**UNIT I**

* **Data Object**
* A data object is usually represented as storage in computer memory.
* A data object is a region of storage that contains a value or group of values.
* a run-time grouping of one or more pieces of data in a virtual computer.
* a location in memory with an assigned name in the actual computer.
* a combination of data objects is a data structure.

**Data objects are of two types -:**

1. **System defined** – which are already present in the system.eg : runtime storage, stacks, file buffers, free space lists.
2. **Programmer defined** – which are defined by the user.eg : variables, constants, arrays, files etc.

**data object**: contains a data value that is manipulated as a unit.

**Data value:** a bit pattern that is recognized by the computer.

**Types of data values** : single numbers, pointers to other objects, characters etc.

**Attributes and Bindings**

* **Type**: determines the set of data values that the object may take and the applicable operations.
* **Name:** the binding of a name to a data object.
* **Component**: the binding of a data object to one or more data objects.
* **Location:** the storage location in memory assigned by the system
* **Value:** the assignment of a bit pattern to a name

*Type, name* and *component* are bound at translation, *location* is bound at loading, *value* is bound at execution

**In programs, data objects are represented as variables and constants**

**Variables :** Data objects defined and named by the programmer explicitly.

**Constants:** Data objects with a name that is permanently bound to a value for its lifetime.

**Life-time of a data object:** persistent and non-persistent data objects

**Persistence:** Existence of data beyond the execution time of a single program

**Non Persistence:** Existence of data at the time of execution of a single program

**Data Types -** *A data type is a class of data objects with a set of operations for creating and manipulating them.* **Examples** of elementary data types: integer, real, character, Boolean, enumeration, pointer.

**Specification of a data type:**

1. The **attributes** that distinguish data objects of that type

Data type, name - invariant during the lifetime of the object

* stored in a descriptor and used during the program execution
* used only to determine the storage representation,

not used explicitly during execution

1. The **values** that data object of that type may have,

* Determined by the type of the object
* Usually an ordered set, i.e. it has a least and a greatest value

1. The **operations** that define the possible manipulations of data objects of that type.

An operation is defined by:

* Domain - set of possible input arguments
* Range - set of possible results
* Action - how the result is produced

**op name: arg type x arg type x … x arg type ® result type**

* **Arrays**
* An array is collection of items stored at continuous memory locations.
* Arrays are reffered to as structured data types.
* An array is defined as **finite ordered collection of homogenous** data, stored in contiguous memory locations.

**Example where arrays are used,**

* to store list of Employee or Student names,
* to store marks of students,
* to store list of numbers or characters etc.

Since arrays provide an easy way to represent data, it is classified amongst the data structures in C. Other data structures in c are **structure**, **lists**, **queues**, **trees** etc. Array can be used to represent not only simple list of data but also table of data in two or three dimensions.

**Types of Arrays :**

* 1. One Dimensional array
  2. Multi Dimensional array

**One Dimensional array**



* **Array declaration by specifying size**

int arr1[10];

* **Array declaration by initializing elements**

int arr[] = { 10, 20, 30, 40 }

* **Array declaration by specifying size and initializing elements**

int arr[6] = { 10, 20, 30, 40 }

**Example program of one dimensional array**

#include<stdio.h>

void main()

{

int arr[4];

int i, j;

printf("Enter array element");

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

{

scanf("%d", &arr[i]); //Run time array initialization

}

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

{

printf("%d\n", arr[j]);

}

}

**Multi dimensional arrays**

C language supports multidimensional arrays also. The simplest form of a multidimensional array is the two-dimensional array. Both the row's and column's index begins from 0.

**Two-dimensional arrays are declared as follows,**

data-type array-name[row-size][column-size]

Example :

int a[3][4];

**An array can also be declared and initialized together. For example,**

int arr[][3] = {

{0,0,0},

{1,1,1}

};

**Example program of two dimensional array**

#include<stdio.h>

void main()

{

int arr[3][4];

int i, j, k;

printf("Enter array element");

for(i = 0; i < 3;i++)

{

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

{

scanf("%d", &arr[i][j]);

}

}

for(i = 0; i < 3; i++)

{

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

{

printf("%d", arr[i][j]);

}

}

}

* **Structures**

Structure is a collection of variables (can be of different types) under a single name.

Example program

#include <stdio.h>

struct student

{

char name[50];

int roll;

float marks;

} s[10];

int main()

{

int i;

printf("Enter information of students:\n");

// storing information

for(i=0; i<10; ++i)

{

s[i].roll = i+1;

printf("\nFor roll number%d,\n",s[i].roll);

printf("Enter name: ");

scanf("%s",s[i].name);

printf("Enter marks: ");

scanf("%f",&s[i].marks);

printf("\n");

}

printf("Displaying Information:\n\n");

// displaying information

for(i=0; i<10; ++i)

{

printf("\nRoll number: %d\n",i+1);

printf("Name: ");

puts(s[i].name);

printf("Marks: %.1f",s[i].marks);

printf("\n");

}

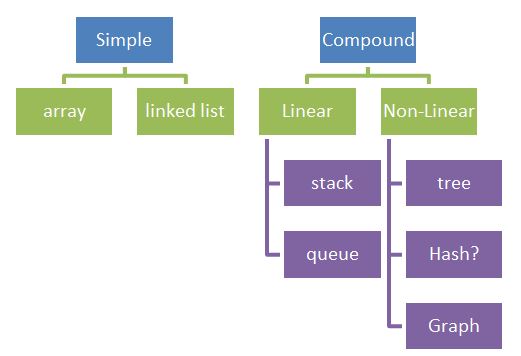
return 0;

}

* **Data Structure**

A data structure is a particular way of organizing data in a computer so that it can be used effectively.

**Types of data structure**



**Below is an overview of some popular linear data structures.**

1. Array

2. Linked List

3. Stack

4. Queue

1.**Array** is a data structure used to store homogeneous elements at contiguous locations. Size of an array must be provided before storing data.

2. A **linked list** is a linear data structure (like arrays) where each element is a separate object. Each element (that is node) of a list is comprising of two items – the data and a reference to the next node.

3. A **stack** or LIFO (last in, first out) is an abstract data type that serves as a collection of elements, with two principal operations: push, which adds an element to the collection, and pop, which removes the last element that was added. In stack both the operations of push and pop takes place at the same end that is top of the stack. It can be implemented by using both array and linked list.

4. A **queue** or FIFO (first in, first out) is an abstract data type that serves as a collection of elements, with two principal operations: enqueue, the process of adding an element to the collection.(The element is added from the rear side) and dequeue, the process of removing the first element that was added. (The element is removed from the front side). It can be implemented by using both array and linked list.

* **Abstract data type (ADT)**

Abstract Data type (ADT) is a type (or class) for objects whose behavior is defined by a set of value and a set of operations.  
The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called “abstract” because it gives an implementation independent view. The process of providing only the essentials and hiding the details is known as abstraction.  
The user of data type need not know that data type is implemented, for example, we have been using int, float, char data types only with the knowledge with values that can take and operations that can be performed on them without any idea of how these types are implemented. So a user only needs to know what a data type can do but not how it will do it. We can think of ADT as a black box which hides the inner structure and design of the data type.

**Types of Abstract data type**

**Stack,Queue,array etc.**

* **Notation to specify Abstract data type**

As ADTs provide an abstract view to describe properties of sets of entities, their use is independent from a particular programming language. We therefore introduce a notation here which is adopted from [[3](http://www.desy.de/gna/html/cc/Tutorial/node12.html#ford:96a)]. Each ADT description consists of two parts:

* **Data**: This part describes the structure of the data used in the ADT in an informal way.
* **Operations**: This part describes valid operations for this ADT, hence, it describes its interface. We use the special operation **constructor** to describe the actions which are to be performed once an entity of this ADT is created and **destructor** to describe the actions which are to be performed once an entity is destroyed. For each operation the provided *arguments* as well as *preconditions* and *postconditions* are given.

As an example the description of the ADT *Integer* is presented. Let *k* be an integer expression:

**ADT *Integer* is**

**Data**

A sequence of digits optionally prefixed by a plus or minus sign. We refer to this signed whole number as *N*.

**Operations**

**constructor**

Creates a new integer.

**add(k)**

Creates a new integer which is the sum of *N* and *k*.

Consequently, the *postcondition* of this operation is *sum* = *N*+*k*. Don't confuse this with assign statements as used in programming languages! It is rather a mathematical equation which yields ``true'' for each value *sum*, *N* and *k* after *add* has been performed.

**sub(k)**

Similar to *add*, this operation creates a new integer of the difference of both integer values. Therefore the postcondition for this operation is *sum* = *N*-*k*.

**set(k)**

Set *N* to *k*. The postcondition for this operation is *N* = *k*.

**...**

**end**

* **Pre conditions and post conditions**

**function’s precondition** refers to what must hold before the function is executed, and a **function’s postcondition** refers to what will be true or changed after the execution of the function in addition to what it returns.  
  
If the function modifies a global variable, for example, this variable’s state should be included in the preconditions. A tremendous number of bugs occur due to incomplete understanding of preconditions and postconditions of a function.

**Example**

**int get\_sum(int first\_int, int second\_int, int &sum)**

**{**

**sum = first\_int + second\_int;**

**return sum;**

**}**

This function computes the sum of the two given integers and stores it in the given argument **sum** as well as returning it. In this case the precondition is that **first\_int** and **second\_int** are valid integers bound by C++’s integer maximum value and minimum value. And the postcondition is that the sum of those integers is returned and variable **sum** is updated to store the value of the sum.  
  
Knowing exactly the state of the program before and after the execution of a function is extremely important. It will greatly reduce logic errors and confusion.

* **Pre-condition** - a statement or set of statements that outlines a condition that should be true, or conditions that should be true, when the operation is called. The operation is not guaranteed to perform as it should unless the pre-conditions have been met.
* **Post-condition** - is a statement or statements describing the condition that will be true when the operation has completed its task. If the operation is correct and the pre-condition(s) met, then the post-condition is guaranteed to be true.

The pre-condition and post-condition are more than just a way to summarise a data type's actions. Stating these conditions should be the first step in designing and writing your algorithms.

When working with ADTs, you must be clear about your own reasoning and in this respect they are an excellent way of clarifying and documenting your thinking process. More importantly, if you are part of a group of programmers working together on a complex project, the operation definition, along with the pre and post-conditions, supply all the information required by another computer programmer to utilise the interface of the data type.

**Pre-condition**: an assertion that must be true in order for the operation to execute correctly:

* **The implementer** assumes it to be true...and codes accordingly
* **The user** must be sure in the code that it is met in order to use.

**Post-condition**: an assertion that prevails upon completion of the operation ie a description of the results of the actions performed by the operation. It must be accurately and precisely stated:

* **The implementer** must make it (through their code) true
* **The user assumes** it will be true

**PRE** and **POST**-conditions provide a set of conditions (a contract) for the programmer of the operation to meet.

**PRE** and **POST**-conditions also help the reader and user to know what they can expect as the result of the operation upon the data structure.

**PRE** and **POST**-conditions for public interface operations **do not** include implementation details.

* **Implementation of data structures – refer the programs(txt file)**

**Note: for array of pointer program, pointer to an array program, pointer to structure,array of structure.. refer your class register notes.**