Abstract Data Types

CMPT 115/117 lecture slides

Notes written by Mark Eramian, Ian McQuillan, Michael Horsch, Lingling Jin, and Dmytro Dyachuk

Objectives

- Explain what a data type is in your own words.
- Give an example of a data type.
- Explain what a data structure is in your own words.
- Give an example of a data structure.
- Explain what an abstract data type is in your own words.
- Give an example of an abstract data type.
- List the three aspects of an ADT.

Outline

Motivation: writing larger applications

- A good strategy: separate large software projects into several components.
  - Some components can be re-used in (or from) other projects.
  - Design decisions should be made for each component, without affecting other components.
- Abstract Data Types help us design such components.
- Abstract Data Types (ADTs) provide guidance for good design.
  - Reduce the number of design decisions
  - Facilitate division of labour
  - Enable modularity
  - Allow code reuse

Abstract Data Types: informally

Informally, an Abstract Data Type (ADT) provides the following ideas:

- An ADT defines a way to store data at a useful level of abstraction (not too much detail)
- An ADT defines what kinds of things can be done with the data, e.g., adding data to a list
- An ADT hides details about how these things are done

We will explore these ideas in detail in this lecture.

ADT: First example

Consider the idea of a *list*

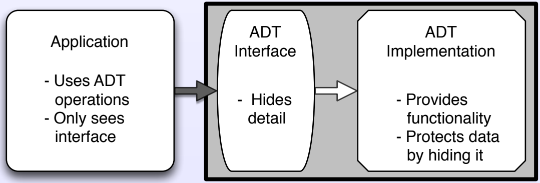
- Use to store a collection of related information.
- The list implies a sequence, which could be meaningful
  - E.g., a TODO list: tasks ordered according to importance
- There are certain things we want to do with lists:
  - Add items, search for items, remove items, and more. . .
- Ideal properties of lists as ADT:
  - Re-usable in many programs
  - The details of how lists are programmed is hidden
  - The operations (add, search, remove, sort, etc) give us functionality we want.
  - Only the operations provided by the implementation are visible.

ADT: Getting a fuller picture

- An ADT will
  - reveal only what operations are available, as function headers
  - hide the details (encapsulate) about how the operations are programmed
- **Application programmers** building applications only know what the operations do (interface)
- **ADT programmers** implement the operations (implementation), but:
  - they are building robust, efficient, reusable tools
  - they know nothing about how the tool will be used in an application

Three useful concepts for ADTs

1. Encapsulation
   – hide the data in a "black box"
2. Operation Interface
   – description of the controls on the black box
3. Implementation
   – what goes on inside the black box



Levels of Data Abstraction

- Atomic Data
- Composite Data
- Data Types
- Data Structures
- Abstract Data Types

Atomic data vs. Composite data

- **Atomic data** are data which consists of a single piece of information.
    - Examples: a temperature, a letter grade, a reference to some memory
- **Composite data** consists of a collection of multiple pieces of information.
    - Examples: A collection of grades, an (x, y, z) point in 3D space, the name of a file
- At this level of abstraction, we are concerned only with the information, not the representation in a program.

Data Types

A *data type* packages together data and ways of manipulating the data.

Data Type

A data type consists of two parts:

1. a conceptual set of data values
2. operations that can be performed on the data values

A data type can be specific to a language (e.g., C++), or generic across all/many languages. E.g., lists are useful everywhere.

Example Data Types: Integer, and Character

Example (Integer Data Type)

The integer data type can consist of:

data values: $-\infty, \ldots, -2, -1, 0, 1, 2, \ldots, \infty$

operations: $+, -, *, \%, /, \ldots$

Example (Character Data Type)

The character data type can consist of:

data values: $\backslash 0, \ldots,$ 'A','B',...,'a','b',..., '1', '2', ...

operations: $<, >, \leq, \geq, ==, \ldots$

Example Data Types: Person, and 2D Points

We can have data types for composite data too.

Example (Person Data Type)

The Person data type can consist of:

data values: { 15, "Bob", "Buckwheat"}, { 50, "Jack", "Bauer" }, . . .

operations: set/get firstname, set/get lastname, set/get age, ==, . . .

Example (2D Point Data Type)

The 2D Point data type can consist of:

data values: $(0,0),(1,0),(3.7,-8.5)\cdots$

operations: get distance between 2 points, move point, add points together, . . .

Data structures: introduction

A *data structure* is a way to organize a collection of data. The organization reflects the meaning of the data itself, or the purpose to which it will be used.

Data Structure

- A data structure stores data, and also stores structural information about the organization.
- The structure reflects the meaning or purpose of the organization.

We use data structures to build real implementations of ADTs.

Text-book definition of data structures

A *data structure* is an aggregation of data elements into a set with defined relationships.

Data Structure

- A combination of elements in which each element is either a data type or another data structure.
- A set of associations or relationships (structure) between elements.

Data Structures: Familiar examples

Example (Data Structure: Array)

- *Elements:* All the same data type (integer, float, Person, etc,.).
- *Structure:* Elements form a contiguous sequence in which each element is numbered with an index.

Example (Data Structure: Matrix (2D array))

- *Elements:* All the same data type (integer, float, Person, etc,.).
- *Structure:* Elements form a grid in which each element has a row and column position.

We will see additional examples of data structures in this course, such as *lists*, and *trees*.

Abstract Data Types: Formal presentation

- An abstract data type (ADT) is an *abstraction* that permits programmers to make use of the data structure without knowing its internal implementation.
- We know *what* an ADT can do, but *how* it is done is hidden.

Abstract Data Type

An abstract data type consists of:

- One or more data structures (definition of data).
- Definition of operations on the data (the interface).
- Encapsulation (hiding) of data and the implementation of the operations.

ADTs

Example (String ADT)

A string consists of:

the data structure: a sequence of characters

the operations (interface): create, compare, concatenate, etc. (in C++, interface specified in cstring)

the encapsulation: Implementation of operations and data structure are hidden in the string library (data structure might be character array like in C++, might be something else...)

ADTs

Example (Matrix ADT)

A matrix consists of:

the data structure: a 2D array of numbers

the operations (interface): create, edit element, $+, -, /, \times, \ldots$

the encapsulation: details of operations hidden in functions (e.g. `matrixMultiply`). Data structure might be 2D array, might be something else.

ADTs

Even the integers can be viewed as an ADT:

Example (Integer ADT)

An integer consists of:

the data structure: a sequence of bits

the operations (interface): $+, -, /, \times, \ldots, \%, \leq, \geq, ==, \ldots$

the encapsulation: details of operations hidden in hardware. Data structure might be 1's complement, 2's complement, signed magnitude, big endian, little endian...

In C/C++ integers are primitive data since the encapsulation happens in hardware rather than software. But some languages (e.g., Java) have an Integer ADT.

Abstract Data Types - Summary

abstract data type

An *abstract data type* consists of

1. a declaration of data,
2. a declaration of operations,
3. an encapsulation of the data and operations,

where the implementation is hidden from the user.

- The hiding of the implementation is key.
- The user does not need to know the underlying data structure or the details of the implementation of operations in order to use the abstract data type.
- The underlying data structure of an ADT might change... but so long as the interface stays the same, the rest of a program is not affected by the change.

Separating the Interface from the Implementation

- The interface lists the operations of the ADT
  - Full function header (pre-, post-, and return) for every function
  - The body of the functions is omitted from the interface
  - Application programmers use this information only
- The implementation describes every detail about the ADT, including
  - Data structures used
  - Full function definitions for every operation
  - Application programmers are not allowed to use this information in any way.

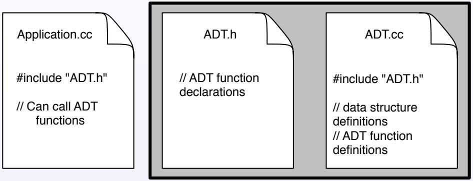
Example: ADT Interface in Pseudocode

```
INTERFACE: an abstract data type for Student

Algorithm PrintStudent (s)
pre: s :: Student
post: the student is printed
returns: nothing

Algorithm MakeStudent (first, last)
pre: first :: String
     last  :: String
post: nothing
returns: a new Student with the provided names
```

Example: ADT Implementation in Pseudocode

```
IMPLEMENTATION: an abstract data type for Student
Student
  String firstname
  String lastname
end Student


Algorithm PrintStudent (s)
pre: s :: Student
post: the student is printed
returns: nothing

    print s.lastname, ", ", firstname, " : student"


Algorithm MakeStudent (first, last)
pre: first :: String
    last  :: String
post: nothing
returns: a reference to a new Student with the provided names

    refToStudent sr ← allocate new Student
    (*sr).firstname ← first
    (*sr).lastname ← last
    return sr
```

Implementing ADTs in C++




```
g++ -o Application -Wall -pedantic Application.cc ADT.cc
```

- `ADT.cc` contains the functions that implement the operations.
- `ADT.h` contains the function headers and typedefs.
- Any `Application.cc` may `#include ADT.h`. This informs the program about the *interface*, but not about the *implementation*.
- All `.cc` files are combined when the program is compiled.
- Further details in the Tutorials.

ADTs Design - Exercise 1

- Describe an abstract data type for a complex number*.
    - declaration of data
    - a list of at least three operations you might want to perform on that data

*A *complex number* is a number that can be expressed in the form $a + bi$, where a and b are real numbers and i is the imaginary unit, which satisfies the equation $i^2 = -1$. In this expression, a is the real part and b is the imaginary part of the complex number.

ADTs Design - Exercise 1 Solution

- Complex number ADT:
  - Data declaration:

```
Complex
    Float real     // the real part of the number
    Float image    // the imaginary part of the number
end Complex
```

  - Operations:
    1. add two numbers
    2. subtract two numbers
    3. multiply two numbers
    4. divide two numbers
    5. exponentiate a number
    6. magnitude (absolute value) of a number
    7. negate a number
    8. ...

ADTs Design - Exercise 2

- Describe an abstract data type for a song in an MP3 library.
  - declaration of data
  - a list of at least three operations you might want to perform on that data

ADTs Design - Exercise 2 Solution

- Song ADT:
  - Data declaration:

```
Song
    refToCharacter title      // string
    refToCharacter artist     // string
    int playCount
end Song
```

  - Operations:
    1. play song
    2. increase play count
    3. reset play count to zero
    4. ...

Implementing ADTs in Java and Object-oriented C++

- In Java and Object-oriented C++, ADTs are usually implemented using classes.
- A `class` is like a `record type` in which you can place functions, in addition to data.
- This permits the packing of the interface and the implementation of an ADT in a single programming language entity while keeping them separate from the point of view of the user.
- We will show you how to use classes in C++ to implement ADTs, and other concepts of Object Oriented Programming, near the end of this course.

Perspective

- This topic defined some abstract terms somewhat vaguely
- These ideas will be clearer in the next few weeks.
- Key outcome: ADTs protect data from unconstained manipulation, and enhance developers' productivity.
- Before this topic, maybe you were a programmer.
- If you embrace this principle, you are on your way to becoming a software developer.