4.1 Preliminaries

The ADT list, as described in the previous chapter, has operations to insert, delete, and retrieve items, given their positions within the list. A close examination of the array-based implementation of the ADT list reveals that an array is not always the best data structure to use to maintain a collection of data. An array has a **fixed size**—at least in most commonly used programming languages—but the ADT list can have an arbitrary length. Thus, in the strict sense, you cannot use an array to implement a list because it is certainly possible for the number of items in the list to exceed the fixed size of the array. When developing implementations for ADTs, you often are confronted with this fixed-size problem. In many contexts, you must reject an implementation that has a fixed size in favor of one that can grow dynamically.

In addition, although the most intuitive means of imposing an order on data is to order it physically, this approach has its disadvantages. In a physical ordering, the successor of an item x is the next data item in sequence after x, that is, the item "to the right" of x. An array orders its items physically and, as you saw in the previous chapter, when you use an array to implement a list, you must shift data when you insert or delete an item at a specified position. Shifting data can be a time-consuming process that you should avoid, if possible. What alternatives to shifting data are available?

To get a conceptual notion of a list implementation that would not involve shifting, consider Figure 4-1. This figure should help free you from the notion that the only way to maintain a given order of data is to store the data in that order. In these diagrams, each item of the list actually *points to* the next item. Thus, if you know where an item is, you can determine its successor, which can be anywhere physically. This flexibility not only allows you to insert and delete

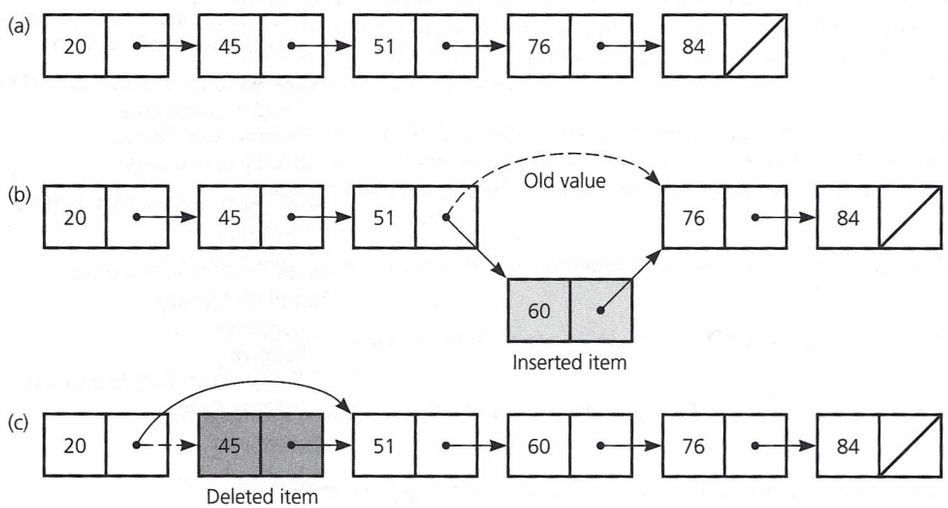


FIGURE 4-1

(a) A linked list of integers; (b) insertion; (c) deletion

data items without shifting data, it also allows you to increase the size of the list easily. If you need to insert a new item, you simply find its place in the list and set two **pointers**. Similarly, to delete an item, you find the item and change a pointer to bypass the item.

An item in a linked list points to its successor

Because the items in this data structure are *linked* to one another, it is called a **linked list**. As you will see shortly, *a linked list is able to grow as needed*, whereas an array can hold only a fixed number of data items. In many applications, this flexibility gives a linked list a significant advantage.

Before we examine linked lists and their use in the implementation of an ADT, we need to know more about pointers. Like many programming languages, C++ has pointers that you can use to build a linked list. The next section discusses the mechanics of these pointers.

Pointers

When you declare an ordinary variable `x` to be `int`, the C++ compiler allocates a memory cell that can hold an integer. You use the identifier `x` to refer to this cell. To put the value 5 in the cell, you could write

```
x = 5;
```

To display the value that is in the cell, you could write

```
cout << "The value of x is " << x << endl;
```

A **pointer variable**, or simply a **pointer**, contains the location, or **address** in memory, of a memory cell. By using a pointer to point to a particular memory cell, you can locate the cell and, for example, determine its content.

Figure 4-2 illustrates a pointer `p` that points to a memory cell containing an integer.

The notion of one memory cell that refers to another memory cell is a bit tricky. In Figure 4-2, keep in mind that the content of `p` is not a typical value. The content of `p` is of interest only because it tells you where in memory to look for the integer value 5. That is, you can get to the integer value *indirectly* by using the address that `p` contains.

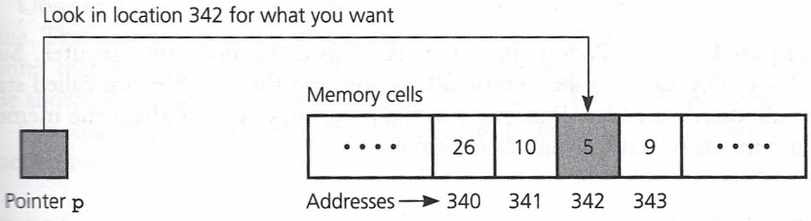


FIGURE 4-2
A pointer to an integer

Now for two big questions:

- How do you get a pointer variable `p` to point to a memory cell?
- How do you use `p` to get to the content of the memory cell to which `p` points?

Before we answer either question, we need to declare `p` as a pointer variable. For example, the declaration

`p` is a pointer variable

```
int *p;
```

declares `p` to be an integer pointer variable; that is, `p` can point only to memory cells that contain integers. You can declare pointers to any type except files.

You need to be careful when declaring more than one pointer variable. The declaration

`q` is not a pointer variable

```
int *p, q;
```

declares `p` to be a pointer to an integer, but declares `q` to be an integer. That is, the statement is equivalent to

```
int *p;
int  q;
```

To declare both `p` and `q` correctly as integer pointer variables, write

```
int *p
int *q;
```

or[1]

```
int *p, *q;
```

Memory for the pointer variables `p` and `q`, and for the integer variable `x` in

```
int x;
```

is allocated at compilation time; that is, before the program executes. Such memory allocation is called **static allocation** and the variables are called statically allocated variables. Execution of the program does not affect the memory requirements of statically allocated variables.

---

1. In the context of pointers, the `*` operator is unary (like the `!` operator) and right-associative. Whether you write `int *p` or `int* p`, `*` applies to the variable `p` and not the data type `int`.

Initially, the contents of p, q, and x are undetermined, as Figure 4-3a illustrates. However, you can place the address of x into p and therefore have p point to x by using the C++ address-of operator &, as follows:

```
p = &x;
```

Figure 4-3b illustrates the result of this assignment. Notice that the assignment statement

```
p = x;  // THIS STATEMENT IS ILLEGAL
```

is illegal because there is a type clash: x is an integer variable, while p is a pointer variable, which can contain only an address of a memory cell that contains an integer.

*p is the memory cell to which p points

Pointer p now points to a memory cell. The notation *p represents *the memory cell to which p points.* You can store a value in the memory cell to which p points by writing the assignment statement

```
*p = 6;
```

as Figure 4-3c illustrates. (You could, of course, write x = 6 to make the same assignment.) After this assignment, the expression *p has the value 6, because 6 is now the value in the memory cell to which p points. Thus, you could, for example, use cout << *p to display 6.

Memory allocation can also occur at execution time and is called **dynamic allocation**. A variable allocated then is called a dynamically allocated variable. C++ enables dynamic allocation of memory by providing the operator new, which acts on a data type, as in

new allocates memory dynamically

```
p = new int;
```

The expression new int allocates a new memory cell that can contain an integer and returns a pointer to this new cell, as Figure 4-3d illustrates. The initial content of this new cell is undetermined. Note that new char would allocate a new memory cell that can contain a character, and so on. If, for some reason, new cannot allocate memory, it throws the exception std::bad_alloc, which is in the <new> header.

Observe that this newly created memory cell has no programmer-defined name. The only way to access its content or to put a value in it is indirectly via the pointer that new creates, that is, by using *p in the previous example. As Figure 4-3e shows, the statement *p = 7 assigns 7 to the newly created memory cell.

Suppose that you now assign to the pointer q the value in p by writing the statement

Copying a pointer

```
q = p;
```

Pointer q now points to the same memory cell that p points to, as Figure 4-3f illustrates. Alternatively, you could let q point to a new memory cell, as Figure

4-3g shows, and assign a value to this new cell. These steps are like the ones pictured in parts d and e of this figure.

Suppose that you no longer need the value in a pointer variable. That is, you do not want the pointer to point to any particular memory cell. C++ environments provide the constant `NULL`,[2] which you can assign to a pointer of any type. By convention, a `NULL` pointer value means that the pointer does not point to anything. Do not confuse a pointer variable whose value is `NULL` with one whose value is not initialized. Until you explicitly assign a value to a newly declared pointer variable, its value—like that of any other variable—is undefined. You should not assume that its value is `NULL`. In Figure 4-3a, `p` and `q` are examples of pointer variables whose values are undefined.

A pointer whose value is `NULL` does not point to anything

Now suppose that you no longer need a dynamically allocated memory cell. Simply changing all pointers to the cell wastes memory, because the cell remains allocated to the program, even though it is no longer accessible. For example, Figure 4-3h shows the result of assigning `NULL` to `p`. (In the figures, a diagonal line represents a `NULL` value.) The cell to which `p` originally pointed—it still contains 7—is in limbo. To avoid this situation—which is called a **memory leak**—C++ provides the operator `delete` as a counterpart to `new`. Conceptually, the expression

```
delete q
```

`delete` returns memory to the system for reuse

returns to the system the memory cell to which `q` points. That is, `delete` in effect deallocates memory from a program, thus freeing the memory for future use by the program. Because `delete` does not deallocate `q` itself and leaves the content of `q` undefined, a reference to `*q` at this point can be disastrous. Thus, you should assign `NULL` to `q` after applying the `delete` operator as a precaution against following `q` to a deallocated memory cell. Figure 4-3i shows the results of these actions.

`delete q` does not deallocate `q`; it leaves `q` undefined

However, consider the situation

```
p = new int;
q = p;
delete p;
p = NULL;
```

A pointer to a deallocated memory cell is possible and dangerous

as illustrated in Figure 4-4. Even though `p` is `NULL`, `q` still points to the deallocated node. Later the system might reallocate this node—via the `new` operator—and `q` might still point to it. You can imagine some of the errors that might ensue if a program mistakenly followed the pointer `q` and reached a node within an entirely unexpected data structure! It would be useful if the `delete` operator could eliminate the potential for this type of program error by setting to `NULL` all pointers to a deallocated node. Unfortunately, this task is

2. Several header files, such as `cstdlib` and often `cstddef`, define `NULL`. Its value is 0. Many C++ programmers prefer to use 0 instead of `NULL`. However, for clarity, this book uses `NULL`.

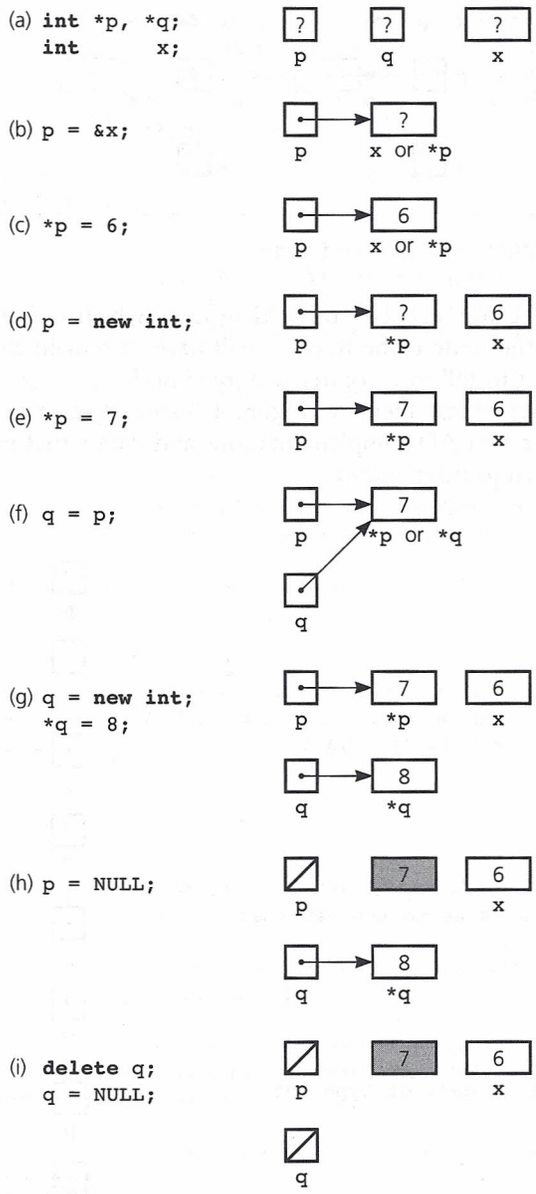


FIGURE 4-3

(a) Declaring pointer variables; (b) pointing to statically allocated memory; (c) assigning a value; (d) allocating memory dynamically; (e) assigning a value; (f) copying a pointer; (g) allocating memory dynamically and assigning a value; (h) assigning ***NULL*** to a pointer variable; (i) deallocating memory

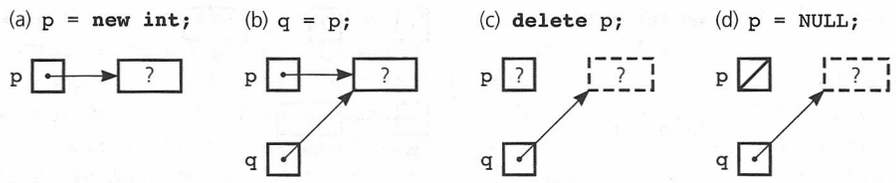


FIGURE 4-4

An incorrect pointer to a deallocated node

very difficult. Because *delete* cannot determine which variables—in addition to *p*—point to the node to be freed, it will have to remain the programmer's responsibility not to follow a pointer to a freed node.

The sequence of statements in Figure 4-5 should serve to illustrate pointers further. Note that ADT implementations and data structures that use C++ pointers are called **pointer-based**.

```
int *p, *q;

p = new int;          // Allocate a cell of type int.

*p = 1;               // Assign a value to the new cell.

q = new int;          // Allocate a cell of type int.

*q = 2;               // Assign a value to the new cell.

cout << *p << " "     // Output line contains: 1 2
     << *q << endl;   // These values are in the
                      // cells to which p and q point.
```

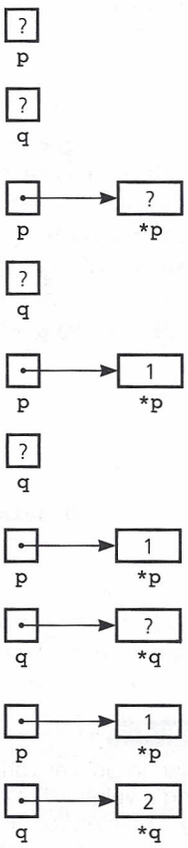


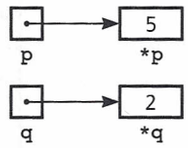
FIGURE 4-5

Programming with pointer variables and dynamically allocated memory

(continues)

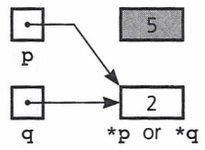
```
*p = *q + 3;          // The value in the cell to which
                      // q points, 2 in this case, and 3
                      // are added together. The result is
                      // assigned to the cell to which
                      // p points.

cout << *p << " "     // Output line contains: 5 2
     << *q << endl;
```



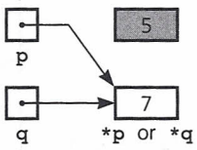

```
p = q;                // p now points to the same cell as q.
                      // The cell p formerly pointed to is
                      // lost; it cannot be referenced.

cout << *p << " "     // Output line contains: 2 2
     << *q << endl;
```

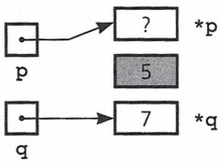



```
*p = 7;               // The cell to which p points (which
                      // is also the cell to which q points)
                      // now contains the value 7.

cout << *p << " "     // Output line contains: 7 7
     << *q << endl;
```




```
p = new int;          // This changes what p points to,
                      // but not what q points to.
```




```
delete p ;            // Return to the system the cell to
                      // which p points.

p = NULL;             // Set p to NULL, a good practice
                      // following delete.

q = NULL;             // The cell to which q previously
                      // pointed is now lost. You cannot
                      // reference it.
```

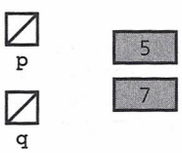


FIGURE 4-5

(continued from previous)

KEY CONCEPTS

C++ Pointer Variables

1. The declaration

```
int *p;
```

(continues)

C++ Pointer Variables *(continued)*

statically allocates a pointer variable `p` whose value is undefined but is not `NULL`. The pointer variable `p` can point to a memory cell that contains an integer in this example.

2. The statement

```
p = new int;
```

dynamically allocates a new memory cell that can contain an integer. The pointer variable `p` points to this new cell. (However, see item 5 on this list.)

3. The expression `*p` represents the memory cell to which the pointer variable `p` points.
4. If the pointer variable `p` contains `NULL`, it does not point to anything.
5. If `new` cannot allocate memory, it throws the exception `std::bad_alloc`. Thus, a statement such as

```
p = new int;
```

needs to be contained within a `try` block followed by an exception handler for the `bad_alloc` exception. (`std::bad_alloc` is defined in the <new> header.)

6. The statement

```
delete p;
```

returns to the system the memory cell to which `p` points. It does not delete `p` itself. Remember that `p` is a variable, and as such, its lifetime is not affected by `delete`.

Dynamic Allocation of Arrays

When you declare an array in C++ by using statements such as

An ordinary C++ array is statically allocated

```
const int MAX_SIZE = 50;
double anArray[MAX_SIZE];
```

the compiler reserves a specific number—`MAX_SIZE`, in this case—of memory cells for the array. This memory allocation occurs before your program executes, so it is not possible to wait until execution to give `MAX_SIZE` a value. We have already discussed the problem this fixed-size data structure causes when your program has more than `MAX_SIZE` items to place into the array.

You just learned how to use the *new* operator to allocate memory dynamically; that is, during program execution. Although the previous section showed you how to allocate a single memory cell, you actually can allocate many cells at one time. If you write

Use the *new* operator to allocate an array dynamically

```
int arraySize = 50;
double *anArray = new double[arraySize];
```

The pointer variable *anArray* will point to the first item in an array of 50 items. Unlike *MAX_SIZE*, *arraySize* can change during program execution. You can assign *arraySize* a value and, thus, determine how large your array will be at execution time. Good, but how do you use this array?

Regardless of how you allocate an array—statically, as in the first example, or dynamically, as in the second—you can use an index and the familiar array notation to access its elements. For example, *anArray[0]* and *anArray[1]* are the first two items in the array *anArray*.

You also can use a pointer offset notation to reference any array element. C++ treats the name of an array as a pointer to its first element. For example,

An array name is a pointer to the array's first element

anArray is equivalent to *anArray[0]*

(anArray+1) is equivalent to *anArray[1]*

and so on. (This notation uses pointer arithmetic.) We are not suggesting that you use this notation, however.

When you allocate an array dynamically, you need to return its memory cells to the system when you no longer need them. As in the previous section, you use the *delete* operator to perform this task. To deallocate the array *anArray*, you write

delete returns a dynamically allocated array to the system for reuse

```
delete [ ] anArray;
```

You write brackets when you apply *delete* to an array.

Now suppose that your program uses all of the array *anArray*, despite having determined its size during execution. You can allocate a new and larger array, copy the old array into the new array, and finally deallocate the old array. The following statements double the size of *anArray*:

You can increase the size of a dynamically allocated array

```
double *oldArray = anArray;           // copy pointer to array
anArray = new double[2*arraySize];    // double array size
for (int index = 0; index < arraySize; ++index)
   anArray[index] = oldArray[index]; // copy old array
delete [ ] oldArray;                  // deallocate old array
```

Subsequent discussions in this book will refer to both statically and dynamically allocated arrays. Our array-based ADT implementations will use statically allocated arrays for simplicity. The programming problems will ask you to create array-based implementations that use dynamically allocated arrays.