File I/O

C file input and output are very similar to terminal input and output.

The `<stdio.h>` system header file provides file input and output functions.

The file input and ouput module define a private data structure, the `FILE`, which refers to an opened file. Because this is a private data structure, we use only pointers to it.

There are three `FILE *` available from `stdio.h`.

`stdin` This is the standard input, often connected to the terminal, from which input is obtained by default.

`stdout` This is the standard output, often connected to the terminal, to which output is directed by default.

`stderr` This is the standard error, to which diagnostic messages and errors should be directed.

`printf` prints to `stdout`.
`scanf` reads from `stdin`.

Opening Files

`FILE *fopen(const char *`*filename*`, const char *mode)`

`fopen` opens a file. The *filename* argument determines the file to be opened, and the *mode* argument determines whether the file will be read, written, or both.

- If the mode is `"r"`, the file is opened for reading. If it does not exist, `fopen` fails.
- If the mode is `"w"`, the file is opened for writing. If it does not exist, it is created. If it *does* exist, it is *truncated*.
- If the mode is `"a"`, the file is opened for appending. If it does not exist, it is created. If it *does* exist, further data will be *added on* to the end.

In the event of an error, `fopen` returns `NULL`. It is *very important* to test for this condition.

Error Reporting

```
extern int errno;
void perror(const char *str);
```

This function is useful in reporting errors to a user. It is particularly important in file I/O, where many different errors such as file not found, permission denied, and device full might be causing a function to fail.

`errno` is set to indicate the nature of a system call failure.

`fopen` may fail for, among others, the following reasons.

ENOENT No such file or directory

EACCES Permission denied

EISDIR Is a directory

ENOMEM Out of memory

Use `perror` to report errors to the user without needing to test for every possible error code.

`perror` will print the argument *str*, then a human-readable error string.

```
perror("Cannot open file");
```

The above should produce, for `EACCES`,

```
Cannot open file:  Permission denied
```

Testing for Files

The `fopen` modes that write to files do not report whether the file exists or not.

The `access` system call is one good way to determine whether a file exists. This function can be found in `<unistd.h>`.

`int access(const char *`*pathname*`, int` *mode*`);`
The `access` function determines whether *pathname* exists and whether it would be possible to read or write it. The precise test depends on the specified *mode*.

`F_OK` The function returns 0 if the file exists and -1 if it does not or if an error, such as a lack of permission on the parent directory, occurred.

`R_OK` The function returns 0 if the file exists and can be read, and -1 if it cannot or in the event of an error.

`W_OK` The function returns 0 if the file exists and can be written, and -1 if it cannot or in the event of an error.

```
void check_file(const char *filename)
{
  char str[256];

  if (access(filename, F_OK) == 0) {
    printf("File \"%s\" exists.\n", filename);
  } else if (errno == ENOENT) {
    snprintf(str, 256, "Cannot access file \"%s\"", filename);
    perror(str);
  }
}
```

Reading from Files

`int fscanf(FILE *`*file*`, const char *`*format*`, ...)`

fscanf is used just as **scanf** is used, except that *file* specifies the open file from which to read data.

scanf(...) is in fact identical to **fscanf(stdin, ...)**.

fscanf returns the number of input items successfully read.

```
float   x, y, z;
int     i;

i = fscanf(f, "%f %f %f", &x, &y, &z);
```

If the **fscanf** successfully reads three floating-point numbers from the file **f**, then **i** will be 3.

If **fscanf** cannot read at least three floating-point numbers, **i** would not be 3.
This might happen if **f** reached the end of the file or if an alphabetical character were encountered after two of the numbers.

Reading from Files

Below is a simple program that reads triplets of numbers from a data file and, considering them as a vector in 3-space, prints their length.

Note the care taken to test for error conditions when using file input and output routines.

```
#include <stdio.h>
#include <math.h>

int main(void)
{
  FILE  *f;
  int    n;
  float  x, y, z;

  f = fopen("vectors.dat", "r");
  if (f == NULL) {
    perror("Error opening \"vectors.dat\"");
    exit(1);
  }

  while (1) {
    n = fscanf(f, "%f %f %f", &x, &y, &z);

    if (n != 3)
      break;

    printf("%f\n", sqrt(x * x + y * y + z * z));
  }

  return 0;
}
```

Writing to Files

`int fprintf(FILE *`*file*`, const char *`*format*`, ...);`

`fprintf` is used just as `printf` is used, except that *file* specifies the open file to which to write data.

`printf(...)` is in fact identical to `fprintf(stdout, ...)`.

`fprintf` returns the number of output characters successfully written. If an error is encountered, a negative value will be returned.

`int fclose(FILE *`*file*`);`

`fclose` closes the file and guarantees that all outstanding writes are on the disk.

`fclose` can be called on any opened file to close it and release resources associated with the open file.

It is particularly important to call `fclose` on files being written and to test its return value. The data written to a file is not guaranteed to be safe on disk until `fclose` is successfully called on the file.

`fclose` should return 0. It is important to test that it is successful and does not instead return `EOF`, indicating an error.

Writing to Files

Below is a simple program that writes the integers from 1 to 10 and their squares into **`squares.dat`**.

```
#include <stdio.h>

int main(void)
{
  FILE *f;
  int   i, err;

  f = fopen("squares.dat", "w");
  if (f == NULL) {
    perror("Error opening \"squares.dat\"");
    exit(1);
  }

  for (i = 1; i <= 10; i++) {
    err = fprintf(f, "%d %d\n", i, i * i);
    if (err < 0) {
      perror("Error writing to \"squares.dat\"");
      exit(1);
    }
  }

  err = fclose(f);
  if (err != 0) {
    perror("Error closing \"squares.dat\"");
    exit(1);
  }

  return 0;
}
```

Random Access

`void rewind(FILE *`*file*`)`

This function "rewinds" *file* to start reading or writing from the beginning.

`long ftell(FILE *`*file*`)`

`ftell` indicates the offset between the current file position of *file* and the beginning of the file.

`long fseek(FILE *`*file*`, long` *offset*`, int` *whence*`)`

`fseek` is used to set the file position of *file* based on the *offset* and *whence* arguments.

whence can be one of the following.

`SEEK_SET` Set the position to *offset* from the start of the file.

`SEEK_CUR` Set the position to *offset* from the current position.

`SEEK_END` Set the position to *offset* from the end of the file.

`int feof(FILE *`*file*`)`

This function returns a true value if and only if the current file position of *file* is at the end of the file.

Reading a Line

`char *fgets(char *`*str*`, int` *size*`, FILE *`*file*`)`

`fgets` is used to read a line from a file safely.

`fgets` reads from *file* into *str*, up to and including a newline.

`fgets` will stop after reading *size*-1 characters even if a newline isn't encountered, leaving room for a terminating 0.

`gets` will read a line from the standard input, but will happily write past the end of a buffer if given a long enough line. Thus, use `fgets(stdin, ...)` and **not** `gets(...)`.

`fgets` will return NULL if it encounters an error or if the current file position is the end of the file.

```
#include <stdio.h>

int main(void)
{
  char  line[256];
  char *err;

  err = fgets(line, 256, stdin);
  if (err == NULL) {
    perror("Error reading from standard input");
    exit(1);
  }

  if (line[strlen(line) - 1] != '\n') {
    fprintf(stderr, "Input line too long, more than 254 characters\n");
    exit(1);
  }

  line[strlen(line) - 1] = '\0';
}
```

Binary I/O

The above functions all write data into text files.

There exist C functions which read and write binary representations of data from and to files.

- Text data files are easier to read and debug.
- Many UNIX utilities exist which handle text data files.

- Binary data files are often more compact than text data files.
- It is also much easier to access arbitrary offsets into arrays of numbers in binary format than in text format.

When designing file formats, it is often best to select a text representation of data.

Binary I/O

`size_t fread(void *`*ptr*`, size_t` *size*`, size_t` *nmemb*`, FILE *`*file*`)`
`size_t fwrite(const void *`*ptr*`, size_t` *size*`, size_t` *nmemb*`,`
`FILE *`*file*`)`

These functions are used to perform binary input and output, reading and writing binary images of an array of objects.

ptr points to the array of objects.

size is the size of a single object, and *nmemb* is the number of objects.

The functions return the number of items read or written. In the event of an error or end-of-file condition, a number of items smaller than *nmemb* will be returned.

To read or write a single object, take its address and treat it as an array of one object.

```
void copy_integer(FILE *in, FILE *out)
{
  int i, err;

  err = fread(&i, sizeof(int), 1, in);
  if (err != 1) {
    perror("Error reading data");
    return;
  }

  err = fwrite(&i, sizeof(int), 1, out);
  if (err != 1) {
    perror("Error writing data");
    return;
  }
}
```

Binary I/O

```
#include <stdio.h>
#include <stdlib.h>

typedef struct vector_struct {
  double x;
  double y;
  double z;
} vector;

static void bin_vector(FILE *in, FILE *out, int nvects);

int main(void)
{
  int     nvects;
  FILE   *in, *out;

  in = fopen("vector.dat", "r");
  if (in == NULL) {
    perror("Cannot read \"vector.dat\"");
    exit(1);
  }

  out = fopen("vector.bin", "w");
  if (out == NULL) {
    perror("Cannot write \"vector.bin\"");
    exit(1);
  }

  if (fscanf(in, "%d", &nvects) != 1) {
    perror("Could not read vector count from \"vector.dat\"");
    exit(1);
  }

  bin_vector(in, out, nvects);

  if (fclose(out) == EOF) {
    perror("Cannot write \"vector.bin\"");
  }

  return 0;
}
```

Binary I/O

```
/* Here we have
 *   in a file open for reading
 *   out a file open for binary writing
 *   nvects the number of vectors to work with.
 */
static void bin_vector(FILE *in, FILE *out, int nvects)
{
  int     i, err;
  vector *vects;

  vects = malloc(sizeof(vector) * nvects);

  for (i = 0; i < nvects; i++) {
    err = fscanf(in, "%lf %lf %lf", &vects[i].x, &vects[i].y, &vects[i].z);

    if (err < 3)
      break;
  }

  if (i < nvects) {
    fprintf(stderr, "Could not read all %d vectors\n", nvects);
    free(vects);
    return;
  }

  err = fwrite(vects, sizeof(vector), nvects, out);

  if (err < nvects) {
    fprintf(stderr, "Could not write all %d vectors\n", nvects);
  }

  free(vects);
}
```

Binary I/O

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

typedef struct vector_struct {
  double x;
  double y;
  double z;
} vector;

int main(int argc, char **argv)
{
  FILE   *bin;
  vector  v;
  int     err;

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

  bin = fopen(argv[1], "rb");
  if (bin == NULL) {
    perror("Cannot read input file");
    exit(1);
  }

  while (1) {
    err = fread(&v, sizeof(vector), 1, bin);
    if (err < 1)
      break;

    printf("%f\n", sqrt(v.x * v.x + v.y * v.y + v.z * v.z));
  }

  return 0;
}
```