Lecture 02-03

- The Concept of Abstraction
- Abstract Data Type

IT205: Data Structures (AY 2023/24 Sem II Sec B) — **Dr. Arpit Rana**

The Concept of Abstraction

Abstraction means ***separating*** the details of ***how something works*** with ***how to use it***.

- This allows for the separation of two groups of people with different goals:
    - the ***creators*** of an entity
      they are responsible for designing, building, and implementing an entity
    - the ***clients*** (or users) of that entity
      they are responsible for using it.

The Concept of Abstraction

Abstraction means ***separating*** the details of ***how something works*** with ***how to use it***.

- The set of rules (implicit or explicit) governing how clients can interact with an entity form an ***interface***.
    - part of the creator's work is to design the interface, while
    - clients are responsible for learning the interface in order to interact with it.

Abstraction in Programs

A program is the combination of data structures and algorithms. Thus, programs may provide two forms of abstraction:

- **Functional abstraction**

  By allowing the client to call a function written by the creator without necessarily understanding how it is implemented.

  Interface: function header and ***docstring***

Abstraction in Programs: Functional abstraction

```
int main() {
    int ans;
    ans = factorial(5);
    cout << ans << endl;
    return 0;
}
```

Client of factorial()

Implementation 1

```
int factorial(int n) {
    if (n == 0)
        return 1;
    return n * factorial(n-1);
}
```

Implementation 2

```
int factorial(int n) {
    int i, result = 1;
    for (i = 2; i <= n; i++)
        result *= i;
    return result;
}
```

Abstraction in Programs: Functional abstraction

```
int main() {
    int ans;
    ans = factorial(5);
    cout << ans << endl;
    return 0;
}
```

Client of factorial()

- `main()` needs to know
  - `factorial()`'s purpose
  - Its parameters and return value
  - Its limitations, 0 ≤ n ≤12 for `int`
- `main()` does not need to know
  - `factorial()`'s internal coding
- Different `factorial()` coding
  - Does not affect its clients!
- We can build a wall to shield `factorial()` from `main()`!

Abstraction in Programs

A program is the combination of data structures and algorithms. Thus, programs may provide two forms of abstraction:

- **Data abstraction**

  To hide information about data and how it is represented in the program.

  Interface: the set of operations defined on that data.

Abstraction in Programs: Data abstraction

```
int main() {
     BankAccount account(1000.0);

     double currentBalance =
     account.getBalance();

     cout << "Current Balance: " <<
     currentBalance << endl;

     return 0;
}
```

Using BankAccount

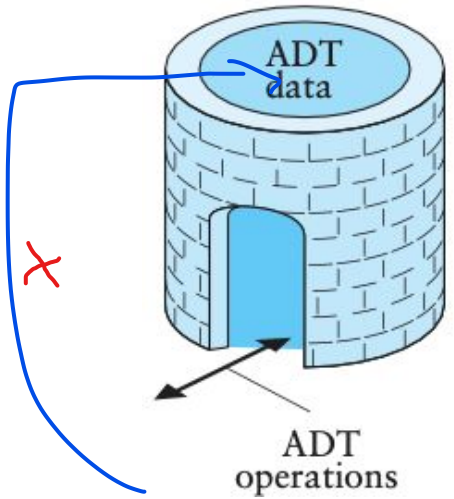
```
class BankAccount {
     private:
          double balance;
     public:
          void deposit(double amount){
          ...}
          bool withdraw(double amount){
          ...}
          double getBalance(){
               return balance;
          }
};
```

Defining BankAccount

Abstract Data Types (ADTs)

ADTs are the ***mathematical concept*** that defines ***what*** data is stored, what we can do with this data—and ***not the how*** a computer actually stores this data or implements these operations.

- They are the end result of the *data abstraction*
- They are independent of any one specific programming language.
- ADT operations provides an ***interface*** to data structure and a secure access.



Abstract Data Types (ADTs)

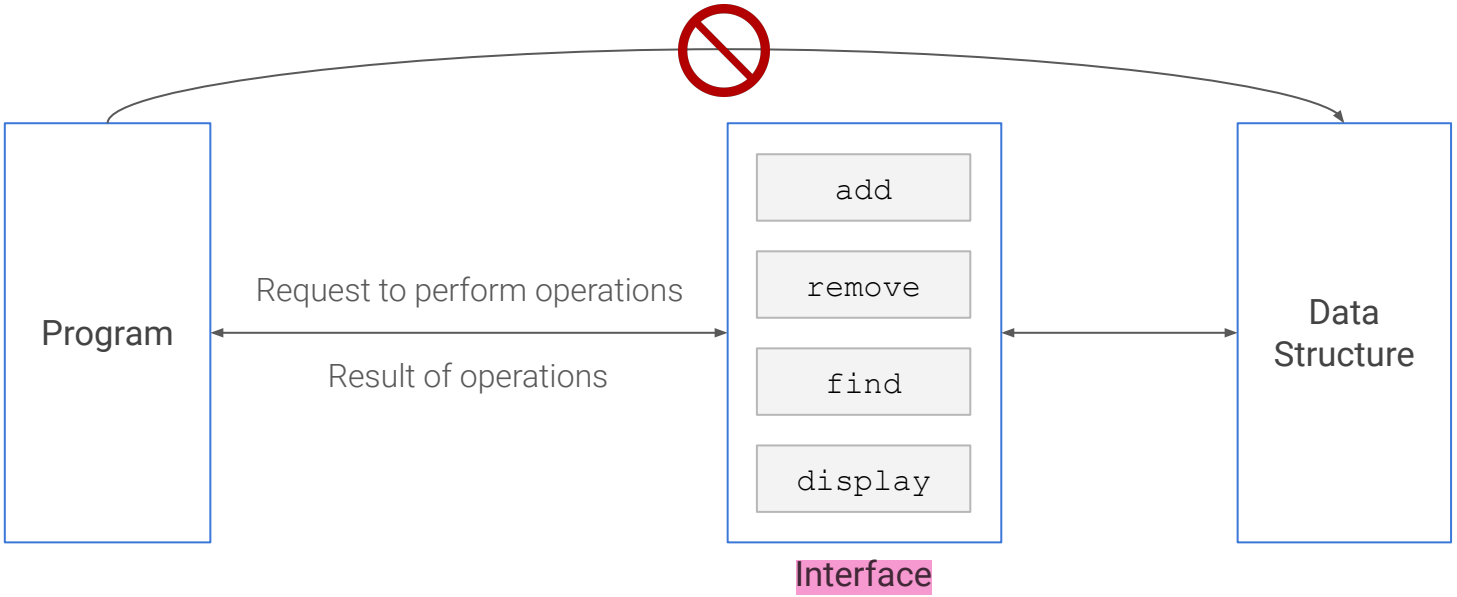
For example,

- An ***integer***, it has properties of being a counting number and various mathematical operations can be performed on it.
- An ***array*** is an abstract idea. We will see that it keeps an ordered collection of homogeneous items. Operations include traversing/indexing, etc.
- We can use C++ ***string*** objects and their operators (length, at, and so on).

We don't know (or care) how these are actually implemented.

Abstract Data Types (ADTs)

- Client applications (i.e., programs) should not: use the underlying data structure directly and also should not depend on implementation details.



Abstract Data Types (ADTs): Components

ADTs are the ***mathematically specified*** entity that defines

- a set of its ***instances,*** a collection of values for the ADT; VALUE HOLDER ex.class
- a specific ***interface***, a collection of signatures of operations that can be invoked on an instance; PERTICULAR OPERTATION THAT CAN BE USE ON THAT VARIABLE
- a set of ***axioms***, that define the ***behaviour*** of the operations: LIKE RULES
  - ***preconditions***: any constraints on the operation's data (input parameters) that must be satisfied before the operation can be applied — Client side;
  - ***postconditions***: any conditions that will become true after performing an operation — Creator side.

Abstract Data Types (ADTs): Components

ADT *interface* may include three types of methods:

- *Constructors/ Initializers*, to create the instances of the data type
- *Access methods*, to access the elements of the data type
- *Manipulation methods*, to manipulate or modify the data type

Dynamic Sets: As an ADT Example

An example dynamic set ADT

Return type

- Methods:
  - *New(): ADT*
  - *IsEmpty(S: ADT): boolean*
  - *Insert(S: ADT, v: element): ADT*
  - *Delete(S: ADT, v: element): ADT*
    precondition: IsEmpty(S) == false
  - *IsIn(S: ADT, v: element): boolean*
    precondition: IsEmpty(S) == false

perameters

is integer

- *New()*, constructor
- *Insert()* and *Delete()*, manipulation methods
- *IsEmpty(), IsIn()*, access methods

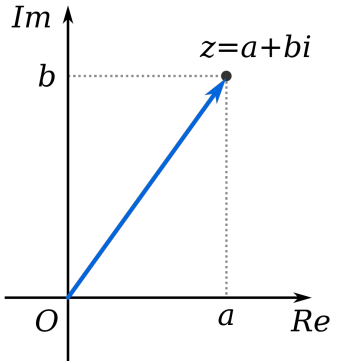
Rational Numbers: As an ADT Example

An example rational number ADT

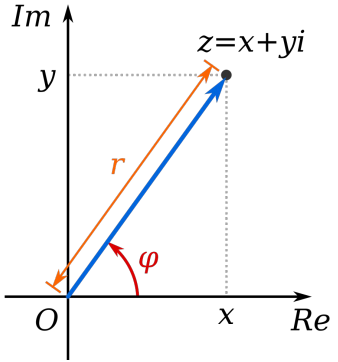
- Methods:
  - *New(n: integer, d: integer): ADT,*
    precondition: $d \neq 0$
  - *Add(r_1: ADT, r_2: ADT): ADT*
    postcondition: $r.n = r_1.n * r_2.d + r_2.n * r_1.d$
    $r.d = r_1.d * r_2.d$
  - *Multiply(r_1: ADT, r_2: ADT): ADT*
    postcondition: $r.n = r_1.n * r_2.n$
    $r.d = r_1.d * r_2.d$
  - *Equal(r_1: ADT, r_2: ADT): boolean – what about its postcondition?*

Complex Numbers: As an ADT Example

Common representations of a complex number:



Rectangular Form
(a + *i*b)



Polar Form
r(cos φ + *i**sin φ)

Complex Numbers: As an ADT Example

An example complex number ADT

- Methods:
  - *New(real: Real, img: Real): ADT,*
  - *Add(c_1: ADT, c_2: ADT): ADT*
  - *Subtract(c_1: ADT, c_2: ADT): ADT*
  - *Multiply(c_1: ADT, c_2: ADT): ADT*
  - *Divide(c_1: ADT, c_2: ADT): ADT*
  - *Equal(c_1: ADT, c_2: ADT): boolean*
  - *Conjugate(c_1: ADT): ADT*
  - *getMagnitude(c_1: ADT): Real*

Exercise

- Specify Complex Number ADT for both the representations and see which operation is more efficient in which representation. Your specification of ADT should be independent of its representation.
- Create an ADT for the Date, i.e., dd/mm/yy format.

Important Properties of ADTs

Specification

Specifying the names, parameters, and return values of the supported operations -

- without specifying how the operations are performed, and
- without specifying how the data is internally represented.

Implementation

- Providing the complete definition of all operations declared in the specification.
- Providing the details as how data structures are internally represented.

Important Properties of ADTs

BETWEEN THEM WE HAVE INTERFACE

Specification and implementation are disjoint:

- One specification
- One or more implementations
  - Using different data structure
  - Using different algorithms

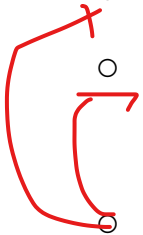
Clients of ADT:

- are aware of the specification ***only***
  - use only base on the specified operations
- do not care / need not know about the actual implementation
  - i.e. different implementations do *not* affect the client

When to Use ADTs?

- When we need to operate on data that are not directly supported by the language
  - E.g. Complex Number, Module Information, Bank Account, etc

- Simple Steps:
  - Design an ADT
  - Carefully specify all operations needed
    - Ignore any implementation related issues
  - Implement them



Next Lecture

- Analysis of Algorithms