Object-oriented Programming

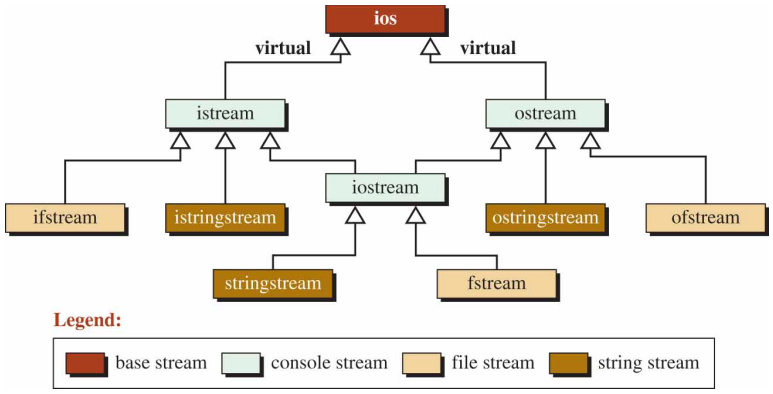
Input / Output Streams Part 2

YoungWoon Cha

Computer Science and Engineering

Stream Classes

To handle input/output operations, the C++ library defines a hierarchy of classes.



The ios Class

At the top of the hierarchy is the *ios* class that serves as a virtual base class for all input/output classes.

It defines data members and member functions that are inherited by all input/output stream classes.

Since the *ios* class is never instantiated, it does not use its data members and member functions. They are used by other stream classes.

Other Classes

For convenience, we refer to *istream*, *ostream*, and *iostream* as console classes (they are used to connect our program to the console).

We refer to *ifstream*, *ofstream*, and *fstream* as file streams (they are used to connect our program to files).

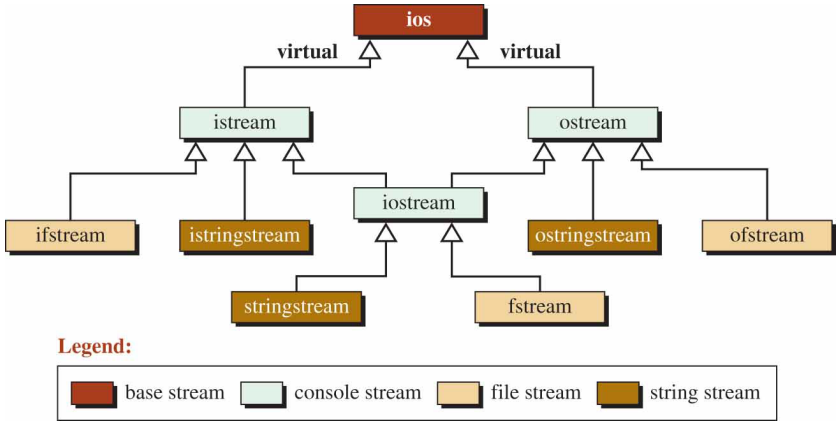
We refer to *istringstream*, *ostringstream*, and *stringstream* as string streams.

KU KONKUK UNIVERSITY

File Streams

FILE STREAMS

We need to use the file stream classes defined in the <fstream> header to connect the files to our program so we can read or write files.



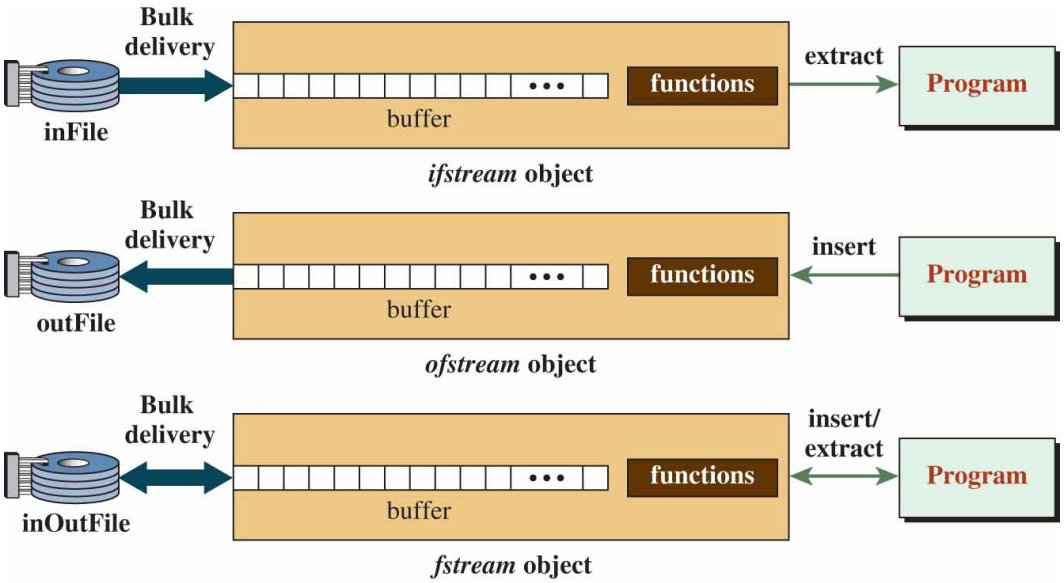
File stream classes are made from three classes: *ifstream*, *ofstream*, and *fstream*.

They are inherited respectively from the *istream*, *ostream*, and *iostream* classes and inherit all member functions of these classes.

The file stream classes define some new functions, mostly for instantiation and opening files.

File Input/Output Part 1

In the case of file input/output, the source is a file or the sink is a file.



Construct the Stream

We can instantiate any of the three streams as shown below.

To do so we need to include the <fstream> header file in our program.

```
ifstream inStream;
ofstream outStream;
fstream inOutStream;
```

File Input/Output Part 2

Connect the File (Open)

To be able to read from or write to a file, we need to connect the instantiated stream to the corresponding file using a member function named *open*.

The *open* function uses the file name (a C-string) as the first parameter and an open mode as the second.

The parameters *inFile*, *outFile*, and *inOut*File are file names.

```
inStream.open(const char* inFile, …);
outStream.open(const char* outFile, …);
inOutStream.open(const char* inOutFile, …);
```

```
inStream.is_open()
outStream.is_open()
inOutStream.is_open()
```

Testing Opening Success

A file is an external entity to our program.

An input file may have been deleted or corrupted or an output file cannot be open because the disk is full.

The three file stream classes provide a function to test that the file was opened successfully and connected to the stream.

The result is a Boolean *true* or *false* value.

File Input/Output Part 3

Read and Write

There are no new read and write member functions defined for the file streams; these streams inherit the member functions defined for the console streams as discussed before.

Disconnect the File (Close)

After we are done with the files, we need to close them using the *close* member function.

Since a stream can be connected to only one file at a time, the close member function has no parameter.

```
inStream.close();
outStream.close();
inOutStream.close();
```

Destruct the Stream

This step is done automatically when the stream object goes out of scope.

File Input/Output Part 4

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

int main ()
{
      // Instantiation of an ofstream object
      ofstream outStrm;
      // Creation of a file and connecting it to the ofstream object
      outStrm.open ("integerFile");
      if (!outStrm.is_open())
      {
           cout << "integerFile cannot be opened!";
           assert (false);
      }
      // Writing to the file using overloaded insertion operator
      for (int i = 1; i <= 10; i++)
      {
           outStrm << i * 10 << " ";
      }
      // Closing the file
      outStrm.close ();
      // The ofstream object is destroyed after return statement
      return 0;
}
```

Nothing will be output to the monitor from the Program when it is run, but we can check the contents of a file named *integerFile* (a data file) with a text editor to find that the following line was stored in it.

```
10 20 30 40 50 60 70 80 90 100
```

File Input/Output Part 5

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

int main ()
{
      int data;
      // Instantiation of an ifstream object
      ifstream inStrm;
      // Connection of the existing file to the ifstream object
      inStrm.open ("IntegerFile");
      if (!inStrm.is_open())
      {
           cout << "integerFile cannot be opened!";
           assert (false);
      }
      // Reading from the ifstream object and writing to the cout object
      for (int i = 1; i <= 10; i++)
      {
           inStrm >> data ;
           cout << data << " " ;
      }
      // Disconnection of the IntegerFile from the ifstream
      inStrm.close ();
      // Destruction of the ifstream object is done after return
      return 0;
}
```

Note that this time we can see the integers printed on the monitor. We read from the *integerFile* using the extraction operator. This operator skips whitespace and reads the data. After each read, we write the data to the *cout* object (monitor).

```
Run:
10 20 30 40 50 60 70 80 90 100
```

Opening Modes Part 1

The opening mode is a type defined in the *ios* class and inherited by all classes in the hierarchy of stream classes.

They are not used in the console streams or string streams because we do not open them; they are used only in the file streams.

The opening mode is defined as a type, a *bitfield*, with six bits that are used independently or in conjunction with the other bits.

`ios :: in`	open for input
`ios :: out`	open for output; the contents are destroyed
`ios :: app`	open for output and write at the end (append)
`ios :: ate`	Move to the end immediately after opening (at end)
`ios :: trunc`	truncate the file to zero length
`ios :: binary`	read and write in binary mode (default is text)

```
enum _Openmode { // constants for file opening options
    _Openmask = 0xff
};

static constexpr _Openmode in         = static_cast<_Openmode>(0x01);
static constexpr _Openmode out        = static_cast<_Openmode>(0x02);
static constexpr _Openmode ate        = static_cast<_Openmode>(0x04);
static constexpr _Openmode app        = static_cast<_Openmode>(0x08);
static constexpr _Openmode trunc      = static_cast<_Openmode>(0x10);
static constexpr _Openmode _Nocreate  = static_cast<_Openmode>(0x40);
static constexpr _Openmode _Noreplace = static_cast<_Openmode>(0x80);
static constexpr _Openmode binary     = static_cast<_Openmode>(0x20);
```

Opening Modes Part 2

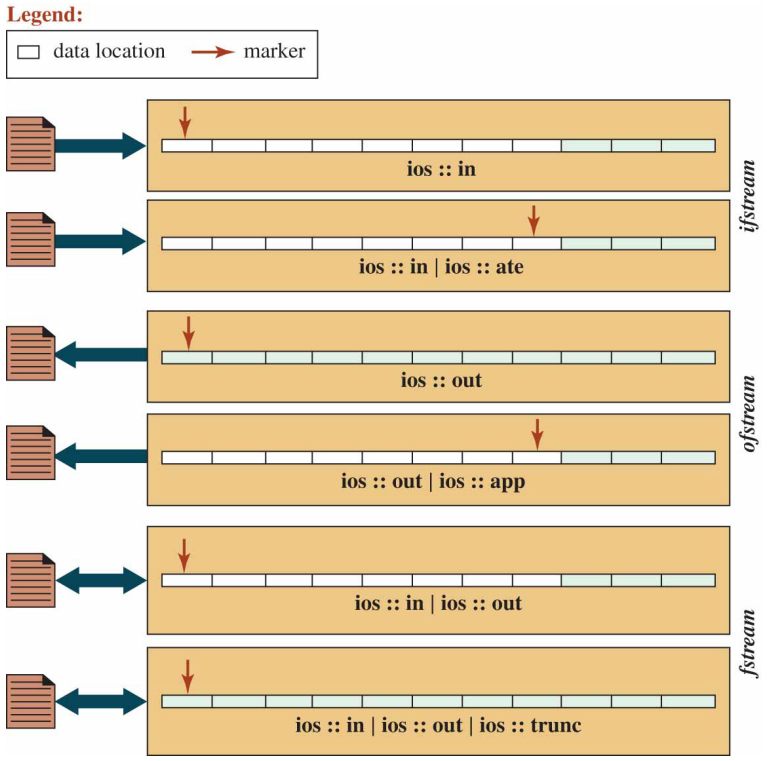
Opening Modes for Input

Normally we use one of the two modes for an input file. The first, (ios :: in), opens the file for input and puts the marker at the first byte of the buffer.

This allows us to start reading the file from the first byte.

With each read the marker moves to the next byte until we reach an empty position.

In this case the *eofbit* is set and the file can no longer be read.



In the second mode, (ios :: in | ios :: ate), the file is opened for reading, but the marker goes to the position after the last byte.

This means the *eofbit* is set and we cannot read.

However, it has other applications such as we can read the bytes in reverse by moving the marker toward the beginning.

Another application is that we can find the size the of the file as we demonstrate later.

Opening Modes Part 3

The Program copies the contents of *file*1 to the monitor.

We have only one stream to instantiate (*cout* is already done for us).

We can use the first mode (ios :: in) to read the contents of a file named file1 whose contents are shown here:

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

int main ()
{
      // Variable declaration
      char ch;
      // Instantiation of an ifstream object
      ifstream istrm ;
      // Opening file1 and testing if it is opened properly
      istrm.open ("file1", ios :: in);
      if (!istrm.is_open())
      {
            cout << "file1 cannot be opened!" << endl;
            assert (false);
      }
      // Reading file1 character by character and writing to monitor
      while (istrm.get (ch))
      {
            cout.put(ch);
      }
      // Closing stream
      istrm.close ();
      return 0;
}
```

This is the file that we want to open and read its contents.

We need to open only one file (*cout* is done for us)

We close only one file (cout is automatically done for us).

We can see the data when we run program.

```
Run:
This is the file that we want to open
and read its contents.
```

Opening Modes Part 4

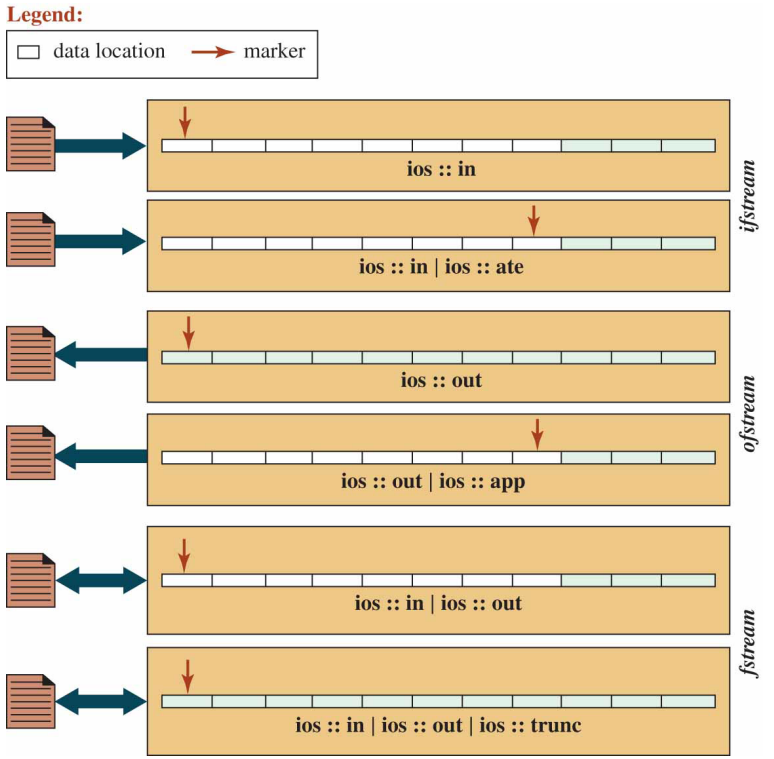
Opening Modes for Output

We can use two output modes to write to a file.

In the first mode (ios :: out) we open the file for output. If the file contains any data, they are deleted.

In the second mode (ios :: out | ios :: app) we open the file and write at the end of the file (append).

The existing data are preserved.



Opening Modes Part 5

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

int main ()
{
	// Variable declaration
	char ch;
	// Instantiation of an ifstream and an ofstream object
	ifstream istr;
	ofstream ostr;
	// Opening file1 and file2 and testing if they are open
	istr.open ("file1", ios :: in);
	if (!istr.is_open())
	{
		cout << "file1 cannot be opened!" << endl;
		assert (false);
	}
	ostr.open ("file2", ios :: out);
	if (!ostr.is_open())
	{
		cout << "file2 cannot be opened!" << endl;
		assert (false);
	}
	// Reading file1 character by character and writing to file2
	while (istr.get (ch))
	{
		ostr.put(ch);
	}
	// Closing file1 and file2
	istr.close ();
	ostr.close ();
	return 0;
}
```

The Program copies the contents of *file*1 to *file*2.

We need to instantiate two streams.

Note that we open *file*1 for input using opening mode (ios :: in), which means that marker is set on the first byte of the buffer.

We open *file*2 using the opening mode (ios :: out), which means the buffer is emptied and the marker is set to the first byte.

We read a character at a time from *file*1 and write it to *file*2.

The file markers in both files are moving in sequence.

We can see nothing, but we can open *file*2 to see that it is the exact copy of *file*1.

Appending to a File

We decided to add the date at the end of *file*1.

We can open the file and append the date, as a C-string, at the end of the file.

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

int main ()
{
      // Instantiate a ostream object
      ofstream ostr;
      // Open file1 and connect it to the ostream object
      ostr.open ("file1", ios :: out | ios :: app);
      if (!ostr.is_open())
      {
           cout << "file1 cannot be opened!";
           assert (false);
      }
      // Append the date as a C-string to file1
      ostr << "\nOctober 15, 2016.";
      // Close the file
      ostr.close ();
      return 0;
}
```

This is the file that we want to open and read its contents.

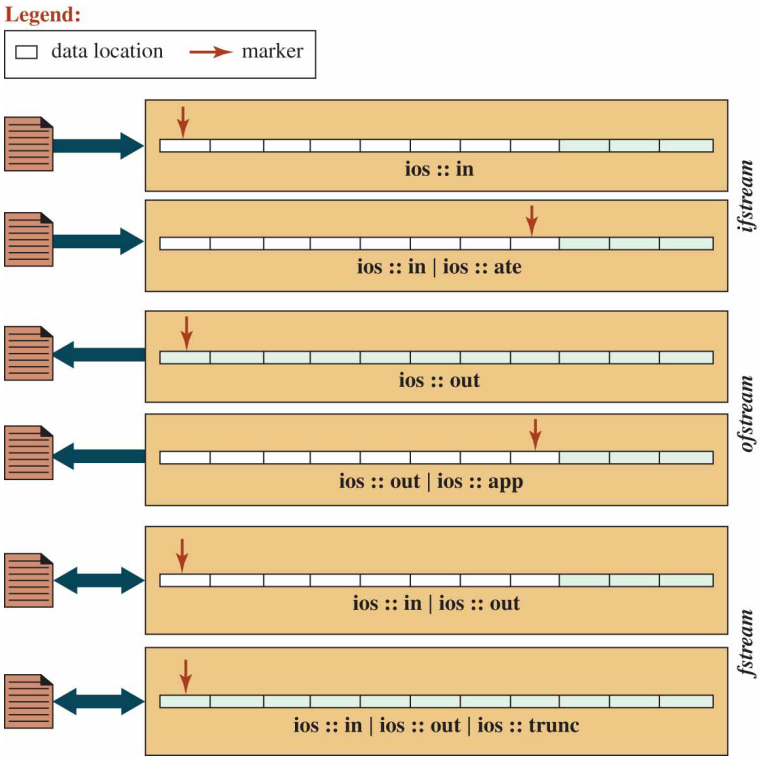
When we open *file1* with a text editor, we see that the date has been added at the end of the file (in a new line) as shown below:

```
This is the file that we want to open and
read its contents.
October 15, 2015.
```

Opening Modes Part 6

Opening for Input/Output

We can open a file for input/output if we connect our file to an *fstream* object using the mode (ios :: in | ios :: out). (r+ in C)



Opening a File for Both Input and Output

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

int main ()
{
      // Instantiate an fstream object
      fstream fstr;
      // Open the intFile and connected to the
      fstream object
      fstr.open ("intFile", ios :: in | ios :: out);
      if (!fstr.is_open())
      {
            cout << "intFile cannot be opened!";
            assert (false);
      }
      // Read all integers and add to the sum until
      end of file is detected
      int num;
      int sum = 0;
      while (fstr >> num)
      {
            sum += num;
      }
      // Clear the file and add the message and the
      sum to the file
      fstr.clear ();
      fstr << "\nThe sum of the numbers is: ";
      fstr << sum;
      // Close the stream
      fstr.close ();
      return 0;
}
```

Assume that we have a file of integers.

```
12 14 17 20 21 25 32 27 56 18
```

We decide to add the sum of the integers at the end of the file.

We can open the file for both read and write (ios :: in | iso :: out).

We need to read the integers, one by one (input), and then write the sum at the end of the file (output).

When the loop terminates the *eofbit* is set and we cannot use the stream.

We clear the stream and then write the message and the sum to the end of the file.

The result is a file with the following contents.

```
12 14 17 20 21 25 32 27 56 18
The sum of the numbers is: 242
```

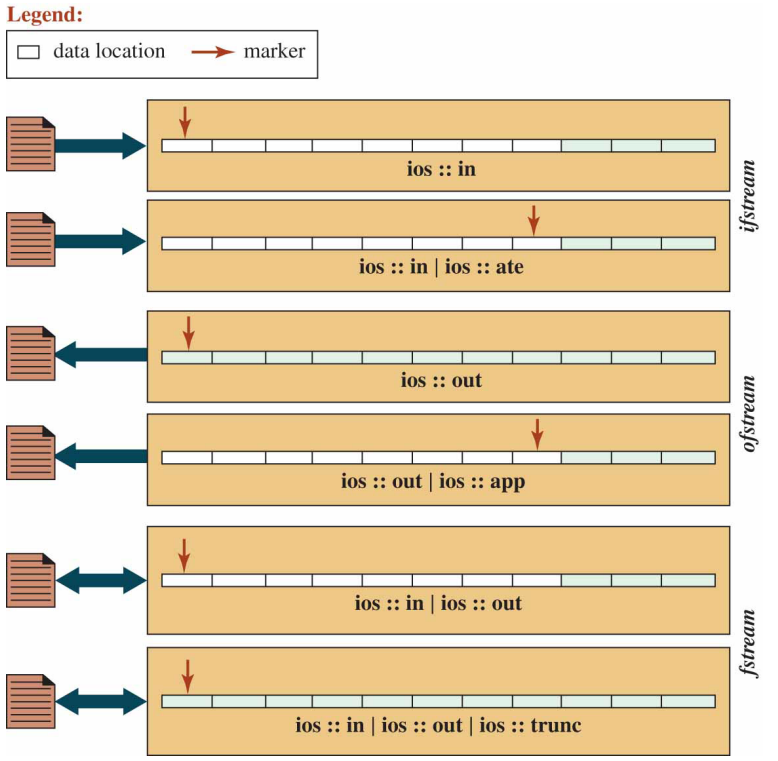
Opening Modes Part 7

The file mode (ios: in | ios: out | ios : trunc) allows us to open the file, truncate its contents, and write new data to it. (w+ in C)

This is the same as the mode (ios :: out) with one difference. (read and write)

We create a new file or delete the contents of an old file and replace it with new data.

The method is used when we want to keep the name of the file, but with new contents.



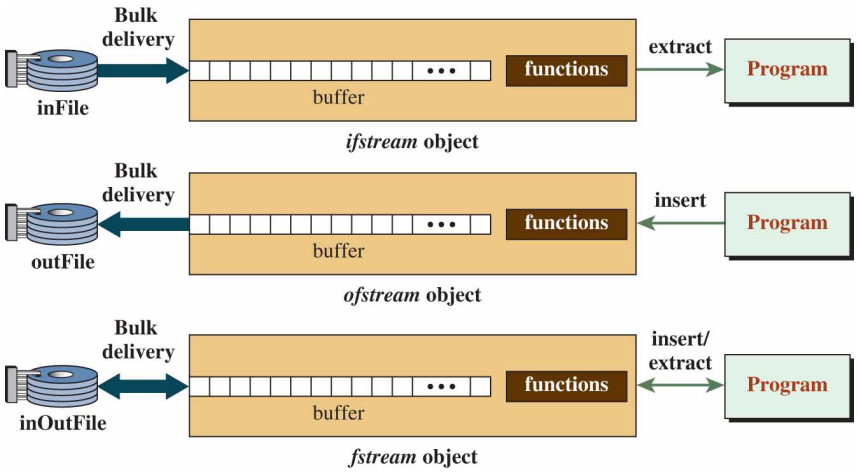
Sequential Versus Random Access Part 1

A file is a sequential collection of 8-bit bytes.

We actually do not read or write to a file directly; we read from or write to the buffer in the stream object.

In reading, the contents of the of the file are copied to the buffer by the system in bulk movement; in writing the contents of the buffer are moved from the buffer to the file in bulk movement.

Transferring bytes from the file to the buffer or from the buffer to the file is out of our control.



When we talk about the sequential or random access, we do not mean that the file is arranged sequentially or randomly (a file is always a sequential collection of bytes), we mean how we access the buffer in the stream object: sequentially or randomly.

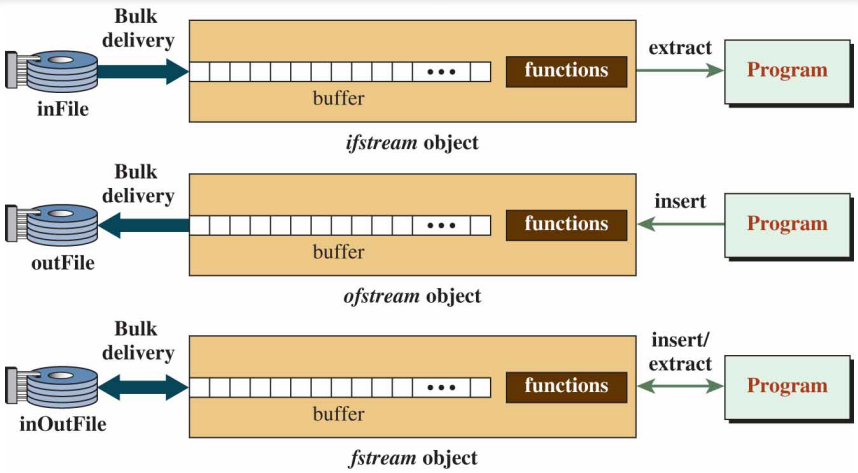
If we think about the buffer, we see that it is like an array; we can always access an array either sequentially (element after element) or randomly (any element we want).

Sequential Versus Random Access Part 2

The buffer in an *ifstream* has a marker that points to the next byte to be read.

The buffer in an *ofstream* has a marker that points to the next byte to be written.

The buffer in an *fstream* has only one marker for read or write.



Sequential Access

The movement of the stream marker is controlled by the read/write functions.

The marker starts at the beginning of the buffer when the file is opened.

It moves toward the end of the buffer with each read or write.

The marker is pointing to the next byte to be read or written, but the number of bytes in each read or write depends on the type of data.

If we are reading or writing characters, movement is one byte at a time; if we are reading or writing formatted data the movement can be a number of byes for each read or write.

Sequential Versus Random Access Part 3

Random Access

Reading or writing using the console system can be done only sequentially, but we can use random access when we use the file stream.

In random access, we can read data from any location and we can write data to any location.

We simply move the stream marker to the corresponding character.

ios :: beg	ios :: cur	ios :: end

direction (dir) values

Input	Output
int tellg() istream& seekg(int *pos*) istream& seekg(int *off*, ios :: dir)	int tellp(); ostream& seekp(int *pos*) ostream& seekp(int *off*, ios :: dir)

```
enum _Seekdir { // constants for file positioning options
    _Seekbeg,
    _Seekcur,
    _Seekend
};

static constexpr _Seekdir beg = _Seekbeg;
static constexpr _Seekdir cur = _Seekcur;
static constexpr _Seekdir end = _Seekend;
```

Finding the Index of Current Location

The first functions in each category, *tellg* or *tellp*, give the index of the current byte pointed to by the marker.

Although there is only one marker, we need to use *tellg* (*g* is for *get*) when we are using an *istream* object and *tellp* (*p* is for *put*) when we are using the *ostream* object.

Printing Location and Value of Characters

The Program shows how we print the location of the marker and the value of the corresponding character in a file that has only one word in it (“Hello!”).

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

int main ()
{
      // Declaration of variables
      char ch;
      int n;
      // Instantiation of stream and opening the file
      ifstream istr;
      istr.open ("sample", ios :: in);
      // Getting characters and thir locations
      n = istr.tellg ();
      while (istr.get(ch))
      {
          cout << n << " " << ch << endl;
          n = istr.tellg ();
      }
      // Closing the file
      istr.close ();
      return 0;
}
```

```
Run:
0 H
1 e
2 l
3 l
4 o
5 !
```

Sequential Versus Random Access Part 4

ios :: beg	ios :: cur	ios :: end

direction (dir) values

Input	Output
int tellg() istream& seekg(int *pos*) istream& seekg(int *off,* ios :: dir)	int tellp(); ostream& seekp(int *pos*) ostream& seekp(int *off,* ios :: dir)

Moving the Marker

We can move the marker to point to the byte we want to read or write next. Movement can be absolute or relative.
If we know the index of the byte we want to move to, we can use the two member functions with one argument: *seekg* (location) and *seekp* (location).

If we want to move the marker to another position relative to the beginning, current, or the end of the buffer, we can use the relative member functions: *seekg* (off, dir) or *seekp* (off, dir) where the offset (*off*) is a positive or negative integer and *dir* is the start position: *cur* for the current position, *beg* (for the beginning of the buffer), and *end* (for the end of the buffer).

Random Access

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

int main ()
{
	std::ofstream outfile;
	outfile.open("test.txt");

	outfile.write("This is an apple", 16);
	long pos = outfile.tellp(); // 16
	outfile.seekp(pos - 7); // 9
	outfile.write(" sam", 4); // This is a sample
	outfile.close();

	ifstream input("test.txt");
	input.seekg(-1, ios::end);
	cout << (char)input.peek() << endl; // e
	input.seekg(0, ios::beg);
	cout << (char)input.peek() << endl; // T
	input.seekg(2, ios::cur);
	cout << (char)input.peek() << endl; // i
	input.seekg(3);
	cout << (char)input.peek() << endl; // s
	input.close();
	return 0;
}
```

```
This is a sample
```

After running the program, the file looks like the following:

```
e
T
i
s
```

Finding the Size of a File

The following shows the contents of the file we have used. Note that when we check the value returned by *tellg*, it gives the index of the character (44) after the period in the file (43), but since the indexes start from 0 not 1, we know that there are exactly 44 characters in the file with indexes 0 to 43.

```
This is the file whose size we want to find.
```

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

int main ()
{
      // Instantiation of stream and connection to the file
      ifstream ifstr;
      ifstr.open ("file4" , ios :: in | ios :: ate);
      // Finding the marker value after the last character
      cout << "File size: " << ifstr.tellg ();
      // Closing the file
      ifstr.close ();
      return 0;
}
```

```
Run:
File size: 44
```

KU KONKUK UNIVERSITY

Binary File Streams

Binary Input / Output

The console streams are not used for binary data because neither the source nor the sink (keyboard or monitor) can handle them.

On the other hand, files connected to a file stream can be used to input and output binary data in which eight bits (a byte) are treated as a byte without being related to any character code (ASCII or Unicode).

This is needed when we want to input or output data that should be interpreted as bytes.

For example, we may write a program that reads a file that has stored audio/video information.

Each picture element (pixel) in a video file can be 8 bits that may be stored as a byte.

We may also need to store objects of user-defined types to be stored as a set of bytes.

In these cases, we can store data in a file as a sequence of bytes.

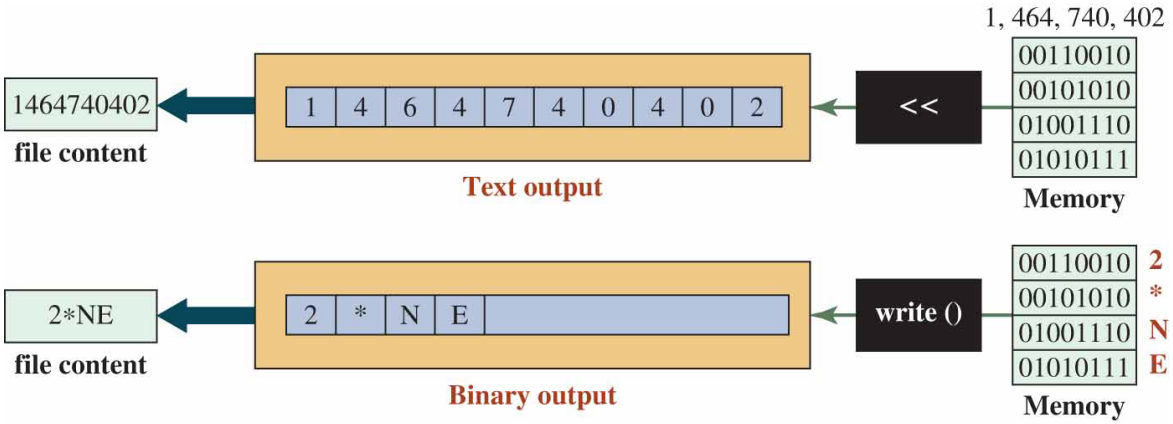
To do so, the file should be opened in the binary mode.

File streams are capable of holding text and binary data.

Binary Input / Output Part 2

In text input/output, values are converted to bytes; in binary input/output the exact pattern of bits is converted to bytes.

We show how a large integer, 1,464,740,402, is stored in memory, in a stream, and in a file both as text and binary .



Note:
The insertion operator changes each digit to a byte.
The function *write*() copies the image of memory to the stream.

The figure shows that in text output, we need to store each digit of the integer as a character (text) in the stream and in the file; in binary output we use only four bytes (the image of data in memory).

Binary Input / Output Part 3

Member Functions

To read and write binary data, the member functions are defined in the *istream* and *ostream* classes, but they are not used there.

The *ifstream* and *ofstream* classes inherit them and use them.

Input	Output
ifstream& read(char* s, int n) ofstream& readsome(char* s, int n)	ifstream& write(char* s, int n)

The *read* function reads up to *n* characters from the stream buffer and fills an array of characters named *s*.

If the number of characters available in the stream is less than the size of the array, the *eofbit* is set.

To prevent this from happening, the second function (*readsome*) was designed that reads characters until no character is available or *n* characters are read.

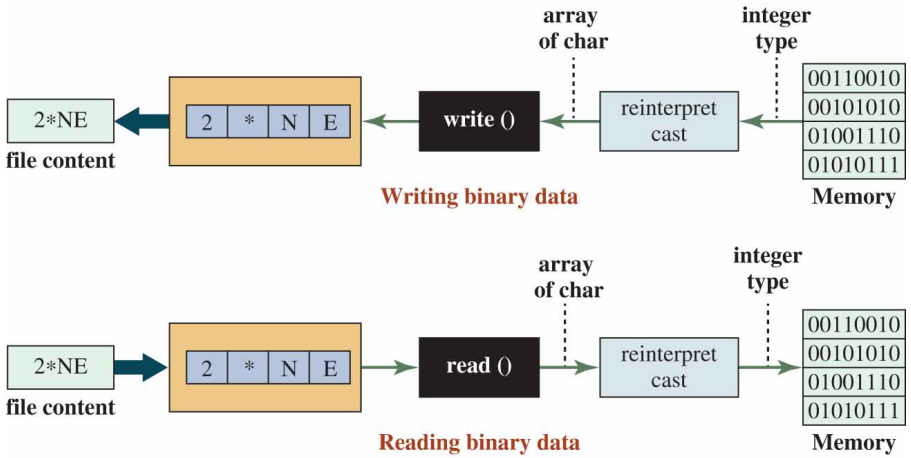
None of the functions add a null character to the array, which means they cannot be used as a C-string.

The *write* function writes exactly *n* characters from an array of characters to the stream buffer.

Binary Input / Output Part 4

Conversion of Fundamental Types

We need a conversion mechanism (*reinterpret_cast*) to change the integer 1,464,740,402 to an array of four characters (2, *, N, E). and vice versa.




```
reinterpret_cast <type2*>(&type1)
```

In the cast operator, *type*1 is a fundamental type (or class type) and *type*2 is a char type.

This means that we need to rewrite the read and write functions as shown below:

```
ifstream& read(reinterpret_cast <char*>(&type), sizeof(type))
ifstream& readsom(reinterpret_cast <char>(&type), sizeof(type))
ofstream& write(reinterpret_cast <char*>(&type), sizeof(type))
```

Writing and Reading Binary Data

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

int main ()
{
	int int1 = 12325;
	double double1 = 45.78;
	// Creating a new file for output and write the two data types
	ofstream strmOut ("Sample", ios :: out | ios :: binary);
	if (!strmOut.is_open())
	{
		cout << "The file Sample cannot be opened for writing!";
		assert (false);
	}
	strmOut.write (reinterpret_cast <char*> (&int1), sizeof (int));
	strmOut.write (reinterpret_cast <char*> (&double1), sizeof (double));
	strmOut.close ();

	int int2;
	double double2;
	// Opening the same file for input and reading the two data types
	ifstream strmIn ("Sample", ios :: in | ios :: binary);
	if (!strmIn.is_open())
	{
		cout << "The file Sample cannot be opened for reading!";
		assert (false);
	}
	strmIn.read (reinterpret_cast <char*> (&int2) , sizeof (int));
	strmIn.read (reinterpret_cast <char*> (&double2) , sizeof (double));
	strmIn.close ();
	// Testing the value of stored data types
	cout << "Value of int2: " << int2 << endl;
	cout << "Value of double2: " << double2 << endl;
	return 0;
} // End main
```

The Program shows how we can write the value of an *int* and a *double* to a file and read them to be sure the values are stored correctly.

Note that we use the *ofstream* object to create a new file and then *ifstream* object to check the file.

```
Run:
Value of int2: 12325
Value of double2: 45.78
```

Binary Input / Output Part 5

Conversion of User-Defined Objects

When reading, we need to have an empty object (created from a default constructor) to be filled by characters from the file.

When we write we need to have a filled object (created from the parameter constructor) to be changed to characters and written to the file as shown below:

```
istream& read(reinterpret_cast <char*>(&object), sizeof(class))
istream& write(reinterpret_cast <char*>(&object), sizeof(class))
```

The Student Class

```
#ifndef STUDEN_H
#define STUDEN_H
#include <iostream>
#include <fstream>
#include <cassert>
#include <iomanip>
#include <cstring>
#include <string>
using namespace std;

class Student
{
      private:
            int stdId;
            char stdName [20];
            double stdGpa;
      public:
            Student (int, const string&, double);
            Student ();
            ~Student ();
            int getId() const;
            string getName() const;
            double getGpa () const;
};
#endif
```

```
#include "student.h"
// Parameter Constructor
Student :: Student (int id, const string& name, double
gpa)
: stdId (id), stdGpa (gpa)
{
      strcpy (stdName, name.c_str() );
      if (stdId < 1 || stdId > 99)
      {
            cout << "Identity is out of range. Program
            aborted.";
            assert (false);
      }
      if (stdGpa < 0.0 || stdGpa > 4.0)
      {
            cout << "The gpa value is out of range.
            Program aborted.";
            assert (false);
      }
}
// Default Constructor
Student :: Student ()
{ }
// Destructor
Student :: ~Student ()
{ }
// Accessor function
int Student :: getId() const
{   return stdId;   }
// Accessor function
string Student :: getName() const
{   return stdName;   }
// Accessor function
double Student :: getGpa () const
{   return stdGpa;   }
```

Application File to Write and Read to a File

```
#include "student.h"

int main ()
{
      // Opening File.dat for binary output
      fstream stdStrm1;
      stdStrm1.open ("File.dat", ios :: binary | ios :: out);
      if (!stdStrm1.is_open())
      {
            cout << "File.dat cannot be opened for writing!";
            assert (false);
      }
      // Instantiation of five objects
      Student std1 (1 , "John", 3.91);
      Student std2 (2 , "Mary", 3.82);
      Student std3 (3 , "Lucie", 4.00);
      Student std4 (4 , "Edward", 3.71);
      Student std5 (5 , "Richard", 3.85);
      // Writing the five objects to the binary file and close it
      stdStrm1.write(reinterpret_cast <char*> (&std1), sizeof (Student));
      stdStrm1.write(reinterpret_cast <char*> (&std2), sizeof (Student));
      stdStrm1.write(reinterpret_cast <char*> (&std3), sizeof (Student));
      stdStrm1.write(reinterpret_cast <char*> (&std4), sizeof (Student));
      stdStrm1.write(reinterpret_cast <char*> (&std5), sizeof (Student));
      stdStrm1.close ();
      // Opening File.dat for input
      fstream stdStrm2;
      stdStrm2.open ("File.dat", ios :: binary | ios :: in);
      if (!stdStrm2.is_open())
      {
            cout << "File.dat cannot be opened for
            reading!";
            assert (false);
      }
      // Reading Student objects, display them, and close
      the File.data
      cout << left << setw (4) << "ID" << " ";
      cout << setw(15) << left << "Name" << " ";
      cout << setw (4) << "GPA" << endl;
      Student std;
      for (int i = 0; i < 5; i++)
      {
            stdStrm2.read(reinterpret_cast <char*> (&std),
            sizeof (Student));
            cout << setw (4) << std.getId() << " ";
            cout << setw(15) << left << std.getName() <<" ";
            cout << fixed << setw (4) << setprecision (2)
            << std.getGpa ();
            cout << endl;
      }
      stdStrm2.close();
      return 0;
}
```

Run:

ID	Name	GPA
1	John	3.91
2	Mary	3.82
3	Lucie	4.00
4	Edward	3.71
5	Richard	3.85

Binary Input / Output Part 6

Random Access

We can also read or write to binary files randomly.

In other words, instead of reading the record of all students, we can read the record of a specific student using her identity. However, this involves two precautions.

First, we need to be sure that all of the objects stored in the file are of the same size to be able to use the *seekg*() and *seekp*() functions.

In the case of the object with only data members of fundamental data types, the size of each object is the sum of the size of its data members.

In the case of the Student class, the size of array representing the name is also fixed.

```
class Student
{
      private:
            int stdId;
            char stdName [20];
            double stdGpa;
      public:
            Student (int, const string&, double);
            Student ();
            ~Student ();
            int getId() const;
            string getName() const;
            double getGpa () const;
};
#endif
```

Second, we need to use some precautions to find the location of the object given some information about the object.

For example, we can use the identity of a student in the previous example to find the location of Student object using:

```
seekg((id - 1) * sizeof(Student))
```

End of Class

Thank you

E-mail: youngcha@konkuk.ac.kr