==========

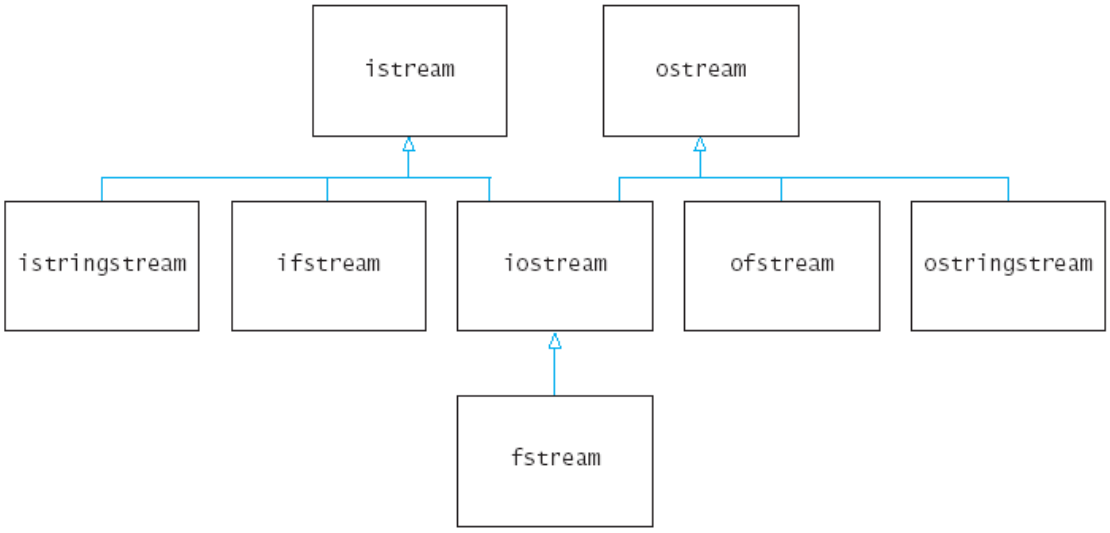
STREAMS / FILES

==========

I/O Streams

- I/O refers to program Input and Output
- I/O is done via stream objects
- A **stream** is a flow of data.
- Input stream: data flows into the program
    - Input can be from
        - the keyboard
        - a file
- Output stream: data flows out of the program
    - Output can be to
        - the screen
        - a file
- Input and Output stream: data flows either into or out of the program
    - only possible with files

- **The C++ input/output library** consists of several classes that are related by inheritance (*inheritance will be treated later in this course*)
- The inheritance hierarchy of stream classes:



- The standard **cin** and **cout** objects
  belong to specialized system-dependent classes with nonstandard names.
- You can assume that
    - **`cin`** belongs to a class that is derived from **`istream`** and
    - **`cout`** belongs to a class derived from **`ostream`**.

cin & cout streams

- `cin`
  - input stream connected to the keyboard
- `cout`
  - output stream connected to the screen
- `cin` and `cout` are declared in the `iostream` header file
  - => `#include <iostream>`
- You can declare your own streams to use with files.

Why use files?

- • Files allow you
  - to use input data over and over
  - to deal with large data sets
  - to access output data after the program ends
  - to store data permanently

Text files vs. Binary files

- Usually files are classified in two categories:
  - ASCII (text) files
  - and binary files.
- While both binary and text files contain data stored as a series of bits,
  - the bits in text files represent characters,
  - while the bits in binary files represent other types of data (`int`, `float`, `struct`, ...)
- Simple text files are usually created by using a text editor like **notepad**, **pico**, etc. (not Word or OpenOffice)
- We work with binary files all the time.
  - **executable files**, **image files**, **sound files**, ... are binary files.
- In effect, ASCII files are basically binary files, because they store binary numbers.
- `cin` & `cout` "behave like" text files.

Accessing file data

- Open the file
  - this operation associates the name of a file in disk to a stream object.
  - NOTE: cin and cout are open automatically on program start.
- Use read/write calls or extraction/insertion operators, to get/put data from/into the file.
- Close the file.

Declaring Stream Variables

- Like other variables, a stream variable must be …
  - declared before it can be used
  - initialized before it contains valid data
    - Initializing a stream means connecting it to a file
- Input-file streams are of type `ifstream`
- Ouput-file streams of are type `ofstream`
- These types are defined in the `fstream` library
  - => `#include <fstream>`

- Example:

```
#include <fstream>
using namespace std;
...
ifstream in_stream;
ofstream out_stream;
```

Connecting a stream to a file / Opening a file

- The opening operation connects a stream to an external file name
  - An external file name is the name for a file that the operating system uses
  - Examples:
    - `infile.txt` and `outfile.txt` used in the following examples
- Once a file is open, it is referred to using the name of the stream connected to it.
- A file can be opened using
  - the **`open()`** member function associated with streams
  - the **constructor** of the stream classes
- Examples:
  - `ifstream in_stream;`
  - `ofstream out_stream;`
  - `in_stream.open("infile.txt");`
    - connects `in_stream` to `"infile.txt"`
  - `out_stream.open("C:\\Mieic\\Prog\\programs\\outfile.txt");`
    - connects out_stream to `"oufile.txt"`
      that is in directory `"C:\Mieic\Prog\programs"`
    - note the double backslash in the string argument
      - necessary in Windows systems
        where the directories of the path are separated by `'\'`
  - Alternatively:
    - `ifstream in_stream("infile.txt");`
    - calls the constructor of `ifstream` class
      that automatically tries to open the file
- The filename does not need to be a constant, as in the previous examples.
  Program users can enter the name of a file to use for input or for output.
  - in this case it must be stored in a string variable
  - In C++11, you can use a `std::string` as argument to **`open()`**
    or to the constructor

```
std::string filename;
cout << "Filename ?"; cin >> filename;
myFile.open(filename);
```

  - In the previous C++ standard,
    `open()` only accepts a C-string for the first parameter.
    The correct way of calling it would then be:
    - `myFile.open(filename.c_str());`
- Note:
  - The name of a text file does not necessarily have the extension `'.txt'`

open() method (C++11)

- `void ifstream::open(const string &filename, ios::openmode mode = ios::in);`
- `void ofstream::open(const string &filename, ios::openmode mode = ios::out);`
- `void fstream::open(const string &filename, ios::openmode mode = ios::in | ios::out);`
    - **filename** is the name of the file (must be a C-string, in pre-C++11 compilers)
    - **mode** determines how the file is opened; can be the OR (|) of several constants
        - **ios::in** – the file is capable of input
        - **ios::out** – the file is capable of output
        - **ios::binary** – causes file to be opened in binary mode;
          by default, all files are opened in text mode
        - **ios::ate** – cause initial seek to end-of-file;
          I/O operations can still occur anywhere within the file
        - **ios::app** – causes all output to the file to be appended to the end
        - **ios::trunc** – the file is truncated to zero length

Using input/output stream for reading/writing from/to text files

- It is very easy to read from or write to a text file.
- Simply use the **<<** and **>>** operators the same way you do when performing console I/O, except that, instead of using cin and cout,
  use a stream that is linked to a file.
- Example 1:

```
ifstream in_stream;
in_stream.open("infile.txt");
int one_number, another_number;
in_stream >> one_number >> another_number;
```

- Example 2:

```
ofstream out_stream;
out_stream.open("outfile.txt");
out_stream << "Resulting data:";
out_stream << one_number << endl << another_number << endl;
```

Closing a file

- After using a file, it should be closed.
  This disconnects the stream from the file
    - Example: `in_stream.close();`
- The system will automatically close files if you forget
  as long as your program ends normally
- Files should be closed:
    - to reduce the chance of a file being corrupted
      if the program terminates abnormally.
    - if your program later needs to read input from the output file.

Errors on opening files

- Opening a file could fail for several reasons.
  Common reasons for open to fail include
  - the file does not exist (or the path is incorect)
  - the external name is incorrect
  - the file is already open
- Member function **`is_open()`**, can be used to test whether the file is already open
- May be no error message if the call to open fails.
  Program execution continues!
- Member function **`fail()`**, can be used to test the success of a stream operation
  (not only the open() operation)
  - Example:

```
in_stream.open("numbers.txt");
if( in_stream.fail( ) )
{
    cerr << "Input file opening failed.\n";
    exit(1) ;  //  sometimes, it is best to stop the program,
               //  with an exit code != 0
}
```

Reading from text files – additional notes

- Stream input is performed with the stream extraction operator **`>>`**, which
  - skips white space characters (`' '`, `'\t'`, `'\n'`)
  - returns **`false`**, after end-of-file (EOF) is encountered
  - Example:

```
double next, sum = 0;
while(in_stream >> next)
{
   sum = sum + next;
}
```

- Stream input causes some stream state flags to be set when an error occurs:
  - **`failbit`** - improper input (internal logic error of the operation)
  - **`badbit`** - the operation failed (failure of I/O on the stream buffer)
  - **`eofbit`** - EOF was reached on the input stream
    - EOF can be tested using the eof() member function
      - **`while( ! in_stream.eof() ) ...`** (*see later*)
  - **`goodbit`** to be set when no error has occured
- Member function **`ignore()`** can be used to skip characters, as with **`cin`** stream.
- NOTE:
  - be careful when mixing **`>>`** and **`getline()`** OR **`>>`** and **`cin.get()`**
  - *remember what has been said about this, in the string section*

How To Test End of File

- In some cases, you will want to know when the end of the file has been reached.
  - For example, if you are reading a list of values from a file, then you might want to continue reading until there are no more values to obtain.
  - This implies that you have some way to know when the end of the file has been reached.
  - C++ I/O system supplies such a function to do this: **`eof( )`**.
  - To detect EOF involves these steps:
    - 1. Open the file being read for input.
    - 2. Begin reading data from the file.
    - 3. After each input operation, determine if the end of the file has been reached by calling **`eof( )`**.
- NOTE:
  - **`eof()`** returns false only when the program tries to read past the end of the file
- Example:
  - This loop reads each character, and writes it to the screen

```
in_stream.get(next);          // NOTE:first, input must be tried
while ( ! in_stream.eof( ) ) // then you can test for EOF
{
   cout << next;
   in_stream.get(next);
}
```

Formatting output to text files

- As for **`cout`**, formatting can be done using:
  - manipulators (defined in **iomanip** library => #include <iomanip>)
    - **`setw()`**
    - **`fixed`**
    - **`setprecision`**
    - ... and some other
  - using **`setf()` member function** of output streams
    - **`out_stream.setf(ios::fixed);`**
    - **`out_stream.setf(ios::showpoint);`**
    - **`out_stream.precision(2);`**
    - ... and some other
- Note:
  - A manipulator is a function called in a nontraditional way used after the insertion operator (<<) as if the manipulator function call is an output item
  - Manipulators in turn call member functions
    - **`setw`** does the same task as the member function **width**
    - **`setprecision`** does the same task as the member function **`precision`**
    - ...
  - Any flag that is set, may be unset, using the **`unsetf`** function
    - Example: **`cout.unsetf(ios::showpos);`** causes the program to stop printing plus signs on positive numbers

Stream names as arguments

- Streams can be arguments to a function
- The function's formal parameter for the stream must be call-by-reference
    - Example:

```
void make_neat(ifstream &messy_file, ofstream &neat_file);
// make_neat() code will be presented in the following pages
```

- Take advantage of the inheritance relationships between the stream classes whenever you write functions with stream parameters.
    - **ifstream** as well as **cin** are objects of type **istream**
    - **ofstream** as well as **cout** are objects of type **ostream**
    - Example:
        - `double get_max(istream &in);`
        - You can now pass parameters of types derived from **istream**, such as an **ifstream** object or **cin**.
            - `max = get_max(in_stream);`
            - `max = get_max(cin);`
            - both **cin** and **in_stream** can be used as arguments of a call to **get_max()**, whose parameter is of type **istream &**

Binary I/O

- While reading and writing text files is very easy it is not always the most efficient way to handle files.
- There will be times when you need to store information in binary format: **int**'s, **double**'s, **struct**'s, ... or **char**'s
- When performing I/O of binary data be sure to open the file using the **ios::binary** mode specifier
- I/O can be performed using the
    - **get()** and **put()** member functions
    -
        - `istream &get(char &ch);`
        - `ostream &put(char ch);`
    - NOTE:
        - In a text stream, some character translations may take place. For example, when the newline character is output, using **<<**, it may be converted into a carriage-return / linefeed sequence.
        - The reverse happens when a carriage-return / linefeed sequence is input from a file: it is converted into a newline char.
        - No such translations occur on binary files:
            - using **get()** you can "see" the carriage-return/linefeed chars in a text file

- `read()` and `write()` member functions
  - can be used to read/write blocks of binary data
    - `istream& read (char *buf, streamsize num);`
      - reads `num` characters from the invoking stream ans puts them into the buffer pointed to by `buf`
    - `ostream& write (const char* buf, streamsize num);`
      - writes `num` characters to the invoking stream from the buffer pointed to by `buf`
- Example: (*see next pages*)

Random access

- The C++ I/O system manages 2 pointers associated with a file:
  - the get pointer, which specifies where in the file the next input operation will occur
  - the put pointer, which specifies where in the file the next output operation will occur
- You can perform random access (in a nonsequential fashion) by using the `seekg()` and `seekp()` functions.
- Generally, random access I/O should only be performed on those files opened for binary operations. WHY?
- Their most common forms are:
  - `istream& seekg (streamoff offset, ios_base::seekdir origin);`
  - `ostream& seekp (streamoff offset, ios_base::seekdir origin);`
    - `origin` can take one of the values: `ios::beg`, `ios::end`, `ios::cur`
    - `offset` is an integer that specifies the displacement of the get/put pointer relative to the specified `origin`
- `seekg()` and `seekp()` are interchangeable for file streams. However, this is not true for other types of streams (ex: stringstreams, *sees next pages*), as they may hold separate pointers for the put and get positions.
- `tellg()` and `tellp()` can be used to obtain the current position of the pointers.
- Note: you can't call **seekp**/**tellp** on an instance of **ifstream** and you can't call **seekg**/**tellg** on an instance of **ofstream**. However, you can use both on an instance of **fstream**.

==========

INPUT/OUTPUT – TEXT FILES

==========

```
/**
   INPUT FROM TEXT FILE
   Reads numbers from a file and finds the maximum value
   @param in the input stream to read from
   @return the maximum value or 0 if the file has no numbers

   (from BIG C++ book)
*/
#include <iostream>
#include <string>
#include <fstream>

using namespace std;

double max_value(ifstream &in) //stream parameters must always be passed by reference
{
   double highest;
   double next;
   if (in >> next) // if file contains at least 1 element
      highest = next;
   else
      return 0;   // If file is empty.  Not the best solution ...!!!

   while (in >> next)
   {
      if (next > highest)
         highest = next;
   }

   return highest;
}

int main()
{
   string filename;
   cout << "Please enter the data file name: "; // numbers.txt
   //located in C:\Users\jsilva\......\Project_folder\numbers.txt
   cin >> filename;

   ifstream infile;
   infile.open(filename);

   if (infile.fail())   // OR if (! infile.is_open())  OR if (! infile)
   {
      cerr << "Error opening " << filename << "\n";
      return 1;   // exit(1);
   }

   double max = max_value(infile);
   cout << "The maximum value is " << max << "\n";

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

==========

```
/**
INPUT FROM TEXT FILE   OR   KEYBOARD
Reads numbers from a file and finds the maximum value
@param in the input stream to read from
@return the maximum value or 0 if the file has no numbers

(adapted from BIG C++ book, by JAS)
*/

#include <iostream>
#include <string>
#include <fstream>

using namespace std;

double max_value(istream &in) // can be called with 'infile' or 'cin'
{
      double highest;
      double next;
      if (in >> next)
            highest = next;
      else
            return 0;

      while (in >> next)
      {
            if (next > highest)
                  highest = next;
      }

      return highest;
}

int main()
{
      double max;

      string input;
      cout << "Do you want to read from a file? (y/n) ";
      cin >> input;

      if (input == "y")
      {
            string filename;
            cout << "Please enter the data file name: ";
            cin >> filename;

            ifstream infile;
            infile.open(filename);

            if (infile.fail())
            {
                  cerr << "Error opening " << filename << "\n";
                  return 1;
            }

            max = max_value(infile);
            infile.close();
      }
```

```
	else
	{
		cout << "Insert the numbers. End with CTRL-Z." << endl;
		max = max_value(cin);
	}

	cout << "The maximum value is " << max << "\n";

	return 0;
}
```

TO DO BY STUDENTS:

- What is the output when the file is empty or the user types CTRL-Z as first input?
- Modify the program to solve the 'problem'.

```
// INPUT/OUTPUT - TEXT FILES
// Reads all the numbers in the file rawdata.dat and writes the numbers
// to the screen and to the file neat.dat in a neatly formatted way.
// Illustrates output formatting instructions.
// Adapted from Savitch book

// DON'T FORGET TO PUT FILE rawdata.txt IN THE PROJECT DIRECTORY
// OR IN THE CURRENT DIRECTORY (IF YOU RUN THE PROGRAM FROM THE COMMAND PROMPT)

#include <iostream>
#include <fstream>
#include <cstdlib>
#include <iomanip>

using namespace std;


/*
The numbers are written one per line, in fixed-point notation
with 'decimal_places' digits after the decimal point;
each number is preceded by a plus or minus sign and
each number is in a field of width 'field_width'.
(This function does not close the file.)
*/
void make_neat(ifstream& messy_file, ofstream& neat_file,
               int field_width, int decimal_places);



int main( )
{
      const int FIELD_WIDTH = 12;
      const int DECIMAL_PLACES = 5;

    ifstream fin;
    ofstream fout;

    fin.open("rawdata.txt");
    if (fin.fail( ))           //Could have tested if(fin.is_open())
    {
        cerr << "Input file opening failed.\n";
        exit(1);
    }

    fout.open("neatdata.txt");
    if (fout.fail( ))
    {
        cerr << "Output file opening failed.\n";
        exit(2);
    }

    make_neat(fin, fout, FIELD_WIDTH, DECIMAL_PLACES);

    fin.close( );
    fout.close( );

    cout << "End of program.\n";
    return 0;
}
```

```
//Uses iostream, fstream, and iomanip:
void make_neat(ifstream &messy_file, ofstream &neat_file,
               int field_width, int decimal_places)
{
      double next;

    neat_file.setf(ios::fixed);           // not in e-notation
    neat_file.setf(ios::showpoint);       // show decimal point ...
                                          // ... even when fractional part is 0
    neat_file.setf(ios::showpos);         // show + sign
    neat_file.precision(decimal_places);
/*
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.setf(ios::showpos);
    cout.precision(decimal_places);
*/
    while (messy_file >> next)
    {
        //cout << setw(field_width) << next << endl;
        neat_file << setw(field_width) << next << endl;
    }
}


/*

rawdata.txt

10.37     -9.89897
2.313   -8.950  15.0

  7.33333   92.8765
-1.237568432e2

neatdata.txt

   +10.37000
    -9.89897
    +2.31300
    -8.95000
   +15.00000
    +7.33333
   +92.87650
  -123.75684

*/
```

```
//FILES
//Detecting the end of a file with eof() method
//Copies file code.txt to file code_numbered.txt,
//but adds a number to the beginning of each line.
//Illustrates the use of get() member function of istream/ifstream
//Assumes code.txt is not empty.

#include <fstream>
#include <iostream>
#include <cstdlib>

using namespace std;

int main( )
{
  ifstream fin;
  ofstream fout;

  fin.open("code.txt");
  if (fin.fail( ))
  {
    cerr << "Input file opening failed.\n";
    exit(1);
  }

  fout.open("code_numbered.txt");
  if (fout.fail( ))
  {
    cerr << "Output file opening failed.\n";
    exit(1);
  }

  char next;
  int n = 1;
  fin.get(next); //THE ARGUMENT OF get() IS PASSED BY VALUE OR BY REFERENCE?
  fout << n << " ";
  while (! fin.eof( )) //returns true if the program has read past the end of the input file;
                       //otherwise, it returns false
  {
    fout << next;  //NOTE: get() READS SPACE AND NEWLINE CHARACTERS
    if (next == '\n')
    {
      n++;
      fout << n << ' ';
    }
    fin.get(next);
  }

  fin.close( );
  fout.close( );

  return 0;
}

TO DO BY STUDENTS:
try with an empty file; see what happens; solve the "problem"
TIP: investigate the use of get()
```

```
//Appending data to the end of a text file

#include <iostream>
#include <fstream>

using namespace std;

int main( )
{
  ofstream fout;
  fout.open("numbers.txt", ios::app); //TO DO: try with a non-existing file
  fout << "Appended data:\n";
  for (int i=10; i<=19; i++)
    fout << i << endl;
  fout.close( );
  return 0;
}
```