Object – Oriented Programming

Week 14

Streams

Weng Kai

Why streams?

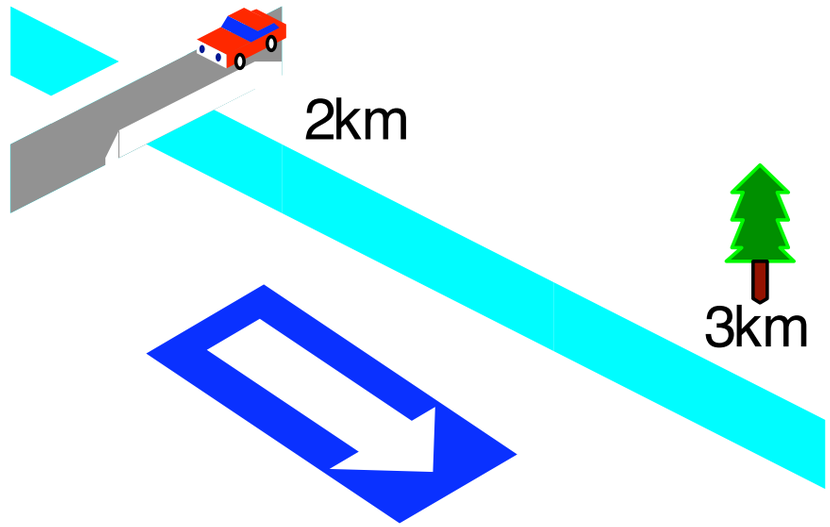
- Original C I/O used printf, scanf
- Streams invented for C++
  - C I/O libraries still work
- Advantages of streams
  - Better type safety
  - Extensible
  - More object-oriented
- Disadvantages
  - More verbose 冗长
  - Often slower

C vs. C++

- C stdio operations work
  - –Don't provide “object-oriented” features
  - –No overloadable operators
- C++
  - –Can overload inserters and extractors
- Moral
  - –When converting C to C++, leave the I/O intact

What is a stream?

- Common logical interface to a device
- Sequential
  - There is a "position" associated with each stream
- Can
  - Produce values
  - Consume values
  - Both



Stream naming conventions

约定

	Input	**Output**	**Header**
Generic	istream	ostream	<iostream>
File	ifstream	ofstream	<fstream>
C string (legacy)	istrstream	ostrstream	<strstream>
C string	istringstream	ostringstream	<sstream>

Stream operations

- Extractors 流提取运算符
  - –Read a value from the stream
  - –Overload the >> operator
- Inserters
  - –Insert a value into a stream
  - –Overload the << operator
- Manipulators
  - –Change the stream state
- Others

Kinds of streams

- Text streams
  - Deal in ASCII text
  - Perform some character translation
    - e.g.: newline -> actual OS file representation
  - Include
    - Files
    - Character buffers
- Binary streams
  - Binary data
  - No translations

Predefined streams

- cin
  - –standard input
- cout
  - –standard output
- cerr
  - –unbuffered error (debugging) output
- clog
  - –buffered error (debugging) output

Example

```
#include <iostream>
int i; float f; char c;
char buffer[80];
```

- Read the next character

```
cin >> c;
```

- Read an integer

```
cin >> i; // skips whitespace
```

- Read a float and a string separated by whitespace

```
cin >> f >> buffer;
```

Predefined extractors

- *istream >> lvalue*

expression type	output format	C I/O
char	character	%c
short, int	integer	%d
long	long decimal integer	%ld
float	floating point	%g
double	double precision floating pt.	%lg
long double	long double	%Lg
char *	string	%s
void *	pointer	%p

- Extractors skip leading whitespace, in general

Defining a stream extractor

- Has to be a 2-argument free function
  - First argument is an `istream&`
  - Second argument is a *reference* to a value

```
istream&
operator>>(istream& is, T& obj) {
    // specific code to read obj
   return is;
}
```

- Return an `istream&` for chaining

```
cin >> a >> b >> c;
((cin >> a) >> b) >> c;
```

Other input operators

- `int get()`
  - Returns the next character in the stream
  - Returns EOF if no characters left
  - Example: copy input to output

```
int ch;
while ((ch = cin.get()) != EOF)
  cout.put(ch);
```

- `istream& get(char& ch)`
  - Puts the next character into argument
  - Similar to int get();

More input operators

- `get(char *buf, int limit, char delim = '\n')`
  - read up to `limit` characters, or to `delim`
  - Appends a null character to `buf`
  - Does not consume the delimiter 分隔符
- `getline(char *buf, int limit,char delim = '\n')`
  - read up to `limit` characters, or to `delim`
  - Appends a null character to `buf`
  - *Does* consume the delimiter
- `ignore(int limit = 1, int delim = EOF)`
  - Skip over limit characters or to delimiter
  - Skip over delimiter if found

More input operators

- `int gcount()`
  - returns number of characters just read

```
char buffer[100];
cin.getline(buffer, sizeof(buffer));
cout << "read " << cin.gcount()
     << " characters"
```

- `void putback(char)`
  - pushes a single character back into the stream
- `char peek()`
  - examines next character without reading it

```
switch (cin.peek()) ...
```

Predefined inserters

- Usage
  - *ostream* << *expression*

expression type	output format	C I/O
char	character	%c
short, int	integer	%d
long	long decimal integer	%ld
float, double	double precision floating pt.	%g
long double	long double	%lg
char *	string	%s
void *	pointer	%p

Creating a stream inserter

- Has to be a 2-argument free function
  - First argument is an ostream&
  - Second argument is any value

```
ostream&
operator<<(ostream& os, const T& obj) {
    // specific code to write obj
   return os;
}
```

- Return an `ostream&` for chaining

```
cout << a << b << c;
((cout << a) << b) << c;
```

Other output operators

- `put(char)`
  - prints a single character
  - Examples

```
cout.put('a');
cerr.put('!');
```

- `flush()`
  - Force output of stream contents
  - Example

```
cout << "Enter a number";
cout.flush();
```

Formatting using manipulators

- Manipulators modify the state of the stream
  - #include <iomanip>
  - Effects hold (usually)
- Example

```
int n;
cout << "enter number in hexadecimal"
     << flush;
cin >> hex >> n;
```

Example

- A simple program

```
#include <iostream>
#include <iomanip>
main() {
    cout << setprecision(2) << 1000.243 <<endl;
    cout << setw(20) << "OK!";
    return 0;
}
```

- Prints

```
1e03
                    OK!
```

Manipulators

manipulator	effect	type
dec, hex, oct	set numeric conversion	I, O
endl	insert newline and flush	O
flush	flush stream	O
setw(int)	set field width	I, O
setfill(ch)	change fill character	I, O
setbase(int)	set number base	O
ws	skip whitespace	I
setprecision(int)	set floating point precision	O
setiosflags(long)	turn on specified flags	I, O
resetiosflags(long)	turn off specified flags	I, O

Creating manipulators

- You can define your own manipulators!

```
// skeleton for an output stream manipulator
ostream& manip(ostream& out) {
    ...
    return out;
}
ostream& tab ( ostream& out ) {
    return out << '\t';
}
cout << "Hello" << tab << "World!" << endl;
```

Stream flags control formatting

flag	purpose (when set)
ios::skipws	skip leading white space
ios::left, ios::right	justification
ios::internal	pad between sign and value
ios::dec, ios::oct, ios::hex	format for numbers
ios::showbase	show base of number
ios::showpoint	always show decimal point
ios::uppercase	put base in uppercase
ios::showpos	display + on positive numbers
ios::scientific, ios::fixed	floating point format
ios::unitbuf	flush on every write

Setting flags

- Using manipulators
  - –setiosflags(flags);
  - –resetiosflags(flags);
- Using stream member functions
  - –setf(flags)
  - –unsetf(flags)

Working with flags

- Code

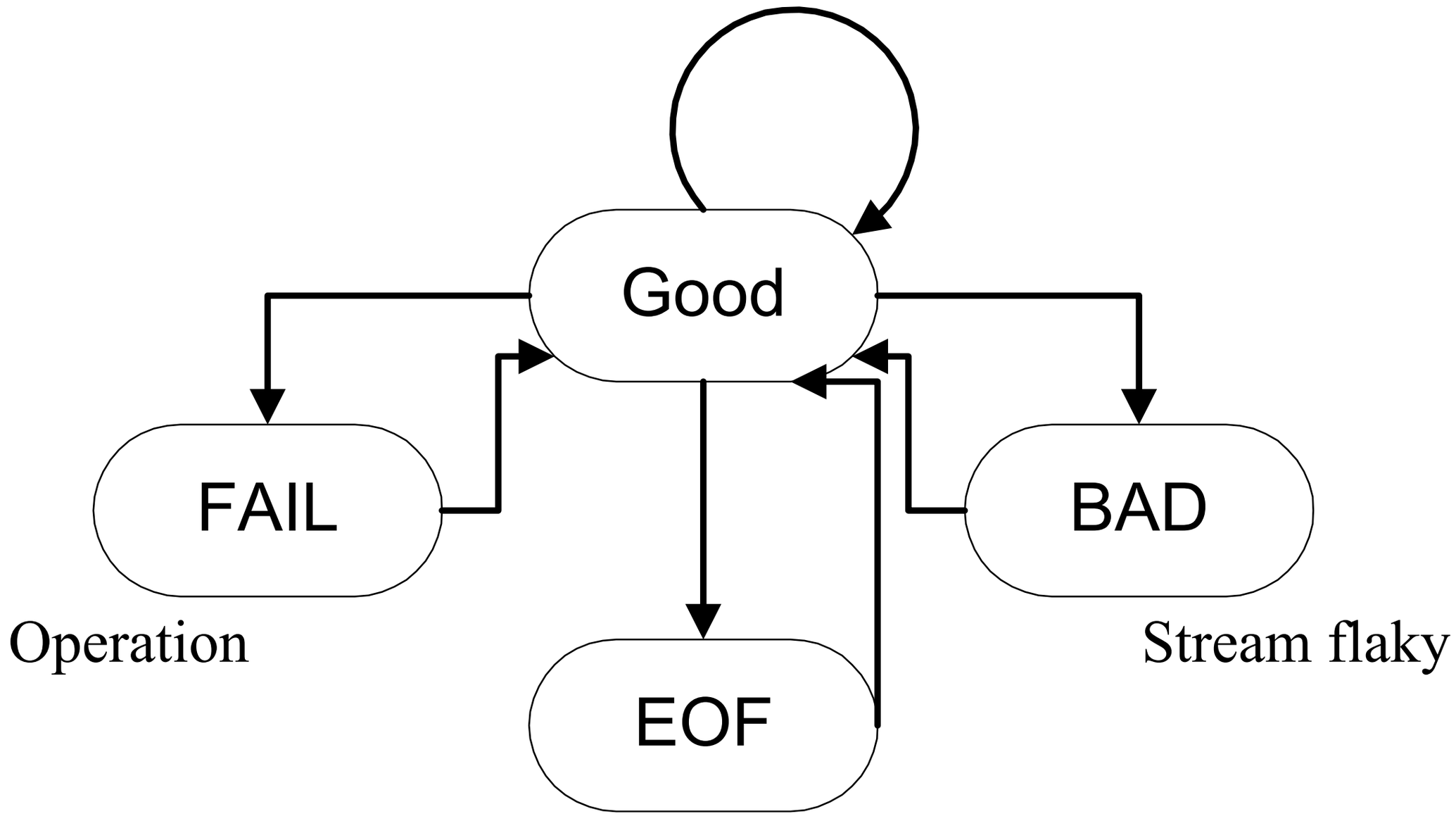
```
#include <iostream>
#include <iomanip>
main() {
    cout. setf(ios::showpos | ios::scientific);
    cout << 123 << " " << 456.78 << endl;
    cout << resetiosflags(ios::showpos) << 123;
    return 0;
}
```

- Prints

```
+123 +4.567800e+02
123
```

Stream error states

clear() returns stream to GOOD



Working with streams

- Error state is set after each operation
- Conversion to void* returns 0 if problem
- Can clear an error state using
  - `clear()` // Resets error state to good()
- Checking status
  - `good()` // Returns true if in valid state
  - `eof()` // Returns true if at EOF
  - `fail()` // Returns true if minor failure or bad
  - `bad()` // Returns true if in bad state

Example

```
int n;
cout << "Enter a value for n, then [Enter]" << flush;
while (cin.good()) {
   cin >> n;
   if (cin) { // input was ok
       cin.ignore(INT_MAX, '\n'); // flush newline
       break;
    }
    if (cin.fail()) {
       cin.clear();  // clear the error state
       cin.ignore(INT_MAX, '\n'); // skip garbage
       cout << "No good, try again!" << flush;
    }
}
```

File streams

- ifstream, ofstream connect files to streams
  - –#include <fstream>
  - –Open modes specify how to create files

mode	purpose
ios::app	append
ios::ate	position at end of file
ios::binary	do binary I/O
ios::in	open for input
ios::out	open for output
ios::nocreate	don’t create file if not there
ios::noreplace	don’t replace file if present
ios::trunc	truncate file if present

File streams

```
#include <iostream>
#include <fstream>
int main(int argc, char *argv[]) {
   if (argc != 3) {
     cerr << "Usage: copy file1 file2" << endl;
     exit(1);
}
ifstream in(argv[1]);

if (!in) {
   cerr << "Unable to open file " << argv[1];
   exit(2);
}
```

File streams

```
ofstream  out(argv[2]);
if (!out) {
   cerr << "Unable to open file " << argv[2];
   exit(2);
}
   char c;
   while (in >> c) {
      out << c;
   }
}
```

More stream operations

- `open(const char *, int flags, int)`
  - Open a specified file

```
ifstream inputS;
inputS.open("somefile", ios::in);
if (!inputS) {
    cerr << "Unable to open somefile";
...
```

- `close()`
  - Closes stream

IO stream buffers

- Every IO stream has a stream buffer
- Class `streambuf` defines the buffer abstraction
- The member function `rdbuf()` returns a pointer to the stream buffer
- The << operation is overloaded for `streambufs`
  - It connects buffers directly!

Copy a file to standard out

```
#include <fstream>
#include <assert>

main(int argc, char *argv[]) {
  assert(argc == 2);
  ifstream in(argv[1]);
  assert(in);  // check that stream opened
  cout << in.rdbuf(); // Drain file!
}
```

String streams (legacy)

- I/O to character buffers is modeled using streams
  - `#include <strstream.h>`
  - Input: `istrstream` class
  - Output: `ostrstream` class

```
istrstream in("2.3 47 This is a stream");
int i; float f; char buf[123];
in >> f >> i >> buf;
cout << " i = " << i;
cout << " f = " << f;
cout << " buf = " << buf << endl;
cout << in.rdbuf(); // print remainder!
```

ostrstreams and storage allocation

- Input streams are initialized with a buffer

```
istrstream mystr("hi bob");
```

- Output streams have two allocation methods
  - User allocates storage

```
char buffer[SIZE];
ostrstream(buffer, SIZE , ios::out);
```

  - Stream handles storage

```
ostrstream A;
A << cin.rdbuf(); // read file into string!
```

    - You can get the buffer, but programming gets messy
      - `char *str()` returns the buffer...

Notes

- use string and stringstream (not strstream)
  –example
- You can create your own manipulators

```
// newline without a flush
ostream & nl ( ostream& os ) {
   return os << '\n';
}

cout << "newline" << nl;
```

C vs. C++

- C stdio operations work
  - –Don't provide “object-oriented” features
  - –No overloadable operators
- C++
  - –Can overload inserters and extractors
- Moral
  - –When converting C to C++, leave the I/O intact