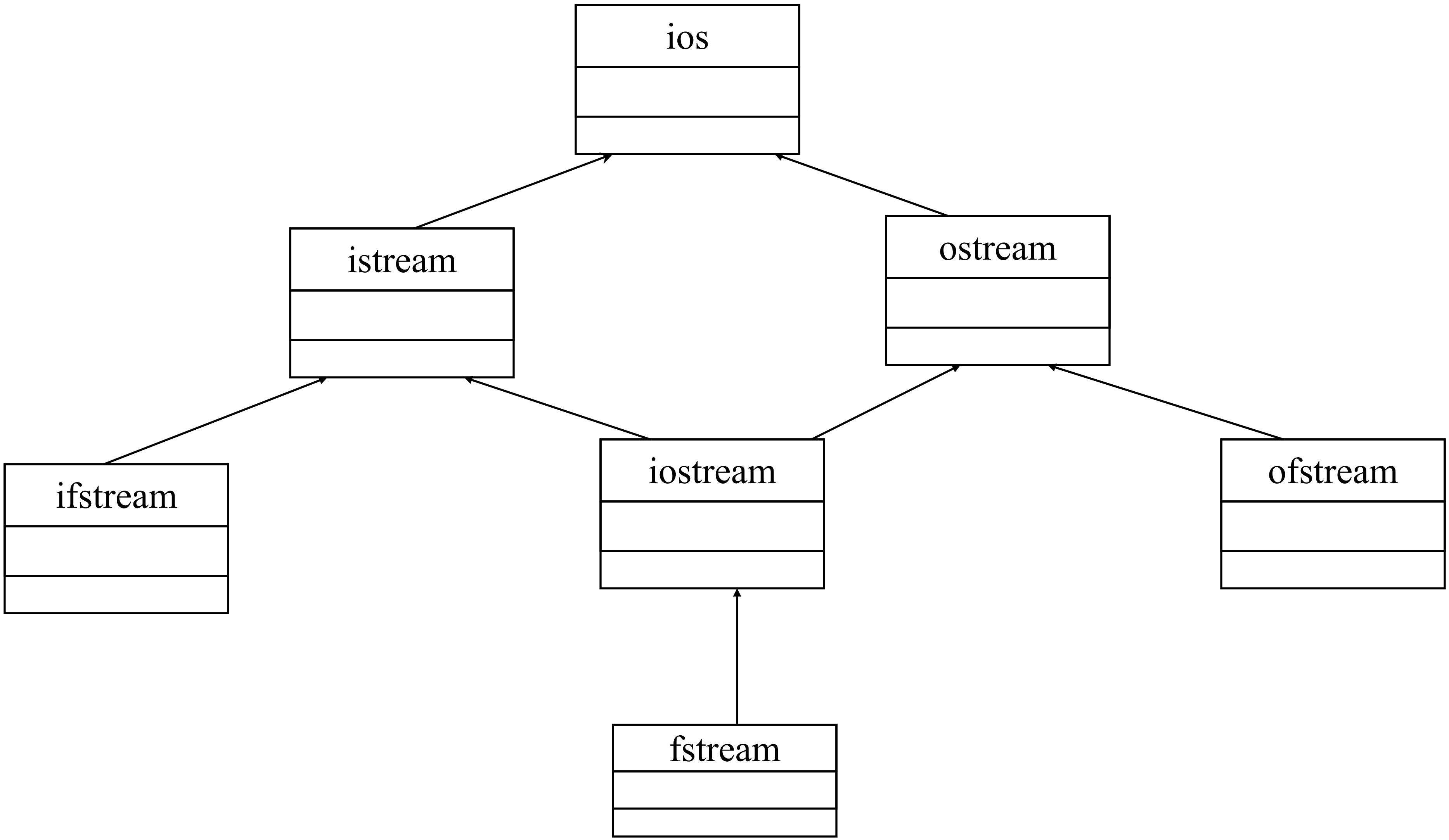
# C++ I/O streams

## The C++ I/O Library - Continued

- Input/output facilities are not defined within the C++ language.
- Its implemented in C++ and provided as a component of a C++ standard library.
- As its lowest level, a file is interpreted simply as a sequence, or stream of bytes. One aspect of the I/O library is to manage the transfer of these bytes.
- At the user level a file consists of a sequence of possibly intermixed data types.

# Simplified I/O Class Hierarchy



## Stream Basics

- In C++ a ***stream*** represents a flow of data; more specifically, it is a sequence of bytes
  - What the bytes mean or how they are to be interpreted is not important to the stream

- A stream is an *object* – an instance of a *class*

- Using streams, you can…
  - Read data into your program and write from your program
  - Communicate with external media

- The above is accomplished by *connecting* streams to things such as files, memory, or the terminal

## Stream Basics

- You have already used streams in your programs:
  - Input stream object/variable: `cin`
  - Output stream object/variable: `cout`

- The `cout` object also has several *member functions* (manipulators) that allow you to format the data (these are found in `iomanip`)

```
setiosflags()   setprecision()  setw()
```

- In C++, all I/O is performed using streams into which you place data you wish to output, and from which you extract information from the user
- There are two general classes of streams:
  - Input stream class (data flows into your program): `istream`
  - Output stream class (data flows out of your program): `ostream`

## Stream Basics

- For any stream, the first step is to connect the stream with the device (e.g., keyboard, monitor, file, etc.) with which you want to communicate

- For standard I/O (`cin` and `cout`) these connections are already made with the keyboard and monitor respectively
  - This means you can just use these streams directly in your programs and functions

- For other types of streams, this connection needs to be done explicitly

## File Streams

- File I/O is basically treated the same as standard I/O
  - Once a file stream is *connected* to a file, then the operations to actually manipulate the data are the same as with standard streams (because they are both streams)

- Files must be connected to streams within your program by ***opening*** a file

- You must define a stream object for *each* file you wish to use *simultaneously*

- The file stream classes are contained in the library **`fstream`**

## File Streams

- There are three steps associated with using files for I/O:
  - Step 1: Declare the file stream objects – create the streams
  - Step 2: Connect the objects to files – open the files
  - Step 3: Disconnect the objects from files – close the files

- Between steps 2 and 3, you can use the stream in much the same way as you use `cin` and `cout` except that instead of reading from the keyboard and writing the screen, you will be reading and writing from file
  - Details on using file streams are presented below

## File Streams

- As with any other variable, file streams must be declared before they can be used

- Use the `ifstream` and `ofstream` types for input and output, respectively

```
ifstream in_stream;      // used for input
ofstream out_stream;     // used for output
```

- The above code declares two variables; one for input and one for output
  - However, these variables are **not yet** connected to any files

## File Streams

- Once the objects are declared, we can connect them to files using the `open()` member function as follows

```
in_stream.open( “input.txt” );
out_stream.open( “output.txt” );
```

- You can also connect to a file when you declare it (similar to initializing a variable)

```
ifstream in_stream( “input.txt” );
ofstream out_stream( “output.txt” );
```

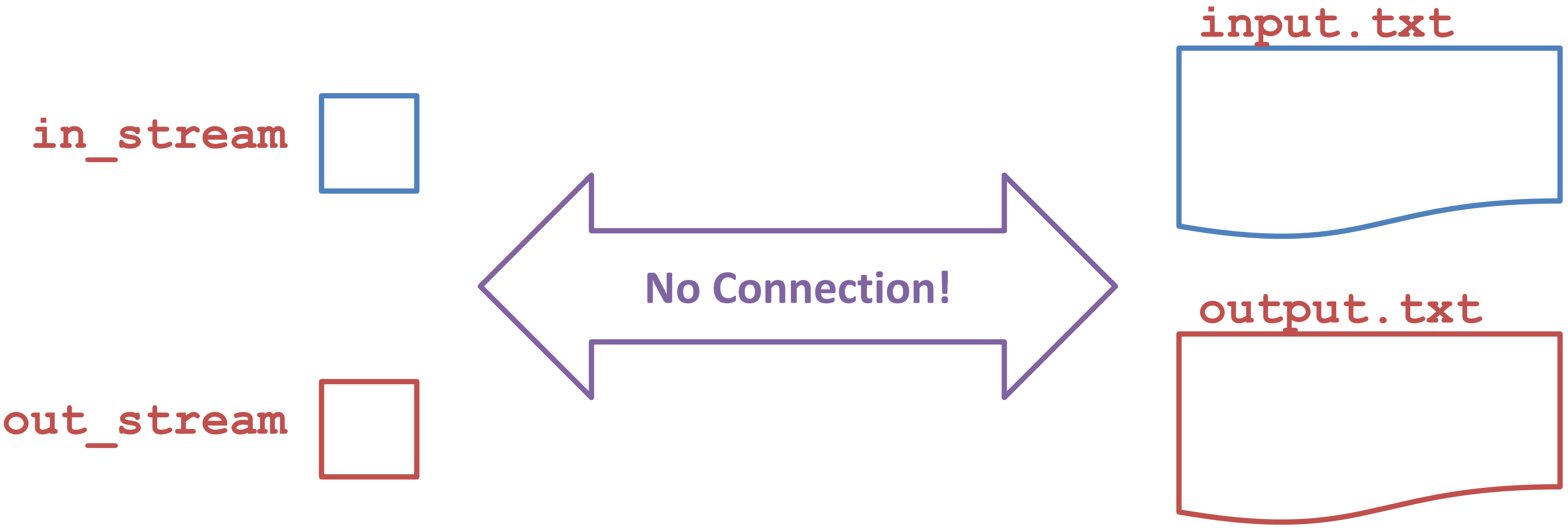
- For now, this is the approach we will use
  – You can also use strings to specify the file name at runtime (e.g. prompt the user), but more on that later

## File Streams

- To understand what is happening when we open a file, consider the following code segment

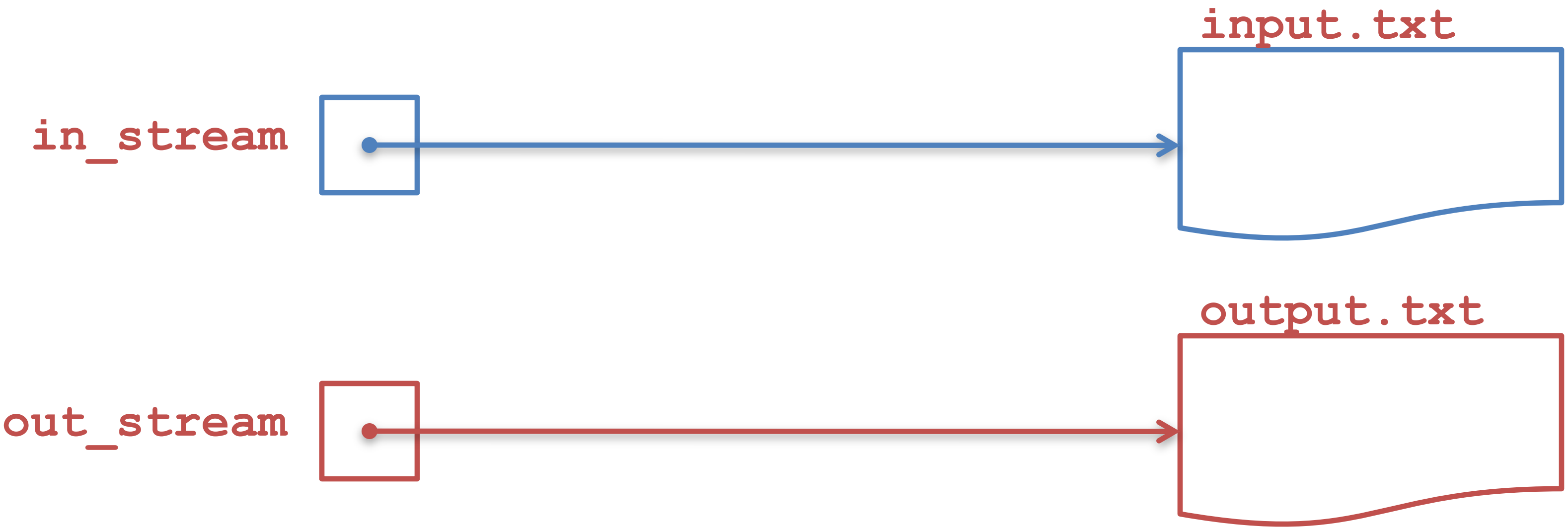
```
ifstream in_stream;
ofstream out_stream;

in_stream.open( “input.txt” );
out_stream.open( “output.txt” );
```

- Before opening the file, there is no link (connection) between the stream objects and the files



## File Streams

- After opening the file, there is now a link (connection) between the stream objects and the files



- We can now read information (i.e., bytes) from input.txt and we can write information (i.e., bytes) to output.txt
  - By analogy to standard I/O, **`input.txt`** is like the keyboard (cin), whereas **`output.txt`** is like the monitor

## File Streams

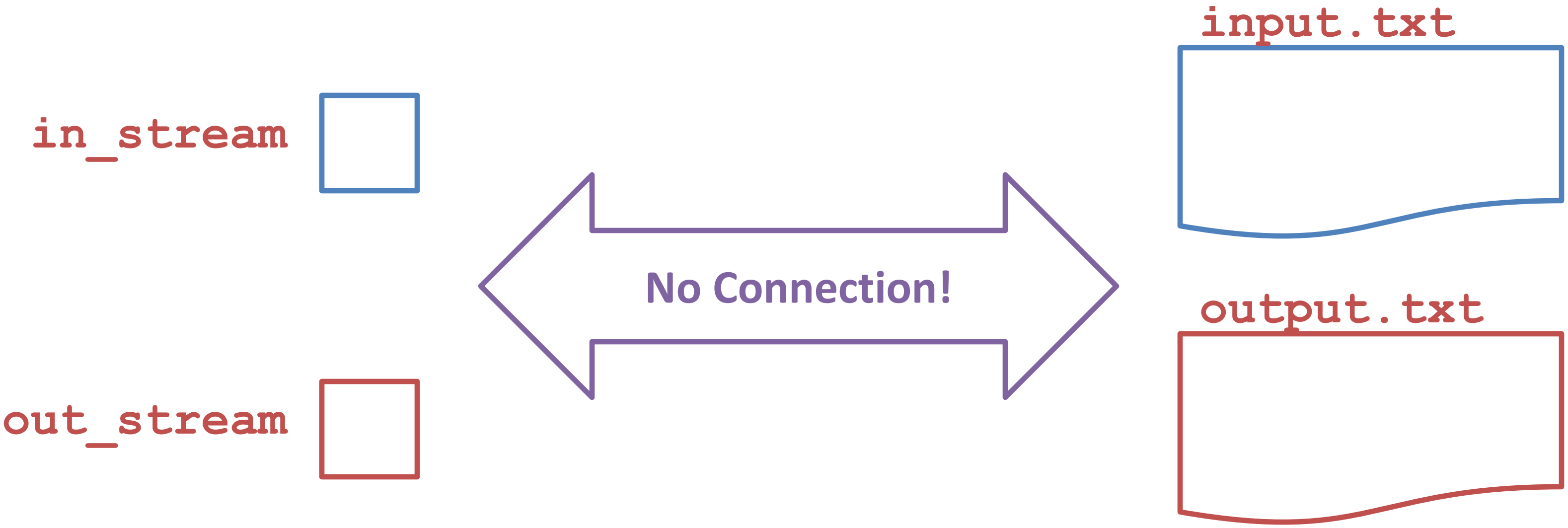
- When you are finished getting the input, or sending the output, you should close the file
  - This *disconnects* the stream from the file
  - Use the `close()` member function to do this

```
in_stream.close();    // no input arguments needed
out_stream.close();   // no input arguments needed
```

- If your program ends normally, the objects will automatically disconnect themselves from the files, even if the `close()` function is not called
  - However, if the program ends abnormally, the file may be corrupted

## File Streams

- By closing a file stream, you break the connection that was made when you opened the file
- Returning to our example from a few slides ago, after closing the two files, there is no connection between the streams and the files



## Testing for Proper Connection

- When connecting to a file for input, it is a good idea to make sure that the connection was successful before trying to read the data
  - For example, what if the file does not exist on your computer?
- Checking that the connection was successful is done using the `fail()` member function
  - This function returns a value of type `bool` which can be used to take appropriate actions
  - A return value of `true` means that a problem occurred

```
ifstream in_file( “input.dat” );

if( in_file.fail() )
{
    cout << “Error opening file...” << endl;

    // other statements to deal with the error
    // (e.g. prompt for new file name)
}
```

## Testing for Proper Connection

- If an input file does not exist, you can do one of the following
  - Allow the user to enter a new file name
  - Terminate the program with an appropriate message

- You can exit a program at any time using the `exit()` function
  - The input argument is an integer
  - By convention, a value of 1 is used in the case of an error, and 0 otherwise
  - To use `exit()`, you must include the `cstdlib` library in your program

```
ifstream in_file( “input.dat” );

if( in_file.fail() )
{
   cout << “Error opening file...quitting\n”;
   exit(1);
}
```

- Only use `exit()` when there is no other alternative!

## Testing for Proper Connection

- If you are opening a file for output that does not exist, the file will automatically be created
- However, if it already exists, then the existing file will be emptied and any data that was in that file will be lost!
- If you don’t want to lose existing data in a file, you can append the output to the end of the file
  - This may screw up your formatting, but at least you keep your data!

```
ofstream out_file( “output.dat”, ios::app );
```

**Specifies that the files should be appended to**

- In this context, **`app`** is short for “append”

## Testing for Proper Connection

- The `clear()` member function clears all the flags associated with a stream (e.g., `fail`)

```
// assume “BadFileName.txt” does NOT exist on your machine
ifstream in( “BadFileName.txt” );

if( in.fail() )
{
    cout << “The Stream Failed\n”;

    in.clear(); // needed to clear the failure flag

    // assume “GoodFileName.txt” DOES exist on your machine
    in.open( “GoodFileName.txt” );

    if( in.fail() )
        cout << “Doesn’t Work\n”;
    else
        cout << “Hurray!!!!\n”;
}
```

- **Note: member function fail can be used for testing other operations such as failure in reading.**

## Testing for End of File

- There are often times when you do not know how many input records you will encounter
  - Ideally, you want to stop reading at the end of the file
  - The `ifstream` and `ofstream` classes have a flag that is set when the end of file (EOF) is reached
  - To access this flag, use the `eof()` member function

- `eof()` returns `true` if the end of file was reached or `false` otherwise

```
ifstream infile( “input.dat” );
int a;

// read until the end of the file...
while( !infile.eof() )
{
   infile >> a;
   cout << a << “ was read from file\n”;
}
```

## Prompting for File Names

```
void main()
{
   string input_file_name;
   string output_file_name;

   cout << “Enter input file name: ” << endl;
   cin >> input_file_name;
   cout << “You entered “ << input_file_name << endl;

   ifstream in_file( input_file_name);
  // check that the file was opened properly (omitted)

   cout << “Enter output file name: ” << endl;
   cin >> output_file_name;
   cout << “You entered “ << output_file_name << endl;

   ofstream out_file( output_file_name.c_str(), ios::app );
  // check that the file was opened properly (omitted)

  // use the streams here

  // close the streams
  in_file.close();
  out_file.close();
}
```

## Reading a Full Line

- We saw that you could read multi-word input from the keyboard using `cin`

```
string line_of_text;
cout << “Enter a line of text: “;
getline( cin, line_of_text, ‘\n’ );
```

- As you might expect you can also read a full line from file using

```
ifstream infile( “input.dat” );
// check that file opened properly here

string line_of_text;
getline( infile, line_of_text );
// default delimiter is newline
```

## Streams – Example 1

```
#include <fstream>     // why not <iostream>?
#include <cstdlib>
using namespace std;

void main()
{
   int num1, num2, num3, num4;

   ifstream infile( “input.dat” );
   // check that input file was opened properly HERE!

   ofstream outfile( “output.dat” );

   infile >> num1 >> num2 >> num3 >> num4;

   outfile << “The four numbers read were:\n”
           << num1 << endl << num2 << endl
           << num3 << endl << num4 << endl;

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

## Streams – Example 1 con’t

- The “input.dat” file may look like any of the following

```
3      OR      3 6 -1 10      OR      3  6
6                                     -1 10
-1                                    14
10
```

- As written, the program does not care about the format of the input, it will just read four consecutive numbers
  - However, the file format can be very important in some cases
- In contrast, the “output.dat” file will always look as follows

```
The four numbers read were:
3
6
-1
10
```

# Part II
# More I/O Functions

## More I/O Stream Functions

- Member functions such as **get,** should be used for reading single characters (including white spaces), or a string of character (including white spaces). The following code reads from a stream (file or keyboard), until hits end of the file:

```
char ch = streamObject.get();
while ( !streamObject.eof() )
{
    streamObject << ch;
    char ch = streamObject.get();
}
```

## More I/O Stream Functions

- Another overloaded get member function can be used to read a sequence of character like a world or a sentence.
- In the following example, get extracts characters from the stream and stores them in *s* as a c-string, until either (n-1) characters have been extracted or the *delimiting character* is encountered:

```
char s[50];
streamObject.get(s, 50, ‘\n’);
while ( !streamObject.eof() )
{
  …



   }
```

- the delimiting character being either the newline character('\n') or other character as specified by the user (can be for example ‘|’, or ‘$’, etc.) The delimiting character is **not** extracted from the input sequence if found and remains there as the next character to be extracted from the stream (see getline for an alternative that does discard the delimiting character).
  A null character ('\0') is automatically appended to the written sequence if n is greater than zero, even if an empty string is extracted.

## More I/O Stream Functions

- Also member function **getline** can be used for reading string of characters (including white spaces) into an array of character.

```
char s[50];

streamObject.getline(s, 50, ‘\n’);
while ( !streamObject.eof() )
{
…
}
```

## More I/O Stream Functions

- The statement: `streamObject.getline(s, 50, '\n');`

  does the followings:
  - Removes at most 49 characters from the input stream up to and including the first occurrence of a newline but not beyond.
  - Stores all of the characters removed from the stream except the newline in s.
  - Places a null character, '\0', at the end of the characters stored in s.
- Other extraction operators
  - **Putback(ch)**: pushes back a character into the iostream.
  - **peek()**: returns next character but does not extract it.
  - **Ignore(int limit =1, int delim=EOF)**, discards up to ***limit*** characters and stops if encounters ***delim*** character.
- The put(char ch) member function can be used as an alternative method of inserting a character into the output stream.

  **streamObject.put(ch);**

# Part III
# Binary File I/O

## What is a Binary File

- Binary files are usually thought of a being a sequence bytes.
  - In fact the data will not be interpreted as a sequence of single characters like in a text file.
  - The data will be stored in the same format and sequence of bytes when used in you program.
    - A variable stored into double on the computer memory will be stored into a binary file in the same order and sequence of bytes.
- Example:
  - double x = 0.00887776665551, will be stored in a an 8-byes memory space. The same data in a text file will be stored in a 16-byte memory space.

## What is a  Binary File (continued)

- A binary file is normally more compressed that a text file.
  - Most digital data are stored in binary files
- Reading ad writing data from and into file are faster, using binary data.
- Binary file can be viewed or read properly like a text file using a text editor. Here is an example of a binary file that I opened by an editor on a Mac computer:

```
oe˙.¸†8![.[__text__TEXT#.‹a(.__debug_frame__DWARF$|‰cdebug_info__DWARF†.K`dc__debug_abbrev__DWARF.P.EW__debug_arange
s__DWARFT
Z__debug_macinfo__DWARFT‹Z__debug_loc__DWARFT‹Z__debug_pubnames__DWARFT.‹Z__debug_pubtypes__DWARF≥T$s[__debug_
str__DWARF◊T.[__debug_ranges__DWARF◊T.[__data__DATA◊T.[__StaIcInit__TEXT‡T{†[‹d.__bss__DATA[__cstring__TEXT[UCÄ__mod_ini
t_func__DATA†U`Ä
```

## Associating an Stream Object to a Binary File

- To connect a binary file to the program for input and/or output, you need to open the file in binary mode.
- Example:
  - ofstream outfile(“myouput.bin”, ios::out | ios::binary);
  - Ifstream infile(“input.bin”, ios::in | ios::binary):
- An fstream class object can be also used to open a file for both reading and writing:
  - fstream in_outfile(“myfile.bin”, ios::in | ios::out |ios::binary);

## Writing into a binary file:

- ‘write’ member function from istream library class can be used to read a sequence of byes. Here is the prototype of the member function:

  **ostream& write (char* address, streamsize n);** // streamsize is long int

- Notice that in the following example the name of the array which is an address is passed to the function read. The **cast** operator convert the double* to a char*.

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

int main(void)
{
   ofstream os ("test.bin", ios::binary);
   double a []= { 2.3, 3, 10, 44};
   os.write ((char*) (a), sizeof(a)); // writes 32 bytes – you can also use

   if (os)
      std::cout << "All 4 element of array a, 32 bytes, are written into test.bin.";
      else
         os.close();
   return 0;
}
```

# Writing into a binary file

You may also write the data in an array using a loop (one element at a time).

In the following example example, notice how the address of each element is used:

```
int main(void)
 {
    ofstream os ("test.bin", ios::binary);
    double a []= { 2.3, 3, 10, 44};

   for(int i =0; i < 4; i++)
    {
        os.write ((char*) (&a[i]),  sizeof(double));   // Writes 8 bytes in each iteration
        if(!os)
            cerr << "Sorry! faild to write into the the output file...\n”;
    }

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

## Reading from binary file

- To read from a binary file you can use member function ‘read’. Here is the prototype of this function:

  ```
  istream& read (char* address,  streamsize n);  // streamsize is a long int
  ```

- Reads n bytes from input stream and stores them in the memory space starting at the given address. Notice that member function gcount returns the number of bytes read from stream
- Example:

```
int main(void)
{
   ifstream is ("test.bin", ios::binary);
   double a [4];

   is.read ((char*) (a), sizeof(a))
   long int g = is.gcount();     // libray function gcount returns number bytes read from input stream
   if (is)
      std::cout << ”Values of 4 element of array are read from binary file test.bin\n" <<
      "Number of bytes read from test.bin is: " << g << endl;
   else
      is.close();
   return 0;
}
```

Program output:

```
Values of 4 element of array are read from binary file
test.bin
Number of bytes read from test.bin is: 32
```

## Reading from binary file

You may also read the data in an array using a loop and read one element at a time. In the following example example, notice how the address of each element is used:

```
int main(void)
{
   ifstream is ("test.bin", ios::binary);
   double a [4];

   for(int i = 0; i < 4; i++){
      is.read ((char*)(&a[i]), sizeof(double));
      long int g = is.gcount();  // returns 8 bytes in each read
      cout << "i = " << i << " and g = " << g << endl;
   }

  for(int i =0; i < 4; i++)
      cout << a[i] << endl;

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

Program output:

```
i = 0 and g = 8
i = 1 and g = 8
i = 2 and g = 8
i = 3 and g = 8

2.3
3
10
44
```