对象和对象的定义

对象的定义格式: <类名><对象名表>TDate date1, date2, *Pdate, date[31];

对象成员的表示方法 <对象名>.<成员名> 或者 <对象名>.<成员名>(<参数表>)前者用来表示数据成员的，后者用来表示成员函数的。 <对象指针名>-><成员名>或者 <对象指针名>-><成员名>(<参数表>)

下面的两种表示是等价的：<对象指针名>-><成员名>与(*<对象指针名>).<成员名>

对象的初始化

构造函数:创建对象时，使用给定的值来将对象初化。

析构函数:释放一个对象,在对象删除前，用它来做一些清理工作.

构造函数的特点如下：

1、构造函数是成员函数，函数体可写在类体内，也可定在类体外。

2、构造函数是一个特殊的函数，该函数的名字与类名相同，该函数不指定类型说明，它有隐含的返回值，该值由系统内部使用。该函数可以一个参数，也可以有多个参数。

3、构造函数可以重载，即可以定义多个参数个数不同的函数。

4、程序中不能直接调用构造函数，在创建对象时系统自动调用构造函数。

析构函数的特点如下：

1、析构函数是成员函数，函数体可写在类体内，也可定在类体外。

2、析构函数也是一个特殊的函数，它的名字同类名，并在前面加“~”字符，用来与构造函数加以区别。析构函数不指定数据类型，并且也没有参数。

3、一个类中只可能定义一个析构函数。

4、析构函数可以被调用，也可以系统调用。在下面两种情况下，析构函数会被自动调用。

在类定义时没有定义任何构造函数时，则编译器自动生成一个不带参数的缺省构造函数，其格式如下：

<类名>::<缺省构造函数名>()

{

}

用缺省构造函数对对象初始化时，则将对象的所有数据成员都初始化为零或空。

同理，如果一个类中没有定义析构函数时，则编译系统也生成一个称为缺省析构函数数，其格式如下：

<类名>::~<缺省析构函数名>

{

}

<缺省析构函数名>即为该类的类名。缺省析构函数是一个空函数。

拷贝初始化构造函数

拷贝初始化构造函数:用一个已知的对象来初始化一个被创建的同类的对象。拷贝初始化构造函数实际上也是构造函数，它是在初始化时被调用来**将一个已知对象的数据成员的值拷贝给正在创建的另一个同类的对象**

拷贝初始化构造函数的特点如下：

1、该函数名同类名，因为它也是一种构造函数，并且该函数也不被指定返回类型。

2、该函数只有一个参数，并且是对某个对象的引用。

3、每个类都必须有一个拷贝初始化构造函数，其格式如下：

<类名>::<拷贝初始化构造函数名>(const<类名>&<引用名>)

其中，<拷贝初始化构造函数名>是与该类名相同的。const 是一个类型修饰符，被它修饰的对象是一个不能被更新的常量。如果类中没有说明拷贝初始化构造函数，则编译系统自动生成一个具有上术形式的缺省拷贝初始化构造函数。作为该类的公有成员。

总结：**拷贝初始化构造函数的功能就是用一个已知的对象来初始化另一个对象**。在下述三种情况下，需要用拷贝初始化构造函数来用一个对象初始化另一个对象。

1、明确表示由一个对象初始化另一个对象时，如：TPoint P2(P1);

2、当对象作为函数实参传递给函数形参时，如：上例 P = f(N);

3、当对象用为函数返回值时，如：上例 return R;

静态成员(static)

静态数据成员：静态成员可实现多个对象之间的数据共享，且静态数据成员不会破坏隐藏的原则，即保证了安全性。因此，静态成员是类的所有对象中共享的成员，而不是某个对象的成员。

使用静态数据成员可以节省内存，因为它是所有对象所公有的，因此，对多个对象来说，静态数据成员只存储一处，供所有对象共用。静态数据成员的值对每个对象都是一样，但它的值是可以更新的。只要对静态数据成员的值更新一次，保证所有对象存取更新后的相同的值，这样可以提高时间效率。

1、静态数据成员在定义或说明时前面加关键字 static。static int Sum;

2、静态数据成员初始化的格式如下：**<数据类型><类名>::<静态数据成员名>=<值>** int Myclass::Sum = 0;

这表明：

(1) 初始化在类体外进行，而前面不加 static，以免与一般静态变量或对象相混淆。

(2) 初始化时不加该成员的访问权限控制符 private，public 等。

(3) 初始化时使用作用域运算符来标明它所属类，因此，静态数据成员**是类的成员，而不是对象的成员。**

3、静态数据成员是静态存储的，它是静态生存期，必须对它进行初始化。

4、**引用静态数据成员时，采用如下格式：<类名>::<静态成员名>**

静态成员函数：属于类的静态成员，不是对象成员。因此，对静态成员的引用不需要用对象名。static void f1(M m); 在静态成员函数的实现中不能直接引用类中说明的非静态成员，可以引用类中说明的静态成员。如果静态成员函数中要引用非静态成员时，可通过对象来引用。**调用静态成员函数使用如下格式：<类名>::<静态成员函数名>(<参数表>);** M::f1(P); //调用时不用对象名。

友元(friend)

友元是一种定义在类外部的普通函数，但它需要在类体内进行说明，说明时前面**加关键字 friend。**友元不是成员函数，但是它可以访问类中的私有成员。友元的作用在于提高程序的运行效率，但是，它破坏了类的封装性和隐藏性，使得非成员函数可以访问类的私有成员。

友元函数：能够访问类中的私有成员的非成员函数。友元函数从语法上看，它与普通函数一样，即在定义上和调用上与普通函数一样，不需要指出所属的类。在调用友元函数时，也是同普通函数的调用一样

友元类：即一个类可以作另一个类的友元。当一个类作为另一个类的友元时，这就意味着这个类的所有成员函数都是另一个类的友元函数。

局部类和嵌套类

局部类：在一个函数体内定义的类称为局部类。局部类中只能使用它的外围作用域中的对象和函数进行联系，因为外围作用域中的变量与该局部类的对象无关。**局部类中不能说明静态成员函数，并且所有成员函数都必须定义在类体内。**

嵌套类：在一个类中定义的类称为嵌套类，定义嵌套类的类称为外围类。定义嵌套类的目的在于隐藏类名，减少全局的标识符，从而限制用户能否使用该类建立对象。这样可以提高类的抽象能力，并且强调了两个类(外围类和嵌套类)之间的主从关系。下面是一个嵌套类的例子：
对嵌套类的若干说明：

1、从作用域的角度看，嵌套类被隐藏在外围类之中，该类名只能在外围类中使用。如果在外围类的作用域内使用该类名时，需要加名字限定。

2、从访问权限的角度来看，嵌套类名与它的外围类的对象成员名具有相同的访问权限规则。不能访问嵌套类的对象中的私有成员函数，也不能对外围类的私有部分中的嵌套类建立对象。

3、嵌套类中的成员函数可以在它的类体外定义。

4、嵌套类中说明的成员不是外围类中对象的成员，反之亦然。嵌套类的成员函数对外围类的成员没有访问权，反之亦然。因此，在分析嵌套类与外围类的成员访问关系时，往往把嵌套类看作非嵌套类来处理。

5、在嵌套类中说明的友元对外围类的成员没有访问权。

6、如果嵌套类比较复杂，可以只在外围类中对嵌套类进行说明，关于嵌套的详细的内容可在外围类体外的文件域中进行定义。

对象的生存周期

按生存期的不同对象可分为如下三种：

1、局部对象：当对象被定义时调用构造函数，该对象被创建，当程序退出定义该对象所在的函数体或程序块时，调用析构函数，释放该对象。

2、静态对象：当程序第一次执行所定义的静态对象时，该对象被创建，当程序结束时，该对象被释放。

3、全局对象：当程序开始时，调用构造函数创建该对象，当程序结束时调用析构函数释放该对象。

局部对象是被定义在一个函数体或程序块内的，它的作用域小，生存期也短。

静态对象是被定义在一个文件中，它的作用域从定义时起到文件结束时止。它的作用域比较大，它的生

存期也比较大。

全局对象是被定义在某个文件中，而它的作用域却在包含该文件的整个程序中，它的作用域是最大的，它的生存期也是长的。

对象指针和对象引用

指向类的成员的指针：

指向数据成员的指针格式如下：**<类型说明符><类名>::*<指针名>**
指向成员函数的指针格式如下：**<类型说明符>(<类名>::*<指针名>)(<参数表>)**

由于类不是运行时存在的对象。因此，在使用这类指针时，需要首先指定 A 类的一个对象，然后，通过对象来引用指针所指向的成员。A a;　　a.*pc = 8;其中，运算符.*是用来对指向类成员的指针来操作该类的对象的。如果使用指向对象的指针来对指向类成员的指针进行操作时，使用运算符->*。A *p = &a; //a 是类 A 的一个对象，p 是指向对象 a 的指针。p ->* pc = 8;

再看看指向一般函数的指针的定义格式：**<类型说明符>*<指向函数指针名>(<参数表>)**
关于给指向函数的指针赋值的格式如下：**<指向函数的指针名>=<函数名>**
关于在程序中，使用指向函数的指针调用函数的格式如下：**(*<指向函数的指针名>)(<实参表>)**
如果是指向类的成员函数的指针还应加上相应的对象名和对象成员运算符。

对象指针和对象引用作函数的参数

1. 对象指针作函数的参数

使用对象指针作为函数参数要经使用对象作函数参数更普遍一些。因为使用对象指针作函数参数有如下两点好处：

(1) 实现传址调用。可在被调用函数中改变调用函数的参数对象的值，实现函数之间的信息传递。

(2) 使用对象指针实参仅将对象的地址值传给形参，而不进行副本的拷贝，这样可以提高运行效率，减少时空开销。

当形参是指向对象指针时，调用函数的对应实参应该是某个对象的地址值，一般使用&后加对象名。

2. 对象引用作函数参数

在实际中，使用对象引用作函数参数要比使用对象指针作函数更普遍，这是因为使用对象引用作函数参数具有用对象指针作函数参数的优点，而用对象引用作函数参数将更简单，更直接。所以，在 C++编程中，人们喜欢用对象引用作函数参数。

this 指针

this 指针是一个隐含于每一个成员函数中的特殊指针。它是一个指向正在被该成员函数操作的对象,也就是要操作该成员函数的对象。当对一个对象调用成员函数时，编译程序先将对象的地址赋给 this 指针，然后调用成员函数，每次成员函数存取数据成员时，由隐含作用 this 指针。

而通常不去显式地使用 this 指针来引用数据成员。同样也可以使用*this 来标识调用该成员函数的对象。下面举一例子说明 this 指针的应用。

对象与数组

对象数组：对象数组是指数组元素为对象的数组。该数组中若干个元素必须是同一个类的若干个对象。对象数组的定义、赋值和引用与普通数组一样，只是数组的元素与普通数组不同，它是同类的若干个对象。

1. 对象数组的定义，**对象数组定义格式如下：<类名><数组名>[<大小>]...** DATE dates[7];

其中，<类名>指出该数组元素是属于该类的对象，方括号内的<大小>给出某一维的元素个数。一维对象数组只有一个方括号，二维对象数组要有两个方括号

2. 对象数组的赋值，对象数组可以被赋初值，也可以被赋值

DATE dates[4]={ DATE(7, 7, 2001), DATE(7, 8, 2001), DATE(7, 9, 2001), DATE(7, 10, 2001) }

1. 指向数组的指针，**指向一般数组的指针定义格式如下：<类型说明符>(*<指针名>)[<大小>]...**

其中，用来说明指针的 * 要与<指针名>括在一起。后面用一个方括号表示该指针指向一维数组，后面用二个方括号表示该指针指向二维数组。<类型说明符>用来说明指针所指向的数组的元素的类型。int (*P)[3]; P 是一个指向一维数组的指针，该数组有 3 个 int 型元素。

指向对象数组的指针，则把<类型说明符>改为<类名>即可：<类名>(*<指针名>)[<大小>]...

指向数组的指针的主要应用思想是：将数组的首地址(二维数组的某个行地址)赋给指针，然后通过循环(for)改变指针指向的地址，从而动态的访问数组中各个元素。

2. 指针数组

所谓指针数组指的是数组元素为指针的那类数组。一个数组的元素可以是指向同一类型的一般指针，也可以是指向同一类类型的对象。

一般指针数组的定义格式如下：<类型名>*<数组名>[<大小>]...

其中，*加在<数组名>前面表示该数组为指针数组。[<大小>]表示某一维的大小，即该维的元素个数，…表示可以是多维指针数组，每一个[<大小>]表示一维。

int * pa[3];　　char * pc[2][5];

对象指针数组的定义如下：对象指针数组是指该数组的元素是指向对象的指针，它要求所有数组元素都是指向同一个类类型的对象的指针。 **<类名>*<数组名>[<大小>]...**

它与前面讲过的一般的指针数组所不同的地方仅在于该数组一定是指向对象的指针。即指向对象的指针用来作该数组的元素。

常类型(const)

1. **一般常量：定义或说明一个常数组可采用如下格式： <类型说明符> const <数组名>[<大小>]…**

或者 const <类型说明符> <数组名>[<大小>]…

2. **常对象：** **常对象是指对象常量，定义格式如下： <类名> const <对象名> 或者 const <类名> <对象名>**

定义常对象时，同样要进行初始化，并且该对象不能再被更新，修饰符 const 可以放在类名后面，也可以放在类名前面。

3. **常指针**

一个指向字符串的常量指针： char * const prt1 = stringprt1;其中，ptr1 是一个常量指针。ptr1 = stringprt2;是非法的。 *ptr1 = "m";是合法的。

一个指向字符串常量的指针： const * ptr2 = stringprt1;ptr2 是一个指向字符串常量的指针。ptr2 所指向的字符串不能更新的，而 ptr2 是可以更新的。 *ptr2 = "x";是非法的， ptr2 = stringptr2;是合法的。

4. **常引用：** 使用 const 修饰符也可以说明引用，被说明的引用为常引用，该引用所引用的对象不能被更新。其定义格式如下： **const <类型说明符> & <引用名>** const double & v;

在实际应用中，常指针和常引用往往用来作函数的形参，这样的参数称为常参数。

5.**常成员函数：** 使用 const 关键字进行说明的成员函数，称为常成员函数。只有常成员函数才有资格操作常量或常对象，没有使用 const 关键字说明的成员函数不能用来操作常对象。**常成员函数说明格式如下：**

<类型说明符> <函数名> (<参数表>) const； 其中，const 是加在函数说明后面的类型修饰符，它是函数类型的一个组成部分，因此，在函数实现部分也要带 const 关键字。

6.常数据成员 类型修饰符 const 不仅可以说明成员函数，也可以说明数据成员。

由于 const 类型对象必须被初始化，并且不能更新，因此，在类中说明了 const 数据成员时，只能通过成员初始化列表的方式来生成构造函数对数据成员初始化。

子对象和堆对象(new 和 delete 的应用)

子对象：当一个类的成员是某一个类的对象时，该对象就为子对象。子对象实际就是对象成员。在类中出现了子对象或称对象成员时，该类的构造函数要包含对子对象的初始化，通常采用成员初始化表的方法来初始化子对象。在成员初始化表中包含对子对象的初始化和对类中其他成员的初始化。

堆对象：所谓堆对象是指在程序运行过程中根据需要随时可以建立或删除的对象。这种堆对象被创建在内存一些空闲的存储单元中，这些存储单元被称为堆。它们可以被创建的堆对象占有，也可以通过删除堆对象而获得释放。

创建或删除堆对象时，需要：new delete

基类和派生类

通过继承机制，可以利用已有的数据类型来定义新的数据类型。所定义的新的数据类型不仅拥有新定义的成员，而且还同时拥有旧的成员。我们**称已存在的用来派生新类的类为基类，又称为父类。由已存在的类派生出的新类称为派生类，又称为子类。**

在 C++语言中，一个派生类可以从一个基类派生，也可以从多个基类派生。**从一个基类派生的继承称为单继承；从多个基类派生的继承称为多继承。**

派生类的定义格式： **单继承的定义格式如下：class <派生类名>:<继承方式><基类名>**

{

<派生类新定义成员>

};

其中，<派生类名>是新定义的一个类的名字，它是从<基类名>中派生的，并且按指定的<继承方式>派生的。<继承方式>常使用如下三种关键字给予表示：

public 表示公有基类；

private 表示私有基类；

protected 表示保护基类；

多继承的定义格式如下：class <派生类名>:<继承方式 1><基类名 1>,<继承方式 2><基类名 2>,…

{

<派生类新定义成员>

};多继承与单继承的区别从定义格式上看，主要是多继承的基类多于一个。

派生类的三种继承方式　公有继承(public)、私有继承(private)、保护继承(protected)是常用的三种继承方式。

1. 公有继承(public)

公有继承的特点是基类的公有成员和保护成员作为派生类的成员时，它们都保持原有的状态，而基类的私有成员仍然是私有的。

2. 私有继承(private)

私有继承的特点是基类的公有成员和保护成员都作为派生类的私有成员，并且不能被这个派生类的子类所访问。

3. 保护继承(protected)

保护继承的特点是基类的所有公有成员和保护成员都成为派生类的保护成员，并且只能被它的派生类成员函数或友元访问，基类的私有成员仍然是私有的。　　不同继承方式的基类和派生类特性

继承方式	基类特性	派生类特性
公有继承	public	public
	protected	protected
	private	不可访问
保护继承	public	protected
	protected	protected
	private	不可访问
私有继承	public	private
	protected	private
	private	不可访问

对于公有继承方式：

(1) 基类成员对其对象的可见性：公有成员可见，其他不可见。这里保护成员同于私有成员。

(2) 基类成员对派生类的可见性：公有成员和保护成员可见，而私有成员不可见。保护成员同于公有成员。

(3) 基类成员对派生类对象的可见性：公有成员可见，其他成员不可见。

所以，在公有继承时，派生类的对象可以访问基类中的公有成员；派生类的成员函数可以访问基类中的公有成员和保护成员。这里，一定要区分清楚派生类的对象和派生类中的成员函数对基类的访问是不同的。

对于私有继承方式：

(1) 基类成员对其对象的可见性：　公有成员可见，其他成员不可见。

(2) 基类成员对派生类的可见性：　公有成员和保护成员是可见的，而私有成员是不可见的。

(3) 基类成员对派生类对象的可见性：　所有成员都是不可见的。

所以，在私有继承时，基类的成员只能由直接派生类访问，而无法再往下继承。

对于保护继承方式：

这种继承方式与私有继承方式的情况相同。两者的区别仅在于对派生类的成员而言，对基类成员有不同的可见性。

上述所说的可见性也就是可访问性。关于可访问性还有另的一种说法。这种规则中，称派生类的对象对基类访问为水平访问，称派生类的派生类对基类的访问为垂直访问。

一般规则如下：

公有继承时，水平访问和垂直访问对基类中的公有成员不受限制；

私有继承时，水平访问和垂直访问对基类中的公有成员也不能访问；

保护继承时，对于垂直访问同于公有继承，对于水平访问同于私有继承。

对于基类中的私有成员，只能被基类中的成员函数和友元函数所访问，不能被其他的函数访问。

基类与派生类的关系

任何一个类都可以派生出一个新类，派生类也可以再派生出新类，因此，基类和派生类是相对而言的。基类与派生类之间的关系可以有如下几种描述：

1. 派生类是基类的具体化

类的层次通常反映了客观世界中某种真实的模型。在这种情况下，不难看出：基类是对若干个派生类的抽象，而派生类是基类的具体化。基类抽取了它的派生类的公共特征，而派生类通过增加行为将抽象类变为某种有用的类型。

2. 派生类是基类定义的延续

先定义一个抽象基类，该基类中有些操作并未实现。然后定义非抽象的派生类，实现抽象基类中定义的操作。例如，虚函数就属此类情况。这时，派生类是抽象的基类的实现，即可看成是基类定义的延续。这也是派生类的一种常用方法。

3. 派生类是基类的组合

在多继承时，一个派生类有多于一个的基类，这时派生类将是所有基类行为的组合。

派生类将其本身与基类区别开来的方法是添加数据成员和成员函数。因此，继承的机制将使得在创建新类时，只需说明新类与已有类的区别，从而大量原有的程序代码都可以复用，所以有人称类是“可复用的软件构件”。

单继承

在单继承中，每个类可以有多个派生类，但是每个派生类只能有一个基类，从而形成树形结构。

派生类的构造函数和析构函数的构造

1. 构造函数

我们已知道，派生类的对象的数据结构是由基类中说明的数据成员和派生类中说明的数据成员共同构成。将派生类的对象中由基类中说明的数据成员和操作所构成的封装体称为基类子对象，它由基类中的构造函数进行初始化。

构造函数不能够被继承，因此，派生类的构造函数必须通过调用基类的构造函数来初始化基类子对象。所以，在定义派生类的构造函数时除了对自己的数据成员进行初始化外，还必须负责调用基类构造函数使基类数据成员得以初始化。如果派生类中还有子对象时，还应包含对子对象初始化的构造函数。

派生类构造函数的一般格式如下：

<派生类名>(<派生类构造函数总参数表>):<基类构造函数>(参数表 1),<子对象名>(<参数表 2>)

{

<派生类中数据成员初始化>

};

B::B(int i, int j, int k):A(i), bb(j), bbb(k)

派生类构造函数的调用顺序如下：

· 基类的构造函数

· 子对象类的构造函数(如果有的话)

· 派生类构造函数

2. 析构函数

当对象被删除时，派生类的析构函数被执行。由于析构函数也不能被继承，因此在执行派生类的析构函数时，基类的析构函数也将被调用。执行顺序是先执行派生类的构造函数，再执行基类的析构函数，其顺序与执行构造函数时的顺序正好相反。这一点从前面讲过的例子可以看出，请读者自行分析。

3. 派生类构造函数使用中应注意的问题

(1) 派生类构造函数的定义中可以省略对基类构造函数的调用，其条件是在基类中必须有缺省的构造函数或者根本没有定义构造函数。当然，基类中没有定义构造函数，派生类根本不必负责调用基类的析构函数。

(2) 当基类的构造函数使用一个或多个参数时，则派生类必须定义构造函数，提供将参数传递给基类构造函数途径。在有的情况下，派生类构造函数的函数体可能为空，仅起到参数传递作用。如本讲第一个例子就属此种情况。

子类型化和类型适应

1. 子类型化

子类型的概念涉及到行为共享，它与继承有着密切关系。

有一个特定的类型 S，当且仅当它至少提供了类型 T 的行为，由称类型 S 是类型 T 的子类型。子类型是类型之间的一般和特殊的关系。

在继承中，公有继承可以实现子类型。

类 B 继承了类 A，并且是公有继承方式。因此，可以说类 B 是类 A 的一个子类型。类 A 还可以有其他的子类型。类 B 是类 A 的子类型，类 B 具备类 A 中的操作，或者说类 A 中的操作可被用于操作类 B 的对象。

子类型关系是不可逆的。这就是说，已知 B 是 A 的子类型，而认为 A 也是 B 的子类型是错误的，或者说，子类型关系是不对称不。

因此，可以说公有继承可以实现子类型化。

2. 类型适应

类型适应是指两种类型之间的关系。例如，B 类型适应 A 类型是指 B 类型的对象能够用于 A 类型的对象所能使用的场合。

前面讲过的派生类的对象可以用于基类对象所能使用的场合，我们说派生类适应于基类。

同样道理，派生类对象的指针和引用也适应于基类对象的指针和引用。

子类型化与类型适应是致的。A 类型是 B 类型的子类型，那么 A 类型必将适应于 B 类型。

子类型的重要性就在于减轻程序人员编写程序代码的负担。因为一个函数可以用于某类型的对象，则它也可以用于该类型的各个子类型的对象，这样就不必为处理这些子类型的对象去重载该函数。

多继承

多继承下派生类的定义格式如下：class <派生类名>:<继承方式 1><基类名 1>,<继承方式 2><基类名 2>,…

{

<派生类类体>

};

其中，<继承方式 1>,<继承方式 2>,…是三种继承方式：public、private、protected 之一。

多继承的构造函数：

多继承派生类的构造函数格式：<派生类名>(<总参数表>):<基类名 1>(<参数表 1>),<基类名 2>(<参数表 2>),…

<子对象名>(<参数表 n+1>),…

{

<派生类构造函数体>

}

其中，<总参数表>中各个参数包含了其后的各个分参数表。

多继承下派生类的构造函数与单继承下派生类构造函数相似，它必须同时负责该派生类所有基类构造函数的调用。同时，派生类的参数个数必须包含完成所有基类初始化所需的参数个数。

派生类构造函数执行顺序是先执行所胡基类的构造函数，再执行派生类本身构造函数，处于同一层次的各基类构造函数的执行顺序取决于定义派生类时所指定的各基类顺序，与派生类构造函数中所定义的成员初始化列表的各项顺序无关。也就是说，执行基类构造函数的顺序取决于定义派生类时基类的顺序。可见，派生类构造函数的成员初始化列表中各项顺序可以任意地排列。

二义性问题

一般说来，在派生类中对基类成员的访问应该是唯一的，但是，由于多继承情况下，可能造成对基类中某成员的访问出现了不唯一的情况，则称为对基类成员访问的二义性问题。

派生类 A 的两基类 B1 和 B2 中都有一个成员函数 print()。如果在派生类中访问 print()函数，到底是哪一个基类的呢？于是出现了二义性。但是在上例中解决了这个问题，其办法是通过作用域运算符::进行了限定。如果不加以限定，则会出现二义性问题。**解决的方法可用前面用过的成员名限定法来消除二义性**，

当一个派生类从多个基类派生类，而这些基类又有一个共同的基类，则对该基类中说明的成员进行访问时，也可能会出现二义性。

虚基类

在《多继承》中讲过的例子中，由类 A，类 B1 和类 B2 以及类 C 组成了类继承的层次结构。在该结构中，类 C 的对象将包含两个类 A 的子对象。由于类 A 是派生类 C 两条继承路径上的一个公共基类，那么这个公共基类将在派生类的对象中产生多个基类子对象。如果要想使这个公共基类在派生类中只产生一个基类子对象，则必须将这个基类设定为虚基类。**引进虚基类的真正目的是为了解决二义性问题。**

虚基类说明格式如下：　virtual <继承方式><基类名>其中，virtual 是虚类的关键字。虚基类的说明是用在定义派生类时，写在派生类名的后面。class B : virtual public A

虚基类的构造函数

前面讲过，为了初始化基类的子对象，派生类的构造函数要调用基类的构造函数。对于虚基类来讲，由于派生类的对象中只有一个虚基类子对象。为保证虚基类子对象只被初始化一次，这个虚基类构造函数必须只被调用一次。由于继承结构的层次可能很深，规定将在建立对象时所指定的类称为最派生类。C++规定，虚基类子对象是由最派生类的构造函数通过调用虚基类的构造函数进行初始化的。如果一个派生类有一个直接或间接的虚基类，那么派生类的构造函数的成员初始列表中必须列出对虚基类构造函数的调用。如果未被列出，则表示使用该虚基类的缺省构造函数来初始化派生类对象中的虚基类子对象。

从虚基类直接或间接继承的派生类中的构造函数的成员初始化列表中都要列出这个虚基类构造函数 的调用。但是，只有用于建立对象的那个最派生类的构造函数调用虚基类的构造函数，而该派生类的基类中所列出的对这个虚基类的构造函数调用在执行中被忽略，这样便保证了对虚基类的对象只初始化一次。

C++又规定，在一个成员初始化列表中出现对虚基类和非虚基类构造函数的调用，则虚基类的构造函数先于非虚基类的构造函数的执行。

多态性和虚函数

多态性是面向对象程序设计的重要特性之一。它与前面讲过的封装性和继承性构成了面向对象程序设计的三大特性。这三大特性是相互关联的。封装性是基础，继承性是关键，多态性是补充，而多态性又必须存在于继承的环境之中。

所谓多态性是指发出同样的消息被不同类型的对象接收时导致完全不同的行为。这里所说的消息主要是指对类的成员函数的调用，而不同的行为是指不同的实现。利用多态性，用户只需发送一般形式的消息，而将所有的实现留给接收消息的对象。对象根据所接收到的消息而做出相应的动作(即操作)。

函数重载和运算符重载是简单一类多态性。重要的多态性是建立在虚函数的概念和方法基础上的。虚函数是本章讨论的重点。此外，本章还将讨论静态联编和动态联编以及纯虚函数和抽象类等概念。

运算符重载

多态性是面向对象程序设计的重要特征之一。它与前面讲过的封装性和继承性构成了面向对象程序设计的三大特征。这三大特征是相互关联的。封装性是基础，继承性是关键，多态性是补充，而多态又必须存在于继承的环境之中。

所谓多态性是指发出同样的消息被不同类型的对象接收时导致完全不同的行为。这里所说的消息主要是指对类的成员函数的调用，而不同的行为是指不同的实现。利用多态性，用户只需发送一般形式的消息，而将所有的实现留给接收消息的对象。对象根据所接收到的消息而做出相应的动作(即操作)。

函数重载和运算符重载是简单一类多态性。函数重载的概念及用法在《函数重载》一讲中已讨论过了，这里只作简单的补充，我们重点讨论的是运算符的重载。

所谓函数重载简单地说就是赋给同一个函数名多个含义。具体地讲，C++中允许在相同的作用域内以相同的名字定义几个不同实现的函数，可以是成员函数，也可以是非成员函数。但是，定义这种重载函数时

要求函数的参数或者至少有一个类型不同，或者个数不同。而对于返回值的类型没有要求，可以相同，也可以不同。那种参数个数和类型都相同，仅仅返回值不同的重载函数是非法的。因为编译程序在选择相同名字的重载函数时仅考虑函数表，这就是说要靠函数的参数表中，参数个数或参数类型的差异进行选择。

由此可以看出，重载函数的意义在于它可以用相同的名字访问一组相互关联的函数，由编译程序来进行选择，因而这将有助于解决程序复杂性问题。如：在定义类时，构造函数重载给初始化带来了多种方式，为用户提供更大的灵活性。

下面我们重点讨论运算符重载。

运算符重载就是赋予已有的运算符多重含义。运算符重载的几个问题

1. 运算符重载的作用是什么？

它允许你为类的用户提供一个直觉的接口。

运算符重载允许 C/C++的运算符在用户定义类型(类)上拥有一个用户定义的意义。重载的运算符是函数调用的语法修饰：

2. 算符重载的好处是什么？

通过重载类上的标准算符，你可以发掘类的用户的直觉。使得用户程序所用的语言是面向问题的，而不是面向机器的。

最终目标是降低学习曲线并减少错误率。

3. 哪些运算符可以用作重载？

几乎所有的运算符都可用作重载。具体包含：

算术运算符：+,-,*,/,%,++,--;

位操作运算符：&,|,~,^,<<,>>;

逻辑运算符：!,&&,||;

比较运算符：<,>,>=,<=,==,!=;

赋值运算符：=,+=,-=,*=,/=,%=,&=,|=,^=,<<=,>>=;

其他运算符：[],(),->,,(逗号运算符),new,delete,new[],delete[],->*。

下列运算符不允许重载：

.,.*,::,?:

4. 运算符重载后，优先级和结合性怎么办？

用户重载新定义运算符，不改变原运算符的优先级和结合性。这就是说，对运算符重载不改变运算符的优先级和结合性，并且运算符重载后，也不改变运算符的语法结构，即单目运算符只能重载为单目运算符，双目运算符只能重载双目运算符。

5. 编译程序如何选用哪一个运算符函数？

运算符重载实际是一个函数，所以运算符的重载实际上是函数的重载。编译程序对运算符重载的选择，遵循着函数重载的选择原则。当遇到不很明显的运算时，编译程序将去寻找参数相匹配的运算符函数。

6. 重载运算符有哪些限制？

(1) 不可臆造新的运算符。必须把重载运算符限制在 C++语言中已有的运算符范围内的允许重载的运算符之中。

(2) 重载运算符坚持 4 个“不能改变”。

·不能改变运算符操作数的个数；

·不能改变运算符原有的优先级；

·不能改变运算符原有的结合性；

·不能改变运算符原有的语法结构。

7. 运算符重载时必须遵循哪些原则？

运算符重载可以使程序更加简洁，使表达式更加直观，增加可读性。但是，运算符重载使用不宜过多，否则会带来一定的麻烦。

使用重载运算符时应遵循如下原则：

(1) 重载运算符含义必须清楚。

(2) 重载运算符不能有二义性。

运算符重载函数的两种形式

运算符重载的函数一般地采用如下两种形式：成员函数形式和友元函数形式。这两种形式都可访问类中的私有成员。

1. 重载为类的成员函数

将运算符重载函数说明为类的成员函数格式如下：　　<类名> operator <运算符>(<参数表>)

其中，operator 是定义运算符重载函数的关键字。

2. 重载为友元函数

运算符重载函数还可以为友元函数。当重载友元函数时，将没有隐含的参数 this 指针。这样，对双目运算符，友元函数有 2 个参数，对单目运算符，友元函数有一个参数。但是，有些运行符不能重载为友元函数，它们是：=,(),[]和->。

重载为友元函数的运算符重载函数的定义格式如下：friend <类型说明符> operator <运算符>(<参数表>){……}

3. 两种重载形式的比较

一般说来，单目运算符最好被重载为成员；对双目运算符最好被重载为友元函数，双目运算符重载为友元函数比重载为成员函数更方便此，但是，有的双目运算符还是重载为成员函数为好，例如，赋值运算符。因为，它如果被重载为友元函数，将会出现与赋值语义不一致的地方。

其他运算符的重载举例

1).下标运算符重载

由于 C 语言的数组中并没有保存其大小，因此，不能对数组元素进行存取范围的检查，无法保证给数组动态赋值不会越界。利用 C++的类可以定义一种更安全、功能强的数组类型。为此，为该类定义重载运算符[]。

在重载下标运算符函数时应该注意：

(1) 该函数只能带一个参数，不可带多个参数。

(2) 不得重载为友元函数，必须是非 static 类的成员函数。

2). 重载增 1 减 1 运算符

增 1 减 1 运算符是单目运算符。它们又有前缀和后缀运算两种。为了区分这两种运算，将后缀运算视为又目运算符。表达式　obj++或 obj--被看作为：　　obj++0 或 obj--0

3). 重载函数调用运算符

可以将函数调用运算符()看成是下标运算[]的扩展。函数调用运算符可以带 0 个至多个参数。

静态联编和动态联编

联编是指一个计算机程序自身彼此关联的过程。按照联编所进行的阶段不同，可分为两种不同的联编方法：静态联编和动态联编。

静态联编： 静态联编是指联编工作出现在编译连接阶段，这种联编又称早期联编，因为这种联编过程是在程序开始运行之前完成的。

在编译时所进行的这种联编又称静态束定。在编译时就解决了程序中的操作调用与执行该操作代码间的关系，确定这种关系又称为束定，在编译时束定又称静态束定。

动态联编：从对静态联编的上述分析中可以知道，编译程序在编译阶段并不能确切知道将要调用的函数，只有在程序执行时才能确定将要调用的函数，为此要确切知道该调用的函数，要求联编工作要在程序运行时进行，这种在程序运行时进行联编工作被称为动态联编，或称动态束定，又叫晚期联编。

动态联编实际上是进行动态识别。在上例中，前面分析过了静态联编时，fun()函数中 s 所引用的对象被束定到 Point 类上。而在运行时进行动态联编将把 s 的对象引用束定到 Rectangle 类上。可见，同一个对象引用 s，在不同阶段被束定的类对象将是不同的。那么如何来确定是静态联编还是动态联编呢？C++规定动态联编是在虚函数的支持下实现的。

从上述分析可以看出静态联编和动态联编也都是属于多态性的，它们是不同阶段对不同实现进行不同的选择。上例中，实现上是对 fun()函数参数的多态性的选择。该函数的参数是一个类的对象引用，静态联编和动态联编和动态联编实际上是在选择它的静态类型和动态类型。联编是对这个引用的多态性的选择。

虚函数

虚函数是动态联编的基础。虚函数是成员函数，而且是非 static 的成员函数。说明虚函数的方法如下：
virtual <类型说明符><函数名>(<参数表>)

其中，被关键字 virtual 说明的函数称为虚函数。

如果某类中的一个成员函数被说明为虚函数，这就意味着该成员函数在派生类中可能有不同的实现。当使用这个成员函数操作指针或引用所标识对象时，对该成员函数调用采取动态联编方式，即在运行时进行关联或束定。

动态联编只能通过指针或引用标识对象来操作虚函数。如果采用一般类型的标识对象来操作虚函数，则将采用静态联编方式调用虚函数。

派生类中对基类的虚函数进行替换时，要求派生类中说明的虚函数与基类中的被替换的虚函数之间满足如下条件：

(1) 与基类的虚函数有相同的参数个数；

(2) 其参数的类型与基类的虚函数的对应参数类型相同；

(3) 其返回值或者与基类虚函数的相同，或者都返回指针或引用，并且派生类虚函数所返回的指针或引用的基类型是基类中被替换的虚函数所返回的指针或引用的基类型的子类型。

满足上述条件的派生类的成员函数，自然是虚函数，可以不必加 virtaul 说明。

总结动态联编的实现需要如下三个条件：

(1) 要有说明的虚函数；

(2) 调用虚函数操作的是指向对象的指针或者对象引用；或者是由成员函数调用虚函数；

(3) 子类型关系的建立。

构造函数中调用虚函数时，采用静态联编即构造函数调用的虚函数是自己类中实现的虚函数，如果自己类中没有实现这个虚函数，则调用基类中的虚函数，而不是任何淫生类中实现的虚函数。

关于析构函数中调用虚函数同构造函数一样，即析构函数所调用的虚函数是自身类的或者基类中实现的虚函数。

一般要求基类中说明了虚函数后，派生类说明的虚函数应该与基类中虚函数的参数个数相等，对应参数的类型相同，如果不相同，则将派生类虚函数的参数的类型强制转换为基类中虚函数的参数类型。

纯虚函数和抽象类

纯虚函数是一种特殊的虚函数，它的一般格式如下：

```
class <类名>
{
  virtual <类型><函数名>(<参数表>)=0;
  …
};
```

在许多情况下，在基类中不能对虚函数给出有意义有实现，而把它说明为纯虚函数，它的实现留给该基类的派生类去做。这就是纯虚函数的作用。下面给出一个纯虚函数的例子。

抽象类

带有纯虚函数的类称为抽象类。抽象类是一种特殊的类，它是为了抽象和设计的目的而建立的，它处于继承层次结构的较上层。抽象类是不能定义对象的，在实际中为了强调一个类是抽象类，可将该类的构造函数说明为保护的访问控制权限。

抽象类的主要作用是将有关的组织在一个继承层次结构中，由它来为它们提供一个公共的根，相关的子类是从这个根派生出来的。

抽象类刻画了一组子类的操作接口的通用语义，这些语义也传给子类。一般而言，抽象类只描述这组子类共同的操作接口，而完整的实现留给子类。

抽象类只能作为基类来使用，其纯虚函数的实现由派生类给出。如果派生类没有重新定义纯虚函数，而派生类只是继承基类的纯虚函数，则这个派生类仍然还是一个抽象类。如果派生类中给出了基类纯虚函数的实现，则该派生类就不再是抽象类了，它是一个可以建立对象的具体类了。

虚析构函数

在析构函数前面加上关键字 virtual 进行说明，称该析构函数为虚析构函数。virtual ~B();

如果一个基类的析构函数被说明为虚析构函数，则它的派生类中的析构函数也是虚析构函数，不管它是否使用了关键字 virtual 进行说明。

说明虚析构函数的目的在于在使用 delete 运算符删除一个对象时，能保析构函数被正确地执行。因为设置虚析构函数后，可以采用动态联编方式选择析构函数。