# File Processing

## Files

- Used for permanent storage of large quantity of data
- Generally kept on secondary storage device, such as a disk, so that the data stays even when the computer is shut off

## Data hierarchy

- Bit
  - Binary digit
  - true or false
  - quarks of computers
- Byte
  - Character (including decimal digits)
  - Atoms
  - Smallest addressable unit (difficult to get by itself)
  - Generally, eight bits to a byte though there used to be 6-bit bytes in the 60s
- Word
  - Collections of bytes
  - Molecules
  - Smallest unit fetched to/from memory at any time
  - Number of bits in word is generally used as a measure of the machine's addressability (32-bit machine)
- Field
  - Collection of bytes or even words
  - Exemplified by the name of an employee (30 characters/bytes, or 8 words on a 32-bit machine)
- Record
  - `struct` in C
  - Collection of fields
- File
  - Collection of records
  - Each record in the file identified with a unique [set of] field[s], called *key*
  - I use student name as a key to keep the file of grades
  - The payroll of a large company may use the social security number as the key
  - Sequential file
    - Records follow one after the other
  - Random access file
    - The location of a record is a function of the key
    - Mostly used in databases
  - Indexed sequential file
    - The location of a record is dependent on an index kept in a separate file

**Files and streams**

- C views each file simply as a sequential stream of bytes
- Each file ends with a special *end-of-file marker*, CTRL-D in Unix, CTRL-Z in Windows
- Streams
  - Any source for input or any destination for output
  - Communication channels between files and programs
  - Accessed through a file pointer of type `FILE *`
- Opening a file
  - Equivalent to associating a stream with the file
  - Returns a pointer to a `FILE` structure
    - Defined in `<stdio.h>`
    - The pointer structure contains information used to process the file
      - `file descriptor` – Index into the operating system array called *open file table*
      - Each element in the open file table contains a *file control block* that is used by the OS to administer the corresponding file
    - `FILE` structure is dependent upon the operating system
      - Members of the structure vary among systems based on how each system handles its files
  - Three files and their associated streams are automatically opened at the beginning of program execution
    1. `stdin` – Standard input
       - Stream to read data from the keyboard
    2. `stdout` – Standard output
       - Print data on the screen
    3. `stderr` – Standard error
       - Prints data on screen as soon as it is available
  - Standard streams can be *redirected* by using the feature from operating system
  - A file must be opened before it is referred to in the program
    - The standard streams (`stdin`, `stdout`, and `stderr`) are automatically opened whenever you run a program
- Standard library
  - Provides many functions for reading data from files and for writing data into the files
  - `fgetc ( fd )`
    - Reads one character from the file stream associated with `fd`
    - If there is no more data in the file, it returns the constant `EOF`
    - `getchar()` can also be written as `fgetc ( stdin )`
  - `fputc ( ch, fd )`
    - Writes the character `ch` into the file stream associated with `fd`
    - `putchar ( ch )` can also be written as `fputc ( ch, stdout );`
    - `ch` is actually of type `int` but only the least significant 8 bits are considered
  - `fgets ( str, n, stream )`
    - Get a string `str` containing `n` characters from the `stream`
    - If the line terminates by reading in `\n` before `n` characters, `fgets` stops reading at that point
    - `gets (str )` is the equivalent function to read in a string `str` from `stdin`
  - `fputs ( str, stream )`
    - Put the string `str` on the `stream`

    - Does not need the specification of size but keeps writing until it encounters the end of string character `'\0'`
    - `puts ( str )` is the equivalent function to write a string to `stdout`
  - Other file I/O functions include `fscanf`, `fprintf`, `fread`, and `fwrite`

## Creating a sequential access file

- No file structure imposed by C, therefore, structure of a file is entirely up to the programmer
- The file variable name must be declared with the `FILE *` type
- Every file is handled by a separate `FILE *` variable, or file descriptor
- Opening a file
  - Process of connecting a program to a file, or associating a stream with a file
  - Use the function `fopen`

```
fd = fopen ( name, mode );
```

    - `FILE * fd` is the file descriptor or the file variable
    - `char * name` is the actual name of the file (`vectors.dat`)
    - `char *mode` indicates if the file is to be read or written into

      Different modes are

| | |
|---|---|
| `r` | Read. Only to be used for existing files |
| `w` | Write. If the file already exists, its old contents are lost; otherwise, the file is created |
| `a` | Append. If the file does not exist, it is created |
| `r+` | Update an existing file (read and write) |
| `w+` | Same as `w` but reading also possible |
| `a+` | Same as `a` but reading also possible |

  - If `fopen` succeeds, it returns a pointer to be used to identify the stream in subsequent operations
  - If `fopen` fails, it returns a `NULL`

```
if ( ( fd = fopen ( "vectors.dat", "w" ) ) == NULL )
{
    printf ( "Error opening the file vectors.dat\n" );
    return ( 1 );
}
```

  - Once the file is open, actual reading and writing can be done by several functions, including `fscanf`, `fgets`, `fputs`, and so on
  - Example

```
/*****************************************************************************/
/* wr_fl.c : Capture user's inputs from the keyboard into a file            */
/*****************************************************************************/

#include <stdio.h>

int main()
{
    FILE *fd;                           /* File descriptor          */
    char line[100];                     /* Buffer for input         */
    char filename[] = "testfile";       /* File to keep information */

    /* Open the file */

    if ( ! ( fd = fopen ( filename, "w" ) ) )
    {
        printf ( "Sorry, could not not open file\n" );
        exit ( 1 );
    }

    /* Get data from the user     */
    printf ( "Please enter the text to write to the file\n" );
    printf ( "Press ctrl-d on a line by itself when finished\n\n" );

    fgets ( line, sizeof ( line ), stdin );
    while ( ! feof ( stdin ) ) /* Check for end of file character */
    {
        fprintf ( fd, "%s", line );
        fgets ( line, sizeof ( line ), stdin );
    }

    fclose ( fd );

    return ( 0 ); /* Normal termination */
}
```

  - Cautions
    - ∗ *Opening an existing file with mode `"w"` discards the current contents of the file without warning*
    - ∗ *You must open a file and attach it to a file descriptor before using it in the program*
    - ∗ *Never open a non-existent file for reading*
    - ∗ *Always check whether the file was opened properly*

- Closing a file
  - Disconnect the file from the program
  - Use the function `fclose`

    `fclose ( fd );`

    - ∗ Writes any buffered data for the named stream `FILE * fd` and then, closes the stream
    - ∗ Also frees any buffers allocated by the standard I/O system
    - ∗ Performed automatically for all open files upon calling `exit`
    - ∗ Returns 0 on success
    - ∗ Returns `EOF` on error (such as trying to write to a file that was not opened for writing)
  - Example – Copying an ASCII file to another

```
/*******************************************************************************/
/* fcpy.c : Copy a file to another                                             */
/* Limitations: Only works with ASCII files, with each text line being less    */
/*             than 100 characters                                             */
/*******************************************************************************/

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

int main()
{
    FILE *infile, *outfile;              /* File descriptors */
    char line[100];                      /* Buffer for I/O   */
    char file_1[40], file_2[40];         /* File names       */
    int n;

    /* Get source and destination file names                 */

    printf ( "What is the source file? " );
    fgets ( file_1, sizeof ( file_1 ), stdin );
    printf ( "What is the destination file? " );
    fgets ( file_2, sizeof ( file_2 ), stdin );

    /* Remove the newline character from the file names      */

    n = strlen ( file_1 );
    file_1[n-1] = NULL;
    n = strlen ( file_2 );
    file_2[n-1] = NULL;

    /* Open the files */

    if ( ! ( infile = fopen ( file_1, "r" ) ) )
    {
        printf ( "Sorry, could not open file %s for reading\n", file_1 );
        exit ( 1 );
    }
```

```
        if ( ! ( outfile = fopen ( file_2, "w" ) ) )
        {
            printf ( "Sorry, could not not open file %s to write\n", file_2 );
            exit ( 2 );
        }

        /* Read from input file and write into the output file  */

        fgets ( line, sizeof ( line ), infile );
        while ( ! feof ( infile ) )
        {
            fprintf ( outfile, "%s", line );
            fgets ( line, sizeof ( line ), infile );
        }

        /* Close the files */

        fclose ( infile );
        fclose ( outfile );

        exit ( 0 );                /* Normal termination */
    }
```

- Other useful functions
  - `freopen ( filename, mode, fd )` opens the file named by `filename` and associates the stream pointed to by `fd` with it
    - The `mode` argument is used just as in `fopen`
    - The original stream is closed, regardless of whether the open ultimately succeeds
    - If the open succeeds, `freopen` returns the original value of `fd`
    - Typically used to attach the preopened streams associated with `stdin`, `stdout`, and `stderr` to other files
  - `feof ( fd )` returns a non-zero if the end of stream `fd` has been reached, and zero otherwise
    - Useful to check end of file
  - `ferror ( fd )` is non-zero if an error has occurred while reading from or writing into the stream `fd`
    - The error indication lasts until the stream is closed, or the error indication is cleared by `clearerr()`
  - `clearerr ( fd )` resets the error indication and `EOF` indication to zero on the stream `fd`
- File position pointer
  - Part of the `FILE *` structure
  - Always points at the location of the byte in file where the file is to be read from, or written into
  - The location from the beginning of the file is expressed in number of bytes and is known as the *file offset*
  - May be manipulated by several commands
  - `rewind ( fd )`
    - Function to reset the file position pointer to the beginning of the file
    - Does not return a value
- Sequential access files are generally not updated in-place; if the file needs to be modified, it should be completely rewritten

### Line input and output

- `char * fgets ( char *str, int n, FILE *fd )`
  - Reads at most `n-1` characters from the stream `fd` into the array `str`
  - Newline character terminates reading after having been read into the `str`
  - Returns pointer to `str`
  - Returns `NULL` if end-of-file is encountered and no characters have been read

- `int fputs ( char *str, FILE *fd )`
  - Writes the string `str` to the stream `fd`
  - A newline character is written only if it is a part of `str`
  - Returns non-zero if an error occurs, otherwise returns zero
- `char * gets ( char * str )`
  - Version of `fgets` to use with `stdin`
  - Reads characters until a newline character is encountered
  - The newline character is not placed into `str`
- `int puts ( char * str )`
  - Version of `fputs` to use with `stdout`
  - A newline character is automatically added

### Unformatted I/O and direct access

- Random access files
  - Structured so as to allow the positioning of file pointer anywhere within the file in a meaningful manner (generally, beginning of record)
  - Preferable to have fixed-size records to easy repositioning
  - This ensures that records do not have to be searched to find the right one
  - Ideal for most large databases, such as airlines reservation system, bank accounts, and inventory files
  - Exact location of the record, relative to the beginning of the file, can be calculated as a function of the record key
  - You can update a specific record in a random access file without modifying other records
- Functions `fread` and `fwrite` for buffered binary I/O
- Allows non-ASCII representation of numbers to be written to a file
- Binary files
  - More efficient to write in the format that is used internally in the machine, as no conversion is needed
  - Take less space than ASCII files
  - Cannot be directly printed or viewed on screen
  - Not as portable as the ASCII files
- Example to write a binary sequence:

```
int i = 19;
FILE * fd;
fd = fopen ( ... );
...
fwrite ( &i, sizeof ( int ), 1, fd );
```

- The syntax is:

```
fread ( buf_ptr, size, nitems, stream );
fwrite ( buf_ptr, size, nitems, stream );
```

  - `fread` transfers a specified number of bytes from the location in the file specified by the file descriptor to a buffer in the memory beginning with the specified address

  - `fwrite` transfers a specified number of bytes beginning at a specific location in memory to the file in the location pointed to by the file descriptor
  - `char * buf_ptr` – Pointer to a buffer (the address of an object in memory)
  - `int size` – The size (in bytes) of one element in the buffer
  - `int nitems` – Number of elements in the buffer
  - `FILE * stream` – Pointer to the stream

- The following two statements achieve the same effect, except for output format (compare number of characters transferred)

```
fprintf ( fd, "%d", number );
fwrite ( &number, sizeof ( int ), 1, fd );
```

- Both functions return an integer value, equal to the number of items actually read or written (normally equal to `nitems`)
- If nothing at all can be read, possibly due to an end-of-file, the returned value is 0 (not `EOF`)
- End-of-file can be distinguished from a read error by one of the functions `feof` or `ferror`
- Direct access (or random access)
  - Used to update a file (read, modify, write)
  - Preferable to have all records to be the same length fixed in advance
    - Allows access to a record directly without having to scan through other records
    - Location of each record can be calculated relative to the beginning of the file
    - Some records in the file may be empty
    - Data can be inserted without destroying surrounding data
  - Locating a position in a file can be accomplished by the function `fseek`
  - `fseek ( FILE * stream, long offset, int whence )`
    - `stream` – file pointer
    - `offset` – Position expressed in bytes, relative to a point specified by `whence`
    - `whence` can have the following three values (defined in `stdio.h`

      `SEEK_SET` – `offset` is relative to the beginning of the file; `offset = 0` specifies the first possible position in file

      `SEEK_CUR` – `offset` is relative to the current position; `offset = -1` moves the file pointer back one byte

      `SEEK_END` – `offset` is relative to the end of the file (and must therefore be negative)
    - Returns zero if the call was successful; non-zero otherwise
    - `fseek` destroys character pushback accomplished through `ungetc`, if called before the `getc` call
  - The current position of the file pointer can be accessed by the function `ftell`
  - `long ftell ( FILE * stream )`
    - Returns the offset to be used by `fseek` if we want to return to the same position in the file stream

- Example: Create a credit processing system capable of storing up to 100 fixed-length records. Each record should consist of an account number that will be used as the record key, a last name, a first name, and a balance. The resulting program should be able to update an account, insert a new account record, delete an account and list all the account records in a formatted text file for printing.

```
/****************************************************************************/
/* types.h                                                                  */
/****************************************************************************/
typedef struct
{
    int      acct_num;
    char     last_name[15];
    char     first_name[15];
    float    balance;
} client_data_t;
```

```
/******************************************************************************/
/* Creating a randomly accessed file                                          */
/* create.c                                                                   */
/******************************************************************************/
#include <stdio.h>
#include "types.h"

int main()
{
    int      i;                      /* Loop counter                          */
    client_data_t blank_client = { 0, "", "", 0.00 };
    FILE   * client_file;

    if ( ( client_file = fopen ( "credit.dat", "w" ) ) == NULL )
    {
        printf ( "Could not open file credit.dat\n" );
        exit ( 1 );
    }

    for ( i = 0; i < 100; i++ )
        fwrite ( &blank_client, sizeof ( client_data_t ), 1, client_file );

    fclose ( client_file );

    return ( 0 );
}
/******************************************************************************/
```

- Using combinations of `fseek` and `fwrite` to store data at specific locations in file

```
/******************************************************************************/
/* Updating a randomly accessed file                                          */
/* update.c                                                                   */
/******************************************************************************/
#include <stdio.h>
#include "types.h"

int main()
{
    FILE   * client_file;
    client_data_t   client;
    char     line[80];                 /* Input buffer for stdin              */

    if ( ( client_file = fopen ( "credit.dat", "r+" ) ) == NULL )
    {
        printf ( "Could not open file credit.dat\n" );
        exit ( 1 );
    }

    printf ( "Enter account number (valid range: 1 -- 100; 0 to quit) : " );
    fgets ( line, sizeof(line), stdin );
    sscanf ( line, "%d", &client.acct_num );

    while ( client.acct_num )
    {
        printf ( "Enter last name, first name, and balance : " );
        fgets ( line, sizeof(line), stdin );
        sscanf (line, "%s%s%f", &client.last_name, &client.first_name, &client.balance);

        fseek (client_file, (client.acct_num-1)*sizeof(client_data_t), SEEK_SET);
        fwrite (&client, sizeof(client_data_t), 1, client_file);

        printf ( "Enter account number (valid range: 1 -- 100; 0 to quit) : " );
        fgets ( line, sizeof(line), stdin );
        sscanf ( line, "%d", &client.acct_num );
    }

    fclose ( client_file );

    return ( 0 );
}
/******************************************************************************/
```

- Illustrating `fread` and `feof`

```
/******************************************************************************/
/* Reading data from a random access file                                     */
/* read.c                                                                     */
/******************************************************************************/
```

```
#include <stdio.h>
#include "types.h"

int main()
{
    FILE   * client_file;
    client_data_t   client;

    if ( ( client_file = fopen ( "credit.dat", "r" ) ) == NULL )
    {
        printf ( "Could not open file credit.dat\n" );
        exit ( 1 );
    }

    printf ( "%-6s %-15s %-15s %10s\n", "Acct", "Last name", "First name", "Balance" );

    while ( ! feof ( client_file ) )
    {
        fread ( &client, sizeof ( client_data_t ), 1, client_file );
        if ( client.acct_num )
            printf ( "%-6d %-15s %-15s %10.2f\n", client.acct_num, \
                     client.last_name, client.first_name, client.balance );
    }

    fclose ( client_file );

    return ( 0 );
}
/****************************************************************************/
```

### Buffering problems

- Buffered I/O stores data in a buffer until the buffer is big enough to write to the disk.
- Look at the following two codes

| | |
|---|---|
| printf ( "starting program\n" ); | printf ( "starting program\n" ); |
| do_step_1(); | fflush ( stdout ); |
| printf ( "step 1 completed\n" ); | do_step_1(); |
| do_step_2(); | printf ( "step 1 completed\n" ); |
| printf ( "step 2 completed\n" ); | fflush ( stdout ); |
| do_step_3(); | do_step_2(); |
| printf ( "step 3 completed\n" ); | printf ( "step 2 completed\n" ); |
| | fflush (stdout ); |
| | do_step_3(); |
| | printf ( "step 3 completed\n" ); |
| | fflush ( stdout ); |

  - In the left code, the `printf` puts the output data in a buffer; the buffer gets flushed when it is full, or at the end of program
  - In the right code, the `fflush` statement forces the buffers to be flushed

- `int fflush ( FILE * stream)`
  - Forces a write of all buffered data for the given output [or update] `stream` via the stream's underlying write function
  - The open status of the `stream` is unaffected
  - If the `stream` argument is `NULL`, `fflush` flushes all open output streams
  - Returns 0 on successful completion and `EOF` on error, setting the global variable `errno` to indicate the error (like not an open stream, or the stream not open for writing)
  - `stdout` is line buffered and output will appear whenever a newline character is encountered
  - `scanf` flushes `stdout` before waiting for input

- Related function `int fpurge ( FILE * stream )`

  - Erases any input or output buffered in the given stream
  - For output streams, discards any unwritten output
  - For input streams, discards any unread data in the stream, including the data pushed back using ungetc

- Setting buffer size
  - Two functions for explicit control over the buffering performed on I/O to a file
  - Must be called before the first read or write on a file but after opening the file

    ```
    int setvbuf ( FILE *fd, char *buffer, int mode, int size );
    void setbuf ( FILE *fd, char *buffer );
    ```

    buffer – Contains the address to be used as the new buffer; if a NULL is passed, a new buffer is automatically created

    mode – Can be assigned values declared in stdio.h

    _IOFBF – Full buffering or block buffering

    * Characters as saved up and written as a block

    _IOLBF – Line buffering

    * Characters are saved up until a newline is encountered, or input is read from stdin, or the buffer is full

    _IONBF – No buffering

    * Information appears on the destination file or screen as soon as it is written

    size – Specifies the number of bytes to be contained in the buffer

  - setvbuf returns zero for success; non-zero for error
  - setbuf is similar to setvbuf except that if buffer is NULL, buffering is turned off; if buffer is not NULL, it is used with full buffering and a buffer size equal to BUFSIZ (declared in stdio.h)
  - Useful in debugging programs

    ```
    #if DEBUG
        setbuf ( stdout, NULL );
    #endif
    ```

### Unbuffered I/O

- Based on system calls
- Conceptually similar to those in the standard library[1]
- Low-level I/O is never buffered
- open system call
  - Open an unbuffered file
  - Invoked by

    ```
    #include <sys/types.h>
    #include <sys/stat.h>
    #include <fcntl.h>

    int fd = open ( char *file_name, int flags );        /* File already exists   */
    int fd = open ( char *file_name, int flags, int mode ); /* Create a new file */
    ```

    * File descriptor is an integer and not a pointer
    * file_name can be an absolute path or relative to the current directory

[1] Standard library functions are generally recommended for portability; System calls are more efficient and may be required in some cases, such as handling I/O for programs that create new processes

    - `flags` is an integer with each bit indicating the type of access; defined in `fcntl.h` or `sys/file.h` depending on ATT or BSD version of Unix, as follows

| | |
|---|---|
| `O_RDONLY` | Open for read |
| `O_WRONLY` | Open for write |
| `O_RDWR` | Open for read and write |
| `O_CREAT` | Create if file not found |
| `O_APPEND` | Write at end of file |
| `O_TRUNC` | Truncate existing file to zero length, if found |
| `O_EXCL` | Fail if file exists |

    - `mode` is the protection mode of the file; used only when the `O_CREAT` flag is set, otherwise ignored

  - Examples

```
int in_fd, out_fd;                        /* File descriptors      */
in_fd = open ( "infile", O_RDONLY, 0 );         /* Read only     */
out_fd = open ( "outfile", O_WRONLY|O_CREAT, 0666 );  /* Write */
```

    - Note that the permissions are specified as octal integer constant `0666`, and not as a decimal integer 666; the prefix zero is very important

- `creat` system call
  - Create a new file or truncate an existing one
  - Defined by

```
int creat ( char *filename, int permissions )
```

  - Returns the file descriptor of the created file, or -1 on error
  - The call

```
creat ( filename, mode );
```

  is equivalent to

```
open ( filename, O_WRONLY | O_CREAT | O_TRUNC, mode );
```

- `close` system call
  - Close the file
  - Frees the file descriptor for later use
  - Defined by

```
int close ( int fd );
```

- `read` system call
  - Read a block of data from file
  - Defined by

```
int read ( int fd, char *buffer, int num );
```

  - Returns the number of bytes read; zero if end of file is encountered; -1 if an error occurs

- `write` system call
  - Write a block of data to a file
  - Defined by

```
int write ( int fd, char *buffer, int num );
```

  - Returns the number of characters written
  - Error is indicated by the returned integer being less than `num`

### Designing file formats

- Important to include file type information with each file
- DOS does it by using an extension, such as `file.dat`
- Unix achieves the same by using a magic number
- *Magic number*
  - Identification number for the type of file
  - The `file(1)` command identifies the type of a file using, among other tests, a test for whether the file begins with a certain *magic number*
  - Magic number is specified in the file `/etc/magic` using four fields
    - Offset: A number specifying the offset, in bytes, into the file of data which is to be tested
    - Type: Type of data to be tested – byte, short (2-byte), long (4-byte), or string
    - Value: Expected value for file type
    - Message: Message to be printed if comparison succeeds
  - Used by the `C` compiler to distinguish between source, object, and assembly file formats
  - Developing magic numbers
    - Start with first four letters of program name (e.g., list)
    - Convert them to hex: 0x6c607374
    - Add 0x80808080 to the number
    - The resulting magic number is: 0xECE0F3F4
    - High bit is set on each byte to make the byte non-ASCII and avoid confusion between ASCII and binary files