Section D: Files and Streams

Files

All the programs we have looked at so far use input only from the keyboard, and output only to the screen. If we were restricted to use only the keyboard and screen as input and output devices, it would be difficult to handle large amounts of input data, and output data would always be lost as soon as we turned the computer off. To avoid these problems, we can store data in some secondary storage device, usually magnetic tapes or discs. Data can be created by one program, stored on these devices, and then accessed or modified by other programs when necessary. To achieve this, the data is packaged up on the storage devices as data structures called files.

A file is a container for data. Think of a file as a drawer in a file cabinet which needs to be opened and closed. Before you can put something into a file, or take something out, you must open the file (the drawer). When you are finished using the file, the file (the drawer) must be closed.

The easiest way to think about a file is as a linear sequence of characters. In a simplifed picture (which ignores special characters for text formatting) these lecture notes might be stored in a file called "Lecture_4" as:

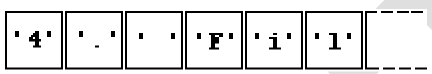


Figure 1

Streams

Before we can work with files in C++, we need to become acquainted with the notion of a stream. We can think of a stream as a channel or conduit on which data is passed from senders to receivers. As far as the programs we will use are concerned, streams allow travel in only one direction. Data can be sent out from the program on an output stream, or received into the program on an input stream. For example, at the start of a program, the standard input stream "cin" is connected to the keyboard and the standard output stream "cout" is connected to the screen.

In fact, input and output streams such as "cin" and "cout" are examples of (stream) objects. So learning about streams is a good way to introduce some of the syntax and ideas behind the object-oriented part of C++. The header file which lists the operations on streams both to and from files is called "fstream". We will therefore assume that the program fragments discussed below are embedded in programs containing the "include" statement

```
#include<fstream>
```

As we shall see, the essential characteristic of stream processing is that data elements must be sent to or received from a stream one at a time, i.e. in serial fashion.

Creating a Sequential Access File

Steps to create (or write to) a sequential access file:

1. Declare a stream variable name:

ofstream fout; //each file has its own stream buffer

ofstream is short for output file stream
fout is the stream variable name
Naming the stream variable "fout" is helpful in remembering that the information is going "out" to the file.

2. Open the file:

fout.open("scores.dat", ios::out);

fout is the stream variable name previously declared **"scores.dat"** is the name of the file
ios::out is the steam operation mode (your compiler may not require that you specify the stream operation mode.)

3. Write data to the file:

```
fout<<grade<<endl;
fout<<"Mr. Spock\n";
```

The data must be separated with space characters or end-of-line characters (carriage return), or the data will run together in the file and be unreadable. Try to save the data to the file in the same manner that you would display it on the screen.

If the **iomanip.h** header file is used, you will be able to use familiar formatting commands with file output.

```
fout<<setprecision(2);
fout<<setw(10)<<3.14159;
```

4. Close the file:

fout.close( );

Closing the file writes any data remaining in the buffer to the file, releases the file from the program, and updates the file directory to reflect the file's new size. As soon as your program is finished accessing the file, the file should be closed. Most systems close any data files when a program terminates. Should data remain in the buffer when the program terminates, you may lose that data.

Reading from a sequential-access file

To read data from a file you need one FileStream object and one StreamReader object. The StreamReader object accepts the FileStream object as its argument.

```
 FileStream fs;
StreamReader fr;
//create file stream object
fs = new FileStream("D:\\test.txt", FileMode.Open , FileAccess.Read );
//create reader objec
fr=new StreamReader(fs);
string content=fr.ReadLine() ;
while (!fr.EndOfStream )
{
Console.WriteLine(content);
content = fr.ReadLine();
}
//close FileStream object
fs.Close();
```

Updating a Sequential File

The data in many sequential files cannot be modified without the risk of destroying other data in the file.

If the name “White” needed to be changed to “Worthington,” the old name cannot simply be overwritten, because the new name requires more space.

Fields in a text file — and hence records — can vary in size.

Records in a sequential-access file are not usually updated in place. Instead, the entire file is usually rewritten.

Rewriting the entire file is uneconomical to update just one record, but reasonable if a substantial number of records need to be updated.

Mater file versus transaction files: the merge operation

RandomAccessFile

To create a random access file, use the class java.io.RandomAccessFile.

The seek( ) method allows repositioning of the filePointer.

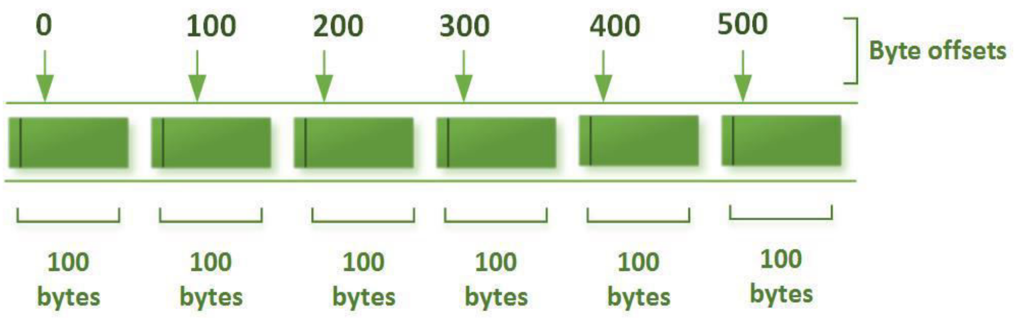
Demo of using RandomAccessFile class to create a file for both read and write operations

Exercise: Examine the content of the file test.out before and after running the application (next page), and explain why the content of the file was changed that way.

ndividual records of a random-access file are normally fixed in length and may be accessed directly (and thus quickly) without searching through other records.

This makes random-access files appropriate for :

- airline reservation systems
- banking systems
- point-of-sale systems
- other kinds of transaction-processing systems that require rapid access to specific data.



Random-Access File

Creating a Random-Access File

Functions fwrite and fread are capable of reading and writing arrays of data to and from disk. The third argument of both fread and fwrite is the number of elements in the array that should be read from or written to disk.

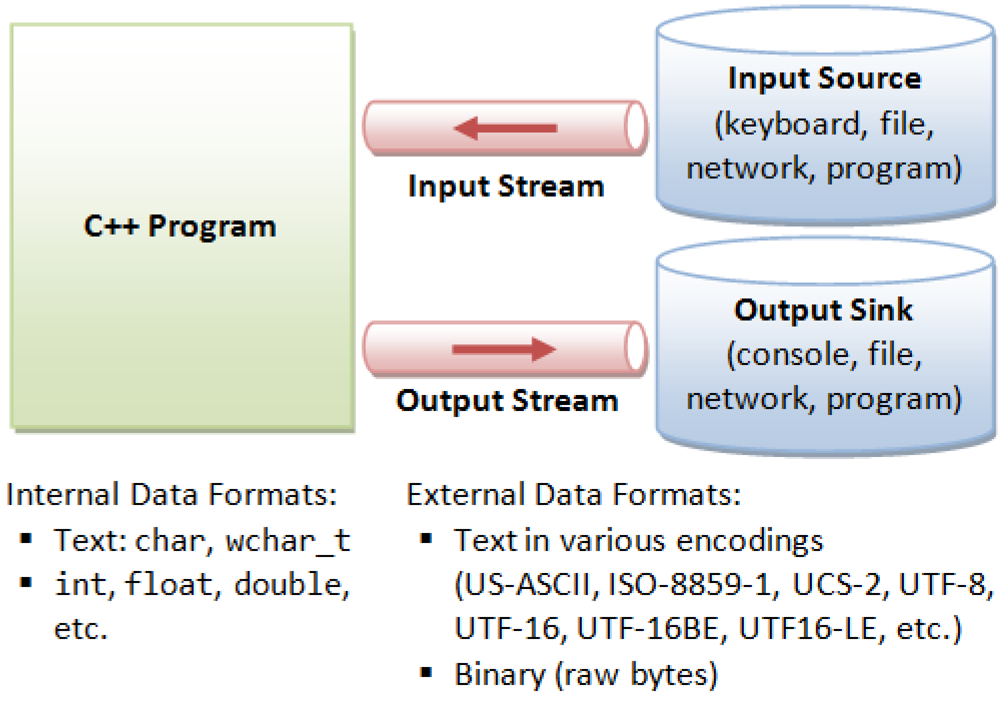
Writing Data Randomly to a Random-Access File

The program writes data to the file "credit.dat". It uses the combination of fseek and fwrite to store data at specific locations in the file.

Stream I/O

Streams

C/C++ IO are based on streams, which are sequence of bytes flowing in and out of the programs (just like water and oil flowing through a pipe). In input operations, data bytes flow from an input source (such as keyboard, file, network or another program) into the program. In output operations, data bytes flow from the program to an output sink (such as console, file, network or another program). Streams acts as an intermediaries between the programs and the actual IO devices, in such the way that frees the programmers from handling the actual devices, so as to archive device independent IO operations.



C++ provides both the formatted and 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.

To perform input and output, a C++ program:

1. Construct a stream object.
2. Connect (Associate) the stream object to an actual IO device (e.g., keyboard, console, file, network, another program).
3. Perform input/output operations on the stream, via the functions defined in the stream's pubic interface in a device independent manner. Some functions convert the data between

the external format and internal format (formatted IO); while other does not (unformatted or binary IO).
4. Disconnect (Dissociate) the stream to the actual IO device (e.g., close the file).
5. Free the stream object.

File Input/Output (Header <fstream>)

C++ handles file IO similar to standard IO. In header <fstream>, the class ofstream is a subclass of ostream; ifstream is a subclass of istream; and fstream is a subclass of iostream for bi-directional IO. You need to include both <iostream> and <fstream> headers in your program for file IO.

To write to a file, you construct a ofsteam object connecting to the output file, and use the ostream functions such as stream insertion <<, put() and write(). Similarly, to read from an input file, construct an ifstream object connecting to the input file, and use the istream functions such as stream extraction >>, get(), getline() and read().

File IO requires an additional step to connect the file to the stream (i.e., file open) and disconnect from the stream (i.e., file close).

File Output

The steps are:

1. Construct an ostream object.
2. Connect it to a file (i.e., file open) and set the mode of file operation (e.g, truncate, append).
3. Perform output operation via insertion >> operator or write(), put() functions.
4. Disconnect (close the file which flushes the output buffer) and free the ostream object.

```
#include <fstream>
.......
ofstream fout;
fout.open(filename, mode);
......
fout.close();

// OR combine declaration and open()
ofstream fout(filename, mode);
```

By default, opening an output file creates a new file if the filename does not exist; or truncates it (clear its content) and starts writing as an empty file.

open(), close() and is_open()

```
void open (const char* filename,
      ios::openmode mode = ios::in | ios::out);
```

// open() accepts only C-string. For string object, need to use c_str() to get the C-string

void **close** ();   // Closes the file, flush the buffer and disconnect from stream object

bool **is_open** (); // Returns true if the file is successfully opened

File Modes

File modes are defined as static public member in ios_base superclass. They can be referenced from ios_base or its subclasses - we typically use subclass ios. The available file mode flags are:

1. ios::**in** - open file for input operation
2. ios::**out** - open file for output operation
3. ios::**app** - output appends at the end of the file.
4. ios::**trunc** - truncate the file and discard old contents.
5. ios::**binary** - for binary (raw byte) IO operation, instead of character-based.
6. ios::**ate** - position 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 the file.

For output, the default is ios::out | ios::trunc. For input, the default is ios::in.

File Input

The steps are:

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

```
#include <fstream>
.......
ifstream fin;
fin.open(filename, mode);
......
fin.close();

// OR combine declaration and open()
ifstream fin(filename, mode);
```

Unformatted Input/Output Functions

put(), get() and getline()

The ostream's member function put() can be used to put out a char. put() returns the invoking ostream reference, and thus, can be cascaded. For example,

```
// ostream class
ostream & put (char c);  // put char c to ostream
// Examples
cout.put('A');
cout.put('A').put('p').put('p').put('\n');
cout.put(65);
```

Stream Manipulators

C++ provides a set of manipulators to perform input and output formatting:

1. <iomanip> header: setw(), setprecision(), setbas(), setfill().
2. <iostream> header: fixed|scientific, left|right|internal, boolalpha|noboolalpha, etc.

States of stream

The steam superclass ios_base maintains a data member to describe the states of the stream, which is a bitmask of the type iostate. The flags are:

- eofbit: set when an input operation reaches end-of-file.
- failbit: The last input operation failed to read the expected characters or output operation failed to write the expected characters, e.g., getline() reads n characters without reaching delimiter character.
- badbit: serious error due to failure of an IO operation (e.g. file read/write error) or stream buffer.
- goodbit: Absence of above error with value of 0.

These flags are defined as public static members in ios_base. They can be accessed directly via ios_base::failbit or via subclasses such as cin::failbit, ios::failbit. However, it is more convenience to use these public member functions of ios class:

- good(): returns true if goodbit is set (i.e., no error).
- eof(): returns true if eofbit is set.
- fail(): returns true if failbit or badbit is set.
- bad(): returns true if badbit is set.

- clear(): clear eofbit, failbit and badbit.

Error State of Streams

It probably comes as no surprise that streams have an error state. Our examples have avoided it to this point, so we'll deal with it now. When an error occurs, flags are set in the state according to the general category of the error.

iostate flag	**Error category**
ios_base::goodbit	Everything's fine
ios_base::eofbit	An input operation reached the end of an input sequence
ios_base::failbit	An input operation failed to read the expected character, or An output operation failed to generate the desired characters
ios_base::badbit	Indicates the loss of integrity of the underlying input or output sequence

There are several situations when both eofbit and failbit are set; however, the two have different meanings and do not always occur in conjunction. The flag ios_base::eofbit is set when there is an attempt to read past the end of an input sequence. This occurs in the following two typical examples:

1. Assume the extraction happens character-wise. Once the last character is read, the stream is still in good state; eofbit is not yet set. Any subsequent extraction, however, will be considered an attempt to read past the end of the input sequence. Thus, eofbit will be set.
2. If you do not read character-wise, but extract an integer or a string, for example, you will always read past the end of the input sequence. This is because the input operators read characters until they find a separator, or hit the end of the input sequence. If the input contains the sequence ... 912749<eof> and an integer is extracted, eofbit will be set.

The flag ios_base::failbit is set as the result of a read or write operation that fails. For example, if you try to extract an integer from an input sequence containing only white spaces, the extraction fails, and the failbit is set. Let's see whether failbit would be set in the previous examples:

1. After reading the last available character, the extraction not only reads past the end of the input sequence; it also fails to extract the requested character. Hence, failbit is set in addition to eofbit.
2. Here it is different. Although the end of the input sequence is reached by extracting the integer, the input operation does not fail and the desired integer will indeed be read. Hence, in this situation only the eofbit will be set.

In addition to these input and output operations, there are other situations that can trigger failure. For example, file streams set failbit if the associated file cannot be opened .

The flag ios_base::badbit indicates problems with the underlying stream buffer. These problems could be:

- ***Memory shortage***. There is no memory available to create the buffer, or the buffer has size zero for other reasons or the stream cannot allocate memory for its own internal data, as with iword and pword.
- ***The underlying stream buffer throws an exception***. The stream buffer might lose its integrity, as in memory shortage, or code conversion failure, or an unrecoverable read error from the external device. The stream buffer can indicate this loss of integrity by throwing an exception, which is caught by the stream and results in setting the badbit in the stream's state.

Generally, you should keep in mind that badbit indicates an error situation that is likely to be unrecoverable, whereas failbit indicates a situation that might allow you to retry the failed operation. The flag eofbit simply indicates the end of the input sequence.

Function Templates

Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type.

A template is a blueprint or formula for creating a generic class or a function. The library containers like iterators and algorithms are examples of generic programming and have been developed using template concept.

There is a single definition of each container, such as **vector**, but we can define many different kinds of vectors for example, **vector <int>** or **vector <string>**.

The general form of a template function definition is shown here:

```
template <class type> ret-type func-name(parameter list) {
  // body of function
}
```

Here, type is a placeholder name for a data type used by the function. This name can be used within the function definition

Function templates are special functions that can operate with *generic types*. This allows us to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type.

In C++ this can be achieved using *template parameters*. A template parameter is a special kind of parameter that can be used to pass a type as argument: just like regular function parameters can be used to pass values to a function, template parameters allow to pass also types to a function. These function templates can use these parameters as if they were any other regular type.

The format for declaring function templates with type parameters is:

```
template <class identifier> function_declaration;
template <typename identifier> function_declaration;
```

The only difference between both prototypes is the use of either the keyword class or the keyword typename. Its use is indistinct, since both expressions have exactly the same meaning and behave exactly the same way.

Class templates

We also have the possibility to write class templates, so that a class can have members that use template parameters as types. For example:

```
template <class T>
class mypair {
   T values [2];
  public:
   mypair (T first, T second)
   {
     values[0]=first; values[1]=second;
   }
};
```

The class that we have just defined serves to store two elements of any valid type. For example, if we wanted to declare an object of this class to store two integer values of type int with the values 115 and 36 we would write:

```
mypair<int> myobject (115, 36);
```

this same class would also be used to create an object to store any other type:

```
mypair<double> myfloats (3.0, 2.18);
```

The only member function in the previous class template has been defined inline within the class declaration itself. In case that we define a function member outside the declaration of the class template, we must always precede that definition with the template <...> prefix:

Non-type parameters for templates

Besides the template arguments that are preceded by the class or typename keywords , which represent types, templates can also have regular typed parameters, similar to those found in functions. As an example, have a look at this class template that is used to contain sequences of elements:

```
/ sequence template
#include <iostream>
using namespace std;

template <class T, int N>
class mysequence {
    T memblock [N];
  public:
    void setmember (int x, T value);
    T getmember (int x);
};

template <class T, int N>
void mysequence<T,N>::setmember (int x, T value) {
  memblock[x]=value;
}

template <class T, int N>
T mysequence<T,N>::getmember (int x) {
  return memblock[x];
}

int main () {
  mysequence <int,5> myints;
  mysequence <double,5> myfloats;
  myints.setmember (0,100);
  myfloats.setmember (3,3.1416);
  cout << myints.getmember(0) << '\n';
  cout << myfloats.getmember(3) << '\n';
  return 0;
}
```

```
100

3.1416
```

Templates and Inheritance

Templates may be used in many combinations with inheritance. It is possible to combine inheritance and templates because a template class is a class, albeit one with a parameter. Combining these language features allows the parameterization ability of templates to be used in conjunction with the specializing abilities of inheritance. Four combinations of templates and inheritance are presented in this section:

- a template class that inherits from another template class,
- a non-template class that inherits from a template class,
- a template class that inherits from a non-template class, and
- a template class that uses multiple inheritance.

Templates and Friends

There are four kinds of relationships between classes and their friends when templates are involved:

- *One-to-many*: A non-template function may be a friend to all template class instantiations.
- *Many-to-one*: All instantiations of a template function may be friends to a regular non-template class.
- *One-to-one*: A template function instantiated with one set of template arguments may be a friend to one template class instantiated with the same set of template arguments. This is also the relationship between a regular non-template class and a regular non-template friend function.
- *Many-to-many*: All instantiations of a template function may be a friend to all instantiations of the template class.

Static Data Members and Templates

A static declaration within a class template declares a static data member for each template class generated from the template. The static declaration can be of template argument type or of any defined type.

Like member function templates , you can explicitly define a static data member of a template class at file scope for each type used to instantiate a template class. For example:

```
template <class T> class key
{
public:
    static T x;
};
int key<int>::x;
char key<char>::x;
void main()
```

```
{
    key<int>::x = 0;
}
```

You can also define a static data member of a template class using a template definition at file scope. For example:

```
template <class T> class key
{
public:
    static T x;
};
template <class T> T key<T> ::x; // template definition
void main()
{
    key<int>::x = 0;
}
```

Exception Handling

When you instantiate a template class, you must have either an explicit definition or a template definition for each static data member, but not both.

An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: **try, catch,** and **throw**.

- **throw:** A program throws an exception when a problem shows up. This is done using a **throw** keyword.
- **catch:** A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The **catch** keyword indicates the catching of an exception.
- **try:** A **try** block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

Assuming a block will raise an exception, a method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

```
try
{
  // protected code
}catch( ExceptionName e1 )
{
  // catch block
}catch( ExceptionName e2 )
{
  // catch block
}catch( ExceptionName eN )
{
  // catch block
}
```

You can list down multiple **catch** statements to catch different type of exceptions in case your **try** block raises more than one exception in different situations.

Throwing Exceptions

Exceptions can be thrown anywhere within a code block using **throw** statements. The operand of the throw statements determines a type for the exception and can be any expression and the type of the result of the expression determines the type of exception thrown.

Following is an example of throwing an exception when dividing by zero condition occurs:

```
double division(int a, int b) {
  if( b == 0 ) {
    throw "Division by zero condition!";
  }
  return (a/b);
}
```

Catching Exceptions

The **catch** block following the **try** block catches any exception. You can specify what type of exception you want to catch and this is determined by the exception declaration that appears in parentheses following the keyword catch.

```
try {
  // protected code
}catch( ExceptionName e ) {
  // code to handle ExceptionName exception
}
```

Above code will catch an exception of **ExceptionName** type. If you want to specify that a catch block should handle any type of exception that is thrown in a try block, you must put an ellipsis, ..., between the parentheses enclosing the exception declaration as follows:

```
try {
   // protected code
}catch(...) {
   // code to handle any exception
}
```

The following is an example, which throws a division by zero exception and we catch it in catch block.

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

double division(int a, int b) {
   if( b == 0 ) {
      throw "Division by zero condition!";
   }
   return (a/b);
}

int main () {
   int x = 50;
   int y = 0;
   double z = 0;

   try {
      z = division(x, y);
      cout << z << endl;
   }catch (const char* msg) {
      cerr << msg << endl;
   }

   return 0;
}
```

Because we are raising an exception of type **const char***, so while catching this exception, we have to use const char* in catch block. If we compile and run above code, this would produce the following result:

```
Division by zero condition!
```