# 12 STREAMS

## Streams

►A ***stream*** is a general name given to a flow of data in an input/output situation. For this reason, streams in C++ are often called ***iostreams***.

►An **iostream** can be represented by an object of a particular class.

►For example, you've already seen numerous examples of the ***cin*** and ***cout*** stream objects used for input and output.

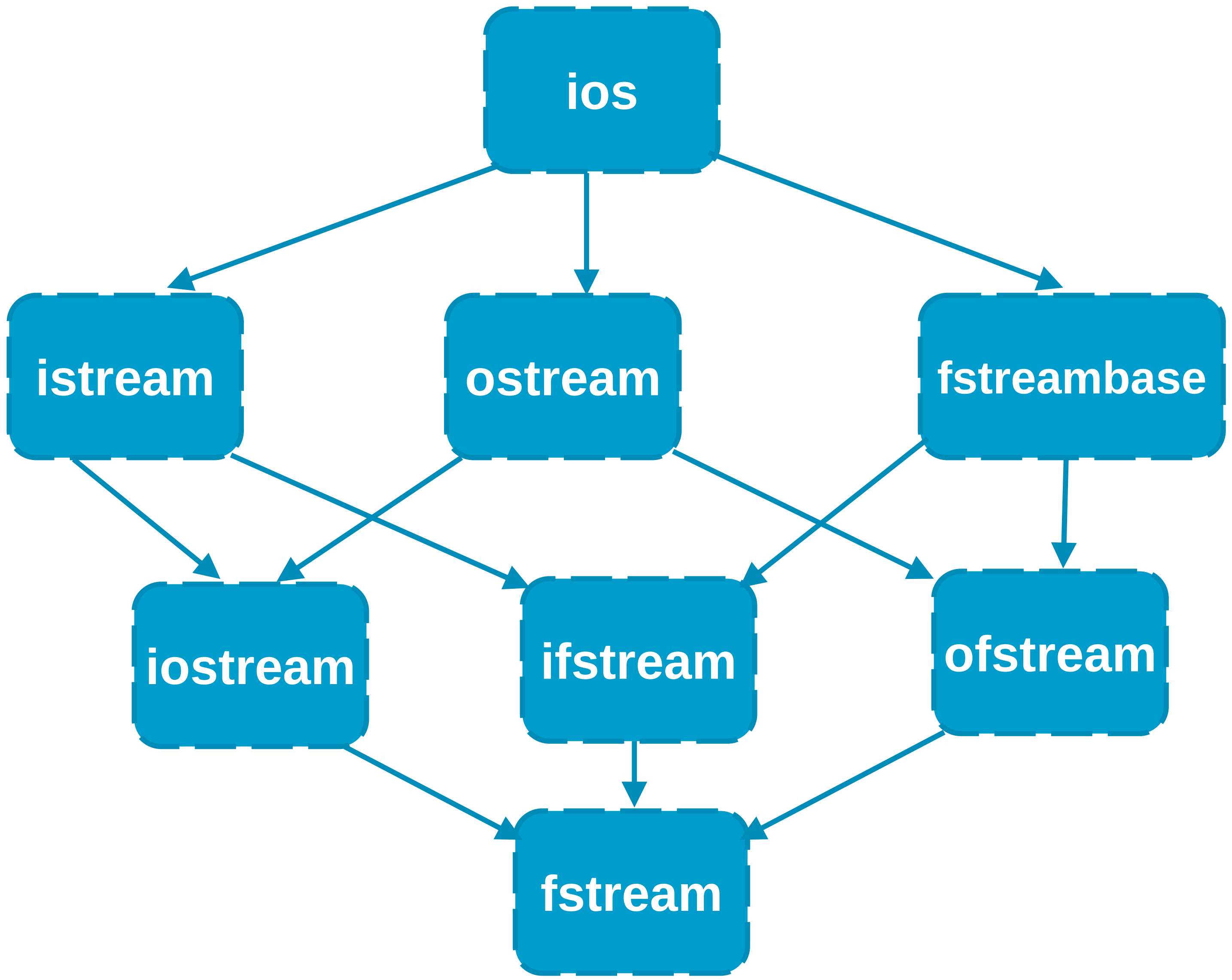
## Advantages of Streams

►Old-fashioned C programmers may wonder what advantages there are to using the stream classes for I/O instead of traditional C functions such as printf() and scanf() and—for files—fprintf(), fscanf(), and so on.

►One reason is that the stream classes are less prone to errors. If you've ever used a %d formatting character when you should have used a %f in printf(), you'll appreciate this. There are no such formatting characters in streams, because each object already knows how to display itself. This removes a major source of program bugs.

►Second, you can overload existing operators and functions, such as the insertion (<<) and extraction (>>) operators, to work with classes you create. This makes your classes work in the same way as the built-in types, which again makes programming easier and more error free (not to mention more aesthetically satisfying).

## Stream Class Hierarchy



## Stream Class Hierarchy

- The **ios** class is the base class for the **iostream** hierarchy.
  - –contains many constants and member functions common to input and output operations of all kinds.
  - –also contains a pointer to the **streambuf** class, which contains the actual memory buffer into which data is read or written and the low-level routines for handling this data.

## Stream Class Hierarchy

- The **istream** and **ostream** classes are derived from **ios** and are dedicated to input and output, respectively.
- The **istream** class contains such member functions as get(), getline(), read(), and the extraction (») operators, whereas **ostream** contains put() and write() and the insertion («) operators.
- The **iostream** class is derived from both **istream** and **ostream** by multiple inheritance.
  - used with devices, such as disk files, that may be opened for both input and output at the same time.

## Stream Class Hierarchy

- ► The **ifstream** class is used for creating input file objects
- ► The **ofstream** class is used for creating input file objects is used for creating output file objects.
- ► To create a read/write file the **fstream** class should be used.

## ios

►The ios class is the grand daddy of all the stream classes and contains the majority of the features you need to operate C++ streams.

►The three most important features are

- the formatting flags,
- the error-status bits,
- the file operation mode.

We'll look at formatting flags and error-status bits now.

## Formatting Flags

Formatting flags are a set of enum definitions in ios. They act as on/off switches that specify choices for various aspects of input and output format and operation.

| | |
|---|---|
| skipws | Skip (ignore) whitespace on input. |
| left | Left adjust output. |
| right | Right adjust output. |
| dec | Convert to decimal. |
| oct | Convert to octal. |
| hex | Convert to hexadecimal. |
| showbase | Use base indicator on output (0 for octal, 0x for hex). |
| showpoint | Show decimal point on output. |
| uppercase | Use uppercase X, E, and hex output letters ABCDEF. |
| showpos | Display ‘+’ before positive integers. |
| scientific | Use exponential format on floating-point output [9.1234E2]. |
| fixed | Use fixed format on floating-point output [912.34]. |
| unitbuf | Flush all streams after insertion. |

## Formatting Flags

►There are several ways to set the formatting flags, and different flags can be set in different ways. Because they are members of the ios class, flags must usually be preceded by the name ios and the scope-resolution operator (e.g., ios::skipws). All the flags can be set using the **setf**() and **unsetf**() ios member functions.

►For example,

```
cout.setf(ios::left); //left justify output text
cout >> "This text is left-justified";
cout.unsetf(ios::left); //return to default
                        //(right justified)
```

►Many formatting flags can be set using manipulators, so let's look at them now.

## Manipulators

►Manipulators are formatting instructions inserted directly into a stream.

►You've seen examples before, such as the manipulator ***endl***, which sends a new line to the stream and flushes it:

```
cout << "To each his own." << endl;
```

►There is also used the setiosflags() manipulator:

```
cout << setiosflags(ios::fixed)  // use fixed decimal point
     << setiosflags(ios::showpoint) //always show decimal point
     << var;
```

## No-argument ios Manipulators

| | |
|---|---|
| ws | Turn on whitespace skipping on input |
| dec | Convert to decimal |
| oct | Convert to octal |
| hex | Convert to hexadecimal |
| endl | Insert new line and flush the output stream |
| ends | Insert null character to terminate an output string |
| flush | Flush the output stream |
| lock | Lock file handle |
| unlock | Unlock file handle |

You insert these manipulators directly into the stream. e.g., to output var in hexadecimal format, you can say

```
cout << hex << var;
```

## ios Manipulators with Arguments

► Manipulators that take arguments affect only the next item in the stream.

► For example, if you use ***setw*** to set the width of the field in which one number is displayed, you'll need to use it again for the next number.

| | | |
|---|---|---|
| setw() | field width (int) | Set field width for output |
| setfill() | fill character (int) | Set fill character for output (default is a space) |
| setprecision() | precision (int) | Set precision (number of digits displayed) |
| setiosflags() | formatting flags (long) | Set specified flags |
| resetiosflags() | formatting flags (long) | Clear specified flags |

## Functions

▶The ios class contains a number of functions that you can use to set the formatting flags and perform other tasks.

▶Most of these functions are shown below:

| | |
|---|---|
| ch=fill() | Return the fill character (fills unused part of field; default is space). |
| fill(ch) | Set the fill character. |
| p=precision() | Get the precision (number of digits displayed for floating point). |
| precision(p) | Set the precision. |
| w=width() | Get the current field width (in characters). |
| width(w) | Set the current field width. |
| setf(flags) | Set specified formatting flags (e.g., ios::left). |
| unsetf(flags) | Unset specified formatting flags. |

►These functions are called for specific stream objects using the normal dot operator. For example, to set the field width to 14, you can say

```
cout.width(14);
```

►Similarly, the following statement sets the fill character to an asterisk (as for check printing):

```
cout.fill('*');
```

►You can use several functions to manipulate the ios formatting flags directly.

For example, to set left justification, use

```
cout.setf(ios::left);
```

To restore right justification, use

```
cout.unsetf(ios::left);
```

## istream

The istream class, which is derived from ios, performs input-specific activities.

***istream*** functions:

| | |
|---|---|
| **>>** | Formatted extraction for all basic (and overloaded) types. |
| **get(ch)** | Extract one character into ch. |
| **get(str)** | Extract characters into array str, until '\0'. |
| **get(str, MAX)** | Extract up to MAX characters into array. |
| **get(str, DELIM)** | Extract characters into array str until specified delimiter (typically '\n'). Leave delimiting char in stream. |

## *istream* Functions

| | |
|---|---|
| get(str, MAX, DELIM) | Extract characters into array str until MAX characters or the DELIM character. Leave delimiting char in stream |
| getline(str, MAX, DELIM) | Extract characters into array str until MAX characters or the DELIM character. Extract delimiting character |
| putback(ch) | Insert last character read back into input stream |
| ignore(MAX, DELIM) | Extract and discard up to MAX characters until (and including) the specified delimiter (typically '\n') |
| peek(ch) | Read one character, leave it in stream |
| count = gcount() | Return number of characters read by a (immediately preceding) call to get(), getline(), or read() |
| read(str, MAX) | For files. Extract up to MAX characters into str until EOF |
| seekg(position) | Sets distance (in bytes) of file pointer from start of file |
| seekg(position, seek_dir) | Sets distance (in bytes) of file pointer from specified place in file: seek_dir can be ios::beg, ios::cur, ios::end |
| position = tellg(pos) | Return position (in bytes) of file pointer from start of file |

## *ostream*

The ostream class handles output or insertion activities.

**ostream functions:**

| | |
|---|---|
| << | Formatted insertion for all basic (and overloaded) types. |
| put(ch) | Insert character ch into stream. |
| flush() | Flush buffer contents and insert new line. |
| write(str, SIZE) | Insert SIZE characters from array str into file. |
| seekp(position) | Sets distance in bytes of file pointer from start of file. |
| seekp(position, seek_dir) | Set distance in bytes of file pointer from specified place in file. seek_dir can be ios::beg, ios::cur, or ios::end. |
| position = tellp() | Return position of file pointer, in bytes. |

## Ostream and _withassign Classes

►The iostream class, which is derived from both istream and ostream, acts only as a base class from which other classes, specifically iostream_withassign, can be derived.

►It has no functions of its own (except constructors and destructors). Classes derived from iostream can perform both input and output.

►There are three _withassign classes:

istream_withassign, derived from istream
ostream_withassign, derived from ostream
iostream_withassign, derived from iostream

►These _withassign classes are much like those they're derived from except they include overloaded assignment operators so their objects can be copied.

## Predefined Stream Objects

| Objects Name | Class | Used for |
|---|---|---|
| cin | istream_withassign | Keyboard input |
| cout | ostream_withassign | Normal screen output |
| cerr | ostream_withassign | Error output |
| clog | ostream_withassign | Log output |

The cerr object is often used for error messages and program diagnostics. Output sent to cerr is displayed immediately, rather than being buffered, as output sent to cout is. Also, output to cerr cannot be redirected. For these reasons, you have a better chance of seeing a final output message from cerr if your program dies prematurely. Another object, clog, is similar to cerr in that it is not redirected, but its output is buffered, whereas cerr's is not.

## Stream Errors

What happens if a user enters the string "nine" instead of the integer 9, or pushes ENTER without entering anything? What happens if there's a hardware failure? We'll explore such problems in this session. Many of the techniques you'll see here are applicable to file I/O as well.

## Error-Status Bits

The stream error-status bits (error byte) are an ios member that report errors that occurred in an input or output operation.

| | |
|---|---|
| goodbit | No errors (no bits set, value = 0). |
| eofbit | Reached end of file. |
| failbit | Operation failed (user error, premature EOF). |
| badbit | Invalid operation (no associated streambuf). |
| hardfail | Unrecoverable error. |

Various ios functions can be used to read (and even set) these error bits.

| | |
|---|---|
| int = eof(); | Returns true if EOF bit set. |
| int = fail(); | Returns true if fail bit or bad bit or hard-fail bit set. |
| int = bad(); | Returns true if bad bit or hard-fail bit set. |
| int = good(); | Returns true if everything OK; no bits set. |
| clear(int=0); | With no argument, clears all error bits; otherwise sets specified bits, as in clear(ios::failbit). |

inp.cpp

```
#include <iostream>
int main() {
    int i;
    char ok=0;
    while(!ok) {                          // cycle until input OK
        cout << "\nEnter an integer: ";
        cin >> i;
        if( cin.good() ) ok=1;                // if no errors
        else {
          cin.clear();                    // clear the error bits
          cout << "Incorrect input";
          cin.ignore(20, '\n');               // remove newline
        }
    }
    cout << "integer is " << i; // error-free integer
}
```

## No-Input Input

- ►Whitespace characters, such as TAB, ENTER , and '\n', are normally ignored (skipped) when inputting numbers. This can have some undesirable side effects. For example, users, prompted to enter a number, may simply press the key without typing any digits. Pressing ENTER causes the cursor to drop down to the next line while the stream continues to wait for the number.
- ►What's wrong with the cursor dropping to the next line?
  - –First, inexperienced users, seeing no acknowledgment when they press , may assume the computer is broken.
  - –Second, pressing repeatedly normally causes the cursor to drop lower and lower until the entire screen begins to scroll upward.
- ►Thus it's important to be able to tell the input stream *not* to ignore whitespace. This is done by clearing the skipws flag:

```
cout << "\nEnter an integer: ";
cin.unsetf(ios::skipws);        // don't ignore whitespace
cin >> i;
if( cin.good() )
   {
   // no error
   }
  // error
```

Now if the user types without any digits, failbit will be set and an error will be generated. The program can then tell the user what to do or reposition the cursor so the screen does not scroll.

## Disk File I/O with Streams

► Disk files require a different set of classes than files used with the keyboard and screen. These are ifstream for input, fstream for input and output, and ofstream for output. Objects of these classes can be associated with disk files and you can use their member functions to read and write to the files.

► The ifstream, ofstream, and fstream classes are declared in the FSTREAM.H file.

► This file also includes the IOSTREAM.H header file, so there is no need to include it explicitly;

► FSTREAM.H takes care of all stream I/O.

```
#include <fstream.h>                     // for file I/O
int main(){
   char ch = 'x';                        // character
   int j = 77;                           // integer
   double d = 6.02;                      // floating point
   char str1[] = "Kafka";                // strings
   char str2[] = "Proust";               // (no embedded spaces)
   ofstream outfile("fdata.txt");        // create ofstream object
   outfile << ch                         // insert (write) data
            << j   <<  ' '               // needs space between numbers
            << d
            << str1 << ' '               // needs space between strings
            << str2;
}
```

Here the program defines an object called outfile to be a member of the ofstream class. At the same time, it initializes the object to the file name FDATA.TXT. This initialization sets aside various resources for the file, and accesses or *opens* the file of that name on the disk. If the file doesn’t exist, it is created. If it does exist, it is truncated and the new data replaces the old. The outfile object acts much as cout did in previous programs, so the insertion operator (<<) is used to output variables of any basic type to the file. This works because the insertion operator is appropriately overloaded in ostream, from which ofstream is derived.

When the program terminates, the outfile object goes out of scope. This calls its destructor, which closes the file, so you don’t need to close the file explicitly.

You must separate numbers (such as 77 and 6.02) with nonnumeric characters. Because numbers are stored as a sequence of characters rather than as a fixed-length field, this is the only way the extraction operator will know, when the data is read back from the file, where one number stops and the next one begins. Second, strings must be separated with whitespace for the same reason. This implies that strings cannot contain embedded blanks. In this example, I use the space character (“ “) for both kinds of delimiters. Characters need no delimiters, because they have a fixed length.

## Reading Data

Any program can read the file generated by previous program by using an ifstream object that is initialized to the name of the file. The file is automatically opened when the object is created. The program can then read from it using the extraction (>>) operator.

```
// reads formatted output from a file, using >>
#include <fstream.h>
const int MAX = 80;
int main(){
   char ch;                          // empty variables
   int j;
   double d;
   char str1[MAX];
   char str2[MAX];
   ifstream infile("fdata.txt");     // create ifstream object
   infile >> ch >> j >> d >> str1 >> str2;  // extract data from it
   cout << ch << endl                // display the data
        << j << endl
        << d << endl
        << str1 << endl
        << str2 << endl;
}
```

## Detecting End-OF-File

► Objects derived from ios contain error-status bits that can be checked to determine the results of operations. When you read a file little by little, you will eventually encounter an end-of-file condition. The EOF is a signal sent to the program from the hardware when there is no more data to read. The following construction can be used to check for this:

```
        while( !infile.eof() )   // until eof encountered
```

► However, checking specifically for an eofbit means that I won't detect the other error bits, such as the failbit and badbit, which may also occur, although more rarely. To do this, I could change the loop condition:

```
while( infile.good() )   // until any error encountered
```

► But even more simply, I can test the stream directly

```
    while( infile )        // until any error encountered
```

Any stream object, such as infile, has a value that can be tested for the usual error conditions, including EOF. If any such condition is true, the object returns a zero value.

► If everything is going well, the object returns a nonzero value. This value is actually a pointer, but the "address" returned has no significance except to be tested for a zero or nonzero value.

## Binary I/O

You can write a few numbers to disk using formatted I/O, but if you're storing a large amount of numerical data, it's more efficient to use binary I/O in which numbers are stored as they are in the computer's RAM memory rather than as strings of characters. In binary I/O an integer is always stored in 2 bytes, whereas its text version might be 12345, requiring 5 bytes. Similarly, a float is always stored in 4 bytes, whereas its formatted version might be 6.02314e13, requiring 10 bytes.

The next example shows how an array of integers is written to disk and then read back into memory using binary format. I use two new functions: write(), a member of ofstream, and read(), a member of ifstream. These functions think about data in terms of bytes (type char). They don't care how the data is formatted, they simply transfer a buffer full of bytes from and to a disk file. The parameters to write() and read() are the address of the data buffer and its length. The address must be cast to type char, and the length is the length in bytes (characters), *not* the number of data items in the buffer.

## Example

```
#include <fstream.h>          // for file streams
const int MAX = 100;          // number of ints
int buff[MAX];                // buffer for integers
int main() {
    int j;
    for(j=0; j<MAX; j++)  // fill buffer with data
    buff[j] = j; // (0, 1, 2, ...)
    ofstream os("edata.dat", ios::binary);     // create output stream
    os.write( (char*)buff, MAX*sizeof(int) );  // write to it
    os.close(); // must close it
    for(j=0; j<MAX; j++)      // erase buffer
        buff[j] = 0;
    ifstream is("edata.dat", ios::binary);       // create input stream
    is.read( (char*)buff, MAX*sizeof(int) );   // read from it
    for(j=0; j<MAX; j++) // check data
        if( buff[j] != j )  std::cerr << "\nData is incorrect";
            else std::cout << "\nData is correct";
 }
```

## Writing an Object to Disk

When writing an object, you generally want to use binary mode. This writes the same bit configuration to disk that was stored in memory and ensures that numerical data contained in objects is handled properly.

```
#include <fstream.h>            // for file streams
class person {                  // class of persons
  protected:
    char name[40];          // person's name
    int age;                // person's age
  public:
    void getData(void) {            // get person's data
        std::cout << "Enter name: "; cin >> name;
        std::cout << "Enter age: "; cin >> age;
     }
};
```

```
int main() {
  person pers;                          // create a person
  pers.getData();                       // get data for person
  ofstream outfile("PERSON.DAT", ios::binary);
  outfile.write( (char*)&pers, sizeof(pers) );  // write to it
 }
```

## Reading an Object from Disk

```
#include <fstream.h>                // for file streams
class person {                      // class of persons
  protected:
    char name[40];                  // person's name
    int age;                        // person's age
  public:
    void showData(void) {           // display person's data
      std::cout << "\n   Name: " << name;
      std::cout << "\n   Age: " << age;
    }
  };
```

```
 int main() {
   person pers;                       // create person variable
   ifstream infile("PERSON.DAT", ios::binary); // create stream
   infile.read( (char*)&pers, sizeof(pers) );  // read stream
   pers.showData();                                // display person
 }
```

To work correctly, programs that read and write objects to files, must be working on the same class of objects. Objects of class person in these programs are exactly 42 bytes long, with the first 40 occupied by a string representing the person’s name and the last 2 containing an int representing the person’s age.

Notice, however, that although the person classes in both programs have the same data, they may have different member functions. The first includes the single function getData(), whereas the second has only showData(). It doesn’t matter what member functions you use, because members functions are not written to disk along with the object’s data. The data must have the same format, but inconsistencies in the member functions have no effect. This is true only in simple classes that don’t use virtual functions.

## I/O with Multiple Objects

```
#include <fstream.h>                  // for file streams
class person {                        // class of persons
   protected:
      char name[40];                  // person's name
      int age;                        // person's age
   public:
      void getData() {             // get person's data
         cout << "\n   Enter name: "; cin >> name;
         cout << "   Enter age: "; cin >> age;
      }
      void showData() {             // display person's data
         cout << "\n   Name: " << name;
         cout << "\n   Age: " << age;
      }
};
```

objfile.cpp

```
int main(){
   char ch;
   person pers;                          // create person object
   fstream file;                         // create input/output file
   file.open("PERSON.DAT", ios::out | ios::binary ); // open for append
   do{                                   // data from user to file
      cout << "\nEnter person's data:";
      pers.getData();                    // get one person's data
      file.write( (char*)&pers, sizeof(pers) ); // write to file
      cout << "Enter another person (y/n)? ";
      cin >> ch;
   } while(ch=='y');                     // quit on 'n'
   file.close();                         // reset to start of file
   file.open("PERSON.DAT", ios::in | ios::binary );
    file.read( (char*)&pers, sizeof(pers) ); // read first person
   while( !file.eof() )                  // quit on EOF
    {
      cout << "\nPerson:";               // display person
      pers.showData();
      file.read( (char*)&pers, sizeof(pers) );  // read another
    }                                    // person
}
```

## Reacting to Errors

The next program shows how errors are most conveniently handled. All disk operations are checked after they are performed. If an error has occurred, a message is printed and the program terminates. We will use the technique, discussed earlier, of checking the return value from the object itself to determine its error status. The program opens an output stream object, writes an entire array of integers to it with a single call to write(), and closes the object. Then it opens an input stream object and reads the array of integers with a call to read().

```
#include <fstream> // for file streams
#include <process> // for exit()
const int MAX = 1000;
int buff[MAX];
int main(){
  for(int j=0; j<MAX; j++) buff[ j ] = j; // fill buffer with data
  ofstream os; // create output stream
  os.open("edata.dat", ios::trunc | ios::binary); // open it
  if(!os) { cerr << "\nCould not open output file"; exit(1); }
  std::cout << "\nWriting..."; // write buffer to it
  os.write( (char*)buff, MAX*sizeof(int) );
  if(!os) { cerr << "\nCould not write to file"; exit(1); }
  os.close(); // must close it
}
```

```
  for(j=0; j<MAX; j++) buff[ j ] = 0; // clear buffer
  ifstream is; // create input stream
  is.open("edata.dat", ios::binary);
  if(!is) { std::cerr << "\nCould not open input file"; exit(1); }
  std::cout << "\nReading...";
  is.read( (char*)buff, MAX*sizeof(int) ); // read file
  if(!is) { std::cerr << "\nCould not read from file"; exit(1); }
  for(j=0; j<MAX; j++) // check data
    if( buff[j] != j ) { std::cerr << "\nData is incorrect"; exit(1); }
  std::cout << "\nData is correct";
}
```

## Analyzing Errors

In the previous example, we determined whether an error occurred in an I/O operation by examining the return value of the entire stream object.

```
if(!is)
   // error occurred
```

However, it’s also possible, using the ios error-status bits, to find out more specific information about a file I/O error.

```
#include <fstream.h>            // for file functions
int main(){
   ifstream file;
   file.open("GROUP.DAT", ios::nocreate);
   if( !file )
      cout << endl <<"Can't open GROUP.DAT";
   else
      cout << endl << "File opened successfully.";
   cout << endl << "file = " << file;
   cout << endl << "Error state = " << file.rdstate();
   cout << endl << "good() = " << file.good();
   cout << endl << "eof() = " << file.eof();
   cout << endl << "fail() = " << file.fail();
   cout << endl << "bad() = " << file.bad();
   file.close();
}
```

This program first checks the value of the object file. If its value is zero, the file probably could not be opened because it didn't exist. Here's the output of the program when that's the case:

```
Can't open GROUP.DAT
file = 0x1c730000
Error state = 4
good() = 0
eof() = 0
fail() = 4
bad() = 4
```

The error state returned by rdstate() is 4. This is the bit that indicates the file doesn't exist; it's set to 1. The other bits are all set to 0. The good() function returns 1 (true) only when no bits are set, so it returns 0 (false). I'm not at EOF, so eof() returns 0. The fail() and bad() functions return nonzero because an error occurred.

In a serious program, some or all of these functions should be used after every I/O operation to ensure that things have gone as expected.

## File Pointers

Each file object has associated with it two integer values called the *get pointer* and the *put pointer*. These are also called the *current get position* and the *current put position*, or—if it's clear which one is meant—simply the *current position*. These values specify the byte number in the file where writing or reading will take place

There are times when you must take control of the file pointers yourself so that you can read from or write to an arbitrary location in the file. The seekg() and tellg() functions allow you to set and examine the get pointer, and the seekp() and tellp() functions perform the same actions on the put pointer.

```
// seeks particular person in file
#include <fstream.h> // for file streams
class person { // class of persons
  protected:
    char name[40]; // person's name
    int age; // person's age
  public:
    void showData() { // display person's data
      cout << "\n Name: " << name; cout << "\n Age: " << age;
    }
};
```

```
int main(){
    person pers; // create person object
    ifstream infile; // create input file
    infile.open("PERSON.DAT", ios::binary); // open file
    infile.seekg(0, ios::end); // go to 0 bytes from end
    int endposition = infile.tellg(); // find where we are
    int n = endposition / sizeof(person); // number of persons
    cout << endl << "There are " << n << " persons in file";
    cout << endl << "Enter person number: "; cin >> n;
    int position = (n-1) * sizeof(person); // number times size
    infile.seekg(position); // bytes from begin
    infile.read( (char*)&pers, sizeof(pers) ); // read one person
    pers.showData(); // display the person
}
```

Here's the output from the program, assuming that the PERSON.DAT file contains 3 persons:

```
There are 3 persons in file
Enter person number: 2
   Name: Rainier
   Age: 21
```

## File I/O Using Member Functions

So far, we've let the main() function handle the details of file I/O. This is nice for demonstrations, but in real object-oriented programs, it's natural to include file I/O operations as member functions of the class.

In the next example, we will add member functions, diskOut() and diskIn() to the person class. These functions allow a person object to write itself to disk and read itself back in.

Simplifying assumptions: First, all objects of the class will be stored in the same file, called PERSON.DAT. Second, new objects are always appended to the end of the file. An argument to the diskIn() function allows me to read the data for any person in the file. To prevent attempts to read data beyond the end of the file, I include a static member function, diskCount(), that returns the number of persons stored in the file.

```
#include <fstream.h> // for file streams
class person {// class of persons
  protected:
    char name[40]; // person's name
    int age; // person's age
  public:
    void getData(){ // get person's data
    cout << "\n Enter name: "; cin >> name; cout << " Enter age: "; cin >> age;}
  void showData(){ // display person's data
          cout << "\n Name: " << name; cout << "\n Age: " << age; }
  void diskIn(int ); // read from file
  void diskOut(); // write to file
  static int diskCount(); // return number of  persons in file
};

void person::diskIn(int pn){ // read person number pn  from file
  ifstream infile; // make stream
  infile.open("PERSON.DAT", ios::binary); // open it
  infile.seekg( pn*sizeof(person) ); // move file ptr
  infile.read( (char*)this, sizeof(*this) ); // read one person
}
```

```
void person::diskOut()                  // write person to end of file
{
  ofstream outfile;                     // make stream
  outfile.open("PERSON.DAT", ios::app | ios::binary);  // open it
  outfile.write( (char*)this, sizeof(*this) ); // write to it
}

int person::diskCount()                 // return number of persons  in file
{
  ifstream infile;
  infile.open("PERSON.DAT", ios::binary);
  infile.seekg(0, ios::end);            // go to 0 bytes from end
  return infile.tellg() / sizeof(person); // calculate number of persons
}
```

```
int main(void){
   person p;                              // make an empty person
   char ch;
   do{                                   // save persons to disk
      cout << "\nEnter data for person:";
      p.getData();                        // get data
      p.diskOut();                        // write to disk
      cout << "Do another (y/n)? ";
      cin >> ch;
   }while(ch=='y');                         // until user enters 'n'
   int n = person::diskCount();         // how many persons in file?
   cout << "\nThere are " << n << " persons in file";
   for(int j=0; j<n; j++) {  // for each one,
      cout << "\nPerson #" << (j+1);
      p.diskIn(j);                        // read person from disk
      p.showData();                          // display person
   }
 }
```

## Overloading the « and » Operators

In this session I'll show how to overload the extraction and insertion operators. This is a powerful feature of C++. It lets you treat I/O for user-defined data types in the same way as for basic types such as int and double. For example, if you have an object of class TComplex called c1, you can display it with the statement

cout << c1; just as if it were a basic data type.

You can overload the extraction and insertion operators so they work with the display and keyboard (cout and cin). With a little more care, you can also overload them so they work with disk files as well.

```
#include<iostream>
class TComplex {
    float   real,img;
    friend std::istream& operator >>(std::istream&, TComplex&);
    friend std::ostream& operator <<(std::ostream&, const TComplex&);
  public:
    TComplex(float rl=0,float ig=0){real=rl;img=ig;}
    TComplex operator+(const TComplex&);
};
```

```
istream& operator >>(istream& stream, TComplex& z){  // Overloading >>
   cout << "Enter real part:";
   stream >> z.real;
   cout << "Enter imaginer part:";
   stream >> z.img;
   return stream;
}
ostream& operator <<(ostream& stream, const TComplex & z){
   stream << "( " << z.real << " , " << z.img << " ) \n";
   return stream;
}
TComplex TComplex::operator+(const TComplex & z){    // Operator +
  return TComplex (real+z.real , img+z.img);
}
int main(){
  TComplex z1,z2,z3;
  std::cin >> z1;
  std::cin >> z2;
  z3=z1+z2;
  std::cout << " Result=" << z3;
}
```

inout.cpp

## Overloading for Files

The next example shows how the << and >> operators can be overloaded so they work with both file I/O and cout and cin.

```
#include<fstream>
class TComplex {
    float real,img;
    friend istream& operator >>(istream&, TComplex&);
    friend ostream& operator <<(ostream&, const TComplex&);
 public:
    TComplex(float rl=0,float ig=0){real=rl;img=ig;}
};
istream& operator >>(istream& stream, TComplex &z){
   char dummy;
   stream >> dummy >> z.real;
   stream >> dummy >> z.img >> dummy;
   return stream;
}
ostream& operator <<(ostream& stream, const TComplex & z){
                  stream << "(" << z.real << " , " << z.img << ") \n";
                  return stream;
};
```

```
int main(){
  char ch;
  TComplex z1;
  ofstream ofile;                    // create and open
  ofile.open("complex.dat");  // output stream
  do {  std::cout << "\nEnter Complex Number:(real,img)";
          cin >> z1;                   // get complex number from user
          ofile << z1;                 // write it to output str
          std::cout << "Do another (y/n)? ";   std::cin >> ch;
   } while(ch != 'n');
   ofile.close();                    // close output stream
   std::ifstream ifile;              // create and open
   ifile.open("complex.dat");  // input stream
   std:.cout << "\nContents of disk file is:";
   while(!ifile.eof()){
      ifile >> z1;  // read complex number from stream
      if (ifile)
         std::cout << "\nComplex Number = " << z1;   // display complex number
   }
}
```

fileio.cpp

## Overloading for Binary I/O

So far, you’ve seen examples of overloading operator<<() and operator>>() for formatted I/O. They also can be overloaded to perform binary I/O. This may be a more efficient way to store information, especially if your object contains much numerical data.

```
#include <fstream.h> // for file streams
class person {// class of persons
  protected:
       char name[40]; // person's name
       int age; // person's age
  public:
   void getData(){ // get data from keyboard
          cout << "\n Enter name: "; cin.getline(name, 40);
          cout << " Enter age: "; cin >> age;
   }
   void putData(){ // display data on screen
          cout << "\n Name = " << name; cout << "\n Age = " << age;
   }
   friend istream& operator >> (istream& s, person& d);
   friend ostream& operator << (ostream& s, person& d);
```

```
        void persin(istream& s){
          s.read( (char*)this, sizeof(*this) );
        }
        void persout(ostream& s) // write our data to file
        {
          s.write( (char*)this, sizeof(*this) );
        }
};   // end of class definiton
istream& operator >> (istream& s, person& d) {
        d.persin(s);
        return s;
}
ostream& operator << (ostream& s, person& d){
        d.persout(s);
        return s;
}
```

```
int main(){
        person pers1, pers2, pers3, pers4;
        cout << "\nPerson 1";
        pers1.getData();  // get data for pers1
        cout << "\nPerson 2";
        pers2.getData(); // get data for pers2
        outfile("PERSON.DAT", ios::binary);
        outfile << pers1 << pers2; // write to file
        outfile.close();
        ifstream infile("PERSON.DAT", ios::binary);
        infile >> pers3 >> pers4; // read from file into
        cout << "\nPerson 3";     // pers3 and pers4
        pers3.putData();  // display new objects
        cout << "\nPerson 4";
        pers4.putData();
}
```