# Chapter 3:
# Data Abstraction: The Walls

**Data Abstraction & Problem Solving with C++**

**Fifth Edition**

**by Frank M. Carrano**




## Abstract Data Types

- Modularity
  - Keeps the complexity of a large program manageable by systematically controlling the interaction of its components
  - Isolates errors
  - Eliminates redundancies




## Abstract Data Types

- Modularity (Continued)
  - A modular program is
    - Easier to write
    - Easier to read
    - Easier to modify




## Abstract Data Types

- Functional abstraction
  - Separates the purpose and use of a module from its implementation
  - A module's specifications should
    - Detail how the module behaves
    - Be independent of the module's implementation




## Abstract Data Types

- Information hiding
  - Hides certain implementation details within a module
  - Makes these details inaccessible from outside the module




## Abstract Data Types

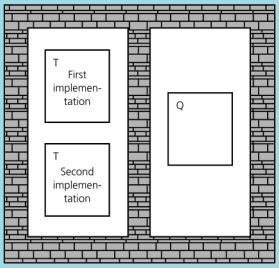


***Figure 3-1***

Isolated tasks: the implementation of task *T* does not affect task *Q*

## Abstract Data Types

- The isolation of modules is not total
  - A function's specification, or contract, governs how it interacts with other modules

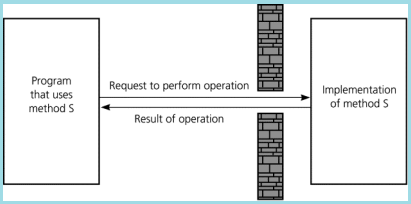


***Figure 3-2*** A slit in the wall




## Abstract Data Types

- Typical operations on data
  - Add data to a data collection
  - Remove data from a data collection
  - Ask questions about the data in a data collection




## Abstract Data Types

- Data abstraction
  - Asks you to think *what* you can do to a collection of data independently of *how* you do it
  - Allows you to develop each data structure in relative isolation from the rest of the solution
  - A natural extension of functional abstraction




## Abstract Data Types

- Abstract data type (ADT)
  - An ADT is composed of
    - A collection of data
    - A set of operations on that data
  - Specifications of an ADT indicate
    - What the ADT operations do, not how to implement them
  - Implementation of an ADT
    - Includes choosing a particular data structure




## ADT vs Data Structure

- ADT
  - Collection of data
  - Set of operations on the data
  - Example: list (we will define shortly)
- Data Structure
  - Construct within programming language
  - Stores a collection of data
  - Example: array




## Abstract Data Types

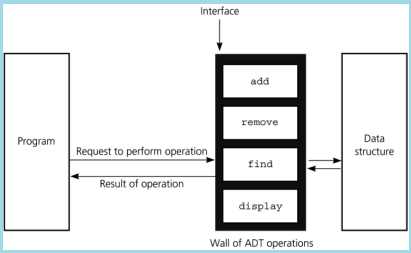


***Figure 3-4***

A wall of ADT operations isolates a data structure from the program that uses it

## Specifying ADTs: The ADT List

- Except for the first and last items in a list, each item has a unique predecessor and a unique successor
- Head (or front) does not have a predecessor
- Tail (or end) does not have a successor




## The ADT List

- Items are referenced by their position within the list
- Specifications of the ADT operations
  - Define an operation contract for the ADT list
  - Do not specify how to store the list or how to perform the operations
- ADT operations can be used in an application without the knowledge of how the operations will be implemented




## The ADT List

- ADT List Operations
  - Create an empty list
  - Destroy a list
  - Determine whether a list is empty
  - Determine the number of items in a list
  - Insert an item at a given position in the list
  - Delete the item at a given position in the list
  - Look at (retrieve ) the item at a given position in the list




## The ADT List

- Operation Contract for the ADT List

createList()

destroyList()

isEmpty():boolean {query}

getLength():integer {query}

insert(in index:integer, in newItem:ListItemType, out success:boolean)

remove(in index:integer, out success:boolean)

retrieve(in index:integer, out dataItem:ListItemType, out success:boolean) {query}




## The ADT List

- Pseudocode to create the list

*milk*, *eggs*, *butter*

```
aList.createList()
aList.insert(1, milk, success)
aList.insert(2, eggs, success)
aList.insert(3, butter, success)
```




## The ADT List

*milk*, *eggs*, *butter*

- Insert *bread* after *milk*

```
aList.insert(2, bread, success)
```

*milk*, *bread*, *eggs*, *butter*

- Insert *juice* at end of list

```
aList.insert(5, juice, success)
```

*milk*, *bread*, *eggs*, *butter*, *juice*

## The ADT List

*milk*, *bread*, *eggs*, *butter*, *juice*

- Remove *eggs*

```
aList.remove(3, success)
```

*milk*, *bread*, *butter*, *juice*

- Insert *apples* at beginning of list

```
aList.insert(1, apples, success)
```

*apples*, *milk*, *bread*, *butter*, *juice*




## The ADT List

*apples*, *milk*, *bread*, *butter*, *juice*

- Pseudocode function that displays a list

```
displayList(in aList:List)
  for (position = 1 to aList.getLength())
  { aList.retrieve(position, dataItem,
                   success)
    Display dataItem
  } // end for
```




## The ADT List

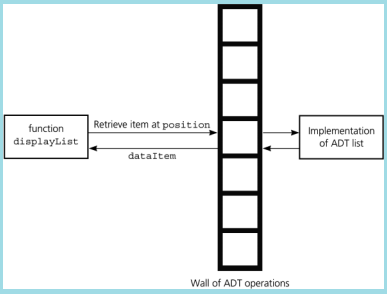


***Figure 3-7***

The wall between ***displayList*** and the implementation of the ADT list




## The ADT Sorted List

- The ADT sorted list
  - Maintains items in sorted order
  - Inserts and deletes items by their values, not their positions




## The ADT Sorted List

- Operation Contract for the ADT Sorted List

```
sortedIsEmpty():boolean{query}
sortedGetLength():integer{query}
sortedInsert(in newItem:ListItemType,
             out success:boolean)
sortedRemove(in index:integer,
             out success :boolean)
sortedRetrieve(in index:integer,
               out dataItem:ListItemType,
               out success :boolean){query}
locatePosition(in anItem:ListItemType,
               out isPresent:boolean):integer{query}
```




## Designing an ADT

- The design of an ADT should evolve naturally during the problem-solving process
- Questions to ask when designing an ADT
  - What data does a problem require?
  - What operations does a problem require?

## Designing an ADT

- For complex abstract data types, the behavior of the operations must be specified using axioms
  - Axiom: A mathematical rule
  - Example: (aList.createList()).size() = 0




## Implementing ADTs

- Choosing the data structure to represent the ADT's data is a part of implementation
  - Choice of a data structure depends on
    - Details of the ADT's operations
    - Context in which the operations will be used




## Implementing ADTs

- Implementation details should be hidden behind a wall of ADT operations
  - A program (client) should only be able to access the data structure by using the ADT operations




## Implementing ADTs

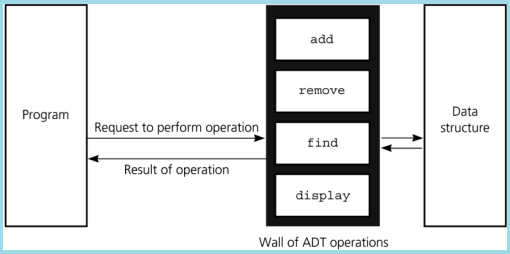


***Figure 3-8***

ADT operations provide access to a data structure




## Implementing ADTs

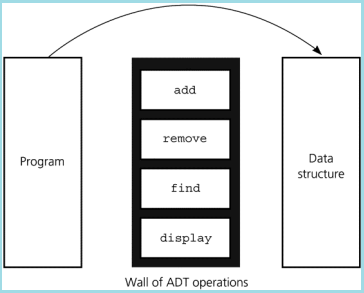


***Figure 3-9*** Violating the wall of ADT operations