PART VII

USING ARRAYS AND STRUCTURES

When you need to handle several objects of the same kind, arrays come in very handy, because they eliminate the need to designate each object by a different name. This tremendously simplifies your programs, because the code is essentially the same no matter which object in the array you manipulate.

In Part VII, you will learn how to manipulate arrays of any class of objects, including arrays of numeric variables. Words, sentences, and text in general can also be manipulated by using arrays of characters. Finally, to further improve your skills in application development, you will make additional improvements to the point-of-sale terminal and to the satellite simulation.

SKILL

NINETEEN

USING ARRAYS

- Declaring and using arrays
- Using arrays of arrays
- Using numeric arrays
- Sorting arrays

In this Skill, you will use athletes to illustrate the purpose of arrays and the use of arrays to denote collections of identical objects of any type. After you grasp the initial concepts using arrays of athletes, you will study and use the more commonplace arrays of numeric data.

You will learn how to declare arrays, how to access elements of arrays, how to use arrays in expressions, how to pass arrays as arguments, and how to receive arrays as parameters in functions.

Understanding and Working with Arrays

An *array* is a collection of objects of the same class in which you can designate one of the objects by its position in line. For example, if the athletes in a fitness class are aligned and you do not know their names, the easiest thing for you to do would be to refer to each one of them by their relative position in line—the first, the second, the third….

It is important that you understand the convenience of using arrays. Let's take a very simple example: Suppose that you want to write a program to tell five athletes, such as the ones shown in Figure 19.1, to exercise.

You would probably consider creating five athletes and giving each one a name:

```
athlete Julia, Andrea, Ricardo, Andy, Michael;
```

Then, you would have to tell each one of them to exercise:

```
JumpJack(Julia);
JumpJack(Andrea);
JumpJack(Ricardo);
JumpJack(Andy);
JumpJack(Michael);
```

Well, maybe you can do this so far, but what if there are one hundred athletes? What if there are one thousand?

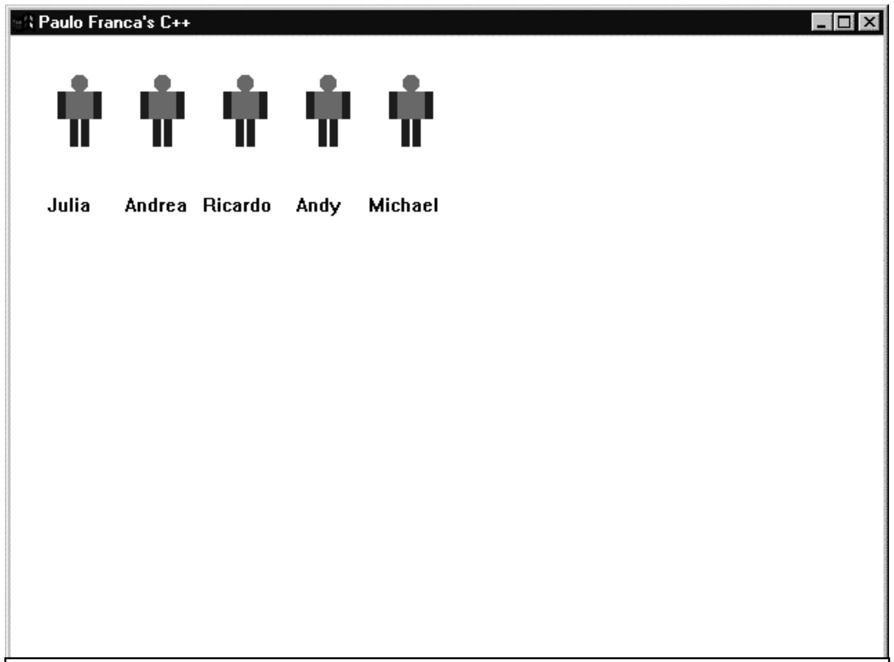


Fig 19.1 Five individual athletes

A convenient way to deal with this problem is to give a name to a collection of athletes, and then to refer to each individual athlete as the first, the second, the third….

The process would be something like the following process:

```
Let Guy be a collection of five athletes.
Repeat for each value of index from 1 to 5:
        Take the Guy in position index and make him do a jumping jack.
```

As it has been with all the repetitions we have studied, the computer will make sure that all the *Guys* will exercise, but you, the almighty programmer, do not have to write one thousand lines of code—no matter how many *Guys* you want to deal with! Figure 19.2 shows this array of athletes.

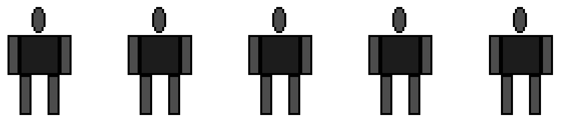


Fig 19.2 Guy is an array of athletes

Now, compare a program that deals with each athlete individually:

```
void mainprog()
{
     athlete Julia, Andrea, Ricardo, Andy, Michael;
     JumpJack(Julia);
     JumpJack(Andrea);
     JumpJack(Ricardo);
     JumpJack(Andy);
     JumpJack(Michael);
}
```

with a program that uses an array of five athletes:

```
void mainprog()
{
     athlete Guy[5];            // Declares an array of five athletes
     for (int which=0; which<=4; which++)
     {
          JumpJack(Guy[which]);// Each athlete does a jumping jack
     }
}
```

The last program will be essentially the same whether you want to deal with 5, 10, or 100 athletes. The first program would have to be modified according to how many athletes you use.

Do you remember that we said an array is composed of elements of the same class? Very well, you can also use objects of the class `runner` if you'd like:

```
void mainprog()
{
     runner Guy[5];         // Declares an array of five runners
     for (int which=0;which<=4;which++)
     {
          Guy[which].run();// Each runner runs
     }
}
```

Ways to Use Arrays

You can have arrays of all kinds of objects: athletes, runners, robots, and circles, and even the most modest `int` and `float` can be elements of an array. In C++, the elements of an array are designated by an index value, which is an integer (`int`). The first element of an array is designated by the index value *0,* the second element by the index value *1,* and so on.

In the previous program, the array `Guy` had five elements:

`Guy[0]` was the first.
`Guy[1]` was the second.
`Guy[2]` was the third.
`Guy[3]` was the fourth.
`Guy[4]` was the fifth.
`Guy[5]` did not exist, since there were only five elements!

☠ It is a common bug to declare an array with *N* elements, and to then try to use element *N,* which, of course, does not exist. The element indexes start with zero, and go to *N-1.* Don't forget to use element zero.

Using Arrays in Functions

If you want to use an array in a function, you must declare it, so the compiler knows that you are using an array. You declare an array much like you declare any other object, but you follow the array name with brackets enclosing the number of elements in the array. For example:

```
int  points[10];
```

declares an array named `points` with 10 elements (numbered 0 through 9), in which each element is an `int`.

```
float  measurements[12];
```

declares an array named `measurements` with 12 elements (numbered 0 through 11), in which each element is a `float`.

The array name denotes the whole collection of elements. The array name followed by an index value inside brackets denotes one particular element. For example, in the first example in this section

`points` …

is a collection of 10 elements.

`points[1]` …

is one particular element (the second) in `points`.

As long as the index value is an `int`, you can use any expression in the brackets:

- `Guy[which]` … denotes one particular *Guy*. If you know the value of `which`, you can determine which element that particular *Guy* is.
- `Guy[which-1]` … denotes another particular *Guy*.

However, the index value should not denote a nonexistent element in the array. In the case of `Guy[which]`, `which` should have values between 0 and 4. Any other value will be outside the bounds of the array. It is your responsibility, as a programmer, to make sure that this does not happen.

ASSIGNING THE WRONG INDEX VALUE

A major cause of errors in programs is to assign an index value that denotes an invalid entry. You may often need to use an expression in brackets to denote the index value, and you can never be sure what values the expression will take during the program execution. For example, suppose that you declare an array `List` with 10 elements (numbered 0 through 9). Suppose that you use an array like the following:

```
List[ (k-3)*j]=0;
```

The values of `k` and `j` determine the index value (which should be between 0 and 9). However, it is unlikely that you will watch everything that happens to `k` and `j` in the program, and you may eventually have a value like *–13* as your result.

In a case like this one, the computer will usually go to the wrong place in the memory, and will place the *zero* where it thinks the `List[-13]` should be. The consequences will be unpredictable!

Using Arrays in Expressions

You can use arrays in expressions—they will usually be arrays that have numeric elements, such as `int` and `float`. Since each element is usually a number, the value of that element will be used when you use that element in an expression.

For example, consider the following piece of program:

```
void mainprog()
{
     int  value[10];                 // Declares an array of 10
integers
          for (int  index=0;index<=9; index++)
          {
               value [index]=9-index;// Computes the value for each
element
          }
...
```

Can you determine the value of each element of the array after this piece of program is executed? What will be the value of element *0*? What about element *1*?

Notice that in the loop, we compute an expression `9-index`, and we store the result in `value[index]`.

Therefore, after the piece of program above, the array `value` will have a number stored in it. We can use this number for anything.

For example, we can use a box to "say" the value of each element:

```
void mainprog()
{
  int  value[10];        // Declares an array of 10 integers
  for (int  index=0;index<=9; index++)
  {
        value [index]=9-index;// Computes each value
  }
  Box Sal;
  for (which=0;which<=9;which++)
  {
       Sal.say(value[which]);
  }
}
```

Can you see that we can use the array element as an argument? I suggest that you try this to determine whether you guessed correctly the values in the array.

Using Arrays as Arguments

Arrays and array elements can also be used as arguments in functions. It is important that you know exactly which type of object you are using. For example, if you declare

```
athlete Guy[10];
```

there is a big difference between using

```
JumpJack(Guy);
```

and using

```
JumpJack(Guy[1]);
```

The first form is wrong! Can you understand why? The `JumpJack` function requires one parameter of the type `athlete`. Indeed, `Guy[1]` is an athlete, but `Guy` is not. `Guy` is an array of athletes!

It is possible to use each athlete as an argument to the `JumpJack` function and to have each one of them do the jumping jacks. However, it is impossible to use the whole array as an argument, since the function expects to deal with only one object. We can define a different version of the `JumpJack` function that takes an array of athletes as an argument. In this case, the following statement:

```
JumpJack(Guy);
```

is correct.

Using Arrays in Parameter Lists

When a function receives an array as a parameter, the function header must indicate this fact. For example:

```
void jumpjack(athlete eachone[10], int howmany_athletes)
{
   for (int i=0;i<howmany_athletes;i++)
   {
      eachone[i].ready();
      eachone[i].up();
      eachone[i].ready(0.);
   }
}
```

This function can be called with an array of athletes as an argument. If you expect to always have 10 athletes, you may want to omit the `howmany_athletes` argument.

In this case, the function header specifies clearly that `eachone` is an array. It tells the compiler to expect that indexes will be associated with it. Another interesting fact is that you don't really need to specify the size of the array. Since the array was declared somewhere else in the program, memory space will not be allocated again in the function, because arrays are passed by reference in C++. All the function needs to know is that `eachone` may be followed by an index.

The function header could then be as follows:

```
void jumpjack (athlete eachone[], int howmany_athletes)
```

If you have a multidimensional array (you will learn about these arrays later in this Skill), only the last bracket can be left blank.

The *c7jack.cpp* Program

The `c7jack.cpp` program demonstrates how to use an array of athletes and how to have each athlete perform a jumping jack.

```
//                                c7jack.cpp
// This programs illustrates use of arrays.
//                         July 31, 1994
//
#include"franca.h";
athlete Guy[7];
void JmpJack(athlete somebody)
{
     somebody.up();
     somebody.ready();
}
void mainprog()
{
 for (int i=0;i<7;i++)
 {
   JmpJack(Guy[i]);
   Guy[i].say("Done!");
 }
}
```

This program will remain essentially unchanged whether you have only one athlete or several athletes. The difference would be as follows:

- The array should be declared with as many elements as needed.
- The loop should go from *zero* to the number denoting the last athlete.

The Number of Elements in an Array

If you declare an array with seven elements and use only three elements, the program should run correctly. The only problem is that you will leave unused space in the computer memory. On the other hand, if you want to use more elements than you have declared, you may cause an error in the program.

In many situations, you may have to deal with an array in which you don't know exactly the number of elements that will be used. For example, suppose that you have to deal with the ages of students in a classroom. You may have 28 students in your classroom, but if you make a program that deals with 28 students, you will not be able to use the same program with another class. When you develop programs, you should try to make them as useful as possible.

One solution would be to declare the array as follows:

```
int student_age[28];
```

Then, you could exchange the number *28* for another number—for example, 32—to deal with a larger class. This is not a good solution, because other places in the program may also use the number of students. For example, if you compute the average age of the students, you will certainly have a piece of program like the following one:

```
sum=0;
for (int i=0; i<28; i++)
   {
      sum=sum+student_age[i];
    }
average=sum/28;
```

Did you notice that there are two other places that you may have to change the value *28*, and use 32 instead? It may look easy to do this in the example, but if you have a very long program, it will not be so easy to locate all the values!

A more important case is the case in which you want to use this piece of program with different classes in a school, in which each class is likely to have a different number of students. As a novice, you may think that all you have to do is to change the number of students in the program for each new class you want to process. This method is extremely unprofessional!

☞ Keep in mind that you, the professional programmer, are not expected to be present while the program runs.

A good solution for this problem is to keep the number of students separately, and to give it a name:

```
int number_of_students;
```

This variable could either be read from the keyboard or be passed as an argument to a function. The array would have been declared with enough elements to accommodate all the students. For example:

```
int student_age[100];
```

Then, the part that computes the average could be as follows:

```
sum=0;
for (int i=0; i<number_of_students; i++)
   {
      sum=sum+student_age[i];
   }
average=sum/number_of_students;
```

Constants and Array Sizes

It is illegal to use a variable to indicate the array size in a declaration. We must use a constant. Why? It is not possible to use a variable in the array declaration. You must use an integer. For example:

```
athlete Guy[n];
```

is incorrect, unless `n` is a constant.

WHY DOES THE ARRAY SIZE NEED TO BE A CONSTANT?

The value of a variable will only be known during program execution. Also, as the name implies, a variable may have its value changed as the program executes. Long before the program is able to execute, the compiler needs to set apart memory space to accommodate all the variables the program will need (including the arrays). This is why the number of elements must be known—the compiler must reserve memory space for all elements. By the time the program starts executing and the variables start having values assigned to them, all the memory has already been allocated. For this reason, the size of the array must be specified as a constant. It is possible to define a named constant and to use it as the array size. For example:

```
const int arraysize;

float prices[arraysize];
```

However, remember that since `arraysize` was defined as a constant, its value cannot be changed during program execution.

Try This for Fun…

Change the program `c7jack.cpp` so you can give the number of athletes that you want to see. Have each athlete do as follows:

- Say “Hi.”
- Perform a jumping jack.
- Say their index value (“0, 1, 2…”).

Numeric Arrays

In many situations, we use arrays containing numeric values. For example, we use arrays of the types `int`, `long`, `float`, or `double`.

An Array of Arrays

You can have arrays of several types of elements. Here's a brain teaser: Can you have an array, in which each element is an array?

Suppose that we have a group of athletes that have shown up for the eight o'clock class. If we treat them as an array, we can denote them by their index values—0, 1, 2… Now, suppose that another group comes in for the nine o'clock class. They can also be treated as an array, can't they?

In that case, we could designate that each group is an array, and then we could have the following groups:

- group 0 (the eight o'clockers)
- group 1 (the nine o'clockers)
- group 2 (the ten o'clockers)

You can now understand that what we call *group* is actually an element of an array. For example, suppose that the array we are talking about consists of three groups of four athletes each. This situation is shown in Figure 19.3.

We could declare this situation as follows:

```
 athlete Guy[3][4];
```

in which `Guy` is an array of three elements, in which each element is an array of four athletes.

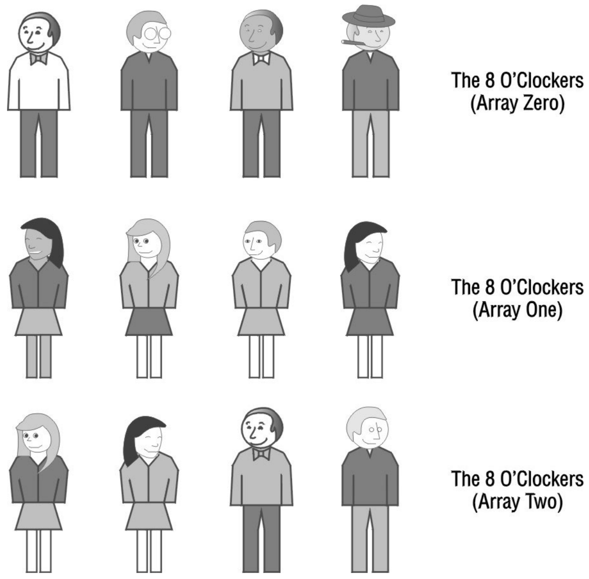


Fig 19.3 Array of arrays

Arrays, Rows, and Columns

If you look at Figure 19.3, you may notice that the athletes are arranged in three rows of four athletes each. You may also notice that each row contains athletes of the same group; therefore, when we refer to *group zero,* we actually refer to *row zero*—each row corresponds to one group.

When elements are arranged in tabular form, such as the athletes in Figure 19.3, you can designate one element by the row and the column where the element is located. Therefore, `Guy[2][3]` denotes the athlete in row 2, column 3. Figure 19.4 shows the rows and columns in the array of athletes.

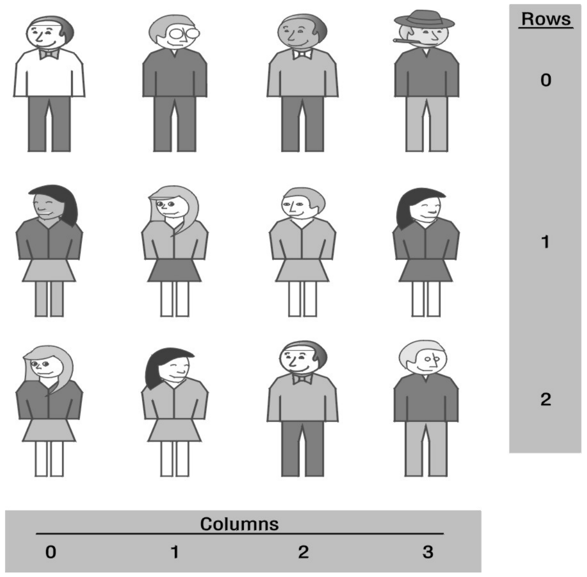


Fig 19.4 Rows and columns in array of athletes

Initialization

In the same way that you are responsible for any object or variable in the program, you, as a programmer, are responsible for setting the appropriate values for the elements of the array. When an array is created, its elements contain any garbage that was in the computer's memory. You should not assume that the values of the elements are zero or any other value.

You can assign a value to each element of an array during the execution of the program, or you can assign an initial value as soon as you declare your array. In the case of an array that holds numeric values, you can set an initial value for each element at the declaration. Simply follow the declaration with an equal sign and a set of braces that enclose the initial value for each element separated by commas. The first value corresponds to the first element (numbered zero), the second value to the second element (numbered 1), and so on.

```
int values[10] = {9,8,7,6,5,4,3,2,1,0};
```

The declaration above assigns the value 9 to element `[0]`, the value 8 to element `[1]`, the value 7 to element `[2]`, and so on.

Here are some notes on the syntax:

- There is an equal sign after the brackets.
- The values of each element are separated from each other by commas.
- All the values are enclosed in braces.

When you initialize a two-dimensional array as follows:

```
int matrix[2][3] = { {0,0,1},{2,3,4} };
```

the last index value is changed first. In this case, the initialization will have the same effect as the following statements:

```
matrix[0][0] =0;
matrix[0][1] =0;
matrix[0][2] =1;
matrix[1][0] =2;
matrix[1][1] =3;
matrix[1][2] =4;
```

In the sequence of statements above, notice that, at first, the first index value was kept at 0, while the second index value changed to become 0, 1, and then 2. Then, the first index value was changed to 1, while the second index value again changed to become 0, 1, and then 2.

Using Arrays

Suppose that you want to keep track of the ages of five athletes in a group in an array. You can declare an array as follows:

```
athlete guy[5];
int age[5];
```

You can have each athlete "say" his or her number, ask you for his or her age, and, finally, "say" his or her age.

The *c7age.cpp* Program

The procedure above can be accomplished by the `c7age.cpp` program, shown below.

```
//                                c7age.cpp
// This program illustrates use of integer arrays.
//                           July 31, 1994
//
#include"franca.h"
athlete Guy[5];
void mainprog()
{
 int age[5];
 for (int i=0;i<=4;i++)
 {
   Guy[i].ready();
   Guy[i].say(i);
   age[i]=Guy[i].ask("What is my age?");
 }
```

Using Numeric Arrays

You may have noticed in the examples above that we did not use the athlete array too often. Most of the work was done by the `age` array. As a matter of fact, we deal very often with simple numeric arrays in real life. For example, you may have to deal with an array that contains the grades of students in a class, or that contains the balances in customer accounts. If we have only the `age` array, instead of having an array of athletes and the `age` array, the program will not be too different.

Try This for Fun…

- Write another program that contains only an `age` array to find out the value and location of the greatest age. Use the `ask` function to request the values of the ages and to tell you the greatest one after the comparisons are made. Use a box to display your result.

Avoiding the Use of Unnecessary Arrays

Arrays are needed only when you are required to keep all the values at hand for later use. Unless other requirements demand the use of an array, you should decide whether you even need an array. For example, if you want to input several values and to compute their average, there is no need to use an array. You can add the numbers as you read them, and divide the total by the number of elements:

```
Box result ("Average:");
int n; // The number of elements
float value, total;
n = ask ("Input number of elements");
total = 0;

for (int i=1;i<=n;i++)
{
   total=total+ask("Input a number");
}
total = total / n;
result.say(total);
```

In the example above, the numbers are added as they are typed. There is no need to use an array to store all the numbers. If you use unnecessary arrays, they may consume computer memory and make your program harder to understand.

Defining a Type with *typedef*

If you are using several arrays that have the same declaration—for example:

```
float  matrix[5][3], prices[5][3], cost[5][3];
```

you may consider creating a new type:

```
typedef float theusual[5][3];
```

and then declaring the arrays that you are going to use:

```
theusual matrix,prices,cost;
```

In this case, you simply create a new type of variable. From this point on, the compiler will recognize `theusual` as a type that represents a floating point array of five by three elements. `typedef` can be used in any situation, not only with arrays.

Here is the syntax:

```
typedef  type declaration    identifier ;
```

in which the identifier of the new type may be followed by an array size.

Try These for Fun…

- Modify the `c7age.cpp` program so the oldest athlete presents him- or herself. You need to include more code that determines who is the oldest athlete. Then, make the athlete say “Here!” Notice that you have to determine whether the oldest athlete is `Guy[0]`, `Guy[1]`… You will need to find the index value of the *Guy* who is the oldest athlete. Notice that the solution to this problem is an algorithm to find the largest element in an array.

> ☞ Use an integer—for example, `oldest`—to examine one athlete at a time, and make the integer equal to zero (to point to the first athlete). Then, compare the age of the athlete examined by `oldest` with the age of every other athlete. Every time you find an age that is greater than the one you are examining, make `oldest` point to this new athlete. This way, `oldest` always indicates the oldest athlete that you have examined. After you have examined all the athletes, `oldest` will point out the oldest athlete.

- The problem of finding the smallest element in an array is very similar to the problem above. Try to find the youngest athlete.
- Write the code of a function `float average(int array[], int from, int to);` to compute and return as a result the average value of the array elements, starting from the position `from` and ending at the position `to`.
- Write the code of a function `int less (int array[], int from, int to, int value);` to compute and return as a value the number of elements whose values are less than the argument `value`, starting from the position `from` and ending at the position `to`. For example, if the array contains the values 21, 32, 15, 80, 75, and 43, and the function call is `x = less (array, 1, 4, 50)`, the result should be 2.

Example—Handling a Numeric Array

In this example, we will read a set of numbers and store them in an array. Some operations are performed, such as searching for the largest number and sorting numbers. The numbers in the array are displayed in a list of boxes, which, in turn, illustrates how to deal with an array of objects. This example deals with an integer array of 10 numbers.

Getting and Showing the Array

In the simplest version of this program, we will read and show the array in boxes on the screen. The steps involved in this program are as follows:

- Declare the objects and the variables that are needed.
- Get the values for the array.
- Show the values for the array.

This simple list of steps suggests that we develop a function to get the values for the array and another function to show the values for the array. We may have a program like the following program:

```
void mainprog()
{
  int values[10];      // Array of numbers
  getarray(values,0,9);
  showarray(values,0,9);
}
```

The program simply consists of two function calls—a call to a function `getarray` (to get the values for the array from the keyboard) and a call to a function `showarray` to display the array on the screen. Both functions will have to be programmed.

There are several ways to conceive of these two functions—the main concern is to determine what kind of arguments should be passed to them.

The *getarray* function

The `getarray` function needs to know how many elements to get from the keyboard, and where to store them. We may consider building a function with no arguments that would be called as follows:

```
getarray();
```

This function would work OK. In this case, you may have to declare the array `number` outside the `mainprog` function to make it global, so the other functions can use it. You also may assume that 10 elements are to be read, and that they will always be read from element 0 to element 9. The disadvantage of this approach is that your functions will not be reusable. You may want to read and show several arrays, but if you restrict yourself to reading always the same array name, the following difficulties arise:

- You cannot use this function if you want to get values for two different arrays. (If you are not convinced, go ahead and try it!)
- If you want to use this function in another program, you must also name the array in the other program `number`. You can give up all hope of selling software with this kind of restriction!

On the other hand, if you include arguments, it will make your functions a little more general and more reusable. A similar reasoning applies to the function `showarray`. The main program is now very simple and complete. All we have to do now is to code the two functions.

Reading the Array

The `getarray` function requests numbers from the keyboard, and stores them in the array. Since several elements are to be read, this process has to be done in a loop. A possible solution to this problem could be as follows:

```
void getarray(int number[],int from,int to)
{
  for (int i=from;i<=to;i++)
  {
    number[i]=ask("Input a number");
  }
}
```

This function is also a very simple function.

In C++, arrays are always passed by reference to functions, whether you precede the parameter name with the ampersand (&) or not. Therefore, the values read from the keyboard will be available to the main program.

Showing the Array

The `showarray` function displays each value of the numeric array in a separate box.

Here is a possible implementation:

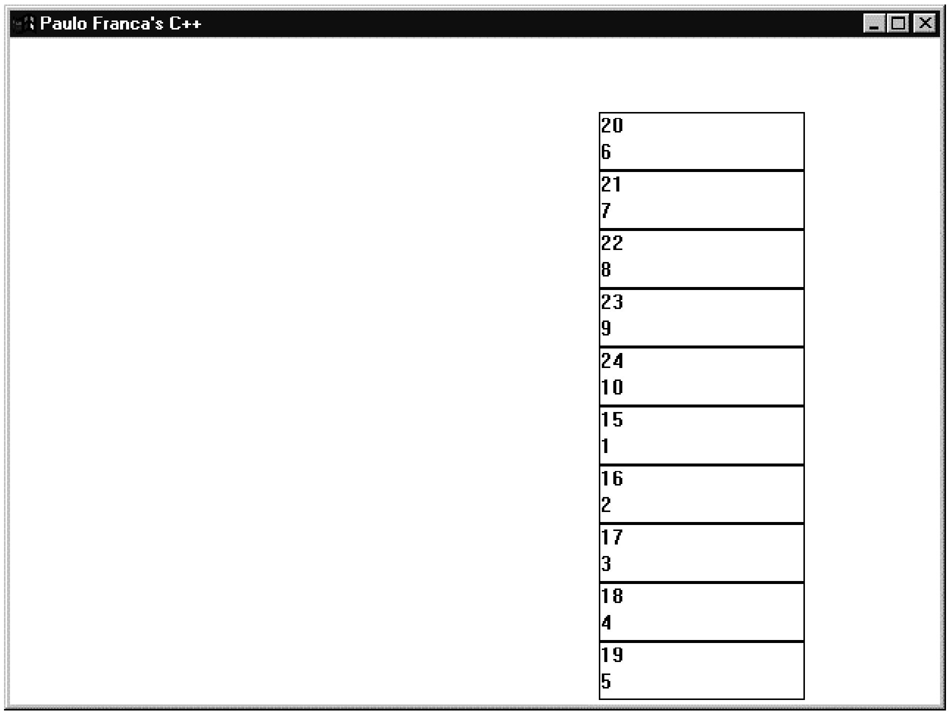
```
void showarray(int number[],int from,int to)
{
  Box list[10];
  for (int i=from;i<=to;i++)
  {
       list[i%10].place(400,50+(i%10)*40);
       list[i%10].label(i);
       list[i%10].say(number[i]);
  }
}
```

Since the range of array indexes is beyond our control, this function displays up to 10 boxes, and places the array elements in the boxes according to the last digit of the array index. For example, if the user requests the display of elements 31 to 40, element 31 is displayed in box 1, element 32 is displayed in box 2…. Element 40 is displayed in box 0, because the index of the array of boxes is the remainder of division by 10. Figure 19.5 shows how this function displays array elements 15 to 24.

This function does not use the feature that automatically places boxes on the screen. Instead, each box is individually placed at given coordinates, so each new box will be automatically placed in a new location, and a new set of boxes is created every time the function is called. After a few calls to this function, the boxes will be placed off the screen.

At this point, you should try to run this program.

The functions used in this section can be found in the header file `c7intfun.h`.



```
20
6
21
7
22
8
23
9
24
10
15
1
16
2
17
3
18
4
19
5
```

Fig 19.5 Displaying 10 array elements

Finding the Largest Element

The next step in this array manipulation is to search the array to find the largest value. We will use the same functions as above to get and to show the array, and essentially the same program. Instead of including the search in the program, we can conceive of another function to find the largest element in the array. This function call could be as follows:

```
findlargest(values,from, to);
```

It is especially useful to include arguments that let us search from a given start to a given limit, as we shall see shortly. This function must, somehow, identify the *largest value found* or *where it was found*! Do you understand clearly the two options?

Suppose that we have the following numbers in the array:

23, 12, 45, 11, 89, 21, 32, 55, 81, 32

If you examine the values, you will determine that the largest value is 89. But you may also say that the largest value is in position 4 (remember to start counting from 0, as is the case in C++). If the result is known by the index position where the largest number was found, you can easily learn its value. But if you know the largest value, it is not easy to find the index position. For this reason, you should return the index position as a result.

Comparing Two Numbers at a Time

How do you find the largest value in an array?

As you know very well by now, the computer can compare two numbers at a time. It cannot look at a whole set, and then find the largest. You must tell the computer to examine each number. A simple solution to this problem is to have the computer remember the largest value that has been found as each value is examined in the array.

In the beginning, you can safely assume that the first value in the array is the largest one found so far (since you have not examined any other value). Then, as you loop through all the elements, check whether the value you think is the largest is still the largest. If it is, fine. If it is not, just dispose of it and keep the new value that is the largest.

An implementation could be as follows:

```
int findlargest(int number[],int from,int to)
{
 int index;
 int guess;
 guess=from;            // Take the first as a guess
 for (index=from;index<=to;index++)
  {
    if(number[found]<number[index])
    {
           guess=index; // Change the guess
       }
  }
  return guess;
}
```

Can you follow the idea? The variable `guess` keeps the position of the largest number found so far. In the beginning, since no numbers have been examined, use the first position (`from`) as your guess. Now, since you guess that this position holds the largest number so far, you assume that `number[guess]` is the largest number.

Next, start looping through the array. Use the integer variable `index` to denote the current position you are examining. The index starts with `from`, which is the first position to examine, and ends with `to`, which is the last position.

☞ You may also consider looping from `from+1` onward, because we are already holding the first element, and it is no help to compare `number[found]` with `number[index]` when both `found` and `index` are equal. It is a good idea, but it may fail in the extreme case in which `from` and `to` have the same value.

As you compare the number you think is the largest (`number [guess]`) with the current element (`number[index]`), you correct the position of the largest number. If the current number is larger, you forget the old position and copy the current position (`index`) to `guess`. When you do so, you remain sure that `guess` always designates the element containing the largest value. As you reach the end of the loop, you can be sure that guess designates the position of the largest value found in the array.

You can include a call `findlargest` in the main program to find the largest element. Of course, once this value is found, it is a good idea to display it. The program below includes a couple extra boxes to display the largest value, as well as its position in the array.

```
void mainprog()
{
  Box largest(50,350,"Largest:");
  Box index(50,400,"Position:");
  int values[10];
  int k;
  getarray(values,0,9);
  showarray(values,0,9);
  k=findlargest(values,0,9);
  largest.say(values[k]);
  index.say(k);
}
```

Sorting Arrays

Since you have functions to help you easily get, show, and find the largest element in an array, you can also sort the array. In other words, you can rearrange it so all the elements appear in ascending order, for example. Sorting is a very common and useful procedure that deserves a lot more study than we present here. We do present a very simple sorting algorithm.

Again, instead of adding more statements to our program, we can build a `sort` function and call it from the main program. It could be as follows:

```
 sort(values,from,to);
```

This function keeps the main program simple, and, in addition, provides a `sort` function that may be useful another time!

The idea of the `sort` function is simple. Since you can easily find the largest element, why not move it to the end of the array, where it belongs? If you know that the largest element is in position `larger`, and that the last element is in position `last`, all you have to do is to exchange `number[larger]` for `number[last]`. At least you can be sure that the last element is in its correct place. How does this help, though? You are still left with elements 0 to 8, which are not sorted. But wait! Can't we now tell the `findlargest` function to search for the largest number from 0 to 8? Of course, we can!

EXCHANGING VALUES

It can be tricky for beginning programmers to exchange values in two variables. To exchange the values in variables `a` and `b`, some beginners are tempted to do as follows:

```
a=b;

b=a;
```

Of course, this will not work. As `b` is copied into `a`, the old value of `a` is lost. Therefore, when `a` is copied into `b`, the new value of `a`, which is the same as the value of `b`, is copied. To solve this problem, we must use a third variable; for example, `temp`.

```
temp=a;

a=b;

b=temp;
```

This problem is similar to the problem of exchanging the contents of two glasses—one containing wine, another containing milk. You need to use a third glass to hold the wine, and then you pour the milk into the glass that contained the wine. It is only after you do this that you can pour the wine into the glass that contained the milk.

As we keep selecting the largest element and moving it to the end, we are left each time with a smaller unsorted array. At some point, we are left with an unsorted array of only one element, and, at this point, we know that the array is sorted!

Here is an implementation:

```
void sort (int array[],int fromwhere,int towhere)
{
  int largest,temp;
  // Loop decreasing "last" each time:
  for (int last=towhere;last>fromwhere;last--)
  {
    // Find largest from beginning to "last":
    largest=findlargest(array,fromwhere,last);
    // Switch largest with "last":
    temp=array[largest];
    array[largest]=array[last];
    array[last]=temp;
  }
}
```

The main program to sort the array would then be as follows:

```
void mainprog()
{
  int values[10];
  getarray(values,0,9);
  showarray(values,0,9);
  sort(values,0,9);
  showarray(values,0,9);
}
```

Recursive Sorting

A variation of the `sort` function uses the following idea: You can check whether there is only one element to be sorted. In that case, you are done. Otherwise, you can find the largest element, exchange it with the element in the last position, and then sort the remaining elements.

When I say *sort the remaining elements*, what do I mean by it? Well, I mean to call the `sort` function again. This is a recursive solution, since the `sort` function includes a call to itself.

An implementation of this variation could be as follows:

```
void recsort(int array[],int fromwhere,int towhere)
{
  // This is a recursive version of sort:
  int largest,temp;
  int last;
    last=towhere;
    // Check whether sort is not complete:
    if(fromwhere<=towhere)
    {
    // Find largest from beginning to "last":
    largest=findlargest(array,fromwhere,last);
    // Switch largest with "last":
    temp=array[largest];
    array[largest]=array[last];
    array[last]=temp;
    // Sort the remaining array
    recsort(array,fromwhere,last-1);
    }
}
```

Are You Experienced?

Now you can…

Manipulate several objects of identical types by using arrays

Use arrays of objects or numbers

Use arrays of arrays

Sort an array

SKILL

TWENTY

WORKING WITH TEXT AND PRACTICING WITH ARRAYS

- Using character arrays
- Manipulating strings with the functions in `string.h`
- Converting between numbers and characters
- Understanding and using structures
- Searching arrays

You can manipulate text in C++ by using character arrays, in which each element of the array contains one character. Most of the usual operations with character strings can be performed using available functions. You don't have to write programs to copy one string to another or to compare strings.

You may also have to deal with information that is organized in several pieces, such as an account number, a customer name, or a current balance. This problem can be solved with objects, but it can also be solved with structures.

Finally, when you have data stored in an array, you may have to search the array to locate a specific element. You will learn how to search at the end of this Skill.

Using Text in Your C++ Programs

Another important application of computers is to manipulate text. Even though the computer stores everything in its memory as a number, characters can be stored and manipulated by using numeric codes. The code the computer uses to store the characters you are dealing with is irrelevant in most cases. You will be able to manipulate characters without knowing their codes.

Using *char* in Your Code

The type `char` can be used for variables that are supposed to store characters. For example:

```
char initial;
```

establishes that the variable `initial` will be used to store a character. Each `char` variable can store only one character. For this reason, it is very likely that handling characters will require the use of arrays. You can use a character variable in an expression, and you can assign a value corresponding to a character by enclosing the character in single quotes. For example:

```
 initial = 'A';
```

assigns the value corresponding to the character `A` (uppercase) to the variable `initial`. We could have also set an initial value at the time that we declared the variable. For example:

```
char choice='y';
```

Notice that codes are not the same for upper- and lowercase letters. The program can differentiate between *a* and *A*.

Comparing by Using *char* Codes

Of course, character variables can also be compared, just like any other variable:

```
char choice;
...
...
if (choice=='N') break;
```

In the example above, the character variable `choice` is compared with the character `N` (uppercase). If `choice` contains the code representing the character `N`, a `break` occurs.

Adding a blank space

It is possible to use a blank space in the quotes to indicate a blank space. A blank space has a code just like any other character:

```
char blank=' ';
```

The statement above will declare a variable `blank` of type `char`, with an initial value of a blank space.

Characters and Character Codes

It is legal to assign a numeric (`int`, `long`, `float`, etc.) value to a character variable. Since the character is represented by a code that is an integer ranging from 0 to 255, it is legal in C++ to assign an integer value. In fact, a character enclosed in single quotes signifies the numeric code of the character. In other words, 'N' is the same as 78, 'A' is the same as 65, 'a' is the same as 97, and so on. Don't worry about the numeric codes—they were established as standard codes for communicating characters in electronic form. You will find more information on this topic in an exercise later in this Skill. If the numeric value is not in the range of 0 to 255, the computer will truncate the number, and the result may not be what you expected.

It is also legal to assign a character to an integer variable. However, it is illegal to assign more than one character (a character array, as we will see in the next section) to a character variable.

Each character position in the computer's memory can hold a numeric code ranging from 0 to 255, which means that a total of 256 characters can be represented. This total is enough to represent the letters of the Roman alphabet in lower- and uppercase, special signs, and control characters. All we have to do is to associate each code with a character. Most computers use the correspondence known as the American Standard Code for Information Interchange (ASCII).

Try This for Fun…

- Assume the following declarations in a program:

  ```
  int i,j,k;
  float x,y;
  char a,b;
  ```

  - Indicate which of the following statements are correct in terms of syntax:

    a. `i=j;`
    b. `a=j;`
    c. `x=a;`
    d. `for (a='a';a<='z';a++)`
    e. `b=b+3;`

Using Character Arrays

Since each variable can hold only one character, the obvious choice to handle words and phrases is an array of characters. Some programming languages have a type to handle *strings*, which are sequences of characters. C++ does not have a built-in string type, so it uses arrays. However, keep in mind that C++ allows classes. Therefore, you can either build a class to handle strings or buy one!

Character arrays are declared in the same way that you declare other arrays. For example:

```
char first_name[20];
```

Each element of the array can be used. For example:

```
first_name[0]='a';
first_name[1]='n';
first_name[2]=first_name[0];
```

The example above results in the name `ana` being stored in the first three positions of the array `first_name`.

It is also possible to initialize the array as you declare it:

```
char last_name[20]= { 'f','r','a','n','c','a'};
```

Keeping Track of Word Length

Since words and names have varying lengths, and since every memory position either is initialized with a value or contains some trash, how will we know that a particular last name contains six characters? Obviously, if this variable was created to hold only that name, we could remember the size—this is not usually the case, though. It is useful to be able to tell where the string ends.

Although there is no string type, C++ offers some functions to manipulate strings, and offers some amenities for programmers. By convention, strings in C++ are marked with a special code that follows the last position. This code is referred to as `null`. It is not a keyword, but it is defined in almost every header file used in C++. In fact, `null` is defined as the value *zero* (not the character *0*). To correctly initialize `last_name` in the example above, we should have done as follows:

```
char last_name[20]= { 'f','r','a','n','c','a',null};
```

or:

```
char last_name[20]= { 'f','r','a','n','c','a',0};
```

The zero is not enclosed in quotes, because we mean to use the value *zero,* not the character *0*.

We are lucky that C++ offers another way to initialize character arrays:

```
char last_name[20] = "franca";
```

This statement works the same as the previous statement worked. When you enclose a sequence of characters in double quotes, you indicate to the compiler that you are representing a string. The compiler automatically inserts `null` after your string, provided you have one array element available to accommodate the `null`. Sequences of characters terminated with `null` are called *null-terminated strings*.

As you may remember, athletes can “say” something that is enclosed in double quotes. Why? A sequence of characters that is enclosed in double quotes is a null-terminated string. In general, you can substitute any message that is enclosed in double quotes with any null-terminated string.

For example, you could do as follows:

```
Sal.say(last_name);
```

Similarly, null-terminated strings can be used as labels in boxes and as messages that ask for data from the keyboard.

String Arrays

It is possible to manipulate an array in which each element is a character array. This process is a simple use of arrays of arrays. For example:

```
char name [5][20];
```

declares an array of five elements, in which each element is an array of 20 characters. This kind of array is useful to store a list of names, for example. Notice that `name[0]` is an array of characters, and so are `name[1]`, `name[2]`, `name[3]`, and `name[4]`. We can store a null-terminated string in each of these arrays. For example, the following loop:

```
for (int i=0;i<=4;i++)
  askwords(name[i],20,"Enter a name");
```

reads a name for each of the five arrays of 20 characters.

A more readable version of the same program could be as follows:

```
typedef char onename[20];  // Creates a type
onename  namelist[5];      // namelist has five names
for (int i=0;i<=4;i++)
     askwords(namelist[i],20,"Enter a name");
```

Suppose that you want to store the names of a few athletes: you could have an array of five athletes, and an array name as declared above. You could store the name of the first athlete in `name[0]`, the name of the second athlete in `name[1]`, and so on.

It is also possible to initialize a string without specifying the length. For example:

```
char first_name []="Mitiko";
```

You must include the brackets, even if they are empty.

In this case, the character array `first_name` will be declared with a size of 6 (the five letters plus the `null` terminator). This feature is convenient because it saves you the time you would have to spend counting how many spaces you need to store a given string. However, keep in mind that in this case, the array will not be able to accommodate a first name that has more than five characters.

Inputting a string with *askwords*

You can use the function `askwords`, which is included in `franca.h`, to input a string from the keyboard into a character array. This function:

```
askwords(char inputstring[],int maxsize,char message[])
```

displays a dialog box showing the message string that was provided as the third argument, as well as the `inputstring` that was supplied as the first argument. The user may type any string to replace `inputstring`, up to the maximum number of characters specified in `maxsize`.

For example, the program below reads a list of names and displays each name under each athlete.

```
#include "franca.h"
void mainprog()             // c7names.cpp
{
   athlete player[5];
   char name [5][20];
   for (int i=0;i<=4;i++)
     askwords(name[i],20,"Enter a name");

   for (i=0;i<=4;i++)
   {
     player[i].ready(0.);
     player[i].say(name[i]);
   }
}
```

Using String Functions

It is regrettable that you cannot compare strings the same way you compare numeric variables. You also cannot copy a string to another string by using the assignment operator as you do with numeric variables.

For example, if a program has the following declarations:

```
char agent[]="Bond",client[]="Flint";
```

the following statements will not work:

```
if (agent=="Bond") ...
        client=agent;
```

However, you don't have to compare or copy one by one all the elements in a character array to accomplish your task. The header file `string.h` has a few functions that come in handy for this job. It is a standard header file that is available with almost every C++ compiler. Use the `#include` directive shown below to access the string functions:

```
#include <string.h>
```

We will not study in detail all the functions provided in `string.h`, only the functions that you may most likely need. Refer to the compiler's Help menu topics on `string.h` to learn more about the other functions. The following functions will be discussed:

`strcmp`	compares two strings (alphabetically)
`stricmp`	compares two strings (alphabetically), disregarding case
`strcpy`	copies a string to another string
`strcat`	appends a string to another string
`strlen`	computes the length of a string

Comparing Strings with *strcmp*

The following function:

```
int strcmp (char s1[],char s2[]);
```

compares the character arrays `s1` and `s2`. If `s1` comes before `s2` in alphabetical order, the function returns a negative integer. If both strings are equal in alphabetical order, the function returns a zero. If `s1` comes after `s2` in alphabetical order, the function returns a positive integer.

For example:

```
char city[20];
Cin>>city;
if(strcmp(city,"Cupertino")==0)
   Cout<<"You live in a good town!";
```

In the example above, a string (`city`) is read from the keyboard, and then compared with `Cupertino`. If the string exactly matches `Cupertino`, the message *You live in a good town!* is displayed. However, if the typed string is `cupertino`, there will be no match, since its case is different from the previous string's case. For this kind of problem, use the function `stricmp`, as described below.

Comparing Strings while Ignoring Cases with *stricmp*

The following function:

```
int stricmp(char s1[],char s2[]);
```

compares the character arrays `s1` and `s2`. This function works the same as the `strcmp` function, with the exception that case is ignored.

Copying Strings with *strcpy*

The following function:

```
strcpy (char dest[], char source[]);
```

copies the contents of the source string (`source`) into the destination string (`dest`). The original contents of the destination string are lost. It is the programmer's responsibility to make sure that the source string can be accommodated in the space available in the destination string.

For example;

```
char name1[]="Brandon",name2[]="Daisy";
strcpy(name1,name2);
...
```

In the example above, the array `name1` will contain `Daisy` after the function is invoked. If you copy `name1` into `name2`, it could lead to an erroneous result, because the string `name2` is not large enough to accommodate the contents of `name1`.

Concatenating Strings with *strcat*

The following function:

```
strcat (char dest[],char source[]);
```

appends the contents of the `source` string to the end of the destination string (`dest`).

For example:

```
char message[20]="The exit ";
strcpy( message, "is near!");
...
```

causes the character array to become *The exit is near!*

Returning String Length with *strlen*

The following function:

```
int strlen(char string[]);
```

returns the length of the string. The length does not include the `null` terminator character.

For example:

```
char sentence[80]="He that shall persevere to the end ...";
Cout<< strlen(sentence);
...
```

The sequence above causes the number *38* to be displayed, since this is the number of characters contained in the array `sentence`. The string length and the array size are not usually the same! The string length is the number of characters from the beginning position to the `null` terminator. This length is very likely to change during program execution, since you may copy different characters into the same string. On the other hand, the array size is usually set when you declare the array. The string length should never exceed the array size.

Determining Declared Size with *sizeof*

Although the array size may be constant throughout program execution, C++ offers an operator to determine the size of any array, object, or structure that you declare in your program—the `sizeof` operator.

The `sizeof` operator returns an integer representing the size of any variable, object, array, or structure that you declared in your program. It is obvious that since you declared the variable, you should be able to determine the size yourself. Why use this operator?

There may be several reasons to use it, including the fact that computations may sometimes become a little tedious. For example, what is the size of the array declared as follows:

```
char listen[]="He that shall persevere to the end, he shall be saved.";
```

This computation presents absolutely no problem. Just count all the letters, the blank spaces, and the period, then add one for the `null` terminator to compute the size. However, you may not only find this tedious, you may also make a mistake somewhere.

Another, and more important, reason stems from the fact that you often have to modify programs, and an array that was originally declared with 40 elements may, in a later version of your program, be changed to accommodate 50 elements. If you have loops that go from 0 to 39 to handle the array, you will have to inspect the program completely to determine which of these constants needs to be changed. You can easily avoid this hassle if, instead of using the following code:

```
char array[40];
...
for ( int i=0; i<=39;i++)
```

you use

```
char array[40];
...
for (int i=0; i<sizeof(array) ; i++)
```

Converting between Numbers and Characters

Each numeric digit—0, 1, 2, 3, 4, 5, 6, 7, 8, and 9—can be represented as a character, and, therefore, any number stored in the computer can also be written if each of its digits is converted to the character representation. C++ offers functions to assist you in doing this kind of type conversion:

`atoi`	converts from alphanumeric array (character) to integer
`atof`	converts from alphanumeric array (character) to floating point number
`itoa`	converts from integer to alpha (character)

These functions are included in the header file `stdlib.h`, which must be included in your program with the following directive:

```
#include <stdlib.h>
```

Converting from Character Array to Integer with *atoi*

The following function:

```
int atoi( char number[])
```

returns an integer, which is represented by the character array (string) `number`. If the array cannot be converted, or if it contains invalid characters, a zero is returned.

For example:

```
char number[20];
int k;
askwords(number,20,"Enter the value");
k=atoi(number);
Cout<<k;
```

Converting from Character Array to Floating Point Number with *atof*

The following function:

```
float atof(char number[])
```

returns a floating point number represented by the string `number`. If this string cannot be converted, a zero is returned.

Converting from Integer to Character Array with *itoa*

The following function:

```
itoa (int value, char number[], int radix)
```

converts the integer value `value`, and stores the result in the string `number`. `radix` is an integer value that specifies the radix of the numbering system to be used. In most cases, we are interested in decimal representation, so the number *10* can be used.

For example:

```
int k=25;
char number[20];
itoa(k, number, 10);
```

Try These for Fun…

- Write a program to read a first name, a middle name, and a last name. Then, have it write the last name, a comma, and then the first name and the middle initial.
- Write a function to compare positions `i` through `j` of a character array with positions of another character array starting at position `k`. Return 1 if the strings match, and zero otherwise.
- Write a program to read an integer *N*, and then to read *N* names while storing them in an array of character arrays.
- Expand the program above to read an integer *m*, and then to display the *m*th name in the array.

Understanding and Using Structures

A *structure* is a set of data of varying types and/or meanings that are all related to the same item. For example, to describe a customer, you may use the following information:

- Name
- Address
- Zip code
- Phone number
- Account number

Each piece of data, called a *field*, has a different meaning. One field keeps the name of a person, another field keeps the name of a street, another field keeps a phone number, etc. However, these fields are related because they describe a single customer. The address reflects the street address of the customer whose name is the *name* field, the telephone number reflects his or her telephone number, etc.

Since objects also have data that belong to the same object, it may be said that objects also contain a structure. For example, athletes have coordinates, a head, a trunk, a left arm, a right arm, a left leg, a right leg, and other data, as well. Indeed, there is a strong resemblance between structures and classes. In fact, it is also possible to include member functions in a structure, and then to declare objects. The only difference is that there are no private or protected members in a structure. They are all public.

Although it is possible to use structures to work as classes, I strongly suggest that you use structures only in the cases for which you do not need member functions.

A structure is declared in a similar way that a class is declared. Use the keyword `struct`, followed by an identifier that will name the structure, then include the declaration of the data that compose the structure in braces.

```
struct    identifier
{
     data or object declarations ;
 };   // Don't forget the semicolon!
```

For example:

```
struct student
{
   char lastname[20];
   char firstname[20];
   long enrollment_number;
};
```

Just like it is with a class, a structure does not imply the creation of an object or a variable—it simply describes what a student looks like. To use an object of a class, you have to declare an object of that class. To use a variable of a given type, you have to declare a variable of that type. To use a structured variable in the program, you must declare a variable that has that structure. For example:

```
student customer;
```

creates a variable `customer` that has a structure such as that defined in `student`.

You can access fields in the structure by qualifying the field with the structured variable name. For example:

```
customer.enrollment_number=924054;
```

Again, this is the same way that we access data in objects.

Structures and Arrays

A field in a structure may be an array. Actually, the *student* example uses character arrays for `firstname` and `lastname`. A field itself may also be a structure. Consider the following structure:

```
struct address
{
   char street[40];
   char city [20];
   char zipcode[10];
};
```

Once this address structure is declared, we can declare a structure to describe an employee to include a field that happens to be an address—in other words, it’s another structure. For example:

```
struct employee
{
   char lastname[20];
   char firstname[20];
   long salary;
   address home;
}
```

If you wonder how to access the address field, here is how to do it:

```
employee person;
...
strcpy(person.home.street,"221 Baker St.");
strcpy(person.home.city,"London");
...
```

Inheritance in Structures

Another alternative you can use to implement the `employee` structure above is to use inheritance to make the `employee` structure a descendent of the `address` structure. In other words, you can think of an employee as an address that has first and last names, as well as a salary.

```
struct employee: address
{
   char lastname[20];
   char firstname[20];
   long salary;
};
```

Arrays of Structures

It is also possible to have arrays in which each element is a structure. For example:

```
employee  staff[100];
student group[30];
```

The first statement creates an array of 100 elements, in which each element is an employee with all the fields described for this structure. The second statement creates an array of 30 students.

To access the `city` field of a given staff member, you will also have to specify an index, so the computer can understand which staff member you are talking about. For example:

```
staff[10].home.city
```

Copying Structures

You can copy a variable to another variable that has the same structure by using the assignment operator. For example, if you have the following declarations:

```
employee worker1,worker2, staff[100];
```

the following statements are correct:

```
worker1=worker2;
staff[5]=worker1;
worker1.home=staff[1].home;
```

because the compiler can copy variables that have the same structure.

Searching Character Arrays

An interesting searching situation is when you have to locate a given element in a character array. For example, you may have to determine which element in an array has the value *zero*.

This situation can be illustrated with a null-terminated string. Remember that a string is terminated by the character `null`, whose value is zero? How do you suppose the `strlen` function computes the size of a string? Can you do it?

All you have to do is to search all the elements of the string to determine where the `null` character is. In fact, it is easier to do this than it is to search for the largest element, because, from the beginning, you know which element for which you are looking.

Go ahead—compare the value you are looking for with the first value, with the second value, with the third value….

If you have the following string:

```
char name[50];
```

and, at some point in the program, if you want to determine the size of the string without using the `strlen` function, you can do as follows:

```
for(int position=0;position<50;position++)
{
  if (name[position]==0) break;
}
```

After this piece of program executes, the variable `position` should contain the number of characters before `null`. However, if there is no terminator, the answer will be 50.

Since you know how to find a `null`, do you think you can find any other character? What if you want to look for a *t*?

Searching with More Arrays

A more interesting situation of array searching is when you use a structure or multidimensional array. For example, consider the following structure:

```
struct student
{
   int idnumber
   char lastname[30];
}
student myclass[50];
```

In this case, there is an array `myclass` with 50 elements. Each element is a structure consisting of an ID number and a last name.

POSITION	ID#	LAST NAME
00	5002	Rogers
01	6754	Smith
02	6003	Adams
03	6532	Rodriguez
…		
19	6021	Arentz

Suppose that all the elements in the array contain valid data. How could you find the last name of the student whose ID number is 6021?

The problem is essentially the same as the problem of searching in an array for a given value. All you have to do is to search the field `idnumber` for a match. While you do this, keep track of the position in the array where you are working.

```
for (int position=0;position<50;position++)
{
   if(myclass[position].idnumber==96021) break; // Found it!
}
```

If a match is found—for example, in position 19—the last name you want is the field `lastname` in position 19 of the array. In this case, you want the following name:

```
myclass[position].lastname
```

What If the Student Is Not Found?

What if there are no students with the ID number you found in the example above? In this case, the loop will check all the elements in the array, and will not execute a break. You may be able to avert this situation by checking the value of `position`—valid values are only between 0 and 49. A complete loop will leave `position` with the value *50*.

> & It is a good idea to leave searching tasks for specialized functions. You may consider returning the index value where the match was found, or a negative number to inform you that there was no match.

Try These for Fun…

- Write the code for a function to examine a string `inputstring` and to return as a result the position where a character `thischaracter` was found. If the character was not found, the result should be zero. The string and the character are passed as arguments.
- Write a program to read an array of 10 integers and to locate the position of a given value. The value is also supplied from the keyboard. Try to use a function to search the array.

Are You Experienced?

Now you can…

Use character arrays to store text data

Use string-manipulation functions to operate on character arrays

Convert between numeric and character format

Use structures

Search for specific values in an array

SKILL

TWENTY-ONE

DEVELOPING APPLICATIONS—SHORT PROJECTS

- Improving the point-of-sale terminal
- Improving the satellite simulation

To further develop your skills in developing applications, you will now make a few changes to two applications you created earlier—the point-of-sale terminal (originally created in Skill 9) and the satellite simulation (created in Skill 18). Here's how you'll improve the code for each of these programs:

- You will incorporate arrays and text manipulation into your point-of-sale terminal.
- You will incorporate arrays into the satellite project to handle a collection of satellites.

Short Project 1—Point-of-Sale Terminal

When you go to the supermarket or to any other store, you hardly ever see a cash register that makes the clerk type in the price anymore. It is most likely that the product code is input using a bar code reader. How does this kind of terminal work?

The computer has a list of all the products that are on sale. In this list, there must be a code, a price, and, possibly, a description of the product. As the clerk inputs the code, the computer searches the list for a product with that particular code. Does this remind you of the searching operation? Well, it should!

In this version of the point-of-sale terminal, a list of items is kept in the computer's memory. Each sale is transacted by entering a product code (or part number) from the keyboard (regrettably, there is no bar code reader available to you). The list is then searched for this particular item, and the price and the product description will be obtained. This new version of the terminal also displays the product description on the screen.

The end-user interface should remain as similar to the previous terminal's interface as possible.

Implementing New Features

The list will be kept in the computer's memory. The following information must be known for each product:

- The product code
- The product description
- The product price

Since several products will be on sale, you may consider declaring an array of structures:

```
struct product
{
  int code;
  char description[20];
  float price;
};
product list[20];
```

Use this array to locate the information for the items that you need. Since the computer's memory cannot hold information while the power is off, you will need to input information every time your program starts. You can think of this array as a *parts catalog* that is checked for every purchase. In the old days, when you wanted to buy parts for your car, the clerk had to look for the part number in a big catalog to find out the price and other information.

A catalog? Should we consider having the program also use a catalog to look for the product information? Actually, this is a great idea! You can design an additional class of objects that acts like a catalog, allowing you to look up information as long as you know the product code. If a catalog is available, the remaining operation is very similar to the old terminal's operation. In fact, why build another class? We can inherit several things from the old one!

The *catalog* Class

The `catalog` class can implement all the operations needed to handle the parts catalog. Although this particular implementation solves only the problem we have at hand, you may find out later that it is a very useful piece of code. Here is one possible declaration:

```
struct product
{
  int code;
  char description[20];
  float price;
};

class catalog
{
 protected:
  product list[20];
  int listsize;
 public:
  catalog();
  virtual product find(int part_number);
};
```

This class has an array and an integer as data members to hold the number of products present in the array. The array itself consists of elements that are structures, as previously described. Since this is not a commercial product, the array uses only 20 elements.

As far as member functions are concerned, there is a constructor and a `find` function. The constructor automatically asks you to input the array elements, so you don't use an empty list. The `find` function searches the catalog for the given part number. Notice that this function returns as a result a structure of type `product`. You enter a part number, and the function returns a complete structure with all the information (code, description, price) that is associated with the given part number.

& The `find` function may need to be replaced by other `find` functions in derived classes. It should be a virtual function.

Here is the code for the constructor:

```
catalog::catalog()
{

  listsize=0;
  for (listsize=0;yesno("Another item?")&&(listsize<20)
                 ;listsize++)
  {
     list[listsize].code=ask("Enter product code:");
     strcpy(list[listsize].description,
           "Product description");
     askwords(list[listsize].description,20,
           "Description:");
     list[listsize].price=ask("Enter the price:");
  }
}
```

The `find` function is very similar to what you have already seen while searching an array:

```
product catalog::find(int part_number)
{
  for(int item=0;item<listsize;item++)
  {
     if(part_number==list[item].code) return list[item];
  }
  product nonexistent;
  nonexistent.code=0;
  return nonexistent;
}
```

If there is no match for the requested code, even in this case, a structure is returned, but the field `code` is zero. This is how you can find out whether the item was found or not. On the other hand, if the match was found in position `item`, the returned structure is the array element indexed by `item`.

& The code for the `catalog` class is included in `c7catalo.h`.

The New *terminal* Class

The new `terminal` class is an expansion of the old one. You can inherit what was available, and just add the expansions. The new class `saleterm` can be declared as follows:

```
class saleterm: public terminal
{
  protected:
   catalog parts;
   product sale;
   Box Part_Number,Part_Description;
   void items();
  public:
   saleterm();
};
```

By deriving this new class from the `terminal` class, you inherit everything that exists in `terminal`. Only the new items have to be declared. The new terminal consists of a catalog, a structure to describe one product (the product currently being sold), a couple of new boxes to display the product code (`Part_Number`), and the product description.

The `items` function is inherited, too. However, since this terminal requests codes instead of prices, you cannot use the same function. You have to provide a new function to handle the items.

You *can* keep the `operate` member function intact and use it as inherited, because this function only catches up after the price has been obtained.

Constructors

An interesting thing happens with the constructors when an object of a derived class is created. (Pay attention—this is new!) The constructor for an object of the base class is invoked, and then the constructor for the derived class is invoked. When the constructor for the derived class is invoked, an object of the base class already exists. This new constructor only has to add the features that are particular to the derived class to finish the construction.

In this example, the constructor only needs to position and to label two additional boxes.

Here is the code for the constructor:

```
saleterm::saleterm()
{
  Part_Number.place(450,300);
  Part_Description.place(450,340);
  Part_Number.label("Part Number:");
  Part_Description.label("Description");
}
```

Here is the code for the new `items` function:

```
void saleterm::items()
{
  int somecode;
  for(;;)
  {
    do
    {
      somecode=ask("Enter part number:");
      sale=parts.find(somecode);
      if(sale.code==0)
          yesno("Wrong part number, please check");
    }
    while (sale.code==0);
    Part_Number.say(sale.code);
    Part_Description.say(sale.description);
    saletotal=saletotal+sale.price;
    Price.say(sale.price);
    Cur_Total.say(saletotal);
    if(!yesno("Another item?")) break;
  }
    saletotal=saletotal*(1+tax);
    Saletotal.say(saletotal);
 }
```

The complete code for this project can be found in `c7term.h` and `c7term.cpp`, shown below.

```
#ifndef C7TERM_H        // c7term.h
#define C7TERM_H
#include "franca.h"
#include "c6term.h"
#include "c7catalo.h"
#include <string.h>
class saleterm: public terminal
{
  protected:
   catalog parts;
   product sale;
   Box Part_Number,Part_Description;
   void items();
  public:
   saleterm();
};
```

```
saleterm::saleterm()
{
  Part_Number.place(450,300);
  Part_Description.place(450,340);
  Part_Number.label("Part Number:");
  Part_Description.label("Description");
}
void saleterm::items()
{
  int somecode;
  for(;;)
  {
    do
    {
      somecode=ask("Enter part number:");
      sale=parts.find(somecode);
      if(sale.code==0) yesno("Wrong part number, please check");
    }
    while (sale.code==0);
    Part_Number.say(sale.code);
    Part_Description.say(sale.description);
    saletotal=saletotal+sale.price;
    Price.say(sale.price);
    Cur_Total.say(saletotal);
    if(!yesno("Another item?")) break;
  }
    saletotal=saletotal*(1+tax);
    Saletotal.say(saletotal);
 }
#endif

// End

#include "franca.h"   // c7term.cpp
#include "c7term.h"
#include <string.h>
void mainprog()
{
   saleterm cashregister;
   cashregister.operate();
}
```

IMPROVEMENTS

Can you save the product information so you don't have to retype it every time your program starts? It will definitely be difficult to sell your terminal otherwise, don't you think?

You can store this information in a disk file, which we will learn about in the next Skill.

Short Project 2—Satellites

Arrays make it easy for you to manipulate a collection of objects. If you want to simulate electrons orbiting around a nucleus, as suggested in Skill 18, without using arrays, you will end up with a large, repetitive program that handles each electron individually.

Because arrays allow you to deal with all elements using the same name (for example, `electron`) and designate each element by a variable index (for example, `electron [n]`), you have to write only one loop to explain what you want done with any element. Not only will you save work, but the program does not have to change if you ever need to change the number of elements in the array.

As an example, we will use the `satellite` class to simulate an atom with two electrons in the first orbit and eight electrons in the second orbit. An array `electron`, consisting of 10 objects of type `satellite`, will be used. Of course, the nucleus can also be a satellite.

Enhancing Capabilities with Arrays

To make things more interesting, you can make the nucleus perform a circular movement, as well. In the example below (`c7atom.cpp`) , a special satellite `ether` is used as the center for the nucleus's orbit.

The declaration and the initialization are as follows:

```
const float pi2=2*3.14159;
Box clock("Time: ");
Clock timer,sidereal;
satellite electron[10];
satellite nucleus,ether;
nucleus.resize(40);
ether.place(320,200);
nucleus.center(ether);
nucleus.dist(80);
nucleus.speed(pi2/800.);
nucleus.move();
```

Initializing Electrons

Each electron has to be initialized, as well. Electrons in the first orbit are initialized as follows:

```
for (int i=0;i<2;i++)
{
  electron[i].center(nucleus);
  electron[i].dist(80);
  electron[i].speed(pi2/300.);
  electron[i].resize(12);
  electron[i].color(5,5);
  electron[i].angle(i*pi2/2.+pi2/4.);
  electron[i].move();
}
```

Electrons in the second orbit are initialized as follows:

```
 for (i=2;i<10;i++)
{
  electron[i].center(nucleus);
  electron[i].dist(120);
  electron[i].speed(pi2/600.);
  electron[i].resize(12);
  electron[i].color(5,5);
  electron[i].angle(i*pi2/8.);
  electron[i].move();
}
```

When all the objects have been initialized, the actual simulation can take place:

```
nucleus.show();
for(;sidereal.time()<20.;)
{
  nucleus.erase();
  nucleus.move();
  nucleus.show();
  for( int i=0;i<10;i++)
  {
    electron[i].erase();
    electron[i].center(nucleus);
    electron[i].move();
    electron[i].show();
  }
  clock.say(sidereal.time()*10);
  timer.watch(.033);
  timer.reset();
}
```

In this case, the nucleus was represented as a separate satellite, but it is not necessary to represent it this way. You can choose `electron[10]` to represent it if you add one more electron to your array.

Try This for Fun…

- The program `c7pool.cpp` (listed below) implements and uses a class `pool`, which

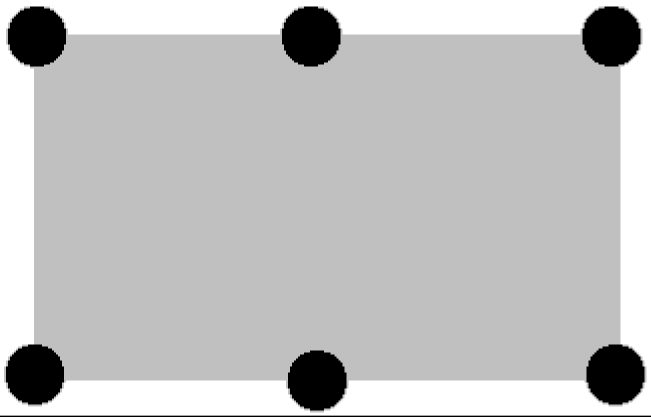


Fig 21.1 The pool table

simulates a ball rolling on a pool table and bouncing against the walls.

  - Derive a class that simulates a pool table by including six black circles, as shown in Figure 21.1. When the ball drops into a pocket (reaches a black circle), you score a point. To determine whether the ball has dropped into a pocket, check whether the ball's coordinates (*x*,*y*) are at a distance that is less than the pocket's radius. In this program, the pocket's radius is 15 and the ball's radius is 10.
  - Here is the listing of `c7pool.cpp`:

```
#include "franca.h"
#include <math.h>
#include <stdlib.h>
  const int ballradius=10,poolradius=15;
  const int poolx=400,pooly=200;
class pool:public Stage
{
 protected:
  Square table;
  Circle ball;
  float xb,yb;     // The ball's coordinates
  float incx,incy; // Incremental movements
  float speed;
 public:
  pool();
  void shoot(float angle);
};

pool::pool()
{
  float angle;
  speed=1.5;
  table.resize(pooly,poolx);
  table.place(320,240);
  table.color(2);
  ball.color(0,0);
  ball.resize(ballradius*2);
  table.show();
  insert(table);
  insert(ball);
  xb=rand()%(poolx-ballradius)+320-poolx/2.+ballradius;
  yb=rand()%(pooly-ballradius)+240-pooly/2.+ballradius;
  ball.place(xb,yb);
  ball.show();
}

void pool::shoot(float angle)
{
  angle=3.14159*angle/180.;
  incx=cos(angle)*speed;
  incy=-sin(angle)*speed;
  Clock cuckoo,timer;
  for (;cuckoo.time()<30;)
  {
    if((xb<=(320-poolx/2.+ballradius))||
       (xb>=(320+poolx/2.-ballradius))) incx=-incx;
    if((yb<=(240-pooly/2.+ballradius))||
       (yb>=(240.+pooly/2.-ballradius))) incy=-incy;
    xb=xb+incx;
    yb=yb+incy;
    ball.place(xb,yb);
    table.show();
    ball.show();
    timer.watch(.033);
    timer.reset();
  }
}
```

```
void mainprog()
{
  pool billiard;
  float angle;
  for(;yesno("Take a shot?");)
  {
    angle=ask("input the angle:");
    billiard.shoot(angle);
  }
}
```

Are You Experienced?

Now you can…

Use character arrays in a point-of-sale or similar application to display product information

Use an array of Screen Objects, such as satellites, to manipulate several objects

SKILL

TWENTY-ONE

DEVELOPING APPLICATIONS—SHORT PROJECTS

- Improving the point-of-sale terminal
- Improving the satellite simulation

To further develop your skills in developing applications, you will now make a few changes to two applications you created earlier—the point-of-sale terminal (originally created in Skill 9) and the satellite simulation (created in Skill 18). Here's how you'll improve the code for each of these programs:

- You will incorporate arrays and text manipulation into your point-of-sale terminal.
- You will incorporate arrays into the satellite project to handle a collection of satellites.

Short Project 1—Point-of-Sale Terminal

When you go to the supermarket or to any other store, you hardly ever see a cash register that makes the clerk type in the price anymore. It is most likely that the product code is input using a bar code reader. How does this kind of terminal work?

The computer has a list of all the products that are on sale. In this list, there must be a code, a price, and, possibly, a description of the product. As the clerk inputs the code, the computer searches the list for a product with that particular code. Does this remind you of the searching operation? Well, it should!

In this version of the point-of-sale terminal, a list of items is kept in the computer's memory. Each sale is transacted by entering a product code (or part number) from the keyboard (regrettably, there is no bar code reader available to you). The list is then searched for this particular item, and the price and the product description will be obtained. This new version of the terminal also displays the product description on the screen.

The end-user interface should remain as similar to the previous terminal's interface as possible.

Implementing New Features

The list will be kept in the computer's memory. The following information must be known for each product:

- The product code
- The product description
- The product price

Since several products will be on sale, you may consider declaring an array of structures:

```
struct product
{
  int code;
  char description[20];
  float price;
};
product list[20];
```

Use this array to locate the information for the items that you need. Since the computer's memory cannot hold information while the power is off, you will need to input information every time your program starts. You can think of this array as a *parts catalog* that is checked for every purchase. In the old days, when you wanted to buy parts for your car, the clerk had to look for the part number in a big catalog to find out the price and other information.

A catalog? Should we consider having the program also use a catalog to look for the product information? Actually, this is a great idea! You can design an additional class of objects that acts like a catalog, allowing you to look up information as long as you know the product code. If a catalog is available, the remaining operation is very similar to the old terminal's operation. In fact, why build another class? We can inherit several things from the old one!

The *catalog* Class

The `catalog` class can implement all the operations needed to handle the parts catalog. Although this particular implementation solves only the problem we have at hand, you may find out later that it is a very useful piece of code. Here is one possible declaration:

```
struct product
{
  int code;
  char description[20];
  float price;
};

class catalog
{
 protected:
  product list[20];
  int listsize;
 public:
  catalog();
  virtual product find(int part_number);
};
```

This class has an array and an integer as data members to hold the number of products present in the array. The array itself consists of elements that are structures, as previously described. Since this is not a commercial product, the array uses only 20 elements.

As far as member functions are concerned, there is a constructor and a `find` function. The constructor automatically asks you to input the array elements, so you don't use an empty list. The `find` function searches the catalog for the given part number. Notice that this function returns as a result a structure of type `product`. You enter a part number, and the function returns a complete structure with all the information (code, description, price) that is associated with the given part number.

& The `find` function may need to be replaced by other `find` functions in derived classes. It should be a virtual function.

Here is the code for the constructor:

```
catalog::catalog()
{

  listsize=0;
  for (listsize=0;yesno("Another item?")&&(listsize<20)
                ;listsize++)
  {
     list[listsize].code=ask("Enter product code:");
     strcpy(list[listsize].description,
           "Product description");
     askwords(list[listsize].description,20,
           "Description:");
     list[listsize].price=ask("Enter the price:");
  }
}
```

The `find` function is very similar to what you have already seen while searching an array:

```
product catalog::find(int part_number)
{
  for(int item=0;item<listsize;item++)
  {
     if(part_number==list[item].code) return list[item];
  }
  product nonexistent;
  nonexistent.code=0;
  return nonexistent;
}
```

If there is no match for the requested code, even in this case, a structure is returned, but the field `code` is zero. This is how you can find out whether the item was found or not. On the other hand, if the match was found in position `item`, the returned structure is the array element indexed by `item`.

& The code for the `catalog` class is included in `c7catalo.h`.

The New *terminal* Class

The new `terminal` class is an expansion of the old one. You can inherit what was available, and just add the expansions. The new class `saleterm` can be declared as follows:

```
class saleterm: public terminal
{
  protected:
   catalog parts;
   product sale;
   Box Part_Number,Part_Description;
   void items();
  public:
   saleterm();
};
```

By deriving this new class from the `terminal` class, you inherit everything that exists in `terminal`. Only the new items have to be declared. The new terminal consists of a catalog, a structure to describe one product (the product currently being sold), a couple of new boxes to display the product code (`Part_Number`), and the product description.

The `items` function is inherited, too. However, since this terminal requests codes instead of prices, you cannot use the same function. You have to provide a new function to handle the items.

You *can* keep the `operate` member function intact and use it as inherited, because this function only catches up after the price has been obtained.

Constructors

An interesting thing happens with the constructors when an object of a derived class is created. (Pay attention—this is new!) The constructor for an object of the base class is invoked, and then the constructor for the derived class is invoked. When the constructor for the derived class is invoked, an object of the base class already exists. This new constructor only has to add the features that are particular to the derived class to finish the construction.

In this example, the constructor only needs to position and to label two additional boxes.

Here is the code for the constructor:

```
saleterm::saleterm()
{
  Part_Number.place(450,300);
  Part_Description.place(450,340);
  Part_Number.label("Part Number:");
  Part_Description.label("Description");
}
```

Here is the code for the new `items` function:

```
void saleterm::items()
{
  int somecode;
  for(;;)
  {
    do
    {
      somecode=ask("Enter part number:");
      sale=parts.find(somecode);
      if(sale.code==0)
          yesno("Wrong part number, please check");
    }
    while (sale.code==0);
    Part_Number.say(sale.code);
    Part_Description.say(sale.description);
    saletotal=saletotal+sale.price;
    Price.say(sale.price);
    Cur_Total.say(saletotal);
    if(!yesno("Another item?")) break;
  }
    saletotal=saletotal*(1+tax);
    Saletotal.say(saletotal);
 }
```

The complete code for this project can be found in `c7term.h` and `c7term.cpp`, shown below.

```
#ifndef C7TERM_H        // c7term.h
#define C7TERM_H
#include "franca.h"
#include "c6term.h"
#include "c7catalo.h"
#include <string.h>
class saleterm: public terminal
{
  protected:
   catalog parts;
   product sale;
   Box Part_Number,Part_Description;
   void items();
  public:
   saleterm();
};
```

```
saleterm::saleterm()
{
  Part_Number.place(450,300);
  Part_Description.place(450,340);
  Part_Number.label("Part Number:");
  Part_Description.label("Description");
}
void saleterm::items()
{
  int somecode;
  for(;;)
  {
    do
    {
      somecode=ask("Enter part number:");
      sale=parts.find(somecode);
      if(sale.code==0) yesno("Wrong part number, please check");
    }
    while (sale.code==0);
    Part_Number.say(sale.code);
    Part_Description.say(sale.description);
    saletotal=saletotal+sale.price;
    Price.say(sale.price);
    Cur_Total.say(saletotal);
    if(!yesno("Another item?")) break;
  }
    saletotal=saletotal*(1+tax);
    Saletotal.say(saletotal);
 }
#endif

// End

#include "franca.h"   // c7term.cpp
#include "c7term.h"
#include <string.h>
void mainprog()
{
   saleterm cashregister;
   cashregister.operate();
}
```

IMPROVEMENTS

Can you save the product information so you don't have to retype it every time your program starts? It will definitely be difficult to sell your terminal otherwise, don't you think?

You can store this information in a disk file, which we will learn about in the next Skill.

Short Project 2—Satellites

Arrays make it easy for you to manipulate a collection of objects. If you want to simulate electrons orbiting around a nucleus, as suggested in Skill 18, without using arrays, you will end up with a large, repetitive program that handles each electron individually.

Because arrays allow you to deal with all elements using the same name (for example, `electron`) and designate each element by a variable index (for example, `electron [n]`), you have to write only one loop to explain what you want done with any element. Not only will you save work, but the program does not have to change if you ever need to change the number of elements in the array.

As an example, we will use the `satellite` class to simulate an atom with two electrons in the first orbit and eight electrons in the second orbit. An array `electron`, consisting of 10 objects of type `satellite`, will be used. Of course, the nucleus can also be a satellite.

Enhancing Capabilities with Arrays

To make things more interesting, you can make the nucleus perform a circular movement, as well. In the example below (`c7atom.cpp`) , a special satellite `ether` is used as the center for the nucleus's orbit.

The declaration and the initialization are as follows:

```
const float pi2=2*3.14159;
Box clock("Time: ");
Clock timer,sidereal;
satellite electron[10];
satellite nucleus,ether;
nucleus.resize(40);
ether.place(320,200);
nucleus.center(ether);
nucleus.dist(80);
nucleus.speed(pi2/800.);
nucleus.move();
```

Initializing Electrons

Each electron has to be initialized, as well. Electrons in the first orbit are initialized as follows:

```
for (int i=0;i<2;i++)
{
  electron[i].center(nucleus);
  electron[i].dist(80);
  electron[i].speed(pi2/300.);
  electron[i].resize(12);
  electron[i].color(5,5);
  electron[i].angle(i*pi2/2.+pi2/4.);
  electron[i].move();
}
```

Electrons in the second orbit are initialized as follows:

```
 for (i=2;i<10;i++)
{
  electron[i].center(nucleus);
  electron[i].dist(120);
  electron[i].speed(pi2/600.);
  electron[i].resize(12);
  electron[i].color(5,5);
  electron[i].angle(i*pi2/8.);
  electron[i].move();
}
```

When all the objects have been initialized, the actual simulation can take place:

```
nucleus.show();
for(;sidereal.time()<20.;)
{
  nucleus.erase();
  nucleus.move();
  nucleus.show();
  for( int i=0;i<10;i++)
  {
    electron[i].erase();
    electron[i].center(nucleus);
    electron[i].move();
    electron[i].show();
  }
  clock.say(sidereal.time()*10);
  timer.watch(.033);
  timer.reset();
}
```

In this case, the nucleus was represented as a separate satellite, but it is not necessary to represent it this way. You can choose `electron[10]` to represent it if you add one more electron to your array.

Try This for Fun…

- The program `c7pool.cpp` (listed below) implements and uses a class `pool`, which

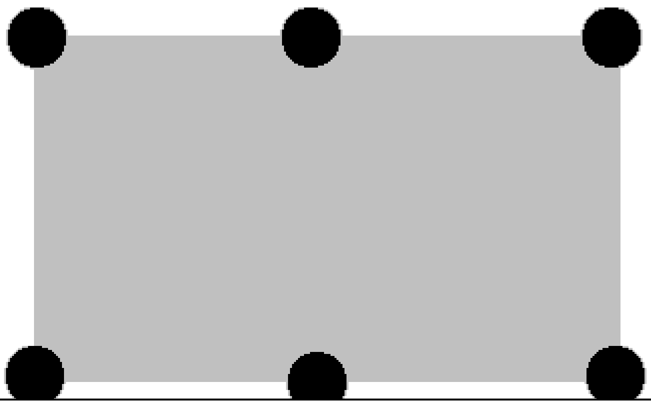


Fig 21.1 The pool table

simulates a ball rolling on a pool table and bouncing against the walls.

  - Derive a class that simulates a pool table by including six black circles, as shown in Figure 21.1. When the ball drops into a pocket (reaches a black circle), you score a point. To determine whether the ball has dropped into a pocket, check whether the ball's coordinates (*x*,*y*) are at a distance that is less than the pocket's radius. In this program, the pocket's radius is 15 and the ball's radius is 10.
  - Here is the listing of `c7pool.cpp`:

```
#include "franca.h"
#include <math.h>
#include <stdlib.h>
  const int ballradius=10,poolradius=15;
  const int poolx=400,pooly=200;
class pool:public Stage
{
 protected:
  Square table;
  Circle ball;
  float xb,yb;     // The ball's coordinates
  float incx,incy; // Incremental movements
  float speed;
 public:
  pool();
  void shoot(float angle);
};

pool::pool()
{
  float angle;
  speed=1.5;
  table.resize(pooly,poolx);
  table.place(320,240);
  table.color(2);
  ball.color(0,0);
  ball.resize(ballradius*2);
  table.show();
  insert(table);
  insert(ball);
  xb=rand()%(poolx-ballradius)+320-poolx/2.+ballradius;
  yb=rand()%(pooly-ballradius)+240-pooly/2.+ballradius;
  ball.place(xb,yb);
  ball.show();
}

void pool::shoot(float angle)
{
  angle=3.14159*angle/180.;
  incx=cos(angle)*speed;
  incy=-sin(angle)*speed;
  Clock cuckoo,timer;
  for (;cuckoo.time()<30;)
  {
    if((xb<=(320-poolx/2.+ballradius))||
       (xb>=(320+poolx/2.-ballradius))) incx=-incx;
    if((yb<=(240-pooly/2.+ballradius))||
       (yb>=(240.+pooly/2.-ballradius))) incy=-incy;
    xb=xb+incx;
    yb=yb+incy;
    ball.place(xb,yb);
    table.show();
    ball.show();
    timer.watch(.033);
    timer.reset();
  }
}
```

```
void mainprog()
{
  pool billiard;
  float angle;
  for(;yesno("Take a shot?");)
  {
    angle=ask("input the angle:");
    billiard.shoot(angle);
  }
}
```

Are You Experienced?

Now you can…

Use character arrays in a point-of-sale or similar application to display product information

Use an array of Screen Objects, such as satellites, to manipulate several objects