高级语言程序设计·下 理论考试重点

 # 说在前面：易错点、难点考点盘点：

改错题目

1.至少会有一个构造类但是不加上逗号的错误
2.纯虚函数不能被调用
3.抽象基类不能有对象

读程序得到输出结果：

1. 输出格式「输入输出流」
2. 分析拷贝构造、析构函数输出结果「训练参考 lecture 9 – 练习」

A a;	直接用类名定义类对象，调用构造函数
A *a = new A();	动态生成类对象，调用构造函数
A a = b; A c(b);	用已有类对象初始化生成类对象，调用拷贝构造函数
非引用类型的函数形参	调用拷贝构造函数
return时的临时类对象	调用拷贝构造函数

图表 1 类对象生成方式总结，具体可以参考 lecture 9PPT

一、 指针

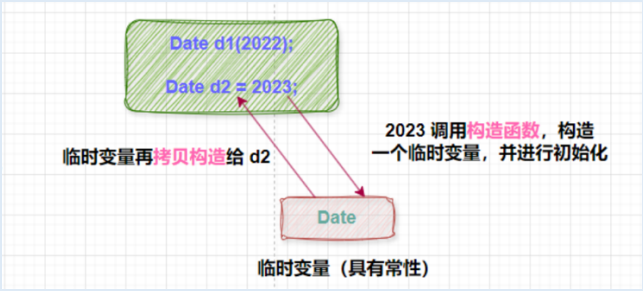
二、 类和对象

1. Explicit 关键字：禁止隐式类型转换[如下图]

```
class Cube
{
private:
    double side;
public:
    explicit Cube(double aSide);
    void TestFunction(Cube aCube);
};
Cube::Cube(double aSide){//带参构造函数
    side = aSide;
}
```

此函数无法实现从double到Cube的隐式类型转换

拓展延伸 - 关于隐式类型构造的过程细节



（参考链接：
https://blog.csdn.net/weixin_45031801/article/details/137796214）

中间发生了调用一次构造函数进行构造，然后再使用拷贝构造函数构造给 d2
其中新创造出来的临时变量不再需要就直接析构

注意类的临时变量的写法和那个分配在栈中的定义不太一样：
A('a',1) | A a('a',1)
临时变量即时构造及时析构 | 分配在栈中的对象随着程序结束析构

2. 分析构造函数、析构函数、拷贝构造函数被调用的情况

「一般来说会在析构函数中写一个输出函数，然后写出程序运行的结果」
一般来说容易出现错误的地方：「一般发生在默认但是未知的拷贝情况下」
（i）代码的本质上是**复制了一个类对象**，那么会隐晦地调用构造函数
（ii）如果你使用函数处理类的对象，那么一般来说如果你不使用引用，那么很有可能出现的情况就是为了创建临时类的对象变量**也会发生拷贝构造**。

构造、析构、拷贝调用分类讨论：
（1）对于**类对象数组**的情况，参考下图

```
class A{
   int x;
};
A *p;
p = new A(); //动态生成一个类A的对象
delete p; //先调用类A的析构函数，再回收A的空间

p = new A[10]; //动态生成10个类A的对象
delete p; //只为第一个类A的对象调用析构函数，然后释放动态空间（错误！）
delete []p; //为数组中每个类A的对象调用析构函数，再回收动态分配的空间
```

图表 2 类对象数组的构造和析构

（2）一种类**内含有其他类对象的构造和析构**情况：

```
class B {
  public:
      int n;
      B(int x){n = x;}
};
class A {
  public:
      int m;
      B b1;
      B b2;
      A (int x1, int x2, int y):b2(x2),b1(x1){
          m = y;}
};
```

构造顺序：b1, b2, A自己
析构顺序：A自己，b2, b1

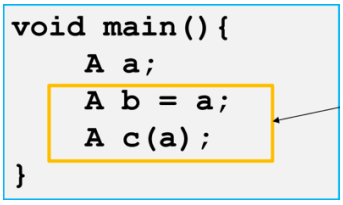
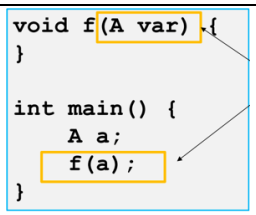
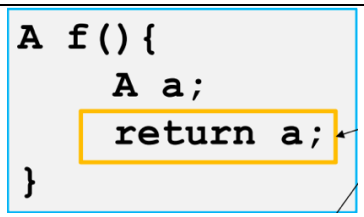
先按照对象成员的声明顺序执行相应的构造函数，再调用自己的构造函数，析构函数顺序相反

图表 3 类内含有其他成员的类成员构造顺序

（3）关于 **new 出来的对象以及析构**的特殊情况

注意：如果代码里面没有 delete 的话，那么你的析构函数就不会在程序结束的时候调用这个析构函数。

（4）拷贝构造发生的几种情况：

		
使用已经构造好的对象给其他对象初始化	类对象为函数实参	函数返回类对象时，系统会产生一个临时变量，临时变量由拷贝构造函数基 于 return 后面的变量生成

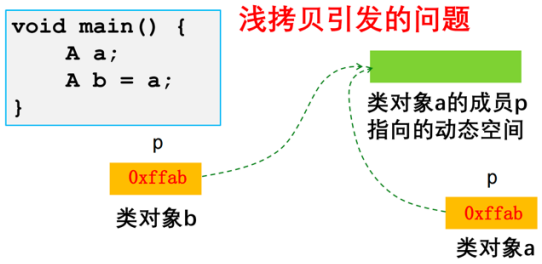
3. 类的基础组成的知识点

（1） 构造函数：*//特征：支持重载、支持参数、有返回类型*

构造函数在类对象生成的时候被系统自动调用；

（2） 析构函数：*//析构函数特征：不支持重载，不支持参数，没有返回类型*

（3） 拷贝构造函数：*//区分浅拷贝、深拷贝*



图表 4 常见浅拷贝样式以及问题

定义：

拷贝构造函数名与类名相同；

拷贝构造函数**没有返回类型**；

形参只能是类对象的引用，拷贝构造函数不能够被重载；

一种浅拷贝的情况 - 类对象赋值

```
class A {
  private:
      int *p;
  public:
      A(){p = new int[2];}
      ~A(){delete []p;}
};
int main() {
    A a;
    A b;
    a = b;
    return 0;
}
```

类对象相互赋值时默认进行对象数据成员的对位拷贝

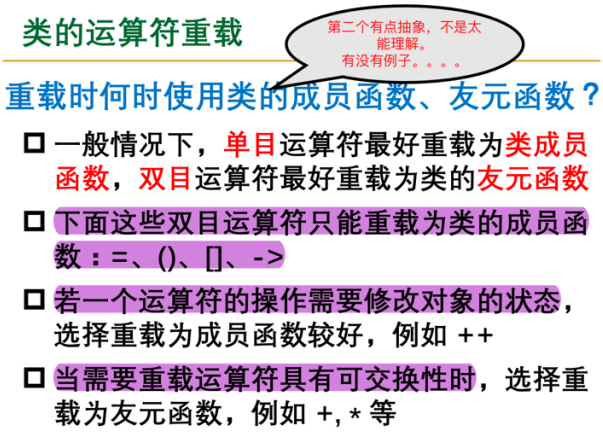
a的成员变量p将被赋值为b的成员变量p的值，两个p指向同一个空间！

但是当 a 和 b 被析构的时候：

（1）a 的原有空间丢失，发生内存泄漏

（2）b 的动态空间被析构两次

4. 运算符重载以及条件



图表 5 类的运算符重载的一般规范条件

5. Static 关键字：

类内 static 成员属于整个类，被所有类对象共享，在类定义的时候生成并且分配空间。

「理解含义：成员包括成员变量、成员函数，那么你就知道下面 static 函数为什么……」

```
class A {
  public:
      static int m;//声明
      int n;
};
int A::m = 1;//定义和初始化
```

定义和初始化必须在类外进行，定义时不再加static

图表 6 static 成员初始化注意事项

类内 static 成员函数「静态成员函数」的权限

静态成员函数**能够访问静态成员变量，但是不能够访问普通成员变量**

但是非静态成员函数**既可以**访问静态成员变量，**也可以**访问普通成员变量

6. Const 关键字

`public:` `const int m;` //常量数据成员定义	常量数据成员初始化之后不能更改， **初始化只能由构造函数完成**
`void f() const ;`	常量成员函数不能修改类的非 static 成员变量
`const int & f(){return m;}`	函数的返回值不能修改

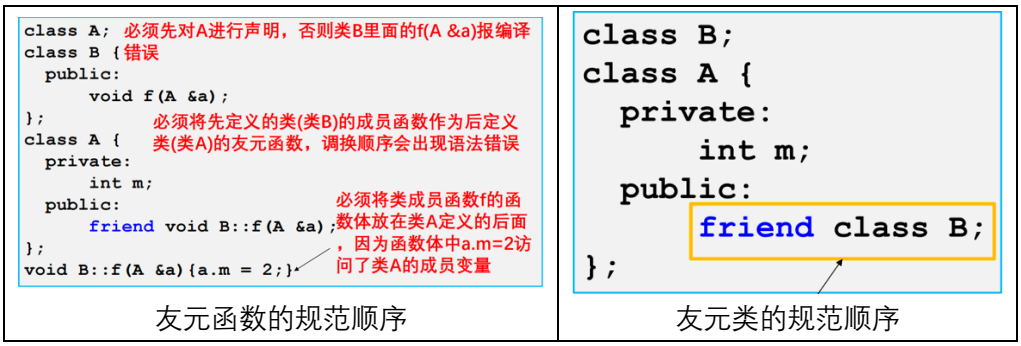
7. This 指针

需要用到 this 指针无非两种情况：
（i）函数参数与类内的成员的名称相同
（ii）类内函数返回的是本身

8. 类的友元

类的友元 friend 关键字的作用就是：使得被修饰的对象可以访问到该类的所有成员

需要注意的点：相关**友元类、友元函数的规范顺序**

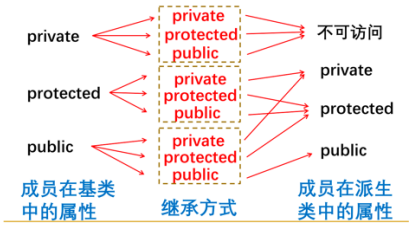


友元函数的规范顺序　　友元类的规范顺序

三、 类的继承

对应 lecture10 考点的 1-5

1.派生类的访问权限



图表 7 派生类的继承前后成员的属性变化图

我们能够显然发现几个显著的特点 – 基类中 private 无论如何继承都不可访问，除此之外只要是 private 的继承方式的在派生类的属性都变成了 private。而 public 只有经过 public 方式

才可能在派生类中是 public 属性。其他的均为 protected；

2.派生类的构造函数

基本知识点：

（1） 参数列表的规范 - 在派生类构造函数中，只要基类不是使用无参的默认构造函数都要显式给出基类名和参数表

（2） 构造/析构顺序 –

i.调用各基类的构造函数，调用顺序为派生继承时的**基类声明顺序.但是如果按照下面图拍呢所示，我们优先按照这个继承的顺序进行调用**

ii.若派生类含有对象成员的话，调用各对象成员的构造函数，**调用顺序按照派生类中对象成员的声明顺序**

```
class CD:public CB,public CC {
    int d;
    CC obcc;//CC类型的对象成员
    CB obcb;//CB类型的对象成员
public:
    CD(int n1,int n2,int n3,int n4)
  :CC(n3,n4), CB(n2), obcb(100+n2),
  obcc(100+n3,100+n4) {
      d=n1;      cout<<"CD::d="<<d<<endl; };
    ~CD(){cout<<"CDobj is destructing"<<endl;}
}; //先基类CB、CC，再对象成员CC、CB，最后派生类CD
```

图表 8 考虑了三种情况后的一般类的构造/析构的顺序

3. 派生类的拷贝构造函数

重要的点 – 关于派生类拷贝构造函数调用了基类的什么函数：

i.派生类的拷贝构造函数，**没有显示地调用基类拷贝构造函数**，将会调用基类无参构造函数！

ii.派生类的拷贝构造函数显示地调用基类拷贝构造函数,将会调用基类的拷贝构造函数。

- 派生类可以使用using关键字，显式地“继承”基类的构造函数（无参构造函数除外）
- 继承来的基类构造函数可以当作派生类的构造函数使用，初始化派生类对象的基类部分

```
class Carton : public Box
{
  using Box::Box; //继承了Box所有的构造函数（无参构造函数除外）
  private:
    std::string material {"Cardboard"};
  public:
    Carton(double lv, double wv, double hv, std::string
mat) : Box {lv, wv, hv}, material {mat}
   {cout<<"Carton(double,double,double,string) called.";}
};
```

图表 9 using Box::Box 的继承关系

4. 派生类的析构函数

功能和之前类中的析构函数的功能差不多

5. 友元的继承、静态成员的继承

友元继承：基类的友元也会成为其派生类的友元

静态成员的继承：基类的成员如果是公有的或者保护的，那么这些静态成员的拷贝都只有一个，为基类和派生类所有对象共享

四、 类的多态

对应 lecture10 考点的 6-11：
6．赋值兼容性问题

根本核心：如果派生类没有重载赋值运算符但是基类重载赋值运算符，那么派生类的派生成员按照系统自定义的按位拷贝，基类的基成员就按照基类的重载的赋值运算符进行赋值。如果派生类和基类都没有重载，那么都按位拷贝。

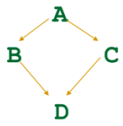
- **基类对象和派生类对象之间允许有下述的赋值关系：**
  - **基类对象 = 派生类对象**
    - 只赋“共性成员”部分
    - 反方向赋值“派生类对象 = 基类对象”不被允许
  - **指向基类对象的指针 = 派生类对象的地址**
    - 访问非基类成员部分时，要经过指针类型的强制转换
    - 下述赋值不允许：指向派生类类型的指针 = 基类对象的地址
  - **基类的引用 = 派生类对象**
    - 通过引用只可以访问基类成员部分
    - 下述赋值不允许：派生类的引用 = 基类对象

由“大”到“小”可以，反方向不行

图表 10 赋值方向：只允许派生类到基类

7．二义性问题的三种情况以及解决方案「**虚基类**」

单继承父类与子类间重名	多继承下二基类间重名	多基混合继承包含两个基类实例

单继承时父类与子类成员重名时，**对于子类而言，不加类名限定时候认为是处理子类成员**，如果要访问父类重名成员，需要加上类名限定	多继承情况下访问二基类成员的重名时，**对于子类而言不加类名限定处理子类成员**，如果要访问父类重名成员，需要加上类名限定	多级混合继承情况下，若类D从两条不同“路径”同时对类A进行了一般性继承，则类D的对象中会同时包含着两个类A的实例，要通过类名限定来指定访问两个类A实例中的哪一个
		`class A`
		`class B : public A`
		`class C : public A`
		`class D : public B, public C`
		 

虚基类：

核心：虚继承可以避免因为菱形继承关系导致的二义性【对应上面**第三种情况**】

多级混合继承情况下，若类D从两条不同“路径”同时对类A进行了虚拟继承的话，则类D的对象中只包含着类A的一个实例，被虚拟继承的基类A被称为虚基类

虚基类的说明：在定义派生类时增加关键字virtual

```
class A
class B : virtual public A
class C : virtual public A
class D : public B, public C
```

图表 11 虚基类的代码书写格式

8．函数重写与静态绑定

函数重写	静态绑定

▪ 仅在基类与其派生类的范围内实现
▪ 允许多个不同函数使用完全相同的函数名、函数参数表以及函数返回类型
`class graphelem {`
`protected:`
`  int color;`
`public:`
`  graphelem(int col) {`
`  color=col;`
`  }`
`  void draw(){cout<<"graphelement"};`
`};`
	`void main() {`
	`  line ln1;`
	`  line *pl = &ln1;`
	`  ln1->draw(); //line::draw()`
	`  ln1->graphelem::draw(); //graphelem::draw()`
	`  graphelem *pg = &ln1; //基类指针指向子类对象`
	`  pg->draw(); //graphelem::draw()`
	`}`
	当基类指针指向子类对象时，通过基类指针只能访问子类里面的基类部分，因此，pg->draw() 指的是基类的draw()，这是因为编译器采用静态绑定！

9．静态绑定问题的解决方案「**虚函数**」

虚函数注重实用而不是理论，我们不再重点关注这个

- 在程序运行时动态地进行，根据当时的情况来确定调用哪个同名函数（静态绑定是在编译阶段完成）
- 当涉及到多态性和虚函数时应该使用动态绑定（非虚函数都是静态绑定）
- 动态绑定的优点是灵活性强，但效率低（静态绑定灵活性差，但效率高）

图表 12 动态绑定的特点

10．多态发生的条件

动态绑定的发生条件	多态发生的条件

■ 必须把函数定义为类的虚函数 ■ 必须通过指针或引用调用虚函数	■ 必须把函数定义为类的虚函数 ■ 类之间应满足子类型关系，通常表现为一个类从另一个类公有派生而来 ■ 必须先使用基类指针(或引用)指向子类的对象，然后使用基类指针或者引用调用虚函数

11．纯虚函数和抽象积累「注重实践，理论考试不是重点，知道即可」

五、 类的模板

1 ．函数模板

本部分不难，只涉及几点提醒：

（1）函数模板实例化的时机：实例化在编译阶段完成，在函数调用处发生！

（2）函数模板实例化后的函数不会执行实参到形参的自动类型转换

```
cout<<max(23,-5.6)<<endl;// 错误！不进行实参到形参类型的自动转换
```

图表 13 Max（double，double）函数模板不接受自动类型转换

（3）函数模板与函数重载的先后调用顺序：**首先检查是否存在重载函数，若匹配成功则调用该函数，否则再去匹配函数模板。**

（4）函数模板返回类型的显式指定要求：

```
template <class T1, class T2, class T3>
     T3 max1 (T1 a, T2 b){
     return a > b ? a : b;
}

void main() {
    max1<int, double>(3, 5);//错误！只指定了T1和T2，T3仍然无法无法推断
    max1<int, ,int>(2.4, 3); //错误！不能跳跃指定

    max1<int, double, char>(3, 5); //OK!
}
```

图表 14 关于多个模板参数后显式指定的要求

2. 类模板

我们不多赘述类模板的一般格式。我们也只涉及提醒几个重要的方面：

（1） 类模板的实例化只会发生在需要类的定义的时候

（2） 类模板的静态成员是模板实例化类的静态成员，**对于每一个实例化类，其所有的对象共享其静态成员。但是不同实例化的类不共享对应的静态成员。**

（3） 类模板的所有函数都是函数模板，也可以是另外一个独立的函数模板（参数不同）

（4） 类模板与友元模板函数的关系

如果类模板的友元函数是**普通函数**，则友元函数是该类模板**任意类实例的友元**
如果友元函数是**与类模板无关的模板函数**，则**友元函数的任意函数实例是任意类实例的友元**
友元函数是**与类模板有关的模板函数**，则友元函数**只是该类模板特定类实例的友元**

3. 类模板参数检测和特例版本

解决方案一：对complex类进行<<运算符重载

解决方案二：在statck类模板里填加showtop函数的特例版本专门支持complex类

此函数的特例版本专门针对complex类

```
void stack<complex>::showtop() {
    cout<<"Top_Member:"<<data[top].real<<"
"<<data[top].image<<endl;
 //分别输出实部real和虚部image
}
```

图表 15 关于这种特例问题的所有解决方案，参考 PPT lecture 11

2 输入输出流

1. 输入流的控制

Get 函数	Getline 函数

提取一个字符（存放在字符型参数），格式为：
`istream& get( char& rch );`
`char ch, ch1;`
`cin.get(ch);`
`//从键盘上读入字符，并存放在字符变量ch中`
`cin.get(ch).get(ch1)<<endl;`
`//连续从键盘上读入两个字符，放到ch和ch1中`
get()函数和运算符“>>”的区别在于：
get()函数能够读取空格、回车符和制表符等空白字符，而“>>”只能读取非空白字符
	从输入设备读入一行，格式为：
	`istream& getline(char* pch, int n, char delim = '\n');`
	▪ 从输入设备读取n-1个字符，并在其后加上字符串结束符'\0'，存入第1个参数所指向的内存空间中
	▪ 若在读取够n-1个字符前遇到由第3个参数指定的终止符，则提前结束读取，终止符的默认值是`\n`
	▪ 若读取成功，函数返回值为真（非0）值，若读取失败（遇到文件结束符EOF），函数返回值为假（0）值
	getline()能够读取空格、回车符和制表符等空白字符

2. 输出流的控制（输出格式 – 只关注常用的一些格式）

通过公有函数控制输出格式：

输出宽度/左右对齐设置	输出数据的格式	输出的进制格式设置

`int main() {`
`  char *s = "Hello";`
`  cout.fill('*');  // 置填充符`
`  cout.width(10);  // 置输出宽度`
`  cout.setf(ios :: left); // 左对齐`
`  cout << s << endl;`
`  cout.width(15); // 置输出宽度`
`  cout.setf(ios :: right, ios :: left);`
`  // 清除左对齐标志位，置右对齐`
`  cout << s << endl;`
`  return 0;`
`}`
Hello*****
**********Hello
	`int main() {`
	`  float a = 123.4567;`
	`  cout.flags(ios::right | ios::fixed);`
	`  // 置右对齐、定点输出，默认保留6位小数`
	`  cout<< a << endl;`
	`  cout.flags(ios::left | ios::scientific);`
	`  // 置左对齐、科学计数法`
	`  cout.precision(2); //保留2位小数`
	`  cout << a << endl;`
	`  return 0;`
	`}`
	123.456700
	1.23e+002
		`int main() {`
		`  int a , b , c ;`
		`  cin.setf(ios::dec, ios::basefield) ;`
		`  cin>>a ; //改为十进制输入，之前的进制清除`
		`  cin.setf(ios::hex, ios::basefield) ;`
		`  cin>>b ; //十六进制输入`
		`  cin.setf(ios::oct, ios::basefield) ;`
		`  cin>>c ; //八进制输入`
		`  cout.setf(ios::dec, ios::basefield);`
		`  cout<<a<<" "<<b<<" "<<c<<endl; //十进制输出`
		`  cout.setf(ios::hex, ios::basefield);`
		`  cout<<a<<" "<<b<<" "<<c<<endl; //十六进制输出`
		`  cout.setf(ios::oct, ios::basefield);`
		`  cout<<a<<" "<<b<<" "<<c<<endl; //八进制输出`
		`  return 0;`
		`}`

通过格式控制符控制输出格式#include<iomanip>
基本上的常见格式控制就是上下展示的这些

```
#include <iomanip>
void main() {
    cout.width(6); //只管随后一个数的域宽，默认右对齐
    cout<<4785<<27.4272<<endl; //□□478527.4272
    cout<<setw(6)<<4785<<setw(8)<<27.4272<<endl;
    //□□4785□27.4272
    cout.width(6);
    cout.precision(3);
    /*当格式为ios::scientific或ios::fixed时，浮点数精度np指
    小数点后的位数，否则指有效数字；此例中没有fixed或
    者scientific，所以指的是有效数字为3*/
    cout<<4785<<setw(8)<<27.4272<<endl;
    //□□4785□□□□27.4
    cout<<setw(6)<<4785<<setw(8)<<setprecision(2);
    cout<<27.4272<<endl<<endl;
    //□□4785□□□□□□27
    //setprecision(2)设置浮点数的有效数字
    cout.setf(ios::fixed, ios::floatfield);
    //今后以定点格式显示浮点数(无指数部分)
    cout.width(6);
    cout.precision(3);
    //当格式为ios::fixed时，设置小数点后的位数
    cout<<4785<<setw(8)<<27.4272<<endl;
    //□□4785□□27.427
}
```

3. 磁盘文件输入输出流对象

ofstream	文件输入流	`ofstream(const char* szName, int nMode=ios::out);`
ifstream	文件输出流	`ifstream(const char* szName, int nMode=ios::in);`
fstream	fstream 由 ifstream 和 ofstream 派生既然支持文件读又支持文件写！	`fstream(const char* szName, int nMode = ios::in | ios::out);`

方式	作用
ios::in	以输入方式打开文件，对文件进行读操作，该文件必须存在
ios::out	以输出方式打开文件，对文件进行写操作
ios::app	以追加方式打开文件，所有输出附加在文件末尾
ios::ate	打开文件时，文件指针定位在文件尾
ios::binary	以二进制方式打开文件，缺省的方式是文本方式
ios::trunc	如果文件已存在，则先删除文件内容

图表 16 所有的 nMode

ios::app一般只和ios::out配合，用于以追加的方式打开输入流
fstream fs("aaa.txt", ios::out | ios::app)

ios::ate一般和ios::in相配合，打开文件时将文件指针定位到文件尾
fstream fs("aaa.txt", ios::in | ios::ate);

如果ios::ate和ios::out相结合，将清空原文件
fstream fs("aaa.txt", ios::out | ios::ate);

图表 17 常见的 nMode 的搭配规则

4. 磁盘文件常用的函数架构

open 函数	其实就是构造函数（活着理解为对象赋值/初始化函数）
Put 函数	File.put(char c):一个字符一个字符地**写入文件**
Get 函数	file.get(char c):一个字符一个字符地**从文件读取**
Getline 函数	按照行读取文件

5. 磁盘读写二进制文件

文本文件	数据按 ASCII 码的形式存储在文件中	12.345以文本方式写入文件时，会自动将每个数字（包括小数点）都转换为ASCII码来存，所以一共要存6个字节，分别是'1', '2', '.', '3', '4', '5'的ASCII码
二进制文件	数据按照二进制数据存储在文件中	12.345在内存中占8个字节，以补码形式存储(0x713D0AD7A3B02840)，用write函数写入文件时，直接将内存中的8个字节拷贝到文件中

Read 函数	`istream& read(char* pch, int nCount )` **功能：从文件中读入nCount个字符放入pch缓冲区中（若读至文件结束尚不足nCount个字符时，也将立即结束本次读取过程）**
Write 函数	`ostream& write(const char* pch, int nCount)` **功能：将pch缓冲区中的前nCount个字符写出到某个文件中**

```
ofstream fout("wrt_read_file.bin",
ios::binary);
//打开用于“写”的二进制磁盘文件
int Len=strlen(str);
fout.write((char*)(&Len), sizeof(int));
fout.write(str, Len);
//数据间无需分割符
fout.write((char*)(&ss), sizeof(ss));
fout.close();

char str2[80];
ifstream fin("wrt_read_file.bin",
ios::binary);
fin.read((char*)(&Len), sizeof(int));
fin.read(str2, Len);
```

图表 18 二进制文件读写的基本格式

6. 磁盘对文件内容任意访问 – 文件指针

Infile.**seekg**();	定位**输入文件**流对象的文件指针	**istream& seekg(long offset, seek_dir origin=ios::beg);** **ostream& seekp(long offset, seek_dir origin=ios::beg);**
Outfile.**seekp**();	定位**输出文件**流对象的文件指针	

ios::beg：文件首
ios::cur：文件指针当前位置
ios::end：文件尾

图表 19 文件指针初始化的位置

同理我们也可以通过 tellg(),tellp()分别获取输入文件/输出文件对象当前的文件指针位置「**从文件首到当前位置的字节数**」

7. 字符串流：使用字符串流必须包含头文件<strstream>

字符串流类	构造函数	常见格式

ostrstream	ostrstream类将不同类型的信息转换为字符串，并存放在(输出到)用户设定的字符数组中
	□ 该类最常用的构造函数的一般格式为：
	ostrstream(char* str, int n, int mode = ios::out);
	//其中str是用户指定的字符数组，n用来指定这个数组最多能存放的字符个数, mode给出流的方式
		int main() {
		ostrstream os; //未指定字符数组，系统来分配
		string str = "abcef";
		int i = 1000;
		os << str << i; //将str和i连成一个字符串，存放在系统分配的空间中
		cout << os.str()<<endl; //输出"abcef1000"
		cout<<os.pcount(); //ostrstream成员函数，统计当前已经输出了多少个字符
		return 0;

istrstream	istrstream类可将用户字符数组中的字符串取出（读入），而后反向转换为各种变量的内部形式
	□ 一参构造函数：istrstream(char* str);
	▪ 由参数str 指定一个以'\0'为结束符的字符串（字符数组）
	□ 二参构造函数：istrstream(char* str, int n);
	▪ 由参数str 指定字符数组，由第二参数n 指出仅使用str的前n个字符
		char *str = "1234 100.35 ";
		istrstream inp(str);
		int nNumber;
		float balance;
		inp>>nNumber; //从字符串中拆出整数1234
		inp>>balance; //从字符串中拆出float 100.35
		cout<<nNumber<<' '<<balance;

stringstream	stringstream ostr("ccc"); //初始字符串为"ccc"
	ostr.put('d'); //向字符串添加'd'，注意从字符串开头增加，即覆盖掉第1个'c'
	ostr.put('e'); //向字符串添加'e'，覆盖掉第2个'c'
	string gstr = ostr.str();
	cout<<gstr<<endl; //输出"dec"
	char a;
	ostr>>a;
	cout<<a;
		stringstream sstr;
		//--------int转string----------
		int a=100;
		string str;
		sstr<<a; //将100写入sstr
		sstr>>str; //将100转为string
		cout<<str<<endl;
		//--------string转char[]--------
		sstr.clear(); //clear()成员函数，清除缓存
		string name = "colinguan";
		char cname[200];
		sstr<<name;
		sstr>>cname;
		cout<<cname;