CS217 OBJECT ORIENTED PROGRAMMING

Spring 2020

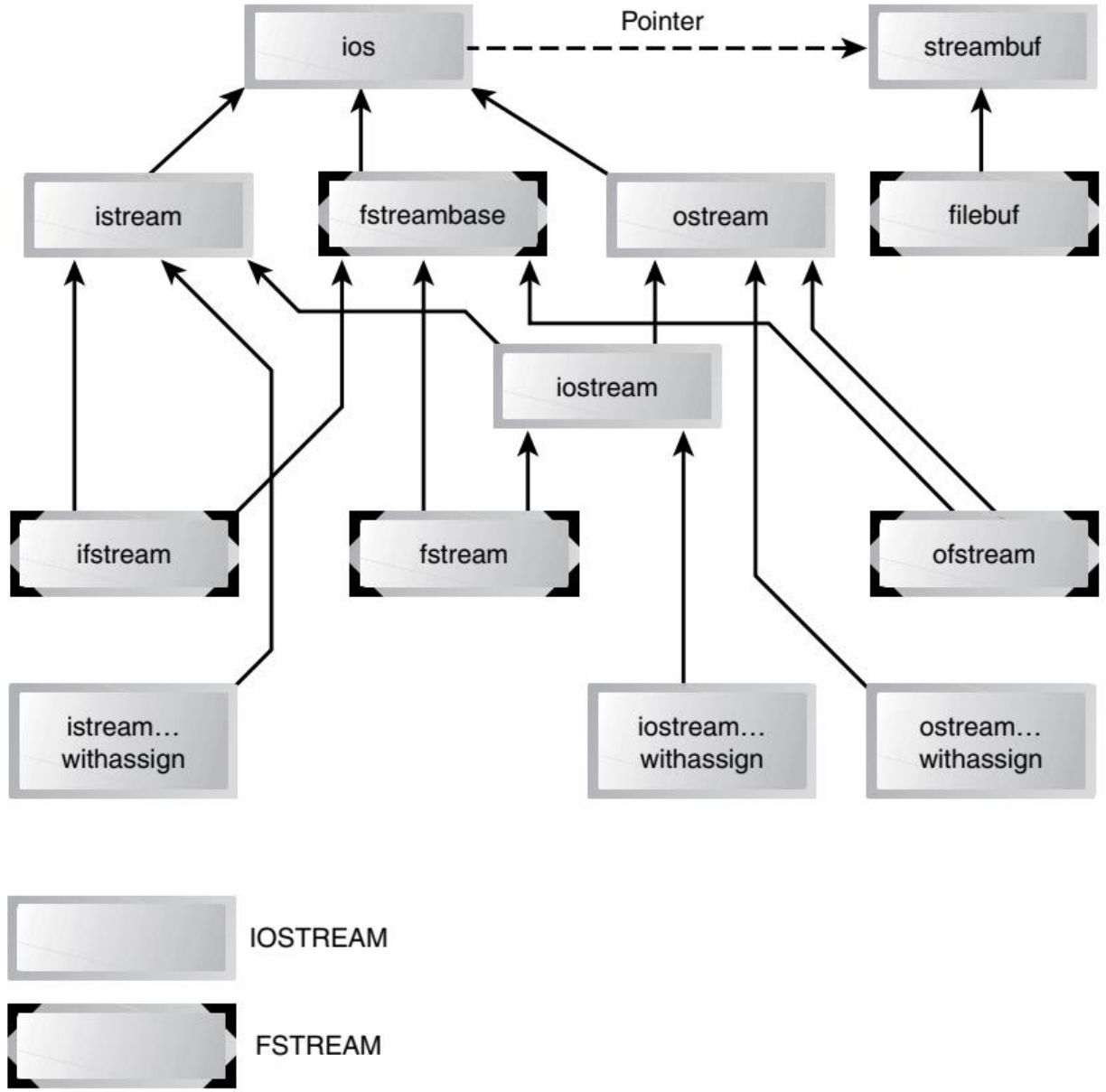


FIGURE 12.1

Stream class hierarchy.

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.

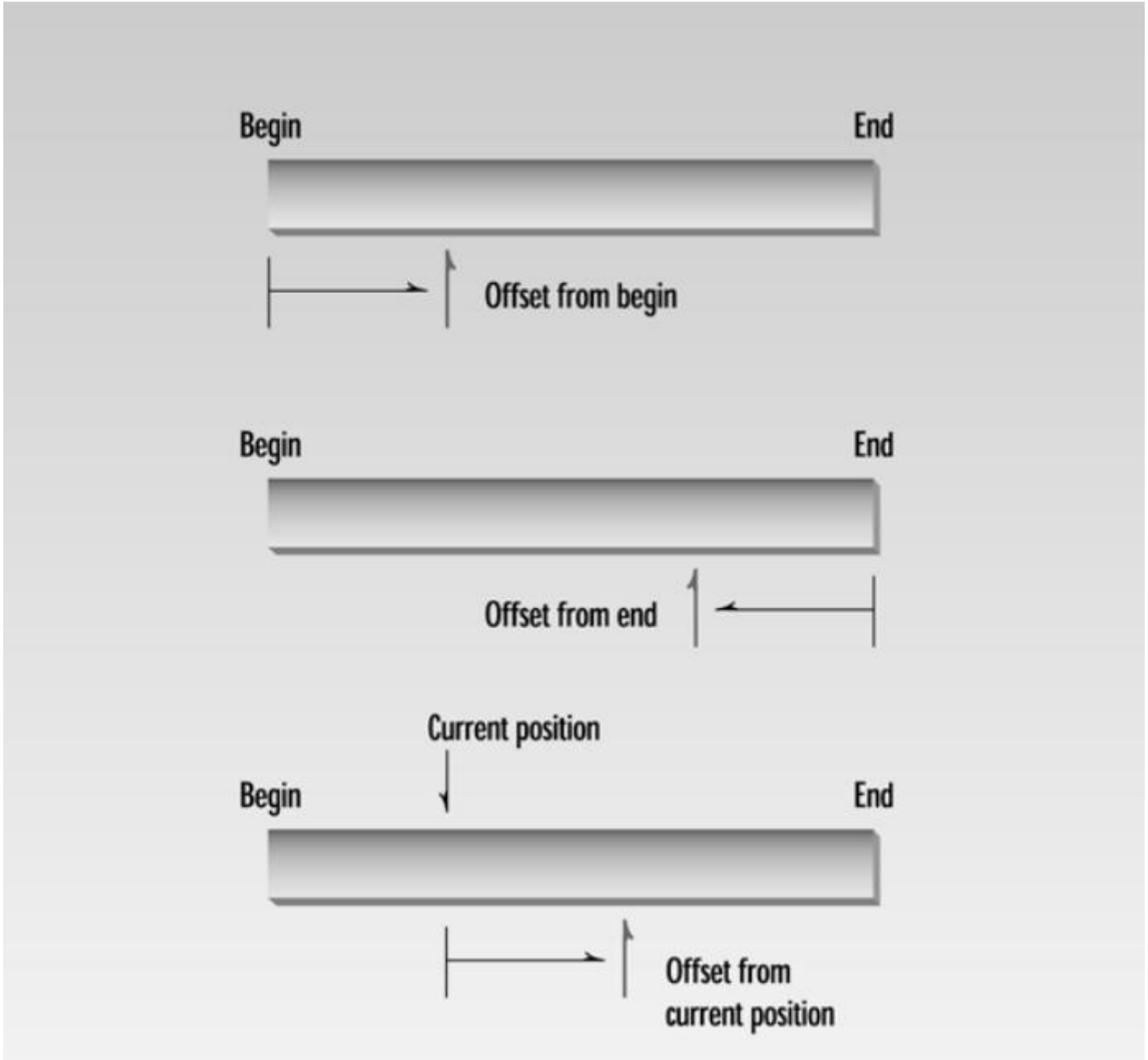
SPECIFYING THE POSITION

- We saw an example of positioning the get pointer in the DISKFUN program, where the **seekg()** function set it to the beginning of the file so that reading would start there.
- The **seekg()** function can be used in two ways:
  1. **With one Argument**: the single argument represents the position from the start of the file. **File.seekg(0)**

  2. **With Two Arguments:** first argument represents an offset from a particular location in the file, and the second specifies the location from which the offset is measured.
     - There are three possibilities for the second argument: **beg** is the beginning of the file, **cur** is the current pointer position, and **end** is the end of the file.

```
seekg(-5, ios::end);
```

     - for example, will set the put pointer to 10 bytes before the end of the file. Figure 12.5 shows how this looks.


Begin
End
Offset from begin
Begin
End
Offset from end
Current position
Begin
End
Offset from
current position

```
#include <fstream>
#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(void) //display person's data
     {
          cout << "\n Name: " << name;
          cout << "\n Age: " << age;
     }
};
```

```
int main()
{
     person pers; //create person object
     ifstream infile; //create input file
     infile.open("GROUP.DAT", ios::in | ios::binary);
     infile.seekg(0, ios::end); //go to 0 bytes from end
     int endposition = infile.tellg(); //find where we are
     int n = endposition / sizeof(person);
     cout << "\nThere are " << n << " persons in file";
     cout << "\nEnter person number: ";
     cin >> n;

     int position = (n-1) * sizeof(person);
     infile.seekg(position); //bytes from start
     //read one person
     infile.read( reinterpret_cast<char*>(&pers),
sizeof(pers) );
     pers.showData(); //display the person
     cout << endl;
return 0;
}
```

THE tellg() FUNCTION

- The **tellg()** function returns the current position of the get pointer.
- The program uses this function to return the pointer position at the end of the file; this is the length of the file in bytes.
- Next, the program calculates how many person objects there are in the file by dividing by the size of a person; it then displays the result

COMMAND-LINE ARGUMENTS

- If you've ever used MS-DOS, you are probably familiar with **command-line arguments**, used when invoking a program.

- They are typically used to pass the name of a data file to an application.

- For example, you can invoke a word processor application and the document it will work on at the same time:

```
C> wordproc afile.doc
```

- How can we get a C++ program to read the command-line arguments?

```
// comline.cpp
// demonstrates command-line arguments
#include <iostream>
using namespace std;

int main(int argc, char* argv[])
{
      cout << "\nargc = " << argc << endl; //number of
arguments
      for(int j=0; j<argc; j++) //display arguments
            cout << "Argument " << j << " = " << argv[j]
<< endl;
return 0;
}
```

```
// otype.cpp
// imitates TYPE command
#include <fstream> //for file functions
#include <iostream>
using namespace std;
#include <process.h> //for exit()
int main(int argc, char* argv[] )
{
	if( argc != 2 )
	{
		cerr << "\nFormat: otype filename";
		exit(-1);
	}
	char ch; //character to read
	ifstream infile; //create file for input
	infile.open( argv[1] ); //open file
	if(!infile ) //check for errors
	{
		cerr << "\nCan\'t open " << argv[1];
		exit(-1);
	}
	while( infile.get(ch) != 0 ) //read a character
		cout << ch; //display the character
return 0;
}
```