

File Input and Output

Objectives

- To use ofstream for output and ifstream for input.
- To test whether a file exists.
- To test the end of a file.
- To write data in a desired format.
- To read and write data using the getline, get, and put functions.
- To use an fstream object to read and write data.
- To open a file with specified modes.
- To use the eof(), fail(), bad(), and good() functions to test stream states.
- To understand the difference between text I/O and binary I/O.
- To write binary data using the write function.
- To read binary data using the read function.
- To cast primitive type values and objects to character arrays using the reinterpret_cast operator.
- To read/write objects.
- To use the seekp and seekg functions to move the file pointers for random file access.
- To open a file for both input and output to update files.

Writing Data to a File


The ofstream class can be used to write primitive data type values, arrays, strings, and objects to a text file. Example below demonstrates how to write data. The program creates an instance of ofstream and writes two lines to the file “scores.txt”. Each line consists of first name (a string), middle name initial (a character), last name (a string), and score (an integer).

TextFileOutput

Writing Data to a File

```
output << "John" << " " << "T" << "Smith" << " " << 90 << endl;
```

scores.txt
file



```
John T Smith 90  
Eric K Jones 85
```

```
output << "Eric" << " " << "K" << "Jones" << " " << 85 << endl;
```

close file

The close() function (line 18) must be used to close the stream for the object. If this function is not invoked, the data may not be saved properly in the file.

file exists?

If a file already exists, the contents of the file will be destroyed without warning.

absolute filename

Every file is placed in a directory in the file system. An *absolute file* name contains a file name with its complete path and drive letter. For example, **c:\example\scores.txt** is the absolute file name for the file **scores.txt** on the Windows operating system. Here **c:\example** is referred to as the *directory path* for the file. Absolute file names are machine-dependent. On Unix, the absolute file name may be **/home/maleki/example/scores.txt**, where **/home/maleki/example** is the directory path for the file **scores.txt**.

\ in file names

The directory separator for Windows is a backslash (\). The backslash is a special character in C++ and should be written as \\ in a string literal.

For example,

```
output.open("c:\\example\\scores.txt");
```


relative filename

Absolute file name is platform dependent. It is better to use relative file name without drive letters. The directory of the relative filename can be specified in the IDE if you use an IDE to run C++.

In Visual Studio C++ you would write:

```
..\scores.txt
```

Which means the directory above project directory.

Reading Data from a File

The ifstream class can be used to read data from a text file. Example below demonstrates how to read data. The program creates an instance of ifstream and reads data from the file scores.txt. scores.txt was created in the preceding example.

TextFileInput

Testing File Existence

If the file does not exist, your program will run and produce incorrect results. Can your program check whether a file exists? Yes. You can invoke the fail() function immediately after invoking the open function. If fail() returns true, it would indicate that the file does not exist.

Can we use exceptions? How?

```
// Open a file
```

```
try {
```

```
    input.open("scores.txt");
```

```
    if (input.fail())
```

```
        throw std::exception("Could not open file");
```

```
}
```

```
catch (std::exception &ex) {
```

```
    cout << ex.what() << endl;
```

```
    //return -1;
```

```
}
```

```
//Maybe a little bit of overuse?!
```

know data format

To read data correctly, you need to know exactly how data is stored. For example, the program in earlier slide would not work if the score is a double value with a decimal point.

Testing End of File

TestEndOfFile reads two lines from the data file. If you don't know how many lines are in the file and want to read them all, how do you know the end of file? You can invoke the eof() function on the input object to detect it. The program reads numbers from file numbers.txt and displays their sum.

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|--|---|---|---|---|--|---|---|---|---|----|
| 9 | 5 | . | 5 | | 6 | | 7 | 0 | . | 2 | | 1 | . | 5 | 5 | \n |
| 1 | 2 | | 3 | . | 3 | | 1 | 2 | . | 9 | | 8 | 5 | . | 6 | |



Reads beyond end of file

TestEndOfFile

Formatting Output

WriteFormatData gives an example that formats the student records to the file named formattedscores.txt.

WriteFormatData

getline, get and put

There is a problem to read data using the stream extraction operator (>>). Data are delimited by whitespace. What happens if the whitespace characters are part of a string? You have learned how to use the getline function to read a string with whitespace. You can use the same function to read strings from a file. Recall that the syntax for the getline function is

```
getline(istream& input, string s, char delimiterChar)
```

ReadCity

getline, get and put

Two other useful functions are get and put. You can invoke the get function on an input object to read a character and invoke the put function on an output object to write a character.

CopyFile

fstream and File Open Modes

In the preceding sections, you used the ofstream to write data and the ifstream to read data.

Alternatively, you can also use the fstream class to create an input stream or output stream. It is convenient to use fstream if your program needs to use the same stream object for both input and output. To open an fstream file, you have to specify a file mode to tell C++ how the file will be used. The file modes are listed in next slide.

File Open Modes

| Mode | Description |
|--------------------------|---|
| <code>ios::in</code> | Opens a file for input. |
| <code>ios::out</code> | Opens a file for output. |
| <code>ios::app</code> | Appends all output to the end of the file. |
| <code>ios::ate</code> | Opens a file for output. If the file already exists, move to the end of the file. Data can be written anywhere in the file. |
| <code>ios::trunc</code> | Discards the file's contents if the file already exists. (This is the default action for <code>ios::out</code>). |
| <code>ios::binary</code> | Opens a file for binary input and output. |

Combining Modes

Several modes can be combined together using the `|` operator. This is a bitwise inclusive OR operator. For example, to open an output file named `city.txt` for appending data, you can use the following statement:

```
fstream.open("city.txt", ios::out | ios::app);
```

AppendFile

Testing Stream States

You have used the eof() function and fail() function to test the states of a stream. C++ provides several more functions in a stream for testing stream states. Each stream object contains a set of bits that act as flags. These bit values (0 or 1) indicate the state of a stream. Next slide lists these bits.

Stream State Bit Values

| Bit | Description |
|----------------------------|---|
| <code>ios::eofbit</code> | Set when the end of an input stream is reached. |
| <code>ios::failbit</code> | Set when an operation failed. |
| <code>ios::hardfail</code> | Set when an unrecoverable error occurred. |
| <code>ios::badbit</code> | Set when an invalid operation has been attempted. |
| <code>ios::goodbit</code> | Set when an operation is successful. |

Stream State Functions

| Function | Description |
|----------------------|---|
| <code>eof()</code> | Returns true if the eofbit flag is set. |
| <code>fail()</code> | Returns true if the failbit or hardfail flags is set. |
| <code>bad()</code> | Returns true if the badbit is set. |
| <code>good()</code> | Returns true if the goodbit is set. |
| <code>clear()</code> | Clears all flags. |

ShowStreamState

Binary I/O

Although it is not technically precise and correct, you can envision a text file as consisting of a sequence of characters and a binary file as consisting of a sequence of bits. For example, the decimal integer 199 is stored as the sequence of three characters, '1', '9', '9', in a text file, and the same integer is stored as a byte-type value C7 in a binary file, because decimal 199 equals hex C7 ($199 = 12 * 16 + 7$).

Text vs. Binary I/O

Computers do not differentiate binary files and text files. All files are stored in binary format, and thus all files are essentially binary files. Text I/O is built upon binary I/O to provide a level of abstraction for character encoding and decoding.

ios::binary

Binary I/O does not require conversions. If you write a numeric value to a file using binary I/O, the exact value in the memory is copied into the file. To perform binary I/O in C++, you have to open a file using the binary mode ios::binary. By default, a file is opened in text mode.

You used the << operator and put function to write data to a text file and the >> operator, get, and getline functions to read data from a text file. To read/write data from/to a binary file, you have to use the read and write functions on a stream.

The write Function

The syntax for the write function is

```
streamObject.write(char* bytes, int size)
```

BinaryCharOutput shows an example of using the write function.

BinaryCharOutput

Write Any Type

Often you need to write data other than characters. How can you accomplish it? C++ provides the reinterpret_cast for this purpose. You can use this operator to cast the address of a primitive type value or an object to a character array pointer for binary I/O. The syntax of this type of casting is:

reinterpret_cast<dataType>(address)

where address is the starting address of the data (primitive, array, or object) and dataType is the data type you are converting to. In this case for binary I/O, it is char *.

BinaryIntOutput

Reinterpret_cast Demo

ReinterpretCastingDemo

The read Function

The syntax for the read function is

```
streamObject.read(char* address, int size)
```

Assume we have a file city.dat.

BinaryCharInput reads the characters using the read function.

BinaryCharInput

Read Any Type

Similarly, to read data other than characters, you need to use the reinterpret_cast operator. Assume we have a file temp.dat.

BinaryIntInput reads the integer using the read function.

BinaryIntInput

Binary Array I/O

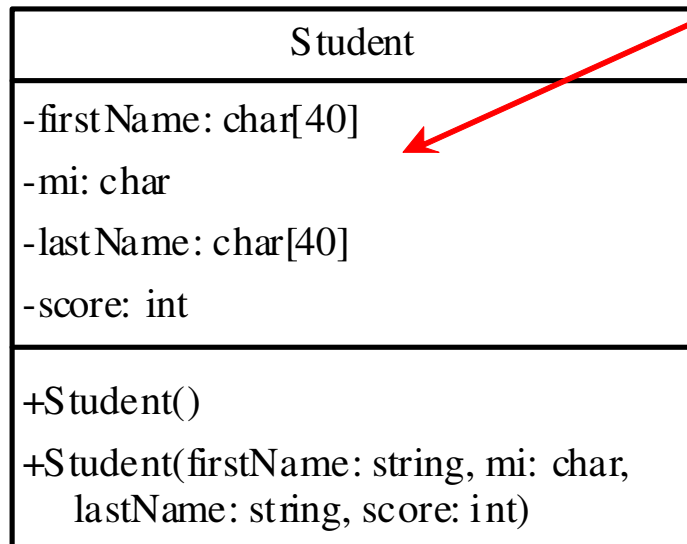
This section gives an example to write an array of double values to a binary file, and read it back from the file.

BinaryArrayIO

Binary Object I/O

We did write student records into a text file. A student record consists of first name, middle name initial, last name, and score. These fields are written to the file separately. A better way to process it is to declare a class to model records. Each record is an object of the Student class.

Binary Object I/O, cont.



The *get* and *set* functions for these data fields are provided in the class but for brevity omitted in the UML diagram.

The first name of this student.

The middle initial of this student.

The last name of this student.

The score of this student.

Constructs a default Student object.

Constructs a student with specified first name, mi, last name, and score

Student.h

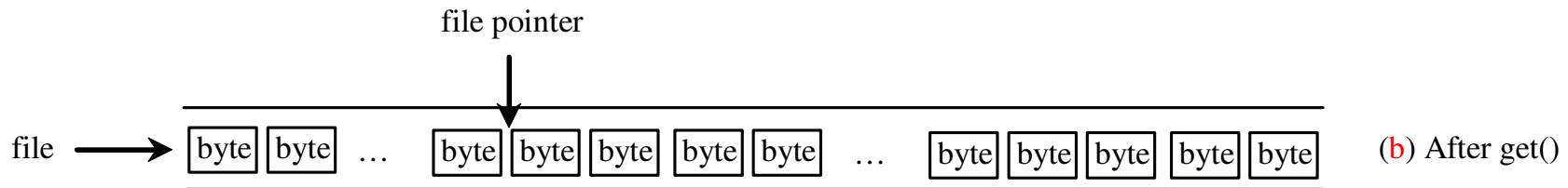
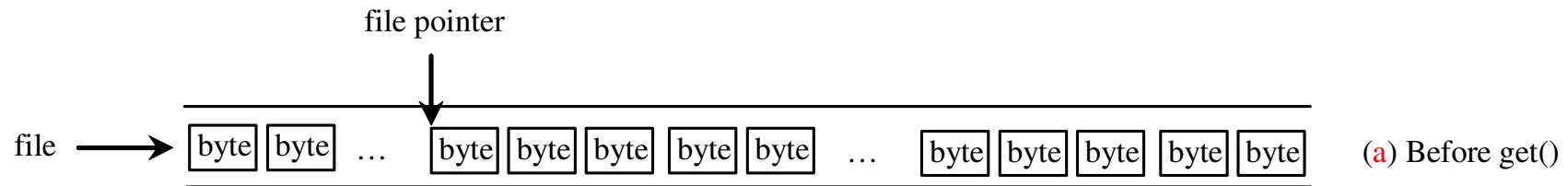
Student.cpp

BinaryObjectIO

Random Access File

A file consists of a sequence of bytes. There is a special marker called *file pointer* that is positioned at one of these bytes. A read or write operation takes place at the location of the file pointer. When a file is opened, the file pointer is set at the beginning of the file. When you read or write data to the file, the file pointer moves forward to the next data item. For example, if you read a character using the get() function, C++ reads one byte from the file pointer and now the file pointer is 1 byte ahead of the previous location.

Random Access File, cont



seekp, seekg, tellp, tellg

| Seek Base | Description |
|-----------------------|---|
| <code>ios::beg</code> | Calculates the offset from the beginning of the file. |
| <code>ios::end</code> | Calculates the offset from the end of the file. |
| <code>ios::cur</code> | Calculates the offset from the current file pointer. |

| Statement | Description |
|--------------------------------------|---|
| <code>seekg(100L, ios::beg);</code> | Moves the file pointer to the 100 th byte from the beginning of the file. |
| <code>seekg(-100L, ios::end);</code> | Moves the file pointer to the 100 th byte backward from the end of the file. |
| <code>seekp(42L, ios::cur);</code> | Moves the file pointer to the 42 nd byte forward from the current file pointer. |
| <code>seekp(-42L, ios::cur);</code> | Moves the file pointer to the 42 nd byte backward from the current file pointer. |
| <code>seekp(100L);</code> | Moves the file pointer to the 100 th byte in the file. |

Random Access File Example

This example demonstrates how to access file randomly. The program first stores 10 student objects into the file, and then retrieves the 3rd student from the file.

RandomAccessFile

Updating Files

This example demonstrates how to update a file. Suppose file `object1.dat` has already been created with ten Student objects from earlier slide. The program first reads the 3rd student from the file, changes the last name, writes the revised object back to the file, and reads the new object back from the file.

UpdateFile