INTRODUCTION TO PROGRAMING

Chapter 3

File Processing

Khoa Công Nghệ Thông Tin
Trường Đại Học Khoa Học Tự Nhiên
ĐHQG-HCM

GV: Thái Hùng Văn

Objectives

In this chapter, you will:

- Understand about Data Hierarchy and File Concepts.
- Learn about Input /Output Streams and File Streams
- Understand and distinguish File Types
- Learn about how to open /create a file to read /write or append data to it
- Learn about how read /write strings from /to a file.
- Learn about how read /write binary contents from /to a file.
- Discover some other file operations to improve the practical programs.
- Examine File Handling in C ++ program.

Outlines

- **Introduction**
- **The Data Hierarchy**
- **File Types**
- **Input /Output Streams**
- **Stream Headers, Templates and Classes**
- **File Streams**
- **File Modes**
- **Writing Data from a Text File**
- **Reading Data from a Text File**
- **Example**
- Issues to expand career knowledge

Introduction

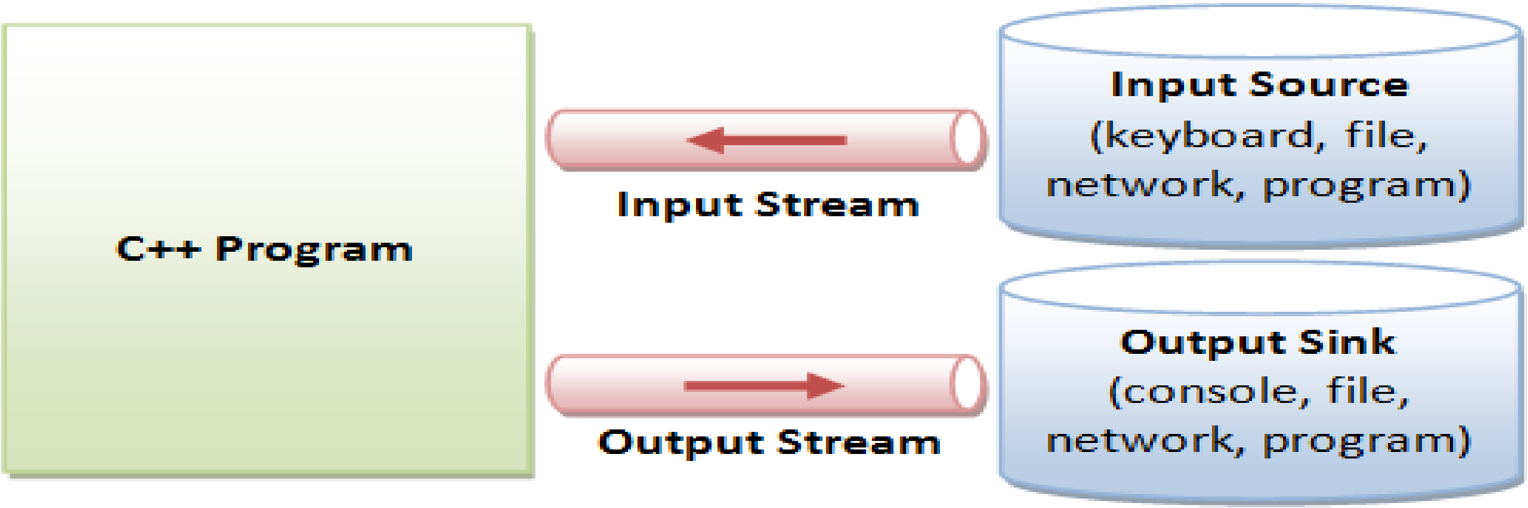
- Storage of data
  - Arrays, strings, structs, and **all variables** in C++ are **temporary** (stored in RAM)
  - **Files** are **permanent** (stored in secondary storage i.e. disk, cards)
- Size of data
  - The total size of the static variables is limited by the size of the stack (very small, most systems don't auto-grow stacks). On Windows, the typical maximum size for a **stack** is **1MB**.
  - The total size of the dynamic variables is limited by the size of the heap. Heap can grow to **all available** (virtual) **memory**, and not too big.
  - The data stored in the **file** is **unlimited** in **size**.
- Data access speed
  - File access speed on HDD or USB disk is much slower than RAM. Not bad with SSD, and will be equivalent if it is RAM disk

Data hierarchy

- **Data hierarchy** refers to the systematic organization of data, often in a hierarchical form. The components of the data hierarchy are listed below:
  - A **Data field** holds a single fact or attribute of an entity. Consider a date field, e.g. "15/10/2019". There are a single date field, or 3 sub fields: day of month, month and year.
  - A **Record** is a collection of related fields. An Employee record may contain a name fields address fields, birthdate field, so on.
  - A **File** is a collection of related records. If there are N employees, then each employee would have a record
  - Files are integrated into a **database**. This is done using a *Database Management System*

Streams

- In C programming, we input/output data using **streams**, which are sequence of bytes flowing in /out of the programs.
- We can associate a stream with a device or with a file.



C++ Program

Input Stream

Output Stream

Input Source
(keyboard, file, network, program)

Output Sink
(console, file, network, program)

Internal Data Formats:

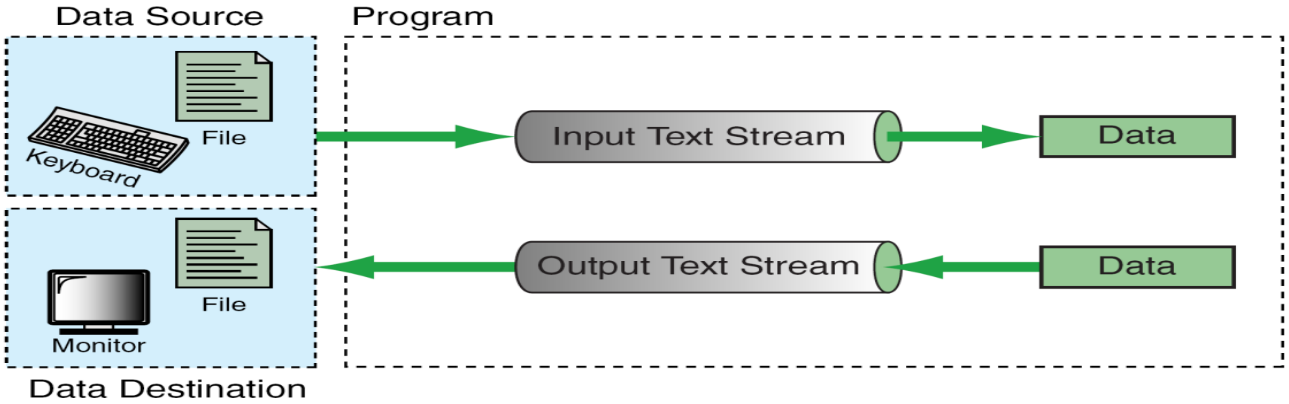
- Text: char, wchar_t
- int, float, double, etc.

External Data Formats:

- Text in various encodings (US-ASCII, ISO-8859-1, UCS-2, UTF-8, UTF-16, UTF-16BE, UTF16-LE, etc.)
- Binary (raw bytes)

Input /Output Streams

- In **input** operations, data bytes flow from an ***input source*** *(such as keyboard, file,..)* into the program.
- In **output** operations, data bytes flow from the program to an ***output sink*** (*console*, *file*, *network*, *another program*,..)
- In data representation form, there are two types of streams:
  - ***Text streams*** consist of sequential characters divided into lines. Each line terminates with the newline character (`\n`).
  - ***Binary streams*** consist of data values such as integers, floats or complex data types, "using their memory representation."



IO stream functions /operations

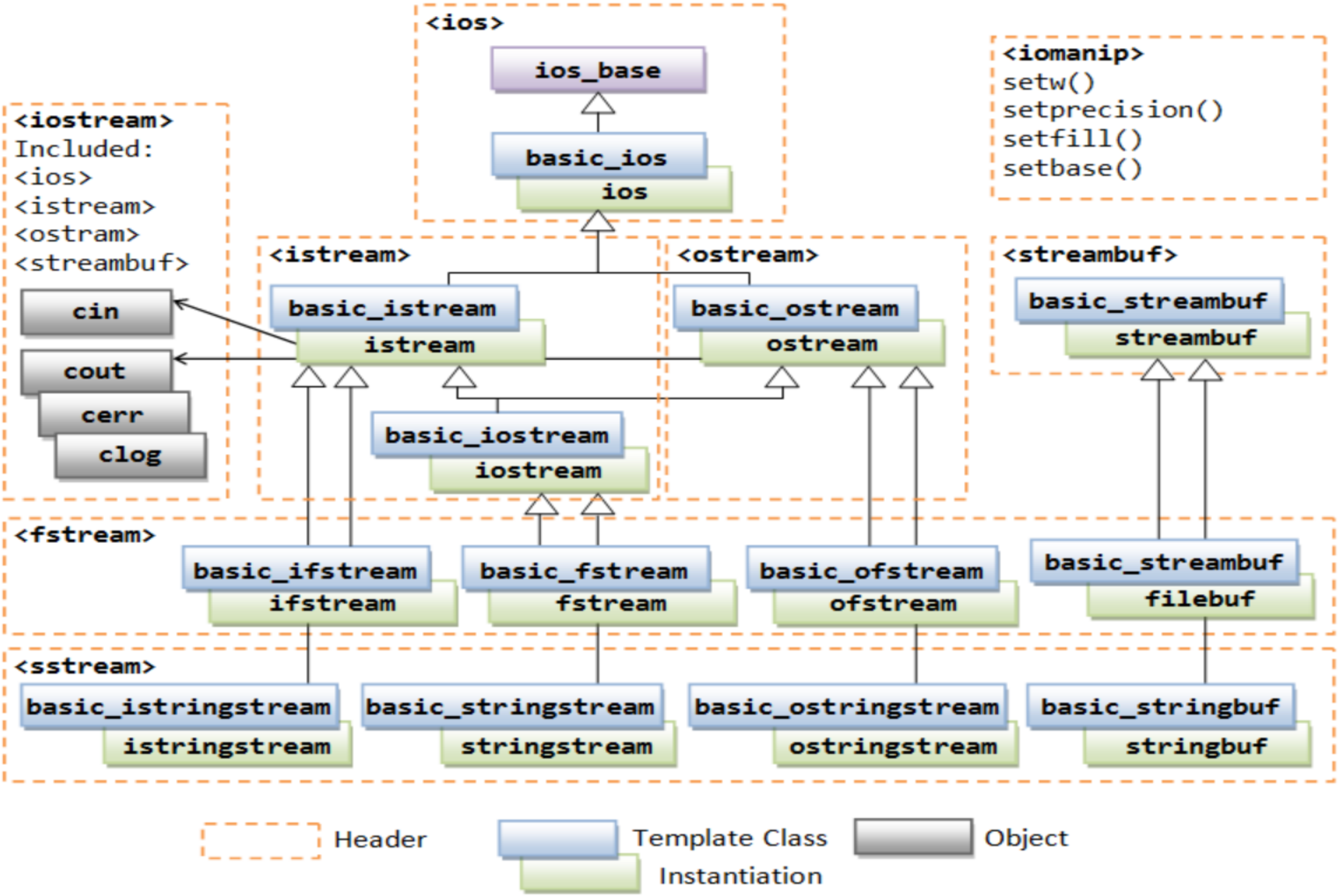
- C++ streams provide both the formatted & unformatted IO functions.
  - In **formatted** or high-level IO, bytes are grouped and converted to types such as **int**, **double**, **string** or **user-defined types**.
  - In **unformatted** or low-level IO, bytes are treated as **raw bytes** and unconverted.
- Formatted IO operations are supported via overloading the stream insertion (<<) and stream extraction (>>) operators, which presents a consistent public IO interface.
- Examples:

```
int a, b, c=65;
cin>>a>>b; // input 2 integer numbers to a and b from keyboard
cout<<a+b<<endl; //output a+b to console and set cursor to newline
cout.put(c); // output 'A' to console ('A' == 65)
```

C++ Stream Headers

- C++ IO stream is provided in some main headers:
  - <**iostream**>: included <ios>, <istream>, <ostream> and <streambuf>; provided basic functions /operations on the standard IO device (keyboard, screen)
  - <**fstream**> : for file IO
  - <**sstream**> : for string IO
  - <**iomanip**> provided manipulators such as setw(), setprecision(), setfill(), setbase(),.. for formatting

Stream Headers, Templates and Classes



https://www.ntu.edu.sg/home/ehchua/programming/cpp/cp10_IO.html

Mechanism of performing IO via Stream

1. Construct a stream object.
2. Connect (associate) the stream object to an actual IO device *(e.g., keyboard, console, file, ..)*
3. Perform input/output operations on the stream, via the functions defined in the stream's pubic interface in a device independent manner.
4. Disconnect (dissociate) the stream to the actual IO device *(e.g., close the file).*
5. Free the stream object.

Files & Streams

- A file is an “independent entity” with a name recorded by the OS.
- A stream is created by a program.
- To work with a file, we must associate stream name (in our program) with the file name (and its path).

Example:

```
ofstream fout; // fout is our stream name (it’s a variable)
fout.open ("D:\\test\\Example.txt") ; // file name is "Example.txt")
```

File Streams

- We have been using the **iostream** standard library, which provides **cin** and **cout** methods for reading /writing from /to standard IO devide respectively.
- For reading /writing from /to Files, we use another standard C++ library called **fstream**, which defines 3 new data types
  - **ifstream** : Stream class represents the input file stream, is used to read data from files.
  - **ofstream** : represents the output file stream, is used to create files and to write data to files.
  - **fstream** : has the capabilities of both **ofstream** and **ifstream** ; it can create files, write data to files, and read data from files.

File Types

- In programming, all files can be categorized into one of two file formats - **binary** or **text**.
- Both binary and text files contain data stored as a series of bytes, and may look the same on the surface, but they encode data differently.
  - **The bytes in text files represent characters**
  - **The bits in binary files represent custom data.**
- **Text files contain only textual data**
- **Binary files may contain both textual and custom binary data.**
- C supports two types of files (text stream files and binary stream files) with similar handling methods.

Main steps in Processing a File

1. Create the file stream.
2. Open file, connect the stream name with the file name.
3. Read or write the data
4. Close the file.

Detailed steps for Processing Files

1: *To access file handling routines:* **#include <fstream>**

2: *To declare variables that can be used to access file:*

```
ifstream in_stream;  ofstream out_stream;
```

3: *To connect program's variable to a file*

```
in_stream.open(InputFileName);
out_stream.open(OutputFileName);
```

4: *To see if the file opened successfully:*

```
if ( !in_stream.fail() || !out_stream.fail() )
```

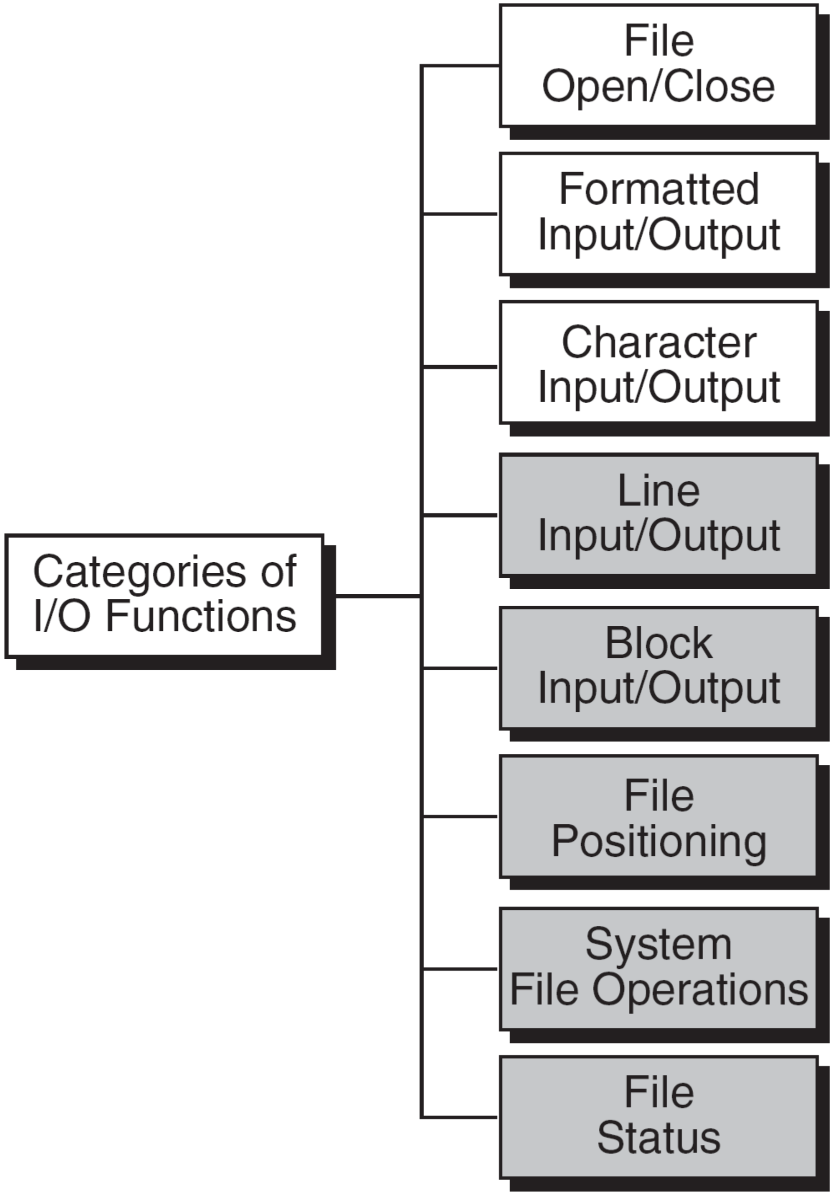
5: *To get data from a file or put data into a file:*

```
in_stream >> a >> b >> c;

out_stream << x << y << z;
```

7: *When done with the file:*

```
in_stream.close(); out_stream.close();
```

Standard Functions in File Processing



Openning a File

- A file must be opened before you can read or write data
  - **ifstream** object is used to open a file for reading purpose only.
  - Either **ofstream** or **fstream** may be used to open a file for writing
- Following is the standard syntax for **open**() function, which is a member of **fstream**, **ifstream**, and **ofstream** objects.

```
void open(const char* filename, ios::openmode mode);
```

- Here, the 1st argument specifies the **filename** and **location** to be opened and the 2nd argument defines the **mode** in which the file should be opened.
- To perform file processing in C++, the header files **<iostream>** and **<fstream>** must be included

File Modes

- File_Mode is an optional parameter with a combination of the following flags:
  - ios::**in** - open file for input operation
  - ios::**out** - open file for output operation
  - ios::**app** - output appends at the end of the file.
  - ios::**trunc** - truncate the file and discard old contents.
  - ios::**binary** - for binary file operation, instead of text file.
  - ios::**ate** - the file pointer "at the end" for input/output.
- You can set multiple flags via bit-OR (|) operator, e.g., ios::out | ios::app to append output at the end of file.
- For output, the default is **ios::out | ios::trunc**. For input, the default is **ios::in**.

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 function takes flushes the buffers and closes the file:

myfile.close();

- Once *close()* 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 this function.

Writing on Text File

- Using operator (<<) with **ofstream** object just as we use that operator to output data to the screen *(similar **cout** object)*
- The ios::**binary** flag is **not included** in the opening mode
- The content of the text file will be similar to the content we see on the screen if we use the **cout**.

Example:

```
ofstream f_out;
f_out.open ("D:/Test/Example.txt");  // open Text File for output
if ( ! f_out.fail() )   {           // open successful
    int a = 2020, b = 6;
    double d = 2020.06;
    f_out << a << " + "<< b << " = " << a+b <<endl;
    f_out << d;
    f_out.close();
 }else  cout << "Unable to open file";
```

Reading from Text File

- Similar with writing to text file, reading from a file can also be performed in the same way that we did with **cin**

- The steps are:

  1. Construct an **ifstream** object.
  2. Connect it to a file (*open file*) and set the file mode operation.
  3. Perform output operation via extraction << operator or **read**(), **get**(), **getline**(),.. functions.
  4. Disconnect (close file) and free the ifstream object.

```
ifstream fin;
fin.open(filename, mode);
......
fin >> a >> b >> c;
......
fin.close();
```

Reading & Writing Text File – Example #1

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
    // Write to file
    ofstream fout ("D:/Example.txt");  // default mode is ios::out | ios::trunc
    if (fout.fail()) return 1;
    fout << "This is a line."<< endl;
    fout << "This is another line."<< endl;
    fout.close();

    // Read from file
    ifstream fin("D:\\Example.txt");  // default mode ios::in
    if (!fin.fail()) return 2;
    char ch;
    while (fin.get(ch))   // till end-of-file
        cout << ch;
    fin.close();
    return 0;
}
```

Reading & Writing Text File – Example #2

```
int a = 2021, b = 4;
float f = 2021.04;
char s[80] = "Testing # ";
ofstream fout;
fout.open ("D:\\test\\Example.txt") ;  if (!fout)  return ;
fout << s << endl << "2021  4.2021  4 \n"
          <<  --a << "  " << ++f << "   " <<  b ;
fout.close();
ifstream fin ("D:/test/Example.txt") ;  if (!fin) return ;
fin.getline(s, 80);
fin >> a >> f  >> b;
cout <<s<<a<<'*'<<b<<'*'<<f<<endl;  // => Testing # 2021*4*4.2021
fin >> a >> b  >> s[2] >> f;
cout <<s<<a<<'*'<<b<<'*'<<f<<endl; // => Te.ting # 2020*2022*4
fin.close();
```

Reading and Writing on Binary File

- For binary files, the operators >> and << is **not efficient.**
- File streams include 2 member functions specifically designed to read and write binary data:
  - **write** (*memory_block, size*); // ***memory_block** is of type **char****
  - **read** (*memory_block, size*); // ***size** is the number of characters*

Example:

```
ofstream fout ("D:/Example.bin", ios::binary);  // open for output
if ( ! fout.fail() )   {          // open successful
    int n = 2020; c = 'A';
    int a[100] = { 22, 6, 2020, 7, 1, 30 };
    fout.write((char*)&n,sizeof(n));
    fout.write((char*)&c, 1);
    fout.write((char*)a, 6*sizeof(int));
    fout.close(); } // file size: 4+1+6*4=29B
```

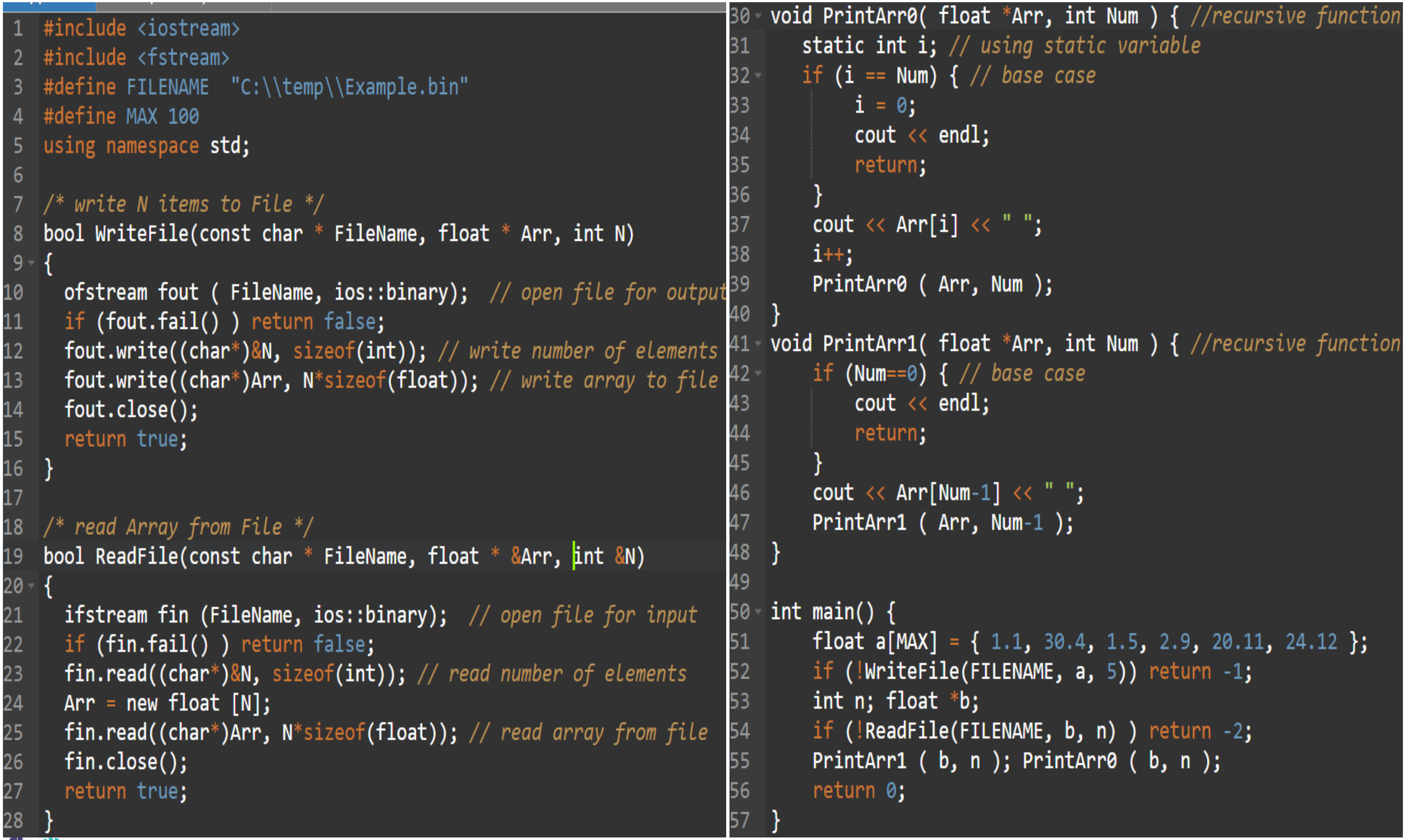
Reading & Writing Binary File – Example *[with header]*

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
    ofstream fout ("D:/Example.bin", ios::binary);  // open binary file for output
    if (fout.fail() ) return 1;
    short int N = 6;
    float A[100] = { 3.1, 6.2, 2021.3, 20.4, 2.5, 2022.6 };
     fout.write((char*)&N, sizeof(short int)); // write number of elements
    fout.write((char*)A, N*sizeof(float)); // write array to file
    fout.close();

    ifstream fin ("D:/Example.bin", ios::binary);  // open file for input
     if (fin.fail() ) return 2;
     int Num;
     fin.read((char*)&Num, sizeof(short int)); // read number of elements
     float * B = new float [Num];
    fin.read((char*)B, Num*sizeof(float)); // read array from file
    fin.close();
}
```

Reading & Writing Binary File – Example *[no header]*

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
    ofstream fout ("D:/Example.bin", ios::binary);  // open binary file for output
    if (fout.fail() ) return 1;
    float A[100] = { 3.1, 6.2, 2021.3, 20.4, 2.5, 2022.6 };
    fout.write((char*)A, 5*sizeof(float)); // write array to file
    fout.close(); // file size: 5*4=20 B

    ifstream fin ("D:/Example.bin", ios::binary | ios::ate);  // open file for input
     if (fin.fail() ) return 2;
     int Num = fin.tellg() / sizeof(float); // get number of elements
     float * B = new float [Num];
    fin.seekg (0, ios::beg);
    fin.read((char*)B, Num*sizeof(float)); // read array from file
    fin.close();
}
```

Reading & Writing Binary File – Example *[advanced]*



```
#include <iostream>
#include <fstream>
#define FILENAME  "C:\\temp\\Example.bin"
#define MAX 100
using namespace std;

/* write N items to File */
bool WriteFile(const char * FileName, float * Arr, int N)
{
  ofstream fout ( FileName, ios::binary);  // open file for output
  if (fout.fail() ) return false;
  fout.write((char*)&N, sizeof(int)); // write number of elements
  fout.write((char*)Arr, N*sizeof(float)); // write array to file
  fout.close();
  return true;
}

/* read Array from File */
bool ReadFile(const char * FileName, float * &Arr, int &N)
{
  ifstream fin (FileName, ios::binary);  // open file for input
  if (fin.fail() ) return false;
  fin.read((char*)&N, sizeof(int)); // read number of elements
  Arr = new float [N];
  fin.read((char*)Arr, N*sizeof(float)); // read array from file
  fin.close();
  return true;
}
```

```
void PrintArr0( float *Arr, int Num ) { //recursive function
   static int i; // using static variable
   if (i == Num) { // base case
        i = 0;
        cout << endl;
        return;
    }
    cout << Arr[i] << " ";
    i++;
    PrintArr0 ( Arr, Num );
}
void PrintArr1( float *Arr, int Num ) { //recursive function
    if (Num==0) { // base case
        cout << endl;
        return;
    }
    cout << Arr[Num-1] << " ";
    PrintArr1 ( Arr, Num-1 );
}

int main() {
    float a[MAX] = { 1.1, 30.4, 1.5, 2.9, 20.11, 24.12 };
    if (!WriteFile(FILENAME, a, 5)) return -1;
    int n; float *b;
    if (!ReadFile(FILENAME, b, n) ) return -2;
    PrintArr1 ( b, n ); PrintArr0 ( b, n );
    return 0;
}
```

Position Functions

- IO streams objects keep internally internal position (>=1):
  - ifstream keeps the location of the element to be read in the next input operation.
  - ofstream keeps location where the next element has to be written.
  - fstream keeps both, the get and the put position, like iostream.
- **tellg**() & **tellp**() : return a value of the member type streampos, which is a type representing the current get or put position
- **seekg**() & **seekp**() : allow to change the location of the get and put positions. Both functions are overloaded with 2 different prototypes:
  - **seekg** (position); / **seekp** (position);
  - **seekg** (offset, direction); / **seekp** (offset, direction);

  // direction = ios::beg / ios::cur / ios::end *(offset counted from it)*

Checking state flags functions

The following member functions check for specific states of a stream *(they return a bool value)*:

- **bad**() : returns true if a reading or writing operation fails..
- **fail**() : returns true in the same cases as bad(), but also in the case that a format error happens, like when an alphabetical character is extracted when we are trying to read an integer number.
- **eof**() : returns true if a file open for reading has reached the end.
- **good**() : returns false in the same cases in which calling any of the previous functions would return true. Note that *good* and *bad* are not exact opposites (*good* checks more state flags at once).
- **clear**() : can be used to reset the state flags.

Quiz

End!