

Conceptually a file is thought of as an array of bytes.

If there are  $k$  bytes in the file, the bytes are indexed from 0 to  $k-1$ .

The bytes in the file are un-interpreted – they have no meaning.

Each byte stores an sequence of 8 bits that can be interpreted as an integer in the range 0 to 255 or as a hex value in the range 0 to FF. That is commonly exploited when displaying binary data.

So, a byte could be interpreted as storing just a `char` value, or one byte of an `int`, or one byte of something else.

42	A0	06	37	FF	04	04	00	B8	...
----	----	----	----	----	----	----	----	----	-----

## Opening a File Stream

Open file as usual with either constructor or open function, except...

Need to specify open mode

```
ifstream inFile("data.bin", ios::in | ios::binary);
ofstream outFile("dataout.bin", ios::out | ios::binary);
fstream inOutFile("data.bin", ios::in | ios::out |
    ios::binary); //both read and write
```

To open a file stream use open mode information in `ios` class.

Actual modes determined by compiler implementation, although those shown here are conformant to the C++ Standard.

Typical modes

<code>app</code>	open so write appends to file
<code>ate</code>	“at the end”
<code>binary</code>	i/o in binary mode instead of text
<code>in</code>	open for reading
<code>out</code>	open for writing
<code>trunc</code>	eliminate contents when open

## Binary File Example

C++ Binary File I/O 3

The display below shows a *hex dump* of a binary file.

The columns to the left and the headers at the top indicate byte offsets within the file

The hex codes for the actual nibbles in the file are shown in the middle columns.

ASCII interpretations of the bytes are shown at the right.

```
      00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  0123456789ABCDEF
-----
0000:0000  A4 00 00 00 51 39 56 32 4C 32 42 00 50 75 74 61  «...Q9V2L2B.Puta
0000:0010  74 69 76 65 20 31 2D 61 6D 69 6E 6F 63 79 63 6C  tive 1-aminocycl
0000:0020  6F 70 72 6F 70 61 6E 65 2D 31 2D 63 61 72 62 6F  opropane-1-carbo
0000:0030  78 79 6C 61 74 65 20 64 65 61 6D 69 6E 61 73 65  xylate deaminase
0000:0040  20 28 45 43 20 33 2E 35 2E 39 39 2E 37 29 01 00  (EC 3.5.99.7)..
0000:0050  11 00 50 79 72 6F 63 6F 63 63 75 73 20 61 62 79  ..Pyrococcus aby
0000:0060  73 73 69 4A 01 4D 48 50 4B 56 44 41 4C 4C 53 52  ssiJ.MHPKVDALLSR
0000:0070  46 50 52 49 54 4C 49 50 57 45 54 50 49 51 59 4C  FPRITLIPWETPIQYL
0000:0080  50 52 49 53 52 45 4C 47 56 44 56 59 56 4B 52 44  PRISRELGVVDVYVKRD
```

## File Pointers

C++ Binary File I/O 4

File pointers are positions in a file for reading and writing.

`get` pointer is used for reading – points to the next byte to read.

`put` pointer is used for writing – points to the next byte location for a write.

Both are usable at the same time only if working with `fstream` open for read/write.

`get` and `put` pointers are independent in the sense that they don't have to point to the same place in the file.

However, moving one invalidates the other.

File pointers are moved by an operation called *seeking*.

Functions `seekg` to move get pointer, and `seekp` to move put pointer

A seek can go to an absolute location

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

Type `streampos` is a large positive integer (long)

Parameter is absolute location where pointer should be positioned

Be careful with types of constants when computing absolute locations

A seek can also move to a relative location

```
istream& seekg(streamoff offset, ios::seek_dir loc);  
ostream& seekp(streamoff offset, ios::seek_dir loc);
```

Specify offset relative to `loc`: `beg`, `cur`, `end`

Type `streamoff` is large integer

Can find location of get and put pointers using tell functions

```
streampos tellg(); //position of get pointer  
streampos tellp(); //position of put pointer
```

Read from either `ifstream` or `fstream` (opened for input)

Use stream class member function `read`:

```
istream& read(char* target, int num)
```

Reads `num` bytes from the file stream into storage pointed to by `target`.

You must be sure that you can store `num` bytes to that location!

```
#include <iostream>  
#include <fstream>  
using namespace std;  
  
int main( int argc, char** argv ) {  
    unsigned int numberOfRecords;  
    ifstream myFile (argv[1], ios::in | ios::binary);  
    myFile.read( (char*) &numberOfRecords,  
                sizeof(unsigned int));  
    . . .  
    return 0;  
}
```

Write to either ofstream or fstream open for write

Use stream member function write:

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

Writes num consecutive bytes to location of put pointer in output stream starting with the byte located in memory at source.

Will overwrite bytes if put pointer is located in middle of file.

Extends file if put pointer at the end.

Stream errors less likely for write.

## Example: Text to Binary

```
int main() {
    string Name;
    int* Scores = new int[5];
    int NameLen;
    ifstream TextIn("Data.txt");
    ofstream BinOut("Data.bin", ios::out | ios::binary);

    getline(TextIn, Name, '\t');
    while (TextIn) {
        for (int Idx = 0; Idx < 5; Idx++)
            TextIn >> Scores[Idx];
        TextIn.ignore(255, '\n');

        NameLen = Name.length();
        BinOut.write((char*) &NameLen, sizeof(int));
        BinOut.write(Name.c_str(), Name.length());
        BinOut.write((char*) Scores, 5*sizeof(int));
        getline(TextIn, Name, '\t');
    }
    TextIn.close();
    BinOut.close();
}
```

**Cast address of integer to char\***

**Convert string object to char\***

**Cast integer array address to char\***

## Example: Binary to Text

C++ Binary File I/O 9

```
ifstream BinIn("Data.bin", ios::in | ios::binary);
ofstream TextOut("reRead.txt");

char* cName = new char[100];
BinIn.read((char*) &NameLen, sizeof(int));

while (BinIn) {
    BinIn.read(cName, NameLen);
    cName[NameLen] = '\\0';

    BinIn.read((char*) Scores, 5*sizeof(int));

    TextOut << cName;
    for (int j = 0; j < 5; j++)
        TextOut << ":" << Scores[j];
    TextOut << endl;

    BinIn.read((char*) &NameLen, sizeof(int));
}

BinIn.close();
TextOut.close();
return 0;
}
```

Cast address of  
integer array to  
char\*

## Example: Input and Results

C++ Binary File I/O 10

Kernighan, Brian	48	-1	-1	-1	-1
Gates, Bill	101	22	-1	-1	-1
Jobs, Steve	46	-1	-1	-1	-1
Stroustrup, Bjarne	21	86	52	93	-1
Lovelace, Ada	57	91	13	54	-1
von Neumann, Jon	7	84	-1	-1	-1

```
Kernighan, Brian:48:-1:-1:-1:-1
Gates, Bill:101:22:-1:-1:-1
Jobs, Steve:46:-1:-1:-1:-1
Stroustrup, Bjarne:21:86:52:93:-1
Lovelace, Ada:57:91:13:54:-1
von Neumann, Jon:7:84:-1:-1:-1
```

## Example: Binary Output File

C++ Binary File I/O 11

```
0000:0000 10 00 00 00 4B 65 72 6E 69 67 68 61 6E 2C 20 42      K e r n i g h a n , B
0000:0010 72 69 61 6E 30 00 00 00 FF FF FF FF FF FF FF FF    r i a n O
0000:0020 FF FF FF FF FF FF FF FF 08 00 00 00 47 61 74 65      G a t e
0000:0030 73 2C 20 42 69 6C 6C 65 00 00 00 16 00 00 00 FF    s , B i l l i e
0000:0040 FF FF FF FF FF FF FF FF FF FF FF 08 00 00 00 4A    J
0000:0050 6F 62 73 2C 20 53 74 65 76 65 2E 00 00 00 FF FF    o b s , S t e v e .
0000:0060 FF FF FF FF FF FF FF FF FF FF FF FF FF 12 00
0000:0070 00 00 53 74 72 6F 75 73 74 72 75 70 2C 20 42 6A    S t r o u s t r u p , B j
0000:0080 61 72 6E 65 15 00 00 00 56 00 00 00 34 00 00 00    a r n e V 4
0000:0090 5D 00 00 00 FF FF FF FF 0D 00 00 00 4C 6F 76 65    ] L o v e
0000:00A0 6C 61 63 65 2C 20 41 64 61 39 00 00 00 5B 00 00    l a c e , A d a 9 [
0000:00B0 00 00 00 00 36 00 00 00 FF FF FF FF 10 00 00      6
0000:00C0 00 76 6F 6E 20 4E 65 75 6D 61 6E 6E 2C 20 4A 6F    v o n N e u m a n n , J o
0000:00D0 6E 07 00 00 00 54 00 00 00 FF FF FF FF FF FF FF    n T
0000:00E0 FF FF FF FF FF 00 00 00 00 00 00 00 00 00 00 00
0000:00F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

## Reading and Writing Complex Data

C++ Binary File I/O 12

The read and write functions deal with un-interpreted bytes

This means can read and write self-contained structured data (objects or structs) by using type casting:

```
#include <fstream>

class Data {
public:
    Data() : key(0), address(0) {}
    Data(int k, unsigned long add) : key(k), address(add) {}
    int getKey() const { return key; }
    unsigned long getAddress() const { return address; }
private:
    int key;
    unsigned long address;
}
```

However, be careful of this if the "object" has dynamic content; e.g.,

```
class Place {
private:
    string Name;
    . . .
};
```

```
int main() {
    Data entry(1,200L);
    Data *array = new Data[10];
    unsigned long location1 = 10L*sizeof(Data);
    fstream myFile("data.bin",
        ios::in | ios::out | ios::binary);

    //move put pointer to location1
    myFile.seekp(location1);
    //write 1 Data object to file
    myFile.write((char*)&entry, sizeof(Data));

    //move get pointer to beginning of file
    myFile.seekg(0);
    //read 10 Data objects into array
    myFile.read((char*)array, sizeof(Data) * 10);

    return 0;
}
```

When a complex object with dynamic content, such as a `string` object, is involved the issue becomes more complex.

Simply writing the correct number of bytes, from the starting address of the object in memory, will NOT produce the desired result.

The following example illustrates storing simple, variable-length objects to a file and recovering them. A simple, inefficient file index is also created and used.

We employ a simple data class to store data about a city:

```
class City {  
private:  
    string Name;  
    string State;  
    int    Population;  
public:  
    City();  
    City(string N, string S, int P);  
    string getName() const;  
    string getState() const;  
    int    getPop() const;  
    int    Size() const;  
};
```

```
int City::Size() const {  
    return (Name.length() +  
           State.length() +  
           sizeof(int));  
}
```

We also employ a simple class to hold the file index data for a City object:

```
class Entry {  
private:  
    string Key;  
    unsigned int Address;  
public:  
    Entry();  
    Entry(string K, unsigned int A);  
    string getKey() const;  
    unsigned int getAddress() const;  
};
```

We assume that city names are unique, and use those as our primary key. No hashing is employed; the `Entry` objects are simply stored in the order that the `City` objects are written to the file.

That is not ideal.

Writing the data fields of a `City` object to the current DB file location is relatively simple:

```
void writeCity(ostream& Out, City* toWrite) {  
  
    int NameLen = (toWrite->getName()).length();  
    int StateLen = (toWrite->getState()).length();  
  
    Out.write((char*) &NameLen, sizeof(int));  
    Out.write((char*) &StateLen, sizeof(int));  
    Out.write((toWrite->getName()).c_str(), NameLen);  
    Out.write((toWrite->getState()).c_str(), StateLen);  
    int Pop = toWrite->getPop();  
    Out.write((char*) &Pop, sizeof(int));  
}
```

To recover the strings we must either store length data (as shown here) or write delimiters to the DB file.

Reading the data fields and reconstructing a `City` object from the current DB file location is also relatively simple:

```
City readCity(istream& In) {  
    int NameLen, StateLen, Population;  
  
    In.read((char*) &NameLen, sizeof(int));  
    char* cName = new char[NameLen + 1];  
    In.read((char*) &StateLen, sizeof(int));  
    char* cState = new char[StateLen + 1];  
    In.read(cName, NameLen);  
    cName[NameLen] = '\\0';  
    In.read(cState, StateLen);  
    cState[StateLen] = '\\0';  
    In.read((char*) &Population, sizeof(int));  
  
    City toReturn(string(cName), string(cState), Population);  
  
    delete [] cName;  
    delete [] cState;  
    return toReturn;  
}
```

A text file of city data is parsed, and an array of `City` objects is created. Then, those `City` objects are written to a binary DB file and an array of index `Entry` objects is created:

```

City G[MAXCITIES];
Entry H[MAXCITIES];

int numCities = initCityList(G);

fstream BinData("Data.bin",
                ios::in | ios::out | ios::binary);

for (int Idx = 0; Idx < numCities; Idx++) {
    unsigned int Offset = BinData.tellp();

    H[Idx] = Entry(G[Idx].getName(), Offset);

    writeCity(BinData, &G[Idx]);
}

```

Finding a city, given its name, is a simple matter of looking up the file offset for the relevant record and then reading in that record:

```

City findCity(string CityName, const Entry Index[], int Size,
             fstream& DB) {

    int Idx = 0;
    while ( (Idx < Size) && (Index[Idx].getKey() != CityName) ) {
        Idx++;
    }
    if (Idx == Size) {
        return City("No such city", "", 0);
    }
    unsigned int Offset = Index[Idx].getAddress();
    DB.seekg(0);
    DB.seekg(Offset);
    City Target = readCity(DB);
    return Target;
}

```