Ch12: Advanced File Operations

- ASCII Data Files
  - fstream, ifstream, ofstream
  - File Modes
  - Stream Bit Flags
  - Reading and Writing
  - Function Parameters
- Binary Data Files
  - Opening
  - Reading and Writing

Data Files

- File is collection of data usually stored on external media
  - Data can be saved to files and then later reused
- Use file stream objects:
  - `include <fstream>`

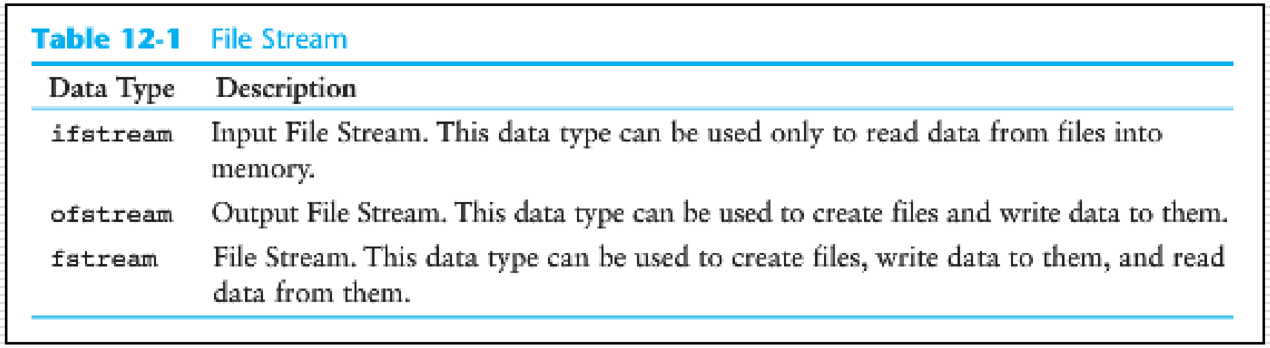


Table 12-1 File Stream

Data Type	Description
`ifstream`	Input File Stream. This data type can be used only to read data from files into memory.
`ofstream`	Output File Stream. This data type can be used to create files and write data to them.
`fstream`	File Stream. This data type can be used to create files, write data to them, and read data from them.

File Streams

- ifstream and ofstream have default open modes

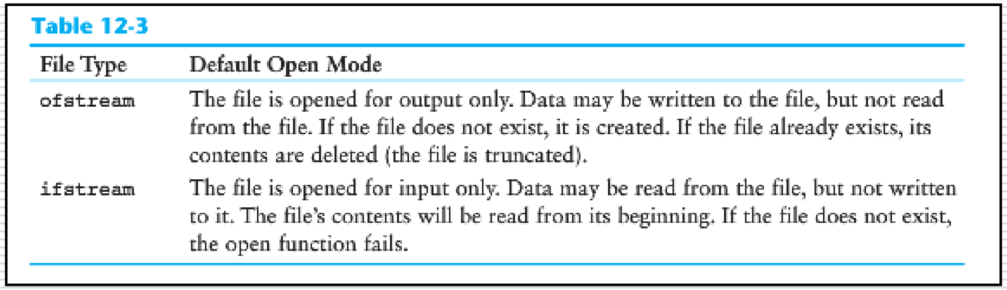


Table 12-3

File Type	Default Open Mode
ofstream	The file is opened for output only. Data may be written to the file, but not read from the file. If the file does not exist, it is created. If the file already exists, its contents are deleted (the file is truncated).
ifstream	The file is opened for input only. Data may be read from the file, but not written to it. The file's contents will be read from its beginning. If the file does not exist, the open function fails.

  - Creating and opening file for input

```
ifstream inputFile;
inputFile.open(“Customers.txt”);
inputFile.open(“C:\\temp\\Customers.txt”);
```

  - Creating and opening file for output

```
ofstream outputFile;
outputFile.open(“C:\\logfile.txt”);
```

fstream File Stream

- Creating and opening file for **input**

```
fstream inputFile;
inputFile.open("Customers.txt", ios::in);
--OR--
fstream inputFile("Customers.txt", ios::in);
```

- Creating and opening file for **output**

```
fstream outputFile;
outputFile.open("logfile.txt", ios::out);
--OR--
fstream outputFile("logfile.txt", ios::out);
```

fstream File Modes

- Combine with | operator
  - `ios:out | ios::app`

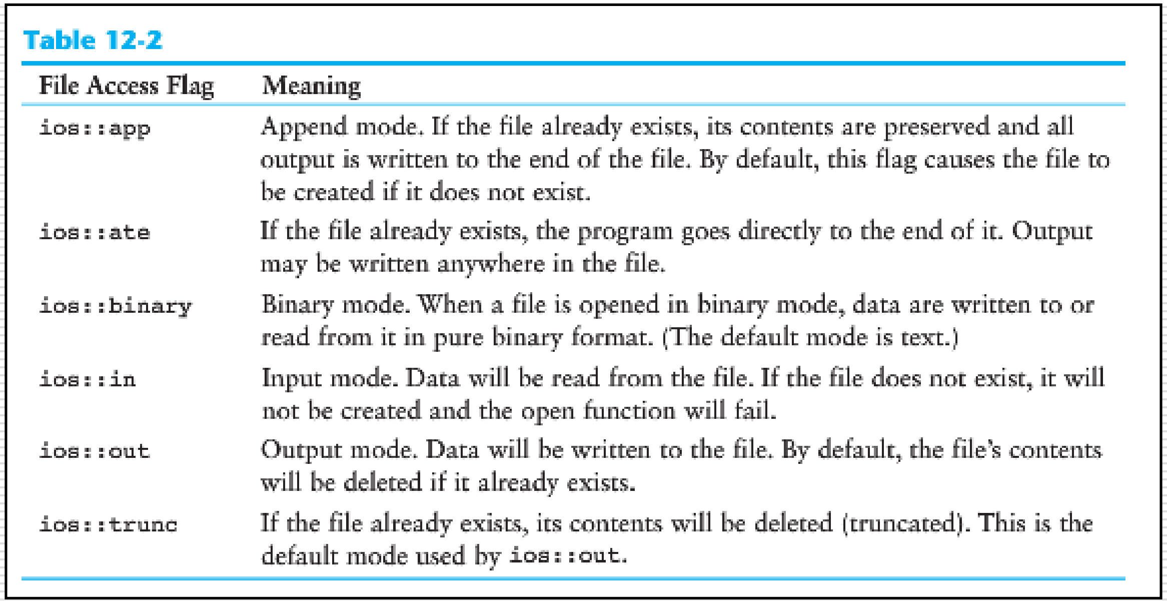


Table 12-2

File Access Flag	Meaning
`ios::app`	Append mode. If the file already exists, its contents are preserved and all output is written to the end of the file. By default, this flag causes the file to be created if it does not exist.
`ios::ate`	If the file already exists, the program goes directly to the end of it. Output may be written anywhere in the file.
`ios::binary`	Binary mode. When a file is opened in binary mode, data are written to or read from it in pure binary format. (The default mode is text.)
`ios::in`	Input mode. Data will be read from the file. If the file does not exist, it will not be created and the open function will fail.
`ios::out`	Output mode. Data will be written to the file. By default, the file's contents will be deleted if it already exists.
`ios::trunc`	If the file already exists, its contents will be deleted (truncated). This is the default mode used by `ios::out`.

File Streams

- Error states are maintained for every stream via bit flags

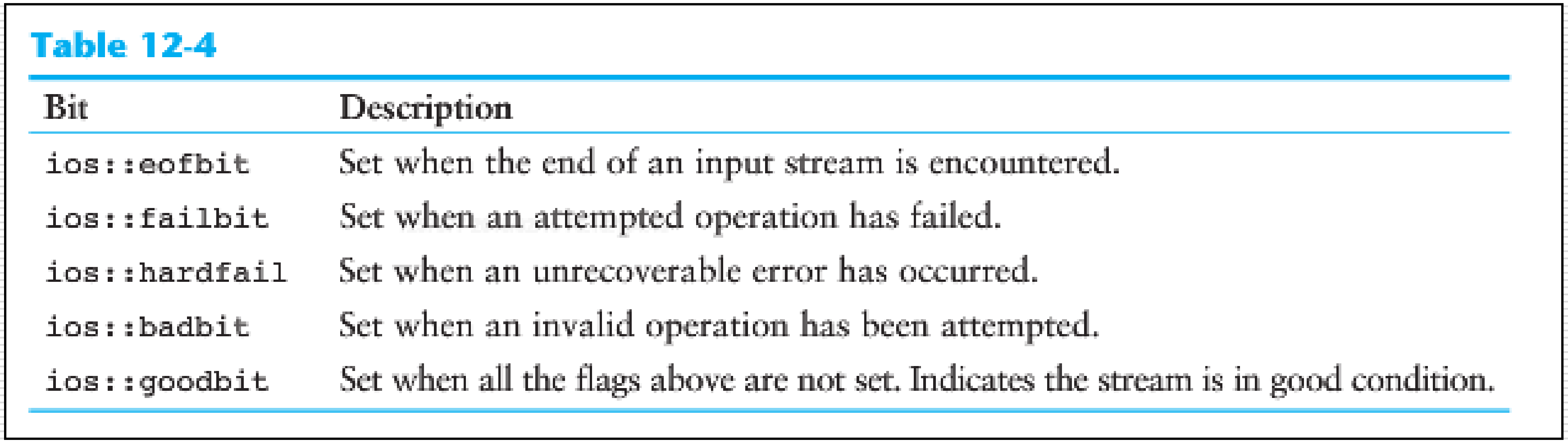


Table 12-4

Bit	Description
`ios::eofbit`	Set when the end of an input stream is encountered.
`ios::failbit`	Set when an attempted operation has failed.
`ios::hardfail`	Set when an unrecoverable error has occurred.
`ios::badbit`	Set when an invalid operation has been attempted.
`ios::goodbit`	Set when all the flags above are not set. Indicates the stream is in good condition.

File Streams

- Bit flags tested by stream functions

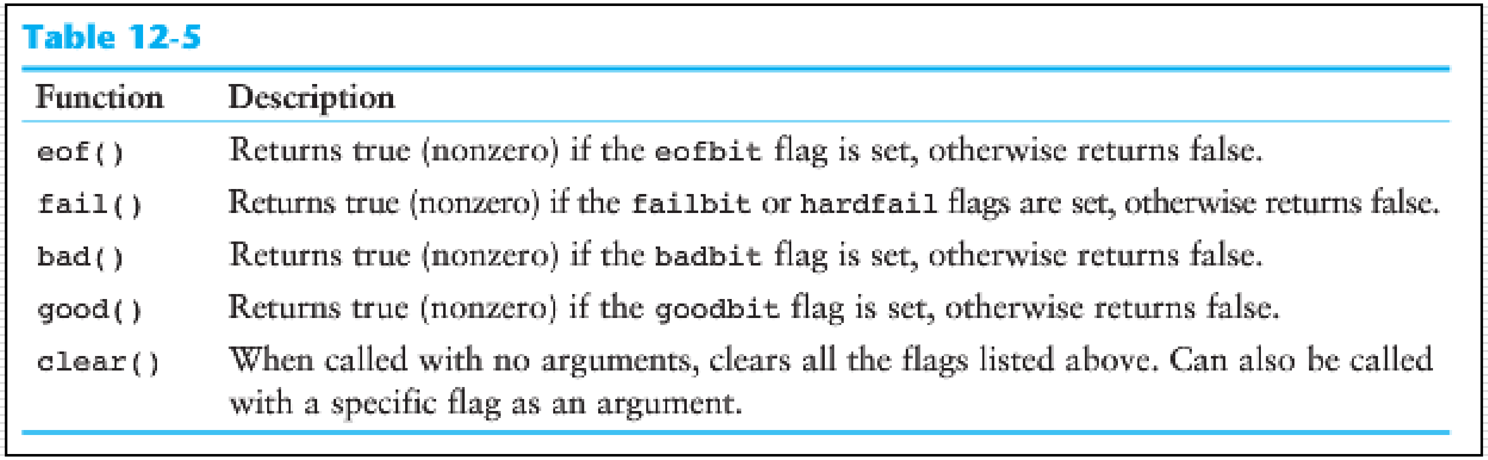


Table 12-5

Function	Description
`eof()`	Returns true (nonzero) if the `eofbit` flag is set, otherwise returns false.
`fail()`	Returns true (nonzero) if the `failbit` or `hardfail` flags are set, otherwise returns false.
`bad()`	Returns true (nonzero) if the `badbit` flag is set, otherwise returns false.
`good()`	Returns true (nonzero) if the `goodbit` flag is set, otherwise returns false.
`clear()`	When called with no arguments, clears all the flags listed above. Can also be called with a specific flag as an argument.

File Streams

- Bit flags tested by stream functions

```
 void showState(fstream &file) {
   cout << "File Status:\n";
   cout << " eof bit: " << file.eof() << endl;
   cout << " fail bit: " << file.fail() << endl;
   cout << " bad bit: " << file.bad() << endl;
   cout << " good bit: " << file.good() << endl;
   file.clear();     // Clear any bad bits
}
```

File Streams

- Testing for open errors before read/write
  - Using stream status

```
if (!inputFile) {
   cout << “Error!” << endl;
    exit(-1);
}
```

  - Using stream function

```
if (inputFile.fail()) {
   cout << “Error!” << endl;
    exit(-1);
}
```

File Streams

- Reading from file stream
  - Numerical
    - `inputFile >> myInt >> myFloat;`
  - Character and String
    - Skips preceding whitespace; stops at whitespace
      - `inputFile >> myChar >> myString;`
    - Reads whitespace
      - `inputFile.get(myChar);`
      - `getline(inputFile, myString, ‘\n’);`

File Streams

- Writing to file stream
  - Numerical, Character, and String

    -
      ```
      outputFile << myInt << myFloat
           << myChar << myString;
      ```
  - Character

    -
      ```
      outputFile.put(myChar);
      ```
  - Formatting output

    -
      ```
      outputFile << showpoint << fixed
           << setprecision(2) << right;
      ```

    -
      ```
      outputFile << setw(20) << myFloat
           << endl;
      ```

File Streams

- Looping to read from file
  - Using stream status

```
while (inputFile >> name) {
    cout << name << endl;
}
```

  - Using stream function

```
while (!inputFile.eof()) {
    inputFile >> name;
     cout << name << endl;
}
```

File Streams

- Closing a file
  - flush pending output and disassociate program stream variable from physical file
    - `inputFile.close();`
    - `outputFile.close();`

File Streams

- As function parameter
  - Use pass by reference type because internal state changes with almost every operation
    - `fstream&`
    - `ifstream&`
    - `ofstream&`

File Streams

- As function parameter:

```
bool openFileIn(fstream &file, string name) {
    bool status;
    file.open(name, ios::in);
    if (file.fail())
        status = false;
    else
        status = true;
    return status;
}
```

```
if (openFileIn(dataFile, “demoFile.txt”)) {
    cout << “File successfully opened!” << endl;
    showContents(dataFile);
    dataFile.close();
}
```

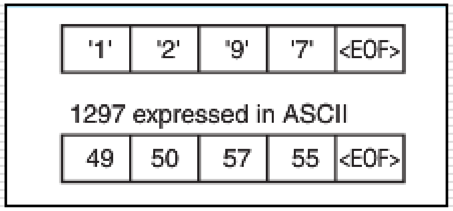
File Streams

- As function parameter:

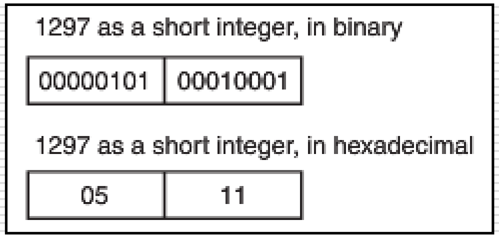
```
void showContents(fstream &file) {
   string line;
   while (file >> line)
   {
      cout << line << endl;
   }
}
```

Binary Files

- Contain data not necessarily stored as ASCII text



Number 1297 represented in ASCII format


1297 as a short integer, in binary
00000101
00010001
1297 as a short integer, in hexadecimal
05
11

Number 1297 represented in binary/hex format

Binary Files

- Use mode argument to open file in binary format
  - `fstream inputFile("Customers.txt", ios::in | ios::binary);`
  - `fstream outputFile("logfile.txt", ios::out | ios::binary);`

Binary Files

- Use write member function to write binary data to file
  - `fileObject.write(address, size);`
    - fileObject is name of file stream object
    - address is the starting address of the section of memory that is to be written to the file
      - expected to be the address of a char
    - size is number of bytes of memory to write
      - must be an integer value

Binary Files

- Use write member function to *write* binary data to file
  - Example (single character):
    ```
    char cVar = ‘A’;
    outFile.write(&cVar, sizeof(cVar));
    ```
  - Example (multiple characters):
    ```
    char cData[] = {‘A’, ‘B’, ‘C’, ‘D’};
    outFile.write(cData, sizeof(cData));
    ```
  - Example (integer):
    ```
    int iVar = 1;
    outFile.write(reinterpret_cast
         <char *>(&iVar), sizeof(iVar));
    ```

Binary Files

- Use read member function to read binary data from file
  - `fileObject.read(address, size);`
    - fileObject is name of file stream object
    - address is the starting address where the data being read from the file is to be stored
      - expected to be the address of a char
    - size is number of bytes of memory to read
      - must be an integer value

Binary Files

- Use read member function to read binary data from file
  - Example (single character):
    ```
    char cVar;
    inFile.read(&cVar, sizeof(cVar));
    ```
  - Example (multiple characters):
    ```
    char cData[] = {‘A’, ‘B’, ‘C’, ‘D’};
    inFile.read(cData, sizeof(cData));
    ```
  - Example (integer):
    ```
    int iVar = 1;
    intFile.read(reinterpret_cast
         <char *>(&iVar), sizeof(iVar));
    ```