C++ Standard Library:

24. Input/Output with Files

C++ provides the following classes to perform output and input of characters to/from files:

- **ofstream:** Stream class to write on files
- **ifstream:** Stream class to read from files
- **fstream:** Stream class to both read and write from/to files.

These classes are derived directly or indirectly from the classes istream, and ostream. We have already used objects whose types were these classes: cin is an object of class istream and cout is an object of class ostream. Therfore, we have already been using classes that are related to our file streams. And in fact, we can use our file streams the same way we are already used to use cin and cout, with the only difference that we have to associate these streams with physical files.
Let's see an example:

```cpp
// basic file operations
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  ofstream myfile;
  myfile.open ("example.txt");
  myfile << "Writing this to a
file.\n";
  myfile.close();
  return 0;
}
```
```
[file example.txt]
Writing this to a file
```

This code creates a file called example.txt and inserts a sentence into it in the same way we are used to do with cout, but using the file stream myfile instead.

But let's go step by step:

Open a file

The first operation generally performed on an object of

one of these classes is to associate it to a real file. This procedure is known as to *open a file*. An open file is represented within a program by a stream object (an instantiation of one of these classes, in the previous example this was `myfile`) and any input or output operation performed on this stream object will be applied to the physical file associated to it.

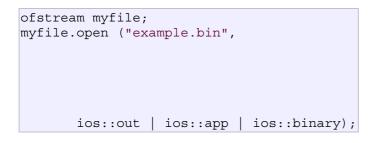In order to open a file with a stream object we use its member function `open()`:

```
open (filename, mode);
```

Where `filename` is a null-terminated character sequence of type `const char *` (the same type that string literals have) representing the name of the file to be opened, and `mode` is an optional parameter with a combination of the following flags:

ios::in	Open for input operations.
ios::out	Open for output operations.
ios::binary	Open in binary mode.
ios::ate	Set the initial position at the end of the file. If this flag is not set to any value, the initial position is the beginning of the file.
ios::app	All output operations are performed at the end of the file, appending the content to the current content of the file. This flag can only be used in streams open for output-only operations.
ios::trunc	If the file opened for output operations already existed before, its previous content is deleted and replaced by the new one.

All these flags can be combined using the bitwise operator OR (`|`). For example, if we want to open the file `example.bin` in binary mode to add data we could do it by the following call to member function `open()`:

```
ofstream myfile;
myfile.open ("example.bin",



       ios::out | ios::app | ios::binary);
```

Each one of the `open()` member functions of the classes `ofstream`, `ifstream` and `fstream` has a default mode that is used if the file is opened without a second argument:

class	default mode parameter
ofstream	ios::out
ifstream	ios::in
fstream	ios::in | ios::out

For `ifstream` and `ofstream` classes, `ios::in` and `ios::out` are automatically and respectively assumed, even if a mode that does not include them is passed as second argument to the `open()` member function.

The default value is only applied if the function is called without specifying any value for the mode parameter. If the function is called with any value in that parameter the default mode is overridden, not combined.

File streams opened in binary mode perform input and output operations independently of any format considerations. Non-binary files are known as *text files*, and some translations may occur due to formatting of some special characters (like newline and carriage return characters).

Since the first task that is performed on a file stream object is generally to open a file, these three classes include a constructor that automatically calls the `open()` member function and has the exact same parameters as this member. Therefor, we could also have declared the previous `myfile` object and conducted the same opening operation in our previous example by writing:

```
ofstream myfile ("example.bin",



    ios::out | ios::app | ios::binary);
```

Combining object construction and stream opening in a single statement. Both forms to open a file are valid and equivalent.

To check if a file stream was successful opening a file, you can do it by calling to member `is_open()` with no arguments. This member function returns a bool value of true in the case that indeed the stream object is associated with an open file, or false otherwise:

```
if (myfile.is_open())




    { /* ok, proceed with output */ }
```

Closing a file

When we are finished with our input and output operations on a file we shall close it so that its resources become available again. In order to do that we have to

call the stream's member function `close()`. This member function takes no parameters, and what it does is to flush the associated buffers and close the file:

```
myfile.close();
```

Once this member function is called, the stream object can be used to open another file, and the file is available again to be opened by other processes.

In case that an object is destructed while still associated with an open file, the destructor automatically calls the member function `close()`.

Text files

Text file streams are those where we do not include the `ios::binary` flag in their opening mode. These files are designed to store text and thus all values that we input or output from/to them can suffer some formatting transformations, which do not necessarily correspond to their literal binary value.

Data output operations on text files are performed in the same way we operated with `cout`:

```
// writing on a text file
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  ofstream myfile
("example.txt");
  if (myfile.is_open())
  {
    myfile << "This is a
line.\n";
    myfile << "This is another
line.\n";
    myfile.close();
  }
  else cout << "Unable to open
file";
  return 0;
}
```
```
[file example.txt]
This is a line.
This is another line.
```

Data input from a file can also be performed in the same way that we did with `cin`:

```cpp
// reading a text file
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main () {
  string line;
  ifstream myfile
("example.txt");
  if (myfile.is_open())
  {
    while (! myfile.eof() )
    {
      getline (myfile,line);
      cout << line << endl;
    }
    myfile.close();
  }

  else cout << "Unable to open
file";

  return 0;
}
```
```
This is a line.
This is another line.
```

This last example reads a text file and prints out its content on the screen. Notice how we have used a new member function, called `eof()` that returns true in the case that the end of the file has been reached. We have created a while loop that finishes when indeed `myfile.eof()` becomes true (i.e., the end of the file has been reached).

Checking state flags

In addition to `eof()`, which checks if the end of file has been reached, other member functions exist to check the state of a stream (all of them return a bool value):

bad()

> Returns true if a reading or writing operation fails. For example in the case that we try to write to a file that is not open for writing or if the device where we try to write has no space left.

fail()

> Returns true in the same cases as bad(), but also

in the case that a format error happens, like when an alphabetical character is extracted when we are trying to read an integer number.

eof()

Returns true if a file open for reading has reached the end.

good()

It is the most generic state flag: it returns false in the same cases in which calling any of the previous functions would return true.

In order to reset the state flags checked by any of these member functions we have just seen we can use the member function `clear()`, which takes no parameters.

get and put stream pointers

All i/o streams objects have, at least, one internal stream pointer:

`ifstream`, like `istream`, has a pointer known as the *get pointer* that points to the element to be read in the next input operation.

`ofstream`, like `ostream`, has a pointer known as the *put pointer* that points to the location where the next element has to be written.

Finally, `fstream`, inherits both, the get and the put pointers, from `iostream` (which is itself derived from both `istream` and `ostream`).

These internal stream pointers that point to the reading or writing locations within a stream can be manipulated using the following member functions:

tellg() and tellp()

These two member functions have no parameters and return a value of the member type `pos_type`, which is an integer data type representing the current position of the get stream pointer (in the case of `tellg`) or the put stream pointer (in the case of `tellp`).

seekg() and seekp()

These functions allow us to change the position of the get and put stream pointers. Both functions are overloaded with two different prototypes. The first prototype is:

```
seekg ( position );
seekp ( position );
```

Using this prototype the stream pointer is changed to the absolute position `position` (counting from the beginning of the file). The type for this parameter is the same as the

one returned by functions `tellg` and `tellp`: the member type `pos_type`, which is an integer value.

The other prototype for these functions is:

```
seekg ( offset, direction );
seekp ( offset, direction );
```

Using this prototype, the position of the get or put pointer is set to an offset value relative to some specific point determined by the parameter `direction`. `offset` is of the member type `off_type`, which is also an integer type. And `direction` is of type `seekdir`, which is an enumerated type (enum) that determines the point from where offset is counted from, and that can take any of the following values:

`ios::beg` offset counted from the beginning of the stream

`ios::cur` offset counted from the current position of the stream pointer

`ios::end` offset counted from the end of the stream

The following example uses the member functions we have just seen to obtain the size of a file:

```cpp
// obtaining file size
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  long begin,end;
  ifstream myfile
("example.txt");
  begin = myfile.tellg();
  myfile.seekg (0, ios::end);
  end = myfile.tellg();
  myfile.close();
  cout << "size is: " << (end-
begin) << " bytes.\n";
  return 0;
}
```
```
size is: 40 bytes.
```

Binary files

In binary files, to input and output data with the extraction and insertion operators (`<<` and `>>`) and functions like `getline` is not efficient, since we do not need to format any data, and data may not use the separation codes used by text files to separate elements (like space, newline, etc…).

File streams include two member functions specifically designed to input and output binary data sequentially: `write` and `read`. The first one (`write`) is a member function of `ostream` inherited by `ofstream`. And `read`

is a member function of `istream` that is inherited by
`ifstream`. Objects of class `fstream` have both members.
Their prototypes are:

```
write ( memory_block, size );
read ( memory_block, size );
```

Where `memory_block` is of type "pointer to char"
(`char*`), and represents the address of an array of
bytes where the read data elements are stored or from
where the data elements to be written are taken. The
`size` parameter is an integer value that specifies the
number of characters to be read or written from/to the
memory block.

```cpp
// reading a complete binary file
#include <iostream>
#include <fstream>
using namespace std;

ifstream::pos_type size;
char * memblock;

int main () {
  ifstream file ("example.txt",
ios::in|ios::binary|ios::ate);
  if (file.is_open())
  {
    size = file.tellg();
    memblock = new char [size];
    file.seekg (0, ios::beg);
    file.read (memblock, size);
    file.close();

    cout << "the complete file
content is in memory";

    delete[] memblock;
  }
  else cout << "Unable to open
file";
  return 0;
}
```
```
the complete file content is in
memory
```

In this example the entire file is read and stored in a
memory block. Let's examine how this is done:

First, the file is open with the `ios::ate` flag, which
means that the get pointer will be positioned at the end
of the file. This way, when we call to member `tellg()`,
we will directly obtain the size of the file. Notice the type
we have used to declare variable `size`:

```
ifstream::pos_type size;
```

`ifstream::pos_type` is a specific type used for buffer

and file positioning and is the type returned by `file.tellg()`. This type is defined as an integer type, therefore we can conduct on it the same operations we conduct on any other integer value, and can safely be converted to another integer type large enough to contain the size of the file. For a file with a size under 2GB we could use `int`:

```
int size;
size = (int) file.tellg();
```

Once we have obtained the size of the file, we request the allocation of a memory block large enough to hold the entire file:

```
memblock = new char[size];
```

Right after that, we proceed to set the get pointer at the beginning of the file (remember that we opened the file with this pointer at the end), then read the entire file, and finally close it:

```
file.seekg (0, ios::beg);
file.read (memblock, size);
file.close();
```

At this point we could operate with the data obtained from the file. Our program simply announces that the content of the file is in memory and then terminates.

Buffers and Synchronization

When we operate with file streams, these are associated to an internal buffer of type `streambuf`. This buffer is a memory block that acts as an intermediary between the stream and the physical file. For example, with an `ofstream`, each time the member function `put` (which writes a single character) is called, the character is not written directly to the physical file with which the stream is associated. Instead of that, the character is inserted in that stream's intermediate buffer.

When the buffer is flushed, all the data contained in it is written to the physical medium (if it is an output stream) or simply freed (if it is an input stream). This process is called *synchronization* and takes place under any of the following circumstances:

- **When the file is closed:** before closing a file all buffers that have not yet been flushed are synchronized and all pending data is written or read to the physical medium.
- **When the buffer is full:** Buffers have a certain

size. When the buffer is full it is automatically synchronized.

- **Explicitly, with manipulators:** When certain manipulators are used on streams, an explicit synchronization takes place. These manipulators are: `flush` and `endl`.

- **Explicitly, with member function sync():** Calling stream's member function `sync()`, which takes no parameters, causes an immediate synchronization. This function returns an `int` value equal to `-1` if the stream has no associated buffer or in case of failure. Otherwise (if the stream buffer was successfully synchronized) it returns `0`.