I/O and File Streams

Standard Output (cout)

- On most program environments, the standard output by default is the screen, and the C++ stream object defined to access it is cout.
- For formatted output operations, cout is used together with the insertion operator, which is written as << (i.e., two "less than" signs).

```
1 cout << "Output sentence"; // prints Output sentence on screen
2 cout << 120;               // prints number 120 on screen
3 cout << x;                 // prints the value of x on screen
```

Standard Input (cin)

- In most program environments, the standard input by default is the keyboard, and the C++ stream object defined to access it is cin.
- For formatted input operations, cin is used together with the extraction operator, which is written as >> (i.e., two "greater than" signs). This operator is then followed by the variable where the extracted data is stored. For example:

```
1 int age;
2 cin >> age;
```

- The extraction operation on cin uses the type of the variable after the >> operator to determine how it interprets the characters read from the input; if it is an integer, the format expected is a series of digits, if a string a sequence of characters, etc.

Standard Input/Output Example

```
1  // i/o example
2
3  #include <iostream>
4  using namespace std;
5
6  int main ()
7  {
8    int i;
9    cout << "Please enter an integer value: ";
10   cin >> i;
11   cout << "The value you entered is " << i;
12   cout << " and its double is " << i*2 << ".\n";
13   return 0;
14 }
```

```
Please enter an integer value: 702
The value you entered is 702 and its double is 1404.
```

cin and strings

- The extraction operator can be used on cin to get strings of characters in the same way as with fundamental data types.
- However, cin extraction always considers spaces (whitespaces, tabs, new-line...) as terminating the value being extracted, and thus extracting a string means to always extract a single word, not a phrase or an entire sentence.

```
1 string mystring;
2 cin >> mystring;
```

getline

- To get an entire line from cin, there exists a function, called getline, that takes the stream (cin) as first argument, and the string variable as second. For example:

```
1  // cin with strings
2  #include <iostream>
3  #include <string>
4  using namespace std;
5
6  int main ()
7  {
8    string mystr;
9    cout << "What's your name? ";
10   getline (cin, mystr);
11   cout << "Hello " << mystr << ".\n";
12   cout << "What is your favorite team? ";
13   getline (cin, mystr);
14   cout << "I like " << mystr << " too!\n";
15   return 0;
16 }
```

```
What's your name? Homer Simpson
Hello Homer Simpson.
What is your favorite team? The Isotopes
I like The Isotopes too!
```

stringstream

- The standard header <sstream> defines a type called stringstream that allows a string to be treated as a stream, and thus allowing extraction or insertion operations from/to strings in the same way as they are performed on cin and cout. This feature is most useful to convert strings to numerical values and vice versa. For example, in order to extract an integer from a string we can write:

```
1 string mystr ("1204");
2 int myint;
3 stringstream(mystr) >> myint;
```

Stringstream Example

```
1  // stringstreams
2  #include <iostream>
3  #include <string>
4  #include <sstream>
5  using namespace std;
6
7  int main ()
8  {
9    string mystr;
10   float price=0;
11   int quantity=0;
12
13   cout << "Enter price: ";
14   getline (cin,mystr);
15   stringstream(mystr) >> price;
16   cout << "Enter quantity: ";
17   getline (cin,mystr);
18   stringstream(mystr) >> quantity;
19   cout << "Total price: " << price*quantity << endl;
20   return 0;
21 }
```

```
Enter price: 22.25
Enter quantity: 7
Total price: 155.75
```

Input/output with files

- C++ provides the following classes to perform output and input of characters to/from files:
  - ofstream: Stream class to write on files
  - ifstream: Stream class to read from files
  - fstream: Stream class to both read and write from/to files.
- These classes are derived directly or indirectly from the classes istream and ostream. We have already used objects whose types were these classes: cin is an object of class istream and cout is an object of class ostream. Therefore, we have already been using classes that are related to our file streams. And in fact, we can use our file streams the same way we are already used to use cin and cout, with the only difference that we have to associate these streams with physical files

Open a file

- The first operation generally performed on an object of one of these classes is to associate it to a real file. This procedure is known as to open a file. An open file is represented within a program by a stream (i.e., an object of one of these classes; in the previous example, this was myfile) and any input or output operation performed on this stream object will be applied to the physical file associated to it.
- In order to open a file with a stream object we use its member function open:

  open (filename, mode);

- Where filename is a string representing the name of the file to be opened, and mode is an optional parameter with a combination of the following flags:

ios::in	Open for input operations.
ios::out	Open for output operations.
ios::binary	Open in binary mode.
ios::ate	Set the initial position at the end of the file. If this flag is not set, the initial position is the beginning of the file.
ios::app	All output operations are performed at the end of the file, appending the content to the current content of the file.
ios::trunc	If the file is opened for output operations and it already existed, its previous content is deleted and replaced by the new one.

Closing a file

- When we are finished with our input and output operations on a file we shall close it so that the operating system is notified and its resources become available again. For that, we call the stream's member function close. This member function takes flushes the associated buffers and closes the file

```
myfile.close();
```

- Once this member function is called, the stream object can be re-used to open another file, and the file is available again to be opened by other processes.
- In case that an object is destroyed while still associated with an open file, the destructor automatically calls the member function close.

Text files

- Text file streams are those where the ios::binary flag is not included in their opening mode. These files are designed to store text and thus all values that are input or output from/to them can suffer some formatting transformations, which do not necessarily correspond to their literal binary value.

```cpp
// writing on a text file
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  ofstream myfile ("example.txt");
  if (myfile.is_open())
  {
    myfile << "This is a line.\n";
    myfile << "This is another line.\n";
    myfile.close();
  }
  else cout << "Unable to open file";
  return 0;
}
```

```
[file example.txt]
This is a line.
This is another line.
```

Reading from a file

```
1  // reading a text file
2  #include <iostream>
3  #include <fstream>
4  #include <string>
5  using namespace std;
6
7  int main () {
8    string line;
9    ifstream myfile ("example.txt");
10   if (myfile.is_open())
11   {
12     while ( getline (myfile,line) )
13     {
14       cout << line << '\n';
15     }
16     myfile.close();
17   }
18
19   else cout << "Unable to open file";
20
21   return 0;
22 }
```

```
This is a line.
This is another line.
```

Get and Put Stream Positioning

- All i/o streams objects keep internally -at least- one internal position:
- ifstream, like istream, keeps an internal get position with the location of the element to be read in the next input operation.
- ofstream, like ostream, keeps an internal put position with the location where the next element has to be written.
- Finally, fstream, keeps both, the get and the put position, like iostream.
- These internal stream positions point to the locations within the stream where the next reading or writing operation is performed.

Stream Positioning

- positions can be observed and modified using the following member functions:
- **tellg() and tellp()**
  - These two member functions with no parameters return a value of the member type streampos, which is a type representing the current get position (in the case of tellg) or the put position (in the case of tellp).
- **seekg() and seekp()**
  - These functions allow to change the location of the get and put positions. Both functions are overloaded with two different prototypes. The first form is:
    - seekg (position);
    - seekp (position);
  - Using this prototype, the stream pointer is changed to the absolute position position (counting from the beginning of the file). The type for this parameter is streampos, which is the same type as returned by functions tellg and tellp.
  - The other form for these functions is:
    - seekg (offset, direction);
    - seekp (offset, direction);

seekg() and seekp()

- Using this prototype, the get or put position is set to an offset value relative to some specific point determined by the parameter direction. offset is of type streamoff. And direction is of type seekdir, which is an enumerated type that determines the point from where offset is counted from, and that can take any of the following values:

ios::beg	offset counted from the beginning of the stream
ios::cur	offset counted from the current position
ios::end	offset counted from the end of the stream

Stream Positioning Example

```cpp
// obtaining file size
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  streampos begin,end;
  ifstream myfile ("example.bin", ios::binary);
  begin = myfile.tellg();
  myfile.seekg (0, ios::end);
  end = myfile.tellg();
  myfile.close();
  cout << "size is: " << (end-begin) << " bytes.\n";
  return 0;
}
```

```
size is: 40 bytes.
```

Binary files

- For binary files, reading and writing data with the extraction and insertion operators (<< and >>) and functions like getline is not efficient, since we do not need to format any data and data is likely not formatted in lines.
- File streams include two member functions specifically designed to read and write binary data sequentially: write and read. The first one (write) is a member function of ostream (inherited by ofstream). And read is a member function of istream (inherited by ifstream). Objects of class fstream have both. Their prototypes are:
  - write (memory_block, size);
  - read (memory_block, size);
- Where memory_block is of type char* (pointer to char), and represents the address of an array of bytes where the read data elements are stored or from where the data elements to be written are taken. The size parameter is an integer value that specifies the number of characters to be read or written from/to the memory block.

Binary Files Example

```cpp
// reading an entire binary file
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  streampos size;
  char * memblock;

  ifstream file ("example.bin", ios::in|ios::binary|ios::ate);
  if (file.is_open())
  {
    size = file.tellg();
    memblock = new char [size];
    file.seekg (0, ios::beg);
    file.read (memblock, size);
    file.close();

    cout << "the entire file content is in memory";

    delete[] memblock;
  }
  else cout << "Unable to open file";
  return 0;
}
```

```
the entire file content is in memory
```

Exercise 1

- Write a c++ code to write all the **EVEN** numbers from 1 to 20 into a text file called “**numbers.txt**”.

Exercise 2

- Write a c++ code to read all the numbers from the text file “number.txt” after executing Exercise 1, print the numbers, and calculate and print the sum of the numbers.