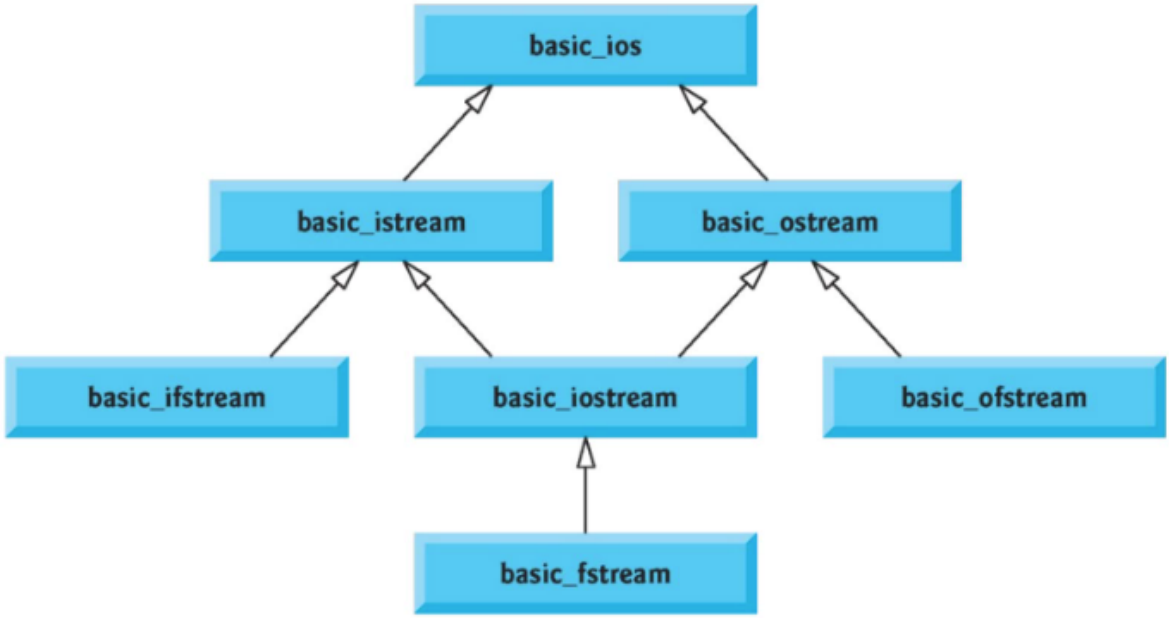
1.I/O template hierarchy

basic_ios作为基础类，作为模板类，不直接用

basic_istream 和basic_ostream继承自它



ifstream: in file stream

ofstream: out file stream

上述二者用来处理文件读写

普通的iostream也就是in stream和out stream只是单独处理普通流输入输出

比如说cin cout等等

```
#include <iostream>
#include <fstream>
#include <cstdlib> //包括程序退出的函数原型
using namespace std;

int main()
{
    //创建一个ofstream对象用于存储读取到的文件对象，类比python中的open as f...
    ofstream outClientFile("clients.txt",ios::out);
    //outClientFile为对象名字
    //ios::out用于写入，在一开始的时候先清空文件
    //ios::in用于只读
    //ios::app用于追加
    //ios::ate用于打开文件并直接把光标移动到文件末尾at end
    //ios::trunc直接清空文件
    if (!outClientFile)
    {
        cerr<<"error"<<endl;//cerr 是console error，用于报错信息
        exit(EXIT_FAILURE);
    }
```

```
    cout<<"success creating file."
}
```

```
int account;
string name;
double balance;

while(cin>>account>>name>>balance){
    outClientFile<<account<<" "<<name<<balance;
    //这里是将这些变量用左插入运算符赋值到文件变量中
    //键盘输入的是cin，屏幕中最后输出的是cout
    cout<<"?";
}
```

```
#include <iostream>  // 用于控制台输入输出
#include <fstream>   // 用于文件输入输出
#include <string>    // 用于std::string类
#include <cstdlib>   // 用于exit函数和EXIT_FAILURE宏
using namespace std; // 使用标准命名空间

enum RequestType { ZERO_BALANCE = 1, CREDIT_BALANCE, DEBIT_BALANCE, END }; // 用户请求的类型
int getRequest(); // 声明一个函数，用于获取用户的请求
bool shouldDisplay(int, double); // 声明一个函数，决定是否显示特定的记录
void outputLine(int, const string &, double); // 声
```

enum为枚举类型，这定义了一个名为 `RequestType` 的枚举类型，它包含四个可能的值：

- `ZERO_BALANCE` 被赋予了显式的值 `1`。
- `CREDIT_BALANCE` 在没有显式赋值的情况下，将会被赋予下一个整数值，即 `2`。
- `DEBIT_BALANCE` 将会被赋予值 `3`。
- `END` 将会被赋予值 `4`。

```
inClientFile>>account>>name>>balance;//文件读取
```

```
while(!inClientFile.eof()){//.eof end of file
    //display record
    if(shouldDisplayFile(request,balance))
        outputLine(account, name, balance);
    inClientFile>>account>>name>>balance;//文件读取

}
inClientFile.clear();//reset eof for next input
inClientFile.seekg(0);//reposition to beginning of file
request = getRequest();
```

关于<<的重载

```
class Point {
private:
    int x, y;

public:
    Point(int x, int y) : x(x), y(y) {}

    // 声明友元函数，为了直接访问到类的私有成员，方便传出成员属性
    friend ostream& operator<<(ostream& os, const Point& pt);//在类内部声明友元函数，注意，这里只是声明，不带大括号的，告诉后面这个函数存在而且是朋友就行
};

// 重载 << 运算符作为友元函数
//如果是outstream则用ostream&
//括号内的传入值，其一必是ostream本身的引用ostream&，另外一个是对象引用，pt为虚对象
ostream& operator<<(ostream& os, const Point& pt) {
    os << "(" << pt.x << ", " << pt.y << ")";
    //当后面用实例化对象调用这个运算符时，就会按照对象的属性依次输出
    return os;//注意这里返回的类型是ostream引用
}

int main() {
    Point p1(1, 2);
    cout << "Point is: " << p1 << endl;
    //这里的p1直接调用了上面的重载运算，变成了<< "(" << pt.x << ", " << pt.y << ")"
    return 0;
}
```

关于cin的>>符号重载也类似

```
 //类内部
friend istream& operator>>(istream& is, Point& pt);

//类外部
// 重载 >> 运算符作为友元函数
istream& operator>>(istream& is, Point& pt) {
    is >> pt.x >> pt.y;
    return is;
}

int main(){
    cin >> p1;
}
```

关于友元函数中第二个参数的传入

对象参数传递： 第二个参数是我们要输出或输入的自定义类的对象。在这里，传递方式取决于我们希望如何处理对象：

- **传递对象的引用：** 如果我们不想在函数中创建对象的副本，我们会传递对象的引用（例如，`const Point&`）。在重载 `<<` 运算符时，通常使用常量引用（`const`），因为输出操作不应修改对象的状态。
- **传递对象的副本：** 如果我们传递对象的副本（例如，直接使用 `Point` 而不是 `Point&`），函数将会收到对象的一个副本。这在大多数情况下是不必要的，因为它会增加额外的内存和处理开销，并且我们通常不需要在输出或输入操作中修改原始对象。

重载 `<<` 运算符： 当重载 `<<` 运算符时，用于输出的对象通常传递为常量引用（`const Type&`），原因如下：

- **避免拷贝：** 传递引用可以避免不必要的对象拷贝，这对于大型对象尤其重要。
- **保持不变性：** 使用常量引用（`const`）是为了确保输出操作不会修改对象的状态。

重载 `>>` 运算符： 当重载 `>>` 运算符时，对象必须传递为非常量引用（`Type&`），原因如下：

- **修改对象：** 输入操作的目的是改变对象的状态，即将输入数据填充到对象中。因此，必须以引用方式传递，以便直接在原始对象上操作。
- **不使用常量引用：** 不使用 `const`，因为我们需要修改引用所指向的对象。

```
ostream& operator<<(ostream& out, const MyClass& obj) {
    // 实现输出
    return out;
}
istream& operator>>(istream& in, MyClass& obj) {
    // 实现输入
    return in;
}
```

关于fill和width函数

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

int main() {
    int n = 77;
    cout.width(5);  // 设置宽度为5
    cout << n << endl;  // 输出：   "   77"（因为默认是右对齐）

    cout.width(5);  // 再次设置宽度为5
    cout << left << n << endl;  // 输出："77   "（左对齐）

    return 0;
}
```

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

int main() {
    cout.fill('*');  // 设置填充字符为 '*'
    cout.width(10);  // 设置宽度为10
    cout << 123 << endl;  // 输出："*******123"（因为默认是右对齐）
```

```
    cout.width(10);  // 再次设置宽度为10
    cout << left << 123 << endl;  // 输出："123*******"（左对齐）

    return 0;
}
```

如果用户输入了一些空白字符后才输入数据，`cin` 会忽略这些空白字符，直接从第一个非空白字符开始读取输入。

检查文件是否打开

if (fin.fail())

if (!fin)

if (!fin.is_open())

binary files

在 C++ 中，二进制文件的读写与文本文件不同，主要在于处理方式和目的。文本文件由字符组成，可以用文本编辑器直接读取和编辑。而二进制文件包含了编码的二进制数据，通常不是为了被人直接读取的，而是为了提供给程序直接读取和写入的结构化数据。

当你打开一个文件进行二进制读写时，你需要指定 `ios::binary` 模式。这告诉C++的文件流库以二进制方式处理文件，不要对数据进行任何特殊的处理，例如不要将换行符 `\n` 转换成 `\r\n`。

读取二进制文件

在你上传的第一段代码中，`ifstream` 对象 `fin` 被用来打开一个名为 "planets.dat" 的文件以进行读取。`ios::in` 标志指定了文件是为了读取而打开的，而 `ios::binary` 指定了文件应该以二进制模式打开。

```
ifstream fin("planets.dat", ios_base::in | ios_base::binary);
```

接着，使用 `read` 函数来从文件中读取数据：

```
fin.read((char *) &pl, sizeof(pl));
```

这里，`read` 函数需要两个参数：一个指向内存中用来存储读取数据的位置的指针，以及要读取的字节数。`(char *) &pl` 是将 `pl` 的地址转换为 `char` 类型指针，这是因为 `read` 函数需要一个 `char` 指针类型的参数。`sizeof(pl)` 指定了要读取的字节数，这通常是目标变量或结构的大小。

写入二进制文件

在你上传的第二段代码中，`ofstream` 对象 `fout` 被用来打开一个名为 "planets.dat" 的文件以进行写入。这里，`ios::out` 标志指定了文件是为了写入而打开的，`ios::app` 是追加模式，`ios::binary` 指定了文件应该以二进制模式打开。

```
ofstream fout("planets.dat", ios_base::out | ios_base::app | ios_base::binary);
```

使用 `write` 函数将数据写入文件：

```
fout.write((char *) &pl, sizeof(pl));
```

与 `read` 类似，`write` 也需要一个指向要写入数据的内存位置的指针，以及要写入的字节数。`(char *) &pl` 将 `pl` 的地址转换为 `char` 类型指针，而 `sizeof(pl)` 指定了要写入的字节数。

在 C++ 中，`ios::out` 和 `ios_base::out` 没有区别。一般用前者，因为前者更加具体一些

seekg以及seekp用法

seekg 为istream成员类型函数，全称为seek get，用于设置输入流的读取位置

```
fin.seekg(0, ios::beg);//将输入流的位置设置到文件开头
//0为绝对位置，相对于后面的ios::beg位置来说
fin.seekg(100,ios::beg);//这里则是移动到beginning后100位
fin.seekg(-10,ios::end);//这里将位置移动到end的前10位
fin.seekg(0,ios::cur);//这里将位置移动到cur光标的位置
```

seekp为ostream成员类型函数