Files

```
#include <stdio.h>

FILE *fopen(const char *path, const char *mode);
int fclose(FILE *stream);
int fprintf(FILE *stream, const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
```

The mode in `fopen()` can have the following possible values.

- r Open text file for reading. The stream is positioned at the beginning of the file.
- r+ Open for reading and writing. The stream is positioned at the beginning of the file.
- w Truncate file to zero length or create text file for writing. The stream is positioned at the beginning of the file.
- w+ Open for reading and writing. The file is created if it does not exist, otherwise it is truncated. The stream is positioned at the beginning of the file.
- a Open for appending (writing at end of file). The file is created if it does not exist. The stream is positioned at the end of the file.
- a+ Open for reading and appending (writing at end of file). The file is created if it does not exist. The initial file position for reading is at the beginning of the file, but output is always appended to the end of the file.

Text versus Binary Files

- A text file consists of formatted lines separated by end of line character(s). Useful for human consumption since they are readable by humans. Various operating systems (unfortunately) use different line separator characters.
  - Linux: the line separator is the *newline* character
  - Mac OS X: *carriage return* character
  - MS Windows: carriage return followed by the newline character
- Binary files are unformatted with data stored directly in its native format. They take less space and are faster to read/write. However, we have to know their layout to be able to read them. Use the `fread()` and `fwrite()` functions in the standard library for binary I/O.

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite (const void *ptr, size_t size, size_t nmemb,
                                                FILE *stream);
```

- See following examples:
  - `file-io/generate_text.c`
  - `file-io/generate_binary.c`

Comparison of Text versus Binary Files

The following example shows two files being generated with the same logical content: one is a text file and the other is a binary file.

```
[amit@onyx file-io]$ time gentxt 10000000 1234

real    0m25.900s
user    0m24.637s
sys     0m0.782s
[amit@onyx file-io]$ time genbin 10000000 1234

real    0m1.410s
user    0m0.763s
sys     0m0.446s

[amit@onyx file-io]$ ls -lh data.*
-rw-r--r-- 1 amit faculty 306M Mar 24 09:16 data.bin
-rw-r--r-- 1 amit faculty 395M Mar 24 09:16 data.txt
[amit@onyx file-io]$
```

The file size was about 30% bigger for text files and the time was about 18 times slower!

Example: Writing a Text File

```
/* C-examples/file-io/generate_text.c */
/* appropriate header files */
struct record {
    int key;
    double value1;
    double value2;
    double value3;
};
const int MAX_KEY = 1000000;

void generate_file(int n, unsigned int seed, FILE *fout)
{
    int i;
    struct record *next = (struct record *) malloc(sizeof(struct record));
    srandom(seed); /* set starting seed for random generator */
    for (i=0; i<n; i++) {
        next->key = random() % MAX_KEY;
        next->value1 = random()/((double)MAX_KEY);
        next->value2 = random()/((double)MAX_KEY);
        next->value3 = random()/((double)MAX_KEY);
        fprintf(fout, "%d %lf %lf %lf\n", next->key,
                next->value1, next->value2, next->value3);
    }
    free(next);
}
int main(int argc, char **argv) {
    /* ... other code */
    FILE *fout = fopen("data.txt", "w");
    generate_file(n ,seed, fout);
    fclose(fout);
    return 0;
}
```

Example: Writing a Binary File

```
/* C-examples/file-io/generate_binary.c */
/* appropriate header files */
struct record {
    int key;
    double value1;
    double value2;
    double value3;
};
const int MAX_KEY = 1000000;

void generate_file(int n, unsigned int seed, FILE *fout)
{
    int i;
    struct record *next = (struct record *) malloc(sizeof(struct record));
    srandom(seed); /* set starting seed for random num. generator */
    for (i=0; i<n; i++) {
        next->key = random() % MAX_KEY;
        next->value1 = random()/((double)MAX_KEY);
        next->value2 = random()/((double)MAX_KEY);
        next->value3 = random()/((double)MAX_KEY);
        fwrite(next, sizeof(struct record), 1, fout);
    }
    free(next);
}

int main(int argc, char **argv) {
    /* ... other code */
    fout = fopen("data.bin", "w");
    generate_file(n ,seed, fout);
    fclose(fout);
    return 0;
}
```

Random Access in Files

- If a file is stored on a media that supports random access, then we can seek to anywhere on the file and read/write instead of reading/writing sequentially.
- Random access files can be text or binary although they are usually binary.
- Databases often use random access files for storing data that is sorted or hashed for fast access.
- We will consider an external search program that reads records from a random access sorted file to search for a record with a matching key value. The data file consists of n records. The program provides a linear and a binary search option. The binary search would take $O(\lg n)$ disk reads versus $O(n)$ disk reads for linear search.

Library Functions for Random Access

```
#include <stdio.h>

 int fseek(FILE *stream, long offset, int whence);
 long ftell(FILE *stream);
 void rewind(FILE *stream);
 int fgetpos(FILE *stream, fpos_t *pos);
 int fsetpos(FILE *stream, fpos_t *pos);
```

- The `fseek` function sets the file position indicator for the stream pointed to by stream. The new position, measured in bytes, is obtained by adding offset bytes to the position specified by whence. If whence is set to `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`, the offset is relative to the start of the file, the current position indicator, or end-of-file, respectively. A successful call to the fseek function clears the end-of-file indicator for the stream.
- The `ftell` function obtains the current value of the file position indicator for the stream pointed to by stream.
- rewind function sets the cursor back to the start of the file.
- The `fgetpos` and `fsetpos` functions are alternate interfaces equivalent to ftell and fseek (with whence set to `SEEK_SET`), setting and storing the current value of the file offset into or from the object referenced by pos

Example: Determining File Size

```
/* C-examples/file-io/filesize.c */
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv)
{
    long int size;
    FILE *fin;

    if (argc !=2 ) {
        fprintf(stderr, "Usage: %s <filename>\n", argv[0]);
        exit(1);
    }

    fin = fopen(argv[1], "r");
    if (!fin) {
        perror("filesize:");
    }
    fseek(fin, 0, SEEK_END);
    size = ftell(fin);
    printf("Size of the file %s = %ld bytes\n", argv[1],  size);
    exit(0);
}
```

Example: External Search

- External search means we are searching for data in files stored on disks. This is useful when the data files are too large to read into an array in memory.
- The files in the example:

```
[amit@kohinoor C-examples]$ ls C-examples/file-io/disk_search/
common.h  ExternalSearch.h  ExternalSearch.c Record.h Record.c
gendata.c SearchTest.c timing.c Makefile
```

- The files `Record.h` and `Record.c` define the data record that is stored in the file.
- The files `ExternalSearch.h` and `ExternalSearch.c` define a linear search and a binary search over the records stored in the data file.
- The file `SearchTest.c` performs specified number of random linear or binary searchers to performance comparison.
- The file `gendata.c` generates a sorted data file that can be used as input to the test program. The file `timing.c` contains functions that allow precise timing of a section of code inside a program.

Example: Record.h

```
/* C-examples/file-io/disk_search/Record.h */
#ifndef  __RECORD_H
#define  __RECORD_H

#include <stdlib.h>
#include <stdio.h>
#include "common.h"

typedef struct record Record;
typedef struct record * RecordPtr;
struct record {
    long int key;
    double value1;
    double value2;
    double value3;
};
char *toString(RecordPtr record);
#endif /*  __RECORD_H */
```

Example: Record.c

```
/* C-examples/file-io/disk_search/Record.c */
#include "Record.h"

#define MAXLEN 128

char *toString(RecordPtr record)
{
  char *buffer;

  buffer = (char *) malloc(sizeof(char)*MAXLEN);
  sprintf(buffer, "key=%ld values=[%lf, %lf, %lf]\n",
    record->key, record->value1, record->value2, record->value3);
  return buffer;
}
```

Example: ExternalSearch.h

```
/* C-examples/file-io/disk_search/ExternalSearch.h */
#ifndef  __EXTERNALSEARCH_H
#define  __EXTERNALSEARCH_H
#include <stdlib.h>
#include <stdio.h>
#include "common.h"
#include "Record.h"

RecordPtr extBinarySearch(FILE *dataFile, unsigned long int key);

RecordPtr extLinearSearch(FILE *dataFile, unsigned long int key);

#endif /*  __EXTERNALSEARCH_H */
```

Example: ExternalSearch.c (1)

```
/* C-examples/file-io/disk_search/ExternalSearch.c */
#include "ExternalSearch.h"
static long int numRecords(FILE *);

RecordPtr extLinearSearch(FILE *dataFile, unsigned long int key)
{
    long int i = 0;
    long int count = 0;
    long int pos = 0; /* offset into the file, in bytes */
    RecordPtr record;

    if (dataFile == NULL) return NULL;
    count = numRecords(dataFile);

    record = (RecordPtr) malloc(sizeof(Record));
    for (i = 0; i < count; i++) {
        pos = i * sizeof(Record); /* calculate byte offset in the file */
        fseek(dataFile, pos, SEEK_SET);
        fread(record, sizeof(Record), 1, dataFile);
        if (record->key == key) {
            return record;
        }
    }
    free(record);
    return NULL;
}
```

Example: ExternalSearch.c (2)

```
/* C-examples/file-io/disk_search/ExternalSearch.c */

RecordPtr extBinarySearch(FILE *dataFile, unsigned long int key)
{
    /* the search window is low..high */
    long int low = 0, mid = 0, high = 0;
    long int pos = 0; /* offset into the file, in bytes */
    RecordPtr record;
    if (dataFile == NULL) return NULL;
    high = numRecords(dataFile);

    record = (RecordPtr) malloc(sizeof(Record));
    while (low <= high) {
        mid = (low+high)/2;
        pos = mid * sizeof(Record); /* calculate byte offset in the file */
        fseek(dataFile, pos, SEEK_SET);
        fread(record, sizeof(Record), 1, dataFile);
        if (record->key == key) {
            return record;
        } else if (record->key < key) {
            low = mid + 1;
        } else { /* record->key > key */
            high = mid - 1;
        }
    }
    free(record);
    return NULL;
}
```

Example: ExternalSearch.c (3)

```
/* C-examples/file-io/disk_search/ExternalSearch.c */

static long int numRecords(FILE *dataFile)
{
    long int count = 0;

    /* figure out the size of the file in bytes */
    fseek(dataFile, 0, SEEK_END);
    count = ftell(dataFile);
    /* set high to number of records in the file */
    count = count/sizeof(Record);
    if (DEBUG >= 2) {
        fprintf(stderr, "data file has %ld records\n", count);
    }
    return count;
}
```

Checkpointing to Disk

- Check-pointing is to save ("freeze-dry") the state of a program to a binary file on disk periodically. So if the power goes out or the process gets killed it can restart ("reconstituted") from the saved state instead of having to start all over again.
- This concept is also known as serialization and is built into Java!
- When do we checkpoint?
  - Checkpoint once every n operations.
  - Check the time periodically from the code and checkpoint every n minutes (or hours).
  - Set up a repeating alarm such that each time the alarm is received from the operating system, the code checkpoints.
- We will look at a checkpoint capable version of the generic doubly-linked list.

Extra Checkpoint Code for Doubly-Linked Lists (1)

```
/* excerpt from C-examples/file-io/checkpoint/Job.c */
int getJobSize(JobPtr job)
{
    int size = 0;
    size = sizeof(int) + sizeof(int) +  job->infoSize;
    return size;
}

void checkpointJob(JobPtr job, FILE *fout)
{
    fwrite(&(job->jobid), sizeof(int), 1, fout);
    fwrite(&(job->infoSize), sizeof(int), 1, fout);
    fwrite(job->info, job->infoSize, 1, fout);
}


JobPtr restoreJob(FILE *fin)
{
    int jobid;
    int infoSize;
    JobPtr job;

    fread(&jobid, sizeof(int), 1, fin);
    fread(&infoSize, sizeof(int), 1, fin);
    job = createJob(jobid, "");
    job->info = (char *) malloc(sizeof(char)*infoSize);
    fread(job->info, infoSize, 1, fin);
    return job;
}
```

Extra Checkpoint Code for Doubly-Linked Lists (2)

```
/* excerpt from C-examples/file-io/checkpoint/Node.c */
int getDataSize(NodePtr node)
{
    return getJobSize(node->data);
}

void checkpointNode(NodePtr node, FILE *fout)
{
    checkpointJob(node->data, fout);
}

NodePtr restoreNode(FILE *fin)
{
    NodePtr node;
    JobPtr data;

    data = restoreJob(fin);
    node = createNode(data);
    return node;
}
```

Extra Checkpoint Code for Doubly-Linked Lists (3)

```
/* excerpt from C-examples/file-io/checkpoint/List.c */
#define MAX_MSG_LENGTH 1024

Boolean checkpointList(ListPtr list, char *saveFile)
{
    NodePtr node;
    FILE *fout;
    char errmsg[MAX_MSG_LENGTH];

    if (list == NULL) return TRUE;
    if (isEmpty(list)) return TRUE;
    if (saveFile == NULL) return FALSE;
    if (strlen(saveFile) == 0) return FALSE;
    fout = fopen(saveFile, "w");
    if (!fout) {
        sprintf(errmsg, "checkpointList: %s",saveFile);
        perror(errmsg);
        return FALSE;
    }
    fwrite(&(list->size), sizeof(int), 1, fout);

    node = list->tail;
    while (node) {
        checkpointNode(node, fout);
        node = node->prev;
    }
    fclose(fout);
    return TRUE;
}
```

Extra Checkpoint Code for Doubly-Linked Lists (4)

```
/* excerpt from C-examples/file-io/checkpoint/List.c */
/* ... */
ListPtr restoreList(char *saveFile)
{
    int size;
    FILE *fin;
    ListPtr list;
    NodePtr node;
    char errmsg[MAX_MSG_LENGTH];

    if (saveFile == NULL) return NULL;
    if (strlen(saveFile) == 0) return NULL;
    fin = fopen(saveFile, "r");
    if (!fin) {
        sprintf(errmsg, "restoreList: %s",saveFile);
        perror(errmsg);
        return NULL;
    }
    fread(&size, sizeof(int), 1, fin);
    if (size <= 0) return NULL;
    printf("restore: list size = %d\n", size);
    list = createList();

    while (size > 0) {
        node = restoreNode(fin);
        addAtFront(list, node);
        size--;
    }
    fclose(fin);
    return list;
}
```

Extra Checkpoint Code for Doubly-Linked Lists (5)

- Build and run the code from the folder `C-examples/file-io/checkpoint`
- Check out the files `SaveList.c` nd `RestoreList.c` to see how to use the checkpointing interface from the list class.
- We will see how to automatically checkpoint based on a timer later this semester.

Serialization in Java

- Java has built-in serialization for objects. A class has to implement the empty interface *Serializable* for it to be marked as serializable.

```
public class Record implements Serializable {
...
}
```

- Serializing an object:

```
FileOutputStream fout = new FileOutputStream("obj.serial");
ObjectOutputStream out = new ObjectOutputStream(fout);
out.writeObject(obj);
```

- De-serializing an object:

```
FileInputStream fileIn = new FileInputStream("obj.serial");
ObjectInputStream in = new ObjectInputStream(fileIn);
Record obj = (Record) in.readObject();
```

- See examples `SaveList.java` and `LoadList.java` in the folder `C-examples/file-io/checkpoint-java`.
- Multiple serialized objects are stored sequentially in a file and can only be accessed sequentially. To do random access requires some custom coding.