IO

传统输入输出的缺点：需要确定数据类型

```
#include <cstdio>
#include <iostream>
using namespace std;
float a = 5;
printf("%d", a);
//%d负责解释处理数据输出;

Class aa{
public:
   int a;
   int b;
};

aa a;
a.a = 1;
a.b = 2;
printf("%? ", a);

//不能直接输出类的对象内容。
```

输入输出对象

```
//类名称为：iostream，iostream.h是该类的头文件

ostream &operator<<(ostream out,char * p);
ostream &operator<<(ostream out,int p);
ostream &operator<<(ostream out,char  p);
ostream &operator<<(ostream out,float p);


//cout、cin为iostream类的全局对象，不可复制
//默认输入输出设备、其它输入输出设备

ostream cout(stdout);   //  标准输出      默认屏幕
istream cin(stdin);     //    标准输入      默认键盘
ostream cerr(stderr);   //  标准出错      默认屏幕
ostream clog(stdprn);  //   标准打印     默认打印机
```

好的，让我给你讲解一下。

首先，前三行代码你给出的注释是正确的，它们分别代表了C++中的标准输入输出流：

1. `ostream cout(stdout);`：这行代码实际上在C++标准库中已经定义，它定义了一个名为`cout`的输出流对象，该对象与标准输出`stdout`相关联。通常，`stdout`指的是终端或屏幕输出。因此，当你向`cout`写入数据时，数据会显示在屏幕上。
2. `istream cin(stdin);`：这行代码也在C++标准库中预先定义，它定义了一个名为`cin`的输入流对象，与标准输入`stdin`相关联。通常，`stdin`指的是从键盘接收输入。所以，当你从`cin`读取数据时，你实际上是从键盘输入数据。
3. `ostream cerr(stderr);`：这同样是在C++标准库中预先定义的一个输出流对象，名为`cerr`，与标准错误输出`stderr`相关联。通常，`stderr`也是输出到屏幕上，但它用于输出错误信息，这样即使标准输出被重定向到其他地方（如文件），错误信息仍然会显示在屏幕上。

然而，第四行代码`ostream clog(stdprn);`是有问题的。在C++标准库中，并没有定义`stdprn`这个标识符，它也不是一个标准的设备文件描述符。C++标准库中定义的与打印相关的流是`clog`，它与`cerr`类似，通常也用于输出到屏幕（尽管它的缓冲行为可能与`cerr`有所不同）。

如果你想将内容发送到打印机，C++标准库并没有直接提供这样的功能。在Windows系统中，你可能需要使用特定的Windows API函数来发送打印任务；在类Unix系统中，你可能需要将输出重定向到打印命令（如`lpr`）或者使用CUPS（Common Unix Printing System）的API。

总结一下：

- `cout`用于向屏幕输出。
- `cin`用于从键盘输入。
- `cerr`用于向屏幕输出错误信息。
- C++标准库中没有直接支持打印到打印机的流，需要使用操作系统提供的特定API或命令。

希望这个解释能帮助你更好地理解这些概念！

在C++中，`clog` 是一个与标准错误流 `cerr` 类似的预定义输出流对象，但它通常提供带缓冲的输出。这意味着写入 `clog` 的数据可能不会立即显示，而是会被暂存到缓冲区中，直到缓冲区满或者程序显式地刷新输出流。

你可以像使用 `cout` 一样使用 `clog`，但是 `clog` 通常用于记录程序运行时的诊断信息或日志。这些信息可能不像 `cout` 输出的信息那样对用户友好，但对于程序调试和日志记录来说是非常有用的。

下面是一个使用 `clog` 的简单示例：

```
#include <iostream>

int main() {
    std::clog << "This is a diagnostic message written to clog." << std::endl;

    // 刷新clog以确保消息被立即输出
    std::clog.flush();

    return 0;
}
```

在这个例子中，我们向 `clog` 写入了一条诊断信息，并使用 `std::clog.flush()` 来确保消息被立即输出到其关联的设备（通常是标准错误输出，stderr）。如果你不调用 `flush()`，那么消息可能会在程序结束前或缓冲区满时才被输出。

值得注意的是，`clog` 和 `cerr` 都通常与标准错误输出（stderr）相关联，因此它们的输出通常不会与 `cout` 的输出混合在一起，这有助于在调试时将日志信息与常规输出分开。

如果你想要将日志信息记录到文件而不是控制台，你可以通过重定向stderr流或者使用一个自定义的日志系统来实现。在Unix-like系统中，你可以通过命令行重定向stderr到文件，例如：

```
bash复制代码

./your_program 2> logfile.txt
```

这条命令会将stderr重定向到名为`logfile.txt`的文件中。在Windows命令提示符下，你可以使用类似的重定向语法。

iomanip

```
#include <iostream>
#include<iomanip>
using  namespace std;
void main()
{
    double a = 0.12345;
    cout.precision(2);
    cout << a << endl;
    cout << a << setprecision(3)<<' ' << a << endl;//setprecison(int)是<iomanip>里的函数
    cout << a << endl;
}
```

0.12 0.12 0.123 0.123

```
#include <iostream>
#include<iomanip>
using  namespace std;
void main()
{
    double a = 0.12345;
    cout.precision(2);
    cout << a << endl;
    cout << a << setprecision(3)<<' ' << a << endl;//这个setprecision起作用
    setprecision(4);//这个不起作用
    cout <<a << endl;
}
```

C++IO机制的优点

```
#include <iostream>
using  namespace std;
void main()
{
    char a[20];
    cin >> a;
    cout << endl << a;

    cin.getline(a, 11, 'a');
    cout << endl << a;
}
```

1. 定义一个字符数组a，长度为20。
2. 使用cin >> a;从标准输入读取一个字符串到数组a中，直到遇到空格、制表符或换行符为止。
3. 输出一个换行符，然后输出数组a的内容。
4. 使用cin.getline(a, 11, 'a');尝试从标准输入读取数据。但这里的getline用法与常见的稍有不同。

关于cin.getline(a, 11, 'a');:

- 通常，std::istream::getline函数的原型是istream& getline (char* s, streamsize n, char delim = '\n');，其中delim参数表示定界符，默认为换行符。
- 但在你的代码中，delim被设置为'a'，这意味着getline会尝试读取输入，直到遇到字符'a'为止，或者读取了10个字符（因为第二个参数是11，但需要留一个位置给字符串的终止字符\0）。
- 这意味着，如果用户输入的是"Hello Worlda"，那么只有"Hello Worl"会被读取到数组a中，并在后面添加一个\0作为字符串的终止符。
- 最后，再次输出数组a的内容。

error

error 23bdfea

23bdfe

---

error

error 12345678910

12345678

解释：cin>>a后回车被保留在缓存区没有被读取，输出error后又按了一个回车，所以cin.getline先读取两回车再读取8个字符，留下一个空间补'\0'。

文件流类

例子1

```
//ofstream   //文件输出类
//ifstream    //文件输入类
//在fstream里面定义   fstream是多继承子类

//没有cout和cin这样的标准全局对象。

Ofstream::ofstream(char * pFilename,int mode=ios::out,int prot=filebuf::openprot);
```

首先，我们来谈谈文件流。在C++中，文件流是用于读写文件的一种机制。C++标准库提供了三个主要的文件流类：ofstream（输出文件流）、ifstream（输入文件流）和fstream（输入/输出文件流）。这些类都定义在<fstream>头文件中，而不是fstream.h（这是旧式C++头文件的命名方式，现代C++使用无后缀的头文件命名方式）。

1. **ofstream（文件输出类）**：

   ofstream用于向文件中写入数据。当你创建一个ofstream对象时，你可以指定一个文件名和打开模式。例如：

   ```
   ofstream outfile("example.txt");
   ```

   这行代码会创建一个名为example.txt的文件，并准备向其中写入数据。如果文件已经存在，它将被覆盖。如果你不希望覆盖现有文件，而是想在其后追加数据，你可以使用ios::app模式：

   ```
   ofstream outfile("example.txt", ios::app);
   ```

   使用ofstream对象，你可以像使用cout一样使用插入运算符（<<）向文件中写入数据。

2. **ifstream（文件输入类）**： ifstream用于从文件中读取数据。创建一个ifstream对象时，你也需要指定一个文件名。例如：

   ```
   ifstream infile("example.txt");
   ```

   这行代码会打开一个名为example.txt的文件，并准备从中读取数据。使用ifstream对象，你可以使用提取运算符（>>）从文件中读取数据，类似于使用cin。

3. **fstream（文件输入/输出类）**： fstream是一个多继承子类，它继承了ifstream和ofstream的功能，允许你对同一个文件进行读写操作。当你需要同时读取和写入同一个文件时，这是非常有用的。例如：

   ```
   fstream file("example.txt", ios::in | ios::out);
   ```

   这行代码会打开一个名为example.txt的文件，允许你进行读写操作。

关于你提到的构造函数参数：

- char * pFilename：这是一个指向C风格字符串的指针，指定了要打开文件的名称。
- int mode：这是一个可选参数，指定了文件打开的模式（如ios::out表示写模式，ios::in表示读模式，ios::app表示追加模式等）。你可以使用按位或运算符（|）来组合多个模式。
- int prot：这是一个保护模式参数，但在现代C++中通常不使用，因此你可以忽略它。在大多数情况下，只需提供文件名和模式即可。

请注意，在使用文件流之前，最好检查文件是否成功打开。你可以通过调用文件流对象的is_open()成员函数来完成这个检查。例如：

```
ifstream infile("example.txt");
if (!infile.is_open()) {
    cerr << "Unable to open file" << endl;
    exit(1); // Exit the program with an error code
}
```

希望这些信息能帮助你更好地理解C++中的文件流类！

例子2

```
#include<fstream>
#include <iostream>
using namespace std;
void fn() {
    ofstream myf("c:\\myname.txt");    //ios::out|ios::trunc
    myf << "学院\n"<<"of 软件\n";
}
void fi() {
    char ch[20];
    ifstream  myf("c:\\myname.txt", ios::nocreate, filebuf::sh_none);
    if (myf.fail())  cerr << "文件打开错误";
    myf >> ch;
    cout << ch;
    myf >> ch;
    cout << ch;
}
void main() {
    fn();
    fi();
}
```

首先，我们来看一下整体结构和功能。这段代码包含两个主要的函数`fn()`和`fi()`，以及程序的主入口`main()`。`fn()`函数用于向文件`c:\\myname.txt`写入内容，而`fi()`函数用于从同一个文件读取内容并打印到控制台。`main()`函数则依次调用这两个函数。

现在，我们逐一分析每个部分：

包含头文件

```
#include<fstream>
#include <iostream>
```

这两行代码包含了必要的头文件。`<fstream>`用于文件操作，`<iostream>`用于输入输出流操作。

使用命名空间

```
using namespace std;
```

这行代码表示我们将使用标准命名空间`std`，这样我们就不必在每次使用标准库中的类或函数时都加上`std::`前缀。

fn() 函数

```
void fn() {
    ofstream myf("c:\\myname.txt");    //ios::out|ios::trunc
    myf << "学院\n"<<"of 软件\n";
}
```

这个函数创建了一个输出文件流对象`myf`，用于向`c:\\myname.txt`文件写入数据。注意，虽然注释中提到了`ios::out|ios::trunc`，但实际上在创建`ofstream`对象时并没有显式指定这些模式。对于`ofstream`来说，`ios::out`是默认模式，而`ios::trunc`表示如果文件已存在，则其内容会被截断（即清空）。在这个函数中，字符串`"学院\n"`和`"of 软件\n"`被写入文件。

fi() 函数

```
void fi() {
    char ch[20];
    ifstream  myf("c:\\myname.txt", ios::nocreate, filebuf::sh_none);
    if (myf.fail())  cerr << "文件打开错误";
    myf >> ch;
    cout << ch;
    myf >> ch;
    cout << ch;
}
```

这个函数尝试从`c:\\myname.txt`文件中读取数据。但是，这里有几个问题需要注意：

1. `ios::nocreate`和`filebuf::sh_none`不是C++标准库中的有效标志。这些可能是特定编译器或老版本C++中的扩展，但在标准C++中并不被识别。在现代C++中，如果文件不存在，`ifstream`的构造函数会自动失败，无需额外标志。
2. 函数使用了一个字符数组`ch[20]`来存储从文件中读取的数据。但是，由于使用了`>>`运算符，它只会读取空白字符（如空格、制表符或换行符）之前的文本。因此，如果文件中的内容是“学院\nof 软件\n”，第一次读取会得到“学院”，第二次读取会得到“of”。注意，这里不会读取整行，也不会读取中文字符后的换行符。
3. 如果文件打开失败，程序会向标准错误流`cerr`输出错误信息。

main() 函数

```
void main() {
    fn();
    fi();
}
```

这是程序的入口点。它首先调用`fn()`来写入文件，然后调用`fi()`来读取并打印文件内容。但是，请注意，标准C++要求`main()`函数返回一个`int`类型，而不是`void`。因此，这里的`main()`函数签名应该改为`int main()`，并在函数结束时返回一个整数值，通常是`0`表示成功。

修正后的代码

考虑到上述解释和问题，修正后的代码可能如下所示：

```
#include <fstream>
#include <iostream>
using namespace std;

void fn() {
```

```
    ofstream myf("c:\\myname.txt"); // 默认是 ios::out 和 ios::trunc 模式
    myf << "学院\n" << "of 软件\n";
}

void fi() {
    char ch[20];
    ifstream myf("c:\\myname.txt");
    if (myf.fail()) {
        cerr << "文件打开错误" << endl;
        return;
    }
    while (myf >> ch) {
        cout << ch << endl; // 假设每行不超过19个字符（留一个位置给字符串结束符'\0'）
    }
}

int main() { // 注意main函数应返回int类型
    fn();
    fi();
    return 0; // 表示程序成功执行完毕
}
```

在这个修正后的版本中，我移除了无效的文件打开标志，修改了`main()`函数的返回类型，并添加了对文件读取的循环处理（尽管在原始代码中只读取了两段文本）。同时，我也改善了错误处理和输出格式。

---

myf.fail()和myf.is_open()` 在功能上是不同的，它们检查的是文件流的不同状态。

1. `myf.fail()`： 这个方法用于检查流是否处于失败状态。当对文件流进行操作（如读取、写入）时，如果操作失败（例如，尝试读取文件末尾之后的数据，或者写入已关闭的文件），则流的失败标志会被设置。`fail()` 方法会检查这个失败标志，如果之前的操作中有任何失败，它将返回 `true`。
2. `myf.is_open()`： 这个方法用于检查文件流是否已成功打开。如果文件流已成功打开并且可用于读写操作，则 `is_open()` 返回 `true`。这个方法并不关心之前的操作是否失败，它只关心流当前是否处于打开状态。

因此，这两个方法检查的是不同的状态：

- `myf.fail()` 关注的是流操作是否出现了错误。
- `myf.is_open()` 关注的是流是否处于打开状态。

在文件操作中，你可能会想先检查文件是否成功打开（使用 `is_open()`），然后再进行读写操作。如果在读写过程中出现问题，`fail()` 方法会返回 `true`。

请注意，如果文件流操作失败，通常需要调用 `myf.clear()` 来重置流的状态标志，然后才能继续使用该流进行其他操作。

例子3

```
Ofstream::ofstream(char * pFilename,int mode=ios::out,int prot=filebuf::openprot);
Mode://打开方式
```

```
ios::ate                    // 如果文件存在末尾追加
ios::trunc                //  如果文件存在清除内容（默认）
ios::in                     //  输入能力（ifstream默认）
ios::out                   //   输出能力（ofstream默认）
ios::nocreate        //   文件不存在返回错误
ios::noreplace       //   文件存在返回错误
ios::binary             //   二进制方式

Prot: //保护方式
filebuf::openprot  // 允许共享
filebuf::sh_none    // 独占
filebuf::sh_read    // 读共享
filebuf::sh_write  //  写共享
```

好的，作为老师，我会详细解释这些参数和它们的作用。

首先，我们来谈谈ofstream的构造函数。ofstream是C++中用于写入文件的类。它的构造函数允许你指定要打开的文件名以及打开模式和保护方式。

打开方式 (mode)

- ios::ate: 当文件存在时，将输出位置定位到文件的末尾，这样新的输出会被追加到文件的末尾，而不是覆盖原有内容。
- ios::trunc: 如果文件已经存在，它的内容会被截断（即清空），然后你可以从头开始写入新内容。这是ofstream的默认行为。
- ios::in: 允许从文件读取数据。通常与ifstream（输入文件流）一起使用，但在某些情况下，你可能想要一个既可以读又可以写的文件流。
- ios::out: 允许向文件写入数据。这是ofstream的默认能力。
- ios::nocreate: 如果指定的文件不存在，不会创建新文件，而是返回一个错误。
- ios::noreplace: 如果指定的文件已经存在，不会替换或修改它，而是返回一个错误。
- ios::binary: 以二进制模式打开文件，这对于非文本文件（如图像、音频等）或需要精确控制字节的文件非常有用。

保护方式 (prot)

保护方式决定了文件是如何被共享和保护的。

- filebuf::openprot: 允许其他进程或线程共享此文件。这是默认的保护方式。
- filebuf::sh_none: 文件以独占方式打开，不允许其他进程或线程访问。这提供了最高的安全性和一致性保证，但可能降低了并发性。
- filebuf::sh_read: 文件以读共享方式打开，允许其他进程或线程同时读取该文件，但不允许写入。
- filebuf::sh_write: 文件以写共享方式打开。这通常意味着多个写入者可以同时访问文件，但这种模式在实际应用中较为少见，因为它可能导致数据不一致。

使用示例

当你创建一个ofstream对象时，你可以这样指定模式和保护方式：

```
std::ofstream outfile("example.txt", ios::out | ios::binary, filebuf::sh_none);
```

在这个例子中，我们打开了一个名为“example.txt”的文件进行写入操作，并指定了二进制模式和独占访问权限。

希望这些解释能帮助你更好地理解ofstream的构造函数和它的参数！

串流类

例子1

```
//ostrstream   //串输出类
//istrstream    //串输入类    在strstream里面定义   strstream是多继承子类

//没有cout和cin这样的标准全局对象。

ostrstream::ostream(char * ,int size);
istrstream::istream(char * ,int size);
//标准输入输出、外存和内存统称I/O流类
```

```
#include <strstream>
#include <iostream>
using namespace std;
char a[128];
char c;
int main() {
    ostrstream oo(a, 128);
    oo << "Computer";
    oo.put(' ');//I/O流类的函数基本是通用的
    cout << a;

    istrstream ii(a);
    ii >> c;
    ii.get(c);//I/O流类的函数基本是通用的
    return 0;
}
```

首先，这段代码涉及到了C++中的<strstream>库，这是一个处理基于字符数组的流的标准库。不过需要注意的是，<strstream>是一个较旧的库，已经被<sstream>库所取代，后者提供了更安全、更易用的字符串流处理功能。但在这里，我们还是按照你给出的代码来讲解。

代码分析：

1. **变量声明**:
   - `char a[128];`: 声明了一个字符数组a，大小为128字节，用于存储字符串流的数据。
   - `char c;`: 声明了一个字符变量c，稍后将用于从输入流中读取数据。
2. **创建输出字符串流**:
   - `ostrstream oo(a, 128);`: 使用字符数组a和其大小（128字节）初始化了一个输出字符串流oo。这意味着，所有写入oo的数据都会被存储在a中，直到达到128字节的限制。
3. **向输出流中写入数据**:
   - `oo << "Computer";`: 这行代码将字符串"Computer"写入到输出流oo中，也就是写入了字符数组a。
   - `oo.put(' ');`: 使用put方法在字符串末尾添加了一个空格字符。
4. **输出字符数组的内容**:
   - `cout << a;`: 由于a是全局字符数组，并且已经被oo流填充了数据，所以这行代码会输出"Computer "（注意字符串末尾的空格）。

5. **创建输入字符串流并读取数据**:
   - `istrstream ii(a);`: 使用字符数组a初始化了一个输入字符串流ii。
   - `ii >> c;`: 从输入流ii中读取一个字符（实际上是读取到第一个空格之前的所有字符），并将其存储在变量c中。但这里有个问题，c只是一个字符，所以这行代码只会读取'C'（即"Computer"的第一个字符）并存储在c中，后续的字符不会被读取到c中，但仍然会从流中消耗掉。
   - `ii.get(c);`: 这行代码会从输入流中读取下一个字符（在本例中是空格后的第一个字符，但由于前面的>>操作已经读取到了空格，所以这里get方法会读取空格之后的那个字符，但在这个例子中，空格之后没有字符了，所以c的值可能会被设置为EOF或其他未定义的值）。
6. **程序结束**:
   - `return 0;`: 表示程序正常结束。

需要注意的问题和改进点：

- `<strstream>`是一个已经被弃用的库，建议使用`<sstream>`代替，它提供了更安全、更灵活的字符串流操作。
- 在使用>>从流中读取数据时，要注意它只会读取到空格、制表符或换行符为止的数据。如果你想要读取整行或更复杂的数据结构，可能需要使用其他方法（如getline）。
- 在本例中，c只能存储一个字符，所以使用>>读取时会丢失大量数据。如果你想要读取整个单词或更多数据，应该使用字符串或其他适当的数据结构来存储。
- 在读取流中的数据时，最好检查流的状态（如是否达到了流的末尾或是否发生了错误），以确保数据的完整性和准确性。

希望这个讲解能帮助你更好地理解这段代码的工作原理和潜在的问题！

为什么我调试的时候发现ii.get(c)后c从'C’变成了'o'?

在您提供的代码中，如果您发现`ii.get(c)`执行后c的值从`'C'`变成了`'o'`，这通常不应该发生。根据您之前的描述，`ii >> c`会从istrstream对象ii中提取一个字符，并将其存储在变量c中。由于c是一个字符变量，它只能存储一个字符，因此在这个操作中，它应该会存储字符串"Computer "中的第一个字符，也就是`'C'`。

紧接着的`ii.get(c)`应该从流中继续读取下一个字符。但是，由于`ii >> c`已经读取了"Computer"的第一个字符`'C'`，并且会继续读取直到遇到空格为止（在这个例子中，就是"Computer "后的空格），`ii.get(c)`理应读取到这个空格后的第一个字符。然而，在这个字符串后面没有字符了，所以理论上`ii.get(c)`应该会将c设置为EOF（文件结束标记），或者保持c不变（这取决于具体的实现）。

如果您观察到c从`'C'`变成了`'o'`，那么可能的原因有：

1. 之前的读取操作没有正确进行，或者c在`ii >> c`和`ii.get(c)`之间被其他代码意外修改了。
2. 如果您的代码中有其他并发执行的部分，可能存在数据竞争导致c的值被意外修改。
3. 调试器可能显示的是旧的值或者缓存的值，而不是当前的实际值。

为了解决这个问题，您可以：

- 确保在调试过程中没有其他代码修改c的值。
- 检查您的调试器是否显示正确的变量值。
- 在`ii >> c`和`ii.get(c)`之间添加断点或打印语句来确认c的值没有在中间被修改。

此外，如果您正在使用的是老版本的编译器或开发环境，可能存在一些已知的或未知的bug。确保您的开发环境是最新的，并查看是否有相关的bug修复。

最后，考虑到`<strstream>`已经被弃用，并且可能存在兼容性问题，我强烈建议您改用`<sstream>`库中的`std::stringstream`来处理字符串流操作。这不仅更加安全，而且也是C++标准推荐的做法。

如果您仍然遇到问题，请提供更多的代码上下文，以便更精确地诊断问题所在。