Managing Files of Records

Last Time

Fundamental File Processing Operations

Today

- 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. Sections 4.1, 5.1, 2.5.

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); // include white space in read
    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 of 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.

ISBN Number

Secondary Keys: other keys that may be used for search

Author Name
Book Title
Author Name + Book Title (probably unique)

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).

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` :

   ```
   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.

Type	**Advantages**	**Disadvantages**
Fixed	Easy to Read/Store	Waste space with padding
Length Based	Easy to jump ahead to the end of the field	Long fields require more than 1 byte (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. Ex: Tags in HTML files	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:
   ```
   87359|CARROLL|ALICE IN WONDERLAND
   03818|FOLK|FILE STRUCTURES
   ```
   ```
   058735907CARROLL19ALICE IN WONDERLAND
   050381804FOLK15FILE STRUCTURES
   ```

2. Fixed number of fields
   It can be combined with length-based or delimited field.

3. Record beginning with length indicator.
   Example:
   ```
   3387359|CARROLL|ALICE IN WONDERLAND
   2603818|FOLK|FILE STRUCTURES
   ```

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.

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.

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 begging 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 fixed length.

RRN = 30 (31st record)
record length = 101 bytes

So, byte offset = 3030

Now, how to go directly to byte 3030 in a file ?

By **seeking** ...

Seeking

General seek function :

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

Example :

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

Moves to byte 3030 in `file`.

<u>In C style</u> :

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

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
```

<u>In C++</u> :

Object of class `fstream` has two file pointers :

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

The previous examples in C style become:

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

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

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