【C++】Day27(2)

Class	C++
Date	@December 29, 2021
Material	
Series Number	
Summary	

【Ch8】The IO Library

8.1 The IO Classes

To support different kinds of IO processing ,the library defines a collection of IO types in addition to the `istream` and `ostream` types that we have already used.

These types are defined in three separate headers:

Header	Type
`iostream`	`istream`, `wistream` reads from a stream `ostream`, `wostream` writes to a stream `iostream`, `wiostream` reads and writes a stream
`fstream`	`ifstream`, `wifstream` reads from a file `ofstream`, `wofstream` writes to a file `fstream`, `wfstream` reads and writes a file

`sstream`	`istringstream`, `wistringstream` reads from a `string`
	`ostringstream`, `wostringstream` writes to a `string`
	`stringstream`, `wstringstream` reads and writes a `string`

- iostream defines the basic types used to read from and write to a stream
- fstream defines the types used to read and write named fiels
- sstream defiens the types used to read and write in-memory strings

Relationships among the IO Types

Conceptually, neither the kind of device nor the character size affects the IO operaetions we want to perform.

For example, we'd like to use `>>` to read data regardless of whether we're reading a console window, a disk file, or a string.

The library lets us ignore the differences among these different kinds of streams by using inheritance.

Briefly, inheritance lets us say that a particular class inherits from another class. Ordinarily, we can use an object of an inherited class as if it were an object of the same type as the class from which it inherits.

The types `ifstream` and `istringstream` inherit from `istream`. Thus, we can use objects of type `ifstream` or `istringstream` as if they were `istream` objects.

We can use objects of these types in the same ways as we have used cin.

For example, we can call `getline` on an `ifstream` or `istringstream` object, and we can use the `>>` to read data from an ifstream or istringstream.

Note: Everything that we cover in the remainder of this section applies equally to plain streams, file streams, and string streams and to the char or side-character stream versions.

8.1.1 No Copy or Assign for IO Objects

We cannot copy or assign objects of the IO tyeps:

```
ofstream out1, out2;
out1 = out2; //error: cannot assign stream objects
ofstream print(ofstream); //error: can't initialize the ofstream parameter
```

Because we cannot copy the IO types, we cannot have a parameter or return type that is one of the stream types. Functions that do IO typically pass and return the stream through references.

Reading or writing an IO object changes its state, so the reference must not be const.

8.1.2 Condition States

The IO classes define functions and flags that let us access and manipulate the condition state of a stream:

strm::iostate	*strm* is one of the IO types listed in Table 8.1 (p. 310). iostate is a machine-dependent integral type that represents the condition state of a stream.
strm::badbit	*strm*::iostate value used to indicate that a stream is corrupted.
strm::failbit	*strm*::iostate value used to indicate that an IO operation failed.
strm::eofbit	*strm*::iostate value used to indicate that a stream hit end-of-file.
strm::goodbit	*strm*::iostate value used to indicate that a stream is not in an error state. This value is guaranteed to be zero.
s.eof()	true if eofbit in the stream s is set.
s.fail()	true if failbit or badbit in the stream s is set.
s.bad()	true if badbit in the stream s is set.
s.good()	true if the stream s is in a valid state.
s.clear()	Reset all condition values in the stream s to valid state. Returns void.
s.clear(flags)	Reset the condition of s to flags. Type of flags is *strm*::iostate. Returns void.
s.setstate(flags)	Adds specified condition(s) to s. Type of flags is *strm*::iostate. Returns void.
s.rdstate()	Returns current condition of s as a *strm*::iostate value.

As an example of an IO error, consider the following code:

```
int val;
cin >> val;
```

If we enter `Boo` on the standard input, the read will fail. The input operator expected to read an int but got the character `B` instead. As a result, cin will be put in an error state.

Similarly, cin will be in an error state if we enter an end-of-file.

Once an error has occurred, subsequent IO operations on that stream will fail. We can read from or write to a stream only when it is in a non-error state.

Because a stream might be in an error state, code ordinarily should check whether a stream is okay before attempting to use it. The easiest way to determine the state of a stream object is to use that object as a condition:

```
while(cin >> word)
  //ok: read operation successful
```

The while condition checks the state of the stream returned from the `>>` expression. If that input operation succeeds, the state remains valid and teh condition will succeed.

Interrogating the State of a Stream

Using a stream as a condition tells us only whether the stream is valid. It does not tell us what happened. Sometimes we also need to know why the stream is invalid.

The IO library defiens a machine-dependent integral type named `iostate` that it uses to convey information about the state of a stream.

The IO classes define four `constexpr` values of type `iostate` that represent particular bit patterns. These values are used to indicate particular kinds of IO conditions. They can be used with the bitwise operators to test or set multiple flags in one operation.

- The `badbit` indicates a system-level failure, such as an unrecoverable read or write error. It is usually not possible to use a stream once badbit has been set.
- The `failbit` is set after a recoverable error, such as reading a character when numeric data was exepected. It is often possible to correct such problems and continue using the stream
- Reaching end-of-file sets both `eofbit` and `failbit`.
- The `goodbit`, which is guaranteed to have the value 0, indicates no failures on the stream.

If any of `badbit`, `failbit`, or `eofbit` are set, then a condition that evaluates that stream will fail.

The library also defines a set of functions to interrogate the state of these flags.

- The `good` operation returns true of none of the error bits is set.
- The `bad`, `fail`, and `eof` operations return true when the corresponding bit is on.
- In addition, `fail` returns true if `bad` is set.

By implication, the right way to determine the overall state of a stream is to use either `good` or `fail`. Indeed, the code that is executed when we sue stream as a condition is equivalent to calling `!fail()`

Managing the Condition State

- The `rdstate()` member returns an iostate value that corresponds to the current state of the stream.
- The `setstate` operation turns on the given condition bits to indicate that problem occurered.
- The `clear` member is overloaded. One version takes no arguments and a second version takes a single argument of type iostate.

  The version of `clear` that takes no arguments turns off all the failure bits. After `clear()`, a call to good returns true.

We might use these members as follows:

```
//remember the current state of cin
auto old_state = cin.rdstate(); //remember the current state of cin
cin.clear(); //make cin valid
process_input(cin); //use cin
cin.setstate(old_state); //now reset cin to its old state
```

The version of clear that takes an argument expects an iostate value that represents the new state of the stream. To turn off a single condition, we use the rdstate member and the bitwise operators to produce the desired new state.

For example, the following turns off fallbit and bad bit but leaves eofbit untouched:

```
//turns off failbit and badbit but all other bits unchanged
cin.clear(cin.rdstate() & ~cin.failbit & ~cin.badbit);
```

Exercise

Exercise 8.1: Write a function that takes and returns an `istream&`. The function should read the stream until it hits end-of-file. The function should print what it reads to the standard output. Reset the stream so that it is valid before returning the stream.

Exercise 8.2: Test your function by calling it, passing `cin` as an argument.

```
std::istream& func(std::istream &is) {
  //auto old_state = is.rdstate();
  is.clear();
  string str;
  while(is >> str)
    cout << str;
  is.clear();
  //is.setstate(old_state);
  return is;
}
```