A C++ program for doing the same task:

```
// listcpp.cpp
#include <fstream>// to use fstream class
using namespace std; // to use standard C++ library

main() {
    char ch;
    fstream infile;

    infile.open("A.txt",ios:in);
    infile.unsetf(ios::skipws);
         // set flag so it doesn't skip white space

    infile >> ch;
    while (! infile.fail())  {
        cout << ch ;
        infile >> ch ;
    }
    infile.close();
}
```


Physical Files and Logical Files

physical file: a collection of bytes stored on a disk or tape

logical file: a "channel" (like a telephone line) that connects the program to a physical file

- The program (application) sends (or receives) bytes to (from) a file through the logical file. The program knows nothing about where the bytes go (came from).

- The operating system is responsible for associating a logical file in a program to a physical file in disk or tape. Writing to or reading from a file in a program is done through the operating system.

Note that from the program point of view, input devices (keyboard) and output devices (console, printer, etc) are treated as files - places where bytes come from or are sent to.

There may be thousands of physical files on a disk, but a program only have about 20 logical files open at the same time.

The physical file has a name, for instance `myfile.txt`

The logical file has a logical name used for referring to the file inside the program. This logical name is a variable inside the program, for instance `outfile`

In C programming language, this variable is declared as follows:

```
FILE * outfile;
```

In C++ the logical name is the name of an object of the class `fstream`:

```
fstream outfile;
```

In both languages, the logical name `outfile` will be associated to the physical file `myfile.txt` at the time of **opening** the file as we will see next.

Opening Files

Opening a file makes it ready for use by the program.
Two options for opening a file :

- open an **existing** file
- create a **new** file

When we open a file we are positioned at the beginning of the file.

How to do it in C:

```
FILE * outfile;
outfile = fopen("myfile.txt", "w");
```

The first argument indicates the physical name of the file.
The second one determines the "mode", i.e. the way, the file is opened. The mode can be:

- `"r"`: open an existing file for input (reading);
- `"w"`: create a new file, or truncate existing one, for output;
- `"a"`: open a new file, or append an existing one, for ouput;
- `"r+"`: open an existing file for input and output;
- `"w+"`: create a new file, or truncate an existing one, for input and output;
- `"a+"`: create a new file, or append an existing one, for input and output;
- `"rb", "wb", "ab","r+b","w+b","a+b"`: same as above but the file is open in binary mode.

How to do it in C++:

```
fstream outfile;
outfile.open("myfile.txt",ios::out);
```

The second argument is an integer indicating the mode.
Its value is set as a "bitwise or" (operator `|`) of constants defined in the class `ios`:

- `ios::in` open for input;
- `ios::out` open for output;
- `ios::app` seek to the end of file before each write;
- `ios::ate` initially position at the end of file;
- `ios::trunc` always create a new file (truncate if exists);
- `ios::binary` open in binary mode (rather than text mode).

C:	`r`	`w`	`a`
C++:	`in`	`out|trunc` or `out`	`out|app`
C:	`r+`	`w+`	`a+`
C++:	`out|in`	`out|in|trunc`	`out|in|app`

All obtions above, if followed by `b` in C, would have `|binary`.
Exercise: Open a physical file `"myfile.txt"` associating it to the logical file `"afile"` and with the following capabilities:

1. input and output (appending mode):
   ```
   afile.open("myfile.txt",
   ios::in|ios::out|ios::app);
   ```
2. create a new file, or truncate existing one, for output:

Closing Files

This is like "hanging up" the line connected to a file.

After closing a file, the logical name is free to be associated to another physical file.

Closing a file used for output guarantees that everything has been written to the physical file.
We will see later that bytes are not sent directly to the physical file one by one; they are first stored in a buffer to be written later as a block of data. When the file is closed the leftover from the buffer is flushed to the file.

Files are usually closed automatically by the operating system at the end of program's execution.
It's better to close the file to prevent data loss in case the program does not terminate normally.

In C :

```
fclose(outfile);
```

In C++ :

```
outfile.close();
```


Reading

Read data from a file and place it in a variable inside the program.

A generic `Read` function (not specific to any programming language):

```
Read(Source_file, Destination_addr, Size)
```

`Source_file`: logical name of a file which has been opened
`Destination_addr`: first address of the memory block were data should be stored
`Size`: number of bytes to be read

In C (or in C++ using C streams):

```
char c;      // a character
char a[100]; // an array with 100 characters
FILE * infile;
  ⋮
infile = fopen("myfile.txt","r");
fread(&c,1,1,infile); /* reads one character */
fread(a,1,10,infile); /* reads 10 characters */
```

`fread`:
1st argument: destination address (address of variable `c`)
2nd argument: element size in bytes (a `char` occupies 1 byte)
3rd argument: number of elements
4th argument: logical file name

In C, read and write operations to files are supported by various functions: `fread, fget, fwrite, fput, fscanf, fprinf`.

In C++ :

```
char c;
char a[100];
fstream infile;
infile.open("myfile.txt",ios::in);
infile >> c;     // reads one character
infile.read(&c,1);
    // alternative way of reading one character
infile.read(a,10); // reads 10 bytes
```

Note that in the C++ version, the operator `>>` communicates the same info at a higher level. Since `c` is a char variable, it's implicit that only 1 byte is to be transferred.

C++ fstream also provide the `read` method, corresponding to `fread` in C.

Contents of today's lecture:

- Field and record organization (textbook: Section 4.1)
- Sequential search and direct access (textbook: Section 5.1)
- Seeking (textbook: Section 2.5)

Reference: Folk, Zoellick and Riccardi, File Structures, 1998. Sections 4.1, 5.1, 2.5.

Files as Streams of Bytes

So far we have looked at a file as a stream of bytes.
Consider the program seen in the last lecture :

```
#include <fstream>
using namespace std;
main() {
    char ch;
    fstream infile;
    infile.open("A.txt",ios:in);
    infile.unsetf(ios::skipws);
        // set flag so it doesn't skip white space
    infile >> ch;
    while (! infile.fail()) {
        cout << ch;
        infile >> ch;
    }
    infile.close();
}
```

Consider the file example: `A.txt`

```
87358CARROLLALICE IN WONDERLAND      <nl>
03818FOLK   FILE STRUCTURES          <nl>
79733KNUTH  THE ART OF COMPUTER PROGR<nl>
86683KNUTH  SURREAL NUMBERS          <nl>
18395TOLKIENTHE HOBITT               <nl>
```

(above we are representing the invisible newline character by `<nl>`)

Every stream has an associated **file position**.

- When we do `infile.open("A.txt",ios::in)` the **file position** is set at the beginning.
- The first `infile >> ch;` will read `8` into `ch` and increment the file position.
- The next `infile >> ch;` will read `7` into `ch` and increment the file position.
- The 38th `infile >> ch;` will read the newline character (referred to as `'\n'` in C++) into `ch` and increment the file position.
- The 39th `infile >> ch;` will read `0` into `ch` and increment the file position, and so on.

A file can been seen as

1. a stream of bytes (as we have seen above); or
2. a collection of records with fields (as we will discuss next ...).

Field and Record Organization

Definitions :

Record	=	a collection of related fields.
Field	=	the smallest logically meaningful unit of information in a file.
Key	=	a subset of the **fields** in a record used to identify (uniquely, usually) the record.

In our sample file "A.txt" containing information about books:
Each line of the file (corresponding to a book) is a record.
Fields in each record: ISBN Number, Author Name and Book Title.

Primary Key: a key that uniquely identifies a record.
Example of primary key in the book file:

Secondary Keys: other keys that may be used for search
Example of secondary keys in the book file:

Note that in general not every field is a key (keys correspond to fields, or combination of fields, that may be used in a search).

Consider the following sample program:

```
#include <fstream>
using namespace std;
int main() {
  fstream myfile;
  myfile.open("test.txt",ios::in|ios::out|ios::trunc
                  |ios::binary);
  myfile<<"Hello,world.\nHello, again.";
  myfile.seekp(12,ios::beg);
  myfile<<'X'<<'X';
  myfile.seekp(3,ios::cur);
  myfile<<'Y';
  myfile.seekp(-2,ios::end);
  myfile<<'Z';
  myfile.close();
  return 0;
}
```

Show "test.txt" after the program is executed:

| | | | | | | | | | | | | | | | | | | | | | | | | | | |

Remove ios::binary from the specification of the opening mode.
Show test.txt after the program is executed under DOS:

| | | | | | | | | | | | | | | | | | | | | | | | | | | |

LECTURE 4: SECONDARY STORAGE DEVICES - MAGNETIC DISKS

Contents of today's lecture:

- Secondary storage devices
- Organization of disks
- Organizing tracks by sector
- Organizing tracks by blocks
- Nondata overhead
- The cost of a disk access
- Disk as a bottleneck

Reference: Folk, Zoellick and Riccardi, File Structures, 1998. Sections 3.1.

Secondary Storage Devices

Since secondary storage is different from main memory we have to understand how it works in order to do good file designs.

Two major types of storage devices:

- Direct Access Storage Devices (DASDs)
  - Magnetic Disks
    Hard Disks (high capacity, low cost per bit)
    Floppy Disks (low capacity, slow, cheap)
  - Optical Disks
    CD-ROM = Compact Disc, read-only memory
    (Read-only/write once, holds a lot of data, cheap reproduction)
- Serial Devices
  - Magnetic tapes (very fast sequential access)