Standard I/O Library

Streams and FILE objects

- stream
  - stream of bytes
  - logical structure used to access physical file
  - when we open or create a file with the standard I/O library, we say that we have associated a stream with the file
- FILE object
  - when we open a stream, fopen returns a pointer to a FILE object

Streams and FILE objects

- contains all the information required by the standard I/O library to manage the stream
  - file descriptor
  - pointer to a buffer for the stream
  - size of the buffer
  - counter of the number of characters in the buffer
  - error flag
- predefined stream
  - stdin, stdout, stderr (<stdio.h>)

Buffering

- goal
  - use the minimum number of read and write system calls
- three types of buffering
  - fully buffered
    - actual I/O takes place when the standard I/O buffer is filled
    - files that reside on disk are normally fully buffered

Buffering

- line buffered
  - performs I/O when a newline character(‘\n’) is encountered on input or output
  - used on a stream when it refers to a terminal (standard input and standard output)
  - actual I/O might take place if we fill the buffer before writing a newline
  - whenever input is requested through the standard I/O library from unbuffered stream or from a line-buffered stream, all line-buffered output streams are flushed

Buffering

- unbuffered
  - does not buffer the characters
  - ex) standard error stream

stream buffering operations

- synopsis
  - #include <stdio.h>
  - int setbuf(FILE *stream, char *buf);
  - int setbuffer(FILE *stream, char *buf, size_t size);
  - int setlinebuf(FILE *stream);
  - int setvbuf(FILE *stream, char *buf, int mode , size_t size);
  - int fflush(FILE *stream);

stream buffering operations

- description
  - int setvbuf(FILE *stream, char *buf, int mode , size_t size);
    - may be used at any time on any open stream to change its buffer
    - mode parameter
      - _IONBF : unbuffered
      - _IOLBF : line buffered
      - _IOFBF : fully buffered

stream buffering operations

- buf parameter
  - should point to a buffer at least size bytes long (except for unbuffered files)
  - buf will be used instead of the current buffer
  - if NULL, only the mode is affected
- a new buffer will be allocated on the next read or write operation
- can only change the mode of a stream when it is not “active”
  - before any I/O
  - immediately after a call to fflush( )

stream buffering operations

- int setbuf(FILE *stream, char *buf);
  - setvbuf(stream, buf, buf ? _IOFBF : _IONBF, BUFSIZ);
  - BUFSIZ
    - default buffer size(8192 bytes)
- int setbuffer(FILE *stream, char *buf, size_t size);
  - the same as setbuf( ), except that the size of the buffer is up to the caller
- int setlinebuf(FILE *stream);
  - setvbuf(stream, (char *)NULL, _IOLBF, 0);

stream buffering operations

- fflush( )
  - forces a write of all buffered data for the given output or update stream via the stream's underlying write function
  - If the stream argument is NULL, fflush flushes all open output streams
  - return value
    - Upon successful completion 0 is returned
    - Otherwise, EOF is returned

opening a stream

- synopsis
  - #include <stdio.h>
  - FILE *fopen (const char *path, const char *mode);
  - FILE *fdopen (int fildes, const char *mode);
  - FILE *freopen (const char *path, const char *mode, FILE *stream);

opening a stream

- description
  - FILE *fopen (const char *path, const char *mode);
    - opens the file whose name is the string pointed to by path and associates a stream with it
    - mode
      - r : open file for reading
      - r+ : open for reading and writing
      - w : truncate file to zero length or create file for writing

opening a stream

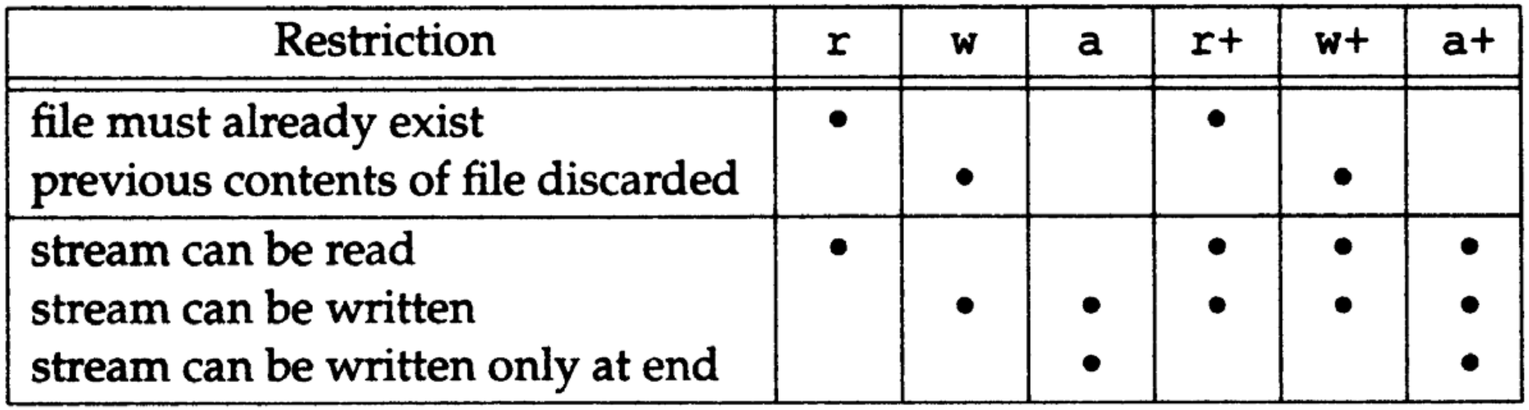
  - w+ : open for reading and writing. the file is created if it does not exist, otherwise it is truncated
  - a : open for writing. The file is created if it does not exist. The stream is positioned at the end of the file
  - a+ : open for reading and writing. The file is created if it does not exist. The stream is positioned at the end of the file
- Any created files will have mode 0666, as modified by the process' umask value

opening a stream

- FILE *fdopen (int fildes, const char *mode);
  - associates a stream with the existing file descriptor, fildes
- FILE *freopen (const char *path, const char *mode, FILE *stream);
  - opens the file whose name is the string pointed to by path and associates the stream pointed to by stream with it.
  - The original stream (if it exists) is closed.
  - The primary use of the freopen function is to change the file associated with a standard text stream (stderr, stdin, or stdout)

opening a stream

- return value
  - Upon successful completion fopen, fdopen and freopen return a FILE pointer. Otherwise, NULL is returned



Restriction	`r`	`w`	`a`	`r+`	`w+`	`a+`
file must already exist	●			●		
previous contents of file discarded		●			●	
stream can be read	●			●	●	●
stream can be written		●	●	●	●	●
stream can be written only at end			●			●

closing a stream

- synopsis
  - #include <stdio.h>
  - int fclose(FILE *stream);
- description
  - dissociates the named stream from its underlying file or set of functions
  - if the stream was being used for output, any buffered data is written first, using fflush( )

reading and writing a stream

- type of I/O operation
  - character-at-a-time I/O (fgetc, fputc)
  - line-at-a-time I/O (fgets, fputs)
  - direct I/O (binary I/O, fread, fwrite)
  - formatted I/O (fprintf, fscanf)

character-at-a-time I/O

- synopsis
  - #include <stdio.h>
  - int fgetc(FILE *stream);
  - int getc(FILE *stream);
  - int getchar(void);
  - int fputc(int c, FILE *stream);
  - int putc(int c, FILE *stream);
  - int putchar(int c);
  - int ungetc(int c, FILE *stream);

character-at-a-time I/O

- description
  - int fgetc(FILE *stream);
    - reads the next character from stream and returns it as an unsigned char cast to an int
    - returns EOF on end of file or error
  - int getc(FILE *stream);
    - equivalent to fgetc( ) except that it may be implemented as a macro
  - int getchar(void);
    - equivalent to getc(stdin)

character-at-a-time I/O

- int fputc(int c, FILE *stream);
  - writes the character c, cast to an unsigned char, to stream
- int putc(int c, FILE *stream);
  - equivalent to fputc( ) except that it may be implemented as a macro
- int putchar(int c);
  - equivalent to putc(c,stdout)
- return value
  - fputc( ), putc( ), putchar( ) return the character written as an unsigned char cast to an int or EOF on error

character-at-a-time I/O

- int ungetc(int c, FILE *stream);
  - pushes c back to stream, cast to unsigned char
  - it is available for subsequent read operation
  - only a single character can be pushed back
  - the character that we push back does not have to be the same character that was read
  - returns c on success, or EOF on error
- cf) int getw(FILE *stream);

  int putw(int c, FILE *stream);
  - the value returned on error is also a legitimate data value.

ferror( ), feof( ), clearerr( )

- synopsis
  - #include <stdio.h>

  - void clearerr(FILE *stream);
  - int feof(FILE *stream);
  - int ferror(FILE *stream);

ferror( ), feof( ), clearerr( )

- description
  - clearerr( )
    - clears the end-of-file and error indicators for the stream pointed to by stream
  - feof( )
    - tests the end-of-file indicator for the stream
    - return nonzero if it is set
  - ferror( )
    - tests the error indicator for the stream
    - returning non-zero if it is set

line-at-a-time I/O

- synopsis
  - #include <stdio.h>
  - char *fgets(char *s, int size, FILE *stream);
  - char *gets(char *s);
  - int fputs(const char *s, FILE *stream);
  - int puts(const char *s);

line-at-a-time I/O

- description
  - gets( )
    - reads a line from stdin into the buffer pointed to by s until either a terminating newline or EOF, which it replaces with '\0'.
    - No check for buffer overrun is performed
  - fgets( )
    - reads in at most one less than size characters from stream and stores them into the buffer pointed to by s

line-at-a-time I/O

  - Reading stops after an EOF or a newline. If a newline is read, it is stored into the buffer
  - '\0' is stored after the last character in the buffer
- fputs( )
  - writes the string s to stream, without its trailing '\0'.
- puts( )
  - writes the string s and a trailing newline to stdout.

line-at-a-time I/O

- return value
  - gets( ), fgets( )
    - return s on success, and NULL on error or when end of file occurs while no characters have been read.
  - puts( ), fputs( )
    - return a non - negative number on success, or EOF on error.

binary I/O

- synopsis
  - #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);

binary I/O

- description
  - fread( )
    - reads nmemb elements of data, each size bytes long, from the stream pointed to by stream, storing them at the location given by ptr
  - fwrite( )
    - writes nmemb elements of data, each size bytes long, to the stream pointed to by stream, obtaining them from the location given by ptr

binary I/O

- return value
  - return the number of items successfully read or written, not the number of characters
  - If an error occurs, or the end-of-file is reached, the return value is a short item count (or zero).
  - fread does not distinguish between end-of-file and error, and callers must use feof( ) and ferror( ) to determine which occurred.

binary I/O

- example

```
struct rec tmp;
struct rec item[10];

fread(&tmp, sizeof(struct rec), 1, fp);
fwrite(item, sizeof(struct rec), 10, fp);
```

formatted I/O - output

- synopsis
  - #include <stdio.h>

  - int printf(const char *format, ...);
  - int fprintf(FILE *stream, const char *format, ...);
  - int sprintf(char *str, const char *format, ...);
  - int snprintf(char *str, size_t size, const char *format, ...);

formatted I/O - output

- description
  - printf( )
    - write output to stdout, the standard output stream
  - fprintf( )
    - write output to the given output stream
  - sprintf( ), snprintf( )
    - write to the character string str
    - snprintf( ) does not write more than size bytes (including the trailing '\0')

formatted I/O - output

- return value
  - These functions return the number of characters printed (not including the trailing `\0' used to end output to strings)
  - negative value if output error
  - snprintf( ) returns -1 if the output was truncated due to size limit

formatted I/O - input

- synopsis
  - #include <stdio.h>
  - int scanf(const char *format, ...);
  - int fscanf(FILE *stream, const char *format, ...);
  - int sscanf(const char *str, const char *format, ...);
- description
  - scanf( )
    - reads input from the standard input stream stdin

formatted I/O - input

- fscanf( )
  - reads input from the stream pointer stream
- sscanf( )
  - reads its input from the character string pointed to by str
- return value
  - return the number of input items assigned
  - EOF if input error or end of file before any conversion

positioning a stream

- synopsis
  - #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);

positioning a stream

- description
  - fseek( )
    - sets the file position indicator for the stream pointed to by stream
    - SEEK_SET, SEEK_CUR, SEEK_END
    - successful call to the fseek( ) clears the end-of-file indicator for the stream and undoes any effects of the ungetc( )
    - return value
      - 0 if OK, -1 on error

positioning a stream

- ftell( )
  - obtains the current value of the file position indicator for the stream
  - return value
    - current file position indicator if OK
    - -1L on error
- rewind( )
  - fseek(stream, 0, SEEK_SET); clearerr(stream);

positioning a stream

- fgetpos( )
  - store the current value of the file offset into the object referenced by pos
- fsetpos( )
  - set the current value of the file offset into the object referenced by pos
- return value
  - 0 if OK, -1 on error

positioning a stream

- example

```
fpos_t pos;
FILE *fp = fopen(“bit”, “r”);
...
fgetpos(fp, &pos);
...
fsetpos(fp, &pos);
```

fileno()

- synopsis
  - #include <stdio.h>
  - int fileno(FILE *fp);

- description
  - examines the argument fp and returns the file descriptor associated with fp.

Temporary Files

- synopsis
  - #include <stdio.h>
  - char *tmpnam(char *s);
  - FILE *tmpfile (void);
  - char *tempnam(const char *dir, const char *pfx);
  - #include <stdlib.h>
  - char *mktemp(char *template);
  - int mkstemp(char *template);

Temporary Files

- description
  - char *tmpnam(char *s);
    - generates a unique temporary filename using the path prefix P_tmpdir defined in <stdio.h>
    - If the argument s is NULL, tmpnam( ) returns the address of an internal static area which holds the filename, which is overwritten by subsequent calls to tmpnam( )
    - If s is not NULL, the filename is returned in s
    - return value
      - a pointer to the unique temporary filename
      - NULL if a unique name cannot be generated

Temporary Files

- FILE *tmpfile(void);
  - generates a unique temporary filename using the path prefix P_tmpdir defined in <stdio.h>
  - the temporary file is then opened in binary read/write (w+b) mode
  - the file will be automatically deleted when it is closed or the program terminates
  - return value
    - returns a stream descriptor
    - NULL if a unique filename cannot be generated or the unique file cannot be opened

Temporary Files

- char *tempnam(const char *dir, const char *pfx)
  - generates a unique temporary filename using up to five characters of pfx, if it is not NULL
  - The directory to place the file is searched for in the following order
    - The directory specified by the environment variable TMPDIR, if it is writable.
    - The directory specified by the argument dir, if it is not NULL.
    - The directory specified by P_tmpdir.
    - The directory /tmp.

Temporary Files

  - The storage for the filename is allocated by malloc(), and so can be free'd by the function free().
  - return value
    - a pointer to the unique temporary filename
    - NULL if a unique filename cannot be generated.
- char *mktemp(char *template);
  - generates a unique temporary filename from template.
  - the last six characters of template must be “XXXXXX” and these these are replaced with a string that makes the filename unique.

Temporary Files

  - template must not be a constant string.
  - returns NULL on error, and template otherwise.
- int mkstemp(char *template);
  - generates a unique temporary filename from template like mktemp().
  - the file is then created with mode read/write.
  - returns the file descriptor of the temporary file or -1 on error

Variable Argument List

- synopsis
  - #include <stdarg.h>

  - va_list arglist;

  - void va_start(va_list ap, last);
  - type va_arg(va_list ap, type);
  - void va_end(va_list ap);

Variable Argument List

- description
  - va_start( )
    - initializes ap for subsequent use by va_arg and va_end, and must be called first
    - The parameter last is the name of the last parameter before the variable argument list, i.e., the last parameter of which the calling function knows the type
  - va_arg( )
    - an expression that has the type and value of the next argument in the call

Variable Argument List

  - Each call to va_arg modifies ap so that the next call returns the next argument
  - The first use of the va_arg macro after that of the va_start macro returns the argument after last
  - Successive invocations return the values of the remaining arguments.
- va_end( )
  - handles a normal return from the function whose variable argument list was initialized by va_start

Variable Argument List

- Example

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

int sum(int val, ...)
{
    va_list arglist;
    int arg, total;
    total = val;
    va_start(arglist, val);
    while ((arg = va_arg(arglist, int)) != 0)
        total += arg;
```

Variable Argument List

```
    va_end(arglist);
    return (total);
}

main()
{
    printf(“Total sum: %d\n”, sum(1,2,3,4,5,0));
}
```

Example – copy

```
#include <stdio.h>
#define MAXBUF 1024

main(int argc, char *argv[])
{
        FILE *source, *dest;
        char buf[MAXBUF];
        int count;
        if (argc != 3) {
            fprintf(stderr, “Usage: %s source destination\n”,
                              argv[0]);
            exit(1);
        }
```

Example - copy

```
if ((source = fopen(argv[1], “r”)) == NULL) {
        fprintf(stderr, “Can’t open %s :”, argv[1]);
        perror(“”);
        exit(1);
}
if ((dest = fopen(argv[2], “w”)) == NULL) {
        fprintf(stderr, “Can’t open %s :”, argv[2]);
        perror(“”);
        exit(1);
}
```

Example - copy

```
    while ((count = fread(buf, 1, MAXBUF, source))) {
            if (fwrite(buf, 1, count, dest) == 0) {
                        perror("fwrite");
                        exit(1);
            }
    }
    if (ferror(source)) {
            perror("fread");
            exit(1);
    }
    fclose(source);
    fclose(dest);
}
```