Files

- A **file** is a collection of data (usually stored on a disk.)

- An **input file** contains items for input to the program (i.e., items you might otherwise type in from the keyboard during interactive data entry).

- An **output file** contains items output from your program (i.e., items you might otherwise have sent to the screen).

- In C++ we access a file by declaring a stream and attaching it to a particular file.

- You must include the fstream library

  ```
  #include <fstream>
  ```

- **Examples: (simple) declaring and opening a file for output**

  ```
  ofstream outfile("c:\\myoutput.txt");       //uses constructor

  ofstream myoutput.open("c:\\chapter13\\myoutput.txt");

  ofstream myoutput;                  //spearate variable declaration
  myoutput.open("c:\\chapter13\\myoutput.txt");  //attachment

  ofstream cout("c:\\mypgms\\hw13.txt"); //this redirects cout

  ofstream myout("con");                      //output to console
  ```

- **Examples: (simple) declaring and opening a file for input**

  ```
  ifstream infile("c:\\myinput.txt");              //uses constructor

  ifstream myinput;                   //separate variable declaration
  myinput.open("c:\\chapter13\\myinput.txt");     //attachment

  ifstream cin("c:\\mydata\\hw13input.txt");
  ```

- **File Open Modes:**

File Mode Flags	Description
iso::app	Append to end of exisiting file (or create new file if file does not exist)
ios::ate	Go to end of existing file; write anywhere
ios::binary	Read/write in binary mode (not text)
ios::in	Open file for input
ios::out	Open (or create) file for output

- **Examples: declaring and opening a file for input and/or output**

```
fstream myfile;                               //declare variable

myfile.open(“c:\\myfile.txt” ios::in);        //open for input

myfile.open(“c:\\myfile.txt” ios::out);       //open for output

myfile.open(“c:\\myfile.txt” ios::in | ios::output);
                                       //open for input and output

fstream myfile(“c:\\myfile.txt” ios::in | ios::output);
                                    // using file stream constructor
```

- **Closing files:**

  If you open a file, you should close it:

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

● **Demo Program - File Write and Read:**

```
//This program demonstrates the use of an fstream object
//and file mode flags.
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
   fstream dataFile;            // file object
   string buffer;               // Used to read line from file

   // Create a new file named myfile.dat to write to.
   dataFile.open("myfile.dat", ios::out);

   // Write two lines to the file.
   dataFile << "Now is the time for all good men" << endl
            << "to come to the aid of their country.";

   // Close the file.
   dataFile.close();

   // Open the file for input.
   dataFile.open("myfile.dat", ios::in);

   // Read a line into a buffer and print the line.
   getline(dataFile, buffer);
   cout << buffer << endl;

   // Read a second line and print it.
   getline(dataFile, buffer);
   cout << buffer << endl;

   // Close the file.
   dataFile.close();

   return 0;
}
```

● **Demo Program - File Append:**

```
// This program writes information to a file, closes the file,
// then reopens it and appends more information.
#include <fstream>
using namespace std;

int main()
{
   fstream dataFile;                    // file object

   // Open a file to write to, and write to it.
   dataFile.open("demofile.txt", ios::out);
   dataFile < < "Jones\n";
   dataFile < < "Smith\n";

   // Close the file.
   dataFile.close();

   // Open the same file in append mode, and write to it.
   dataFile.open("demofile.txt", ios::out|ios::app);
   dataFile < < "Willis\n";
   dataFile < < "Davis\n";

   // Close the file.
   dataFile.close();

   return 0;
}
```

Checking for EOF using an Input File

Using infile.eof() to Check for the End of a Data Set

- **Note:**
  When an ifstream object (e.g., infile) successfully reads in a data value, the method infile.eof() returns 0 (false). When infile reaches the end of an input stream (e.g., end of a file), then infile.eof() returns 1 (true).

The Function readdata():

```
  /* ... */
  void readdata(int numbers[], int &n)
  {
     // declare and open input file
     ifstream infile(“c:\\mypgms\\myinput.txt”);
//   ifstream infile(“con”);                //un-comment for debugging

     // read and count the number of marks
     n = 0;
     cout << "Enter the marks - end with EOF: " << endl;

     infile >> numbers[n];
     while (!infile.eof())                  //read until EOF
     {
        n++;
        infile >> numbers[n];
     }

     infile.close();                        //close input file
     return;
  }
```

Alternate way to Check for the End of a Data Set

- **Note:**
  When infile successfully reads in a data value, it returns1 (true). When infile reaches the end of the file, it returns 0 (false).

The Function readdata():

```
   /* ... */
   void readdata(int numbers[], int &n)
   {
       // declare and open input file
       ifstream infile(“c:\\mypgms\\myinput.txt”);
//     ifstream infile(“con”);                   //un-comment for debugging

       // read and count the number of marks
       n = 0;

       while (infile >> numbers[n])        //read until EOF
          n++;

       infile.close();                     //close input file
       return;
   }
```

Checking for File Open Errors

- **Note:**
  When an ifstream object (e.g., infile) successfully opens a file, the method infile.fail() returns 0 (false). Otherwise, it returns 1 (true).

The Function readdata():

```
   /* ... */
   void readdata(int numbers[], int &n)
   {
      // declare and open input file
      ifstream infile("c:\\mypgms\\myinput.txt");
//    ifstream infile("con");               //un-comment for debugging

      if (infile.fail())                    //test for file open failure
      {
         cout << "Error: the file could not be opened" < endl;
         exit(1);
      }

      // read and count the number of marks
      n = 0;

      while (infile >> numbers[n])          //read until EOF
         n++;

      infile.close();                       //close input file
      return;
   }
```

Alternate Method for Checking for File Open Errors

- **Note:**
  When an ifstream object (e.g., infile) fails to open a file, its value will be 0 (false). Otherwise, it will be true (non-zero).

The Function readdata():

```
   /* ... */
   void readdata(int numbers[], int &n)
   {
       // declare and open input file
       ifstream infile(“c:\\mypgms\\myinput.txt”);
//     ifstream infile(“con”);                   //un-comment for debugging

       if (!infile)                              //test for file open failure
       {
          cout << “Error: the file could not be opened” < endl;
          exit(1);
       }

       // read and count the number of marks
       n = 0;

       while (infile >> numbers[n])              //read until EOF
          n++;

       infile.close();                           //close input file
       return;
   }
```

Output Formatting

- **Use the regular I/O stream manipulators (#include <iomanip>):**

- **I/O Stream Manipulators:**

Manipulator	Description
dec	Display in decimal format
endl	Write newline and flush output buffer
fixed	Use fixed point numbers for real numbers
flush	Flush output buffer
hex	Inputs and outputs using hexadecimal
left	Left justify output
oct	Inputs and outputs using octal
right	Right justify output
scientific	Use scientific notation for real numbers
setfill(ch)	Makes ‘ch’ the fill character
setprecision(n)	Sets floating-point precision to n
setw(n)	Set width of output field to n
showpoint	Show decimal point and trailing zeroes
noshowpoint	(Opposite of showpoint)
showpos	prints a + sign with nonnegative numbers
noshowpos	(Opposite of showpos)

● **Demo Program - Output Formatting:**

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

int main()
{
   const int ROWS = 3, COLS = 3;
   fstream outFile("table.txt", ios::out);
   int nums[ROWS][COLS] =  {  {2897,    5,  837},
                              {34,    7, 1623},
                              {390, 3456,   12}
                           };

   // Write the three rows of numbers.
   for (int row = 0; row < ROWS; row + + )
   {
      for (int col = 0; col < COLS; col + + )
      {
         outFile < < setw(4) < < nums[row][col] < < "  ";
      }
      outFile < < endl;
   }

   // Close the file.
   outFile.close();

   return 0;
}
```

Output Formatting Using sstream Objects

- A library class called **ostringstream** provides stream objects that write to an array of characters in memory instead of writing to a disk file.

- This class allows in-memory formatting of output before writing to a file.

- To use, **#include <sstream>**

- **Demo Program - Output Formatting:**

```
// This program demonstrates the use of an ostringstream
// object to do sophisticated formatting.
#include <iostream>
#include <iomanip>
#include <sstream>
using namespace std;

string dollarFormat(double);          // Function Prototype

int main()
{
   const int ROWS = 3, COLS = 2;
   double amount[ROWS][COLS] = {   {184.45,    7},
                                   {59.13,     64.32},
                                   {7.29,      1289}
                                };

   // Format table of dollar amounts in columns of width 10
   for (int row = 0; row< ROWS; row++)
   {
      for (int column = 0; column < COLS; column++)
         cout<<setw(10) << dollarFormat(amount[row][column]);
      cout << endl;
   }
   return 0;
}

//*******************************************
// formats a dollar amount
//*******************************************
string dollarFormat(double amount)
{
   // Create ostringstream object
   ostringstream outStr;

   // Set up format information and write to outStr.
   outStr << showpoint << fixed << setprecision(2);
   outStr << '$' << amount;

   // Extract and return the string inside outStr.
   return outStr.str();
}
```

Output Formatting with Member Functions

- **Formatting Member Functions:**

Method	Description
width(n)	Set width of output field to n
precision(n)	Sets floating-point precision to n
setf()	Sets the specified format flag
unsetf()	Disables the specified format flag

- **Examples:**

```
dataOut.width(n);          //same as dataOut << setw(n);

dataOut.precision(n);      //same as dataOut <<setprecision(n);

dataOut.setf(ios::fixed);  // same as dataOut << fixed;

dataOut.setf(ios::fixed | ios::showpoint | ios::left);
        // same as dataOut << fixed << showpoint << left
```

Passing File Stream Objects to Functions

```
   /* multiplication table for the integers 1 to 10 */
   #include <iostream>
   #include <fstream>
   using namespace std;

   //function prototypes
   void printheadings(ofstream &);
   void printrow(ofstream &, int);

   int main()
   {
      // declare and open the output file
//    ofstream outfile("output.txt"); //comment-out for debugging
      ofstream outfile("con");        //un-comment for debugging

      printheadings(outfile);
      for (int m1 = 1; m1 < = 10; m1 + +)
         printrow(outfile, m1);             //m1 = multiplicand

      outfile.close();                      //close the output file

      return 0;
   }
```

```
/* Function printheadings()
 * Input:
 *     out1 - a reference to the output file
 * Process:
 *     prints headings for a multiplication chart
 * Output:
 *     prints the table headings
 */
void printheadings(ofstream &out1)
{
   out1 << "\tThis is a Multiplication Table from 1 to 10"
      << endl << endl;
   out1.width(5);
   out1 << "X";
   /* loop to print the heading of multipliers */
   for (int m2 = 1; m2 <= 10; m2++)
   {
      out1.width(5);
      out1 << m2;
   }
   out1 << endl;
   return;
}
```

```
/* Function printrow()
 * Input:
 *     out2 - a reference to the output file
 *     m1 - the current multiplicand
 * Process:
 *     prints a row of a multiplication table by calculating the
 *     first 10 multiples of the multiplicand.
 * Output:
 *     prints a row of the table
 */
void printrow(ofstream &out2, int m1)
{
   out2.width(5);
   out2 < < m1;              //prints the multiplicand
   for (int m2 = 1; m2 < = 10; m2+ +)  //m2=multiplier
   {
      out2.width(5);
      out2 < < m1 * m2;   //prints the product
   }
   out2 < < endl;
   return;
}
```

More Detailed Error Testing

- All stream objects have **error state bits** that indicate the condition of the stream.

- **File Condition Bit Flags:**

Bit	Description
ios::eofbit	set when end of input stream encountered
ios::failbit	set when an attempted operation has failed
ios::hardfail	set when unrecoverable error has occurred
ios::badbit	set when an invalid operation has been attempted
ios::goodbit	set when all of the other flags are not set

- **Member Functions that Report on the Bit Flags:**

Method	Description
eof()	Returns true if eofbit is set; otherwise false
fail()	Returns true when failbit or hardfail flags set
bad()	Returns true when badbit flag is set
good()	Returns true when goodbit flag is set
clear()	When called with no arguments, it clears all of the above flags (and sets the goodbit). It can also be called with a specific flag as an argument.

Member Functions for Reading and Writing Files

- **Member Function getline():**

infile.getline(*C_string_var, max_num + 1,delimiting_character*);

getline() reads characters (incluing whitespace characters) from the calling input stream (infile) into the specified *C_string_var* until either *max_num* characters are read or the *delimiting_character* is encountered. The third parameter is optional. If it is omitted, the default is the newline character (Enter).

- **Example:**

```
char str[81];

infile.getline(str, 81, '.');     //user types: Hello there.
                                  // str = Hello there
                                  // (Note: no period)
infile.getline(str, 81, '\n');    //user types: Great Idea!
                                  // str = Great Idea!
                                  // (Note: with exclamation point)
infile.getline(str, 81);          //reads infile until 80 characters
                                  // are read or Enter pressed
```

- **Member Function get():**
  - Reads a single character from the input stream.

```
infile.get(ch);
```

- **Member Function put():**
  - Writes a single character from the output stream.

```
outfile.get(ch);
```

- **Member Functions seekg() (seek get) and seekp() (seek put):**
  - Move the read/write position in a file.

```
infile.seekg(offset, place);      //used with input files
outfile.seekp(offset, place);     //used with output files
```

  - "offset" is number of bytes to seek from "place"
  - "place" can be ios::beg; ios::cur; ios::end

- **Examples:**

```
infile.seekg(0L, ios::beg);        //rewind to beginning of file

infile.seekg(50, ios::cur);        //move forward 50 bytes

infile.seekg(-75, ios::end);       //move to 75th byte from EOF

outfile.seekp(0L, ios::beg);       //rewind to beginning of file

outfile.seekp(50, ios::cur);       //move forward 50 bytes

outfile.seekp(-75, ios::end);      //move to 75th byte from EOF
```

```
// Program shows how to rewind a file. It writes a text file and opens it
// for reading, then rewinds it to the beginning and reads it again.
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    // Variables needed to read or write file one
    // character at a time.
    char ch;
    fstream ioFile("rewind.txt", ios::out);

    // Open file.
    if (!ioFile)
    {
        cout << "Error in trying to create file";
        return 0;
    }

    // Write to file and close.
    ioFile<<"All good dogs "<<endl<<"growl, bark, and eat."<<endl;
    ioFile.close();

    //Open the file.
    ioFile.open("rewind.txt", ios::in);
    if (!ioFile)
    {
        cout << "Error in trying to open file";
        return 0;
    }

    // Read the file and echo to screen.
    while (ioFile.get(ch))
        cout.put(ch);
    cout << endl;

    //Rewind the file.
    ioFile.clear();
    ioFile.seekg(0, ios::beg);

    //Read file again and echo to screen.
    while (ioFile.get(ch))
        cout.put(ch);
    return 0;
}
```

Working with Multiple Files

- **Demo Program:**

```
// This program demonstrates reading from one file
// and writing to a second file.
#include <iostream>
#include <fstream>
#include <cctype>                // Needed for the toupper function
using namespace std;

int main()
{
   // Variables needed to read and write files.
   const int LENGTH = 81;
   ifstream inFile;              // For input file
   ofstream outFile("out.txt");  // For output file
   char fileName[LENGTH], ch, ch2;

   // Open input file
   cout << "Enter a file name: ";
   cin >> fileName;
   inFile.open(fileName);
   if (!inFile)
   {
      cout << "Cannot open " << fileName << endl;
      return 0;
   }

   // Read characters from  input file and write
   // uppercased versions of the character to the
   // output file.
   while (inFile.get(ch))
   {
      ch2 = toupper(ch);
      outFile.put(ch2);
   }

   // Close files.
   inFile.close();
   outFile.close();
   cout << "File conversion done.\n";
   return 0;
}
```

Binary Files

- By default, files are stored in (ASCII) **text mode.**

- **Binary files** contain data that is unformatted and is not necessarily stored as ASCII text.

- To open a file in binary mode, use the ios::binary mode flag.

  infile.open(“myfile.dat”, ios::binary);

- Reading and writing binary files requires special methods:

  infile.read(char *buffer, int numberBytes);
  outfile.write(char *buffer, int numberBytes);

  - Note: the address of the “buffer” needs to be type cast to char * (or you can use **reinterpret_cast<char *>**)

- **Demo Program:**

```
//This program uses the write and read functions.
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    // File object used to access file.
    fstream file("nums.dat", ios::out | ios::binary);
    if (!file)
    {
        cout << "Error opening file.";
        return 0;
    }

    // Integer data to write to binary file.
    int buffer[ ] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int size = sizeof(buffer)/sizeof(buffer[0]);

    // Write the data and close the file.
    cout << "Now writing the data to the file.\n" << endl;
    file.write(reinterpret_cast<char *>(buffer), sizeof(buffer));
    file.close();

    // Open the file and use a binary read to read
    // contents of the file into an array.
    file.open("nums.dat", ios::in);
    if (!file)
    {
        cout << "Error opening file.";
        return 0;
    }

    cout << "Now reading the data back into memory.\n";
    file.read(reinterpret_cast<char *>(buffer), sizeof(buffer));

    // Write out the array entries.
    for (int count = 0; count < size ; count++)
        cout << buffer[count] << " ";
    cout << endl;

    // Close the file.
    file.close();
    return 0;
}
```

Creating Records with Structures

- Structures can be used to store fixed length records to a file.

- Since structures can contain a mixture of data types, you should always use the ios::binary mode when opening a file to store structures.

- Structures written to a file **must not contain pointers.** (This is because if the structure is read into memory on a subsequent run of the program, it can not be guaranteed that all program variables will be at the same memory locations; hence, the pointer values will be wrong.)

- Since string objects use pointers and dynamic memory internally, **structures written to a file must not contain string objects.**

Random Access Files

- With **sequential access**, you start at the beginning of a file and access the records in order.

- With **random access**, you can access data in a file in any order (i.e., you can access any record directly).

- **Member Functions seekg() (seek get) and seekp (seek put):**

  - Move the read/write position in a file.

```
infile.seekg(offset, place);        //used with input files
outfile.seekp(offset, place);       //used with output files
    - “offset” is number of bytes to seek from “place”
    - “place” can be ios::beg; ios::cur; ios::end
```

- **Member Functions tellg() and tellp():**

  - Return the current read/write byte position in a file.

```
offset = infile.tellg();            //used with input files
offset = outfile.tellp();           //used with output files
```

Opening a File for Both Input and Output

- **See Demo Project:**