Files

http://www.savie-crp.ca/wp-content/uploads/2015/08/archives.jpg

Character Operations

- Comparing a character c and an element in a string s1

```
if ( s1[i] == 'c' ) …
```

Single quotes

- Changing a single character in an array

```
s1[i] = 'c';
```

Note: You do not need the **single** quotes if the character is a variable.

```
char letter;
char str[10];
letter = 'm';
str[3] = letter;
```

ASCII

- **ASCII** - American Standard Code for Information Interchange
- International counterpart is ISO646.
- It is a 7-bit code (128 values).
- Uses one byte (8 bits) to store a character.

ASCII

- Upper and lower case differ by one bit.
- First bit is always 0 in pure ASCII.

Character	Decimal	Bits	Hex
A	65	0100 0001	41
Z	90	0101 1010	5A
a	97	0110 0001	61
z	122	0111 1010	7A
<space>	32		20
<line feed \n>	10		0A
<tab \t>	9		09
0	48		30
9	57		39
.	46		2E

ASCII

- Upper and lower case differ by one bit.
- First bit is always 0 in pure ASCII.

'a' is a 1 byte number

We think of it as *a* ,
C thinks of it as 97

Character	Decimal	Bits	Hex
A	65	0100 0001	41
Z	90	0101 1010	5A
a	97	0110 0001	61
z	122	0111 1010	7A
<space>	32		20
<line feed \n>	10		0A
<tab \t>	9		09
0	48		30
9	57		39
.	46		2E

ASCII

- Upper and lower case differ by one bit.
- First bit is always 0 in pure ASCII.

Character	Decimal	Bits	Hex
A	65	0100 0001	41
Z	90	0101 1010	5A
a	97	0110 0001	61
z	122	0111 1010	7A
<space>	32		20
<line feed \n>	10		0A
<tab \t>	9		09
0	48		30
9	57		39
	46		2E

'a' is a 1 byte number

We think of it as *a* ,
C thinks of it as 97

'a' < 'b' same as
97 < 98 true

ASCII

- Upper and lower case differ by one bit.
- First bit is always 0 in pure ASCII.

Character	Decimal	Bits	Hex
A	65	0100 0001	41
Z	90	0101 1010	5A
a	97	0110 0001	61
z	122	0111 1010	7A
<space>	32		20
<line feed \n>	10		0A
<tab \t>	9		09
0	48		30
9	57		39
	46		2E

'a' is a 1 byte number

We think of it as *a* ,
C thinks of it as 97

'a' > 'A' same as
97 > 65 true

ASCII

- Upper and lower case differ by one bit.
- First bit is always 0 in pure ASCII.

Character	Decimal	Bits	Hex
A	65	0100 0001	41
Z	90	0101 1010	5A
a	97	0110 0001	61
z	122	0111 1010	7A
<space>	32		20
<line feed \n>	10		0A
<tab \t>	9		09
0	48		30
9	57		39
	46		2E

'a' is a 1 byte number

We think of it as *a* ,
C thinks of it as 97

'C' + ('a' - 'A') == 'c' same as
67 + 32 == 99 true

ASCII

- Upper and lower case differ by one bit.
- First bit is always 0 in pure ASCII.

Character	Decimal	Bits	Hex
A	65	0100 0001	41
Z	90	0101 1010	5A
a	97	0110 0001	61
z	122	0111 1010	7A
<space>	32		20
<line feed \n>	10		0A
<tab \t>	9		09
0	48		30
9	57		39
	46		2E

'a' is a 1 byte number

We think of it as *a* ,
C thinks of it as 97

char c = (char) 97;
c == 'a' true

Unicode

- Other character encodings exist for characters that do not appear in the ASCII set, *i.e.* international character sets.
- **Unicode** Text Format (UTF) is supported by Java (and other languages) and uses between 1 and 4 bytes to store a character. An ASCII character is a 1-byte UTF character .
- *From Wikipedia*: Unicode contains 137,994 characters (consisting of 137,766 graphic characters, 163 format characters and 65 control characters) covering 150 modern and historic scripts, as well as multiple symbol sets and emoji.

Text File

- A text file is a **sequence** of **ASCII** characters stored on a disk.
- Each character is stored as **one** byte.
- Similar to strings in C.

Text File Format Example

- Store the string “cost 2.98” in a text file. Each character is 1 byte.
- When the text file is read every byte is read into a char.

c	o	s	t		2	.	9	8
01100011	*01101111*	*01110011*	*01110100*	*00100000*	*00110010*	*00101110*	*00111001*	*00111000*

- Note: numbers are stored as character strings so 2.98 needs 4 bytes or characters - it is not stored as a float! The integer 2 would be stored in 1 byte.

Text Files

- Text editors are simply applications that read / display / write text files.
- The only difference between **text** and **binary** files is the type of data stored in the files.
- **Text** files contain only **ASCII** characters.
- Both types are read from and written to using the same techniques.

Binary Files

- **Binary** files contain **non-ASCII** characters.
  - **Text** is still stored using **ASCII**
  - **Numbers** are stored in their **binary** format
  - **Structures** can also be stored

Binary Files

- Store the integer number 10383 in a binary file.
- An integer is stored as **4** bytes in a binary format.

  10383 = 00000000 00000000 00010100 10001111

- If it was stored in a text file it would occupy **5** bytes.
- Binary representations can take less space but cannot be edited by a text editor.

Interpreting Data in Files

- Text and binary data can be mixed in the same file.
- Unix/Linux does not care what is in the file, it is all dependant on how it is interpreted by a program.
- When reading a file you must read it in the same order in which the file was written or the data will be corrupted.
- The system considers the file to be a **collection of bytes**.

0000 0000 0001 0100 1000 1111

- These 4 bytes could be interpreted as 4 characters (1 byte each) or an integer (4 bytes).
- But if we interpret this as 4 characters - nonsense!
- As an integer it is: 5263

	Binary	Hex	ASCII
1st Byte	0000 0000	0	NUL
2nd Byte	0000 0000	0	NUL
3rd Byte	0001 0100	14	DC4
4th Byte	1000 1111	8F	undefined (> 127)

0000 0000 0001 0100 1000 1111

- In a C program:

```
fscanf ( fptr, “%d”, &i );
fscanf ( fptr, “%c%c%c%c”, &a, &b, &c, &d );
```

- If this bit string (4 bytes worth) was in a file than it could not be read in a text editor since some of the characters are not ASCII.
- See program: *interpretBytes.c*

```c
#include <stdio.h>

int main ( )
{
   unsigned int num = 5263;
   char a, b, c, d;

   FILE *fptr;
   fptr = fopen ( "testme", "wb" );
   fwrite ( &num, sizeof(num),1, fptr );
   fclose ( fptr );

   fptr = fopen ( "testme", "r" );
   fscanf ( fptr, "%d", &num );
   printf ( "Number = %d\n", num );
   fclose ( fptr );

   fptr = fopen ( "testme", "r" );
   fscanf ( fptr, "%c%c%c%c", &a, &b, &c, &d );
   printf ( "a = %c \n", a );
   printf ( "b = %c \n", b );
   printf ( "c = %c \n", c );
   printf ( "d = %c \n", d );
   fclose ( fptr );

   return(0);
}
```

```
$ ./interpretBytes
Number = 5263
a = ?
b =
c =
d =
$ od -d testme
0000000      5263       0
0000004
$ od -c testme
0000000  217 024  \0  \0
0000004
$
```

File Operations

- Files in C are treated as a **stream**:
  - an ordered **sequence** of **bytes** (similar to an array)
  - can be read, written, and moved through
  - separate streams to several files can be opened at one time

Opening a File

- A file can be **open**ed for **reading** or **writing**.
- The filename is associated with a **file pointer**.
- The pointer points to a structure that contains the file information which is managed by the OS (bookkeeping information that the OS has to share with program so that it can access the file).

Declaring a File Pointer

- A file pointer is declared using the `FILE` type:

```
FILE *fp;
FILE *infile;
```

- The file is opened using open.

```
fp = fopen ( "filename", "r" );
```

file pointer *name of the file that you wish to open* *mode*

File Modes

- Mode indicates what you can do with the file

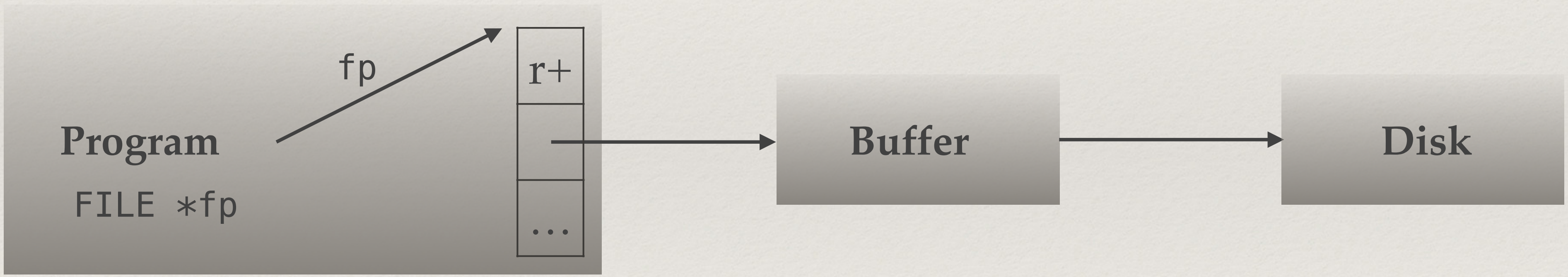
Mode	Meaning
r	**open existing** file for **reading**; start at the *beginning*
w	**create** a **new** file for **writing**
a	**append** - open an **existing** file; **write** at the *end* of the file
r+	**open existing** file for **reading** and **writing**; start at the *beginning*
w+	**create new** file for **reading** and **writing**; start at the *beginning;* **truncates existing** file
a+	**open existing** file for **reading** and **writing**; write to the *end* of the file

Files, Buffers & C

- There is not a direct path to the disk.
- It would be inefficient to actually write to the disk (hardware) after each program write.

Files, Buffers & C

- There is not a direct path to the disk.
- It would be inefficient to actually write to the disk (hardware) after each program write.



Reading from a File

- Reading from a file containing text can be done using the same commands that are used to read from the keyboard (*i.e.* the `stdin` stream).
- You need to use the "file" versions: `fscanf`, `fgets`.
- Both commands are exactly the same as reading from the keyboard/`stdin` with the addition of a file pointer.

fscanf and fgets

- Read individual values from a specified stream (file).

```
fscanf ( fp, "%d %c", &num, &ch );
```

Read from the stream associated with `fp` in the `fopen` command

Exactly the same as scanf

- Read a string from an input stream.

```
fgets ( str, 100, fp );
```

char array to store input

maximum length of input

read from file associated with `fp` in `fopen`

Writing to a File

- Writing text to a file is performed using `fprintf` and `fputs`.
- Similar to output using `printf` but writes to a file instead of the screen (`stdout`).

```
fprintf ( fp, “%s %d”, stringA, numB );
```

File pointer from `fopen`

Same as `printf`

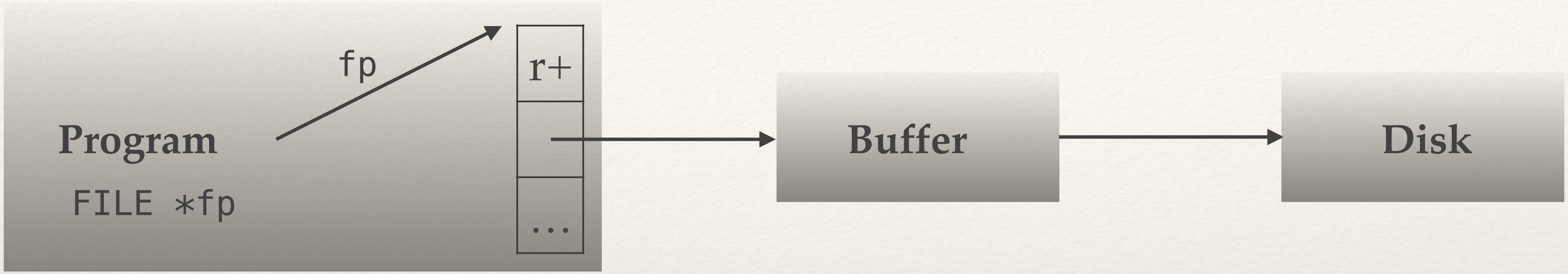
```
fgets( strPtr, length, fp);
```

Pointer to string

maximum length of input

File pointer from `fopen`

Writing to a File



- Note: the program could have written to the buffer, but the buffer not yet stored on the disk
- Buffers are written to disk on
  - `fprintf ( fp, “\n” )`
  - `fflush ( fp )`

Closing a File

- When you have finished reading or writing to a file it must be closed using `fclose`.

```
fclose ( fp );
```

- This forces any buffered data to be written to the disk.
- If the file is not closed then data may be lost (*i.e.* not written to the file).
- It also frees the memory used to store file information.

Writing Binary Data to a File

- Writing binary data to a file requires the `fwrite()` command.
- This writes the data in the form that it is stored in by the system (**raw** bits) and not as ASCII.

```
fwrite ( ptr, size, number, fp );
```

Pointer to data that will be written to disk

Number of bytes in each element written

Number of elements to be written

Stream to write to (*i.e.* file)

Binary File Write Example

```
#include <stdio.h>
int main ( )
{
   int numArray[100], i;
   FILE *fptr;

   fptr = fopen ( "tmpfile", "wb" );

   for ( i=0; i<100; i++ ) {
      numArray[i] = i;
   }

   fwrite ( numArray, sizeof(int), 100, fptr );

   fclose ( fptr );
}
```

open for writing

write 100 elements that are the size of an int (4 bytes), taken from array `numArray` *and put into the file associated with the file pointer* `fptr`

- Creates a 400 byte file:

```
$ ls -l tmpfile
•-rw-r--r--  1 dastacey  staff  400  6 Dec 14:51 tmpfile
```

```
$ od -d tmpfile
0000000      0     0     1     0     2     0     3     0
0000020      4     0     5     0     6     0     7     0
0000040      8     0     9     0    10     0    11     0
0000060     12     0    13     0    14     0    15     0
0000100     16     0    17     0    18     0    19     0
0000120     20     0    21     0    22     0    23     0
0000140     24     0    25     0    26     0    27     0
0000160     28     0    29     0    30     0    31     0
0000200     32     0    33     0    34     0    35     0
0000220     36     0    37     0    38     0    39     0
0000240     40     0    41     0    42     0    43     0
0000260     44     0    45     0    46     0    47     0
0000300     48     0    49     0    50     0    51     0
0000320     52     0    53     0    54     0    55     0
0000340     56     0    57     0    58     0    59     0
0000360     60     0    61     0    62     0    63     0
0000400     64     0    65     0    66     0    67     0
0000420     68     0    69     0    70     0    71     0
0000440     72     0    73     0    74     0    75     0
0000460     76     0    77     0    78     0    79     0
0000500     80     0    81     0    82     0    83     0
0000520     84     0    85     0    86     0    87     0
0000540     88     0    89     0    90     0    91     0
0000560     92     0    93     0    94     0    95     0
0000600     96     0    97     0    98     0    99     0
0000620
```

Reading a Binary File

- Reading binary data from a file is done with `fread()`.
  - `fread()` has the same parameters as `fwrite()`.

```
fread ( ptr, size, number, fptr );
```

ptr: array that the data will be stored in or address of a data structure

size: size of the array elements

number: number of array elements

fptr: file to be read from

Binary File Read Example

```
#include <stdio.h>
int main ( )
{
    int numArray[100];
    FILE *fptr;

    fptr = fopen ( “tmpfile”, “rb” );
    fread ( numArray, sizeof(int), 100, fptr );
    fclose ( fptr );

    printf ( “numbers: %d %d\n”, numArray[0], numArray[99] );
}
```

given the file created in the previous program to write a binary file

reads 100 integers from the file (`tmpfile`) into the array (`numArray`)

```
$ ./binaryFileRead
numbers: 0 99
```

Reading a Binary File

- The system is really reading/writing the number of bytes equal to `size * number`.
- It does not read one element at a time.
- It reads all the requested bytes and places them in the data structure (*e.g.* array).
  - Boundaries between elements are ignored.
- In the previous two examples, the file created is a collection of 100 integers (400 bytes).

Reading and Writing Structures

- Reading and writing structures can be achieved using fread / fwrite.
- Example: write and read an array of 10 structures to disk

```
#include <stdio.h>
struct data {
    char letter;
    int number;
    double bignum;
};
typedef struct data datatype;

int main ( )
{
    /* code that uses datatype */
}
```

replaces `struct data` *with one word* (`datatype`)

Reading and Writing Structures

- Read
  fread
- Exam
  disk

```
int main ( )
{
    struct data *mydata

    /* code that uses mydata struct pointer  */
}
```

same as

```
int main ( )
{
    datatype *mydata

    /* code that uses mydata struct pointer  */
}
```

a *with*
)

```
{
    /* code that uses datatype */
}
```

```
FILE *fptr;
int i;
datatype labels[10];

…
/*
 *   Write the data structures
 */
fptr = fopen ( "outfile", “wb" );
fwrite ( labels, sizeof ( datatype ), 10, fptr );
fclose ( fptr );

…
/*
 *   Read the data structures
 */
fptr = fopen ( "outfile", "rb" );
fread ( labels, sizeof ( datatype ), 10, fptr );
fclose ( fptr );
```

Reading and Writing Structures

- How big is the file `outfile` when written?
- filesize = sizeof (datatype) * 10
- `datatype` contains:

1 char	1 byte
1 integer	4 bytes
1 double	8 bytes
	13 bytes

- The file should be 13 * 10 = 130 bytes.
- But structures can be padded so that each element begins on a 4 byte boundary.

Reading and Writing Structures

- Padding

1 char	**4 bytes**
1 integer	4 bytes
1 double	8 bytes
	16 bytes

- Why?
- Variables that start on a 4-byte boundary are faster to manipulate by the machine.
- The actual file size is 16 * 100 = 160 bytes

Reading and Writing Structures

- The difference between passing a structure versus passing an array to `fread()` and `fwrite()`?
    - Arrays provide their own address
    - Their name = their address

```
datatype a[10];
datatype b;

fwrite (  a, sizeof ( datatype ), 10, fptr );

fread  ( &b, sizeof ( datatype ),  1, fptr );
```

array name = its address

must take address of structure (unless supplying a struct pointer)

Sequential Access Files

- All the files that we have examined have been read/written in a sequence.
  - Sequential Access File

item *1*	item *2*	item *3*	item *4*	…	item *m*

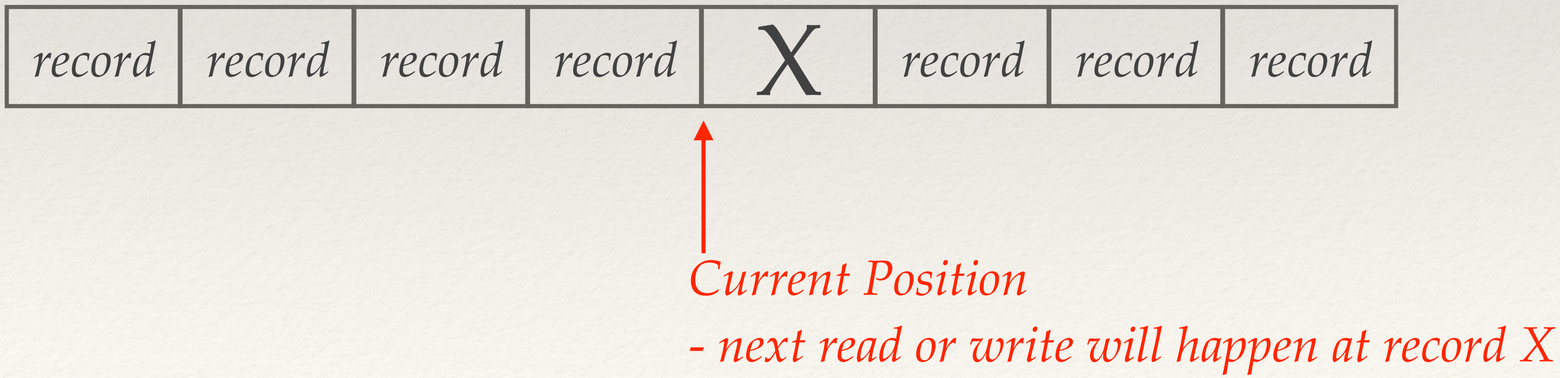
  - item could be a primitive type (int, float, *etc.*) or a structure (struct)
  - records (or items) are read/written in order

Random Access Files

- But often you do not want to read or write only in order; your algorithm might need to "*jump*" around the file getting records in **random** order.
- Random Access allows files to be read/written in **any** order.
- The files themselves are the same (sequence of bytes) as all other files but there are two functions used to control where access occurs.

Random Access Files

- `fseek()` - move to a location in the file
- `ftell()` - report to your program its current location in the file
- Both `fseek()` and `ftell()` use a current position value to indicate where the file will next be accessed.



fseek()

```
fseek ( fptr, offset, from );
```

file pointer

how far to move in the file

one of 3 values that tell where to seek **from**

`fseek()` returns 0 on **success** and non-zero on *failure*

SEEK_SET	start of the file
SEEK_CUR	from the current position
SEEK_END	end of the file

`fseek()`

```
fseek ( fptr,sizeof(datatype)*3,SEEK_SET );
```

in file associated with `fptr`

move 3 times the size of the `datatype` structure

move from the start of the file

SEEK_SET	start of the file
SEEK_CUR	from the current position
SEEK_END	end of the file

fseek()

```
fseek (fptr,sizeof(datatype)*-3,SEEK_END);
```

in file associated with `fptr`

move **back** 3 times the size of the `datatype` structure

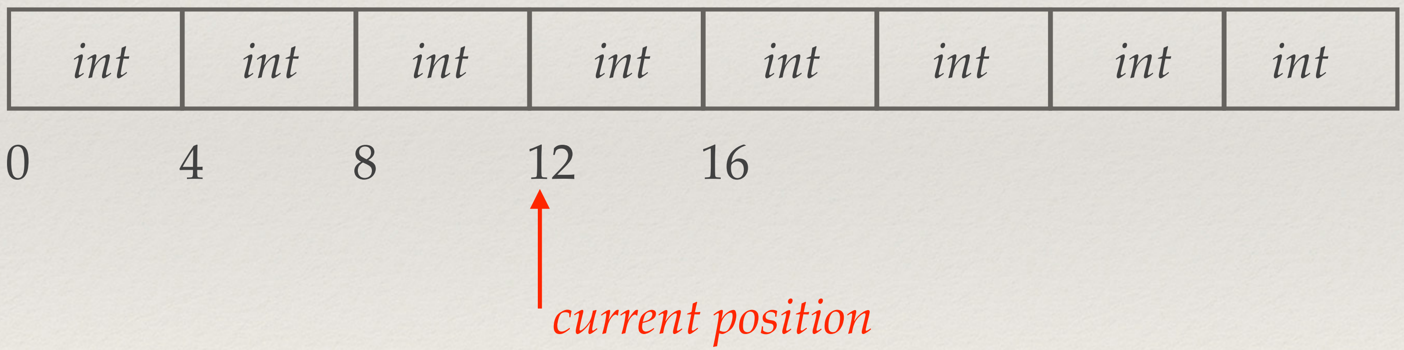
move from the end of the file

SEEK_SET	start of the file
SEEK_CUR	from the current position
SEEK_END	end of the file

ftell()

```
long tell ( fptr );
```

- Return value is the number of bytes from the start of the file
- Example: in a file of integers (4 bytes each)



- `ftell()` would return a value of 12

```c
#include <stdio.h>
#include <string.h>

int main ()
{
   struct info {
      char name[10];
      int  id;
   };
   typedef struct info record;
   record item;
   record newItem[10];

   FILE *fptr;
   int i;
   char names[40] = "FredJohnMaryFordGertHammWillBillPaulSamm";

   fptr = fopen ( "data", "w+" );
   for ( i=0; i<10; i++ ) {
      strncpy ( item.name, &names[i*4], 4  );
      strncpy ( &item.name[4],"\0",1 );
      item.id = i;
      fwrite ( &item, sizeof(item), 1, fptr );
   }
   fseek ( fptr, 0, SEEK_SET );
   for ( i=0; i<10; i++ ) {
      fread ( &newItem[i], sizeof(record), 1, fptr );
   }

   for ( i=0; i<10; i++ ) {
      printf ( "(%d) %s - %d\n", i+1, newItem[i].name, newItem[i].id );
   }
```

Open new file for writing and reading

Write a record to the file

Move to the beginning of the file

Read a record from the file

Start at the beginning of the new file and write 10 records

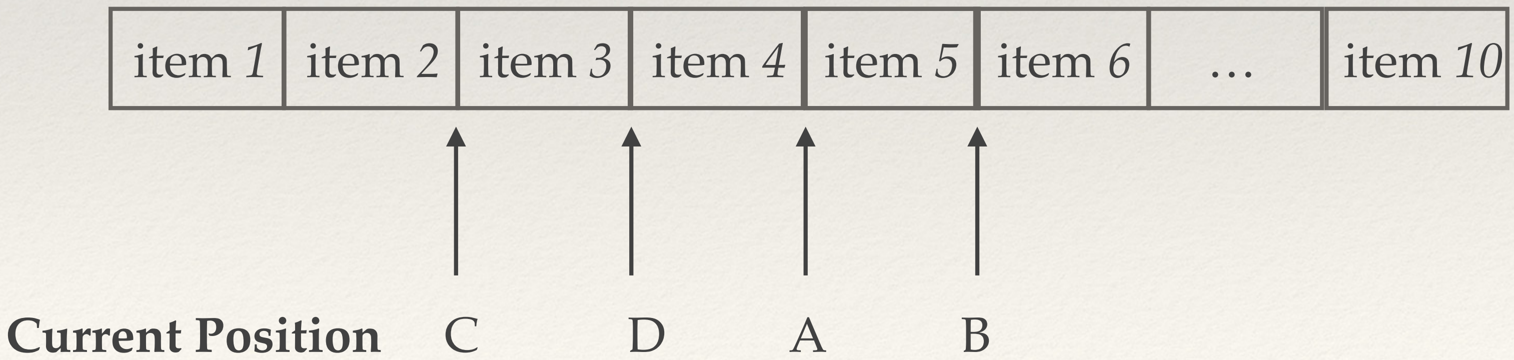
```
A  fseek ( fptr, ( sizeof(item) * 4 ), SEEK_SET );
B  fread ( &item, sizeof(item), 1, fptr );
   printf ( "Record 5: %s - %d\n", item.name, item.id );
C  fseek ( fptr, sizeof(item) * -3, SEEK_CUR );
D  fread ( &item, sizeof(item), 1, fptr );
   printf ( "Record 3: %s - %d\n", item.name, item.id );
   fclose ( fptr );
}
```

Move to the end of Record 4 (starting from the beginning of the file)

Read Record 5

Move 3 records backwards from the current location

Read Record 3



```
$./seekTell
(1) Fred - 0
(2) John - 1
(3) Mary - 2
(4) Ford - 3
(5) Gert - 4
(6) Hamm - 5
(7) Will - 6
(8) Bill - 7
(9) Paul - 8
(10) Samm - 9
Record 5: Gert - 4
Record 3: Mary - 2
```

Note: The file is padded since the structure was not a multiple of 4 bytes.

1	2	3	4	5	6	7	8	9	10	P	P	1	2	3	4

name —————————————— *Padding* — **id** ——————