`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:

```
// 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.

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

ifstream::pos_type size;
char * memblock;

int main () {
  ifstream file ("example.bin",
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`.