Data Abstraction: The Walls

Chapter 1

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, do
- Object-oriented design
  - Describe solution to problem
  - Express solution in terms of software objects
  - Create one or more models of solution

Aspects of Object-Oriented 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 time.

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

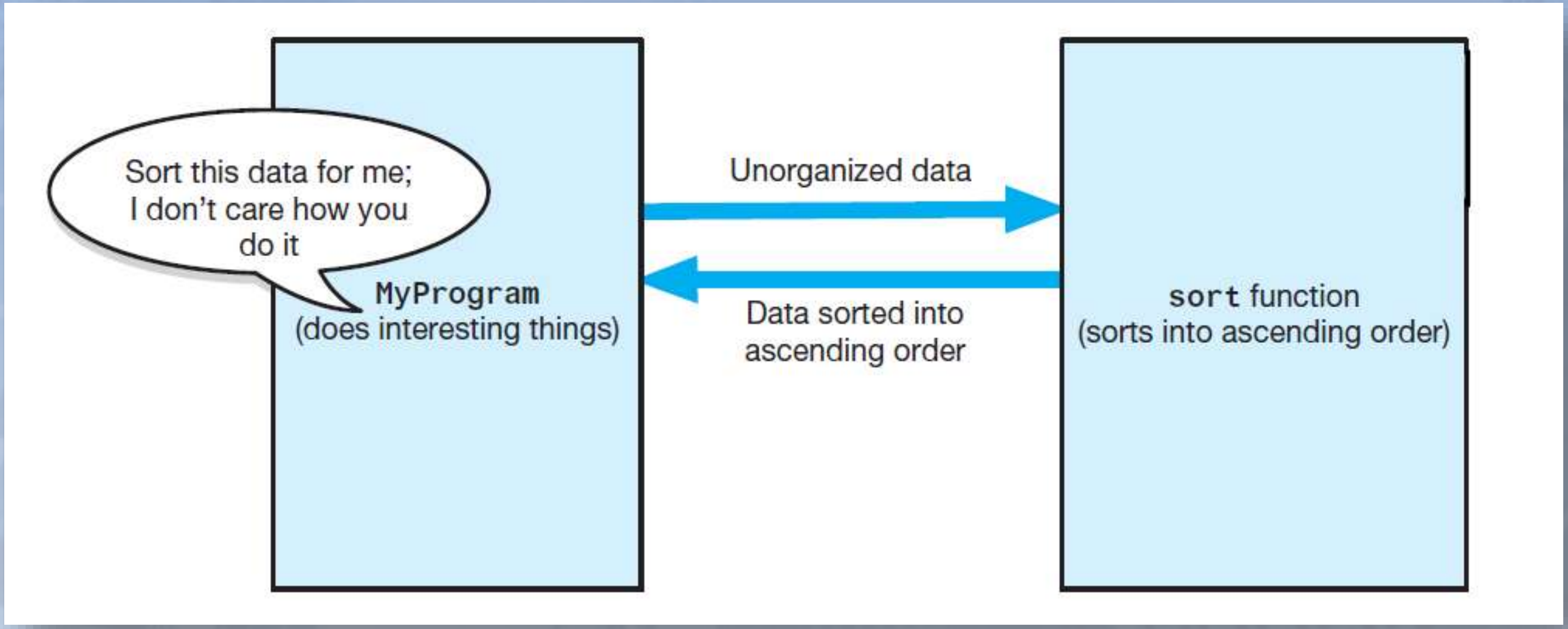

Sort this data for me; I don't care how you do it
MyProgram
(does interesting things)
Unorganized data
Data sorted into ascending order
sort function
(sorts into ascending order)

FIGURE 1-1 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

Ways to address invalid conditions:

- Assume they will not happen
- Ignore such situations
- Guess at client’ s intentions
- Return value that signals problem
- Throw an exception

Abstraction

- Separate purpose of a module from its implementation
- Specifications do not indicate how to implement
  - Able to use without knowing implementation

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

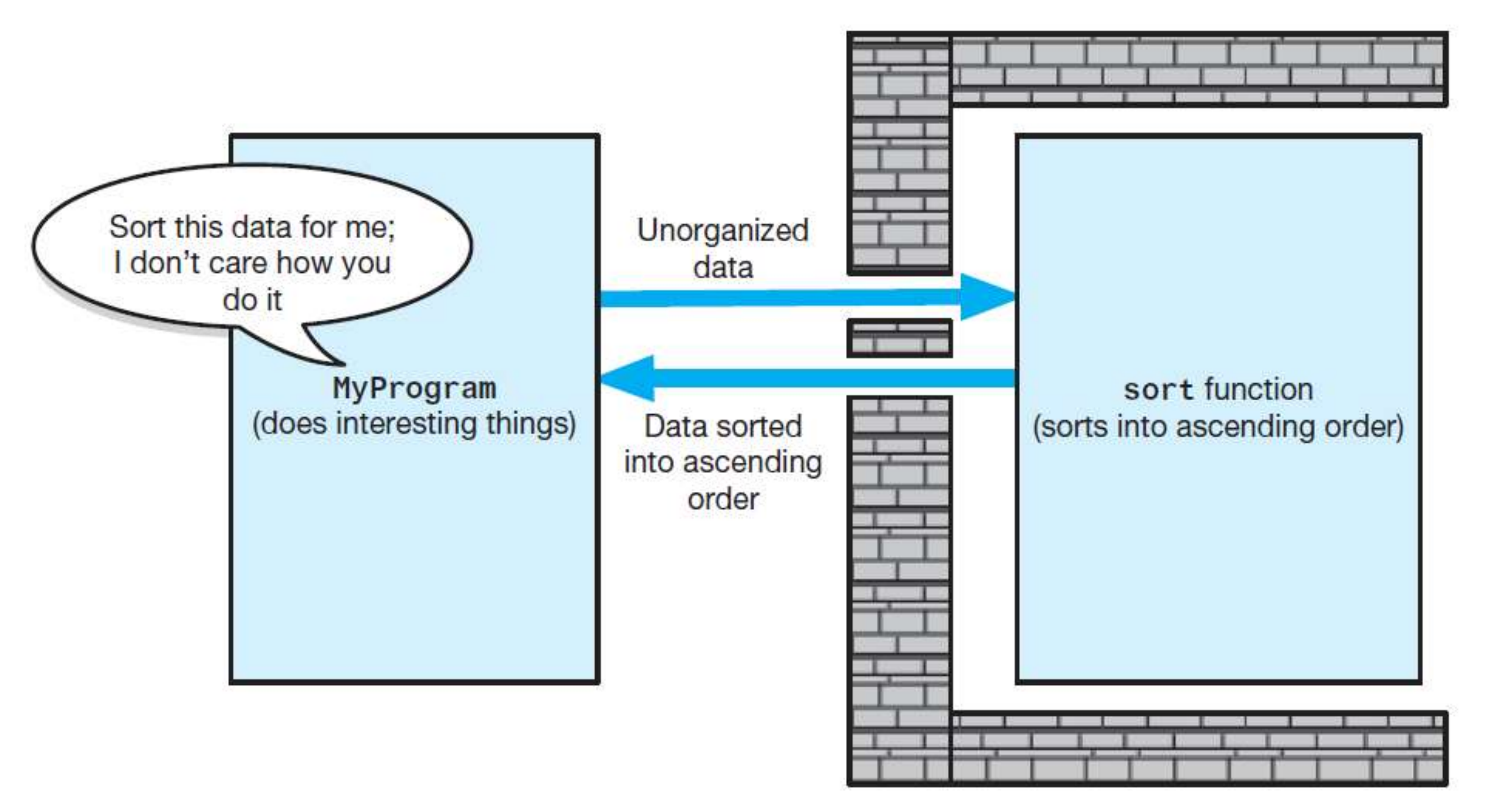

Sort this data for me;
I don't care how you do it
MyProgram
(does interesting things)
Unorganized data
Data sorted into ascending order
sort function
(sorts into ascending order)

FIGURE 1-2 Tasks communicate through a slit in the wall

Information Hiding

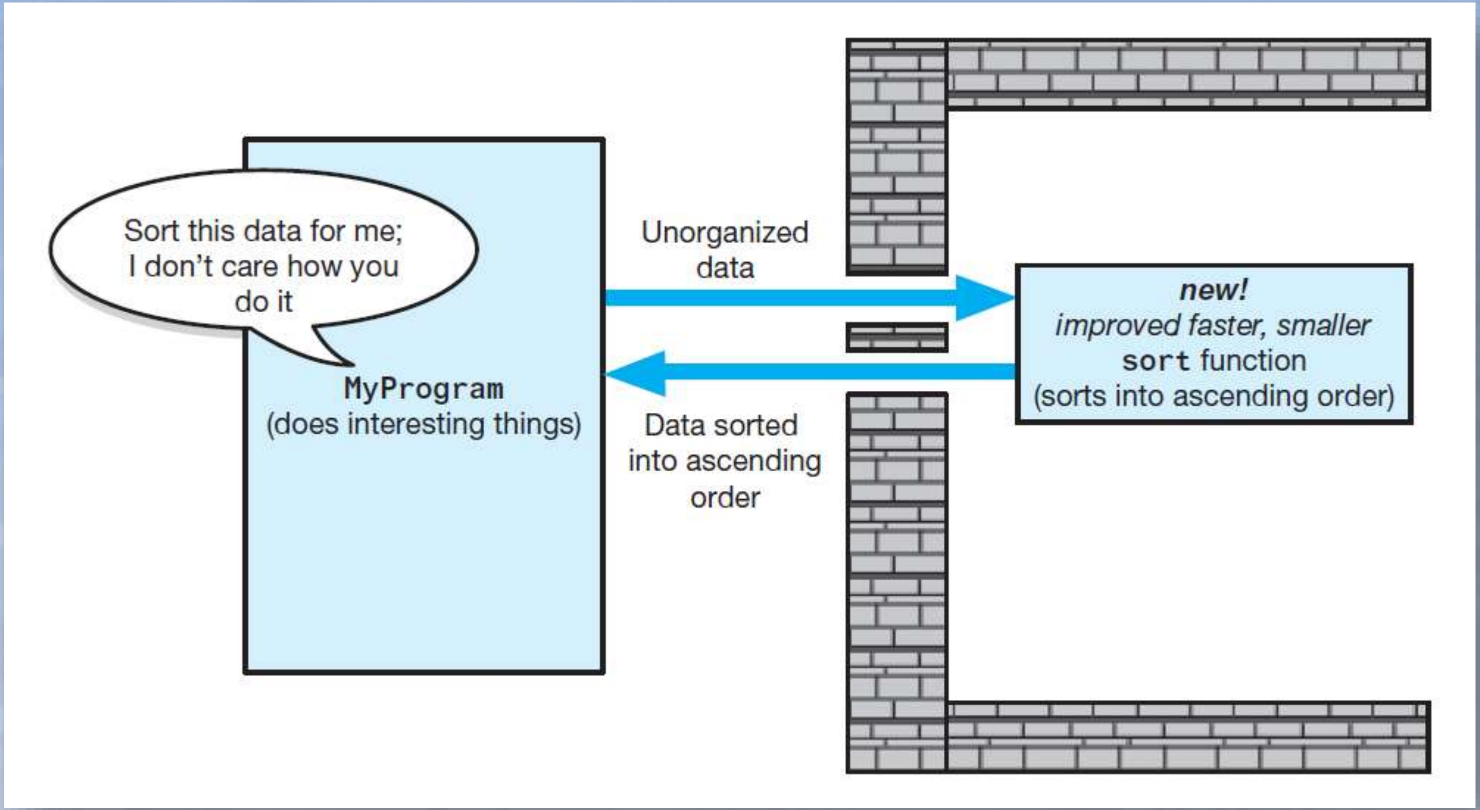


FIGURE 1-3 A revised implementation communicates through the same slit in the wall

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

Abstract **D**ata **T**ypes (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 Types (ADT)

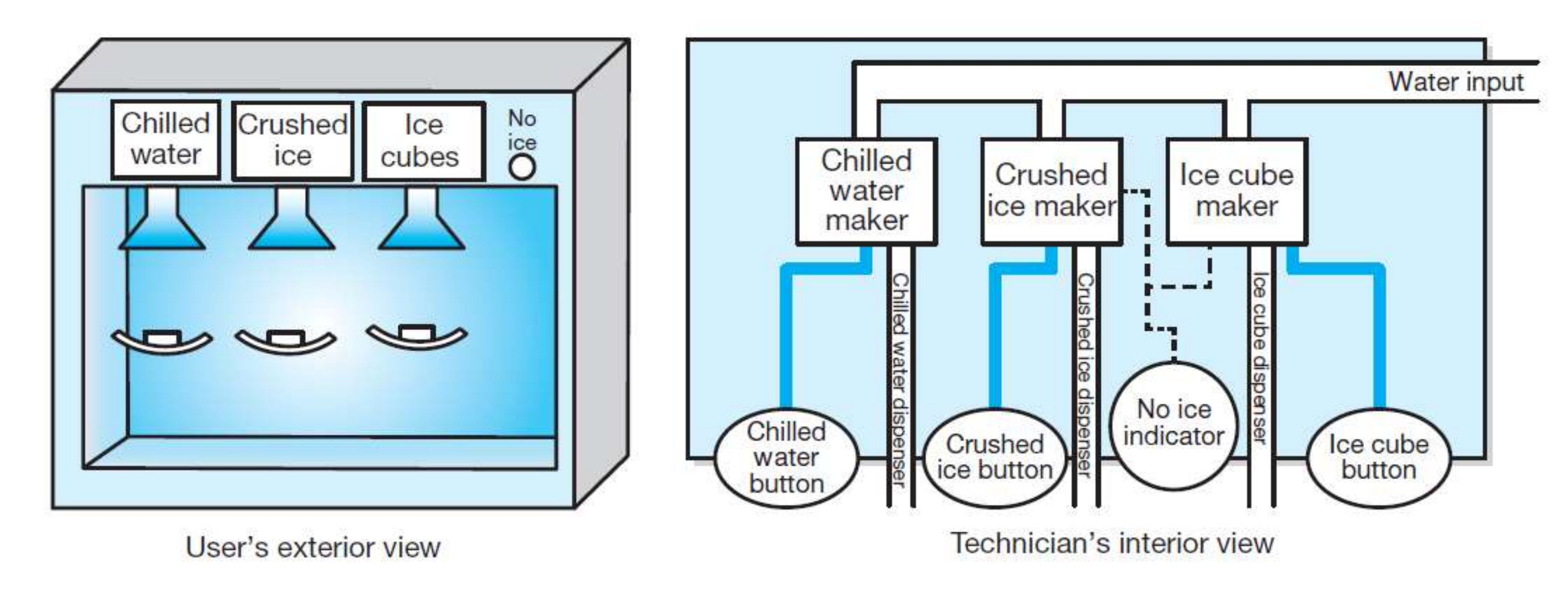


FIGURE 1-4 A dispenser of chilled water, crushed ice, and ice cubes

Abstract Data Types (ADT)

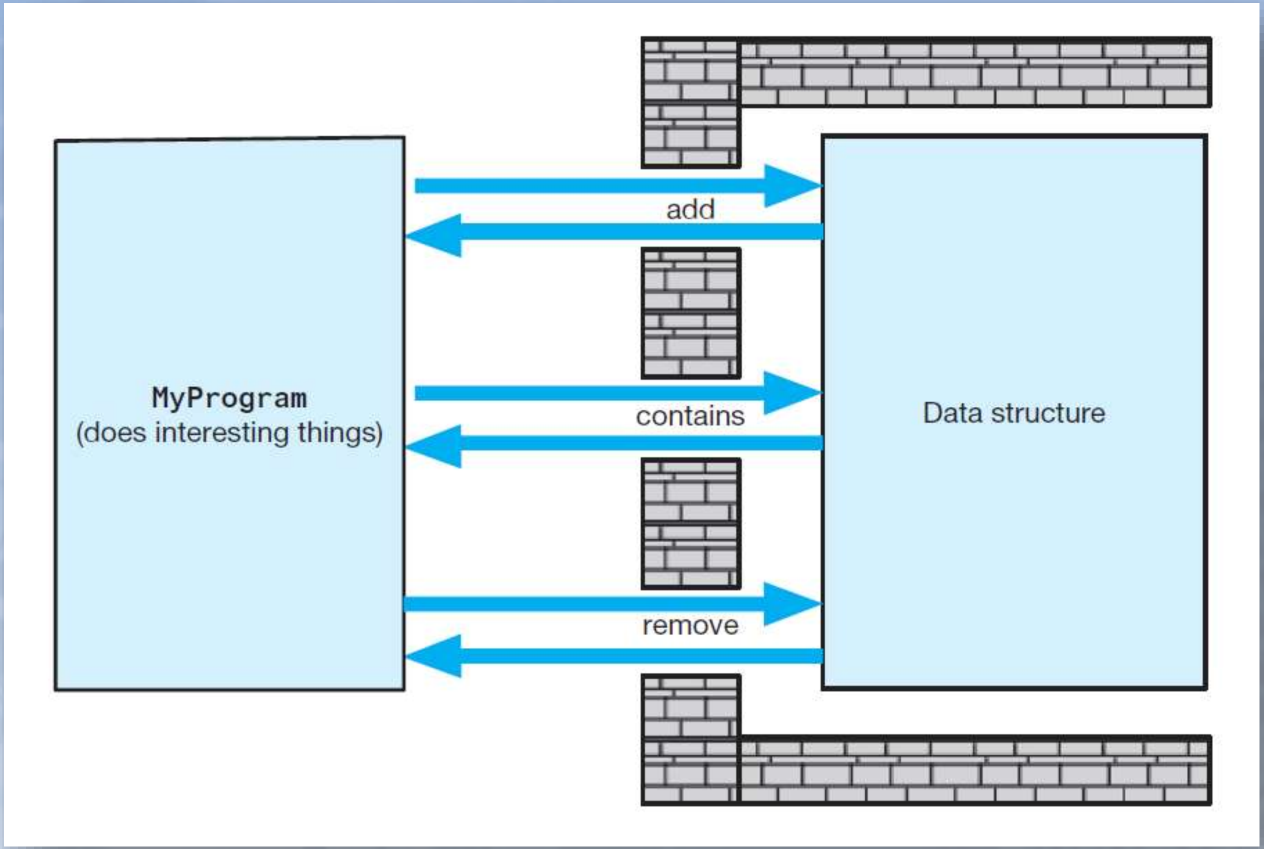


FIGURE 1-5 A wall of ADT operations isolates a data structure from the program that uses it

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

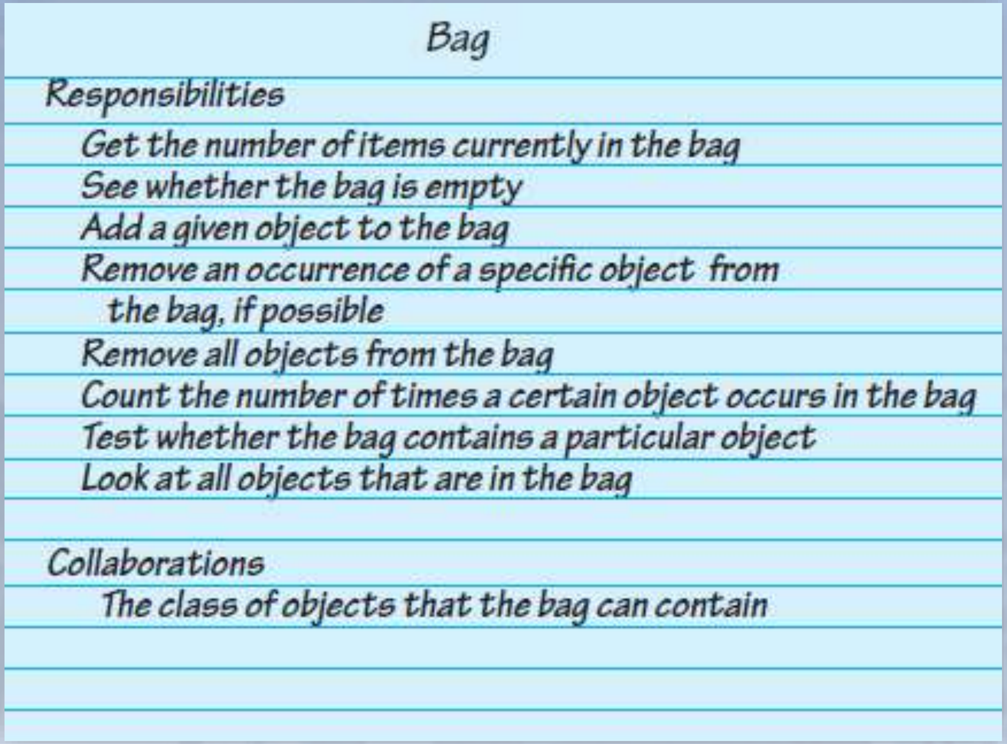
$$\left\{ \frac{a}{b} \,\middle|\, a,\ \mathrm{b} \in \text{Integers},\ b \neq 0 \right\}$$

to use in some other object which required fractions

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



Bag

Responsibilities

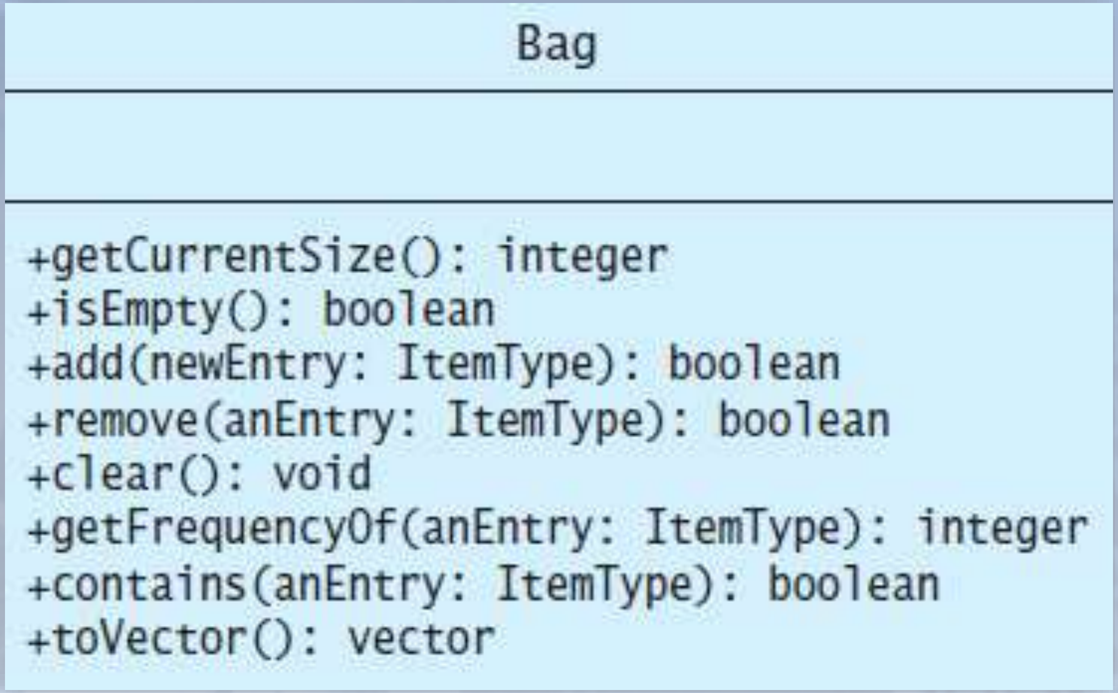
Get the number of items currently in the bag
See whether the bag is empty
Add a given object to the bag
Remove an occurrence of a specific object from the bag, if possible
Remove all objects from the bag
Count the number of times a certain object occurs in the bag
Test whether the bag contains a particular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

Collaborations

The class of objects that the bag can contain

FIGURE 1-6 A CRC card for a class Bag

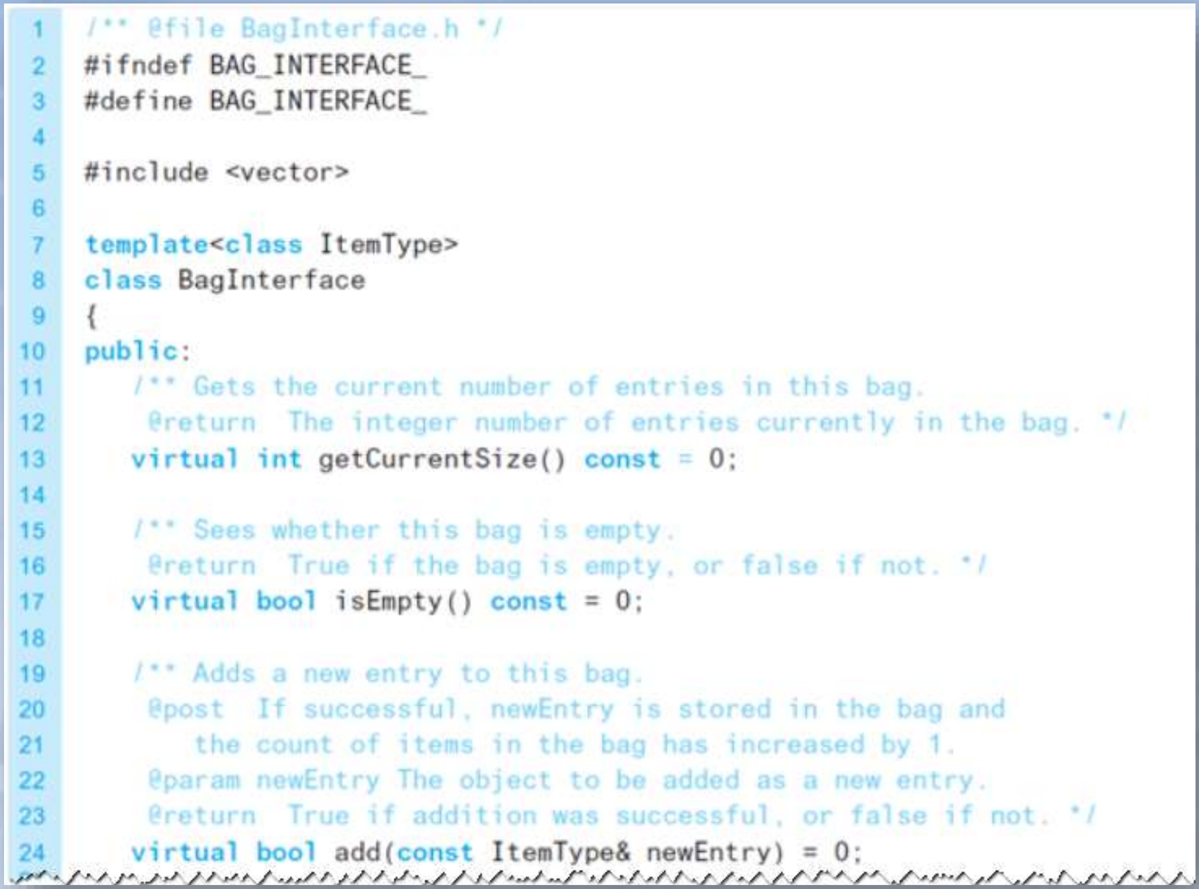
Specifying Data and Operations



```
Bag

+getCurrentSize(): integer
+isEmpty(): boolean
+add(newEntry: ItemType): boolean
+remove(anEntry: ItemType): boolean
+clear(): void
+getFrequencyOf(anEntry: ItemType): integer
+contains(anEntry: ItemType): boolean
+toVector(): vector
```

FIGURE 1-7 UML notation for the class Bag

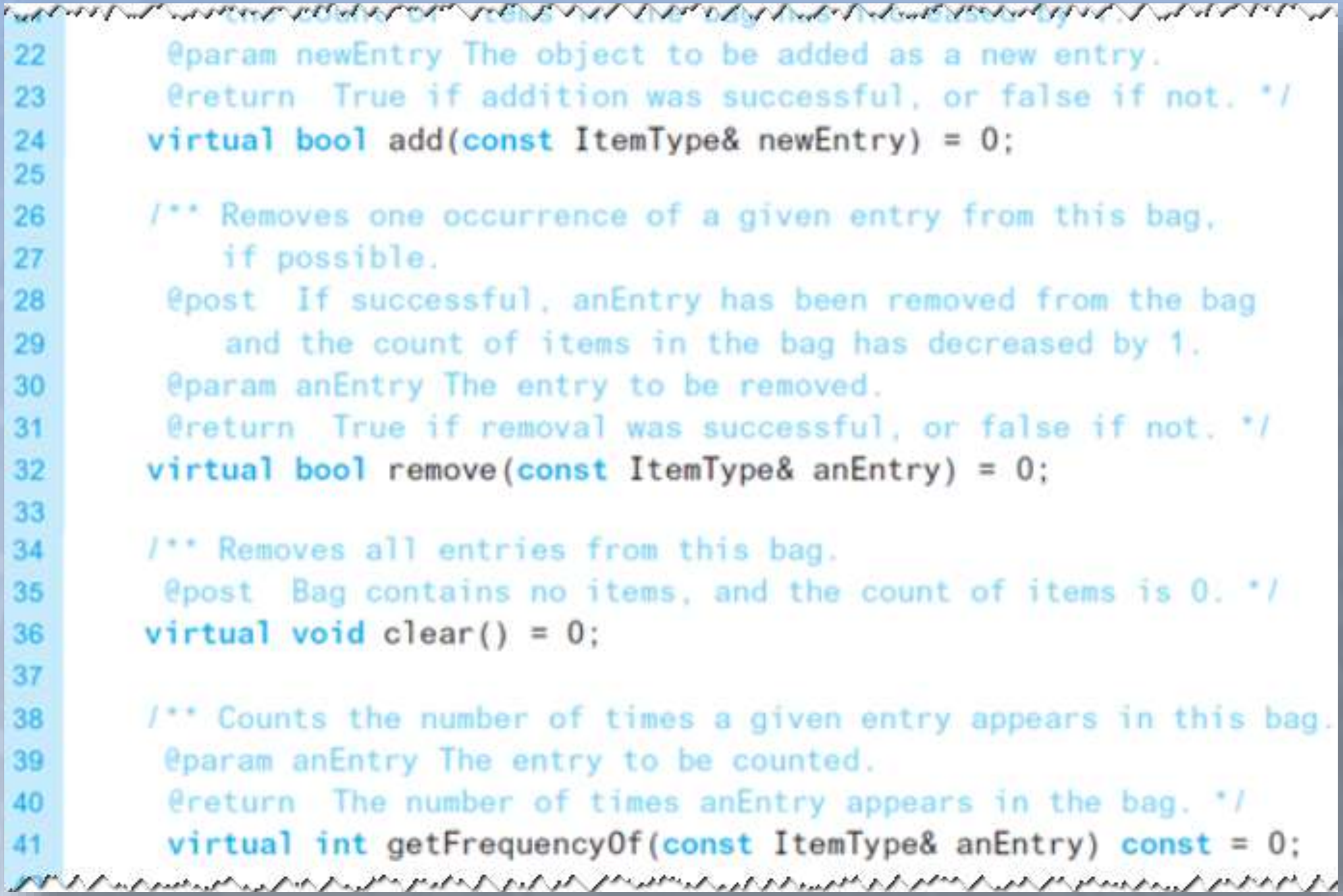
An Interface Template for the ADT



```
/** @file BagInterface.h */
#ifndef BAG_INTERFACE_
#define BAG_INTERFACE_

#include <vector>

template<class ItemType>
class BagInterface
{
public:
   /** Gets the current number of entries in this bag.
    @return  The integer number of entries currently in the bag. */
   virtual int getCurrentSize() const = 0;

   /** Sees whether this bag is empty.
    @return  True if the bag is empty, or false if not. */
   virtual bool isEmpty() const = 0;

   /** Adds a new entry to this bag.
    @post  If successful, newEntry is stored in the bag and
       the count of items in the bag has increased by 1.
    @param newEntry The object to be added as a new entry.
    @return  True if addition was successful, or false if not. */
   virtual bool add(const ItemType& newEntry) = 0;
```

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

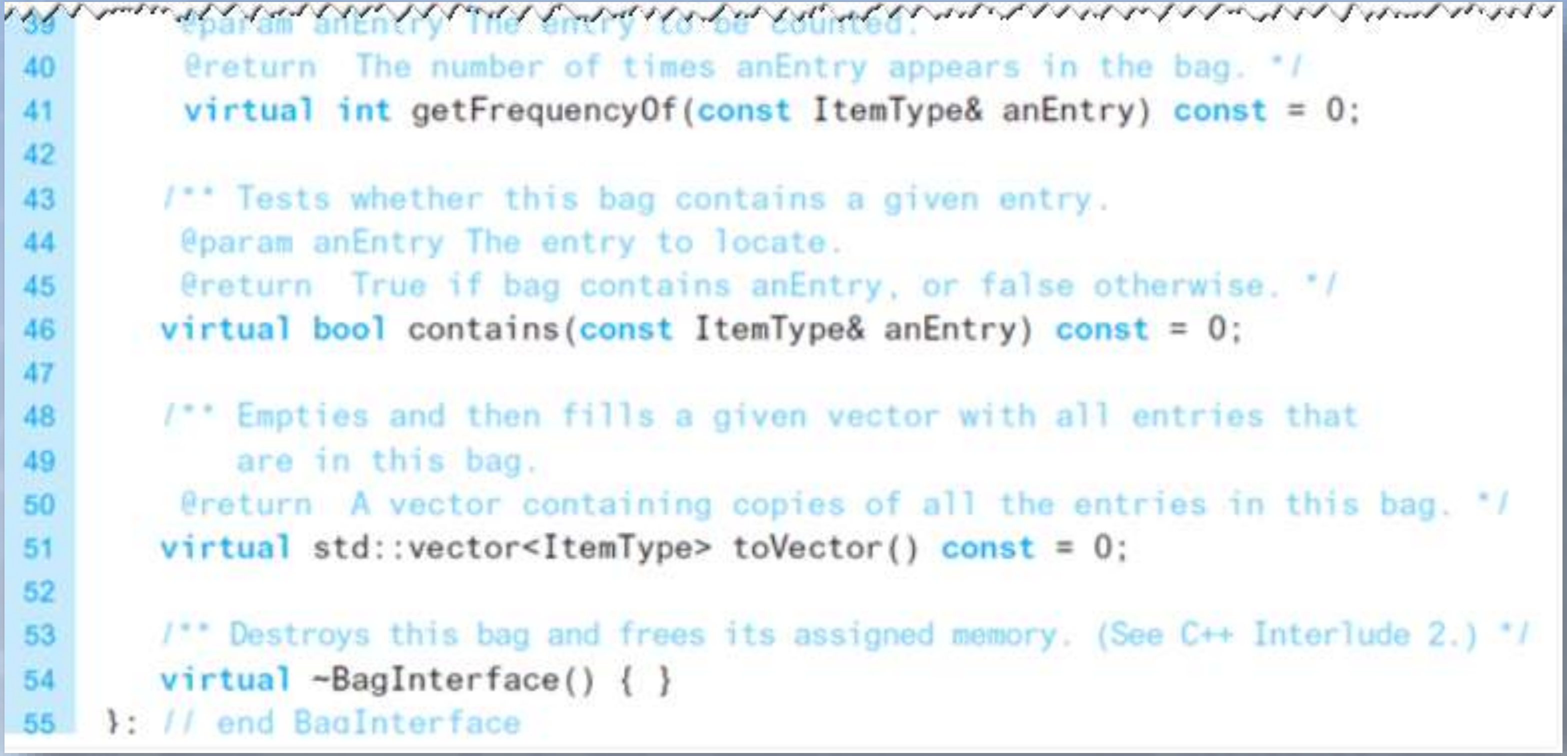
An Interface Template for the ADT



```
    @param newEntry The object to be added as a new entry.
    @return  True if addition was successful, or false if not. */
   virtual bool add(const ItemType& newEntry) = 0;

   /** Removes one occurrence of a given entry from this bag,
       if possible.
    @post  If successful, anEntry has been removed from the bag
       and the count of items in the bag has decreased by 1.
    @param anEntry The entry to be removed.
    @return  True if removal was successful, or false if not. */
   virtual bool remove(const ItemType& anEntry) = 0;

   /** Removes all entries from this bag.
    @post  Bag contains no items, and the count of items is 0. */
   virtual void clear() = 0;

   /** Counts the number of times a given entry appears in this bag.
    @param anEntry The entry to be counted.
    @return  The number of times anEntry appears in the bag. */
    virtual int getFrequencyOf(const ItemType& anEntry) const = 0;
```

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

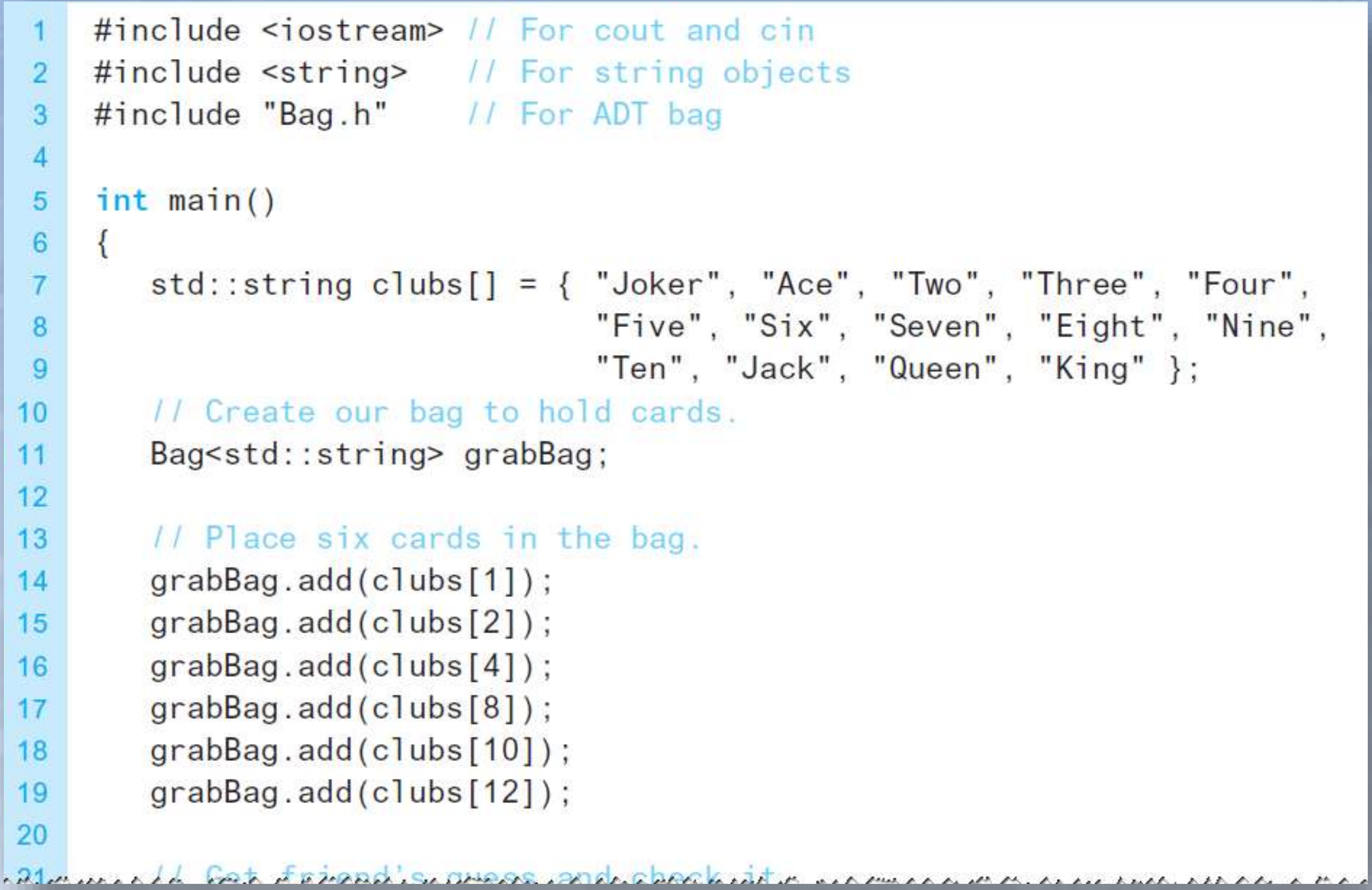
An Interface Template for the ADT



```
      @param anEntry The entry to be counted.
      @return  The number of times anEntry appears in the bag. */
      virtual int getFrequencyOf(const ItemType& anEntry) const = 0;

     /** Tests whether this bag contains a given entry.
      @param anEntry The entry to locate.
      @return  True if bag contains anEntry, or false otherwise. */
     virtual bool contains(const ItemType& anEntry) const = 0;

     /** Empties and then fills a given vector with all entries that
         are in this bag.
      @return  A vector containing copies of all the entries in this bag. */
     virtual std::vector<ItemType> toVector() const = 0;

     /** Destroys this bag and frees its assigned memory. (See C++ Interlude 2.) */
     virtual ~BagInterface() { }
  }; // end BagInterface
```

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

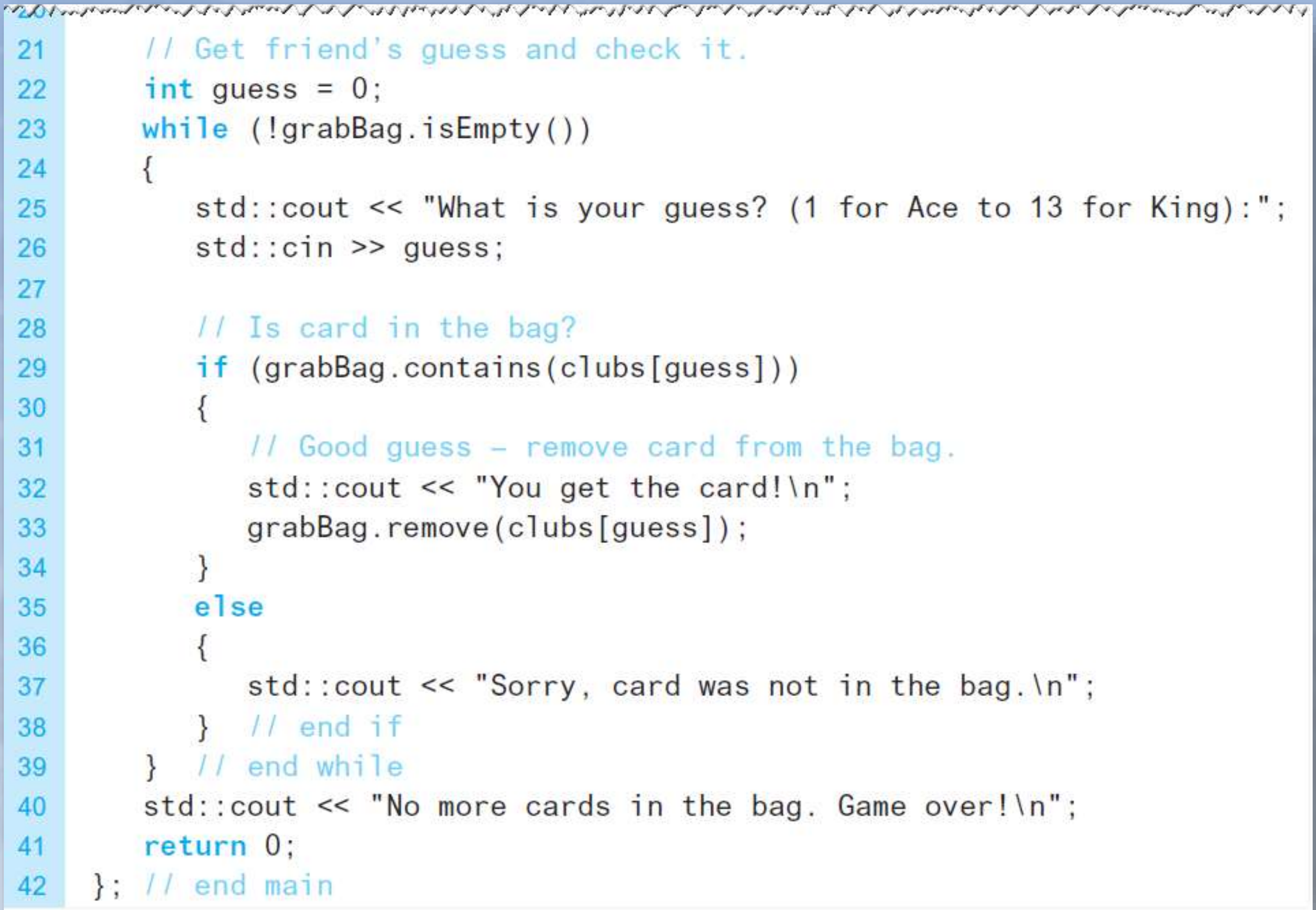
Using the ADT Bag



```
#include <iostream> // For cout and cin
#include <string>   // For string objects
#include "Bag.h"    // For ADT bag

int main()
{
   std::string clubs[] = { "Joker", "Ace", "Two", "Three", "Four",
                           "Five", "Six", "Seven", "Eight", "Nine",
                           "Ten", "Jack", "Queen", "King" };
   // Create our bag to hold cards.
   Bag<std::string> grabBag;

   // Place six cards in the bag.
   grabBag.add(clubs[1]);
   grabBag.add(clubs[2]);
   grabBag.add(clubs[4]);
   grabBag.add(clubs[8]);
   grabBag.add(clubs[10]);
   grabBag.add(clubs[12]);

   // Get friend's guess and check it.
```

LISTING 1-2 A program for a card guessing game

Using the ADT Bag



```
   // Get friend's guess and check it.
   int guess = 0;
   while (!grabBag.isEmpty())
   {
      std::cout << "What is your guess? (1 for Ace to 13 for King):";
      std::cin >> guess;

      // Is card in the bag?
      if (grabBag.contains(clubs[guess]))
      {
         // Good guess – remove card from the bag.
         std::cout << "You get the card!\n";
         grabBag.remove(clubs[guess]);
      }
      else
      {
         std::cout << "Sorry, card was not in the bag.\n";
      }  // end if
   }  // end while
   std::cout << "No more cards in the bag. Game over!\n";
   return 0;
};  // end main
```

LISTING 1-2 A program for a card guessing game

End

Chapter 1