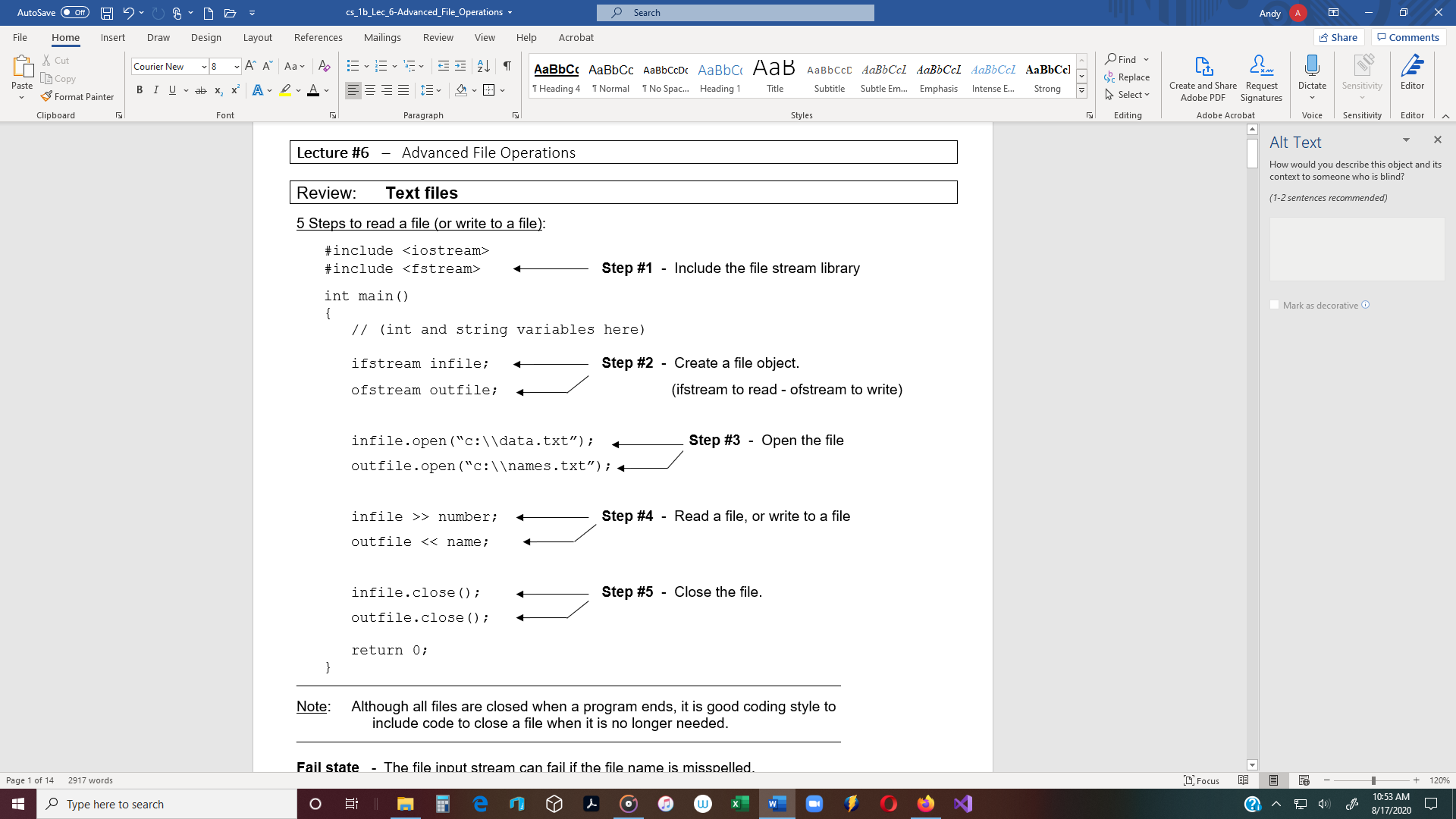
# **Lecture #6** – Advanced File Operations

Review: **Text files**



Note: Although all files are closed when a program ends, it is good coding style to

include code to close a file when it is no longer needed.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Fail state** - The file input stream can fail if the file name is misspelled.

* C++ does not automatically generate an error message if a file does not open.
* The programmer must include code to check to see if a file fails to open.

Two ways to check the file stream**:** (next page)

**1.) Call the fail() function**

if (outFile.fail()) // Function returns true or false.

{

cout << “File did not open\n”;

return 1; // When this statement is in main(), 1 is returned to

} // the operating system, and the program closes.

if (**!**inFile) // if the file does not open, ……….

{

cout << “Error opening file. “;

exit(1); // When this statement is used in any function, 1 is

// returned to the OS, and the program closes.

} // Defined in <**cstdlib**>, but not required in Visual Studio )

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**End-of-file marker** - When a file is written, the operating system automatically

writes an EOF marker at the end of the file.

* The end-of-file character is different for different operating systems.
* **.eof() function** - Use this function to detect the end of a file.

Ex 1: while (!inFile.eof()) // While it’s not the end of the file

{ // read numbers and output them.

inFile >> number;

cout << number << endl;

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Another way to read a text file 🡪 while (inFile >> number)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Run-time input of file names:**

Ex 2: ifstream inFile;

char fileName[30];

cout << “Enter the input file name: “;

cin >> fileName; // (or 🡪 cin.getline(fileName, 30)

inFile.open(fileName);

**Using the fstream Data Type**

Ex 3: #include <fstream>

int main()

{

int num1, num2, num3;

fstream dataFile; // fstream – The I/O mode must be specified.

// ***open()*** function takes 2 arguments:

dataFile.open(“info.txt”, ios::out);

filename **file access flag** – Open in output mode.

**File Access Flags** - C++ Access Flags specifies how a file will be read or written (for text and

binary files).

* **ios::in** - Input mode.
* **ios::out** - Output mode.
  + By default, data that is already in the file is overwritten.
* **ios::trunc** - If a file already exists, its contents will be deleted (truncated).
  + This is the default mode used by ios::out.
* **ios::app** - Append mode.
  + If a file exists, its contents are preserved and all ouput is

written to the end of the file.

* **ios::binary** - Binary mode.
  + When a file is opened in binary mode, data is read or

written in binary form. (Text is default)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

* Flags can be used with ifstream and ofstream objects.
* Several flags can be used together, by using the | operator.

Note: Opening for input and output at the same time

does not work with text (**binary only**).

Ex 4: dataFile.open(“info.txt”, **ios::out** **| ios::app**);

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Note: Another way to open a file is to open it when the file object is declared.

* This eliminates the open() function.

Ex 5: fstream dataFile(“names.txt”, ios::out | ios::app);

if (!dataFile)

cout << “Error opening file”;

NOTE: A file can be opened in input mode and output mode, in binary only (not text files).

**File Output Formatting** - File output may be formatted in the same way screen output

is formatted.

Ex 6: #include <fstream>

#include <iomanip> // for setprecision()

int main()

{

fstream dataFile(“numbers.txt”, ios::out);

double number = 17.123456;

dataFile << fixed << showpoint << setprecision(3)

<< number << endl << setprecision(1);

OUTPUT:

17.123

17.1

<< number << endl;

dataFile.close();

return 0;

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_

Ex 7: Writing 2 rows of data to a file:

#include <fstream>

#include <iomanip> // for setw()

const int ROWS = 2;

const int COLS = 2;

int main()

{

ASCII Code

Dec Hex

0 - 30

1 - 31

2 - 32

3 - 33

4 - 34

5 - 35

6 - 36

7 - 37

8 - 38

9 - 39

fstream outFile(“table.txt”, ios::out);

int numbers[ROWS][COLS] = {{2897, 5},{34, 7}};

for (int row = 0; row < ROWS; row++)

{

for (int col = 0; col < COLS; col++

{

outFile << setw(6) << numbers[row][col];

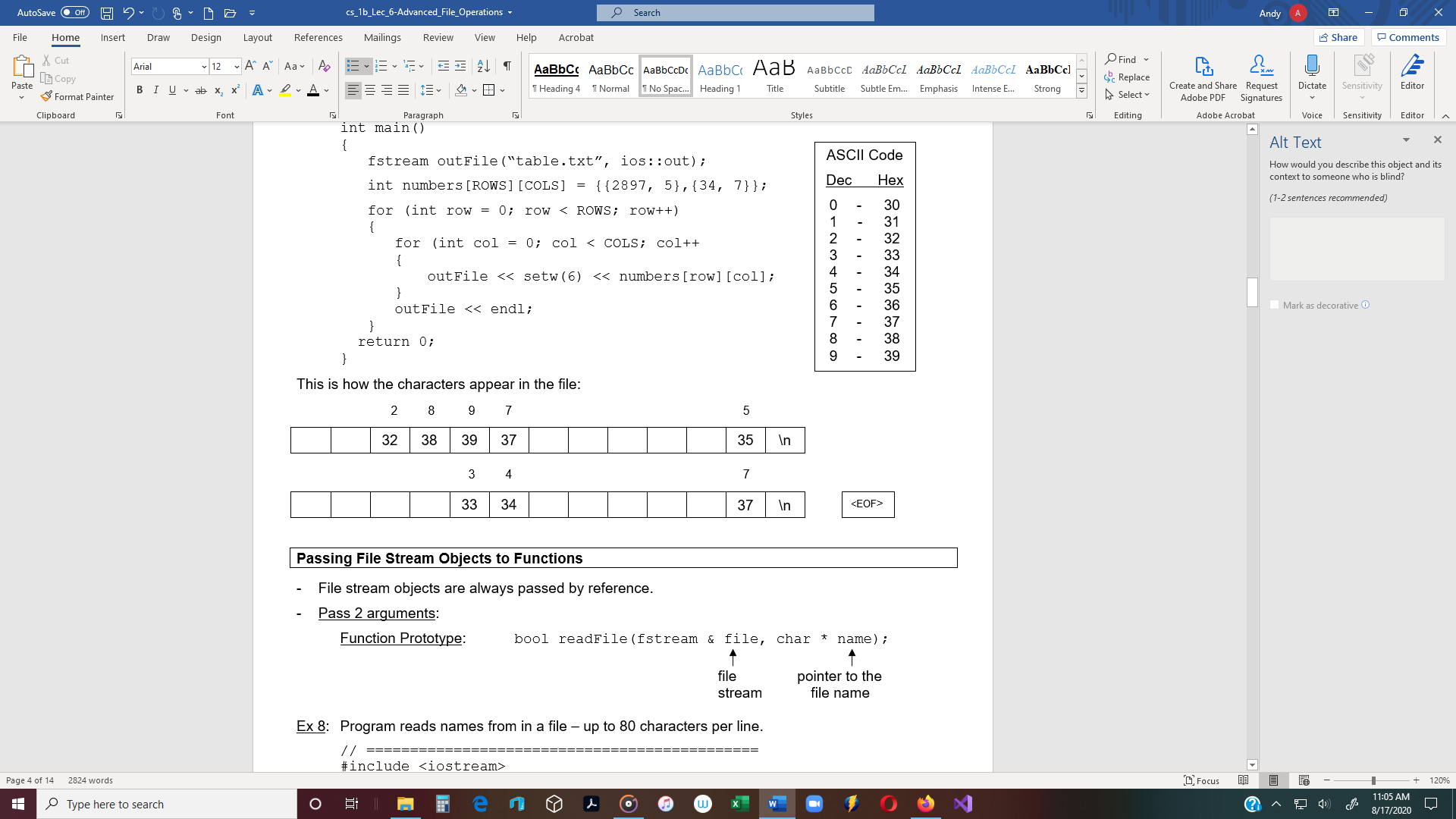
}

outFile << endl;

}

return 0;

}



**Passing File Stream Objects to Functions**

* File stream objects are always passed by reference.
* Pass 2 arguments:

Function Prototype: bool readFile(fstream & file, char \* name);

file pointer to the

stream file name

Ex 8: Program reads names from in a file – up to 80 characters per line.

// =============================================

#include <iostream>

#include <fstream>

using namespace std;

const int SIZE = 81;

bool readFile(fstream &, char \*);

void showContents(fstream &);

int main()

{

fstream inFile;

if (!readFile(inFile, “data.txt”))

{

cout << “Error opening file!\n”;

return 0;

}

cout << “File opened successfully.\n”;

cout << “Now reading file\n”;

showContents(inFile);

inFile.close();

return 0;

}

// ==== **readFile()** =============================

bool readFile(fstream & file, char \* name)

{

file.open(name, ios::in);

if (file.fail())

return false;

else

return true;

}

// ==== **showContents()** =========================

void showContents(fstream & file)

{

char line[SIZE];

while (file >> line) // (or file.getline(line, SIZE);

{

/\* OUTPUT:

File opened successfully.

Now reading file.

Jones

Smith

Willis

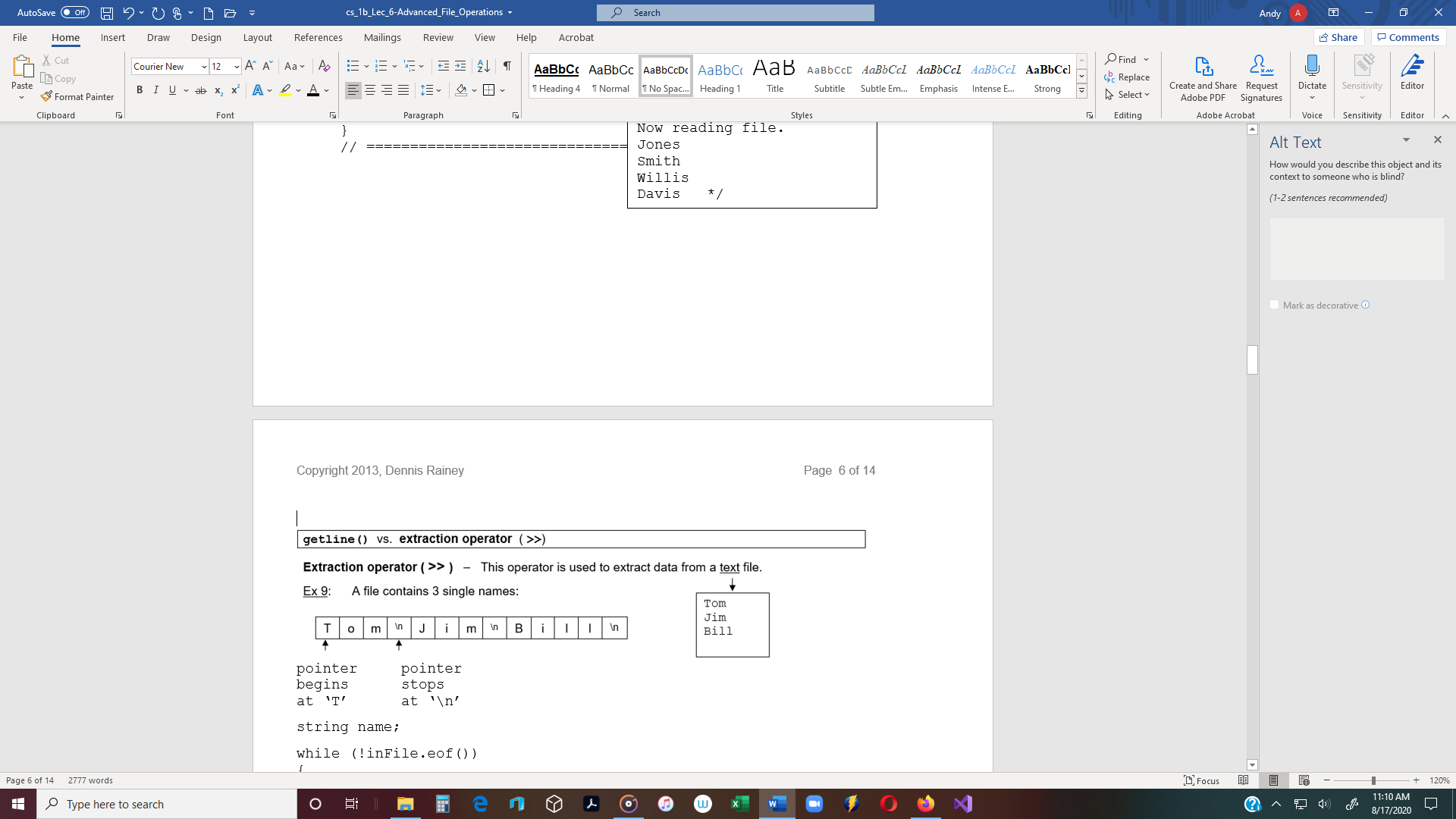
Davis \*/

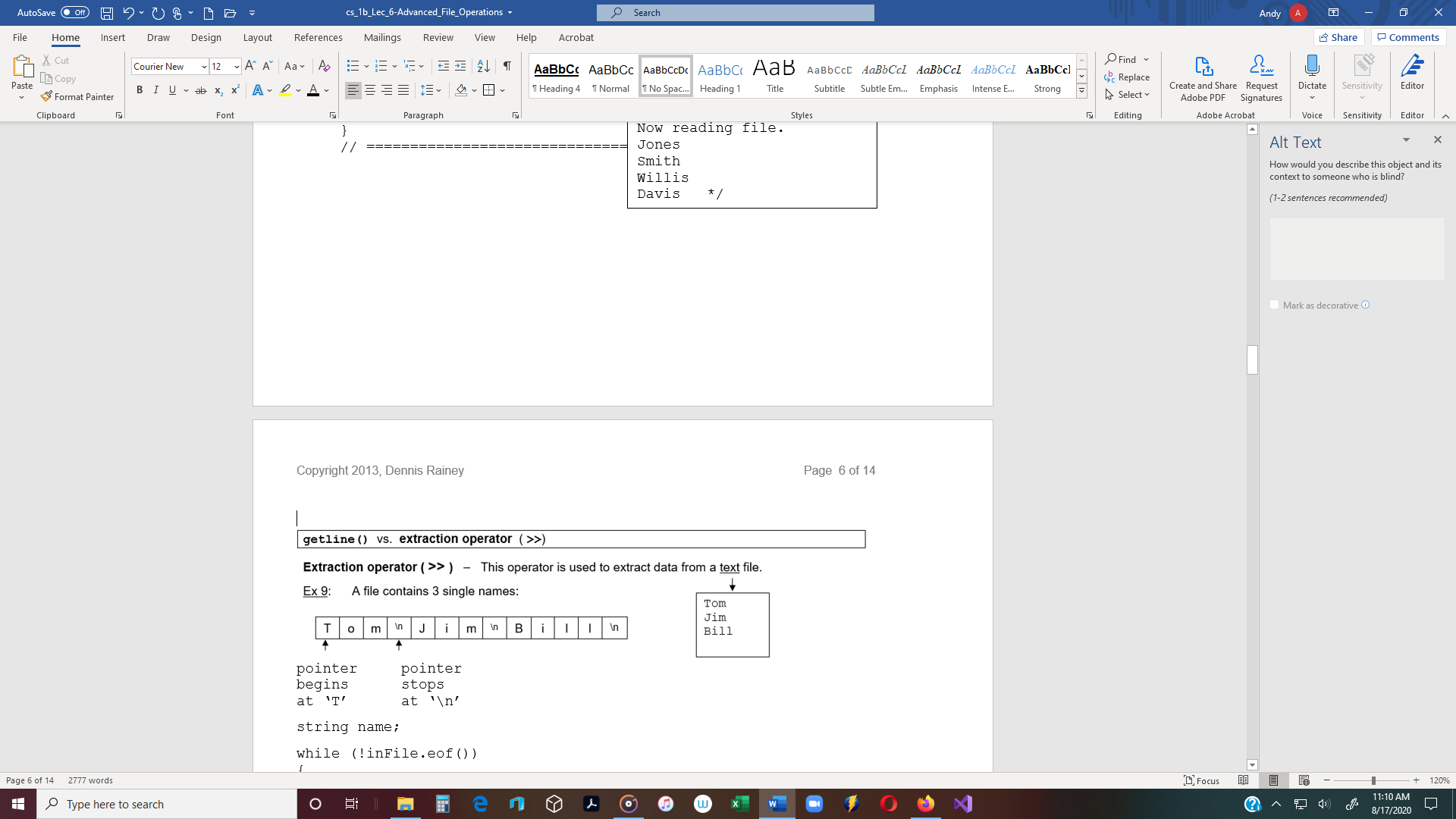
cout << line << endl;

}

}

// =============================================





string name;

while (!inFile.eof())

{

inFile >> name; // The extraction operator reads Tom and stops when it

// encounters any white space **('\n', '\t',** '')

**Problem**: The extraction operator cannot read

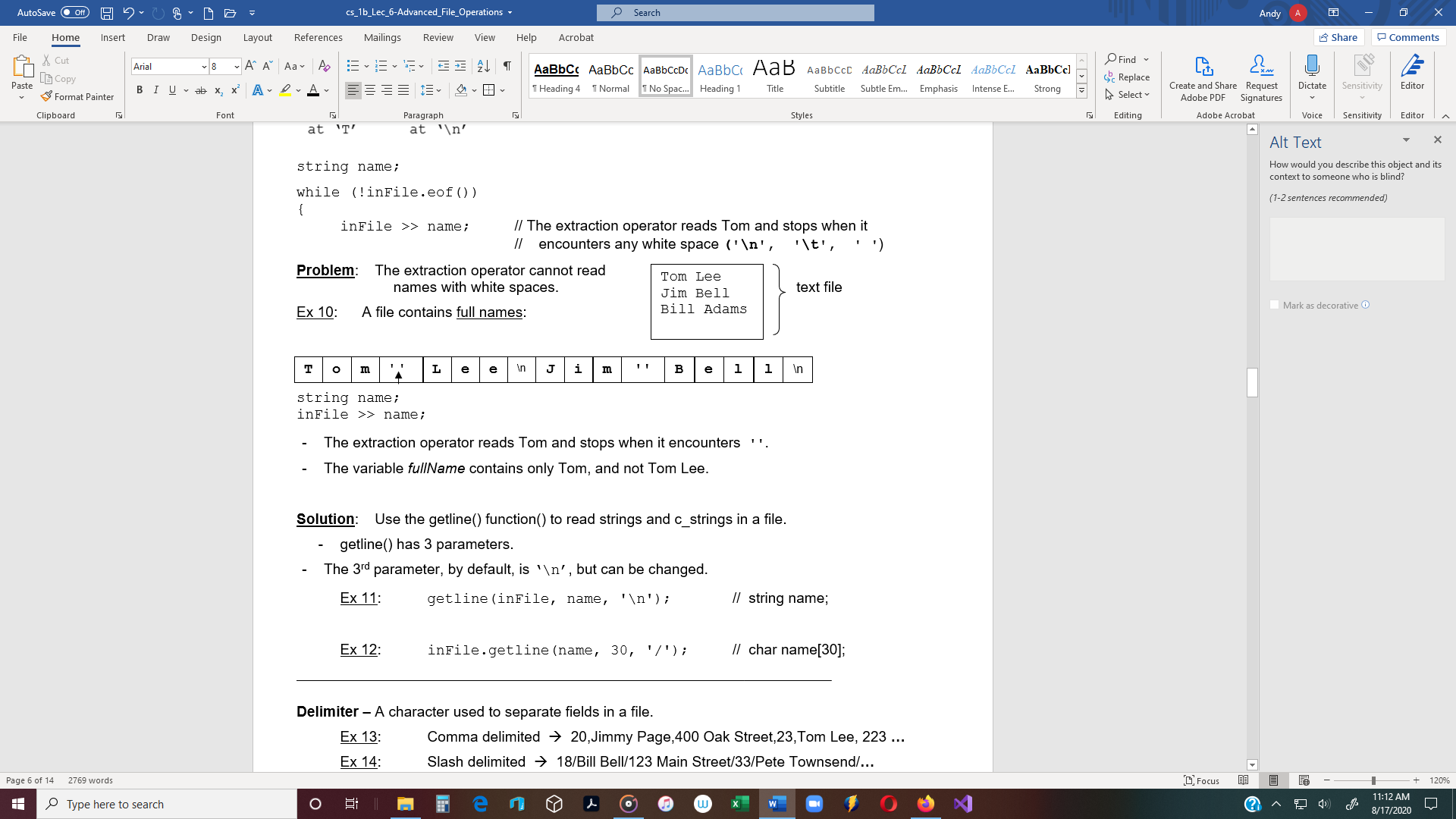
Tom Lee

Jim Bell

Bill Adams

names with white spaces. text file

Ex 10: A file contains full names:



* The extraction operator reads Tom and stops when it encounters ''.
* The variable *fullName* contains only Tom, and not Tom Lee.

**Solution**: Use the getline() function() to read strings and c\_strings in a file.

* getline() has 3 parameters.
* The 3rd parameter, by default, is ‘\n’, but can be changed.

Ex 11: getline(inFile, name, '\n'); // string name;

Ex 12: inFile.getline(name, 30, '/'); // char name[30];

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Delimiter –** A character used to separate fields in a file.

Ex 13: Comma delimited 🡪 20,Jimmy Page,400 Oak Street,23,Tom Lee, 223 **…**

Ex 14: Slash delimited 🡪 18/Bill Bell/123 Main Street/33/Pete Townsend/**…**

* To read a slash-delimited file, specify the delimiter in the 3rd argument.

Ex 15: **Read a delimited text file containing strings:**

string name;

while (true)

{

if (file.eof())

break;

getline(file, name, '/'); // getline reads text and stops

// at the '/' delimiter.

// The '/' is absorbed and the

// pointer advances.

cout << name << endl;

}

Ex 16: **Read a comma delimited text file, containing c\_strings**

char name[30];

while (!file.eof())

{

file**.**getline(name,30, ','); // getline() reads characters until

// it has read 29 characters, or

// it encounters a ',' character

// The ',' is absorbed and the

// pointer advances.

cout << name << endl;

}

**get() function**

* The ***get()*** function reads the next character from a text file.
* It will read any character, but only one, including white spaces ('\n', '\t', ' ')

Ex 17: The following can read the Gettysburg Address in a file and output it

to the screen.

inFile.get(character);

while(!inFile.eof())

{

cout << character;

inFile.get(character); // char character;

}

**putback() function -** The function moves the ***get*** ***pointer*** back one character in a text file

Ex 19: The following reads text, one character at a time from a text file.

* + The character is then checked to see if it is a ‘<’ or ‘>’.
  + If it is, then the ***get pointer*** is moved back one character.

inFile.get(character);

while(!inFile.eof())

{

if (character == '<')

{

inFile.putback(character);

// Put the character back on the buffer and then do something

}

else

inFile.get(character); // Otherwise get the character

} // end of while loop

**peek() function**

* The ***peek()*** function looks ahead to the next character in the text file, before getting it.

Ex 20: The following reads text, one character at a time.

inFile.get(character);

while(!inFile.eof())

{

if ((inFile.peek(character) == '>')

// do something;

else

inFile.get(character);

}

**Text vs. Binary Files**

**Text File** – Default is text file.

* A text file contains ASCII characters.
* Bytes are interpreted as ASCII characters.
* Data is stored character by character.
* The length of each line is not fixed.
* The extraction ( **>>** ) and Insertion ( **<<** ) operators work only with text files.
* The extraction operator ( >> ) and the getline() function read in bytes and

interpret them as ASCII characters.

* Searching a text file is linear (slower than searching a binary file).

**Binary File** – Contains strings of bits, un-interpreted by the file system

* Data is stored based on data type declarations, which have fixed lengths.

Ex 21: float gpa and int age require 4 bytes each.

* **binary** must be specified when creating a binary file.
* Use the ‘|’ operator specify binary and out

Ex 22: file**.**open(“c:\\data.bin”, ios**::**binary | ios::out);

* Records in a binary file can be accessed, changed, and re-inserted back into the

same location within the file.

Note for binary: Instead of the insertion operator (<<) - use the **write()** function.

Instead of the extraction operator (>>) - use the **read()** function.

**sizeof( )** – This function returns the number of bytes of the parameter.

* This function can return the number of bytes of memory a data type requires.
* It works with standard (native) C++ data types, as well as struct and class objects.

Ex 23: cout << sizeof(int); // output = 4 (bytes)

cout << sizeof(float); // output = 4 (bytes)

cout << sizeof(double); // output = 8 (bytes)

cout << sizeof(char); // output = 1 (byte)

**write()** - The write() function is defined in <fstream> and is used to write data to

a binary file.

* The general format is: fileObject.write(address, size);

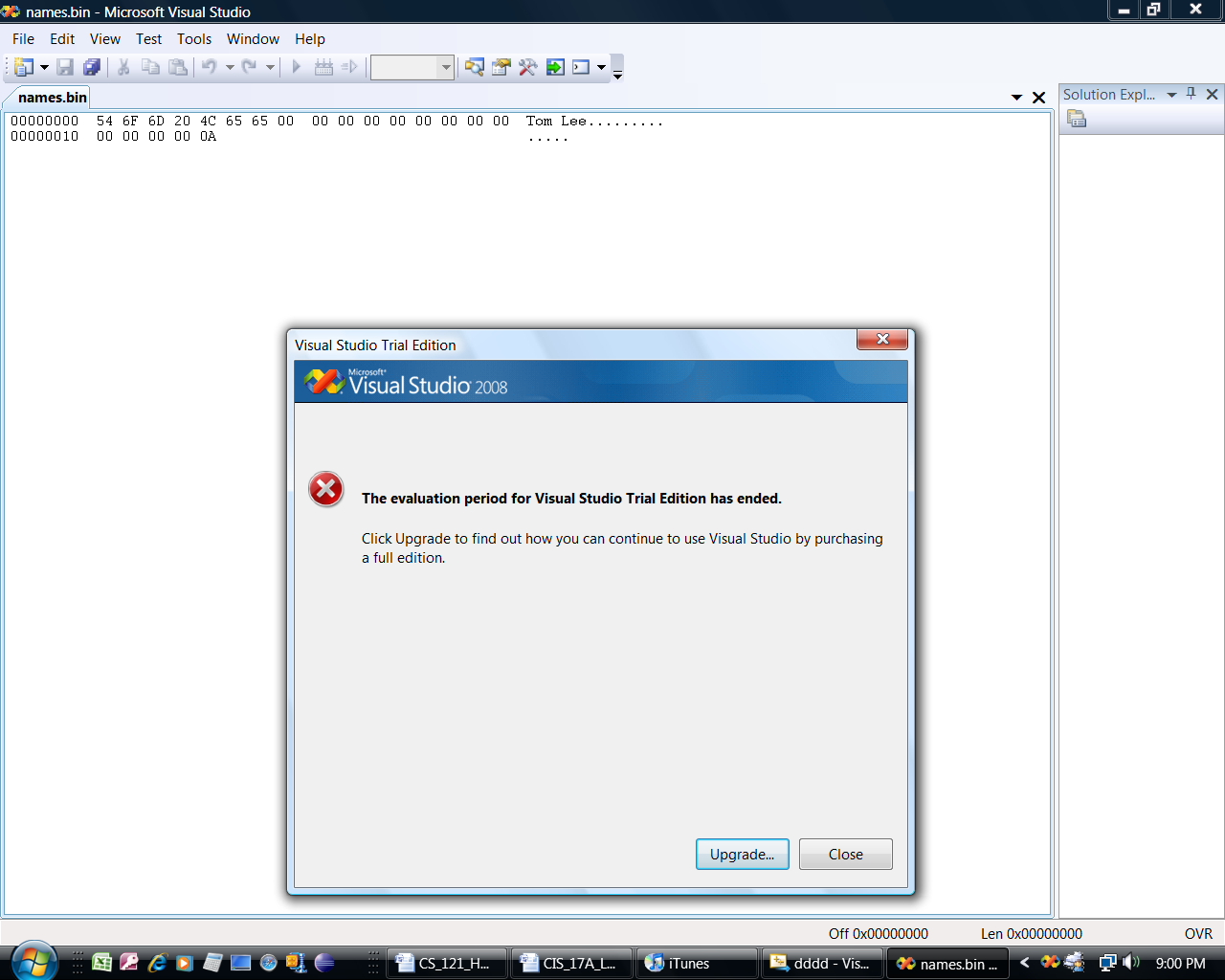
Important: The first parameter of the *write()* function is a pointer to a char variable.

* Therefore, the following works fine, because name is a character array.

Ex 24: char name[20] = “Tom Lee”;

outFile.write(name, sizeof(name); // Pass the address of ***name***,

// and the size of ***name***.



**T o m ‘ ‘ L e e ‘\0’ (NULL)**

(2 Hex digits = 1 character (byte)

**read()** - The **read()** function is defined in <fstream> and is used to read a binary file.

* **read()** - The read() function reads un-interpreted characters.(binary files)
* **Un-interpreted characters** - Include ASCII characters plus other non-printable formatting

characters (provided by such programs as Word).

* The general format is: fileObject.read(address, size);

Important: Like the *write()* function, the first parameter is a pointer to a char variable.

* + Therefore, the following works fine, because name is a character array.

Ex 25: char name[20] = “Tom Lee”

file.read(name, sizeof(name); // Pass the address of ***name***,

// and the size of ***name***.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Ex 26: #include <iostream>

#include <fstream>

using namespace std;

const int SIZE = 4;

int main()

int main()

{

char data[SIZE] = {‘a’,‘b’,‘c’,‘d’};

fstream file;

file.open(“test.bin”, ios::out | ios::binary);

cout << “Writing characters to file.\n”;

file.write(data, sizeof(data));

file.close();

file.open(“test.bin”, ios::in | ios::binary);

cout << “Now reading the data back into memory.\n”;

file.read(data, sizeof(data);

cout << “Here is the data:\n”;

for (int i = 0; i < SIZE; i++)

cout << data[i] << “ ”;

cout << endl;

file.close();

return 0;

}

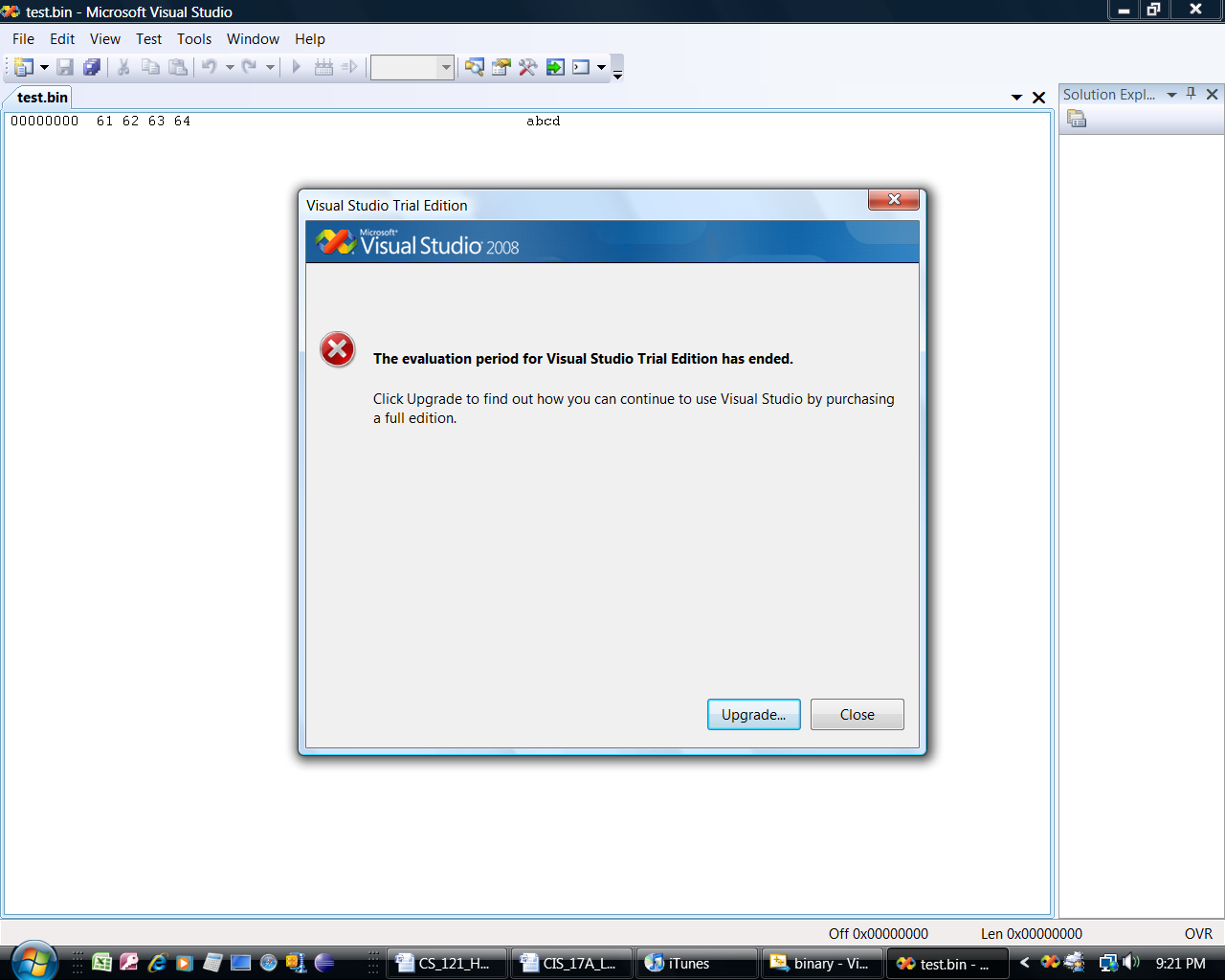
/\* **OUTPUT**

Writing characters to file.

Now reading the data back into memory.

a b c d

**test.bin**



**To read and write data that is not char data type:**

* The first parameter in the *read()* and *write()* functions must be typecast.

Ex 27:

#include <iostream>

#include <fstream>

using namespace std;

const int SIZE = 5;

int main()

Typecast an *int* \* pointer to

a *char* \* pointer, because ages

is an *int* array.

{

int ages[SIZE] = {21,27,13,42,15};

fstream file;

file.open(“data.bin”, ios::out | ios::binary);

cout << “Writing data to file.\n”;

**file.write(reinterpret\_cast<char \*>(ages), sizeof(ages));**

Typecast *int* \* pointer to *cha*r \*

file.close();

file.open(“data.bin”, ios::in | ios::binary);

cout << “Now reading the data back into memory.\n”;

file.read(**reinterpret\_cast<char \*>(ages)**, **sizeof(ages));**

cout << “Here is the data:\n”;

/\* **OUTPUT**

Writing data to file.

Now reading the data back

into memory.

21 27 13 42 15

Press any key to continue.\*/

for (int i = 0; i < SIZE; i++)

cout << ages[i] << “ ”;

file.close();

return 0;

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Note: In the above program, an *int* array is written to a file by typecasting the *int* \* pointer to

a *char* \* pointer.

* If an object of a class or struct is written to a binary file, then the pointer to the object

must be typecast to *char* \*, like this:

Typecast *Person* \* pointer to *char* \*

**Person person;**

**file.write(reinterpret\_cast<char \*>(& person), sizeof(person));**

This also works:

**file.write((char\*)&person, sizeof(person));**

**Sequential File Access** - All previous examples have dealt with sequential file access.

* When a file is opened, the position to read or write is at the beginning of the file,

(unless ios::app is used).

* To read a file sequentially, the ***read()*** function begins reading at the beginning

of the file and continues until done.

* Problem: Sequential file access can be very inefficient, because to read one record in

a file, all records preceding the record must be read first.

Solution: Random-Access of files

**Random File Access** - With this type of file, a program can jump to any byte in the file

without reading the preceding bytes.

**Seeking** – The action of moving to a certain position in a file is called seeking.

* Two **seek** functions are included in C++ stream classes: **seekp()** and **seekg()**
* C++ has a **get pointer** that holds the address where to read the next byte.
* C++ has a **put pointer** that holds the address where to write the next byte.
* The **get** and **put** pointers can be moved by using functions **seekg()** and **seekp()**.
* These 2 functions move the *read / write* position to any byte in a file.
  1. **seekp()** - Use the seekp() function with files opened for output. (p for put)

seekp() moves the **put pointer** to a location before writing to a file.

Syntax: **file.seekp(byteOffset, origin);**

* + **byteOffset** – The number of bytes from the origin.
  + **origin** – The present position of the pointer (beg, end, cur).

Ex 28: **file.seekp(0L, ios::beg);** // Moves the **put** pointer to the

// beginning of the file.

Ex\_29: file.seekp(0L, ios::end); // Moves the put pointer to the end of file

Ex\_30: **file.seekp(20L, ios::cur);** // Moves the put pointer 20 bytes from

// its current position in the file.

**Note:** After moving the *put pointer*, then use the *write()* function to write to the file.

* 1. **seekg()** - Use seekg() function with files opened for input. (g for get)
* seekg() moves the *get pointer* to a location before beginning to read a file.

Syntax: **file.seekg(byteOffset, origin);**

Ex 31: **file.seekg(10L, ios::beg);** // Moves the **get** pointer 10 bytes from

// the beginning

Ex\_32: **file.seekg(-10L, ios::end);** // Moves the **get** pointer 10 bytes from

// the end of the file.

**Note:** After moving the *get pointer*, then use the *read()* function to read the file.

**SUMMARY Binary Files**

* **Declare a file object and open a file**

fstream file("students.bin", ios::out | ios::in | ios::binary);

* **Write to a file** - When a file opens, the "put" pointer points to the beginning of the file.

file.write(reinterpret\_cast<char\*>(&student), sizeof(Student));

* When something is written to a file, both "put" and "get" pointers move.

Therefore, to read the first record in a file, move the get pointer to the beginning of the file

by using the seekg() function.

file.seekg(0L, ios::beg);

* **Read the first record**

file.read(reinterpret\_cast<char\*>(&student), sizeof(Student));

* Assuming the first record exists in a file, it can be overwritten. But first,

the "put" pointer must be moved to the beginning of the file.

file.seekp(sizeof(student), ios::beg);

* **Write a record**

file.write(reinterpret\_cast<char\*>(&s1), sizeof(Student));

* **To write a record at the end, move the "put" pointer to the end.**

file.seekp(0L, ios::end);

**Variable-length file** - A variable-length file is one in which each record in the file can change

in size, and all records do not have to be the same size.

* For example, objects of the following struct could be written to a file, and each object

could be a different size.

* A string class object of string class can vary in size.
* Ex 33: If a person’s name is Bob Jones, the string uses 10 bytes (approximately).

However, if the name is Orenthal James Simpson, the string occupies

about 23 bytes – so the size of the record varies depending on the data.

**Fixed-length file** - In a fixed-length file, all records are the same size.

* For example, objects of the following struct could be written to a file, and each object

would be the same size (158 bytes).

* For c\_strings, the number of bytes of memory or file space is the size of the array.

Ex 34: If a person’s name is Bob Jones, the array uses 50 bytes, even though the

name is only 10 bytes (including the NULL character).

struct student

{

int id; 🡨 4 bytes

char name[50]; 🡨 50 bytes

char address[100]; 🡨 100 bytes

float gpa; 🡨 4 bytes

}; 158 bytes