C++ Binary File I/O

Contents:

C++ file input and output are typically achieved by using an object of one of the following classes:

- `ifstream` for reading input only.
- `ofstream` for writing output only.
- `fstream` for reading and writing from/to one file.

All three classes are defined in `<fstream.h>`. Throughout this page, the term "file stream" will be used when referring to features that apply equally to all three classes.

Normally, for binary file i/o you **do not** use the conventional text-oriented << and >> operators! It can be done, but that is an advanced topic.

Basic Model for File I/O

In C++, the file stream classes are designed with the idea that a file should simply be viewed as a stream or array of uninterpreted bytes. For convenience, the "array" of bytes stored in a file is indexed from zero to *len*-1, where *len* is the total number of bytes in the entire file.

Each open file has two "positions" associated with it:

1. The current reading position, which is the index of the next byte that will be read from the file. This is called the "get pointer" since it points to the next character that the basic `get` method will return.
2. The current writing position, which is the index of the byte location where the next output byte will be placed. This is called the "put pointer" since it points to the location where the basic `put` method will place its parameter.

These two file positions are independent, and either one can point anywhere at all in the file.

Getting The Size of a File

The typical way to get the size of a file is to use the C library function `stat`:

```
#include <sys/stat.h>
...
struct stat results;

if (stat("input.bin", &results) == 0)
    // The size of the file in bytes is in
    // results.st_size
else
    // An error occurred
```

Note that the second parameter to `stat` is a pointer. It is your responsibility to create and manage the memory where `stat` will place its results, and the address of that memory is what you should pass in as this second parameter. The above example shows the use of a local variable to hold the results returned by `stat`.

Opening a File

A file stream object can be opened in one of two ways. First, you can supply a file name along with an i/o mode parameter to the constructor when declaring an object:

```
ifstream myFile ("data.bin", ios::in | ios::binary);
```

Alternatively, after a file stream object has been declared, you can call its `open` method:

```
ofstream myFile;
...
myFile.open ("data2.bin", ios::out | ios::binary);
```

Either approach will work with an `ifstream`, an `ofstream`, or an `fstream` object.

Normally, when manipulating text files, one omits the second parameter (the i/o mode parameter). However, in order to manipulate binary files, you should always specify the i/o mode, including `ios::binary` as one of the mode flags. For read/write access to a file, use an `fstream`:

```
fstream myFile;
myFile.open ("data3.bin", ios::in | ios::out | ios::binary);
```

Note: If you are a GNU g++ user (version 2.7.x or earlier), then do **not** use i/o mode flags when opening `ifstream` objects. Because of a bug in the GNU libg++ implementation, the flags will not be correctly interpreted. If you are working under Unix, omit the i/o mode flags entirely; if you are working with g++ under MS-DOS, then use an `fstream` object. This note applies to g++ users *only*.

Reading From a File

To read from an `fstream` or `ifstream` object, use the `read` method. This method takes two parameters:

```
istream& read(char*, int);
```

The `read` member function extracts a given number of bytes from the given stream, placing them into the memory pointed to by the first parameter. It is your responsibility to create and manage the memory where `read` will place its result, as well as to ensure that it is large enough to hold the number of bytes requested. The bytes that are read and not interpreted, the method does not assume anything about line endings, and the `read` method does **not** place a null terminator at the end of the bytes that are read in.

If an error occurs while reading (for example, if you read off the end of a file), the stream is placed in an error state. If that occurs, you can use the `gcount` method to find out the number of characters that were actually read, and use the `clear` method to reset the stream to a usable state. Once a stream goes into an error state, all future `read` operations will fail.

An example:

```
#include <fstream.h>
...
char buffer[100];
ifstream myFile ("data.bin", ios::in | ios::binary);
myFile.read (buffer, 100);
if (!myFile) {
    // An error occurred!
    // myFile.gcount() returns the number of bytes read.
    // calling myFile.clear() will reset the stream state
    // so it is usable again.
}
...
if (!myFile.read (buffer, 100)) {
    // Same effect as above
}
```

Repositioning the "Get" Pointer

To change the position of the "get" pointer (the file reading position) of an `fstream` or `ifstream` object, use the `seekg` method. The basic form of this operation takes a single parameter:

```
istream& seekg(streampos pos);
```

A `streampos` is essentially an `unsigned long` integer value. `seekg` moves the get pointer to the specified absolute file position (where 0 is the start of the file).

When calling `seekg` be careful of the types of your arguments:

```
#define BLKSIZE 1024
int blk_number;
...
myFile.seekg (blk_number * BLKSIZE);  // Error!
```

The problem above is that files can be relatively large, so `streampos` can hold very large numbers. But above, if `blk_number` is above **63**, because of the types of `blk_number` and `BLKSIZE` (both `int`s), on a PC their product can only be 16 bits wide. To correct this, ensure constants used in file positioning are `long` quantities, or that all variables used in such positioning are `long` quantities, or both.

You can determine the current get pointer position using "`myFile.tellg()`", a method with no parameters that returns the index of the get pointer on the given stream.

There is also a variant of `seekg` that allows you to specify a position relative to the current get pointer location, or relative to the end of the file.

Writing To a File

To write to an `fstream` or `ofstream` object, use the `write` method. This method takes two parameters:

```
ostream& write(const char*, int);
```

The `write` member function writes a given number of bytes on the given stream, starting at the position of the "put" pointer. If the put pointer is current at the end of the file, the file is extended. If the put pointer points into the middle of the file, characters in the file are overwritten with the new data. The bytes that are written and not interpreted, no carriage return is added after the data, and the `write` method does **not** assume there is a null terminator at the end of the bytes that are being written.

If an error occurs while writing (for example, if you run out of disk space), the stream is placed in an error state. Such errors are not as common as read errors, and are often not checked.

An example:

```
#include <fstream.h>
...
char buffer[100];
ofstream myFile ("data.bin", ios::out | ios::binary);
myFile.write (buffer, 100);
```

Repositioning the "Put" Pointer

To change the position of the "put" pointer (the file reading position) of an `fstream` or `ofstream` object, use the `seekp` method. The basic form of this operation takes a single parameter:

```
ostream& seekp(streampos pos);
```

A `streampos` is essentially an `unsigned long` integer value. `seekp` moves the put pointer to the specified absolute file position (where 0 is the start of the file).

You can determine the current put pointer position using "`myFile.tellp()`", a method with no parameters that returns the index of the put pointer on the given stream.

There is also a variant of `seekp` that allows you to specify a position relative to the current put pointer location, or relative to the end of the file.

Reading and Writing Complex Data

Although the `read` and `write` methods accept a `char*` pointer, there is no requirement that the data you read

and/or write be held in a `char` array. You can read or write complex data objects using simple type casting of pointers:

```
#include <fstream.h>
...
class Data {
    int    key;
    double value;
};

Data x;
Data *y = new Data[10];

fstream myFile ("data.bin", ios::in | ios::out | ios::binary);
myFile.seekp (location1);
myFile.write ((char*)&x, sizeof (Data));
...
myFile.seekg (0);
myFile.read ((char*)y, sizeof (Data) * 10);
```

Closing a File

For all file stream objects, use:

```
myFile.close();
```

CS2604 class account (cs2604@courses.cs.vt.edu)

Wed Jan 28 1997