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 ();
};
```

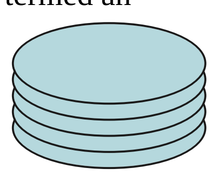
Some languages (but not C) include the concept of an *interface*. An interface is typically a collection of functions that are united in serving a common purpose. Once again, the interface shows only the function names and argument types (this is termed the function *signature*), and not the bodies, 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

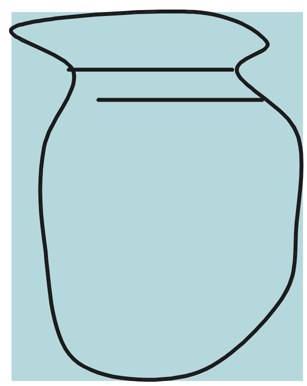
The study of data structures is concerned largely with the need to maintain *collections* of values. These are sometimes termed *containers*. Even without discussing how these collections can be implemented, a number of different types of containers can be identified purely by their purpose or behavior.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

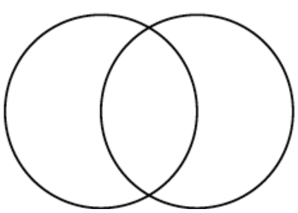
Notice several aspects of this description. The first is the important part played by *metaphor*. The characteristics of the collection are described by appealing to a common experience with non-computer related examples. The second is that it is the *behavior* that is important in defining the type 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

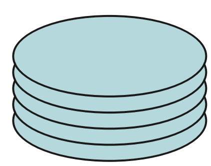


There are several abstract data types that are so common that the study of these collection types is considered to be the heart of a fundamental understanding of computer science. These can be described as follows:

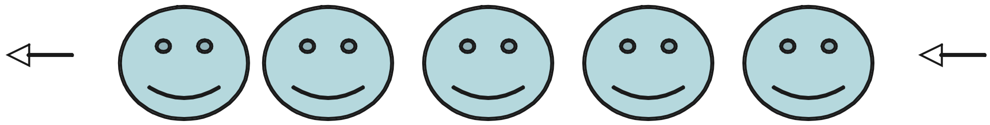
A *bag* is the simplest type of abstraction. A good metaphor is a bag of marbles. Operations on a bag include adding a value to the collection, asking if a specific value is or is not part of the collection, and removing a value from the collection.



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.



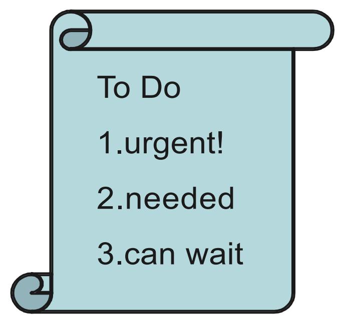
The order that elements are placed into a bag is completely unimportant. That is not true for the next three abstractions. For this reason these are sometimes termed *linear* collections. The simplest of these is the *stack*. The stack abstraction was described earlier. The defining characteristic of 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.

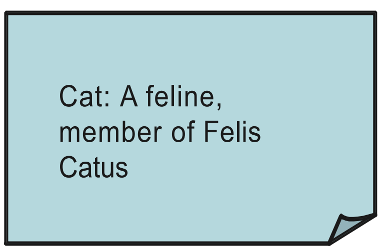


A *queue*, on the other hand, removes values in exactly the same order that they were inserted. This is termed FIFO order (first-in, first-out). A queue of people waiting in line to enter a theater is a useful metaphor.

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.

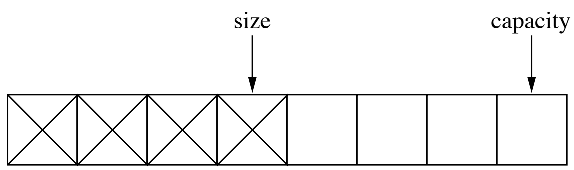
Implementations

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.

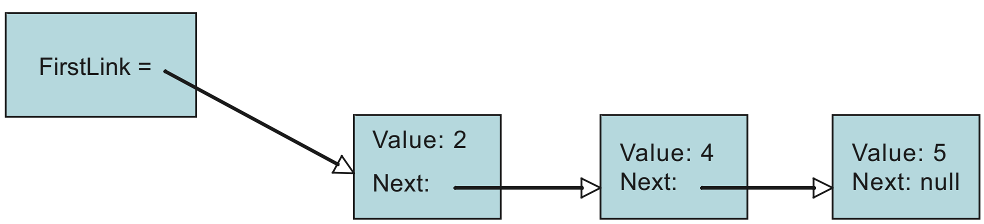
The most basic way to store a collection of values is an *array*. An array is nothing more than a fixed size block of memory, with adjacent cells in memory holding each element in the collection:

element 0	element 1	element 2	element 3	element 4

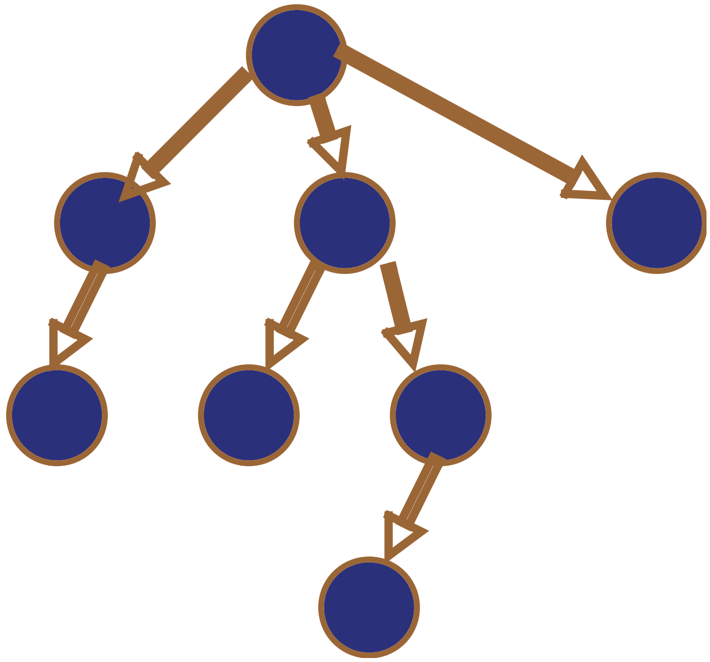
A disadvantage of the array is the fixed size, which typically cannot be changed during the lifetime of the container. To overcome this we can place one level of indirection between the user and the storage. A *dynamic array* stores the size and capacity of a container, and a pointer to an array in which 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.



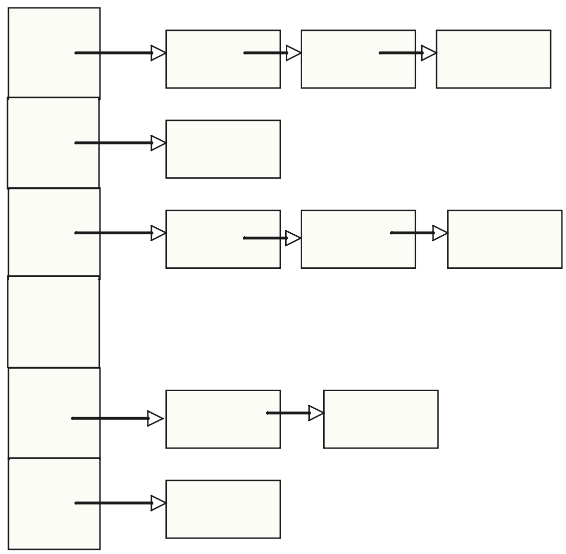
The fact that elements in both the array and the dynamic array are stored in a single block is both an advantage and a disadvantage. When collections remain roughly the same size during their lifetime the array uses only a small amount of memory. However, if a collection changes size dramatically then 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.



Many more variations on these themes, such as skip lists (a randomized tree structure imposed on a simple linked list), or heaps (a binary tree organized as a priority queue) will be presented as we explore this topic.

Some Words Concerning C

In the first part of this book we have made little reference to the type of values being held in a collection. Where we have used a data structure, such as the array used in sorting algorithms, the element type has normally been a simple floating-point number (a double). In the development that follows we want to generalize our containers so that they can maintain values of many different types. Unfortunately, C provides only very primitive facilities for doing so.

A major tool we will use is symbolic name replacement provided by the C preprocessor. This facility allows us to define a name and value pair. Prior to compilation, the C preprocessor will systematically replace each occurrence of the name with a value. For example, we will define our element type as follows:

```
# define EleType double
```

Following this definition, we can use the name EleType to represent the type of value our container will hold. This way, the user need only change the one definition in order to modify the type of value a collection can maintain.

Another feature of the preprocessor allows us to make this even easier. The statement #ifndef informs the preprocessor that any text between the statement and a matching #endef statement should only be included if the argument to the ifndef is *n*ot already *def*ined. The definition of TYPE in the interface file will be written as follows

```
# ifndef TYPE
# define TYPE double
# endif
```

These statements tell the preprocessor to only define the name EleType if it has not already been defined. In effect, it makes double into our default value, but allows the user to provide an alternative, by preceding the definition with an alternative definition. If the user wants to define the element type as an integer, for example, they simple precede the above with a line

```
# define TYPE integer
```

A second feature of C that we make extensive use of in the following chapters is the equivalence between arrays and pointers. In particular, when an array must be allocated dynamically (that is, at run time), it is stored in a pointer variable. The function used to allocate memory is termed malloc. The malloc function takes as argument an integer representing the number of bytes to allocate. The computation of this quantity is made easier by another function, sizeof, which computes the size of its argument type.

You saw an example of the use of malloc and sizeof in the merge sort algorithm described in Chapter 4. There the malloc was used to create a temporary array. Here is another example. The following bit of code takes an integer value stored in the variable n, and allocates an array that can hold n elements of whatever type is represented by EleType. Because the malloc function can return zero if there is not enough memory for the request, the result should always be checked. Because malloc returns an indetermined pointer type, the result must be cast to the correct form.

```
int n;
TYPE * data;
…
n = 42; /* n is given some value */
…
data = (TYPE *) malloc(n * sizeof(TYPE)); /* array of size n is allocated */
assert (data != 0);  /* check that allocation worked */
…
free (data);
```

Dynamically allocated memory must be returned to the memory manager using the free operation. You should always make sure that any memory that is allocated is eventually freed.

This is an idiom you will see repeatedly starting in worksheet 14. We will be making extensive use of pointers, but treating them as if they were arrays. Pointers in C can be indexed, exactly as if were arrays.

Study questions

1. What is abstraction? Give three examples of abstraction from real life.

2. What is information hiding? How is it related to abstraction?

3. How is encapsulation related to abstraction?

4. Explain how a function can be viewed as a type of encapsulation. What information is being hidden or abstracted away?

5. What makes an ADT description abstract? How is it different from a function signature or an interface?

6. Come up with another example from everyday life that illustrates the behavior of each of the six classic abstractions (bag, stack, queue, deque, priority queue, map).

7. For each of the following situations, describe what type of collection seems most appropriate, and why. Is order important? Is time of insertion important?

a. The names of students enrolled in a class.

b. Files being held until they can be printed on a printer.

c. URLs for recently visited web pages in a browser.

d. Names of patients waiting for an operating room in a hospital emergency ward.

e. Names and associated Employee records in a company database.

8. In what ways is a set similar to a bag? In what ways are they different?

9. In what ways is a priority queue similar to a queue? In what ways are they different?

10. Once you have completed worksheet 14, answer this question and the ones that follow. What is a dynamic array?

11. What does the term capacity refer to in a dynamic array?

12. What does the term size refer to in a dynamic array?

13. Can you describe the set of legitimate subscript positions in a dynamic array? Is this different from the set of legal positions in an ordinary array?

14. How does the add function in a dynamic array respond if the user enters more values than the current capacity of the array?

15. Suppose the user enters 17 numbers into a dynamic array that was initialized with a capacity of 5? What will be the final length of the data array? How many times will it have been expanded?

16. What is the algorithmic execution time for the `_dyArrayDoubleCapacity`, if n represents the size of the final data array?

17. Based on your answer to the previous question, what is the worst case algorithmic complexity of the function `dyArrayAdd`?

Analysis Exercises

1. Explain, in your own words, how calling a function illustrates the ideas of abstraction and information hiding. Can you think of other programming language features that can also be explained using these ideas?

2. Even without knowing the implementation, you can say that it would be an error to try to perform a *pop* operation on a stack if there has not been a preceding *push*. For

each of the classic abstractions describe one or more sequences of actions that should always produce an error.

3. This question builds on the work you began with the preceding question. Without even looking at the code some test cases can be identified from a specification alone, independent of the implementation. As you learned in Chapter 3, this is termed *black box testing*. For example, if you push an item on to a stack, then perform a pop, the item you just pushed should be returned. For each of the classic data structures come up with a set of test cases derived simply from the description.

4. Contrast an interface description of a container with the ADT description of behavior. In what ways is one more precise than the other? In what ways is it less precise?

5. Once you have completed worksheet 14 you should be able to answer the following. When using a partially filled array, why do you think the data array is doubled in size when the size exceeds the capacity? Why not simply increase the size of the array by 1, so it can accommodate the single new element? Think about what would happen if, using this alternative approach, the user entered 17 data values. How many times would the array be copied?

Programming Projects

1. In worksheet 15 you explore why a dynamic array doubles the size of its internal array value when the size must be increased. In this project you can add empirical evidence to support the claim that this is a good idea. Take the dynamic array functions you developed in the worksheets 14 and 15 and add an additional variable to hold the "unit cost". As we described in worksheet 15, add 1 each time an element is added without reallocation, and add the size of the new array each time a reallocation occurs, plus 1 for the addition of the new element. Then write a main program that will perform 200 additions, and print the average cost after each insertion. Do the values remain relatively constant?

On the Web

Wikipedia (http://en.wikipedia.org/wiki/Main_Page) has a good explanation of the concept of abstract data types. Links from that page explore most of the common ADTs. Another definition of ADT, as well as definitions of various forms of ADTs, can be found on DADS (http://www.nist.gov/dads/) the *Dictionary of Algorithms and Data Structures* maintained by the National Institute of Standards and Technology. Wikipedia has entries for many common C functions, such as malloc. There are many on-line tutorials for the C programming language. A very complete tutorial has been written in Brian Brown, and is mirrored at many sites. You can find this by googling the terms "Brian Brown C programming".