Chapter 3: Data Abstraction

1 Abstract Data Types

In Chapter 1 it was stated that:

- Data abstraction separates *what* can be done to data from *how* it is actually done.
- An abstract data type (ADT) is a collection of data and a set of operations that act on the data.
- An ADT's operations can be used without knowing their implementations or how the data is stored, as long as the interface to the ADT is precisely specified.
- A data structure is a data storage container that can be defined by a programmer in a programming language. It may be part of an object or even implement an object, but **it is not an ADT!**

A soft drink vending machine is a good analogy. From a consumer's perspective, a vending machine contains soft drinks that can be selected and dispensed when the appropriate money has been inserted and the appropriate buttons pressed. The consumer sees its interface alone. Its inputs are money and button-presses. The vending machine outputs soft drinks and change. The user does not need to know how a vending machine works, only what it does, and in this sense a vending machine can be viewed as an ADT.

The vending machine company's support staff need to know the vending machines internal workings, its data structure. The person that restocks the machine has to know the internal structure, i.e., where the spring water goes, where the sodas go, and so on. That person is like the programmer who implements the ADT with data structures and other programming constructs.

The ADT is sometimes characterized as a wall with slits in it for data to pass in and out. The wall prevents outside users from accessing the internal implementation, but allows them to request services of the ADT.

ADT operations can be broadly classified into the following types:

- operations that add new data to the ADT's data collection,
- operations that remove data from the ADT's data collection,
- operations that modify the data in the ADT's collection,
- operations to query the ADT about the data in the collection.

2 UML Syntax

We will occasionally use the Uniform Modelling Language syntax for describing the interfaces to an ADT. This syntax is relatively simple.

To describe the ADT's **attributes** (which will become data members of a class), use the syntax

[visiblity] name [:type] [=defaultValue] [{property}]

where:

- square brackets `[]` indicate optional elements
- *visibility* is one of the symbols `+`, for public accessibility, `-`, for private accessibility, or `#`, for protected accessibility. If this element is not included, the *visibility* defaults to private access.
- *name* is the attribute's name.
- *type* is the data type of the element.
- *defaultValue* is the element's default value. It will not have a default value if this is omitted.
- *property* is one of the values `changeable`, indicating an attribute that can be modified, such as a variable, or `frozen`, indicating a constant value.

Examples

```
-count: integer  {changeable}      // a private integer variable
-count: integer                    // same as above - changeable is the default
+MaxItems : integer = 100 {frozen} // a public constant, defaulting to 100
-name: string                      // a private string variable
```

The **operations** of an ADT are described by the following syntax:

[visiblity] name ([parameter_list]) [:type] [{property}]

where:

- square brackets `[]` indicate optional elements
- *visibility* is the same as for attributes except that the default is public access.
- *name* is the operation's name.
- *type* is the return value of the operation. It can be `void` or nothing if the operation does not return a value.
- *property* is usually either omitted or is the value `query`, which means that the operation does not modify any attributes of the object on which this operation is called.
- The parameter list is either empty (and the parentheses are not omitted – they are outside of the square brackets on purpose) or it is a comma-separated list whose elements are of the form

  [direction] name:type [=defaultValue]

  where

  - square brackets are again optional.
  - *direction* is one of the words `in`, meaning the parameter is an input to the function, `out`, meaning it is an output, or `inout`, meaning it is both.
  - *type* is the parameter's type and is mandatory.
  - an optional default value can be specified as an argument if there is no actual argument in the call.

Examples

```
+insert( in position_to_insert: integer,
         in new_item: list_item_type,
         out success: boolean): void

+fill( inout input_stream: input_file_stream,
       in number_to_read: integer,
       out success: boolean): void
```

Both of the above operations are public and return nothing. Both also have a third parameter for indicating the success of the operation. It is a style of coding to put each parameter on a separate line. This makes them easier to read.

3 Specification of an ADT

The first step in any object-oriented solution to a problem is to specify the ADTs that are in it. Each module's specification must be written before the module itself.

We will use the example of a list to demonstrate how to specify an ADT. Lists are ubiquitous containers. People use lists every day in their lives: shopping lists, lists of assignments to finish for school, lists of people to call, lists of movies to watch, and so on. Computer scientists use lists in a multitude of ways, such as:

- lists of blocks allocated to files in a file system
- lists of active processes in a computer system
- lists of memory blocks to be written to disk after a write operation to them
- lists of atm transactions to be performed
- lists of packets waiting to be routed
- polynomials (as lists of terms)

What then is a list, assuming it is not empty? What characterizes a list? In other words, what makes something a list? Obviously it is a *collection of things*, but there are many different types of collections of things. The defining property of a list is that it is **sequential**; it is a collection of items with the property that it has a first item and a last item (which may be the same item) and that each item that is not the first has a unique predecessor and each item that is not the last has a unique successor.

This is a characterization of the data in the list, but not the things that we can do with a list. What do we do with lists? We insert items into them. We remove items from them. We check whether a specific item is in the list. We count how many items are in the list. Sometimes we might want to know what item is in position k in a list (e.g., which horse came in third place in the fourth race today?)

Where can an item be inserted in a list? Only at the end? Only at the beginning or end? Anywhere? Which items can be deleted? Any of them? Only those at the end? Only those at either end? The answers to this question characterize the list's type. We will see soon that stacks and queues are two kinds of lists that constrain where insertions and deletions take place. Can lists be sorted? Is sorting something that a list, as an ADT, should support?

There are two operations that people normally do not perform with lists but computer programs must: creation and destruction. People do not normally say they are creating an empty list, but that is an operation that a list ADT must provide to client software. Similarly, computer programs need to be able to destroy lists because they are no longer needed and they hold resources. People just crumple up the paper with the list on it and dispose of it that way.

4 The List ADT

We will define a list ADT. We know that a list contains a sequence of elements as its data, so we will assume that the sequence is a_1, a_2, a_3, ..., a_N. Notice that the list index values are 1-based, not 0-based. There is no item at position 0. The first item is at position 1. Based on the above considerations, we will assume that the list operations are:

1. Create an empty list.
2. Destroy a (possibly non-empty) list.
3. Report whether it is empty or not.
4. Report how many elements it contains.
5. Insert an element at a given position in the list.
6. Delete the element at a given position in the list.
7. Retrieve the element at a given position in the list.
8. (Optional) Report whether a particular element is a member of the list. (e.g., is milk in the shopping list?)

The last operation does not have to be a supported operation, as we can repeatedly use the previous operation to determine whether a particular element is in the list, but doing so may not be efficient. We will address this issue later. In fact we do not need an operation to determine whether it is empty if we have one that tells us how many items are in the list, but it is convenient.

Now we make the specified operations more precise by writing them with enough detail so that they are unambiguous and clear. They are written first with a combination of C++ and pseudocode. None of the operations have the list as a parameter because it is assumed that they are called on a list object of some kind. This level of detail will come when they are converted to actual code.

4.1 The List Interface (Operation Contract)

This description does not use the UML syntax.

```
create_list();
/* Creates an empty list
   A new empty list is created. This is essentially a C++ constructor.
*/

destroy_list();
/* Destroys the list
   This is a destructor. It deallocates all memory belonging to the list.
*/

bool is_empty() const;
/* Checks if the list is empty
   This returns true if the list is empty and false if it is not.
*/
int  length() const;
/* Determines the length of a list. This returns the length of the list.*/
```

```
void insert( [in] int position_to_insert,
             [in] list_item_type new_item,
             [out] bool& success);
// Inserts new_item into a list
// If  1 <= position_to_insert <= list.length() + 1, then new_item
// is inserted into the list at position position_to_insert and
// success is true afterward. Otherwise success is false.
// The items in positions >= position_to_insert are shifted so that
// their positions in the list are one greater than before the insertion.
// Note: Insertion will not be successful if
// position_to_insert < 1 or > ListLength()+1.

void delete( [in] int position,
             [out] bool& success);
// Deletes an item from a list.
// Precondition: position indicates where the deletion
//               should occur.
// Postcondition: If 1 <= position <= list.length(),
//               the item at position position in the list is
//               deleted, other items are renumbered accordingly,
//               and success is true; otherwise success is false.

void retrieve([in] int position,
              [out] list_item_type& DataItem,
              [out] bool& success) const;
// Retrieves a list item by position number.
// Precondition: position is the number of the item to
//               be retrieved.
// Postcondition: If 1 <= position <= list.length(),
//               DataItem is the value of the desired item and
//               success is true; otherwise success is false.
```

The insert, delete, and retrieve operations require a bit of explanation. Each has a position parameter and a parameter to indicate whether the operation succeeded. The position parameter is 1-based:

```
insert(1,Sam, succeeded)
```

will always succeed because position 1 is at least 1 and never greater than the list length +1. Therefore the effect is to put Sam into the first position, shifting all elements as necessary. In contrast

```
insert(0, Sam, succeeded)
```

fails because $0 < 1$. If the list is empty, then any delete operation will fail because list length < 1 and so the condition

```
1 <= position <= listlength()
```

can never be true. If the list has length N, then the operation

```
delete(k, succeeded)
```

will succeed as long as 1 <= k <= N, and the effect will be to remove the item a_k, shifting items $a_{k+1}, a_{k+2}, ..., a_N$ to positions $k, k+1, ..., N-1$ respectively. Similarly, the retrieve operation must be given a position parameter whose value lies between 1 and the list length, otherwise it will fail, so if the list is empty, it will fail.

The above is just the specification of the interface. The job of the programmer is to convert this interface to code and to implement it. Those steps come later.

4.2 A UML Description of the List Interface (Operation Contract)

```
+create_list(): void
/* Creates an empty list
   A new empty list is created. This is essentially a C++ constructor.
*/
+destroy_list(): void
/* Destroys the list
   This is a destructor. It deallocates all memory belonging to the list.
*/
+is_empty(): boolean {query}
/* Checks if the list is empty
   This returns true if the list is empty and false if it is not.
*/
+length(): integer {query}
/* Determines the length of a list. This returns the length of the list. */

+insert( in position_to_insert: integer,
         in new_item: list_item_type,
         out success: boolean): void
// Inserts new_item into a list
// If  1 <= position_to_insert <= list.length() + 1, then new_item
// is inserted into the list at position position_to_insert and
// success is true afterward. Otherwise success is false.
// The items in positions >= position_to_insert are shifted so that
// their positions in the list are one greater than before the insertion.
// Note: Insertion will not be successful if
// position_to_insert < 1 or > ListLength()+1.

+delete( in position: integer,
        out success: boolean): void
// Deletes an item from a list.
// Precondition: position indicates where the deletion
//               should occur.
// Postcondition: If 1 <= position <= list.length(),
//               the item at position position in the list is
//               deleted, other items are renumbered accordingly,
//               and success is true; otherwise success is false.

+retrieve(in position: integer,
          out DataItem: list_item_type,
          out success: boolean): void {query}
// Retrieves a list item by position number.
// Precondition: position is the number of the item to
//               be retrieved.
// Postcondition: If 1 <= position <= list.length(),
```

```
//              DataItem is the value of the desired item and
//              success is true; otherwise success is false.
```

5 The Sorted List ADT

Now we turn to a slightly different ADT, also a list, but one that provides a different set of operations. A **sorted list** is a list in which the elements are ordered by their values. It is distinguished from an unsorted list in the following ways:

- Insertions and deletions must preserve the ordering of the elements.
- The insert operation does not insert an item into a specific position; it is inserted without a supplied position; the list decides where it belongs based on its value.
- The delete operation is given the value of an element, called a **key**, and if it finds the element, it deletes it. It does not delete by position.
- It is often endowed with a "find" operation, which returns the position of an element in the list. This operation might be named locate, or search, or even something like get_position, but the traditional (historical) name is find.

Notice that the list does not need a sort operation! As long as the insert and delete operations perform as described above, the list will always be in sorted order whenever an operation is about to be called. (It may not be in sorted order at certain points in the middle of operations.)

The find operation may seem unnecessary at first. Why would you want the position of an item? The answer is that the retrieve operation is still present in a sorted list, and retrieve still expects a position. Therefore, if you want to get the element whose key is, for argument's sake, "samantha", you would first find the position of that key in the list and then retrieve the element atthat position.

The ADT for a sorted list is therefore similar to that of an unsorted list except that insert and delete are replaced by different ones and find is added:

```
bool is_empty() const;
// Same semantics as the unsorted list operation
// Check if list is empty
// This returns true if the list is empty and false if it is not.

int  length() const;
// Same semantics as the unsorted list operation
// Determines the length of a list
// This returns the length of the list.

void insert( [in] list_item_type new_item,
             [out] bool& success);
// Inserts new_item into a list in a position such that
// the items before new_item are not greater than it and the
// items after new_item are not smaller than it.
// success is true if the insertion succeeded, and false otherwise

void delete( [in] list_item_type new_item,
             [out] bool& success);
// If the list contains an element with value new_item,
// then that element is deleted from the list and success is set to
```

```
// true.
// If not, no deletion takes place and success is set to false.
// NOTE: if the list contains multiple items with value new_item, this
// operation may either:
// remove the first, or
// remove all of them, or
// allow another parameter to specify which way it should behave

void retrieve([in] int position,
              [out] list_item_type& DataItem,
              [out] bool& success) const;
// Same semantics as the unsorted list operation
// Retrieves a list item by position number.

int find( [in] list_item_type DataItem) const;
// If there is an element with key DataItem in the list,
// this returns the index of the first occurrence of DataItem
// otherwise it returns -1 to indicate it is not in the list
```

6 Implementing ADTs

If you are handed a description of an ADT, how do you implement it? The following steps are a good guide.

1. Refine the ADT to the point where the ADT itself is written completely in programming language code. This way you see clearly what the operations must do. The ADT will then be the interface of a C++ class.
2. Make a decision about the internal data structures that will store the data. This decision must be made simultaneously with decisions about major algorithms that manipulate the data. For example, if the data is stored in sorted order, search operations are faster than if it is unsorted. But insertions will take longer in order to maintain the order. Will insertions be very frequent, even more frequent than searches, or will the data be inserted once and never removed, followed by many more searches (like a dictionary and like the symbol table created by a compiler as it compiles a program.) The data structures will be the private data of the C++ class.
3. Write the implementations of all of the functions described in the interface. These implementations should be in a separate implementation file for the class. The section "Separating Class Interface and Implementation" below describes the details of separate interface and implementation files.
4. Make sure that the only operations in the interface are those that clients need. Move all others into the file containing the implementation. In other words, only expose the operations of the ADT, not those used to implement it.

7 C++ Review

This is a review of selected topics that are important for the development of classes in C++. This material was supposed to be covered in the prerequisite courses.

7.1 Separating Class Interface and Implementation

A class definition should always be placed in a `.h` file (called a header file) and its implementation in a `.cpp` file (called an implementation file.) If you are implementing a class that you would like to distribute to

many users, you should distribute the header file and the compiled implementation file, i.e., the object code (either in a `.o` file or compiled into a library file.) The header file should be thoroughly documented. If the implementation needs the interface, which it usually does, put an `#include` directive in the `.cpp` file. The `#ifndef` directive is used to prevent multiple includes of the same file, which would cause compiler errors. `#ifndef X` is evaluated to true by the preprocessor if the symbol `X` is not defined at that point. `X` can be defined by either a `#define` directive, or by a `-DX` in the compiler's command-line options. The convention is to write a header file in the folowing format:

```
#ifndef __HEADERNAME_H #define __HEADERNAME_H

//  interface definitions appear here

#endif // __HEADERNAME_H
```

For those wondering why we need this, remember that the `#include` incorporates the named file into the current file at the point of the directive. If we do not enclose the header file in this pair of directives, and two or more included files contain an include directive for the header file, then multiple definitions of the same class (or anything else declared in the header file) will occur and this is a syntax error.

7.2 Functions with Default Parameters

Any function may have default arguments. A default argument is the value assigned to a formal parameter in the event that the calling function does not supply a value for that parameter. The syntax is:

`return_type function_name (` $t_1\ p_1$`,` $t_2\ p_2$`, ...,` $t_k\ p_k$ `=` d_k`, ...,` $t_n\ p_n$ `=` d_n`);`

If parameter p_k has a default value, then all parametersp_i, with i > k must also have default values. (I.e., all parameters to its right in the list of parameters must have default values also.)

If a function is declared prior to its definition, as in a class interface, the defaults should not be repeated again – it is not necessary and will cause an error if they differ in any way.

If default parameters are supplied and the function is called with fewer than n arguments, the arguments will be assigned in left to right order. For example, given the function declaration,

```
void carryOn( int count, string name = "", int max = 100);
```

`carryOn(6)` is a legal call that is equivalent to `carryOn(6, "", 100)`, and `carryOn(6, "flip")` is equivalent to `carryOn(6, "flip", 100)`.

If argument k is not supplied an initial value, then all arguments to the right of k must be omitted as well. Thus,

```
void bad( int x = 0, int y);
```

is a compile-time error because x has a default but the parameter y to its right does not.

This topic comes up here because default arguments are particularly useful in reducing the number of separate constructor declarations within a class.

7.3 Member Initializer Lists

Consider the following class definitions.

```
class MySubClass
{
public:
    MySubClass(string s) { m_name = s; }
private:
    string m_name;
};

class X
{
public:
    X(int n, string str ): x(n), quirk(str) { }
private:
    const int x;
    MySubClass quirk; MySubClass& pmc;
};
```

The constructor

```
X(int n, string str ): x(n), quirk(str), pmc(quirk) { }
```

causes the constructors for the `int` class and the `MySubClass` class to be called prior to the body of the constructor, in the order in which the members appear in the definition, i.e., first `int` then `MySubClass`. Member initializer lists are necessary in three circumstances:

- To initialize a constant member, such as `x` above;
- To initialize a member that does not have a default constructor, such as `quirk`, of type `MySubClass`;
- To initialize a reference member, such as `pmc` above. References are explained below.

7.4 Declaring and Creating Class Objects

This is a review of the different methods of declaring class objects. The three most common ways to declare an object statically are

```
MyClass obj;                  // invokes default constructor
MyClass obj(params);          // invokes constructor with specified # of parameters
MyClass obj = initial value;  /* if constructor is not explicit and
                                 initial value can be converted to a MyClass object,
                                 this creates a temporary object with the initial
                                 value and assigns to obj using the copy constructor.
                              */
```

If `initial_value` is a `MyClass` object already, just the copy constructor is called. A copy constructor is a special constructor whose only argument is a parameter whose type is the same as the class object being constructed. If a program does not supply a user-defined copy constructor, a (shallow) copy constructor is provided by the compiler. The call to a copy constructor can be avoided by using the following declaration format instead:

```
MyClass obj = MyClass(params); /* This is an exception to the above rule.
                                  The ordinary constructor is used to create the object
                                  obj directly. The right-hand side is not a call
                                  to a constructor.
                               */
```

Although the above declaration looks very much like the right-hand side is invoking a constructor, it is not. The C++ standard allows this notation as a form of type conversion. The right-hand side is like a cast of the parameter list into an object that can be given to the ordinary constructor of the object. Thus no copy constructor is invoked.

```
MyClass obj4(); // syntax error
```

If a member function does not modify any of the class's data, it should be qualified with the `const` qualifier.

7.5 Function Return Values

A function should generally return by value (as opposed to by-reference), as in

```
double sum(const vector<double> & a);
string reverse(const string & w);
```

`sum` returns a double and `reverse` returns a string. Returning a value requires constructing a temporary object in a part of memory that is not destroyed when the function terminates. If an object is large, like a class type or a large array, it may be better to return a reference or a constant reference to it, as in

```
const vector<string> & findmax(const vector<string> & lines);
```

Suppose that `findmax()` searches through the string vector `lines` for the largest string and returns a reference to it. But this can be error-prone – if the returned reference is to an object whose lifetime ends when the function terminates, the result is a runtime error. In particular, if you write

```
int& foo ( )
{
    int temp = 1;
    return temp;
}
```

then your function is returning a reference to `temp`. Because temp is a local variable (technically an automatic variable) of `foo()`, it is destroyed when the function terminates. When the calling function tries to dereference the returned value, a bad run-time error will occur.

Usually you return a reference when you are implementing a member function of a class, and the reference is to a private data member of the class or to the object itself.

8 A C++ Interface for the List ADT

Below is part of the interface for the List ADT described above. It is missing the actual data structures to be used and all private members, which will be supplied only after we learn about lists and linked lists in particular.

```
typedef actual_type_to_use list_item_type;

class List
{
public:
   List();  // default constructor
   ~List(); // destructor

   bool is_empty() const;
   // Determines whether a list is empty.
   // Precondition: None.
   // Postcondition: Returns true if the list is empty,
   //                otherwise returns false.

   int length() const;
   // Determines the length of a list.
   // Precondition: None.
   // Postcondition: Returns the number of items that are currently in the list.

   void insert(int new_position, list_item_type new_item,
               bool& Success);
   // Inserts an item into a list.
   // Precondition: new_position indicates where the
   //               insertion should occur. new_item is the item to be inserted.
   // Postcondition: If 1 <= position <= this->length()+1, new_item is
   //                at position new_position in the list, other items are
   //                renumbered accordingly, and Success is true;
   //                otherwise Success is false.

   void delete(int position, bool& Success);
   // Deletes an item from a list.
   // Precondition: position indicates where the deletion should occur.
   // Postcondition: If 1 <= position <= this->length(),
   //                the item at position position in the list is
   //                deleted, other items are renumbered accordingly,
   //                and Success is true; otherwise Success is false.

   void retrieve(int position, list_item_type & DataItem,
                 bool& Success) const;
   // Retrieves a list item by position number.
   // Precondition: position is the number of the item to be retrieved.
   // Postcondition: If 1 <= position <= this->length(),
   //                DataItem is the value of the desired item and
   //                Success is true; otherwise Success is false.

private:
   // to be determined later

};  // end class
```

When we cover inheritance, you will see that we can make this interface an abstract base class without any private data, and make all of the different solutions to the List ADT subclasses with actual private data.