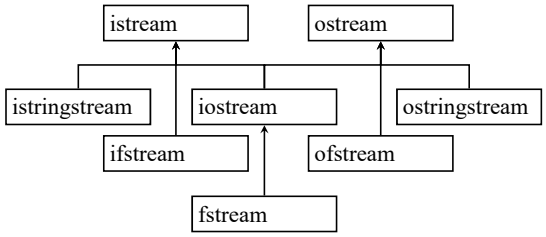
CISC/CMPE320

- Notices:
- Assignment 1 due next Friday at 7pm. The rest of the assignments will also be moved ahead a week.
- Teamwork: Let me know who the "team leader" is and of any more membership changes. I know there are problems using bitbucket and hipchat. Let me know if you cannot log in.

Today

- File I/O – Text, Random and Binary.
- A Text File I/O Demo.
- Re-Structure the Demo:
  - Separate Declaration and Implementation.
  - Create a Simple Class.

ISO C++ Stream Hierarchy



- Note that iostream extends both istream and ostream.

File I/O

- We've been including the iostream library for console I/O.
- For file I/O: `#include <fstream>`
- So, fstream inherits all the functionality that we are used to from iostream and adds additional functions like **`open()`** and **`close()`**.

Text File I/O – Reading

- Two ways to open a file for reading. Assume **`filename`** is a **`string`**:

```
ifstream fileIn;
fileIn.open(filename.c_str());
```

- Or:

```
ifstream fileInAgain(filename.c_str());
```

- (Or you can supply a **`char*`** literal).

Text File I/O – Reading, Cont.

- The filename can contain a drive letter and/or folder names – a path.
- If it is a Unix path you are OK, but for a Windows path, remember to use \\ as a folder delimiter.
- If no path is supplied, the open() function looks in the same folder as your source code (not the *.exe folder…).

Text File I/O – Reading, Cont.

- Use the **`fail()`** function to check to make sure the file has opened OK:

```
if (fileIn.fail()) {
  cerr << "Unable to open file: " << filename << endl;
  return 1;
}
```


Text File I/O – Reading, Cont.

- To read a **`char`** use the **`get()`** function:
  `fileIn.get(aChar);`
- To read a **`string`**, use **`getline()`**:

```
string line;
getline(fileIn, line);
```

- **`getline()`** returns **`false`** (or 0) when there is nothing to read in.

Text File I/O – Reading, Cont

- Other ways to check for the end of the file, if you don't know how much to read in advance:

```
while(!fileIn.eof())
```

- Note that **`eof()`** returns **`true`** after you have tried to read past the end of the file.
- Turns out that **`>>`** also returns a boolean **`false`** if there is nothing to be read:

```
while(fileIn >> aWord)
```


Text File I/O – Reading, Cont.

- You can also use the re-direction operators, just as you would with screen input:

```
int aVal;
fileIn >> aVal;
```

- Reads an **`int`** value into **`aVal`**.
- And when you are finished:

```
fileIn.close();
```


Text File I/O – Reading, Cont.

- Note that the **`>>`** operator reads what is expected by examining the type of the variable supplied.
- It ignores whitespace (tabs, spaces and linefeeds).
- So, if you wish to read whitespace, you will need to use a "get…" method instead.

Text File I/O – Writing

- To create the stream:

```
ofstream outFile("TestOut.txt");
```

- Use the **`<<`** operator to write to **`outFile`** and then **`close()`** the file, when done.
- To append to a file:

```
ofstream outFile("TestOut.txt", ios::app);
```

- **`ios::app`** is a constant defined in the iostream library, so no new includes are required.

Text File I/O – Writing, Cont.

- You can also use the **`fail()`** function to see if you have had problems opening the file (no write privilige, bad folder, etc.).
- And, remember the stuff you did in part 3 of exercise 1 (stream manipulators)? This is how you can format things for output.

Random File Access

- Text I/O is all sequential access – unidirectional.
- To open a file for input and output at the same time and to read any position in any order, you need to use random I/O.
- Same library: fstream.

Random File Access, Cont.

- To open for input and output, for example:

```
fstream rwStream;
rwStream.open("stuff.txt", ios::in | ios::out);
```

- Last time we used ifstream for input and ofstream for output. fstream extends both of these classes, allowing you to do input and output at the same time.

Random File Access, Cont.

- File open modes:

Mode	Meaning
`ios::app`	Append to end of file and you cannot move anywhere else.
`ios::ate`	Starts like append, but then you can move anywhere in the file.
`ios::in`	Open for input.
`ios::out`	Open for output.
`ios::trunc`	Discard contents (default behavior for ios::out).
`ios::binary`	Open for binary I/O.

- Combine modes using the binary "or" |.

Binary Random File Access

- Imagine two pointers in the file - a read pointer ("get") and a write pointer ("put").
- To read or write you first have to position the pointer at a certain byte position. For example:

```
rwStream.seekp(1000, ios::beg);
```

- Positions the write pointer 1000 bytes from the beginning of the file.

Binary Random File Access, Cont.

```
rwStream.seekg(1000, ios::end);
```

- Positions the read pointer 1000 bytes from the end of the file.
- You can also use **`ios::cur`** to position the pointer in relative terms.
- After the seek, you read and write the normal way.
- But how to know where to position the "pointers"?

Binary Random File Access, Cont.

- This only works when you have a structured file.
- You need to know what is stored, in which order it is stored and how big each item is.
- Remember that you can use the **`sizeof()`** operator to get the size of anything – primitive types or objects.
- This way you could write an entire object to the file and read it back the same way.

Text File I/O – Example

- See TextIODemo.cpp
- Stuff that also snuck into this demo:
  - Use of vector and string classes.
  - Function prototypes.
  - Passing by constant reference.
- We will also look at an OOP version of this program.

Function Prototypes

- Allow you to implement functions in any order, since the function names have already been declared.
- Easier to see the structure of what you are coding.
- Parameters must be typed – optionally supply the parameter name for style.
- Next, the declarations will be placed in a separate file from the implementation.

File I/O Demo as a Class

- Restrucure TextIODemo.cpp by separating out a header file and an implementation file.
- Might as well make a simple class out of the result.
- See:
  - textfileio.h
  - textfileio.cpp
  - TestFileIOClass.cpp

Header File - textfileio.h

- This header file only contains the declaration of the TextFileIO class, but it could also have contained:
  - enums
  - structs
  - non-member function prototypes
  - other class declarations
  - constants
  - documentation
- Putting any implementation in the header file is considered poor structure (*and poor style*).

Declaration in Separate File

- Now, the implementation can be completely hidden from anyone using your class.
- You should be able to use the class without knowing anything about the implementation!

textfileio.h, Cont.

- Note the use of **`const`** in the member function declaration:

```
vector<string> readFile() const;
```

- This contract promises that the member function will not change member variables (attributes).
- Optional, but good programming practice, particularly for accessors.

textfileio.h, Cont.

- `#pragma once` ensures that the declarations in this file will only be made once.
- The *.h file will be included in any file that is going to use this class using:

```
#include "textfileio.h"
```

- You can have as many of these #includes as you want in a project without worry!

Class Declaration

- The **`public:`** and **`private:`** sections control access to class members from instantiations.
- As you would expect, encapsulation dictates that your attributes are declared in the **`private:`** section!

Implementation File – textfileio.cpp

- Has the same name (different extension) as the header file, by convention.
- Implements member and non – member functions.
- Few comments, or just comments for the developer. Users are not going to see this file.

textfileio.cpp, Cont.

- The constructor:

```
TextFileIO::TextFileIO(const string& fname) :
filename(fname) {}
```

- Note the “shortcut” notation in the initialization section.
- You can still do things the old-fashioned way, especially if you are going to check the arguments for legality.

textfileio.cpp, Cont.

- Also note the membership operator ::
- It allows you to associate the member with the class.
- You can implement members or non-member functions for any header file that you have included.

Aside - `private` Members

- Member function definitions and their implementations can access **`private`** members – even if this is in a different file.
- Non-member functions cannot access **`private`** members, only **`public`** ones.

TestFileIOClass.cpp

- Some class in your project must have a main function, or your application will not run.
- (But only one main function per project!)
- TextFileIO is instantiated on the run-time stack (more about this later) in the usual way.
- You can only access public: members from here.

Default Constructors

- Invoked without parameters.
- If we had one for TextFileIO it would be invoked as:

  `TextFileIO test;` No round brackets!

- What does: **`TextFileIO test();`** look like to you?