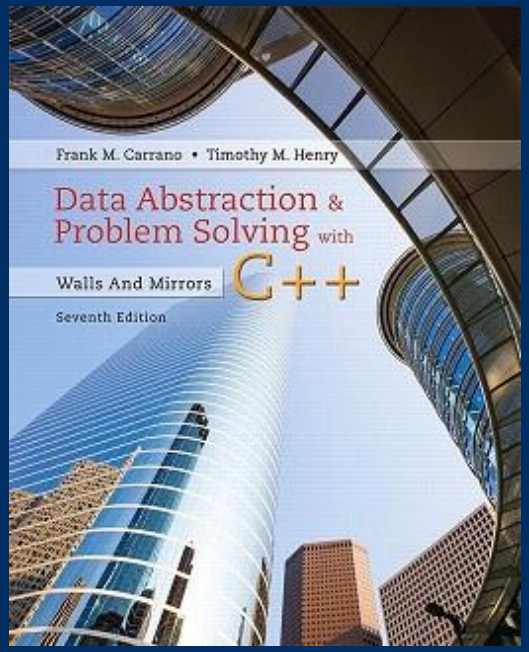
Chapter 1

# Data Abstraction: The Walls

## CS 302 - Data Structures

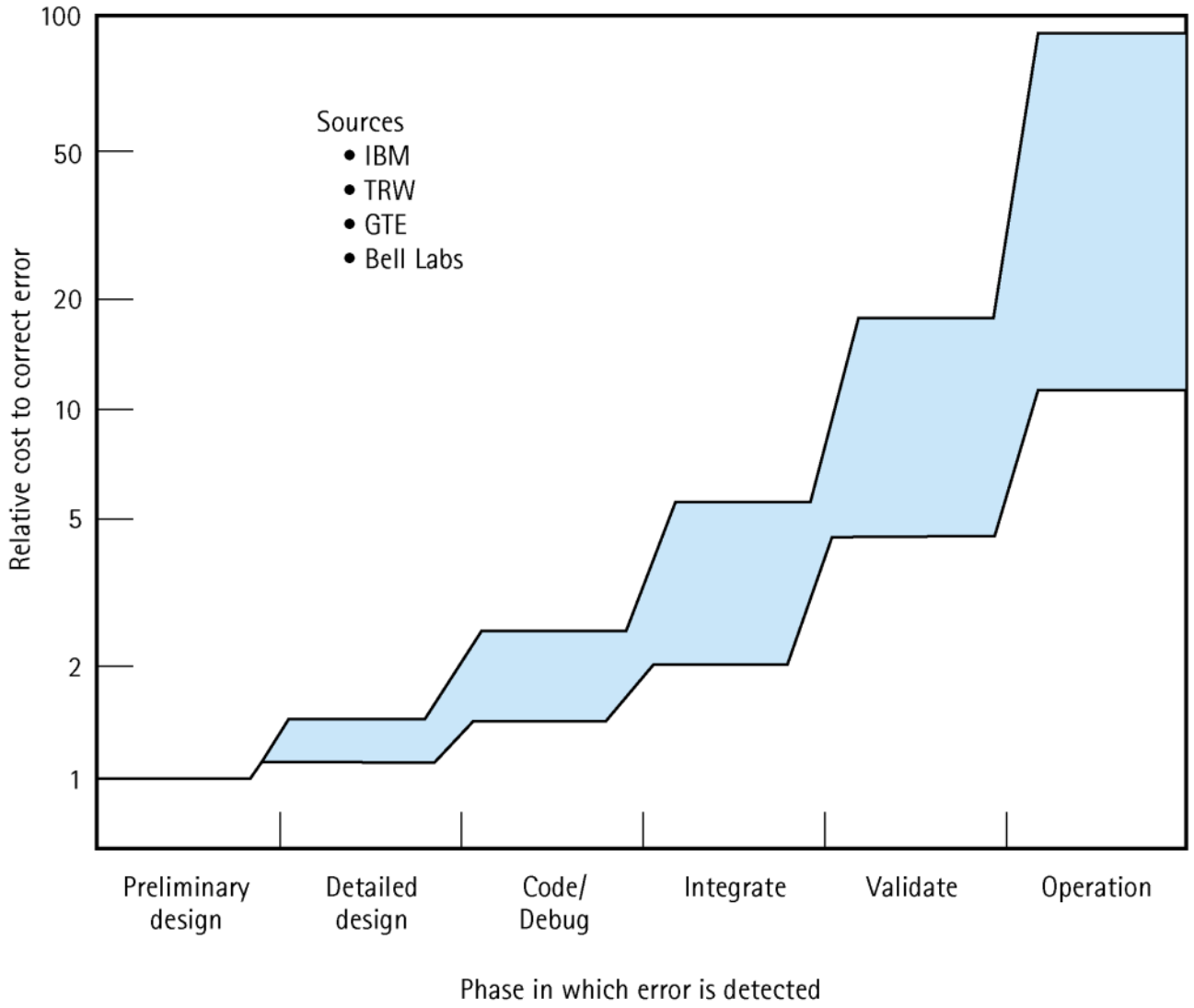



Modified from authors' slides

# The Software Life Cycle

- Problem analysis
- Requirements elicitation
- Software specification
- High- and low-level design
- Implementation
- Testing and Verification
- Delivery
- Operation
- Maintenance

# Cost of an Error



# Controlling Errors

- **Robustness**: The ability of a program to recover following an error
  - the ability of a program to continue to operate within its environment
- **Preconditions:** Assumptions that must be true on entry into an operation or function for the postconditions to be guaranteed
- **Postconditions:** Statements that describe what results are to be expected at the exit of an operation or function
  - assuming that the preconditions are true

# Procedural vs. Object-Oriented Code

- “Read the specification of the software you want to build.
- Underline the verbs if you are after procedural code,
- the nouns if you aim for an object-oriented program.”

Grady Booch, “What is and isn’t Object Oriented Design,” 1989

# Procedural vs. Object-Oriented Code

The “Amazing Lunch Indicator” is a GPS-based mobile application which helps people to find the closest restaurants based on the user’s current position and other specification like price, restaurant type, dish and more. The application should be free to download from either a mobile phone application store or similar services.

Restaurant owners can provide their restaurant information using the web-portal. This information will act as the bases for the search results displayed to the user. An administrator also uses the web-portal in order to administer the system and keep the information accurate. The administrator can, for instance, verify restaurant owners and manage user information.

Furthermore, the software needs both Internet and GPS connection to fetch and display results. All system information is maintained in a database, which is located on a web-server. The software also interacts with the GPS-Navigator software which is required to be an already installed application on the user’s mobile phone. By using the GPS-Navigator, users can view desired restaurants on a map and be navigated to them. The application also has the capability of representing both summary and detailed information about the restaurants.

# Approaches to Building Manageable Modules

## PROCEDURAL DECOMPOSITION

Divides the problem into more easily handled subtasks, until the functional modules (subproblems) can be coded

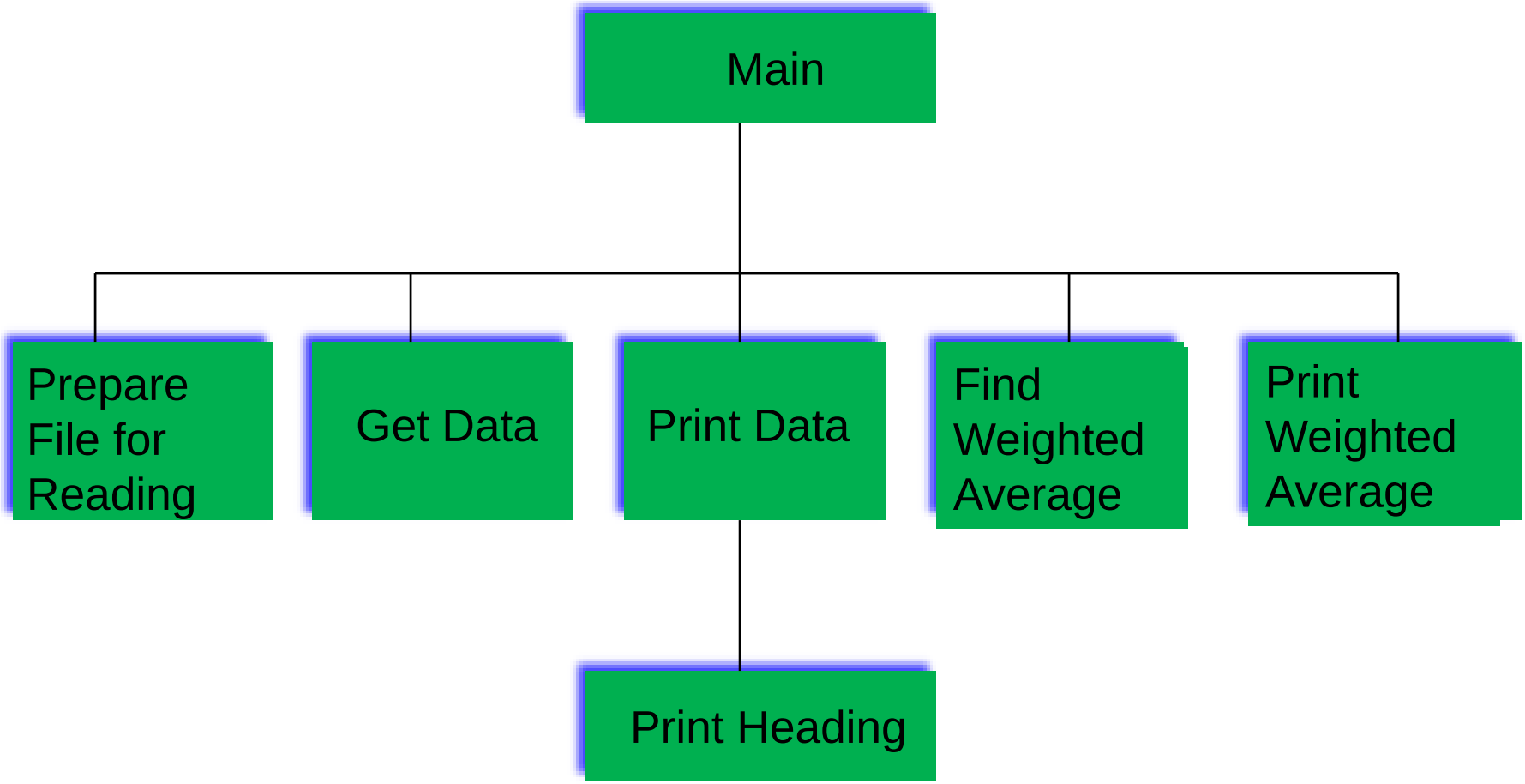
FOCUS ON: processes

## OBJECT-ORIENTED DESIGN

Identifies various objects composed of data and operations, that can be used together to solve the problem

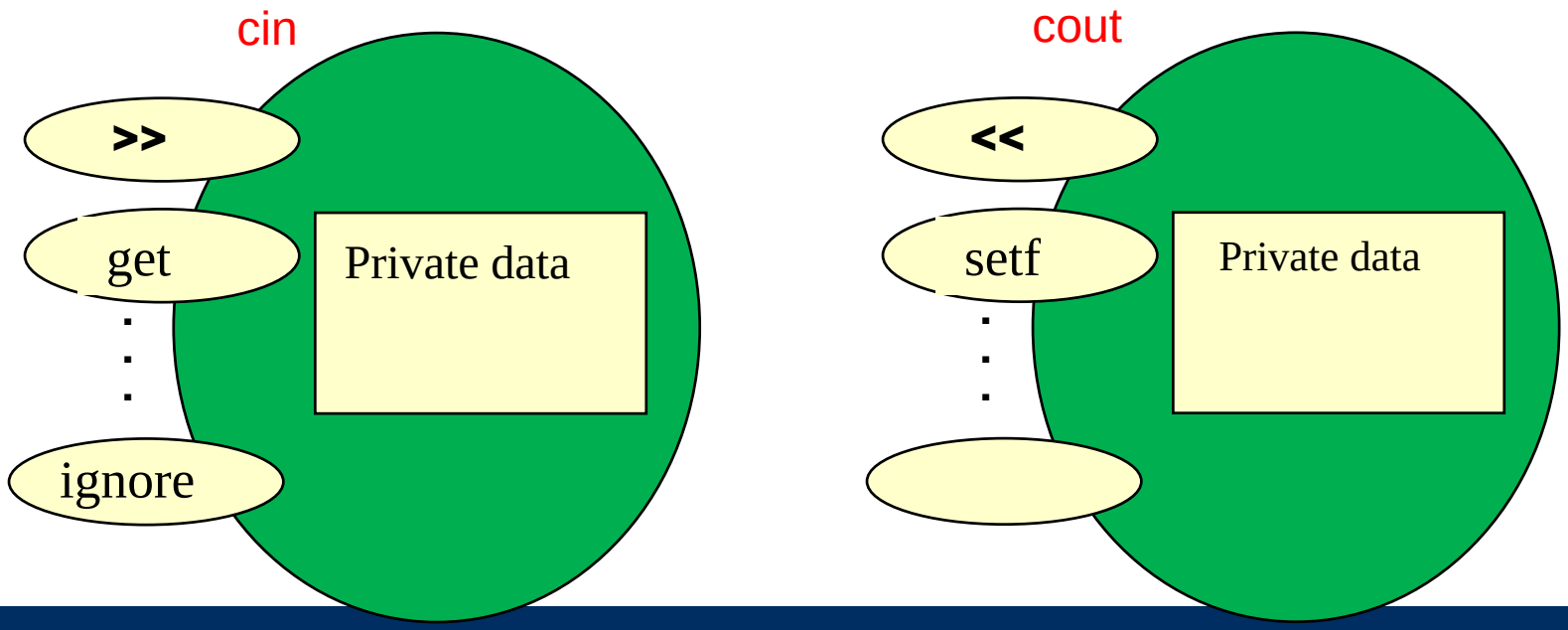
FOCUS ON: data objects

# Functional Design Modules



# Object-Oriented Design

- A technique for developing a program in which the solution is expressed in terms of objects
  - self- contained entities composed of data and operations on that data



# Object-Oriented Concepts

- Object-oriented analysis and design (OOAD)
  - Process for solving problems
- Solution
  - Computer program consisting of system of interacting classes of objects
- Object
  - Has set of characteristics, behaviors related to solution

# Object-Oriented Analysis & Design

- Requirements of a solution
  - What solution must be doing?
- Object-oriented design
  - Describe solution to problem
  - Express solution in terms of software objects
  - Create one or more models of solution

# Principles of Object-Oriented Programming

- **Encapsulation**:
  - Objects combine data and operations
- **Inheritance**:
  - Classes inherit properties from other classes
- **Polymorphism**:
  - Objects determine appropriate operations at execution

# Cohesion

- Each module should perform one well-defined task

- Benefits
  - Well named, self-documenting
  - Easy to reuse
  - Easier to maintain
  - More robust

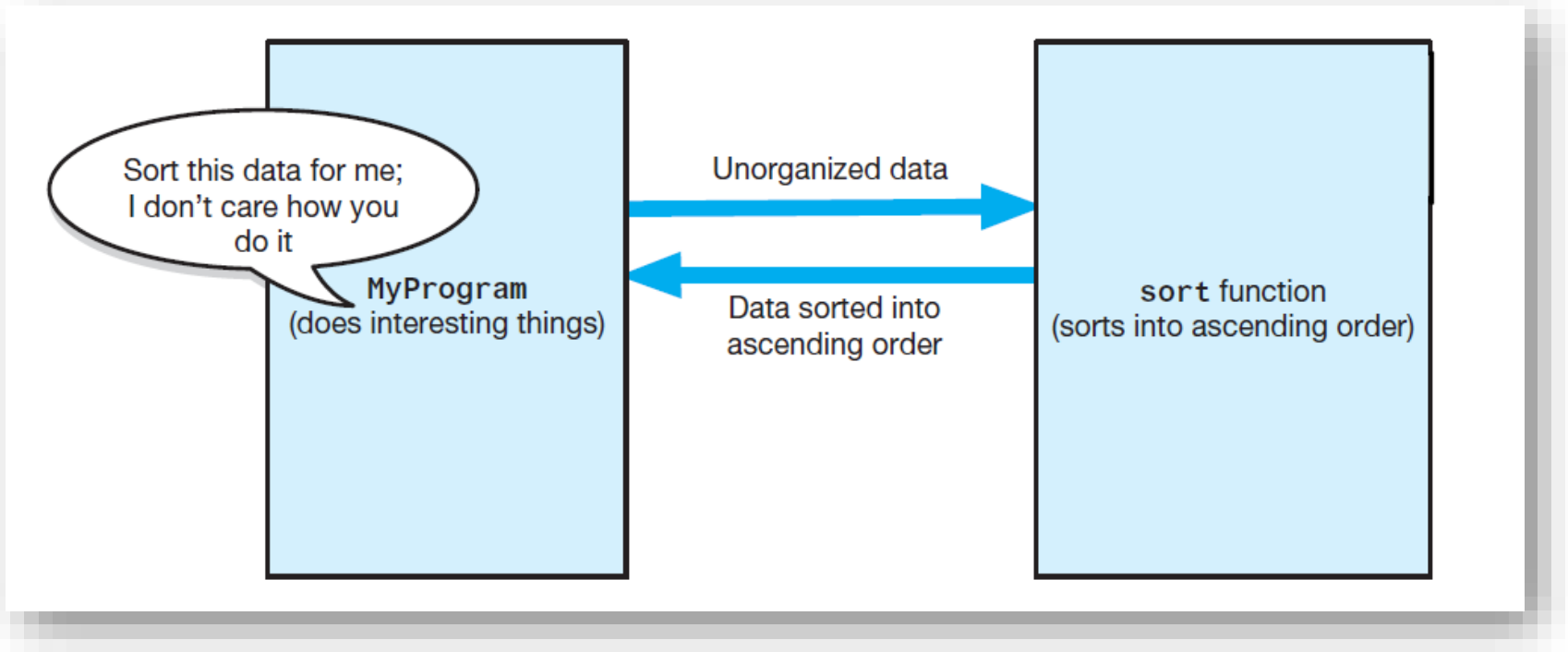
# Coupling

- Measure of dependence among modules
- Dependence
  - Sharing data structures or calling each other's methods
- Modules should be loosely coupled
  - Highly coupled modules should be avoided

# Coupling

- Benefits of loose coupling in a system
  - More adaptable to change
  - Easier to understand
  - Increases reusability
  - Has increased cohesion

# Specifications




- The task sort is a module separate from the MyProgram module

# Operation Contracts

- Documents
  - How method can be used
  - What limitations it has
- Specify
  - Purpose of modules
  - Data flow among modules
  - Pre-, post-condition, input, output of each module

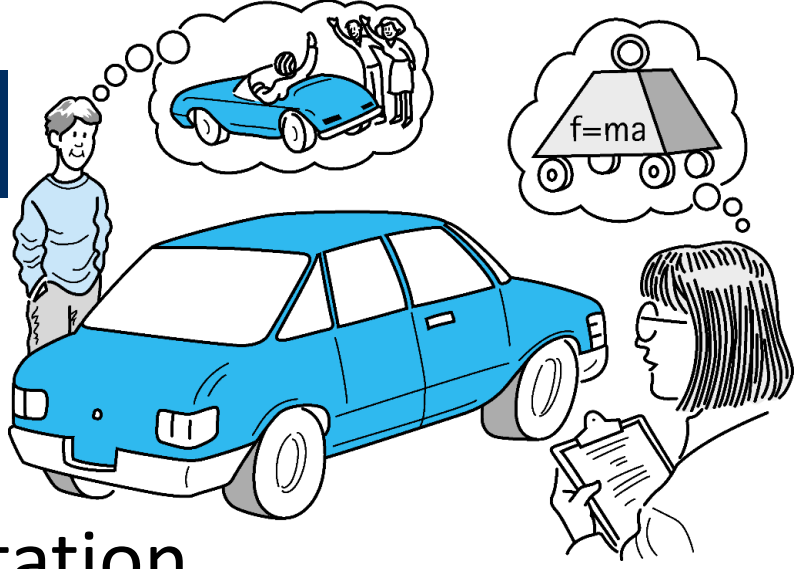
# Unusual Conditions

Options

- Assume they never happen
- Ignore invalid situations
- Guess at client's intent
- Return value that signals a problem
- Throw an exception

!!! Has security implications !!!

# Abstraction



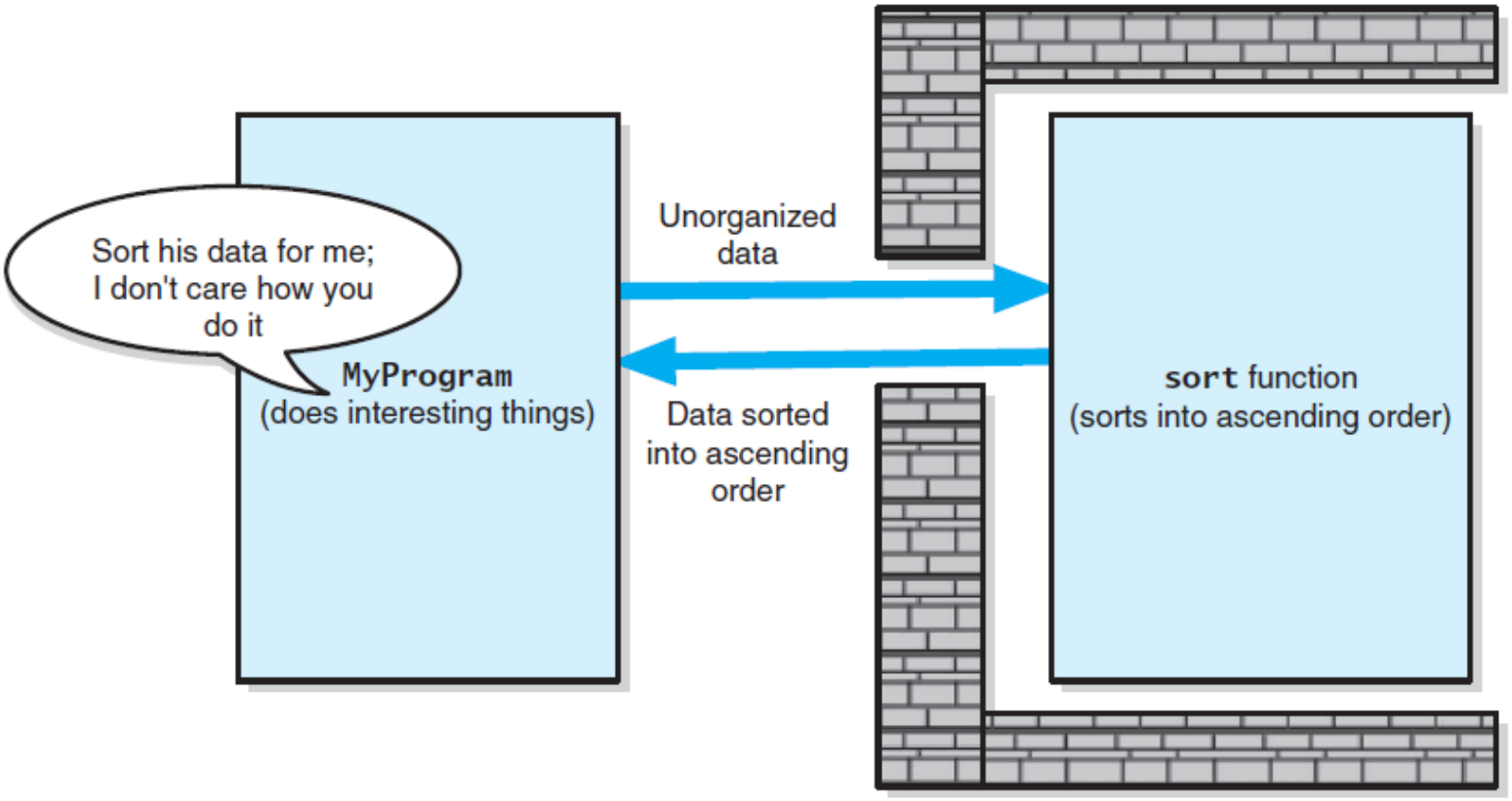

- Separates purpose of a module from its implementation
- Specifications do not indicate how to implement
  - Able to use without knowing implementation
- Think “what” not “how”

# Information Hiding

- Abstraction helps identify details that should be hidden from public view
  - Ensured no other module can tamper with these hidden details.
- Isolation of the modules cannot be total, however
  - Client must know what tasks can be done, how to initiate a task

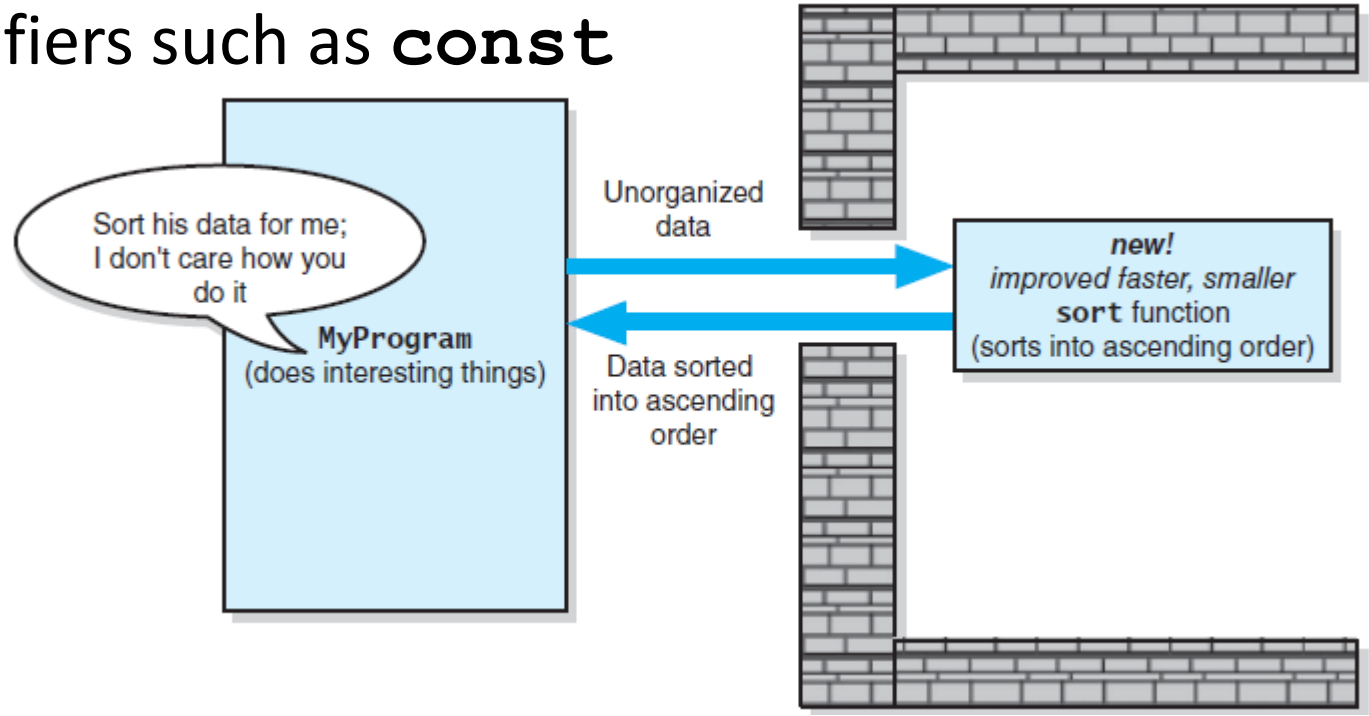
# Information Hiding

- Tasks communicate through a slit in wall



# Minimal and Complete Interfaces

- **Signature**
  - function's name;
  - the number, order, and type of arguments;
  - any qualifiers such as **`const`**



# Minimal and Complete Interfaces

- Interface for a class made up of publicly accessible methods and data
- Complete interface for a class
  - Allows programmer to accomplish any reasonable task
- Minimal interface for a class
  - Contains method if and only if that method is essential to class's responsibilities

# Data

- The representation of information in a manner suitable for communication or analysis by humans or machines

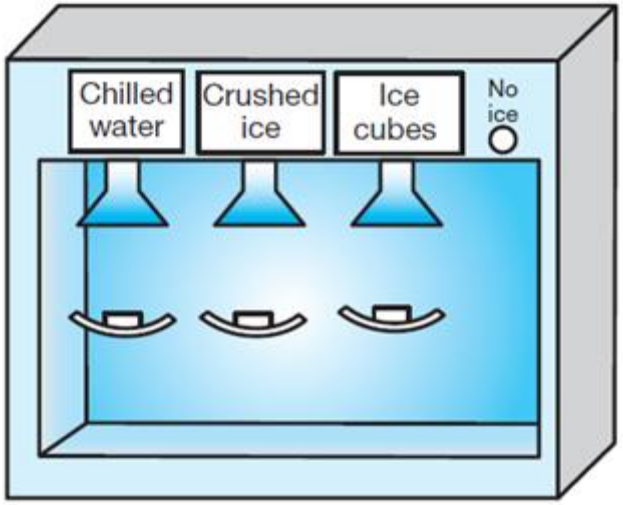
- Data are the nouns of the programming world:
  - The objects that are manipulated
  - The information that is processed

# Abstract Data Types (ADT)

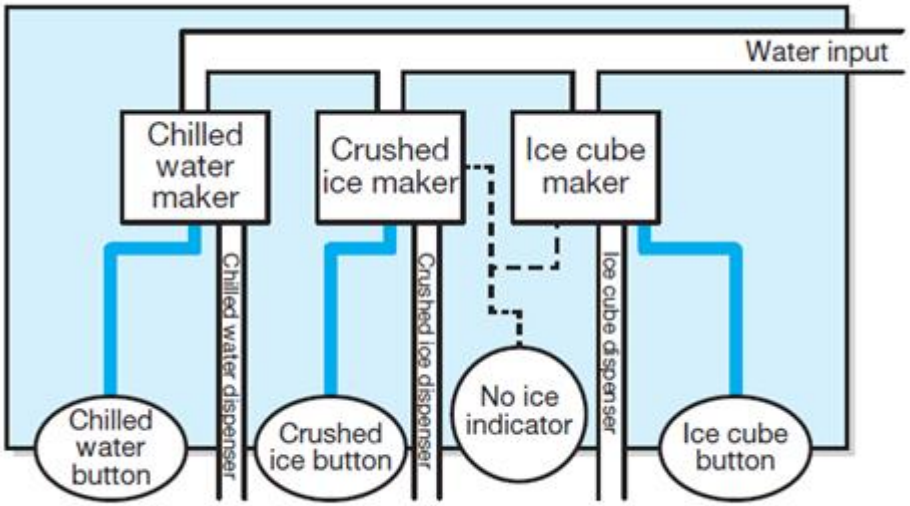
- Typical operations on data
  - Add data to a data collection.
  - Remove data from a data collection.
  - Ask questions about the data in a data collection.
- **An ADT** : a collection of data *and* a set of operations on data
- **A data structure** : an implementation of an ADT within a programming language

# Abstract Data Type

- ADT is a specification for a group of values and the operations on those values
- A dispenser of chilled water, crushed ice, and ice cubes



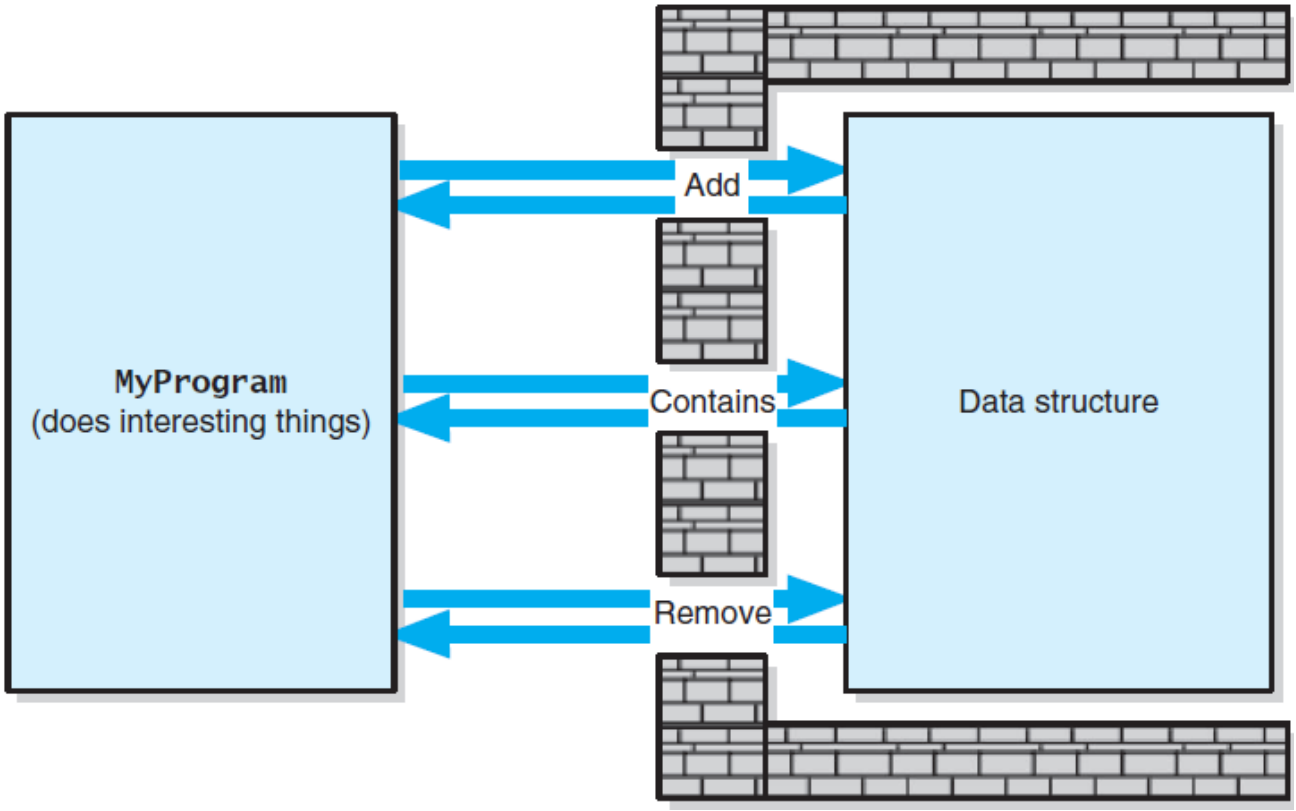

User's exterior view




Technician's interior view

# Abstract Data Type

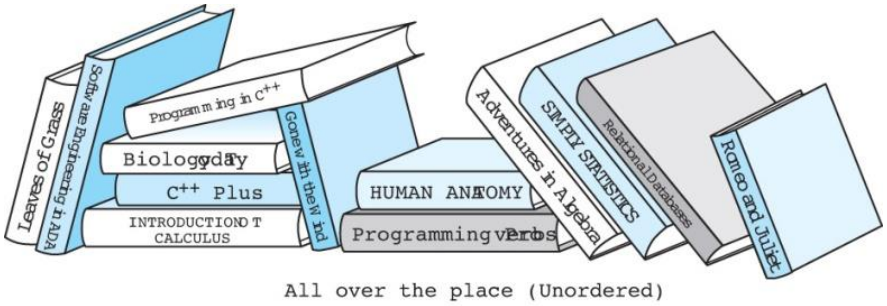
- A wall of ADT operations isolates a data structure from the program that uses it



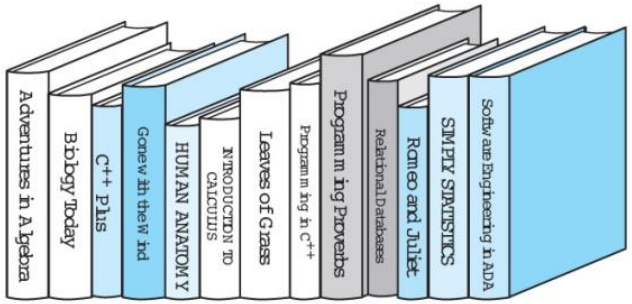
# Data from 3 different levels

- **Application (or user) level**: modeling real-life data in a specific context. WHY
- **Logical (or ADT) level**: abstract view of the domain and operations. WHAT
- **Implementation level**: specific representation of the structure to hold the data items, and the coding for operations. HOW

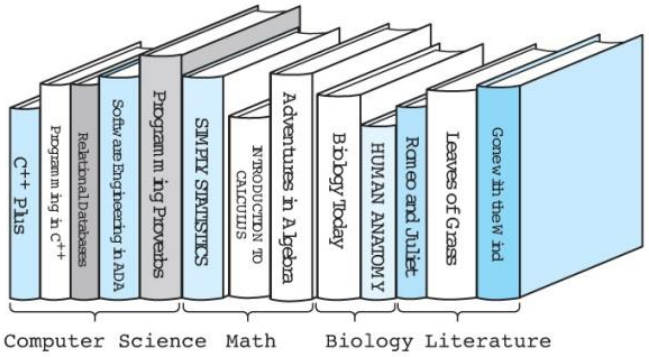
# Collection ordered in different ways




All over the place (Unordered)




Alphabetical order by title




Ordered by subject

# Viewing a library from 3 different levels

- **Application (or user) level**: Library of Congress, or Baltimore County Public Library.
- **Logical (or ADT) level**: domain is a collection of books;
  - operations include: check book out, check book in, pay fine, reserve a book.
- **Implementation level**: representation of the structure to hold the "books", and the coding for operations.

# Designing an ADT

- Evolves naturally during the problem-solving process
  - What data does a problem require?
  - What operations does a problem require?
- ADTs typically have initialization and destruction operations
  - Assumed but not specified at this stage

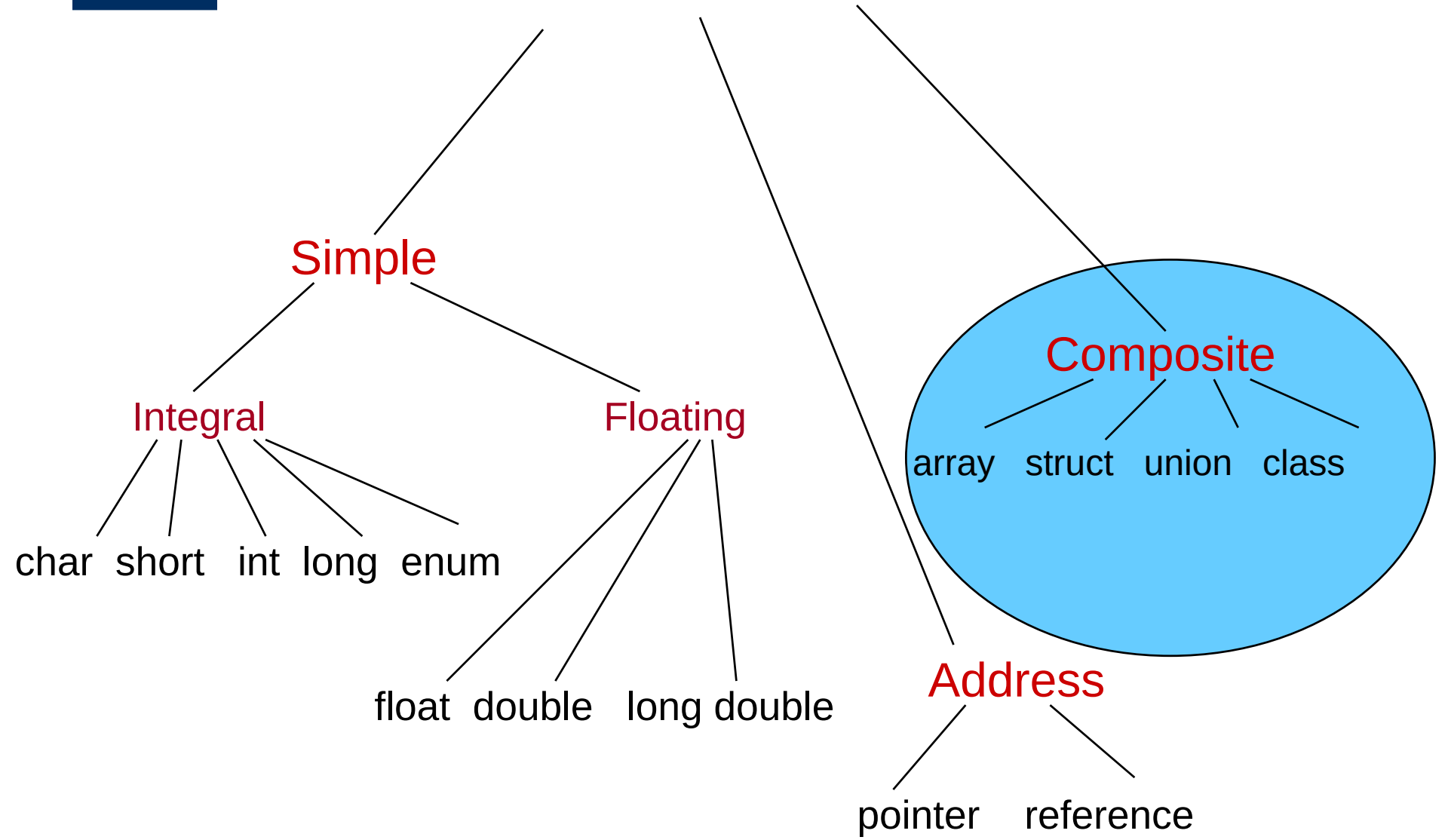
# ADTs That Suggest Other ADTs

- You can use an ADT to implement another ADT
  - Example: Date-Time objects available in C++ for use in various contexts
  - Possible to create your own fraction object

$$\left\{ \frac{a}{b} \mid a,\ \mathrm{b} \in \text{Integers},\ b \neq 0 \right\}$$

to use in some other object which required fractions

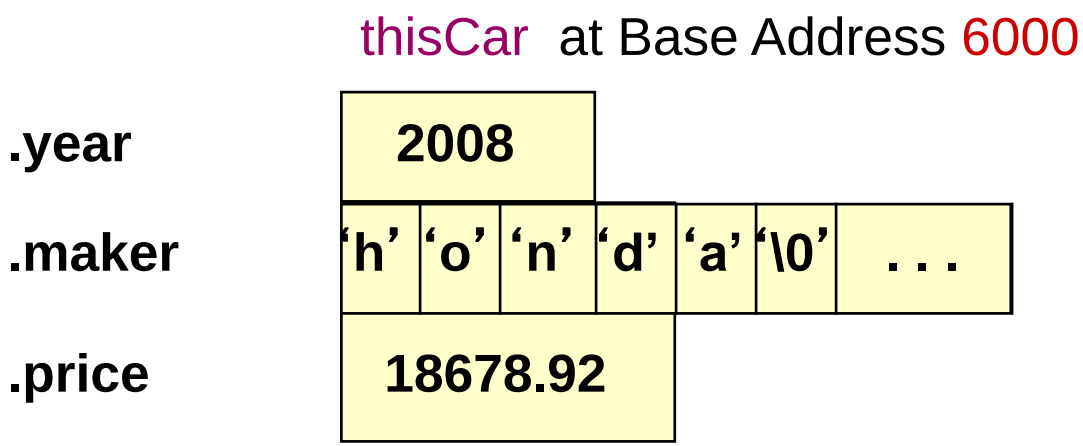
# C++ Built-In Data Types



# Records

- A composite data type made up of a finite collection of not necessarily homogeneous elements called *members* or *fields*

- For example . . .



# The ADT Bag

# The ADT Bag

- A bag is a container
  - Contains finite number of data objects
  - All objects of same type
  - Objects in no particular order
  - Objects may be duplicated

# The ADT Bag

- Consider the bag to be an abstract data type.
  - We are specifying an abstraction inspired by an actual physical bag
  - Doesn't do much more than contain its items
  - Can unordered and possibly duplicate objects
  - We insist objects be of same or similar types
- Knowing just its interface
  - Can use ADT bag in a program

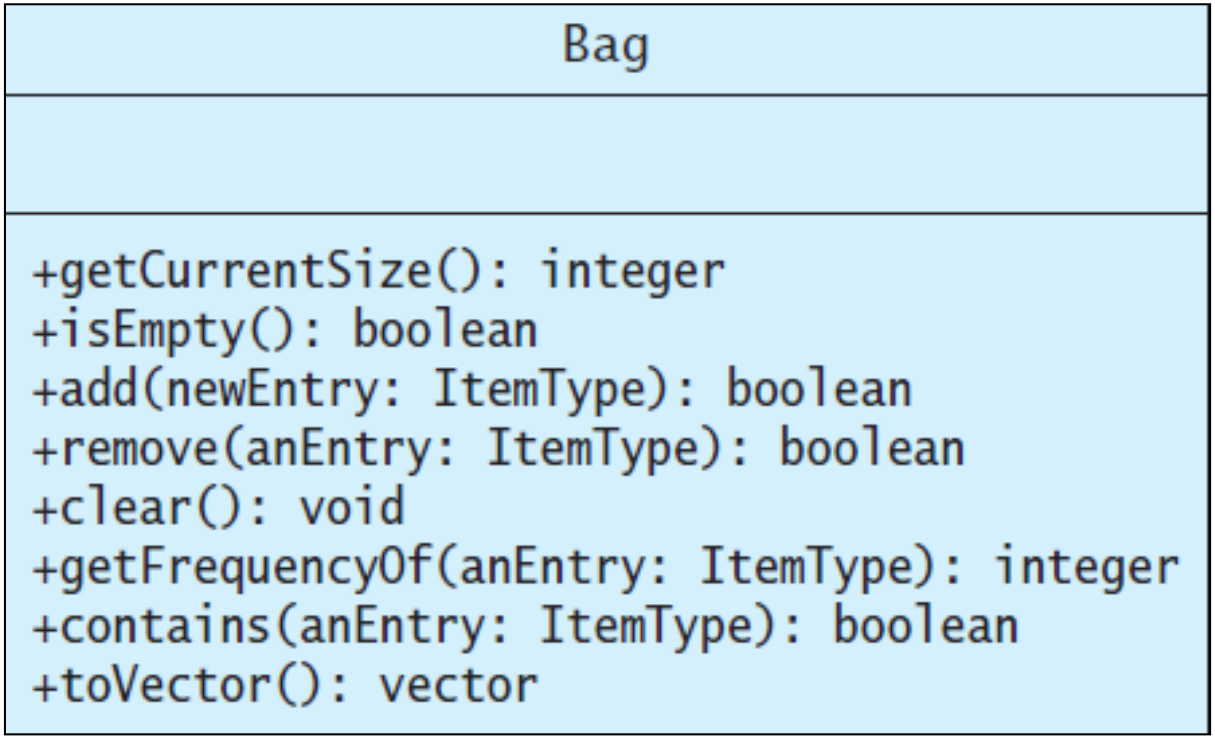
# Identifying Behaviors

- A CRC card for a class **`Bag`**

| Bag |
| --- |
| Responsibilities |
| Get the number of items currently in the bag |
| See whether the bag is empty |
| Add a given object to the bag |
| Remove an occurrence of a specific object from the bag, if possible |
| Remove all objects from the bag |
| Count the number of times a certain object occurs in the bag |
| Test whether the bag contains a particular object |
| Look at all objects that are in the bag |
| |
| Collaborations |
| The class of objects that the bag can contain |
| |
| |

# Specifying Data and Operations

- UML notation for the class **`Bag`**



# An Interface Template for the ADT

```
1  /** @file BagInterface.h */
2  #ifndef BAG_INTERFACE_
3  #define BAG_INTERFACE_
4
5  #include <vector>
6
7  template<class ItemType>
8  class BagInterface
9  {
10 public:
11    /** Gets the current number of entries in this bag.
12     @return  The integer number of entries currently in the bag. */
13    virtual int getCurrentSize() const = 0;
14
15    /** Sees whether this bag is empty.
16     @return  True if the bag is empty, or false if not. */
17    virtual bool isEmpty() const = 0;
18
19    /** Adds a new entry to this bag.
20     @post  If successful, newEntry is stored in the bag and
21        the count of items in the bag has increased by 1.
22     @param newEntry The object to be added as a new entry.
23     @return  True if addition was successful, or false if not. */
24    virtual bool add(const ItemType& newEntry) = 0;
```

LISTING 1-1 A file containing a C++ interface for bags

# An Interface Template for the ADT

```
22      @param newEntry The object to be added as a new entry.
23      @return  True if addition was successful, or false if not. */
24     virtual bool add(const ItemType& newEntry) = 0;
25
26     /** Removes one occurrence of a given entry from this bag,
27         if possible.
28      @post  If successful, anEntry has been removed from the bag
29         and the count of items in the bag has decreased by 1.
30      @param anEntry The entry to be removed.
31      @return  True if removal was successful, or false if not. */
32     virtual bool remove(const ItemType& anEntry) = 0;
33
34     /** Removes all entries from this bag.
35      @post  Bag contains no items, and the count of items is 0. */
36     virtual void clear() = 0;
37
38     /** Counts the number of times a given entry appears in this bag.
39      @param anEntry The entry to be counted.
40      @return  The number of times anEntry appears in the bag. */
41      virtual int getFrequencyOf(const ItemType& anEntry) const = 0;
```

LISTING 1-1 A file containing a C++ interface for bags

# An Interface Template for the ADT

```
39       @param anEntry The entry to be counted.
40       @return  The number of times anEntry appears in the bag. */
41       virtual int getFrequencyOf(const ItemType& anEntry) const = 0;
42
43     /** Tests whether this bag contains a given entry.
44       @param anEntry The entry to locate.
45       @return  True if bag contains anEntry, or false otherwise. */
46     virtual bool contains(const ItemType& anEntry) const = 0;
47
48     /** Empties and then fills a given vector with all entries that
49         are in this bag.
50       @return  A vector containing copies of all the entries in this bag. */
51     virtual std::vector<ItemType> toVector() const = 0;
52
53     /** Destroys this bag and frees its assigned memory. (See C++ Interlude 2.) */
54     virtual ~BagInterface() { }
55   }; // end BagInterface
```

LISTING 1-1 A file containing a C++ interface for bags

# Using the ADT Bag

```
1  #include <iostream> // For cout and cin
2  #include <string>   // For string objects
3  #include "Bag.h"    // For ADT bag
4
5  int main()
6  {
7     std::string clubs[] = { "Joker", "Ace", "Two", "Three", "Four",
8                             "Five", "Six", "Seven", "Eight", "Nine",
9                             "Ten", "Jack", "Queen", "King" };
10    // Create our bag to hold cards.
11    Bag<std::string> grabBag;
12
13    // Place six cards in the bag.
14    grabBag.add(clubs[1]);
15    grabBag.add(clubs[2]);
16    grabBag.add(clubs[4]);
17    grabBag.add(clubs[8]);
18    grabBag.add(clubs[10]);
19    grabBag.add(clubs[12]);
20
21    // Get friend's guess and check it
```

LISTING 1-2 A program for a card guessing game

# Using the ADT Bag

```
21      // Get friend's guess and check it.
22      int guess = 0;
23      while (!grabBag.isEmpty())
24      {
25         std::cout << "What is your guess? (1 for Ace to 13 for King):";
26         std::cin >> guess;
27
28         // Is card in the bag?
29         if (grabBag.contains(clubs[guess]))
30         {
31            // Good guess - remove card from the bag.
32            std::cout << "You get the card!\n";
33            grabBag.remove(clubs[guess]);
34         }
35         else
36         {
37            std::cout << "Sorry, card was not in the bag.\n";
38         }  // end if
39      }  // end while
40      std::cout << "No more cards in the bag. Game over!\n";
41      return 0;
42   }; // end main
```

LISTING 1-2 A program for a card guessing game

# Reminders

- Assignment 1 is on
  - **Due on Mon, Feb 5th at 2:00 pm**
  - **20 % deduction policy**

- TAs are there for discussions
  - Not for debugging your code!