LESSON SET

12 Advanced File Operations

PURPOSE

1. To review the basic concept of files
2. To understand the use of random access files
3. To understand and use various types of files (binary and text)

PROCEDURE

1. Students should read the Pre-lab Reading Assignment before coming to lab.
2. Students should complete the Pre-lab Writing Assignment before coming to lab.
3. In the lab, students should complete labs assigned to them by the instructor.

Contents	Pre-requisites	Approximate completion time	Page number	Check when done
Pre-lab Reading Assignment		20 min.	214	
Pre-lab Writing Assignment	Pre-lab reading	10 min.	231	
LESSON 12A				
Lab 12.1 Introduction to Files (Optional)	General understanding of basic I/O	15 min.	231	
Lab 12.2 Files as Parameters and Character Data	Understanding of `get` function and parameters	20 min.	233	
Lab 12.3 Binary Files and the `write` Function	Completion of all previous labs	30 min.	235	
LESSON 12B				
Lab 12.4 Random Access Files	Completion of all previous labs	20 min.	238	
Lab 12.5 Student Generated Code Assignments	Completion of all previous labs	30 min.	240	

PRE-LAB READING ASSIGNMENT

Review of Text Files

Chapter three introduced the basic use of files for input and output. We briefly review those concepts in this section.

A file is a collection of information stored (usually) on a disk. Files, just like variables, have to be defined in the program. The data type of a file depends on whether it is used as an input file, output file, or both. Output files have a data type called `ofstream`, input files have a data type of `ifstream`, and files used as both have the data type `fstream`. We must add the `#include <fstream>` directive when using files.

Examples:

```
ofstream outfile;     // defining outfile as an output file
ifstream infile;      // defining infile as an input file
fstream datafile;     // defining datafile to be both an input and
                      // output file
```

After their definition, files must still be opened, used (information stored to or data read from the file), and then closed.

Opening Files

A file is opened with the `open` function. This ties the logical name of the file that is used in the definition to the physical name of the file used in the secondary storage device (disk). The statement `infile.open("payroll.dat");` opens the file `payroll.dat` and lets the program know that `infile` is the name by which this file will be referenced within the program. If the file is not located in the same directory as the C++ program, the full path (drive, etc.) MUST be indicated: `infile.open("a:\\payroll.dat");` This tying of the **logical name** `infile` with the **physical name** `payroll.dat` means that wherever `infile` is used in the program, data will be read from the physical file `payroll.dat`. A program should check to make sure that the physical file exists. This can be done by a conditional statement.

Example:

```
ifstream infile;
infile.open("payroll.dat");
if (!infile)
{
   cout << Error opening file.  It may not exist were indicated.\n;
   return 1;
}
```

In the previous example, `return 1` is used as an indicator of an abnormal occurrence. In this case the file in question can not be found.

Reading from a File

Files have an "invisible" end of line marker at the end of each line of the file. Files also have an invisible end of file marker at the end of the file. When reading from an input file within a loop, the program must be able to detect that marker as the sentinel data (data that meets the condition to end the loop). There are several ways to do this.

Sample Program 12.1:

```
#include <fstream>
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    ifstream infile;       // defining an input file
    ofstream outfile;      // defining an output file

    infile.open("payroll.dat");
    // This statement opens infile as an input file.
    // Whenever infile is used, data from the file payroll.dat
    // will be read.

    outfile.open("payment.out");
    // This statement opens outfile as an output file.
    // Whenever outfile is used, information will be sent
    // to the file payment.out

    int hours;             // The number of hours worked
    float payRate;         // The rate per hour paid
    float net;             // The net pay

    if (!infile)
    {
        cout << "Error opening file.\n";
        cout << "Perhaps the file is not where indicated.\n";
        return 1;

    }

    outfile << fixed << setprecision(2);
    outfile << "Hours     Pay Rate   Net Pay"  << endl;

    infile >> hours;    // priming the read

    while (infile)
    {
        infile >> payRate;
        net = hours * payRate;
```

continues

```
            outfile    << hours << setw(10) << "$ " << setw(6)
                       << payRate << setw(5) << "$ " << setw(7)
                       << net << endl;

            infile >> hours;
        }

        infile.close();
        outfile.close();

        return 0;
}
```

Notice the statement `outfile << fixed << setprecision(2);` in the above program. This shows that the format procedures learned for `cout` can be used for output files as well. Remember that `setw(x)` can be used as long as the `iomanip` header file is included.

This program assumes that a data file exists and contains an undetermined number of records with each record consisting of two data values, `hours` and `payRate`. Suppose the input data file (`payroll.dat`) contains the following:

```
40  10.00
30   6.70
50  20.00
```

The program will produce the following output file (`payment.out`).

```
Hours     Pay Rate     Net Pay
40         $ 10.00    $ 400.00
30         $  6.70    $ 201.00
50         $ 20.00    $1000.00
```

The input file contains data for one employee on each line. Each time through the `while` loop, information is processed for one employee. The loop executes the same number of times as there are lines (employee records in this case) in the data file. Since there are two items of data for each line (`hours` and `payRate`), these items are read in each time through the loop. Notice that one of the input variables was input before the `while` loop. This is called "priming the read." Input can be thought of as a stream of values taken one at a time. Before the `while` loop condition can be tested, there has to be something in that stream. We **prime** the read by reading in at least one variable before the loop. Observe that the statement `infile >> hours;` is given twice in the program: once before the input loop and as the last statement in the loop. The other item, `payRate`, is read in at the very beginning of the loop. This way each variable is read every time through the loop. Also notice that the heading of the output file is printed outside the loop before it is executed.

There are other ways of determining when the end of the file is reached. The `eof( )` function reports when the end of a file is encountered. The loop in Sample Program 12.1 can be replaced with the following:

```
infile >> hours;
while (!infile.eof())
{
```

```
        infile >> payRate;
        net = hours * payRate;
        outfile << hours << setw(10) << "$ " << setw(6) << payRate << setw(5)
                << "$ " << setw(7) << net << endl;
        infile >> hours;
    }
```

In addition to checking to see if a file exists, we can also check to see if it has any data in it. The following code checks first if the file exists and then if it is empty.

```
inData.open("sample2.dat");

if(!inData)
   cout << "file does not exist" << endl;
else if((else if (inData.peek()) == EOF)

   cout << "File is empty" << endl;
else
    //rest of program
```

The `peek` function actually looks ahead in the file for the next data item, in this case to determine if it is the end of file marker. `ch` must be defined as `char` data type.

Since the `peek` function looks "ahead" for the next data item, it can be used to test for end of file in reading values from a file within a loop without priming the read.

The following program accomplishes the same thing as Sample Program 12.1 without priming the read. The portions in bold differ from Sample Program 12.1.

Sample Program 12.2:

```
#include <fstream>
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
       ifstream infile;          // defining an input file
       ofstream outfile;         // defining an output file

       infile.open("payroll.dat");
       // This statement opens infile as an input file.
       // Whenever infile is used, data from the file payroll.dat
       // will be read.

       outfile.open("payment.out");
       // This statement opens outfile as an output file.
```

continues

```
// Whenever outfile is used, information will be sent
// to the file payment.out

int hours;              // The number of hours worked
float payRate;          // The rate per hour paid
float net;              // The net pay
char ch;                // ch is used to hold the next value
                        // (read as character) from the file

if (!infile)
{
      cout << "Error opening file.\n";
      cout << "Perhaps the file is not where indicated.\n";
      return 1;
}

outfile << fixed << setprecision(2);
outfile << "Hours     Pay Rate   Net Pay"  << endl;


while ((ch = infile.peek()) != EOF)
{
      infile >> hours;
      infile >> payRate;
      net = hours * payRate;

      outfile  << hours << setw(10) << "$ " << setw(6)
               << payRate << setw(5) << "$ " << setw(7)
               << net << endl;


}

infile.close();
outfile.close();

return 0;
}
```

Output Files

Output files are opened the same way: `outfile.open("payment.out")`. Whenever the program writes to `outfile`, the information will be placed in the physical file `payment.out`. Notice that the program generates a file stored in the same location as the source file. The user can indicate a different location for the file to be stored by indicating the full path (drive, etc.).

Files Used for Both Input and Output

A file can be used for both input and output. The `fstream` data type, which is used for files that can handle both input and output, must have a **file access flag** as an argument to the `open` function so that the mode, input or output, can be determined. There are several access flags that are used to indicate the use of the file. The following chart lists frequently used access flags.

Flag mode	Meaning
`ios::in`	Input mode. The file is used for "reading" information. If the file does not exist, it will not be created.
`ios::out`	Output mode. Information is written to the file. If the file already exists, its contents will be deleted.
`ios::app`	Append mode. If the file exists, its contents are preserved and all output is written to the end of the file. If it does not exist then the file will be created. Notice how this differs from `ios::out`.
`ios::binary`	Binary mode. Information is written to or read from in pure binary format (discussed later in this chapter).

Example:

```
#include <fstream>
using namespace std;

int main()
{
   fstream test ("grade.dat", ios::out)
   // test is defined as an fstream file first used for output
   // ios::out is the file access flag

   // code of the program goes here
   // the code will put values in the test file

   test.close();            // close the file as an output file
   test.open("grade.dat", ios::in)
   // the same file is reopened now as an input file
   // ios::in is the file access flag

   //  other code goes here

   test.close();            // close the file

}
```

In the following example, we check for a file's existence before opening it. We first attempt to open the file for input. If the file does not exist, then the open operation fails and we open it as an output file.

Example:

```
fstream dataFile;
dataFile.open("grades.txt", ios::in);
if (dataFile.fail())
       // The fail directive indicates the file did not open
       // since it does not exist
{
       // The file does not exist, so create it.
dataFile.open("grades.txt", ios::out);
       // file is processed here:  data sent to the file
}
else  // the file already exists.

{
cout << "The file grades.txt already exists. \n";

      // process file here
 dataFile.close();

}
```

Just as `cin >>` is used to read from the keyboard and `cout <<` is used to write to the screen, `filename >>` is used to read from the file `filename` and `filename <<` is used to write to the file `filename`.

Closing a File

Files should be closed before the program ends to avoid corrupting the file and/or losing valuable data.

```
infile.close();
outfile.close();
dataFile.close();
```

Passing Files as Parameters to Functions

Files can be passed as parameters just like variables. Files are always passed by reference. The `&` symbol must be included after the data type in the function heading and prototype.

Example:

```
void GetData(ifstream& infile, ofstream& outfile); // prototype of function
                                                   // with files as parameters

void GetData(ifstream& infile, ofstream& outfile)  // heading of function with
                                                   // files as parameters
```

Review of Character Input

Chapter 10 introduced the basics of characters and strings. We briefly review those concepts since they apply to files as well.

Recall that each file has an end of line marker for each line as well as the end of file marker at the end of the file. Whenever whitespace (blanks, newlines, controls, etc.) is part of a file, a problem exists with the traditional >> operator in inputting character data. When reading input characters into a string object, the >> operator skips any leading whitespace. It then reads successive characters into the character array, stopping at the first trailing whitespace character (which is NOT consumed, but rather which remains as the next character to be read in the file). The >> operator also takes care of adding the null character to the end of the string. Stopping at the first trailing whitespace creates a problem for names containing spaces. A program reading first names into some string variable (array of characters) has a problem reading a name like Mary Lou since it has a blank space in it. The blank space between Mary and Lou causes the input to stop when using the >> operator. The `get` function can be used to input such strings.

```
infile.get(firstname, 20);
```

The `get` function does NOT skip leading whitespace characters but rather continues to read characters until it either has read, in the example above, 19 characters or it reaches the newline character \n (which it does NOT consume). Recall from Lesson Set 10 that the last space is reserved for the null character.

Since the `get` function does not consume the end of line character, there must be something done to consume it so that a new line can be read.

Example: Given the following data file

```
Mary Lou <eol>
Becky    <eol>
Debbie   <eol>
<eof>
```

Note: Both the `<eol>` and `<eof>` are NOT visible to the programmer or user.

There are several options for reading and printing this data.

```
char dummy;                       // created to read the end of line character
char  firstname[80];              // array of characters for the first name
outfile << "Name  " << endl;
infile.get(firstname,80);         // priming the read   inputs the first name

while(infile)
{
   infile.get(dummy);             // reads the end of line character into dummy
   outfile << firstname << endl;  // outputs the name
   infile.get(firstname,80);      // reads the next name
}
```

In the above example, `dummy` is used to consume the end of line character. `input.get(firstname,80);` reads the string Mary Lou and stops just before reading the `<eol>` end of line character. The `infile.get(dummy)` gets the end of line character into `dummy`.

Another way to do this is with the `ignore` function, which reads characters until it encounters the specific character it has been instructed to look for or until it has

skipped the allotted number of characters, whichever comes first. The statement `infile.ignore(81,'\n')` skips up to 81 characters stopping if the new line `'\n'` character is encountered. This newline character IS consumed by the function, and thus there is no need for a dummy character variable.

Example:

```
char  firstname[80];

outfile << "Name  " << endl;
infile.get(firstname,80);

while(!infile.fail())
{
   infile.ignore(81,'\n');      // read and consume the end of line character
   outfile << firstname << endl;
   infile.get(firstname,80);
}
```

The following sample program shows how names with embedded whitespace along with numeric data can be processed. Parts in bold indicate the changes from Sample Program 12.2. Assume that the `payroll.dat` file has the following information:

John Brown	40	10.00
Kelly Barr	30	6.70
Tom Seller	50	20.00

The program will produce the following information in `payment.out`:

Name	**Hours**	**Pay Rate**	**Net Pay**
John Brown	40	$10.00	$ 400.00
Kelly Barr	30	$ 6.70	$ 201.00
Tom Seller	50	$20.00	$1000.00

Sample Program 12.3:

```
#include <fstream>
#include <iostream>
#include <iomanip>
using namespace std;

const int MAX_NAME = 11;

int main()
{
      ifstream infile;           // defining an input file
      ofstream outfile;          // defining an output file

      infile.open("payroll.dat");
      // This statement opens infile as an input file.
      // Whenever infile is used, data from the file payroll.dat
      // will be read.
```

```
    outfile.open("payment.out");
    // This statement opens outfile as an output file.
    // Whenever outfile is used, information will be sent
    // to the file payment.out

    int hours;                // The number of hours worked
    float payRate;            // The rate per hour paid
    float net;                // The net pay
    char ch;                  // ch is used to hold the next value
                              // (read as character) from the file
    char name[MAX_NAME];      // array of characters for the name of
                              // a student, with at most 10 characters

    if (!infile)
    {
       cout << "Error opening file.\n";
       cout << "Perhaps the file is not where indicated.\n";
       return 1;

    }

    outfile << fixed << setprecision(2);
    outfile << "Name             Hours     Pay Rate   Net Pay"  << endl;


    while ((ch = infile.peek()) != EOF)    // no need to prime the read
    {
          infile.get(name,MAX_NAME);     // gets names with blanks
          infile >> hours;
          infile >> payRate;
          infile.ignore(81,'\n');        // ignores the rest of the line
                                         // and consumes end of line marker
          net = hours * payRate;

          outfile << name << setw(10) << hours << setw(10) << "$ "
                  << setw(6) << payRate << setw(5) << "$ " << setw(7)
                  << net << endl;


    }

    infile.close();
    outfile.close();

    return 0;
}
```

Another way to read in lines containing whitespace characters is with the getline member function.

Example:

```
char firstName[80];
outfile << "Name " << endl;
infile.getline(firstName,81);

while (infile)
{
  outfile << firstName << endl;
  intfile.getline(firstName,81);
}
```

Binary Files

So far all the files we have talked about have been text files, files formatted as ASCII text.[1] Even the numbers written to a file with the << operator are changed to ASCII text. ASCII is a code that stores every datum (letter of the alphabet, digit, punctuation mark, etc.) as a character with a unique number. Although ASCII text is the default method for storing information in files, we can specify that we want to store data in pure binary format by "opening" a file in binary mode with the `ios::binary` flag. The `write` member function is then used to write binary data to the file. This method is particularly useful for transferring an entire array of data to a file. Binary files are efficient since they store everything as 1s or 0s rather than as text.

Example:

```
fstream test("grade.dat", ios::out | ios::binary); // This defines and opens
                                                   // the file test as an
                                                   // output binary file

int grade[arraysize] = {98, 88, 78, 77, 67, 66, 56, 78, 98, 56}; // creates and
                                                                 // initializes
                                                                 // an integer
                                                                 // array

test.write((char*)grade, sizeof(grade));   // write all values of array to file

test.close();                              // close the file
```

`test.write((char*)grade, sizeof(grade));` in the above example calls the `write` function. The name of the file to be written to is `test`. The first argument is a character pointer pointing to the starting address of memory, in this case to the beginning of the `grade` array. The second argument is the size in bytes of the item written to the file. `sizeof` is a function that determines the size.

[1] Or some other alphanumberic code.

The following sample program initializes an array and then places those values into a file as binary numbers. The program then adds 10 to each element of the array and prints those values to the screen. Finally the program reads the values from the same file and prints them. These values are the original numbers. Study the program and its comments very carefully.

Sample Program 12.4:

```
#include <fstream>
#include <iostream>
using namespace std;

const int ARRAYSIZE = 10;

int main()
{
       fstream test("grade.dat", ios::out | ios::binary);
       // note the use of | to separate file access flags
       int grade[ARRAYSIZE] = {98,88,78,77,67,66,56,78,98,56};
       int count;     // loop counter

       test.write((char*)grade, sizeof(grade));
                     // write values of array to file
       test.close();  // close file

       // now add 10 to each grade

       cout << "The values of grades with 10 points added\n";

       for (count =0; count < ARRAYSIZE; count++)
       {
          grade[count] = grade[count] + 10;
                    // this adds 10 to each elemnt of the array
          cout << grade[count] << endl;
                   // write the new values to the screen
       }

       test.open("grade.dat", ios::in);
                  // reopen the file but now as an input file

       test.read((char*) grade, sizeof(grade));

       /*  The above statement reads from the file test and places the
           values found into the grade array.      As with the write
           function, the first argument is a character pointer even
           though the array itself is an integer.  It points to the
           starting address in memory where the file information is to
           be transferred
       */
```

continues

```
    cout << "The grades as they were read into the file"  << endl;

    for (count =0; count < ARRAYSIZE; count++)
    {
         cout << grade[count] << endl;
              // write the original values to the screen
    }

    test.close();
    return 0;
}
```

The output to the screen from this program is as follows:

```
The values of grades with 10 points added
108
98
88
87
77
76
66
88
108
66
The grades as they were read into the file
98
88
78
77
67
66
56
78
98
56
```

Files and Records

Files are often used to store records. A "field" is one piece of information and a "record" is a group of fields that logically belong together in a unit.

Example:

Name	**Test1**	**Test2**	**Final**
Brown	89	97	88
Smith	99	89	97

Each record has four fields: `Name`, `Test1`, `Test2`, and `Final`. When records are stored in memory, rather than in files, C++ structures provide a good way to organize and store them.

```
struct Grades
{
     char name[10];
     int test1;
```

```
    int test2;
    int final;
};
```

An identifier defined to be a `Grades` structure can hold one record.

The `write` function, mentioned in the previous section, can be used to write records to a file.

```
fstream test("score.dat", ios::out|ios::binary);
Grades student;                          // defines a structure variable

// code that gets information into the student record
test.write((char *) &student, sizeof(student));
```

The `test.write` function used to write a record stored as a `struct` is similar to the `write` function used for an array with one big difference. Notice the inclusion of `(&)`. Why is this necessary here and not when writing an array? In this example we need to pass by reference which requires the `(&)` symbol. Arrays are passed by pointer. The following sample program takes records from the user and stores them into a binary file.

Sample Program 12.5:

```
#include <fstream>
#include <iostream>
#include <cctype>           // for toupper function
using namespace std;

const int NAMESIZE = 31;

struct Grades               // declaring a structure
{
            char name[NAMESIZE];
            int  test1;
            int  test2;
            int final;
};

int main()
{
            fstream tests("score.dat", ios::out | ios::binary);
            // defines tests as an output binary file

            Grades student; // defines student as a record (struct)
            char more;      // used to determine if there is more input

            do
            {
               cout << "Enter the following information"  << endl;
               cout << "Student's name: ";
               cin.getline(student.name,NAMESIZE);
```

continues

```
            cout << "First test score :";
            cin >> student.test1;
            cin.ignore();    // ignore rest of line

            cout << "Second test score: ";
            cin >> student.test2;
            cin.ignore();

            cout << "Final test score: ";
            cin >> student.final;
            cin.ignore();

            // write this record to the file
            tests.write((char *) &student, sizeof(student));

            cout << "Enter a y if you would like to input more data\n ";
            cin >> more;
            cin.ignore();

        }   while (toupper(more) == 'Y');

        tests.close();
        return 0;
}
```

Random Access Files

All the files studied thus far have performed **sequential file access**, which means that all data is read or written in a sequential order. If the file is opened for input, data is read starting at the first byte and continues sequentially through the file's contents. If the file is opened for output, bytes of data are written sequentially. The writing usually begins at the beginning of the file unless the `ios::app` mode is used, in which case data is written to the end of the file. C++ allows a program to perform **random file access**, which means that any piece of data can be accessed at any time. A cassette tape is an example of a sequential access medium. To listen to the songs on a tape, one has to listen to them in the order they were recorded or fast forward (reverse) through the tape to get to a particular song. A CD has properties of a random access medium. One simply jumps to the track where a song is located. It is not truly random access, however, since one can not jump to the middle of a song.

There are two file stream member functions that are used to move the read/write position to any byte in the file. The `seekp` function is used for output files and `seekg` is used for input files.

Example:

```
dataOut.seekp(30L, ios::beg);
```

This instruction moves the marker position of the file called `dataOut` to 30 posi- tions from the beginning of the file. The first argument, `30L` (`L` indicates a long integer), represents an offset (distance) from some point in the file that will be

used to move the read/write position. That point in the file is indicated by the second argument (`ios::beg`). This access flag indicates that the offset is calculated from the beginning of the file. The offset can be calculated from the end (`ios::end`) or from the current (`ios::cur`) position in the file.

If the `eof` marker has been set (which means that the position has reached the end of the file), then the member function `clear` must be used before `seekp` or `seekg` is used.

Two other member functions may be used for random file access: `tellp` and `tellg`. They return a long integer that indicates the current byte of the file's read/write position. As expected, `tellp` is used to return the write position of an output file and `tellg` is used to return the read position of an input file.

Assume that a data file `letterGrades.txt` has the following single line of information:

ABCDEF

Marker positions always begin with 0. The mapping of the characters to their position is as follows:

A B C D E F
0 1 2 3 4 5

The following sample program demonstrates the use of `seekg` and `tellg`.

Sample Program 12.6:

```
#include <iostream>
#include <fstream>
#include <cctype>
using namespace std;


int main()
{
            fstream inFile("letterGrades.txt", ios::in);
            long offset;     // used to hold the offset
                             // of the read position from
                             // some point
            char ch;         // holds character read at some
                             // position in the file
            char more;       // used to indicate if more information
                             // is to be given

            do
            {
               cout << "The read position is currently at byte "
                    << inFile.tellg() << endl;
```

continues

```
                    // This prints the current read position (found by the tellg
                    // function)

                    cout << "Enter an offset from the beginning of the file: ";
                    cin >> offset;

                    inFile.seekg(offset, ios::beg);

                    // This moves the position from the beginning of the file.
                    // offset contains the number of bytes that the read position
                    // will be moved from the beginning of the file

                    inFile.get(ch);
                    // This gets one byte of information from the file

                    cout << "The character read is " << ch << endl;

                    cout << "If you would like to input another offset enter a Y"
                         << endl;
                    cin >> more;

                    inFile.clear();
                    // This clears the file in case the eof flag was set

               }   while (toupper(more) == 'Y');
               inFile.close();
               return 0;
}
```

Sample Run:

The read position is currently at byte 0
Enter an offset from the beginning of the file: 2
The character read is C
If you would like to input another offset enter a Y: y
The read position is currently at byte 3
Enter an offset from the beginning of the file: 0
The character read is A
If you would like to input another offset enter a Y: y
The read position is currently at byte 1
Enter an offset from the beginning of the file: 5
The character read is F
If you would like to input another offset enter a Y: n

CAUTION: If you enter an offset that goes beyond the data stored, it prints the previous character offset.

PRE-LAB WRITING ASSIGNMENT

Fill-in-the-Blank Questions

1. The _____member function moves the read position of a file.
2. Files that will be used for both input and output should be defined as _____data type.
3. The _____member function returns the write position of a file.
4. The `ios::`_____file access flag indicates that output to the file will be written to the end of the file.
5. _____files are files that do not store data as ASCII characters.
6. The _____member function moves the write position of a file.
7. The _____function can be used to send an entire record or array to a binary file with one statement.
8. The >> operator _____any leading whitespace.
9. The _____function "looks ahead" to determine the next data value in an input file.
10. The _____ and _____functions do not skip leading whitespace characters.

LESSON 12A

LAB 12.1 Introduction to Files (Optional)

(This is a good exercise for those needing a review of basic file operations)

Retrieve program `files.cpp` from the Lab 12 folder. The code is as follows:

```
// This program uses hours, pay rate, state tax and fed tax to determine gross
// and net pay.

#include <fstream>
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
      // Fill in the code to define payfile as an input file
      float gross;
      float net;
      float hours;
      float payRate;
      float stateTax;
      float fedTax;
```

continues