CS217 OBJECT ORIENTED PROGRAMMING

Spring 2020

STREAMS IN C++

- A *stream* is a general name given to a flow of data.
- In C++ a stream is represented by an object of a particular class.
- So far we've used the **cin** and **cout** stream objects.
- Different streams are used to represent different kinds of data flow.

Advantages of Streams

- One reason is simplicity.
  - If you've ever used a **%d** formatting character when you should have used a **%f** in **printf()**, you'll appreciate this.
  - There are no such formatting characters in streams, since each object already knows how to display itself.
- Another reason is that you can overload existing operators and functions, such as the insertion (<<) and extraction (>>) operators, to work with classes that you create.

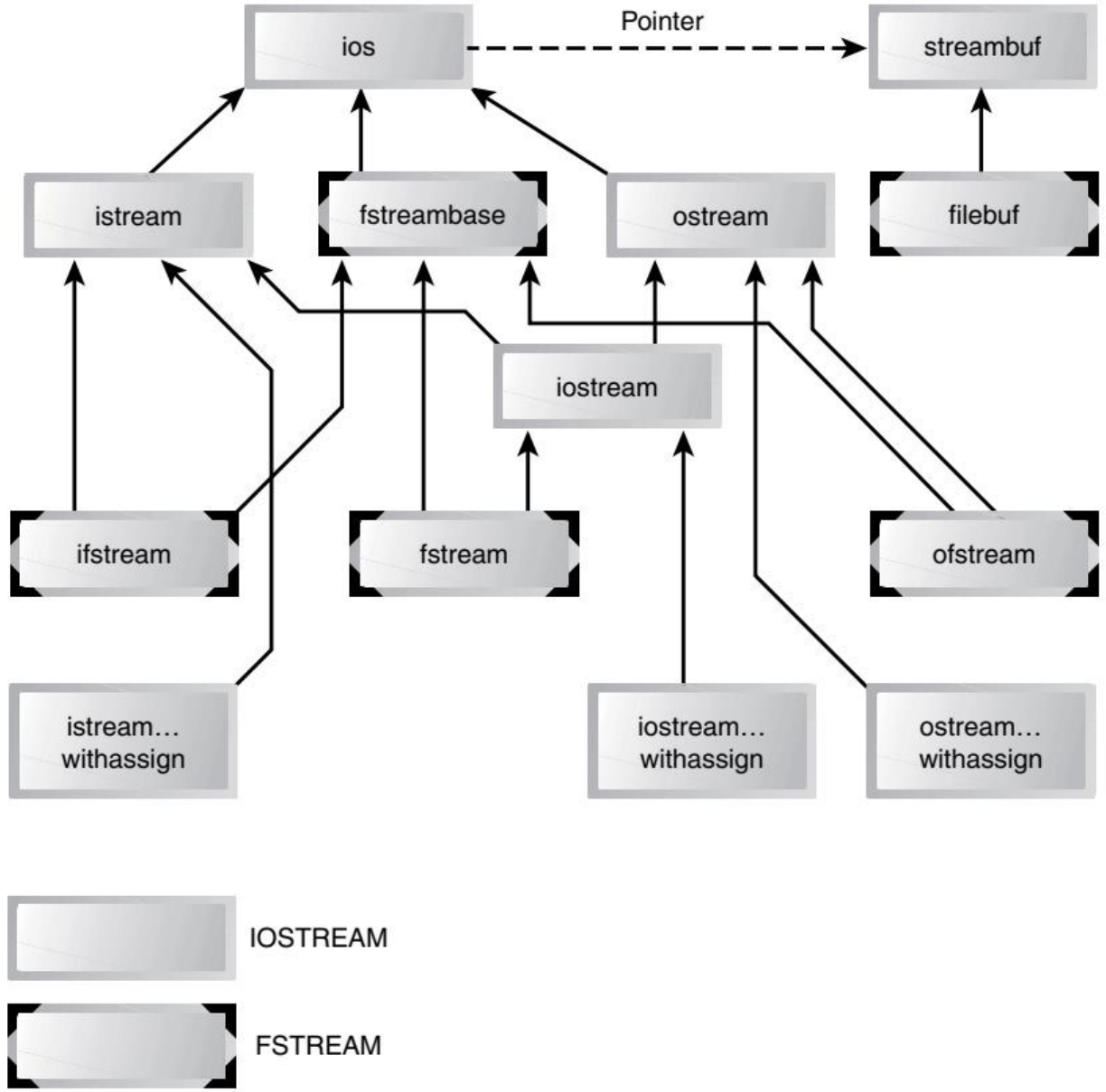

FIGURE 12.1

Stream class hierarchy.

STREAMS IN C++

- We've already made extensive use of some stream classes. The extraction operator **>>** is a member of the **istream** class, and the insertion operator **<<** is a member of the **ostream** class.
  - Both of these classes are derived from the **ios** class

- The ios class is the *granddaddy* of all the stream classes, and contains the majority of the features you need to operate C++ streams.

- **Manipulators** are formatting instructions inserted directly into a stream.
  - We've seen examples before, such as the manipulator endl, which sends a newline to the stream and flushes it:

```
cout << "To each his own." << endl;
```

THE istream CLASS

- The **istream** class, which is derived from **ios**, performs input-specific activities, or extraction.

TABLE 12.6 `istream` Functions

Function	*Purpose*
`>>`	Formatted extraction for all basic (and overloaded) types.
`get(ch);`	Extract one character into `ch`.
`get(str)`	Extract characters into array `str`, until '`\n`'.

THE ostream CLASS

- The **ostream** class handles output or insertion activities.

TABLE 12.7 ostream Functions

Function	*Purpose*
`<<`	Formatted insertion for all basic (and overloaded) types.
`put(ch)`	Insert character `ch` into stream.
`flush()`	Flush buffer contents and insert newline.
`write(str, SIZE)`	Insert `SIZE` characters from array `str` into file.
`seekp(position)`	Set distance in bytes of file pointer from start of file.
`seekp(position, seek_dir)`	Set distance in bytes of file pointer, from specified place in file. `seek_dir` can be `ios::beg`, `ios::cur`, or `ios::end`.
`pos = tellp()`	Return position of file pointer, in bytes.

DISK FILE I/O WITH STREAMS (FILING)

- Most programs need to save data to disk files and read it back in.
- Working with disk files requires another set of classes: **ifstream** for input, **ofstream** for output and **fstream** for both input and output.
- Objects of these classes can be associated with disk files, and we can use their member functions to read and write to the files .

WRITING DATA (TO DISK FILES)

```
#include <fstream>              //for file I/O
#include <iostream>
#include <string>
using namespace std;

int main(){
    char ch = 'x';
    int j = 77;
    double d = 6.02;
    string str1 = "Kafka";
    string str2 = "Proust";

    ofstream outfile(“fdata.txt"); //create ofstream object

    outfile<< ch   << j << ' ' << d<< str1 << ' '<< str2;

    cout << "File written\n";

    return 0;
    }
```

READING DATA (FROM DISK FILE)

```
#include <fstream>                          //for file I/O
#include <iostream>
#include <string>
using namespace std;

int main(){
        char ch;
        int j;
        double d;
        string str1;
        string str2;
        ifstream infile("fdata.txt");    //create ifstream object

        infile >> ch >> j >> d >> str1 >> str2;
        cout << ch << endl << j << endl<< d << endl<< str1 << endl<< str2 << endl;
        return 0;
        }
```

STRINGS WITH EMBEDDED BLANKS

- The technique of our last examples won't work with char* strings containing embedded blanks.

- To handle such strings, you need to write a specific delimiter character after each string, and use the **getline()** function, rather than the extraction operator, to read them in.

- To extract the strings from the file, we create an **ifstream** and read from it one line at a time using the **getline()** function, which is a member of **istream**. This function reads characters, including whitespace, until it encounters the '\n' character, and places the resulting string in the buffer supplied as an argument

```
#include <fstream>
using namespace std;
int main()
{
    ofstream outfile("TEST.TXT");
    outfile << "I fear thee, ancient Mariner!\n";
    outfile << "I fear thy skinny hand\n";
    outfile << "And thou art long, and lank, and brown,\n";
    outfile << "As is the ribbed sea sand.\n";

return 0;
}
```

```
#include <fstream>
#include <iostream>
using namespace std;
int main()
{
    const int MAX = 80;                 //size of buffer
    char buffer[MAX];                   //character buffer

    ifstream infile("TEST.TXT");        //create file for input

    while( !infile.eof() )              //until end-of-file
    {
        infile.getline(buffer, MAX); //read a line of text
        cout << buffer << endl;         //display it
    }
return 0;
}
```

CHARACTER I/O

- The **put()** and **get()** functions, which are members of **ostream** and **istream**, respectively, can be used to output and input single characters

```
#include <fstream>                      //for file functions
#include <iostream>
#include <string>
using namespace std;

int main()
   {
   string str = "Time is a great teacher, but unfortunately "
                "it kills all its pupils.  Berlioz";

   ofstream outfile("TEST.TXT");        //create file for output
   for(int j=0; j<str.size(); j++)      //for each character,
      outfile.put( str[j] );            //write it to file
   cout << "File written\n";
   return 0;
   }
```

```
int main()
   {
   char ch;                          //character to read
   ifstream infile("TEST.TXT");      //create file for input
   while( infile )                   //read until EOF or error
      {
      infile.get(ch);                //read character
      cout << ch;                    //display it
      }
   cout << endl;
   return 0;
   }
```

- Another approach to reading characters from a file is the **rdbuf()** function, a member of the **ios** class.

- This function returns a pointer to the **streambuf** (or **filebuf**) object associated with the stream object.

- This object contains a buffer that holds the characters read from the stream, so you can use the pointer to it as a data object in its own right.

```
// ichar2.cpp
// file input with characters

#include <fstream> //for file functions
#include <iostream>
using namespace std;

int main()
{
    ifstream infile("TEST1.TXT");    //create file for input
    cout << infile.rdbuf();          //send its buffer to cout
    cout << endl;
    infile.close();       //
return 0;
}
```

CLOSING FILES

- So far in our example programs there has been no need to close streams explicitly because they are closed automatically when they go out of scope; this invokes their destructors and closes the associated file.
- However, if both the output stream os and the input stream is are associated with the same file, the first stream must be closed before the second is opened.
- We use the **close()** member function for this

```
#include <fstream> //for file functions
#include <iostream>
#include <string>
using namespace std;
int main()
{
string str = "Time is a great teacher, but unfortunately "
"it kills all its pupils. Berlioz";

ofstream outfile("TEST1.TXT");       //create file for output
for(int j=0; j<str.size(); j++)      //for each character,
     outfile.put( str[j] );          //write it to file

cout << "File written\n";

outfile.close();

char ch;                             //character to read
ifstream infile("TEST1.TXT");        //create file for input
while(infile )                       //read until EOF or error
{
     infile.get(ch);                 //read character
     cout << ch;                     //display it
}
cout << endl;

return 0;
}
```

BINARY I/O

- You can write a few numbers to disk using formatted I/O, but if you're storing a large amount of numerical data it's more efficient to use **binary I/O**, in which numbers are stored as they are in the computer's RAM memory.
- In binary I/O an **int** is stored in 4 bytes, **float** in 4 bytes ..etc.
- Our next example shows how an array of integers is written to disk and then read back into memory, using binary format.
- We use two new functions: **write()**, a member of **ofstream**; and **read()**, a member of **ifstream**.
- These functions think about data in terms of **bytes** (type char).
- They don't care how the data is formatted, they simply transfer a buffer full of bytes from and to a disk file

- The parameters to **write()** and **read()** are the address of the data buffer and its length.
  - The address must be cast, using **reinterpret_cast**, to type **char***, and the length is the length in bytes, *not* the number of data items in the buffer.

```
// binio.cpp
// binary input and output with integers
#include <fstream>                      //for file streams
#include <iostream>
using namespace std;
const int MAX = 100;                    //size of buffer
int buff[MAX];                          //buffer for integers
int main()
{
     for(int j=0; j<MAX; j++)           //fill buffer with data
          buff[j] = j; //(0, 1, 2, ...)

//create output stream
ofstream os("edata.dat", ios::binary);

//write to it
os.write( reinterpret_cast<char*>(buff), MAX*sizeof(int));
os.close();                    //must close it

for(int j=0; j<MAX; j++) //erase buffer
     buff[j] = 0;

//create input stream
ifstream is("edata.dat", ios::binary);
//read from it
is.read(reinterpret_cast<char*>(buff), MAX*sizeof(int) );
for(int j=0; j<MAX; j++) //check data
if( buff[j] != j )
{
     cerr << "Data is incorrect\n"; return 1;
}
     cout << "Data is correct\n";
return 0;
}
```

- You must use the **ios::binary** argument in the second parameter to **write()** and **read()** when working with binary data.

- In the BINIO program we use the **reinterpret_cast** operator to make it possible for a buffer of type int to look to the **read()** and **write()** functions like a buffer of type **char**.

OBJECT I/O: WRITING OBJECT TO DISK FILE

```
// opers.cpp
// saves person object to disk
#include <fstream>                //for file streams
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class person                      //class of persons
   {
   protected:
      char name[80];              //person's name
      short age;                  //person's age
   public:
      void getData()              //get person's data
         {
         cout << "Enter name: "; cin >> name;
         cout << "Enter age: "; cin >> age;
         }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   person pers;                   //create a person
   pers.getData();                //get data for person
                                  //create ofstream object
   ofstream outfile("PERSON.DAT", ios::binary);
                                  //write to it
   outfile.write(reinterpret_cast<char*>(&pers), sizeof(pers));
   return 0;
   }
```

OBJECT I/O: READING OBJECT FROM DISK

```
#include <fstream>                //for file streams
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class person                      //class of persons
   {
   protected:
      char name[80];              //person's name
      short age;                  //person's age
   public:
      void showData()             //display person's data
         {
         cout << "Name: " << name << endl;
         cout << "Age: " << age << endl;
         }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   person pers;                   //create person variable
   ifstream infile("PERSON.DAT", ios::binary); //create stream
                                               //read stream
   infile.read( reinterpret_cast<char*>(&pers), sizeof(pers) );
   pers.showData();                            //display person
   return 0;
   }
```

COMPATIBLE DATA STRUCTURES

- To work correctly, programs that read and write objects to files, as do **OPERS** and **IPERS**, must be talking about the same class of objects.

- Objects of class person in these programs are exactly 82 bytes long: The first 80 are occupied by a string representing the person's name, and the last 2 contain an integer of type short, representing the person's age.

- If two programs thought the name field was a different length, for example, neither could accurately read a file generated by the other.

```
// diskfun.cpp
// reads and writes several objects to disk
#include <fstream>                //for file streams
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class person                      //class of persons
   {
   protected:
      char name[80];              //person's name
      int age;                    //person's age
   public:
      void getData()              //get person's data
         {
         cout << "\n   Enter name: "; cin >> name;
         cout << "   Enter age: "; cin >> age;
         }
      void showData()             //display person's data
         {
         cout << "\n   Name: " << name;
         cout << "\n   Age: " << age;
         }
   };
```

```
int main()
   {
   char ch;
   person pers;                   //create person object
   fstream file;                  //create input/output file
                                  //open for append
   file.open("GROUP.DAT", ios::app | ios::out |
                                      ios::in | ios::binary );
   do                             //data from user to file
      {
      cout << "\nEnter person's data:";
      pers.getData();             //get one person's data
                                  //write to file
      file.write( reinterpret_cast<char*>(&pers), sizeof(pers) );
      cout << "Enter another person (y/n)? ";
      cin >> ch;
      }
   while(ch=='y');                //quit on 'n'
   file.seekg(0);                 //reset to start of file
                                  //read first person
   file.read( reinterpret_cast<char*>(&pers), sizeof(pers) );
   while( !file.eof() )           //quit on EOF
      {
      cout << "\nPerson:";        //display person
      pers.showData();            //read another person
      file.read( reinterpret_cast<char*>(&pers), sizeof(pers) );
      }
   cout << endl;
   return 0;
   }
```

- In **DISKFUN** we want to create a file that can be used for both input and output.

- This requires an object of the **fstream** class, which is derived from **iostream**, which is derived from both **istream** and **ostream** so it can handle both input and output.

- In DISKFUN we use a different approach: We create the file in one statement and open it in another, using the **open()** function, which is a member of the fstream class.
  - You can create a stream object once, and then try repeatedly to open it, without the overhead of creating a new stream object each time.

- We've seen the mode bit **ios::binary** before. In the open() function we include several new mode bits.
- The mode bits, defined in ios, specify various aspects of how a stream object will be opened.

TABLE 12.10 Mode Bits for the `open()` Function

Mode Bit	*Result*
`in`	Open for reading (default for `ifstream`)
`out`	Open for writing (default for `ofstream`)
`ate`	Start reading or writing at end of file (`AT End`)
`app`	Start writing at end of file (`APPend`)
`trunc`	Truncate file to zero length if it exists (`TRUNCate`)
`nocreate`	Error when opening if file does not already exist
`noreplace`	Error when opening for output if file already exists, unless `ate` or `app` is set
`binary`	Open file in binary (not text) mode

- In DISKFUN we use ios::app because we want to preserve whatever was in the file before.
  - That is, we can write to the file, terminate the program, and start up the program again, and whatever we write to the file will be added following the existing contents.

- We use **ios:in** and **ios:out** because we want to perform both input and output on the file, and we use **ios:binary** because we're writing binary objects.

- We write one person object at a time to the file, using the **write()** function.

- When we've finished writing, we want to read the entire file. Before doing this we must reset the file's current position. We do this with the **seekg()** function.

FILE POINTERS

- Each file object has associated with it two integer values called the ***get pointer*** and the ***put pointer***.

- These are also called the *current get position* and the *current put position*, or—if it's clear which one is meant—simply the ***current position.***

- These values specify the byte number in the file where writing or reading will take place .

- The **seekg()** and **tellg()** functions allow you to set and examine the get pointer, and the **seekp()** and **tellp()** functions perform these same actions on the put pointer.