Object-oriented Programming

Input / Output Streams Part 1

YoungWoon Cha

Computer Science and Engineering

KU KONKUK UNIVERSITY

Streams

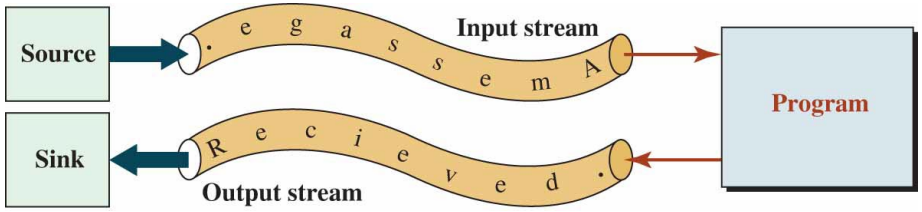
INTRODUCTION

When we run a program, data needs to be stored in memory.

The data stored in memory for processing comes from an external source and goes to an external sink.

A source can be a keyboard, a file, or a string.

A sink can be a monitor, a file, or a string.



Although, we can receive data from a source and we can send data to a sink in text or binary format, the data are organized as a sequence of bytes (8-bit chunks).

A source can only send a sequence of bytes;
a sink can receive only a sequence of bytes.

A source and a sink of data cannot be directly connected to a program.

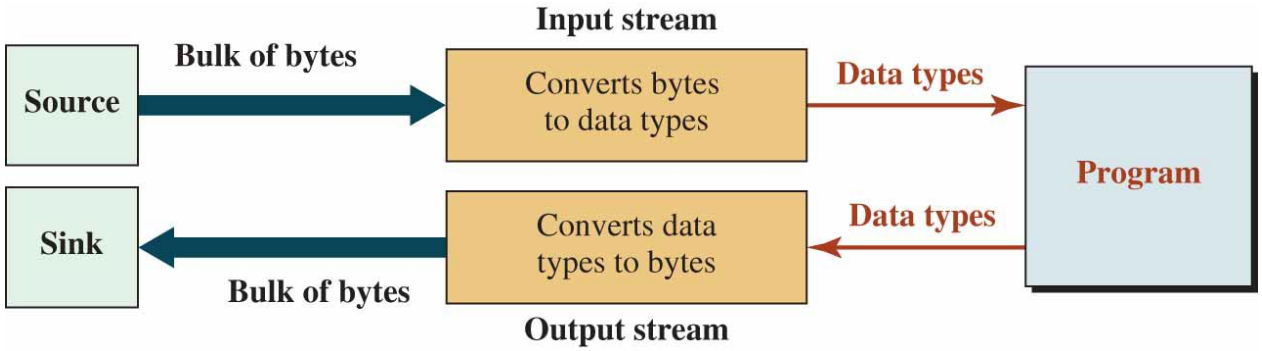
We need a mediator to stand between them to interpret data when reading or writing.

Streams

Role of Streams

The C++ standard created *input/output streams* to change bytes of data that are received from a source to the data type the program needs and to change the data types that the program sends to a sink into bytes that the sink can store.

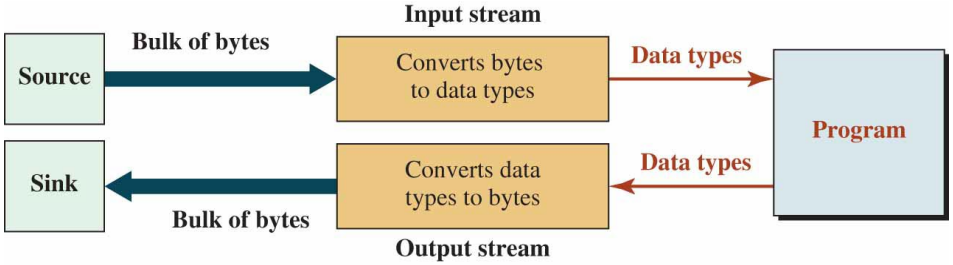
In other words, the main job of streams can be thought of as converting bytes to data types and data types to bytes.



Data Representation Part 1

Text versus Binary Input/Output

Let us see how data are stored in memory, a stream, and a sources/sink locations.



Storage of Data in Memory

The memory of a computer is a collection of bytes (8-bit) chunks, but we do not store data as individual bytes, we store data as a set of contiguous bytes.

For example, each data object of type *int* occupies four bytes of memory no matter what the value of the variable is.

In other words, a small integer of value 23 and a large integer of value 4,294,967,294 both occupy 4 bytes of memory or 32 bits of data.

Data in the memory are stored as binary.

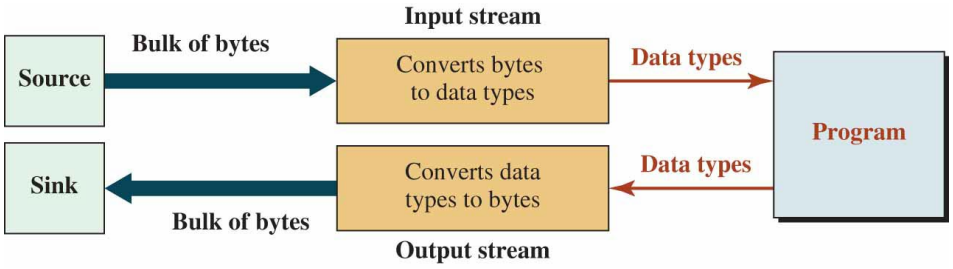
Data Representation Part 2

Storage of Data in Sources or Sinks

Sources and sinks of data also store data in a chunk of bytes, but data can be either binary or text.

For example, the integer 23 can be stored as four bytes (binary) or it can be stored as two characters '2' and '3' (text).

The integer 4294967294 can be stored as four bytes in binary, but 10 bytes in text.

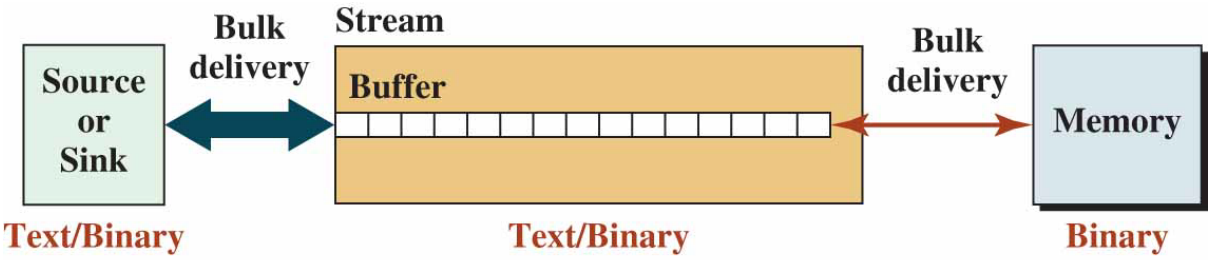


- **The keyboard as a source of data can accept data only as text. Similarly, the monitor as a sink of data shows data only as a set of characters.**
- **Files as sources or sinks of data can handle both text and binary input/output. If we want to store an object in a file, the format should be binary.**
- **Strings as sources or sinks of data must be input/output as text; a string is a collection of characters.**

Data Representation Part 3

Storage of Data in Buffer of Streams

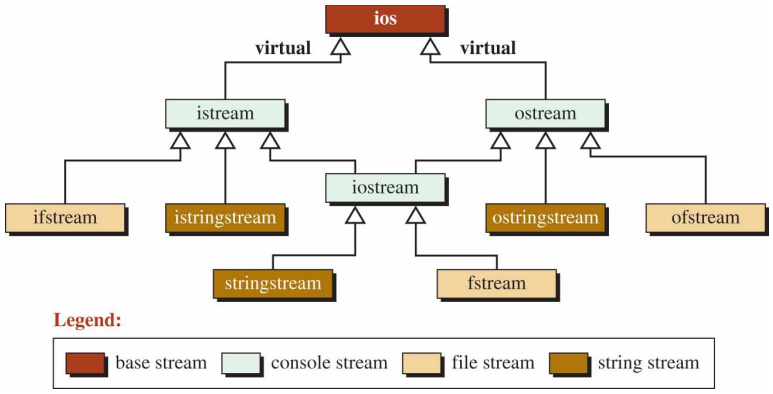
Streams deliver data to a sink or receive data from a source in bulk.



Note:
The stream is responsible for changing data from text to binary and vice versa.

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).

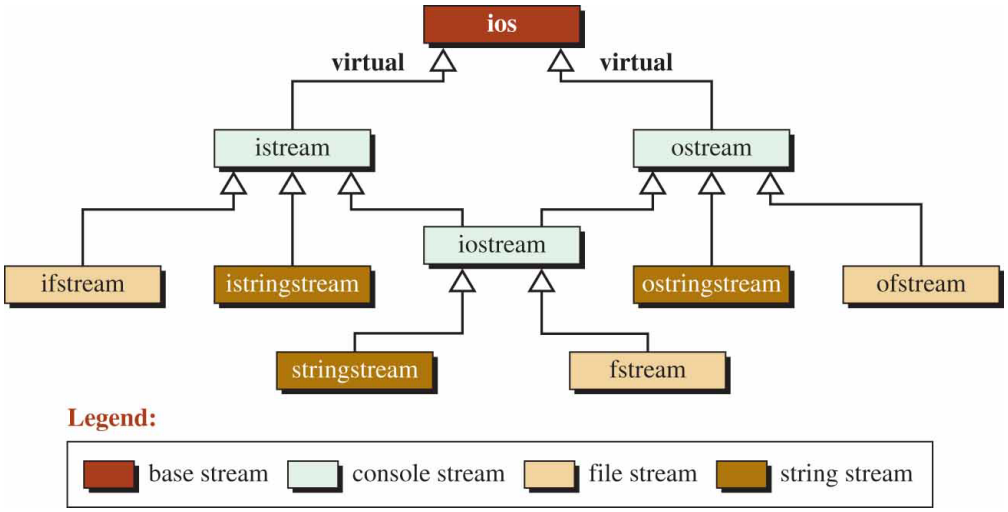
We refer to *ifstream*, *ofstream*, and *fstream* as file streams (they are used to connect our program to files).

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

KU KONKUK UNIVERSITY

Console Streams

CONSOLE STREAMS



We first discuss the console streams: *istream*, *ostream*, and *iostream*.

We can use the first two, but the *iostream* class cannot be instantiated.

It is defined as a superclass for other classes in the hierarchy.

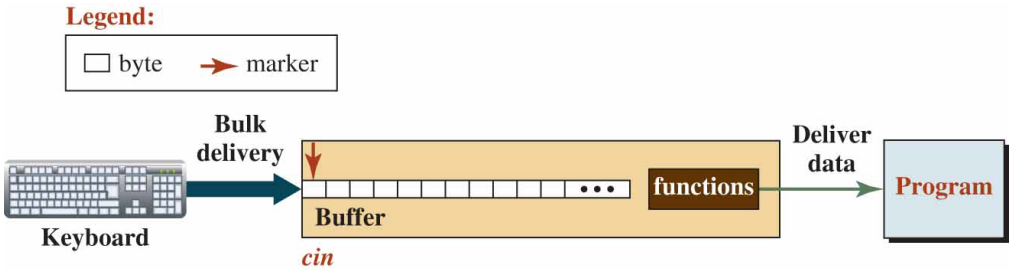
Console Objects Part 1

We first discuss the objects created from the two console stream classes.

The istream Object: cin

The *istream* class defines a class that allows us to read data from the keyboard into our program.

We cannot instantiate this class. However, the system has created an object of this class, named *cin*, which is stored in the <iostream> header file.



When our program goes out of scope, the *cin* object is destroyed and it is automatically disconnected from the keyboard.

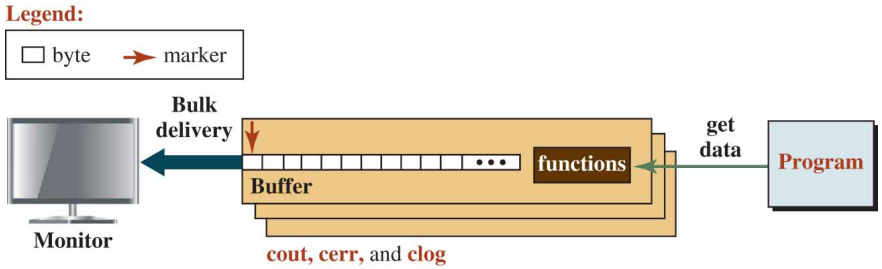
The system has created an object from istream class, named cin, and stored it in the <iostream> header.

Console Objects Part 2

The ostream Objects: cout, cerr, and clog

The *ostream* class defines a class that allows us to write data from our program to the monitor.

We cannot instantiate this class. However, the system has created three objects of this class (named *cout*, *cerr*, and *clog*) which are stored in the <iostream> header file.



Note:
There are three simulatneous objects.
The marker is at the beginning of the buffer when the monitor is connected to the streams.
After a write operation, the marker moves to the right.

When our program goes out of scope, these objects are automatically destroyed and are automatically disconnected from the monitor.

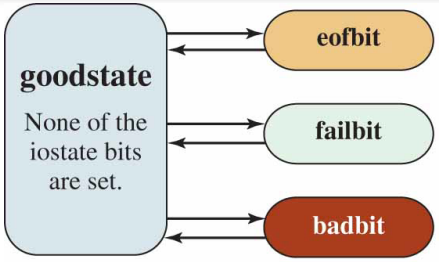
The *cout* object is tied to the *cin* object, which means every time we input data through the *cin* object, the *cout* object is flushed (emptied).

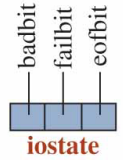
Both *cerr* and *clog* sends errors to the console. The difference is that the *cerr* flushes its contents immediately after each operation; the *clog* collects the error messages and is flushed whenever the program terminates or when it is explicitly flushed.

Stream State Part 1

State Data Member

The *ios* stream defines a type called *iostate* (input/output state).





Note:
A stream can contiue to read or write only in the good state.

```
enum _Iostate { // constants for stream states
    _Statmask = 0x17
};

static constexpr _Iostate goodbit = static_cast<_Iostate>(0x0);
static constexpr _Iostate eofbit  = static_cast<_Iostate>(0x1);
static constexpr _Iostate failbit = static_cast<_Iostate>(0x2);
static constexpr _Iostate badbit  = static_cast<_Iostate>(0x4);
```

Constants	Input Stream	Output Stream
ios :: eofbit	No more characters to extract.	Not applicable.
ios :: failbit	An invalid read operation.	An invalid write operation.
ios :: badbit	Stream integrity is lost.	Stream integrity is lost.
ios :: goodbit	Everything is fine.	Everything is fine.

The *eofbit* is set when we try to extract a byte that does not exist in the stream buffer.

The *failbit* is set when we try to extract a byte that does not correspond to what we want.

The *badbit* is also set when the integrity of the stream is lost such as a memory shortage, a conversion problem, an exception is thrown, and so on.

When none of the three bits are set, the stream is in a good state and can be used.

Stream State Part 2

State-Related Member Functions

The *ios* class provides member functions to check the condition of the state.

Functions	Return values
bool eof ()	true if *eofbit* is set; false otherwise.
bool fail ()	true if *failbit* or *badbit* is set; false otherwise.
bool bad ()	true if *badbit* is set; false otherwise.
bool good ()	true if the *stream* is in good condition; false otherwise.
clear ();	It clears all three bits (sets to zero).
operator void* ();	It returns a pointer; interpreted true if not null.
const bool operator! ();	It returns true if badbit or failbit is set.

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

int main ()
{
      int n;
      cout << "Enter a line of integers and eof at the end: " << endl;
      while (cin >> n)
      {
            cout << n * 2 << " ";
      }
      return 0;
}
```

```
Run:
Enter a line of integers and eof at the end:
14 24 11 78 19 32 ^Z
28 48 22 156 38 64
```

The expression *while* (*cin* >> *n*), a conversion must occur to change the *cin* object to a Boolean value.

This is done by calling *operator void** (), which returns a pointer (*true*) if the previous operation is successful and a null pointer (*false*) if it is not successful.

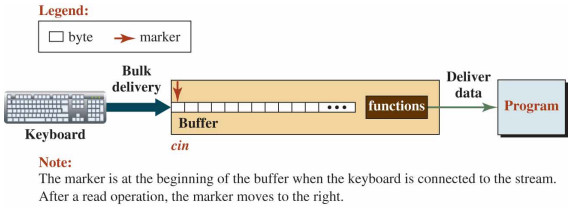
This means that the loop continues until one of the three bits are set.

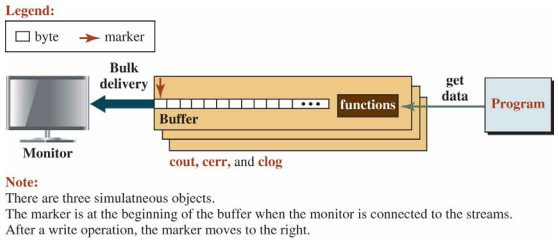
When we type ^Z on the keyboard, it means that we terminate inputting data into the *cin* buffer and the *eofbit* is set.

Input/Output

Since the objects of the *istream* and *ostream* classes have already been instantiated and connected to the console, our only task is to use the member functions to read data from the buffer or write data to the buffer.

The buffers store only a set of 8-bit bytes interpreted individually as characters.





Character Member Functions

The *istream* and *ostream* classes provide member functions to read a single character from the keyboard and to write a single character to the monitor.

The istream class	The ostream class
int get() istream& get (char& c)	ostream& put (char& c);

The *istream* class provides two versions of the *get* function. The first returns an integer (the ASCII) value of the character in the buffer which is stored in memory as an integer (converted to four bytes).

The second function reads a character which is stored in its parameter.

The *ostream* object, provides only one put function that writes a copy of a character defined in its parameter to the buffer.

A Program to Test the Get and Put Functions

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

int main ()
{
      int x;
      cout << "Enter five characters (no spaces): ";
      for (int i = 0; i < 5; i++)
      {
           x = cin.get ();
           cout << x << " ";
      }
      return 0;
}
```

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

int main ()
{
      char c;
      cout << "Enter a multi-line text and EOF as the last line."
      << endl;
      char pre = '\n';
      while (cin.get(c ))
      {
           if (pre == ' ' || pre == '\n' )
           {
                cout.put (toupper (c ));
           }
           else
           {
                cout.put (c );
           }
           pre = c;
      }
      return 0;
}
```

```
Run:
Enter five characters (no spaces): ABCDE
65 66 67 68 69
```

```
Run:
Enter a multi-line text and EOF as the last line.
This is the text that we want to capitalize
This Is The Text That We Want To Capitalize
each word.
Each Word.
^Z
```

Input/Output Part 2

Reading A C-String

The *istream* class provides member functions to read a set of characters and create a C-string out of them.

The first function, *get*, reads *n* -1 characters from the keyboard and stores them as a C-string (a *null* character is added to the end of the array).

The second function, *getline*, reads n - 1 characters or a set of characters terminated by a delimiter, which is defaulted to '\n', whichever comes first.

The *istream* class	The *ostream* class
istream& get(char* s, int n) istream& getline(char* s, int n, char d = '\n')	

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

int main ()
{
      char str2 [80];
      cin.getline (str2, 80, '\n');
      cout << str2;
      return 0;
}
```

```
Run:
This is a line to make a string out of it.
This is a line to make a string out of it.
```

Input/Output Part 3

Input/Output of Fundamental Data Types

Inputting and outputting of fundamental data types is done by two overloaded operators called the *insertion* operator and the *extraction* operator.

Characters → A fundamental data type

A fundamental data type → Characters

The following shows the general declaration of these two operators in which the type can be *bool*, *char*, *short* (signed and unsigend), *int* (signed and unsigend), *long* (signed and unsigend), *float*, *double*, *long double* or *void.**

```
istream& operator >> (type&  x);   // Extracting next data item
ostream& operator << (type&  x);   // Inserting next data item
```

Input/Output Part 4

```
istream& operator >> (type&  x);   // Extracting next data item
ostream& operator << (type&  x);   // Inserting next data item
```

Extracting Fundamental Data Type

The extraction operator for the fundamental data types uses the following four rules when it encounters a statement such as *cin >> variable*.

The operator reads the characters one by one until it finds a character that does not belong to the syntax of the corresponding type:

a. **If it is extracting a Boolean value, it needs to see a value that can be interpreted as 0 or 1.**

b. **If it is extracting a character value, it reads the next character in the input stream (including a whitespace character) and stores it in the corresponding variable.**

c. **If it is extracting an integer value, it reads digits until it encounters a non-digit character. It makes an integer value of the extracted characters. The un-extracted characters remain in the stream.**

d. **If it is extracting a floating-point value, it reads digits with only one decimal point between digits. If it encounters a second decimal point or a non-digit character, it stops and makes the floating-point value.**

Input/Output Part 5

istream& operator >> (type& x);	// Extracting next data item
ostream& operator << (type& x);	// Inserting next data item

Inserting Fundamental Data Type

Fundamental data types can be inserted into streams using the insertion operator (<<). The value is considered as a set of characters and inserted into the stream.

For example, the integer 124 is inserted as three characters (1, 2, and 4) into the stream.

Input/Output of C++ Strings

The extractor and the inserter operator are overloaded for the string class; they can be used as they have defined.

However, the input stops when the first whitespace is encounter.

If we want to read the whitespace characters (the whole line), we need to use the *getline* function (a global function) defined in the <string> header. It has the following signature.

```
istream& getline(istream& in, string& str);
```

End of Class

Thank you

E-mail: youngcha@konkuk.ac.kr