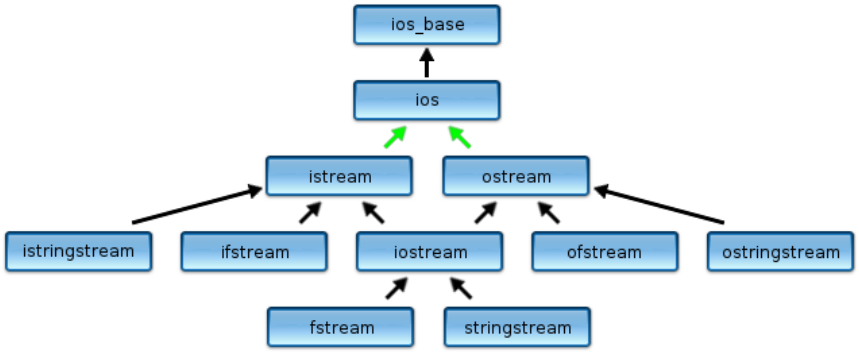
Chapter

12

Processing Files

The C++ standard library stream headers contain several different classes that form a class hierarchy, designed using the object-oriented facility known as **inheritance**.



Note **headers**, not header. Until now, have one stream header: `<iostream>`. To read and write to files (instead of the standard streams, we'll use the **file stream** classes—*`ifstream`* and *`ofstream`*—found in the `<fstream>` header. The name *`ifstream`* stands for `input-file-stream`, while the name *`ofstream`* stands for `output-file-stream`.

In the diagram above, each class is a **derived class** (or **subclass**), of the class above it. Thus, *`istream`* and *`ostream`* are both **derived from** *`ios`*, and are **specialized** kinds of *`ios`* objects. In the opposite direction, *`ios`* is a **base class** (or **superclass**) of both *`istream`* and *`ostream`*. Similarly, *`ifstream`* is derived from *`istream`* and *`ostream`* is the base class of *`ofstream`*.

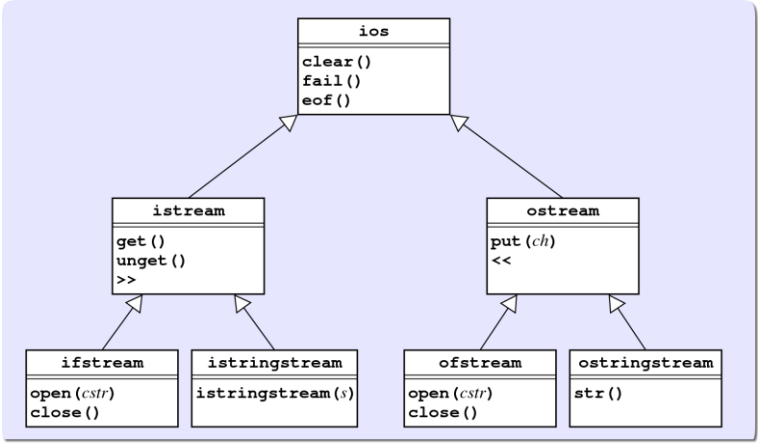
This relationship—between base and derived classes—is conveyed by the words **is a**. Every *`ifstream`* object **is an** `istream` and, by continuing up the hierarchy, an `ios`. This means that characteristics of any class are **inherited** by its derived classes.

UML Diagrams

Simple diagrams that show the relationships among classes are useful, but often we want to expand them to **include the methods** exposed at each level. This diagram is a standard way of displaying a class hierarchy called the **Unified Modeling Language**, or **UML**. In UML, each class appears as a rectangular box whose upper portion contains the name of the class.

Inheritance Arrows

The **public member functions** of the class appear in the lower portion. In UML, derived classes use open arrowheads to **point to** their base classes.



Inherited Member Functions

UML diagrams make it easy to determine which **inherited member functions** are available to each of the classes in the diagram. Because each class inherits all of the methods of **every class** in its ancestor chain, an object of a particular class can call **any member function** defined in any of those classes.

For example, the diagram shows that **any** ***ifstream*** object can call these functions:

- **`open()`** and **`close()`** from the ***`ifstream`*** class itself
- **`get()`** and **`unget()`** and the **`>>`** operator from ***`istream`***
- **`clear()`**, **`fail()`**, and **`eof()`** from the ***`ios`*** class

Easy File I/O

File processing in C++ is fairly straightforward:

1. **Declare a stream variable to refer to the file**. Here's an example with both an input file stream and an output file stream.

```
ifstream infile;
ofstream outfile;
```

2. **Open the file.** To **establish an association** between that variable and an actual physical file on disk you need to **open the file** calling **`open()`**.

```
infile.open("myfile.txt");
```

Alternatively, you can use perform **both steps at once** using the **stream constructors**. Here's an example:

```
ifstream infile{"myfile.txt"};
```

If the **file is missing** the stream will **fail to open**; you can check for that by calling the member function `fail()`. There will be **no other error messages**:

```
ifstream infile{"myfile.txt"};
if (infile.fail()) { /* handle error */ }
```

3. Transfer the data. Read and write data using these techniques:
   - Read or write character by character using **unformatted I/O**.
   - Process the file **line by line**, using **line-oriented I/O**.
   - Read and write **formatted data**, mixing numeric data with strings and other data types. This is known as **token-based file I/O**.

Processing Lines

Since text files are usually arranged by lines, it is often useful to **read an entire line of data at once**. The easiest way to do that is to use the function named `getline()` in the `<string>` library. `getline()` is not a member function, and it takes two arguments:

- the **input stream from which** the line is read (open this as shown above)
- a `string` variable **into which** the result is written

By default, `getline()` stops when it encounters a newline, which is **removed** from the stream and **discarded**. It **is not** stored as part of the `string`. Like `get()`, the `getline()` member function **returns** the input stream, which allows you to test for end-of-file.

```
string line;
while (getline(in, line))
    cout << line << endl;
```

This `while` loop reads each line of data from the stream into the string variable `line` until the stream reaches the end of the file. For each line, the body of the loop uses `<<` to send the line to `cout`, followed by a newline character to replace the one which was discarded by `getline()`.

Try it Yourself

Click on the "Running Man" to start with a function named `searchFile()` that takes two `string` arguments. The first is the name of the file to open, and the second is the word or phrase to search for. Neither `string` may be modified; there is no return value.

- Open the file and **read it line by line**.
- If the phrase you are looking for is found in that line, then print the line number (in a field `5` wide), a space, a colon, another space and then the line from the file, followed by a newline.
- Assume the first line number in the file is line #1.
- Your output should be printed with `cout`.
- If the file cannot be found, then print an error message (using `cerr`, of course): "`File fname cannot be opened`".

Mechanics (First Draft)

You shouldn't have to think about this part; memorize and practice these steps until they become second nature.

1. **Open the file using the supplied filename**. If it can't be opened, then print the error message using `cerr`. To check if the file was opened, **explicitly** use the `fail()` function or, **implicitly** check using the stream variable itself. If you use the second method, don't forget the **not operator**.

   Instead of adding an `else` to the `if` statement, add a `return` statement to end the function if nothing else can be done.

2. Add the **line-oriented I/O pattern**. You will have read and printed every line, and so you'll be ready to go on to the next step, where you actually solve the problem.

Searching & Numbering

Searching is easy. Place the output **inside an *if* statement**. Use `find()` as part of your condition. If the word is found, then print the line. Numbering the line is also very easy.

- Create a **line counter** right before the loop starts.
- In the loop, **increment the counter** each time a line is read.
- Instead of printing the line when the phrase is found, print the line number, using **formatted output** before printing the contents.

To print the line number in a field **five character wide** you'll need to use `setw()`. Follow that with a space, a colon and another space, and finally the line itself.

Processing Tokens

We can also process input **token by token** or word by word. The word **token** means "a chunk of meaningful data". A token may be an integer, a number, a string, or a custom type, like stars or points.

As you've already seen, you read a token using the **extraction operator** `>> var`. If `var` is a `string`, this reads a single word. When `var` is an `int`, it reads an integer, and so on.

The input operation **returns the stream**, just with raw input and line-oriented input. We could process input **word-by-word**, like this:

```
string word;
while (in >> word) // process the word
```

Of course, word-by-word is not exactly correct, since a word may include punctuation, or be a number, and so on. Technically, it is **token-by-token**.

Validating Data

With raw, line-by-line or string-based token-oriented input, a data loop **only** ends when it reaches end-of file. However, consider this filter, which reads and processes integers:

```
int n;
while (cin >> n) // process n
```

This loop **fails** when `cin` cannot read an integer; it also fails when it reaches end-of-file. When this occurs, the `cin` object is placed in a failed state. **No error message is printed**; the rest of the input is simply not processed. To fix:

1. Check to make sure you haven't reached `cin.eof()`
2. **Reset** the **error state**, by calling the member function: `cin.clear()`
3. **Remove** the offending token from the stream with `cin >> bad_data` where `bad_data` is a `string`. This will remove one token and throw it away.
4. You may also want to **print an error message** to `cerr`.

```
int n, sum{0};
while (cin)
{
    if (cin >> n) { sum += n; }
    else if (! cin.eof())
    {
        cin.clear(); // Clear error
        string bad_data;
        cin >> bad_data; // Remove token
        cerr << "Invalid: " << bad_data << endl;
    }
}
cout << "sum: " << sum << endl;
```

Finish Up

- Complete the **reading exercises (REX)** for this chapter.
- Complete the homework using the **CS50 IDE**. The link is on Canvas.
  a. Make sure you submit the assignment using `make submit`.
  b. Make sure you check the CS150 Homework Console to see that your scores got reported, before the beginning of the next lecture.
- Take the **pre-class reading quiz** on Canvas. You have two attempts.

See you in class or on the Canvas discussion board.