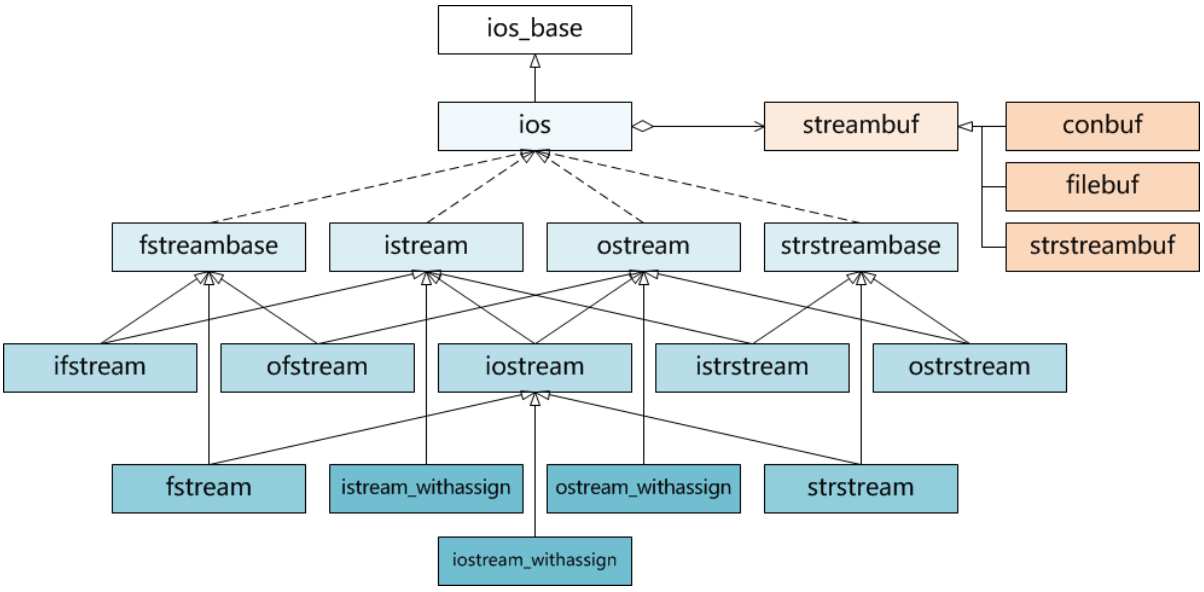
7 I/O流

7.2 I/O流的基本概念

- 在C++中，数据I/O通过流来实现
  - 输出流：针对数据写入目标的流
  - 输入流：针对数据读取来源的流
  - C++的流（stream）源自ANSI C中的标准I/O流，它与UNIX系统V的STREAMS I/O系统是两个完全不同的概念

7.2 C++的I/O流体系



- istream及其子类支持输入操作
- ostream及其子类支持输出操作
- 由istream和ostream类共同派生的子类iostream既支持输入操作，也支持输出操作

7.3 文本I/0

7.3.1 文本文件的格式化I/O

```cpp
// text.cpp

// 文本文件的格式化I/O

#include <iostream>
#include <fstream>

using namespace std;

int main(void) {
    // 如果参数文件名所代表的文件并不存在，那么程序将按照缺省行为创建这
    // 个文件。如果这个文件原先已经存在，它的内容将从开始位置全部被覆盖
    ofstream ofs("text.txt");
    if (!ofs) {
        cout << "Unable to open file for writing" << endl;
```

```
        return -1;
    }

    ofs << 1234 << ' ' << 56.78 << ' ' << "apples" << '\n';

    ofs.close();

    ofs.open("text.txt", ios::app); // 把数据追加到原有内容的后面
    if (!ofs) {
        cout << "Unable to open file for appending" << endl;
        return -1;
    }

    ofs << "append_a_line\n";

    ofs.close();

    ifstream ifs("text.txt"); // 要求文件必须存在
    if (!ifs) {
        cout << "Unable to open file for reading" << endl;
        return -1;
    }

    int n;
    double d;
    string str1, str2;

    // 经重载定义的提取操作符“>>”能够按照预期的数据类型进行
    // 格式化提取操作。缺省情况下，提取操作符会自动跳过输入
    // 文件中各个数据项间的空白字符（空格、制表符、换行符等）

    ifs >> n >> d >> str1 >> str2;

    cout << n << ' ' << d << ' ' << str1 << ' ' << str2 << endl;

    ifs.close();

    return 0;
}
```

7.3.1.1 I/O流的打开模式

```
ifstream(const char* filename, openmode mode);
ofstream(const char* filename, openmode mode);
fstream(const char* filename, openmode mode);
```

```
void open(const char* filename, openmode mode = default_mode);
void close(void);
```

- ios::in
  - 打开文件用于读取，不存在则失败，存在不清空
  - 适用于ifstream（缺省）和fstream
- ios::out

    - 打开文件用于写入，不存在则创建，存在则清空
    - 适用于ofstream（缺省）和fstream
- ios::app
    - 打开文件用于追加，不存在则创建，存在不清空
    - 适用于ofstream和fstream
- ios::trunc
    - 打开时清空原内容
    - 适用于ofstream和fstream
- ios::ate
    - 打开时定位文件尾
    - 适用于ifstream、ofstream和fstream
- ios::binary
    - 以二进制模式读写
    - 适用于ifstream、ofstream和fstream
- 打开模式可以组合使用，如：
    - ios::in|ios::out表示既读取又写入
- 打开模式不能随意组合，如：
    - ios::in|ios::trunc不合理，清空同时读取没有意义
    - ios::in|ios::out|ios::trunc合理，清空原内容，写入新内容，同时读取

7.3.1.2 I/O流的状态

- I/O流对象中包含一个类型为ios::iostate的成员变量，表示I/O流的当前状态，可通过状态函数访问该状态

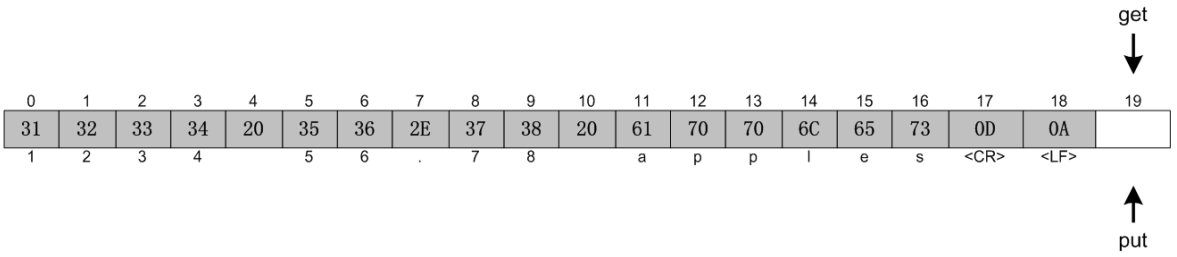
状态函数	说明
bool ios::good(void);	状态位全零即流可用返回true，否则返回false
bool ios::bad(void);	badbit位为1返回true，否则返回false
bool ios::eof(void);	eofbit位为1返回true，否则返回false
bool ios::fail(void);	badbit或failbit位为1返回true，否则返回false
iostate ios::rdstate(void);	获取当前状态
void ios::clear(iostate s=ios::goodbit);	设置（复位）状态
void ios::setstate(iostate s);	添加状态

- I/O流对象在任何时候都可被隐式转换为布尔值，只在状态位全零时，该布尔值为true，其它均为false

```
...
if (!ofs) {
    cout << "Unable to open file for writing" << endl;
    return -1;
}
...
```

7.3.2 文件位置

7.3.2.1 get和put位置




```
// position.cpp

// 文件位置

#include <iostream>
#include <fstream>

using namespace std;

int main(void) {
    // fstream类的缺省打开模式是ios::in|ios::out，
    // 以该模式打开文件，要求文件必须存在
    fstream fs("position.txt", ios::in|ios::out);
    if (!fs) {
        cout << "Unable to open file for reading and writing" << endl;
        return -1;
    }

    fs << 1234 << ' ' << 56.78 << ' ' << "apples" << '\n';

    // 在某些操作系统上，以文本方式写入流时，系统会将一个换行符自动
    // 扩展为一个回车符加一个换行符的形式（可通过hexdump -C命令查看）

    cout << fs.tellg() << endl; // 获得当前get位置
    cout << fs.tellp() << endl; // 获得当前put位置

    // get和put位置本应用独立的变量表示，其中任何一个都
    // 可以指向文件的任何位置，但大多数实现并没有这么做

    fs.seekg(ios::beg); // 设置get位置

    cout << fs.tellg() << endl; // 在将get位置调整到文件开始的同时，
    cout << fs.tellp() << endl; // put位置也随之被调整到了文件开始

    int n;
    double d;
```

```
    string str;

    fs >> n >> d >> str;

    cout << n << ' ' << d << ' ' << str << endl;

    fs.close ();

    return 0;
}
```

7.3.2.2 seek函数的两参数版本

```
istream& istream::seekg(off_type offset, ios::seekdir origin);
ostream& ostream::seekp(off_type offset, ios::seekdir origin);
```

- origin表示偏移量offset的起点
  - ios::beg表示从文件头（文件的第一个字节）偏移offset字节
  - ios::cur表示从当前位置（最近一次被读写字节的下一个位置）偏移offset字节
  - ios::end表示从文件尾（文件的最后一个字节的下一个位置）偏移offset字节
- offset表示偏移字节数
  - 正数表示向文件尾方向偏移
  - 负数表示向文件头方向偏移
- 读写指针不能被设置到文件头之前或文件尾之后，否则流状态异常

```
// seek.cpp

// seek函数的两参数版本

#include <iostream>
#include <fstream>

using namespace std;

int main(void) {
    fstream fs("seek.txt", ios::in|ios::out);
    if (!fs) {
        cout << "Unable to open file for reading and writing" << endl;
        return -1;
    }

    fs << 1234 << ' ' << 56.78 << ' ' << "apples" << '\n';

    cout << fs.tellg() << endl;
    cout << fs.tellp() << endl;

    fs.seekg(0); // seekg(0)等价于seekg(ios::beg)，将get位置移至文件开始

    cout << fs.tellg() << endl;
    cout << fs.tellp() << endl;

```

```
    int n;
    string str;

    fs >> n;
    fs.seekg(6, ios::cur); // 从当前位置向文件尾方向偏移6个字节
    fs >> str;

    cout << n << ' ' << str << endl;

    fs.close();

    return 0;
}
```

7.3.3 非格式化I/O

```
ostream& put(char ch);
int get();
istream& get(char& ch);
istream& get(char* buffer, streamsize num, char delim);
istream& getline(char* buffer, streamsize num, char delim);
```

```
// putget.cpp

// 非格式化I/O

#include <iostream>
#include <fstream>

using namespace std;

int main(void) {
    // put函数

    ofstream ofs("putget.txt");
    if (!ofs) {
        cout << "Unable to open file for writing" << endl;
        return -1;
    }

    for (char c = ' '; c <= '~'; ++c)
        // 将参数字符插入到其调用输出流中，并返回该输出流对象
        if (!ofs.put(c)) {
            cout << "Unable to write file" << endl;
            return -1;
        }

    ofs.put('\n');

    ofs.close();

    // 无参get函数

    ifstream ifs("putget.txt");
```

```
    if (!ifs) {
        cout << "Unable to open file for reading" << endl;
        return -1;
    }

    char c;

    while ((c = ifs.get()) != EOF)
        cout << c;

    if (!ifs.eof()) {
        cout << "Unable to read file" << endl;
        return -1;
    }

    // 单参get函数

    ifs.clear();
    ifs.seekg(ios::beg);

    // 单参get函数返回调用该函数的流对象，当该流已
    // 到文件尾或发生错误是返回false，否则返回true
    while (ifs.get(c))
        cout << c;

    if (!ifs.eof()) {
        cout << "Unable to read file" << endl;
        return -1;
    }

    // 三参get函数

    ifs.clear();
    ifs.seekg(ios::beg);

    char buf[4];

    // 读取从当前位置开始直到作为定界符的第三个参数之间，最多不超过第二个
    // 参数-1个字符，到第一个参数所指向的字符数组中，追加结尾空字符，并返
    // 回调用流对象。缺省定界符是换行符（\n）。每调用一次get函数，定界符
    // 并不被提取，读写指针停在定界符处，这时可通过ignore函数，跳过定界符
    while (ifs.get(buf, sizeof(buf), '\n')) {
        cout << buf;

        // 当get函数试图从定界符上读取内容时，将发生错误
        if (ifs.peek() == '\n')
            ifs.ignore();
    }

    cout << endl;

    if (!ifs.eof()) {
        cout << "Unable to read file" << endl;
        return -1;
    }

```

```
    // getline函数

    ifs.clear();
    ifs.seekg(0, ios::beg);

    char line[128];

    // 当getline函数试图读取多于第二个参数-1个字符时，将发生错误
    // getline函数会可以自动跳过定界符，无需通过ignore函数忽略之
    while (ifs.getline(line, sizeof(line), '\n'))
        cout << line;

    cout << endl;

    if (!ifs.eof()) {
        cout << "Unable to read file" << endl;
        return -1;
    }

    ifs.close();

    return 0;
}
```

7.4 二进制I/O

7.4.1 分块读写

```
istream& read(char* buffer, streamsize num);
ostream& write(const char* buffer, streamsize num);
```

```
// binary.cpp

// 分块读写

#include <iostream>
#include <fstream>

using namespace std;

int main(int argc, char* argv[]) {
    if (argc < 3) {
        cout << "Usage: " << argv[0] << " <input_file> <output_file>" <<
endl;
        return -1;
    }

    ifstream ifs(argv[1], ios::binary);
    if (!ifs) {
        cout << "Unable to open file for reading" << endl;
        return -1;
    }

    ofstream ofs(argv[2], ios::binary);
```

```
    if (!ofs) {
        cout << "Unable to open file for writing" << endl;
        return -1;
    }

    char buf[32];

    // read函数返回调用流对象，读至文件尾或出错时返回false
    while (ifs.read(buf, sizeof(buf)))
        ofs.write(buf, sizeof(buf));

    if (!ifs.eof()) {
        cout << "Unable to read file" << endl;
        return -1;
    }

    // 通过gcount函数获取最后一次读取到的字节数
    ofs.write(buf, ifs.gcount());

    ofs.close();
    ifs.close();

    return 0;
}
```

7.4.2 完整读写

```
// binentire.cpp

// 完整读写

#include <iostream>
#include <fstream>

using namespace std;

int main(int argc, char* argv[]) {
    if (argc < 3) {
        cout << "Usage: " << argv[0] << " <input_file> <output_file>" <<
endl;
        return -1;
    }

    // 在打开文件流时使用ios::ate选项，将文件的get位置置于文件尾
    ifstream ifs(argv[1], ios::binary|ios::ate);
    if (!ifs) {
        cout << "Unable to open file for reading" << endl;
        return -1;
    }

    size_t bytes = ifs.tellg(); // 通过tellg函数获取当前get位置，即文件总字节数
    ifs.seekg(ios::beg); // 将get位置重置于文件头

    char* buf = new char[bytes]; // 分配足量缓冲区

```

```
    ifs.read(buf, bytes); // 一次读取文件全部内容
    if (!ifs) {
        cout << "Unable to read file" << endl;
        return -1;
    }

    ifs.close();

    ofstream ofs(argv[2], ios::binary);
    if (!ofs) {
        cout << "Unable to open file for writing" << endl;
        return -1;
    }

    ofs.write(buf, bytes);
    if (!ofs) {
        cout << "Unable to write file" << endl;
        return -1;
    }

    ofs.close();

    delete[] buf;

    return 0;
}
```

7.4.3 字节序问题

向文件写入多字节整数，按内存地址从低到高的顺序，依次写入每一个字节：

```
// byteorder.cpp

// 从低地址到高地址写文件

#include <iostream>
#include <fstream>

using namespace std;

int main(void) {
    ofstream ofs ("byteorder.bin", ios::binary);
    if (!ofs) {
        cout << "Unable to open file for writing" << endl;
        return -1;
    }

    int n = 0x12345678;

    ofs.write((char*)&n, sizeof(n));
    if (!ofs) {
        cout << "Unable to write file" << endl;
        return -1;
    }

```

```
    ofs.close();

    return 0;
}
```

7.4.4 数组读写

```
// array.cpp

// 数组读写

#include <iostream>
#include <fstream>

using namespace std;

int main(void) {
    ofstream ofs("array.bin", ios::binary);
    if (!ofs) {
        cout << "Unable to open file for writing" << endl;
        return -1;
    }

    int oarr[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    ofs.write((char*)oarr, sizeof(oarr));
    if (!ofs) {
        cout << "Unable to write file" << endl;
        return -1;
    }

    ofs.close();

    ifstream ifs("array.bin", ios::binary|ios::ate);
    if (!ifs) {
        cout << "Unable to open file for reading" << endl;
        return -1;
    }

    size_t bytes = ifs.tellg();
    ifs.seekg(ios::beg);

    size_t num = bytes/sizeof(int);
    int* iarr = new int[num];

    ifs.read ((char*)iarr, bytes);
    if (!ifs) {
        cout << "Unable to read file" << endl;
        return -1;
    }

    ifs.close();

    for (size_t i = 0; i < num; ++i)
        cout << iarr[i] << ' ';
    cout << endl;
```

```
49
50      delete[] iarr;
51
52      return 0;
53  }
```

7.5 格式控制

7.5.1 流函数

流函数	功能
int ios::precision(int);	设置浮点精度，返回原精度
int ios::precision(void) const;	获取浮点精度
int ios::width(int);	设置显示域宽，返回原域宽
int ios::width(void) const;	获取显示域宽
char ios::fill(char);	设置填充字符，返回原字符
char ios::fill(void) const;	获取填充字符
long ios::flags(long);	设置格式标志，返回原标志
long ios::flags(void) const;	获取格式标志
long ios::setf(long);	添加格式标志位，返回原标志
long ios::setf(long, long);	添加格式标志位，返回原标志 先用第二个参数将互斥域清零
long ios::unsetf(long);	清除格式标志位，返回原标志

- 般而言，对I/O流格式的改变都是持久的，即只要不再设置新格式，当前格式将始终保持下去
- 显示域宽是个例外，通过`ios::width(int)`所设置的显示域宽，只影响紧随其后的第一次输出，再往后的输出又恢复到默认状态

格式标志	互斥域	效果
ios::left	ios::adjustfield	左对齐
ios::right	ios::adjustfield	右对齐
ios::internal	ios::adjustfield	数值右对齐，符号左对齐
ios::dec	ios::basefield	十进制
ios::oct	ios::basefield	八进制
ios::hex	ios::basefield	十六进制
ios::fixed	ios::floatfield	用定点小数表示浮点数
ios::scientific	ios::floatfield	用科学计数法表示浮点数

格式标志	互斥域	效果
ios::showpos	—	正整数前面显示+号
ios::showbase	—	显示进制前缀0或0x
ios::showpoint	—	显示小数点和尾数0
ios::uppercase	—	数中字母显示为大写
ios::boolalpha	—	用字符串表示布尔值
ios::unitbuf	—	每次插入都刷流缓冲
ios::skipws	—	以空白字符作分隔符

7.5.2 流控制符

流控制符	功能	输入	输出
left	左对齐	☐	☒
right	右对齐	☐	☒
internal	数值右对齐，符号左对齐	☐	☒
dec	十进制	☒	☒
oct	八进制	☒	☒
hex	十六进制	☒	☒
fixed	用定点小数表示浮点数	☐	☒
scientific	用科学计数法表示浮点数	☐	☒
(no)showpos	正整数前面（不）显示+号	☐	☒
(no)showbase	（不）显示进制前缀0或0x	☐	☒
(no)showpoint	（不）显示小数点和尾数0	☐	☒
(no)uppercase	数中字母（不）显示为大写	☐	☒
(no)boolalpha	（不）用字符串表示布尔值	☒	☒
(no)unitbuf	（不）每次插入都刷流缓冲	☐	☒
(no)skipws	（不）以空白字符作分隔符	☒	☐
ws	跳过前导空白字符	☒	☐
ends	空字符	☐	☒
endl	换行符，刷流缓冲	☐	☒
flush	刷流缓冲	☐	☒
setprecision(int)	设置浮点精度	☐	☒

流控制符	功能	输入	输出
setw(int)	设置显示域宽	☐	☒
setfill(int)	设置填充字符	☐	☒
setiosflags(long)	设置格式标志	☒	☒
resetiosflags(long)	清除格式标志	☒	☒

```cpp
// format.cpp

// 格式控制

#include <math.h>


#include <iostream>
#include <iomanip> // 带参数的流控制符
#include <fstream>

using namespace std;

int main(void) {
    // 流函数
    cout.precision(10);
    cout << cout.precision () << endl; // 无参precision函数，返回当前精度
    cout << sqrt(200) << endl;
    cout.setf(ios::scientific);
    cout << sqrt(200) << endl;
    cout.unsetf(ios::scientific);

    // 流控制符
    cout << setprecision(5) << sqrt(200) << ' ' <<
        scientific << sqrt(200) << resetiosflags(ios::scientific) << endl;

    // 四舍五入而非简单截断
    cout << setprecision(2) << 1.24 << ' ' << 1.25 << ' ' << 1.26 << endl;

    // 大多数用于格式化的流函数和流控制符对流的影响会一直持续到下一次对该格式的改变
    cout.precision(5);
    cout << 1.23456 << endl;
    cout << 1.234567 << endl;
    cout << setprecision(8) << 1.234567 << endl;

    // 以十进制、八进制和十六进制表示一个整数
    cout << oct << 127 << endl;
    cout << hex << 127 << endl;
    cout << dec << 127 << endl;

    // 流控制符setw仅对其下一个输出的域宽进行控制，不足域宽部分以填充字符填充
    cout << setw(7) << oct << 127 << endl;
    cout << setw(7) << hex << 127 << endl;
    cout << setw(7) << dec << 127 << endl;

```

```
    // 缺省填充字符是空格，但可以通过流函数fill或流控制符setfill指定
    cout << setfill('#') << setw(7) << oct << 127 << endl;
    cout << setfill('@') << setw(7) << hex << 127 << endl;
    cout << setfill('%') << setw(7) << dec << 127 << endl;
    cout.fill('$');
    cout << setw(7) << oct << 127 << endl;
    cout.fill('&');
    cout << setw(7) << hex << 127 << endl;
    cout.fill('*');
    cout << setw(7) << dec << 127 << endl;

    // 指定对齐方式
    cout << left << setfill('#') << setw(7) << oct << 127 << endl;
    cout << setfill('@') << setw(7) << hex << 127 << endl;
    cout << setfill('%') << setw(7) << dec << 127 << endl;
    cout.setf(ios::right);
    cout.fill('$');
    cout << setw(7) << oct << 127 << endl;
    cout.fill('&');
    cout << setw(7) << hex << 127 << endl;
    cout.fill('*');
    cout << setw(7) << dec << 127 << endl;

    // 必须为每个输出独立设置域宽，即使它们的域宽相同
    cout << setfill('#') << setw(7) << oct << 127 << hex << 127 << dec <<
127 << endl;
    cout << setw(7) << oct << 127 << setw(7) << hex << 127 << setw(7) <<
dec << 127 << endl;

    // 显示进制前缀
    cout << showbase << oct << 127 << endl;
    cout << hex << 127 << endl;
    cout << dec << 127 << endl;
    cout.setf(ios::showbase);
    cout << oct << 127 << endl;
    cout << hex << 127 << endl;
    cout << dec << 127 << endl;

    // 不显示进制前缀
    cout << noshowbase << oct << 127 << endl;
    cout << hex << 127 << endl;
    cout << dec << 127 << endl;
    cout.unsetf(ios::showbase);
    cout << oct << 127 << endl;
    cout << hex << 127 << endl;
    cout << dec << 127 << endl;

    // 显示小数点
    cout << 12.00 << endl;
    cout << setprecision(10) << 12.00 << endl;
    cout << showpoint << 12.00 << endl;
    cout.setf(ios::showpoint);
    cout << 12.00 << endl;

    // 不显示小数点
    cout << noshowpoint << 12.00 << endl;
```

```
    cout.unsetf(ios::showpoint);
    cout << 12.00 << endl;

    // 科学计数法和定点小数
    cout.precision(4);
    cout << sqrt(200) << endl; // 对于定点小数法，精度限制整个浮点数的显示位数
    cout.setf(ios::scientific);
    cout << sqrt(200) << endl; // 对于科学计数法，精度限制底数小数的显示位数
    cout.precision(8);
    cout << sqrt(200) << endl;
    cout.unsetf(ios::scientific);
    cout << sqrt(200) << endl;

    // 流控制符noskipws或流函数unsetf(ios::skipws)用于取消跳过空白
    // 流控制符skipws或流函数setf(ios::skipws)用于启用跳过空白
    ifstream ifs("format.txt");
    if (!ifs) {
        cout << "Unable to open file for reading" << endl;
        return -1;
    }

    ifs.unsetf(ios::skipws);

    char c;

    while (ifs >> c)
        cout << c;

    cout << endl << "---------" << endl;

    ifs.clear();
    ifs.seekg(ios::beg);

    ifs.setf(ios::skipws);

    while (ifs >> c)
        cout << c;

    cout << endl << "---------" << endl;

    ifs.close();

    cout.width(10);
    cout.fill('-');
    cout.setf(ios::internal, ios::adjustfield);
    cout.setf(ios::showpos);
    cout << 12345 << endl;

    return 0;
}
```

7.6 字符串流

```
// sstream.cpp

```

```cpp
// 字符串流

#include <iostream>
#include <sstream>

using namespace std;

int main(void) {
    // ostringstream对象的一个优点是，可以利用插入操作符（<<）
    // 的格式化能力把数值类型的数据转换为它们的字符串表示形式

    ostringstream oss;
    oss << 1234 << ' ' << 56.78 << ' ' << "apples";

    // 通过字符串流的str成员函数，访问与之相关联的字符串对象

    cout << oss.str() << endl;

    // istringstream对象的一个优点是，可以利用提取操作符（>>）
    // 的格式化能力从字符串中解析出不同类型的数值

    // istringstream iss(oss.str());
    istringstream iss;
    iss.str(oss.str());

    int n;
    double d;
    string str;

    iss >> n >> d >> str;

    cout << n << ' ' << d << ' ' << str << endl;

    return 0;
}
```

7.7 流缓冲

缺省情况下，所有C++流都是带缓冲的。与一个流相关联的缓冲区是streambuf类型的对象。缓冲意味着当向一个输出文件流写入数据时，所输出的字节在一般情况下并不会立即被写到文件中。相反，它们会被暂时转移到一块作为输出流缓冲区的内存块中。如有以下情况之一发生：

- 缓冲区被填满
- 流被关闭
- 显式调用该流的flush成员函数或向该流插入endl控制符
- 调用该流的sync成员函数以建立同步

流缓冲区立即被刷新，其中所有的数据全部转移到输出文件中。