Advanced Object Oriented Programming

Advanced I/O Concepts

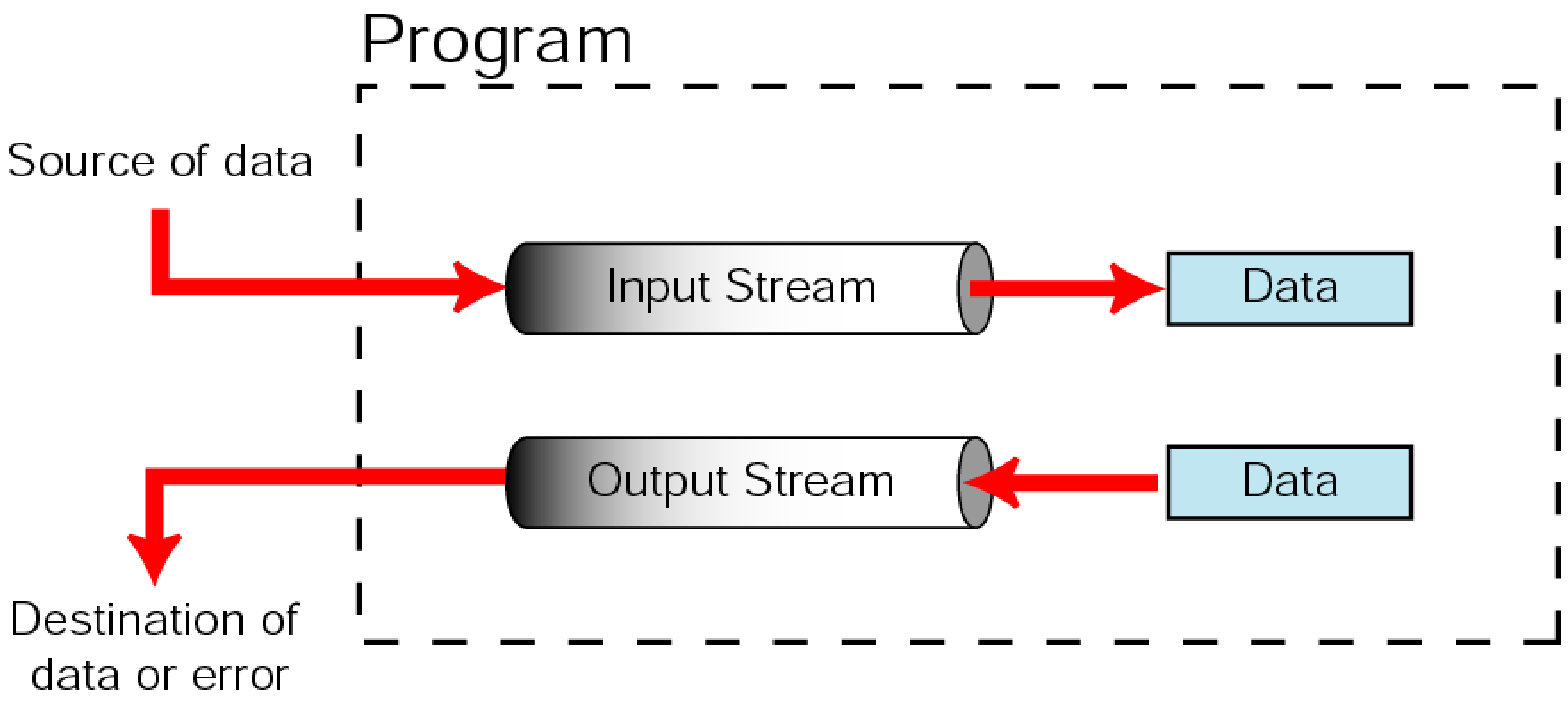
Seokhee Jeon
Department of Computer Engineering
Kyung Hee University
jeon@khu.ac.kr

Streams

- Diversity of input sources or output destinations
  - disk, CD/DVD, tape, printer, ...
- Should a programmer know the operation of each data source or destination?
- A stream is an abstract representation of an input data source or output data destination
- With the stream, the details of reading and writing data to and from a source or destination are left to the operating system

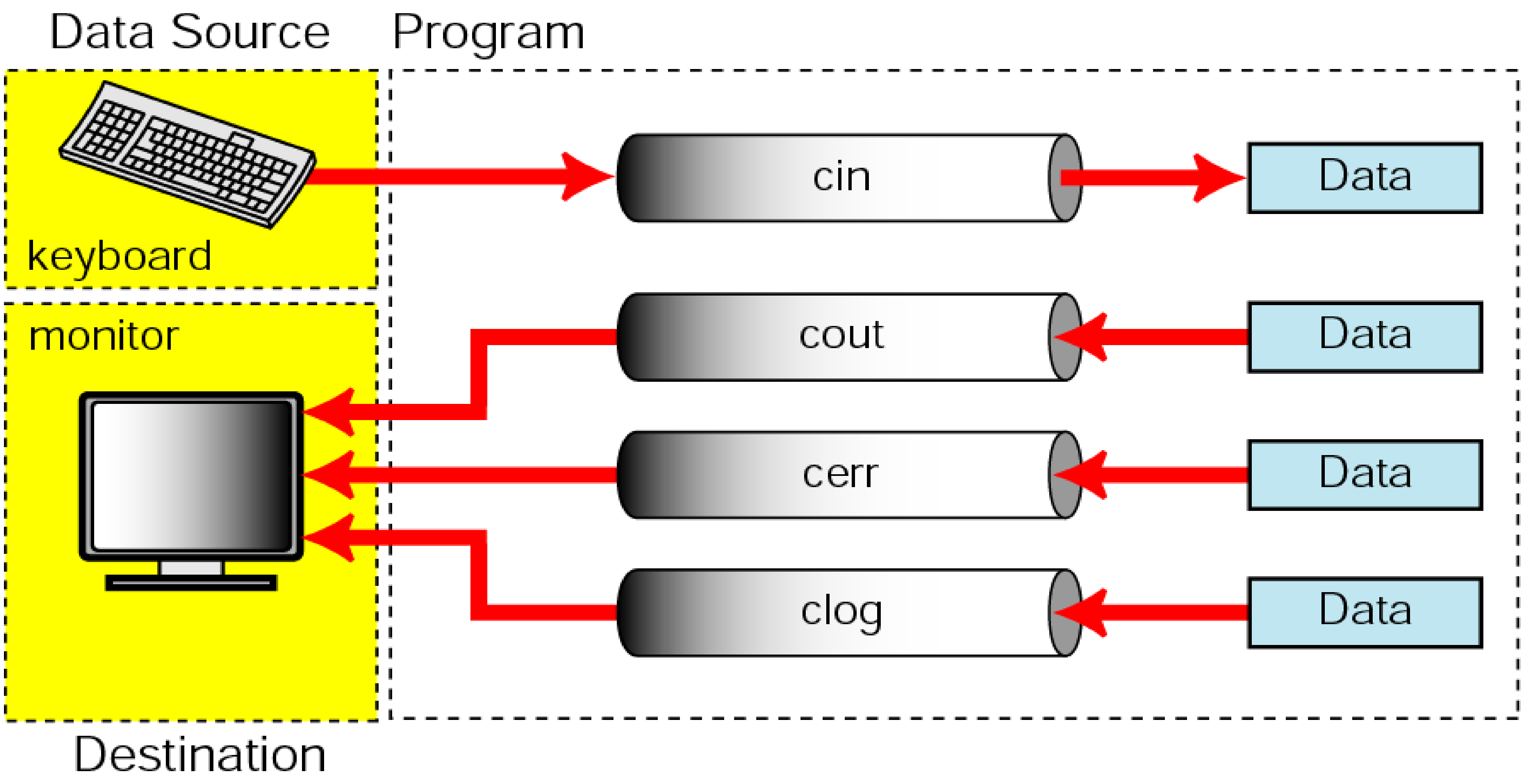
Concepts of Stream

- a sequence of elements in time
- Only one stream element, the current one, is available at a time



Standard Streams

- Standard streams are created, connected, and disconnected automatically



File Streams

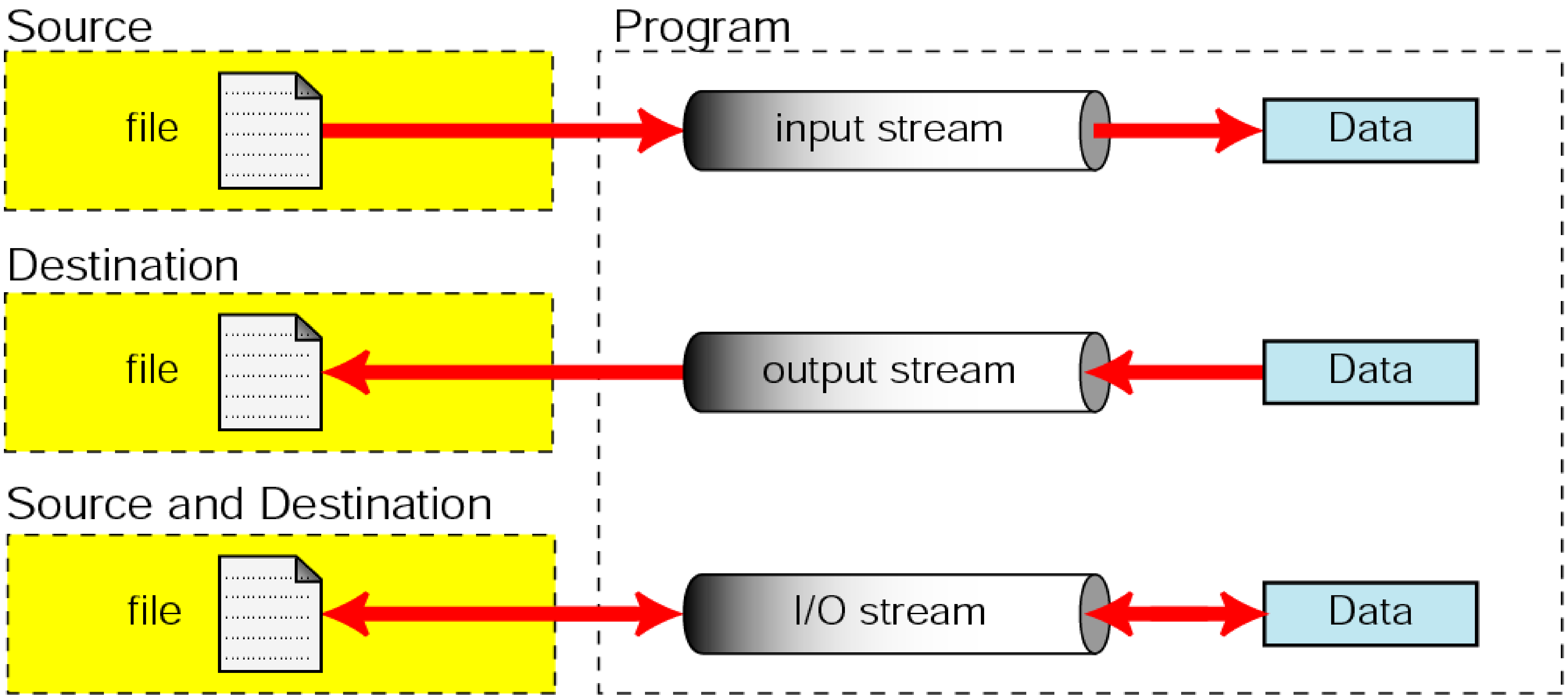
- Besides standard input sources or output destinations, programmers should create their own streams for reading from or writing to files

- Three types of file streams
  - input file stream: *ifstream*
  - output file stream: *ofstream*
  - input/output file stream: *iofstream*

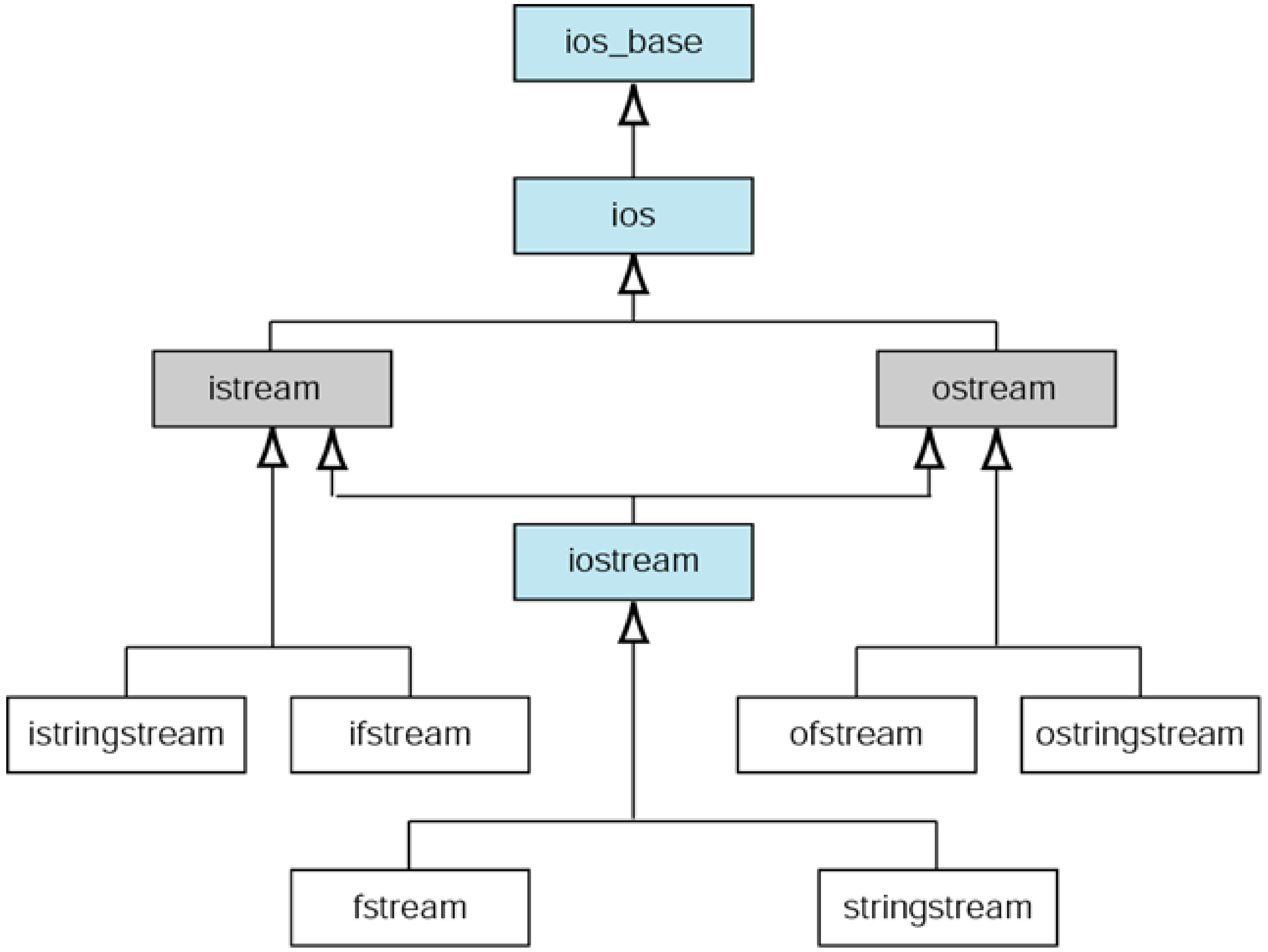
경희대학교 Kyung Hee University

File Streams

- After we create a file stream, it must be connected to the physical device
  - This id done by the *open* function



I/O Class hierarchy



File Streams

- *ios_base* keeps track of the stream state and has function for formatting
- *ios* tests and sets the stream state
- *istream* allows sequential or random input access to disk and standard input files
- *ostream* allows sequential or random output access to disk and standard output files
- *iostream* allows sequential or random input/output access to disk and standard input files
- *ifstream* defines the functions that read from a file
- *ofstream* defines the functions that write to a file
- *fstream* defines the functions that read and write to a file
- *istringstream* defines the functions that read from a string
- *ostringstream* defines the functions that write to a string
- *stringstream* defines the functions that read and write to a string

File States

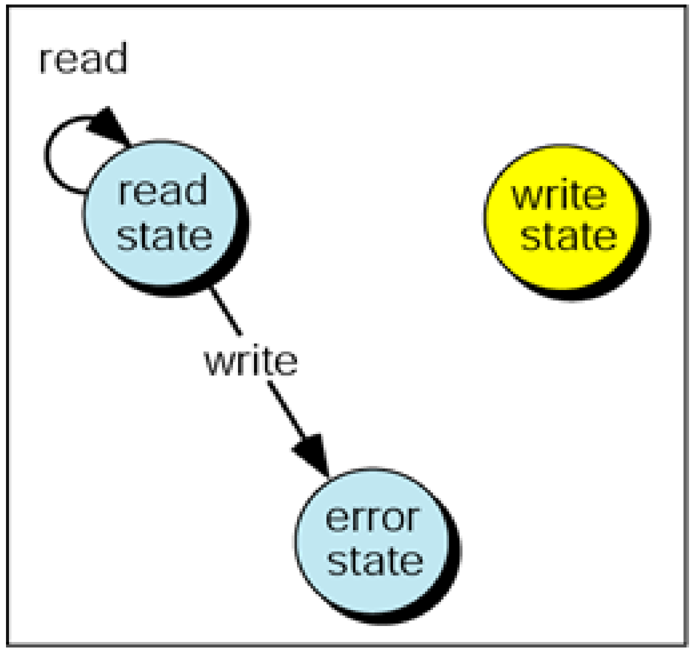
- An opened file is in only of the following three states at a time
  - read state: We can read from the file
  - write state: We can write to the file
  - error state: The result of an error. We cannot read or write.

- When opening files, we decide the possible states
  - If we open a file for reading, only two states, read and error, are possible
  - If we open a file for updating (reading + writing), all three states are possible

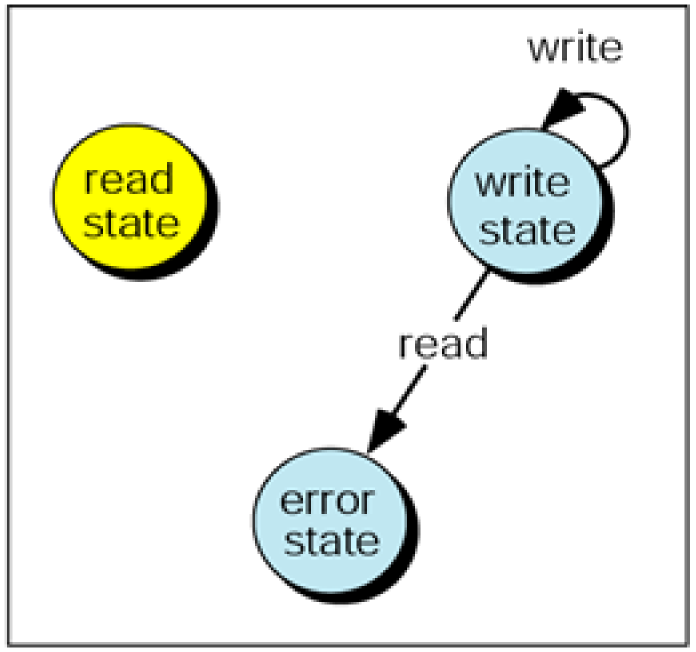
Transitions between States

- Between read and write states
  - Possible only when the file is opened for updating
  - Use positioning functions
- From a read or write state to an error state
  - changed when the previous operation incurs an error
  - logical error vs. physical error
- From an error state back to the previous normal state
  - use the *clear* function

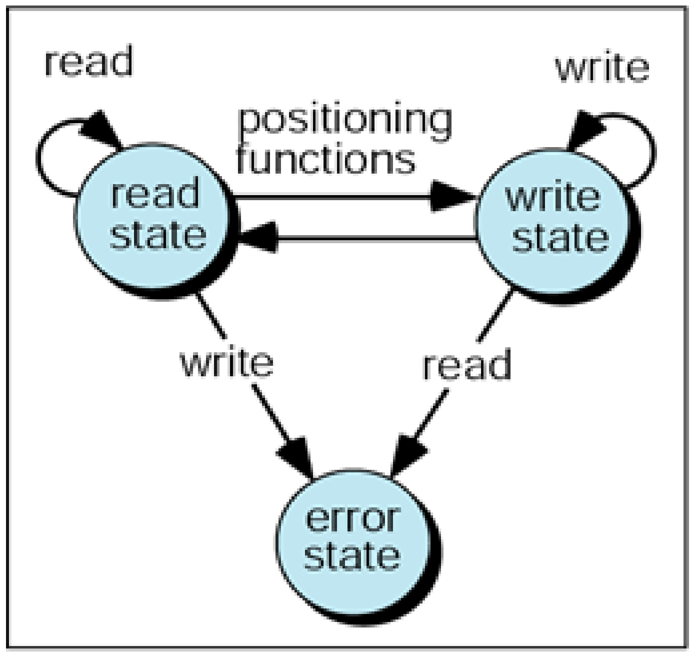
Transitions among File States



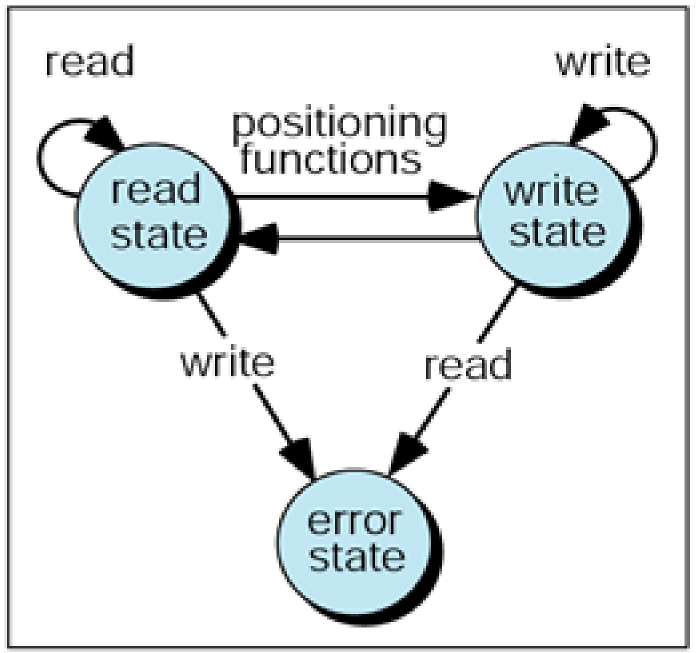
read state



write state



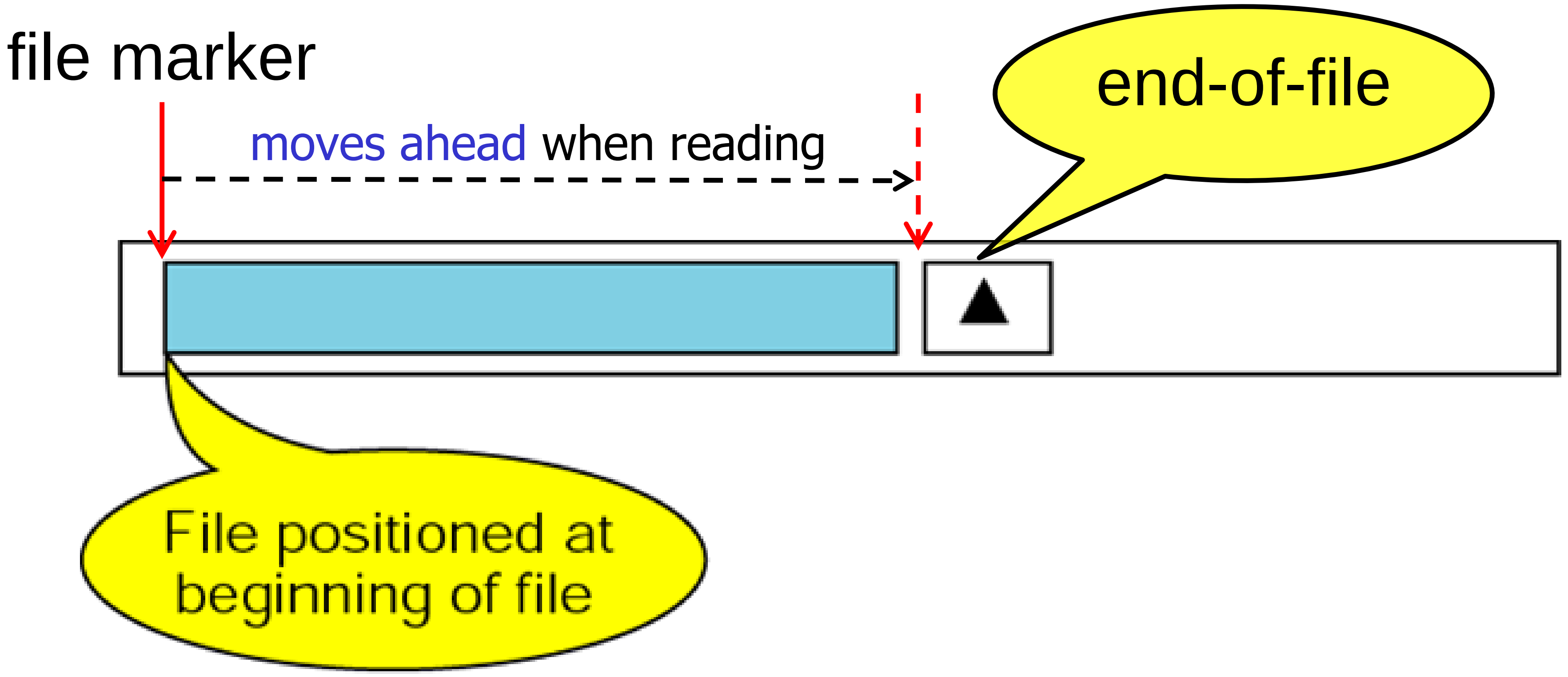
update/read state



update/write state

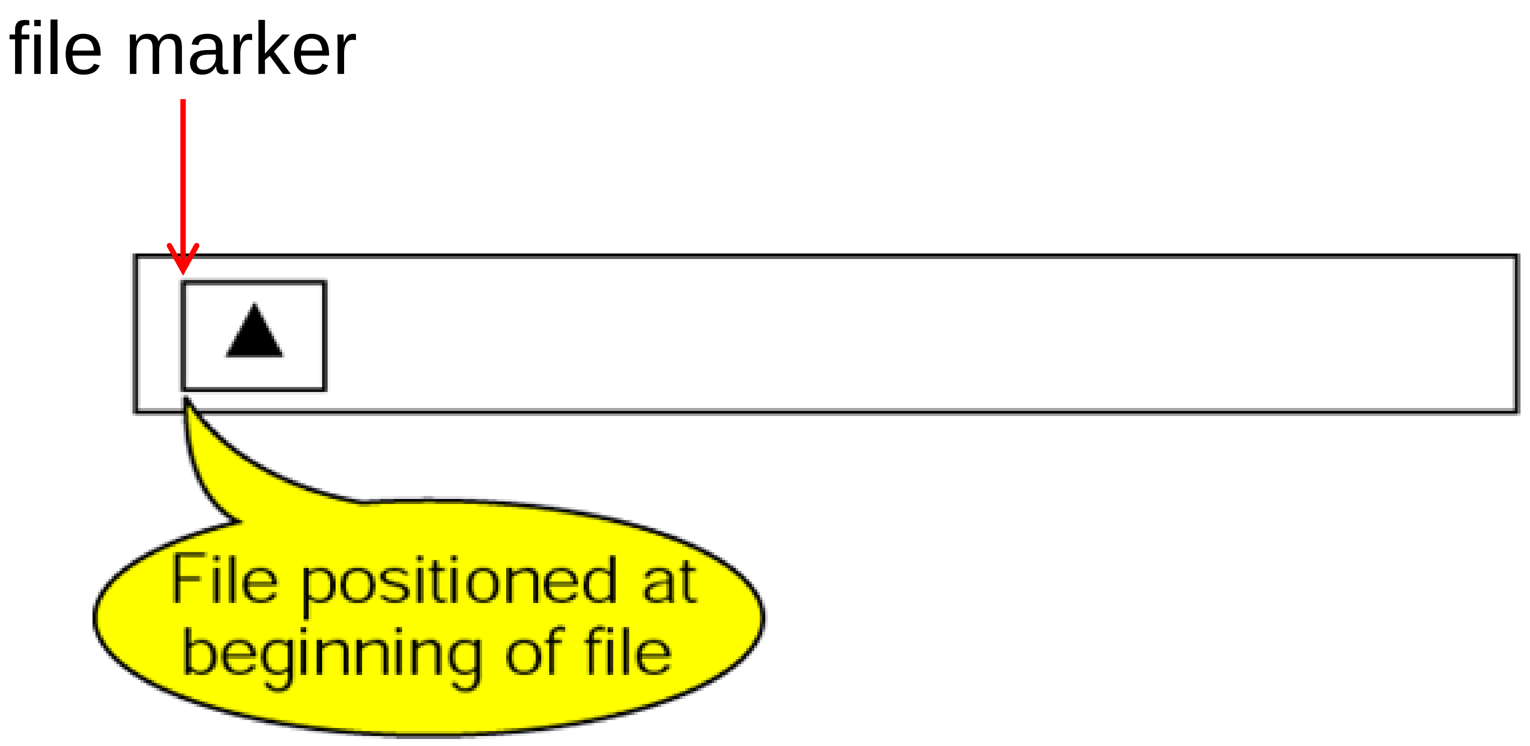
Open file in read state

```
ifstream fsIn;
…
fsIn.open(“file1”);
```



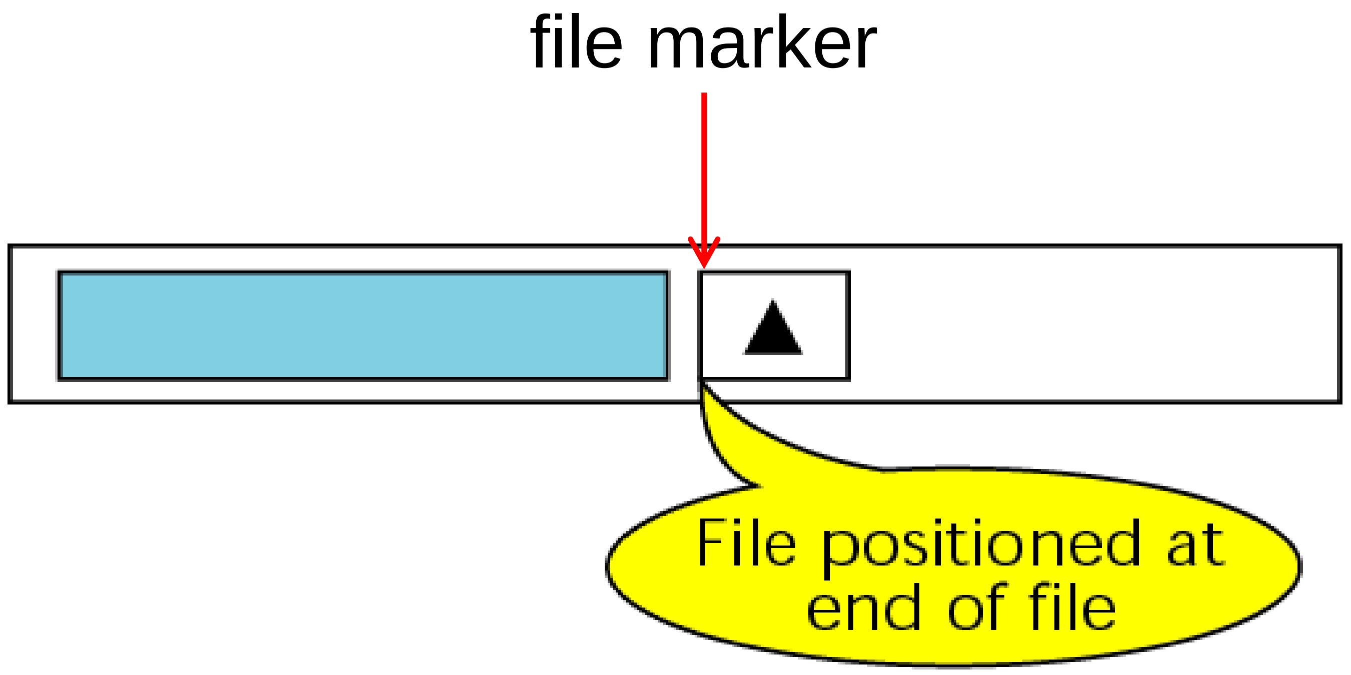
Open file in write state: create new file

```
ofstream fsOut;
...
fsOut.open("file1");
```



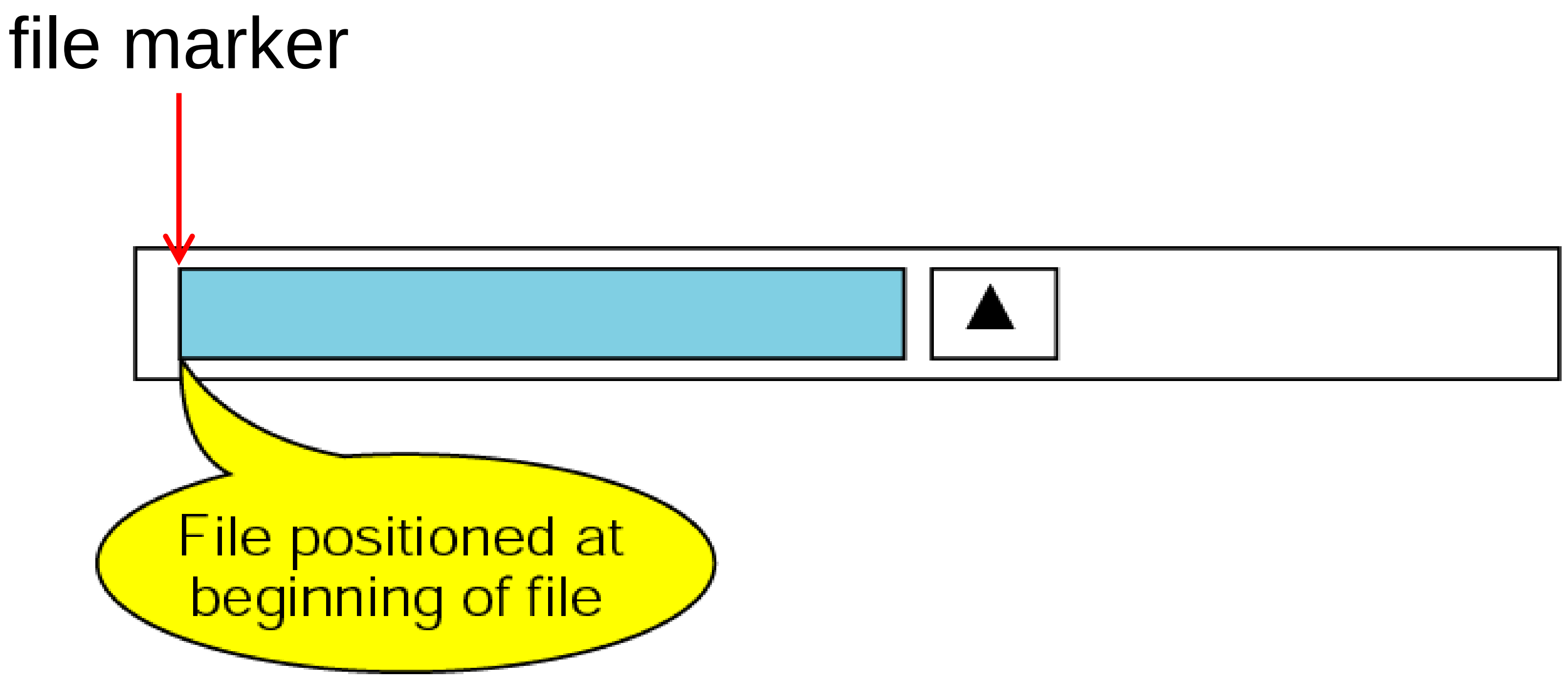
Open file in write state: append to a file

```
ofstream fsAppend;
...
fsAppend.open (“file1”, ios::out | ios::app);
```



Open File for Updating

```
fstream fsUpdate;
…
fsUpdate.open (“file1”);
```



Input/Output system flags

- defines the state of a file

```
file_stream.open (file_id, ios_flags)
```

For specifying more than one flag,
they need to be bitwise OR'd

ios::in	Input
ios::out	Output
ios::app	Append
ios::ate	At the End of the file
ios::trunc	Truncate the current contents
ios::binary	Binary mode

Testing If a File is Open

- To verify that a file is currently open and connected to a stream, we use the *is_open* function

```
if ( fs.is_open() )
{
    …
}
```

Open and Overloaded Constructors

- All stream classes have an overloaded constructor to open a file when the stream is instantiated

```
// Traditional Open
ifstream fsIn;
...
fsIn.open (“file1”);
```

```
// Constructor Open
ifstream fsIn (“file1”);
```

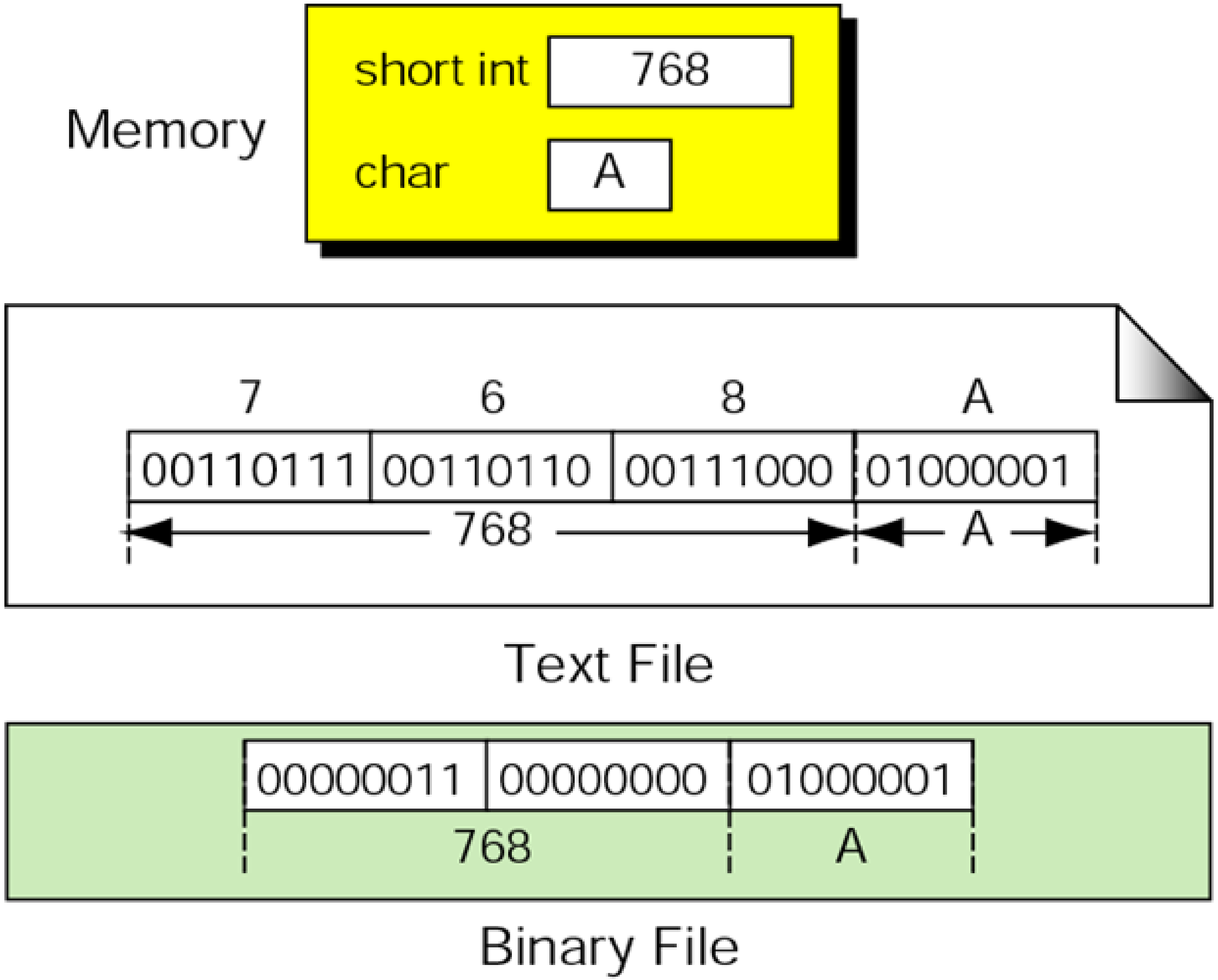
Text Files

- Contains human-readable graphic characters encoded with the ASCII code
- They should be converted to internal formats when read into the memory
  - E.g., integral data must be converted to the appropriate binary number
- Two special characters
  - end-of-line
  - end-of-file

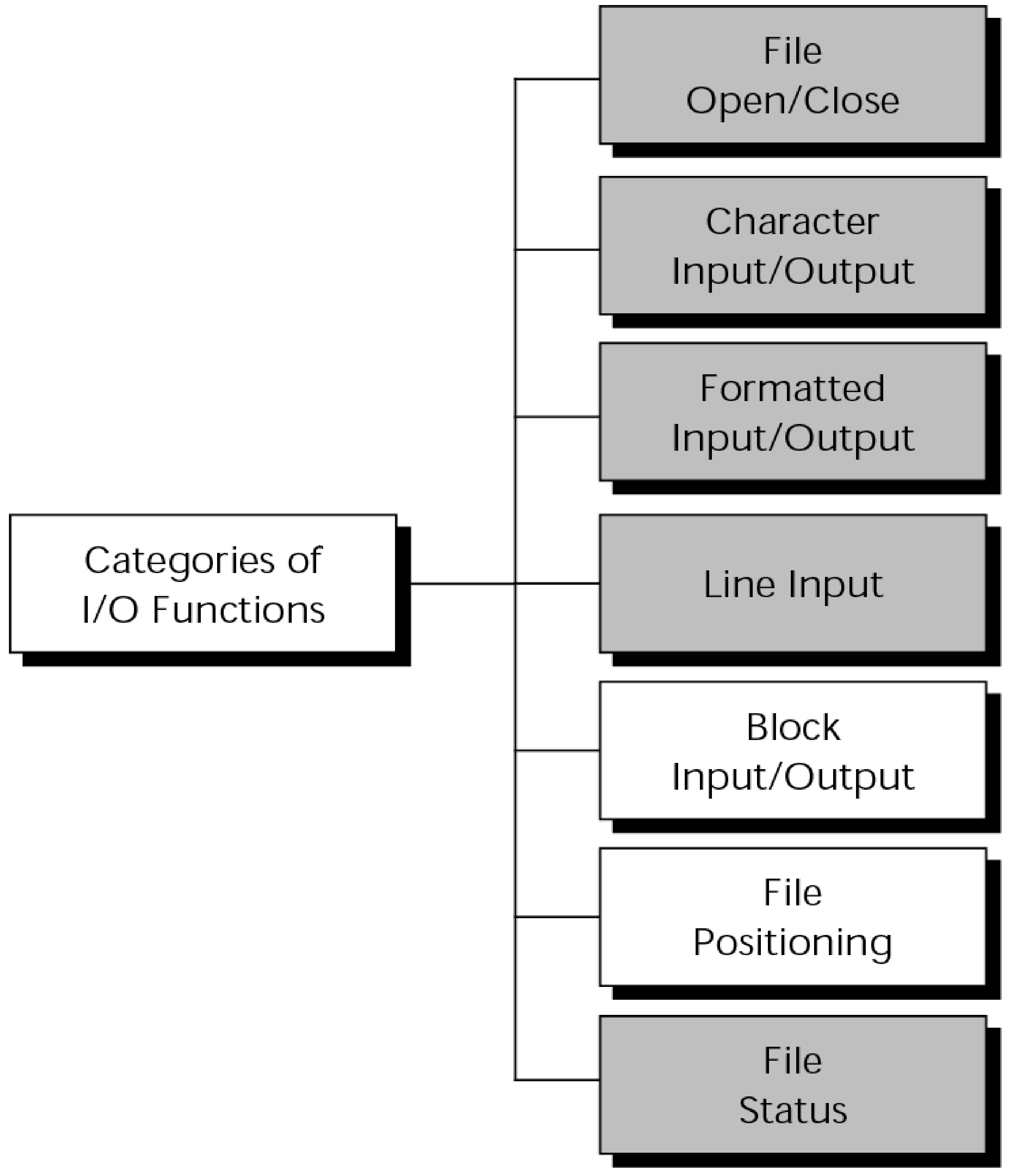
Binary Files

- Data are stored in the same format as they are stored in memory
  - An *int* in C++ is stored in its binary format, usually 4 bytes in a PC
  - A character is stored in its ASCII format, usually 1 byte
- There are no lines or a new line characters
- There is an end-of-file marker

Binary and text files



Types of standard input/output functions



(Chapter 7)

(Chapter 7)

(Chapter 7)

(Chapter 14)

(Chapter 7)

Block Input/Output Functions

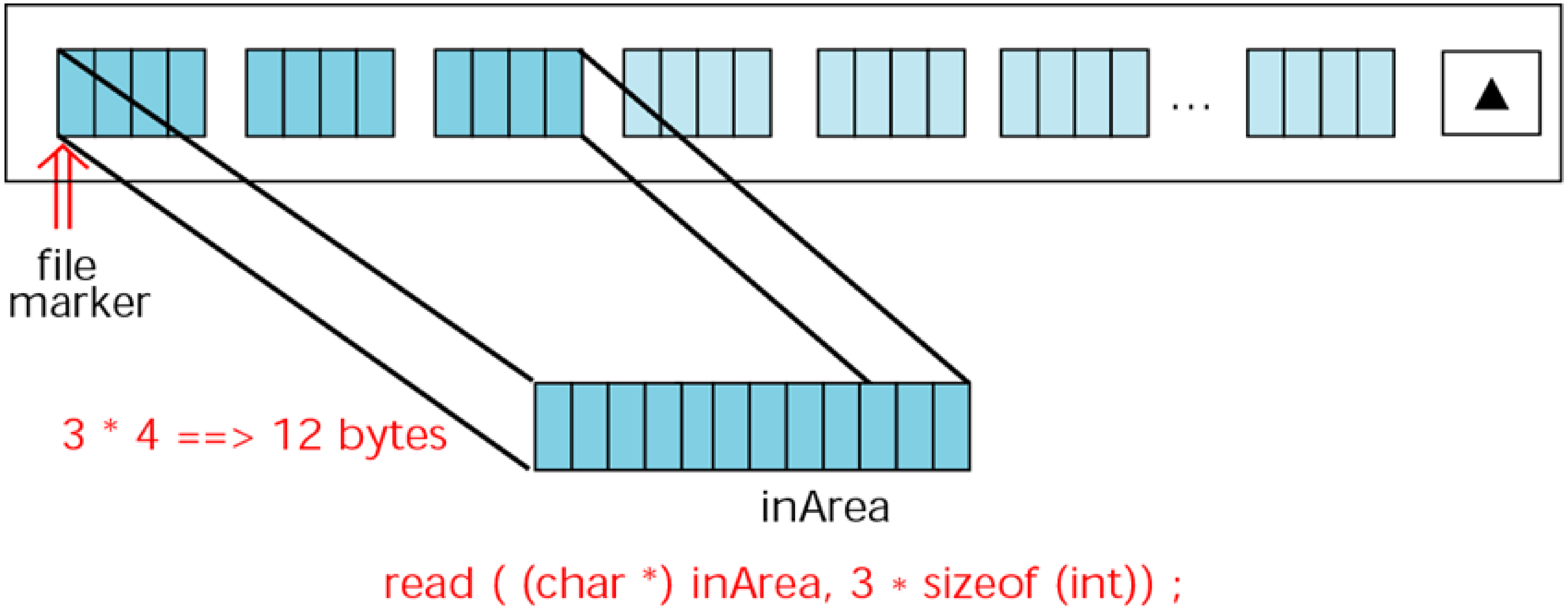
- Used to read and write data to binary files
- Remind that there are no format conversions when the data are transferred between binary files and memory
- The block read function is file read (*read*)
- The block write function is file write (*write*)

read operation

- Reads a specified number of bytes from a binary file and places them into memory at the specified location

```
istream& read ( char* buffer, int size );
```

buffer:	a pointer to the input area in memory
size:	specify how many bytes are to be read



Program: Read file of integers

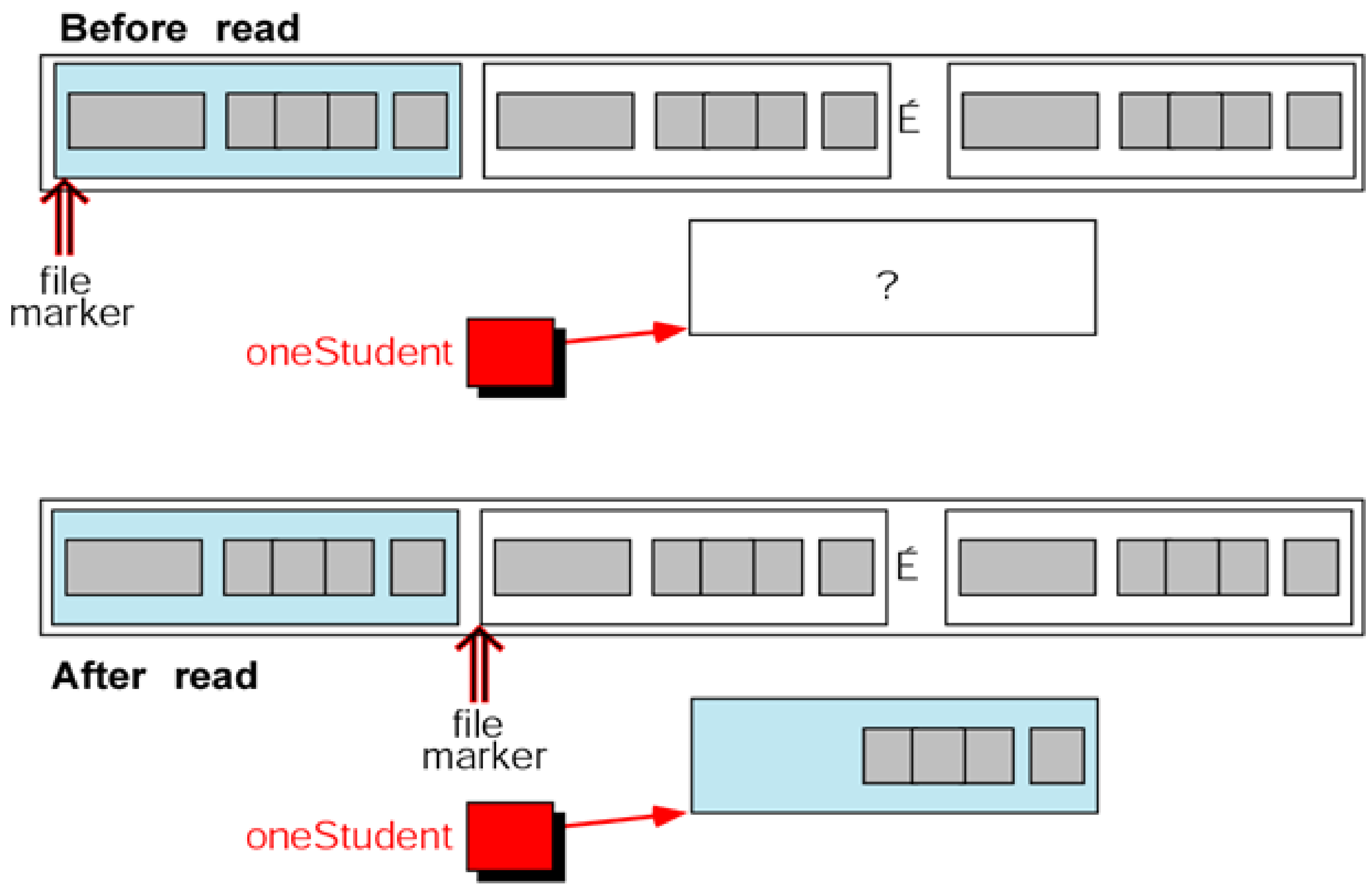
```
#include <iostream>
#include <fstream>
using namespace std;

int main ()
{
    ifstream fsIn;
    fsIn.open("P16-01.dat", ios::binary | ios::in);
    if (!fsIn) {
        cerr << "Input file open failure□a□n";
        exit (100);
    } // open error

    int intAry[3];
    while (fsIn.read((char *) intAry, 3 * sizeof(int))) {
        int numRead;
        numRead = fsIn.gcount() / sizeof (int);

        // process array
        for (int i = 0; i < numRead; i++)
           cout << intAry[i] << "   ";
        cout << endl;
    } // while
}   // main
```

```
/*
Results:
           1    3    4
           6    7    9
           10   12   13
*/
```

Reading a structure

structures (records): a named collection of fields grouped together for processing a unit of information

```
struct STU
  {
   char     name[20];
   int      exams[3];
   char     grade;
  } // STU
```



Program: Read student file

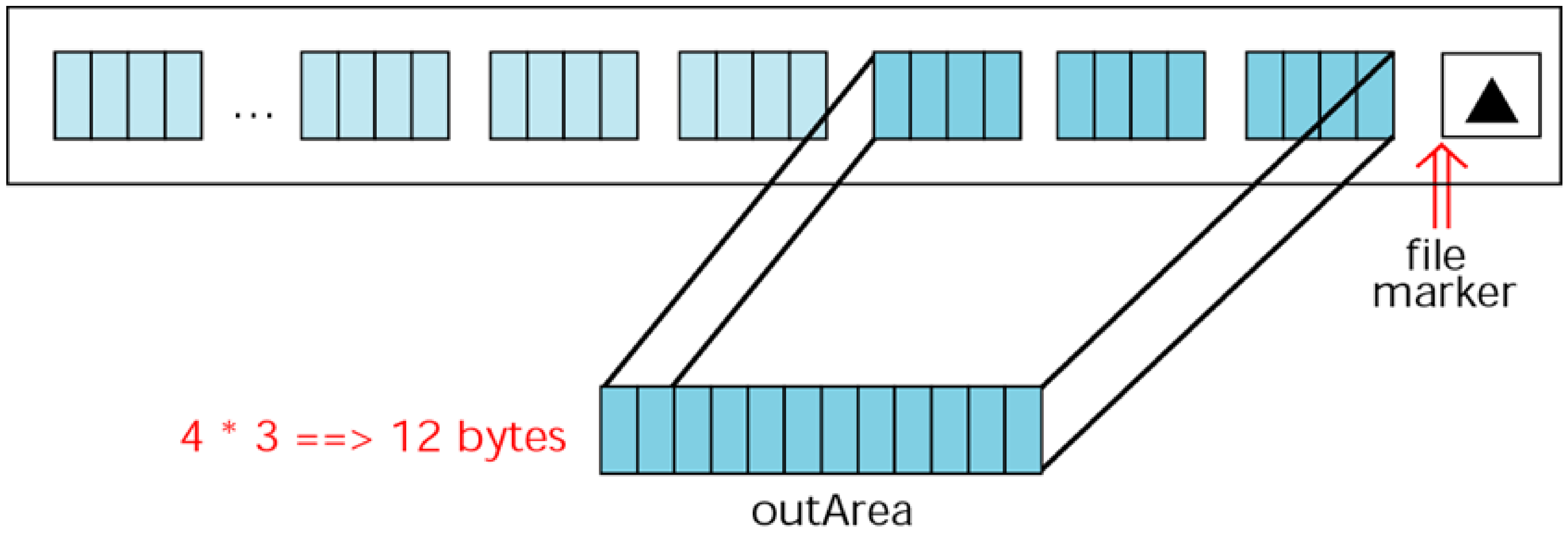
```
/*Reads one student's data from a file.
      Pre   stuFile is opened for reading
      Post  stu data structure filled
            returns true if successful/false if not
*/
bool readStudent  (STU& oneStudent, ifstream& fsStudent)
{
   fsStudent.read((char *) &oneStudent, sizeof(STU));
   bool ioResult = fsStudent.good();
   return ioResult;
} // readStudent
```

write operation

- writes a specified number of items to a binary file

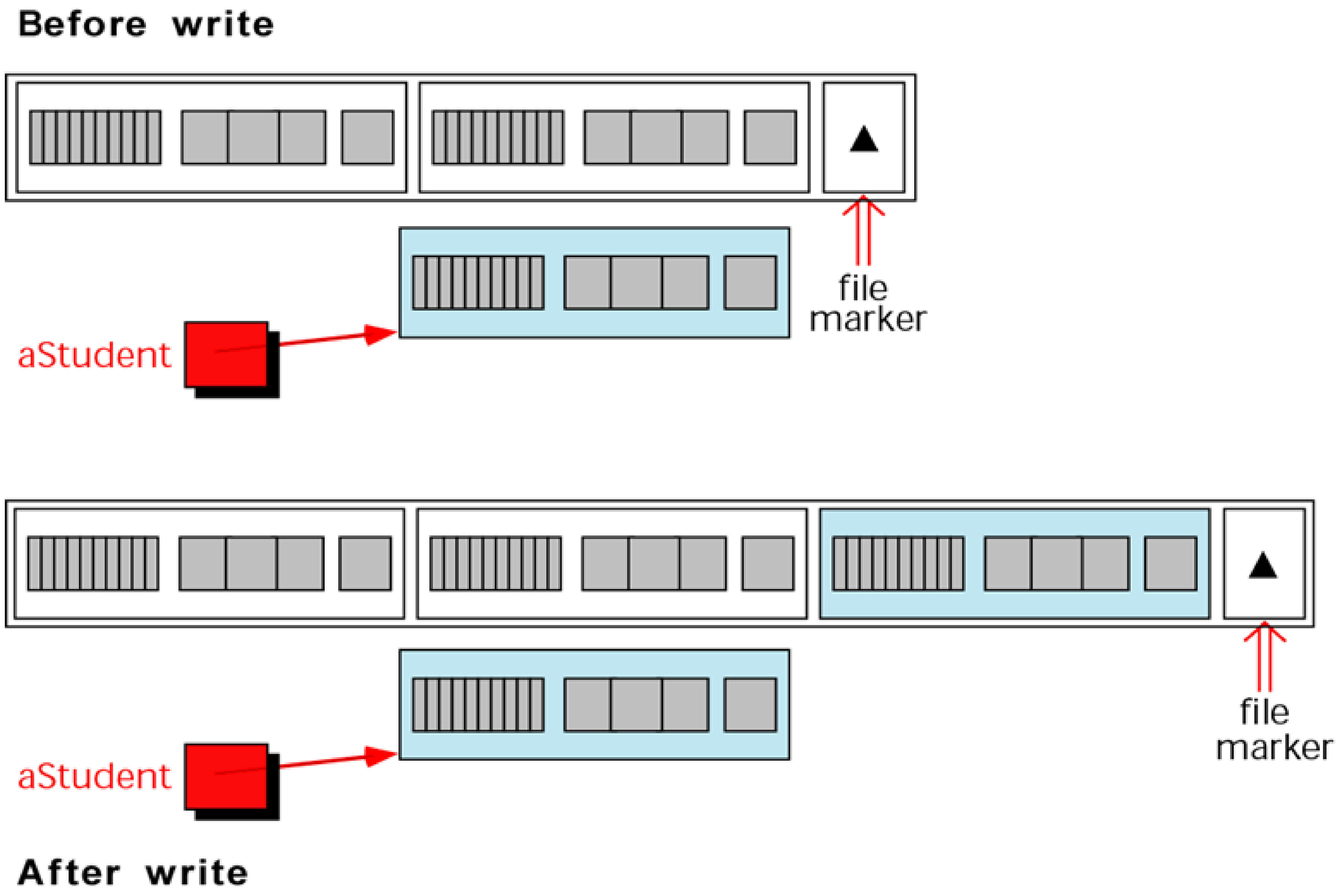
```
ostream& write ( const char* buffer, int size );
```

buffer: a pointer to the output area in memory
size: specify how many bytes are to be written



```
fsOut.write ( (char *) outArea, 3 * sizeof (int)) ;
```

Writing a structure



Program: Write structured data

```
/* Writes one student's record to a binary file.
      Pre   aStudent has been filled
            fileOut is open for writing
      Post  aStudent written to fileOut
*/
void writeStudent (STU&      aStudent,
                   ofstream& fsStuOut)
{
   fsStuOut.write ((char*) &aStudent, sizeof(STU));
   if (!fsStuOut.good())
      {
       cout << "□aError 100 writing student file□a□n";
       exit (100);
      } // if
   return;
}  // writeStudent
```

Tell & Seek operation (for positioning)

- Used to randomly process data in disk files; or
- to change a file's state (e.g., from write state to read state)

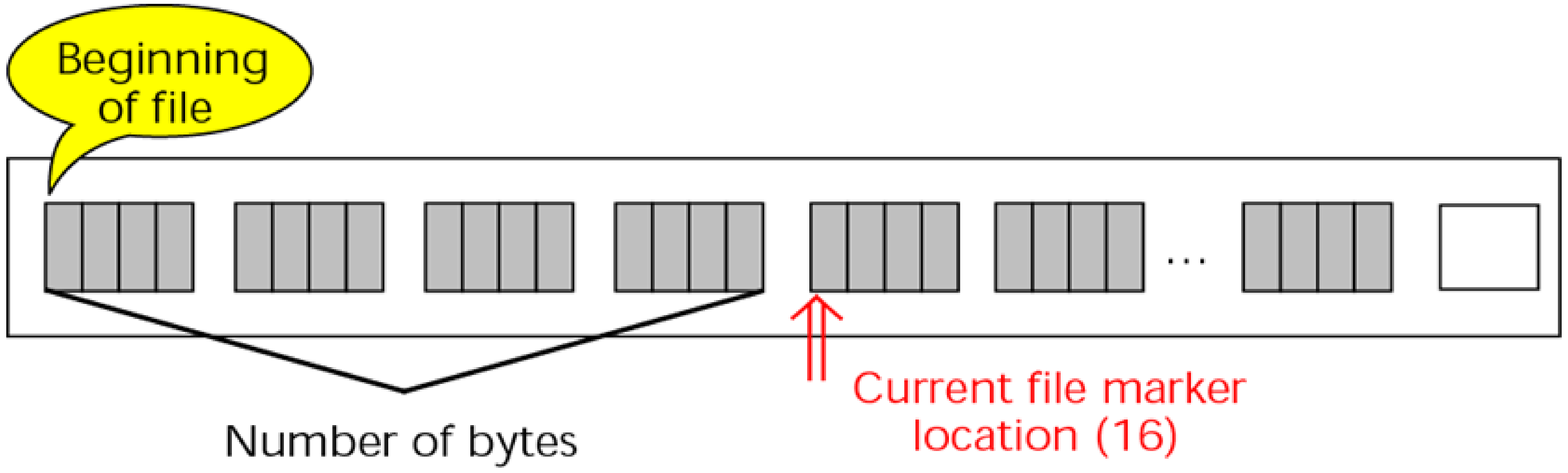
Get position Functions

```
streampos location;

location = fsIn.tellg();
location = fsOut.tellp();
```

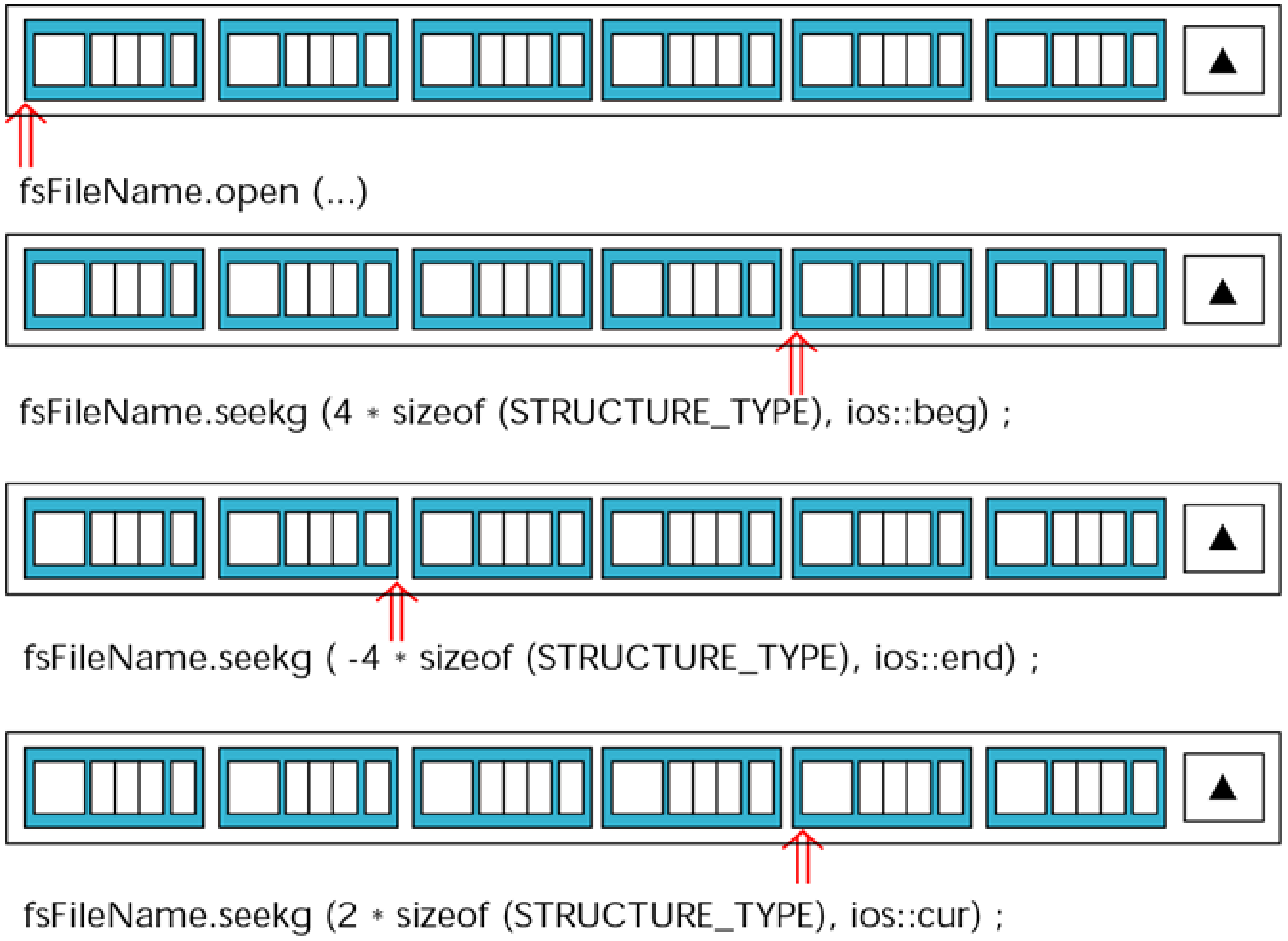
Set position Functions

```
enum seek_dir {beg, cur, end};

istream& seekg ( long offset, ios::seek_dir wherefrom );
ostream& seekp ( long offset, ios::seek_dir wherefrom );
```

tell operation (for positioning)



Seek operation and ios::seek_dir values


fsFileName.open (...)
fsFileName.seekg (4 * sizeof (STRUCTURE_TYPE), ios::beg) ;
fsFileName.seekg (-4 * sizeof (STRUCTURE_TYPE), ios::end) ;
fsFileName.seekg (2 * sizeof (STRUCTURE_TYPE), ios::cur) ;

Examples: Set Position

```
fsStreamName.seekg(99L);  // set the file makrer to byte 100 on a file
fsStreamName.seekg(99L, ios::beg);      // same as the above

fsBinFile.seekg(sizeof(int), ios::cur);      // advance one integer forward
fsStuFile.seekg(sizeof(STU), ios::cur);    // move to the next record

fsStuOut.seekp(0L, ios::end);                // position the file at the end

fsFileName.seekg(tell_location, ios::beg);  // go back to saved location
fsFileName.seekp(tell_location, ios::beg);
```

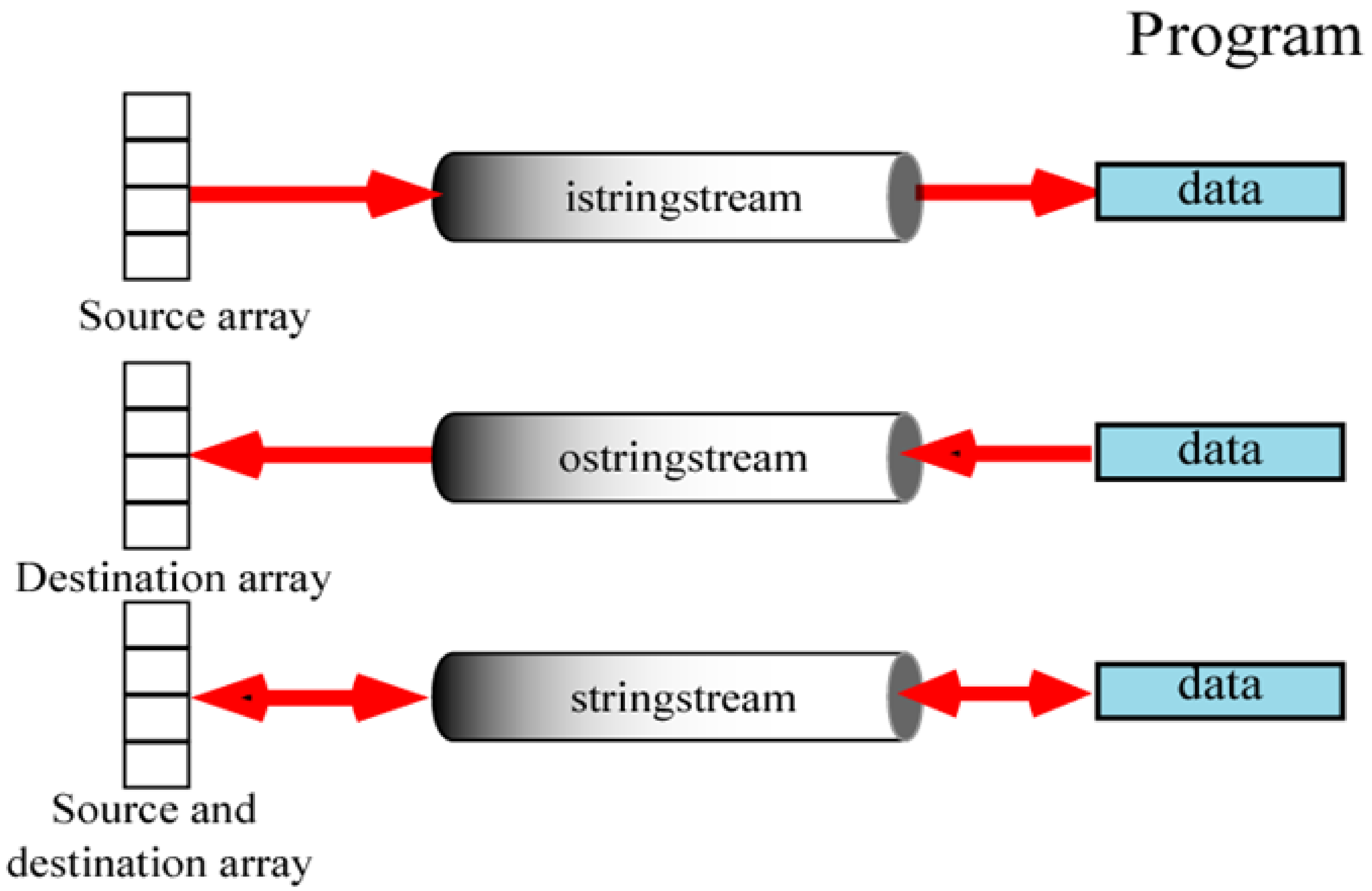
String Streams

- Formatting capabilities found in text files are very useful
  - ➔ Can we use these capabilities to deal with strings in a program?

- C++ defines three I/O classes, whose input source or output destination is a string
  - *istringstream*
  - *ostringstream*
  - *stringstream*

- These streams allows us to connect streams and strings so that we can *read* a string and store its data in a set of variables or *write* a set of variables to a string

Stringstream objects



Program: Writing to a string

```
#include <sstream>
#include <iostream>
#include <iomanip>
#include <string>
using namespace std;

int main ()
{
    cout << "Begin ostringstream demonstration□n";
    ostringstream ssOut;

    ssOut << setw(4) << 23
          << setw(4) << 'a'
          << setw (8) << 23.6 << endl ;
    cout << ssOut.str ();
    cout << "End of ostringstream demonstration□n";
    return 0;
}   // main
```

```
/*          Results:
Begin ostringstream
demonstration
  23   a    23.6
End of ostringstream
demonstration
*/
```

Program: Reading from a string

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

int main ()
{
    string s = "22 A 34.2";
    istringstream ssIn (s);

    int i;
    ssIn >> i;
    char charA;
    ssIn >> charA;
    float fNum;
    ssIn >> fNum;

    cout << i     << " "
         << charA << " "
         << fNum  << endl;
    return 0;
}   // main
```

```
/*        Results:
Begin ostringstream
demonstration
22 A 34.2
End of ostringstream
demonstration
*/
```

Questions?