DATA STRUCTURES

LECTURE-5

ABSTRACT DATA TYPE

Dr. Sumitra Kisan

Two important things about data type:

1. Defines a certain domain of values.
2. Defines operations allowed on those values.

Example:

int type

-- takes only integer values

-- Operations- Addition, Subtraction, Multiplication, Division, Bitwise operations etc.

float type

-- takes only floating point values

-- Operations- Addition, Subtraction, Multiplication, Division.. etc are allowed. Bitwise and % operations are not allowed

Used defined data types

The operations and values of user defined data types are not specified in the language itself but is specified by the user.

Example: Structure, union and Enumeration

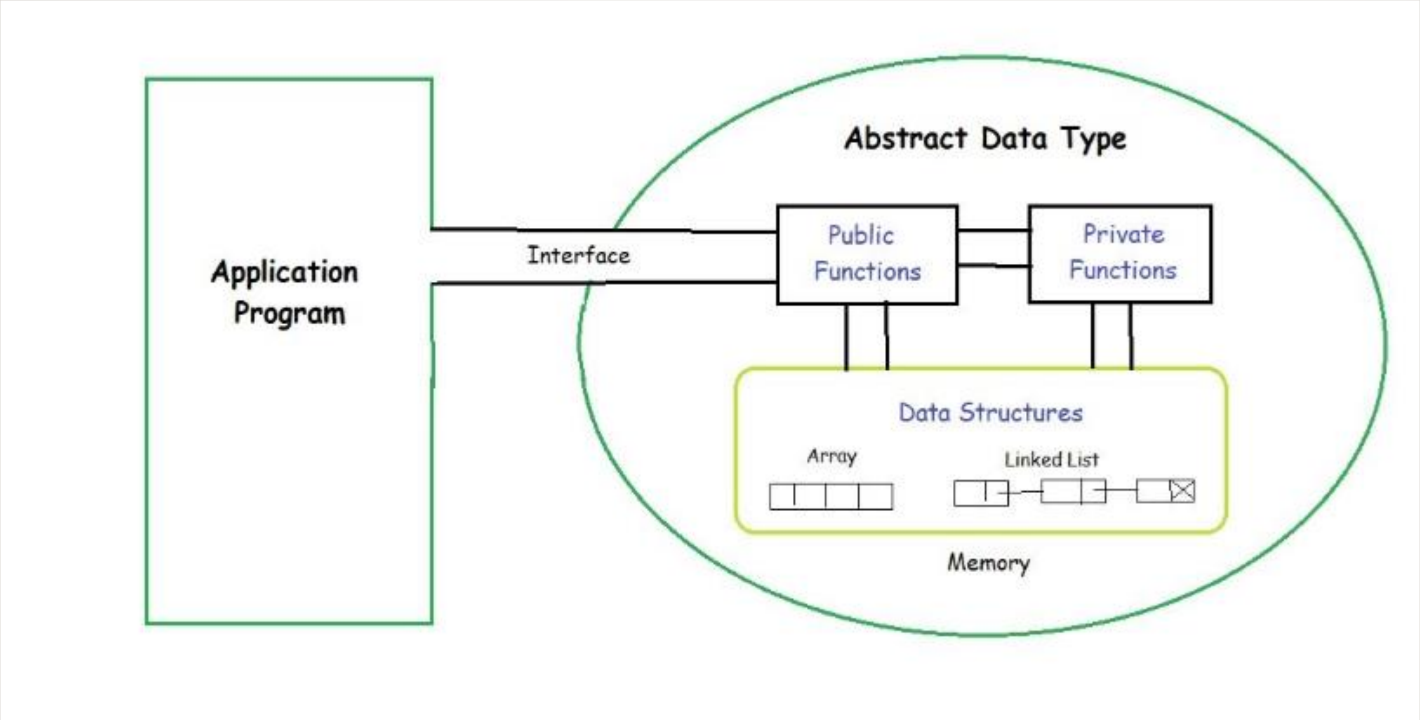
By using structure, we are defining our own type by combining other data types.

```
struct point {
        int x;
        int y;
    };
```

- **Abstract Data type (ADT)** is a type (or class) for objects whose behavior is defined by a set of values 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.

ADTs are like user defined data types which defines operations on values using functions without specifying what is there inside the function and how the operations are performed.

The process of providing only the essentials and hiding the details is known as abstraction.



The program which uses data structure is called a client program. It has access to the ADT i.e. interface
The program which implements the data structure is known as the implementation.

List ADT

The **List ADT Functions** is given below:

- get() – Return an element from the list at any given position.
- insert() – Insert an element at any position of the list.
- remove() – Remove the first occurrence of any element from a non-empty list.
- removeAt() – Remove the element at a specified location from a non-empty list.
- replace() – Replace an element at any position by another element.
- size() – Return the number of elements in the list.
- isEmpty() – Return true if the list is empty, otherwise return false.
- isFull() – Return true if the list is full, otherwise return false.

Stack ADT

- push() – Insert an element at one end of the stack called top.
- pop() – Remove and return the element at the top of the stack, if it is not empty.
- peek() – Return the element at the top of the stack without removing it, if the stack is not empty.
- size() – Return the number of elements in the stack.
- isEmpty() – Return true if the stack is empty, otherwise return false.
- isFull() – Return true if the stack is full, otherwise return false.

Queue ADT

- enqueue() – Insert an element at the end of the queue.
- dequeue() – Remove and return the first element of the queue, if the queue is not empty.
- peek() – Return the element of the queue without removing it, if the queue is not empty.
- size() – Return the number of elements in the queue.
- isEmpty() – Return true if the queue is empty, otherwise return false.
- isFull() – Return true if the queue is full, otherwise return false.

Features of ADT:

- **Abstraction:** The user does not need to know the implementation of the data structure only essentials are provided.
- **Better Conceptualization:** ADT gives us a better conceptualization of the real world.
- **Robust:** The program is robust and has the ability to catch errors.
- **Encapsulation**: ADTs hide the internal details of the data and provide a public interface for users to interact with the data. This allows for easier maintenance and modification of the data structure.
- **Data Abstraction**: ADTs provide a level of abstraction from the implementation details of the data. Users only need to know the operations that can be performed on the data, not how those operations are implemented.
- **Data Structure Independence**: ADTs can be implemented using different data structures, such as arrays or linked lists, without affecting the functionality of the ADT.
- **Information Hiding:** ADTs can protect the integrity of the data by allowing access only to authorized users and operations. This helps prevent errors and misuse of the data.
- **Modularity**: ADTs can be combined with other ADTs to form larger, more complex data structures. This allows for greater flexibility and modularity in programming.

Advantages:

- **Encapsulation**: ADTs provide a way to encapsulate data and operations into a single unit, making it easier to manage and modify the data structure.
- **Abstraction**: ADTs allow users to work with data structures without having to know the implementation details, which can simplify programming and reduce errors.
- **Data Structure Independence**: ADTs can be implemented using different data structures, which can make it easier to adapt to changing needs and requirements.
- **Information Hiding**: ADTs can protect the integrity of data by controlling access and preventing unauthorized modifications.
- **Modularity**: ADTs can be combined with other ADTs to form more complex data structures, which can increase flexibility and modularity in programming.

Disadvantages:

- **Overhead**: Implementing ADTs can add overhead in terms of memory and processing, which can affect performance.
- **Complexity**: ADTs can be complex to implement, especially for large and complex data structures.
- **Learning** Curve: Using ADTs requires knowledge of their implementation and usage, which can take time and effort to learn.
- **Limited Flexibility:** Some ADTs may be limited in their functionality or may not be suitable for all types of data structures.
- **Cost**: Implementing ADTs may require additional resources and investment, which can increase the cost of development.