

Hard Disk Architecture

1 millisecond 1 msec = $0.001 \left(\frac{1}{1,000} \right)$ second

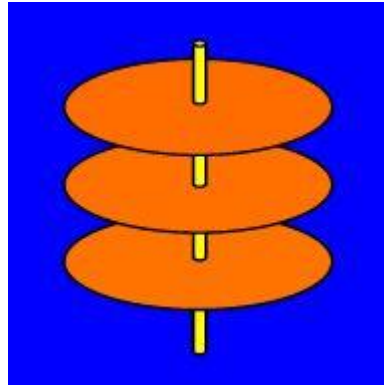
1 microsecond 1 μ sec = $0.000\ 001 \left(\frac{1}{1,000,000} \right)$ second

1 nanosecond 1 nsec = $0.000\ 000\ 001 \left(\frac{1}{1,000,000,000} \right)$ second

Disk drives have one or more platters.

Each platter has one or two **sides**.

Each side has a read/write head.



Each side is divided up into concentric tracks.

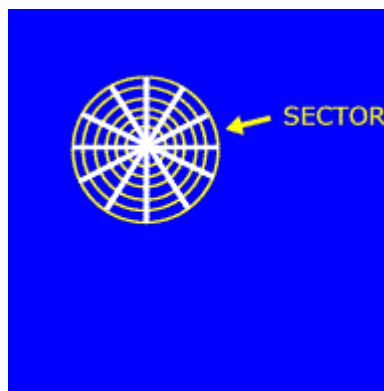
The platters and their read/write heads are stacked, hence all read/write heads are accessing the same track# on their sides.

The set of tracks (one per side) that are accessed simultaneously is called a **cylinder**.

All data on a cylinder can be accessed without moving the heads.

Each track is divided up into **sectors**.

On a PC, the sector size is 512 bytes.



A sector is the smallest addressable part of a hard disk. It is the smallest quantity of data that can be transferred between RAM and the hard disk.

If the program writes exactly one byte, your operating system must do the following:

1. Read the sector that contains the byte into a 512-byte buffer in RAM.
2. Copy the byte into its correct position in the buffer.
3. Write the 512-byte buffer back to the disk sector.

The operating system addresses each sector by three numbers

1. Cylinder #
2. Head # within cylinder
3. Sector # within track

Access time is the total time to move data between the hard disk and RAM

Access time is the sum of:

The seek time is the time for the hard drive's read/write heads to move to the requested cylinder.

The latency is the time for the requested sector to rotate under a read/write head.

The transfer time. how long it takes for data to move between the hard disk and RAM.

Seagate Baracuda 7200.11

Random read seek time = 8.5 msec

Random write seek time = 9.5 msec

Average latency = 4 msec (Rotation speed = 7,200 revolutions / minute)

Rotation speed of 7,200 revolutions / minute

=> 120 revolutions / second

=> 8 msec for one rotation

=> 4 msec average to rotate a sector under the read/write head

The transfer rate is the how fast data moves between the hard disk and RAM.

The transfer rate cannot be greater than the rate the data is coming off the disk.

If there are 63 sectors per track, and the disk rotates at 7200 rpm, then at most 3.9MB can be transferred in one second.

$63 * 512$ bytes can transfer in $\frac{1}{7,200}$ minutes

$32,256$ bytes can transfer in $\frac{60}{7,200}$ seconds

$32,256$ bytes can transfer in $\frac{1}{120}$ seconds

$120 * 32,256$ bytes can transfer in 1 second

$3,870,720$ bytes per second

=> 0.13 msec for one sector (negligible relative to the latency and seek times)

=> 8 msec for all 63 sectors

Access time to read 1 sector is 12.5 msec (= 8.5 seek + 4msec latency)

RAM

DDR3-1600

Transfer rate 1600 Million data transfers per second x 8 bytes

=> 12.8 GB / second

=> 0.08 nsec to transfer 1 byte

| RAM | Hard Disk |
|------------|--------------------------|
| 0.08 nsec | 12.5 msec |
| | 144 million times slower |

Analogy: 5 seconds to fetch a pencil from a desk drawer (RAM), => 23 years to fetch the pencil from the cupboard (hard disk)

DISKS ARE SLOW.

Advantages of hard disks over RAM

| |
|---|
| 1. Lower cost per byte. |
| 2. Moreover, they can store data when the computer is turned off. |

A **file structure** is an ADT where the data is stored on a storage device such as a hard disk.

Like primary memory ADTs, a file structure is a combination of :

- [1] data
- [2] methods

Typical methods are:

- read a record
- write a record
- search for a record, via a search key
- traverse the records in ascending key order

How a file structure is implemented, can make a tremendous difference in the speed of an application program.

Sequential access.

Accessing records one after the other, starting with the first record, then moving to the next record, etc

Direct access.

Accessing a record via a search key.

Sequential Access vs Random Access

Assuming:

| | |
|-----------------------------|--|
| Average seek | 9 msec |
| Average latency | 4 msec |
| Transfer time for one track | 8 msec |
| sectors / track | 64 |
| sector size | 512 bytes |
| Each logical record | 128 bytes (4 logical records per sector) |
| cluster size | 8 sectors (32 logical records per cluster) (8 clusters per track) |

=> transfer time for one sector 8 / 64 msec

=> transfer time for one cluster 8 / 8 msec

Assuming that a sequential file is extremely fragmented - the clusters are scattered all over the disk.

The time to transfer a sequential file of 256 logical records
= time to transfer 8 = 256 / 32 clusters (since there are 32 logical records / cluster)
= 8 * (9msec + 4msec + 8/8 msec)
= 112 msec

The time to access 256 records randomly
= 256 * (9msec + 4msec + 8/64 msec)
= 3360 msec

Roughly speaking, if there are 32 logical records per cluster we can expect random access to be 32 times SLOWER.

AND, if the sequential file is not extremely fragmented it will perform EVEN BETTER.

If a set of operations to be performed on a file involves more than 5% to 10% of the records, use sequential access!

History of the development of file structures

Binary tree in RAM

↓

AVL tree (a balanced tree) in RAM

The tree is balanced. The tree never becomes lopsided.
The time to locate an item is $\log_2 N$

↓

B-tree on a hard disk

The tree is balanced.
Each node has multiple child nodes ($m \gg 2$)
Each node fills up one or more sectors.
The time to locate an item is $\log_k N$ (k is the number of children per node)
=> **Only a few disk reads to locate a record**

↓

B⁺ tree

The leaf nodes are linked together to permit fast sequential access.

====

Hashing

Search keys are transformed by a function (the hashing function) to record addresses.

↓

Extendible dynamic hashing

Hashing that adjusts the hashing function as the number of records grows.
Can retrieve data with 1 or 2 disk accesses, no matter how big the file becomes.

Physical files and logical files

Consider a program that prints out the contents of a text file A:\EMPLOYEE.TXT

```
OPEN "a:\\Employee.txt"
LOOP
|   READ line FROM "a:\\Employee.txt"
|   IF eof( "a:\\Employee.txt" )
|       BREAK
|   PRINT line
CLOSE "a:\\Employee.txt"
```

One problem with this program is that if we want to print out the contents of "c:\\Client.txt", we have to edit the program and replace "a:\\Employee.txt" with "c:\\Client.txt" 4 times.

We can solve this problem by coding:

```
inputFile = "a:\\Employee.txt"

OPEN inputFile
LOOP
|   READ line FROM inputFile
|   IF eof( inputFile )
|       BREAK
|   PRINT line
CLOSE inputFile
```

a:\Employee.txt and c:\Client.txt are the **physical names** of files. i.e. the name of the file on the hard disk.

Within the program, we refer to the file using a **logical name** inputFile.

When the program starts running, we bind the logical filename to a physical filename. We can use the same program to print different text files by reading in the physical filename at runtime.

```
cin >> inputFile
```

```
OPEN inputFile
LOOP
|   READ line FROM inputFile
|   IF eof( inputFile )
|       BREAK
|   PRINT line
CLOSE inputFile
```

Programming languages don't use string variables for the logical filenames, instead they actually use other types:

| | |
|-----|---|
| C | <code>int</code> - for low-level file i/o <code>FILE*</code> - for high-level file i/o |
| C++ | <code>ifstream</code> - for input files, <code>ofstream</code> - for output files, <code>fstream</code> - for i/o files |

In C++, the physical filename can be bound to the logical filename either when the logical file is instantiated, or when the file is opened.

```
ifstream fIn1( "a:\\Employee.txt" );
```

```
ifstream fIn2;  
fIn2.open( "a:\\Employee.txt", ... );
```

C++ automatically creates some logical files for every program:

| logical filename | initially bound to |
|-------------------------|---------------------------|
| <code>cin</code> | the keyboard |
| <code>cout</code> | the screen |
| <code>cerr</code> | the screen |

The programmer can bind any logical filename to the keyboard and screen, by using the special "con" physical filename.

```
ofstream fOut;  
fOut.open( "con", ... ); // output goes to the screen
```

Opening files

When the programmer wishes to work with a file, he can either:

- [1] Create a new file and open it, or
- [2] Open an existing file.

C++ uses the open method for both creating and opening a file.

Open a file for input

```
const char EMPLOYEE_FILENAME[] = "A:\\Employee.txt";  
fstream file1;  
file1.open(EMPLOYEE_FILENAME, ios_base::in );
```

| | If the file exists | If the file does not exist |
|----------------------|------------------------------|-----------------------------------|
| | The file is opened for input | (nothing) |
| file1.fail() returns | false | true |

Open a file for input and output

```
const char EMPLOYEE_FILENAME[] = "A:\\Employee.txt";  
fstream file1;  
file1.open(EMPLOYEE_FILENAME, ios_base::in | ios_base::out );
```

| | If the file exists | If the file does not exist |
|----------------------|------------------------------|-----------------------------------|
| | The file is opened for input | (nothing) |
| file1.fail() returns | false | true |

Create a file for output

```
const char EMPLOYEE_FILENAME[] = "A:\\Employee.txt";  
fstream file1;  
file1.open(EMPLOYEE_FILENAME, ios_base::out | ios_base::trunc );
```

| | If the file exists | If the file does not exist |
|----------------------|---|--|
| | The file is opened output. The previous contents is destroyed. | file is created and opened for output. |
| file1.fail() returns | false | false |

WARNING

The calls of `open(...)` do NOT automatically clear the error flags (`badbit`, `eofbit`, and `failbit`) before each call.

```
fail() returns true if either badbit = true or failbit == true
eof() returns true if eofbit == true
```

C++ will accumulate errors unless you explicitly **clear()** them.

Teachers recommendation: **Before calling `fail()`, always call `clear()`.**

Closing files

Buffered i/o

To minimize the number of disk accesses, an operating system will temporarily save in RAM data that is being written to a file.

Closing a file ensures that all buffered data is written to the file.

In C++ code:

```
file.close();
```

Reading and Writing binary files

Get pointer

For each input file, C++ maintains a **get pointer**.

Initially the file's get pointer is positioned at the file's first byte.

After the first byte is read from the file, the get pointer is advanced to the second byte, etc.

The file's get pointer always is positioned at the next byte to be read.

After reading from the file's last byte, the get pointer is positioned at the end of the file.

Put pointer

Similarly, for each output file a **put pointer**.

The file's put pointer is always positioned at where the next byte will be written in the file.

A file opened for input and output has independent get and put pointers.

Seeking

The programmer can move a file's put pointer to any byte position in a file.

```
file.seekp ( byte_number );
```

The next read will be written to the put pointer's new position.

The bytes are numbered from 0.

```
E..g file.seekp( 1000 ); // move the put pointer to the 1000th byte in the file.
```

Similarly, the programmer can move a file's get pointer to any byte position in a hard disk file.

```
file.seekg ( byte_number );
```

The next read will be read from the put pointer's new position.

file.write(...)

The write method of a `fstream` object has two parameters:

```
write( /* in */ const char* buffer, /* in */ int n );
```

A pointer to a constant char array

The char array contains the bytes of data to be written.

In practice, the actual argument isn't usually a `const char` array, so the argument must be cast to `const char *`

The number of bytes to be written.

```
struct Employee
{
    ...
};

Employee employee = { ... };

file.write( (const char*) &employee, sizeof employee );
```

file.read(...)

The read method of a `fstream` object has two parameters:

```
read( /* out */ const char* buffer, /* in */ int n );
```

A pointer to a char array.

The bytes of data will be read into this array.

In practice, the actual argument isn't usually a char array, so the argument must be cast to `char *`

The number of bytes to be read.

The dimension of the char array must be greater than or equal to the number of bytes to be read.

```
file.read( (char*) &employee, sizeof employee );
```

```

#include <fstream.h>
#include <assert.h>

const char FILESPEC[] = "employee.dat";
const int  MAX_LENGTH_NAME = 20;

struct Employee
{
    int    id;
    char   lastname[MAX_LENGTH_NAME+1];
    char   firstname[MAX_LENGTH_NAME+1];
    double salary;
};

void main()
{
    fstream file;

    Employee employees[] =
    {
        { 0, "Jones", "Jim", 30000 },
        { 1, "Smith", "Sam", 35000 },
        { 2, "Tremblay", "Tina", 40000 }
    };

    file.clear();
    file.open( FILESPEC, ios::binary | ios::in | ios::out | ios::trunc );
    if ( file.fail() )
    {
        cerr << "Can't create " << FILESPEC << endl;
        return;
    }

    file.seekp( 2 * sizeof Employee );
    file.write( (const char*) &employees[2], sizeof Employee );

    file.seekp( 0 * sizeof Employee );
    file.write( (const char*) &employees[0], sizeof Employee );

    file.seekp( 1 * sizeof Employee );
    file.write( (const char*) &employees[1], sizeof Employee );

    Employee employee;

    file.seekp( 0 * sizeof Employee );
    file.read( (char*) &employee, sizeof Employee );
    assert ( employee.id == employees[0].id );

    file.seekp( 2 * sizeof Employee );
    file.read( (char*) &employee, sizeof Employee );
    assert ( employee.id == employees[2].id );

    file.seekp( 1 * sizeof Employee );
    file.read( (char*) &employee, sizeof Employee );
    assert ( employee.id == employees[1].id );
}

```

RandomFile file structure

```
bool Create (  
    /* in */ const char filespec[],  
    /* in */ int      sizeRecord );
```

Create a new file.

The file is opened for binary input/output.

The file has fixed sized records.

```
bool RandomFile::Create(  
    /* in */ const char filespec[],  
    /* in */ int      sizeRecord_ )  
{  
    if ( file.is_open() )  
        file.close();  
  
    sizeRecord = sizeRecord_;  
  
    file.clear();  
    file.open(  
        filespec,  
        ios::binary | ios::in | ios::out | ios::trunc );  
    return ! file.fail();  
}
```

```
bool Open (  
    /* in */ const char filespec[],  
    /* in */ int      sizeRecord );
```

Open an existing file.

The file is opened for binary input/output.

sizeRecord must be the same record size as specified when the file was created.

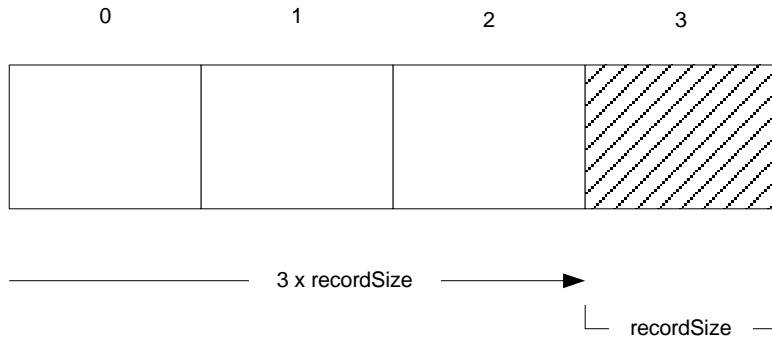
```
bool RandomFile::Open(  
    /* in */ const char filespec[],  
    /* in */ int      sizeRecord_ )  
{  
    if (file.is_open() )  
        file.close();  
  
    sizeRecord = sizeRecord_;  
  
    file.clear();  
    file.open( filespec,  
        ios::binary | ios::in | ios::out );  
    return ! file.fail();  
}
```

```
bool Close ();
```

```
bool RandomFile::Close()  
{  
    if ( file.is_open() )  
        file.close();  
    return true;  
}
```

```
bool Write(  
    /* in */ int recordNr,  
    /* in */ char record[] )
```

Seek the **put pointer** to byte offset `recordNr * recordSize`.



Write `recordSize` bytes from `record[]`.

```
bool RandomFile::Write(  
    /* in */ int recordNr,  
    /* in */ char record[] )  
{  
    if ( ! file.is_open() )  
        return false;  
  
    file.clear();  
    file.seekp( (long)recordNr * sizeRecord );  
    if ( file.fail() )  
        return false;  
  
    file.write( record, sizeRecord );  
  
    file.clear();  
    if ( file.fail() )  
        return false;  
  
    return true;  
}
```

```
bool Read(
    /* in */ int recordNr,
    /* out */ char record[] )
```

Seek the **get pointer** to byte offset `recordNr * recordSize`.

Read `recordSize` bytes into `record[]`

```
bool RandomFile::Read(
    /* in */ int recordNr,
    /* out */ char record[] )
{
    if ( ! file.is_open() )
        return false;

    file.clear();
    file.seekg( (long)recordNr * sizeRecord );
    if ( file.fail() )
        return false;

    file.clear();
    file.read( record, sizeRecord );
    if ( file.fail() )
        return false;

    return true;
}
```

The buffer **MUST** be big enough to receive `recordSize` bytes.

```
bool IsOpen()
```

Determine if the file is open.

```
bool RandomFile::IsOpen()
{
    return ( file.is_open() != 0 );
}
```

ManagedRandomFile file structure

We wish to manage a file of fixed sized records.

Applications can allocate records.

Application programs can read and write to their allocated records.

When the application program no longer needs a record it can free it.

| application | file structure |
|--|--------------------------------|
| allocate a record | --> |
| | <-- ok, you can have record #3 |
| write record #3 | --> |
| read record #3 | --> |
| free record #3 <i>(I no longer need it)</i> | --> |

When the application frees a record, the file structure's implementation pushes the record on a linked list of free records.

When the application allocates a record, the file structure's implementation will first check to see if there is a free record on the free list.

If the free list is not empty, a record is popped and allocated to the application.
The record # is returned to the application.

If the free list is empty, the file is extended by one new record (by writing a new record at the end of the file).

The record# of the new record is returned to the application.

The file structure implementation reserves record #0 to store

the record size
the number of allocated records
the record# of the record on the top of the free list
the number of records in the underlying random file.

The file structure implementation minimizes the number of reads and writes of record #0 by caching it in RAM.

When the file is opened, record #0 is read into RAM.

When the file is closed, record #0 is written to disk.

The file structure implementation also allows the application to store additional data in record #0.