



# 输入/输出流

吴清锋



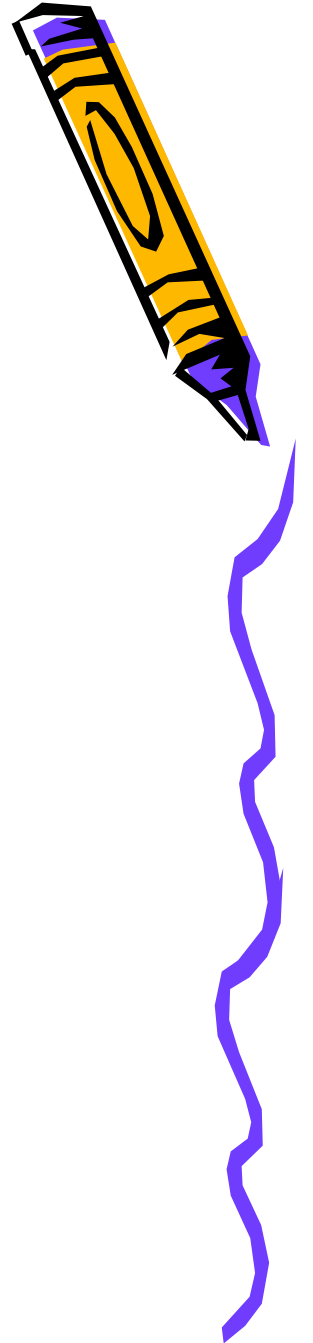
# 目 录

- C++流的概念
- 格式化I/O
- 检测流操作的错误
- 文件流
- 字符串流



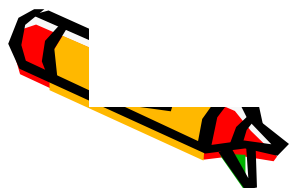
# 回忆：C语言中的文件

- 为何要讨论C语言中的文件？
  - 数据能够长期存放
- 如何让C程序来定位和处理文件？
  - 文件指针：是一桥梁
- 文件是如何操作的？



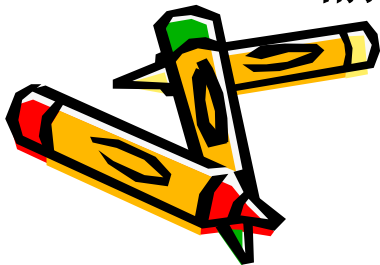
# 概述

- C++语言中**没有**专门的输入/输出（I/O）语句；
- C++中的I/O操作是通过一组**标准I/O函数**和**I/O流**来实现的。
  - C++的标准I/O函数是从C语言**继承**而来的，同时对C语言的标准I/O函数进行了**扩充**。  
即： `printf`, `getchar`, `gets`等
  - C++的**I/O流**不仅拥有标准I/O函数的功能，而且比标准I/O函数功能更强、更方便、更可靠。



# C++流的分类

- 在C++语言中，数据的I/O，包括：
  - 标准I/O: 对标准输入设备键盘和标准输出设备显示器
  - 文件I/O: 对在外存磁盘上的文件
  - 串I/O: 对内存中指定的字符串存储空间进行输入输出



# C++流的流向

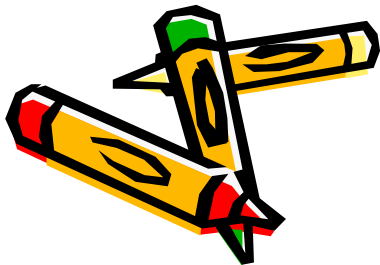
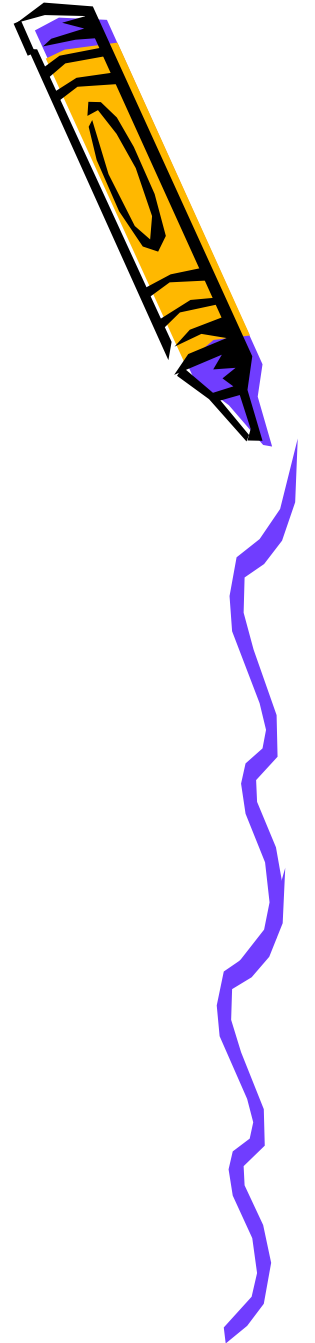
C++中把数据之间的传输操作称作流。流包括输入和输出流，它们是相对于内存而言的。

- 在C++中，流既可以表示数据从内存传送到某个载体或设备中，即输出流；一个输出流对象是信息流动的目标。
- 也可以表示数据从某个载体或设备传送到内存缓冲区变量中，即输入流。一个输入流对象是数据流出的源头



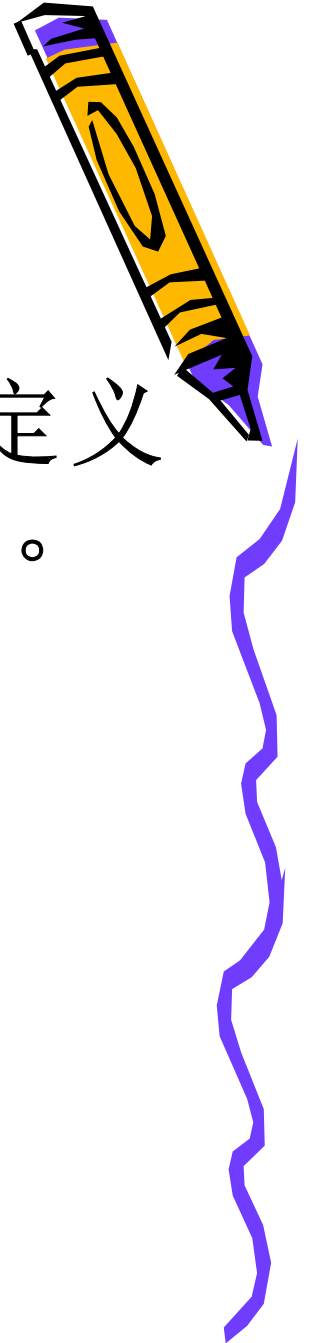
# 如何展开讨论？

- 包含的头文件
- 类
- 对象
- 运算符
  - 运算符重载



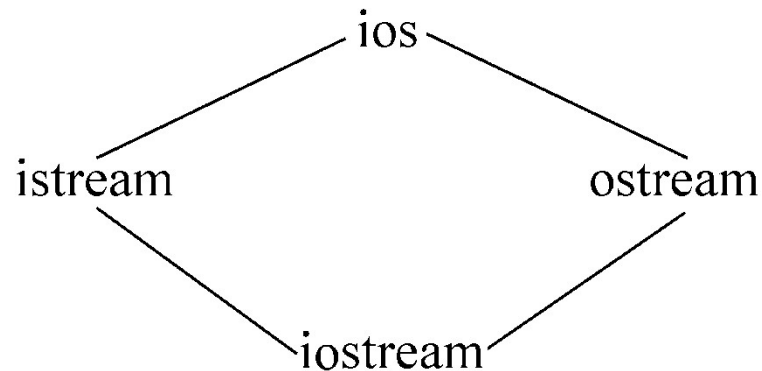
# C++流的类体系结构

- C++为实现数据的输入和输出定义了一个庞大的类库（**I / O**类库）。



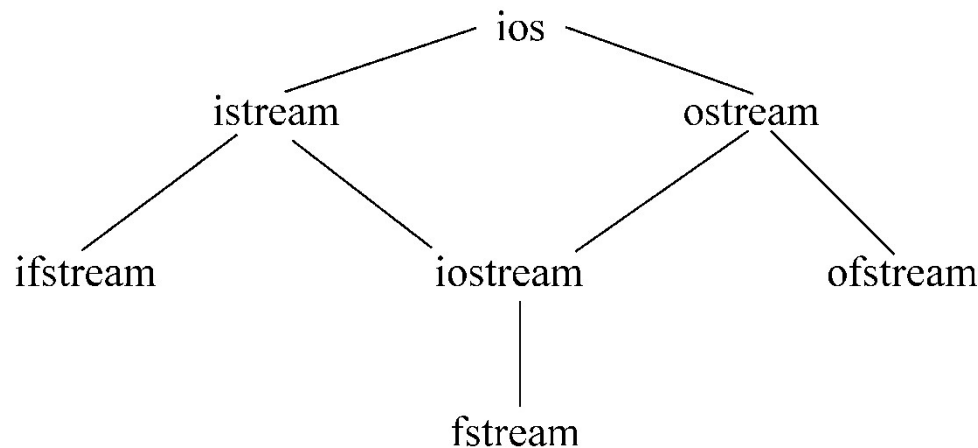


# iostream类库中各类间的关系图如下所示：



其中**ios**为抽象基类

- 输入流类**istream**：支持输入操作
- 输出流类**ostream**：支持输出操作
- iostream**：支持输入输出操作.是从上述两类通过多重继承而派生的类




C++对**文件的输入输出**需要用：  
`ifstream`和**`ofstream`**类

- `ifstream`支持对文件的输入操作
- `ofstream`支持对文件的输出操作
- 类**`ifstream`**继承了类**`istream`**
- 类**`ofstream`**继承了类**`ostream`**
- 类**`fstream`**继承了类**`iostream`**

# 包含的头文件



- 类库中不同的类的声明被放在不同的头文件中。包含头文件的过程，就是用户在本程序中声明所需要类的过程。
- 在一个程序或一个编译单元（即一个程序文件）中当需要进行**标准I/O**操作时，则必须包含头文件**iostream**,
- 当需要进行**文件I/O**操作时，则必须包含头文件**fstream**,
- 同样，当需要进行**字符串I/O**操作时，则必须包含头文件**sstream**。
-  当使用**setw, fixed**等**操作符进行格式控制**，包含头文件**iomanip**

- 标准的I/O



# iostream头文件中预定义对象

C++不仅定义有现成的I/O类库供用户使用，而且还为用户进行标准I/O操作定义了四个类对象，它们分别是**cin**，**cout**，**cerr**和**clog**，其中：

- **cin**为istream流类的对象，代表：标准输入设备键盘，也称为**cin流**或标准输入流，
- 后三个为ostream流类的对象，**cout**代表：标准输出设备显示器，也称为**cout流**或标准输出流，**cerr**和**clog**含义相同，代表：错误信息输出设备显示器。
- 因此当进行键盘输入时使用**cin流**，当进行显示器输出时使用**cout流**，当进行错误信息输出时使用**cerr**或**clog**。



## 关于cerr

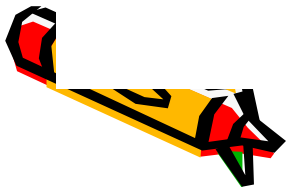
**cerr**与**cout**的差别在于：

(1) **cerr**是**不能重定向**的（即无法到磁盘文件）；

(2) **cerr****不能被缓冲**，它的输出总是直接传达到标准输出设备上。

如果使用下列语句，则错误信息“**Error**”总能保证在显示器上显示出来：

```
cerr << “Error” << “\n”;
```





## 关于 **clog**

- **clog(console log)**是不能重定向的，但是可以被缓冲。
- 在某些系统中，由于缓冲，使用**clog**代替**cerr**可以改进显示速度：

### **cerr**和**clog**的选择

**cerr**对象可用于给程序的用户生成警告或错误信息；

**clog**对象用于生成程序的执行情况信息。



有一元二次方程 $ax^2+bx+c=0$ ，其一般解为

$x_{1,2}=(-b \pm \sqrt{b^2-4ac})/2a$ ，但若 $a=0$ ，或 $b^2-4ac<0$ 时，用此公式出错。

编程序，从键盘输入 $a,b,c$ 的值，求 $x_1$ 和 $x_2$ 。如果 $a=0$ 或 $b^2-$

▶  $4ac<0$ ，输出出错信息。

```
#include <iostream>
```

```
#include <cmath>
```

```
using namespace std;
```

```
int main( )
```

```
{float a,b,c,disc;
```

```
cout<<"please input a,b,c:";
```

```
cin>>a>>b>>c;
```

```
if (a==0)
```

```
cerr<<"a is equal to zero,error!"<<endl;
```

```
//将有关出错信息插入cerr流，在屏幕输出
```

```
else
```



```
if ((disc=b*b-4*a*c)<0)
```

```
    cerr<<"disc=b*b-4*a*c<0"<<endl; //将有关出错信息插入  
                                         //cerr流，在屏幕输出
```

```
else {
```

```
    cout<<"x1="<<(-b+sqrt(disc))/(2*a)<<endl;
```

```
    cout<<"x2="<<(-b-sqrt(disc))/(2*a)<<endl;
```

```
}
```

```
return 0;
```

```
}
```

运行情况如下：

① please input a,b,c: 0 2 3✓

a is equal to zero,error!

② please input a,b,c: 5 2 3✓

sc=b\*b-4\*a\*c<0

③ please input a,b,c: 1 2.5 1.5✓

x1=-1

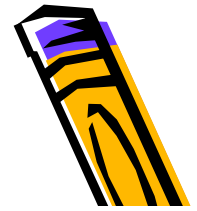
x2=-1.5



- 自定义类型对象的**I/O**



# <<、>>运算符的基本功能



- 基本功能

- “<<” 和 “>>” 能用作标准类型数据的输入和输出运算符;

- 在用 “**cout<<**”输出基本类型的数据时，可以不必考虑数据是什么类型，系统会判断数据的类型，并根据其类型选择调用与之匹配的运算符。

- 例子: **cout<<“C++”;**

实际解读为: **cout.operator<< (“C++”)**



# <<、>>运算符的重载

- 若想将<<和>>应用于自己声明的类型，就需要对运算符进行重载

```
friend ostream & operator << (ostream & dest, Date  
    &date)
```

```
friend istream & operator >> (istream & sour, Date  
    &date)
```





# 格式控制

- 在输出数据时，希望数据按指定的格式输出：
  - 第一种方法：控制符的方法
  - 第二种方法：使用流对象的有关成员函数





## 第一种方法：格式控制操作符

- 使用系统头文件**iomanip**中提供的**操纵符**。
- 使用这些操纵符不需要调用成员函数，只要把它们作为插入操作符<<（个别作为提取操作符>>）的输出对象即可。
- 格式控制操作符，分为：
  - 带参数
  - 不带参数





- 操纵符中: **dec**, **oct**, **hex**, **endl**, **ends**, **flush**和**ws**除了在**iomanip.h**中有定义外, 在**iostream.h**中也有定义。
- 所以当程序或编译单元中只需要使用这些不带参数的操纵符时, 可以只包含**iostream.h**文件, 而不需要包含**iomanip.h**文件。

- 关于进制的信息: **dec oct hex**

- 关于格式的信息: **flush**

**setiosflags (long f)**

**resetiosflags (long f)**

**setfill (int c)**

**setprecision (int n)**

**setw (int w)**



## 例 用控制符控制输出格式。

```
#include <iostream>
#include <iomanip> //不要忘记包含此头文件
using namespace std;
int main()
{int a;
  cout<<"input a:";
  cin>>a;
  cout<<"dec:"<<dec<<a<<endl;           //以十进制形式输出整数
  cout<<"hex:"<<hex<<a<<endl;           //以十六进制形式输出整数a
  cout<<"oct:"<<setbase(8)<<a<<endl;     //以八进制形式输出整数a
  char *pt="China";                      //pt指向字符串"China"
  cout<<setw(10)<<pt<<endl;               //指定域宽为10，输出字符串
  cout<<setfill('*')<<setw(10)<<pt<<endl; //指定域宽10，输出字符串，空白处以'*'填充
  double pi=22.0/7.0;                    //计算pi值
  cout<<setiosflags(ios::scientific)<<setprecision(8); //按指数形式输出，8位小数
  cout<<"pi="<<pi<<endl;                 //输出pi值
  cout<<"pi="<<setprecision(4)<<pi<<endl;   //改为4位小数
  cout<<"pi="<<setiosflags(ios::fixed)<<pi<<endl; //改为小数形式输出
  return 0;
}
```



## 第二种方法：用流对象的成员函数控制输出格式






- 除了可以用控制符来控制输出格式外，还可以通过调用流对象**cout**中用于控制输出格式的成员函数来控制输出格式。
- 流成员函数**setf**和控制符**setiosflags**括号中的参数表示格式状态，它是通过格式标志来指定的。格式标志在类**ios**中被定义为枚举值。因此在引用这些格式标志时要在前面加上类名**ios**和域运算符“**::**”。





## 例：用流控制成员函数输出数据。

```
#include <iostream>
using namespace std;
int main( )
{int a=21
  cout.setf(ios::showbase);//显示基数符号(0x或0)
  cout<<"dec:"<<a<<endl;      //默认以十进制形式输出a
  cout.unsetf(ios::dec);      //终止十进制的格式设置
  cout.setf(ios::hex);        //设置以十六进制输出的状态
  cout<<"hex:"<<a<<endl;      //以十六进制形式输出a
  cout.unsetf(ios::hex);      //终止十六进制的格式设置
  cout.setf(ios::oct);        //设置以八进制输出的状态
  cout<<"oct:"<<a<<endl;      //以八进制形式输出a
  cout.unsetf(ios::oct);
  char *pt="China";          //pt指向字符串"China"
  cout.width(10);             //指定域宽为10
  cout<<pt<<endl;            //输出字符串
  cout.width(10);             //指定域宽为10
```



```
cout.fill('*');           //指定空白处以'*'填充
cout<<pt<<endl;          //输出字符串
double pi=22.0/7.0;       //输出pi值
cout.setf(ios::scientific); //指定用科学记数法输出
cout<<"pi=";              //输出"pi="
cout.width(14);            //指定域宽为14
cout<<pi<<endl;           //输出pi值
cout.unsetf(ios::scientific); //终止科学记数法状态
cout.setf(ios::fixed);     //指定用定点形式输出
cout.width(12);            //指定域宽为12
cout.setf(ios::showpos);   //正数输出“+”号
cout.setf(ios::internal);  //数符出现在左侧
cout.precision(6);         //保留6位小数
cout<<pi<<endl;           //输出pi，注意数符“+”的位置
return 0;
}
```



运行情况如下:

dec:21(十进制形式)

hex:0x15 (十六进制形式, 以0x开头)

oct:025 (八进制形式, 以0开头)

China (域宽为10)

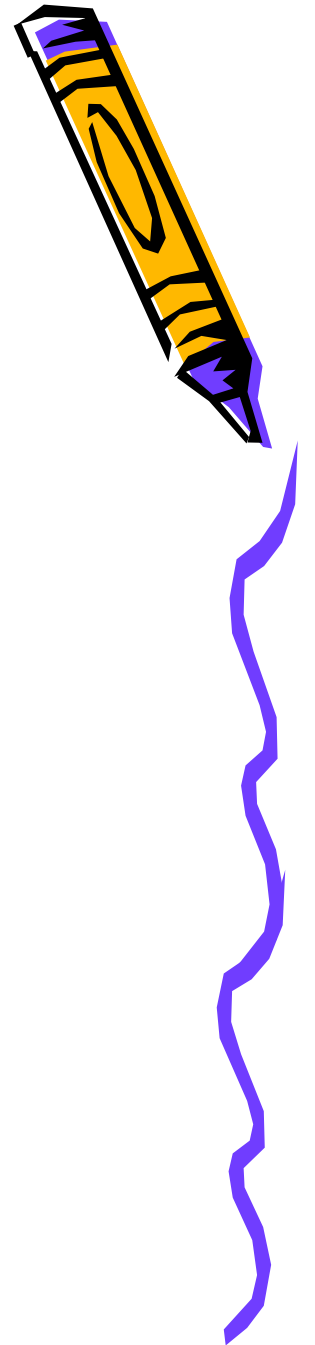
\*\*\*\*\*China (域宽为10, 空白处以'\*'填充)

pi=\*\*3.142857e+00 (指数形式输出, 域宽14, 默认6位小数)

+\*\*\*3.142857 (小数形式输出, 精度为6, 最左侧输出数符“+”)

,

- 面向文件的**I/O**



# 文件流

- 文件概述

在磁盘上保存的信息是**按文件的形式组织**的，每个文件都对应一个文件名，并且属于某个物理盘或逻辑盘的目录层次结构中一个确定的目录之下。

一个文件名由**文件主名和扩展名**两部分组成，它们之间用圆点（即小数点）分开，扩展名可以省略，当省略时也要省略掉前面的圆点。





- 在C++程序中使用的保存数据的文件按存储格式分为两种类型：

- 一种为字符格式文件，简称**字符文件**，  
字符文件又称**ASCII码文件**或**文本文件**
- 另一种为内部格式文件，简称**字节文件**  
字节文件又称**二进制文件**。

- 在字符文件中（文本文件），每个字节单元的内容为字符的**ASCII码**，被读出后能够直接送到显示器或打印机上显示或打印出对应的字符，供人们直接阅读。
- 在字节文件中（二进制文件），**文件内容是数据的内部表示**，是从内存中直接复制过来的。
- 当然对于**字符信息**，数据的内部表示就是ASCII码表示，所以在字符文件和在字节文件中保存的字符信息**没有差别**，
- 但对于**数值信息**，数据的内部表示和ASCII码表示**截然不同**，所以在字符文件和在字节文件中保存的数值信息也截然不同。



# 步骤概述（给出图示意）



- 要在程序中使用文件时，首先要在开始包含 **#include<fstream>** 命令。
- 由它提供的输入文件流类 **ifstream**、输出文件流类 **ofstream** 和输入输出文件流类 **fstream** 定义用户所需要的 **文件流对象**。
- 然后利用该对象（直接）或调用相应类中的 **open成员函数（间接）**，按照一定的 **打开方式** 打开一个文件。
- 文件被打开后，就可以通过流对象访问（读写）它（实质是通过函数来实现）。
- 访问结束后再 **通过流对象关闭** 它。



# 文件的打开

- 流可以分为3类：输入流、输出流以及输入/输出流，相应地必须将流说明为**ifstream**、**ofstream**以及**fstream**类的对象。例如：

**ifstream ifile;**      //说明一个输入流对象

**ofstream ofile;**      //说明一个输出流对象

**fstream iofile;**      //说明一个输入/输出流对象

- 定义了流对象之后，**建立对象与文件的关联**，有两种方式：

1、直接方式：定义对象的同时，来建立关联；

2、间接方式：可使用函数**open( )**打开文件；**文件的打开即是在流与文件之间建立一个连接。**

**open ( )** 的函数原型为：

**void open(const char \* filename,int mode,int prot=filebuf::openprot);**



`void open(const char * filename,int mode,int prot=filebuf::openprot);`

其中，

- **filename**是文件名字，它可包含路径说明。
- **mode**说明文件打开的模式，它对文件的操作影响重大，**mode**的取值必须是以下值之一：

**ios::in**            打开文件进行读操作

**ios::out**          打开文件进行写操作

**ios::ate**          打开时文件指针定位到文件尾

**ios::app**          添加模式，所有增加都在文件尾部进行

**ios::trunc**        如果文件已存在,则清空原文件

**ios::nocreate**    如果文件不存在则打开失败

**ios::noreplace**   如果文件存在则打开失败

**ios::binary**      二进制文件（非文本文件）



对于**ifstream**流，**mode**的默认值为**ios::in**；对于**ofstream**流，**mode**的默认值为**ios::out**。



- **prot**决定文件的访问方式，取值为：

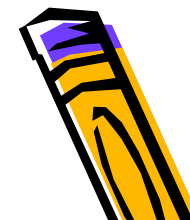
- 0 普通文件
- 1 只读文件
- 2 隐含文件
- 4 系统文件

一般情况下，该访问方式使用默认值。

- 值得一提的是，与其他状态标志一样，**mode**的符号常量可以用位或运算“|”组合在一起，如：  
**ios::in|ios::binary**表示以只读方式打开二进制文件。



# 文件打开的路径书写



- 打开文件时，可以指定绝对路径

例：打开C盘custom文件夹中invtry.dat

```
ofstream outputFile;
```

```
outputFile.open("c:\\custom\\invtry.dat");
```

因为反斜线是一种特殊的字符，在字符串中，两个反斜线表示一个反斜线。

- 在open函数中，可以采用**字符数组或string对象**作为参数；

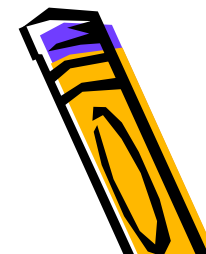
```
ifstream inputFile;  char fileName[20];
```

```
strcpy(fileName,"Myfile.dat");
```

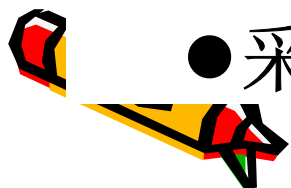
```
inputFile.open(fileName);
```



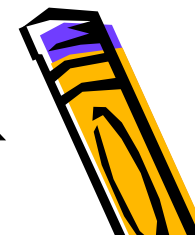
# 打开成功与否的判断



- 打开文件操作并不能保证总是正确的，如文件不存在、磁盘损坏等原因可能造成打开文件失败。
- 如果打开文件失败后，程序还继续执行文件的读/写操作，将会产生严重错误。在这种情况下，应使用异常处理以提高程序的可靠性。
  - 如果使用构造函数或`open()`打开文件失败，流状态标志位中的`failbit`、`badbit`或`hardbit`将被置为1，并且在`ios`类中重载的运算符“!”将返回非0值。通常可以利用这一点检测文件打开操作是否成功，如果不成功则作特殊处理。
  - 采用`fail`函数来判断



# 打开成功与否的判断方法



- 第一种方法:通过 “! 操作符” 测试open函数是否失败

```
fstream dataFile;
```

```
dataFile.open(“custom.dat”,ios::in)
```

```
if (!datafile ) {
```

```
    cout<<“文件打开失败\n”; exit(0);
```

```
}
```

- 第二种方法:采用fail函数来测试文件打开是否失败

```
dataFile.open(“custom.dat”,ios::in);
```

```
if (dataFile.fail() ) { //打开失败时，成员函数返回true
```

```
    cout<<“文件打开失败\n” ;
```

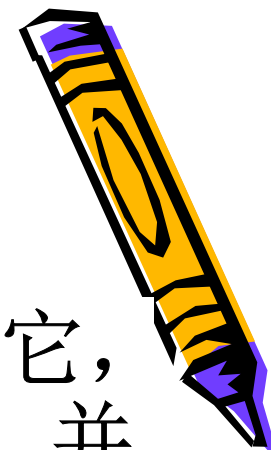
```
    exit(0);
```

```
}
```



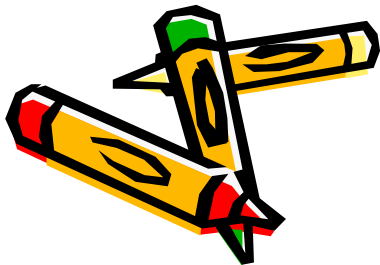
# 文件的关闭（1）

- 当打开的文件操作结束后，就需要关闭它，使文件流与对应的物理文件**断开联系**，并能够保证最后输出到文件缓冲区中的内容，无论是否已满，都将立即**写入到对应的物理文件中**。
- 每个文件流类中都提供有一个关闭文件的成员函数**close（）**，
- 文件流对应的文件被关闭后，还可以利用该文件流调用**open**成员函数打开其他的文件。



## 文件的关闭（2）

- 关闭任何一个流对象所对应的文件，就是用这个流对象调用**close**（）成员函数即可。
- 如要关闭**fout**流所对应的**a:\xxk.dat**文件，
  - 则关闭语句为：  
**fout.close();**



# 文件中信息的读写

文件读写有两种方法：

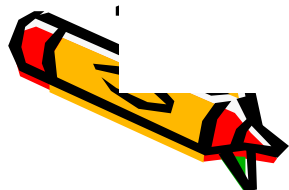
- 使用流运算符直接读写

文件的读/写操作可以直接使用流的插入运算符“<<”和提取运算符“>>”，这些运算符将完成文件的字符转换工作。

- 使用流成员函数

常用的输出流成员函数为：**put**函数、**write**函数

常用的输入流成员函数如下：**get**函数、**getline**函数、**read**函数





例：有一个整型数组，含**10**个元素，从键盘输入**10**个整数给数组，将此数组送到磁盘文件中存放。

```
#include <fstream>
```

```
using namespace std;
```

```
int main( )
```

```
{int a[10]; //内存是一个重要的沟通的渠道，键盘和文件之间的沟通不可逾越
```

```
ofstream outfile("f1.dat",ios::out); //定义文件流对象，打开磁盘文件"f1.dat"
```

```
if(!outfile) //如果打开失败，outfile返回0值
```

```
{cerr<<"open error!"<<endl;
```

```
exit(1);
```

```
}
```

```
cout<<"enter 10 integer numbers:"<<endl;
```

```
outfile<<"The data in the file :\n"; //在文件中存字符串信息
```

```
for(int i=0;i<10;i++)
```

```
{cin>>a[i];
```

```
    outfile<<a[i]<<" "; //向磁盘文件"f1.dat"输出数据，等效于显示器
```

```
outfile.close(); //关闭磁盘文件"f1.dat"
```

```
return 0;
```

```
}
```

例：从上程序建立的数据文件**f1.dat**中读入**10**个整数放在数组中，找出并输出**10**个数中的最大者和它在数组中的序号。

```
#include <fstream>
```

```
int main( )
```

```
{int a[10],max,i,order;
```

```
    ifstream infile("f1.dat",ios::in|ios::nocreate);
```

```
//定义输入文件流对象，以输入方式打开磁盘文件f1.dat
```

```
    if(!infile)
```

```
    {cerr<<"open error!"<<endl;
```

```
        exit(1);
```

```
    }
```



```
    for(i=0;i<10;i++)
```

```
        {infile>>a[i]; //从磁盘文件读入整数，通过空白字符区分数据，数据顺序存  
                //放在a数组中，等效于键盘
```



```
        cout<<a[i]<<" "; //在显示器上顺序显示10个数
```

```
    cout<<endl;
```

```
    max=a[0]; //读取到内存中，来处理数据
```



```
order=0;
for(i=1;i<10;i++)
    if(a[i]>max)
        {max=a[i];           //将当前最大值放在max中
          order=i;           //将当前最大值的元素序号放在order中
        }
cout<<"max="<<max<<endl<<"order="<<order<<endl;
infile.close();
return 0;
}
```



运行情况如下:

1 3 5 2 4 6 10 8 7 9(在磁盘文件中存放的10个数)



max=10 (最大值为10)

order=6 (最大值是数组中序号为6的元素)

备注: 程序采用**顺序的方式**从文件中读取数据。当打开文件时, 流对象的“光标”位于文件第一个字节的位置。因此, 首次读操作是在第一个字节读取数据, 随着数据的读取, 光标会自动向后移动。

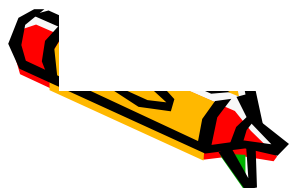


# 文本文件读写的注意事项

- 文本文件中每个字节单元的内容为字符的 **ASCII码**
  - 处理文本文件时将自动作一些**字符转换**，如：输出换行字符时将转换为回车与换行两个字符存入文本文件；读入时也会将回车与换行两个字符合并为一个换行字符，这样**内存中的字符与写入文件中的字符之间就不再是一一对应关系**。
  - 文本文件的结束以ASCII码的控制字符**0x1A**表示。
- 
- 

# 二进制文件的读写

- 读写二进制文件的字符不作任何转换，读写的字符与文件之间是完全一致的。
- 对二进制文件的读写常采用方法：
  - **read**（）：一次读取一个记录（定长块）；
  - **write**（）：适合一次写一个数据块（如数组、结构体或对象等）。



# 文件的随机读写

在磁盘文件中有一个文件指针（光标），用来指明当前应进行读写的位置。对于二进制文件，允许对指针进行控制，使它按用户的意图移动到所需的位置，以便在该位置上进行读写。文件流提供一些有关文件指针的成员函数。

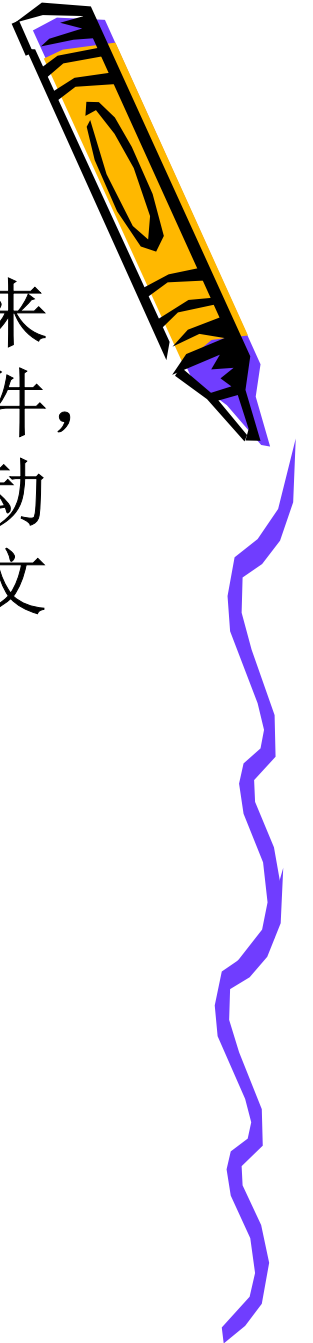
## ● 文件的随机读写

（1）输出流随机访问函数。

输出流随机访问函数有**seekp**和**tellp**。

（2）输入流随机访问函数。

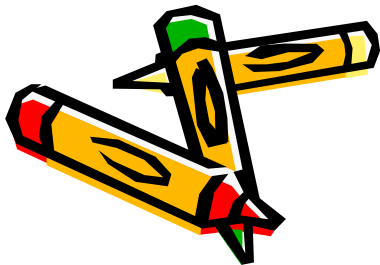
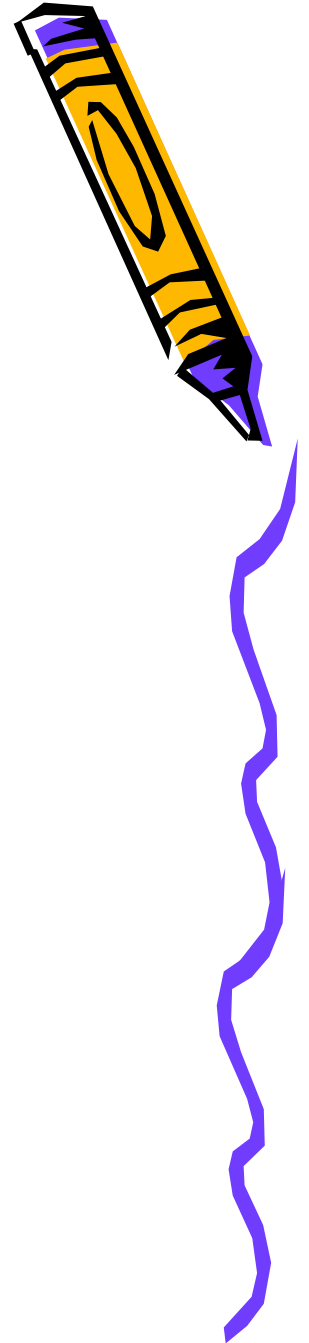
输入流随机访问函数有**seekg**和**tellg**。



# 文件的光标的参照位置

**seekg**和**seekp**函数可以根据参照位置，来进行偏移。

- **ios::beg** 文件
- **ios::cur** 当前位位置
- **ios::end** 文件尾



# 检测文件的结束

- 对无法准确知道文件中存储多少数据时，可以通过成员函数**eof( )**检测“光标”是否已经达到文件的尾部。
- 若“光标”已经到文件的尾部，再次进行读取，那么**eof**函数将返回一个非零值。

```
if (inFile.eof() )  
    inFile.close( );
```

或

```
while (! inFile.eof( ) )  
    inFile>>var;
```



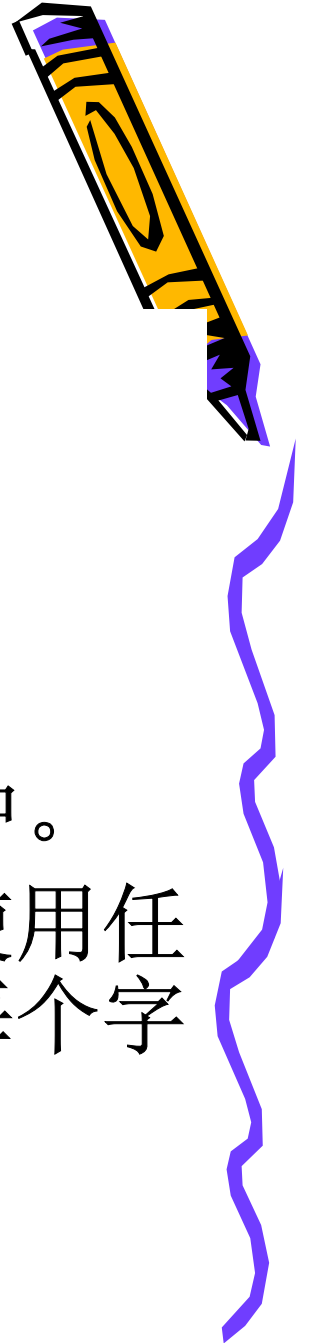


- 面向字符串变量的**I/O**



# 字符串流

- 字符串流类包括三类：
  - 输入字符串流类**istream**
  - 输出字符串流类**ostream**
  - 输入输出字符串流类**stringstream**三种。
- 它们都被定义在系统头文件**sstream**中。
- 只要在程序中带有该头文件，就可以使用任何一种字符串流类定义**字符串流对象**。每个字符串流对象简称为字符串流。



# 从构造函数看对象的初始化

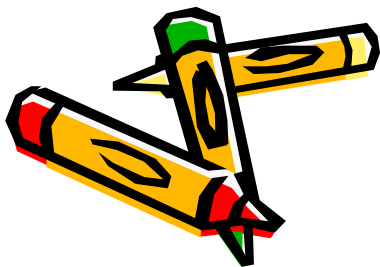
- 三种字符串流类的构造函数声明格式分别如下：

**istream(const char\* buffer);**

**ostream(char\* buffer, int n);**

**stringstream(char\* buffer, int n, int mode);**

- 对字符串流的操作方法通常与对字符文件流的操作方法相同。





- 下面给出定义相应字符串流的例子。

(1) **ostream** sout(a1,50);

(2) **istream** sin(a2);

(3) **stringstream** sio(a3,sizeof(a3),ios::in | ios::out);

- 第(1)条语句定义了一个输出字符串流**sout**，使用的字符数组为**a1**，大小为**50**个字节，以后对**sout**的输出都将被写入到字符数组**a1**中。
- 第(2)条语句定义了一个输入字符串流**sin**，使用的字符数组为**a2**，以后从**sin**中读取的输入数据都将来自字符数组**a2**中。
- 第(3)条语句定义了一个输入输出字符串流**sio**，使用的字符数组为**a3**，大小为**a3**数组的长度，打开方式规定为既能够用于输入又能够用于输出，当然进行输入的数据来自数组**a3**，进行输出的数据写入数组**a3**。

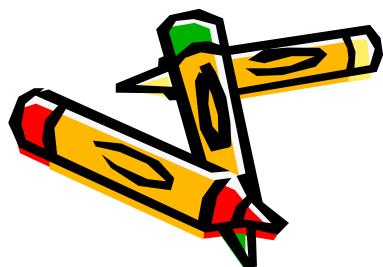
例：从一个字符串流中输入用逗号分开的每一个整数并显示出来。

```
• #include<strstream>
• using namespace std;
• void main() {
•     char a[]="38,46 ,55, 78,42  ,77,60,93@";
•     cout<<a<<endl; //输出a字符串
•     istream sin(a);
•     //定义一个输入字符串流sin，使用的字符数组为a
•     char ch=' '; int x;
•     while(ch!='@') { //使用'@'字符作为字符串流结束标志
•         sin>>ws>>x>>ws;
•         //从流中读入一个整数，并使用操作符ws读取(去除)
•         //一个整数前后的空白字符
•         cout<<x<<' ';
•         //输出x的值并后跟一个空格
•         sin.get(ch);
•         //从sin流中读入一个字符，实际读取的是','或'@'字符
•     }
•     cout<<endl;
• }
```

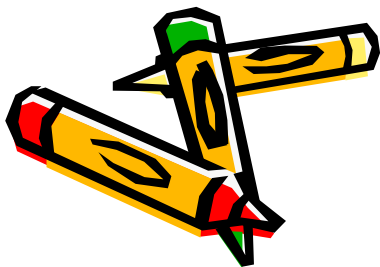
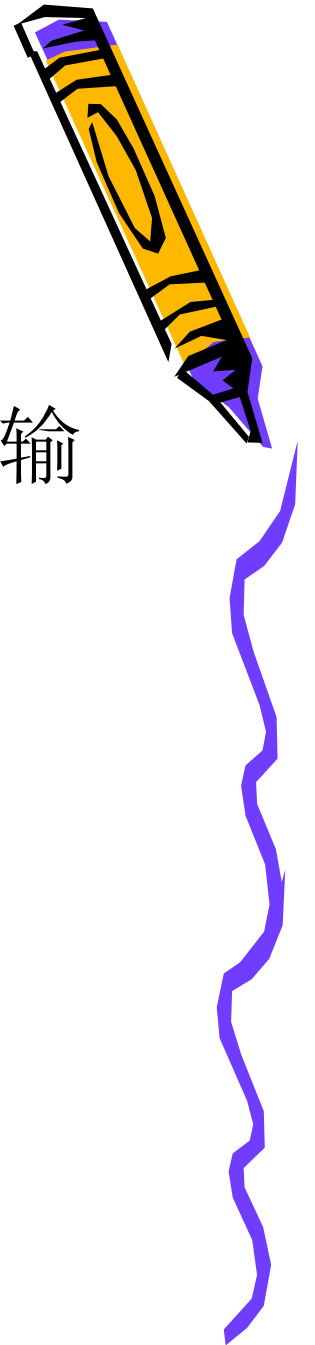
该程序运行结果如下：

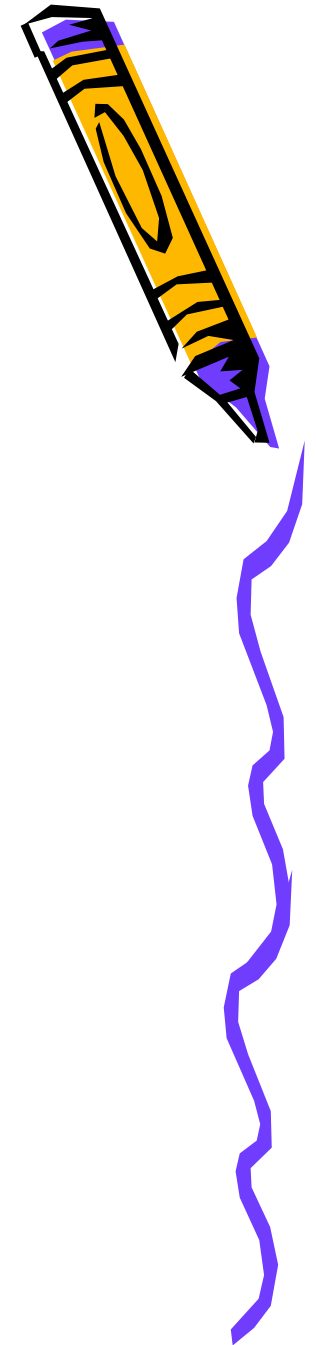
```
• 38,46 ,55, 78,42  ,77,60,93@
• 38 46 55 78 42 77 60 93
```

- 补充阅读



- 下面以标准输出流对象**cout**为例说明输出的格式化控制。





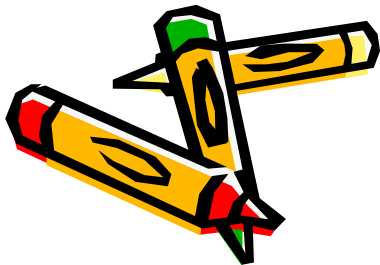
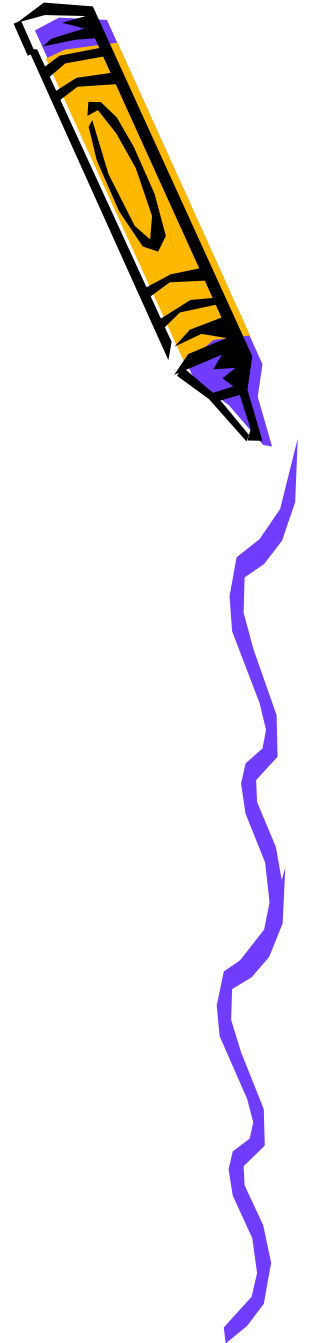
• 程序1:

```
• #include<iostream.h>
• void main()
• {
•     int x=30, y=300, z=1024;
•     cout<<x<<' '<<y<<' '<<z<<endl; //按十进制输出
•     cout.setf(ios::oct); //设置为八进制输出
•     cout<<x<<' '<<y<<' '<<z<<endl; //按八进制输出
•     cout.unsetf(ios::oct);
•     //取消八进制输出设置, 恢复按十进制输出
•     cout.setf(ios::hex); //设置为十六进制输出
•     cout<<x<<' '<<y<<' '<<z<<endl; //按十六进制输出
•     cout.setf(ios::showbase | ios::uppercase);
•     //设置基指示符输出和数值中的字母大写输出
•     cout<<x<<' '<<y<<' '<<z<<endl;
•     cout.unsetf(ios::showbase | ios::uppercase);
•     //取消基指示符输出和数值中的字母大写输出
•     cout<<x<<' '<<y<<' '<<z<<endl;
•     cout.unsetf(ios::hex);
•     //取消十六进制输出设置, 恢复按十进制输出
•     cout<<x<<' '<<y<<' '<<z<<endl;
```





- 此程序的运行结果如下:
- 30 300 1024
- 36 454 2000
- 1e 12c 400
- 0X1E 0X12C 0X400
- 1e 12c 400
- 30 300 1024



• 程序2:

• `#include<iostream.h>`

• `void main()`

• `{`

• `int x=468;`

• `double y=-3.425648;`

• `cout<<"x=";`

• `cout.width(10); //设置输出下一个数据的域宽为10`

• `cout<<x; //按缺省的右对齐输出，剩余位置填充空格字符`

• `cout<<"y=";`

• `cout.width(10); //设置输出下一个数据的域宽为10`

• `cout<<y<<endl;`

• `cout.setf(ios::left); //设置按左对齐输出`

• `cout<<"x=";`

• `cout.width(10);`

• `cout<<x;`

• `cout<<"y=";`

• `cout.width(10);`





- `cout<<y<<endl;`
- `cout.fill('*'); //设置填充字符为'*'`
- `cout.precision(3); //设置浮点数输出精度为3`
- `cout.setf(ios::showpos); //设置正数的正号输出`
- `cout<<"x=";`
- `cout.width(10);`
- `cout<<x;`
- `cout<<"y=";`
- `cout.width(10);`
- `cout<<y<<endl;`
- `}`

此程序运行结果如下:

`x= 468y= -3.42565`

`x=468 y=-3.42565`

`x=+468*****y=-3.43*****`



程序3:

```
#include<iostream.h>
void main()
{
    float x=25, y=-4.762;
    cout<<x<<' '<<y<<endl;
    cout.setf(ios::showpoint); //强制显示小数点和无效(
    cout<<x<<' '<<y<<endl;
    cout.unsetf(ios::showpoint); //恢复缺省输出
    cout.setf(ios::scientific); //设置按科学表示法输出
    cout<<x<<' '<<y<<endl;
    cout.setf(ios::fixed); //设置按定点表示法输出
    cout<<x<<' '<<y<<endl;
```



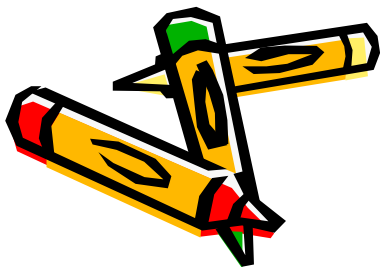
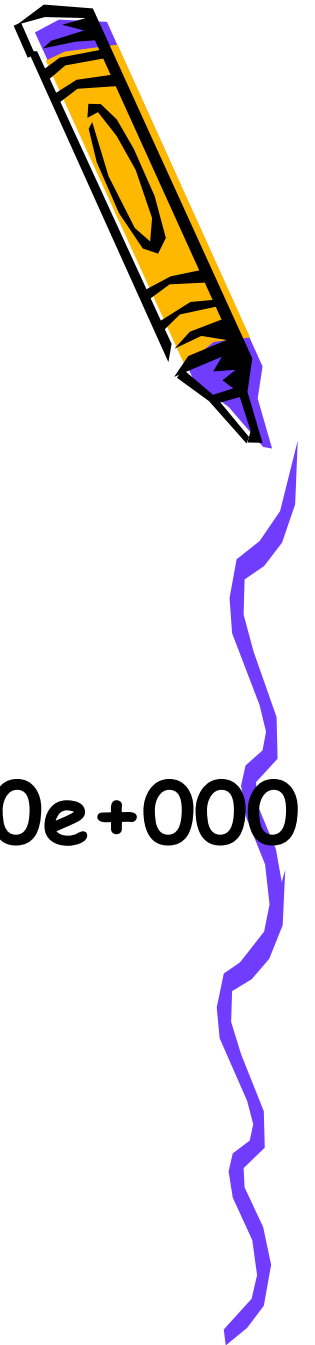
• 程序运行结果如下：

• **25 -4.762**

• **25.0000 -4.76200**

• **2.500000e+001 -4.762000e+000**

• **25 -4.762**



- 下面以标准输出流对象**cout**为例，说明使用操作符进行的输出格式化控制。

- 





• 程序4:

• `#include<iostream.h>`

• `//因iomanip.h中包含有iostream.h, 所以该命令可省略`

• `#include<iomanip.h>`

• `void main()`

• `{`

• `int x=30, y=300, z=1024;`

• `cout<<x<<' '<<y<<' '<<z<<endl; //按十进制输出`

• `cout<<oct<<x<<' '<<y<<' '<<z<<endl; //按八进制输出`

• `cout<<hex<<x<<' '<<y<<' '<<z<<endl; //按十六进制输出`

• `cout<<setiosflags(ios::showbase | ios::uppercase);`

• `//设置基指示符和数值中的字母大写输出`

• `cout<<x<<' '<<y<<' '<<z<<endl; //仍按十六进制输出`

• `cout<<resetiosflags(ios::showbase | ios::uppercase);`

• `//取消基指示符和数值中的字母大写输出`

• `cout<<x<<' '<<y<<' '<<z<<endl; //仍按十六进制输出`

• `cout<<dec<<x<<' '<<y<<' '<<z<<endl; //按十进制输出`

• 程序的功能和运行结果都与程序1完全相同。





程序5:

```
#include<iostream.h>
#include<iomanip.h>
void main()
{
    int x=468;
    double y=-3.425648;
    cout<<"x="<<setw(10)<<x;
    cout<<"y="<<setw(10)<<y<<endl;
    cout<<setiosflags ios::left; //设置按左对齐输出
    cout<<"x="<<setw(10)<<x;
    cout<<"y="<<setw(10)<<y<<endl;
    cout<<setfill('*'); //设置填充字符为'*'
    cout<<setprecision(3); //设置浮点数输出精度为3
    cout<<setiosflags ios::showpos; //设置正数的正号输出
    cout<<"x="<<setw(10)<<x;
    cout<<"y="<<setw(10)<<y<<endl;
    cout<<resetiosflags ios::left | ios::showpos);
    cout<<setfill(' ');
}
```



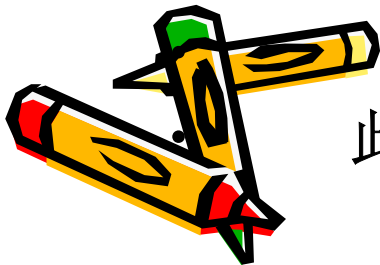
该程序的功能和运行结果完全与程序2相同。



- 程序6:

- `#include <iomanip.h>`
- `void main()`
- `{`
- `float x=25, y=-4.762;`
- `cout<<x<<' '<<y<<endl;`
- `cout<<setiosflags(ios::showpoint);`
- `cout<<x<<' '<<y<<endl;`
- `cout<<resetiosflags(ios::showpoint);`
- `cout<<setiosflags(ios::scientific);`
- `cout<<x<<' '<<y<<endl;`
- `cout<<setiosflags(ios::fixed);`
- `cout<<x<<' '<<y<<endl;`
- `}`

此程序的功能和运行结果也完全与程序3相同。



- 下面给出定义文件流对象和打开文件的一些例子：



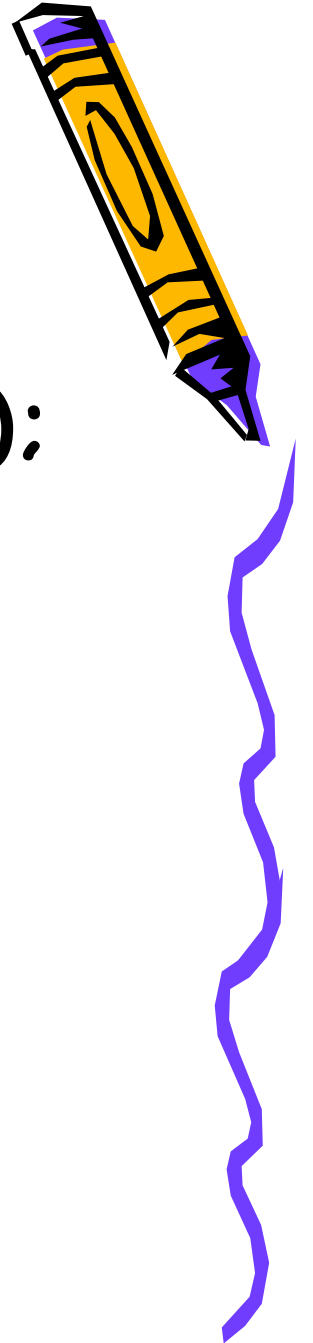
# 间接打开方式



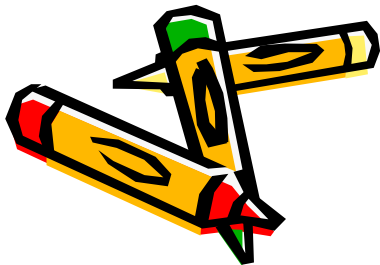
- (1) ofstream fout;  
• fout.open("a:\\\\xxk.dat"); //字符串中的双反斜线表示一个反斜线
- (2) ifstream fin;  
• fin.open("a:\\\\wr.dat", ios::in | ios::nocreate);
- (3) ofstream ofs;  
• ofs.open("a:\\\\xxk.dat", ios::app);
- (4) fstream fio;  
• fio.open("a:\\\\abc.ran", ios::in | ios::out | ios::binary);



# 直接打开方式



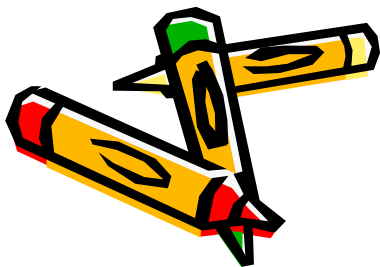
- 依次与下面的文件流定义语句功能相同。
- (1) `ofstream fout("a:\\\\xxk.dat");`
- (2) `ifstream fin("a:\\\\wr.dat",  
ios::in | ios::nocreate);`
- (3) `ofstream ofs("a:\\\\xxk.dat",  
ios::app);`
- (4) `fstream ofs(a:\\\\abc.ran",  
ios::in | ios::out | ios::binary);`





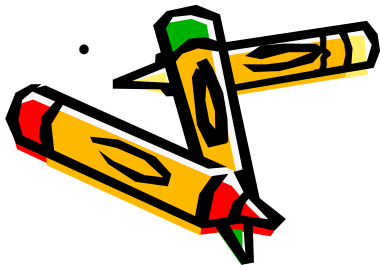
• 例1. 向a盘上的wr1.dat文件输出0~20之间的整数，含0和20在内

```
• #include<iostream.h>
• #include<stdlib.h>
• #include<fstream.h>
• void main(void)
• {
•     ofstream f1("a:wr1.dat");
•     //定义输出文件流，并打开相应文件，若打开失败则f1带回0值
•     if (!f1) { //当f1打开失败时进行错误处理
•         cerr<<"a:wr1.dat file not open!"<<endl;
•         exit(1);
•     }
•     for(int i=0;i<21;i++)
•         f1<<i<<" "; //向f1文件流输出i值
•     f1.close(); //关闭f1所对应的文件
• }
```



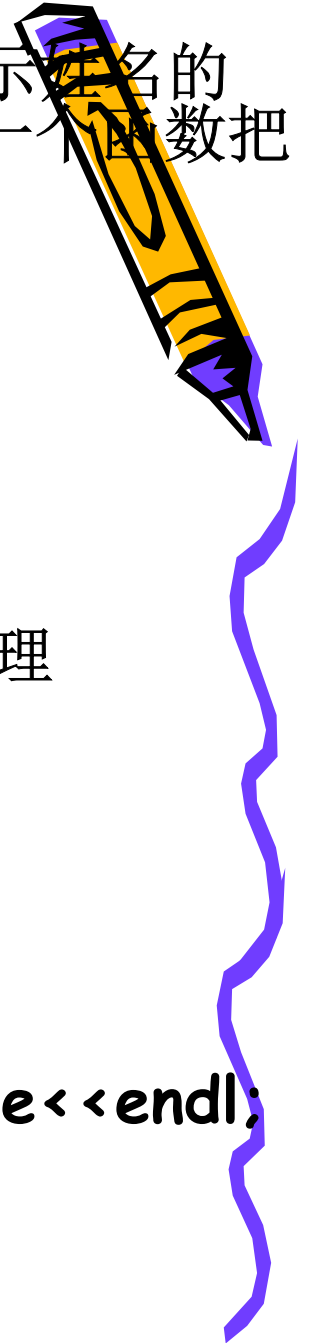
- 例2. 把从键盘上输入的若干行文本字符原原本本地存入到a盘上wr2.dat文件中，直到从键盘上按下Ctrl+z组合键为止。此组合键代表文件结束符EOF。

```
#include<iostream.h>
#include<stdlib.h>
#include<fstream.h>
void main(void)
{
    char ch;
    ofstream f2("a:wr2.dat");
    if (!f2) { //当f1打开失败时进行错误处理
        cerr<<"File of a:wr2.dat not open!"<<endl;
        exit(1);
    }
    ch=cin.get(); //从cin流中提取一个字符到ch中
    while(ch!=EOF) {
        f2.put(ch); //把ch字符写入到f2流中，此语句也
        //可用f2<<ch代替
        ch=cin.get(); //从cin流中提取下一个字符到ch中
    }
    f2.close(); //关闭f2所对应的文件
```



- 例3. 假定一个结构数组a中的元素类型pupil包含有表示姓名的字符指针域name和表示成绩的整数域grade，试编写一个函数把该数组中的n个元素输出到字符文件“a:wr3.dat”中。

```
• #include<stdlib.h>
• #include<fstream.h>
• void ArrayOut(pupil a[], int n)
• {
•     ofstream f3("a:wr3.dat");
•     if (!f3) { //当f3打开失败时进行错误处理
•         cerr<<"File of a:wr3.dat not
open!"<<endl;
•         exit(1);
•     }
•     for(int i=0; i<n; i++)
•         f3<<a[i].name<<endl<<a[i].grade<<endl;
•     f3.close();
• }
```



- 若已经为输出 **pupil** 类型的数据定义了如插入操作符重载函数：

- ```
ostream&  
operator<<(ostream& ostr, pupil& x)  
{  
  
    ostr<<x.name<<endl<<x.grade<<endl;  
  
    return ostr;  
}
```

- 则可将上述函数中 **for** 循环体语句修改为 “**f3<<a[i];**”。







例1. 从一个字符串流中输入用逗号分开的每一个整数并显示出来。

```
#include<strstream.h>
```

```
void main()
```

```
{
```

```
    char a[]="38,46 ,55, 78,42  ,77,60,93@";
```

```
    cout<<a<<endl; //输出a字符串
```

```
    istream sin(a);
```

```
    //定义一个输入字符串流sin，使用的字符数组为a
```

```
    char ch=' ';
```

```
    int x;
```

```
    while(ch!='@') { //使用'@'字符作为字符串流结束标志
```

```
        sin>>ws>>x>>ws;
```

```
        //从流中读入一个整数，并使用操作符ws读取
```

```
        //一个整数前后的空白字符
```

```
        cout<<x<<' ';
```

```
        //输出x的值并后跟一个空格
```

```
        sin.get(ch);
```

```
        //从sin流中读入一个字符，实际读取的是' ,'或' '@'字符
```

```
    }
```

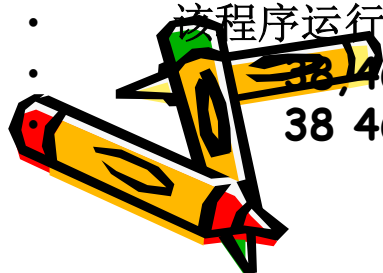
```
    cout<<endl;
```

```
}
```

该程序运行结果如下：

```
38,46 ,55, 78,42  ,77,60,93@
```

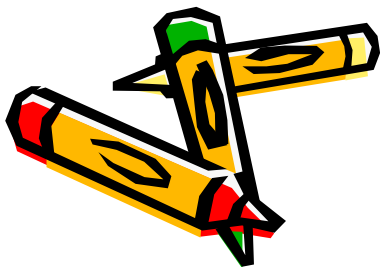
```
38 46 55 78 42 77 60 93
```



# 检测流操作的错误

在I/O流的操作过程中可能出现各种错误，每一个流都有一个**状态标志字**，以指示是否发生了错误以及出现了哪种类型的错误，这种处理技术与格式控制标志字是相同的。

**ios**类定义了以下枚举类型：



```
enum io_state
```

```
{
```

```
    goodbit = 0x00,
```

```
    eofbit  = 0x01,
```

```
    failbit = 0x02,  
    用
```

```
    badbit  = 0x04,  
    可用
```

```
    hardfail = 0x80
```

```
};
```

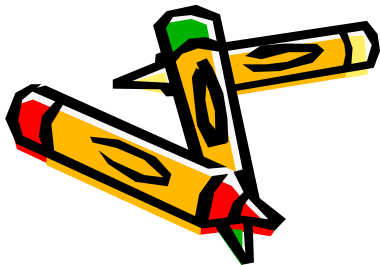
```
//不设置任何位，一切正常
```

```
//输入流已经结束，无字符可读入
```

```
//上次读/写操作失败，但流仍可使
```

```
//试图作无效的读/写操作，流不再
```

```
//不可恢复的严重错误
```



- 对应于这个标志字各状态位，**ios**类还提供了以下成员函数来检测或设置流的状态：

**int rdstate();** //返回流的当前状态标志字

**int eof();** //返回非**0**值表示到达文件尾

**int fail();** //返回非**0**值表示操作失败

**int bad();** //返回非**0**值表示出现错误

**int good();** //返回非**0**值表示流操作正常

**int clear(int flag=0);** //将流的状态设置为**flag**

- 为提高程序的可靠性，应在程序中检测**I/O**流的操作是否正常。
- 当检测到流操作出现错误时，可以通过异常处理来解决问题。

