PART VIII

GETTING REAL

In real-life programming, you will not be able to use the Screen Objects that were available with our special software. You will have to deal with standard text input and output (i/o) using the keyboard and the display. In Part VIII, you will develop your ability to perform i/o operations using the standard C++ i/o streams. You will also learn how to format input and output.

Also, in real-life programming, you don't want to have to type data all the time. It is most likely that you will have your data stored in a disk file, so you can read it and update it at will. In Part VIII, you will develop your ability to use files.

Finally, to keep improving your skills in developing applications, you will work on further improvements to the point-of-sale terminal. The most remarkable improvement this time is that you will store your product catalog in a disk file.

SKILL

TWENTY-TWO

LIVING WITHOUT *FRANCA.H*

- Inputting with streams
- Outputting with streams
- Formatting

In the real world, there are no `franca.h` header files or `ScreenObj`, `athlete`, `Clock`, or most of the other object types we have used to learn C++. Now that you are a programmer, you must learn to live on your own, without the help of the class libraries that were developed to help you take your initial programming steps.

It is regrettable that it is still too hard for a beginning programmer to use a graphic interface in Windows. In this Skill, we will resort to the more modest text interface to enable you to face the real world without fear. In fact, the main difference between what we have been using so far and what is available in the real world is the way that you interact with your computer.

Our predefined classes made it easy for you to develop programs that could produce pictures and animations on the screen. Our communication with the user was a bit more attractive due to the graphic interface. When you use everyday C++, you will be restricted to writing data to the screen and to reading data from the keyboard.

In this Skill, we will concentrate on reading textual data from the keyboard, on displaying textual data on the screen, and on using files—with no help from prebuilt libraries.

Of course, you are always welcome to use the classes declared in `franca.h`. You may also continue to study C++ and, more specifically, Windows programming, so you may learn how to communicate with users of your programs through a graphic interface.

LIVING WITHOUT I/O STATEMENTS

It is amazing that C++ does not have special statements to deal with inputting and outputting data. Essentially, data to be output are sent to special objects in charge of output, and data to be input are retrieved from other special objects.

The Real World of C++ Programming

The best way to learn what "real" C++ programming is like is to study examples. The following example is a very simple program that runs in C++ using only standard header files. This program asks you for your first name, and writes a message back to you. When you execute this program, notice the first difference between what you have done so far and what you can do now—there are no more projects!

You do not have to use projects anymore. Instead of opening a project, removing the previous `.cpp` file, and including the new one, all you have to do is to open your program file and then run it.

You can type or load a program, and then just run it!

```
#include <iostream.h>
#include <iomanip.h>
void main()             // c8cngrat.cpp
{
  char yourname[30];
  cout<<"Hello!"<<endl<<"What is your name?";
  cin>yourname;
  cout<<endl;
  cout<<"Congratulations, "<<yourname;
  cout<<", you are now a programmer!";
}
```

The code snippet above shows clearly a few differences from the exercises you've been working with throughout this book:

- There is no `franca.h`, so there are no athletes, runners, Screen Objects, Clocks, etc. There are also no functions such as `ask`, `yesno`, etc.
- It is most likely that you will use the header files `iostream.h` and `iomanip.h`. You may also have to use other header files.

Do not use `franca.h` with `iostream.h`—it may cause unpredictable problems!

- There is no `void mainprog`. Instead, your main function is called `main`.
- You don't have to use a project—just type or load your program, and then run it.

Make sure there are no projects open. If there is a project open, close it.

- As you run the program, no graphic interfaces will be available. (Well, you are not a kid anymore….)

Execute this program and see how it works. Experiment with typing your full name instead of your first name only. What happens?

LIVING WITHOUT PROJECTS

The following hints may help you run your first few programs that are not part of a project:

- Make sure there are no projects or workspaces open. If there are, close them.
- In the main menu, choose File ➢ Open to open the program that you want.
- Run the program using the same procedure you used to run a project.
- Microsoft compilers will create a standard project to run your program. Make sure to close the project or workspace after you run your program.

Using C++ Streams

Two alternatives govern input and output in C++. It is possible to use a set of functions to perform these operations, or to use a set of objects to perform them. In fact, the use of functions was created with the C programming language, the ancestor of C++.

The use of objects to perform input and output was specifically developed for C++, and is simpler and more powerful. Your program generates data that flow to an output. This is the concept of C++ output streams. You take all the data that you want to output and "move" it to a special object (`cout`) that forwards your data to the screen.

Actually, it is very simple—if you have an integer variable `number`, you can write this variable to the screen with the following statement:

```
 cout<< number;
```

which means *put the variable `number` in the C output.*

Similarly, you request another kind of object to provide you with data collected from the keyboard. You "retrieve" the data from this input object (`cin`) and bring them to variables in your program. Using the same example, you can input a value to the variable `number` with the following statement:

```
 cin>> number;
```

which means *extract data from the C input and bring it into the variable `number`.*

The names *cin* and *cout* are derived from *C input* and *C output.*

As you may notice, the input and output operations use special operators, << and >>. These operators remind you that the data flow from data to output (`cout<<number`) and from input to data (`cin>>number`).

This syntax is very similar to what you used with `Cin` and `Cout` in `franca.h`.

INPUT AND OUTPUT

Novice programmers are often confused by the terms *input* and *output.* Keep in mind that these terms refer to the computer. What you type from the keyboard is *input to* the computer; values that you display on the screen are *output from* the computer.

The *iostream.h* Header File

To use stream input and output (stream i/o), you must include a new header file in your programs. Do not use `franca.h` with stream i/o. Instead, type the following line at the beginning of your program file:

```
#include <iostream.h>
```

Now, the header file is enclosed by < and >, instead of by double quotes (“”). When you use header files that are in the compiler directories, you should use this new form.

The `iostream.h` file contains the class declarations and definitions that allow you to use the objects `cin` and `cout`.

Stream i/o handles all the fundamental data types of C++. The following data types may be directly used:

- `int`
- `char`
- `float`
- `double`

You cannot use arrays. You can only input and output an element of an array that happens to be of a fundamental data type. You also cannot use a structure. You can only input and output a field in a structure that happens to be of a fundamental data type.

However, there is a notable exception: null-terminated strings, although a special case of arrays can be used, too! For example:

```
char name[] = "Alfred E. Newman";
...
cout<<name;
```

Once you have included `iostream.h` in your program, the objects `cin` and `cout` are automatically declared and usable.

Outputting with Streams

Stream output consists of the object `cout`, followed by the operator `<<` and the variable (or constant) to be output:

```
cout <<    variable identifier  ;
```

For example:

```
cout<< number;
```

If several variables are to be output, each one has to be preceded by another operator (`<<`):

```
cout<<"The result is:"<<number<<" and that is final";
```

The output is displayed on the screen with no spaces between the values.

For example:

```
int number=32;
char name[]="Sonny Bonds";
cout<<name<<number;
```

results in the following output:

```
Sonny Bonds32
```

Two output statements do not cause two lines. In other words, in the example above, if the output was split into two statements:

```
cout<<name;
cout<<number;
```

the result would be the same.

Spacing

If you desire a space between values, an easy solution is to insert a blank string between them:

```
cout<<name<<"  "<<number;
```

Or, a more detailed explanation could also be used:

```
cout<<"Customer Name:  " << name << "  code is: "<<number;
```

Starting on a New Line

It is also possible to instruct the output to start on a new line. There are two ways to do this: you can write the control character `\n` (`newline`) to the output, or you can include the i/o manipulator `endl` in the output. For example:

```
cout<<"Customer Name:"<<name<<'\n'<<" code is:   "<<number;
```

or:

```
cout<<"Customer Name:<<name<<endl<<" code is:   "<<number;
```

Either one will cause the same output:

```
Customer Name:Sonny Bonds
 code is:   32
```

Inputting with Streams

Stream input consists of the object `cin`, followed by the operator `>>` and the variable to which the input value should be assigned:

```
cin >>     variable identifier    ;
```

For example:

```
cin>>i;
```

fetches a value from the keyboard and assigns it to the variable `i`. It is possible to assign values to more than one variable with a single input:

```
cin>>i>>j>>k;
```

In this case, the first value will be assigned to `i`, the second value to `j`, and the third value to `k`. To make the computer separate one input from another input, use a blank space or hit Enter after each value when you type the values. If you choose to separate them with blank spaces, you still have to hit Enter after the last value.

You can experiment with the following program, which requests three integer values to be input and then displays them:

```
#include <iostream.h>
#include <iomanip.h>
void main()
{
  int i,j,k;
  cout<<"Enter three values:\n";
  cin>>i>>j>>k;
  cout<<endl<<"Your values are:"<<'\n';
  cout<<i<<" "<<j<<" "<<k;
}
```

Run this program to make sure that the numbers that you input are correctly displayed.

You can move to a new line on the screen either by including the i/o manipulator `endl` in the stream, or by including the control character `newline`, which is represented in C++ by a backslash followed by the character `n`. Wherever this control character is found in a literal string or in the output stream, it is interpreted as an instruction to move to the next line. A `newline` can be inserted in the output stream either by including it in another string (`Enter three values:\n`), or by including it by itself (as in the second `cout` in the program above). Whether you use single quotes or double quotes to enclose the character, they will achieve the same effect. However, double quotes generate a null-terminated string, instead of a single character.

When you run the program, observe the following points:

- You can type one or more blank spaces between each number.
- You can press Enter between each number.
- You cannot use blank spaces in a number.
- You cannot use other characters (such as a comma) between numbers.

Example—Using Streams to Input and Output Arrays

Suppose that we want to work with some numeric arrays. It is very likely that we will want to list the whole array to determine the values. We can use a function `show` to do this:

```
void show (int array[], int from, int to)
{
   cout<<endl<<"Contents of array:"<<endl<<"Index"
       <<"  Value";
   for (int k=from;k<=to;k++)
     cout<<endl<<k<<"      "<<array[k];
}
```

This function takes three parameters: the array itself and the index positions that denote where to start and stop the listing. Remember that you do not have to specify the size of the array.

Similarly, we can also use a `readarray` function to input the values:

```
void readarray(int array[],int from,int to)
{
    for (int i=from;i<=to;i++)
    {
     cout<<endl<<"Please input element index "<<i<<':';
     cin>>array[i];
    }
}
```

This function illustrates the good practice of keeping the user informed of what the computer is expecting them to type. Remember that it is very likely that you will make programs for other people to use. The more they know what the computer wants them to do, the fewer mistakes they will make. On the other hand, be careful to avoid including unnecessary messages in your functions. If your software is going to be embedded in other pieces of software, it may jam the user screen with useless information.

You can test the functions above with the program `c8show.cpp`:

```
void main()         // c8show.cpp
{
   int number[10];
   readarray(number,0,9);
   show(number,0,9);
}
```

Formatting

A large amount of a professional programmer's time is spent getting data and producing reports. Professional reports have to look good. Names and numbers must be displayed in convenient places in the report, and numbers must be well aligned.

This is what we call *formatting*. Both input data and output data must be well formatted, so people can readily understand the information.

Elementary formatting was achieved by moving to a new line and by inserting blanks, but this is not enough. You still need to know a few more items to conveniently format your reports.

Assuring Floating Point Precision

If you print floating point values, you may not like the way they appear. For example:

```
#include<iomanip.h>
void main()
{
 float price,tax=6.75,total;
 cout<<"Enter the price:";
 cin>>price;
 total=price+tax*price/100;
 cout<<endl<<"Please pay: $"<<total;
}
```

Although this example is a very simple program to compute the price after sales tax, you may not be very happy with the results. If you enter the price as 10, you will see the following result:

```
Please pay: $10.675
```

There is no such thing as a half cent—you really expected the result to come out with only two decimal places. How can you ensure this? We will learn how to implement floating point precision a little later in this Skill.

Aligning Fields

Another important issue is to make sure that all fields are aligned. Consider the program that reads and shows an array. If you execute it using numbers with varying lengths, you may end up with the following screen:

```
Contents of the array:
Index     Value
0         1
1         23456
2         23
3         -4567
4         3
5         23
6         -21005
7         32
8         21
9         6789
```

This is not desirable because the array elements have an unusual alignment. They should all finish, not start, in the same vertical line. To avoid this problem, specify that all these numbers occupy a given width, in columns, in the output.

Using I/O Manipulators

The header file `iomanip.h` defines several manipulators that can be included in the i/o stream to format the data to be input or output. So far, we have seen `endl`, but there are others (see Table 22.1).

Table 22.1: Manipulators in `iomanip.h`

MANIPULATOR	ITS PURPOSE
`Endl`	Start on new line
`Ends`	Insert `null` in output
`Flush`	Flush stream
`setiosflags(long flag)`	Set i/o flag bits
`resetiosflags(long flag)`	Clear i/o flag bits
`setfill(char fillchar)`	Set fill character to `fillchar`
`setprecision(int places)`	Set precision to `places`
`setw(int width)`	Set total field width

The manipulators `setiosflags`, `resetiosflags`, `setfill`, and `setprecision` remain effective until you specify otherwise. For example, once the fill character is set to a dot, it will remain a dot until another fill character is specified. However, the manipulator `setw` is only effective once.

COMPILER DIFFERENCES

Microsoft compilers do not allow you to alternate the alignment by setting the flags to `ios::left` or `ios::right`. Once you have set the alignment to left:

```
setiosflags(ios::left)
```

you have to reset it:

```
resetiosflags(ios::left)
```

Borland compilers allow you to set the alignment without resetting.

Manipulators are inserted in the stream in the same way that you insert a variable that you want to read or write. Do you remember how `endl` was used? For example, if we want to make sure that all the numbers in the array are displayed correctly, we can modify the `show` function:

```
void show (int array[], int from, int to)
{
   cout<<endl<<"Contents of array:"<<endl<<"Index"
       <<"  Value";
   for (int k=from;k<=to;k++)
     cout<<endl<<setw(3)<<k<<"      "<<setw(8)<<array[k];
}
```

Manipulating Field Width with *setw*

The change that we just made to the `show` function illustrates the use of the manipulator `setw`. In this case, each index value `k` will occupy exactly three spaces, no matter how many spaces are actually needed to write `k`. Similarly, each value in the array will occupy exactly eight columns. You should specify a width that is wide enough to accommodate the value you expect to show. If you specify a width that is not sufficient, the computer uses more columns so the number can be correctly written (which ruins your format). The default is `setw(0)`, which means that the minimum number of columns that are needed to represent the value will be used.

Manipulating Decimal Digits with *setprecision*

When you display floating point numbers, you may want to limit the number of decimal places shown. For example, if you are dealing with money, you probably want your results to contain up to two decimal places only. The `setprecision` manipulator can be used for this purpose. Once you have specified the number of decimal places to be displayed, this specification remains in effect until you change your settings with another `setprecision` manipulator. For example, to avoid displaying a price with more than two decimal places, you could do as follows:

```
float price;
cout<<setprecision(2);
...
cout<<price;
```

Any floating point variable that you forward to `cout` is displayed with only two decimal places until you use the manipulator again. Unfortunately, you may be left with other problems that will not be solved completely by this manipulator.

If your floating point variable needs fewer than two decimal places, it prints with fewer than two decimal places, and the decimal point may even be omitted. For example:

```
float x[3]={12.35,10.,5.};
cout<<setprecision(2);
for(int i=0;i<3;i++)
cout<<endl<<setw(8)<<x[i];
```

will display results as follows:

```
12.35
   10
    5
```

which may not be appropriate in a report. To align the decimal point, you will need to use the `setiosflags` manipulator.

The other problem you face is that, due to your settings of width and/or precision, the computer may choose to display the value in exponential (scientific) notation. If this does not suit you, you will have to resort to `setiosflags` to change it.

Manipulating Fill Characters with *setfill*

When you specify the width for a field (by using `setw`), and when the value to be displayed requires fewer columns than are available, the remaining columns are filled with blank spaces. In this case, we say that the fill character is *blank*. It is possible to use any other character as a fill character, instead of using blank. This is done by the `setfill` manipulator. Once a fill character is specified, it remains in use until you use `setfill` again. If you want to go back to using blank as the fill character, all you have to do is to use `setfill(' ')`.

For example, it is common to fill dollar values with a nonblank character such as * when printing checks. This is easy to implement:

```
cout<<"US$"<<setfill('*')<<value;
```

There may be other situations in which you want to include leading zeros, dots, or other characters, as well.

Manipulating I/O Controls with *setiosflags* and *resetiosflags*

The `setiosflags` and `resetiosflags` manipulators can be used to change the controls of a set of flags that affect the input/output operations. The flags are shown in Table 22.2.

Table 22.2: Flags in `iomanip.h`

FLAG	ITS PURPOSE
`skipws`	Ignore white spaces in input
`left`	Left align
`right`	Right align
`showpoint`	Show decimal places and point
`scientific`	Use scientific (exponential) notation
`fixed`	Use fixed floating point notation

To set a flag, use the `setiosflags` manipulator with an argument that consists of the sequence `ios::` followed by the flag you want to set. To reset the flag, use `resetiosflags` with the same argument. For example:

```
cin>>resetiosflags(ios::skipws);
```

resets the `skipws` flag and allows you to read blank (white) spaces in the input.

Reading White Spaces with *skipws*

The `skipws` flag is set as the default. Blank spaces are used as separators between values, and you cannot read them. If you read a string to an array of characters, the input stream assumes that the string is terminated when the first blank space is reached.

You may try to run the following program:

```
#include <iomanip.h>
void main()
{
   char mark[]="---------+";
   char name[30];
   cin>>name;
   cout<<name;
}
```

If you input a string such as `Sonny Bonds` as data, you will see the result `Sonny`, because the first blank space indicates the end of the string.

If you reset the `skipws` flag by including the following statement:

```
cin>>resetiosflags(ios::skipws);
```

you will get the same result! How can you read the blank spaces?

The only way you can read the blank spaces is to undo the automatic feature of C++ that reads character arrays as strings. You have to read the characters one at a time. Consider the following program:

```
#include <iomanip.h>
void main()
{                              // c8skipws.cpp
   const char enter=10;
   char mark[]="---------+";
   char name[30];
   cin>>resetiosflags(ios::skipws);
   cout<<mark<<mark<<mark<<endl;
   for (int i=0;i<29;i++)
   {
     cin>>name[i];
     if(name[i]==enter) break;
   }
   name[i]=0;]
   cout<<endl<<setw(5)<<i<<" characters read";
   cout<<endl<<name;

}
```

This program resets the `skipws` flag and reads the input string one character at a time. To avoid reading all] 30 characters, this program also tests whether the user has hit Enter. This is done by comparing the input with the code for Enter, as defined in the constant (the code for Enter is equal to the numeric value *10)*.

If you remove `resetiosflags`, the whole string will still be read, but all the blank spaces will be ignored. Worse than this, the code for Enter will not be detected, and you will have to actually complete the typing of 30 nonblank characters (which means that control characters such as Enter are also skipped).

The character array `mark` is used merely to help you keep track of how many characters you have typed.

Aligning Values with Left and Right I/O Flags

When a value needs fewer columns than there are available, you can align it to the left or to the right by setting either the left or the right flag. The default is to align to the right, which is probably what you want to do when displaying numeric quantities, but is less acceptable when displaying character strings.

Consider the following program:

```
#include <iomanip.h>
void main()
{
   char name[5][20];
   int code[5];
   for (int i=0;i<=4;i++)
   {
     cout<<endl<<"Enter a name:";
     cin>>name[i];
     cout<<endl<<"Enter a code:";
     cin>>code[i];
   }
   for (i=0;i<=4;i++)
   {
     cout<<endl<<setw(20)<<name[i]<<setw(6)<<code[i];
   }
}
```

This program reads a sequence of names and numbers from the keyboard, and then displays them on the screen. However, your results may look as follows:

```
Clarice              12
    Liz            4532
  Lucia             435
 Marcos              21
Claudio              43
```

Text on the Left, Numbers on the Right

The formatting above may not be your idea of good formatting—you may want to make sure that the names are left aligned and that the numbers are right aligned. The new listing would be as follows:

```
#include <iomanip.h>
void main()
{                                                // c8align.cpp
   char name[5][20];
   int code[5];
   for (int i=0;i<=4;i++)
   {
     cout<<endl<<"Enter a name:";
     cin>>name[i];
     cout<<endl<<"Enter a code:";
     cin>>code[i];
   }
   for (i=0;i<=4;i++)
   {
    cout<<endl<<setiosflags(ios::left)<<setw(20)<<name[i]

<<setiosflags(ios::right)<<setiosflags(ios::right)<<setw(6)<<code[i];
   }
}
```

The new output would then look as follows:

```
Clarice              12
Liz                4532
Lucia               435
Marcos               21
Claudio              43
```

& The left-alignment flag was reset so that you could use this program with Microsoft compilers.

Including the Decimal Point with *showpoint*

The `showpoint` flag forces the output of floating point numbers to include the decimal point and the trailing zeros to complete the maximum number of decimal places as set by `setprecision`. This flag is necessary because the decimal point and the trailing zeros may be omitted otherwise.

Outputting Numbers in Exponential Format with *scientific*

The `scientific` flag specifies that floating point numbers are to be output using the exponential format.

Outputting Numbers in Fixed Point Format with *fixed*

The `fixed` flag is the opposite of the `scientific` flag, and specifies that floating point numbers are to be output using the fixed point format.

Using Character Codes

In Skill 20, you were introduced to the type `char`. An ASCII code, consisting of an integer number from 0 to 255, is used to represent characters. It is not important to memorize which code corresponds to which character, but you should check out the following program:

```
#include <iostream.h>
void main()               // c8ascii.cpp
{
 // This program shows the character
 //      equivalent of a numeric code.
 char code,choice='y';
 int number;
 while(choice=='y')
 {
   cout<<endl<<"Input the code you want to know:";
   cin>>number;
   code=number;
   cout<<endl<<"This code corresponds to the
              character:"<<code;
   cout<<endl<<"Do you want to continue (y/n)?";
   cin>>choice;
 }
}
```

This program requests that you input an integer number, and it prints a character whose code corresponds to the number that you input. Notice that `code` was declared as a character, while `number` was declared as an integer number. When you output a character, the computer understands that you don't want to print the number, but, instead, that you want to print the character whose code is represented by that number. Thus, even though both `code` and `number` may be equal to 65, when you output `number`, you will see the number *65,* but when you output `code`, you will see the character *A*.

The program above simply reads an integer number and copies that number to a character variable (`code`), and then outputs the character variable.

& If you want to make it clear that the integer value is converted to a character, you can cast the type in the assignment by using `code=(char) number;`. However, this is done automatically by the compiler.

Using *ask, askwords, yesno,* and Boxes

The functions `ask`, `askwords`, and `yesno`, as well as the `Box` objects, were very useful for handling input and output. Although you cannot use the Windows graphic interface with the standard C++ input and output streams, I am providing an alternative that will still let you make limited use of these functions and boxes.

The *nofranca.h* Header File

The `nofranca.h` header file allows you to use the functions listed above and to use boxes in your programs, even though you are not using `franca.h` anymore. All you have to do is to use the following directive:

```
#include "nofranca.h"
```

You must include this header file before you attempt to include `franca.h`. Once this new header file is included in your program, it will prevent the standard `franca.h` from also being included. Any calls to these functions and the use of boxes will be interpreted by this new version, and simulated in the text interface provided by C++.

You may benefit from looking at this header file. By now, you should be able to understand the complete program. Notice the directive that tests `FRANCA_H` and `_CANVAS_H`. This directive prevents the header file `franca.h` from being loaded.

The Complete Code

Here is the complete code for `nofranca.h`:

```
#ifndef NOFRANCA_H
#define NOFRANCA_H
#define FRANCA_H
#define _CANVAS_H
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
#include <iomanip.h>

float ask(char question[])
{
   float answer;
   cout<<endl<<"Asking: "<<question;
   cin>>answer;
   return answer;
}

void askwords(char sentence[],int size,char question[])
{
   cout<<endl<<"Asking (sentence): "<<question;
   cin>>sentence;
}

int yesno(char question[])
{
   char answer;
   cout<<endl<<"yes or no: "<<question;
   cout<<endl<<"Please enter y or n: ";
   cin>>answer;
   if (answer=='y') return 1;
   return 0;
}
```

```
class Box
{
  char title[40];
  char message[40];
 public:
  Box();
  Box(char alabel[]);
  void say(float);
  void say(char msg[]);
  void label(int);
  void label(char msg[]);
  void place(int x,int y);
};
void Box::Box()
{
  strcpy(title,"");
  strcpy(message,"");
}
void Box::Box(char msg[])
{
  strcpy(title,msg);
}
void Box::place(int x,int y)
{

}
void Box::say(float value)
{
  cout<<endl<<title;
  cout<<endl<<setprecision(2)<<setiosflags(ios::showpoint);
  cout<<setiosflags(ios::fixed)<<value;
}
void Box::say(char msg[])
{
  cout<<endl<<title;
  cout<<endl<<msg;
}
void Box::label(char msg[])
{
  strcpy(title,msg);
}
void Box::label(int value)
{
  itoa(value,title,10);
}
#endif
```

Try These for Fun…

- Modify the program `c8ascii.cpp` to read a character and to display the corresponding numeric code.
- Write a program to display a table that uses two columns: the first column should contain integers starting with zero and going up to 255. The second column should contain characters corresponding to the codes displayed in the first column.
- Modify the program above to display the table in 10 groups of two columns, so all the codes can fit on a single screen.

Are You Experienced?

Now you can…

Survive in the real world without the `franca.h` software

Use stream input and output

Format input and output

SKILL

TWENTY-THREE

USING FILES

- Understanding and using files
- Using preprocessor directives
- Making a class for text files
- Inputting and displaying files
- Understanding basic file operations

In Skill 23, you will develop your ability to use files with your data. You will learn how to use the stream files available in standard C++. You will also learn how to develop a class of your own to handle files. Finally, you will become familiar with the most common file operations—searching and matching. You will use these techniques to further improve the point-of-sale terminal in the next Skill.

Understanding and Using Files

Since the beginning of the book, you have stored your programs on the computer's hard drive, so that you don't have to retype them every time you need them. In general, any set of information that is stored and that can be used later is called a *file*. In this section, you learn how to store data in files, so you can read them later or transfer them to another computer.

You work with files in essentially the same way that you have been working with the display and the keyboard. Of course, since you are already experienced with program files, you know that there are some subtle differences between working with files and working with the display and the keyboard:

- A file is identified by its name on the disk.
- You can write information (output from the computer) or read information (input to the computer) using the same file.

Since you choose the file name, it is not possible for the compiler to declare a file object beforehand. Input and output stream objects were declared with `cin` and `cout`, because you couldn't choose their names. You used `cin` for input and `cout` for output. When it comes to files, however, it is different. There may be several files you want to deal with, instead of just one file for input and one file for output. Therefore, you must declare your files as objects of the class `fstream` and give them names.

Once an object is declared, you can use member functions of the class `fstream`, and you will be able to establish an input and/or output stream using the file in the same way that you used `cin` and `cout`. Even the manipulators and the flags can be used!

Here is an example:

```
#include <fstream.h>
void main()
{
 fstream myfile;
 myfile.open("list.txt",ios::out);
 myfile<<"Anything goes.";
 myfile.close();
}
```

This program creates a file with the name `list.txt`, and writes the string `Anything goes.` in it.

& To use files, you must include the header file `fstream.h`.

You can run this program and look at the contents of the file `list.txt` using any word processor. You can also print the contents of the file from any word processor. Since we did not specify the directory in which the file will be located, it will be created in the current directory.

Manipulating Files

You can manipulate two kinds of files:

- Text files
- Binary files

Text files are organized in lines, and each line is marked with an `end of line` (`endl`). Data remain in readable form, and blank spaces separate each data field, so you can easily understand the file's contents. On the other hand, binary files are not organized in lines. One piece of information is written after another piece, and numeric values are stored in the way they are operated on in the computer memory—in binary notation. These numbers are not converted to a sequence of decimal digits so that you can easily read them. As a result, the printout of a binary file is not easily understood. In this book, we will deal exclusively with text files.

To use a file, the file must first be declared like any other object.

File Declaration

Since files are objects of the class `fstream`, the declaration is very simple:

```
fstream   identifier  ;
```

The *identifier* is simply a name that you use in your program to refer to the file—it is not the name stored on the disk! The correspondence between this identifier and the file name on the disk will be left for the `open()` member function.

Once a file is declared, we can use member functions to perform operations with it. The most important member functions are as follows:

- `open()`
- `close()`
- `eof()`

The *open()* Member Function

The following `fstream` member function:

```
open(char filename[], int access_mode);
```

establishes a correspondence between the file object used in your program and data on the disk. It also establishes how the file will be used: to input, to output, to append, etc. This function takes two arguments: a character string (`filename`), which determines the file name on the disk, and a flag (`access_mode`), which indicates the access mode.

File name

The file name is a null-terminated string. It can contain a full path to the file, including the drive, the directory, etc. However, when you include backslashes (\) in the string, they must be replaced by two backslashes (\\).

Access mode

The access mode will be one of the following modes:

For example:

`ios::out`	open file for output (write)
`ios::in`	open file for input (read)
`ios::app`	append file (start at the end of the file and write)
`ios::nocreate`	open file only if it exists (do *not* create it)

```
// This opens a file list.txt located on drive a,
//      directory \franca for input:
   myfile.open("a:\\franca\\list.txt,ios::in);
// This opens a file whose name is contained in the
//      character array "filename" for output:
   myfile.open(filename,ios::out);
```

Open attributes can be combined by using a logical `or` operator (|). For example:

```
yourfile.open(ìroster.txtî, ios::nocreate|ios::in);
```

In this case, the system tries to open the file `roster.txt` for input (`ios::in`). If this file is not found in the current directory, it is *not* created and the open will fail.

& Microsoft compilers usually create a new file if you try to open a nonexistent file. To avoid this, you must use the `ios::nocreate` option. Borland compilers will not create a new file if you try to open a nonexistent file.

Testing whether the file was successfully opened

It is a good practice to test whether the file was successfully opened. For example, a file may fail to open if an input file does not exist where specified, or if an output file cannot be created. This test is performed differently by Microsoft and Borland compilers.

With Microsoft compilers, you can test whether a file was correctly opened by invoking the `is_open()` member function after trying to open the file. If the function returns a zero, there was a problem opening the file. For example:

```
yourfile.open(ìrosterî,ios::in|ios::nocreate);
if( yourfile.is_open()==0 ) cout <<ìError: File was not openî;
```

With Borland compilers, you must compare the `fstream` object with a *NULL*. If this comparison is true, the file is not open. For example:

```
yourfile.open(ìrosterî,ios::in|ios::nocreate);
if(yourfile==NULL) cout<<ìError: File was not openî;
```

The *close()* Member Function

The `close()` member function is used to make sure that all data are written to the disk. In many situations, the data are not immediately written to the disk, and if the file is not properly closed, some data may be lost. There are no arguments to this member function.

The *eof()* Member Function

The `eof()` member function is very useful for inputting files. It returns a 1 if you try to read past the last data in the file. This end-of-file (eof) condition is very handy, because in most cases, you don't know beforehand how many pieces of data will be present in the file. What is the solution? Keep reading until `eof()` is reached.

& When you read the last piece of information in the file, you will not cause `eof()`. It is only when you try to read *past* the last piece of information that you will cause `eof()`.

Example—Reading and printing personnel information

The file `c8persnl.txt` contains personnel information. You can use any word processor to read and to print the contents of this file. Each line in this file contains a numeric identification, a first name and a last name, and a floating point number representing the hourly wage of the employee.

The contents of this file should look as follows:

```
 1  Clark Kent     20.00
 2  Alfred Newman   6.00
 3  Oliver Twist   15.00
 4  Huck Finn      17.00
 7  James Bond     32.00
10  Saint Nick     22.00
```

Notice that the identification numbers do not contain a complete sequence (some numbers, such as 5, 6, 8, and 9, are missing), which is the usual case in most files. In real life, you don't really know how many employees you will be reading. Therefore, you must check for *end of file*.

We will now write a program to list the employees, their ID numbers, and their hourly wages. Although it is a simple program, it requires that you pay attention to formatting. The approach is simple:

1. Declare the file and the variables.
2. Read the employee information until the end of the file is reached.
3. Display the ID number, the first and last name, and the hourly wage of each employee.

In fact, we can start the program by including the steps above as comments, and then inserting the appropriate code:

```
#include <fstream.h>   // c8wage.cpp
#include <iomanip.h>

void main()
{
  // Declare the file and the variables:
   char filename[]="c8persnl.txt";
   int id;
   char fname[20],lname[20];
   float wage;
   fstream employees;
   employees.open(filename,ios::in|ios::nocreate);
  // Read the employee information until
  //      the end of the file is reached:

    for (;;)
      {
         employees>>id;
         if (employees.eof()) break;
         employees>>fname>>lname>>wage;
  // Display the data of each employee:
         cout<<endl;
         cout<<setw(5)<<setiosflags(ios::right)<<id
             <<setw(20)<<setiosflags(ios::left)<<fname
             <<setw(20)<<lname
             <<setw(6)<<setprecision(2)<<setiosflags(ios::fixed)

<<resetiosflags(ios::left)<<setiosflags(ios::showpoint)<<wage;
       }
       char choice;
       cout<<endl<<ìEnter any character to finishî;
       cin>>choice;
}
```

Notice the following items in the code above:

- We initialized the character array `filename` with the file name. This works only if the file is in the same directory as your program. Otherwise, you have to specify the full path name.
- We did not check whether the file was successfully opened.
- We detected the end-of-file condition—it is important that you check the end-of-file condition after you attempt to read the beginning of new employee data. Remember that the end-of-file condition is set only after you try to read *past* the last information. If you check the end-of-file condition at the beginning of the loop (including by using a `while (!employees.eof()…)` statement), you will work with invalid data on the last loop iteration.
- We formatted the output—it is a laborious task to format the output, because sometimes you have to align the data to the left (names), and sometimes you have to align them to the right (numbers). Microsoft compilers require you to reset the left flag, instead of simply allowing you to set the right flag. Also, it is important that you set the fixed flag so that Microsoft compilers show decimal zeros.
- The last three lines are optional. Some compilers erase the screen soon after the program is run. When this happens, you barely have time to see the output. If you request the user to type anything, you will force the computer to wait until you are done reading.

It may still be desirable to add a couple of features to this program:

- Check whether the file was opened successfully
- Format the first and last names

To check whether the file was opened successfully, you could include an `if` statement, which would be different according to the compiler you are using. With a Borland compiler, you could use

```
if(employee==NULL) cout<<ìError: file does not openî;
else ...
```

With a Microsoft compiler, you could use

```
if(employee.is_open()==0) cout<<ìError: file does not openî;
else ...
```

Notice that only the condition inside the `if` statement is different.

Isn’t it disturbing that you may have to modify the code according to the compiler you are using?

In fact, one of the benefits of using a programming language like C++ is programs that do not have to be modified from one machine to another machine, or from one compiler to another compiler. However, many compilers have minor discrepancies in the way they implement a few features.

There is still hope—you will soon learn how to use the preprocessor to ease the pains of dealing with different compilers.

The output of the program above may still not be satisfactory, because the first name and the last name are printed in separate columns. We can fix this by copying the first name to a new character array, and then appending a blank space followed by the last name. For example:

```
char full_name[40];
...
strcpy(full_name,fname);
strcat(full_name," ");
strcat(full_name,lname);
```

After we use this sequence, the character array `full_name` will contain the full name of the employee, which we can print instead of the first and last names. In the next example, there is a complete implementation.

Example—Representing employees with objects

Another implementation of the program above is shown below. This new version uses an object to represent the employee; the formatting is embedded in the member function `display()`. To illustrate an alternative approach, the object was implemented using a `struct` instead of a `class`. The class `hired_person` can be implemented as follows:

```
#include <fstream.h>
#include <iomanip.h>
#include <string.h>
struct hired_person
{
   int id;
   char fname[20],lname[20];
   float wage;
   void display();
};
void hired_person::display()
{
   char full_name[40];
   strcpy(full_name,fname);
   strcat(full_name," ");
   strcat(full_name,lname);
   cout<<setiosflags(ios::right)<<setw(6)<<id<<"  "<<setw(40)
       <<setiosflags(ios::left)<<full_name
       <<setw(6)<<setprecision(2)<<resetiosflags(ios::left)
       <<setiosflags(ios::right)
       <<setiosflags(ios::fixed)
       <<setiosflags(ios::showpoint)<<wage;
}
```

The main program can be adapted as follows to use this class in the `c8wage1.cpp` program:

```
void main()                        // c8wage1.cpp
{
  // Declare the variables:
   char filename[]="c:\\franca\\c8persnl.txt";
   hired_person worker;
   fstream employees;
  // Open the file for input:
   employees.open(filename,ios::in);
  // Loop through all the employees:
  {


      for (;;)
      {
       // Read the employee data:
       employees>>worker.id;
       // Check whether it's the end of the file:
       if (employees.eof()) break;
       employees>>worker.fname>>worker.lname>>worker.wage;
       cout<<endl;
       // Print the employee data:
       worker.display();
      }
      cout<<endl<<endl<<ìEnter any character to finish:î;
      char enter;
      cin>>enter;
  }
}
```

If you move `display()` into a member function, you will simplify the main program—not only in its conception, but also in its maintenance—because, if the file information is changed, the main program will be less affected. However, to fully benefit from this, all operations with the file should be encapsulated in the class itself, or, at least, the reading should be moved from the main program into the class.

& Some programmers prefer the shorter form, `if(!employees.is_open()) cout<<ìError opening fileî;`, instead of `if(employees.is_open()==0) cout<<ìError opening fileî;`. Both forms produce the same result.

Using the Preprocessor to Overcome Incompatibilities

You will now learn a few more compiler directives:

- `#define`
- `#ifdef`
- `#ifndef`
- `#endif`

These directives can be useful in several situations, but our main goal in this section is to use them to overcome differences in compiler implementations.

Keep in mind, however, that the preprocessor is a program that manipulates your source program *before* it is compiled. The task of the preprocessor is not accomplished while your program is running.

The *#define* directive

The `#define` directive can be used to replace a given identifier with a given string. For example, if we use

```
#define pi 3.1416
```

the preprocessor will search the rest of the program for the identifier `pi` and change it to the sequence of digits `3.1416`. The result is the same as the result of the following statement:

```
const float pi=3.1416;
```

However, you can do much more than this—you can replace any identifier with any string, as long as the string does not contain blank spaces. For example, you can use

```
#define condition1 employee.is_open()
#define condition2 employee==NULL
```

and, later in your program, you can have

```
if(condition1) cout<<ìErrorî;
```

However, you still have to use either `condition1` or `condition2`. Wouldn’t it be nice if you could somehow choose which condition to use? Hold on!

The *#ifdef* directive

The `#ifdef` directive checks whether a given identifier was defined by a `#define` directive. It does not matter *how* it was defined, only *whether* it was defined. For example, if you are going to use a Borland compiler, you could use

```
#define Borland
```

If you are going to use a Microsoft compiler, you could use

```
#define Microsoft
```

If you then define one or the other, you can later determine which compiler is in use by simply checking whether the appropriate identifier was defined. For example:

```
#ifdef Borland
    #define condition employee==NULL
#endif
#ifdef Microsoft
    #define condition employee.is_open()==0
#endif
```

The *#endif* directive

In C++, we use braces ({}) to indicate where the `if` statement starts and where it ends. In the preprocessor, the `if` statement starts immediately after `#ifdef` and ends at the first `#endif`. This is why it was necessary to include the two `#endif` directives above.

If the identifier `Borland` was defined, the condition will be defined as `employee==NULL`, which is precisely what we want inside an `if` statement when using a Borland compiler. If `Borland` was not defined, the preprocessor will skip through the program until the `#endif`. At that point, it will check whether the identifier `Microsoft` was defined.

The identifier `condition` will be set to either `employee==NULL` or `employee.is_open()==0`, according to which compiler was indicated.

Therefore, you could use the following statement later in the program:

```
if (condition) cout<<ìErrorî;
```

What if neither option was defined?

In that case, the condition will not be defined and your program will cause an error. If you don't want this to happen, you may consider setting the condition to a string that will never be true. Then, if neither `Borland` nor `Microsoft` were defined, you will simply skip the test. For example:

```
#define condition 0!=0
#ifdef Borland
    #define condition employee==NULL
#endif
#ifdef Microsoft
    #define condition employee.is_open()==0
#endif
```

& The `#ifndef` directive works the opposite of how the `#ifdef` directive works. If the identifier is *not* defined, the preprocessor will work on the lines that follow. If the identifier *is* defined, the preprocessor will skip to the next `#endif`.

Testing whether different files were successfully opened

Our preprocessor technique to check whether a file is open has one major inconvenience: it only checks a given file (`employee`). What if you need a different file? What if you use several files?

In these cases, it is a good idea to combine the preprocessor directives with a function in which you pass the file as a parameter.

This function, which I named `fileopen`, will check whether a given file is open and will return either a 1 (yes, it is open) or a zero (no, it is not open).

Here is a possible implementation:

```
#include <fstream.h>
int fileopen(fstream & datafile)
{
   int yesopen=1;
   #ifdef Microsoft
        yesopen=datafile.is_open();
   #endif
   #ifdef Borland
        yesopen=!(datafile==NULL);
   #endif
   return yesopen;
}
```

You could also make this function an inline function. How?

When you compile this function, the preprocessor will check whether you defined either `Borland` or `Microsoft` (don't define both!). If you chose `Borland`, the `Microsoft` code will be skipped. Your function will look as follows:

```
int yesopen=1;
     yesopen=!(datafile==NULL);
return yesopen;
```

If you chose `Microsoft`, the `Borland` code will be skipped.

All you have to do is to invoke the `fileopen` function in the program:

```
if(fileopen(employee)==0) cout<<ìErrorî;
```

The code for this function is included in `c8tfile.h`, and is used by the `textfile` class, which will be discussed in the next section.

Try This for Fun…

- Modify the program `c8wage1.cpp` by moving the *read* operation into the class. Include a member function `read` to read the information of the next employee from the file. This function should test for the end-of-file condition, and return a 1 if data were successfully read or a zero if otherwise.

Making Classes to Deal with Files—the *textfile* Class

After you have used files in a couple of applications, you may wonder whether you can reuse part of your work to deal with different kinds of text files. You *can* define a class to deal with text files that handle all the operations we have used so far.

You should remove all operations that depend on files in general and on the record structure from the main function in particular. An object of the new class `textfile` (or of a class derived from this class) will be able to operate by itself.

In our payroll example, if a class `workers` (derived from `textfile`) is available, the operation of the main function will look as follows:

```
{
 workers staff; // Declare an object of class workers
 for(;;)
 {
   if (staff.read()==0) break;
   staff.display();
 }
}
```

Did you notice what this class gives you?

- You don't have to open the file—it will be automatically opened by the constructor when you declare an object.

- You don't have to mention an `fstream` object in your program, or perform file operations at all.
- The main function does not have to know the structure of the records. You don't have to read each field like you did before. You can use the member function `read` to read one complete record at a time.
- The member function `read` can be designed so an integer is returned as a result. If the function returns a zero, the end-of-file condition has been reached.

Of course, the structure of each record will be different for each kind of file with which you work. What should you do? If you use inheritance, you can design a class that deals with an empty structure, and then redefine the member functions as needed in your files.

You can also implement in this class the ability to locate a record if you are given the value of the first field (which acts like a key). For example, you can ask your file object to locate the employee whose ID number is 7.

The `textfile` class is completely implemented in this section. I hope that this class will be useful in your programs. However, it is most important that you understand how to use classes to manipulate files and reuse your work. The code for this class, as well as for the `fileopen` function, is included in `c8tfile.h`.

For the sake of convenience, the `textfile` class uses the header file `nofranca.h`, which was presented in a previous Skill. There are two advantages to using `nofranca.h`:

- You can use the functions `askwords`, `yesno`, etc.
- You can choose either the plain, text interface of C++ or the graphic interface of `franca.h`. If you want to use the plain interface, do nothing. If you want to use `franca.h`, all you have to do is to use `#include "franca.h"` in your program before you include `c8tfile.h`. The files `franca.h` and `nofranca.h` are exclusive. Whichever file you include first will prevail and prevent the other file from being included.

Class Declaration

Here is the declaration for our `textfile` class:

```
class textfile
{
  protected:
   fstream data;
   char datafile[40];
   int filemode;
  public:
   char id[40];
   textfile();
   textfile(char filename[],int mode=ios::in|ios::app|ios::nocreate);
   ~textfile();
   void display();
   virtual int read();
   int find(char id[]);
   int input();
   void write();
};
```

Data Members

Three data members are protected, because you may want to use them in the classes you derive from `textfile`:

- `data` is an `fstream` object. It is clear that you need one of these objects to handle the file operations.

- `datafile` is a character array that holds the name of the file you will be using on the disk.
- `filemode` is an integer that holds information about what is being done with the file.

There is a public data member `id`, which is a character array that holds the ID field (key) of the record being examined. It is public, because any piece of program can access it.

Member Functions

Most member functions have a straightforward purpose:

- `textfile` has two constructors—the parameterless, default constructor asks you for the file name to be used. The other constructor takes the file name as a parameter.
- `~textfile` is a special member function called a *destructor*. In most cases, you do not have to include a destructor in a class. Destructors are invoked automatically when the object you declared is discarded. In this case, the destructor will be used to close the file. Destructors work in an opposite way from constructors.
- `display()` is a function that displays the contents of the current record on the screen.
- `input()` is a function that inputs contents to a record from the keyboard.
- `find()` is a function that searches the file for a record whose first field matches the parameter. Notice that the parameter is a character string. This case is a very general case, because, in a text file, any kind of data is represented by a character string. The only restrictions that we impose are the size of the key (39 characters) and the position of the key (the key must be the first field in the record).
- `write()` is a function that writes the current record to the file.
- `read()` is a function that reads a record from the disk.

Constructors

Here is the code for the constructors:

```
#include ìnofranca.hî
textfile::textfile()
{
   askwords(datafile,40,"enter the file name:");
   filemode=ios::in|ios::app|ios::nocreate;
   data.open(datafile,filemode);
   if(fileopen(data)==0)
   {
      if(yesno("File not open, create?"))
      {
        filemode=ios::out;
        data.open(datafile,filemode);
        if(fileopen(data)==0) // This tests whether the file is now
open
        {
          yesno("Sorry, file does not open!");
          exit(0);
        }
        else;
      }
      else exit(0);
   }
}

textfile::textfile(char filename[],int mode)
{
   if(sizeof (datafile)>strlen(filename))
             strcpy(datafile,filename);
   data.open(datafile,mode);
   filemode=mode;
   if(fileopen(data)==0)
   {
      yesno("Error in opening, will you check?");
      exit(0);
   }
}
```

☞ The functions in the listing above use `nofranca.h`, which allows you to use the functions `askwords` and `yesno`. If you would like to use the graphic interface, all you have to do is to use `#include ìfranca.hî` before you include `c8tfile.h`. Also, remember that if you use `franca.h`, your main program is `mainprog`, not `main`.

& When you use `franca.h`, you cannot use formatting in `cin` and `cout`. Use boxes instead.

The `exit(0)` function can be used in an extreme error situation to terminate the program.

The constructors obtain the file name, and open the file for inputting and appending.

& The expression `ios::in|ios::app` may seem strange to you. The vertical bar represents the logical `or` operator. It allows you to use the file either for inputting or for appending.

Destructor

The code for the destructor is also very simple:

```
textfile::~textfile()
{
   data.close();
}
```

Inputting from the Keyboard and Displaying to the Screen

The code for the `display()` and `input()` functions is very interesting:

```
void textfile::display()
{
}
int textfile::input()
{
   askwords(id,40,"Enter id code");
   if(id[0]=='0') return 0;
   return 1;
}
```

What happens in these functions? The `display()` function does not show anything! Does it look right? Where are the IDs, names, and hourly wages?

Remember that you cannot solve all problems at once. If you include names, hourly wages, and other information in your class, it will not be reusable! (In other applications, you may be dealing with license plates, vehicle manufacturer, etc.) You should use the `textfile` class as a base from which to derive other classes. In fact, the `display()` function should be replaced by your own version of the `display()` function when you derive a new class. You will then display all the data members in a format that suits you.

The `input()` function requests input for the ID field. You may later provide an `input()` function that uses the previous `input()` function to input the ID, and that continues to input the other data members, as well. Or, you can totally override the previous version of the `input()` function.

The `input()` function in the code above requests input from the keyboard and tests whether you have entered a null string (for example, if you just hit Enter). This is one way to determine that you no longer want to input data. The function then returns a 1 or a zero. If you like this idea, you should use it. If you don't like it, just use something different in your derived class!

Reading and Writing from a File

Next, we will examine the code for the `read()` and `write()` member functions:

```
int textfile::read()
{
// This function reads the next record from the file:
   data>>id;
   if (data.eof()) return 0;
   return 1;
}
```

```
void textfile::write()
{
   data.close();
   data.open(datafile,ios::app);
   if(fileopen(data)==0)
   {
     yesno("Failed reopen for append, will you check?");
     return;
   }
   data<<endl<<id<<' ';
   data.close();
   data.open(datafile,filemode);
}
```

The `read()` function simply reads the next ID from the file. Again, you should complement this function with a member function of your own derived class, so the other fields are also read from the file.

The `write()` function operates in a similar way, except that it closes the file, and then reopens it for output. Notice that a blank space is written after the ID. It is important to use blank spaces to separate fields in a text file. After the ID is written, the file is closed again, and then opened in the previous mode. You should also complement this function with a member function of your own derived class.

Finding a Given Record in a File

Finally, here is the code for the `find()` member function:

```
int textfile::find(char idnumber[])
{
    data.close();
    data.open(datafile,filemode);
    if(fileopen(data)==0)
    {
      yesno("Failed to reopen for find, will you check?");
      exit(0);
    }
    for(;;)
    {
     if(data.eof())break;
     if(strcmp(id,idnumber)!=0)  read();// Virtual read!
     else return 1;
    }
    data.close();
    data.open(datafile,filemode);
    return 0;
}
```

This function simply reads all the records, and compares the key field with the ID number for which you are looking. If this record is found, the function returns a 1. Otherwise, it returns a zero. The search starts at the beginning of the file every time. It may not be the smartest way to find a record, but it is simple enough. It is necessary to close and to reopen the file, so it will be positioned at the beginning.

A very important aspect of the `find()` function is the reading of a record. This function reads a record from the file by invoking the `read()` member function.

Notice there is a standard `read()` that comes from the original `textfile` class and another `read()` that you wrote for your derived class. Which `read()` am I talking about?

If you use the standard `read()`, as defined for `textfile`, only the ID field will be read, and the next attempt to read may erroneously use either a name or an hourly wage as a key. This is because each time you read something from the file, the file is positioned to read the next information recorded.

For example, if your derived class reads records that have an ID and a name, the original `read()` member function is not aware of the name field and will only read the ID. When you next read (using the original `read()`), the computer will start reading where it last stopped and read a piece of the name assuming it is the next ID.

It is imperative that you use the `read()` function that you defined for your derived class instead, which is why `read()` is a virtual function in `textfile`.

You will find the `textfile` class in the header file `c8tfile.h` in your directory.

& If you need to pass a `textfile` object as an argument to a function, pass it by reference only. If you pass it by value, you will create and destroy a copy. After the copy has been destroyed, the file will be closed.

Try This for Fun…

- Derive a class from `textfile` to operate with the `workers` (`c8persnl.txt`) file. Write a program to list the file's contents.

Understanding Basic File Operations

You should have noticed by now that files are excellent for keeping information that you may need to use several times. When you use a file, you avoid retyping all the data every time you need to use them. Furthermore, it may be impossible to retype a large quantity of data. The example discussed in the previous section illustrates how you can keep employee information conveniently stored.

Notice that the employee identification as well as the first and last names are not supposed to change. The hourly wage may change, but it is unlikely that it will change very often. For this reason, the information kept in this file is somewhat permanent.

The `textfile` class provides the basic functions to do as follows:

- Input data from the keyboard
- Write data to the file
- Read data from the file
- Write data to the screen
- Find and read a record from the file

This class allows us to include new employees in the file, to check the data of an existing employee, etc. However, in real life, other operations are needed to maintain a file.

- What if an employee leaves the company? We should be able to remove the record from the file. This operation is called *record removal* or *delete*.
- What if an employee needs his or her hourly wage raised? We should be able to fetch the employee's record, to modify the value of the wage, and to write it back. This operation is called *record update*.

File Updating

It is regrettable that the *update* operation is not easily performed in a text file. Why not? The records of each employee may have different sizes. Although this may not seem like a problem when you regard each record as a line on a page, it is a real problem in the file, because the lines are written one after the other. Therefore, if the updated record exceeds the size of the previous record, there is no way to fit it in the same place. Besides, each time you read from the file, the computer gets ready to read from or write to the next available position. You must then make sure that you rewrite the record in the same place from which it was read.

However, it is possible to implement update operations in text files. The easiest solution is to create a second file with the updated information, so you do not update data in place, but, instead, you update data to the new file. On the other hand, binary files can be a little easier to deal with in this situation, if you constrain all the records to fit in a given size.

& You will benefit from forcing your files to have records of fixed lengths.

File Matching

Another important operation that you may want to perform in a file is *matching*. You use matching when you have to operate on two or more files, and when you fetch a record from one file and need to find a record in the second file that matches it.

For example, consider a file `workhours`, which contains an employee code and the number of hours the employee worked. To compute the value of the paycheck, you have to look into each record of the `workhours` file, and then use the employee code to find a match in the `worker` file to determine the name and the hourly wage of that employee. With this information in hand, you can compute the value of the paycheck.

The file-matching operation is simple when you use `textfile`, because the member function `find()` can locate a record with a given code.

In this example, we will produce the payroll. We are given two files:

- `c8persnl.txt` contains the employee code, the first name, the last name, and the hourly wage.
- `c8payrll.txt` contains the employee code and the number of hours the employee worked.

Our task is to display a list of all the employees that are to be paid, with their personal data and the amount to which they are entitled. Since the hourly wage and the number of hours worked are located in different files, it is necessary to match the employee record from one file with its record in the other file.

It is possible to get each record from the `payroll` file, and then find a match in the `worker` file, or to get each record from the `worker` file and find a match in the `payroll` file. It is very likely that you will read the file that contains transient information (in this case, the `payroll` file), and update the permanent file (the `worker` file).

We have to do the following things for each record we read from the `payroll` file:

- Get the ID code and the number of hours worked for this employee.
- Find a record in the `worker` file with the same ID code.
- Compute the paycheck value for this employee.

Of course, we have to decide on the action to be taken when there is no match. We will greatly simplify the matching operation if we use the `hired_person` class to handle the `worker` file, because the `find()` member function can be used to locate an employee's record, if we are given the ID code.

The `payroll` file has not been defined in a class, but it can easily be derived from the `textfile` class. It will be a very simple task, because we have to include only one data member—the number of hours worked—and one member function. All we actually need to do is to read from this file. There is no need to code the functions for displaying, writing, inputting, or finding!

Example—the *payfile* Class

One implementation for the class that handles the `payroll` file could be as follows:

```
#include ìnofranca.hî
class payfile: public textfile
{
   public:
    float hours;
    virtual int read();
};
int payfile::read()
{
   if(textfile::read()==0)return 0;
   data>>hours;
   return 1;
}
```

The `read()` function that is implemented for the `payfile` class calls the `read()` function of the base class `textfile`. It is always possible to do this. All you have to do to explain that you are not using the member function of the current class is to fully qualify the function name. How do you do this? Use the class name, two colons, and the function name:

```
class name ::  function name  ( );
```

For example:

```
if(textfile::read()==0)return 0;
```

When both file classes are available, the payroll program becomes quite simple:

```
void main()                        // c8payrll.cpp
{
 // Declare the variables:
   char filename[40];
   cout<<"Opening the employee's file:";
   hired_person worker;
   cout<<"Opening the payroll file:";
   payfile workhours;
   float paycheck,payroll=0;
   char idnumber[40];


 // Loop through the primary file:
      for (;;)
      {
       // Read the record from workhours
       //      if not eof:
       if( workhours.read()==0) break;
       // Find the employee data in the worker file:
       if( worker.find(workhours.id))
       {
          paycheck=worker.wage*workhours.hours;
          payroll=payroll+paycheck;
          worker.display();
          cout<<"  $"<<setw(12)
              <<setiosflags(ios::fixed)<<paycheck;
       }
       else
         cout<<endl<<"No record found for worker:"
             <<worker.id;
      }
      cout<<endl<<"Payroll total: $"<<setw(12)<<payroll;
}
```

Try This for Fun…

- Implement the member functions `input()`, `write()`, and `display()` for the `payroll` file. In your implementation, all records should be written to have exactly the same size (use blank spaces to fill space, if necessary).

Are You Experienced?

Now you can…

Store data in files

Use the preprocessor directives `#define`, `#ifdef`, `#ifndef`, and `#endif`

Develop special classes to deal with your files

Read and write files in your applications

Develop applications that handle more than one file

SKILL

TWENTY-FOUR

IMPLEMENTING A REAL-WORLD POINT-OF-SALE TERMINAL

- Enhancing the point-of-sale terminal
- Manipulating a catalog on disk file
- Reusing previously defined classes

To further improve your skills in developing applications, you will work on a real-world implementation of the point-of-sale terminal. This new terminal will get the product code from the keyboard and locate the product description and price in a file.

Revisiting the Point-of-Sale Terminal

The point-of-sale terminal implemented in Skill 21 was inconvenient because the entire parts catalog had to be input for each program execution. By keeping the data in a disk file instead of in memory, it is possible to reuse the same data after shutting down the computer and restarting it.

This implementation uses the same terminal, but, this time, you will implement the catalog using a disk file.

Implementing the New Features

If you carefully examine the latest version of the point-of-sale terminal, you may notice that the `catalog` class encapsulates all the catalog operations. In fact, if the `catalog` class is altered to keep the data in a disk file as opposed to in an array, the problem will be solved.

Why do unnecessary work? All we have to do is to design an alternative `catalog` class!

Since you have undoubtedly become an enthusiast of inheritance, you may think of deriving a new class from `catalog` and using it in this implementation. Indeed, this can be done.

However, the existing `catalog` class contains a few things that we don't really need. For example, why would we need an array if all the data are already in a disk file? In this case, it may be worthwhile to redesign the `catalog` class.

ABSTRACT CLASSES

If both `catalog` classes are needed, the smart thing to do would be to design a `catalog` class whose functions are virtual and inoperative (such as the `display` member function in the `textfile` class) and then derive two classes from it, `arraycatalog` and `filecatalog`, for example. Both classes can still be regarded as catalogs, but each one has a distinctive way of operating. The base class, in this case, will never be used to create an object; it is what we call an *abstract* class.

Deriving the *catalog* Class from *textfile*

The `catalog` class can be easily derived from `textfile`. Consider the following declaration:

```
struct product
{
   char code[40];
   char description[40];
   float price;
};

class catalog: public textfile
{
   product saleitem;
   virtual int read();
   void write();
  public:
   virtual product find(char part_number[]);
   virtual product find(int somecode);
};
```

Two *find()* functions?

You may be wondering why we declared two functions with the name `find()`. Indeed, the `textfile` class needs only one function to handle a character array as the key. However, the terminal program that is already working uses an integer as the key. If you include this new function, you provide compatibility with the current terminal, without having to change the code!

Another look at virtual functions

Even though virtual functions were first introduced in Skill 17 and were used again in the `textfile` class, it may be a good idea to review them here.

When you first learned about virtual functions, you had a runner and a skater that could run. Each one ran in a particular way. So, if you had

```
runner julia;
skater mike;
julia.run();
mike.run();
```

when your program was compiled, the compiler translated these with no trouble. The compiler made sure that `julia` used the `run()` function defined for a runner and that `mike` used the `run()` function defined for a skater. Therefore, as long as the compiler knew which kind of object would be using the function, it could easily determine which function to call in each place.

However, there are situations in which it is impossible for the compiler to determine which class of object will be used. Only when the program is actually executing will the true nature of the object be known. By then, however, the compiler is long done with its task!

The example in Skill 17 was a function that received a parameter of the class `runner`:

```
void march(runner volunteer)
{
   volunteer.run();
}
```

The problem is that due to inheritance, you can actually receive a runner or a skater (since a skater is also a runner, by inheritance). In fact, since this is a function, you can use it sometimes with a runner and sometimes with a skater. The compiler needs to bind one `run()` function, but the compiler does not know whether you will be using a runner or a skater (or any other class you may decide to create).

When you declare a function to be virtual, the compiler postpones the actual binding of the code to the appropriate function until your program executes. The compiler simply adds extra code to determine at runtime which function to use. While your program runs, this extra code determines the appropriate function to bind (this is also called *late binding*).

Ideally, all functions should behave as virtual functions to implement perfect polymorphism. Why do we have to state that the function is virtual then?

For practical reasons. The extra code needed for the late binding makes your programs longer and slower, and, as long as the compiler can determine the class of the object, the virtual attribute is not really needed. Therefore, C++ gives the programmer the burden of specifying which functions need to have the late binding treatment.

An interesting situation in which late binding (a virtual function) is needed occurs in the `textfile` class. The `find` function calls the `read()` function at some point to read the next record. Remember there is an original `read()` function that only reads the ID field, and there may be other functions that read additional fields. You need to read the complete record, not only the ID. However, inside the `find` function, you have no idea which kind of object with which you are dealing. Well, it does not matter. If the `read()` function is virtual, the late binding will figure out which one to use.

Displaying and Inputting

Classes derived from `textfile` should provide their own versions of displaying and inputting. Where are they?

Nowhere! You don't really need to input data to the catalog, nor do you need to display data. You can input data and check them using any word processor. A sample file, `c8parts.txt`, is included with some items. You can use this file, or you can add more items to the list.

The Constructor

There is no need to specify a constructor. The standard constructor for the `textfile` class can be used—it will ask for the file name to be used and then open it.

The *read()* and *write()* Functions

The `read()` and `write()` functions transfer the contents of the current record, the structure `saleitem`, to and from the disk file. Here is the code for these functions:

```
int catalog::read()
{
  int testeof=1;
  testeof=textfile::read();
  if(testeof==0) strcpy(id,"");
  strcpy(saleitem.code,id);
  data>>saleitem.description;
  data>>saleitem.price;
  return testeof;
}
void catalog::write()
{
  textfile::write();
  data<<' '<<saleitem.description;
  data<<' '<<saleitem.price;
}
```

Notice that both functions use their base class counterparts (`textfile::read()` and `textfile::write()`). It is not strictly necessary, but it is a good idea to reuse existing code. In this case, the product code is duplicated, because `catalog` inherited the character array `id` from `textfile`, and the structure defines a product code. For this reason, whenever you read information into the structure, the code is also copied to `id`.

Finding the Parts

Finally, the functions to locate a record when the product code is given are listed below.

```
product catalog::find(char part_number[])
{
  int kfind;
  kfind=textfile::find(part_number);
  if (kfind==0)
  {
    saleitem.code[0]=0;
    id[0]=0;
  }
  return saleitem;
}

product catalog::find(int somecode)
{
  char thecode[40];
  itoa(somecode,thecode,10);
  return find(thecode);
}
```

You will find the complete code for this project in the files `c8catlg.h` and `c8term.h`. The program `c8term.cpp` uses `franca.h` and the Windows graphic interface.

```
// Choose either:                c8term.cpp
//      #define Microsoft
//      or
//      #define Borland
#include "franca.h"
#include "c8catlg.h"
#include "c8term.h"
void mainprog()
{
   saleterm cashregister;
   cashregister.operate();
}
```

You should use the appropriate `#define` for your compiler to check that the files were successfully opened. You can also use the program above with the standard nongraphic interface. All you have to do is to remove `#include Òfranca.hÓ` and change `mainprog` to `main`.

Are You Experienced?

Now you can…

Manipulate a catalog on disk file

Reuse previously defined classes

PART IX

WHAT A LIFE!

In Part IX, you will be exposed to some of the common problems a computer programmer has to face, and you will develop another application using object-oriented programming.

Finally, you must realize that it is not possible to learn everything about C++ in an introductory book, much less about computer programming in general. We will discuss the relevant skills that you may want to develop during your career.

SKILL

TWENTY-FIVE

DEVELOPING APPLICATIONS WITH OBJECT-ORIENTED PROGRAMMING

- Understanding object-oriented programming
- Reusing software
- Completing a small project—a spelling game
- Implementing this project with object-oriented programming

In this Skill, we will summarize the important techniques of object-oriented programming (OOP), and we will propose, discuss, and implement a short project using these techniques.

The recommendations you will find here are not meant to be followed by the letter. There are several different opinions on how to design and maintain software. You should read this material and use your own judgment when the issue is brought to your professional attention.

The OOP Mentality

Object-oriented programming provides tools that make it easier for the programmer to complete his or her professional work. However, a new tool is useless until you understand how to use it. Keep in mind that a piece of software that you write today may be used again in the future. The more this software is reused, the more time and money you will save! Make your software easy to use, easy to expand (preferably, by using inheritance), and easy to maintain.

An efficient programmer can develop a reliable piece of code in a short amount of time. How can we significantly improve programming efficiency? Of course, being experienced and smart will help, but what else can you do?

You can save a significant amount of time in development and maintenance by reusing software. The following sections examine two interesting aspects of reuse:

- Reuse what is available.

- Build for reuse.

There are other, more limited ways of using a piece of software that has already been developed: software redoing and software recycling.

Redoing, Recycling, and Reusing Software

Suppose you wrote a piece of software that reads words from the keyboard and stores them in a disk file. A few months later, someone asks you to develop software that, among other things, reads words from the keyboard and stores them in a disk file.

Here are some of the actions that programmers may choose:

- Sit in front of the computer and write a piece of program. It is just an easy piece after all, and can be done in a couple of hours! You may think that this is *not* a good idea. Indeed, it is not. However, it is what most programmers end up doing. There is no established terminology to denote this approach. For our discussion, we will call this *software redoing*.
- Remember that a similar piece of program was written some time ago. Locate this piece of program on the disk, copy it to your new program, and take the opportunity to make it a little better. We call this *software recycling*.
- If the previous piece of software was built for reuse, it was probably stored in a header file. Include this file in the program, and reuse it in this new project. We call this *software reusing*.
- Another possibility may occur: the previous piece of software was built for reuse, but what you need now is slightly different from what you had then. If you can still use what was previously available, it is still reuse, as long as you do not make any changes to the original software that was available on the disk. You may derive new classes from it and even add them to the original file, as long as the original classes remain intact.

Redoing Software instead of Recycling

This is, regrettably, what most programmers end up doing. After all, the purpose of a programmer is to program, not to dig through buried code! The harm of redoing is twofold:

1. A programmer’s memory is not as reliable as he or she thinks it is. In the process of redoing a program, a programmer may spend the same amount of time as when they first developed it. Even during the second time around, bugs (often the same bugs) appear and consume time.
2. If you continue with this type of approach, it may yield projects with several pieces of code that are not exactly the same, but that end up doing the same things. If maintenance is required, all these pieces will have to be located, examined, and changed.

Recycling essentially eliminates the first inconvenience of redoing, because it saves the programmer time and may jump directly to a tested version of the software. Still, it does not do much for maintenance.

Reusing Pieces of Software

Reuse really helps with maintenance. As a reusable piece of software is updated, it becomes relatively easy to promote this update throughout all the software projects. Any improvements in the reused pieces can promptly benefit all the software.

What is software reuse? We say that software is reused if the same piece of code is used over and over. It could be in the same program, in the same package, or in completely unrelated software.

Indeed, software reuse started with the use of functions (or *subroutines*, in the early nomenclature). Instead of developing a piece of code all the time, programmers learned to use functions from a library to speed up their work. Reuse not only made programming faster, but also resulted in more reliable programs. Why? Since these functions were extensively used, they were subjected to more exhaustive testing than a unique piece of program.

Objects introduced a new dimension to reuse. If you had functions in a library and you wanted something slightly different, you could not use them anymore. Either you had to modify them (using different, untested code) or you had to redo them. Also, you could use any data with any function. There was no way to prevent your data from being inappropriately modified.

With classes, it became possible to limit the use of data and to modify part of the operations while inheriting the rest.

Extensive reuse requires strict discipline. A significant amount of time can be consumed just checking for software that can be used. Worse than this, some class libraries are really difficult to understand, and the programmer will be tempted to develop his own solution, instead of losing time while learning how somebody else solved the problem. Management must use good judgment in these cases. It is true that a programmer may often come up with a piece of code in less time than would have been needed to learn how the available software works.

The problem is maintenance. Chances are it will cost more to maintain software built out of pieces that are not reusable. It is just like maintaining a house built out of custom-sized windows and doors!

An interesting compromise, when you are tempted to build your own solution, is to build your own reusable solution. Instead of only developing something that fits your current need, develop something that can also be reused.

BUILDING FOR REUSE

Some programmers think they are reusing a great deal just because they use a class library. This is not the whole story! They will still lose a lot of time redoing things that could already be done. It is OK to be too lazy to use the keyboard, but never be too lazy to think!

Anytime you are asked to do a task, think of the future. Chances are someone will come up with a similar task again. Can you create a class that will be useful to that problem? Can you make this class easy to upgrade? Can you keep your own class library?

Investing instead of spending

It may take a little longer in the beginning to build your library, since you will be trying to solve your current problem as well as some of your future problems. It is an investment, though. As more tasks are assigned to you, you will start using your class library and be able to finish your job earlier!

Not to mention that your software will become more tested and reliable each time.

Short Project—Spelling Game

In this section, we will describe a project and its implementation. This project requires a computer with a sound board. The purpose is to develop software to assist children in checking their spelling abilities.

This software plays the sound of a word that has been previously digitized and stored in a file. Then, a message box requests the player to type the word that was heard. If the spelling is correct, the player will be notified and the score will be updated. The time the player takes to respond is also kept.

This procedure is repeated for several words. When the list of words is completed, a score with the number of hits and misses and the total time consumed are displayed.

The program can also play the sound and show the spelling of the corresponding word once before it starts to ask questions.

Suggested Implementation

Use your sound board to record each word in a separate file. Restrict yourself to words that are less than eight characters, which simplifies the problem.

You can keep the list of words in an array of characters in memory or in a disk file. Consider storing them in a disk file and loading them to an array. If you use up to only eight characters, you will be able to use the same word to identify the file where the sound is kept.

The list of words can be input directly into your file using any word processor. Later, you may want to allow your program to request words from the keyboard, as well.

Procedure

Here is a general outline of the algorithm:

1. Consider *wordnumber*—the total number of words (and sound files) available.
2. Consider *testnumber*—the total number of words per spelling test (typically, from 6 to 20).
3. Load the words from the file into the memory array.
4. For each word in the array:
    a. Display the word in a box for a few seconds.
    b. Play the corresponding sound.
5. Repeat the following items *testnumber* times:
    a. Choose a random number from 1 to *wordnumber*.
    b. Play the sound corresponding to the word in position *wordnumber*–1 in the array.
    c. Start a timer.
    d. Request the player to spell the word.
    e. Check the time spent.
    f. Check whether spelling is correct and notify the player.
    g. Update the score.
6. Inform the player of the score and the time spent.

Extensions

Once this software is developed and runs satisfactorily, you may improve it by

- Avoiding the repetition of words already spelled correctly
- Graphically illustrating the progress by either building a picture when spelling is correct or building a picture when spelling is incorrect

For example, you can build pieces of a hangman when spelling is incorrect. Another suggestion, which is implemented, is to build one pile of coins when spelling is correct and another pile when spelling is incorrect.

User Interface

In this case, the user interface may be the key factor in determining the success of your product. Assume that users are supposed to be in grade school. Be cool! If you illustrate the session with pictures and sounds, you will grab the attention of your users. If you forget this fact for a few seconds, your players will be bored and will turn your product off!

Figure 25.1 shows the computer screen after a session. Each green coin (in the left-hand pile) represents one word spelled correctly, and each red coin (in the right-hand pile) represents one word spelled incorrectly. In addition, every time the student hits a correct spelling, a cheerful *All right!* is played.

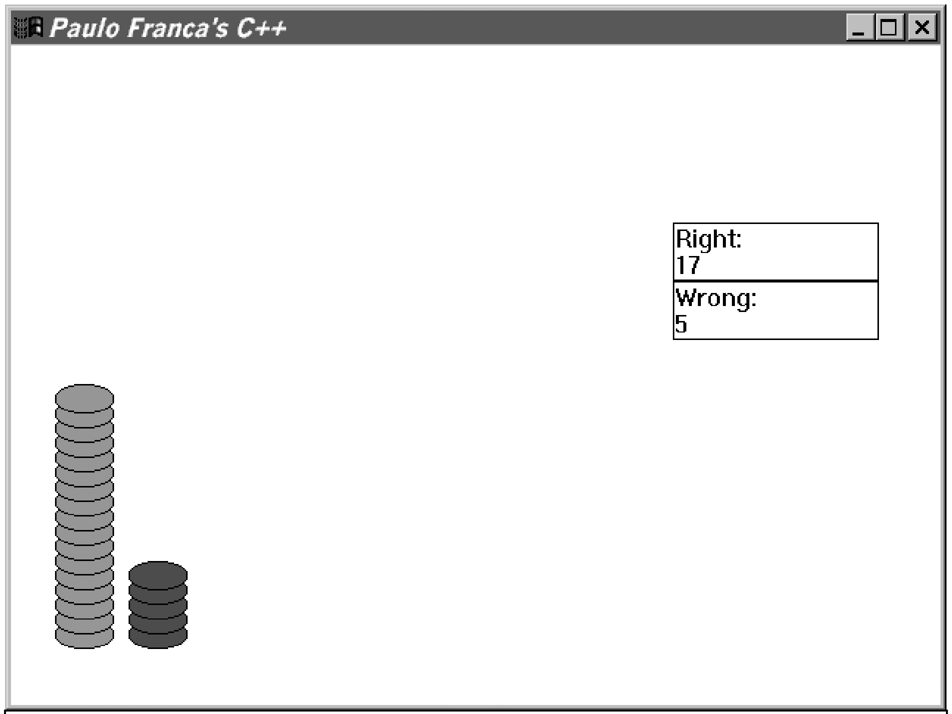

Fig 25.1 Result of a spelling session

Object-Oriented Implementation

The development of a software project using the object-oriented approach does not have to follow any specific rules. Nevertheless, some steps can assist you:

1. Never fear getting started. Explain how you think the problem can be solved. Review this explanation. Change it and expand it—do whatever you need to make it a good description of how to solve the problem.
2. Split the program into smaller pieces until you can comfortably deal with each piece. If you try to employ reusable pieces, either you will use pieces that you already know or you will build pieces in the hope they can be reused.
    - Remember that any time something is reused, you save a lot of time and, presumably, have a more reliable solution. You must identify classes of objects that are available and that can be used in your project. You must also identify classes of objects that may be useful, even if they are not available—you can develop new classes of your own.
    - Allow some generality and room for expansion when you design your new classes. Do not solve the problem that you face today. On the fly, prepare to solve the problems that you may face in the future.

3. Write the class declarations once you have a good idea of the classes you will use.
   - The class declarations establish how other pieces of software interact with the classes. You can write the code for the functions later.
4. Work on the algorithm of the software that uses objects of the classes that you selected.
   - At this point, you can write the code for the program.
5. Write the class definitions—the actual code for the member functions.

However, software development is not a sequential process. Often, you may need to review some previous steps. Whatever you do to write more reliable and readable code is worth it. It will cost you much more to fix something once the software is on the market!

Developing the Project

We can use the project description that was provided as our initial step. We may try to identify useful objects that we can develop to solve the problem. Some objects available in `franca.h`, such as boxes and Screen Objects, will definitely be useful. These objects, of course, will not be enough. Should we identify new classes of objects that we can use?

If we examine the problem and try to split it into reusable pieces (according to step 2), we may end up with some objects that, when they are put together, will handle the problem.

If we imagine how this situation would work in real life, we may picture a teacher testing a student. The teacher has a list of words, which have possibly come from a dictionary, and she dictates these words to the student.

The resulting code for this project is included in `c9spell.cpp`. The sound files are assumed to be in the subdirectory `sounds`.

Working with Words

This imagining could give us the idea to use an object of the class `teacher` that would read the words and administer the tests. The essential operations that the teacher would perform are as follows:

- Display all the words in the test.
- Pronounce all the words in the test.
- Administer the test.

The list of words will come from somewhere, possibly from a dictionary. It would be interesting to allow the teacher to draw words from different lists. We could also devise an object that is a mix of a dictionary and a jukebox, since we have recorded sounds. The jukebox keeps a list of words (both in written format and in sound format), and should be able to

- Pronounce a given word
- Spell a word
- Give the number of words available in the list

The teacher will, of course, use this jukebox to retrieve the spelling and to pronounce each word.

It may prove very useful in the future to implement a class of `jukebox` objects. There may be several situations in which we could use an object that can pronounce words kept in a list.

Since we will often deal with character arrays of size 9, the following type definition will be useful:

```
 typedef char words[9];
```

Consider the following declaration for this class:

```
class jukebox
{
   protected:
     char directory[30],filename[40];// Stores directory path
                                     //      and full path
     int wordnumber;
     words list[40];
     void whatfile(words name);      // Generates file name
   public:
     int howmanywords();
     void pronounce(words thisword);
     void pronounce(int whichword);
     void spell(int which,words spelled);
     jukebox(char path[]);
     jukebox();
     int find(words whichone);
};
```

Notice there are two member functions with the name `pronounce`. One of them takes the word in a character array and pronounces it (by playing the file with the same name). The other one takes an integer that is an index to the list of words. The function `howmanywords()` simply gives the number of words available in the list.

The function `spell()` has an argument that is an index to the list of words. This function retrieves the character array that contains the spelling of the word and then copies it into the second argument.

As you can see, given the integer value, let's use *k*, the teacher can pronounce a word that is in position *k* in the list and can also pronounce the same word using the `jukebox` member functions.

Using Constructors

In addition, there are two constructors for `jukebox`, which initialize it. The argument to one of the constructors is a character array containing the path to the directory where the sound files are recorded. For example, `c:\\franca\\sounds\\` (remember that, in a literal string, backslashes are represented by double backslashes). This string prefixes the file name when `jukebox` tries to play a sound. It is important to include the parameterless constructor in this case, because the teacher declares `jukebox` without parameters. If this constructor is not available, `jukebox` could not be created in this case.

☞ If the sound files are located in the subdirectory `sounds` of the current directory, the path can be `sounds\\`.

The function `whatfile()` puts together the directory path, the file, and the suffix to give a string a complete file name. This complete file name is stored in the character array `filename`. This function is not public, because we intend to make it available only for the other member functions.

Instead of declaring private members, I declare protected members, so that any descendent classes can access them without needing to update the source code. If we suppose objects of the class `jukebox` are available, we can define the `teacher` class to use `jukebox`.

Declaring Classes

Objects of the class `teacher` should be able to write and play each word in the list and to administer the test. Teachers may need a clock to keep track of time, and may also need a place to write the words and scores. We could consider also creating an object such as a blackboard, where the teacher could write these things. However, the available Box objects could serve just as well.

As data members, teacher will have three boxes: yourword (to display the spelling of a word), plus (to display the number of correct answers), and wrong (to display the number of incorrect answers). A jukebox is also needed, so that the teacher can operate with the words.

The *teacher* class declaration

Here is the declaration for the class teacher:

```
class teacher
{
  bank FirstNational;
  Box yourword;
  Clock myclock;
  Box plus,minus;
  jukebox speaker;
  int right,wrong;
  public:
   void playlist();
   void test();
   teacher(jukebox &neon);
};
```

The constructor for teacher has an argument that is a jukebox. This means that a teacher must have a jukebox to work! Why is the argument passed as a reference? Just to avoid an unnecessary copy.

How about the data member that happens to be a bank? It is an idea that came up to help students visualize their performance. I thought of displaying green and red coins to illustrate progress. Since we may need similar tools in other educational software, I thought it would be a good idea to use a class of bank objects. The bank simply builds a pile of green coins (right) and a pile of red coins (wrong).

The *bank* class declaration

Here is the declaration for the class bank:

```
class bank
{
  protected:
   Circle greencoin,redcoin;
   float xgreen,xred;
   float nextygreen,nextyred;
  public:
   void plus();
   void minus();
   bank();
};
```

The plus() function adds a green coin, and the minus() function adds a red coin. A constructor is needed to create the coins and the initial location when the object is started.

The bank would be more reusable if we could specify the location where the coins can be piled. If you use default parameters, you will be allowed to omit the coordinates if you want.

The prototype for the constructor can then be changed to

```
bank( float x=50,float y=400);
```

Once the member functions become available, the main program can be as simple as this:

```
void mainprog()
{
   jukebox speaker("sounds\\");
   teacher MrRoberts(speaker);
   MrRoberts.playlist();
   MrRoberts.test();
   speaker.pronounce("allright");
   speaker.pronounce("byebye");
}
```

Pay attention to the declarations for the `jukebox` and `teacher` objects. The `jukebox` is informed of the path to follow to find the sound files. This path is stored in the `jukebox,` so that all sounds are retrieved from that directory.

The `teacher` object has one argument, which is the `jukebox` we just created. This argument makes sure that this `teacher` will use this particular `jukebox` to administer the test.

Storing Words in Separate Directories

Finally, a couple of words, *alright* and *bye-bye*, are pronounced to indicate that the test is completed. It is clear that these words must be available in digital form in the same directory to which the jukebox `speaker` was assigned. What if you want to keep these words in a different place? After all, you may want to perform spelling tests with different sets of words, and it would be a great idea to have one set in a separate directory. Do we need to have a copy of `allright` and `byebye` in all these directories?

Of course not! Suppose you want the test words in the directory `c:\franca\sounds\test1\` and the greetings in the directory `c:\franca\sounds\`. What can you do?

It is simple. All you need to do is to create two jukeboxes: one associated with the words to test and another associated with the greetings:

```
jukebox speaker("c:\\franca\\test1\\");
jukebox greeting("c:\\franca\\sounds1\\");
```

The `speaker` jukebox will be attached to the `teacher`, as before. The `greeting` jukebox will be used instead of `speaker` in the last couple of statements to pronounce the words *alright* and *bye-bye*.

📖 In the current implementation, the `teacher` also pronounces the word *alright* when the student spells a word correctly. This uses the same jukebox, and assumes that this word is in the same directory as the other words.

Defining Classes

Now that we have settled on which objects to use and specified all the functions in the declarations, we can define the functions for each of the classes. Still, at this stage, we are going to manually load the list of words in the jukebox.

The *jukebox* class definition

Here is the definition of the class `jukebox`:

```
jukebox::jukebox()
{
    wordnumber=0;
    directory[0]=0;
}
jukebox::jukebox(char path[])
{
     fstream wordlist;
     if(strlen(directory)>30)directory[0]=0;
     strcpy(directory,path);         // Directory
     strcpy(filename,directory);
     strcat(filename,"list");
     strcat(filename,".txt");
     wordlist.open(filename,ios::in|ios::nocreate);// File name
     if(fileopen(wordlist)==0)
     {                               // Check whether file is available
       Box errormsg("Error:");
       errormsg.say("No List!");
       exit(0);
     }
     for (int k=0;k<100;k++)
     {                               // Read words
        if(wordlist.eof()) break;
        wordlist>>list[k];
     }
     wordnumber=k;                   // Number of words
}
int jukebox::howmanywords()
{
   return wordnumber;
}
void jukebox::spell(int which,words spelled)
{
     if (which>=wordnumber)
     {
       strcpy(spelled,"");           // Restrict size!
       return;
     }
     strcpy(spelled,list[which]);    // Copy to spelled
}

void jukebox::whatfile(words list)
{
   strcpy(filename,directory);       // Prefix
   strcat(filename,list);            // Name
   strcat(filename,".wav");          // Suffix
}
void jukebox::pronounce(int whichword)
{
   whatfile(list[whichword]);
   sound(filename);
}

void jukebox::pronounce(words thisword)
{
   whatfile(thisword);
   sound(filename);
}
```

You should choose either `#define Microsoft` or `#define Borland` so that the appropriate test for determining whether a file was opened successfully can be applied. This definition must be right at the beginning of the program.

The *bank* class definition

Here is the definition for the class `bank`:

```
bank::bank(float x,float y)
{
   xgreen=x;
   xred=x+50;
   nextygreen=y;
   nextyred=y;
   greencoin.resize(20,40);
   greencoin.color(2,7);
   redcoin.resize(20,40);
   redcoin.color(1,7);
}

void bank::plus()
{
   greencoin.place(xgreen,nextygreen);
   greencoin.show();
   nextygreen=nextygreen-10;
}

void bank::minus()
{
   redcoin.place(xred,nextyred);
   redcoin.show();
   nextyred=nextyred-10;
}
```

The *teacher* class definition

Finally, here is the definition for the class `teacher`:

```
teacher::teacher(jukebox &neon)
{
  speaker=neon;                      // Copy to teacher's jukebox
  right=wrong=0;
  plus.label("Right:");
  plus.say(right);
  minus.label("Wrong:");
  minus.say(wrong);
}
```

```
void teacher::playlist()
{
  words oneword;
  int n=speaker.howmanywords();
  for (int k=0;k<n;k++)
   {
     speaker.spell(k,oneword);
     yourword.say(oneword);
     speaker.pronounce(k);
     myclock.wait(2);
     yourword.erase();
   }
}

void teacher::test()
{
  char spelling[20];                  // Student's word
  char correct[20];                   // Correct word
  int j;
  int count=2*speaker.howmanywords();// Number of questions
  for(int k=1;k<=count;k++)
  {
    j=rand()%speaker.howmanywords();
    strcpy(spelling,"");              // Clear display
    speaker.pronounce(j);
    askwords(spelling,9,"Please spell:");
    speaker.spell(j,correct);
    if(stricmp(spelling,correct)==0)
    {
       speaker.pronounce("allright");
       plus.say(++right);
       FirstNational.plus();
    }
    else
    {
       yesno(" Wrong! Will you study harder?");
       minus.say(++wrong);
       FirstNational.minus();
    }
  }
}
```

Are You Experienced?

Now you can…

Approach application development in an object-oriented way

Reuse software

Understand more of the design issues involved in developing an application

Implement a simple object-oriented application

SKILL

TWENTY-SIX

INTRODUCING THE MISSING C++ SKILLS

- Understanding what else there is in C++
- Understanding what else there is in computer programming

The main goal of this book was to teach you how to solve problems using a computer and the C++ programming language. As you progress in your career, you will notice that it is far more important to devise algorithms than it is to memorize programming recipes.

The concepts you learn while developing algorithms will help you even if you need to program in another language.

To keep things simple for your learning process, some features of C++ were not covered in this beginner's book. As you develop more programs, you may want to find a more advanced text or to experiment with advanced features by yourself.

In this last Skill, some of these missing features are summarized. Also, there is much more to learn about computer programming than what you can learn in one programming book. Issues that may interest you for future learning are also mentioned.

What Else Is There in C++?

This book does not cover all the features of the C++ programming language. Several features were deliberately omitted. In this section, we will briefly discuss some of them.

Pointers

Pointers are variables that hold the addresses of other variables in the computer's memory. There are several applications of pointers in C++. However, you can deal with some of these applications by using arrays (which, in fact, use pointers) and by passing parameters by reference. In C, it is not possible to pass a parameter by reference, and, for this reason, it is extremely important to know how to deal with pointers.

If you use pointers, it will require a lot of attention and may result in hard-to-read code.

AN APPLICATION OF POINTERS

An interesting application of pointers can be found in the implementation of the class `Stage` in `franca.h`. An object of class `Stage` is a Screen Object (derived from `ScreenObj`) that has an array of pointers to Screen Objects. Every time you insert an object in a `Stage` object, the address of this new object is copied into the array. For example, a call to the `show()` member function simply loops through the whole array and shows each object to which is pointed. A similar thing happens with the other member functions.

Combined Operators

There is a special set of assignment operators that you can use when updating a variable. They are as follows:

```
+=
–=
*=
/=
%=
```

For example, you can use

```
x+=5;
```

instead of

```
x=x+5;
```

If you use these operators, you may indeed save a few keyboard strokes. There is nothing much to gain otherwise, and the readability of your program may suffer. Nevertheless, they are very popular constructs that you may find in other programs, and they are not difficult to understand.

Operator Overloading

One of the most interesting features allowed by C++, but not covered in this book, is *operator overloading*. It is possible to define the behavior of any of the operators used in C++. For example, the operator + is defined in C++ to operate with integers and floating point numbers. It is not defined for any other type or class of variables.

You can define classes of your own: strings, complex numbers, coordinates, matrices, or even screen objects. If you do this, you can define a member function `operator+` that will explain how the operator + will be used when one (or both) operands are objects of the new classes. This allows us to operate with strings.

```
if (stringA == stringB) stringA="a copy";
```

will work if you overload the operators == and = to operate with the string class you defined.

An Example of Operator Overloading

In `franca.h`, you can insert a Screen Object into a `Stage` object by invoking the `insert()` member function. For example:

```
Stage soldier;
soldier.insert(rightarm);
soldier.insert(leftarm);
Or, you can use the << operator:
soldier<<rightarm;
soldier<<leftarm;
```

This is because a member function, such as

```
Stage::operator<<(ScreenObj &something)
```

is also defined to belong to the `Stage` class.

The `operator` functions explain what should be done if the particular operator (in this case, <<) shows up between an object of the given class (in this case, `Stage`) and an object of the class defined for the parameter (in this case, `ScreenObj`). In practice, all this function does is call the `insert` member function.

Templates

Templates are another useful C++ feature. In the same way that you can make your functions more versatile by using arguments, you can make your classes more versatile by using *templates*.

A SIMPLE APPLICATION OF TEMPLATES

The `textfile` class had a restriction that the ID field had to be an alphanumeric string of up to 40 characters. The ID field was declared to be part of the class:

```
class textfile

{

   ...

  public:

   char id[40];
```

If you want to have an ID field whose size you can determine for each program, you can use a template:

```
template<int size>

class textfile

{
   ...

   char id[size];
```

Then, in your program, declare your class:

```
textfile myfile<65>;
```

This would work as if you had made your classes using the constant *65* to declare the size of the ID field.

However, the power of templates goes far beyond this simple example. Not only can you change the value of a number, you can also change the type or class of the data with which you are working. Therefore, instead of having ID as a type `char`, you could have it as any generic type:

```
...

anytype id;
```

Keep in mind, however, that the work done by templates is limited to substituting what is in the template for what you provide in the class declaration. Since character arrays are compared with `strcmp` and values are compared with the relational operators, you may still have some work left.

Templates are very powerful tools to build reusable software, since you can define classes in a generic way and customize them to a specific need.

Friend Functions and Classes

Well, who said C++ is not friendly? At least, it is the only programming language that I know that allows "friends" in the software. *Friends* are honorary members of a class. In a class declaration, class *A* can specify that class *B* is a `friend`. If class *B* is a `friend` class, all the member functions of class *B* can access private and protected data members of class *A*. A similar effect happens when a class declares a function to be a `friend`.

Multiple Inheritance

A class can be derived from more than one base class. In this case, data members and member functions of both base classes will be inherited by the derived class.

Memory Allocation

Throughout this book, we have dealt with only one mechanism to allocate the computer's memory to accommodate our variables and objects. This mechanism was the so-called *automatic allocation*. If you use this mechanism, the variable is created, and space is allocated when the program reaches the point where the variable is declared. When the variable goes out of scope, its contents are discarded, and the space is made available for other variables.

There are other ways to allocate memory space. Static variables remain in existence even when they go out of scope.

It is also possible, by using pointers, to dynamically allocate space for a variable during program execution. Suppose you need an array, but you do not know beforehand how many elements the array will use. Instead of allocating a maximum number, you can start the program, and only after the program figures out how many elements are needed, your program requests the computer to allocate the space.

Logical Operators

Special operators are available to perform the logical operations *and*, *or*, and *not* with corresponding pieces of variables. `&&` stands for *and*, `||` stands for *or*, and `!` stands for *not*.

What Else Is There in Computer Programming?

There is no way to learn how to ride a bike by just using a blackboard. If you simply learn all the features of a programming language, it will not make you a great programmer. Knowing how to organize your data, carefully examining the most adequate algorithms, and developing good programming habits are key elements for a successful programmer.

Use your computer, and program! Be curious, and experiment!

Software Engineering Principles

One can still argue whether software development is mostly science or engineering. One may even argue that it involves art, as well! No matter which view you adopt, you must agree that there is a substantial amount of engineering-like requirements when developing software.

It is important that you meet deadlines, budgets, and performance specifications. It is no use to develop the most wonderful software in the world if it is not available when it is needed, or if no one can afford to buy it!

There are several principles that were established to help software development become more efficient. A slightly more difficult problem for you to understand is the problem of software maintenance. I bet you would rather be given programs to develop than be given programs to fix that somebody else developed. Am I right?

Nevertheless, as a programmer, you may spend most of your professional life maintaining software, instead of developing it. Software has to be maintained not only because it was wrong, but also in response to external events—the market has changed, the laws have changed, the problem has changed….

If the software is easy to maintain, the maintenance costs will be lower. How do we make software easy to maintain?

It is not a simple question. You may have noticed, though, that readable code, reuse of software components, and good documentation help a lot. When you develop your software, if you can foresee some of the possible needs for modifications, you can make your software easier to maintain.

Are You Experienced?

Now you can…

Understand what else there is in C++

Understand what else there is in computer programming