Model of File I/O

- 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
- So, a byte could store just a char value, or it could store one byte of an int, or it could store one byte of something else

165	244	56	4	…
				…
				…
…	…	…	…	…

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

File Pointers

- 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 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

Reading Data

- Read from either ifstream or fstream (open for read)
- Use 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 write num characters to that location

```
#include <iostream>
#include <fstream>
void main() {
  char buffer[100];
  ifstream myFile (“data.bin”, ios::in | ios::binary);
  myFile.read(buffer,100);
  if (!myFile) { //error occurred
    cerr << “Error reading from file data.bin, only “
         << myFile.gcount() << “ bytes read” << endl;
    myFile.clear(); //reset file for reading
  }
}
```

Writing Data

- Write to either ofstream or fstream open for write
- Use 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

```
void 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) {
      int Idx;
      for (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

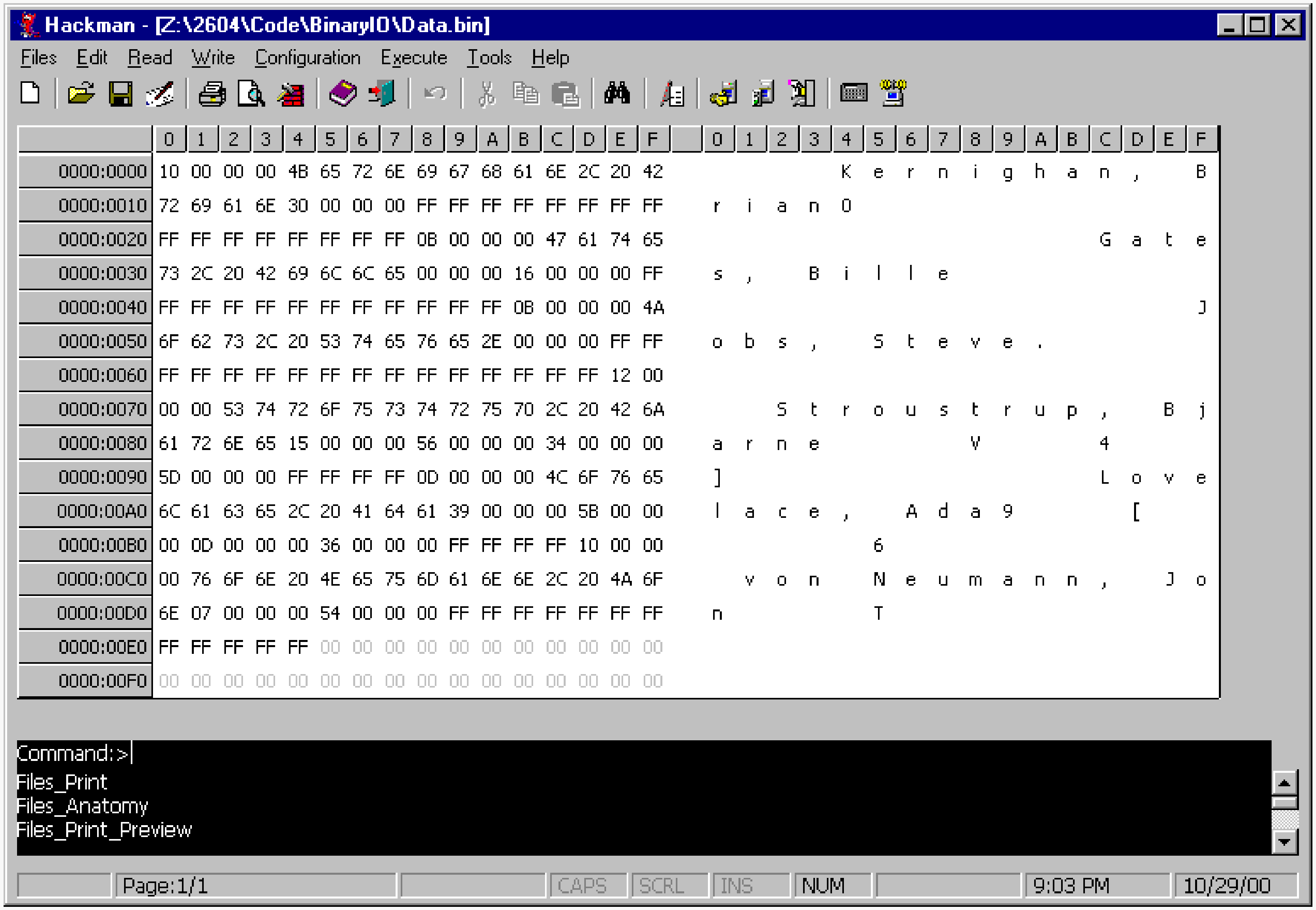
```
   ifstream BinIn("Data.bin", ios::in | ios::binary);
   ofstream TextOut("reRead.txt");

   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();
}
```

Example: Input and Results

```
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


Hackman - [Z:\2604\Code\BinaryIO\Data.bin]
Files Edit Read Write Configuration Execute Tools Help
Page:1/1
CAPS
SCRL
INS
NUM
9:03 PM
10/29/00

```
           0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F   0123456789ABCDEF
0000:0000  10 00 00 00 4B 65 72 6E 69 67 68 61 6E 2C 20 42      Kernighan, B
0000:0010  72 69 61 6E 30 00 00 00 FF FF FF FF FF FF FF FF   rian0
0000:0020  FF FF FF FF FF FF FF FF 0B 00 00 00 47 61 74 65               Gate
0000:0030  73 2C 20 42 69 6C 6C 65 00 00 00 16 00 00 00 FF   s, Bille
0000:0040  FF FF FF FF FF FF FF FF FF FF FF 0B 00 00 00 4A                  J
0000:0050  6F 62 73 2C 20 53 74 65 76 65 2E 00 00 00 FF FF   obs, Steve.
0000:0060  FF 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     Stroustrup, Bj
0000:0080  61 72 6E 65 15 00 00 00 56 00 00 00 34 00 00 00   arne    V   4
0000:0090  5D 00 00 00 FF FF FF FF 0D 00 00 00 4C 6F 76 65   ]           Love
0000:00A0  6C 61 63 65 2C 20 41 64 61 39 00 00 00 5B 00 00   lace, Ada9   [
0000:00B0  00 0D 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    von Neumann, Jo
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 00
```

```
Command:>|
Files_Print
Files_Anatomy
Files_Print_Preview
```

Moving File Pointers

- 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);`
  - `istream& 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`

Reading and Writing Complex Data

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;
. . .
};
```

Complex Data (cont)

```
void 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);
}
```

Dynamic Objects

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.

Data Class

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));
}
```

Index Entry Class

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 a City Object to the DB File

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 a City Object from 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;
}
```

Building an Index for the DB File

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 Object in the DB File

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;
}
```