来一份代码以防误删:

#include<iostream>

#include<fstream>

#include<iomanip>

using namespace std;

bool fx(int n) {//判断是否为质数

for (int i = 2; i \* i <= n; i++) {

if (n % i == 0) {

return false;

}

}

return true;

}

int main() {

ofstream outfile("ASCII.txt", ios::out | ios::\_Nocreate | ios::trunc);

if (!outfile) {

cerr << "Open Error" << endl;

}

outfile << "输出到文本:" << endl;

int mark = 0;

for (int i = 2; i <= 100; i++) {

if (fx(i)) {

mark++;

outfile.setf(ios::left);

outfile << setw(6) << i;

if (mark % 10 == 0) {

outfile << endl;

}

}

}

outfile.close();

char c;

c = 'y';

int b[100] = {};

mark = 0;

int i = 0;

if (c == 'y' || c == 'Y') {

ifstream infile("ASCII.txt", ios::in);

if (!infile) {

cerr << "Open Error" << endl;

}

char c;

while (infile.get(c)) {

if (c == '\n') {

break;

}

}

while (infile >> b[i]) {

mark++;

cout.setf(ios::left);

cout << setw(6) << b[i];

if (mark % 10 == 0) {

cout << endl;

}

i++;

}

if (mark % 10 != 0) {

cout << endl;

}

infile.clear();

infile.close();

}

fstream bit("Binary.txt", ios::out | ios::in|ios::binary);

if (!bit) {

cerr << "Open Error" << endl;

}

for (int j = 0; j < i; j++) {

bit.write((char\*)&b[j], sizeof(int));

}

int m[100] = {};

bit.clear();

bit.flush();

//bit.seekp(0, ios::cur);

cout << bit.tellg() << endl;

char ch;

bit.get(ch);

cout << bit.tellg() << endl;

for (int j = 0; j < i+2; j++) {

if (bit.eof()) {

break;

}

bit.read((char\*)&m[j], sizeof(int));

cout << bit.tellg() << endl;

}

bit.clear();

bit.close();

return 0;

}

1. 首先，eof和EOF是不同的，在fstream声明的对象bit中有一个eof状态，在文件stdio.h中有一个EOF的宏定义-1。
2. eof状态十分的特殊，当该对象设置为eof状态时，许多原本可以正常使用的函数都将不能使用，或者说返回值将会十分特殊，比如-1。在上述的代码中就有cout<<bit.tellg()<<endl;其在显示器上输出就是-1，但是指针是不可能指到-1这个位置的。
3. 当流对象的设置eof状态时，可以使用clear()函数来重置这个状态，才可以使流对象再次得以使用。
4. 一般情况下流对象指针按顺序从头到尾正常到结尾再次向后读取时设置eof，此后再次关联其它文件但是eof状态并未重置，会导致作为其它文件关联时，无法正常调用函数，解决方法就是调用clear()函数。
5. 经过多次测试，发现调用seekp()函数时，除了一种情况，其它均会将流对象中的内容输出到文件中(但目前并不知道输出后该文件流中的内容是否清空【下面又说】，感觉会，但是输出当前指针位置时它并不是显示0【因为输出指针永远是文件指针,而代码里使用过read缓冲区指针和文件指针都向后4,所以不会显示0】)，也就是说文件大小将会改变。
6. 5中唯一情况为seekp(0,ios::cur);也就是根据当前指针位置，不动。这句话无论放在什么情况下都是一句废话，这句话的有无在任何情况下都不会改变指针位置。但是我们不能理解为，指针不动就不会将缓冲区输出到文件，因为上述代码在执行seekp(0,ios::end)时，指针本身就在文件的末尾，实质上并没有改变位置，但这并不是本质，或者说，这句话在其它情况下是有作用的，比如指针在开头时，执行这句话就会将指针移动到文件末尾。对于第一种特殊废物情况，函数内部可能直接判断是否为这种情况然后直接什么都不做退出来。而执行seekp(1,ios:::cur)或seekp(-1,ios::cur)都会将文件流输出。【当然seekp(0,ios::beg)也会输出文件流】。
7. 就算文件流指针在不断地游走，输出时都还是从文件的首开始的，无论什么情况都一样，我也不知道怎么移动文件的指针【看来理解错了，移动的永远是文件的指针，也就是说我其实只能移动文件指针】。当文件原本大小超过将要输出的文件流的大小时，执行seekp或者close等带有输出的操作时，文件本身小于文件流的就不用说了。当文件超过文件流大小时，~~文件超过文件流大小的部分也一起消失~~【不会消失，忘了当时是什么情况】，~~也就是说，文件将会被清空后再输出到文件(这一点还只是猜测的)，这也就是为什么eof函数在没有正式输出之前无法正确判断是否为文件末尾的原因，因为原本文件本身的大小根本就没被关注，自然不晓得eof在哪。eof函数的判断标准据说就是通过比较指针位置和文件大小，文件大小未知，固然无法正确比较。~~
8. Ios::end这句话很厉害，在执行这句话之前，文件并没有输出，也就是说文件的大小未知，而end是根据文件大小来放置的，这很矛盾，但是若将这一句话分解为三个操作，先将文件流输出，文件大小就有了，再将指针放在文件大小处，完成。
9. 现阶段认为，read,wirte都将从同一个缓冲区中读入或写入数据，指针永远向后走，这里面并没有涉及到文件的大小，也没有一句有关输出的代码，也就更本没有eof这种说法，只要条件允许，指针可以一直往后走。
10. 检查read和write是否在同一个缓冲区的方法就是反复调用这两个函数并查看最后输出。经测试，确实是同一个缓冲区。
11. 疑问：为什么只有输出之后才能检测eof，虽然上面有说到文件没大小就不能判断，但是没大小就不能理解为0吗?
12. 答案：因为read读取的是缓冲区，而eof出现的条件是文件尾再读取，read被困在了缓冲区读不了文件就无法出eof。
13. 有一个比较有趣的测试，当我open一个空的二进制文件时，此时执行read函数，立刻意识到这是文件末尾。【因为此时没有write创建缓冲区，所以read可以访问到文件】
14. write为read创建了一个缓冲区，read将实现在不知晓文件大小的情况下无限向后移动的功能。因为一般来说，read是ifstream的函数，关联时文件大小就应当已知【这句话还是有问题，因为没有缓冲区，能访问到文件，才能知道是否是文件末尾】，文件大小为0自然无法向后。而此时作为一个fstream声明的二进制文件，即可输入又可输出，read函数混乱了，它本应该读取文件的内容，但此时指针却引向了write创建的缓冲区，也就是说write创建的缓冲区的优先级更高，read先访问缓冲区而不是访问文件，也就不管文件的大小了。而不用write创建缓冲区，read将直接访问文件，若文件是空的，直接就会知道是文件末尾，就会返回-1。【并没有报错，之前报错是因为没有取地址】
15. 输出会清空缓冲区(测试方法:多次使用flush)【不是真正意义上的清除】

综上所述，当使用fstream声明bit时，并且使用ios::in|ios::out|ios::binary时，write函数为read函数创建缓冲区时，read函数将会优先读取缓冲区里的内容，也就是将目光转向了缓冲区而非文件，也就不在知晓文件到底在哪结束了，好比ntr中的女主read被黄毛缓冲区勾走后苦主文件无人问津。

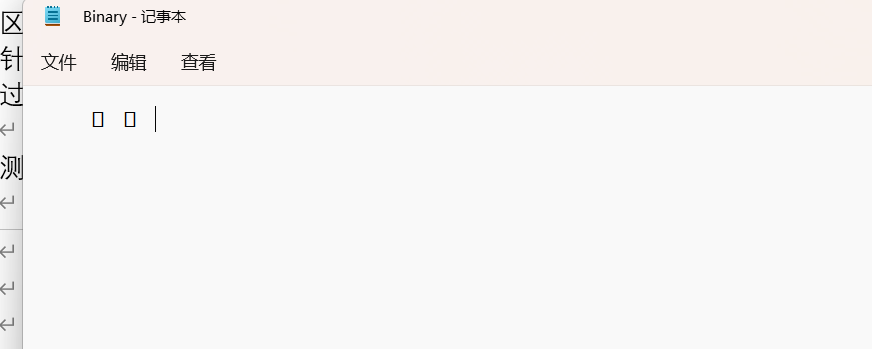
当然我还有另一种解释，这就要求对read函数有深入理解了，(以下全是我的胡扯)，在这里read函数是将缓冲区的内容赋给程序内部变量的(而非从文件读取到变量)，而缓冲区的内容是在与文件建立关联时得到的。此时指针指在缓冲区的0处Write函数也从0开始写，直接完全覆盖上去，read再去读的时候已经全部被write重写过了。【果然是错的】

关键：测试是否在建立关联时给缓冲区赋值

答：没有

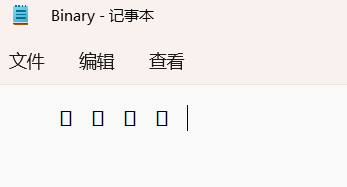
猜想（与上下文无关联）：read函数的目的是为了让缓冲区里存在数据，所以当缓冲区没有数据时，read会从文件中读取数据，并将两个指针都向后移动，当缓冲区里本身存在数据，就没有必要从文件中读取数据了，但是仍会将两个指针向后移动。

当使用ios::trunc时



0 7 14 21

当没有使用ios::trunc时



0 7 14 21 5 6

原始文件为1 2 3 4 5 6

疑问:5 6 到底是原先文件里的还是和0 7 14 21一起输出的？

~~当我将文件指针放置在末尾处，write一个int数后flush，其输出的只是在末尾添加了一个数，也就是说，在建立关联后并没有将所有内容放置在缓冲区中，否则应当是原先数据的两倍在加上这个数。~~

这句话有问题，若关联后文件全部存入缓冲区的假设成立，当我将文件指针放置到末尾时也就输出了，文件内容照假设来说就是用同样的数据从头覆盖上去了，此时write一个int后flush也是仅在文件末尾增加一个数而不是两倍后加一个数，所以这个测试不成立，假设并未被证实或证否。

不过嘛，网上说缓冲区并没有在建立关联时填充。

最后吐糟一点：文件输出时都是从头开始的，若输出的长度不够长，原文件多出来的内容就不会改变，所以若想合适地改变原文件的内容，~~应当read所有数据后输出~~【这句没用】，应当保证输出数据比原先长或相等。想像word一样只修改部分文字应该是不可能的了。

【其实是有可能的，因为所有对指针移动的操作都是对文件指针移动的操作，也就是说输出的位置是可以选择的，不过嘛移动指针可是会将缓冲区直接输出的，所以在write前应当事先把文件指针位置调好】

测试：移动指针操作到底是先输出呢，还是先移动呢？

结果：~~先从头开始输出~~【是从当前文件指针位置开始输出】，再移动指针

当文件为空时，先write一个数后设置文件末尾，文件得到内容，输出指针当前位置为4

当文件本身包含两个数，先write一个数后设置文件末尾，文件第一个数被修改，后面不变，输出当前指针位置为8

这两句黄的我忘了当时想表达什么

int p = 99;

bit.write((char\*)&p, sizeof(int));

p = 102;

bit.write((char\*)&p, sizeof(int));

bit.seekp(0, ios::end);

cout << bit.tellg() << endl;

int o = 84;

bit.write((char\*)&o, sizeof(int));

int m[100] = {};

bit.clear();

bit.flush();

结论：

Write创建文件流缓冲区，往里面写入数据，此后执行seekp(0,ios::end)时，文件流将会被输出，~~缓冲区被清空，或者说消失~~，缓冲区数据被设置为无效数据，缓冲区指针回到0处，即开头处，【但是缓冲区的数据并没有被清除，就像delete一样，以供下一次使用】，当原文件大于文件流时，与文件流大小相等的那一块将会被修改，(注：此时文件的指针一直都是指在0处的，而不是和缓冲区里的指针一致，但可以事先修改文件指针位置，方法在上面)，大于文件流的部分将不会被修改，但是会影响到seekp(0,ios::end)的值，因为新的文件已经产生，但是原本多出来的部分并没有消失而是成为了新文件的一部分。此时因为缓冲区 ~~已经消失~~，缓冲区的指针 ~~就已失去意义~~ 置0，此时输出当前指针位置是根据新文件的大小来的，也就是文件指针。

【缓冲区消失这一点是错的，以后再改】

int p = 103;

bit.write((char\*)&p, sizeof(int));

p = 105;

bit.write((char\*)&p, sizeof(int));

bit.seekp(0, ios::end);

cout << bit.tellg() << endl;

int o = 86;

bit.write((char\*)&o, sizeof(int));

int m[100] = {};

for (int i = 0; i < 100; i++) {

m[i] = -1;

}

for (int j = 0; j < 10; j++) {

if (bit.eof()) {

break;

}

cout << bit.tellg() << endl;

bit.read((char\*)&m[j], sizeof(int));

cout << bit.tellg() << endl;

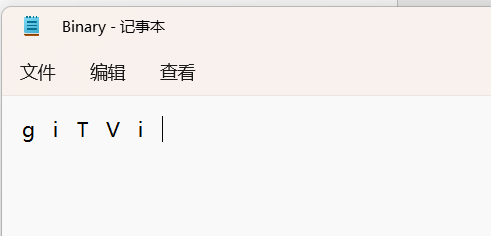
}

bit.clear();

bit.flush();

cout << bit.tellg() << endl;

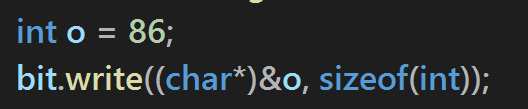
bit.flush();



原文件c f T ，在执行上述代码之后得到了上图

c=99 f=102 T=84 V=86 i=105

一开始对输出V i很奇怪，但是现在理解了。

V的出现就是这句话的问题，其后执行到flush时就会输出V。

但是这个i的出现就很有问题，我明明没有把i write进缓冲区，只是read的时候确实读了个105到m数组里，却在输出时带上来i。

一种理解就是，write创建了缓冲区，往缓冲区里写入了g i两个数，执行seekp(0,ios::end)后输出覆盖原文件c f，此时文件指针指在图二中的T之后，【但是缓冲区假清空，缓冲区指针移到0处即g i前】，之后执行write，写了个V覆盖了g【此时的缓冲区为V i，V是正常的，i可能还是被设置为不可使用状态(经测试，确实还是不可使用状态，方法：取消read后flush只有V输出)】其后read再读入时 ~~将文件里的东西先要读到缓冲区里再赋给变量~~ 读取的是缓冲区的内容【read读了i,并将i活化了，可使用了】。但是之后并没有将缓冲区里的内容给清空，它也应该不能清空，因为它不能干涉到其它的数据。【此时缓冲区数据为V 和i,都可正常输出】所以flush输出时会将i也一并输出。【修改后上下两段差不多】

至于为什么会跳过g，我有一个设想，首先缓冲区并没有真正意义上的消失，它只是在输出过的缓冲区设置了一种无法读取的状态，因为执行过seekp(0,ios::end)，缓冲区的指针重新指向0处，而此时write执行后，覆盖了原先的g，并把指针向后移动4个字节，也就是移到了i的前面，此时read就会直接读取i(为什么不读取文件前面已经说过了，有缓冲区先读缓冲区)，并向后移动4个字节，因为这是原先的缓冲区【里面只有c f】，所以它根本就不知道T的存在，其后继续read读取到的都是空的数据。唯一有疑问的就是输出当前指针位置时，它并不是从0开始。

经测试，将原先c f T改成c f T T后，其输出从g i T V i变成d i j T V i j，基本可以证明上述说明是正确的。

那么指针输出问题的答案就是，输出的是文件的指针而非缓冲区的指针，而read函数的操作是同时移动缓冲区指针和文件指针。【这也是为什么read执行后再输出会把文件扩大(即使read到的都是空的数据)】<——其实这句话因果性不强，不能理解就无视吧。

具体过程需要用到这个操作。

让我注意到这一点的主要是监视里的m数组，之前read到m数组是，m数组总是0，而这一次操作时，m数组竟然读到了105(或者105、106)，这是一开始我无法理解的。我将m数组全置-1的原因是，不知道read后到底有没有对m数组执行赋值操作，毕竟0变成0看不出开。经测试，read后除第一个数据改为105，其后的-1均变为0。

重点：1、缓冲区并没有真正意义上的清空，只是原先输出过的数据被设置了不可使用的状态。

2、tellg()输出的是文件指针而非缓冲区指针。