Chapter 5: Abstraction and Abstract Data Types

Abstraction is the process of trying to identify the most important or inherent qualities of an object or model, and ignoring or omitting the unimportant aspects. It brings to the forefront or highlights certain features, and hides other elements. In computer science we term this process *information hiding*.

Abstraction is used in all sorts of human endeavors. Think of an atlas. If you open an atlas you will often first see a map of the world. This map will show only the most significant features. For example, it may show the various mountain ranges, the ocean currents, and other extremely large structures. But small features will almost certainly be omitted.

A subsequent map will cover a smaller geographical region, and will typically possess more detail. For example, a map of a single continent (such as South America) may now include political boundaries, and perhaps the major cities. A map over an even smaller region, such as a country, might include towns as well as cities, and smaller geographical features, such as the names of individual mountains. A map of an individual large city might include the most important roads leading into and out of the city. Maps of smaller regions might even represent individual buildings.

Notice how, at each level, certain information has been included, and certain information has been purposely omitted. There is simply no way to represent all the details when an artifact is viewed at a higher level of abstraction. And even if all the detail could be described (using tiny writing, for example) there is no way that people could assimilate or process such a large amount of information. Hence details are simply left out.

Abstraction is an important means of controlling complexity. When something is viewed at an abstract level only the most important features are being emphasized. The details that are omitted need not be remembered or even recognized.

Another term that we often use in computer science for this process is *encapsulation*. An encapsulation is a packaging, placing items into a unit, or capsule. The key consequence of this process is that the encapsulation can be viewed in two ways, from the inside and from the outside. The outside view is often a description of the task being performed, while the inside view includes the implementation of the task.

```
double sqrt (double n) {
  double result = n/2;
  while (… ) {
    …
  }
  return result;
}
```

An example of the benefits of abstraction can be seen by imagining calling the function used to compute the square root of a double precision number. The only information you typically need to know is the name of the function (say, sqrt), the argument types, and perhaps what it will do in exceptional conditions (say, if you pass it a negative number). The computation of the square root is actually a nontrivial process. As we described in Chapter 2, the function will probably use some sort of approximation technique, such as Newtons iterative method. But the details of how the result is produced have been abstracted away, or encapsulated within the function boundary, leaving you only the need to understand the description of the desired result.

Programming languages have various different techniques for encapsulation. The previous paragraph described how functions can be viewed as one approach. The function cleanly separates the outside, which is concerned with the “what” – what is the task to be performed, from the inside, the “how” – how the function produces its result. But there are many other mechanisms that serve similar purposes.

```
public interface Stack {
   public void push (Object a);
   public Object top ();
   public void pop ();
   public boolean isEmpty ();
};
```

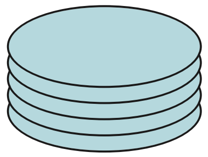
Some languages (but not C) include the concept of an *interface*. An interface is typically a collection of functions that are united in serving a common purpose. Once again, the interface shows only the function names and argument types (this is termed the function *signature*), and not the bodies, or implementation of these actions. In fact, there might be more than one implementation for a single interface. At a higher level, some languages include features such as modules, or packages. Here, too, the intent is to provide an encapsulation mechanism, so that code that is outside the package need only know very limited details from the internal code that implements the package.

Interface Files

The C language, which we use in this book, has an older and more primitive facility. Programs are typically divided into two types of files. Interface files, which traditionally end with a .h file extension, contain only function prototypes, interface descriptions for individual files. These are matched with an implementation file, which traditionally end with a .c file extension. Implementation files contain, as the name suggests, implementations of the functions described in the interface files, as well as any supporting functions that are required, but are not part of the public interface. Interface files are also used to describe standard libraries. More details on the standard C libraries are found in Appendix A.

Abstract Data Types

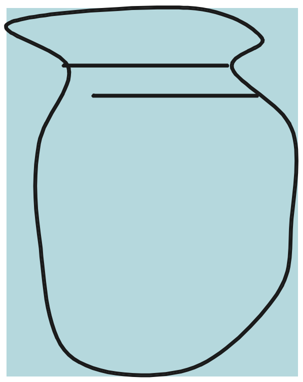
The study of data structures is concerned largely with the need to maintain *collections* of values. These are sometimes termed *containers*. Even without discussing how these collections can be implemented, a number of different types of containers can be identified purely by their purpose or behavior. This type of description is termed an *abstract data type*.



A simple example will illustrate this idea. A *stack* is a collection in which the order that elements are inserted is critically important. A metaphor, such as a stack of plates, helps in envisioning the idea. Only the topmost item in the stack (the topmost plate, for example), is accessible. To second element in the stack can only be accessed by first removing the topmost item. Similarly, when a new item is placed into the collection (a new plate placed on the stack, for example), the former top of the stack is now inaccessible, until the new top is removed.

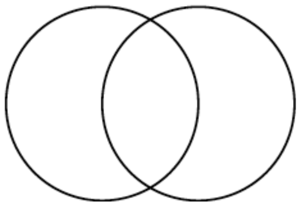
Notice several aspects of this description. The first is the important part played by *metaphor*. The characteristics of the collection are described by appealing to a common experience with non-computer related examples. The second is that it is the *behavior* that is important in defining the type of collection, not the particular names given to the operations. Eventually the operations will be named, but the names selected (for example, push, add, or insert for placing an item on to the stack) are not what makes the collection into a stack. Finally, in order to be useful, there must eventually be a concrete realization, what we term an *implementation*, of the stack behavior. The implementation will, of course, use specific names for the operations that it provides.

The Classic Abstract Data Types

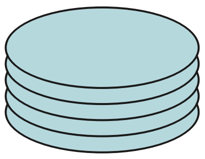


There are several abstract data types that are so common that the study of these collection types is considered to be the heart of a fundamental understanding of computer science. These can be described as follows:

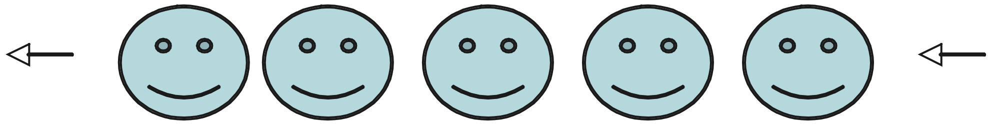
A *bag* is the simplest type of abstraction. A good metaphor is a bag of marbles. Operations on a bag include adding a value to the collection, asking if a specific value is or is not part of the collection, and removing a value from the collection.



A *set* is an extension of a bag. In addition to bag operations the set makes the restriction that no element may appear more than once, and also defines several functions that work with entire sets. An example would be set intersection, which constructs a new set consisting of values that appear in two argument sets. A venn diagram is a good metaphor for this type of collection.



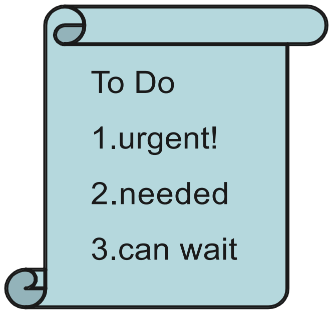
The order that elements are placed into a bag is completely unimportant. That is not true for the next three abstractions. For this reason these are sometimes termed *linear* collections. The simplest of these is the *stack*. The stack abstraction was described earlier. The defining characteristic of the stack is that it remembers the order that values were placed into the container. Values must be removed in a strict LIFO order (last-in, first-out). A stack of plates is the classic metaphor.

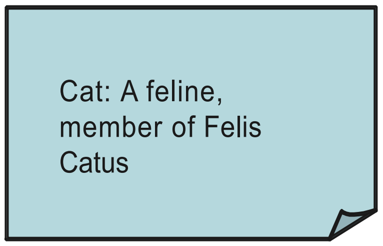


A *queue*, on the other hand, removes values in exactly the same order that they were inserted. This is termed FIFO order (first-in, first-out). A queue of people waiting in line to enter a theater is a useful metaphor.

The *deque* combines features of the stack and queue. Elements can be inserted at either end, and removed from either end, but only from the ends. A good mental image of a deque might be placing peas in a straw. They can be inserted at either end, or removed from either end, but it is not possible to access the peas in the middle without first removing values from the end.

A *priority queue* maintains values in order of importance. A metaphor for a priority queue is a to-do list of tasks waiting to be performed, or a list of patients waiting for an operating room in a hospital. The key feature is that you want to be able to quickly find the most important item, the value with highest priority.





A *map*, or *dictionary*, maintains pairs of elements. Each key is matched to a corresponding value. They keys must be unique. A good metaphor is a dictionary of word/definition pairs.

Each of these abstractions will be explored in subsequent chapters, and you will develop several implementations for all of them.

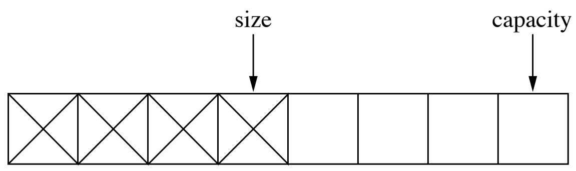
Implementations

Before a container can be used in a running program it must be matched by an *implementation*. The majority of this book will be devoted to explaining different implementations techniques for the most common data abstractions. Just as there are only a few classic abstract data types, with many small variations on a common theme, there are only a handful of classic implementation techniques, again with many small variations.

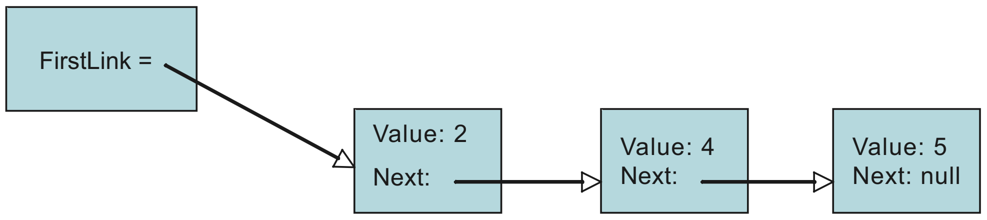
The most basic way to store a collection of values is an *array*. An array is nothing more than a fixed size block of memory, with adjacent cells in memory holding each element in the collection:

element 0	element 1	element 2	element 3	element 4

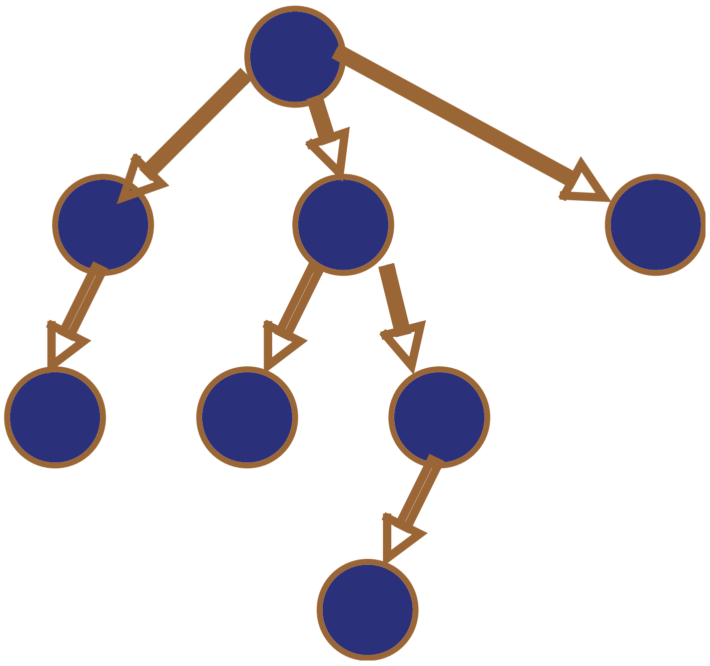
A disadvantage of the array is the fixed size, which typically cannot be changed during the lifetime of the container. To overcome this we can place one level of indirection between the user and the storage. A *dynamic array* stores the size and capacity of a container, and a pointer to an array in which the actual elements are stored. If necessary, the internal array can be increased during the course of execution to allow more elements to be stored. This increase can occur without knowledge of the user. Dynamic arrays are introduced in Worksheet 14, and used in many subsequent worksheets.



The fact that elements in both the array and the dynamic array are stored in a single block is both an advantage and a disadvantage. When collections remain roughly the same size during their lifetime the array uses only a small amount of memory. However, if a collection changes size dramatically then the block can end up being largely unused. An alternative is a *linked list*. In a linked list each element refers to (points to) the next in sequence, and are not necessary stored in adjacent memory locations.



Both the array and the linked list suffer from the fact that they are linear organizations. To search for an element, for example, you examine each value one after another. This can be very slow. One way to speed things up is to use a tree, specifically a *binary tree*. A search in a binary tree can be performed by moving from the top (the root) to the leaf (the bottom) and can be much faster than looking at each element in turn.



There are even more complicated ways to organize information. A *hash table*, for example, is basically a combination of an array and a linked list. Elements are assigned positions in the array, termed their *bucket*. Each bucket holds a linked list of values. Because each list is relatively small, operations on a hash table can be performed very quickly.