9 Simple use of files

9.1 DEALING WITH MORE DATA

Realistic examples of programs with loops and selection generally have to work with largish amounts of data. Programs need different data inputs to test the code for all their different selections. It is tiresome to have to type in large amounts of data every time a program is tested. Also, if data always have to be keyed in, input errors become more likely. Rather than have the majority of the data entered each time the program is run, the input data can be keyed into a text file once. Then, each time the program is subsequently run on the same data, the input can be taken from this file.

While some aspects of file use have to be left to later, it is worth introducing simple ways of using files. Many of the later example programs will be organized as shown in Figure 9.1.

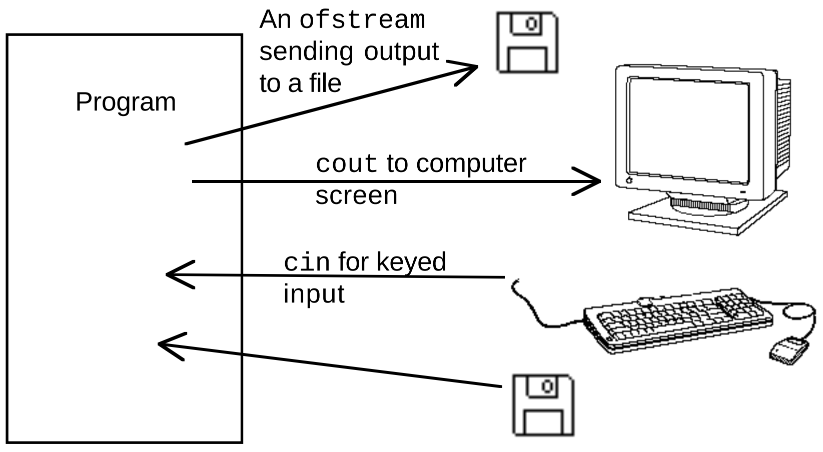

Figure 9.1 Programs using file i/o to supplement standard input and output.

These programs will typically send most of their output to the screen, though some may also send results to output files on disks. Inputs will be taken from file or, in some cases, from both file and keyboard. (Input files can be created with the editor part of the integrated development environment used for programming, or can be created using any word processor that permits files to be saved as "text only".)

Text files

Input files (and any output files) will be simply text files. Their names should be chosen so that it obvious that they contain data and are not "header" (".h") or "code" (".cp") files. Some environments may specify naming conventions for such data files. If there are no prescribed naming schemes, then adopt a scheme where data files have names that end with ".dat" or ".txt".

Binary files

Data files based on text are easy to work with; you can always read them and change them with word processors etc. Much later, you will work with files that hold data in the internal binary form used in programs. Such files are preferred in advanced applications because, obviously, there is no translation work (conversion from internal to text form) required in their use. But such files can not be read by humans.

"Redirection of input and output".

Some programming environments, e.g. the Unix environment, permit inputs and outputs to be "redirected" to files. This means that:

1) you can write a program that uses `cin` for input, and `cout` for output, and test run it normally with the input taken from keyboard and have results sent to your screen,

then

2) you can subsequently tell the operating system to reorganize things so that when the program needs input it will read data from a file instead of trying to read from the keyboard (and, similarly, output can be routed to a file instead of being sent to the screen).

Such "redirection" is possible (though a little inconvenient) with both the Borland Symantec systems. But, instead of using `cin` and `cout` redirected to files, all the examples will use explicitly declared "filestream" objects defined in the program.

The program will attach these filestream objects to named files and then use them for input and output operations.

9.2 DEFINING FILESTREAM OBJECTS

If a program needs to make explicit connections to files then it must declare some "filestream" objects.

fstream.h header file

The header file fstream.h contains the definitions of these objects. There are basically three kinds:

- `ifstream` objects --- these can be used to read data from a named file;

- `ofstream` objects --- these can be used to write data to a named file;

and

- `fstream` objects --- these are used with files that get written and read (at different times) by a program.

The examples will mainly use `ifstream` objects for input. Some examples may have `ofstream` objects.

The `fstream` objects, used when files are both written and read, will not be encountered until much later examples (those dealing with files of record structures).

A program that needs to use filestreams must include both the iostream.h and fstream.h header files:

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

int main()
{
       ...;
```

ifstream objects

`ifstream` objects are simply specialized kinds of input stream objects. They can perform all the same kinds of operations as done by the special `cin` input stream, i.e. they can "give" values to integers, characters, doubles, etc. But, in addition, they have extra capabilities like being able to "open" and "close" files.

ofstream objects

Similarly, `ofstream` objects can be asked to do all the same things as `cout` – print the values integers, doubles etc – but again they can also open and close output files.

The declarations in the file fstream.h make `ifstream`, `ofstream`, and `fstream` "types". Once the fstream.h header file has been included, the program can contain definitions of variables of these types.

Often, filestream variables will be globals because they will be shared by many different routines in a program; but, at least in the first few examples, they will be defined as local variables of the main program.

There are two ways that such variables can be defined.

Defining a filestream attached to a file with a known fixed name

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

int main()
{
    ifstream   input("theinput.txt", ios::in);
    ...
```

Here variable `input` is defined as an input filestream attached to the file called theinput.txt (the token `ios::in` specifies how the file will be used, there will be more examples of different tokens later). This style is appropriate when the name of the input file is fixed.

The alternative form of definition is:

Defining a filestream that will subsequently be attached to a selected file

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

int main()
{
    ifstream   in1;
    ...
```

This simply names `in1` as something that will be an input filestream; `in1` is not attached to an open file and can not be used for reading data until an "open" operation is performed naming the file. This style is appropriate when the name of the input file will vary on different runs of the program and the actual file name to be used is to be read from the keyboard.

The open request uses a filename (and the `ios::in` token). The filename will be a character string; usually a variable but it can be a constant (though then one might as well have used the first form of definition). With a constant filename, the code is something like:

Call to open()

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

void main()
{
    ifstream   in1;
    ...
    in1.open("file1.txt", ios::in);
    // can now use in1 for input ...
    ...
```

(Realistic use of `open()` with a variable character string has to be left until arrays and strings have been covered.)

9.3 USING INPUT AND OUTPUT FILESTREAMS

Once an input filestream variable has been defined (and its associated file opened either implicitly or explicitly), it can be used for input. Its use is just like `cin`:

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

void main()
{
    ifstream   input("theinput.txt", ios::in);
    long l1, l2; double d3; char ch;
    ...
    input >> d3; // read a double from file
    ...
    input >> ch; // read a character
    ...
    input >> l1 >> l2; // read two long integer
```

Similarly, `ofstream` output files can be used like `cout`:

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

void main()
{
    // create an output file
    ofstream out1("results.txt", ios::out);
    int i; double d;
    ...
    // send header to file
    out1 << "The results are :" << endl;      ...
    for(i=0;i<100; i++) {
           ...
           // send data values to file
           out1 << "i : " << i << ",    di " << d << endl;
           ...
           }
    out1.close(); //finished, close the file.
```

This code illustrates "close", another of the extra things that a filestream can do that a simple stream cannot:

```
out1.close();
```

Programs should arrange to "close" any output files before finishing, though the operating system will usually close any input files or output files that are still open when a program finishes.

9.4 STREAM' STATES

All stream objects can be asked about their state: "Was the last operation successful?", "Are there any more input data available?", etc. Checks on the states of streams are more important with the filestreams that with the simple `cin` and `cout` streams (where you can usually see if anything is wrong).

It is easy to get a program to go completely wrong if it tries to take more input data from a filestream after some transfer operations have already failed. The program may "crash", or may get stuck forever in a loop waiting for data that can't arrive, or may continue but generate incorrect results.

Operations on filestreams should be checked by using the normal stream functions `good()`, `bad()`, `fail()`, and `eof()`. These stream functions can be applied to any stream, but just consider the example of an `ifstream`:

```
ifstream in1("indata.txt", ios::in);
```

The state of stream `in1` can be checked:

```
in1.good()         returns "True" (1) if in1 is OK to use
in1.bad()          returns "True" (1) if last operation on
```

```
                   in1 failed and there is no way to recover
                   from failure
in1.fail()         returns "True" (1) if last operation on
                   in1 failed but recovery may be possible
in1.eof()          returns "True" (1) if there are no more
                   data to be read.
```

If you wanted to give up if the next data elements in the file aren't two integers, you could use code like the following:

```
ifstream in1("indata.txt", ios::in);
long       val1, val2;
in1>> val1 >> val2;

if(in1.fail()) {
    cout << "Sorry, can't read that file" << endl;
    exit(1);
    }
```

(There are actually two status bits associated with the stream – the badbit and the failbit. Function `bad()` returns true if the badbit is set, `fail()` returns true if either is set.)

Naturally, this being C++ there are abbreviations. Instead of phrasing a test like:

```
if(in1.good())
    …
```

your can write:

```
if(in1)
    …
```

and similarly you may substitute

```
if(!in1)
```

for

```
if(in1.bad())
```

Most people find these particular abbreviated forms to be somewhat confusing so, even though they are legal, it seems best not to use them. Further, although these behaviours are specified in the iostream header, not all implementations comply!

One check should always be made when using ifstream files for input. *Was the file opened?*

There isn't much point continuing with the program if the specified data file isn't there.

The state of an input file should be checked immediately after it is opened (either implicitly or explicitly). If the file is not "good" then there is no point in continuing.

A possible way of coding the check for an OK input file should be as follows:

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

int main()
{
    ifstream in1("mydata.txt", ios::in);
    switch(in1.good()) {
case 1:
    cout << "The file is OK, the program continues" << endl;
    break;
case 0:
    cout << "The file mydata.txt is missing, program"
                  " gives up"  << endl;
        exit(1);
     }
    ...
```

It is a pity that this doesn't work in all environments.

Implementations of C++ aren't totally standardized. Most implementations interpret any attempt to open an input file that isn't there as being an error. In these implementations the test `in1.good()` will fail if the file is non-existent and the program stops.

Unfortunately, some implementations interpret the situation slightly differently. If you try to open an input file that isn't there, an empty file is created. If you check the file status, you are told its OK. As soon as you attempt to read data, the operation will fail.

The following style should however work in all environments:

```
int main()
{
    ifstream in1("mydata.txt", ios::in | ios::nocreate);
    switch(in1.good()) {
           …
```

The token combination ios::in | ios::nocreate specifies that an input file is required and it is *not* to be created if it doesn't already exist.

9.5 OPTIONS WHEN OPENING FILESTREAMS

You specify the options that you want for a file by using the following tokens either individually or combination:

ios::in	*Open for reading.*
ios::out	*Open for writing.*
ios::ate	*Position to the end-of-file.*
ios::app	*Open the file in append mode.*
ios::trunc	*Truncate the file on open.*
ios::nocreate	*Do not attempt to create the file if it*

```
                         does not exist.
ios::noreplace    Cause the open to fail if the file exists.
ios::translate    Convert CR/LF to newline on input and
                         vice versa on output.
```

(The translate option may not be in your selection; you may have extras, e.g. `ios::binary`.) Obviously, these have to make sense, there is not much point trying to open an ifstream while specifying ios::out!

Typical combinations are:

```
ios::in | ios::nocreate         open if file exists,
                                       fail otherwise
ios::out | ios::noreplace       open new file for output,
                                       fail if filename
                                       already exists
ios::out | ios::ate             (re)open an existing output
                                       file, arranged so
                                       that new data added
                                       at the end after
                                       existing data
ios::out | ios::noreplace | ios::translate
                                open new file, fail if name
                                       exists, do newline
                                       translations
ios::out | ios::nocreate | ios::trunc
                                open an existing file for
                                       output, fail if file
                                       doesn't exist, throw
                                       away existing
                                       contents and start
                                       anew
```

"Appending" data to an output file, `ios::app`, might seem to mean the same as adding data at the end of the file, `ios::ate`. Actually, `ios::app` has a specialized meaning – writes always occur at the end of the file irrespective of any subsequent positioning commands that might say "write here rather than the end". The `ios::app` mode is really intended for special circumstances on Unix systems etc where several programs might be trying to add data to the same file. If you simply want to write some extra data at the end of a file use `ios::ate`.

"Bitmasks"

The tokens `ios::in` etc are actually constants that have a single bit set in bit map. The groupings `like ios::open | ios::ate` build up a "bitmask" by bit-oring together the separate bit patterns. The code of fstream's `open()` routines checks the individual bit settings in the resulting bit mask, using these to select processing options. If you want to combine several bit patterns to get a result, you use an bit-or operation, operator `|`.

Don't go making the following mistakes!

```
ofstream   out1("results.dat", ios::out || ios::noreplace);
ifstream   in1("mydata.dat", ios::in & ios::nocreate);
```

The first example is wrong because the boolean or operator, `||`, has been used instead of the required bit-or operator `|`. What the code says is "If either the constant `ios::out` or `ios::noreplace` is non zero, encode a one bit here". Now both these constants are non zero, so the first statement really says `outl("results.dat",1)`. It may be legal C++, but it sure confuses the run-time system. Fortuitously, code 1 means the same as `ios::in.` So, at run-time, the system discovers that you are opening an output file for reading. This will probably result in your program being terminated.

The second error is a conceptual one. The programmer was thinking "I want to specify that it is an input file and it is not to be created". This lead to the code `ios::in & ios::nocreate`. But as explained above, the token combinations are being used to build up a bitmask that will be checked by the `open()` function. The bit-and operator does the wrong combination. It is going to leave bits set in the bitmask that were present in both inputs. Since `ios::in` and `ios::nocreate` each have only one bit, a different bit, set the result of the `&` operation is 0. The code is actually saying `in1("mydata.dat", 0)`. Now option 0 is undefined for `open()` so this isn't going to be too useful.

9.6 WHEN TO STOP READING DATA?

Programs typically have loops that read data. How should such loops terminate?

Sentinel data

The sentinel data approach was described in section 8.5.1. A particular data value (something that cannot occur in a valid data element) is identified (e.g. a 0 height for a child). The input loop stops when this sentinel value is read. This approach is probably the best for most simple programs. However, there can be problems when there are no distinguished data values that can't be legal input (the input requirement might be simply "give me a number, any number").

Count

The next alternative is to have, as the first data value, a count specifying how many data elements are to be processed. Input is then handled using a `for` loop as follows:

```
int         num;
ifstream    in1("indata.txt", ios::in);
…
// read number of entries to process
in1 >> num;
for(int i = 0; i < num; i++) {
    // read height and gender of child
    char gender_tag;
    double height;
    cin >> height >> gender_tag;
    …
    }
```

The main disadvantage of this approach is that it means that someone has to count the number of data elements!

eof()

The third method uses the `eof()` function for the stream to check for "end of file". This is a sort of semi-hardware version of a sentinel data value. There is conceptually a mark at the end of a file saying "this is the end". Rather than check for a particular data value, your code checks for this end of file mark.

This mechanism works well for "record structured files", see Figure 9.2A. Such files are explained more completely in section 17.3. The basic idea is obvious. You have a file of records, e.g. "Customer records"; each record has some number of characters allocated to hold a name, a double for the amount of money owed, and related information. These various data elements are grouped together in a fixed size structure, which would typically be a few hundred bytes in size (make it 400 for this example). These blocks of bytes would be read and written in single transfers. A record would consist of several successive records.

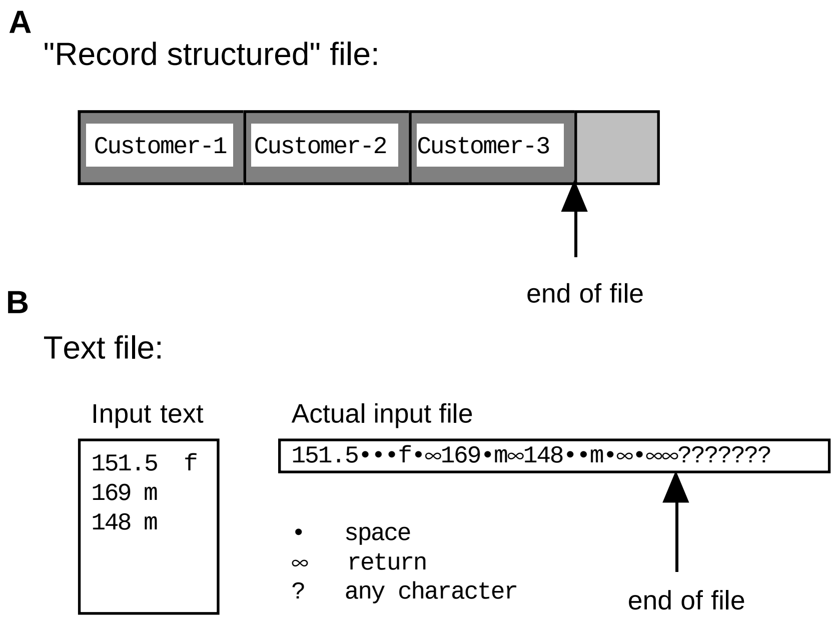

Figure 9.2 Ends of files.

Three "customer records" would take up 1200 bytes; bytes 0…399 for Customer-1, 400…799 for Customer-2 and 800…1199 for Customer-3. Now, as explained in Chapter 1, file space is allocated in blocks. If the blocksize was 512 bytes, the file would be 1536 bytes long. Directory information on the file would specify that the "end of file mark" was at byte 1200.

You could write code like the following to deal with all the customer records in the file:

```
ifstream   sales("Customers.dat", ios::in | ios::nocreate);
if(!sales.good()) {
    cout << "File not there!" << endl;
    exit(1);
    }
while(! sales.eof()) {
    read and process next record
    }
```

As each record is read, a "current position pointer" gets advanced through the file. When the last record has been read, the position pointer is at the end of the file. The test `sales.eof()` would return true the next time it is checked. So the while loop can be terminated correctly.

The `eof()` scheme is a pain with text files. The problem is that text files will contain trailing whitespace characters, like "returns" on the end of lines or tab characters or spaces; see Figure 9.2B. There may be no more data in the file, but because there are trailing whitespace characters the program will not have reached the end of file.

If you tried code like the following:

```
ifstream   kids("theHeights.dat", ios::in | ios::nocreate);
if(!kids.good()) {
    cout << "File not there!" << endl;
    exit(1);
    }
while(! kids.eof()) {
    double height;
    char gender_tag;
    kids >> height >> gender_tag;
    …
    }
```

You would run into problems. When you had read the last data entry (148 m) the input position pointer would be just after the final 'm'. There remain other characters still to be read – a space, a return, another space, and two more returns. The position marker is not "at the end of the file". So the test `kids.eof()` will return false and an attempt will be made to execute the loop one more time.

But, the input statement `kids >> height >> gender_tag`; will now fail – there are no more data in the file. (The read attempt will have consumed all remaining characters so when the failure is reported, the "end of file" condition will have been set.)

You can hack around such problems, doing things like reading ahead to remove "white space characters" or "breaking" out of a loop if an input operation fails and sets the end of file condition.

But, really, there is no point fighting things like this. Use the `eof()` check on record files, use counts or sentinel data with text files.

9.7 MORE FORMATTING OPTIONS

The example in section 8.5.1 introduced the `setprecision()` format manipulator from the iomanip library. This library contains some additional "manipulators" for changing output formats. There are also some functions in the standard iostream library than can be used to change the ways output and input are handled.

You won't make much use of these facilities, but there are odd situations where you will need them.

The following iomanip manipulators turn up occasionally:

```
setw(int)         sets a width for the next output
setfill(int)      changes the fill character for next output
```

along with some manipulators defined in the standard iostream library such as

```
hex        output number in hexadecimal format
dec        output number in decimal format
```

The manipulators setprecision(), hex, and dec are "sticky". Once you set them they remain in force until they are reset by another call. The width manipulators only affect the next operation. Fill affects subsequent outputs where fill characters are needed (the default is to print things using the minimum number of characters so that the filler character is normally not used).

The following example uses several of these manipulators:

```
int main()
{
    int           number = 901;

    cout << setw(10) << setfill('#') << number << endl;
    cout << setw(6) << number << endl;

    cout << dec << number << endl;
    cout << hex << number << endl;
    cout << setw(12) << number << endl;
    cout << setw(16) << setfill('@') << number << endl;

    cout << "Text" << endl;
    cout << 123 << endl;

    cout << setw(8) << setfill('*') << "Text" << endl;

    double d1 = 3.141592;
    double d2 = 45.9876;
    double d3 = 123.9577;

    cout << "d1 is " <<  d1 << endl;

    cout << "setting precision 3 " << setprecision(3)
              << d1 << endl;
    cout << d2 << endl;
    cout << d3 << endl;
    cout << 4214.8968 << endl;

    return EXIT_SUCCESS;
}
```

The output produced is:

```
#######901        901, 10 spaces, # as fill
###901            901, 6 spaces, continue # fill
901               just print decimal 901
385               print it as a hexadecimal number
#########385      hex, 12 spaces, # still is filler
```

```
@@@@@@@@@@@@@385  change filler
Text              print in minimum width, no filler
7b                still hex output!
****Text          print text, with width and fill
d1 is 3.141592    fiddle with precision on doubles
setting precision 3 3.142
45.988
123.958
4.215e+03
```

Alternative form of specification

There are alternative mechanisms that can be used to set some of the format options shown above. Thus, you may use

```
cout.width(8)     instead of    cout << setw(8)
cout.precision(4) ..            cout << setprecision(4)
cout.fill('$')    ..            cout << setfill('$')
```

Other strange format options

There are a number of other output options that can be selected. These are defined by more of those tokens (again, these are actually defined bit patterns); they include:

```
ios::showpoint    (printing of decimal point and trailing 0s)
ios::showpos      (require + sign with positive numbers)
ios::uppercase    (make hex print ABCDE instead of abcde)
```

These options are selected by telling the output stream object to "set flags" (using its `setf()` function) and are deselected using `unsetf()`. The following code fragment illustrates their use:

```
int main()
{
    long   number = 8713901;

    cout.setf(ios::showpos);
    cout << number << endl;
    cout.unsetf(ios::showpos);
    cout << number << endl;
    cout << hex << number << endl;
    cout.setf(ios::uppercase);
    cout << number << endl;
    return EXIT_SUCCESS;
}
```

The code should produce the output:

```
+8713901   positive sign forced to be present
8713901    normally it isn't shown
84f6ad     usual hex format
84F6AD     format with uppercase specified
```

Two related format controls are:

```
ios::left
```

```
ios::right
```

these control the "justification" of data that is printed in a field of specified width:

```
int main()
{
    cout.width(24);
    cout.fill('!');
    cout << "Hello" << endl;
    cout.setf(ios::left, ios::adjustfield);
    cout.width(24);
    cout << "Hello" << endl;
    cout.setf(ios::right, ios::adjustfield);
    cout.width(24);
    cout << "Hello" << endl;

    return EXIT_SUCCESS;
}
```

should give the output:

```
!!!!!!!!!!!!!!!!!!!Hello
Hello!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!Hello
```

Note the need to repeat the setting of the field width; unlike fill which lasts, a width setting affects only the next item. As shown by the output, the default justification is right justified. The left/right justification setting uses a slightly different version of the `setf()` function. This version requires two separate data items – the left/right setting and an extra "adjustfield" parameter.

You can specify whether you want doubles printed in "scientific" or "fixed form" styles by setting:

```
ios::scientific
ios::fixed
```

as illustrated in this code fragment:

```
int main()
{
    double number1 = 0.00567;
    double number2 = 1.746e-5;
    double number3 = 9.43214e11;
    double number4 = 5.71e93;
    double number5 = 3.08e-47;

    cout << "Default output" << endl;

    cout << number1 << ", " << number2 << ", "
                << number3 << endl;
    cout << "      " << number4 << ", " << number5 << endl;

    cout << "Fixed style" << endl;
```

```
        cout.setf(ios::fixed, ios::floatfield);

        cout << number1 << ", " << number2 << ", "
                      << number3 << endl;
    //  cout << "      " << number4 ;
        cout << number5 << endl;

        cout << "Scientific style" << endl;
        cout.setf(ios::scientific, ios::floatfield);

        cout << number1 << ", " << number2 << ", "
                      << number3 << endl;
        cout << "      " << number4 << ", " << number5 << endl;

        return EXIT_SUCCESS;
        }
```

This code should produce the output:

```
Default output
0.00567, 1.746e-05, 9.43214e+11
      5.71e+93, 3.08e-47
Fixed style
0.00567, 0.000017, 943214000000
0
Scientific style
5.67e-03, 1.746e-05, 9.43214e+11
      5.71e+93, 3.08e-47
```

The required style is specified by the function calls `setf(ios::fixed, ios::floatfield)` and `setf(ios::scientific, ios::floatfield)`. (The line with the code printing the number 5.71e93 in "fixed" format style is commented out – some systems can't stand the strain and crash when asked to print such a large number using a fixed format. Very small numbers, like 3.08e-47, are dealt with by just printing zero.)

Maybe you should use stdio instead!

The various formatting options available with streams should suffice to let you layout any results as you would wish. Many programmers prefer to use the alternative stdio library when they have to generate tabular listings of numbers etc. You IDE environment should include documentation on the stdio library; its use is illustrated in all books on C.

Changing options for input streams

You don't often want to change formatting options on input streams. One thing you might occasionally want to do is set an input stream so that you can actually read white space characters (you might need this if you were writing some text processing program and needed to read the spaces and return characters that exist in an input file). You can change the input mode by unsetting the ios::skipws option, i.e.

```
cin.unsetf(ios::skipws);
```

By default, the `skipws` ("skip white space") option is switched on. (You may get into tangles if you unset `skipws` and then try to read a mixture of character and numeric data.)

The following program illustrates use of this control to change input options:

```
int main()
{
/*
Program to count the number of characters preceding a
period '.' character.
*/
    int    count = 0;
    char ch;

//  cin.unsetf(ios::skipws);
    cin >> ch;
    while(ch != '.') {
                  count++;
                  cin >> ch;
           }
    cout << endl;
    cout << "I think I read " << count << " characters."
                  << endl;
    return EXIT_SUCCESS;
}
```

Given the input:

```
Finished, any questions? OK, go; wake that guy in the back row,
he must have been left here by the previous lecturer.
```

the output would normally be:

```
I think I read 95 characters.
```

but if the `cin.unsetf(ios::skipws);` statement is included, the output would be:

```
I think I read 116 characters.
```

because the spaces and carriage returns will then have been counted as well as the normal letters, digits, and punctuation characters.

9.8 EXAMPLE

Problem

Botanists studying soya bean production amassed a large amount of data relating to different soya plantations. These data had been entered into a file so that they could be analyzed by a program.

Some soya plantations fail to produce a harvestable crop. The botanists had no idea of the cause of the problem – they thought it could be disease, or pest infestation, or poor nutrition, or maybe climatic factors. The botanists started by recording information on representative plants from plots in each of the plantations.

Each line of the file contained details of one plant. These details were encoded as 0 or 1 integer values. The first entry on each line indicated whether the plant was rated healthy (1) or unhealthy (0) at harvest time. Each of the other entries represented an attribute considered in the study. These attributes included "late spring frosts", "dry spring", "beetles", "nitrogen deficient soil", "phosphate deficient soil", etc; there were about twenty attributes in the study (for simplicity we will reduce it to five).

The file contains hundreds of lines with entries. Examples of individual entries from the file were:

```
0   1      0      0      0      0
```

(an unhealthy plant that had experienced a late spring frosts but no other problems), and

```
1   1      0      1      0      0
```

(a plant that was healthy despite both a late spring frost and beetles in summer).

The botanists wanted these data analyzed so as to identify any correlations between a plant's health and any chosen one of the possible attributes.

Consider a small test with results for 100 plants. The results might show that 45 were unhealthy, and that beetles had been present for 60 of them. If we assume that the presence of beetles is unrelated to plant health, then there should be a simple random chance distribution of beetles on both healthy and the unhealthy plants. So here, 60% of the 45 diseased plants (i.e. 27 plants) should have had beetle infestations as should 33 of the healthy plants.

```
Expected distribution
                  beetles       clean
unhealthy         27            18            = 45
healthy           33            22            = 55
                  60            40
```

While it is unlikely that you would see exactly this distribution, if health and beetle presence are unrelated then the observed results should be something close to these values.

The actual observed distribution might be somewhat different. For example one might find that 34 of the unhealthy plants had had lots of beetles:

```
Observed distribution
                  beetles       clean
unhealthy         34            11            = 45
healthy           26            29            = 55
                  60            40
```

Results such as these would suggest that beetles, although certainly not the sole cause of ill health, do seem to make plants less healthy.

Of course, 100 samples is really quite small; the differences between expected and observed results could have arisen by chance. Larger differences between observed and expected distributions occur with smaller chance. If the differences are very large, they are unlikely to have occurred by chance. It does depend on the number of examples studied; a smallish difference may be significant if it is found consistently in very large numbers of samples.

Statisticians have devised all sorts of formulae for working out "statistics" that can be used to estimate the strength of relationships among values of different variables. The botanists' problem is one that can be analyzed using the χ^2 statistic (pronounced ≈ "kiye-squared") This statistic gives a single number that is a measure of the overall difference between observed and expected results and which also takes into account the size of the sample.

The χ^2 statistic is easily computed from the entries in the tables of observed and expected distributions:

$$\chi^2 = \sum_{i=1}^{4} (O_i - E_i)^2 / E_i$$

O_i	value in observed distribution
E_i	value in expected distribution
Σ	sum is over the four entries in the table

For the example above,

$$\chi^2 = (34-27)^2/27 + (11-18)^2/18 + (26-33)^2/33 + (29-22)^2/22$$

$$\approx 8$$

If the observed values were almost the same as the expected values ($O_i \approx E_i$) then the value of χ^2 would be small or zero. Here the "large" value of χ^2 indicates that there is a significant difference between expected and observed distributions of attribute values and class designations. It is very unlikely that such a difference could have occurred solely by chance, i.e. there really is some relation between beetles and plant health.

You can select a chance level at which to abandon the assumption that an attribute (e.g. beetle presence) and a class (e.g. unhealthy) are unrelated. For example, you could say that if the observed results have a less than 1 in 20 chance of occurring then the assumption of independence should be abandoned. You could be more stringent and demand a 1 in 100 chance. Tables exist that give a limit value for the χ^2 statistic associated with different chance levels. In a case like this, if the χ^2 statistic exceeds ≈3.8 then an assumption of independence can be abandoned provided that you accept that a 1 in 20 chance as too unlikely. Someone demanding a 1 in 100 chance would use a threshold of ≈5.5.

The botanists would be able to get an idea of what to focus on by performing a χ^2 analysis to measure the relation between plant health and each of the attributes for which they had recorded data. For most attributes, the observed distributions would turn out to be pretty similar to the expected distributions and the χ^2 values would be small (0…1). If any of the attributes had a large χ^2 value, then the botanists would know that they had identified something that affected plant health.

Specification:

1. The program is to process the data in the file "plants.dat".

   This file contains several hundred lines of data. Each line contains details of one plant. The data for a plant take the form of six integer values, all 0 or 1;. The first is the plant classification – unhealthy/healthy. The remaining five are the values recorded for attributes of that plant.

   This data value is terminated by a sentinel. This consist of a line with a single number, -1, instead of a class and set of five attributes.

2. The program is to start by prompting for and reading input from `cin`. This input is to be an integer in the range 1 to 5 which identifies the attribute that is to be analyzed in this run of the program.

3. The program is to then read all the data records in the file, accumulating the counts that provide the values for the observed distribution and overall counts of the number of healthy plants etc.

4. When all the data have been read, the program is to calculate the values for the expected distribution.

5. The program is to print details of both expected and observed distributions; these should be printed in tabular form with data entries correctly aligned.

6. Finally, the program is to calculate and print the value of the χ^2 statistic.

Program design

Preliminary design

A first iteration through the design process gives a structure something like the following:

```
open the data file for input
prompt for and get number identify attribute of interest

read the first class value (or a sentinel if someone
    is trying to trick you with an empty file!)

initialize all counters

while class != sentinel
    read all five attribute values
    select value for the attribute of interest
```

```
        increment appropriate counters
        read next class value or sentinel

    calculate four values for expected distribution

    print tables of observed and expected distributions

    calculate  χ²

    print  χ²
```

Second iteration through design process

There are several points in the initial design outline that require elaboration.

What counters are needed and how can the "observed" and "expected" tables be represented? How can the "attribute of interest" be selected? How should an "appropriate counter" be identified and updated? All of these issues must be sorted out before the complete code is written. However, it will often be the case that the easiest way to record a design decision will be to sketch out a fragment of code.

Each of the tables really needs four variables:

```
Observed                                Expected
                  attribute                    attribute
                  0      1                     0      1
unhealthy 0       Obs00  Obs01                 Exp00  Exp01
healthy 1         Obs10  Obs11                 Exp10  Exp11
```

Three additional counters are needed to accumulate the data needed to calculate the "expected" values `Exp00 … Exp11`; we need a count of the total number of plants, a count of healthy plants, and a count for the number of times that the attribute of interest was true.

Selecting the attribute of interest is going to be fairly easy. The value (1…5) input by the user will have been saved in some integer variable `interest`. In the while loop, we can have five variables `a1 … a5`; these will get values from the file using an input statement like:

```
infile >> a1 >> a2 >> a3 >> a4 >> a5;
```

We can have yet another variable, `a`, which will get the value of the attribute of interest. We could set a using a switch statement:

```
    switch(interest) {
case 1: a = a1; break;
…
case 5: a = a5; break;
           }
```

(Use of arrays would provide an alternative mechanism for holding the values and selecting the one of interest.)

The values for variables `a` and `pclass` (plant's 0/1 class value) have to be used to update the counters. There are several different ways that these update operations could be coded.

Select an approach that makes the meaning obvious even if this results in somewhat wordier and less efficient code than one of the alternatives. The updating of the counters is going to be a tiny part of the overall processing cost (most time will be devoted to conversion of text input into internal numeric data). There is no point trying to "optimize" this code and make it "efficient" because this is not the code where the program will spend its time.

The counts could be updated using "clever" code like:

```
// Update count of cases with attribute true
atotalpos += a;
healthy += pclass; // and count of healthy plants

Obs11 += pclass && a;
Obs10 += pclass && !a;
```

It should take you less than a minute to convince yourself that those lines of code are appropriate. But fifteen seconds thought per line of code? That code is too subtle. Try something cruder and simpler:

```
#define TRUE 1
…
atotalpos += (a == TRUE) ? 1 : 0;
healthy += (pclass == TRUE) ? 1 : 0;

Obs11 += ((pclass == TRUE) && (a == TRUE)) ? 1 : 0;
```

or even

```
if(a == TRUE) atotalpos++;
…
```

Data

Before implementation, you should also compose some test data for yourself. You will need to test parts of your program on a small file whose contents you know; there is no point touching the botanists' file with its hundreds of records until you know that your program is doing the right thing with data. You would have to create a file with ten to twenty records, say 12 healthy and 6 unhealthy plants. You can pick the attribute values randomly except for one attribute that you make true (1) mainly for the unhealthy plants.

Implementation

This program is just at the level of complexity where it becomes worth building in stages, testing each stage in turn.

First stage partial implementation

The first stage in implementation would be to create something like a program that tries to open an input file, stopping sensibly if the file is not present, otherwise continuing by simply counting the number of records.

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

```
#include <fstream.h>

int main()
{
    // Change file name to plants.dat when ready to fly
    ifstream infile("test.dat", ios::in | ios::nocreate);
    int    count = 0;

    if(!infile.good()) {
           cout << "Sorry. Couldn't open the file." <<endl;
           exit(1);
    }

    int pclass;

    infile >> pclass;

    while(pclass != -1) {
           count++;
           int a1, a2, a3, a4, a5;
           infile >> a1 >> a2 >> a3 >> a4 >> a5;
           // Discard those data values
           // Read 0/1 classification of next plant
           // or sentinel value -1
           infile >> pclass;
    }

    cout << "I think there were " << count
                  << " records in that file." << endl;
    return EXIT_SUCCESS;

}
```

Second stage partial implementation

With a basic framework running, you can start adding (and testing) the data processing code. First we could add the code to determine the attribute of interest and to accumulate some of the basic counts.

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

#define TRUE 1
#define FALSE 0

int main()
{
    // Change file name to plants.dat when ready to fly
    ifstream infile("test.dat", ios::in | ios::nocreate);
    int    count = 0;

    if(!infile.good()) {
           cout << "Sorry. Couldn't open the file." <<endl;
           exit(1);
    }

    int pclass;
```

```
       int atotalpos = 0;
       int healthy = 0;
       int interest;

       cout << "Please specify attribute of interest"
                             "(1...5) : ";
       cin >> interest;

       if((interest < 1) ||(interest >5)) {
              cout << "Sorry, don't understand, wanted"
                             " value 1..5" << endl;
              exit(1);
       }

       infile >> pclass;

       while(pclass != -1) {
              count++;
              if(pclass == TRUE)
                     healthy++;

              int a1, a2, a3, a4, a5;
              infile >> a1 >> a2 >> a3 >> a4 >> a5;
              int a;
              switch(interest) {
   case 1:           a = a1; break;
   case 2:           a = a2; break;
   case 3:           a = a3; break;
   case 4:           a = a4; break;
   case 5:           a = a5; break;
              }
              if(a == TRUE)
                     atotalpos++;
              infile >> pclass;
       }

       cout << "Data read for " << count << " plants." << endl;
       cout << healthy << " were healthy." << endl;
       cout << "Attribute of interest was " << interest << endl;
       cout << "This was true in " << atotalpos << " cases."
                     << endl;
       return EXIT_SUCCESS;

   }
```

Note the "sanity check" on the input. This is always a sensible precaution. If the user specifies interest in an attribute other than the five defined in the file there is really very little point in continuing.

The next sanity check is one that you should perform. You would have to run this program using the data in your small test file. Then you would have to check that the output values were appropriate for your test data.

Third stage partial implementation

The next stage in the implementation would be to i) define initialized variables that represent the entries for the "observed" and "expected" tables, ii) add code inside the while loop to update appropriate "observed" variables, and iii) provide code that prints the "observed" table in a suitable layout. The variables for the "observed" variables could be integers, but it is simpler to use doubles for both observed and expected values. Getting the output in a good tabular form will necessitate output formatting options to set field widths etc. You will almost certainly find it necessary to make two or three attempts before you get the layout that you want.

These extensions require variable definitions just before the while loop:

```
double Obs00, Obs01, Obs10, Obs11;
double Exp00, Exp01, Exp10, Exp11;
Obs00 = Obs01 = Obs10 = Obs11 = Exp00 = Exp01 =
           Exp10 = Exp11 = 0.0;
```

Code in the while loop:

```
if((pclass == FALSE) && (a == FALSE)) Obs00++;
if((pclass == FALSE) && (a == TRUE)) Obs01++;
if((pclass == TRUE) && (a == FALSE)) Obs10++;
if((pclass == TRUE) && (a == TRUE)) Obs11++;
```

Output code like the following just after the while loop:

```
cout.setf(ios::fixed, ios::floatfield);

cout << "Observed distribution" << endl;
cout << "               Attribute " << interest << endl;
cout << "                0       1" << endl;
cout << "Unhealthy" << setw(8) << setprecision(1) << Obs00;
cout << setw(8) << Obs01;
cout << endl;
cout << "Healthy  " << setw(8) << Obs10 <<setw(8) << Obs11;
cout << endl;
```

Note that the use of `setprecision()` and `setw()` requires the iomanip library, i.e. #include <iomanip.h>. You might also chose to use set option `ios::showpoint` to get a value like 7 printed as 7.0.

Final stage of implementation

Finally, you would add the code to compute the expected distributions, print these and calculate the χ^2 statistic. There is a good chance of getting a bug in the code to calculate the expected values:

buggy version

```
Exp10 = healthy*(count-atotalpos)/count;
Exp11 = healthy*atotalpos/count;
```

The formulae are correct in principle. Value `Exp11` is the expected value for the number of healthy plants with the attribute value also being positive. So if, as in the example, there are 55 healthy plants, and of the 100 total plants 60 have beetles, we should get 55 * 60/100 or 33 as value for `Exp11.` If you actually had those numeric values, there would be no problems and `Exp11` would get the value 33.0. However, if 65 of the 100 plants had had beetles, the calculation would be 55*65/100 with the correct result 35.75. But this wouldn't be the value assigned to `Exp11`; instead it would get the value 35.

All of the variables `healthy`, `atotalpos`, and `count` are integers. So, by default the calculation is done using integer arithmetic and the fractional parts of the result are discarded.

You have to tell the compiler to code the calculation using doubles. The following code is OK:

```
Exp10 = double(healthy*(count-atotalpos))/count;
Exp11 = double(healthy*atotalpos)/count;
```

bug fixed

The code to print the expected distribution can be obtained by "cutting and pasting" the working code that printed the observed distribution and then changing variable names.

The final few statements needed to calculate the statistic are straightforward:

```
double chisq;
chisq = (Obs00 - Exp00)*(Obs00-Exp00)/Exp00;
chisq += (Obs01 - Exp01)*(Obs01-Exp01)/Exp01;
chisq += (Obs10 - Exp10)*(Obs10-Exp10)/Exp10;
chisq += (Obs11 - Exp11)*(Obs11-Exp11)/Exp11;

cout << "Chisquared value " << chisq << endl;
```

All the variables here are doubles so that there aren't the same problems as for calculating `Exp11` etc. You could make these calculations slightly more efficient by introducing an extra variable to avoid repeated calculation of similar expressions:

```
double chisq;
double temp;
temp = (Obs00 - Exp00);
chisq = temp*temp/Exp00;
temp = (Obs01 - Exp01);
chisq += temp*temp/Exp01;
```

But honestly, it isn't worth your time bothering about such things. The chances are pretty good that your compiler implements a scheme for spotting "common subexpressions" and it would be able to invent such temporary variables for itself.

You would have to run your program on your small example data set and you would need to check the results by hand. For a data set like the following:

```
1 0 0 0 0 0
1 0 1 0 1 0
1 1 0 0 0 1
1 0 1 1 0 1
1 0 1 0 1 1
1 0 1 1 0 1
1 1 0 0 1 0
1 0 1 0 1 0
1 0 0 1 1 0
1 0 1 0 0 1
1 0 0 1 0 1
1 0 0 0 0 1
```

```
0 1 0 1 0 1
0 0 1 0 0 0
0 1 0 1 1 0
0 1 0 1 0 1
0 1 1 0 1 1
0 0 0 1 1 0
0 1 1 0 1 0
-1
```

test runs of the program should produce the following outputs:

```
Observed distribution
               Attribute 1
                0       1
Unhealthy       2       5
Healthy        10       2
Expected distribution
               Attribute 1
                0       1
Unhealthy     4.4     2.6
Healthy       7.6     4.4
Chisquared value 5.7
```

and

```
Observed distribution
               Attribute 4
                0       1
Unhealthy       3       4
Healthy         7       5
Expected distribution
               Attribute 4
                0       1
Unhealthy     3.7     3.3
Healthy       6.3     5.7
Chisquared value 0.4
```

Once you had confirmed that your results were correct, you could start processing the real file with the hundreds of data records collected by the botanists.

With larger programs, the only way that you can hope to succeed is to complete a fairly detailed design and then implement stage by stage as illustrated in this example.

EXERCISES

1. Modify the bank account program, exercise 3 in Chapter 8, so that it reads transaction details from a file "trans.dat".

2. The file "temperatures.dat" contains details of the oral temperatures of several hundred pre-med, public health, and nursing students. Each line of the file has a number (temperature in degrees Centigrade) and a character ('m' or 'f') encoding the gender of the student. The file ends with a sentinel record (0.0 x). Example:

```
37.3      m
36.9      f
37.0      m
…
37.1      f
0.0       x
```

Write a program to process these data. The program is to calculate and print:

mean (average) temperature, and standard deviation, for all students,
mean and standard deviation for male students,
mean and standard deviation for female students.

minimum and maximum temperatures for all students;
minimum and maximum temperatures for male students;
minimum and maximum temperatures for female students.

A preliminary study on a smaller sample gave slightly different mean temperatures for males and females. A statistical test is to be made on this larger set of data to determine whether this difference is significant.

This kind of test involves calculation of the "Z" statistic:

Let
μM = average temperature of males
μF = average temperature of females

SM = standard deviation of temperature of males
SF = standard deviation of temperature of females

NM = number of males
NF = number of females

SEM = square of standard error of temperature of males = $SM*SM \div NM$
SEF = square of standard error of temperature of females = $SF*SF \div NF$

$Z = (\mu M - \mu F) \div \sqrt{}(SEM + SEF)$

The statistic Z should be a value with a mean of 0 and a standard deviation of 1.

If there is no significant difference in the two means, then values for Z should be small; there is less than one chance in twenty that you would find a Z with an absolute value exceeding 2.

If a Z value greater than 2 (or less than -2) is obtained, then the statistic suggests that one should NOT presume that the male and female temperatures have the same mean.

Your program should calculate and print the value of this statistic and state whether any difference in means is significant.