MANAGING FILES OF RECORDS

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 :

```
// listcpp.cpp
#include <fstream.h>
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).

Field Structures

1. Fixed-length fields:
   Like in our file of books (field lengths are 5, 7, and 25).

```
87358CARROLLALICE IN WONDERLAND
03818FOLK   FILE STRUCTURES
79733KNUTH  THE ART OF COMPUTER PROGR
```

2. Field beginning with length indicator:

```
058735907CARROLL19ALICE IN WONDERLAND
050381804FOLK15FILE STRUCTURES
```

3. Place delimiter at the end of fields:

```
87359|CARROLL|ALICE IN WONDERLAND|
03818|FOLK|FILE STRUCTURES|
```

4. Store field as `keyword = value` (self-describing fields):

```
ISBN=87359|AU=CARROLL|TI=ALICE IN WONDERLAND|
ISBN=03818|AU=FOLK|TI=FILE STRUCTURES|
```

   Although the delimiter may not always be necessary here, it is convenient for separating a key value from the next keyword.

Field structures: advantages and disadvantages

Type	Advantages	Disadvantages
Fixed	Easy to Read/Store	Waste space with padding
with length indicator	Easy to jump ahead to the end of the field	Long fields require more than 1 byte to store length (when maximum size is > 256)
Delimited Fields	May waste less space than with length-based	Have to check every byte of field against the delimiter
Keyword	Fields are self describing, allows for missing fields.	Waste space with keywords

Record Structures

1. Fixed-length records.
   It can be used with fixed-length records, but can also be combined with any of the other variable length field structures, in which case we use padding to reach the specified length.

   Examples:

   Fixed-length records combined with fixed-length fields:

```
87358CARROLLALICE IN WONDERLAND
03818FOLK   FILE STRUCTURES
79733KNUTH  THE ART OF COMPUTER PROGR
```

   Fixed-length records combined with variable-length fields:

   delimited fields:

```
87359|CARROLL|ALICE IN WONDERLAND
03818|FOLK|FILE STRUCTURES
79733|KNUTH|THE ART OF COMPUTER PROGR
```

   fields with length indicator:

```
058735907CARROLL19ALICE IN WONDERLAND
050381804FOLK15FILE STRUCTURES
```

2. Records with fixed number of fields (variable-length)

   It can be combined with any of the variable-length field structure.

   Examples: Number of fields per record = 3.

   with delimited fields:

   `87359|CARROLL|ALICE IN WONDERLAND|03818|FOLK|···`

   with fields with length indicator:

   `0587359 07CARROLL19ALICE IN WONDERLAND0503818···`

   In the situations above, how would the program detect that a record has ended ?

3. Record beginning with length indicator.

   Example:
   with delimited field:

   `3387359|CARROLL|ALICE IN WONDERLAND`
   `2603818|FOLK|FILE STRUCTURES`

   Can this method be combined with fields having length indicator or fields having keywords?

4. Use an index to keep track of addresses

   The index keeps the byte offset for each record; this allows us to search the index (which have fixed length records) in order to discover the beginning of the record.

   datafile:

   ```
   87359|CARROLL|ALICE IN WONDERLAND|03818|FOLK|···
   ```

   Complete information on the index file:

   indexfile:

5. Place a delimiter at the end of the record.
   The end-of-line character is a common delimiter, since it makes the file readable at our console.

   ```
   87358|CARROLL|ALICE IN WONDERLAND|<nl>
   03818|FOLK|FILE STRUCTURES|<nl>
   79733|KNUTH|THE ART OF COMPUTER PROGR|<nl>
   ```

Summary :

Type	**Advantages**	**Disadvantages**
Fixed Length Record	Easy to jump to the i-th record	Waste space with padding
Variable Length Record	Saves space when record sizes are diverse	Cannot jump to the i-th record, unless through an index file

Sequential Search and Direct Access

Search for a record matching a given key.

- **Sequential Search**
  Look at records sequentially until matching record is found.
  Time is in $O(n)$ for n records.

  Example when appropriate :
  Pattern matching, file with few records.

- **Direct Access**
  Being able to seek directly to the beginning of the record.
  Time is in $O(1)$ for n records.

  Possible when we know the Relative Record Number (RRN):
  First record has RRN 0, the next has RRN 1, etc.

Direct Access by RRN

Requires records of <u>fixed length</u>.

RRN = 30 (31st record)
record length = 101 bytes
So, byte offset = ________

Now, how to go directly to byte ________ in a file ?
By **seeking** ...

Seeking

Generic seek function :

```
Seek(Source_File, Offset)
```

Example :

```
Seek(infile, 3030)
```

Moves to byte 3030 in `file`.

In C style :

Function prototype:

```
int fseek(FILE *stream, long int offset, int origin);
```

`origin`: 0 = `fseek` from the beginning of file
1 = `fseek` from the current position
2 = `fseek` from the end of file

Examples of usage:

```
fseek(infile,0L,0); // moves to the beginning
                                //of the file

fseek(infile,0L,2); // moves to the end of the file

fseek(infile,-10L,1); // moves back 10 bytes from
                          // the current position
```

In C++ :

Object of class `fstream` has two file pointers :

- `seekg` = moves the get pointer.
- `seekp` = moves the put pointer.

General use:

```
file.seekg(byte_offset,origin);
file.seekp(byte_offset,origin);
```

Constants defined in class ios:

`origin`: ios::beg = `fseek` from the beginning of file
ios::end = `fseek` from the current position
ios::cur = `fseek` from the end of file

The previous examples, shown in C style, become in C++ style:

```
infile.seekg(0,ios::beg);

infile.seekg(0,ios::end);

infile.seekg(-10,ios::cur);
```

Consider the following sample program:

```
#include <fstream.h>
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:

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