The old I/O library

Warning: This chapter describes classes that are *obsolete*. These classes are normally not available when libg++ is installed normally. The sources are currently included in the distribution, and you can configure libg++ to use these classes instead of the new iostream classes. This is only a temporary measure; you should convert your code to use iostreams as soon as possible. The iostream classes provide some compatibility support, but it is very incomplete (there is no longer a **File** class).

File-based classes

The **File** class supports basic IO on Unix files. Operations are based on common C stdio library functions.

File serves as the base class for istreams, ostreams, and other derived classes. It contains the interface between the Unix stdio file library and these more structured classes. Most operations are implemented as simple calls to stdio functions. **File** class operations are also fully compatible with raw system file reads and writes (like the system **read** and **lseek** calls) when buffering is disabled (see below). The **FILE*** stdio file pointer is, however maintained as protected. Classes derived from File may only use the IO operations provided by File, which encompass essentially all stdio capabilities.

The class contains four general kinds of functions: methods for binding **File**s to physical Unix files, basic IO methods, file and buffer control methods, and methods for maintaining logical and physical file status.

Binding and related tasks are accomplished via **File** constructors and destructors, and member functions **open**, **close**, **remove**, **filedesc**, **name**, **setname**.

If a file name is provided in a constructor or open, it is maintained as class variable **nm** and is accessible via **name**. If no name is provided, then **nm** remains null, except that **Files** bound to the default files stdin, stdout,

and stderr are automatically given the names **(stdin)**, **(stdout)**, **(stderr)** respectively. The function **setname**} may be used to change the internal name of the **File**. This does not change the name of the physical file bound to the File.

The member function **close** closes a file. The **~File** destructor closes a file if it is open, except that stdin, stdout, and stderr are flushed but left open for the system to close on program exit since some systems may require this, and on others it does not matter. **remove** closes the file, and then deletes it if possible by calling the system function to delete the file with the name provided in the **nm** field.

Basic IO

- **read** and **write** perform binary IO via stdio **fread** and **fwrite**.
- **get** and **put** for chars invoke stdio **getc** and **putc** macros.
- **put(const char* s)** outputs a null-terminated string via stdio **fputs**.
- **unget** and **putback** are synonyms. Both call stdio **ungetc**.

File Control

flush, **seek**, **tell**, and **tell** call the corresponding stdio functions.

flush(char) and **fill()** call stdio _**flsbuf** and _**filbuf** respectively.

setbuf is mainly useful to turn off buffering in cases where nonsequential binary IO is being performed. **raw** is a synonym for **setbuf(_IONBF)**. After a **f.raw()**, using the stdio functions instead of the system **read**, **write**, etc., calls entails very little overhead. Moreover, these become fully compatible with intermixed system calls (e.g., **lseek(f.filedesc(), 0, 0)**). While intermixing **File** and system IO calls is not at all recommended, this technique does allow the **File** class to be used in conjunction with other functions and libraries already set up to operate on file descriptors. **setbuf** should be called at most once after a constructor or open, but before any IO.

File Status

File status is maintained in several ways.

A **File** may be checked for accessibility via **is_open()**, which returns true if the File is bound to a usable physical file, **readable()**, which returns true if the File can be read from (opened for reading, and not in a _fail state), or **writable()**, which returns true if the File can be written to.

File operations return their status via two means: failure and success are represented via the logical state. Also, the return values of invoked stdio and system functions that return useful numeric values (not just failure/success flags) are held in a class variable accessible via **iocount**. (This is useful, for example, in determining the number of items actually read by the **read** function.)

Like the AT&T i/o-stream classes, but unlike the description in the Stroustrup book, p238, **rdstate()** returns the bitwise OR of **_eof**, **_fail** and **_bad**, not necessarily distinct values. The functions **eof()**, **fail()**, **bad()**, and **good()** can be used to test for each of these conditions independently.

_fail becomes set for any input operation that could not read in the desired data, and for other failed operations. As with all Unix IO, **_eof** becomes true only when an input operations fails because of an end of file. Therefore, **_eof** is not immediately true after the last successful read of a file, but only after one final read attempt. Thus,

for input operations, _**fail** and _**eof** almost always become true at the same time. **bad** is set for unbound files, and may also be set by applications in order to communicate input corruption. Conversely, _**good** is defined as 0 and is returned by
rdstate() if all is well.

The state may be modified via **clear(flag)**, which, despite its name, sets the corresponding state_value flag. **clear()** with no arguments resets the state to _**good**. **failif(int** *cond***)** sets the state to _**fail** only if *cond* is true.

Errors occuring during constructors and file opens also invoke the function **error**. **error** in turn calls a resetable error handling function pointed to by the non-member global variable **File_error_handler** only if a system error has been generated. Since **error** cannot tell if the current system error is actually responsible for a failure, it may at times print out spurious messages. Three error handlers are provided. The default, **verbose_File_error_handler** calls the system function **perror** to print the corresponding error message on standard error, and then returns to the caller. **quiet_File_error_handler** does nothing, and simply returns. **fatal_File_error_handler** prints the error and then aborts execution. These three handlers, or any other user-defined error handlers can be selected via the non-member function **set_File_error_handler**.

All read and write operations communicate either logical or physical failure by setting the _**fail** flag. All further operations are blocked if the state is in a _**fail** or _**bad** condition. Programmers must explicitly use **clear()** to reset the state in order to continue IO processing after either a logical or physical failure. C programmers who are unfamiliar with these conventions should note that, unlike the stdio library, **File** functions indicate IO success, status, or failure solely through the state, not via return values of the functions. The **void*** operator or **rdstate()** may be used to test success. In particular, according to c++ conversion rules, the **void*** coercion is automatically applied whenever the **File&** return value of any **File** function is tested in an **if** or **while**. Thus, for example, an easy way to copy all of stdin to stdout until eof (at which point **get** fails) or some error is **char c; while(cin.get(c) && cout.put(c));**.

The current version of istreams and ostreams differs significantly from previous versions in order to obtain compatibility with AT&T 1.2 streams. Most code using previous versions should still work. However, the following features of **File** are not incorporated in streams (they are still present in **File**): **scan(const char* fmt...)**,

remove(), **read()**, **write()**, **setbuf()**, **raw()**. Additionally, the feature of previous streams that allowed free intermixing of stream and stdio input and output is no longer guaranteed to always behave as desired.