Files

Class 39

Chapter 12

- we will mostly focus on sections 12.7 through 12.10
- most of what you need is in these slides

ASCII Encoding

- remember a byte is 8 contiguous bits
- and the ASCII encoding scheme for bytes

Binary	Decimal	Category
0000 0000 – 0001 1111	0 – 31	control characters
0010 0000	32	space
0010 0001 – 0010 1111	33 – 47	punctuation
0011 0000 – 0011 1001	48 – 57	digits
0011 1010 – 0100 0000	58 – 64	punctuation
0100 0001 – 0101 1011	65 – 90	uppercase
0101 1100 – 0110 0000	91 – 96	punctuation
0110 0001 – 0111 1001	97 – 122	lowercase
0111 1010 – 0111 1110	123 – 126	punctuation
0111 1111	127	delete

Files

- a disk file contains a sequence of bytes
- when you ask the operating system for some stuff from a file, you just get bytes
- the information you get depends on how the bytes are interpreted
- it is up to the programmer to use the correct IO functions so the bytes will be interpreted correctly

Files

- a disk file contains a sequence of bytes
- when you ask the operating system for some stuff from a file, you just get bytes
- the information you get depends on how the bytes are interpreted
- it is up to the programmer to use the correct IO functions so the bytes will be interpreted correctly
- there are two main flavors of file
  1. text files
  2. binary files
- all files can be considered binary files (every file just contains bytes)
- text files, however, contain only bytes that correspond to ASCII characters plus the newline character for end-of-line
- thus text files are easy to interpret as a sequence of lines each of which is a sequence of characters

Text Files

- consider a CSV text file
- three comma-separated fields, a 9-digit ID, an unsigned min, and an unsigned max (both in the range 0 – 1,000)

```
234567890,25,125
123456789,100,1000
321645746,1,100
```

- there are 52 characters, including commas and newlines
- each character is one byte
- thus the file contains 52 bytes
- notice that the lines and fields are different lengths even though the format of the lines and fields is the same

Binary Files

- binary files contain bytes that do not (necessarily) correspond to ASCII characters
- they are not designed to be human-readable
- their bytes encode information, but not using ASCII (remember, ASCII is just one encoding scheme)
- some files are open standard formats, e.g., jpeg and pdf
- some are proprietary, e.g., xlsx and dwg

Integer Types

- from class 4 on 28 August, remember two of the integer types:

Name	# Bytes (ice)	Range
unsigned short	2 bytes	0 – 65,535
unsigned int	4 bytes	0 – 4,294,967,295

Integer Types

- from class 4 on 28 August, remember two of the integer types:

Name	# Bytes (ice)	Range
unsigned short	2 bytes	0 – 65,535
unsigned int	4 bytes	0 – 4,294,967,295

- different computers use different sizes
- for safety, if size matters, use explicit sizes
- defined in cstdint library

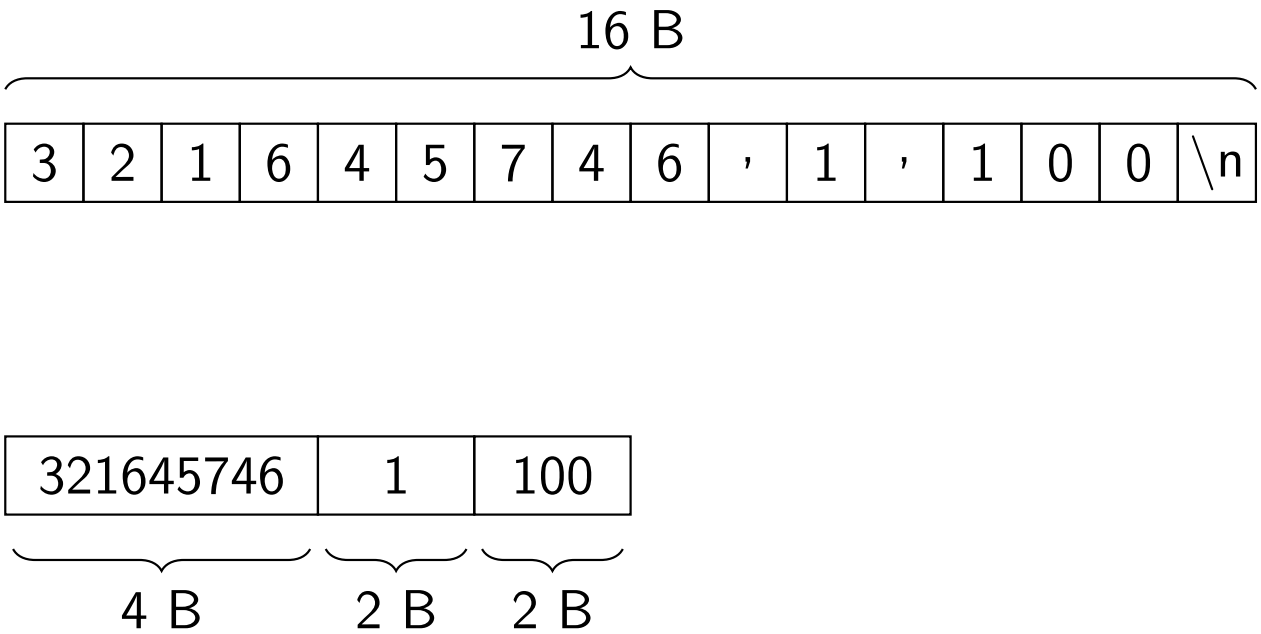
Name	# Bytes	Range
uint16_t	2 bytes	0 – 65,535
uint32_t	4 bytes	0 – 4,294,967,295

A Binary File

- the ID field is a 9-digit number, and thus can be encoded in a 4-byte unsigned integer (but not the smaller unsigned 2-byte type)
- the max and min fields are in the range 0 – 1,000, and thus can each be encoded in a 2-byte unsigned integer (but not a smaller 1-byte type)
- thus each line requires $4 + 2 + 2 = 8$ bytes, with no need for separators or newlines, for a 3-line total of 24 bytes
- the ID, min, and max information can be encoded in a binary file using only 24 bytes, less than half as much as ASCII

Text vs. Binary Files

- the same information in a text CSV file and a binary file
- text: comma-separated, newline-terminated, variable-length ASCII data
- binary: byte-structured fixed-length binary data



Sequential IO

- text files normally are read and written from beginning to end
- line by line, start to finish
- this is how we have read and written all files this semester

Sequential IO

- text files normally are read and written from beginning to end
- line by line, start to finish
- this is how we have read and written all files this semester

- they have to be processed start to finish, because each line may be a different length
- each item in the line may be a different length
- the processing doesn't know where in the file a newline character will be reached until one is actually encountered

Sequential IO

- text files normally are read and written from beginning to end
- line by line, start to finish
- this is how we have read and written all files this semester

- they have to be processed start to finish, because each line may be a different length
- each item in the line may be a different length
- the processing doesn't know where in the file a newline character will be reached until one is actually encountered

- going back to our simple example, each line is of a different length, 17 bytes, 19 bytes, and 16 bytes, respectively

```
234567890,25,125
123456789,100,1000
321645746,1,100
```

Binary Files

- binary files, however, do not have lines
- lines are a useful fiction in text files
- binary files can be made with fixed-size records
- if an ID, a min, and a max are going to be in a program as a struct, the struct would look like this, with exactly 8 bytes for each record:

```
struct Record
{
  uint32_t id;  // exactly 4 bytes
  uint16_t min; // exactly 2 bytes
  uint16_t max; // exactly 2 bytes
};
```

Binary Files

- if we store information in a file not as variable-length ASCII lines
- but instead as fixed-length binary structs
- we get:
  1. far less disk space used
  2. exact placement of each record

The Binary File Toolkit

- just like with text files, we need the fstream library

The Binary File Toolkit

- just like with text files, we need the fstream library
- unlike text files, binary files are normally opened for input and output at the same time

```
fstream file;
file.open("bindata.dat", ios::out | ios::in | ios::binary);
```

- fstream file type, not ifstream or ofstream
- ios::out etc are enumeration values (explained in section 11.11, which we did not cover)
- the single vertical bar is the bitwise-or operator (not the same as the logical-or double vertical bar)
- opening a file this way does not erase the contents as the text file open does

The Binary File Toolkit: Writing a Record

- binary files are just bytes; the only thing that can be read from or written to binary files are single, individual bytes
- the 1-byte data type in C++ is char
- everything must be converted to or from char
- this is done with with reinterpret_cast

```
Record record {234567890, 50, 75};
file.write(reinterpret_cast<char*>(&record), sizeof record);
```

- write has two arguments, the address of the first byte to write, and the count of how many bytes to write to the file

The Binary File Toolkit: Reading a Record

- reading a record is almost the same

```
Record record;
file.read(reinterpret_cast<char*>(&record), sizeof record);
```

The Binary File Toolkit: Reading a Record

- reading a record is almost the same

```
Record record;
file.read(reinterpret_cast<char*>(&record), sizeof record);
```

- like all other file situations, before the program ends, the file must be closed

```
file.close();
```

Strings

- strings are a big problem
- in our programs, we want to use only C++ strings
- but a C++ string is a complex object
- especially because it has variable length

- the whole point of binary files is fixed-length records
- so when we store string data in binary files, we must use C-string fields

- in general: C-strings on disk, C++ strings in program
- convert back and forth as we read and write

Converting Strings

- C-string $\rightarrow$ C++ string: `string cppstr = cstr;`
- extract C-string from C++ string: `cppstr.c_str();`

- to actually "convert" a C++ string to a C-string:

```
char cstr[SIZE]; // must make sure size is big enough!
strcpy(cstr, cppstr.c_str());
```

The Binary File Toolkit: Seeking

- the hardest part of processing binary files is choosing the correct place for reading and writing
- binary files are typically not processed sequentially
- rather we use random access for binary files
- a binary file has a read marker and a write marker
- the read marker is positioned with the seekg (for get) function and the write marker is positioned with the seekp (for put) function

The Binary File Toolkit: Seeking

- both seek functions have a parameter that is a long number of bytes from the beginning of the file
- the following reads the third record in the file:

```
file.seekg(2L * sizeof record);
file.read(reinterpret_cast<char*>(&record), sizeof record);
```

Pros and Cons

Text File Pros

- the contents are human readable
- can easily view or modify them with any text editor
- can retrieve their information even if the original program is lost

Binary File Pros

- they are smaller than text files, often much smaller
- reading and writing is faster, often much faster
- can read and write just one record without reading and writing the whole file
- portable across platforms, with no end-of-line issues
- can use disk file as database (essentially unlimited size) instead of RAM