, regardless of the drive type.

/4 - Formats a double-density diskette in a high-density type disk drive. Files written to a double-sided disk using a high-density drive may not be reliable.

/8 - Formats at 8 sectors per track. If /8 is not specified, FORMAT defaults to 9 or 15 sectors per track, depending upon the disk drive type. The /V option cannot be used with the /8 option.

/F:(size) - Formats disks to specific sizes. You can specify the target disk to be a size value from 160Kb to 2.88Mb. Do not format a floppy disk at a size higher than it was designed for.

/N:(sectors) - Specifies the number of sectors per track on the disk. Used to format a 3.5 inch disk with the number of sectors per track specified using this option. For 720 K-byte disks, this value should be entered as ³N:9.²

/T:(tracks) - Specifies the number of tracks on the disk. Used to format a 3.5 inch disk with the number of tracks specified using this option. For both 720 K-byte disks and 1.44 K-byte disks, this value should be entered as T:80.

/B - Formats a disk reserving room on the disk to later copy the DOS system files.

/S - Copies the operating system files to the disk after formatting. These system files are hidden files and will not appear in a directory listing. Using some versions of DOS, this must be the last option entered.

/C - Causes FORMAT to retest badclusters, otherwise FORMAT will mark the clusters as bad but will not retest them. (In DOS versions before Version 6, FORMAT always retested any bad clusters.)

/V:(label) - Causes FORMAT to prompt for a volume label after the disk is formatted. The label can be of 1 to 11 characters. The same characters acceptable in filenames are acceptable in the volume label (however, you cannot add a file name extension). The /8 option cannot be used with the /V option. DOS Version 5 automatically assigns Label as the disk label and creates a unique serial number in the boot sector of the disk. The serial number is displayed at the end of the formatting process.

/Q - Provides a quick way to format a disk . This option erases the file allocation table and the root directory, but does not identify bad sectors.

/U - Completely erases all data on the target disk making it impossible to perform an UNFORMAT later.

/V - Displays a prompt so that a volume label can be entered.

EXAMPLE

If you want to format a floppy disk as a double-density disk in a 1.2M drive, you should enter the following:

format a:/4

SOFTWARE PRIVACY AND COPYRIGHT

INTRODUCTION

- 💡 Because of human nature, computers systems may be used for both good and bad purposes.
- 💡 Some questions to look at include:
 - 💡 *What are the consequences of the widespread use of computing technology?*
 - 💡 *Does technology make it easy for others to invade our personal privacy?*
 - 💡 *Does technology make it easy for other to invade the security of business organizations like our banks or our employees?*
- 💡 Competent end users need to be aware of the potential impact of technology on people and how to protect themselves on the Web.
- 💡 They need to be sensitive to and knowledgeable about personal privacy, organizational security, ergonomics, and the environmental impact of technology.

PEOPLE

- 💡 People are one key component of information systems, the others including procedures, software, hardware, and data.
- 💡 Most everyone would agree that technology has had a very positive impact on people, but there are some negative impacts as well.
- 💡 Some of these negative concerns include:

PRIVACY

- 💡 What are the threats to personal privacy and how can we protect ourselves

SECURITY

- 💡 How can access to sensitive information be controlled and how can we secure hardware and software?

ERGONOMICS

- 💡 What are the physical and mental risks of using technology, and how can these risks be minimized?

ENVIRONMENT

- 💡 What can individuals and organizations do to minimize the impact of technology on our environment?

PRIVACY

- 💡 Ethics are standards of moral conduct
- 💡 Computer ethics are guidelines for the morally acceptable use of computers in our society
- 💡 Four primary computer ethic issues include:

PRIVACY

- 💡 Concerns the collection and use of data about individuals

ACCURACY

- 💡 Relates to the responsibility of those who collect data to ensure that the data is correct

PROPERTY

- 💡 Relates to who owns data and rights to software

ACCESS

- 💡 Relates to the responsibility of those who have data to control and who is able to use that data

LARGE DATABASES

- 💡 Every day, data is gathered about us and stored in large databases

- 📁 The federal government alone has over 2,000 databases
- 📁 Our Social Security number has become a national identification number
- 📁 Information resellers (aka information brokers) make up an entire industry that collects and sells personal data.
- 📁 Electronic profiles are built containing highly detailed and personalize descriptions of individuals
- 📁 Some concerns involve the possibility of:

IDENTITY THEFT

- 📁 The illegal assumption of someone's identity for the purposes of economic gain

MISTAKEN IDENTITY

- 📁 The electronic profile of one person can be switched with another.
- 📁 The Freedom of Information Act entitles you to look at information kept by credit bureaus and government agencies

PRIVATE NETWORKS

- 📁 Snoop ware is software that allows organizations to search electronic mail and files
- 📁 One survey found nearly 75% of all businesses have done this
- 📁 Some argue that these are private networks, and the owners can do what they want
- 📁 Others argue that the U.S. is becoming a nation linked by electronic mail, therefore the government has to provide protection for users against other people reading or censoring messages

THE INTERNET AND THE WEB

- 📁 Illusion of anonymity is the problem when people believe they are safe from others invading their personal privacy
- 📁 Many organizations monitor the email sent and received on their servers

- 📁 Some individuals “eavesdrop” on e-mail sent over the web
- 📁 Your use of the Web may be monitored, including the creation of a “history file” including a list of all the sites you’ve visited
- 📁 Cookies are special files that capture information about the web sites that you visit. Two basic types include:

TRADITIONAL COOKIES

- 📁 Monitor your activities at a single site
- 📁 Often used to provide customer service

AD NETWORK (AKA ADWARE) COOKIES

- 📁 Monitor activities across all sites you visit
- 📁 Examples include Double Click and Avenue A
- 📁 Programs called “cookie cutter programs” help to filter out these “bad” cookies
- 📁 Spyware is software that are designed to secretly record and report on an individual’s activities on the Internet. Adware is just one type of spyware

MAJOR LAWS ON PRIVACY

- 📁 Most information collected by non-governmental organizations is NOT covered by existing laws
- 📁 The Code of Fair Information Practice has been established to encourage organizations to follow its recommended practices

SECURITY

COMPUTER CRIMINALS

- 📁 Computer crime is an illegal action which the perpetrator uses special knowledge of computer technology.
- 📁 Five types of computer criminals include:

EMPLOYEES

- 📖 The largest category of computer criminals

OUTSIDE USERS

- 📖 Criminal Suppliers and clients that have access to an organization's computers

HACKERS AND CRACKERS

- 📖 Hackers are people who gain unauthorized access to a computer for the fun and challenge of it
- 📖 Crackers do the same for malicious reasons
- 📖 A "bomb" is a destructive computer program put into a system

ORGANIZED CRIME

- 📖 Use computers just like legitimate businesses, only for illegal purposes
- 📖 Counterfeiters and forgers use computer technology

TERRORISTS

- 📖 Knowledgeable terrorist groups can disrupt computer and communication systems

COMPUTER CRIME

- 📖 FBI estimates computer crime losses at over \$1.5 trillion

MALICIOUS PROGRAMS

VIRUSES

- 📖 Programs that migrate through networks and operating systems most attach themselves to programs and databases
- 📖 Computer Abuse Amendments Act of 1994

WORMS

- 📖 Special type of virus that doesn't attach itself to programs or databases, but fills a system with self replicating information

TROJAN HORSES

- 📄 Programs that come into a system disguised as something else

DENIAL OF SERVICE

- 📄 Like a worm, it attempts to slow down a system
- 📄 DOS attacks flood a system with requests for information or data, typically via the Internet

INTERNET SCAMS

- 📄 A scam is a fraudulent or deceptive act or operation designed to trick an individual into spending their time or money for little or no return
- 📄 An Internet scam is a scam using the Internet

THEFT

- 📄 Can take many forms, including theft of hardware, software, data or computer time
- 📄 Software piracy is the unauthorized copying of programs for personal gain

The Software Copyright Act of 1980 says it IS LEGAL to make a backup copy of software, however, these copies may NOT be sold or given away

DATA MANIPULATION

Finding entry into a system and leaving a message may seem like a prank, but it IS AGAINST the law

Computer Fraud and Abuse Act of 1986 makes it a crime for unauthorized persons to view, copy, or damage data using any computer across state lines.

It also prohibits the unauthorized use of any federal (or federally insured financial institution's) computer.

OTHER HAZARDS

NATURAL HAZARDS

- 📖 Fires, floods, wind, hurricanes, tornadoes, earthquakes all require remote backup and redundancy plans

CIVIL STRIFE AND TERRORISM

- 📖 Wars, riots, and terrorist activities are real risks

TECHNOLOGICAL FAILURE

- 📖 Hardware and software will fail, so you need plans for this contingency
- 📖 Surge protectors can help protect against voltage surges (aka spikes)

HUMAN ERRORS

- 📖 Put in validation plans to help reduce the number of errors

MEASURES TO PROTECT COMPUTER SECURITY

- 📖 Security is concerned with protecting information, hardware, and software which much be protected from both man made and natural disasters.
- 📖 Some techniques used to protect computer systems include:

ENCRYPTING MESSAGES

- 📖 Encoding messages and data so it can not be read by someone without the decoding scheme

RESTRICTING ACCESS

- 📖 Keep unauthorized people away from systems by using some things as:
- 📖 Biometric scanning devices such as fingerprint or retinal (eye) scanners
- 📖 Passwords – secret words or codes that must be entered to access the system

- 📄 Firewalls – hardware and software that acts as a security buffer between the corporation's private network and all external networks, including the Internet

ANTICIPATING DISASTERS

- 📄 Physical security: protecting hardware from human and natural disasters
- 📄 Data security: protecting software and information from unauthorized tampering or damage
- 📄 Disaster Recovery Plans: contingencies for continuing operations during an emergency until normal operations can be restored

Hot sites: fully equipped backup computer centers

- 📄 Cold sites: building with hook ups, but no equipment

BACKING UP DATA

- 📄 Equipment can usually be replaced, but data may be irreplaceable
- 📄 Data should be encrypted if sent over networks, restricted from unauthorized viewing and modification, and backed up at a remote location

ERGONOMICS

- 📄 Ergonomics is the study of human factors related to things people use
- 📄 It is concerned with fitting the job to the working rather than forcing the worker to contort to fit the job

PHYSICAL HEALTH

- 📄 Sitting too long and working with computers can lead to:

EYESTRAIN AND HEADACHE

- 📄 Take a 15 minute break every hour or two
- 📄 Avoid computer screens that flicker
- 📄 Screen should be 3-4 times brighter than the background light

BACK AND NECK PAIN

- 📖 Make sure tables and chairs are adjustable
- 📖 Monitor should be at or slightly below eye level
- 📖 Keyboards should be detachable
- 📖 Use a footrest to avoid leg strain

REPETITIVE STRAIN INJURY

- 📖 RSI (aka cumulative trauma disorder) is a name given to a number of injuries
- 📖 Carpal tunnel syndrome consists of damage to nerves and tendons in the hands
- 📖 May use ergonomically designed keyboards, take breaks and rests from working on computers

MENTAL HEALTH

NOISE

- 📖 Women have been found to be more sensitive to noisy conditions, especially high pitched equipment noises
- 📖 Use head mounted microphones and earphones, as well as room soundproofing to reduce noise

ELECTRONIC MONITORING

- 📖 Research shows that people suffer more from electronic surveillance than from human
- 📖 Fed Ex and Bell Canada removed some electronic surveillance and found that productivity went up

Techno stress is the tension that arises when we have to unnaturally adapt to computers rather than having computers adapt to us

DESIGN

- 📖 “less may be more” when it comes to computer design – some people prefer fewer features if the system is easier to use.

THE ENVIRONMENT

- 📄 Microcomputers use 5% of the electricity used in the workplace
- 📄 The Environmental Protection Agency (EPA) has created the Energy Star program to discourage waste in the microcomputer industry.
- 📄 The microcomputer industry has responded with the Green PC to address the reduction of power consumption by computers

THE GREEN PC

SYSTEM UNIT

- 📄 Use an energy saving microprocessor
- 📄 Employ microprocessor and hard drives that switch to sleep mode when not in operation
- 📄 Replace supply unit with an adapter that uses less electricity
- 📄 Eliminate the cooling fan

DISPLAY

- 📄 Use flat panel displays (which use less electricity than CRT)
- 📄 Use “power-down” monitors
- 📄 Use screen saver software that clears the display

MANUFACTURING

- 📄 Reduce the amounts of chlorofluorocarbons (CFCs) in the manufacturing process
- 📄 Other toxic chemicals and metals (nickel, other heavy metals) are removed from the manufacturing process

PERSONAL RESPONSIBILITY

CONSERVE

- 📄 EPA estimates 30-40% of computers are left on all the time
- 📄 EPA estimates that 80% of the time no one is looking at a monitor
- 📄 You can use a screen saver to help save energy

RECYCLE

- 🗑️ You can recycle paper, ink cartridges, packaging materials, as well as computer components yourself

EDUCATE

- 🗑️ You can learn more on how to recycle, and encourage others to do the same

A LOOK TO THE FUTURE

Presence Technology Makes Finding People Easy.

- Researchers are developing technology to alert others when you are doing something, such as watching TV, in your car, etc.
- Advantage is people would know the “best” way to reach you
- Disadvantage is people would always know what you are doing.

VISUAL SUMMARY AT A GLANCE – PRIVACY AND SECURITY

PRIVACY

LARGE DATABASES

PRIVATE NETWORKS

PRIVACY

INTERNET AND THE WEB

TRADITIONAL COOKIES

AD NETWORK (ADWARE) COOKIES

MAJOR PRIVACY LAWS

SECURITY

THREATS TO COMPUTER SECURITY

COMPUTER CRIME

MALICIOUS PROGRAMS

DENIAL OF SERVICE (DOS) ATTACKS

INTERNET SCAMS

THEFT

DATA MANIPULATION

SECURITY

MEASURES TO PROTECT COMPUTER SECURITY

ENCRYPTING

RESTRICTING ACCESS

ANTICIPATING DISASTERS

BACKING UP DATA

ERGONOMICS

PHYSICAL HEALTH

EYESTRAIN AND HEADACHE

BACK AND NECK PAIN

REPETITIVE STRAIN INJURY (RSI)

MENTAL HEALTH

NOISE

STRESS/TECHNOSTRESS

DESIGN

THE ENVIRONMENT

THE GREEN PC

SYSTEM UNITS

DISPLAY UNITS

MANUFACTURING PROCESS

PERSONAL RESPONSIBILITY

CONSERVING ENERGY

RECYCLING

EDUCATING

OPEN-ENDED

DISCUSS THE RELATIONSHIP BETWEEN DATABASES AND PRIVACY

- 📖 Every day, data is gathered about us and stored in large databases
- 📖 The federal government alone has over 2,000 databases
- 📖 Our Social Security number has become a national identification number
- 📖 Information resellers (aka information brokers) make up an entire industry that collects and sells personal data.
- 📖 Electronic profiles are built containing highly detailed and personalize descriptions of individuals
- 📖 Some concerns involve the possibility of:

IDENTITY THEFT

- 📖 The illegal assumption of someone's identity for the purposes of economic gain

MISTAKEN IDENTITY

- 📖 The electronic profile of one person can be switched with another.

- 📖 The Freedom of Information Act entitles you to look at information kept by credit bureaus and government agencies

DISCUSS THE CODE OF FAIR PRACTICE. WHY HAS THIS PRACTICE NOT BEEN MADE INTO LAW?

- 📖 The Code of Fair Practice is a set of guidelines established by former Secretary of Health, Education, and Welfare Elliot Richardson. It addresses a number of privacy concerns, and is supported by many privacy advocates.
- 📖 The fact that it is not law can be argued for a variety of reasons, some political.

DISCUSS THE VARIOUS KINDS OF COMPUTER CRIMINALS.

- 📖 Several types can include employees, outside users, hackers and crackers, organized criminals, and terrorists

WHAT ARE THE PRINCIPAL MEASURES USED TO PROTECT (PROVIDE?) COMPUTER SECURITY? WHAT IS ENCRYPTION? HOW IS IT USED BY CORPORATIONS AND INDIVIDUALS?

- 📖 Some methods include using encrypted messages, restricting access via log in/password combinations, biometric scanning, and firewalls, anticipated disasters, and backing up data
- 📖 Encryption takes an email message and scrambles it so it is unreadable by anyone but the intended recipient
- 📖 Both corporations and individuals can send email this way, including the use of Pretty Good Privacy (PGP)

WHAT IS ERGONOMICS? HOW DOES COMPUTER USE IMPACT MENTAL HEALTH? PHYSICAL HEALTH? WHAT STEPS CAN BE TAKEN TO ALLEVIATE TECHNOSTRESS? WHAT IS ERGONOMIC DESIGN?

- 📖 Ergonomics is the study of human factors related to things people use.
- 📖 Impacts on mental health can include noise, electronic monitoring, and technostress
- 📖 Impacts on physical health can include problems with eyestrain and headaches, back and neck pain, and repetitive strain injuries such as Carpal Tunnel Syndrome.
- 📖 Steps to alleviate technostress include trying to adapt computers to users rather than the other way around. Other ways to reduce stress include taking breaks, using ergonomically designed equipment, and eliminating electronic monitoring

USING TECHNOLOGY

SPYWARE

- 📖 This section refers you to Making IT Work

ERGONOMIC WORKSTATIONS

- 📖 This section refers you to Tim's toolbox section on Ergonomics

PRIVACY AVOCATION ONLINE

AIR TRAVEL DATABASE

- 📖 Have students search the web to find information about the Computer Assisted Passenger Pre-screening System (CAPPS).
- 📖 Discuss the usefulness of such a system – what are the advantages and disadvantages of using this in terms of security and privacy.

FIREWALLS

- 📖 Students are asked to write a one page paper titled "Firewall Security" and answer questions discussed in the text, such as a) Define firewall, etc.

PLAGIARISM

- 📖 Students are asked to write a one page paper titled “Plagiarism” and answer questions discussed in the text, such as a) How is copying another person’s work easier, etc.

WHAT ARE FOUR PRIMARY COMPUTER ETHICS ISSUES?

PRIVACY

- 📖 Concerns the collection and use of data about individuals

ACCURACY

- 📖 Relates to the responsibility of those who collect data to ensure that the data is correct

PROPERTY

- 📖 Relates to who owns data and rights to software

ACCESS

- 📖 Relates to the responsibility of those who have data to control and who is able to use that data

WHAT IS AN INFORMATION BROKER? WHAT IS IDENTITY THEFT? WHAT IS MISTAKEN IDENTITY?

- 📖 An information broker is an organization that collects and sells information about private individuals
- 📖 Identity theft is the illegal assumption of someone’s identity for the purposes of economic gain
- 📖 Mistaken identity is the electronic profile of one person can be switched with another. The Freedom of Information Act entitles you to look at information kept by credit bureaus and government agencies, helping you to avoid mistaken identity.

WHAT ARE HISTORY FILES? WHAT ARE COOKIES? DESCRIBE THE TWO TYPES OF COOKIES.

- 📖 Your use of the Web may be monitored, including the creation of a “history file” including a list of all the sites you’ve visited
- 📖 Cookies are special files that capture information about the web sites that you visit. Two basic types include:

TRADITIONAL COOKIES

- 📖 Monitor your activities at a single site
- 📖 Often used to provide customer service

AD NETWORK (AKA ADWARE) COOKIES

- 📖 Monitor activities across all sites you visit
- 📖 Examples include Double Click and Avenue A
- 📖 Programs called “cookie cutter programs” help to filter out these “bad” cookies
- 📖 Spyware is software that are designed to secretly record and report on an individual’s activities on the Internet. Adware is just one type of spyware

DESCRIBE THE CODE OF FAIR INFORMATION PRACTICE.

- 📖 The Code of Fair Information Practice has been established to encourage organizations to follow its recommended practices. Note it is not law, but a recommended practice.

IDENTIFY FIVE TYPES OF COMPUTER CRIMINALS.

EMPLOYEES

- 📖 The largest category of computer criminals

OUTSIDE USERS

- 📖 Criminal Suppliers and clients that have access to an organization’s computers

HACKERS AND CRACKERS

- 📖 Hackers are people who gain unauthorized access to a computer for the fun and challenge of it
- 📖 Crackers do the same for malicious reasons
- 📖 A “bomb” is a destructive computer program put into a system

ORGANIZED CRIME

- 📖 Use computers just like legitimate businesses, only for illegal purposes

- 📁 Counterfeiters and forgers use computer technology

TERRORISTS

- 📁 Knowledgeable terrorist groups can disrupt computer and communication systems

DESCRIBE FIVE FORMS OF COMPUTER CRIME.

MALICIOUS PROGRAM

VIRUSES

- 📁 Programs that migrate through networks and operating systems most attach themselves to programs and databases
- 📁 Computer Abuse Amendments Act of 1994

WORMS

- 📁 Special type of virus that doesn't attach itself to programs or databases, but fills a system with self replicating information

TROJAN HORSES

- 📁 Programs that come into a system disguised as something else

DENIAL OF SERVICE

- 📁 Like a worm, it attempts to slow down a system
- 📁 DOS attacks flood a system with requests for information or data, typically via the Internet

INTERNET SCAMS

- 📁 A scam is a fraudulent or deceptive act or operation designed to trick an individual into spending their time or money for little or no return

An Internet scam is a scam using the Internet

THEFT

- 📁 Can take many forms, including theft of hardware, software, data or computer time

- 💡 Software piracy is the unauthorized copying of programs for personal gain
- 💡 The Software Copyright Act of 1980 says it IS LEGAL to make a backup copy of software, however, these copies may NOT be sold or given away

DATA MANIPULATION

- 💡 Finding entry into a system and leaving a message may seem like a prank, but it IS AGAINST the law
- 💡 Computer Fraud and Abuse Act of 1986 makes it a crime for unauthorized persons to view, copy, or damage data using any computer across state lines.
- 💡 It also prohibits the unauthorized use of any federal (or federally insured financial institution's) computer.

LIST FOUR WAYS TO PROTECT COMPUTER SECURITY.

ENCRYPTING MESSAGES

- Encoding messages and data so it can not be read by someone without the decoding scheme

RESTRICTING ACCESS

- 💡 Keep unauthorized people away from systems by using some things as:
- 💡 Biometric scanning devices such as fingerprint or retinal (eye) scanners
- 💡 Passwords – secret words or codes that must be entered to access the system
- 💡 Firewalls – hardware and software that acts as a security buffer between the corporation's private network and all external networks, including the Internet

ANTICIPATING DISASTERS

- Physical security: protecting hardware from human and natural disasters

- Data security: protecting software and information from unauthorized tampering or damage
- Disaster Recovery Plans: contingencies for continuing operations during an emergency until normal operations can be restored
- Hot sites: fully equipped backup computer centers
- Cold sites: building with hook ups, but no equipment

BACKING UP DATA

- 📁 Equipment can usually be replaced, but data may be irreplaceable
- 📁 Data should be encrypted if sent over networks, restricted from unauthorized viewing and modification, and backed up at a remote location

WHAT IS ERGONOMICS AND WHY IS IT IMPORTANT?

- 📁 Ergonomics is the study of human factors related to things people use
- 📁 It is concerned with fitting the job to the working rather than forcing the worker to contort to fit the job

DISCUSS THE MOST SIGNIFICANT PHYSICAL CONCERNS (WITH COMPUTERS) AND HOW THEY CAN BE AVOIDED.

- 📁 Sitting too long and working with computers can lead to:

EYESTRAIN AND HEADACHE

- Take a 15 minute break every hour or two
- Avoid computer screens that flicker
- Screen should be 3-4 times brighter than the background light

BACK AND NECK PAIN

- 📁 Make sure tables and chairs are adjustable
- 📁 Monitor should be at or slightly below eye level
- 📁 Keyboards should be detachable
- 📁 Use a footrest to avoid leg strain

REPETITIVE STRAIN INJURY

- 💡 RSI (aka cumulative trauma disorder) is a name given to a number of injuries
- 💡 Carpal tunnel syndrome consists of damage to nerves and tendons in the hands
- 💡 May use ergonomically designed keyboards, take breaks and rests from working on computers

DISCUSS THE MOST SIGNIFICANT MENTAL CONCERNS (WITH COMPUTERS) AND HOW THEY CAN BE AVOIDED.

NOISE

- 💡 Women have been found to be more sensitive to noisy conditions, especially high pitched equipment noises
- 💡 Use head mounted microphones and earphones, as well as room soundproofing to reduce noise

ELECTRONIC MONITORING

- 💡 Research shows that people suffer more from electronic surveillance than from human
- 💡 Fed Ex and Bell Canada removed some electronic surveillance and found that productivity went up
- 💡 Technostress is the tension that arises when we have to unnaturally adapt to computers rather than having computers adapt to us

WHAT IS A GREEN PC?

- 💡 Microcomputers use 5% of the electricity used in the workplace
- 💡 The Environmental Protection Agency (EPA) has created the Energy Star program to discourage waste in the microcomputer industry.
- 💡 The microcomputer industry has responded with the Green PC to address the reduction of power consumption by computers

WHAT ARE THE BASIC ELEMENTS OF A GREEN PC?

SYSTEM UNIT

- 📖 Use an energy saving microprocessor
- 📖 Employ microprocessor and hard drives that switch to sleep mode when not in operation
- 📖 Replace supply unit with an adapter that uses less electricity
- 📖 Eliminate the cooling fan

DISPLAY

- 📖 Use flat panel displays (which use less electricity than CRT)
- 📖 Use “power-down” monitors
- 📖 Use screen saver software that clears the display

MANUFACTURING

- 📖 Reduce the amounts of chlorofluorocarbons (CFCs) in the manufacturing process
- 📖 Other toxic chemicals and metals (nickel, other heavy metals) are removed from the manufacturing process

WHAT OTHER ACTIONS CAN YOU TAKE TO HELP PROTECT THE ENVIRONMENT?


CONSERVE

- 📖 EPA estimates 30-40% of computers are left on all the time
- 📖 EPA estimates that 80% of the time no one is looking at a monitor
- 📖 You can use a screen saver to help save energy

RECYCLE

- 📖 You can recycle paper, ink cartridges, packaging materials, as well as computer components yourself

EDUCATE

-  You can learn more on how to recycle, and encourage others to do the same

I.T AND ENVIRONMENT

WHAT IS ICT?

Information and Communication Technology (ICT), as the term suggests, includes both information processing technology and information communication technology. Information processing technology includes prediction, simulation, and databases, among many others, while information communication technology encompasses the Internet, cell phone systems, and sensor networks, to name just a few.

ICT can be defined as the generic term used to express the convergence of information technology, broadcasting and communications

ICT may be considered one of mankind's greatest inventions of the 20th century comparable to nuclear energy. One notable feature of ICT compared with other energy-intensive system technologies is that it involves many persons: inventors, designers, developers and above all, users. ICT requires extensive support from many people and also has a great impact on many people. This characteristic is conspicuous in the ICT fields of ubiquitous computing, pervasive computing, or even ambient intelligence.

2. ICT FOR THE ENVIRONMENT

ICT has already been used to address environmental problems. For example, sensing technology has been used to monitor environmental conditions; simulation and prediction to support decision-making on environmental issues; eg weather forecasting and database technology to support the accumulation of monitored data sets.

There is still much scope for ICT to be applied to environmental problems, for ubiquitous computing is expanding the real-world applications of ICT. Further, there are many intelligent information processing techniques involving adaptive and learning capabilities that would provide flexibility for decision making.

3. ICT BY THE ENVIRONMENT

Some ICT, on the other hand, has been motivated and guided by environmental concerns. Concerns about global warming have led to the development of not only sophisticated combustion control with low emission engines but also hybrid cars and electric cars that involve the extensive use of ICT in their development.

Environmental concerns have also motivated the development of smart grids for supplying electric power, while concerns about saving energy have led to the development of intelligent homes, buildings, and factories that allow energy to be managed flexibly.

Since ICT itself involves much energy consumption and load on the environment, green ICT (or “green computing”), which means ICT that considers the environment, has attracted growing attention.

4. ICT WITH THE ENVIRONMENT

ICT is particularly important when considering interface technology between humans and the environment: it may support and enhance the symbiotic relationship between humans and the natural environment. Although technologies have mostly focused on exploiting the earth’s natural resources, the power and scale of exploitation could lead to irreversible damage to the environment. Metaphorically, I consider that ICT could serve as global eyes that monitor the environment and humans, and feed back enough information about the global state of the environment to enable humans to avoid causing irreparable damage.

I.T CAREER OPPORTUNITIES

INTRODUCTION

The progress and actions taken by a person throughout a lifetime, especially those related to that person's occupations. A career is often composed of the jobs held, titles earned and work accomplished over a long period of time, rather than just referring to one position.

While employees in some cultures and economies stay with one job during their career, there is an increasing trend to employees changing jobs more frequently. For example, an individual's career could involve being a **Programmer**, though the individual could work for several different firms and in several different areas of IT over a lifetime.

Effect of ICT on Patterns of Employment

The personal computer (PC) was developed in the early 1980s. Before this date, computers were huge, expensive machines that only a few, large businesses owned. Now PCs are found on almost every desk in every office, all over the world.

Because companies now have access to so much cheap, reliable computing power, they have changed the way they are **organized** and the way they **operate**. As a result, many people's jobs have changed.

Areas of Increased Unemployment

Some jobs have been lost as a result of computers being used to do the same work that people used to do.

Some examples of areas have suffered job losses:

Manufacturing

Many factories now have **fully automated production lines**. Instead of using people to build things, **computer-controlled robots** are used.



fig1- people working in manufacturing industry.

Robots can run day and night, never needing a break, and don't need to be paid! (Although the robots cost a lot to purchase, in the long-term the factory saves money.)

Secretarial Work

Offices used to employ many secretaries to produce the documents required for the business to run.



fig 2. secretaries working using typewriter machine.



Now people have personal computers, they tend to type and print their own documents.

Accounting Clerks

Companies once had large departments full of people whose job it was to do calculations (e.g. profit, loss, billing, etc.)

A personal computer running a spreadsheet can now do the same work.

Newspaper Printing

It used to take a team of **highly skilled printers** to typeset (layout) a newspaper page and to then print thousands of newspapers.

The same task can now be performed far more quickly using computers with **DTP** software and computer-controlled printing presses.

Areas Of Increased Employment

Although many employment areas have suffered job losses, other areas have grown and **jobs have been created**.

Sometimes people who have lost their old job have been able to **re-train** and get a **new job** in one of these growth areas.

Some examples of areas where jobs have been created:

➤ IT Technicians

All of the computers in a business need to be maintained: **hardware fixed, software installed**, etc.



fig 3 .show a pc- technician fixing a computer
IT technicians do this work.

➤ Computer Programmers

All of the software that is now used by businesses has to be created by computer programmers.

Hundreds of thousands of people are now employed in the '**software industry**'

➤ Web Designers

Much of modern business is conducted on-line, and company websites are very important.

Company websites need to be designed and built which is the role of web designers.

➤ Help-Desk Staff

People often need help using computers, and software applications.

Computer and Software Company have help-desks staffed by trained operators who can give advice.

WEBSITE DEVELOPMENT

INTRODUCTION

HTML, an initialism of Hypertext Markup Language, is the predominant markup language for web pages. It provides a means to describe the structure of text-based information in a document by denoting certain text as lines, headings, paragraphs, lists and so on. HTML is written in the form of labels (known as tags or elements), surrounded by angle brackets.

HTML pages are used for specifying web page content. They contain information and instructions to web browsers that inform them of what to display, and how it should be displayed. It is a simple format, easily learnt, and can support a number of media devices, such as sound, graphic images, and video.

HTML documents are ASCII files, and are created using a simple text editor (or an editor like Front Page). With a text editor, you cannot see what the code looks like in the browser, unless you save the page and then load it into the browser for viewing. WYSIWYG (What you see is what you get) editors like Front Page allow you to view the page as it is constructed in the editor window.

One advantage of using a simple text editor is that you have more control over the HTML code; the disadvantage is you must know the code and have a picture in your mind as to what it looks like. Another disadvantage, is since HTML is becoming more complex, it is harder to write, and sophisticated editors like Front Page support the advanced features making it easier for you to implement them.

HTML is a series of tags enclosed in `<and>` brackets. For instance, `<H1>` is an HTML tag that defines a head section of an HTML document. Certain characters are reserved, such as `&` which are interpreted as HTML codes.

Explanation:

The first tag in your HTML document is `<HTML>`. This tells your browser that, this is the start of an HTML document. The last tag in your document is `</HTML>`. This tag tells your browser that this is the end of the HTML document.

The text between the tag and the `<HEAD>` tag and the `</HEAD>` is header information. Header information is not displayed in the browser window. The text between the `<TITLE>` tag is the title of your document. The title is displayed in your browser. The text between the `` and `` tags will be displayed in a bold font.

Each HTML page adheres to a basic structure. This looks like


```
<html>
<head>
<title>Title of Document</title>
</head>

<body>
Textual Information to be displayed
</body>
</html>
```

When viewed in the browser, this page looks like.

Textual Information to be displayed

HTML TEXT FORMATTING

Tag	Description
	Defines bold text
<big>	Defines big text
	Defines emphasized text
<i>	Defines italic text
<small>	Defines small text
	Defines strong text
<sub>	Defines subscripted text
<sup>	Defines superscripted text
<ins>	Defines inserted text
	Defines deleted text
<s>	Deprecated. Use instead
<strike>	Deprecated. Use instead
<u>	Deprecated. Use styles instead
<body>	Defines body of a page
<title>	Defines title of a page
<head>	Defines heading of a page
<ahref> 	Defines a link of two pages
<table>	Defines table

HTML TAGS

HTML codes are not case sensitive. Most HTML tags need an end-tag to end it. An example of an HTML tag is , <HEAD> and the end tag for this is </HEAD>.

BOLD TYPE

The **BOLD** print tag starts with and ends with so that all text in-between the tags is printed in bold.

Example of using the bold tag

- the source in the HTML page looks like
This is bold text and this is not.
- the resultant output by the browser looks like
This is bold text and this is not.

SPACES, TABS AND FORMATTING

Newlines, spaces and tabs are ignored (single spaces accepted).

Example of spaces in the input text

- the source in the HTML page looks like
- This line contains heaps of spaces.
- the resultant output by the browser looks like
This line contains heaps of spaces.

To format text literally, the <PRE> and </PRE> tags are used. This is how the above line that contains all the spaces was inserted into this document.

LARGE SIZE TEXT

There are 6 header sizes, from H1, the smallest, to H6, the largest. The HTML tags <H1> to <H6> are used to define the size of text. The normal size is about<H2>.

Examples of using the <H.> tag to implement larger style text

- the source in the HTML page looks like <H3> This is header size 3 </H3>
- the resultant output by the browser looks like:
This is header size 3

ITALIC TEXT

The **ITALIC** print tag starts with <I> ends with </I> so that all text in-between the tags is printed in italics.

Example of using the italics tag

- the source in the HTML page looks like
<I>This is italic text</I> and this is not.
- the resultant output by the browser looks like
This is italic text and this is not.

A PAGE TITLE

A page title, specified on an HTML page, appears in the title window of the browser.

- add a title which appears in the title bar of the browser
<TITLE>This appears in the title window</TITLE>

A HORIZONTAL DIVIDER

This is used for breaking pages up, by separating sections using a horizontal line.

- the source in the HTML page looks like `<HR>`
- the resultant output by the browser looks like



In addition, a number of effects can be applied to a horizontal rule, such as color, size and width tags.

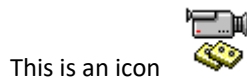
- the source in the HTML page looks like
- the resultant output by the browser looks like



ADDING IMAGES

Graphic images are added to an HTML page using the `` tag.

- the source in the HTML page looks like
`This is an icon`
- the resultant output by the browser looks like



LINKING TO OTHER PAGES

This is called a hyper-link. It shows up in the document as underlined text, and allows the user to load another page.

- the source in the HTML page looks like
` Goto next page `
- the resultant output by the browser looks like
Goto next page

FONT SIZES, FACES AND COLOR

You can specify the font face, size and color using the `` tag. Some systems may not have the desired font installed on their system.

- specify font size and color and type of font, the source in the HTML page looks like
``

- the resultant output by the browser looks like
Hello There

BACKGROUND IMAGES

An image (.gif or .jpg) can be used as a background. It should be reasonably pale so as not to distract from the displayed information.

- the source in the HTML page looks like `<BODY background="../../backgnds/blue_pap.gif">`

You can also specify a background color rather than an image. The sixteen basic colors are AQUA, BLACK, BLUE, FUCHSIA, GRAY, GREEN, LIME, MAROON, NAVY, OLIVE, PURPLE, RED, SILVER, TEAL, WHITE, and YELLOW.

- the source in the HTML page looks like `<BODY BGCOLOR="WHITE">`

BACKGROUND SOUNDS

A background sound is loaded and plays when the HTML page is loaded by the browser. The HTML tag specifies the filename to play (which is generally a .wav file for window systems), whilst the LOOP statement specifies the number of times to play the sound.

- the source in the HTML page looks like `<BGSOUND src="../../sounds/whales.wav" loop="1">`

IMAGES AS HYPERLINKS

It is common to use little pictures or Icons as links to other pages, for example, the little red up triangle's used in this document are used as hyperlinks to the top of this page.

- the source in the HTML page looks like
``
- the resultant output by the browser looks like

This is the same as inserting an image, then making the image a hyperlink.

EMBEDDED VIDEO

Some browsers such as Internet Explorer support video embedded on the HTML page. The `` tag is extended to include DYN SRC which specifies the location of the video file, LOOP which specifies how many times to play, and START, which specifies how the movie will play (MOUSEOVER or FILEOPEN).

- the source in the HTML page looks like
``
- the resultant output in the browser looks like



You should be aware that browsers that do not support embedded video need the SRC statement so a graphic image is displayed if the movie file is not found or is not supported.

Using the Reserved Characters & < > as text

As was stated earlier, the characters & < > are reserved to implement HTML tags. To use these as text requires the use of a special code sequence,

HTML Tag	Resultant Output
&	&
<	<
>	>
©	©
"	"
	insert a space
®	©

Embedding a video or sound object

One of the problems of using the background sound tag is that it is only supported by Internet Explorer. To use video and sound that will play in both Netscape Navigator and Internet Explorer, use the EMBED tag. It's format is shown below.

```
<EMBED src="../../../sounds/intro2.wav" border="0" width="145" height="60" autostart="TRUE"></EMBED>
```

The src is the path to the object, the height and width is the screen space allocated to displaying the object.

To create a table:

Tables are defined with the <table> tag

<Table>

</table>

<Tr> Is the table row element which begins each new row.

</Tr> Must end each row.

<Table>

<Tr>

</Tr>

</Table>

Example:

<html>

<title> First table</title>

<head> Mypage </head>

<body>

<table width = "300" border = "2">

<tr><td width = "210" align = "center">A</td><td width = "45">B</td>

<td width = "45">C</td></tr>

<tr><td width = "210" align = "center">A</td><td width = "45">B</td>

<td width = "45">C</td></tr>

</table>

</body>

</html>

Save your codes as tabe.html

Then open your work where you save it.

OUT PUT

A	B	C
A	B	C

HTML Forms:

A form is a graphical user interface with text entry fields and buttons, check boxes, pull down menus, scrolling lists.

Etc. form elements are elements that allow the user to enter information in a form.

To create a simple Form.

```
<form> Define a form
  <html>
  <title> User form </title>
  <heads> First form</head>
    </body>
  </form>
```

First name:

```
<Input type = "text" name = "first name">
```

```
<br>
```

Last name:

```
<Input type = "text" name = "last name">
```

```
</form>
```

```
</body>
```

```
</html>
```

Save the page as form.html

Open form.html where you save

OUTPUT

First name:

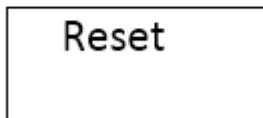
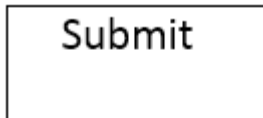
Last name:

Send and Reset Buttons

⟨Input Type = "Submit"⟩

⟨Input Type = "Reset"⟩

OUT PUT:



INTRODUCTION TO JAVASCRIPT

JavaScript

JavaScript is a scripting language designed to enhance HTML pages and provide functionality such as data validation, forms and interaction with form elements. It is a language created by Netscape, and is not Java. It is embedded into an HTML document and interpreted by the client browser. It is supported by both Internet Explorer and Netscape Navigator.

Inserting JavaScript into HTML Documents

JavaScript statements are inserted into HTML documents using the `<SCRIPT>` tag surround the JavaScript code. In addition, this is further wrapped using the embedded comment tags. The embedded comment tags prevents browsers which do not support the script tag from displaying the JavaScript code.

In this particular case, the opening

The use of `<!--` for browsers which do not support JavaScript, so if you were using such a browser, the line would have read

Last modified 19th March 97.

Example 1

The following example embeds the current date and time directly into the HTML document.

The date is Sat Dec 13 2014 10:10:35 GMT-0800 (Pacific Standard Time)

The code for this is as follows.....

```
<SCRIPT language="javascript">  
<!-- start of javascript hide from old browsers document.write(date())-->  
</SCRIPT>
```

The <SCRIPT> and </SCRIPT> tags surround the JavaScript code. In addition, this is further wrapped using the embedded comment tags <!--and-->. The embedded comment tags prevents browsers which do not support the script tag from displaying the JavaScript code.

In this particular case, the opening <SCRIPT> tag specifies the language to be used as JavaScript.

Another example shown below displays the last modified date and time of the document.

Last Modified 01/14/1999 07:00:00

The code for this is as follows.

Last modified

```
<SCRIPT language="javascript">  
<!-- start of javascript hide from old browsers document.write(document.lastmodified);//-->  
</SCRIPT>
```

```
<NOSCRIPT>  
19th march 97.  
</NOSCRIPT>
```

The use of <NOSCRIPT> and </NOSCRIPT> is for browsers which do not support JavaScript, so if you were using such a browser, the line would have read

Last modified 19th March 97.

The Window Status Bar

The bottom line of the browser window is called the STATUS BAR. Using JavaScript, you can write directly to the status bar. Consider the following example, which displays a message in the status bar.

Look at the status bar to see this text displayed there now.

```
<script language="javascript">  
<!--start of javascript hide from old browsers window.status='welcome to my page';!-->  
</script>
```

OnMouseOver

Well, that was pretty neat. But it would be nice to have the contents of the status bar change from time to time, for instance, when the mouse cursor moves over a hyper-link, the status bar could display a message which informs you of what the hyper-link is all about. To do this, the *mouseover* statement is added. Consider the following example.

```
<A href="whatisjs.htm" target="_top"  
onmouseover="window.status='Click here to learn about JavaScript'; return true">
```

<IMG



Message Box Alerts

The next thing we will discuss in message boxes. Sometimes, when a user clicks on something, you want to pop up a message box. The following example does just that.

```
<A href="whatisjs" target="_top" onclick="alert('sorry,that document is not available right now..');return false">
<IMG src="../images/ballloc.gif" border="0">
</A>
```

Note the **return false**, this causes the click on the href to not perform the action of loading the referenced page. As an example, try creating a new document and change it to **return true** and note the different effect this has.

Confirming Actions

In some cases, where a user clicks on a link that might download a rather large file, you would consider giving the user a means of canceling the request before the action is performed. An example of this follows,

```
<A href="../sounds/intro1.wav" onclick="return window.confirm('this might take a while to download.download
sound file?');">
<IMG src="../images/ballloc.gif" border="0">
</A>
```

The confirm function returns either true or false, depending upon the user choices. If it returns true, the action is performed, and the browser will load the referenced wave file.

Detecting the Browser Type

Because browsers display HTML slightly differently, and not all statements are supported by every browser, JavaScript is often used to perform actions based on a particular browser type. This calls for some code which will detect the browser type. An example follows,

```
<script language="javascript">
<!-- start of javascript hide from old browsers,this tests for internet explorer
4if((navigtion.appversion.substring(0,1)=="4")&&(navigation.appName=="microsoft internet Explorer"))>{
document.write('This is Internet Explorer 4');
}
else{

document.write('this is not Internet Explorer 4');
}
//-->
</script>
```

This is not Internet Explorer 4

Here is another code example, which detects various versions of a browser.

```
<SCRIPT language="javascript">
<--determine if running IE 3.02 or less
var ua = window.navigator.userAgent
var msie = ua.indexOf( "MSIE " )
var major = parseInt( ua.substring (msie+5, ua.indexOf(".", msie)));
var minor = parseInt( ua.substring (msie+7, ua.indexOf(";", msie)));
// document.write("msie = "+msie+" ua="+ua);
if( msie > 0 ) // we are running MSIE {
if( major <= 3 ) {
if( minor <= 1 ) &&
confirm("You need to upgrade your version of Internet Explorer. Do you want to upgrade now?"))
{
location.href = "http://www.microsoft.com/ie/"
}
}
}
else {
alert("Please use Internet Explorer version 4.0 or greater.");} // -->
</SCRIPT >
```

Functions

JavaScript supports functions. These are a series of JavaScript statements combined into a single unit and assigned a name. A function could validate a users entry in a dialog box, write specific details to the document, access a database or manipulate an ActiveX control. Functions often return values, like true or false.

Here is an example of a function which displays a text input box and when the user enters data into it, displaying it in an alert box. In further examples, this will be expanded. The JavaScript code contains a function GetTemp, which references the InputBox, and uses the alert function to read the temperature and display it respectively. In addition, a standard <FORM>element and submit button are used to activate the JavaScript procedure.

Enter Number in Degrees Fahrenheit:

The code for this is as follows.....

```
<SCRIPT language="javascript">
<start of JavaScript hide from old browsers
function DisplayTemp( value ) {
alert("The temperature in Fahrenheit is " + value);
}
// -->
</SCRIPT >
```



```
<form name="getTemp1">
Enter Number in Degrees Fahrenheit:
<input type="text" size="20" name="degreesF">
<input type="button" name="CheckTemp1" value="Temperature"
onclick="DisplayTemp(document.GetTemp1.DegreesF.value)">
</form>
```

A FORM is used to hold a text entry box labeled DegreesF and a push button labeled CheckTemp1. When the user clicks on this button, the associated JavaScript event DisplayTemp() is invoked. This function procedure prints out the value of the variable passed to it, which was document.GetTemp1.DegreesF.value

The use of `alert` displays the temperature by using concatenated strings, the `+` character is used to indicate concatenation and the quotes to define text strings.

Expanding the above example to perform conversion

A function in JavaScript is a series of JavaScript statements that are enclosed by the `function` and `{ }` statements. A function that does not accept any values has the name of the function followed by an empty set of parentheses. In addition, a function may, or may not, return a value for the function.

It would be more meaningful if we actually did something better than just echo users responses. Lets consider the previous example, and this time, expand it to convert the temperature from Farenheight to Celcius. To do this, we will add a function `Convert Celcius()` which will take the value in degrees Fahrenheit and display the answer in degrees Celcius.

It looks like

Enter Number in Degrees Fahrenheit:

And the JavaScript code to implement this change looks like

```
<SCRIPT language="javascript">
<!--start of JavaScript hide from old browsers
function Display2( value ) {
var DegreesC
DegreesC = ConvertCelcius( value );
alert("The temperature in Celcius is " + DegreesC + " degrees C.");
}
Function ConvertCelcius( value ) {
return (eval(value) - 32) * (5 / 9);
}
// -->
</SCRIPT>
```

```
<FORM name="gettemp2">
Enter Number in Degrees Fahrenheit:
<input type="text" size="20" name="DegreesF">
<input type="button" name="CheckTemp2" value="Temperature"
onclick="Display2(document.GetTemp2.DegreesF.value)">
</FORM>
```

The function `Display2()` also indicates the declaration of a local variable called `DegreesC` using the `var` statement. The statement `DegreesC = Convert Celcius(value)` passes the value of `DegreesF` to the function `Convert Celcius()` which returns the result into the variable `DegreesC`.

Validating Numeric Entry

In this example, the scripts are expanded to ensure that the entry is numeric. Obviously, the function `GetCelcius3()` should not be called if the entry is not numeric. It uses the `parseInt()` function in JavaScript to determine if the field is numeric. If `parseInt` cannot convert the string, it returns `Nan`, which is checked for by the `is Nan` function.

It looks like

Enter Number in Degrees Fahrenheit:

The code looks like

```
<script language="JavaScript">
<!--start of JavaScript hide from old browsers
function Display3( value ) {
var DegreesC
var DegreesF
if( isNaN( parseInt( value, 10 ) ) ){
alert("Please enter a numeric Value.");
}
else {
DegreesC = GetCelcius3( parseInt( value, 10 ) )
alert("The temperature in Celcius is " + DegreesC + " degrees C.");
}
}

function GetCelcius3( value ) {
return (value - 32) * (5 / 9)
}
// -->

</script>
```

```
<form name="GetTemp3">
Enter Number in Degrees Fahrenheit:
<input type="button" name="CheckTemp3" value="Temperature"
onclick="Display3(document.GetTemp3.DegreesF.value)">
</form>
```

This demonstrates the use of an *If Else* statement to check if the data entry is numeric.

Validating the Number Range

In this example, we expand the previous by including a range check on the input numeric data, in this case, we limit the input data to a range of 32 to 100 inclusive. This is done using an additional *if else* statement and a compound relational test.

It looks like

Enter Number in Degrees Fahrenheit:

The code looks like

```
<script language="JavaScript">
<!--start of JavaScript hide from old browsers
function Display4( value ) {
var DegreesC
var DegreesF
DegreesF = parseInt( value, 10 );
if( isNaN( DegreesF ) ){
alert("Please enter a numeric Value.");
```

```

}
else {
if( (DegreesF < 32) || (DegreesF > 100) ) {
alert("Please enter a value between 32 and 100.");
}
else {
DegreesC = GetCelcius4( parseInt( value, 10) );
alert("The temperature in Celcius is " + DegreesC + " degrees C.");
}
}
}
}

```

```

function GetCelcius4( value ) {
return (value - 32) * (5 / 9)
}
// -->
</script >

```

```

<form name="GetTemp4">
Enter Number in Degrees Fahrenheit:Enter number in degrees Fahrenheit: input type="text" size="20"
name="DegreesF">
<input type="button" name="CheckTemp4" value="Temperature"
onclick="Display4(document.GetTemp4.DegreesF.value)">
</form>

```

COOKIES

In this section we shall show the use of cookies. A cookie is a file that is placed on the users computer, and saves some information which can be later accessed using JavaScript. Consider applications like a shopping cart, where a user browses a web site, selecting items, each time the item is added to the cookie file. Once the user has finished shopping, the JavaScript code can read back the cookie file, determining the items the user wants to purchase, and generate an invoice.

Provided below are some basic JavaScript statements for implementing cookies. These are available in the public domain. The entire JavaScript code is placed in the head section of the HTML document.

Now, let us show you how to use them. The following HTML code displays a message box and allows the user to enter their name, when the page is accessed the first time. This happens because the JavaScript function who() is called, which determines the cookie does not exist (this is the first time the user has accessed the page), so it presents a dialog box. On subsequent accesses to the page, the function finds the cookie, extracts the username stored there, and prints it as part of the document.

```

<script language="JavaScript">
<!--begin script
var expDays = 365;
var exp = new Date();
exp.setTime(exp.getTime()+(expDays*24*60*60*1000));

function Who(info){
// who
var VisitorName = GetCookie('VisitorName');
if( VisitorName == null ) {
VisitorName = prompt("Please enter your first name", "");
SetCookie('VisitorName', VisitorName, exp );
}
}

```

```

return VisitorName;
}

function set() {
  VisitorName = prompt("Please enter your first name","");
  SetCookie('VisitorName', VisitorName, exp );
}

function getCookieVal( offset) {
  var endstr = document.cookie.indexOf(";", offset);
  if( endstr == -1 )
    endstr = document.cookie.length;
  return unescape( document.cookie.substring(offset,endstr));
}

function GetCookie(name) {
  var arg = name + "=";
  var alen = arg.length;
  var clen = document.cookie.length;
  var i = 0;
  while( i < clen ) {
    var j = i + alen;
    if( document.cookie.substring(i, j) == arg)
      return getCookieVal(j);
    i = document.cookie.indexOf(" ",i) + 1;
    if( i == 0) break;
  }
  return null;
}

function SetCookie( name, value ) {
  var argv = SetCookie.arguments;
  var argc = SetCookie.arguments.length;
  var expires = (argc > 2) ? argv[2] : null;
  var path = (argc > 3) ? argv[3] : null;
  var domain = (argc > 4) ? argv[4] : null;
  var secure = (argc > 5) ? argv[5] : false;
  document.cookie = name + "=" + escape(value) +
    ((expires == null) ? "" : ("; expires=" + expires.toGMTString())) +
    ((path == null) ? "" : ("; path=" + path)) +
    ((domain == null) ? "" : ("; domain=" + domain)) +
    ((secure == null) ? "" : ("; secure=" + ""));
}

// end of script -->
</script >

```

Now, let us show you how to use them. The following HTML code displays a message box and allows the user to enter their name, when the page is accessed the first time. This happens because the JavaScript function `who()` is called, which determines the cookie does not exist (this is the first time the user has accessed the page), so it presents a dialog box. On subsequent accesses to the page, the function finds the cookie, extracts the username stored there, and prints it as part of the document.

Hello **null**. It's good to see you here.

DETERMINING THE SCREEN WIDTH

The following JavaScript code determines the screen width.

```
<!-- this must be placed in the header section-->
< body onload = "init()" >
< img name = "H" src = "javascript: single_pixel_xbm" width = "100" height = 1 >

<script language => "JavaScript"
<!--
var single_pixel_xbm = "#definex_width1 \n#definex_height1 \nstatic char
x _bits = {0x00};\n";
function init(){
alter(Width= + document.images'H'.Width;
}
//-->
</script >
```

General Web Design Issues

In this section, we discuss issues concerned with the design of Web Pages. Much of the information covered here is a summary of looking at features that work successfully, derived from looking at top rated sites and sites that have proved to be effective. The topics below show the areas that this section deals with.

THE ELEVEN BASIC RULES OF DESIGN WEB PAGE

1. Page layout, Size, Titles
2. Text, Fonts and Sizes
3. Colors
4. Images, Backgrounds, Icons
5. Multiple Platforms
6. Client Side versus Server Side
7. Use of Html, Java, JavaScript, VBScript, Active X, Plug ins
8. Download times, hits per page
9. Copyrights
10. Accessibility
11. Ease of Navigation

Page

layout,

Size,

Titles

Each page should have a unique title. A page title is defined in the<HEAD> section of an HTML (Hyper-Text markup Language) document, using the<TITLE> tag. This is covered in the HTML overview section. A unique page title aids

in maintaining the web site using a package such as Front Page explorer. The page title is also displayed in the top of the browser window, and appears at the top of the page when it is printed.

Each page should be no more than about one to two pages longer than the screen depth. This minimizes scrolling. In addition, the user should not have to scroll horizontally across the screen. Any information should fit easily on the page and not appear cluttered. Ample use of white space is important for readability and quick identification of the information presented. Obviously, this page does not conform to these requirements!

Page size is linked to download speeds, the more information the page contains, the longer it is, and the more time it will take to download. The use of horizontal lines or other methods to divide sections on a page are important to give the impression of separation to users.

Also, avoid the use of a page that has little to say, classic pages such as "Click here to enter" are a waste of time and convey little to users.

Text, Fonts and Sizes

While it is tempting to use a lot of font sizes and styles, users should avoid this for a number of reasons. From a design point of view, consistency in the use of headings and font styles is important. At most, you should use three different sizes, one for the body of text, a slightly larger one for titles, and a larger one again for the main title, which appears at the top of the page. Do not make the large title too big, it will detract from the information, and consume valuable screen real estate that is better suited to displaying information.

Avoid the use of too many font styles or faces. A font face is "Times Roman" or "Courier". Be aware that a user might be using a web browser that does not have the specified font installed on their system. Instead, stick to a common font, and use attributes such as bold and italic sparingly.

Avoid the use of underlining. In web pages, underlining most often refers to hyper-links. Using underline for text will confuse users, making them think they are hyper-links to other material.

In addition, when font faces are specified using Front Page, statements are embedded into the HTML document. This has the potential to treble the size (in bytes) of the document, which will take longer to download, consume more disk space, and increase network bandwidth (because the file is larger, it consumes more of the bandwidth over a longer period).

Font sizes are specified using the HTML statement . In general, text should use font style 2, if a font size is not specified, most browsers will default to this size. A font size of 6 is the largest, with 1 being the smallest. Avoid font size 1 where possible.

Colors

The use of too much color confuses and detracts from the information being presented. Use color sparingly, for titles, headings, or important keywords that you want to highlight. Certain colors cause reactions with users, an example is Red.

Color is embedded using statements in HTML. Try to stick to one of the sixteen default colors specified in HTML 2.0, the reason being that you have no control over what color setting the user has specified for their screen display (it might be 16 colors), and your selection might make the information unreadable on their screen.

Some of the standard 16 colors available in HTML 2.0 are RED, YELLOW, GREEN, CYAN, BLUE, WHITE, BLACK and MAGENTA.

Images,

Backgrounds,

Icons

Background images or wallpaper should be sparingly used. It is better to use a background color, as this does not involve any download time. Take notice again of contrasting colors, so that text is clearly visible. If using a background image, ensure that it is very pale, so that any information placed on it is clear and easy to read.

Large background images can increase download time considerably, and take into account what will happen if the user resizes the window, or uses a much larger display than what you designed for. For instance, you create a wallpaper of 640 by 480, but the user views this on 800 by 600. Try to use a background image that is expandable. What this means is that the left side matches the right side, and the top side matches the bottom side. If you take one background image, then place four more of the same image, one on the top and bottom, and one on the right and left, the background image looks symmetrical.

For images embedded within the document, use appropriate sizes. This means no more than half the viewing area, so for a screen viewing size of 640 by 480, the largest image should be 320 x 240. Again, as with text colors, use images that conform where possible to 256 colors. Sometimes, file sizes of images are important. It takes time to download images across the network, and in general, images tend to be larger than the HTML page itself. Consider a 10K page, which has four images on it, each of 32K bytes each. The total size is thus 138K (10K + 4 * 32K), which on a 28.8Kbps dial up link will take a minimum of 38.3 seconds to download.

Saving an image in an alternative format (for instance, from .gif to .jpg) often results in a smaller file size. Where larger images are necessary, an example being maps, consider using a thumbnail (a much smaller image typically 100 by 100 or less) as a hyperlink to the much larger image.

Screen real estate is always at a premium. When using images, rather than leave white space around the image, it is better to wrap the text around the image. This is done using two column tables that span the width of the page, one column of the table holds the image, the other the text.

Icons are little symbols that, in web pages, represent shortcuts or links to material. In this document, we have used the image to refer to the top of the page. If the user clicks on this little icon, they return to the top of this document. When using icons, always use them in the same context, with only one meaning, so that left icons always refer to the previous page, up icons refer to the top, down icons to the bottom, and so on. This avoids confusion. Also, use the same icons across all your web pages, users will expect that a particular icon on one page will behave the same way on another page. Icons should resemble the function that they perform or the action associated with them. If this is not the case, use simple text links. In addition, icons should be small, and an alternative should be provided in case the user has graphics disabled on their web browser, or is sight impaired.

Multiple Platforms

One thing you have little control over is what browser the user has. With Active X Server Pages (ASP), or JavaScript, it is possible to detect the type of browser and present the information suited for a particular type of browser (though JavaScript only works for JavaScript enabled browsers). In addition, other factors such as screen size and color depth is beyond your control, so if you intend to use high resolution screen sizes and 16 bit color depths, that fact should be announced to your potential viewers on the home page.

Client

Side

versus

Server

Side

Client side (CS) means that the information or request is handled within the web browser. Once the page is loaded, an Internet or network connection is no longer required. All the information is embedded within the page. Client side is normally written in a script language like JavaScript or Visual Basic Script.

Server side (SS) means that the user sends the information to a dedicated server, which is required to perform some action and return results to the user. This requires an Internet or network connection. An example of server side is CGI (common gateway interface). Without the server, the information cannot be processed. Server side scripting allows you to send different pages to users, dependant upon a number of factors (such as browser type and personal interests). Shopping cart systems are generally implemented using SS.

The down side of SS is that the files must reside on the server, and the user requires a connection to the server. The server must perform the action required, which takes resources away from other things the server might be doing (in other words, places an additional load on the server).

Another advantage of using CS is the pages can readily be stored on a CD-ROM and are portable, so an Internet connection is not required to use them.

Clickable image maps should be implemented client side, as this will be faster (since no requests to the server will be done), however, client side will increase the size of the web pages.

Use of Java, JavaScript, VBScript, Active X, Plug ins

Because all browsers do not support scripting, ensure that you provide alternatives, so users can view the information, or get access to a browser which does support what you implement.

Remember that Active X and VBScript are only supported by Internet Explorer, a Microsoft browser. If you implement pages using this technology, you are limiting it to clients with that particular browser and a Windows based platform.

If you use plugins (components which extend the capability of the browser), provide links to where the plugins are located so the user can download them and install them on their computer. Some users may not have permissions to install plugins on their computer, so it is wise to limit their use. The other problem with plugins is some tend to make the computer unstable, and there is also the issue of support and how long the plugin will be available for. The same applies to Active X components.

Download times, hits per page

Users access information from a wide variety of sources, using a wide variety of communication links, some of which are high speed, and some of which are still very slow. In general, design for a lower speed limit of 14.4Kbps, with an average page download time of around 20 seconds.

The more information you place on a page, the longer it will take to download. Every image and sound reference on a page often creates another hit on the server (most browsers can be configured to limit the number of simultaneous connections it can open), which the server must handle by allocating resources to it. If a page has a lot of embedded references, the download time can increase dramatically, by having all these open connections at once, attempting to download all the images across a narrow bandwidth connection.

It's a good idea to limit the number of hits per page to less than 10. If you use frames, a lot of hits can be generated at once, one for each page in the frameset, one for the default document, and one for each graphic referenced on each page within the frameset. It is not uncommon to have 20 or more hits generated from accessing a single document that specifies a frame set. You would need to add up all the page sizes, and image sizes referenced on each, to get an indication of how long it will take to be downloaded at 14.4Kbps.

Copyrights

It is important that a copyright message be placed on the page. This identifies the page as having ownership, either by the organization or by an individual. Other aspects you might include are the files last modified date, so users readily know how old the page is and when it was last modified.

If you use images, sounds or other resources from other sources, you must first obtain permission to do so (unless those files have been placed in the public domain as freely useable). Place the "permission to use" or "copyright .." statements as close as possible to the image that you are using.

If quoting text or information from other sources, acknowledge the source either directly, or at the bottom of the page in a reference. It is only fair that people who have placed information on the web be acknowledged for their efforts.

Accessibility

This refers to designing pages for users who have disabilities in one form or another. With web pages, this usually means visually impaired persons. Obvious things that stand out are the use of icons as hyperlinks, with no text alternatives.

In a browser such as Internet Explorer, you can enlarge the display font easily (In IE4.0, from the menu bar, View, Fonts, Largest). This option increases the size of all text displayed by the browser, BUT, it does not enlarge the graphics as well. Thus symbols or Icons which act as hyper-links remain invisible to sight impaired users. Providing the option of text links and pages with text only is a necessary solution.

Navigation

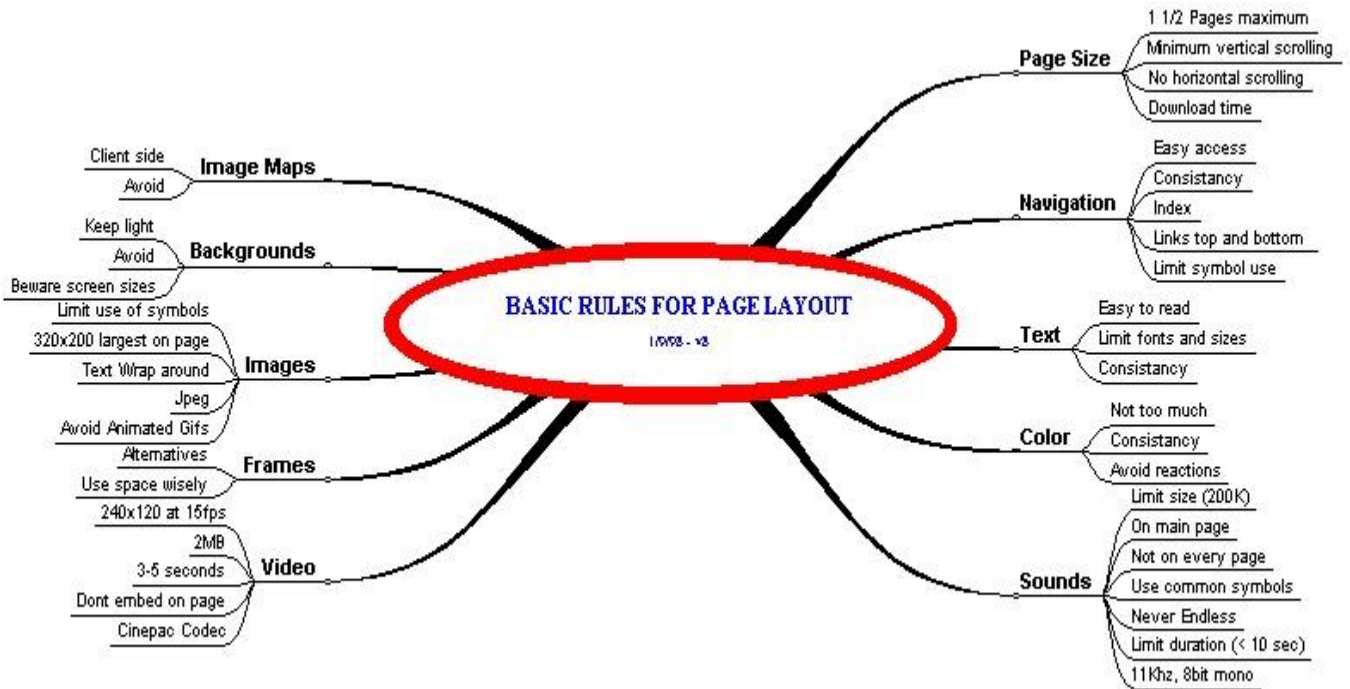
Issues

Browsing pages on the web is not like reading a book. A book is structured, and its organization is quickly found. Information is presented in chapters, index, table of contents and a reference or appendix section.

Its easy for a user to flip pages, but still know where they are in a book. With web pages, that is significantly more difficult. It's hard to see where you are and how this page relates to the previous page, or the next one for that matter. The major cause for this is hyperlinks. Consider a page where a user is discussing various makes of automobiles, as has a hyperlink to a web site for the Ford motor company. A user browsing the current web site clicks on the hyperlink, and is suddenly deep inside another web site, with its own culture, its own symbols, its own navigation systems. User context is lost; they no longer know where they are. If there is an icon at the top of the page called BACK, and the user clicks on it, they will not go to the page they have just come from. This might be self explanatory to those familiar computers or the Internet, but thousands of new people join the Internet daily, and for them, this is a major issue.

Each page should have clear links to the previous and next page (where appropriate), and the home page. If you implement your web site using frames, one frame is usually used for navigation purposes, with various subsections always accessible from within frame. The net effect is that each topic is always only one click away, and if a user gets lost, the navigation frame is there to help them.

Always use the same Icons or symbols throughout your web site, remember, this creates a culture for those that visit your site, and consistent use of symbols is essential to avoid confusion. Avoid changing the style of navigation. If you present navigation via a frame, don't suddenly change to something else, as this will only confuse users. You might also consider offering a navigation alternative, or several methods of navigation. However, the more information and navigation you place on a page, the more overwhelming it becomes, the longer it takes to download, and the less likely a user is to read all of it or access the information contained therein. Simple is often best.



C PROGRAMMING

AN INTRODUCTION TO C PROGRAMMING

It is one of the most used programming language in software development area. At the beginning, C was designed and developed for the development of UNIX operating system. Today C has been widely used to develop many types of application software such as basic calculator word processor games, operating system etc.

Editing in C language

- Editing is a process of writing the C source code. programmers write C source code using editor program. widely used editing programs are Linux, platform via and Emacs.
On windows platform, we can use Notepad as editor *program but most* use software package for C such as C++ builder, DOS C++. This software provides complete programming environment for writing, managing, compiling, debugging and testing C source code. This is known as Integrated Development Environment (IDE)

Compiling in C language

The C source code has to be translated into machine language code. Machine language is a language which are in the form of binary numbers that a computer understand. The process of code translation is called compiling. During compilation process, the compiler can detect any syntax error.

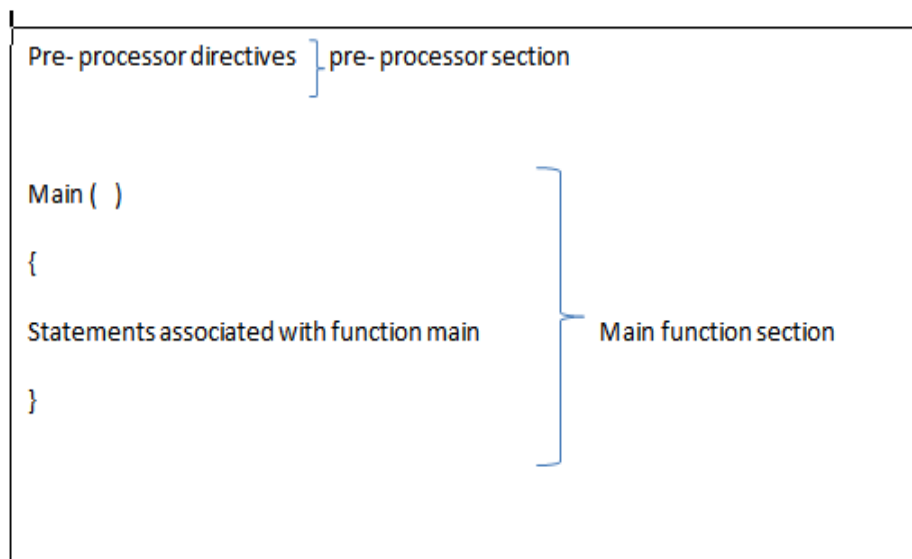
Execution

The execution stage is where you can run the program to check whether it produce the desired output. This stage can be a testing stage where you can check if you get the desired output.

C program layout

Basically, a C program consists of two sections

1. Pre- processor directives
2. Main function



A simple C program

The following program is written in the C programming language which can display the word "program in C is easy"

```
#include

main()

{

printf("Programming in C is easy.\n");
```

```
}
```

Sample Program Output

```
Programming in C is easy.
```

```
—
```

A NOTE ABOUT C PROGRAM

In C, lowercase and uppercase characters are very important! All commands in C must be lowercase. The C programs starting point is identified by the word

```
main()
```

This informs the computer as to where the program actually starts. The brackets that follow the keyword *main* indicate that there are no arguments supplied to this program. The two braces, { and }, signify the begin and end segments of the program. The purpose of the statement

```
#include
```

is to allow the use of the *printf* statement to provide program output. Text to be displayed by *printf()* must be enclosed in double quotes. The program has only one statement

```
printf("Programming in C is easy.\n");
```

printf() is actually a function (procedure) in C that is used for printing variables and text. Where text appears in double quotes "", it is printed without modification. There are some exceptions however. This has to do with the \ and % characters. These characters are modifiers, and for the present the \ followed by the n character represents a newline character. Thus the program prints

```
Programming in C is easy.
```


and the cursor is set to the beginning of the next line. As we shall see later on, what follows the \ character will determine what is printed, ie, a tab, clear screen, clear line etc. Another important thing to remember is that all C statements are terminated by a semi-colon

Summary of major points so far

- program execution begins at *main()*
- keywords are written in lower-case
- statements are terminated with a semi-colon
- text strings are enclosed in double quotes
- C is case sensitive, use lower-case and try not to capitalise variable names
- \n means position the cursor on the beginning of the next line
- *printf()* can be used to display text to the screen
- the curly braces {} define the beginning and end of a program block

DATA TYPES AND CONSTANTS

The four basic data types are

- **INTEGER**

These are whole numbers, both positive and negative. Unsigned integers (positive values only) are supported. In addition, there are short and long integers.

The keyword used to define integers is,

```
int
```

An example of an integer value is 32. An example of declaring an integer variable called **sum** is,

```
int sum;
```

```
sum = 20;
```

- **FLOATING POINT**

These are numbers which contain fractional parts, both positive and negative. The keyword used to define float variables is,

-

- *float*

An example of a float value is 34.12. An example of declaring a float variable called **money** is,

```
float money;  
money = 0.12;
```

- **DOUBLE**

These are exponential numbers, both positive and negative. The keyword used to define double variables is,

-

- *double*

An example of a double value is 3.0E2. An example of declaring a double variable called **big** is,

```
double big;  
big = 312E+7;
```

- **CHARACTER**

- These are single characters. The keyword used to define character variables is,

-

- *char*

An example of a character value is the letter **A**. An example of declaring a character variable called **letter** is,

```
char letter;  
letter = 'A';
```

Note the assignment of the character *A* to the variable *letter* is done by enclosing the value in **single quotes**. Remember the golden rule: Single character - Use single quotes.

Sample program illustrating each data type

```
#include < stdio.h >

main()

{

    int sum;

    float money;

    char letter;

    double pi;

    sum = 10; /* assign integer value */

    money = 2.21; /* assign float value */

    letter = 'A'; /* assign character value */

    pi = 2.01E6; /* assign a double value */

    printf("value of sum = %d\n", sum );

    printf("value of money = %f\n", money );

    printf("value of letter = %c\n", letter );

    printf("value of pi = %e\n", pi );

}
```

Sample program output

```
value of sum = 10  
  
value of money = 2.210000  
  
value of letter = A  
  
value of pi = 2.010000e+06
```

PREPROCESSOR STATEMENTS

The **define** statement is used to make programs more readable. Consider the following examples,

```
#define TRUE 1 /* Don't use a semi-colon , # must be first character on  
line */  
  
#define FALSE 0  
  
#define NULL 0  
  
#define AND &  
  
#define OR |  
  
#define EQUALS ==  
  
  
game_over = TRUE;  
  
while( list_pointer != NULL )  
  
.....
```

Note that preprocessor statements begin with a # symbol, and are NOT terminated by a semi-colon. Traditionally, preprocessor statements are listed at the beginning of the source file.

Preprocessor statements are handled by the compiler (or preprocessor) before the program is actually compiled. All # statements are processed first, and the symbols (like TRUE) which occur

in the C program are replaced by their value (like 1). Once this substitution has taken place by the preprocessor, the program is then compiled.

In general, preprocessor constants are written in **UPPERCASE**.

Class Exercise C4

Use pre-processor statements to replace the following constants

0.312

W

37

Answer

```
#define smallvalue 0.312
#define letter      'W'
#define smallint    37
```

LITERAL SUBSTITUTION OF SYMBOLIC CONSTANTS USING #define

Lets now examine a few examples of using these symbolic constants in our programs. Consider the following program which defines a constant called TAX_RATE.

```
#include
```

```
#define TAX_RATE 0.10
```

```
main()
```

```
{
```

```
float balance;  
  
float tax;  
  
balance = 72.10;  
  
tax = balance * TAX_RATE;  
  
printf("The tax on %.2f is %.2f\n", balance, tax );  
  
}
```

The pre-processor first replaces all symbolic constants before the program is compiled, so after preprocessing the file (and before its compiled), it now looks like,

```
#include  
  
#define TAX_RATE 0.10  
  
main()  
{  
  
float balance;  
  
float tax;  
  
balance = 72.10;  
  
tax = balance * 0.10;  
  
printf("The tax on %.2f is %.2f\n", balance, tax );  
  
}
```

YOU CANNOT ASSIGN VALUES TO THE SYMBOLIC CONSTANTS

Considering the above program as an example, look at the changes we have made below. We have added a statement which tries to change the TAX_RATE to a new value.

```
#include

#define TAX_RATE 0.10

main()
{
    float balance;

    float tax;

    balance = 72.10;

    TAX_RATE = 0.15;

    tax = balance * TAX_RATE;

    printf("The tax on %.2f is %.2f\n", balance, tax );

}
```

This is **illegal**. You cannot re-assign a new value to a symbolic constant.

- **ITS LITERAL SUBSTITUTION, SO BEWARE OF ERRORS**

As shown above, the preprocessor performs literal substitution of symbolic constants. Lets modify the previous program slightly, and introduce an error to highlight a problem.

```
#include

#define TAX_RATE 0.10;

main()

{

float balance;

float tax;


balance = 72.10;

tax = (balance * TAX_RATE )+ 10.02;

printf("The tax on %.2f is %.2f\n", balance, tax );

}
```

In this case, the error that has been introduced is that the *#define* is terminated with a semi-colon. The preprocessor performs the substitution and the offending line (which is flagged as an error by the compiler) looks like

```
tax = (balance * 0.10; )+ 10.02;
```

However, you do not see the output of the preprocessor. If you are using TURBO C, you will only see

```
tax = (balance * TAX_RATE )+ 10.02;
```

flagged as an error, and this actually looks okay (but its not! after substitution takes place).

MAKING PROGRAMS EASY TO MAINTAIN BY USING #define

The whole point of using *#define* in your programs is to make them easier to read and modify.

Considering the above programs as examples, what changes would you need to make if the TAX_RATE was changed to 20%.

Obviously, the answer is once, where the *#define* statement which declares the symbolic constant and its value occurs. You would change it to read

```
#define TAX_RATE = 0.20
```

Without the use of symbolic constants, you would hard code the value 0.20 in your program, and this might occur several times (or tens of times).

This would make changes difficult, because you would need to search and replace every occurrence in the program. However, as the programs get larger, **what would happen if you actually used the value 0.20 in a calculation that had nothing to do with the TAX_RATE!**

SUMMARY OF #define

- allow the use of symbolic constants in programs
- in general, symbols are written in uppercase
- are not terminated with a semi-colon
- generally occur at the beginning of the file
- each occurrence of the symbol is replaced by its value
- makes programs readable and easy to maintain

PREPROCESSOR STATEMENTS

The **define** statement is used to make programs more readable. Consider the following examples,

```
#define TRUE 1 /* Don't use a semi-colon , # must be first character on  
line */
```

```
#define FALSE 0
```

```
#define NULL 0
```

```
#define AND &
```

```
#define OR |
```

```
#define EQUALS ==
```



```
game_over = TRUE;  
  
while( list_pointer != NULL )  
  
.....
```

Note that preprocessor statements begin with a # symbol, and are NOT terminated by a semi-colon. Traditionally, preprocessor statements are listed at the beginning of the source file.

Preprocessor statements are handled by the compiler (or preprocessor) before the program is actually compiled. All # statements are processed first, and the symbols (like TRUE) which occur in the C program are replaced by their value (like 1). Once this substitution has taken place by the preprocessor, the program is then compiled.

In general, preprocessor constants are written in **UPPERCASE**.

Class Exercise C4

Use pre-processor statements to replace the following constants

0.312

W

37

CLASS EXERCISE C4

Use pre-processor statements to replace the following constants

0.312

W

37

```
#define smallvalue 0.312  
#define letter 'W'  
#define smallint 37
```

LITERAL SUBSTITUTION OF SYMBOLIC CONSTANTS USING #define

Lets now examine a few examples of using these symbolic constants in our programs. Consider the following program which defines a constant called TAX_RATE.

```
#include

#define TAX_RATE 0.10

main()

{

float balance;

float tax;


balance = 72.10;

tax = balance * TAX_RATE;

printf("The tax on %.2f is %.2f\n", balance, tax );

}
```

The pre-processor first replaces all symbolic constants before the program is compiled, so after preprocessing the file (and before its compiled), it now looks like,

```
#include

#define TAX_RATE 0.10

main()
```

```
{  
  
float balance;  
  
float tax;  
  
  
balance = 72.10;  
  
tax = balance * 0.10;  
  
printf("The tax on %.2f is %.2f\n", balance, tax );  
  
}
```

YOU CANNOT ASSIGN VALUES TO THE SYMBOLIC CONSTANTS

Considering the above program as an example, look at the changes we have made below. We have added a statement which tries to change the TAX_RATE to a new value.

```
#include  
  
  
#define TAX_RATE 0.10  
  
  
main()  
  
{  
  
float balance;  
  
float tax;  
  
  
balance = 72.10;  
  
TAX_RATE = 0.15;
```

```
tax = balance * TAX_RATE;

printf("The tax on %.2f is %.2f\n", balance, tax );

}
```

This is **illegal**. You cannot re-assign a new value to a symbolic constant.

ITS LITERAL SUBSTITUTION, SO BEWARE OF ERRORS

As shown above, the preprocessor performs literal substitution of symbolic constants. Lets modify the previous program slightly, and introduce an error to highlight a problem.

```
#include

#define TAX_RATE 0.10;

main()

{

float balance;

float tax;


balance = 72.10;

tax = (balance * TAX_RATE )+ 10.02;

printf("The tax on %.2f is %.2f\n", balance, tax );

}
```

In this case, the error that has been introduced is that the *#define* is terminated with a semi-colon. The preprocessor performs the substitution and the offending line (which is flagged as an error by the compiler) looks like

```
tax = (balance * 0.10; )+ 10.02;
```

However, you do not see the output of the preprocessor. If you are using TURBO C, you will only see

```
tax = (balance * TAX_RATE )+ 10.02;
```

flagged as an error, and this actually looks okay (but its not! after substitution takes place).

MAKING PROGRAMS EASY TO MAINTAIN BY USING #define

The whole point of using *#define* in your programs is to make them easier to read and modify. Considering the above programs as examples, what changes would you need to make if the TAX_RATE was changed to 20%.

Obviously, the answer is once, where the *#define* statement which declares the symbolic constant and its value occurs. You would change it to read

```
#define TAX_RATE = 0.20
```

Without the use of symbolic constants, you would hard code the value 0.20 in your program, and this might occur several times (or tens of times).

This would make changes difficult, because you would need to search and replace every occurrence in the program. However, as the programs get larger, **what would happen if you actually used the value 0.20 in a calculation that had nothing to do with the TAX_RATE!**

SUMMARY OF #define

- allow the use of symbolic constants in programs
- in general, symbols are written in uppercase
- are not terminated with a semi-colon
- generally occur at the beginning of the file
- each occurrence of the symbol is replaced by its value
- makes programs readable and easy to maintain

HEADER FILES

Header files contain definitions of functions and variables which can be incorporated into any C program by using the pre-processor *#include* statement. Standard header files are provided with each compiler, and cover a range of areas, string handling, mathematical, data conversion, printing and reading of variables.

To use any of the standard functions, the appropriate header file should be included. This is done at the beginning of the C source file. For example, to use the function *printf()* in a program, the line

```
#include
```

should be at the beginning of the source file, because the definition for *printf()* is found in the file *stdio.h*. All header files have the extension *.h* and generally reside in the */include* subdirectory.

```
#include  
#include "mydecls.h"
```

The use of angle brackets *<>* informs the compiler to search the compilers include directory for the specified file. The use of the double quotes *""* around the filename inform the compiler to search in the current directory for the specified file.

Practice Exercise 1: Defining Variables

1. Declare an integer called sum
2. Declare a character called letter
3. Define a constant called TRUE which has a value of 1
4. Declare a variable called money which can be used to hold currency
5. Declare a variable called arctan which will hold scientific notation values (+e)
6. Declare an integer variable called total and initialise it to zero.
7. Declare a variable called loop, which can hold an integer value.
8. Define a constant called GST with a value of .125

Answers to Practice Exercise 1: Defining Variables

1. Declare an integer called sum

```
int sum;
```

2. Declare a character called letter

```
char letter;
```

3. Define a constant called TRUE which has a value of 1

```
#define TRUE 1
```

4. Declare a variable called money which can be used to hold currency

```
float money;
```

5. Declare a variable called arctan which will hold scientific notation values (+e)

```
double arctan;
```

6. Declare an integer variable called total and initialise it to zero.

```
int total;
```

```
total = 0;
```

7. Declare a variable called loop, which can hold an integer value.

```
int loop;
```

8. Define a constant called GST with a value of .125

```
#define GST 0.125
```

ARITHMETIC OPERATORS

The symbols of the arithmetic operators are:-

Operation	Operator	Comment	Value of Sum before	Value of sum after
Multiply	*	sum = sum * 2;	4	8
Divide	/	sum = sum / 2;	4	2
Addition	+	sum = sum + 2;	4	6
Subtraction	-	sum = sum -2;	4	2
Increment	++	++sum;	4	5
Decrement	--	--sum;	4	3
Modulus	%	sum = sum % 3;	4	1

The following code fragment adds the variables *loop* and *count* together, leaving the result in the variable *sum*

```
sum = loop + count;
```

Note: If the modulus % sign is needed to be displayed as part of a text string, use two, ie %%

```
#include
```



```
main()

{

    int sum = 50;

    float modulus;


    modulus = sum % 10;

    printf("The %% of %d by 10 is %f\n", sum, modulus);

}
```

Sample Program Output

The % of 50 by 10 is 0.000000

CLASS EXERCISE C5

What does the following change do to the printed output of the previous program?

```
printf("The %% of %d by 10 is %.2f\n", sum, modulus);
```

ANSWERS: CLASS EXERCISE C5

```
#include

main()
{
    int sum = 50;
    float modulus;


    modulus = sum % 10;
    printf("The %% of %d by 10 is %.2f\n", sum, modulus);
}
```

The % of 50 by 10 is 0.00

—

Practice Exercise 2: Assignments

1. Assign the value of the variable `number1` to the variable `total`
2. Assign the sum of the two variables `loop_count` and `petrol_cost` to the variable `sum`
3. Divide the variable `total` by the value 10 and leave the result in the variable `discount`
4. Assign the character `W` to the char variable `letter`
5. Assign the result of dividing the integer variable `sum` by 3 into the float variable `costing`. Use type casting to ensure that the remainder is also held by the float variable.

Answers: Practise Exercise 2: Assignments

1. Assign the value of the variable `number1` to the variable `total`

```
total = number1;
```

2. Assign the sum of the two variables `loop_count` and `petrol_cost` to the variable `sum`

```
sum = loop_count + petrol_cost;
```

3. Divide the variable `total` by the value 10 and leave the result in the variable `discount`

```
discount = total / 10;
```

4. Assign the character `W` to the char variable `letter`

```
letter = 'W';
```

5. Assign the result of dividing the integer variable `sum` by 3 into the float variable `costing`. Use type casting to ensure that the remainder is also held by the float variable.

```
costing = (float) sum / 3;
```

PRE/POST INCREMENT/DECREMENT OPERATORS

PRE means do the operation first followed by any assignment operation

,

POST means do the operation after any assignment operation. Consider the following statements

```
++count; /* PRE Increment, means add one to count */
```

```
count++; /* POST Increment, means add one to count */
```

In the above example, because the value of *count* is not assigned to any variable, the effects of the PRE/POST operation are not clearly visible.

Lets examine what happens when we use the operator along with an assignment operation. Consider the following program,

```
#include
```

```
main()
```

```
{
```

```
int count = 0, loop;
```

```
loop = ++count; /* same as count = count + 1; loop = count; */
```

```
printf("loop = %d, count = %d\n", loop, count);
```

```
loop = count++; /* same as loop = count; count = count + 1; */
```

```
printf("loop = %d, count = %d\n", loop, count);
```

```
}
```

Sample Program Output

```
loop = 1, count = 1
```

```
loop = 1; count = 2
```

If the operator precedes (is on the left hand side) of the variable, the operation is performed first, so the statement

```
loop = ++count;
```

really means increment *count* first, then assign the new value of *count* to *loop*.

Which way do you write it?

Where the increment/decrement operation is used to adjust the value of a variable, and is not involved in an assignment operation, which should you use,

```
++loop_count;
```

or

```
loop_count++;
```

The answer is, it really does not matter. It does seem that there is a preference amongst C programmers to use the post form.

Something to watch out for

Whilst we are on the subject, do not get into the habit of using a space(s) between the variable name and the pre/post operator.

```
loop_count ++;
```

Try to be explicit in *binding* the operator tightly by leaving no gap.

GOOD FORM

Perhaps we should say *programming style* or *readability*. The most common complaints we would have about beginning C programmers can be summarized as,

- they have poor layout
- their programs are hard to read

Your programs will be quicker to write and easier to debug if you get into the habit of actually formatting the layout correctly as you write it.

For instance, look at the program below

```
#include

main()

{

int sum,loop,kettle,job;

char Whoknows;


sum=9;

loop=7;

whoKnows='A';

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

}
```

It is our contention that the program is hard to read, and because of this, will be difficult to debug for errors by an inexperienced programmer. It also contains a few deliberate mistakes!

Okay then, lets rewrite the program using good form.

```
#include

main()

{

int sum, loop, kettle = 0, job;

char whoknows;

sum = 9;

loop = 7;

whoknows = 'A';

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

}
```

We have also corrected the mistakes. The major differences are

- the { and } braces directly line up underneath each other
- This allows us to check indent levels and ensure that statements belong to the correct block of code. This becomes vital as programs become more complex
- spaces are inserted for readability
- We as humans write sentences using spaces between words. This helps our comprehension of what we read (if you don't believe me, try reading the following sentence. Wish I had a dollar for every time I made a mistake. The insertion of spaces will also help us identify mistakes quicker.
- good indentation
- Indent levels (tab stops) are clearly used to block statements, here we clearly see and identify functions, and the statements which belong to each { } program body.
- initialization of variables
- The first example prints out the value of *kettle*, a variable that has no initial value. This is corrected in the second example.

• C Programming

• KEYBOARD INPUT

There is a function in C which allows the programmer to accept input from a keyboard. The following program illustrates the use of this function,

```
#include

main()      /* program which introduces keyboard input */
{
    int  number;

    printf("Type in a number \n");
    scanf("%d", &number);
    printf("The number you typed was %d\n", number);
}
```

Sample Program Output

```
Type in a number
23
The number you typed was 23
```

- An integer called *number* is defined. A prompt to enter in a number is then printed using the statement

```
printf("Type in a number \n:");
```

- The *scanf* routine, which accepts the response, has two arguments. The first ("%d") specifies what type of data type is expected (ie char, int, or float). List of formatters for scanf() found here.
- The second argument (&number) specifies the variable into which the typed response will be placed. In this case the response will be placed into the memory location associated with the variable *number*.
- This explains the special significance of the & character (which means the address of).

• Sample program illustrating use of scanf() to read integers, characters and floats

```
#include < stdio.h >

main()
{
    int sum;
    char letter;
```

```

•      float money;
•
•      printf("Please enter an integer value ");
•      scanf("%d", &sum );
•
•      printf("Please enter a character ");
•      /* the leading space before the %c ignores space
characters in the input */
•      scanf(" %c", &letter );
•
•      printf("Please enter a float variable ");
•      scanf("%f", &money );
•
•      printf("\nThe variables you entered were\n");
•      printf("value of sum = %d\n", sum );
•      printf("value of letter = %c\n", letter );
•      printf("value of money = %f\n", money );
•  }
•
•
•
•

```

Sample Program Output

```

•      Please enter an integer value
•      34
•      Please enter a character
•      W
•      Please enter a float variable
•      32.3
•      The variables you entered were
•      value of sum = 34
•      value of letter = W
•      value of money = 32.300000
•
•

```

This program illustrates several important points.

- the c language provides no error checking for user input. The user is expected to enter the correct data type. For instance, if a user entered a character when an integer value was expected, the program may enter an infinite loop or abort abnormally.
- It's up to the programmer to validate data for correct type and range of values.

Practise Exercise 3: printf() and scanf()

1. Use a printf statement to print out the value of the integer variable sum
2. Use a printf statement to print out the text string "Welcome", followed by a newline.

3. Use a printf statement to print out the character variable letter
4. Use a printf statement to print out the float variable discount
5. Use a printf statement to print out the float variable dump using two decimal places
6. Use a scanf statement to read a decimal value from the keyboard, into the integer variable sum
7. Use a scanf statement to read a float variable into the variable discount_rate
8. Use a scanf statement to read a single character from the keyboard into the variable operator. Skip leading blanks, tabs and newline characters.

Answers: Practise Exercise 3: printf() and scanf()

1. Use a printf statement to print out the value of the integer variable sum

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

2. Use a printf statement to print out the text string "Welcome", followed by a newline.

```
printf("Welcome\n");
```

3. Use a printf statement to print out the character variable letter

```
printf("%c", letter);
```

4. Use a printf statement to print out the float variable discount

```
printf("%f", discount);
```

5. Use a printf statement to print out the float variable dump using two decimal places

```
printf("%.2f", dump);
```

6. Use a scanf statement to read a decimal value from the keyboard, into the integer variable sum

```
scanf("%d", &sum);
```

7. Use a scanf statement to read a float variable into the variable discount_rate

```
scanf("%f", &discount_rate);
```

8. Use a scanf statement to read a single character from the keyboard into the variable operator. Skip leading blanks, tabs and newline characters.

```
scanf(" %c", &operator);
```

THE RELATIONAL OPERATORS

These allow the comparison of two or more variables.

Operator	Meaning
==	<i>equal to</i>
!=	<i>not equal</i>
<	<i>less than</i>
<=	<i>less than or equal to</i>
>	<i>greater than</i>
>=	<i>greater than or equal to</i>

In the next few screens, these will be used in *for* loops and *if* statements.

The operator

```
<>
```

may be legal in Pascal, **but is illegal in C.**

ITERATION, FOR LOOPS

The basic format of the for statement is,

```
for( start condition; continue condition; re-evaluation )  
  
program statement;
```

```
/* sample program using a for statement */
```

```
#include
```

```
main() /* Program introduces the for statement, counts to ten */
```

```
{
```

```
int count;
```

```
for( count = 1; count <= 10; count = count + 1 )
```

```
printf("%d ", count );
```

```
printf("\n");
```

```
}
```

Sample Program Output

```
1 2 3 4 5 6 7 8 9 10
```

The program declares an integer variable *count*. The first part of the *for* statement

```
for( count = 1;
```

initialises the value of *count* to 1. The *for* loop continues whilst the condition

```
count <= 10;
```

evaluates as TRUE. As the variable *count* has just been initialised to 1, this condition is TRUE and so the program statement

```
printf("%d ", count );
```

is executed, which prints the value of *count* to the screen, followed by a space character.

Next, the remaining statement of the *for* is executed

```
count = count + 1 );
```

which adds one to the current value of *count*. Control now passes back to the conditional test,

```
count <= 10;
```

which evaluates as true, so the program statement

```
printf("%d ", count );
```

is executed. *Count* is incremented again, the condition re-evaluated etc, until count reaches a value of 11.

When this occurs, the conditional test

```
count <= 10;
```

evaluates as FALSE, and the *for* loop terminates, and program control passes to the statement

```
printf("\n");
```

which prints a newline, and then the program terminates, as there are no more statements left to execute.

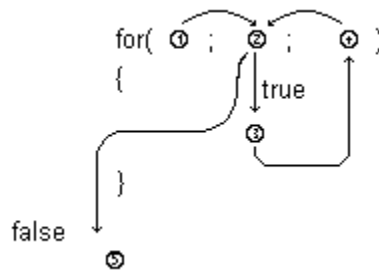
```
/* sample program using a for statement */  
  
#include  
  
main()  
{  
  
    int n, t_number;  
  
    t_number = 0;  
  
    for( n = 1; n <= 200; n = n + 1 )  
  
        t_number = t_number + n;  
  
    printf("The 200th triangular_number is %d\n", t_number);  
  
}
```

Sample Program Output

The 200th triangular_number is 20100

The above program uses a *for* loop to calculate the sum of the numbers from 1 to 200 inclusive (said to be the *triangular number*).

The following diagram shows the order of processing each part of a *for*



An example of using a for loop to print out characters

```
#include
```

```
main()
```

```
{
```

```
char letter;
```

```
for( letter = 'A'; letter <= 'E'; letter = letter + 1 ) {
```

```
printf("%c ", letter);
```

```
}
```

```
}
```

Sample Program Output

```
A B C D E
```

An example of using a for loop to count numbers, using two initialisations

```
#include

main()

{

    int total, loop;

    for( total = 0, loop = 1; loop <= 10; loop = loop + 1 ){

        total = total + loop;

    }

    printf("Total = %d\n", total );

}
```

Sample Program Output

Total = 55

In the above example, the variable *total* is initialised to 0 as the first part of the for loop. The two statements,

```
for( total = 0, loop = 1;
```

are part of the initialisation. This illustrates that more than one statement is allowed, as long as they are separated by **commas**.

Graphical Animation of *for* loop

To demonstrate the operation of the *for* statement, let's consider a series of animations.

The code we will be using is

```
#include

main() {

    int x, y, z;

    x = 2;

    y = 2;

    z = 3;

    for( x = 1; x <= 6; x = x + 1 ) {

        printf("%d", y );

        y = y + 1;

    }

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

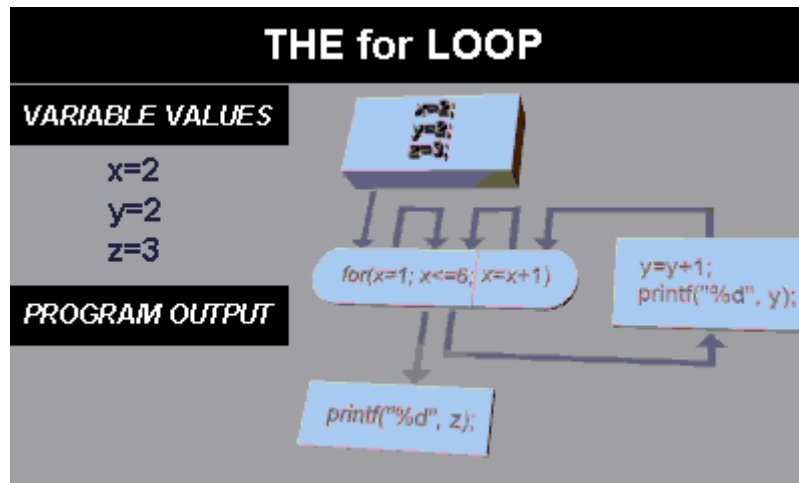
}
```

Sample Program Output

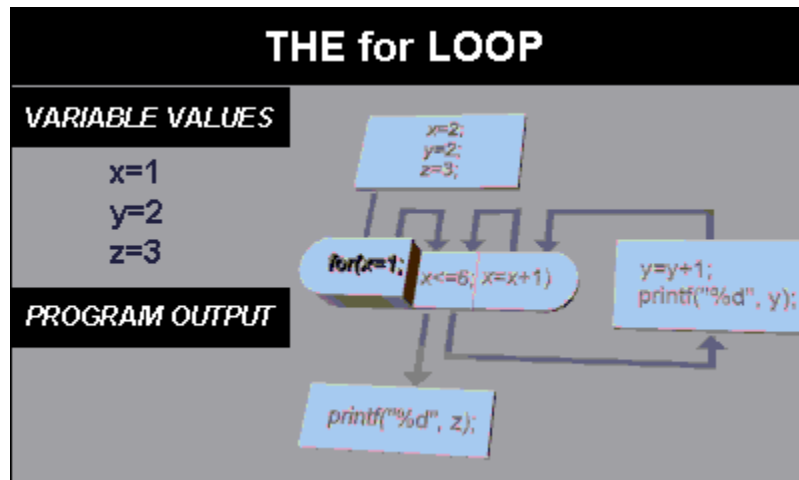
2 3 4 5 6 7

3

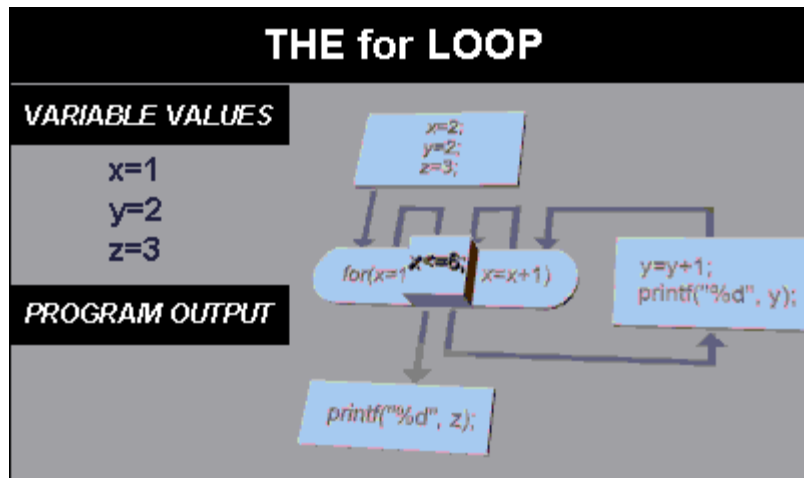
The following diagram shows the initial state of the program, after the initialization of the variables *x*, *y*, and *z*.



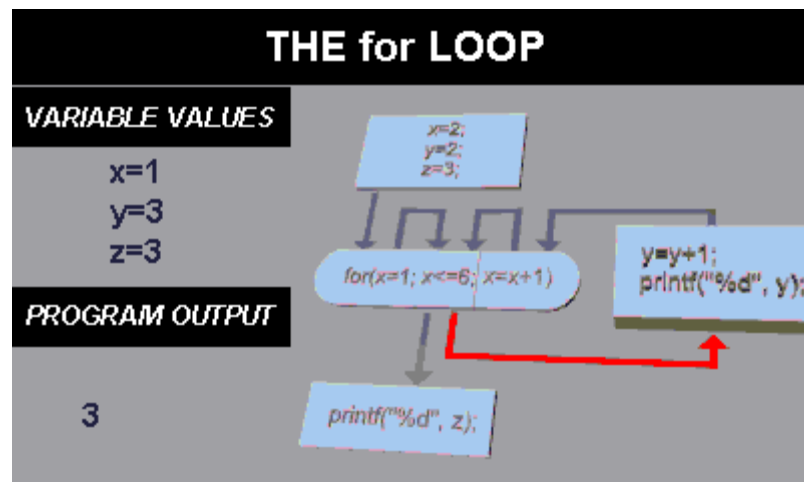
On entry to the *for* statement, the first expression is executed, which in our example assigns the value 1 to *x*. This can be seen in the graphic shown below (Note: see the Variable Values: section)



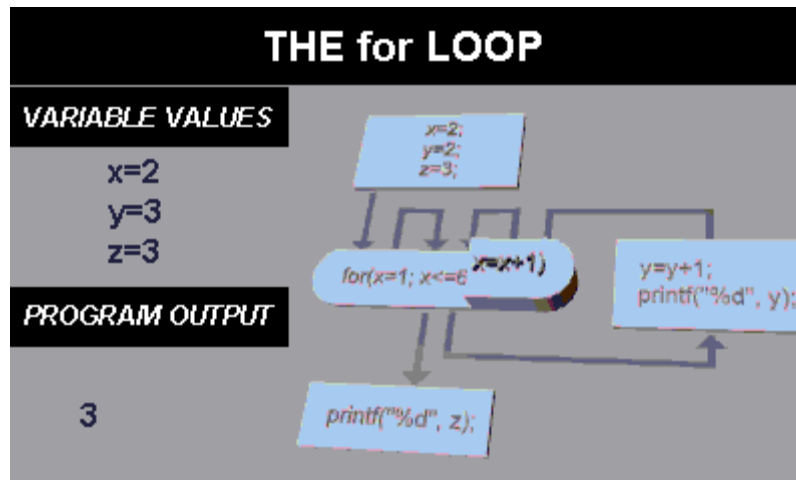
The next part of the *for* is executed, which tests the value of the loop variable *x* against the constant 6.



It can be seen from the variable window that *x* has a current value of 1, so the test is successful, and program flow branches to execute the statements of the *for body*, which prints out the value of *y*, then adds 1 to *y*. You can see the program output and the state of the variables shown in the graphic below.

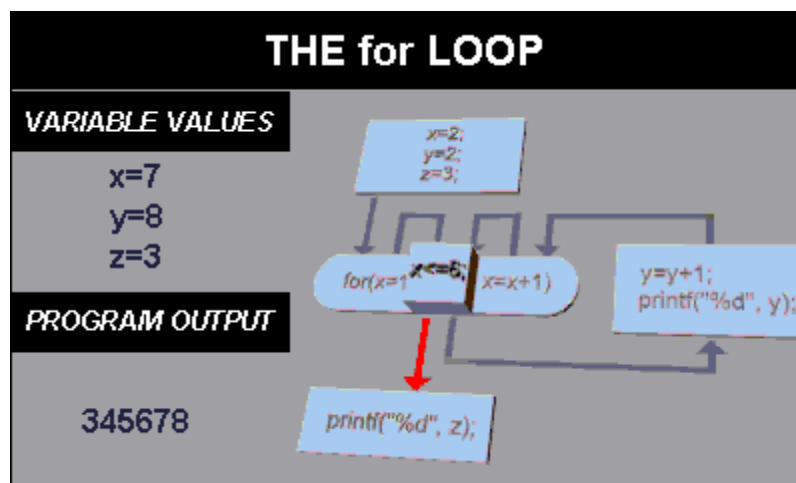


After executing the statements of the *for body*, execution returns to the last part of the *for* statement. Here, the value of *x* is incremented by 1. This is seen by the value of *x* changing to 2.



Next, the condition of the *for* variable is tested again. It continues because the value of it (2) is less than 6, so the body of the loop is executed again.

Execution continues till the value of *x* reaches 7. Let's now jump ahead in the animation to see this. Here, the condition test will fail, and the *for* statement finishes, passing control to the statement which follows.



EXERCISE C6

Rewrite the previous program by calculating the 200th triangular number, and make the program shorter (if possible).

CLASS EXERCISE C7

What is the difference between the two statements,

```
a == 2
```

```
a = 2
```

CLASS EXERCISE C8

Change the printf line of the above program to the following,

```
printf(" %2d %2d\n",n,t_number);
```

What does the inclusion of the 2 in the %d statements achieve?

EXERCISE C9

Create a C program which calculates the triangular number of the users request, read from the keyboard using **scanf()**. A triangular number is the sum of the preceding numbers, so the triangular number 7 has a value of

$7 + 6 + 5 + 4 + 3 + 2 + 1$

Answer: EXERCISE C6

```
#include
```

```
main()
```

```
{
```

```
int n = 1, t_number = 0;
```

```
for( ; n <= 200; n++ )
```

```
t_number = t_number + n;
```

```
printf("The 200th triangular_number is %d\n", t_number);  
  
}
```

Answer: CLASS EXERCISE C7

a == 2 equality test

a = 2 assignment

Answer: CLASS EXERCISE C8

The inclusion of the 2 in the %d statements achieves a field width of two places, and prints a leading 0 where the value is less than 10.

Answer: EXERCISE C9

```
#include  
  
main()  
{  
  
int n = 1, t_number = 0, input;  
  
printf("Enter a number\n");  
  
scanf("%d", &input);
```

```
for( ; n <= input; n++ )  
  
    t_number = t_number + n;  
  
printf("The triangular_number of %d is %d\n", input, t_number);  
  
}
```

Practice Exercise 4: for loops

1. Write a for loop to print out the values 1 to 10 on separate lines.
2. Write a for loop which will produce the following output (hint: use two nested for loops)

```
1  
22  
333  
4444  
55555
```

3. Write a for loop which sums all values between 10 and 100 into a variable called *total*. Assume that *total* has NOT been initialized to zero.
4. Write a for loop to print out the character set from A-Z.

PRACTISE EXERCISE 4

for loops

1. Write a for loop to print out the values 1 to 10 on separate lines.

```
for( loop = 1; loop <= 10; loop = loop + 1 )  
    printf("%d\n", loop) ;
```

2. Write a for loop which will produce the following output (hint: use two nested for loops)

```
1  
22  
333  
4444  
55555
```

```
for( loop = 1; loop <= 5; loop = loop + 1 )
{
    for( count = 1; count <= loop; count = count + 1
)
        printf("%d", loop );
    printf("\n");
}
```

3. Write a for loop which sums all values between 10 and 100 into a variable called *total*. Assume that *total* has NOT been initialized to zero.

```
for( loop = 10, total = 0; loop <= 100; loop = loop + 1 )
    total = total + loop;
```

4. Write a for loop to print out the character set from A-Z.

```
for( ch = 'A'; ch <= 'Z'; ch = ch + 1 )
    printf("%c", ch );
printf("\n");
```

THE WHILE STATEMENT

The *while* provides a mechanism for repeating C statements whilst a condition is true. Its format is,

```
while( condition )
    program statement;
```

Somewhere within the body of the *while* loop a statement must alter the value of the condition to allow the loop to finish.

```
/* Sample program including while */
#include

main()
{
    int loop = 0;

    while( loop <= 10 ) {
        printf("%d\n", loop);
        ++loop;
    }
}
```

Sample Program Output

```
0
1
...
10
```

The above program uses a *while* loop to repeat the statements

```
printf("%d\n", loop);
++loop;
```

whilst the value of the variable *loop* is less than or equal to 10.

Note how the variable upon which the *while* is dependent is initialised prior to the *while* statement (in this case the previous line), and also that the value of the variable is altered within the loop, so that eventually the conditional test will succeed and the *while* loop will terminate.

This program is functionally equivalent to the earlier *for* program which counted to ten.

THE DO WHILE STATEMENT

The *do { } while* statement allows a loop to continue whilst a condition evaluates as TRUE (non-zero). The loop is executed as least once.

```
/* Demonstration of DO...WHILE */

#include

main()

{

    int value, r_digit;

    printf("Enter the number to be reversed.\n");

    scanf("%d", &value);
```



```
do {  
  
    r_digit = value % 10;  
  
    printf("%d", r_digit);  
  
    value = value / 10;  
  
} while( value != 0 );  
  
printf("\n");  
  
}
```

The above program reverses a number that is entered by the user. It does this by using the modulus % operator to extract the right most digit into the variable *r_digit*. The original number is then divided by 10, and the operation repeated whilst the number is not equal to 0.

It is our contention that this programming construct is improper and should be avoided. It has potential problems, and you should be aware of these.

One such problem is deemed to be *lack of control*. Considering the above program code portion,

```
do {  
  
    r_digit = value % 10;  
  
    printf("%d", r_digit);  
  
    value = value / 10;  
  
} while( value != 0 );
```

there is **NO** choice whether to execute the loop. Entry to the loop is automatic, as you only get a choice to continue.

Another problem is that the loop is always executed at least once. This is a by-product of the lack of control. This means its possible to enter a *do { } while* loop with invalid data.

Beginner programmers can easily get into a whole heap of trouble, so our advice is to avoid its use. This is the only time that you will encounter it in this course. Its easy to avoid the use of this construct by replacing it with the following algorithms,

```
initialise loop control variable

while( loop control variable is valid ) {

process data

adjust control variable if necessary

}
```

Okay, lets now rewrite the above example to remove the *do {} while* construct.

```
/* rewritten code to remove construct */

#include

main()

{

int value, r_digit;

value = 0;

while( value <= 0 ) {

printf("Enter the number to be reversed.\n");

scanf("%d", &value);

if( value <= 0 )

printf("The number must be positive\n");

}
```

```
while( value != 0 )  
  
{  
  
    r_digit = value % 10;  
  
    printf("%d", r_digit);  
  
    value = value / 10;  
  
}  
  
printf("\n");  
  
}
```

Sample Program Output

Enter the number to be reversed.

-43

The number must be positive

Enter the number to be reversed.

423

324

SELECTION (IF STATEMENTS)

The *if* statements allows branching (decision making) depending upon the value or state of variables. This allows statements to be executed or skipped, depending upon decisions. The basic format is,

```
if( expression )  
    program statement;
```

Example;

```
if( students < 65 )  
    ++student_count;
```

In the above example, the variable *student_count* is incremented by one only if the value of the integer variable *students* is less than 65.

The following program uses an *if* statement to validate the users input to be in the range 1-10.

```
#include  
  
main()  
{  
    int number;  
    int valid = 0;  
  
    while( valid == 0 ) {  
        printf("Enter a number between 1 and 10 --  
>");  
  
        scanf("%d", &number);  
        /* assume number is valid */  
        valid = 1;  
        if( number < 1 ) {  
            printf("Number is below 1. Please  
re-enter\n");  
            valid = 0;  
        }  
        if( number > 10 ) {  
            printf("Number is above 10. Please  
re-enter\n");  
            valid = 0;  
        }  
    }  
    printf("The number is %d\n", number );  
}
```

Sample Program Output

```
Enter a number between 1 and 10 --> -78  
Number is below 1. Please re-enter  
Enter a number between 1 and 10 --> 4  
The number is 4
```

EXERCISE C10

Write a C program that allows the user to enter in 5 grades, ie, marks between 0 -

100. The program must calculate the average mark, and state the number of marks less than 65.

Consider the following program which determines whether a character entered from the keyboard is within the range A to Z.

```
#include

main()
{
    char letter;

    printf("Enter a character -->");
    scanf(" %c", &letter );

    if( letter >= 'A' ) {
        if( letter <= 'Z' )
            printf("The character is within A
to Z\n");
    }
}
```

Sample Program Output

```
Enter a character --> C
The character is within A to Z
```

The program does not print any output if the character entered is not within the range A to Z. This can be addressed on the following pages with the *if else* construct.

Please note use of the leading space in the statement (before %c)

```
scanf(" %c", &letter );
```

This enables the skipping of leading TABS, Spaces, (collectively called whitespaces) and the ENTER KEY. If the leading space was not used, then the first entered character would be used, and *scanf* would not ignore the whitespace characters.

COMPARING float types FOR EQUALITY

Because of the way in which float types are stored, it makes it very difficult to

compare float types for equality. Avoid trying to compare float variables for equality, or you may encounter unpredictable results.

if else

The general format for these are,

```
if( condition 1 )
    statement1;
else if( condition 2 )
    statement2;
else if( condition 3 )
    statement3;
else
    statement4;
```

The *else* clause allows action to be taken where the condition evaluates as false (zero).

The following program uses an *if else* statement to validate the users input to be in the range 1-10.

```
#include

main()
{
    int number;
    int valid = 0;

    while( valid == 0 ) {
        printf("Enter a number between 1 and 10 --
>");

        scanf("%d", &number);
        if( number < 1 ) {
            printf("Number is below 1. Please
re-enter\n");

            valid = 0;
        }
        else if( number > 10 ) {
            printf("Number is above 10. Please
re-enter\n");

            valid = 0;
        }
        else
            valid = 1;
    }
    printf("The number is %d\n", number );
}
```

Sample Program Output

Enter a number between 1 and 10 --> 12

```
Number is above 10. Please re-enter
Enter a number between 1 and 10 --> 5
The number is 5
```

This program is slightly different from the previous example in that an *else* clause is used to set the variable *valid* to 1. In this program, the logic should be easier to follow.

```
/* Illustrates 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
```

The above program acts as a simple calculator.

Practice Exercise 5: while loops and if else

1. Use a while loop to print the integer values 1 to 10 on the screen

```
12345678910
```

2. Use a nested while loop to reproduce the following output

```
1
22
333
4444
55555
```

3. Use an if statement to compare the value of an integer called sum against the value 65, and if it is less, print the text string "Sorry, try again".

4. If total is equal to the variable good_guess, print the value of total, else print the value of good_guess.

Answers: Practice Exercise 5: while loops and if else

1. Use a while loop to print the integer values 1 to 10 on the screen

```
12345678910
```

```
#include
```

```
main()
```

```
{
```

```
int loop;
```

```
loop = 1;
```

```
while( loop <= 10 ) {
```



```
printf("%d", loop);  
  
loop++;  
  
}  
  
printf("\n");  
  
}
```

2. Use a nested while loop to reproduce the following output

```
1  
  
22  
  
333  
  
4444  
  
55555
```

```
#include  
  
main()  
{  
  
    int loop;  
  
    int count;  
  
    loop = 1;  
  
    while( loop <= 5 ) {  
  
        count = 1;  
  
        while( count <= loop ) {  
  
            printf("%d", count);
```

```
count++;  
  
}  
  
loop++;  
  
}  
  
printf("\n");  
  
}
```

3. Use an if statement to compare the value of an integer called sum against the value 65, and if it is less, print the text string "Sorry, try again".

```
if( sum < 65 )  
  
printf("Sorry, try again.\n");
```

4. If total is equal to the variable good_guess, print the value of total, else print the value of good_guess.

```
if( total == good_guess )  
  
printf("%d\n", total );  
  
else  
  
printf("%d\n", good_guess );
```

COMPOUND RELATIONALS (AND, NOT, OR, EOR)

Combining more than one condition

These allow the testing of more than one condition as part of selection statements. The symbols are

LOGICAL AND &&

Logical and requires all conditions to evaluate as TRUE (non-zero).

LOGICAL OR ||

Logical or will be executed if any ONE of the conditions is TRUE (non-zero).

LOGICAL NOT !

logical not negates (changes from TRUE to FALSE, vsvs) a condition.

LOGICAL EOR ^

Logical eor will be excuted if either condition is TRUE, but NOT if they are all true.

The following program uses an *if* statement with logical OR to validate the users input to be in the range 1-10.

```
#include

main()
{
    int number;
    int valid = 0;

    while( valid == 0 ) {
        printf("Enter a number between 1 and 10 --
>");

        scanf("%d", &number);
        if( (number < 1 ) || (number > 10) ){
            printf("Number is outside range 1-
10. Please re-enter\n");
            valid = 0;
        }
        else
            valid = 1;
    }
    printf("The number is %d\n", number );
}
```

Sample Program Output

```
Enter a number between 1 and 10 --> 56
Number is outside range 1-10. Please re-enter
```

```
Enter a number between 1 and 10 --> 6
The number is 6
```

This program is slightly different from the previous example in that a LOGICAL OR eliminates one of the *else* clauses.

COMPOUND RELATIONALS (AND, NOT, OR, EOR)

NEGATION

```
#include

main()
{
    int flag = 0;
    if( ! flag ) {
        printf("The flag is not set.\n");
        flag = ! flag;
    }
    printf("The value of flag is %d\n", flag);
}
```

Sample Program Output

```
The flag is not set.
The value of flag is 1
```

The program tests to see if *flag* is not (!) set; equal to zero. It then prints the appropriate message, changes the state of *flag*; *flag* becomes equal to not *flag*; equal to 1. Finally the value of *flag* is printed.

COMPOUND RELATIONALS (AND, NOT, OR, EOR)

Range checking using Compound Relational

Consider where a value is to be inputted from the user, and checked for validity to be within a certain range, lets say between the integer values 1 and 100.

```
#include

main()
{
    int number;
    int valid = 0;

    while( valid == 0 ) {
        printf("Enter a number between 1 and
100");
        scanf("%d", &number );
    }
}
```

```
        if( (number < 1) || (number > 100) )  
            printf("Number is outside legal  
range\n");  
        else  
            valid = 1;  
    }  
    printf("Number is %d\n", number );  
}
```

Sample Program Output

```
Enter a number between 1 and 100  
203  
Number is outside legal range  
Enter a number between 1 and 100  
-2  
Number is outside legal range  
Enter a number between 1 and 100  
37  
Number is 37
```

The program uses *valid*, as a flag to indicate whether the inputted data is within the required range of allowable values. The while loop continues whilst *valid* is 0.

The statement

```
if( (number < 1) || (number > 100) )
```

checks to see if the number entered by the user is within the valid range, and if so, will set the value of *valid* to 1, allowing the while loop to exit.

Now consider writing a program which validates a character to be within the range A-Z, in other words *alphabetic*.

```
#include  
  
main()  
{  
    char ch;  
    int valid = 0;  
  
    while( valid == 0 ) {  
        printf("Enter a character A-Z");  
        scanf(" %c", &ch );  
        if( (ch >= 'A') && (ch <= 'Z') )  
            valid = 1;  
        else
```

```
printf("Character is outside legal  
range\n");  
}  
printf("Character is %c\n", ch );  
}
```

Sample Program Output

```
Enter a character A-Z  
a  
Character is outside legal range  
Enter a character A-Z  
0  
Character is outside legal range  
Enter a character A-Z  
R  
Character is R
```

In this instance, the AND is used because we want validity between a range, that is all values between a low and high limit. In the previous case, we used an OR statement to test to see if it was outside or below the lower limit or above the higher limit.

witch() case:

The *switch case* statement is a better way of writing a program when a series of *if else*s occurs. The general format for this is,

```
switch ( expression ) {  
    case value1:  
        program statement;  
        program statement;  
        .....  
        break;  
    case valuen:  
        program statement;  
        .....  
        break;  
    default:  
        .....  
        .....  
        break;  
}
```

The keyword *break* must be included at the end of each case statement. The default clause is optional, and is executed if the cases are not met. The right brace at the end signifies the end of the case selections.

Rules for switch statements

values for 'case' must be integer or character constants
the order of the 'case' statements is unimportant
the default clause may occur first (convention places it
last)
you cannot use expressions or ranges

```
#include

main()
{
    int menu, numb1, numb2, total;

    printf("enter in two numbers -->");
    scanf("%d %d", &numb1, &numb2 );
    printf("enter in choice\n");
    printf("1=addition\n");
    printf("2=subtraction\n");
    scanf("%d", &menu );
    switch( menu ) {
        case 1: total = numb1 + numb2; break;
        case 2: total = numb1 - numb2; break;
        default: printf("Invalid option
selected\n");
    }
    if( menu == 1 )
        printf("%d plus %d is %d\n", numb1, numb2,
total );
    else if( menu == 2 )
        printf("%d minus %d is %d\n", numb1,
numb2, total );
    }
```

Sample Program Output

```
enter in two numbers --> 37 23
enter in choice
1=addition
2=subtraction
2
37 minus 23 is 14
```

The above program uses a *switch* statement to validate and select upon the users input choice, simulating a simple menu of choices.

EXERCISE C11

Rewrite the previous program, which accepted two numbers and an operator, using the *switch case* statement

THE switch case STATEMENT

EXERCISE C11

Rewrite the previous program, which accepted two numbers and an operator, using the *switch case* statement.

```

/* Illustrates 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");
}

```

Solution

```

/* Illustrates switch */
#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);

switch( operator ) {
    case '*' : result = number1 * number2;
break;
    case '/' : result = number1 / number2;
break;
    case '+' : result = number1 + number2;
break;
    case '-' : result = number1 - number2;
break;
    default : invalid_operator = 1;
}
switch( invalid_operator ) {
    case 1 : printf("Invalid operator.\n");
break;
    default : printf("%f %c %f is %f\n",
number1, operator, number2, result );
}
}
```

Practice Exercise 6

Compound Relational and switch

1. if sum is equal to 10 and total is less than 20, print the text string "incorrect."
2. if flag is 1 or letter is not an 'X', then assign the value 0 to exit_flag, else set exit_flag to 1.
3. rewrite the following statements using a switch statement

```
if( letter == 'X' )
    sum = 0;
else if ( letter == 'Z' )
    valid_flag = 1;
else if( letter == 'A' )
    sum = 1;
else
    printf("Unknown letter -->%c\n", letter );
```

Answers: Practice Exercise 6

Compound Relational and switch

1. if sum is equal to 10 and total is less than 20, print the text string "incorrect."

```
if( (sum == 10) && (total < 20) )  
    printf("incorrect.\n");
```

2. if flag is 1 or letter is not an 'X', then assign the value 0 to exit_flag, else set exit_flag to 1.

```
if( (flag == 1) || (letter != 'X') )  
    exit_flag = 0;  
else  
    exit_flag = 1;
```

3. rewrite the following statements using a switch statement

```
if( letter == 'X' )  
    sum = 0;  
else if ( letter == 'Z' )  
    valid_flag = 1;  
else if( letter == 'A' )  
    sum = 1;  
else  
    printf("Unknown letter -->%c\n", letter );  
  
switch( letter ) {  
    case 'X' : sum = 0; break;  
    case 'Z' : valid_flag = 1; break;  
    case 'A' : sum = 1; break;  
    default  : printf("Unknown letter -->%c\n", letter  
);  
}
```

ACCEPTING SINGLE CHARACTERS FROM THE KEYBOARD

getchar

The following program illustrates this,

```
#include  
  
main()  
{  
    int i;  
    int ch;  
  
    for( i = 1; i<= 5; ++i ) {  
        ch = getchar();  
        putchar(ch);  
    }
```

```
}  
}
```

Sample Program Output

```
AACCddEEtt
```

The program reads five characters (one for each iteration of the for loop) from the keyboard. Note that *getchar()* gets a single character from the keyboard, and *putchar()* writes a single character (in this case, *ch*) to the console screen.

The file `ctype.h` provides routines for manipulating characters.

BUILT IN FUNCTIONS FOR STRING HANDLING

string.h

You may want to look at the section on arrays first!. The following macros are built into the file `string.h`

<i>strcat</i>	<i>Appends a string</i>
<i>strchr</i>	<i>Finds first occurrence of a given character</i>
<i>strcmp</i>	<i>Compares two strings</i>
<i>strcmpi</i>	<i>Compares two strings, non-case sensitive</i>
<i>strcpy</i>	<i>Copies one string to another</i>
<i>strlen</i>	<i>Finds length of a string</i>
<i>strlwr</i>	<i>Converts a string to lowercase</i>
<i>strncat</i>	<i>Appends n characters of string</i>
<i>strncmp</i>	<i>Compares n characters of two strings</i>
<i>strncpy</i>	<i>Copies n characters of one string to another</i>
<i>strnset</i>	<i>Sets n characters of string to a given character</i>
<i>strrchr</i>	<i>Finds last occurrence of given character in string</i>
<i>strrev</i>	<i>Reverses string</i>
<i>strset</i>	<i>Sets all characters of string to a given character</i>
<i>strspn</i>	<i>Finds first substring from given character set in string</i>
<i>strupr</i>	<i>Converts string to uppercase</i>

To convert a string to uppercase

```
#include  
#include  
  
main()  
{  
    char name[80]; /* declare an array of characters  
0-79 */
```

```
printf("Enter in a name in lowercase\n");
scanf( "%s", name );
strupr( name );
printf("The name is uppercase is %s", name );
}
```

Sample Program Output

```
Enter in a name in lowercase
samuel
The name in uppercase is SAMUEL
```

BUILT IN FUNCTIONS FOR CHARACTER HANDLING

The following character handling functions are defined in ctype.h

<i>isalnum</i>	<i>Tests for alphanumeric character</i>
<i>isalpha</i>	<i>Tests for alphabetic character</i>
<i>isascii</i>	<i>Tests for ASCII character</i>
<i>iscntrl</i>	<i>Tests for control character</i>
<i>isdigit</i>	<i>Tests for 0 to 9</i>
<i>isgraph</i>	<i>Tests for printable character</i>
<i>islower</i>	<i>Tests for lowercase</i>
<i>isprint</i>	<i>Tests for printable character</i>
<i>ispunct</i>	<i>Tests for punctuation character</i>
<i>isspace</i>	<i>Tests for space character</i>
<i>isupper</i>	<i>Tests for uppercase character</i>
<i>isxdigit</i>	<i>Tests for hexadecimal</i>
<i>toascii</i>	<i>Converts character to ascii code</i>
<i>tolower</i>	<i>Converts character to lowercase</i>
<i>toupper</i>	<i>Converts character to uppercase</i>

To convert a string array to uppercase a character at a time using *toupper()*

```
#include
#include
main()
{
    char name[80];
    int loop;

    printf("Enter in a name in lowercase\n");
    scanf( "%s", name );
    for( loop = 0; name[loop] != 0; loop++ )
        name[loop] = toupper( name[loop] );

    printf("The name is uppercase is %s", name );
}
```

```
}
```

Sample Program Output

```
Enter in a name in lowercase
samuel
The name in upercase is SAMUEL
```

Validation Of User Input In C

Basic Rules

- Don't pass invalid data onwards.
- Validate data at input time.
- Always give the user meaningful feedback
- Tell the user what you expect to read as input

```
/* example one, a simple continue statement */
#include
#include

main()
{
    int    valid_input;    /* when 1, data is valid and loop
is exited */
    char    user_input;    /* handles user input, single
character menu choice */

    valid_input = 0;
    while( valid_input == 0 ) {
        printf("Continue (Y/N)?\n");
        scanf(" %c", &user_input );
        user_input = toupper( user_input );
        if((user_input == 'Y') || (user_input == 'N') )
            valid_input = 1;
        else printf("\007Error: Invalid choice\n");
    }
}
```

Sample Program Output

```
Continue (Y/N)?
b
Error: Invalid Choice
Continue (Y/N)?
N
```

```
/* example two, getting and validating choices */
```

```
#include
#include

main()
{
    int      exit_flag = 0, valid_choice;
    char      menu_choice;

    while( exit_flag == 0 ) {
        valid_choice = 0;
        while( valid_choice == 0 ) {
            printf("\nC = Copy File\nE = Exit\nM =
Move File\n");
            printf("Enter choice:\n");
            scanf(" %c", &menu_choice );
            if((menu_choice=='C') ||
(menu_choice=='E') || (menu_choice=='M'))
                valid_choice = 1;
            else
                printf("\007Error. Invalid menu
choice selected.\n");
        }
        switch( menu_choice ) {
            case 'C' : .....();
break;

            case 'E' : exit_flag = 1; break;
            case 'M' : .....(); break;
            default : printf("Error--- Should not
occur.\n"); break;
        }
    }
}
```

Sample Program Output

```
C = Copy File
E = Exit
M = Move File
Enter choice:
X
Error. Invalid menu choice selected.
C = Copy File
E = Exit
M = Move File
Enter choice:
E
```

Other validation examples

THE CONDITIONAL EXPRESSION OPERATOR

This conditional expression operator takes THREE operators. The two symbols used to denote this operator are the ? and the :. The first operand is placed before the ?, the second operand between the ? and the :, and the third after the :. The general format is,

condition ? expression1 : expression2

If the result of condition is TRUE (non-zero), expression1 is evaluated and the result of the evaluation becomes the result of the operation. If the condition is FALSE (zero), then expression2 is evaluated and its result becomes the result of the operation. An example will help,

*s = (x < 0) ? -1 : x * x;*

If x is less than zero then s = -1

*If x is greater than zero then s = x * x*

Example program illustrating conditional expression operator

```
#include
```

```
main()
```

```
{
```

```
int input;
```

```
printf("I will tell you if the number is positive, negative or  
zero!\n");
```

```
printf("please enter your number now-->");
```

```
scanf("%d", &input );
```

```
(input < 0) ? printf("negative\n") : ((input > 0) ?  
printf("positive\n") : printf("zero\n"));
```

```
}
```

Sample Program Output

I will tell you if the number is positive, negative or zero!

please enter your number now---> 32

positive

CLASS EXERCISE C12

Evaluate the following expression, where a=4, b=5

```
least_value = ( a < b ) ? a : b;
```

Answers: CLASS EXERCISE C12

Evaluate the following expression, where a=4, b=5

```
least_value = ( a < b ) ? a : b;
```

least_value = 4

ARRAYS

Little Boxes on the hillside

Arrays are a data structure which hold multiple variables of the same data type.

Consider the case where a programmer needs to keep track of a number of people within an organisation. So far, our initial attempt will be to create a specific variable for each user. This might look like,

```
int name1 = 101;
```



```
int name2 = 232;
```

```
int name3 = 231;
```

It becomes increasingly more difficult to keep track of this as the number of variables increase. Arrays offer a solution to this problem.

An array is a multi-element box, a bit like a filing cabinet, and uses an indexing system to find each variable stored within it. In C, indexing starts at **zero**.

Arrays, like other variables in C, must be declared before they can be used.

The replacement of the above example using arrays looks like,

```
int names[4];
```

```
names[0] = 101;
```

```
names[1] = 232;
```

```
names[2] = 231;
```

```
names[3] = 0;
```

We created an array called *names*, which has space for four integer variables. You may also see that we stored 0 in the last space of the array. This is a common technique used by C programmers to signify the end of an array.

Arrays have the following syntax, using square brackets to access each indexed value (called an **element**).

```
x[i]
```

so that *x[5]* refers to the sixth element in an array called *x*. In C, array elements start with 0. Assigning values to array elements is done by,

```
x[10] = g;
```

and assigning array elements to a variable is done by,

```
g = x[10];
```

In the following example, a character based array named *word* is declared, and each element is assigned a character. The last element is filled with a zero value, to signify the end of the character string (in C, there is no string type, so character based arrays are used to hold strings). A printf statement is then used to print out all elements of the array.

```
/* Introducing array's, 2 */

#include

main()

{

char word[20];

word[0] = 'H';

word[1] = 'e';

word[2] = 'l';

word[3] = 'l';

word[4] = 'o';

word[5] = 0;

printf("The contents of word[] is -->%s\n", word );

}
```

Sample Program Output

The contents of word[] is Hello

DECLARING ARRAYS

Arrays may consist of any of the valid data types. Arrays are declared along with all other variables in the declaration section of the program.

```
/* Introducing array's */
#include

main()
{
    int    numbers[100];
    float  averages[20];

    numbers[2] = 10;
    --numbers[2];
    printf("The 3rd element of array numbers is %d\n",
numbers[2]);
}
```

Sample Program Output

The 3rd element of array numbers is 9

The above program declares two arrays, assigns 10 to the value of the 3rd element of array *numbers*, decrements this value (*--numbers[2]*), and finally prints the value. The number of elements that each array is to have is included inside the square brackets.

ASSIGNING INITIAL VALUES TO ARRAYS

The declaration is preceded by the word *static*. The initial values are enclosed in braces, eg,

```
#include
main()
{
    int x;
    static int  values[] = { 1,2,3,4,5,6,7,8,9 };
    static char word[] = { 'H','e','l','l','o' };
    for( x = 0; x < 9; ++x )
```

```
printf("Values [%d] is %d\n", x,  
values[x]);  
}
```

Sample Program Output

```
Values[0] is 1  
Values[1] is 2  
....  
Values[8] is 9
```

The previous program declares two arrays, *values* and *word*. Note that inside the squarebrackets there is no variable to indicate how big the array is to be. In this case, C initializes the array to the number of elements that appear within the initialize braces. So *values* consist of 9 elements (numbered 0 to 8) and the char array *word* has 5 elements.

The following program shows how to initialise all the elements of an integer based array to the value 10, using a for loop to cycle through each element in turn.

```
#include  
main()  
{  
    int count;  
    int values[100];  
    for( count = 0; count < 100; count++ )  
        values[count] = 10;  
}
```

MULTI DIMENSIONED ARRAYS

Multi-dimensioned arrays have two or more index values which specify the element in the array.

```
multi[i][j]
```

In the above example, the first index value *i* specifies a row index, whilst *j* specifies a column index.

Declaration and calculations

```
int m1[10][10];

static int m2[2][2] = { {0,1}, {2,3} };

sum = m1[i][j] + m2[k][l];
```

NOTE the strange way that the initial values have been assigned to the two-dimensional array *m2*. Inside the braces are,

```
{ 0, 1 },
{ 2, 3 }
```

Remember that arrays are split up into row and columns. The first is the row, the second is the column. Looking at the initial values assigned to *m2*, they are,

```
m2[0][0] = 0
m2[0][1] = 1
m2[1][0] = 2
m2[1][1] = 3
```

EXERCISE C13

Given a two dimensional array, write a program that totals all elements, printing the total.

CLASS EXERCISE C14

What value is assigned to the elements which are not assigned initialised.

EXERCISE C13

Given a two dimensional array write a program that totals all elements printing the total.

```
#include

main()

{

static int m[][] = { {10,5,-3}, {9, 0, 0}, {32,20,1}, {0,0,8} };

int row, column, sum;

sum = 0;

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

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

sum = sum + m[row][column];

printf("The total is %d\n", sum );

}
```

CLASS EXERCISE C14

They get initialised to **ZERO**.

CHARACTER ARRAYS [STRINGS]

Consider the following program,

```
#include

main()
```

```
{  
  
static char name1[] = {'H','e','l','l','o'};  
  
static char name2[] = "Hello";  
  
printf("%s\n", name1);  
  
printf("%s\n", name2);  
  
}
```

Sample Program Output

Helloxghifghjkloqw30-=kl`'

Hello

The difference between the two arrays is that *name2* has a null placed at the end of the string, ie, in *name2*[5], whilst *name1* has not. This can often result in garbage characters being printed on the end. To insert a null at the end of the *name1* array, the initialization can be changed to,

```
static char name1[] = {'H','e','l','l','o','\0'};
```

Consider the following program, which initialises the contents of the character based array *word* during the program, using the function *strcpy*, which necessitates using the include file *string.h*

```
#include
```

```
#include
```

```
main()
```

```
{
```

```
char word[20];

strcpy( word, "hi there." );

printf("%s\n", word );

}
```

Sample Program Output

hi there.

SOME VARIATIONS IN DECLARING ARRAYS

```
int numbers[10];

static int numbers[10] = { 34, 27, 16 };

static int numbers[] = { 2, -3, 45, 79, -14, 5, 9, 28, -
1, 0 };

static char text[] = "Welcome to New Zealand.";

static float radix[12] = { 134.362, 1913.248 };

double radians[1000];
```

READING CHARACTER STRINGS FROM THE KEYBOARD

Character based arrays are often referred to in C as strings. C does not support a string type, so character based arrays are used in place of strings. The %s modifier to *printf()* and *scanf()* is used to handle character based arrays. This assumes that a 0 or NULL value is stored in the last element of the array. Consider the following, which reads a string of characters (excluding spaces) from the keyboard.

```
char string[18];
scanf("%s", string);
```

NOTE that the & character does not need to precede the variable name when the formatter %s is used! If the users response was


```
then      Hello
          string[0] = 'H'
          string[1] = 'e'
          ....
          string[4] = 'o'
          string[5] = '\0'
```

Note how the enterkey is not taken by *scanf()* and the text string is terminated by a NULL character '\0' after the last character stored in the array.

Practice Exercise 7: Arrays

2. Assign the character value 'Z' to the fourth element of the letters array
3. Use a for loop to total the contents of an integer array called numbers which has five elements. Store the result in an integer called total.
4. Declare a multidimensioned array of floats called balances having three rows and five columns.
5. Write a for loop to total the contents of the multidimensioned float array balances.
6. Assign the text string "Hello" to the character based array words at declaration time.
7. Assign the text string "Welcome" to the character based array stuff (not at declaration time)
8. Use a printf statement to print out the third element of an integer array called totals
9. Use a printf statement to print out the contents of the character array called words
10. Use a scanf statement to read a string of characters into the array words.
11. Write a for loop which will read five characters (use scanf) and deposit them into the character based array words, beginning at element 0.

Answers: Practice Exercise 7: Arrays

1. Declare a character based array called letters of ten elements

```
char letters[10];
```

2. Assign the character value 'Z' to the fourth element of the letters array

```
letters[3] = 'Z';
```

3. Use a for loop to total the contents of an integer array called numbers which has five elements. Store the result in an integer called total.

```
for( loop = 0, total = 0; loop < 5; loop++ )  
  
total = total + numbers[loop];
```

4. Declare a multidimensioned array of floats called balances having three rows and five columns.

```
float balances[3][5];
```

5. Write a for loop to total the contents of the multidimensioned float array balances.

```
for( row = 0, total = 0; row < 3; row++ )  
  
for( column = 0; column < 5; column++ )  
  
total = total + balances[row][column];
```

6. Assign the text string "Hello" to the character based array words at declaration time.

```
static char words[] = "Hello";
```

7. Assign the text string "Welcome" to the character based array stuff (not at declaration time)

```
char stuff[50];
```

```
strcpy( stuff, "Welcome" );
```

8. Use a printf statement to print out the third element of an integer array called totals

```
printf("%d\n", totals[2] );
```

9. Use a printf statement to print out the contents of the character array called words

```
printf("%s\n", words);
```

10. Use a scanf statement to read a string of characters into the array words.

```
scanf("%s", words);
```

11. Write a for loop which will read five characters (use scanf) and deposit them into the character based array words, beginning at element 0.

```
for( loop = 0; loop < 5; loop++ )  
  
scanf("%c", &words[loop] );
```

FUNCTIONS

A function in C can perform a particular task, and supports the concept of modular programming design techniques.

We have already been exposed to functions. The main body of a C program, identified by the keyword *main*, and enclosed by the left and right braces is a function. It is called by the operating system when the program is loaded, and when terminated, returns to the operating system.

Functions have a basic structure. Their format is

```
return_data_type  function_name  ( arguments, arguments )  
data_type_declarations_of_arguments;  
{  
    function_body  
}
```

It is worth noting that a return_data_type is assumed to be type *int* unless otherwise specified, thus the programs we have seen so far imply that *main()* returns an integer to the operating system.

ANSI C varies slightly in the way that functions are declared. Its format is

```
return_data_type  function_name  (data_type variable_name,  
data_type variable_name, .. )  
{  
    function_body  
}
```

This permits type checking by utilizing function prototypes to inform the compiler of the type and number of parameters a function accepts. When calling a function, this information is used to perform type and parameter checking.

ANSI C also requires that the return_data_type for a function which does not return data must be type *void*. The default return_data_type is assumed to be

integer unless otherwise specified, but must match that which the function declaration specifies.

A simple function is,

```
void print_message( void )
{
    printf("This is a module called
print_message.\n");
}
```

Note the function name is *print_message*. No arguments are accepted by the function, this is indicated by the keyword *void* in the accepted parameter section of the function declaration. The return_data_type is *void*, thus data is not returned by the function.

An ANSI C function prototype for *print_message()* is,

```
void print_message( void );
```

Function prototypes are listed at the beginning of the source file. Often, they might be placed in a users .h (header) file.

FUNCTIONS

Now lets incorporate this function into a program.

```
/* Program illustrating a simple function call */

#include

void print_message( void ); /* ANSI C function prototype */

void print_message( void ) /* the function code */
{
    printf("This is a module called print_message.\n");
}
```

```
main()  
  
{  
  
    print_message();  
  
}
```

Sample Program Output

This is a module called print_message.

To call a function, it is only necessary to write its name. The code associated with the function name is executed at that point in the program. When the function terminates, execution begins with the statement which follows the function name.

In the above program, execution begins at *main()*. The only statement inside the main body of the program is a call to the code of function *print_message()*. This code is executed, and when finished returns back to *main()*.

As there is no further statements inside the main body, the program terminates by returning to the operating system.

FUNCTIONS

In the following example, the function accepts a single data variable, but does not return any information.

```
/* Program to calculate a specific factorial number */  
#include  
  
void calc_factorial( int );    /* ANSI function prototype  
*/  
  
void calc_factorial( int n )  
{  
    int i, factorial_number = 1;  
  
    for( i = 1; i <= n; ++i )  
        factorial_number *= i;  
  
    printf("The factorial of %d is %d\n", n,  
factorial_number );  
}
```

```
main()
{
    int  number = 0;

    printf("Enter a number\n");
    scanf("%d", &number );
    calc_factorial( number );
}
```

Sample Program Output

```
Enter a number
3
The factorial of 3 is 6
```

Lets look at the function `calc_factorial()`. The declaration of the function

```
void calc_factorial( int n )
```

indicates there is no return data type and a single integer is accepted, known inside the body of the function as *n*. Next comes the declaration of the [local variables](#),

```
int  i, factorial_number = 0;
```

It is more correct in C to use,

```
auto int  i, factorial_number = 0;
```

as the keyword auto designates to the compiler that the variables are local. The program works by accepting a variable from the keyboard which is then passed to the function. In other words, the variable *number* inside the main body is then copied to the variable *n* in the function, which then calculates the correct answer.

RETURNING FUNCTION RESULTS

This is done by the use of the keyword *return*, followed by a data variable or constant value, the data type of which must match that of the declared `return_data_type` for the function.

```
float add_numbers( float n1, float n2 )

{

    return n1 + n2; /* legal */
}
```

```
return 6; /* illegal, not the same data type */  
  
return 6.0; /* legal */  
  
}
```

It is possible for a function to have multiple return statements.

```
int validate_input( char command )  
  
{  
  
    switch( command ) {  
  
        case '+' :  
  
        case '-' : return 1;  
  
        case '*' :  
  
        case '/' : return 2;  
  
        default : return 0;  
  
    }  
  
}
```

Here is another example

```
/* Simple multiply program using argument passing */  
  
#include  
  
  
  
int calc_result( int, int ); /* ANSI function prototype */  
  
  
  
int calc_result( int numb1, int numb2 )  
  
{
```

```
auto int result;

result = numb1 * numb2;

return result;

}

main()

{

int digit1 = 10, digit2 = 30, answer = 0;

answer = calc_result( digit1, digit2 );

printf("%d multiplied by %d is %d\n", digit1, digit2, answer );

}
```

Sample Program Output

10 multiplied by 30 is 300

NOTE that the value which is returned from the function (ie result) must be declared in the function.

NOTE: The formal declaration of the function name is preceded by the data type which is returned,

```
int calc_result ( numb1, numb2 )
```

EXERCISE C15

Write a program in C which incorporates a function using parameter passing and

performs the addition of three numbers. The main section of the program is to print the result.

Answer: EXERCISE C15

Write a program in C which incorporates a function using parameter passing and performs the addition of three numbers. The main section of the program is to print the result.

```
#include
int calc_result( int, int, int );

int calc_result( int var1, int var2, int var3 )
{
    int sum;

    sum = var1 + var2 + var3;
    return( sum );          /* return( var1 + var2 + var3
); */
}

main()
{
    int numb1 = 2, numb2 = 3, numb3=4, answer=0;

    answer = calc_result( numb1, numb2, numb3 );
    printf("%d + %d + %d = %d\n", numb1, numb2, numb3,
answer);
}
```

LOCAL AND GLOBAL VARIABLES

Local

These variables only exist inside the specific function that creates them. They are unknown to other functions and to the main program. As such, they are normally implemented using a stack. Local variables cease to exist once the function that created them is completed. They are recreated each time a function is executed or called.

Global

These variables can be accessed (ie known) by any function comprising the program. They are implemented by associating memory locations with variable names. They do not get recreated if the function is recalled.

DEFINING GLOBAL VARIABLES

```
/* Demonstrating Global variables */
#include
int add_numbers( void );          /* ANSI function
prototype */

/* These are global variables and can be accessed by
functions from this point on */
int value1, value2, value3;

int add_numbers( void )
{
    auto int result;
    result = value1 + value2 + value3;
    return result;
}

main()
{
    auto int result;
    value1 = 10;
    value2 = 20;
    value3 = 30;
    result = add_numbers();
    printf("The sum of %d + %d + %d is %d\n",
           value1, value2, value3, final_result);
}
```

Sample Program Output

The sum of 10 + 20 + 30 is 60

The scope of global variables can be restricted by carefully placing the declaration. They are visible from the declaration until the end of the current source file.

```
#include
void no_access( void ); /* ANSI function prototype */
void all_access( void );

static int n2;          /* n2 is known from this point
onwards */

void no_access( void )
{
    n1 = 10;            /* illegal, n1 not yet known */
    n2 = 5;            /* valid */
}

static int n1;          /* n1 is known from this point
onwards */

void all_access( void )
{
```

```
        n1 = 10;          /* valid */
        n2 = 3;           /* valid */
    }
```

AUTOMATIC AND STATIC VARIABLES

C programs have a number of segments (or areas) where data is located. These segments are typically,

```
    _DATA    Static data
    _BSS     Uninitialized static data, zeroed out before call
to main()
    _STACK   Automatic data, resides on stack frame, thus
local to functions
    _CONST   Constant data, using the ANSI C keyword const
```

The use of the appropriate keyword allows correct placement of the variable onto the desired data segment.

```
    /* example program illustrates difference between static
and automatic variables */
    #include
    void demo( void );          /* ANSI function
prototypes */

    void demo( void )
    {
        auto int avar = 0;
        static int svar = 0;

        printf("auto = %d, static = %d\n", avar, svar);
        ++avar;
        ++svar;
    }

    main()
    {
        int i;

        while( i < 3 ) {
            demo();
            i++;
        }
    }
```

Sample Program Output

```
auto = 0, static = 0
auto = 0, static = 1
```

```
auto = 0, static = 2
```

Static variables are created and initialized once, on the first call to the function. Subsequent calls to the function do not recreate or re-initialize the static variable. When the function terminates, the variable still exists on the `_DATA` segment, but cannot be accessed by outside functions.

Automatic variables are the opposite. They are created and re-initialized on each entry to the function. They disappear (are de-allocated) when the function terminates. They are created on the `_STACK` segment.

PASSING ARRAYS TO FUNCTIONS

The following program demonstrates how to pass an array to a function.

```
/* example program to demonstrate the passing of an array
*/
#include
int maximum( int [] );          /* ANSI function
prototype */

int maximum( int values[5] )
{
    int max_value, i;

    max_value = values[0];
    for( i = 0; i < 5; ++i )
        if( values[i] > max_value )
            max_value = values[i];

    return max_value;
}

main()
{
    int values[5], i, max;

    printf("Enter 5 numbers\n");
    for( i = 0; i < 5; ++i )
        scanf("%d", &values[i] );

    max = maximum( values );
    printf("\nMaximum value is %d\n", max );
}
```

Sample Program Output

```
Enter 5 numbers
7 23 45 9 121
Maximum value is 121
```

Note: The program defines an array of five elements (values) and initializes each element to the users inputted values. The array *values* is then passed to the function. The declaration

```
int maximum( int values[5] )
```

defines the function name as *maximum*, and declares that an integer is passed back as the result, and that it accepts a data type called *values*, which is declared as an array of five integers. The values array in the main body is now known as the array values inside function maximum. **IT IS NOT A COPY, BUT THE ORIGINAL.**

This means any changes will update the original array.

A local variable *max_value* is set to the first element of values, and a for loop is executed which cycles through each element in values and assigns the lowest item to *max_value*. This number is then passed back by the return statement, and assigned to *max* in the main section.

Functions and Arrays

C allows the user to build up a library of modules such as the maximum value found in the previous example.

However, in its present form this module or function is limited as it only accepts ten elements. It is thus desirable to modify the function so that it also accepts the number of elements as an argument also. A modified version follows,

```
/* example program to demonstrate the passing of an array
*/
#include

int findmaximum( int [], int );          /* ANSI
function prototype */

int findmaximum( int numbers[], int elements )
{
    int largest_value, i;

    largest_value = numbers[0];

    for( i = 0; i < elements; ++i )
        if( numbers[i] > largest_value )
            largest_value = numbers[i];

    return largest_value;
}
```

```
main()
{
    static int numb1[] = { 5, 34, 56, -12, 3, 19 };
    static int numb2[] = { 1, -2, 34, 207, 93, -12 };

    printf("maximum of numb1[] is %d\n",
findmaximum(numb1, 6));
    printf("maximum of numb2[] is %d\n",
findmaximum(numb2, 6));
}
```

Sample Program Output

```
maximum of numb1[] is 56
maximum of numb2[] is 207
```

PASSING OF ARRAYS TO FUNCTIONS

If an entire array is passed to a function, any changes made also occur to the original array.

PASSING OF MULTIDIMENSIONAL ARRAYS TO FUNCTIONS

If passing a multidimensional array, the number of columns must be specified in the formal parameter declaration section of the function.

EXERCISE C16

Write a C program incorporating a function to add all elements of a two dimensional array. The number of rows are to be passed to the function, and it passes back the total sum of all elements (Use at least a 4 x 4 array).

```
#include
```

```
int add2darray( int [][][5], int ); /* function prototype */
```

```
int add2darray( int array[][5], int rows )
```

```
{
```

```
int total = 0, columns, row;
```

```
for( row = 0; row < rows; row++ )

for( columns = 0; columns < 5; columns++ )

total = total + array[row][columns];

return total;

}

main()

{

int numbers[][] = { {1, 2, 35, 7, 10}, {6, 7, 4, 1, 0} };

int sum;

sum = add2darray( numbers, 2 );

printf("the sum of numbers is %d\n", sum );

}
```

UNCTION PROTOTYPES

These have been introduced into the C language as a means of provided type checking and parameter checking for function calls. Because C programs are generally split up over a number of different source files which are independently compiled, then linked together to generate a run-time program, it is possible for errors to occur.

Consider the following example.

```
/* source file add.c */
void add_up( int numbers[20] )
{
    ....
}

/* source file mainline.c */
static float values[] = { 10.2, 32.1, 0.006, 31.08 };

main()
```

```
{  
    float result;  
    ...  
    result = add_up( values );  
}
```

As the two source files are compiled separately, the compiler generates correct code based upon what the programmer has written. When compiling `mainline.c`, the compiler assumes that the function `add_up` accepts an array of float variables and returns a float. When the two portions are combined and ran as a unit, the program will definitely not work as intended.

To provide a means of combating these conflicts, ANSI C has function prototyping. Just as data types need to be declared, functions are declared also. The function prototype for the above is,

```
/* source file mainline.c */  
void add_up( int numbers[20] );
```

NOTE that the function prototype ends with a semi-colon; in this way we can tell its a declaration of a function type, not the function code. If `mainline.c` was re-compiled, errors would be generated by the call in the main section which references `add_up()`.

Generally, when developing a large program, a separate file would be used to contain all the function prototypes. This file can then be included by the compiler to enforce type and parameter checking.

ADDITIONAL ASSIGNMENT OPERATOR

Consider the following statement,

```
numbers[loop] += 7;
```

This assignment `+=` is equivalent to add equals. It takes the value of `numbers[loop]`, adds it by 7, then assigns the value to `numbers[loop]`. In other words it is the same as,

```
numbers[loop] = numbers[loop] + 7;
```


CLASS EXERCISE C17

What is the outcome of the following, assuming time=2, a=3, b=4, c=5

```
time -= 5;  
  
a *= b + c;
```

CLASS EXERCISE C17

What is the outcome of the following, assuming time=2, a=3, b=4, c=5

```
time -= 5;  
a *= b + c;  
  
time = -3  
a = 27
```

A SIMPLE EXCHANGE SORT ALGORITHM

The following steps define an algorithm for sorting an array,

1. Set i to 0
2. Set j to i + 1
3. If $a[i] > a[j]$, exchange their values
4. Set j to j + 1. If $j < n$ goto step 3
5. Set i to i + 1. If $i < n - 1$ goto step 2
6. a is now sorted in ascending order.

Note: n is the number of elements in the array.

EXERCISE C18

Implement the above algorithm as a function in C, accepting the array and its size,

returning the sorted array in ascending order so it can be printed out by the calling module. The array should consist of ten elements.

A SIMPLE EXCHANGE SORT ALGORITHM

The following steps define an algorithm for sorting an array,

1. Set i to 0
2. Set j to $i + 1$
3. If $a[i] > a[j]$, exchange their values
4. Set j to $j + 1$. If $j < n$ goto step 3
5. Set i to $i + 1$. If $i < n - 1$ goto step 2
6. a is now sorted in ascending order.

Note: n is the number of elements in the array.

EXERCISE C18

Implement the above algorithm as a function in C, accepting the array and its size, returning the sorted array in ascending order so it can be printed out by the calling module. The array should consist of ten elements.

```
#include
```

```
void sort( int [], int );
```

```
void sort( int a[], int elements )
```

```
{
```

```
int i, j, temp;
```

```
i = 0;

while( i < (elements - 1) ) {

    j = i + 1;

    while( j < elements ) {

        if( a[i] > a[j] ) {

            temp = a[i];

            a[i] = a[j];

            a[j] = temp;

        }

        j++;

    }

    i++;

}


main()

{

    int numbers[] = { 10, 9, 8, 23, 19, 11, 2, 7, 1, 13, 12 };

    int loop;


    printf("Before the sort the array was \n");

    for( loop = 0; loop < 11; loop++ )

        printf(" %d ", numbers[loop] );

    sort( numbers, 11 );

    printf("After the sort the array was \n");
```

```
for( loop = 0; loop < 11; loop++ )

printf(" %d ", numbers[loop] );

}
```

RECURSION

This is where a function repeatedly calls itself to perform calculations. Typical applications are games and Sorting trees and lists.

Consider the calculation of 6! (6 factorial)

```
ie 6! = 6 * 5 * 4 * 3 * 2 * 1
    6! = 6 * 5!
    6! = 6 * ( 6 - 1 )!
    n! = n * ( n - 1 )!

/* bad example for demonstrating recursion */
#include

long int factorial( long int );          /* ANSI function
prototype */

long int  factorial( long int n )
{
    long int result;

    if( n == 0L )
        result = 1L;
    else
        result = n * factorial( n - 1L );
    return ( result );
}

main()
{
    int j;

    for( j = 0; j < 11; ++j )
        printf("%2d! = %ld\n", factorial( (long)
j) );
}
```

EXERCISE C19

Rewrite example c9 using a recursive function

RECURSIVE PROGRAMMING: EXERCISE C19

Rewrite example c9 using a recursive function.

```
#include
long int triang_rec( long int );

long int triang_rec( long int number )
{
    long int result;

    if( number == 01 )
        result = 01;
    else
        result = number + triang_rec( number - 1 );
    return( result );
}

main ()
{
    int request;
    long int triang_rec(), answer;

    printf("Enter number to be calculated.\n");
    scanf( "%d", &request);

    answer = triang_rec( (long int) request );
    printf("The triangular answer is %l\n", answer);
}
```

Note this version of function triang_rec

```
#include
long int triang_rec( long int );

long int triang_rec( long int number )
{
    return((number == 01) ? 01 : number*triang_rec(
number-1));
}
```

Practise Exercise 8: Functions

1. Write a function called menu which prints the text string "Menu choices". The function does not pass any data back, and does not accept any data as parameters.
2. Write a function prototype for the above function.
3. Write a function called print which prints a text string passed to it as a parameter (ie, a character based array).
4. Write a function prototype for the above function print.

5. Write a function called `total`, which totals the sum of an integer array passed to it (as the first parameter) and returns the total of all the elements as an integer. Let the second parameter to the function be an integer which contains the number of elements of the array.
6. Write a function prototype for the above function.

Practise Exercise 8: Functions

1. Write a function called `menu` which prints the text string "Menu choices". The function does not pass any data back, and does not accept any data as parameters.

```
void menu( void )  
  
{  
  
    printf("Menu choices");  
  
}
```

2. Write a function prototype for the above function.

```
void menu( void );
```

3. Write a function called `print` which prints a text string passed to it as a parameter (ie, a character based array).

```
void print( char message[] )  
  
{  
  
    printf("%s, message );  
  
}
```

4. Write a function prototype for the above function print.

```
void print( char [] );
```

5. Write a function called total, which totals the sum of an integer array passed to it (as the first parameter) and returns the total of all the elements as an integer. Let the second parameter to the function be an integer which contains the number of elements of the array.

```
int total( int array[], int elements )  
  
{  
  
    int loop, sum;  
  
    for( loop = 0, sum = 0; loop < elements; loop++ )  
  
        sum += array[loop];  
  
    return sum;  
  
}
```

6. Write a function prototype for the above function.

```
int total( int [], int );
```

Handling User Input In C

scanf() has problems, in that if a user is expected to type an integer, and types a string instead, often the program bombs. This can be overcome by reading all input as a string (use getchar()), and then converting the string to the correct data type.

```

/* example one, to read a word at a time */
#include
#include
#define MAXBUFFERSIZE    80

void cleartoendofline( void ); /* ANSI function prototype */

void cleartoendofline( void )
{
    char ch;
    ch = getchar();
    while( ch != '\n' )
        ch = getchar();
}

main()
{
    char    ch; /* handles user input */
    char    buffer[MAXBUFFERSIZE]; /* sufficient to handle
one line */
    int     char_count; /* number of characters
read for this line */
    int     exit_flag = 0;
    int     valid_choice;

    while( exit_flag == 0 ) {
        printf("Enter a line of text (
ch = getchar();
char_count = 0;
while( (ch != '\n')  &&  (char_count <
MAXBUFFERSIZE)) {
            buffer[char_count++] = ch;
            ch = getchar();
        }
        buffer[char_count] = 0x00; /* null terminate
buffer */

        printf("\nThe line you entered was:\n");
        printf("%s\n", buffer);

        valid_choice = 0;
        while( valid_choice == 0 ) {
            printf("Continue (Y/N)?\n");
            scanf(" %c", &ch );
            ch = toupper( ch );
            if((ch == 'Y') || (ch == 'N') )
                valid_choice = 1;
            else
                printf("\007Error: Invalid
choice\n");

                cleartoendofline();
        }
        if( ch == 'N' ) exit_flag = 1;
    }
}

```


Another Example, read a number as a string

```
/* example two, reading a number as a string */
#include
#include
#include
#define MAXBUFFERSIZE 80

void cleartoendofline( void );          /* ANSI function
prototype */

void cleartoendofline( void )
{
    char ch;
    ch = getchar();
    while( ch != '\n' )
        ch = getchar();
}

main()
{
    char    ch;                          /* handles user input */
    char    buffer[MAXBUFFERSIZE];      /* sufficient to handle
one line */
    int     char_count;                  /* number of characters
read for this line */
    int     exit_flag = 0, number, valid_choice;

    while( exit_flag == 0 ) {
        valid_choice = 0;
        while( valid_choice == 0 ) {
            printf("Enter a number between 1 and
1000\n");

            ch = getchar();
            char_count = 0;
            while( (ch != '\n') && (char_count <
MAXBUFFERSIZE)) {
                buffer[char_count++] = ch;
                ch = getchar();
            }
            buffer[char_count] = 0x00;      /* null
terminate buffer */
            number = atoi( buffer );
            if( (number < 1) || (number > 1000) )
                printf("\007Error. Number outside
range 1-1000\n");
            else
                valid_choice = 1;
        }
        printf("\nThe number you entered was:\n");
        printf("%d\n", number);

        valid_choice = 0;
        while( valid_choice == 0 ) {
            printf("Continue (Y/N)?\n");
```

```
scanf(" %c", &ch );
ch = toupper( ch );
if((ch == 'Y') || (ch == 'N') )
    valid_choice = 1;
else
    printf("\007Error: Invalid
choice\n");
    cleartoendofline();
}
if( ch == 'N' ) exit_flag = 1;
}
}
```

Other validation examples

More Data Validation

Consider the following program

```
#include

main() {
    int number;

    printf("Please enter a number\n");
    scanf("%d", &number );
    printf("The number you entered was %d\n", number
);
}
```

The above program has several problems

- the input is not validated to see if its the correct data type
- it is not clear if there are explicit number ranges expected
- the program might crash if an incorrect data type was entered

Perhaps the best way of handling input in C programs is to treat all input as a sequence of characters, and then perform the necessary data conversion.

At this point we shall want to explore some other aspects also, like the concepts of

- trapping data at the source
- the domino/ripple effect

Trapping Data At The Source

This means that the validation of data as to its correct range/limit and data type is best done at the point of entry. The benefits of doing this at the time of data entry are

- less cost later in the program maintenance phase (because data is already validated)
- programs are easier to maintain and modify
- reduces the chances of incorrect data crashing the program later on

The Ripple Through Effect

This refers to the problem of incorrect data which is allowed to propagate through the program. An example of this is sending invalid data to a function to process.

By trapping data at the source, and ensuring that it is correct as to its data type and range, we ensure that bad data cannot be passed onwards. This makes the code which works on processing the data simpler to write and thus reduces errors.

An example

Lets look at the case of wanting to handle user input. Now, we know that users of programs out there in user-land are a bunch of annoying people who spend most of their time inventing new and more wonderful ways of making our programs crash.

Lets try to implement a sort of general purpose way of handling data input, as a replacement to *scanf()*. To do this, we will implement a function which reads the input as a sequence of characters.

The function is *readinput()*, which, in order to make it more versatile, accepts several parameters,

- a character array to store the inputted data
- an integer which specifies the data type to read, STRING, INTEGER, ALPHA
- an integer which specifies the amount of digits/characters to read

We have used some of the functions covered in [ctype.h](#) to check the data type of the inputted data.

```
/* version 1.0 */
#include
#include

#define MAX      80          /* maximum length of buffer
*/
#define DIGIT    1          /* data will be read as digits 0-9
*/
#define ALPHA    2          /* data will be read as alphabet A-Z
*/
#define STRING   3          /* data is read as ASCII
*/
```

```
void readinput( char buff[], int mode, int limit ) {
    int ch, index = 0;

    ch = getchar();
    while( (ch != '\n') && (index < limit) ) {
        switch( mode ) {
            case DIGIT:
                if( isdigit( ch ) ) {
                    buff[index] = ch;
                    index++;
                }
                break;
            case ALPHA:
                if( isalpha( ch ) ) {
                    buff[index] = ch;
                    index++;
                }
                break;
            case STRING:
                if( isascii( ch ) ) {
                    buff[index] = ch;
                    index++;
                }
                break;
            default:
                /* this should not occur */
                break;
        }
        ch = getchar();
    }
    buff[index] = 0x00; /* null terminate input */
}

main() {
    char buffer[MAX];
    int number;

    printf("Please enter an integer\n");
    readinput( buffer, DIGIT, MAX );
    number = atoi( buffer );
    printf("The number you entered was %d\n", number );
}
```

Of course, there are improvements to be made. We can change *readinput* to return an integer value which represents the number of characters read. This would help in determining if data was actually entered. In the above program, it is not clear if the user actually entered any data (we could have checked to see if buffer was an empty array).

So lets now make the changes and see what the modified program looks like

```
/* version 1.1 */
#include
```

```
#include

#define MAX      80          /* maximum length of buffer
*/
#define DIGIT    1          /* data will be read as digits 0-9
*/
#define ALPHA    2          /* data will be read as alphabet A-Z
*/
#define STRING   3          /* data is read as ASCII
*/

int readinput( char buff[], int mode, int limit ) {
    int ch, index = 0;

    ch = getchar();
    while( (ch != '\n') && (index < limit) ) {
        switch( mode ) {
            case DIGIT:
                if( isdigit( ch ) ) {
                    buff[index] = ch;
                    index++;
                }
                break;
            case ALPHA:
                if( isalpha( ch ) ) {
                    buff[index] = ch;
                    index++;
                }
                break;
            case STRING:
                if( isascii( ch ) ) {
                    buff[index] = ch;
                    index++;
                }
                break;
            default:
                /* this should not occur */
                break;
        }
        ch = getchar();
    }
    buff[index] = 0x00; /* null terminate input */
    return index;
}

main() {
    char buffer[MAX];
    int number, digits = 0;

    while( digits == 0 ) {
        printf("Please enter an integer\n");
        digits = readinput( buffer, DIGIT, MAX );
        if( digits != 0 ) {
            number = atoi( buffer );
            printf("The number you entered was %d\n",
number );
        }
    }
}
```

```
}  
}
```

The second version is a much better implementation.

Controlling the cursor position

The following characters, placed after the \ character in a *printf()* statement, have the following effect.

Modifier	Meaning
\b	<i>backspace</i>
\f	<i>form feed</i>
\n	<i>new line</i>
\r	<i>carriage return</i>
\t	<i>horizontal tab</i>
\v	<i>vertical tab</i>
\\	<i>backslash</i>
\"	<i>double quote</i>
\'	<i>single quote</i>
\	<i>line continuation</i>
\nnn	<i>nnn = octal character value</i>
\0xnn	<i>nn = hexadecimal value (some compilers only)</i>

```
printf("\007Attention, that was a beep!\n");
```

FORMATTERS FOR scanf()

The following characters, after the % character, in a *scanf* argument, have the following effect.

Modifier	Meaning
d	read a decimal integer
o	read an octal value

x	read a hexadecimal value
h	read a short integer
l	read a long integer
f	read a float value
e	read a double value
c	read a single character
s	read a sequence of characters, stop reading when an enter key or whitespace character [tab or space]
[...]	Read a character string. The characters inside the brackets indicate the allow-able characters that are to be contained in the string. If any other character is typed, the string is terminated. If the first character is a ^, the remaining characters inside the brackets indicate that typing them will terminate the string.
*	this is used to skip input fields

Example of scanf() modifiers

```
int number;
char text1[30], text2[30];

scanf("%s %d %*f %s", text1, &number, text2);
```

If the user response is,

```
Hello 14 736.55 uncle sam
```

then

```
text1 = hello, number = 14, text2 = uncle
```

and the next call to the scanf function will continue from where the last one left off, so if

```
scanf("%s ", text2);
```

was the next call, then

```
text2 = sam
```

PRINTING OUT THE ASCII VALUES OF CHARACTERS

Enclosing the character to be printed within single quotes will instruct the compiler to print out the Ascii value of the enclosed character.

```
printf("The character A has a value of %d\n", 'A');
```

The program will print out the integer value of the character A.

EXERCISE C20

What would the result of the following operation be?

```
int c;

c = 'a' + 1;

printf("%c\n", c);
```

PRINTING OUT THE ASCII VALUES OF CHARACTERS

EXERCISE C20

What would the result of the following operation be?

```
int c;
c = 'a' + 1;
printf("%c\n", c);
```

The program adds one to the value 'a', resulting in the value 'b' as the value which is assigned to the variable c.

BIT OPERATIONS

C has the advantage of direct bit manipulation and the operations available are,

Operation	Operator	Comment	Value of Sum before	Value of sum after
AND	&	sum = sum & 2;	4	0
OR		sum = sum 2;	4	6
Exclusive OR	^	sum = sum ^ 2;	4	6

1's Complement	~	sum = ~sum;	4	-5
Left Shift	<<	sum = sum << 2;	4	16
Right Shift	>>	sum = sum >> 2;	4	1

```

/* Example program illustrating << and >> */

#include

main()

{

    int n1 = 10, n2 = 20, i = 0;

    i = n2 << 4; /* n2 shifted left four times */

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

    i = n1 >> 5; /* n1 shifted right five times */

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

}

```

Sample Program Output

320

0

```

/* Example program using EOR operator */

#include

```

```
main()

{

int value1 = 2, value2 = 4;


value1 ^= value2;

value2 ^= value1;

value1 ^= value2;

printf("Value1 = %d, Value2 = %d\n", value1, value2);

}
```

Sample Program Output

Value1 = 4, Value2 = 2

```
/* Example program using AND operator */

#include


main()

{

int loop;


for( loop = 'a'; loop <= 'f'; loop++ )

printf("Loop = %c, AND 0xdf = %c\n", loop, loop & 0xdf);

}
```

Sample Program Output

Loop = a, AND 0xdf = A

Loop = b, AND 0xdf = B

Loop = c, AND 0xdf = C

Loop = d, AND 0xdf = D

Loop = e, AND 0xdf = E

Loop = f, AND 0xdf = F

STRUCTURES

A Structure is a data type suitable for grouping data elements together. Lets create a new data structure suitable for storing the date. The elements or fields which make up the structure use the four basic data types. As the storage requirements for a structure cannot be known by the compiler, a definition for the structure is first required. This allows the compiler to determine the storage allocation needed, and also identifies the various sub-fields of the structure.

```
struct date {  
    int month;  
    int day;  
    int year;  
};
```

This declares a NEW data type called *date*. This date structure consists of three basic data elements, all of type integer. **This is a definition to the compiler.** It does not create any storage space and cannot be used as a variable. In essence, its a new data type keyword, like *int* and *char*, and can now be used to create variables. Other data structures may be defined as consisting of the same composition as the *date* structure,

```
struct date todays_date;
```

defines a variable called *todays_date* to be of the same data type as that of the newly defined data type struct *date*.

ASSIGNING VALUES TO STRUCTURE ELEMENTS

To assign todays date to the individual elements of the structure *todays_date*, the statement

```
todays_date.day = 21;  
  
todays_date.month = 07;  
  
todays_date.year = 1985;
```

is used. NOTE the use of the .element to reference the individual elements within *todays_date*.

```
/* Program to illustrate a structure */  
  
#include  
  
  
struct date { /* global definition of type date */  
  
    int month;  
  
    int day;  
  
    int year;  
  
};  
  
  
main()  
  
{  
  
  
  
  
    struct date today;  
  
  
  
    today.month = 10;  
  
    today.day = 14;  
  
    today.year = 1995;
```

```
printf("Todays date is %d/%d/%d.\n", \
today.month, today.day, today.year );
}
```

CLASS EXERCISE C21

Write a program in C that prompts the user for todays date, calculates tomorrows date, and displays the result. Use structures for todays date, tomorrows date, and an array to hold the days for each month of the year. Remember to change the month or year as necessary.

CLASS EXERCISE C21

Write a program in C that prompts the user for todays date, calculates tomorrows date, and displays the result. Use structures for todays date, tomorrows date, and an array to hold the days for each month of the year. Remember to change the month or year as necessary.

```
#include

struct date {
int day, month, year;
};

int days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

struct date today, tommorrow;

void gettodaysdate( void );
```

```
printf("Enter in the current day (1-%d)-->", days[today.month-1]);
```

```
scanf("%d", &today.day);

if( (today.day < 1) || (today.day > days[today.month-1]) )

printf("\007Invalid day\n");

else

valid = 1;

}

}

main()

{

gettodaysdate();

tomorrow = today;

tomorrow.day++;

if( tomorrow.day > days[tomorrow.month-1] ) {

tomorrow.day = 1;

tomorrow.month++;

if( tomorrow.month > 12 ) {

tomorrow.year++;

tomorrow.month = 1;

}

}

printf("Tomorrows date is %02d:%02d:%02d\n", \

tomorrow.day, tomorrow.month, tomorrow.year );

}
```

```

/* TIME.C Program updates time by 1 second using functions */
#include

struct time {
    int hour, minutes, seconds;
};

void time_update( struct time );      /* ANSI function
prototype */

/* function to update time by one second */
void time_update( struct time new_time )
{
    ++new_time.seconds;
    if( new_time.seconds == 60) {
        new_time.seconds = 0;
        ++new_time.minutes;
        if(new_time.minutes == 60) {
            new_time.minutes = 0;
            ++new_time.hour;
            if(new_time.hour == 24)
                new_time.hour = 0;
        }
    }
}

main()
{
    struct time current_time;

    printf("Enter the time (hh:mm:ss):\n");
    scanf("%d:%d:%d", \

&current_time.hour,&current_time.minutes,&current_time.seconds);
    time_update ( current_time);
    printf("The new time is
%02d:%02d:%02d\n",current_time.hour, \
        current_time.minutes,
current_time.seconds);
}

```

INITIALIZING STRUCTURES

This is similar to the initialization of arrays; the elements are simply listed inside a pair of braces, with each element separated by a comma. The structure declaration is preceded by the keyword *static*

```
static struct date today = { 4,23,1998 };
```

ARRAYS OF STRUCTURES

Consider the following,


```
struct date {  
    int month, day, year;  
};
```

Lets now create an array called *birthdays* of the same data type as the structure *date*

```
struct date birthdays[5];
```

This creates an array of 5 elements which have the structure of *date*.

```
birthdays[1].month = 12;  
birthdays[1].day   = 04;  
birthdays[1].year  = 1998;  
--birthdays[1].year;
```

STRUCTURES AND ARRAYS

Structures can also contain arrays.

```
struct month {  
    int number_of_days;  
    char name[4];  
};  
  
static struct month this_month = { 31, "Jan" };  
  
this_month.number_of_days = 31;  
strcpy( this_month.name, "Jan" );  
printf("The month is %s\n", this_month.name );
```

Note that the array *name* has an extra element to hold the end of string nul character.

VARIATIONS IN DECLARING STRUCTURES

Consider the following,

```
struct date {  
    int month, day, year;  
} todays_date, purchase_date;
```

or another way is,

```
struct date {  
    int month, day, year;  
} todays_date = { 9,25,1985 };
```

or, how about an array of structures similar to date,

```
struct date {  
    int month, day, year;  
} dates[100];
```

Declaring structures in this way, however, prevents you from using the structure definition later in the program. The structure definition is thus bound to the variable name which follows the right brace of the structures definition.

CLASS EXERCISE C22

Write a program to enter in five dates, Store this information in an array of structures.

CLASS EXERCISE C22

```
#include  
  
struct date {          /* Global definition of date */  
    int day, month, year;  
};  
  
main()  
{  
    struct date dates[5];  
    int i;  
  
    for( i = 0; i < 5; ++i ) {  
        printf("Please enter the date (dd:mm:yy) "  
);  
        scanf("%d:%d:%d", &dates[i].day,  
            &dates[i].month,  
                &dates[i].year );  
    }  
}
```

STRUCTURES WHICH CONTAIN STRUCTURES

Structures can also contain structures. Consider where both a date and time structure are combined into a single structure called *date_time*, eg,

```
struct date {  
  
    int month, day, year;  
  
};  
  
struct time {  
  
    int hours, mins, secs;  
  
};
```

```
struct date_time {  
  
    struct date sdate;  
  
    struct time stime;  
  
};
```

This declares a structure whose elements consist of two other previously declared structures. Initialization could be done as follows,

```
static struct date_time today = { { 2, 11, 1985 }, { 3, 3, 33 } };
```

which sets the *sdate* element of the structure *today* to the eleventh of February, 1985. The *stime* element of the structure is initialized to three hours, three minutes, thirty-three seconds. Each item within the structure can be referenced if desired, eg,

```
++today.stime.secs;  
  
if( today.stime.secs == 60 ) ++today.stime.mins;
```

BIT FIELDS

Consider the following data elements defined for a PABX telephone system.

flag = 1 bit

off_hook = 1 bit

status = 2 bits

In C, these can be defined as a structure, and the number of bits each occupy can be specified.

```
struct packed_struct {  
  
    unsigned int flag:1;  
  
    unsigned int off_hook:1;  
  
    unsigned int status:2;  
  
} packed_struct1;
```

The :1 following the variable *flag* indicates that flag occupies a single bit. The C compiler will assign all the above fields into a single word.

Assignment is as follows,

```
packed_struct1.flag = 0;  
  
packed_struct1.status = 3;  
  
if( packed_struct1.flag )  
  
.....
```

Practise Exercise 9: Structures

1. Define a structure called *record* which holds an integer called *loop*, a character array of 5 elements called *word*, and a float called *sum*.
2. Declare a structure variable called *sample*, defined from a structure of type *record*.

3. Assign the value 10 to the field *loop* of the *sample* structure of type record.
4. Print out (using printf) the value of the word array of the *sample* structure.
5. Define a new structure called *birthdays*, whose fields are a structure of type *time* called *btime*, and a structure of type *date*, called *bdate*.

Practice Exercise 9: Structures(Answers)

1. Define a structure called *record* which holds an integer called *loop*, a character array of 5 elements called *word*, and a float called *sum*.

```
struct record {  
  
    int loop;  
  
    char word[5];  
  
    float sum;  
  
};
```

2. Declare a structure variable called *sample*, defined from a structure of type *struct record*.

```
struct record sample;
```

3. Assign the value 10 to the field *loop* of the *sample* structure of type *struct record*.

```
sample.loop = 10;
```

4. Print out (using printf) the value of the word array of the *sample* structure.

```
printf("%s", sample.word );
```

5. Define a new structure called *birthdays*, whose fields are a structure of type *struct time* called *btime*, and a structure of type *struct date*, called *bdate*.

```
struct birthdays {  
  
    struct time btime;  
  
    struct date bdate;  
  
};
```

DATA CONVERSION

The following functions convert between data types.

atof() converts an ascii character array to a float

atoi() converts an ascii character array to an integer

itoa() converts an integer to a character array

Example

```
/* convert a string to an integer */  
  
#include  
  
#include  
  
  
char string[] = "1234";  
  
  
main()  
{  
  
    int sum;
```

```
sum = atoi( string );

printf("Sum = %d\n", sum );

}


/* convert an integer to a string */

#include

#include


main()

{

int sum;

char buff[20];


printf("Enter in an integer ");

scanf(" %d", &sum );

printf( "As a string it is %s\n", itoa( sum, buff, 10 ) );

}
```

Note that itoa() takes three parameters,

- the integer to be converted
- a character buffer into which the resultant string is stored
- a radix value (10=decimal,16=hexadecimal)

In addition, itoa() returns a pointer to the resultant string.

FILE INPUT/OUTPUT

To work with files, the library routines must be included into your programs. This is done by the statement,

```
#include
```

as the first statement of your program.

- **Associate the variable with a file using fopen()**

Before using the variable, it is associated with a specific file by using the *fopen()* function, which accepts the pathname for the file and the access mode (like reading or writing).

```
in_file = fopen( "myfile.dat", "r" )
```

In this example, the file **myfile.dat** in the current directory is opened for **read** access.

Process the data in the file

Use the appropriate file routines to process the data

When finished processing the file, close it

Use the *fclose()* function to close the file.

```
fclose( in_file );
```

USING FILES

- **Declare a variable of type FILE**

To use files in C programs, you must declare a file variable to use. This variable must be of type **FILE**, and be declared as a pointer type.

FILE is a predefined type. You declare a variable of this type as

```
FILE *in_file;
```

This declares *infile* to be a pointer to a file.

The following illustrates the *fopen* function, and adds testing to see if the file was opened successfully.

```
#include
```

```
/* declares pointers to an input file, and the fopen function */
```



```
FILE *input_file, *fopen ();

/* the pointer of the input file is assigned the value returned from
the fopen call. */

/* fopen tries to open a file called datain for read only. Note that */

/* "w" = write, and "a" = append. */

input_file = fopen("datain", "r");

/* The pointer is now checked. If the file was opened, it will point to
the first */

/* character of the file. If not, it will contain a NULL or 0. */

if( input_file == NULL ) {

printf("*** datain could not be opened.\n");

printf("returning to dos.\n");

exit(1);

}
```

NOTE: Consider the following statement, which combines the opening of the file and its test to see if it was successfully opened into a single statement.

```
if(( input_file = fopen ("datain", "r" )) == NULL ) {

printf("*** datain could not be opened.\n");

printf("returning to dos.\n");

exit(1);

}
```

INPUTTING/OUTPUTTING SINGLE CHARACTERS

Single characters may be read/written with files by use of the two functions, *getc()*, and *putc()*.

```
int ch;  
  
ch = getc( input_file ); /* assigns character to ch */
```

The *getc()* also returns the value EOF (end of file), so

```
while( (ch = getc( input_file )) != EOF )  
.....
```

NOTE that the *putc/getc* are similar to *getchar/putchar* except that arguments are supplied specifying the I/O device.

```
putc('\n', output_file ); /* writes a newline to output file */
```

CLOSING FILES

When the operations on a file are completed, it is closed before the program terminates. This allows the operating system to cleanup any resources or buffers associated with the file. The *fclose()* function is used to close the file and flush any buffers associated with the file.

```
fclose( input_file );  
  
fclose( output_file );
```

COPYING A FILE

The following demonstrates copying one file to another using the functions we have just covered.

```
#include
```

```
main() /* FCOPY.C */

{

char in_name[25], out_name[25];

FILE *in_file, *out_file, *fopen ();

int c;


printf("File to be copied:\n");

scanf("%24s", in_name);

printf("Output filename:\n");

scanf("%24s", out_name);


in_file = fopen ( in_name, "r");


if( in_file == NULL )

printf("Cannot open %s for reading.\n", in_name);

else {

out_file = fopen (out_name, "w");

if( out_file == NULL )

printf("Can't open %s for writing.\n",out_name);

else {

while( (c = getc( in_file)) != EOF )

putc (c, out_file);

putc (c, out_file); /* copy EOF */

printf("File has been copied.\n");
```

```
fclose (out_file);  
  
}  
  
fclose (in_file);  
  
}  
  
}
```

TESTING FOR THE End Of File TERMINATOR (feof)

This is a built in function incorporated with the *stdio.h* routines. It returns 1 if the file pointer is at the end of the file.

```
if( feof ( input_file ))  
    printf("Ran out of data.\n");
```

THE fprintf AND fscanf STATEMENTS

These perform the same function as *printf* and *scanf*, but work on files. Consider,

```
fprintf(output_file, "Now is the time for all..\n");  
fscanf(input_file, "%f", &float_value);
```

THE fgets AND fputs STATEMENTS

These are useful for reading and writing entire lines of data to/from a file. If *buffer* is a pointer to a character array and *n* is the maximum number of characters to be stored, then

```
fgets (buffer, n, input_file);
```

will read an entire line of text (max chars = *n*) into *buffer* until the newline character or *n*=max, whichever occurs first. The function places a NULL character after the last character in the buffer. The function will be equal to a NULL if no more data exists.

```
fputs (buffer, output_file);
```

writes the characters in *buffer* until a NULL is found. The NULL character is not written to the *output_file*.

NOTE: `fgets` does not store the newline into the buffer, `fputs` will append a newline to the line written to the output file.

Practise Exercise 9A: File Handling

1. Define an input file handle called *input_file*, which is a pointer to a type `FILE`.
2. Using *input_file*, open the file *results.dat* for read mode as a text file.
3. Write C statements which tests to see if *input_file* has opened the data file successfully. If not, print an error message and exit the program.
4. Write C code which will read a line of characters (terminated by a `\n`) from *input_file* into a character array called *buffer*. NULL terminate the buffer upon reading a `\n`.
5. Close the file associated with *input_file*.

Practise Exercise 9A: File Handling (Answers)

1. Define an input file handle called *input_file*, which is a pointer to a type `FILE`.

```
FILE *input_file;
```

2. Using *input_file*, open the file *results.dat* for read mode.

```
input_file = fopen( "results.dat", "r" );
```

3. Write C statements which tests to see if *input_file* has opened the data file successfully. If not, print an error message and exit the program.

```
if( input_file == NULL ) {  
  
    printf("Unable to open file.\n");\n}
```

```
exit(1);  
  
}
```

4. Write C code which will read a line of characters (terminated by a `\n`) from *input_file* into a character array called *buffer*. NULL terminate the buffer upon reading a `\n`.

```
int ch, loop = 0;  
  
ch = fgetc( input_file );  
  
while( (ch != '\n') && (ch != EOF) ) {  
  
    buffer[loop] = ch;  
  
    loop++;  
  
    ch = fgetc( input_file );  
  
}  
  
buffer[loop] = NULL;
```

5. Close the file associated with *input_file*.

```
fclose( input_file );
```

File handling using `open()`, `read()`, `write()` and `close()`

The previous examples of file handling deal with File Control Blocks (FCB). Under MSDOS v3.x (or greater) and UNIX systems, file handling is often done using handles, rather than file control blocks.

Writing programs using handles ensures portability of source code between different operating systems. Using handles allows the programmer to treat the file as a stream of characters.

open()

```
#include
```

```
int open( char *filename, int access, int permission );
```

The available access modes are

```
O_RDONLY O_WRONLY O_RDWR
```

```
O_APPEND O_BINARY O_TEXT
```

The permissions are

```
S_IWRITE S_IREAD S_IWRITE | S_IREAD
```

The *open()* function returns an integer value, which is used to refer to the file. If unsuccessful, it returns -1, and sets the global variable *errno* to indicate the error type.

read()

```
#include
```

```
int read( int handle, void *buffer, int nbyte );
```

The *read()* function attempts to read *nbytes* from the file associated with *handle*, and places the characters read into *buffer*. If the file is opened using *O_TEXT*, it removes carriage returns and detects the end of the file.

The function returns the number of bytes read. On end-of-file, 0 is returned, on error it returns -1, setting *errno* to indicate the type of error that occurred.

write()

```
#include
```

```
int write( int handle, void *buffer, int nbyte );
```

The *write()* function attempts to write *nbytes* from *buffer* to the file associated with *handle*. On text files, it expands each LF to a CR/LF.

The function returns the number of bytes written to the file. A return value of -1 indicates an error, with *errno* set appropriately.

close()

```
#include
```

```
int close( int handle );
```

The *close()* function closes the file associated with *handle*. The function returns 0 if successful, -1 to indicate an error, with *errno* set appropriately.

File handling example of a goods re-ordering program

The following program handles an ASCII text file which describes a number of products, and reads each product into a structure with the program.

```
/* File handling example for PR101 */
```

```
/* processing an ASCII file of records */
```

```
/* Written by B. Brown, April 1994 */
```

```
/* */
```

```
/* process a goods file, and print out */
```

```
/* all goods where the quantity on */
```

```
/* hand is less than or equal to the */
```

```
/* re-order level. */
```



```
#include

#include

#include

#include


/* definition of a record of type goods */

struct goods {

    char name[20]; /* name of product */

    float price; /* price of product */

    int quantity; /* quantity on hand */

    int reorder; /* re-order level */

};


/* function prototypes */

void myexit( int );

void processfile( void );

void printrecord( struct goods );

int getrecord( struct goods * );


/* global data variables */

FILE *fopen(), *input_file; /* input file pointer */


/* provides a tidy means to exit program gracefully */
```

```
void myexit( int exitcode )

{

if( input_file != NULL )

fclose( input_file );

exit( exitcode );

}


/* prints a record */

void printrecord( struct goods record )

{

printf("\nProduct name\t%s\n", record.name );

printf("Product price\t%.2f\n", record.price );

printf("Product quantity\t%d\n", record.quantity );

printf("Product reorder level\t%d\n", record.reorder );

}


/* reads one record from inputfile into 'record', returns 1 for success
*/

int getrecord( struct goods *record )

{

int loop = 0, ch;

char buffer[40];

ch = fgetc( input_file );

/* skip to start of record */
```

```
while( (ch == '\n') || (ch == ' ') && (ch != EOF) )

ch = fgetc( input_file );

if( ch == EOF ) return 0;


/* read product name */

while( (ch != '\n') && (ch != EOF)) {

buffer[loop++] = ch;

ch = fgetc( input_file );

}

buffer[loop] = 0;

strcpy( record->name, buffer );

if( ch == EOF ) return 0;


/* skip to start of next field */

while( (ch == '\n') || (ch == ' ') && (ch != EOF) )

ch = fgetc( input_file );

if( ch == EOF ) return 0;


/* read product price */

loop = 0;

while( (ch != '\n') && (ch != EOF)) {

buffer[loop++] = ch;

ch = fgetc( input_file );

}

buffer[loop] = 0;
```

```
record->price = atof( buffer );

if( ch == EOF ) return 0;


/* skip to start of next field */

while( (ch == '\n') || (ch == ' ') && (ch != EOF) )

ch = fgetc( input_file );

if( ch == EOF ) return 0;


/* read product quantity */

loop = 0;

while( (ch != '\n') && (ch != EOF)) {

buffer[loop++] = ch;

ch = fgetc( input_file );

}

buffer[loop] = 0;

record->quantity = atoi( buffer );

if( ch == EOF ) return 0;


/* skip to start of next field */

while( (ch == '\n') || (ch == ' ') && (ch != EOF) )

ch = fgetc( input_file );

if( ch == EOF ) return 0;


/* read product reorder level */

loop = 0;
```

```
while( (ch != '\n') && (ch != EOF)) {

    buffer[loop++] = ch;

    ch = fgetc( input_file );

}

buffer[loop] = 0;

record->reorder = atoi( buffer );

if( ch == EOF ) return 0;


return 1; /* signify record has been read successfully */

}


/* processes file for records */

void processfile( void )

{

    struct goods record; /* holds a record read from inputfile */


    while( ! feof( input_file )) {

        if( getrecord( &record ) == 1 ) {

            if( record.quantity <= record.reorder )

                printrecord( record );

        }

        else myexit( 1 ); /* error getting record */

    }

}
```

```
main()

{

char filename[40]; /* name of database file */


printf("Example Goods Re-Order File Program\n");

printf("Enter database file ");

scanf(" %s", filename );

input_file = fopen( filename, "rt" );

if( input_file == NULL ) {

printf("Unable to open datafile %s\n", filename );

myexit( 1 );

}

processfile();

myexit( 0 );

}
```

The datafile (a standard ASCII text file) used for this example looks like

```
baked beans

1.20

10

5

greggs coffee

2.76
```

5

10

walls ice-cream

3.47

5

5

cadburys chocs

4.58

12

10

POINTERS

Pointers enable us to effectively represent complex data structures, to change values as arguments to functions, to work with memory which has been dynamically allocated, and to more concisely and efficiently deal with arrays. A pointer provides an indirect means of accessing the value of a particular data item. Lets see how pointers actually work with a simple example,

```
int count = 10, *int_pointer;
```

declares an integer *count* with a value of 10, and also an integer pointer called *int_pointer*. Note that the prefix *** defines the variable to be of type pointer. To set up an indirect reference between *int_pointer* and *count*, the *&* prefix is used, ie,

```
int_pointer = &count
```

This assigns the memory address of *count* to *int_pointer*, not the actual value of *count* stored at that address.

POINTERS CONTAIN MEMORY ADDRESSES, NOT VALUES!

To reference the value of *count* using *int_pointer*, the *** is used in an assignment, eg,

```
x = *int_pointer;
```

Since *int_pointer* is set to the memory address of *count*, this operation has the effect of assigning the contents of the memory address pointed to by *int_pointer* to the variable *x*, so that after the operation variable *x* has a value of 10.

```
#include
```

```
main()
```

```
{
```

```
int count = 10, x, *int_pointer;
```

```
/* this assigns the memory address of count to int_pointer */
```

```
int_pointer = &count;
```

```
/* assigns the value stored at the address specified by int_pointer to  
x */
```

```
x = *int_pointer;
```

```
printf("count = %d, x = %d\n", count, x);
```

```
}
```

This however, does not illustrate a good use for pointers.

The following program illustrates another way to use pointers, this time with characters,

```
#include

main()

{

char c = 'Q';

char *char_pointer = &c;

printf("%c %c\n", c, *char_pointer);

c = 'Z';

printf("%c %c\n", c, *char_pointer);

*char_pointer = 'Y';

/* assigns Y as the contents of the memory address specified by
char_pointer */

printf("%c %c\n", c, *char_pointer);

}
```

CLASS EXERCISE C23

Determine the output of the pointer programs P1, P2, and P3.

```
/* P1.C illustrating pointers */
#include

main()
{
    int count = 10, x, *int_pointer;
```

```

        /* this assigns the memory address of count to
int_pointer */
        int_pointer = &count;

        /* assigns the value stored at the address specified by
int_pointer to x */
        x = *int_pointer;

        printf("count = %d, x = %d\n", count, x);
    }

/* P2.C Further examples of pointers */
#include

main()
{
    char c = 'Q';
    char *char_pointer = &c;

    printf("%c %c\n", c, *char_pointer);

    c = '/';
    printf("%c %c\n", c, *char_pointer);
    *char_pointer = '(';
    /* assigns ( as the contents of the memory address specified by
char_pointer */
    printf("%c %c\n", c, *char_pointer);
}

```

CLASS EXERCISE C24

```

/* P3.C Another program with pointers */
#include

main()
{
    int i1, i2, *p1, *p2;

    i1 = 5;
    p1 = &i1;
    i2 = *p1 / 2 + 10;
    p2 = p1;

    printf("i1 = %d, i2 = %d, *p1 = %d, *p2 = %d\n", i1, i2,
*p1, *p2);
}

```

CLASS EXERCISE C25

Determine the output of the pointer programs P1, P2, and P3.

```
/* P1.C illustrating pointers */

#include

main()

{

int count = 10, x, *int_pointer;

/* this assigns the memory address of count to int_pointer */

int_pointer = &count;

/* assigns the value stored at the address specified by int_pointer to
x */

x = *int_pointer;

printf("count = %d, x = %d\n", count, x);

}

count = 10, x = 10;
```

```
/* P2.C Further examples of pointers */
```

```
#include
```

```
main()
```

```
{
```

```
char c = 'Q';
```

```
char *char_pointer = &c;
```

```
printf("%c %c\n", c, *char_pointer);
```

```
c = '/';
```

```
printf("%c %c\n", c, *char_pointer);
```

```
*char_pointer = '(';
```

```
/* assigns ( as the contents of the memory address specified by  
char_pointer */
```

```
printf("%c %c\n", c, *char_pointer);
```

```
}
```

```
Q Q
```

```
/ /
```

```
( (
```

```
/* P3.C Another program with pointers */
```

```
#include

main()

{

int i1, i2, *p1, *p2;

i1 = 5;

p1 = &i1;

i2 = *p1 / 2 + 10;

p2 = p1;

printf("i1 = %d, i2 = %d, *p1 = %d, *p2 = %d\n", i1, i2, *p1, *p2);

}

i1 = 5, i2 = 12, *p1 = 5, *p2 = 5
```

Practice Exercise 10: Pointers

1. Declare a pointer to an integer called *address*.
2. Assign the address of a float variable *balance* to the float pointer *temp*.
3. Assign the character value 'W' to the variable pointed to by the char pointer *letter*.
4. What is the output of the following program segment?

```
int count = 10, *temp, sum = 0;

temp = &count;
*temp = 20;
```

```
temp =  $\Sigma$ 
*temp = count;
printf("count = %d, *temp = %d, sum = %d\n", count, *temp, sum
);
```

5. Declare a pointer to the text string "Hello" called *message*.

Practise Exercise 10: Pointers

1. Declare a pointer to an integer called *address*.

```
int *address;
```

2. Assign the address of a float variable *balance* to the float pointer *temp*.

```
temp = &balance;
```

3. Assign the character value 'W' to the variable pointed to by the char pointer *letter*.

```
*letter = 'W';
```

4. What is the output of the following program segment?

```
int count = 10, *temp, sum = 0;
```

```
temp = &count;
```

```
*temp = 20;
```

```
temp =  $\Sigma$ 
```

```
*temp = count;
```

```
printf("count = %d, *temp = %d, sum = %d\n", count, *temp, sum );
```

```
count = 20, *temp = 20, sum = 20
```

5. Declare a pointer to the text string "Hello" called *message*.

```
char *message = "Hello";
```

POINTERS AND STRUCTURES

Consider the following,

```
struct date {  
    int month, day, year;  
};  
  
struct date  todays_date, *date_pointer;  
  
date_pointer = &todays_date;  
  
(*date_pointer).day = 21;  
(*date_pointer).year = 1985;  
(*date_pointer).month = 07;  
  
++(*date_pointer).month;  
if ((*date_pointer).month == 08 )  
    .....
```

Pointers to structures are so often used in C that a special operator exists. The structure pointer operator, the \rightarrow , permits expressions that would otherwise be written as,

$(*x).y$

to be more clearly expressed as

$x \rightarrow y$

making the if statement from above program

```

if( date_pointer->month == 08 )
    .....

/* Program to illustrate structure pointers */
#include

main()
{
    struct date { int month, day, year; };
    struct date today, *date_ptr;

    date_ptr = &today;
    date_ptr->month = 9;
    date_ptr->day = 25;
    date_ptr->year = 1983;

    printf("Todays date is %d/%d/%d.\n", date_ptr->month, \
        date_ptr->day, date_ptr->year % 100);
}

```

So far, all that has been done could've been done without the use of pointers. Shortly, the real value of pointers will become apparent.

STRUCTURES CONTAINING POINTERS

Naturally, a pointer can also be a member of a structure.

```

struct int_pointers {
    int *ptr1;
    int *ptr2;
};

```

In the above, the structure *int_pointers* is defined as containing two integer pointers, *ptr1* and *ptr2*. A variable of type struct *int_pointers* can be defined in the normal way, eg,

```

struct int_pointers ptrs;

```

The variable *ptrs* can be used normally, eg, consider the following program,

```

#include
main() /* Illustrating structures containing pointers */
{
    struct int_pointers { int *ptr1, *ptr2; };
    struct int_pointers ptrs;
    int i1 = 154, i2;

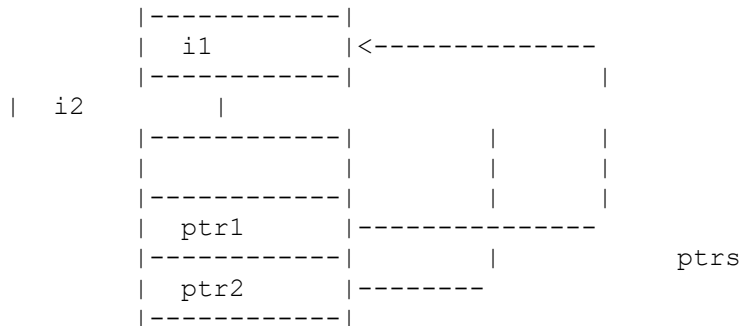
    ptrs.ptr1 = &i1;
}

```



```
ptrs.ptr2 = &i2;
*ptrs.ptr2 = -97;
printf("i1 = %d, *ptrs.ptr1 = %d\n", i1, *ptrs.ptr1);
printf("i2 = %d, *ptrs.ptr2 = %d\n", i2, *ptrs.ptr2);
}
```

The following diagram may help to illustrate the connection,



POINTERS AND CHARACTER STRINGS

A pointer may be defined as pointing to a character string.

```
#include

main()
{
    char *text_pointer = "Good morning!";

    for( ; *text_pointer != '\0'; ++text_pointer)
        printf("%c", *text_pointer);
}
```

or another program illustrating pointers to text strings,

```
#include

main()
{
    static char *days[] = {"Sunday", "Monday", "Tuesday",
"Wednesday", \
                                "Thursday",
"Friday", "Saturday"};
    int i;

    for( i = 0; i < 6; ++i )
        printf( "%s\n", days[i]);
}
```

Remember that if the declaration is,

```
char *pointer = "Sunday";
```

then the null character { '\0' } is automatically appended to the end of the text string. This means that %s may be used in a *printf* statement, rather than using a *for* loop and %c to print out the contents of the pointer. The %s will print out all characters till it finds the null terminator.

Practise Exercise 11: Pointers & Structures

1. Declare a pointer to a structure of type *date* called *dates*.
2. If the above structure of type *date* comprises three integer fields, *day*, *month*, *year*, assign the value 10 to the field *day* using the *dates* pointer.
3. A structure of type *machine* contains two fields, an integer called *name*, and a char pointer called *memory*. Show what the definition of the structure looks like.
4. A pointer called *mpu641* of type *machine* is declared. What is the command to assign the value NULL to the field *memory*.
5. Assign the address of the character array *CPUtype* to the field *memory* using the pointer *mpu641*.
6. Assign the value 10 to the field *name* using the pointer *mpu641*.
7. A structure pointer *times* of type *time* (which has three fields, all pointers to integers, *day*, *month* and *year* respectively) is declared. Using the pointer *times*, update the field *day* to 10.
8. An array of pointers (10 elements) of type *time* (as detailed above in '7.'), called *sample* is declared. Update the field *month* of the third array element to 12.

Practice Exercise 11: Pointers & Structures (Answers)

1. Declare a pointer to a structure of type *date* called *dates*.

```
struct date *dates;
```

2. If the above structure of type *date* comprises three integer fields, *day*, *month*, *year*, assign the value 10 to the field *day* using the *dates* pointer.

```
dates->day = 10;
```

3. A structure of type *machine* contains two fields, an integer called *name*, and a char pointer called *memory*. Show what the definition of the structure looks like.

```
|-----| <-----|
|         | name      |
|-----|           | machine
|         | memory    |
|-----| <-----|
```

4. A pointer called *mpu641* of type *machine* is declared. What is the command to assign the value NULL to the field *memory*.

```
mpu641->memory = (char *) NULL;
```

5. Assign the address of the character array *CPUtype* to the field *memory* using the pointer *mpu641*.

```
mpu641->memory = CPUtype;
```

6. Assign the value 10 to the field *name* using the pointer *mpu641*.

```
mpu641->name = 10;
```

7. A structure pointer *times* of type *time* (which has three fields, all pointers to integers, day, month and year respectively) is declared. Using the pointer *times*, update the field *day* to 10.

```
*(times->day) = 10;
```

8. An array of pointers (10 elements) of type *time* (as detailed above in 7.), called *sample* is declared. Update the field *month* of the third array element to 12.

```
*(sample[2]->month) = 12;
```

```
#include

struct machine {
    int name;
    char *memory;
};

struct machine p1, *mpu641;

main()
{
    p1.name = 3;
    p1.memory = "hello";
    mpu641 = &p1;
    printf("name = %d\n", mpu641->name );
    printf("memory = %s\n", mpu641->memory );

    mpu641->name = 10;
    mpu641->memory = (char *) NULL;
    printf("name = %d\n", mpu641->name );
    printf("memory = %s\n", mpu641->memory );
}
```

```
#include

struct time {
    int *day;
    int *month;
    int *year;
};

struct time t1, *times;

main()
{
    int d=5, m=12, y=1995;

    t1.day = &d;
    t1.month = &m;
    t1.year = &y;

    printf("day:month:year = %d:%d:%d\n", *t1.day, *t1.month, *t1.year
);

    times = &t1;

    *(times->day) = 10;
    printf("day:month:year = %d:%d:%d\n", *t1.day, *t1.month, *t1.year
);
}
```

Practice Exercise 11a: Pointers & Structures

This program introduces a structure which is passed to a function *editrecord()* as a reference and accessed via a pointer *goods*.

Determine the output of the following program.

```
#include
#include

struct record {
    char name[20];
    int id;
    float price;
};

void editrecord( struct record * );

void editrecord( struct record *goods )
{
    strcpy( goods->name, "Baked Beans" );
    goods->id = 220;
    (*goods).price = 2.20;
    printf("Name = %s\n", goods->name );
    printf("ID = %d\n", goods->id);
    printf("Price = %.2f\n", goods->price );
}

main()
{
    struct record item;

    strcpy( item.name, "Red Plum Jam");
    editrecord( &item );
    item.price = 2.75;
    printf("Name = %s\n", item.name );
    printf("ID = %d\n", item.id);
    printf("Price = %.2f\n", item.price );
}
```

1. Before call to editrecord()
2. After return from editrecord()
3. The final values of values, item.name, item.id, item.price

Practice Exercise 11a: Pointers & Structures(Answers)

Determine the output of the following program.

```
#include
#include

struct record {
    char name[20];
    int id;
    float price;
};

void editrecord( struct record * );

void editrecord( struct record *goods )
{
    strcpy( goods->name, "Baked Beans" );
    goods->id = 220;
    (*goods).price = 2.20;
    printf("Name = %s\n", goods->name );
    printf("ID = %d\n", goods->id);
    printf("Price = %.2f\n", goods->price );
}

main()
{
    struct record item;

    strcpy( item.name, "Red Plum Jam");
    editrecord( &item );
    item.price = 2.75;
    printf("Name = %s\n", item.name );
    printf("ID = %d\n", item.id);
    printf("Price = %.2f\n", item.price );
}
```

1. Before call to editrecord()

```
item.name = "Red Plum Jam"
item.id = 0
item.price = 0.0
```

2. After return from editrecord()

```
item.name = "Baked Beans"
item.id = 220
item.price = 2.20
```

3. The final values of values, item.name, item.id, item.price

```
item.name = "Baked Beans"
```

```
item.id = 220  
item.price = 2.75
```

C25: Examples on Pointer Usage

This program introduces a structure which contains pointers as some of its fields.

Determine the output of the following program.

```
#include  
#include  
  
struct sample {  
    char *name;  
    int *id;  
    float price;  
};  
  
static char product[]="Red Plum Jam";  
  
main()  
{  
    int code = 312, number;  
    char name[] = "Baked beans";  
    struct sample item;  
  
    item.name = product;  
    item.id = &code;  
    item.price = 2.75;  
    item.name = name;  
    number = *item.id;  
    printf("Name = %s\n", item.name );  
    printf("ID = %d\n", *item.id);  
    printf("Price = %.2f\n", item.price );  
}
```

C25: Examples on Pointer Usage};

```
static char product[]="Red Plum Jam";
```

```
main()
```

```
{
```

```
int code = 312, number;
```

```
char name[] = "Baked beans";
```

(Answers)

Determine the output of the following program.

```
#include
```

```
#include
```

```
struct sample {
```

```
char *name;
```

```
int *id;
```

```
float price;
```

```
struct sample item;
```

```
item.name = product;
```

```
item.id = &code;
```

```
item.price = 2.75;
```

```
item.name = name;
```

```
number = *item.id;
```

```
printf("Name = %s\n", item.name );
```

```
printf("ID = %d\n", *item.id);
```

```
printf("Price = %.2f\n", item.price );
```

```
}
```


Name = Baked Beans

ID = 312

Price = 2.75

C26: Examples on Pointer Usage

This program introduces a structure which has pointers as some of its fields. The structure is passed to a function *printrecord()* as a reference and accessed via a pointer *goods*. This function also updates some of the fields.

Determine the output of the following program.

```
#include
#include

struct sample {
    char *name;
    int *id;
    float price;
};

static char product[] = "Greggs Coffee";
static float price1 = 3.20;
static int id = 773;

void printrecord( struct sample * );

void printrecord( struct sample *goods )
{
    printf("Name = %s\n", goods->name );
    printf("ID = %d\n", *goods->id);
    printf("Price = %.2f\n", goods->price );
    goods->name = &product[0];
    goods->id = &id;
    goods->price = price1;
}

main()
{
    int code = 123, number;
    char name[] = "Apple Pie";
    struct sample item;

    item.id = &code;
    item.price = 1.65;
    item.name = name;
    number = *item.id;
```

```
    printrecord( &item );  
    printf("Name = %s\n", item.name );  
    printf("ID = %d\n", *item.id);  
    printf("Price = %.2f\n", item.price );  
}
```

C26: Examples on Pointer Usage (Answers)

Determine the output of the following program.

```
#include  
  
#include  
  
struct sample {  
  
    char *name;  
  
    int *id;  
  
    float price;  
  
};  
  
static char product[] = "Greggs Coffee";  
  
static float price1 = 3.20;  
  
static int id = 773;  
  
void printrecord( struct sample * );  
  
void printrecord( struct sample *goods )  
{  
  
    printf("Name = %s\n", goods->name );  
  
    printf("ID = %d\n", *goods->id);  

```

```
printf("Price = %.2f\n", goods->price );  
  
goods->name = &product[0];  
  
goods->id = &id;  
  
goods->price = price1;  
  
}
```

```
main()  
  
{  
  
int code = 123, number;  
  
char name[] = "Apple Pie";  
  
struct sample item;  
  
  
item.id = &code;  
  
item.price = 1.65;  
  
item.name = name;  
  
number = *item.id;  
  
printrecord( &item );  
  
printf("Name = %s\n", item.name );  
  
printf("ID = %d\n", *item.id);  
  
printf("Price = %.2f\n", item.price );  
  
}
```

What are we trying to print out?

What does it evaluate to?

eg,

```
printf("ID = %d\n", *goods->id);
```

%d is an integer

we want the value to be a variable integer type

goods->id,

what is id, its a pointer, so we mean contents of,

*therefor we use *goods->id*

which evaluates to an integer type

Name = Apple Pie

ID = 123

Price = 1.65

Name = Greggs Coffee

ID = 773

Price = 3.20
