This document is part of the HTML publication "**An Introduction to the Imperative Part of C++**"

The original version was produced by **Rob Miller** at **Imperial College London**, September 1996.

Version 1.1 (modified by **David Clark** at **Imperial College London**, September 1997)

Version 1.2 (modified by **Bob White** at **Imperial College London**, September 1998)

Version 1.3, 1.4, 2.0, ..., 2.15 (modified by **William Knottenbelt** at **Imperial College London**, September 1999-September 2016)

4. Files and Streams

4.1. Why Use Files?

All the programs we have looked at so far use input only from the keyboard, and output only to the screen. If we were restricted to use only the keyboard and screen as input and output devices, it would be difficult to handle large amounts of input data, and output data would always be lost as soon as we turned the computer off. To avoid these problems, we can store data in some secondary storage device, usually magnetic tapes or discs. Data can be created by one program, stored on these devices, and then accessed or modified by other programs when necessary. To achieve this, the data is packaged up on the storage devices as data structures called *files*.

The easiest way to think about a file is as a linear sequence of characters. In a simplifed picture (which ignores special characters for text formatting) these lecture notes might be stored in a file called "Lecture_4" as:

Figure 4.1.1

(BACK TO COURSE CONTENTS)

4.2 Streams

Before we can work with files in C++, we need to become acquainted with the notion of a *stream*. We can think of a stream as a channel or conduit on which data is passed from senders to receivers. As far as the programs we will use are concerned, streams allow travel in only one direction. Data can be sent out from the program on an *output stream*, or received into the program on an *input stream*. For example, at the start of a program, the standard input stream "`cin`" is connected to the keyboard and the standard output stream "`cout`" is connected to the screen.

In fact, input and output streams such as "`cin`" and "`cout`" are examples of (stream) *objects*. So learning about streams is a good way to introduce some of the syntax and ideas behind the object-oriented part of C++. The header file which lists the operations on streams both to and from files is called "fstream". We will therefore assume that the program fragments discussed below are embedded in programs containing the "include" statement

```
#include<fstream>
```

As we shall see, the essential characteristic of stream processing is that data elements must be sent to or received from a stream one at a time, i.e. in *serial* fashion.

Creating Streams

Before we can use an input or output stream in a program, we must "create" it. Statements to create streams look like variable declarations, and are usually placed at the top of programs or function implementations along with the variable declarations. So for example the statements

```
ifstream in_stream;
ofstream out_stream;
```

respectively create a stream called "`in_stream`" belonging to the *class* (like type) "`ifstream`" (input-file-stream), and a stream called "`out_stream`" belonging to the class "`ofstream`" (output-file-stream). However, the analogy between streams and ordinary variables (of type "`int`", "`char`", etc.) can't be taken too far. We cannot, for example, use simple assignment statements with streams (e.g. we can't just write "`in_stream1 = in_stream2`").

Connecting and Disconnecting Streams to Files

Having created a stream, we can connect it to a file using the *member function* "`open(...)`". (We have already come across some member functions for output streams, such as "`precision(...)`" and "`width(...)`", in Lecture 2.) The function "`open(...)`" has a different effect for ifstreams than for ofstreams (i.e. the function is polymorphic).

To connect the ifstream "`in_stream`" to the file "Lecture_4", we use the following statement:

```
in_stream.open("Lecture_4");
```

This connects "`in_stream`" to the beginning of "Lecture_4". Diagramatically, we end up in the following situation:

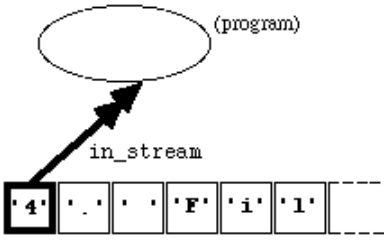

Figure 4.2.1

To connect the ofstream "`out_stream`" to the file "Lecture_4", we use an analogous statement:

```
out_stream.open("Lecture_4");
```

Although this connects "`out_stream`" to "Lecture_4", it also deletes the previous contents of the file, ready for new input. Diagramatically, we end up as follows:

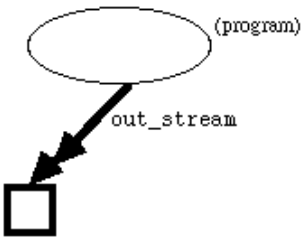

Figure 4.2.2

To disconnect connect the ifstream "in_stream" to whatever file it is connected to, we write:

```
in_stream.close();
```

Diagramatically, the situation changes from that of Figure 4.2.1 to:

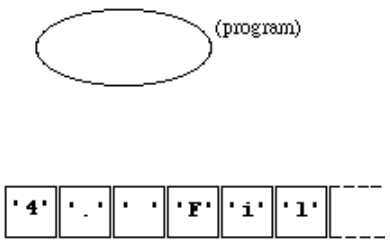

Figure 4.2.3

The statement:

```
out_stream.close();
```

has a similar effect, but in addition the system will "clean up" by adding an "end-of-file" marker at the end of the file. Thus, if no data has been output to "Lecture_4" since "out_stream" was connected to it, we change from the situation in Figure 4.2.2 to:

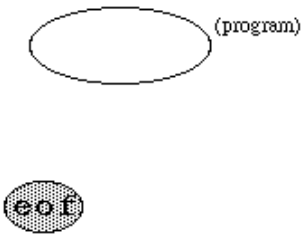

Figure 4.2.4

In this case, the file "Lecture_4" still exists, but is *empty*.

(BACK TO COURSE CONTENTS)

4.3 Checking for Failure with File Commands

File operations, such as opening and closing files, are a notorious source of errors. Robust commercial programs should always include some check to make sure that file operations have completed successfully, and error handling routines in case they haven't. A simple checking mechanism is provided by the member function "fail()". The function call

```
in_stream.fail();
```

returns True if the previous stream operation on "in_stream" was not successful (perhaps we tried to open a file which didn't exist). If a failure has occurred, "in_stream" may be in a corrupted state, and it is best not to attempt any more operations with it. The following example program fragment plays very safe by quitting the program entirely, using the "exit(1)" command from the library "cstdlib":

```
#include <iostream>
#include <fstream>
#include <cstdlib>

using namespace std;
```

```
int main()
{
        ifstream in_stream;

        in_stream.open("Lecture_4");
        if (in_stream.fail())
        {
                cout << "Sorry, the file couldn't be opened!\n";
                exit(1);
        }
        ...
```

(BACK TO COURSE CONTENTS)

4.4 Character Input and Output

Input using "`get(...)`"

Having opened an input file, we can extract or read single characters from it using the member function "`get(...)`". This function takes a single argument of type "`char`". If the program is in the state represented in Figure 4.2.1, the statement

```
in_stream.get(ch);
```

has two effects: (i) the variable "`ch`" is assigned the value "`'4'`", and (ii) the ifstream "`in_stream`" is re- positioned so as to be ready to input the next character in the file. Diagramatically, the new situation is:

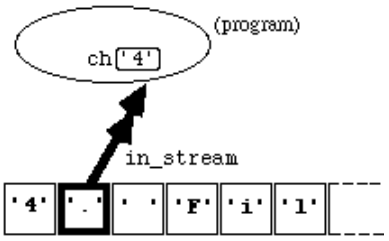

Figure 4.4.1

Output using "`put(...)`"

We can input or *write* single characters to a file opened via an ofstream using the member function "`put(...)`". Again, this function takes a single argument of type "`char`". If the program is in the state represented in Figure 4.2.2, the statement

```
out_stream.put('4');
```

changes the state to:

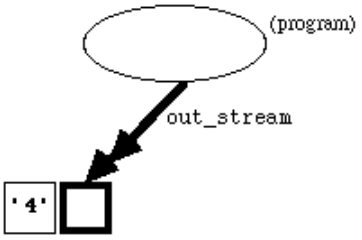

Figure 4.4.2

The "`putback(...)`" Function

C++ also includes a "putback(...)" function for ifstreams. This doesn't really "put the character back" (it doesn't alter the actual input file), but behaves as if it had. Diagramatically, if we started from the state in Figure 4.4.1, and executed the statement

```
in_stream.putback(ch);
```

we would end up in the state:

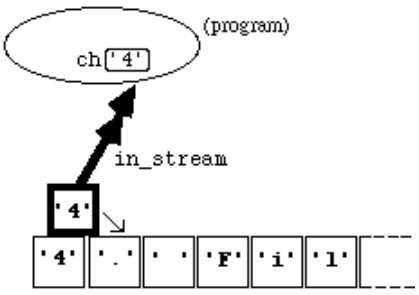

Figure 4.4.3

Indeed, we can "putback" any character we want to. The alternative statement

```
in_stream.putback('7');
```

would result in:

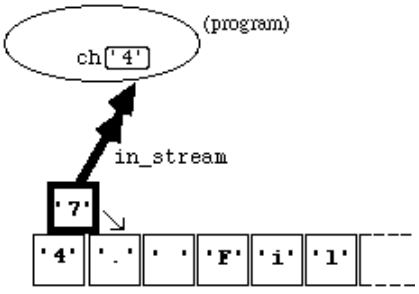

Figure 4.4.4

(BACK TO COURSE CONTENTS)

4.5 Checking for the End of an Input File

4.5.1 End-of-file Checking For Systems Which Implement "eof()"

Special care has to be taken with input when the end of a file is reached. Most versions of C++ (including GNU g++ and Microsoft Visual C++) incorporate an end-of-file (EOF) *flag*, and a member function called "eof()" for ifstreams to test if this flag is set to True or False. It's worth discussing such systems briefly, since many text books (including Savitch) assume this useful facility.

In such systems, when an ifstream is initially connected to a file, the EOF flag is set to False (even if the file is empty). However, if the ifstream "in_stream" is positioned at the end of a file, and the EOF flag is False, the statement

```
in_stream.get(ch);
```

will leave the variable "ch" in an unpredictable state, and set the EOF flag to True. Once the EOF flag is set to True, no attempt should be made to read from the file, since the results will be unpredictable. To illustrate with a diagramatic example, if we start from

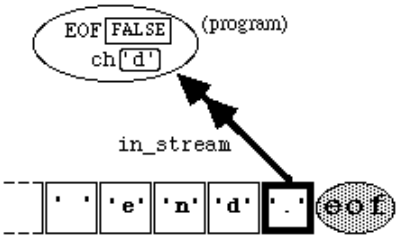

Figure 4.5.1

and then execute the statement

```
in_stream.get(ch);
```

this results in the state

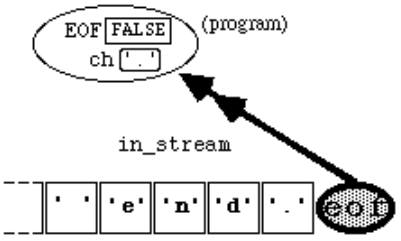

Figure 4.5.2

If we again execute the statement

```
in_stream.get(ch);
```

this now results in the state

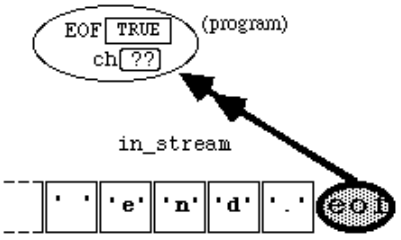

Figure 4.5.3

The boolean expression

```
in_stream.eof()
```

will now evaluate to True.

Below is a simple program which uses these techniques to copy the file "Lecture_4" simultaneously to the screen and to the file "Copy_of_4". Note the use of a *while loop* in this program. "While" loops are simplified versions of "for" loops, without the initialisation and update statements in the "()" parentheses (we will look at such statements again in the next lecture).

```
#include <iostream>
#include <fstream>

using namespace std;

int main()
{
        char character;
```

```
        ifstream in_stream;
        ofstream out_stream;

        in_stream.open("Lecture_4");
        out_stream.open("Copy_of_4");

        in_stream.get(character);
        while (!in_stream.eof())
        {
                cout << character;
                out_stream.put(character);
                in_stream.get(character);
        }

        out_stream.close();
        in_stream.close();

        return 0;
}
```

Program 4.5.1

(BACK TO COURSE CONTENTS)

4.6 Streams as Arguments in Functions

Streams can be arguments to functions, but must be reference parameters (not value parameters). Below is another version of Program 4.5.2 which uses the function "`copy_to(...)`".

```
#include <iostream>
#include <fstream>

using namespace std;

void copy_to(ifstream& in, ofstream& out);

/* MAIN PROGRAM: */
int main()
{
        ifstream in_stream;
        ofstream out_stream;

        in_stream.open("Lecture_4");
        out_stream.open("Copy_of_4");
        copy_to(in_stream, out_stream);
        out_stream.close();
        in_stream.close();

        return 0;
}
/* END OF MAIN PROGRAM */

/* FUNCTION TO COPY A FILE TO ANOTHER FILE AND TO THE SCREEN: */
void copy_to(ifstream& in, ofstream& out)
{
        char character;

        in.get(character);
        while (!in.eof())
        {
                cout << character;
                out.put(character);
                in.get(character);
```

```
            }
        }
        /* END OF FUNCTION */
```

Program 4.6.1

(BACK TO COURSE CONTENTS)

4.7 Input and Output Using ">>" and "<<"

So far we have only talked about writing and reading individual characters to and from files. At the lowest level, ofstreams and ifstreams only deal with files which are sequences of characters. So data of other types ("`int`", "`double`", etc.) has to be converted into character sequences before it can be written to a file, and these character sequences have to be converted back again when they are input.

However, the operators "`>>`" and "`<<`", which we have already met in the context of keyboard input and screen output, do some of this conversion automatically. For example, from the state

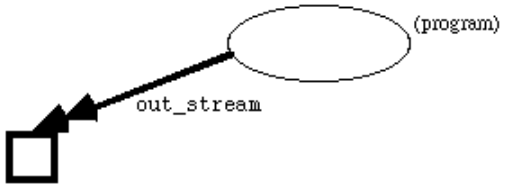

Figure 4.7.1

execution of the statement

```
out_stream << 437 << ' ';
```

will result in the new state

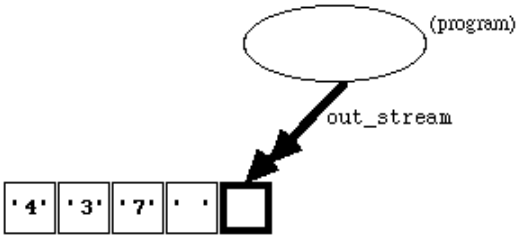

Figure 4.7.2

When using these higher level facilities, it is important to include at least one '` `' (blank) character (or a new-line character) after each item of data written. This ensures that the data items are correctly separated in the file, ready for input using "`>>`". For example, from the state

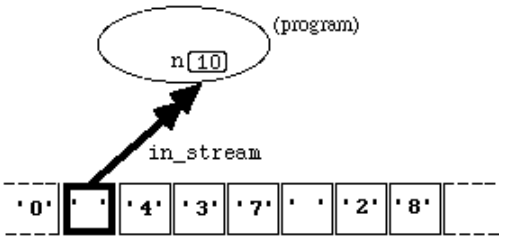

Figure 4.7.3

where "`n`" has been declared as of data type "`int`", execution of the statement

```
in_stream >> n;
```

will result in the new state

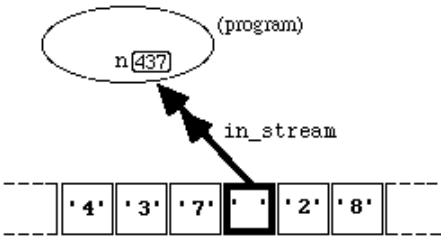

Figure 4.7.4

Notice that the operation ">>" has skipped over the blank space "' '" before the number 437. It always does this, no matter what data type it has been reading or expects to read (even characters).

The following program, which first creates a file called "Integers" containing the integers 51, 52, 53, 54 and 55, further illustrates these techniques.

```
#include <iostream>
#include <fstream>

using namespace std;

int main()
{
        char character;
        int number = 51;
        int count = 0;
        ofstream out_stream;
        ifstream in_stream1;   /* Stream for counting integers. */
        ifstream in_stream2;   /* Stream for counting characters. */

        /* Create the file */
        out_stream.open("Integers");
        for (count = 1 ; count <= 5 ; count++)
                out_stream << number++ << ' ';
        out_stream.close();

        /* Count the integers in the file */
        in_stream1.open("Integers");
        count = 0;
        in_stream1 >> number;
        while (!in_stream1.eof())
        {
                count++;
                in_stream1 >> number;
        }
        in_stream1.close();
        cout << "There are " << count << " integers in the file,\n";

        /* Count the non-blank characters */
        in_stream2.open("Integers");
        count = 0;
        in_stream2 >> character;
        while (!in_stream2.eof())
        {
                count++;
                in_stream2 >> character;
        }
        in_stream2.close();
        cout << "represented using " << count << " characters.\n";
```

```
        return 0;
}
```

Program 4.7.1

This program produces the following output:

```
There are 5 integers in the file,
represented using 10 characters.
```

Once again, notice that, unlike the function "`get(...)`", the operator "`>>`" has jumped over the blank (i.e. space) characters in the file (which separate the five integers) when used in counting characters in the last part of the program.

(BACK TO COURSE CONTENTS)

4.8 Summary

In this lecture we have seen how programs may be connected to external files for input and output using streams. We have seen how "low-level" character input and output can be achieved using the functions "`get(...)`" and "`put(...)`", and how "high-level" input and output of other kinds of data types can be achieved using the stream operators "`>>`" and "`<<`". The material here is also covered in more detail in Savitch, Chapter 6.

Exercises
