File Structures

Lab Programs and Mini-Project Demonstrations and Hands-On

1947

Williams-Kilburn tube - used a cathode ray tube (similar to an analog TV picture tube) to store bits as dots on the screen's surface.

1949

Electronic Delay Storage Automatic Calculator (EDSAC) - a stored program computer, used **mercury delay line memory**.

1950

magnetic drum memory

1951

first tape storage device

1,440,000 decimal digits

300 Kg

1953

IBM 726 Magnetic Tape

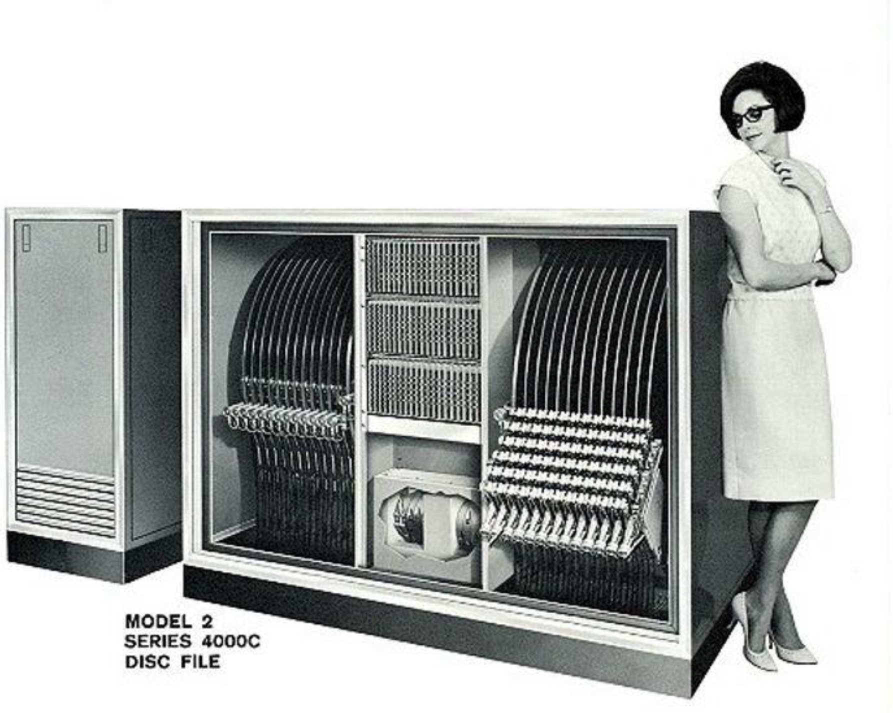
could store 2 million digits per tape

rented for $850 a month.

1959

computer drum

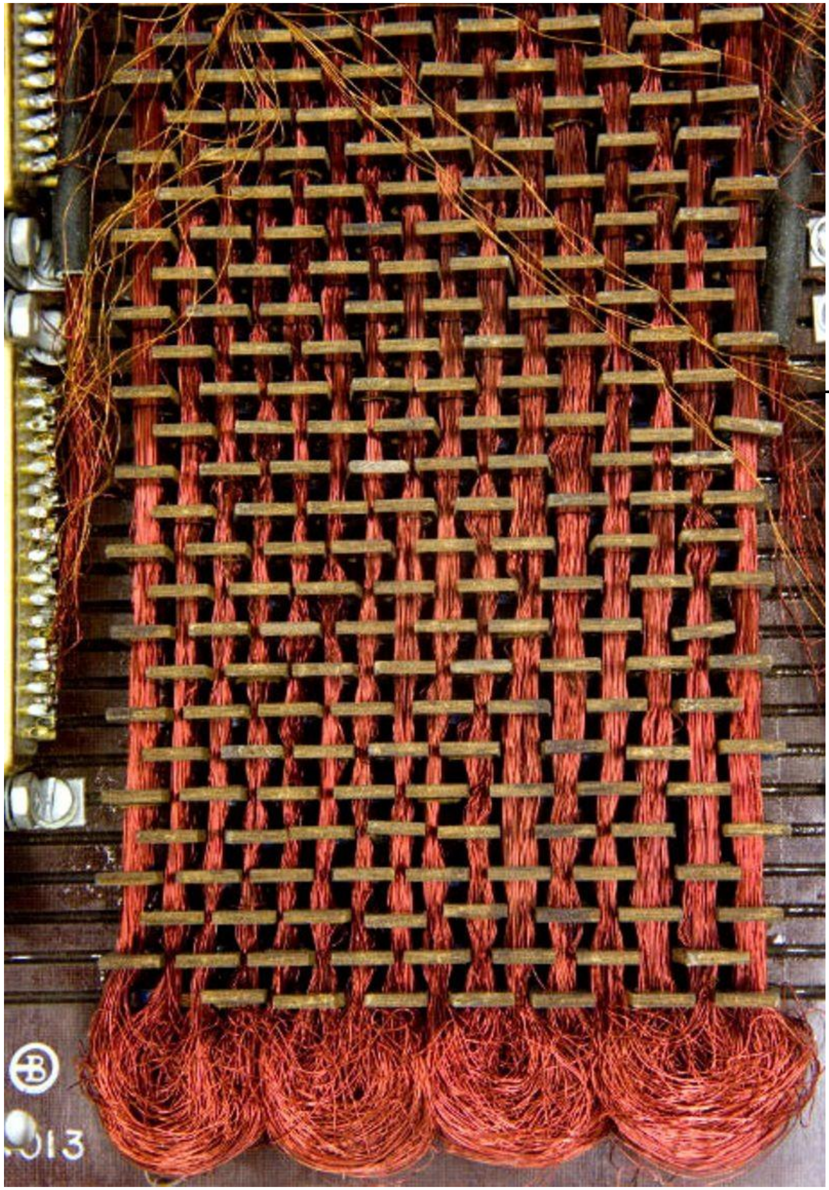
5 million characters of data



1960

Card Random Access Memory (CRAM)

Each CRAM deck of 256 cards recorded about 5.5 MB.



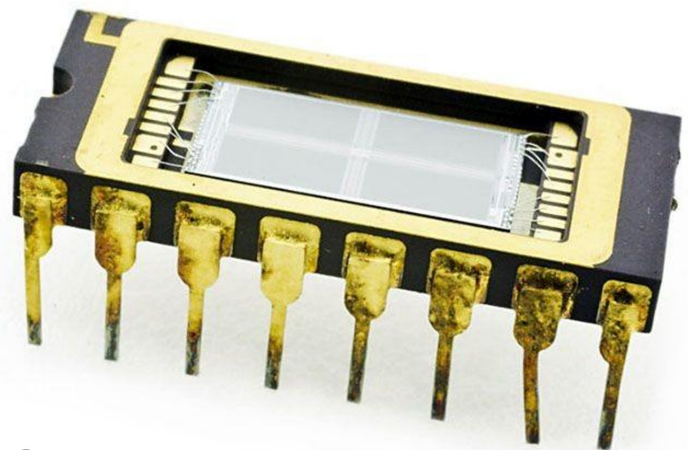
1968

Apollo Guidance Computer read-only rope memory

72 KB of storage

Dynamic Random-Access Memory (DRAM)

64K DRAMs



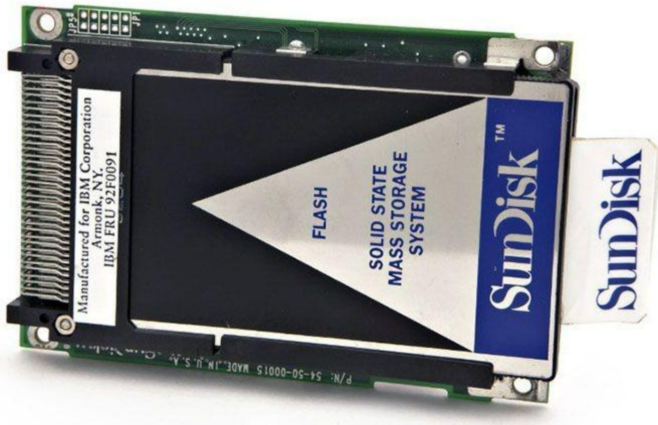
1970

1975

550 megabytes of pre-recorded data

1980

Seagate Hard Disk



1992

Solid State Disk (SSD)

No rotating disks

2000

USB Flash drives

2009

1TB HDD

Dropbox's server farms

2015

12 exabytes = 1000 petabytes =10^18

2006

Cloud Services

Buzzwords

Hierarchy of storage

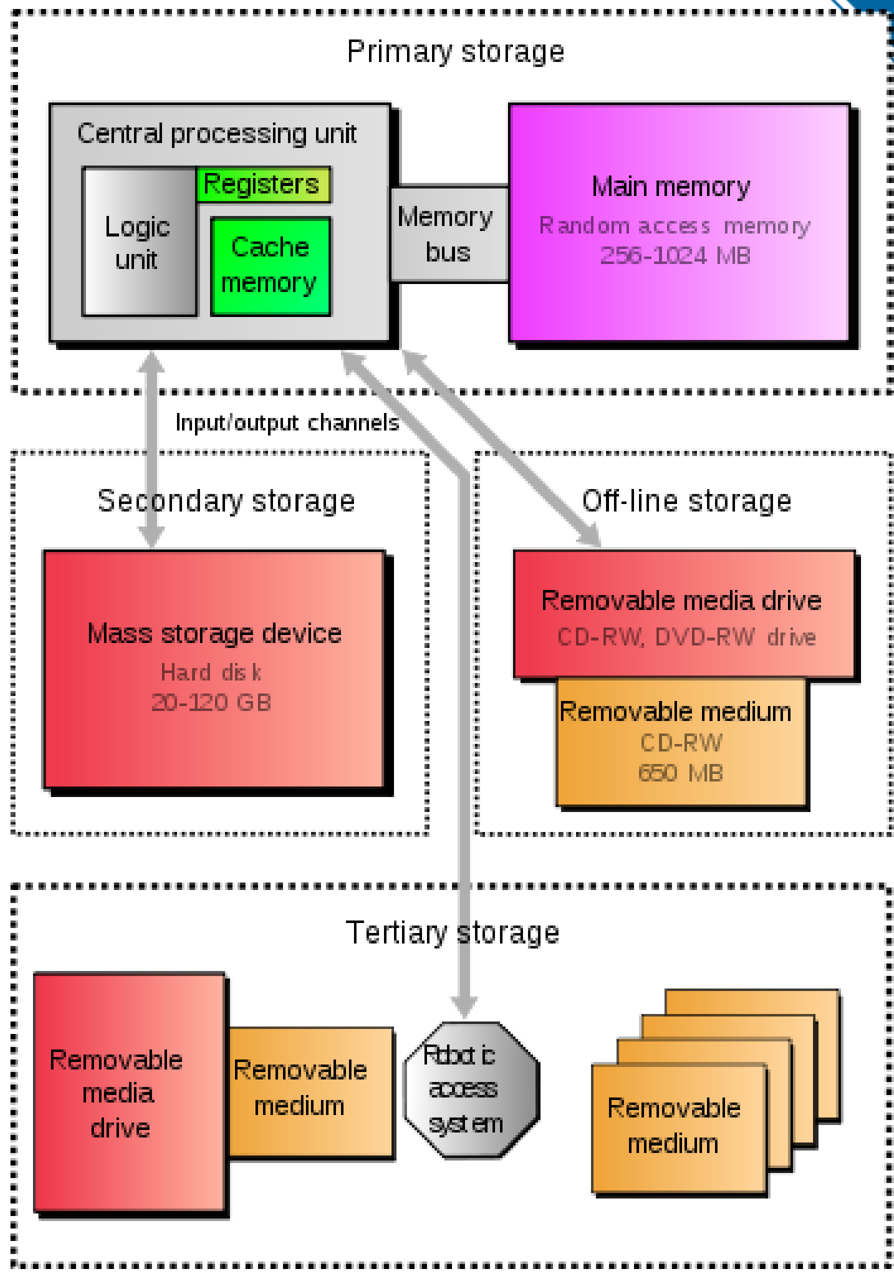
- Primary storage
- Secondary storage
- Tertiary storage
- Off-line storage

Storage media

- Semiconductor
- Magnetic
- Optical
- Paper
- Others

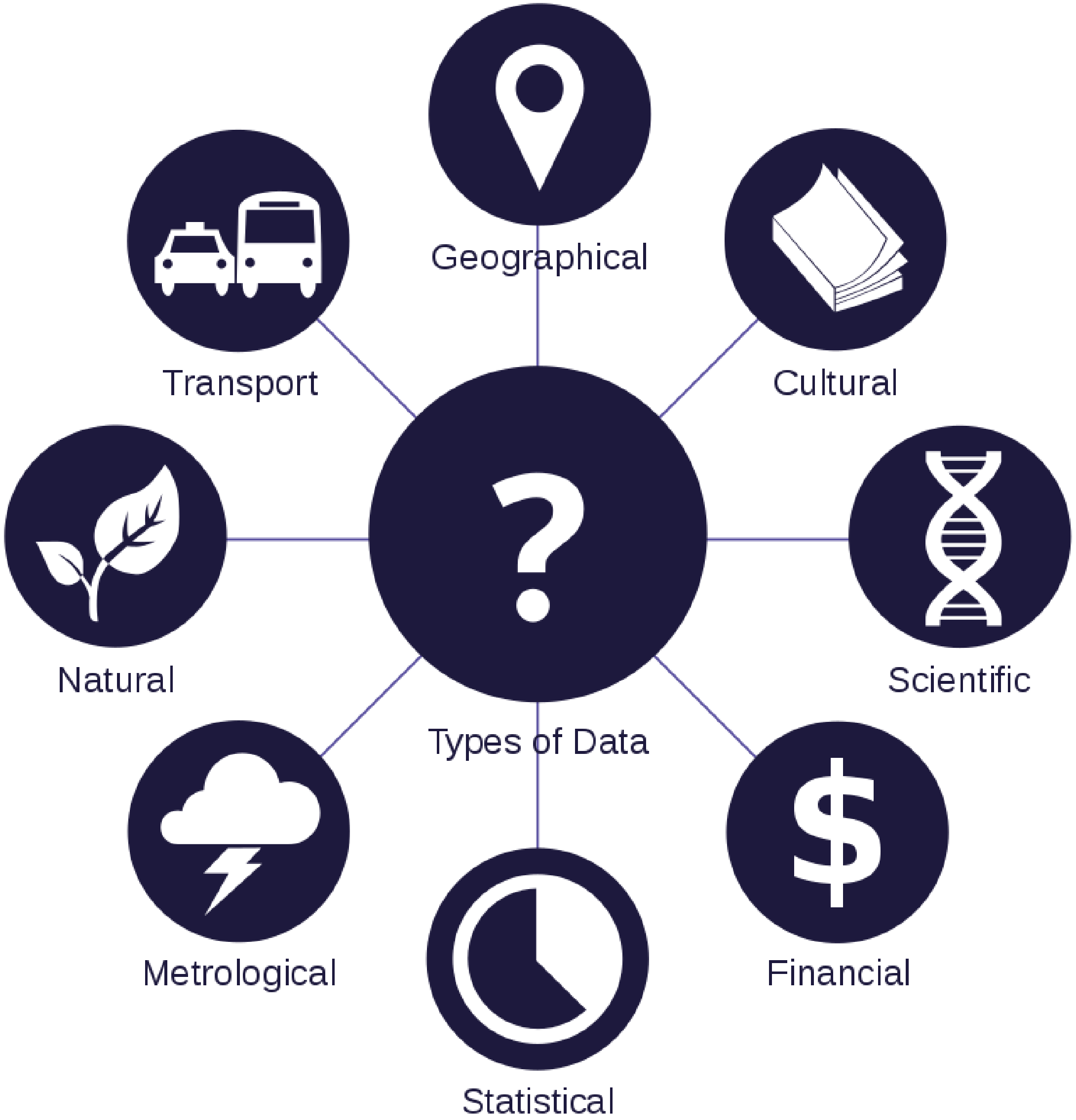
Characteristics of storage

- Volatility
- Mutability
- Accessibility
- Addressability
- Capacity
- Performance
- Energy use
- Security



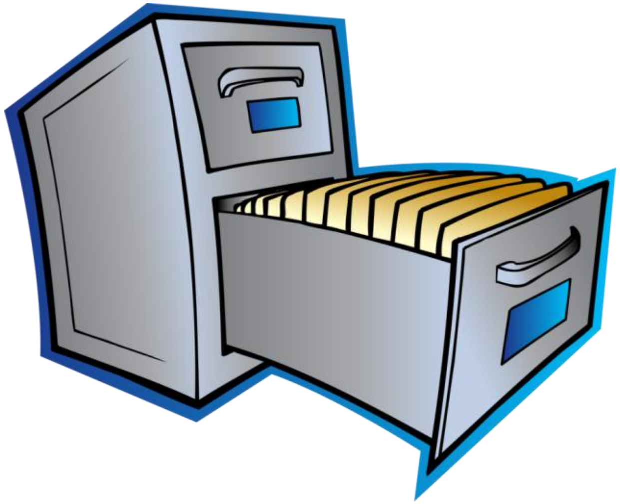
Data?

Information?



Motivation for File Structures

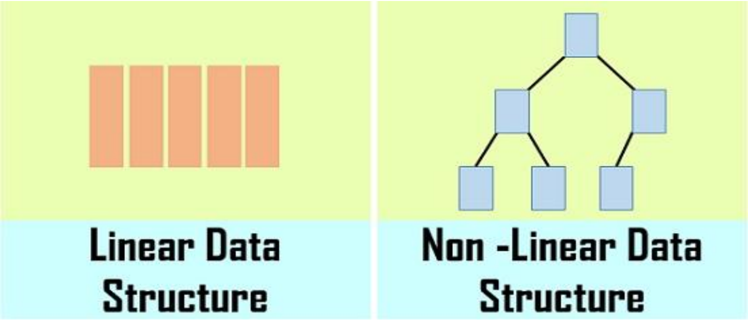
- Storage of data
- Organization of data
- Access to data
- Processing of data



Data Structures vs. File Structures

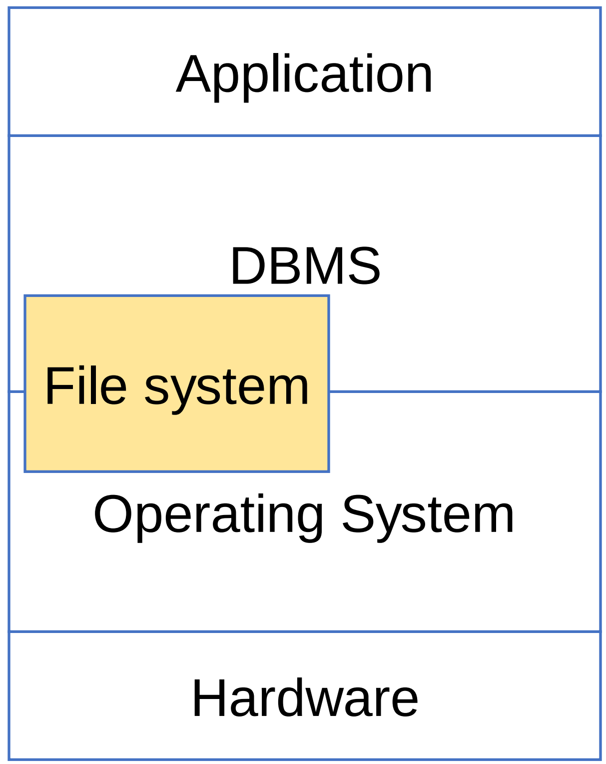
- Both involve:
  - Representation of Data

    +
  - Operations for accessing data



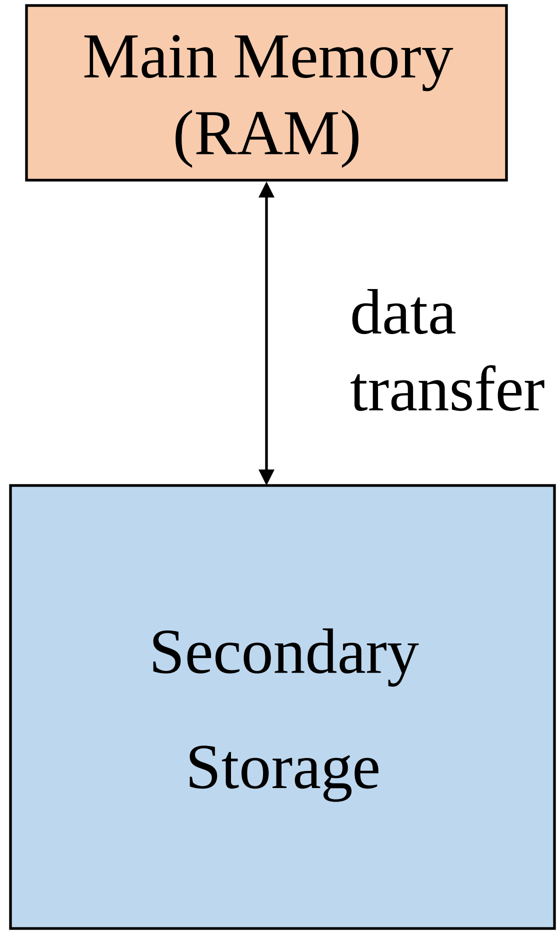
- Difference:
  - <u>Data structures</u>: deal with data in main memory
  - <u>File structures</u>: deal with data in secondary storage

Where do File Structures fit in Computer Science?



Computer Architecture

data is manipulated here



- Semiconductors
- Fast, expensive, volatile, small

data is stored here

- Disks, tape
- Slow, cheap, stable, large

Advantages

- Main memory is fast
- Secondary storage is big (because it is cheap)
- Secondary storage is stable (non-volatile) i.e. data is not lost during power failures

Disadvantages

- Main memory is small. Many databases are too large to fit in main memory (MM).
- Main memory is volatile, i.e. data is lost during power failures.
- Secondary storage is slow (10,000 times slower than MM)

How fast is main memory?

- Typical time for getting info from:

  Main memory: ~12 nanosec = 120×10^{-9} sec

  Magnetic disks: ~30 milisec = 30×10^{-3} sec

- An analogy keeping same time proportion as above:

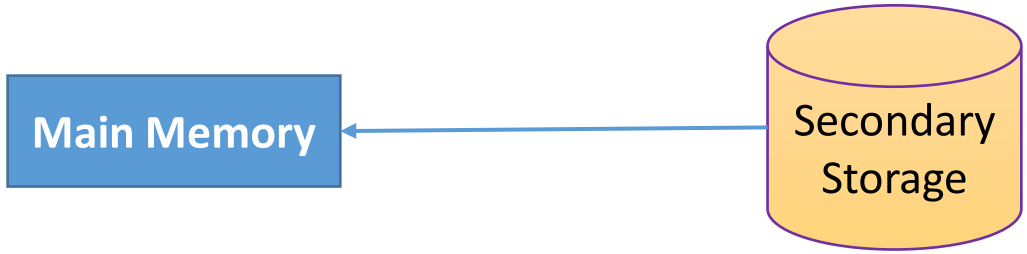
  Looking at the index of a book : 20 sec

  Versus

  Going to the library: 58 days

Focus

- Secondary storage (SS) provides reliable, long-term storage for large volumes of data
- At any given time, we are usually interested in only a small portion of the data
- This data is loaded temporarily into main memory, where it can be rapidly manipulated and processed.
- As our interests shift, data is transferred automatically between MM and SS, so the data we are focused on is always in MM.



Goal of the file structures

- Minimize the number of trips to the disk in order to get desired information.

- Grouping related information so that we are likely to get everything we need with only one trip to the disk.

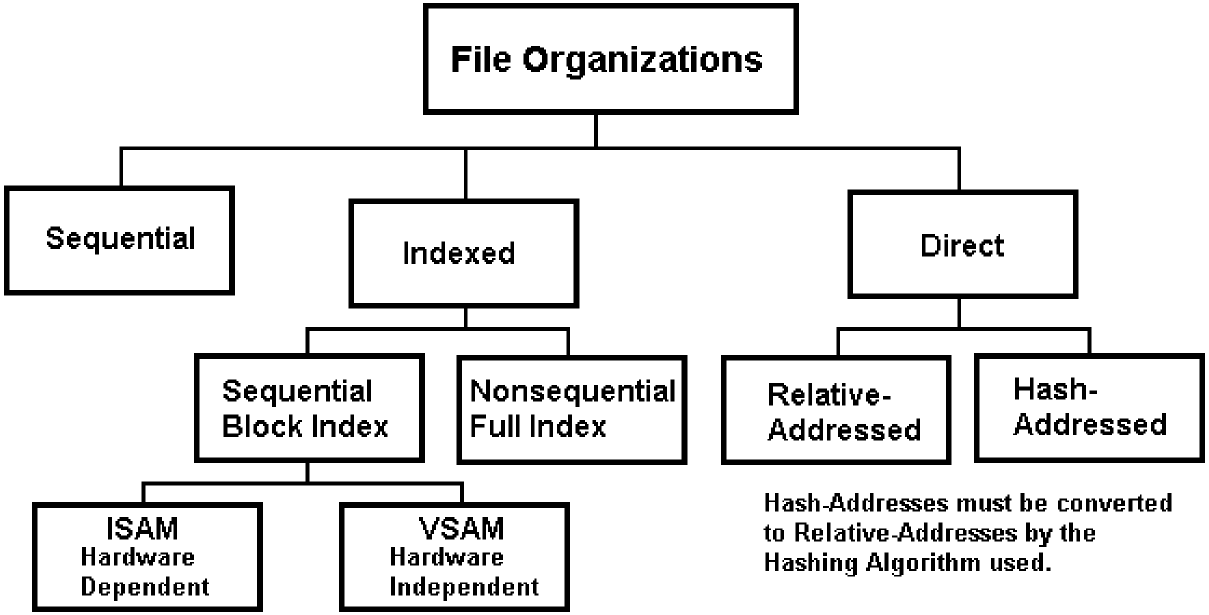
What is a file?

- A **file** is a sequence of records stored in binary format.
- Relative data is stored collectively in file formats.
- A disk drive is formatted into several blocks that can store records.
- Files are organized into one-dimensional arrays of bytes.

Stream of bytes: A file which is regarded as being without structure beyond separation into a sequential set of bytes.

File Organization

- There are two main ways a file can be organized:



- **Sequential Access** — Data is processed in sequence, one after another. To reach a particular item of data, all the data that proceeds it first must be read.
- **Random Access** — Data can be processed in any order. A particular item of data can be reached by going directly to it, without looking at any other data.

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 <u>program</u> (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 <u>operating system</u> 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.

Figure 8.6

Files

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

- The logical file has a logical name (a varibale) inside the program.
  - **In C :**

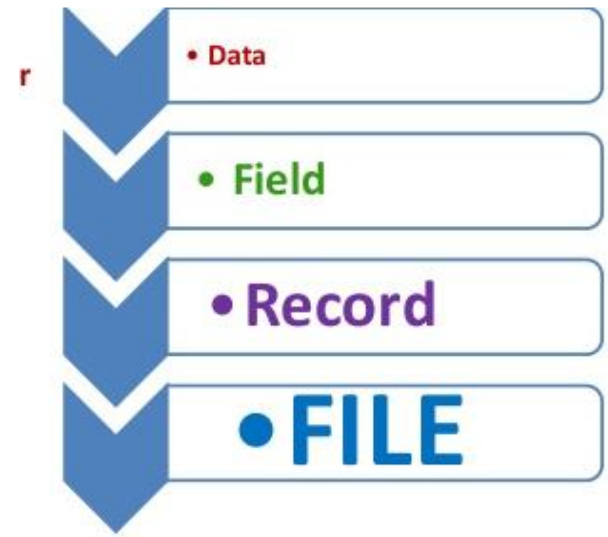
    ```
    FILE * outfile;
    ```

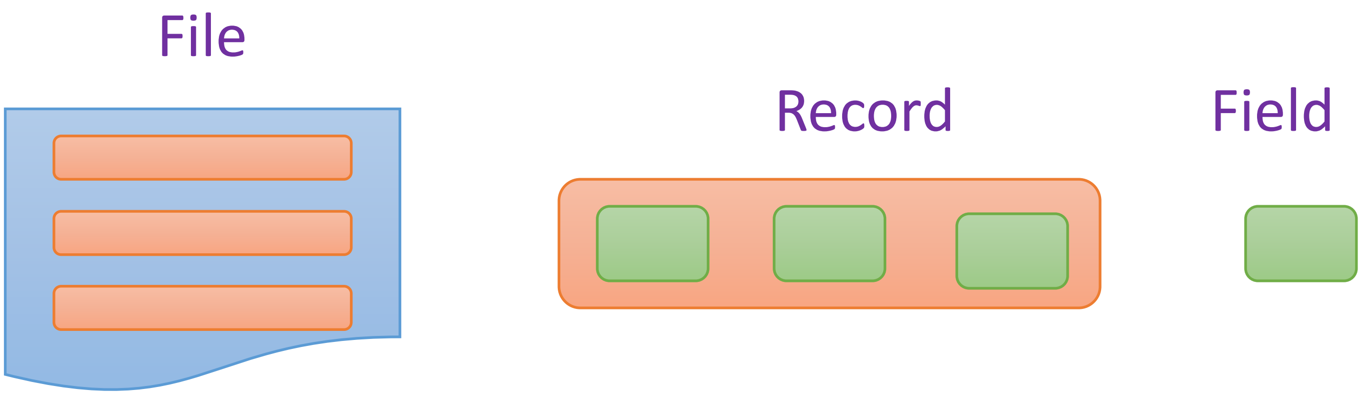
  - **In C++:**

    ```
    fstream outfile;
    ```

File Systems

- Data is not scattered on the disk.
- Instead, it is organized into **<u>files</u>**.
- Files are organized into **<u>records</u>**.
- Records are organized into **<u>fields</u>**.



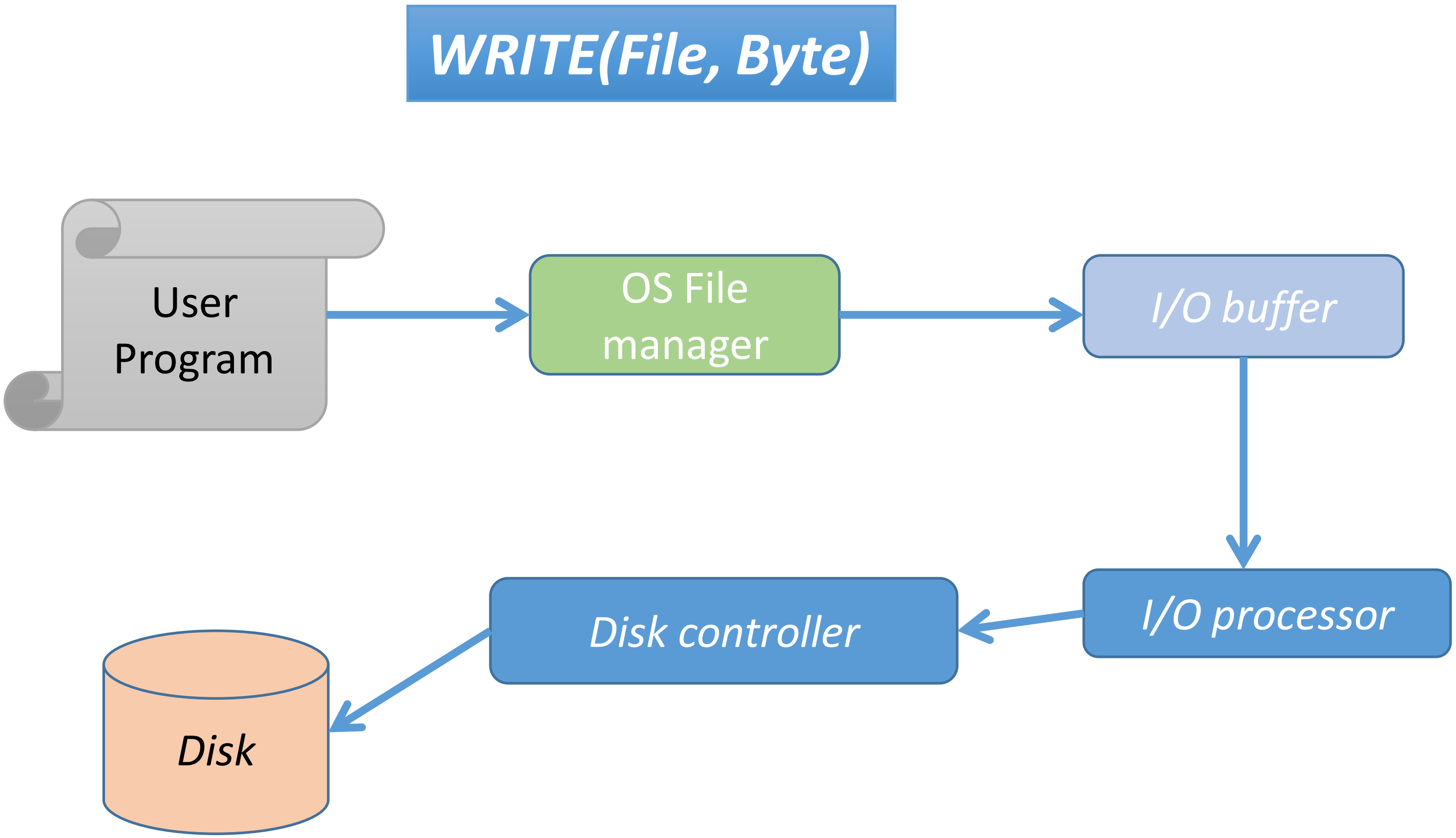


Example

- A student file may be a collection of student records, one record for each student
- Each student record may have several fields, such as
  - Name
  - Address
  - Student number
  - Gender
  - Age
  - GPA
- Typically, each record in a file has the same fields.

A Journey of a Byte

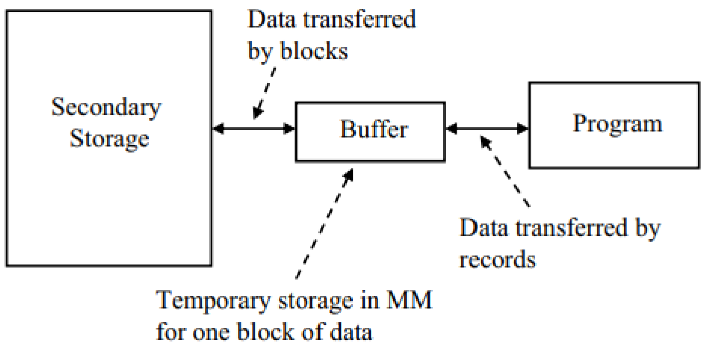
- What happens when a program writes a byte to a file on a disk?



Buffer

- Definition - the part of main memory available for storage of copies of disk blocks
- Program buffers vs. System I/O buffers
- Buffer manager
- subsystem responsible for the allocation for blocks
- goal:
  - minimize the number of disk access
  - utilize the memory space effectively

System I/O Buffer



Field Organization

- Field: The smallest logically meaningful unit of information in a file (not physical)
- Field structures (4 methods)
  - Fix the length of fields
  - Begin each field with a length indicator
  - Separate the fields with delimiters
  - Use a “Keyword = value” expression

Fixed Length Fields

Field 1	Field 2	Field 3	Field 4	Field 5

- Each record is divided into fields of correspondingly equal size.
- Different fields within a record have different sizes.
- Different records can have different length fields.
- Programs which access the record must know the field lengths.
- There is no external overhead for field separation.
- There may be internal fragmentation (unused space within fields.)

Ames	John	123	Maple	Stillwater	OK74075377-1808
Mason	Alan	90	Eastgate	Ada	OK74820

Delimited Variable Length Fields

Field 1	!	Field 2	!	Field 3	!	Field 4	!	Field 5	!

- The fields within a record are followed by a delimiting byte or series of bytes.
- Fields within a record can have different sizes.
- The delimiter cannot occur within the data. There is external overhead for field separation equal to the size of the delimiter per field.

Ames|John|123 Maple|Stillwater|OK|74075|377-1808|
Mason|Alan|90 Eastgate|Ada|OK|74820||

Length Prefixed Variable Length Fields

12	Field 1	4	Field 2	10	Field 3	8	Field 4	7	Field 5

- The fields within a record are prefixed by a length byte or bytes.
- Fields within a record can have different sizes.
- Different records can have different length fields.
- Programs which access the record must know the size and format of the length prefix.
- There is external overhead for field separation equal to the size of the length prefix per field.
- There should be no internal fragmentation (unused space within fields.)

Tagged Fields

- Tags, in the form "Keyword=Value", can be used in fields.
- Use of tags does not in itself allow separation of fields, which must be done with another method.
- Use of tags adds significant space overhead to the file.
- Use of tags does add flexibility to the file structure.
- Fields can be added without affecting the basic structure of the file.
- Tags can be useful when records have sparse fields - that is, when a significant number of the possible attributes are absent.

SURNAME=Ames|FIRSTNAME=John|STREET=123 Maple|...|ZIP=74075|PHONE=377-1808|#...

Record Organization

- Record: a set of fields that belong together
- Record organization(5 methods)
  - Make records a predictable number of bytes (Fixed-length records)
  - Make records a predictable number of fields
  - Begin each record with a length indicator
  - Use an index to keep track of addresses
  - Place a delimiter at the end of each record

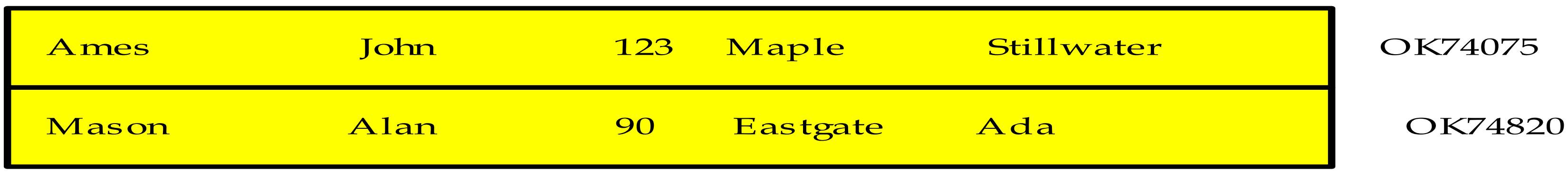
Fixed Length Records

Record 1	Record 2	Record 3	Record 4	Record 5

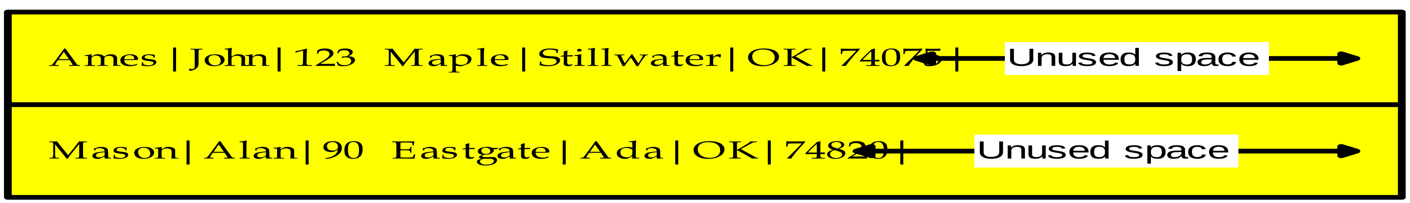
- The file is divided into records of equal size.
- There is no external overhead for record separation.
- There may be internal fragmentation (unused space within records.)
- There will be no external fragmentation (unused space outside of records) except for deleted records.
- Individual records can always be updated in place.

The method for organizing records

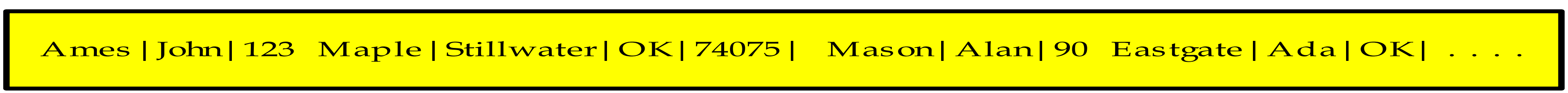
- Three ways of making the lengths of records constant and predictable
  - Fixed-length record w/ fixed-length fields
  - Fixed-length record w/ variable-length fields
  - Six fields per record



(a)

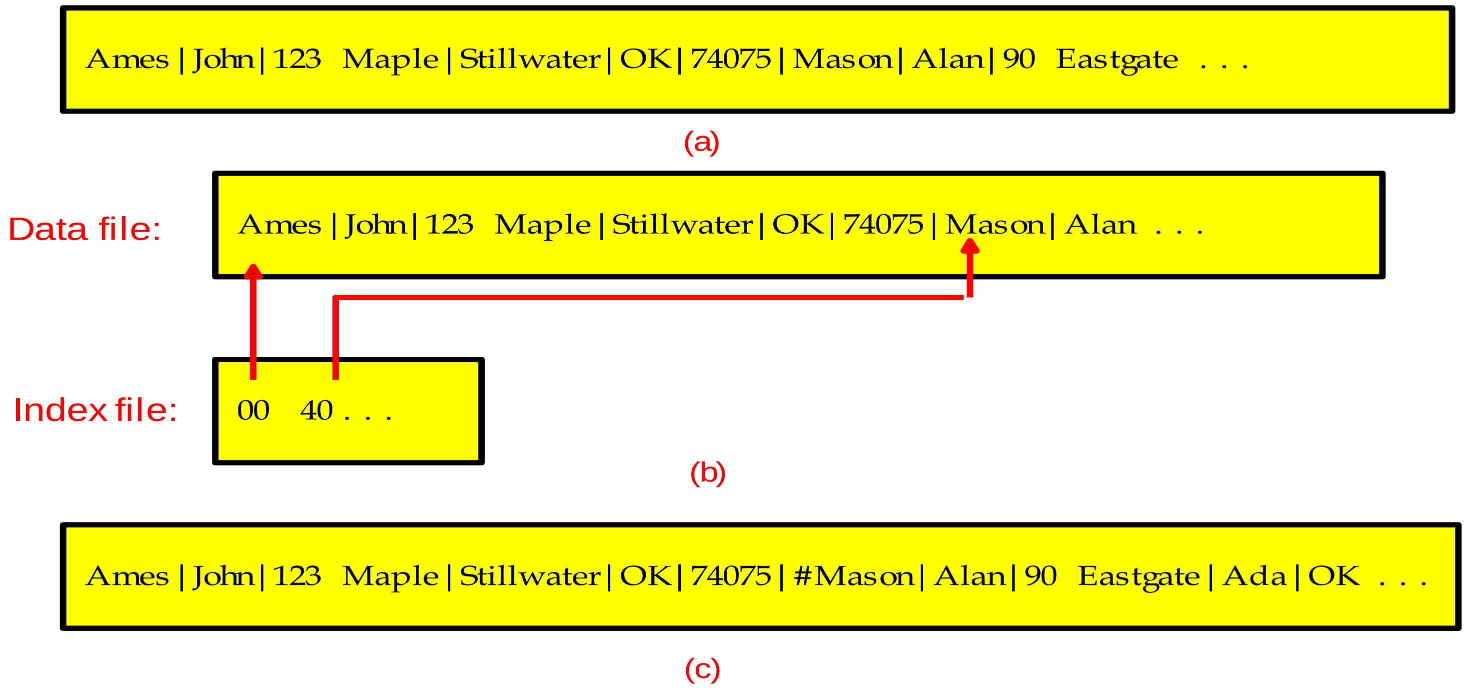


(b)

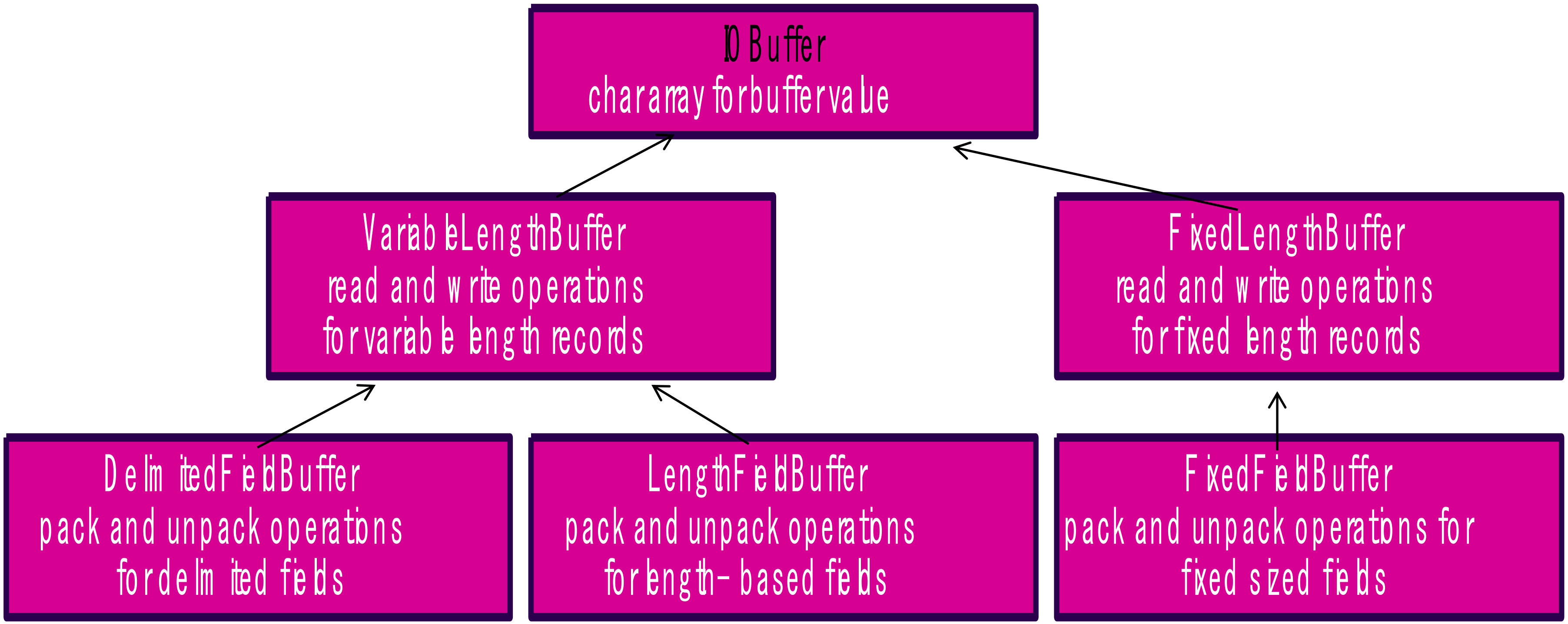


(c)

- Record structure for variable record
  - with a length indicator
  - using a index file
  - with delimiter(#)



Implementation



Types of Files

- Ordinary / Regular Files – data, text, executable binaries
- Directories – folders in Windows
- Special / Device Files – Character, Block

Streams and Access Modes

Standard UNIX streams

- stdin – file descriptor is 0
- stdout – file descriptor is 1
- stderr – file descriptor is 2

File Access Modes

- read
- write
- execute

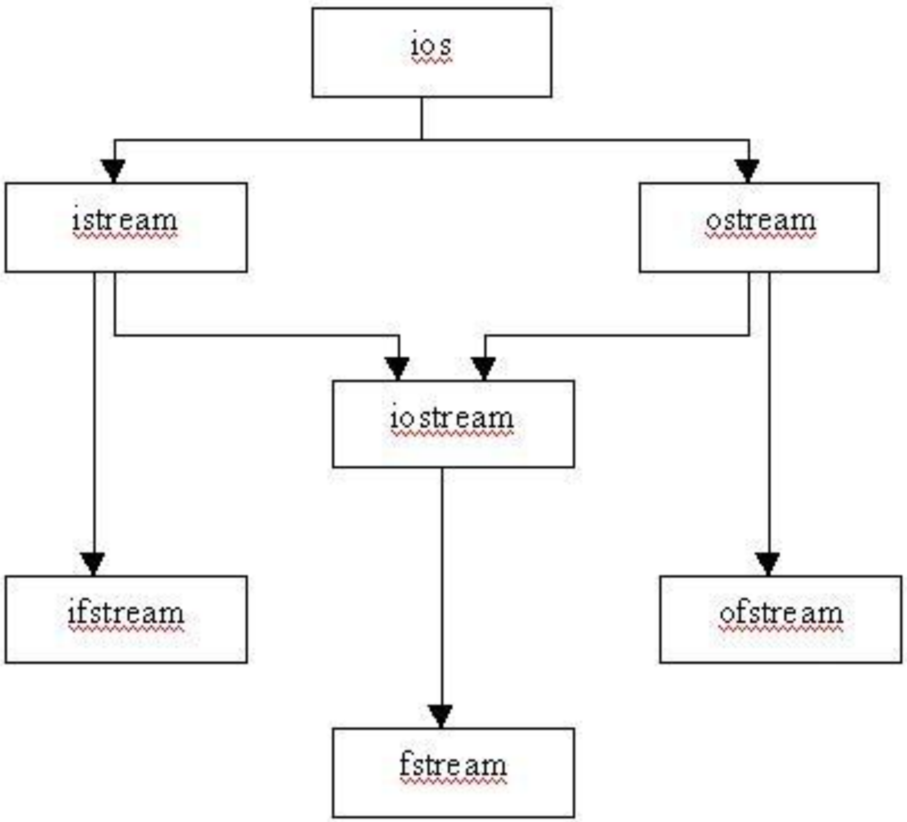
C++ Stream Classes

- ofstream : write on files
  - ofstream obj (const char* FileName, int FileMode);
- ifstream : read from files.
  - ifstream obj (const char* FileName, int FileMode);
- fstream : both read and write from/to files
  - fstream obj (const char* FileName, int FileMode);

FileName – a constant string that represents the file that you want to open.

FileMode – argument is a natural number that follows the table.

Hierarchy of Stream Classes



File Modes

- ios::app
- ios::ate
- ios::in
- ios::out
- ios::trunk
- ios::nocreate
- ios::noreplace

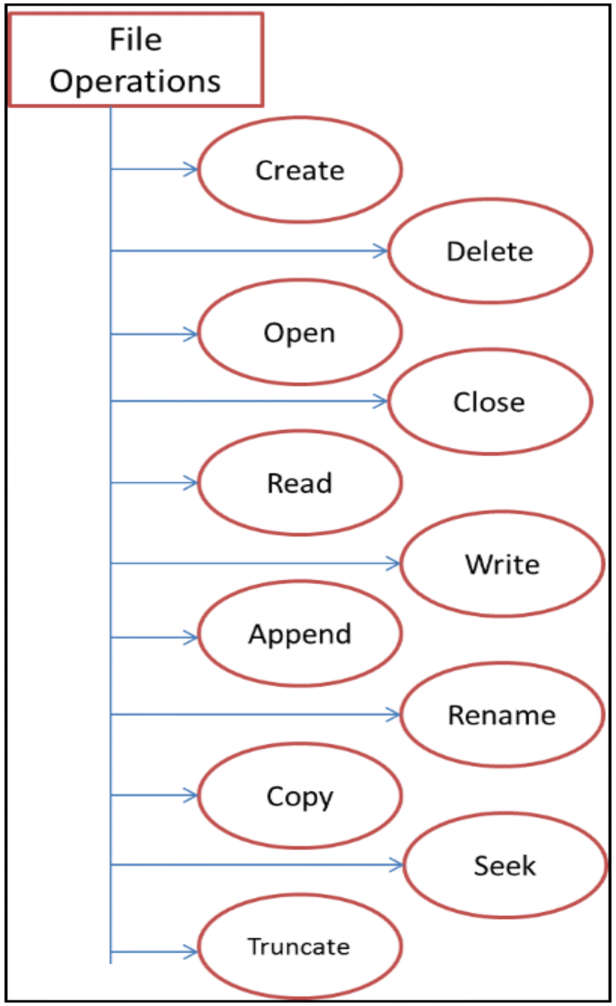
class	default mode parameter
ofstream	ios::out
ifstream	ios::in
fstream	ios::in | ios::out

Checking state flags

- eof() - Returns true if a file open for reading has reached the end.
- good() - Checks whether the stream is ready for input/output operations.
- 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.

Basic File Processing Operations

- Opening
- Closing
- Reading
- Writing
- Seeking



Creating /Opening a File

```
#include <iostream>
#include <conio>
#include <fstream>

using namespace std;
int main()
{
    fstream st;    // Step 1: Creating object of fstream class
    st.open("example.txt ",ios::out);  // Step 2: Creating new file
    if(!st)   // Step 3: Checking whether file exist
  {
     cout<<"File creation failed";
  }
  else
 {
    cout<<"New file created";
    st.close(); // Step 4: Closing file
  }
  getch();
  return 0;
}
```

Output:
[file example.txt]
New file created

Writing to a File

```
#include <iostream>
#include <fstream>

using namespace std;
int main ()
{
   ofstream myfile("example.txt");
   myfile << “Learning file operations.\n";
   myfile.close();
   return 0;
}
```

Output:
[file example.txt]
Learning file operations

Reading from a File

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main ()  {
     string line;
     ifstream myfile ("example.txt");
    if (myfile.is_open()) {
       while ( myfile.good() )  {
           getline (myfile,line);
           cout << line << endl;
       }
    myfile.close();   }
   else   cout << "Unable to open file";
   return 0;
 }
```

Output:
Learning file operations.

Get and Put stream Pointers

- get pointer - ifstream pointer that points to the element to be read in the next input operation.
  - tellg, seekg

- put pointer - ofstream pointer that points to the location where the next element has to be written.
  - tellp, seekp

Program using stream pointers

```
// obtaining file size
#include <iostream>
#include <fstream>
int main ()
{
    long begin,end;
    ifstream myfile ("example.txt");
    begin = myfile.tellg();
    myfile.seekg (0, ios::end);
    end = myfile.tellg();
    myfile.close();
    cout << "size is: " << (end-begin) << " bytes.\n";
    return 0;
}
```

Output:
size is: 40 bytes

UNIX

- An operating system developed in the 1970s.
- A stable, multi-user, multi-tasking system for servers, desktops and laptops.
- Widely used in modern servers, workstations, and mobile devices.

Characteristics of UNIX as an OS

- Less resource intensive
- Over 49 years old
- Large number of applications
- Multi-user and Multi-tasking
- Free applications and even a free OS
- Internet development

Flavours of UNIX

- Linux
- Solaris
- Mac OS X
- Ultrix
- AIX
- BSD unix
- IRIX

Inter-process communication (IPC)

- Mechanisms an operating system provides to allow the processes to manage shared data.

- Typically, applications can use IPC, categorized as clients and servers.

- Methods for doing IPC are divided into categories which vary based on software requirements, such as performance and modularity requirements, and system circumstances, such as network bandwidth and latency.

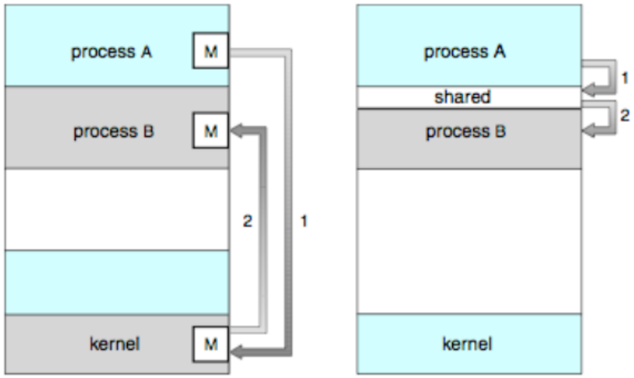
UNIX IPC mechanisms

- Signals
- Pipes
  - Unnamed Pipes
  - Named Pipes of FIFOs
- Message Queues
- Shared Memory
- Remote Procedure Calls (RPC)
- Sockets
- Semaphores

Message Passing vs. Shared-Memory

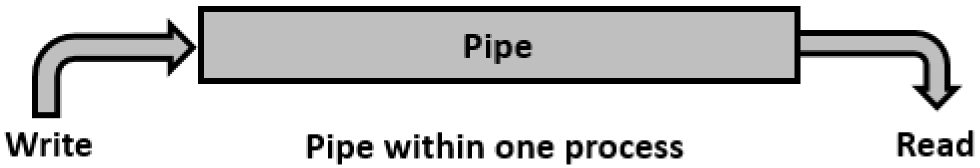
- Message Passing
  - ✓ Why good ? All sharing is explicit -> less chance for error.
  - ✓ Why bad ? Overhead . Data copying, cross protection domains.
- Shared Memory
  - ✓ Why good? Performance. Set up shared memory once, then access w/o crossing protection domains.
  - ✓ Why bad ? Things change behind your back -> error prone.

Message Passing **Shared Memory**



Pipe : Communication Buffer

- A communication medium between two or more related or interrelated processes.
- It can be either within one process or a communication between the child and the parent processes.
- Communication can also be multi-level such as communication between the parent, the child and the grand-child, etc.
- Communication is achieved by one process writing into the pipe and other reading from the pipe.
- To achieve the pipe system call, create two files, one to write into the file and another to read from the file.



- Create a pipe: system call pipe()

  #include <unistd.h>

  int pipe(int fildes[2]);

- Create a **unidirectional communication buffer with two file descriptors**: fildes[0] for read and fildes[1] for write.
- Data write and read on a **first-in-first-out basis.**
- **No external or permanent name, and can only be accessed through two file descriptors.**

fork() in C

- Fork system call is used to create a new process, which is called ***child process***, which runs concurrently with process (which process called system call fork) and this process is called ***parent process***.
- After a new child process created, both processes will execute the next instruction following the fork() system call.
- A child process uses the same pc(program counter), same CPU registers, same open files which use in the parent process.
- It takes no parameters and returns an integer value.
- Different values returned by fork():

  Negative Value: creation of a child process was unsuccessful.

  Zero: Returned to the newly created child process.

  Positive value: Returned to parent or caller. The value contains process ID of newly created child process.

fork() in C

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    // make two process which run same
    // program after this instruction
    fork();
    printf("Hello world!\n");
    return 0;
}
```

Output:
Hello world!
Hello world!

```
fork ();   // Line 1
fork ();   // Line 2
fork ();   // Line 3

       L1       // There will be 1 child process
    /     \     // created by line 1.
  L2       L2   // There will be 2 child processes
 /  \     /  \  //  created by line 2
L3  L3  L3  L3  // There will be 4 child processes
                // created by line 3
```

Thank You!

Questions & Answers

Santosh Pattar

Research Scholar,
Department of Computer Science and Engineering,
University Visvesvaraya College of Engineering.
t: +91-8861135707
e: santoshpattar01@gmail.com