鸡啄米的这套 C++编程入门教程到上一讲--C++编程入门系列之五十（异常处理）为止，就算圆满完成了。大家学到这里应该对 C++的基础知识和程序设计都有了一定的认识了，以后要做的就是多练习多积累经验了。

一.目录

之前鸡啄米都是一节一节的讲，也没有明确给出本套教程的组织结构，大家只能一节一节的跟着学。此教程共有五十二讲：前言、五十讲入门知识和这一节的总结。下面鸡啄米就概括出这套 C++编程入门教程的目录：

鸡啄米：C++编程入门系列之二十四（C++程序设计必知：多文件结构和编译预处理命令）

第六部分：数组、指针和字符串

鸡啄米：C++编程入门系列之二十五（数组、指针和字符串：数组的声明和使用）

鸡啄米：C++编程入门系列之二十六（数组、指针和字符串：数组的存储与初始化、对象数组、数组作为函数参数）

鸡啄米：C++编程入门系列之二十七（数组、指针和字符串：指针变量的声明、地址相关运算--“*”和“&”）

鸡啄米：C++编程入门系列之二十八（数组、指针和字符串：指针的赋值和指针运算）

鸡啄米：C++编程入门系列之二十九（数组、指针和字符串：指向数组元素的指针和指针数组）

鸡啄米：C++编程入门系列之三十（数组、指针和字符串：指针用作函数参数、指针型函数和函数指针）

鸡啄米：C++编程入门系列之三十一（数组、指针和字符串：对象指针）

鸡啄米：C++编程入门系列之三十二（数组、指针和字符串：动态内存分配和释放）

鸡啄米：C++编程入门系列之三十三（数组、指针和字符串：用字符数组存放和处理字符串）

鸡啄米：C++编程入门系列之三十四（数组、指针和字符串：string 类）

第七部分：继承与派生

鸡啄米：C++编程入门系列之三十五（继承与派生：概念介绍与派生类的声明）

鸡啄米：C++编程入门系列之三十六（继承与派生：派生类从基类继承的过程）

鸡啄米：C++编程入门系列之三十七（继承与派生：派生类对基类成员的访问控制之公有继承）

鸡啄米：C++编程入门系列之三十八（继承与派生：派生类对基类成员的访问控制之保护继承与私有继承）

鸡啄米：C++编程入门系列之三十九（继承与派生：派生类的构造函数）

鸡啄米：C++编程入门系列之四十（继承与派生：派生类的析构函数）

鸡啄米：C++编程入门系列之四十一（继承与派生：作用域分辨符）

鸡啄米：C++编程入门系列之四十二（继承与派生：虚基类及其派生类的构造函数）

鸡啄米：C++编程入门系列之四十三（继承与派生：赋值兼容规则）

第八部分：多态性

鸡啄米：C++编程入门系列之四十四（多态性：多态的概念和类型）

鸡啄米：C++编程入门系列之四十五（多态性：运算符重载的概念和规则）

鸡啄米：C++编程入门系列之四十六（多态性：运算符重载为类的成员函数）

鸡啄米：C++编程入门系列之四十七（多态性：运算符重载为类的友元函数）

鸡啄米：C++编程入门系列之四十八（多态性：虚函数）

鸡啄米：C++编程入门系列之四十九（多态性：纯虚函数和抽象类）

第九部分：异常处理

鸡啄米：C++编程入门系列之五十（异常处理）

二.总结

平时写程序总会用到 C++语言的各种基础知识，但从来没有这样根据以前自己的学习，将 C++语法知识和编程方法及思想等总结之后写下来。

在工作之余写下这套教程，不但让我自己对 C++基础知识掌握的更牢固，对 C++编程思想有了更深入的认识，而且更重要的是能与很多想步入 C++编程殿堂的朋友分享这些知识。

大家学完这套教程以后可能还是感觉不太自信，认为自己还是写不了程序。没关系，看看别人写的程序，可以是稍微复杂点的，自己稍加修改再测试运行看看效果，经过这样的练习以后就可以自己试着设计并编写程序了。谁都是这时候过来的，要有耐心和勇于尝试。

记住，学 C++就是在练内功，练好了内功再就是招式上的学习和熟悉了。学好了 C++，可以做 VC 界面开发，也可以做 Linux 下的嵌入式开发，又或者是网络方面等等，但 C++是基础（当然也不是只能用 C++语言进行这些开发）。

建议大家可以再学学 MFC（微软基础类库），做一些简单的界面，这样能清楚的看到 C++能够实现的功能，增加自己的成就感，还能学习微软的代码风格和设计方式。

完成了此套 C++编程入门教程以后，鸡啄米认为还有很多知识有必要跟大家分享，有时间我还会为大家带来 MFC、网络编程、数据库等方面的入门知识。希望大家继续关注鸡啄米博客，继续支持鸡啄米，谢谢大家！

第一部分：C++编程概述

鸡啄米：C++编程入门系列之前言

从今天开始鸡啄米将给大家讲解一些 C++编程入门的知识，对于鸡啄米也是个温习巩固的过程。鸡啄米将会用深入浅出的方式，尽量用最简单的语言让读者学会 C++语言，并爱上这门语言。

本前言讲述的是编程语言的发展过程、面向对象方法、面向对象的基本概念等内容。

语言大家都知道，计算机语言跟人类语言一样有语法等规则，它是计算机可以识别的语言，用来描述解决问题的方法，计算机阅读了它就可以做相应的工作。利用语言描述解决问题的方法就生成了程序，程序是由很多指令组成的，计算机所能识别的指令是 0 和 1 的组合，所有这种指令组成的语言叫做机器语言。可以想象，全是 1 和 0，对于软件开发者来说编程是很难的，开发周期长，功能也做不复杂。后来出现了汇编语言，它就是把机器语言映射成一些人能读懂的助记符，这样就使得软件开发容易点了，但是还是与人的思维差别很大，但这是一个跳跃式的进步。最后出现了高级语言，抽象层次比较高了，程序中的数据命名都能很容易看出数据的含义，而且执行语句也很容易理解。 20 世纪 60 年代出现的结构

化编程语言更是进了一大步，但是程序中的数据和操作分离，各自独立，不能跟现实中的事物对应起来。最后出现了面向对象编程语言。

面向对象编程语言把事物看成是具有属性和行为的对象，然后通过抽象找出属于同一类的事物的属性和行为，就形成了类。通过类的继承和多态可以很好的实现代码复用，提高软件开发效率。如果这块不懂的话可以先尽量理解，通过后面的学习会慢慢理解这些概念。

20 世纪 80 年代比较风行结构化程序设计方法，它的设计思路是，自上而下，逐步细化，将程序结构按功能分成多个模块，每个模块可能又会分成多个模块，这样就形成了一个模块的树状结构，各个模块间的关系尽可能简单，功能上相对独立。每个模块都是由顺序、选择和循环三种基本结构组成。这种方法就将一个复杂的程序设计问题分成很多简单细化的子问题，更便于开发维护。但是这种方法的缺点是数据和处理数据的方法各自独立，如果数据结构改变，所有相关的方法都要改变。这时就出现了面向对象设计方法，它将数据和方法放到一个整体里，这个整体就叫对象，同类型对象抽象成类，类中大部分数据只能用本类中的方法处理，类通过简单的外部接口与外界发生关系，而内部的各种关系对外部是透明的。

下面介绍面向对象的几个基本概念：

1.对象。每个对象都是描述客观存在事物的一个实体，都是由数据和方法（也可以叫属性和行为）构成。属性是描述事物特征的数据，行为描述对对象属性的一些操作。

2.类。类是具有相同属性和行为的一些对象的集合，它为所有属于这个类的对象提供抽象的描述，比如麻雀和杜鹃都可以看作是对象，而鸟就可以看作类。

3.封装。封装就是把对象的所有属性和行为结合成一个独立的单位，对外隐藏对象的内部细节，只保留有几个接口与外界联系。

4.继承。一个类（叫做子类）可以通过继承另一个类（叫做父类）来拥有另一个类的所有属性和行为。比如车和汽车，汽车从车继承，车的所有属性和行为都继承到了汽车上。

5.多态性。多态性就是说父类中的属性和行为被子类继承后，子类可以有自己不同于父类的属性或行为。比如定义一个类“动物”，它具有“吃”这个行为，但是具体怎么吃吃什么，不知道，因为不知道到底是个什么“动物”，如果从这个类继承出子类“羊”和“老虎”，“吃”就成了具体的行为，怎么吃吃什么就都知道了。

最后鸡啄米再次声明，本前言内容可能对于新手来说讲的还是有点难懂，但是不要担心，通过后续的学习肯定会慢慢领会这些的，到那时你再回来看这些就会觉得很简单了。

鸡啄米：C++编程入门系列之一（进制数）

鸡啄米在前言已经讲了程序的发展历程，程序设计的进步和编程入门的一些基本概念。这一节主要讲数据在计算机中的存储结构。计算机执行程序需要控制信息和数据信息，控制信息涉及硬件方面，鸡啄米主要讲数据信息的存储。这部分内容可能有点郁闷，但是这是基础之基础，在编程入门时还是掌握下吧。

大家应该知道比较常用的数制是十进制、二进制、八进制和十六进制。二进制就是逢二进一，每位都是小于二的数，其他进制类推。计算机存储数据使用的是二进制编码。

对于一个 R 进制的数 X 来说，其值可以通过下面的公式算出来：

$V(X)=\sum_{i=0}^{n-1} X_i R^i + \sum_{i=-1}^{-m} X_i R^i$，前面是整数部分，后面是小数部分，m，n 为正整数，表示第 i 位上的数字乘以进制数的 i 次方。比如二进制数$(11.01)_2=1\times2^1+1\times2^0+0\times2^{-1}+1\times2^{-2}=3.25$。当 R 进制转换到十进制时，就这样转换。

那十进制怎么转换为 R 进制呢，可以将十进制数连续除以 R，其每个余数就是 R 进制数各个数位上的数字，最后的余数是最高位。比如将 68 转换成二进制数，用除 2 取余法：

2∟68　　余数
2∟34 .....0　　低位
2∟17 0
2∟8 1
2∟4 0
2∟2 0
2∟1 0
0 .....1　　高位

结果就是 $68_{10}=1000100_2$，将 68 转换为八进制数就用除 8 取余法。

十进制小数转换成 R 进制小数，整数部分仍按上述方法转换，小数部分则乘以 R，然后将结果的整数位取出作为转换最终结果的一位，再用去掉整数位的小数再乘以 R，之后循环这样执行，直到小数部分为 0 或者达到所要求的精度为止，取出的这些整数位第一位为最高位。例如，将十进制小数 0.3125 转换成二进制小数：

0.3125×2 = 0.625　　取出 0，为最高位
0.625×2 = 1.25　　取出 1，剩下的小数为 0.25
0.25×2 = 0.5　　取出 0
0.5×2 = 1.0　　取出 1，小数位为 0，不再继续

结果就是 $0.3125_{10}=0.0101_2$。所以将十进制数 68.3125 转换成二进制数就是 1000100.0101。

二进制和八进制十六进制之间怎样转换呢？每位八进制数相当于三位二进制数，每位十六进制数相当于四位二进制数，举个例子说明下就清楚了：

$(1000100)_2=(1\ \ 000\ \ 100)_2=(104)_8$
$(1000100)_2=(100\ \ 0100)_2=(44)_{16}$

上面这个等式反过来当然也成立，就是八进制十六进制转换为二进制反过来计算就可以了。

下面鸡啄米再跟大家说下计算机数据的存储单位，常用的单位有“位”，“字节”，“字”。

位 bit：这是数据的最小单位，表示一位二进制数据。

字节 byte：由八位二进制数据组成。字节是数据存储中最常用的基本单位。我们常说内存有 2G，这里的单位就是字节。1K 等于 1024 字节，1M 等于 1024K，1G 等于 1024M。

字 word：位的组合，作为一个独立的信息单位处理。取决于机器的类型、字长及使用者的要求。常用的固定字长有 8 位、16 位、32 位等。

机器字长：讨论信息单位时，有个与机器硬件指标有关的单位就是机器字长，一般指参加运算的寄存器所含有的二进制数的位数，它代表了机器的精度，如 32 位、64 位等。就是我们常说的 32 位机器还是 64 位机器。

这一节鸡啄米就讲到这里了，因为确实是比较死的东西，大家大部分需要记住就行了。如果还有什么不明白的，欢迎大家到鸡啄米博客交流学习。谢谢大家！

鸡啄米：C++编程入门系列之二（原码、反码与补码）

上一节进制数中鸡啄米讲了二进制、八进制、十进制和十六进制数的表示方法和相互转换关系。这节主要讲解二进制的几种编码表示方法。

计算机存储数据信息都是以二进制编码存储的，机器内存储的数据的表达形式称为“机器数”，而它代表的数制称为这个机器数的“真值”。数有正负之分，那么在计算机里怎么表示正负呢，0 和 1 不就刚好吗？呵呵，没错，就是用“0”表示正号，“1”表示负号，符号位放在数的最高位。例如，二进制数 X=(+1010100)，Y=(-1010100)，则他们在机器中就存为

X：

0	1	0	1	0	1	0	0

Y：

1	1	0	1	0	1	0	0

最左边那一位是符号位，跟它后面的数字一起组成一个数。

为了让计算机计算起来更简单，硬件设计起来也比较简单，人们研究了多种二进制编码方法。其实就是对负数的不同编码，正数基本不变。

1.原码

刚才鸡啄米讲到的最高位是符号位，后面是绝对值来表示一个数，这种编码叫做“原码”。但是有个问题，就是 0 的表示不唯一，+0 就是 000...0，-0 就是 1000...0。另外，进行四则运算时，对于符号位都要单独处理，同号怎样运算，异号又怎样运算，有时还需要借位，这对计算机来说是很麻烦的，所以必须找更好的编码方法。

2.反码

其实反码不怎么用，但是怎么说是一种编码方式，而且是求补码的中间码，我们还是需要学一下的。

正数的反码跟原码一样，负数的反码的符号位跟原码一样，但其余各位取反，也就是 0 变 1,1 变 0。例如，二进制数+1100111 的原码是 01100111，反码也是 01100111，-110 0111 的原码是 11100111，反码则是 10011000。

3.补码

大家想下，如果现在是 7 点，但是鸡啄米的表时间是 9 点，那怎样才能把它调准呢？鸡啄米可以往后调 2 个小时，也可以往前调 10 个小时，结果都是 9 点。这里就涉及到取模运算（以前我们叫求余运算），9-2 = 7，(9+10)%12 = 7，这里的%就是 C++里的取模运算符。2 与 10 对模数 12 是互为补数的。补码就是利用的这个原理，利用补码可以把减法运算变成两个补码相加，具体就是将其变为一个正数和一个负数的加法运算，然后计算这个正数和负数的补码，两个补码相加。因为补码的符号位作为数值参与运算，所以就不存在符号位单独处理的问题。

正数的原码、反码和补码是一模一样的。

负数的补码是其反码的最末位加 1 得到的，我们经常顺口记为负数的补码就是取反加 1。

必须要指出的是补码运算的结果也是补码，还要把它算到原码才得到最后结果。那么知道补码怎么计算原码，很简单，就是对补码再求补码就是源码了。

鸡啄米给大家个例子：

10-67 = ？，10 的原码是 00001010，补码跟原码一样，-67 的原码是 11000011，补码是 10111101。两个补码相加是 11000111，这是结果的补码，求原码要对这个补码再求补码，取反加 1 后就是 10111001。

最后鸡啄米再跟大家说下浮点数的存储方式。浮点数可以理解为小数。浮点数 N 的科学表示法为：。E 表示 2 的幂，叫做 N 的阶码，它的位数反应了此浮点数的范围。M 表示 N 的全部有效数字，叫做 N 的尾数，它的位数反应了浮点数的精度。

浮点数的存储格式随着机器的不同而不同，比如，如果机器是 16 位机，其二进制浮点数的组成为 4 位阶码，12 位尾数，存储格式如下：

15 14 12 11 10 0

阶码符号	阶码	尾数符号	尾数

这节的内容也是比较基础的内容，但是当你学会编程后可能用的很少，到了一定程度才会用到，鸡啄米希望到那时你再回来温习温习啊。

第二部分：C++简单程序设计

鸡啄米：C++编程入门系列之三（VS2010 的使用介绍）

大家好，鸡啄米上节课给大家讲了原码、反码和补码的知识点。这节课我主要跟大家讲解 VS2010 的使用方法，并不会面面俱到，我只大概讲下经常会用到的功能。至于其他功能用的不会很多，读者完全可以自己摸索，或者查相关资料，不难的。

在讲 VS2010 之前鸡啄米先跟大家讲下程序开发过程中的几个基本概念：源程序、目标程序和翻译程序。

源程序就是我们用各种语言写的程序，我们常说写代码其实就是在写源程序。源程序可以是汇编语言或者高级语言等其他语言所写。

目标程序是经过翻译之后生成的程序，可以是机器语言表示的，也可以是汇编语言或其他中间语言表示的。我们写**程序编译之后的 obj 文件就是目标程序**。目标程序理论说应该能运行，但是程序运行离不开操作系统的支持，它必须经过装载和链接，生成可执行程序（exe 文件）才能运行。

翻译程序就是把源程序翻译成目标程序的程序。它有三种类型：汇编程序、编译程序和解释程序。

汇编程序是把用汇编语言程序翻译成目标程序的程序。编译程序是把高级语言程序翻译成目标程序的程序，C++是高级语言，当然需要编译程序，我们要用的就是 VS2010 平台。解释程序也是将高级语言翻译成目标程序，但它是输入一句，翻译一句，执行一句，直到整个源程序被翻译执行完毕，效率比较低。

下面鸡啄米开始简单给大家介绍 VS2010 的使用。鸡啄米曾经用过的编译器有 VC6.0，VS2005，VS2010，它们中间还有个 VS2008，但鸡啄米没用过。以前 VC6.0 很火的，现在估计还有很多人在用，但是它已经过时了，界面开发方面已经远没有后面的版本好，有些新的 C++标准支持的不是很好。现在最新的版本就是 VS2010 了，所以鸡啄米推荐大家使用。VS2010 可以编译 C、C++、C#等语言。

大家可以到网上搜下 VS2010 英文版下载，可以选择完全安装也可以自己选择要安装的项目，不明白的话选完全安装就可以。VS2010 占用硬盘空间是很大的，安装文件就有 2

G 多安装完大概 3G 多吧。具体安装步骤就不介绍了，相信大家都会的，记得安装 msdn 哦，就是 VS2010 的帮助文件，大家以后做开发少不了用 msdn 的。

安装完第一次打开使用会出现一个对话框，我们选择 Visual C++那一项就可以了。下图是 VS2010 打开后的画面，当然可能你的界面跟我的不一样，没关系，那只是设置问题。

菜单栏我们常用的是 File，View 和 Debug 菜单下的菜单项。工具栏各项其实在菜单栏都有与它们对应的菜单项，功能是一样的。左侧面板可以放多个视图，我这里放了解决方案浏览器、类视图和资源视图，解决方案浏览器中我们可以看到所有头文件和源文件构成的树，头文件就是.h 后缀的文件，源文件就是.cpp 后缀的文件，类视图中显示了每个工程的每个类，也是树状结构，在解决方案浏览器或类视图中双击每一项，中间区域都会打开相应的文件或者到类的位置。资源视图中显示了界面工程中使用的一些界面资源。中间区域默认是打开 start page，打开一个工程后我们在左侧面板上双击某项时，中间区域会出现相应的界面。右侧面板上有个 Toolbox，是在界面工程中用来往界面资源模板上添加控件的，里面包含了各种控件，直接拖到模板上就行了。底侧还有几个标签，其中一个是 output，这是输出视图，用来输出程序运行信息和我们程序中写的一些调试信息，还有一个 Find Result 视图，用来显示我们搜索任何字母或单词的结果。

File->New->Project 项可以创建一个工程，也可以在 start page 中点击 New Project...，然后出现的 New Project 窗口中有多个工程类型可以选择，在这里鸡啄米给大家讲几个，Win32 Console Application 是 Win32 控制台程序，MFC Application 是 MFC 应用程序，可以建立以开发界面程序，Win32 Project 是 Win32 程序，Empty Project 是空程序，MF

C DLL 用来建立生成动态链接库的工程，其它的不常用就不介绍了。我们要建立一个没有应用程序界面的只有 Dos 命令行界面的工程就选 Win32 Console Application 项，我们学习 C++基础知识用这个就可以了。它跟 Win32 Project 的区别是，它的入口是 main 函数，而 Win32 Project 的入口是 WinMain 函数。

File->Open->Project/Solution 可以打开以前建立的工程或者解决方案，一个解决方案中可以包含多个工程，你可以把它理解为多个有关系或者没关系的工程的集合，有时把多个工程放到一个解决方案里调试起来很方便。

上面鸡啄米跟大家说了，你的界面跟我的不一样是因为设置不一样，在 View 下的菜单项我们就可以控制它。View->Solution Explorer 可以打开或关闭解决方案浏览器视图，View->Class View 用来控制类视图的显示，View->Resource View 控制资源视图的显示，当然只有界面工程的资源视图中才会有内容。View->Output 和 View->FindResults 等项大家应该也知道它们的含义了吧。

Debug->Start Debugging（快捷键是 F5）用来开始调试，Debug->Toggle Breakpoint（快捷键是 F9）用来设置断点，设置断点后然后开始调试，程序运行到断点时会停下来，我们用鼠标放到断点处的各个变量上时可以看到当时这些变量的值。Debug->Start Without Debugging 表示不调试直接运行，也就是到断点处也不会停止运行。Debug->Step Into（快捷键是 F11）用来单步执行，遇到子函数就进入并且继续单步执行；Debug->Step Over（快捷键是 F10）也是单步执行，但在单步执行时，在函数内遇到子函数时不会进入子函数内单步执行，而是将子函数整个执行完而跳到下一步，也就是把子函数整个作为一步。

工具栏上有个下拉列表框，它包含有 Debug、Release 和 Configuration Manager，选择 Debug 时为调试模式，生成的可执行程序中包含调试信息，我们可以调试并清楚的看到变量值，选择 Release 时生成的可执行程序中不含调试信息，在设置断点后看到的变量值可能不准确。

msdn 帮助可以点 Help->View Help（快捷键是 F1），也可以点开始->Microsoft Visual Studio 2010->Microsoft Visual Studio 2010 Documentation 打开。

鸡啄米讲的使用方法有点琐碎，有的可能现在只能靠想象一下，只有在实践了之后才会清楚，没关系，能接收多少接收多少，自己可以先熟悉下 VS2010 的界面，找下感觉，以后会慢慢熟悉的。下节课鸡啄米就开始正式讲解 C++的语法知识了，欢迎关注哦！

鸡啄米：C++编程入门系列之四（数据类型）

上一讲鸡啄米给大家讲了 VS2010 的使用介绍，大家以后就可以用它来进行软件开发了，在编程入门的时候可以写些简单的程序先玩玩，实际编写代码试验下鸡啄米讲过的内容。

C++是从 C 中发展来的，可以兼容 C。所以 C++的一些基本语法跟 C 几乎是一样的。

先给大家看下一个简单的 C++程序。

```
#include  <iostream.h>

void  main(void)

{

      Std::cout  <<  "Hello!  \n";

}
```

#include <iostream.h>告诉编译器对程序进行预处理时，就是在编译前的一个阶段，将文件 iostream.h 的代码嵌入到这条指令所在的位置。也就是对编译器来讲这句话相当于它所包含的全部代码。**#include 叫做编译指令，用来包含一些文件。**这里包含的 iostream.h 是 C++标准库里的文件，大家可以直接用，它声明了输入输出的有关信息。这类文件常放在程序开始的地方，所以叫做头文件。

main 是主函数名，函数体用一对大括号括住。**函数是 C++程序里最小的功能单位。**C++程序里必须有且只能有一个 main 函数，它是程序执行的入口，就是从它开始执行。main 函数前的 void 表示此函数没有返回值，后面的 void 表示没有参数。cout<<"Hello! \n";是条语句，每条语句由分号结束。cout 是 C++标准库里预定义的对象，它的信息就放在之前包含的头文件 iostream.h 中，它是一个输出流对象，用操作符<<将后面的字符串输出到标准输出设备，一般是显示器。

我们写完这个程序后存成后缀名为.cpp 的文件，叫做 C++源文件，经过编译链接后生成 exe 可执行文件。此程序运行时会在屏幕上输出：Hello!（后面还会有个换行）。

下面鸡啄米先讲下 C++语言所用的字符集，就是写 C++程序时会用到的一些字符。C++语言的字符集由下述字符构成：1.英文字母：A~Z，a~z；2.数字字符：0~9；3.特殊字符：空格 ！# % ^ & * _ + = - ~ < > / \ ' " ； . ，() [] {}。

然后鸡啄米再给大家讲下 C++语言的词法记号，就是编程中用的关键字、变量名、运算符等等。这里介绍下 C++语言的关键字、标识符、文字、运算符、分隔符和空白。

关键字：关键字是 C++预定义的一些单词，我们定义变量常量时是不能使用的，它们有不同的用处，后面大家会看到。给大家列一下吧：auto bool break case catch char class const const_case continue default delete do double dynamic_cast else enum explicit extern false float for friend goto if inline int long mutable nam espace new operate private protected public register reinterpret_cast return sh

ort signed sizeof static static_cast struct switch template this throw true try typedef typeid typename union unsigned using virtual void volatile while。

标识符：标识符是我们软件开发者自己声明的单词，用来命名一些实体，比如：函数名、变量名、类名、对象名等。它的构成规则：1.以大写字母、小写字母或下划线开头；2.可由大写字母、小写字母、下划线或数字组成；3.字母区分大小写，大写字母和小写字母表示不同的标识符；4.不能用 C++关键字。(不能以数字开头)

文字：指在程序中用符号表示的数据，包括数字、字符、字符串和布尔文字（true 和 false）。后面会具体讲各种文字。

运算符：用来进行运算的符号，比如：+，-，*，/等等。

分隔符：起分隔作用，用来分隔词法记号或程序正文，分隔符有：() {} ，：；。 这些分隔符不进行实际的操作，只是用来构造程序，用法后面也会介绍。

空白：编译分析代码的时候会将代码分成词法记号和空白，空白包括空格、制表符（TAB 键产生的字符）、换行符（回车键产生的字符）和注释。空白用来表示词法记号的开始和结束位置，其余的空格将被编译器忽略。比如 int i;跟 int i;是等价的。注释是对代码进行必要的注解和说明，编译时不会理会注释部分，C++语言有两种注释方法：1.使用“/*”和“*/”括起注释文字，例如，/*this is a comment*/；2.使用“//”，从“//”开始直到它所在行的行尾，所有的字符都被作为注释处理，比如，// this is a comment。

鸡啄米再给大家讲 C++的各个数据类型。

1.基本数据类型。

C++的基本数据类型有 bool（布尔型）、char（字符型）、int（整型）、float（浮点型，表示实数）、double（双精度浮点型）。除了 bool 型外，有两大类：整数和浮点数。因为 char 型本质上就是整型，只不过是一个字节的整数，用来存放字符的 ASCII 码。还有几个关键字 signed 和 unsigned、short 和 long 起修饰作用。

short 修饰 int 时，short int 表示短整型，占 2 个字节，这里 int 可以省略，也就是说 short 就是 short int。long 可以修饰 int 和 double，long int 表示长整型，占 4 个字节，同样 int 也可以省略。**int 在不同的编译环境中可能占的字节数不一样，大多数环境中占用 4 个字节。short 类型固定占 2 个字节，long 类型固定占 4 个字节。**

signed 和 unsigned 可以用来修饰 char 型、int 型、short 型和 long 型。signed 表示有符号数，unsigned 表示无符号数。有符号数以二进制补码形式存储，最高位为符号位，“0”表示正，“1”表示负。无符号整数只能是正数。char 型、int 型、short 型和 long 型默认情况也就是不加修饰的情况下是有符号（signed）的。

bool 型数据取值只能是 false（假）或 true（真）。它所占的字节数在不同的编译系统中可能也不一样，但大多数是占 1 个字节。

2.常量

所谓常量就是在程序运行过程中始终不会变的量，就是直接用文字表示的值，例如，1，23，true，‘B’都是常量。常量又分整型常量、实型常量、字符常量、字符串常量和布尔常量。

整型常量包括正整数、负整数和零。整型常量的形式有十进制、八进制和十六进制。十进制我们都知道了，八进制常量的数字必须以数字 0 开头，比如 0324，-0123。十六进制整型常量的数字必须以 0x 开头，比如 0x3af。

实型常量就是数学上的小数，有两种表示形式：一般形式和指数形式。一般形式比如：13.7，-22.5。指数形式比如：0.2E+2 表示 $0.2*10^2$。

字符常量是单引号括起来的一个字符，比如：'b'，'?'。还有一些不可显示字符，例如响铃、换行、制表符等等，C++提供了一种转义序列的表示法来表示这些字符。比如：\a 表示响铃，\n 表示换行，\t 表示水平制表符，\b 表示退格，\r 表示回车，\\表示字符'\'，\"表示双引号，\'表示单引号。ASCII 字符常量占用 1 个字节。

字符串常量是用双引号括起来的字符序列，比如："China"。字符串常量会在字符序列末尾添加'\0'作为结尾标记。

布尔常量只有两个：false（假）和 true（真）。

3.变量

变量与常量一样也有自己的类型，在使用之前必须首先声明它的类型和名称。变量名也是标识符，因此命名规则应遵从标识符的命名规则。同一个语句中可以声明同一个类型的多个变量，变量声明语句的形式是这样的：数据类型　变量名 1,变量名 2,...,变量名 n;。例如下面两条语句分别声明了两个 int 变量和两个 float 变量：int　num,sum;　float　a,b;。在声明一个变量的同时可以给我赋一个初值，int　num=3;　double d=2.53;　char　c='a';。赋初值还有两一种形式，比如：int　num(3);。

4.符号常量

我们除了可以用文字表示常量以外，还可以给常量起个名字，这就是符号常量。这个符号常量就代表了那个常量。符号常量在使用之前必须声明，跟变量相似。符号常量声明形式：const　数据类型说明符　常量名=常量值;　或　数据类型说明符　const　常量名=常量值;。例如，我们给圆周率起个名字，就是符号常量，const　float　pi=3.1415926;。还有一点必须注意，**符号常量声明时必须赋初值**，在其他时候不能改变它的值。使用符号常量与文字常量相比有很多好处：程序的可读性更高，我们看到这个名字就能看出它的具体意思，再就是最重要的，如果我们多个地方都用了上面那个 pi 常量，但后来圆周率的值精度我想改一下，只用 3.14，这个时候怎么把所有的 pi 都换掉呢？我们只需要修改 pi 的声明就行了：const　float　pi=3.14;，但是如果使用文字常量即所有用圆周率的地方直接写的 3.1415926，那么就必须全找到再换掉，这样不但麻烦而且容易漏掉。

好啦，今天鸡啄米就讲到这里了，这些都是C++语言语法的一些基本点，大家一定要掌握好哦。

鸡啄米：C++编程入门系列之五（运算符和表达式）

上一讲鸡啄米给大家讲了一些数据类型，这一讲主要讲解编程入门知识-运算符和表达式。运算符，顾名思义，就是用于计算的符号，比如+，-，*，/。表达式是用于计算的公式，由运算符、运算量（操作数）和括号组成。

有些运算符需要两个操作数，使用形式为：操作数 运算符 操作数，这样的运算符就叫做二元运算符或双目运算符，只需要一个操作数的运算符叫做一元运算符或单目运算符。运算符具有优先级和结合性。如果一个表达式中有多个运算符则先进行优先级高的运算，后进行优先级低的运算。结合性就是指当一个操作数左边和右边的运算符优先级相同时按什么样的顺序进行运算，是自左向右还是自右向左，下面会具体讲到。鸡啄米来详细介绍几种类型的运算符和表达式。

1.算术运算符和算术表达式

算术运算符包括基本算术运算符和自增自减运算符。由算术运算符、操作数和括号组成的表达式称为算术表达式。基本算术运算符有：+（加），-（减或负号），*（乘），/（除），%（求余）。其中"-"作为负号时为一元运算符，作为减号时为二元运算符。优先级跟我们数学里的是一样的，先乘除，后加减。**"%"是求余运算，它的操作数必须是整数，**比如a%b是要计算a除以b后的余数，它的优先级与"/"相同，这里要注意的是，"/"用于两个整数相除时，结果含有小数的话小数部分会舍掉，比如2/3的结果是0。

C++的自增运算符"++"和自减运算符"--"都是一元运算符，这两个运算符都有前置和后置两种形式，比如i++是后置，--j是前置。无论是前置还是后置都是将操作数的值增1或减1后再存到操作数内存中的位置。如果i的原值是2，则i++这个表达式的结果是2，i的值则变为3,。如果j的原值也是2，则--j这个表达式的结果是1，j的值也变为1。自增或自减表达式包含到更复杂的表达式中时，比如假设i的原值是1，cout<<i++这个表达式会先输出i的值1，然后i再自增1，变为2，而cout<<++i这个表达式会先使i先自增1变为2，然后再输出i的值2。

2.赋值运算符和赋值表达式

最简单的赋值运算符就是"="，带有赋值运算符的表达式被称为赋值表达式。例如n=n+2就是一个赋值表达式，赋值表达式的作用就是把等号右边表达式的值赋给等号左边的对象。赋值表达式的类型是等号左边对象的类型，它的结果值也是等号左边对象被赋值后的值，赋值运算符的结合性是自右向左。什么叫自右向左呢？请看这个例子：a=b=c=1这个表达式会先从右边算起，即先算c=1，c的值变为1这个表达式的值也是1，然后这个表达式就变成了a=b=1，再计算b=1，同样b也变为1，b=1这个表达式的值也变成1，所以a也就变成了1。

除了"="外，赋值运算符还有+=，-=，*=，/=，%=，<<=，>>=，&=，^=，|=。其中前五个是赋值运算符和算术运算符组成的，后五个是赋值运算符和位运算符组成的，这几个赋值运算符的优先级跟"="相同，结合性也是自右向左。鸡啄米举几个例子说明下，a+=5 就等价于 a=a+5，x*=y+3 等价于 x=x*(y+3)。

3.逗号运算符和逗号表达式

逗号也是一个运算符，它的使用形式为：表达式 1,表达式 2。求这个表达式的值就要先解表达式 1，然后解表达式 2，最终这个逗号表达式的值是表达式 2 的值。比如计算 a=1*2,a+3，应先计算 a=1*2，结果为 2，再计算 a+3 的值，a 的值已经变成了 2，再加上 3 为 5，这个逗号表达式的最终结果就是 5。

4.逻辑运算和逻辑表达式

C++中提供了用于比较的关系运算符和用于逻辑分析的逻辑运算符。

关系运算符包括<（小于）、<=（小于等于）、>（大于）、>=（大于等于）、==（等于）、!=（不等于）。前四个的优先级相同，后两个的优先级相同，而且前四个比后两个的优先级高。用关系运算符把两个表达式连起来就是关系表达式，关系表达式的结果类型为 bool，值只能是 true 或 false。比如，a>b，a 大于 b 时表达式 a>b 表达式的值是 true，否则就是 false。更复杂的表达式也算是一个道理。

逻辑运算符包括!（非）、&&（与）、||（或），优先级依次降低。用逻辑运算符将关系表达式连起来就是逻辑表达式，逻辑表达式的结果也是 bool 类型，值也只能是 true 或 false。"!"是一元运算符，使用形式是!操作数。非运算是对操作数取反。比如!a，a 的值是 true，则!a 的值是 false。"&&"是二元运算符，用来求两个操作数的逻辑与，只有两个操作数的值都是 true，逻辑与的结果才是 true，其他情况下结果都是 false。"||"也是二元运算符，用来求两个操作数的逻辑或，只有两个操作数的值都是 false 时，逻辑或的结果才是 false，其他情况下结果都是 true。比如，int a=3,b=5,c=2,d=1; 则逻辑表达式(a>b)&&(c>d)的值为 false。

5.条件运算符和条件表达式

C++中唯一的一个三元运算符是条件运算符"?"。条件表达式的使用形式是：表达式 1?表达式 2:表达式 3。表达式 1 是 bool 类型的，表达死 2,3 可以是任何类型，并且类型可以不同。条件表达式的类型是表达式 2 和 3 中较高的类型，类型的高低后面介绍。条件表达式会先解表达式 1，如果表达式 1 的值是 true，则解表达式 2，表达式 2 的值就是条件表达式的值，而如果表达式 1 的值是 false，则解表达式 3，其值就是条件表达式的最终结果。比如：(a<b)?a:b，如果 a 小于 b，则结果为 a，如果 a 大于 b，则结果为 b。

6.sizeof 运算符

sizeof 运算符用来计算某个对象在内存中占用的字节数。此运算符的使用形式为：sizeof(类型名)或 sizeof(表达式)。计算结果是这个类型或者这个表达式结果在内存中占的字节数。

7.位运算

（1）按位与（&）。它是对两个操作数的二进制形式的每一位分别进行逻辑与操作。比如 3 的二进制形式为 00000011，5 的二进制形式为 00000101，按位与后结果是 00000001。

（2）按位或（|）。它对两个操作数的二进制形式的每一位分别进行逻辑或操作。还是比如 3 和 5 按位或运算后结果是 00000111。

（3）按位异或（^）。它对两个操作数的每一位进行异或，也就是如果对应位相同则运算结果为 0，若对应位不同则计算结果为 1。例如 3 和 5 按位异或后结果为 00000110。

（4）按位取反（~）。这是一个一元运算符。它对一个二进制数的每一位求反。比如，3 按位取反就是 11111100。

（5）移位。包括左移运算（<<）和右移运算（>>），都是二元运算符。移位运算符左边的数是需要移位的数值，右边的数是移动的位数。左移是按指定的位数将一个数的二进制值向左移位，左移后，低位补 0，移出的高位舍弃。右移是按照指定的位数将一个数的二进制值向右移位，右移后，移出的低位舍弃，如果是无符号数则高位补 0，如果是有符号数，则高位补符号位或 0，一般补符号位。比如，char 型变量的值是-8，则它在内存中的二进制补码值是 11111000，所以 a>>2 则需要将最右边两个 0 移出，最左边补两个 1，因为符号位是 1，则结果为 11111110，对其再求补码就得到最终结果-2。

8.混合运算时数据类型的转换。

表达式中的类型转换分为：隐含转换和强制转换。

在算术运算和关系运算中如果参与运算的操作数类型不一样，则系统会对其进行类型转换，这是**隐含转换，转换的原则就是将低类型的数据转换为高类型数据。各类型从低到高依次为 char，short，int，unsigned int，long，unsigned long，float，double。类型越高范围越大，精度也越高。隐含转换是安全的，因为没有精度损失。**逻辑运算符的操作数必须是 bool 型，如果不是就需要将其转换为 bool 型，非 0 数据转换为 true，0 转换为 false。位运算操作数必须是整数，如果不是也会自动进行类型转换，也是低类型数据转换为高类型数据。赋值运算要求赋值运算符左边的值和右边的值类型相同，不同的话也要进行自动转换，但这个时候不会遵从上面的原则而是一律将右值转换为左值的类型。比如，int iVal; float fVal; double dVal;则 dVal=iVal*fVal;计算时先将 iVal 转换为跟 fVal 一样的 float 型，乘法的结果再转换为 double 型。

强制类型转换是由类型说明符和括号来实现的，使用形式为：类型说明符(表达式)或　(类型说明符)表达式。它是将表达式的结果类型强制转换为类型说明符指定的类型。比如，float fVal=1.2; int iVal = (int)fVal;计算后面表达式的值时会将 1.2 强制转换成 1，舍弃小数部分。

好了，鸡啄米今天就讲到这里了，又是很重要的基础知识，以后学习中有很多地方会混淆，注意要记牢哦。

鸡啄米：C++编程入门系列之六（算法的基本控制结构之选择结构）

上一讲鸡啄米为大家讲了运算符和表达式的知识，这一讲鸡啄米主要讲算法基本控制结构中的选择结构，同时教大家写比较简单的程序。

为了能写出简单的程序并看见它们的输出效果，鸡啄米先讲下数据的输入和输出。数据从一个对象到另一个对象的流动我们一般称之为“流”，比如程序中的数据在屏幕上显示出来，我们可以想成数据从程序流向屏幕，就是输出流。从键盘输入数据就是输入流了。从流中获取数据叫做提取操作，向流中添加数据叫做插入操作。cin 是系统预定义的输入流，用来处理标准输入即键盘输入。cout 是预定义的输出流，用来处理标准输出，即屏幕输出。

“<<”是预定义的插入符，它用在 cout 上可以实现屏幕输出。使用形式如下：cout<<表达式<<表达式...。这里可以连着多个表达式，输出多个数据到屏幕。这里的表达式可以是很复杂的表达式，系统会计算出这些表达式的值只把结果传给插入符<<，然后显示到屏幕上。例如，cout<<"a+b="<<a+b;会把"a+b="这个字符串和 a+b 的计算结果输出到屏幕。如果 a=1，b=2；则屏幕上显示 a+b=3。

“>>”是提取符，用到 cin 上用来把键盘上输入的数赋值给变量。使用形式为：cin>>表达式>>表达式...。这里的提取符也可以有多个，每个后边跟一个表达式，这里的表达式一般是用来存放输入值的变量。比如，int a,b; cin>>a>>b;后面这个语句要求从键盘上输入两个整型数，两个数之间用空格分隔，如果输入 3 4，则变量 a 的值为 3，b 的值为 4。

再介绍几个操纵符，操纵符用来把输出值以特殊的格式显示出来。比如 setw(int)用来设置域宽，就是设置数值的显示位数，setprecision(int)用来设置浮点数的小数位数（包括小数点），endl 插入换行符，并刷新流。还有 Dec（decimal 十进制），Hex（hexadecimal 十六进制），Oct（octal 八进制）（binary 二进制）是要求以几进制显示。

下面鸡啄米开始讲算法的基本控制结构。算法的基本控制结构包括顺序结构、选择结构和循环结构。顺序结构就是按照事物的逻辑一条语句一条语句的写下来，顺序执行，就像流水账，这种结构最常见，也最简单，这里就不讲了。这一讲鸡啄米主要讲选择结构，循环结构下一讲会讲到。

大家想想，如果有这样一个问题，若 x<0，则 y=-1，若 x=0，则 y=0，若 x>0，则 y=1。这里涉及到好几种选择，顺序结构肯定不行了，就要用到选择结构。

一.用 if 语句实现选择结构

if 语句专门用来实现选择结构，使用形式为：

```
if (表达式)
    语句 1
else
    语句 2
```

这个结构的执行顺序是，先计算表达式的值，如果为 true，则执行语句 1，否则就执行语句 2。比如，

```
if (x>y)
    cout<<x;
else
    cout<<y;
```

这段程序可以用来输出 x 和 y 中比较大的那个数。if 语句中的 else 和语句 2 可以没有，变成：if (表达式) 语句。比如，if (x>y) cout<<x。

这里鸡啄米给大家一个程序例子，大家可以在 vs2010 中点 File->New->Project 创建一个 Win32 Console Application。比如创建的新工程名叫 Test，然后找到 Test.cpp，里面有 _tmain 函数，实际上就是鸡啄米前面说的 main 函数，是程序的入口函数。我们就在这个 main 函数里写一段代码，用来判断输入一个年份时这个年份是不是闰年。大家知道，可以被 4 整除不能被 100 整除，或者能被 400 整除的年份都是闰年。下面是程序：

```
#include "stdafx.h"
#include <iostream>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
    int nYear;
    bool bIsLeapYear;
    cout<<"Enter the year:";
    cin>>nYear;
    bIsLeapYear = ((nYear%4==0 && nYear%100!=0) || (nYear%400==0));
    if (bIsLeapYear)
        cout<<nYear<<"is a leap year."<<endl;
    else
        cout<<nYear<<"is not a leap year."<<endl;
```

```
    return 0;
}
```

因为 vs2010 里已经没有以前 iostream.h 文件，所以这里包含了 iostream 文件，这是最新的 C++标准，里面同样包含了输入输出函数，using namespace std 是说要用到 std 名字空间，名字空间鸡啄米以后会讲。开始 debug 运行，屏幕上会显示：Enter the year:，然后我们输入 2000，点回车屏幕上接着显示：2000 is a leap year.可能你看不到这一句窗口就消失了，没关系，可以点 debug->start without debugging，这样窗口就不会消失了。

二.多重选择结构

有些问题可能需要我们进行多次判断，这样就可以有几种方法：

1.嵌套的 if 语句

使用的语法形式：

```
if(表达式 1)
   if(表达式 2)   语句 1
   else           语句 2
else
   if(表达式 3)   语句 3
   else           语句 4
```

语句 1、2、3、4 可以是复合语句。每一层的 if 都要与 else 配对，如果省略掉一个 else 则要使用{}把这一层的 if 语句括起来。鸡啄米建议大家写程序的时候最好每层都用大括号括起来这样会大大减少出错的概率，也比较整齐，即使是熟手也一样。鸡啄米再给大家举个例子：

```
#include "stdafx.h"
#include <iostream>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
    int x,y;
    cout<<"Enter x and y:";
    cin>>x>>y;
    if (x!=y)
    {
        if (x>y)
            cout<<"x>y"<<endl;
        else
            cout<<"x<y"<<endl;
    }
```

```
    else
    {
        cout<<"x=y"<<endl;
    }
    return 0;
}
```

运行这个程序，屏幕上会显示，Enter x and y:，然后输入 3 5，按回车接着会显示 x <y。

2.if...else if 语句

若 if 语句的嵌套都在 else 分支下，就可以使用 if...else if 语句。使用的语法形式为：

```
if (表达式 1)          语句 1
else if (表达式 2)    语句 2
else if (表达式 3)    语句 3
...
else                   语句 n
```

这里的执行逻辑就是，如果表达式 1 为 true，则执行语句 1，如果表达式 1 为 false，且表达式 2 为 true 则执行语句 2，如果表达式 1、表达式 2 为 false，且表达式 3 为 true 则执行语句 3...就这样一层一层判断着执行下去。

3.switch 语句

我们进行判断选择的时候，有可能每次都是对同一个表达式的值进行判断，那么就没有必要在每一个嵌套的 if 语句里都计算一下它的值，而是使用 switch 来解决这个问题。语法形式为：

```
switch (表达式)
{
case 常量表达式 1:  语句 1
case 常量表达式 2:  语句 2
...
case 常量表达式 n:  语句 n
default:          语句 n+1
}
```

此类语句的执行顺序是，先计算表达式的值，然后在 case 语句中寻找与之相等的常量表达式，跳到此处开始执行，若没有与之相等的则跳到 default 开始执行。使用 switch 语句时要注意以下几点：这些常量表达式的值不能相同，顺序可以随便；每个 case 语句的最后都要加 break 语句，不然会一直把下面所有的语句执行完；switch 括号里的表达式必须是整型、字符型和枚举型的一种；每个 case 下的语句不需要加{}；如果多个 case 下执行一样的操作，则多个 case 可以共用一组语句，比如

```
    case 1:
    case 2:
    case 3:   a++;
              break;
```

鸡啄米再给大家举个 switch 的例子。如果整型变量 a 为 0 则输出“小鸡”,如果为 1 则输出“小啄”，如果为 2 则输出“小米”，其他情况下输出“鸡啄米”。程序如下：

```
#include "stdafx.h"
#include <iostream>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
        int a;
        cout<<"Enter a:";
         cin>>a;
        switch (a)
        {
         case 0:
                cout<<"小鸡"<<endl;
                 break;
        case 1:
                cout<<"小啄"<<endl;
                break;
        case 2:
                cout<<"小米"<<endl;
                break;
        default:
                cout<<"鸡啄米"<<endl;
                break;
        }
        return 0;
}
```

运行时屏幕显示 Enter a:，我们输入 2，则会接着显示“小米”。

这一讲鸡啄米首先讲了输入输出流对象，然后讲了选择控制结构，举了几个程序例子，大家可以按照鸡啄米的程序上机调试运行下，也可以自己在此基础上改改程序然后看看结果是否如你所想，大家先找找编程的感觉，在编程入门的时候不要着急。

鸡啄米：C++编程入门系列之七（算法的基本控制结构之循环结构）

大家好啊，鸡啄米在C++编程入门上一讲中讲了算法的基本控制结构当中的选择结构，这一讲来讲讲另一种控制结构-循环结构。

大家想想，我们要统计一个班上所有同学的分数，如果使用顺序结构就是一个分数一个分数的加，语句多，而且执行的动作都是一样的，这个时候就需要用到循环结构了。

循环结构有三种循环控制语句，就是有三种写法：while语句、do-while语句和for语句。

1.while语句

使用的语法形式是：

```
while(表达式)    语句
```

它的执行顺序是，先计算while括号里表达式的值是true还是false，如果是true则执行循环体也就是后面的语句，如果是false则跳出while循环，继续执行while循环下面的程序。while后面的语句可以是多条语句，是多条语句的话要用大括号括起来。大家注意，循环体里应该有可以改变表达式值的语句，以便让它能循环到一定程度时跳出循环，不然可就死循环了。这里给大家一个while循环的例子：

```
#include<iostream>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
        int  i=1, sum=0;
        while(i<=10)
        {
                sum+=i;  //相当于 sum=sum+i;
                 i++;
        }
        cout<<"sum="<<sum<<endl;
        return 0;
}
```

这段代码是计算1一直累加到10的和，输出结果为：sum=55。循环体一定要用大括号括起来，不然它只会循环运行sum+=i;这一句了。

2.do-while语句

使用的语法形式如下：

```
do   语句
while(表达式)
```

它的执行顺序是：先执行循环体语句，就是 do 后面的语句，再判断循环条件表达式的值也就是 while 括号里表达式的值，如果为 true 则继续执行循环体，如果为 false 则结束循环，继续执行 do-while 循环后面的语句。这里还要强调下，跟 while 循环一样，循环体中一定要由改变循环条件表达式的语句，不然也会死循环。do 后面的语句也可以是多条语句，是多条语句的话也要用大括号括起来。再给大家看看 do-while 的例子：

```
#include<iostream>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
        int i=1, sum=0;
        do{
                sum+=i;
                i++;
            }while(i<=10);
        cout<<"sum="<<sum<<endl;
        return 0;
}
```

这段代码看起来跟上面差不多，只是用 do-while 语句代替了 while 语句。运行结果也是一样的，也会输出 sum=55。

那大家是不是认为 do-while 和 while 语句是一样的了？那就错了。因为它们的执行顺序不同，while 语句是先判断循环条件表达式的值再执行循环体，do-while 是先执行循环体再判断循环条件表达式的值。大家想想，如果上面这两个例子当中的 i 我们都定义成 int i(11)或者 int i=11，就是 i 的初始值是 11，那结果怎么样呢？上面那段 while 语句的代码会判断 11 是不是<=10，当然是 false 了，循环体不会执行 while 循环就退出了，所以 sum 的值还是初始值，输出 sum=0。但是下面的 do-while 语句会先执行一下 do 后面大括号里的语句，sum+=i 将 sum 的值变成了 11，然后执行 i++之后就判断 while 里 i<=10 的值了，当然也是 false，不会继续执行 do 里的语句了，最后输出 sum=11。大家可以上机调试下，鸡啄米给大家总结下，如果一开始循环条件就为 false 的话，两种语句的执行结果是不一样的，其他情况下一样。

3.for 语句

for 语句是最灵活的循环语句，可用于循环次数已知的情况也可用于循环次数未知的情况。for 语句的语法形式如下：

for(表达式 1;表达式 2;表达式 3)

语句

它的执行顺序是，先计算表达式 1 的值，再计算表达式 2 的值，根据表达式 2 的值决定是否执行循环体，若是 true 则执行一次循环体，若为 false 则退出循环。每执行一次循环体后都计算表达式 3 的值，然后再计算表达式 2 的值根据其值判断是否执行循环体，就这样循环了。表达式 1 的值只在刚进入 for 语句的时候执行一次，做一些初始化，后面就不执行了。鸡啄米再给大家找个例子：

```
#include <iostream>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
        int n, k;
        cout << "Enter a positive integer: ";
        cin >> n;
        cout << "Number " << n << " Factors ";
        for (k=1; k <= n; k++)
            if (n % k == 0)
                cout << k << " ";
        cout << endl;
        return 0;
}
```

这个程序是让我们在屏幕上先设一个 n 值，然后它会从 1 到 n 判断哪些整数是 n 的因子，就是 n 能被谁整除。运行结果是：

```
Enter a positive integer: 36
Number 36 Factors 1 2 3 4 6 9 12 18 36
```

这里的 36 只是鸡啄米举的一个例子，大家可以输入别的值。

鸡啄米就 for 语句要跟大家说几个注意的地方：1.表达式 1、2、3 都可以省略，分号不能省略，都省略后成为 for(;;) 语句，相当于 while(true)，成死循环；2.表达式 2 是循环控制条件，只要是省略掉它，就会成为死循环；3.表达式 1 一般是给循环控制条件赋初值，比如上面的 k=1，也可以是跟循环变量无关的表达式，如果是表达式 1 省略或是跟循环条件无关的表达式，就应该在进入 for 语句前给循环条件赋初值；4.表达式 3 为改变循环控制条件的值，如果它被省略掉或者是跟循环条件无关的表达式，就应该在循环体里另有语句来改变循环条件，保证不死循环。比如，for(i=1;i<=10;) { sum+=i; i++; }，这里因为 for 后面括号里没有表达式 3，则它的循环体里就要加类似 i++这样的语句来改变循环条件 i。5.如果省略掉表达式 1 和表达式 3 只剩下表达式 2 则 for 语句完全等同于 while 语句。比如 for(;i<=10;) { i++; } 相当于 while(i<=10) { i++; }。

三种循环语句鸡啄米就介绍完了，下面说说**循环结构的嵌套**。一个循环体里边可以包含另一个循环结构。while、do-while 和 for 三种循环语句可以相互嵌套。给大家一个例子：

```
#include <iostream>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
        int i=1, a=0;
        for(; i<=5; i++)
        {
              do{
                    i++;
                   a++;
                }while(i<3);
              i++;
        }
        cout << a << "," << i <<endl;
        return 0;
}
```

这里大家可以上机编译运行下看看结果。鸡啄米还要说明一点，上面程序里的 int i=1, a=0;最好写成两个语句:int i=1; int a=0;一个是比较清晰，不容易犯错，也不容易让人误解，比如 int i, a=0;这样的语句其实只对 a 赋了个初值 0，并没有给 i 赋初值，但有可能会给别人造成误解，或者别人那一会有点晕，以为也给 i 赋了个初值。

循环结构也可以和选择结构嵌套，鸡啄米再给大家个例子：

```
#include<iostream>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
      int n；
      for(n=100; n<=200; n++)
      {
            if (n%3!=0)
                cout<<n<<" ";
      }
      return 0;
}
```

这个程序是要求 100 到 200 之间不能被 3 整除的数，然后输出。

经常用来控制循环结构的语句还有几个，鸡啄米再跟大家简单说一下，1.break 语句。break 出现在 switch 语句或者循环体中时，程序直接从 switch 语句中或者循环体中跳出，继

续执行下面的程序。2.continue 语句。continue 语句用在循环体中时，用来结束本次循环，接着判断决定是否执行下一次循环，它跟 break 的区别是，比如，for(int i=0; i<5; i++) { if(i==1) break; sum+=i; }这里如果循环到 i 等于 1 的时候 for 循环就会直接退出，而 for(int i=0; i<5; i++) { if(i==1) continue; sum+=i; }的情况是如果循环执行到 i 等于 1 的时候则 sum+=i 这个语句不执行了直接执行 for 后面括号里表达式 3-i++，也就是进入 i 等于 2 的循环。3.goto 语句。语法形式是，goto 语句标号;。其中语句标号是用来表示语句的标识符，放在语句的最前面，并用冒号跟语句分开，它的作用是让程序执行时直接跳到语句标号指定的语句，这个语句有些危险，尽量少用或不用。

循环结构的知识差不多就这些了，主要是以后要多练习练习，就熟悉了。算法的基本控制结构就这三种：顺序结构、选择结构（上节讲过）和循环结构。大家要自己多试试，编译运行下鸡啄米给出的程序看看运行结果，然后改改程序再看看，编程入门时基本知识学踏实点，以后就可以更好的理解后面的知识。

鸡啄米：C++编程入门系列之八（自定义数据类型）

上两节中鸡啄米讲了编程入门知识-算法的选择结构和循环结构，这一讲鸡啄米给大家介绍下自定义数据类型。

什么是自定义数据类型呢？大家记得像 char、int、float 等类型吗？这些都是 C++内置的基本数据类型，这些数据类型是不能满足我们在编程中的要求的，需要自己定义一些数据类型使用。自定义数据类型有：枚举类型、结构类型、联合类型、数组类型和类等。这一讲鸡啄米主要讲枚举类型、结构类型和联合类型，其他类型后面鸡啄米会细讲。

1.枚举类型

我们在现实当中经常遇到这种情况，描述一件事情时只能取几个有限的值，比如，一周只有星期一、星期二...星期日七天，比赛结果只有胜、负、平和比赛取消等四种结果。我们可以用 0 到 6 的整数表示星期一到星期日，7,8,9 等都不是合法数据，如果我们用整数来表示星期几那就需要专门判断下是不是在 0 到 6 之间，但是如果用枚举类型定义就没有这种问题，因为枚举类型就那几个可能的值。枚举类型的声明形式为： **enum** 枚举类型名 { 变量值列表 }; **变量值列表里都是整型变量，另外不要忘记最后面的分号！**

比如，enum weekday { sun,mon,tue,wed,thu,fri,sat };

这里的枚举元素 sun、mon...都没有指定值，它们就使用默认值，依次为 0,1,2...。即 sun 就是 0，mon 就是 1，...sat 是 6。也可以在声明枚举类型时指定枚举元素的值，比如

enum weekday { sun=7, mon=1, tue, wed, thu, fri, sat };

这里 sun 就是 7，mon 是 1，后面的值在 mon 的基础上依次加 1，即 tue 等于 2，wed 为 3...sat 是 6。枚举元素按常量处理，不能对它们赋值除了声明的时候，像 sun=0;这样的语句就是非法的。

枚举类型我们声明了以后就能像 int、float 等类型那样用了，可以定义枚举类型的变量。整数值需要赋给枚举变量时，应进行强制类型转换。

鸡啄米给大家个例子：设某次体育比赛的结果有四种可能：胜（win）、负（lose）、平局（tie）、比赛取消（cancel），编写程序顺序输出这四种情况。分析：由于比赛结果只有四种可能，所以可以声明一个枚举类型，声明一个枚举类型的变量来存放比赛结果。

```
#include <iostream>
using namespace std;
enum   game_result     {WIN, LOSE, TIE, CANCEL};

int _tmain(int argc, _TCHAR* argv[])
{
        game_result   result;                          // 声明变量时，可以不写关键字 enum
        enum   game_result   omit = CANCEL;       // 也可以在类型名前写上关键字 enum
         int count;
        for (count = WIN ; count <= CANCEL ; count++)
        {
                result = (game_result)count;
                if (result == omit)
                {
                        cout << "The game was cancelled\n";
                }
                else
                {
                        cout << "The game was played ";
                        if (result == WIN)
                                cout << "and we won!";
                         if (result == LOSE)
                                cout << "and we lost.";
                         cout << "\n";
                }
        }
         return 0;
}
```

运行结果是：

```
The game was played and we won!
The game was played and we lost.
The game was played
The game was cancelled
```

2.结构体

有时候我们需要将一些不同类型的数据组合成一个整体，例如，一个学生的学号、姓名、年龄和成绩等，它们数据类型不同，但都属于一个人。这就要用到结构体了，结构体就是由不同数据类型也可以是相同数据类型的若干数据组成的结合体，声明形式为：

```
struct  结构体名
{
    数据类型    成员名 1；
    数据类型    成员名 2；
      :
    数据类型    成员名 n；
};
```

这里**最后那个分号也不要忘记**哦。鸡啄米把上面提到的那个学生的结构体可以定义成如下形式：

```
struct student    //学生信息结构体
{
    int        num; //学号
    char    name[20]; //姓名
    char     sex; //性别
     int        age; //年龄
    float     score; //成绩
    char     addr[30]; //住址
};
```

定义了结构体类型，我们就可以用它来定义结构体变量，就像 int a;那样。结构体型变量的定义形式也类似 int 等类型： 结构体类型名 结构体变量名;。比如，定义一个上面的那个结构体变量，student s;。可以像这样声明了结构体类型以后再定义结构体变量，也可以把它们放在一起。结构体变量占用内存的大小可以用 sizeof(结构体类型名或变量名)计算。在定义结构体变量时可以直接赋初值。结构体成员的使用形式是：**结构体变量名.成员名。**

鸡啄米给大家一个结构体变量的例子。

```
#include <iostream>
using namespace std;
struct student        //学生信息结构体
{
    int        num;//学号
    char    name[20];//姓名
    char    sex;//性别
    int        age;//年龄
}stu={97001,"Lin Lin",'F',19};        // 声明结构体类型的同时，定义结构体类型的变量，并赋初值。也可以分开。
int _tmain(int argc, _TCHAR* argv[])
{
     cout<<stu.num<<"  "<<stu.name<<"  "<<stu.sex<<"  "<<stu.age;
```

```
    return 0;
}
```

大家可以放到 vs2010 上运行下看看结果。

3.联合体

有的时候需要几个不同类型的变量共用一组内存单元，这时就需要用联合体，联合体类型声明的语法形式为：

```
union 联合名
{
    数据类型    成员名 1;
    数据类型    成员名 2;
      :
    数据类型    成员名 n;
};
```

联合体类型变量声明的语法形式是：联合体类型名　联合体变量名;。联合体类型变量的引用形式是：联合体变量名.成员名。

鸡啄米给大家举个联合体类型的例子：

```
union uarea
{
    char   c_data;
    short  s_data;
    long   l_data;
}
```

联合体类型在内存中是怎样占用内存单元的呢？联合体占用的内存大小是其占内存最大的成员需要的内存大小，就像上边的 uarea，它占用的内存大小就是最大成员 long 需要的 4 个字节。联合体里同时只能放一个成员的值。uarea 放 c_data 时 c_data 占用第一个字节，放 s_data 时 s_data 占用前两个字节，放 l_data 时 l_data 占用全部的 4 个字节。

联合体也可以没有名称，叫做无名联合体。无名联合体只是声明一个成员项的集合，这些成员有相同的内存地址，可以用成员项的名字直接访问。鸡啄米给大家一个无名联合体例子：

```
union
{
    int    i;
    float  f;
}
```

在程序总可以直接用成员项 i 和 f 的名字访问 i 和 f，比如 i=2; f=3.2;。无名联合体一般是用作结构体的内嵌成员。鸡啄米给写一个使用联合体的例子：

```
#include <iostream>
using namespace std;
union myun
{
        int    k;
        struct { int x; int y; int z; }u;
}a;
int _tmain(int argc, _TCHAR* argv[])
{
       a.u.x =4;
       a.u.y =5;
       a.u.z =6;
       a.k = 0;
        cout<<a.u.x<<" "<<a.u.y<<" "<<a.u.z<<" ";
        return 0;
}
```

这个程序输出 0 5 6。为什么呢？不应该是 4,5,6 吗?因为联合体的成员共用内存，所以 a.u.x 占用的内存里的 4 被后来的 a.k 覆盖了，变成了 0。

4.类型定义语句--typedef

我们编程时除了可以使用基本数据类型名和自定义的数据类型名以外还可以给一个已经存在的数据类型起个别名，这样做有几个好处，可以给已有的类型起一个有意义的名字，我们一看就知道它表示什么，提高可读性；还可以给较长的类型名另起一个短名。使用的语法形式是：typedef 已有类型名 新类型名表;。新类型名表中可以有多个标识符，它们之间用逗号分开，就是在一个 typedef 语句中可以为一个已有数据类型声明多个别名，比如，

```
typedef   double   length,width;
length   a;
width    b;
```

鸡啄米讲了几个复杂点的自定义数据类型，大家好好消化吧，结构体类型用的比较多，而且结构体里可以放函数的，只是我们一般不那样用，跟我们以后要讲的 C++中的核心概念-类很相似，可以说基本一样。就到这里了，大家辛苦了!

2013.6.14

第三部分：函数

鸡啄米：C++编程入门系列之九（函数的定义与调用）

八月十五了，鸡啄米祝大家中秋快乐，多吃几块月饼啊。

上一讲鸡啄米给大家讲了自定义数据类型，这一节鸡啄米给大家讲讲编程入门时另一个比较重要的概念，函数。一个复杂的问题往往可以分为若干子问题，然后对每个子问题分别解决。C++和 C 语言就是用函数来解决子问题的。函数写好以后，可以被重复调用，我们调用时只需要关注它的功能和使用方法，至于它是怎样实现的我们不需要关心。这样有利于代码重用，提高开发效率，便于分工开发和维护。

一个 C++程序可以由一个主函数和若干子函数组成，主函数是程序执行的开始点，一般就是 main 函数，主函数调用子函数，子函数还可以调用其他子函数。调用其他子函数的函数称为主调函数，被其他函数调用的子函数称为被调函数。

1.函数的定义

函数定义的语法形式是：

```
类型标识符  函数名(形式参数表)
{
    语句序列
}
```

类型标识符是函数的类型，就是常说的函数的返回值类型。函数的返回值可以返回给主调函数使用，由 return 语句给出，比如：return 0。没有返回值的函数的类型标识符为 void，不需要写 return 语句。

形式参数简称为形参，形参表的形式如下：type1 name1, type2 name2,...,typen namen。其中，type1，type2，...，typen 是类型标识符，表示形参的类型，name1,name2,..., namen 是形参名。形参的作用就是实现主调函数和被调函数之间的联系，形参一般是函数需要处理的数据、影响函数功能的因素，没有形参的函数在形参表的位置写 void 或者不写形参表。

函数在没有被调用的时候形参只是一个符号，它只表示形参的位置应该有一个什么样的数据。函数被调用时才由主调函数将实际参数赋予形参，实际参数通常简称实参。**简单说，函数声明时的参数叫形参，被调用时传入的参数叫做实参。**

2.函数的调用

a.函数的调用形式

函数调用之前必须先声明函数原型，可以在主调函数中声明，也可以在所有函数之前声明，只要在调用它之前声明就可以。声明形式为：

```
类型说明符  被调函数名(含类型说明的形参表)
```

若是在主调函数内部声明了被调函数原型，那么该原型只能在这个函数内部有效，也就是只能在这个函数中调用。若是在所有函数之前声明，则该函数原型在本程序文件中任何地方都有效，也就是在本程序文件中任何地方都可以按照该原型调用相应的函数。

声明了函数原型以后，就可以按如下形式调用子函数：函数名(实参表)。实参表中应该给出与函数原型中形参个数相同、类型相符的实参。函数调用可以作为一条语句，这时函数可以没有返回值；函数调用也可以出现在表达式中用于计算，这时就必须有返回值了。

鸡啄米给大家一个函数调用的例子：

```
#include <iostream>
using namespace std;
double power(double x, int n);
int _tmain(int argc, _TCHAR* argv[])
{
    cout<<"5 to the power 2 is "<<power(5,2)<<endl;
    return 0;
}
double power(double x, int n)
{
    double val=1.0;
    while (n--)
        val *= x;
    return(val);
}
```

运行结果是：5 to the power 2 is 25

b.函数调用的执行过程

鸡啄米再跟大家讲下函数调用的执行过程。C++程序经过编译以后生成可执行程序，存在硬盘中，程序启动时首先从硬盘把代码装载到内存的代码区，然后从入口地址也就是 main 函数的起始处开始执行。程序执行时如果遇到对其他函数的调用，则暂停当前函数的执行，保存下一条指令的地址，也就是返回地址，作为从子函数返回后继续执行的切入点，并保存变量状态等现场，然后转到子函数的入口地址执行子函数。遇到 return 语句或者子函数结束时则恢复先前保存的现场，并从先前保存的返回地址开始继续执行。

c.函数的嵌套调用

函数允许嵌套调用，如函数 1 调用函数 2，函数 2 又调用了函数 3 就形成了函数的嵌套调用。鸡啄米再给大家一个函数嵌套调用的例子：

```
#include <iostream>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
    int a,b;
```

```
    int fun1(int x,int y);
    cin>>a>>b;
    cout<<"a、b 的平方和："<<fun1(a,b)<<endl;
    return 0;
}
int fun1(int x,int y)
{
    int fun2(int m);
    return (fun2(x)+fun2(y));
}
int fun2(int m)
{
   return (m*m);
}
```

大家可以把这段代码拷到 VS2010 上运行下，修改再试下，理解理解函数的嵌套调用。

d.递归调用

递归调用就是函数直接或间接的调用自身。调用自身就是指在一个函数的函数体中出现了对自身的调用语句。直接调用自身如：void fun(void) { ...; fun(); ...}。间接调用自身如：void fun1(void) { ...; fun2(); ...} void fun2(void) { ...; fun1(); ...},这里 fun1 调用了 fun2，fun2 又调用了 fun1，这样就构成了递归。

递归调用就是将原有的问题分解成新的问题，而解决新问题时又用到了原有问题的解法。就这样循环分解下去，直到最终分解出来的问题时一个已知解的问题，这就是有限的递归调用，也才是有意义的，无限的递归调用永远也得不到解，没有实际意义。

递归调用有两个阶段，第一个阶段是递推，将原问题不断分解成新的子问题，逐渐向最终的解推进，最后达到已知的条件，也就是递归结束的条件，递推阶段就结束了。比如，计算 4!，可以这样分解递推：

4!=4×3!→3!=3×2!→2!=2×1!→1!=1×0!→0!=1

未知---------->已知

第二个阶段是回归，从已知的条件出发，按照递推的逆过程，逐一求值回归，最后到达递推的开始处，结束回归完成递归调用。还看求 4!的例子，回归阶段如下：

4!=4×3!=24←3!=3×2!=6←2!=2×1!=2←1!=1×0!=1←0!=1

未知<----------已知

鸡啄米举个递归调用的例子：

分析：计算 n!的公式是：n=0 时，n!=1，n>0 时，n!=n(n-1)!。这是一个递归形式的公式，编程求 n!时采用递归算法，递归的结束条件是 n=0。程序代码如下：

```
#include <iostream>
using namespace std;
long fac(int n)
{
	long f;
	if (n<0)
		cout<<"n<0,data error!"<<endl;
	else if (n==0)
		f=1;
	else
		f=fac(n-1)*n;
	return(f);
}
int _tmain(int argc, _TCHAR* argv[])
{
	long fac(int n);	//
	int n;
	long y;
	cout<<"Enter a positive integer:";
	cin>>n;
	y=fac(n);
	cout<<n<<"!="<<y<<endl;
	return 0;
}
```

运行结果：

```
Enter a positive integer:8
8!=40320
```

递归算法在编程入门时可能有点难，不好吸收，没关系，现在大家可以先理解下，以后等功力强了回来看下就很容易掌握了。而且递归算法的效率不是很高，能用其他算法尽量用效率高的算法。

鸡啄米今天就给大家讲到这儿了，中秋节了，看中秋晚会去了。

鸡啄米：C++编程入门系列之十（函数的参数传递与内联函数）

上一讲鸡啄米讲了函数的定义和调用，这一讲接着给大家讲函数的相关知识。让大家了解编程入门时需要知道的函数的参数是怎样传递的以及内联函数的概念。

一.函数的参数传递

上一讲中鸡啄米提到了函数的参数有形参和实参两种，那两者到底有什么区别呢？我们在声明及定义函数的时候跟的参数叫做形参，调用函数时传进去的参数称为实参。比如，函数 int fun(int a)，这里的 a 就是形参，我们在其他地方调用函数 fun，将外部的变量 b 传进去，即 fun(b)，这里的 b 就是实参。实参可以是常量、变量或表达式，它的类型必须和形参相符。函数没有被调用时，形参并不占用内存，只有在调用时才会分配内存空间，然后将实参传进去。函数参数的传递有两种方式，值调用和引用调用。

1.值调用

值调用就是调用函数时，给形参分配内存空间，将实参拷贝给形参，之后的函数执行中形参和实参就脱离了关系，谁都不影响谁。也就是值调用时，只是用实参的值初始化下形参，之后两者互不影响。

鸡啄米给大家个例子就很明白了：

```
#include<iostream>
using namespace std;
void Swap(int a, int b);
int _tmain(int argc, _TCHAR* argv[])
{
        int x=5, y=10;
        cout<<"x="<<x<<"    y="<<y<<endl;
         Swap(x,y);
        cout<<"x="<<x<<"    y="<<y<<endl;
         return 0;
}
void Swap(int a, int b)    //无返回值
{
        int t;
        t=a;
       a=b;
        b=t;
 }
```

运行结果是：

```
x=5     y=10
x=5     y=10
```

从运行结果中，大家可以看出，本来我们想调用 Swap 函数把 x 和 y 的值交换过来，但是事与愿违。这就是因为，这是值调用的方式，x 和 y 分别传值给 a 和 b 后，a 跟 b 的值在 Swap 内怎样变化都影响不到 x 和 y，所以两次输出 x 和 y 的值没有变化。

2.引用调用

如果我们想让被调函数中对形参做的修改同样对主调函数中的实参有效，用值调用不行，那怎么办呢？答案是用引用调用。

引用是一种特殊类型的变量，我们可以认为它是另一个变量的别名，利用引用名和原来的变量名访问变量的效果是一样的。引用的形式是：类型标识符 &引用名。比如：

```
int  i, j;
int &ri=i;    // 建立一个 int 型的引用 ri,并将其初始化为变量 i 的一个别名
j=10;
ri=j;         // 相当于 i=j;
```

使用引用鸡啄米提醒大家必须注意以下问题：声明一个引用时，必须同时对它进行初始化，使它指向一个已存在的对象；一旦一个引用被初始化后，就不能改为指向其它对象。简单说就是引用定义的时候就指定它指向的变量，之后就不能变了。

引用可以作为形参，比如 void swap(int& a, int& b) {...}。这个时候引用就不需要对其初始化了，因为形参只是类型说明，主调函数调用这个函数时才会为形参分配内存，也才会用实参来初始化形参。用引用调用后，形参就是实参的别名而已，对形参做的任何改变都会影响实参发生同样的改变。鸡啄米再拿上面那个例子改成引用调用说明下：

```
#include<iostream>
using namespace std;
void Swap(int& a, int& b);
int _tmain(int argc, _TCHAR* argv[])
{
        int x=5, y=10;
        cout<<"x="<<x<<"    y="<<y<<endl;
        Swap(x,y);
        cout<<"x="<<x<<"    y="<<y<<endl;
        return 0;
}
void Swap(int& a, int& b)
{
        int t;
        t=a;
        a=b;
        b=t;
}
```

运行结果就跟我们想的一样了：

```
x=5     y=10
x=10    y=5
```

引用调用后 x 和 y 的值成功的交换了。值调用和引用调用的区别只是函数的形参写法不同，主调函数调用被调函数时的调用语句是一样的。

二.带默认形参值的函数

函数在声明时可以预先定义默认的形参值。调用时若给出实参则用实参初始化形参，如果没有给出实参则采用预先定义的默认形参值。例如：

```
int add(int x=5,int y=6)      // 定义默认形参值
{
        return x+y;
}
int main()
{
        add(10,20);           // 用实参来初始化形参，实现 10+20
        add(10);              // 形参 x 采用实参值 10，y 采用默认值 6，实现 10+6
        add();                // x 和 y 都采用默认值，分别为 5 和 6，实现 5+6
        return 0;
}
```

默认参数值必须按照从右向左的顺序定义。在有默认值的形参右面，不能出现无默认值的形参。也就是说应该把有默认值的形参都一块放到右边，不能让有默认值的跟没默认值的形参穿插着放。因为在调用时，实参初始化形参是按从左向右的顺序。比如：

```
int add(int x,int y=5,int z=6);   //正确
int add(int x=1,int y=5,int z);   //错误
int add(int x=1,int y,int z=6);   //错误
```

调用出现在函数体实现之前时，默认形参值必须在函数原型中也就是声明时给出；而当调用出现在函数体实现之后时，默认形参值需在函数实现时给出。例如：

调用在实现前时：

```
int add(int x=5,int y=6);
int main()
{
        add(); //调用在实现前
        return 0;
}
int add(int x,int y)
{
        return x+y;
}
```

调用在实现后时：

```
int add(int x=5,int y=6)
{
    return x+y;
}
int main()
{
    add(); //调用在实现后
    return 0;
}
```

在相同的作用域内，默认形参值的说明应保持惟一，但如果在不同的作用域内，允许说明不同的默认形参。这里的作用域是指直接包含函数原型说明的大括号所界定的范围，具体鸡啄米会在后面跟大家讲。例：

```
int add(int x=1,int y=2);
int main()
{
        int add(int x=3,int y=4);
        add();//使用局部默认形参值（实现 3+4）
        return 0;
}
void fun()
{  ...
       add();//使用全局默认形参值（实现 1+2）
}
```

三.内联函数

函数虽然有很多优点，比如代码复用，便于维护等。但是调用函数时需要保存现场和返回地址，被调函数执行完后还要取出这些值继续执行，这些过程在时间和空间方面都有开销。对于一些规模小、功能简单的函数可以定义成内联函数。内联函数在调用时不需要那些转移带来的开销，它是在编译的时候把函数体代码嵌入到所有调用它的语句处，我们可以认为直接把函数体的代码放那里了，当然也不完全一样，毕竟它可能有参数。

内联函数定义时使用关键字 inline，语法形式如下：

```
inline  类型标识符  被调函数名(含类型说明的形参表)
```

鸡啄米提醒大家使用内联函数时必须注意：**内联函数体内不能有循环语句和 switch 语句；内联函数的定义必须出现在内联函数第一次被调用之前；对内联函数不能进行异常接口声明，就是不能声明可能抛出的异常，**异常接口声明鸡啄米以后会讲到。

内联函数应该是语句比较少、结构比较简单的函数，不应该是复杂的函数，因为它对编译器来说就是代码，如果很复杂会造成代码膨胀，反而增大开销，这种情况下其实多数编译器就都会自动把它作为普通函数来处理了。一般来说，包含循环语句的函数一定不能定义成内联函数。鸡啄米再给大家一个内联函数的例子：

```
#include<iostream>
using namespace std;
inline double CalArea(double radius)
{
    return 3.14*radius*radius;
}
int _tmain(int argc, _TCHAR* argv[])
{
    double r(3.0);
    double area;
    area=CalArea(r);
    cout<<area<<endl;
    return 0;
}
```

运行结果是：28.26

这一讲鸡啄米给大家的源代码的例子比较多，大家看看这些代码就会更好的理解鸡啄米所讲的内容，希望大家在编程入门的时候不要着急，好好消化教程。

鸡啄米：C++编程入门系列之十一（重载函数与函数模板）

hello，everyone，鸡啄米上一节讲到的是函数的参数传递和内联函数，这一讲鸡啄米会给大家讲点稍微有点难度的函数重载和函数模板，如果大家把之前的编程入门课程都掌握了，这一讲的内容自己上机试下就很容易理解了。

一.重载函数

现实生活中的一个词可能有多种含义，比如，洗衣服、洗头、洗车，都有一个洗字，但是他们的操作方式是不一样的。函数也一样，有时候它们的操作不完全一样但是名字一样，这就是重载函数。

重载函数就是，两个以上的函数取相同的函数名，但是函数形参的个数或者类型不同，编译器会根据实参与形参的类型和个数进行最佳匹配，自动确定调用哪一个函数。

为什么要有重载函数呢？因为如果没有重载函数，那么对不同类型的数据进行类似的操作也要定义不同名称的函数，比如加法函数，就必须对整数加法和浮点数加法分别定义不同的函数名：

```
int nadd(int a, int b);
float fadd(float a, float b);
```

这样调用需要记住的函数名太多，而且功能类似，很不方便。鸡啄米给大家举几个重载函数的例子：

形参类型不同的例子：

```
int   add(int x,int y);
float add(float x,float y);
```

形参个数不同的例子：

```
int add(int x,int y);
int add(int x,int y,int z);
```

鸡啄米在这里要提醒几个需要注意的地方：

1.重载函数的形参不管是**类型**还是**个数**必须有一样是不同的。因为编译器就是看实参和哪个函数的形参的类型及个数匹配，来判断调用哪个函数，如果函数名、形参类型和个数相同，即使函数返回值类型不同，编译器也会认为是函数重复定义的语法错误，就是说它认为是一个函数。以下两种是错误的重载函数：

```
int add(int x,int y);
int add(int a,int b);
```

上面两个函数虽然形参名不同，但是编译器不会以形参名来区分函数，它会认为这是一个函数。我们以后也要记住，若只改变形参名则函数不会变，调用的地方也不必变。

```
int add(int x,int y);
void add(int x,int y);
```

上面这两个函数返回值不同，确实是两个函数，但是编译器也不会以返回值来区分函数，也会认为是重复定义。

2.重载函数都是进行类似的操作，不要把不同的功能定义成重载函数，否则会让人对调用有误解，比如：

```
int add(int x,int y)
{
        return x+y;
}
float add(float x,float y)
{
```

```
        return x-y;
    }
```

这两个函数一个是实现的两个数的加法，一个是实现减法，在语法上并没有问题。但是功能不一样但是都叫 add，我们调用的时候是不是会混淆？

鸡啄米给大家一个重载函数的例子，大家仔细研究下代码，上机试下，通过上面的讲解，很容易理解：

```
#include<iostream>
using namespace std;
int add(int m,int n);      // 函数调用前必须先声明函数原型
float add(float x,float y);

int _tmain(int argc, _TCHAR* argv[])
{
        int m, n;
        float x, y;
        cout<<"Enter two integer: ";
        cin>>m>>n;
        cout<<"integer <<m<<'+'<<n<<"="<<add(m,n)<<endl;
        cout<<"Enter two float: ";
        cin>>x>>y;
        cout<<"float "<<x<<'+'<<y<<"= "<<add(x,y)<<endl;
        return 0;
}
int add(int m,int n)
{
        return m+n;
}
float add(float x,float y)
{
       return x+y;
}
```

屏幕先输出 Enter two integer:，我们输入 2 3，则屏幕会接着显示 integer 2+3=5，然后继续提示 Enter two float:，我们继续输入 2.1 3.4，最后屏幕会出现 float 2.1+3.4=5.5。

二.函数模板

有时候我们使用重载函数还不能达到最优的效果，比如，两个求绝对值的函数：

```
int abs(int x)
{
        return x<0 ? -x:x;
}
double abs(double x)
{
       return x<0 ? -x:x;
}
```

大家观察下这两个函数，这两个函数只是返回值类型和参数类型不同，功能完全一样，如果能有办法写一段通用的代码适用于多种不同的数据类型，就是不用像上面那样写两个函数而只是一段代码就能实现两个函数的功能，那代码的复用性不是更高了吗？开发效率也会提高的。这就要函数模板来实现了。

模板是由可以使用和操作任何数据类型的通用代码构成，这种程序设计叫做参数化程序设计，因为它把数据类型当成了参数，可以用来创建一个通用功能的函数，支持多种不同类型的形参和返回值。函数模板的定义形式是：

```
template <typename 标识符>
函数定义
```

鸡啄米给大家一个上面那样求绝对值的函数模板示例：

```
#include<iostream>
using namespace std;
template <typename T>
T abs(T x)
{
        return x<0 ? -x:x;
}


int _tmain(int argc, _TCHAR* argv[])
{
       int     n = 5;
       double  d = -2.3;
       cout << abs(n) << endl;
       cout << abs(d) << endl;
```

```
        return 0;
}
```

鸡啄米给大家分析下这个程序，编译器会根据调用 abs 函数时传入实参的类型来确定函数模板的类型参数是什么类型。上面使用调用表达式 abs(n)时，因为 n 是 int 类型，所以模板中类型参数 T 就是 int，然后编译器会以函数模板为样板生成一个函数：

```
int abs(int x)
{
        return x<0 ? -x:x;
}
```

同样，对于调用表达式 abs(d)，因为 d 是 double 类型，所以类型参数 T 就是 double，编译器会根据函数模板生成下面的函数：

```
double abs(double x)
{
      return x<0 ? -x:x;
}
```

所以，abs(n)实际上调用的是函数模板生成的函数：int abs(int x)。而 abs(d)调用的是由函数模板生成的函数：double abs(double x)。

最后，鸡啄米跟大家简单说下 C++系统函数。C++系统库中提供了几百个函数供我们使用，我们只要用 include 指令包含相应的头文件就可以使用了。比如，包含了 math.h 文件我们就可以使用 abs、sqrt、sin 等函数。如果需要用到一些库函数可以 msdn 上查也可以上网查。

好了，今天鸡啄米就讲到这里了，重点掌握下重载函数吧，函数模板可能大家现在还不能很好的理解，不过可以先了解下，以后你会理解的。

2013.6.24

第四部分：类与对象

鸡啄米：C++编程入门系列之十二（类与对象：面向对象设计的基本思想和特点）

上一讲鸡啄米讲了函数重载和函数模板以后，C++和 C 语言相似的语法就剩下后面要讲的数组了，这一讲开始鸡啄米就要讲 C++所特有的一些概念和思想了。本节编程入门教程主要讲面向对象程序设计的基本思想和特点。另外，说明一点，以后鸡啄米会加一个大的教程

分类，让大家更好的对琐碎的知识进行分类，比如这一讲的大的分类叫“类与对象”，标题是面向对象设计的基本思想和特点。

在编程入门教程的前言中，鸡啄米已经简单讲到了面向对象程序设计的几个基本概念，这一讲将做更深入的讨论。

计算机技术飞速发展，软件开发技术也同样经历了各种各样的变化，从早期的面向过程的结构化程序设计，到面向对象的程序设计，其实都是在寻找一种更能够提高可扩展性和可复用性的程序设计方法，这样才能更大限度的提高软件开发效率。现在的面向对象程序设计方法（OOP）通过抽象、封装、继承等手段，已经成为目前最能提高可扩展性和可复用性的程序设计方法，能够最大限度的提高软件生产能力，降低软件开发和维护的费用。

在面向对象程序设计方法之前普遍使用的是面向过程的结构化程序设计方法，它的主要思想就是，将一个复杂问题根据功能等分成一个个子问题，然后再接着细分，直到分解成具体的语句，这种方法固然有很多优点，但是有个比较大的缺点就是，这种方法中数据和函数是分开的，如果我们要修改数据结构，则有关的函数都必须修改，这样不但维护成本比较大，而且很容易遗漏一些应该修改的地方。

面向对象程序设计方法是对面向过程程序设计方法的继承和发展。这种程序设计方法认为，现实世界是由一些互相关联的实体组成的，这些实体就是面向对象方法中的对象，而对一些对象的共性的抽象描述，就是面向对象程序设计方法中最核心的概念--类。面向对象的程序设计方法就是运用面向对象的思维来描述现实问题，再用计算机语言解决该问题，这里的解决就是靠类和对象实现的，是对现实问题的高度概括、分类和抽象。

鸡啄米下面讲下面向对象程序设计方法的基本特点：**抽象、封装、继承和多态。**

1.抽象

面向对象方法中的抽象是指对具体问题即对象进行概括，抽出一类对象的共性并加以描述的过程。面向对象的软件开发中，首先应该对要解决的问题抽象成类，然后才是解决问题的过程。抽象有两个方面：数据抽象和行为抽象。数据抽象是描述某类对象的属性或状态，，行为抽象是描述某类对象的共同行为或共同功能。

鸡啄米就拿时钟给大家举个例子，要实现有关时钟的程序，首先要对时钟进行抽象。时钟有时、分、秒，我们用三个整型变量来存储，这就是数据抽象。时钟有显示时间和设置时间等功能，这就是行为抽象。用 C++语言描述就是：

时钟（Clock）：
数据抽象：
int Hout; int Minute; int Second;
行为抽象：
ShowTime(); SetTime();

上面并不是真正的 C++代码，只是简单的列出了数据成员和函数成员的代码片段。

2.封装

把抽象出来的数据成员和函数成员结合形成一个整体，就是封装。封装的时候，我们可以把一些成员作为类和外界的接口，把其他的成员隐藏起来，以达到对数据访问权限的控制，这样可以使程序的各个部分改变时最低程度的影响其他部分，程序会更安全。

把数据和函数封装为一个可复用的模块，开发时可以利用已有的成果而不必每次都重复编写。我们只需要通过类提供的外部接口访问模块，并不需要知道内部的细节。C++中就是利用类的形式来实现封装的。下面是鸡啄米上面提到的时钟的具体封装形式及时钟类：

```
class  Clock                                    // class 是关键字  Clock 是类名
{
public:                                          // 提示下面是外部接口
    void  SetTime(int  NewH,int  NewM,int  NewS);     // 行为，函数成员
    void  ShowTime();                            // 行为，函数成员
private:                                         // 特定的访问权限
    int  Hour,Minute,Second;                     // 属性，数据成员
};
```

这是一个完整的类的声明。它声明了一个名为 Clock 的类，其中的数据成员和函数成员是前面分析得到的抽象结果。关键字 public 和 private 是用来指定成员的不同访问权限的，至于具体访问权限的问题后面课程中会讲到。声明为 public 的两个函数为类提供了外部接口，外界只能通过这两个接口跟 Clock 类联系。声明为 private 的三个整型数据是类的私有数据，外部无法直接访问。我们可以看到，这种访问权限的机制有效实现了对数据的隐藏。

3.继承

我们在软件开发过程中，可能已经有了前人的一些现有的成果，我们没有必要再重新去编写，那么我们怎样利用这些已有的模块呢？还有可能我们对以前写的程序有了更新的认识，需要融入一些新的认识，那怎么办呢？

这些都可以通过继承来实现，C++语言提供了类的继承机制，让我们软件开发者可以在保持原有特性的基础上，进行更具体、更详细的说明。通过继承我们可以利用之前已经有的程序模块，还可以添加一些新的数据和行为，这在很大程度上提高了程序的复用性，大大节约开发成本。关于继承，后面的课程也会具体讲。

4.多态

多态就是类中具有相似功能的不同函数使用同一个名称。上一讲中讲的重载函数就实现了多态。利用多态可以对类的行为再抽象，抽象成同一个名称的功能相似的函数，减少程

序中标识符的个数。多态是通过重载函数和虚函数等技术来实现的，虚函数后面也会详细介绍。

这一讲鸡啄米认为，是属于软件开发思想方面的东西，在编程入门时可能不是很好理解，但是大家能掌握多少就尽量掌握吧，在以后大家真的实践了一些方法再来理解的话就很简单了，一句话，不要认为它有多难！

鸡啄米：C++编程入门系列之十三（类与对象：类的声明、成员的访问控制和对象）

鸡啄米上讲中介绍了面向对象设计的基本思想和特点，主要是让大家在编程入门时于思维中初步建立面向对象设计的概念。这一讲开始讲面向对象设计中最重要的概念--类，及由类引申出来的一些知识。

在面向过程的设计中，程序的模块是函数构成的，而面向对象设计中程序模块是类构成的。函数只是语句和数据的封装，而类是函数与数据的封装，对比下肯定是面向对象设计更重量级了，更适合大型程序的开发。

其实，类就是一种自定义数据类型，跟一般的类型如 int、char 等有很多相似之处。我们可以定义 int 型的变量，同样也可以定义某个类类型的变量，用类定义的变量叫做类的对象，这种定义对象的过程叫做实例化。

1.类的声明

类声明的语法形式如下：

```
class 类名称
{
public:
        公有成员（外部接口）
protected:
        保护型成员
private:
        私有成员
}
```

当然，这里的 public、protected 和 private 关键字可以任意换顺序，比如先声明私有成员再声明其他的也可以，每个关键字也可以出现多次，比如声明一些 public 的成员，后面又出了个 public 声明了另一些成员，也是可以的，但是一般我们还是按照上面的形式来声明类。

这里还是拿时钟当例子，声明一个类来描述时钟：

```
class  Clock
{
public:
        void  SetTime(int  NewH,int  NewM,int  NewS);
        void  ShowTime();
private:
        int   Hour,Minute,Second;
};
```

类 Clock 封装了时钟的数据和行为，分别叫 Clock 类的数据成员和函数成员。在类的声明中只声明函数的原型，函数的实现也就是函数体可以在类外定义，当然也可以写在类声明里，那样就成为隐式声明的内联函数。后面会讲。下面是在类外写函数实现的方式：

```
void  Clock::SetTime(int  NewH,  int  NewM,  int  NewS)
{
        Hour=NewH;
        Minute=NewM;
        Second=NewS;
}
void  Clock::ShowTime()
{
        cout<<Hour<<":"<<Minute<<":"<<Second<<endl;
}
```

注意：函数名前面要加上它所属的类，用来说明它属于哪个类.

这样就完成了 Clock 类的声明。首先用关键字 class 声明类的名称 Clock，然后说明 Clock 类的数据成员和函数成员，通过 public、private 关键字来说明类成员的访问控制属性，最后给出成员函数的实现.

2.类成员的访问控制

我们再通过时钟来看看访问控制机制。正常使用时我们只能通过时钟面板来看时间，通过按钮或旋钮来调整时间，所以，面板和按钮或旋钮就是时钟的外部接口，时钟类 Clock 的 SetTime 和 ShowTime 就是 Clock 类的外部接口。除了这样使用时钟，我们不能拆开时钟在内部直接改时间，那样会弄坏时钟的，所以时钟的时间是私有成员，Clock 类的 Hour、Minute、Second 都是其私有成员，外部不能直接访问，只能通过外部接口来访问。

类成员访问权限的控制是通过设置成员的访问控制属性来实现的。访问控制属性有三种：公有类型（public）、私有类型（private）和保护类型（protected)。

公有类型声明了类的外部接口。公有类型成员用 public 关键字声明。外部访问必须通过外部接口进行。比如，对于 Clock 类，外部想查看或改变时间只能通过 SetTime 和 ShowTime 两个公有类型的函数实现。

private 后面声明的是类的私有类型成员。如果没有标明访问控制属性则默认为 private。比如，类 Clock 声明中，如果那个 public 没有的话，那么 SetTime 和 ShowTime 函数就都默认是 private 的。私有类型成员只能由本类中的成员函数访问，外部不能访问。Clock 类中 Hour、Minute、Second 都是私有类型成员。

保护类型的成员和私有类型成员权限相似，差别就是某个类派生的子类中的函数能够访问它的保护成员，这个后面会讲到。

3.类的成员函数

函数原型的声明要写在类主体中，原型说明了函数的参数类型和个数及返回值类型。而函数的具体实现是类声明之外的，但是跟普通函数不同的是，写函数实现时要在前面加上类名，指明所属的类。具体的形式为：

```
返回值类型  类名::函数成员名(参数表)
{
        函数体
}
```

类的成员函数也可以由默认参数值，它的调用规则跟之前讲的普通函数相同。

类的比较简单的成员函数也可以声明为内联函数，和普通内联函数一样，编译时也会将内联函数的函数体插入到每个调用它的地方。内联函数的声明有两种方式：**隐式声明和显式声明。**

把函数体直接放到类主体内，这种方式就是隐式声明。比如，将时钟类的 ShowTime 声明为内联函数，可以写成：

```
class  Clock
{
public:
        void  SetTime(int  NewH,int  NewM,int  NewS);
        void  ShowTime()
        { cout<<Hour<<":"<<Minute<<":"<<Second<<endl;
}
private:
```

```
        int  Hour,Minute,Second;
};
```

为了程序的可读性，让大家一看就知道是内联函数，一般还是用关键字 inline 显式声明。就像普通内联函数那样，在函数实现时在函数返回值类型前加上 inline，声明中不加入函数体，还是拿时钟类的 ShowTime 举例：

```
inline  void  Clock::ShowTime()
{
        cout<<Hour<<":"<<Minute<<":"<<Second<<endl;
}
```

效果上，显式声明和隐式声明内联函数的效果是完全一样的。

4.对象

类的对象就是具有该类类型的特定实体。就像一般类型的变量一样，类是自定义数据类型，对象就是该类类型的变量。声明一个对象和声明变量的方式是一样的: 类名 对象名;。比如，声明一个时钟类的对象：Clock myClock;。声明了对象后就可以访问对象的公有成员，这种访问需要采用“.”操作符，调用公有成员函数的一般形式是：对象名.公有成员函数名(参数表)，访问公有数据成员的形式是：对象名.公有数据成员。当然，一般数据是私有类型的，但是也不排除有时是公有类型的。例如，可以用 myClock.ShowTime()的形式访问对象 myClock 的成员函数 ShowTime()。

鸡啄米给大家一个完整的程序举例：

```
#include<iostream>
using  namespace  std;
//  第一部分
class  Clock
{
public:
        void  SetTime(int  NewH,int  NewM,int  NewS);
        void  ShowTime();
private:
        int   Hour,Minute,Second;
};
//  第二部分
void  Clock::SetTime(int  NewH,  int  NewM,  int  NewS)
{
        Hour=NewH;
        Minute=NewM;
        Second=NewS;
```

```
}
void  Clock::ShowTime()
{
        cout<<Hour<<":"<<Minute<<":"<<Second<<endl;
}
// 第三部分
int  _tmain(int  argc,  _TCHAR*  argv[])
{
        Clock   myClock;
        myClock.SetTime(8,30,30);
        myClock.ShowTime();
        return  0;
}
```

本程序就像注释的那样可以分为三个比较独立的部分，第一个部分是类 Clock 的定义，第二个部分是成员函数的具体实现，第三个部分是主函数 main。大家可以在 vs2010 上运行这个程序，然后加些语句实现自己想要的一些其他功能。

鸡啄米这节给大家讲了类的声明、类成员的访问控制、类的成员函数及对象，内容不少，大家稳住心态慢慢理解，有问题在鸡啄米博客留言讨论吧。

鸡啄米：C++编程入门系列之十四（类与对象：构造函数和析构函数）

鸡啄米上一节中给大家讲解了类的声明、成员的访问控制和对象，今天鸡啄米给大家讲 C++编程入门时同样必须掌握的构造函数和析构函数。从上一讲开始已经涉及到了很多面向对象设计的细节，大家慢慢跟着学习思考吧，实际上跟我们现实中的很多做事的思想是一致的，这也正是面向对象语言的特点，它是以现实中的事物和围绕事物处理问题的思路为基础的。

某个类的对象之间都有哪些不同呢？首先是对象名不同，其次就是对象的数据成员的值不同。我们在声明一个对象时，也可以同时给它的数据成员赋初值，称为对象的初始化。

1.构造函数

我们在声明一个变量时，如果对它进行了初始化，那么在为此变量分配内存空间时还会向内存单元中写入变量的初始化。声明对象有相似的过程，程序执行时遇到对象声明语句时会向操作系统申请一定的内存空间来存放这个对象，但是它能像一般变量那样初始化时写入指定的初始值吗？类的对象太复杂了，要实现这一点不太容易，这就需要构造函数来实现。

构造函数的作用就是在对象被创建时利用特定的初始值构造对象，把对象置于某一个初始状态，它在对象被创建的时候由系统自动调用，我们只需要使用默认的构造函数或者自己定义构造函数，而不用管怎么调用的。

构造函数也是类的成员函数，除了有成员函数的所有特征外，还有一些不同之处：（1）**构造函数的函数名跟类名一样，而且没有返回值。**（2）构造函数一般被声明为公有函数，除非我们不允许某个类生成对象则将它声明为 private 或 protected 属性。编译器碰到对象声明语句时，会自动生成对构造函数的调用语句，所以我们常说构造函数是在对象声明时由系统自动调用的。

上一讲那个时钟例子中，鸡啄米没有定义 Clock 类的构造函数，编译器编译时会自动生成一个默认形式的构造函数，这个构造函数不做任何事，那么为什么还要生成它呢？因为C++在对象建立时都会调用构造函数，所以如果没有自己定义构造函数，那么即使是什么都不做的构造函数也是要有的。鸡啄米现在在 Clock 类中加入自己定义的构造函数：

```
class Clock
{
public:
    Clock(int NewH, int NewM, int NewS);          //构造函数
    void SetTime(int NewH, int NewM, int NewS);
    void ShowTime();
private:
    int Hour, Minute, Second;
};
```

构造函数的实现：

```
Clock::Clock(int NewH, int NewM, int NewS)
{
    Hour=NewH;
    Minute=NewM;
    Second=NewS;
}
```

建立对象时构造函数的作用：

```
int main()
{
    Clock c(0,0,0); //隐含调用构造函数，将初始值作为实参。
    c.ShowTime();
    return 0;
}
```

main 函数中，创建对象 c 时，其实隐含了构造函数的调用，将初始值 0,0,0 作为构造函数的实参传入。因为上面 Clock 类定义了构造函数，那么编译器就不会再为它生成默认构造函数了。这里的构造函数有三个形参，那么建立对象就必须给出初始值了。

因为构造函数也是一个成员函数，所以它可以直接访问类的所有数据成员，可以是内联函数，可以带有形参表，可以带默认的形参值，也可以重载，就是有若干个名字相同但形参个数或者类型不同的构造函数。

2.拷贝构造函数

我们可以将一个变量的值赋给另一个同类型的变量，那么可以将一个对象的内容拷贝给相同类的另一个对象吗？可以，我们可以将第一个对象的数据变量的值分别赋给另一个对象的数据变量，但是，如果数据变量数很多的话那将是很麻烦的，这时候我们就需要有拷贝构造函数。

拷贝构造函数是一种特殊的构造函数，因为它也是用来构造对象的。它具有构造函数的所有特性。**拷贝构造函数的作用是用一个已经存在的对象去初始化另一个对象**，这两个对象的类类型应该是一样的。定义拷贝构造函数的形式是：

```
class 类名
{
 public :
        类名(形参);                  //构造函数
        类名(类名 &对象名);    //拷贝构造函数
    ...
};
类名::类(类名 &对象名)    //拷贝构造函数的实现
 {
       函数体
}
```

拷贝构造函数的形参是本类的对象的引用。

程序中如果没有定义拷贝构造函数系统会生成一个默认的拷贝构造函数，它会将作为初始值的对象的数据成员的值都拷贝到要初始化的对象中。下面鸡啄米给大家一个坐标点类的例子，X 和 Y 数据成员分别为点的横坐标和纵坐标：

```
class Point
{
public:
        Point(int xx=0,int yy=0)    {X=xx; Y=yy;}
        Point(Point &p);
        int GetX() {return X;}
        int GetY() {return Y;}
private:
        int X, Y;
};
```

此类中声明了内联构造函数和拷贝构造函数。拷贝构造函数的实现如下：

```
Point::Point(Point &p)
{
        X=p.X;
        Y=p.Y;
        cout<<"拷贝构造函数被调用"<<endl;
}
```

拷贝构造函数在以下三种情况下会被调用：

a.当用类的一个对象去初始化该类的另一个对象时系统自动调用拷贝构造函数实现拷贝赋值。

```
int main()
{
        Point A(1,2);
        Point B(A); //拷贝构造函数被调用
        cout<<B.GetX()<<endl;
        return 0;
}
```

b.若函数的形参为类对象，调用函数时，实参赋值给形参，系统自动调用拷贝构造函数。例如：

```
void fun1(Point p)
{
        cout<<p.GetX()<<endl;
}
int main()
{
        Point A(1,2);
        fun1(A); //调用拷贝构造函数
        return 0;
}
```

c.当函数的返回值是类对象时，系统自动调用拷贝构造函数。例如：

```
Point fun2()
{
      Point A(1,2);
      return A; //调用拷贝构造函数
}
int main()
{
     Point B;
```

```
    B=fun2();
    return  0;
}
```

最后这种情况怎么调用的拷贝构造函数呢？对象 A 是局部对象，在 fun2 函数执行完就释放了，那怎么将它拷贝给对象 B 呢？编译器在执行 B=fun2()时会创建一个临时的无名对象，在执行 return A 时实际上是调用了拷贝构造函数将 A 的值拷贝到了临时对象中，A 就释放了，然后将临时对象的值再拷贝到对象 B 中。

3.析构函数

自然万物都是有生有灭的，类的对象也一样是有生命周期的，一样会消亡。鸡啄米给大家讲一种情况：如果在函数中声明了一个对象，那么在这个函数运行完返回调用函数时，声明的对象也会释放，就像上面说的 fun2 函数中对象 A 那样。

在对象释放时都有什么工作要做呢？我们经常遇到的情况就是：构造函数时动态申请了一些内存单元，在对象释放时就要同时释放这些内存单元。动态分配内存的知识后面鸡啄米会讲。

析构函数和构造函数的作用是相反的，**它会在对象被删除之前做一些清理工作。析构函数是在对象要被删除时由系统自动调用的**，它执行完后对象就消失了，分配的内存空间也释放了。

析构函数是类的一个公有函数成员，它的名称是在类名前加“~”形成，不能有返回值，大家注意下，它和构造函数不同的是它不能有任何形参。如果没有定义析构函数系统也会自动生成一个默认析构函数，默认析构函数也不会做任何工作。一般如果我们想在对象被删除之前做什么工作就可以把它写到析构函数里。

鸡啄米还是在上面那个坐标点类中加入析构函数给大家看下析构函数怎么用：

```
class  Point
{
public:
        Point(int  xx,  int  yy);
        ~Point();
        //...其他函数原型
private:
       int  X,  int  Y;
       char  *p;
};
```

下面是构造函数和析构函数的实现：

```
Point::Point(int  xx,int  yy)
{
        X=xx;
        Y=yy;
        p=new  char[20];      //  构造函数中动态分配 char 型内存
}
Point::~Point()
{
        delete  []char;       //  在类析构时释放之前动态分配的内存
}
//...其他函数的实现略
```

new 和 delete 的用法鸡啄米后面会讲，这里只是让大家看下析构函数有什么作用。

构造函数和析构函数是大家 C++编程入门时必须掌握的内容，很多公司笔试或面试时都会考到，大家掌握好啊，呵呵。鸡啄米谢谢大家关注我的教程！我们一起努力！

鸡啄米：C++编程入门系列之十五（类与对象：类的组合）

上一讲鸡啄米给大家讲了构造函数和析构函数，让大家了解了类的创建和释放过程。这一节讲讲类的组合。

在我们对现实中的某些事物抽象成类时，可能会形成很复杂的类，为了更简洁的进行软件开发，我们经常把其中相对比较独立的部分拿出来定义成一个个简单的类，这些比较简单的类又可以分出更简单的类，最后由这些简单的类再组成我们想要的类。比如，我们想要创建一个计算机系统的类，首先计算机由硬件和软件组成，硬件又分为 CPU、存储器等，软件分为系统软件和应用软件，如果我们直接创建这个类是不是很复杂？这时候我们就可以将 CPU 写一个类，存储器写一个类，其他硬件每个都写一个类，硬件类就是所有这些类的组合，软件也是一样，也能做成一个类的组合。计算机类又是硬件类和软件类的组合。

类的组合其实描述的就是在一个类里内嵌了其他类的对象作为成员的情况，它们之间的关系是一种包含与被包含的关系。简单说，一个类中有若干数据成员是其他类的对象。以前的教程中我们看到的类的数据成员都是基本数据类型的或自定义数据类型的，比如 int、float 类型的或结构体类型的，现在我们知道了，数据成员也可以是类类型的。

如果在一个类中内嵌了其他类的对象，那么创建这个类的对象时，其中的内嵌对象也会被自动创建。因为内嵌对象是组合类的对象的一部分，所以在构造组合类的对象时不但要对基本数据类型的成员进行初始化，还要对内嵌对象成员进行初始化。

组合类构造函数定义（注意不是声明）的一般形式为：

```
类名::类名(形参表):内嵌对象 1(形参表),内嵌对象 2(形参表),...
{
```

```
        类的初始化
}
```

其中，“内嵌对象 1(形参表),内嵌对象 2(形参表),...”称为初始化列表，可以用于完成对内嵌对象的初始化。其实，一般的数据成员也可以这样初始化，就是把这里的内嵌对象都换成一般的数据成员，后面的形参表换成用来初始化一般数据成员的变量形参，比如，Point::Point(int xx, int yy):X(xx),Y(yy) { }，这个定义应该怎么理解呢？就是我们在构造 Point 类的对象时传入实参初始化 xx 和 yy，然后用 xx 的值初始化 Point 类的数据成员 X，用 yy 的值初始化数据成员 Y。

声明一个组合类的对象时，不仅它自身的构造函数会被调用，还会调用其内嵌对象的构造函数。那么，这些构造函数的调用是什么顺序呢？**首先，根据前面说的初始化列表，按照内嵌对象在组合类的声明中出现的次序，依次调用内嵌对象的构造函数，然后再执行本类的构造函数的函数体。**比如下面例子中对于 Distance 类中的 p1 和 p2 就是先调用 p1 的构造函数，再调用 p2 的构造函数，因为 Point p1,p2;是先声明的 p1 后声明的 p2。最后才是执行 Distance 构造函数的函数体。

如果声明组合类的对象时没有指定对象的初始值的话，就会自动调用无形参的构造函数，构造内嵌对象时也会对应的调用内嵌对象的无形参的构造函数。**析构函数的执行顺序与构造函数正好相反。**

这里鸡啄米给大家一个类的组合的例子，其中，Distance 类就是组合类，可以计算两个点的距离，它包含了 Point 类的两个对象 p1 和 p2。

```
#include <iostream>
using namespace std;
class Point
{
public:
        Point(int xx,int yy)  { X=xx; Y=yy; } //构造函数
        Point(Point &p);
        int GetX(void)    { return X; }       //取 X 坐标
        int GetY(void)    { return Y; } //取 Y 坐标
private:
        int X,Y; //点的坐标
};
Point::Point(Point &p)
{
        X = p.X;
        Y = p.Y;
        cout << "Point 拷贝构造函数被调用" << endl;
}
```

```
class Distance
{
public:
    Distance(Point a,Point b); //构造函数
    double GetDis()  { return dist; }
private:
    Point p1,p2;
    double dist;          // 距离
};
// 组合类的构造函数
Distance::Distance(Point a, Point b):p1(a),p2(b)
{
    cout << "Distance 构造函数被调用" << endl;
    double x = double(p1.GetX() - p2.GetX());
    double y = double(p1.GetY() - p2.GetY());
    dist = sqrt(x*x + y*y);
    return;
}
int _tmain(int argc, _TCHAR* argv[])
{
    Point myp1(1,1), myp2(4,5);
    Distance myd(myp1, myp2);
    cout << "The distance is:";
    cout << myd.GetDis() << endl;
    return 0;
}
```

这段程序的运行结果是：

```
Point 拷贝构造函数被调用
Point 拷贝构造函数被调用
Point 拷贝构造函数被调用
Point 拷贝构造函数被调用
Distance 构造函数被调用
The distance is:5
```

Point 类的构造函数是内联成员函数，内联成员函数在[鸡啄米：C++编程入门系列之十三（类与对象：类的声明、成员的访问控制和对象）]中已经讲过。

鸡啄米给大家分析下这个程序，首先生成两个 Point 类的对象，然后构造 Distance 类的对象 myd，最后输出两点的距离。Point 类的拷贝构造函数被调用了 4 次，而且都是在 Distance 类构造函数执行之前进行的，在 Distance 构造函数进行实参和形参的结合时，也就是传入 myp1 和 myp2 的值时调用了两次，在用传入的值初始化内嵌对象 p1 和 p2 时又调

用了两次。两点的距离在 Distance 的构造函数中计算出来，存放在其私有数据成员 dist 中，只能通过公有成员函数 GetDis()来访问。

鸡啄米再跟大家说下类组合时的一种特殊情况，就是两个类可能相互包含，即类 A 中有类 B 类型的内嵌对象，类 B 中也有 A 类型的内嵌对象。我们知道，C++中，要使用一个类必须在使用前已经声明了该类，但是两个类互相包含时就肯定有一个类在定义之前就被引用了，这时候怎么办呢？就要用到**前向引用声明**了。前向引用声明是在引用没有定义的类之前对该类进行声明，这只是为程序声明一个代表该类的标识符，类的具体定义可以在程序的其他地方，简单说，就是声明下这个标识符是个类，它的定义你可以在别的地方找到。

比如，类 A 的公有成员函数 f 的形参是类 B 的对象，同时类 B 的公有成员函数 g 的形参是类 A 的对象，这时就必须使用前向引用声明：

```
class  B;   //前向引用声明
class  A
{
public:
        void  f(B  b);
};
class  B
{
public:
        void  g(A  a);
};
```

这段程序的第一行给出了类 B 的前向引用声明，说明 B 是一个类，它具有类的所有属性，具体的定义在其他地方。

今天鸡啄米讲了类的组合，这个知识还是很实用的，可以让我们的程序更灵活，不至于太庞杂让人摸不着头脑。面向对象是不是很强大啊？希望大家从思想上领会它。

鸡啄米：C++编程入门系列之十六（类与对象：类模板）

鸡啄米上一节中给大家讲了类的组合的相关知识，这一讲跟大家介绍下类模板。类模板的指导思想跟函数模板类似。

代码复用是面向对象设计中的重要的软件开发思想，对于软件开发效率很是关键。怎样做好代码复用呢？越是通用的代码越好复用，将类型作为参数，这种程序设计类型就是参数化程序设计。模板就是 C++进行参数化设计的工具。利用模板我们可以使用同一段程序处理不同类型的对象。

什么是类模板呢？**类模板就是为类声明一种模板，使得类中的某些数据成员，或某些成员函数的参数，又或者是某些成员函数的返回值可以取任意的数据类型，包括基本数据类型和自定义数据类型。**

类模板的声明形式如下：

template <模板参数表>
类声明

我们看到，在类的声明之前要加上一个模板参数表，模板参数表里的类型名用来说明成员数据和成员函数的类型，等会可以看下面的例子。

模板参数表中可以以下两种模板参数：

1.class 标识符（指明可以接受一个类型参数，就是说这是个不固定的类型，用它生成类时才会产生真正的类型）

2.类型说明符 标识符（指明可以接受一个由“类型说明符”所指定类型的常量作为参数）

模板参数表可以包含一个或多个以上两种参数，多于一个时各个参数之间用逗号分隔。鸡啄米提醒大家注意的是，类模板的成员函数必须是函数模板。实际上，类模板并不是有实际意义的代码，它只是一些具有相似功能的类的抽象，就是把这些类的共有部分写成模板，类型作为参数，只有用类模板生成类时才会根据需要生成实际的类的代码。

用类模板建立对象时的声明形式为：

模板名<模板参数表> 对象名 1,...,对象名 n;

此处的模板参数表是用逗号分隔开的若干类型标识符或常量表达式构成。它与上面类模板声明时“模板参数表”中的参数是一一对应的。类型标识符与类模板中的“class 标识符”对应，常量表达式与“类型说明符 标识符”对应。这样声明对象之后系统会根据指定的参数类型和常量值生成一个类，然后建立该类的对象。

鸡啄米最后给大家一个简单明了的例子：

```
#include <iostream>
using namespace std;
// 定义结构体 Student
struct Student
{
    int   id;            // 学号
    float average;   // 平均分
};
// 类模板，实现对任意类型的数据进行存取
template <class T>
class Store
{
public:
```

```
    Store(void);            // 默认形式（无形参）的构造函数
    T GetElem(void);    // 获取数据
    void PutElem(T x);  // 存入数据
private:
    T item;                 // item 用来存放任意类型的数据
    int haveValue;      // 标识 item 是否被存入数据
};
// 以下是成员函数的实现，注意，类模板的成员函数都是函数模板
// 构造函数的实现
template <class T>
Store<T>::Store(void):haveValue(0)
{
}
// 获取数据的函数的实现
template <class T>
T Store<T>::GetElem(void)
{
    // 若 item 没有存入数据，则终止程序
    if (haveValue == 0)
    {
        cout << "item 没有存入数据!" << endl;
        exit(1);
    }
    return item;
}
// 存入数据的函数的实现
template <class T>
void Store<T>::PutElem(T x)
{
    haveValue = 1;     // 将其置为 1，表示 item 已经存入数据
    item = x;              // 将 x 的值存入 item
}
int _tmain(int argc, _TCHAR* argv[])
{
    // 声明 Student 结构体类型变量，并赋初值
    Student g = { 103, 93 };
    // 声明两个 Store 类的对象，数据成员 item 为 int 类型
    Store<int> S1, S2;
    // 声明 Store 类对象 S3，数据成员 item 为 Student 结构体类型
    Store<Student> S3;
```

```
    S1.PutElem(7);    // 向对象 S1 中存入数值 7
    S2.PutElem(-1);   // 向对象 S2 中存入数值-1
    // 输出 S1 和 S2 的数据成员的值
    cout << S1.GetElem() << "  " << S2.GetElem() << endl;
    S3.PutElem(g);    // 向对象 S3 中存入 Student 结构体类型变量 g
    // 输出对象 S3 的数据成员
    cout << "The student id is " << S3.GetElem().id << endl;
    return 0;
}
```

上面这个程序的运行结果是：

```
7  -1
The student id is 103
```

类是对对象的抽象，类模板是对类的抽象。鸡啄米最后跟大家说明下，如果觉得类模板太难理解，也没关系，这毕竟不是软件开发中必须用到的东西，等大家对 C++有了一些心得，再想想如何用类模板实现代码复用也是可以的。

鸡啄米：C++编程入门系列之十七（类与对象：UML 简介）

大家好，鸡啄米上一讲给大家介绍了类模板的概念，这一讲给大家简单介绍下 UML 的知识。如果要系统的学习 UML 还要找专门讲 UML 的教程来研究。

我们在进行软件开发的时候，如果只靠脑子想，只有一个看不见的软件规划、软件架构，可能写程序时会影响你的思路的清晰，或者中间间断了以后会忘记当初的规划而要重新回忆或重新规划。如果我们可以把程序设计用图形表达出来，就会让我们的思路很清晰，也很容易进行合理的优化，我们和其他的软件开发人员或者用户就能够进行更好的沟通。

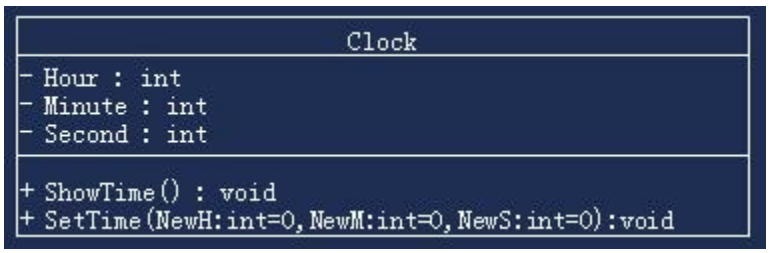
面向对象设计图应运而生，它能清楚的描述以下几个问题：

1.**类**，包括数据成员和函数成员。
2.**对象**，类的实例。
3.**类及对象的关系**，继承或者包含。
4.**类及对象之间的联系**，相互作用与消息传递等。

现在国际上标准的面向对象标记方法称为 UML，即统一建模语言。这种标记分为两类图形符号：表示符号和连接符号。表示符号用来表示类和对象，连接符号用来表示类和对象之间的关系和联系。

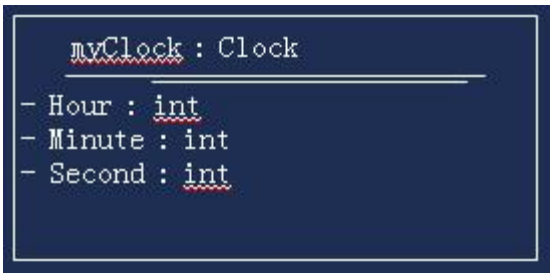
UML 中有 9 种图：类图、对象图、用例图、顺序图、协作图、状态图、活动图、组件图和实施图。这里只介绍下类图和对象图。

鸡啄米用第十三讲类与对象：类的声明、成员的访问控制和对象中的 Clock 类做例子，给出它的类图：



大家可以看出，最上面的框中是类名，中间框中是数据成员，最下面的框中是函数成员。

再用 Clock 类的对象 myClock 给大家一个对象图的例子：



UML 中类及对象的关系有以下几种：依赖、关联、聚合、组合、泛化和实现。

鸡啄米下面分别讲讲这几种关系：

1.依赖关系。

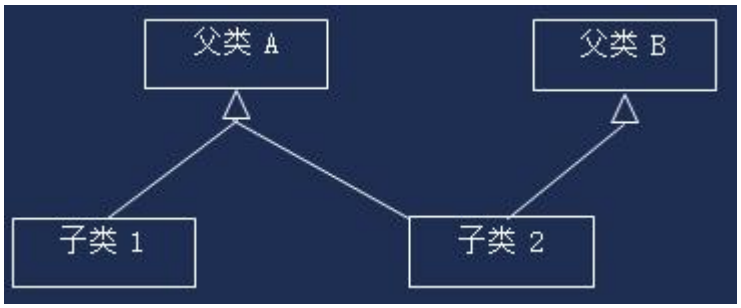
如果类 A 使用了类 B，或者说如果类 B 的变化会影响类 A，则说类 A 依赖于类 B。一般有以下几种情况属于依赖关系：类 A 调用类 B 的成员函数；类 B 的对象是类 A 的成员变量；类 A 的成员函数使用了类 B 类型的参数。依赖关系用带箭头的虚线表示，如下图：

2.关联关系。

上图中，重数 A 表示类 B 的每个对象与类 A 的多少个对象相关联，重数 B 则表示类 A 的每个对象与类 B 的多少个对象发生作用。比如，老师和学生的关联，老师类的重数应该是 1，学生类的重数可能是 n。

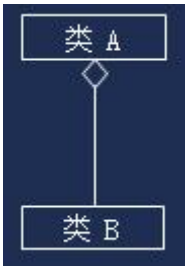
3.泛化关系。

如果类 A 和类 B 是从类 C 继承的子类，那么类 C 就是类 A 和类 B 的泛化。泛化关系用带空心三角形的实线表示：



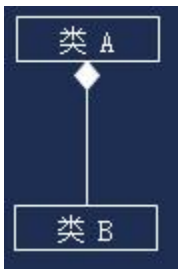
4.聚合关系。

聚合表示类之间的关系是整体和部分的关系，但是聚合关系中的整体和部分是可以分开的。比如，我们可以选择某个主板、硬盘、机箱等配件组装一台电脑，以后这台电脑可以随时更换配件，还一样能够组成一台电脑。聚合关系用带空心菱形的实线表示。



5.组合关系。

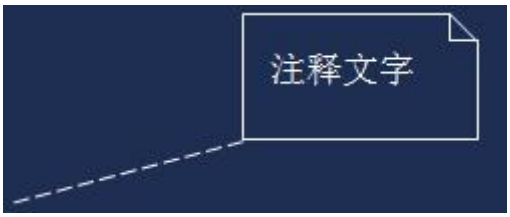
组合关系也是整体和部分的关系，但是它与聚合关系不同的是，整体和部分是不可以分开的。比如，我们的房子由客厅、卧室、厨房等组合而成，房子不能和别人的房子对换客厅等任何房间，客厅、卧室等不能和房子分开。组合关系用带实心菱形的实线表示。



6.实现关系。

实现关系用来说明接口和实现接口的类之间的关系。实现关系图用带空心三角形的虚线表示：

最后，鸡啄米再说下 UML 中的**注释**。在 UML 图形上，注释表示为带有褶角的矩形，然后用虚线连接到 UML 的其他元素上，它是一种用于在图中附加文字注释的机制。



UML 的知识鸡啄米就大概讲这些了，具体系统的 UML 内容大家可以看专门讲解的相关书籍。

第五部分：C++程序设计必知

鸡啄米：C++编程入门系列之十八（C++程序设计必知：作用域和可见性）

上一讲鸡啄米大概介绍了下 UML 知识，这一讲开始鸡啄米就要讲讲 C++程序设计中必知的一些结构和语法的知识点。这些都是很基础但是很有必要掌握的知识，能够很好的利用这些知识就表示你有一些内功了哦。

这些必知的知识包括作用域、可见性和生存期，还有局部变量、全局变量、类的数据成员、静态成员及友元和数据等。这一讲鸡啄米会给大家详细讲讲作用域和可见性。作用域是用来表示某个标识符在什么范围内有效，可见性是指标识符是否可见，可引用。

1.作用域

作用域是这样一个区域，标识符在程序的这个区域内是有效的。C++的作用域主要有四种：函数原型作用域、块作用域、类作用域和文件作用域。

a.函数原型作用域

函数原型大家还记得吗？比如：void fun(int x);这个语句就是函数原型的声明。函数原型声明中的形参的作用范围就是函数原型作用域。fun 函数中形参 x 有效的范围就在左、右

两个括号之间，出了这两个括号，在程序的其他地方都无法引用 x。标识符 x 的作用域就是所谓的函数原型作用域。函数原型如果有形参，声明时一定要有形参的类型说明，但是形参名比如 x 可以省略，不会对程序有任何影响，一般为了程序可读性好，我们一般都写上一个容易理解的形参名。函数原型作用域是最小的作用域。

b.块作用域

这里说的块就是一对大括号括起来的一段程序，块中声明的标识符在什么范围内有效呢？从标识符声明处到块结束的大括号之间都有效。也就是说块中标识符的作用域就是从标识符声明处开始，到块结束的大括号为止。鸡啄米给大家个例子说明下：

```
void fun(int x)
{
    int a(x);            // a 的作用域开始
    cin>>a;
    if (a>0)
    {
        int b;           // b 的作用域开始
        ......
    }                    // b 的作用域结束
}                        // a 的作用域结束
```

在 fun 的函数体内声明了整型变量 a，又在 if 语句的分支内声明了变量 b，a 和 b 都具有块作用域，但是它们的块作用域并不同。a 的作用域从其声明处开始，到其所在块的结束处也就是整个函数体结束的地方为止，b 的作用域是从 b 声明处开始到其所在块结束也就是 if 分支体结束 0 的地方为止。

c.类作用域

假设有一个类 A，A 中有一个数据成员 x，x 在 A 的所有函数成员中都有效（除非函数成员中也定义了一个名称为 x 的变量），这样的 x 就具有类作用域。为什么要排除含有另一个名称也为 x 的变量的函数成员呢？因为那样的话 A 的数据成员 x 在此函数成员被函数成员中的另一个 x 覆盖，不可见了，关于可见性下面会讲。

函数成员访问的大多数数据成员都具有类作用域。我们一般用 a.x 的方式访问类 A 的对象 a 的数据成员 x，这里的 x 就具有类作用域。

符号“.”用于访问对象的成员，包括函数成员。比如，a.fun(x)用来调用对象 a 的函数 fun。如果 ptr 是指向类 A 的一个对象的指针，则访问其数据成员 x 的方式为 ptr->x，访问函数成员的形式如：ptr->fun(x)。

d.文件作用域

如果一个标识符没有在前三种作用域中出现，则它具有文件作用域。这种标识符的作用域从声明处开始，到文件结尾处结束。

鸡啄米举个例子说明下文件作用域：

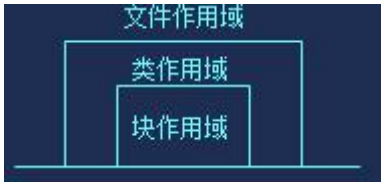
```
#include<iostream>
using namespace std;
int i;   //文件作用域
int main()
{
        i=1;
        {                                   //子块
              int i;                        //块作用域
              i=2;
               cout<<"i="<<i<<endl;  //输出 2
        }
        cout<<"i="<<i;   //输出 1
        return 0;
 }
```

上面的例子中，在 main 函数之前声明了变量 i，i 在整个源文件中都有效，即它具有文件作用域。而在子块中也声明一个变量 i，这个 i 具有块作用域。进入 main 函数后给 i 赋了初值 1，在子块中又声明了一个同名变量 i，并赋初值 2，第一次输出 i 时输出 i=2，为什么呢？因为子块里具有块作用域的 i 把外面具有文件作用域的 i 屏蔽掉了，就是说在子块中，具有文件作用域的 i 是不可见的。出了子块后，具有块作用域的 i 就无效了，所以就输出具有文件作用域的 i 的值 i=1。

2.可见性

标识符的可见性是指在程序的某个地方是否是有效的，是否能够被引用被访问。程序运行到某一处时，能够访问的标识符就是在此处可见的标识符。

上面说的四种作用域中，最大的是文件作用域，其次是类作用域，再次是块作用域。它们的包含关系是：



作用域可见性要注意的几点是：

a.我们要引用标识符时，必须先声明标识符。

b.在一个作用域内，不能声明多于一个的同名的标识符。

c.在不同的作用域，并且这些作用域间没有互相包含关系，则可以在其中声明同名标识符，这些同名标识符不会互相影响。

d.如果在有包含关系的作用域中声明了同名标识符，则外层作用域中的标识符在内层作用域中不可见。

我们再来看下文件作用域中的那个例子，此例就是文件作用域中包含了块作用域的例子。在子块之前可以引用具有文件作用域的变量 i，此时它是可见的，但是进入子块后，就只能引用具有块作用域的变量 i 了，这时具有文件作用域的变量 i 就不可见了，这就是上面 d 中说的外层的标识符被内层的同名标识符屏蔽，这也叫做同名覆盖。

作用域和可见性的知识还算是比较简单的。鸡啄米相信大家很容易就能够理解。如果有什么问题大家可以在鸡啄米博客中留言讨论，大家辛苦了。

鸡啄米：C++编程入门系列之十九（C++程序设计必知：生存期）

大家十一玩的高兴吗？可能大家都旅游散心去了，没有那么多时间跟着鸡啄米的教程学习了，但是鸡啄米再加班给大家写写教程吧。上一讲鸡啄米讲的是作用域与可见性，今天鸡啄米给大家讲讲生存期。

自然界中的事物都有产生和消亡，都有生存期，程序中的简单变量和类的对象也一样有它们的生存期，它们也会产生和消亡。这一讲中鸡啄米用对象来统一表示简单变量和类的对象。对象的生存期分为静态生存期和动态生存期两种。什么叫做静态生存期和动态生存期呢？下面将分别讲解。

1.静态生存期

若某个对象它的生存期与程序的运行期相同，我们就说它具有静态生存期，就是说在程序运行期间它都不会释放。

具有文件作用域的对象都具有静态生存期。

函数内部具有块作用域的对象怎样才能具有静态生存期呢？我们可以使用关键字 static 修饰对象。比如，static int x;这个语句就是将 x 声明为具有静态生存期的变量，也成为静态变量。

2.动态生存期

除了上述情况的对象具有静态生存期外，其余对象都具有动态生存期。具有动态生存期的对象产生于声明处，于该对象的作用域结束处释放。

鸡啄米给大家举个例子说明下简单变量和类的对象的生存期，其中涉及到了全局变量和局部变量，这两个概念下一讲会讲到。现在大家可以先这样简单的理解，全局变量就是具有文件作用域的变量，局部变量就是具有块作用域的变量。

```
#include <iostream>
using namespace std;
int a=1;          // a 为全局变量，它具有静态生存期。
void fun(void);
int main()
{
        static int x;    // 局部变量，具有静态生存期，可以叫做静态局部变量，局部可见。
        int y=5;         // y，z 为局部变量，具有动态生存期。
        int z=1;

        cout<<"---MAIN---\n";
        cout<<" a: "<<a<<" x: "<<x<<" y: "<<y<<" z: "<<z<<endl;
        z=z+2;
        fun();
        cout<<"---MAIN---\n";
        cout<<" a: "<<a<<" x: "<<x<<" y: "<<y<<" z: "<<z<<endl;
        a=a+10;
        fun();
        return 0;
}
void fun(void)
{
       // x,y 为静态局部变量，具有全局寿命，局部可见。只有第一次进入函数时被初始化。
        static int x=4;
        static int y;
        int z=10;   // z 为局部变量，具有动态生存期，每次进入函数时都初始化。
        a=a+20;
        x=x+3;
        z=z+4;
        cout<<"---FUN---\n";
        cout<<" a: "<<a<<" x: "<<x<<" y: "<<y<<" z: "<<z<<endl;
        y=x;
}
```

运行结果：

```
---MAIN---
a: 1 x: 0 y: 5 z: 1
---FUN---
a: 21 x: 7 y: 0 z: 14
---MAIN---
a: 21 x: 0 y: 5 z: 3
---FUN---
a: 51 x: 10 y: 7 z: 14
```

这里要说明下，静态局部变量如果没有进行显式初始化，那么它将被默认初始化为 0。当第二次进入 fun 函数时，其静态局部变量 x 和 y 将不再初始化，x 和 y 将继续使用 fun 函数第一次被调用后最后的值来参加下面的运算。具体点说，第一次调用完 fun 函数后，fun 中变量 x 的值是 7，在第二次调用 fun 函数时，x 不会被赋值为 4，而是继续使用 7 作为自己的值。

鸡啄米再用时钟类来给大家一个例子，让大家回想温习下以前和本讲的一些知识。

```
#include <iostream>
using namespace std;
 class Clock //时钟类声明
{
 public: //外部接口
        Clock();
        void SetTime(int NewH, int NewM, int NewS);   //三个形参均具有函数原型作用域
        void ShowTime();
       ~Clock(){}
 private: //私有数据成员
        int Hour,Minute,Second;
};
//时钟类成员函数实现
Clock::Clock() //构造函数
{
       Hour=0;
        Minute=0;
        Second=0;
}
void Clock::SetTime(int NewH,int NewM,int NewS)
{
        Hour=NewH;
```

```
        Minute=NewM;
        Second=NewS;
    }
    void Clock::ShowTime()
    {
        cout<<Hour<<":"<<Minute<<":"<<Second<<endl;
    }
    //声明全局对象 g_Clock，具有文件作用域，静态生存期
    Clock g_Clock;
    int main() //主函数
    {
        cout<<"文件作用域的时钟类对象:"<<endl;
        //引用具有文件作用域的对象：
        g_Clock.ShowTime();
        g_Clock.SetTime(10,20,30);

        Clock myClock(g_Clock);    //声明具有块作用域的对象 myClock，并通过默认拷贝构造函数用 g_Clock 初始化 myClock
        cout<<"块作用域的时钟类对象:"<<endl;
        myClock.ShowTime(); //引用具有块作用域的对象
        return 0;
    }
```

运行结果：

```
文件作用域的时钟类对象:
0:0:0
块作用域的时钟类对象:
10:20:30
```

这个程序中定义了具有各种作用域的对象，Clock 类声明中，SetTime 函数声明的三个参数具有函数原型作用域，数据成员具有类作用域，全局对象 g_Clock 具有文件作用域，对象 myClock 具有块作用域。除 g_Clock 具有静态生存期外，其余均有动态生存期。

我们通过学习这个例子，复习了类的声明、类成员函数的实现、类构造函数和拷贝构造函数的使用、作用域和可见性，还有本讲的生存期等知识点。

鸡啄米今天就讲到这了，最后祝大家假期愉快！

鸡啄米：C++编程入门系列之二十（C++程序设计必知：数据和函数）

十一期间鸡啄米给大家做了关于生存期的课程，因为是假期，所以鸡啄米也就做了一节课程。今天继续给大家讲讲数据和函数的一些需要了解的具体知识。

在结构化程序设计中，有人提出“数据结构+算法=程序设计”，数据结构就是数据的组织，算法是用函数实现的，可见数据和函数很早就被看作程序设计的重点了。面向对象程序设计中，这种观点应稍作一下修改：“数据结构+算法=对象”。就是数据和函数构成了类的对象。

面向对象程序设计中，数据用来描述对象的属性，函数是行为，用来处理数据。将数据和函数封装到一个类里，类中的函数成员可以访问数据成员，函数成员之间可以实现数据共享。

函数之间实现数据共享有以下几种方式：局部变量、全局变量、类的数据成员、类的静态成员和友元。如何共享局部变量呢？可以在主调函数和被调函数之间通过参数传递来共享。全局变量具有文件作用域，所以作用域中的各个函数都能共享全局变量。类的数据成员具有类作用域，能够被类的函数成员共享。这一讲鸡啄米先介绍这三种共享方式，类的静态成员和友元共享的方式后面两讲会讲到。

1.局部变量

鸡啄米前面简单提到过，局部变量其实一般就是说具有块作用域的变量。如果要在不同的块之间共享存储在局部变量中的数据，只能通过参数传递来实现。这种共享只能在主调函数和被调函数之间进行。因为局部变量具有块作用域，所以不同函数中的局部变量是互不可见的，这也是函数之间的一种数据隐藏，在结构化程序设计中这是实现数据隐藏的唯一方式。而在面向对象设计中主要靠封装来隐藏数据。

2.全局变量

全局变量具有文件作用域，在整个作用域中，除了定义了同名局部变量的块以外的地方都可以直接访问全局变量。不同的函数在不同的地方都能访问相同的全局变量，这样就实现了函数之间对全局变量的共享。

鸡啄米给大家一个全局变量的例子：

```
#include<iostream>
using namespace std;
int global;             // 定义全局变量 global
void f()
{
    global=2;    // 给全局变量 global 赋值
}
void g()
{
    cout<<global<<endl;   // 输出全局变量 global 的值
}
```

```
int _tmain(int argc, _TCHAR* argv[])
{
        f();            // 调用函数 f 给全局变量 global 赋值 2
       g();             // 输出全局变量 global 的值“2”
      return 0;
}
```

我们在进行软件开发的时候使用全局变量会很方便，不用那么多的传递参数，但是它也有很多负面作用。我们在多处使用了全局变量以后就不清楚此变量在哪些函数中使用过了，除非逐个查找，这样就会带来一个问题，如果全局变量要做一些大的改动，我们就不知道它会影响到哪些函数，这时就要通过大量的测试来排除全局变量变动带来的问题。

结构化程序设计中，只能使用前面提到的局部变量参数传递和全局变量这两种方式共享数据。这两种方式各有利弊，局部变量隐藏性好，但是传递的时候会有很大的系统开销，所以一般只传递少量数据，大量数据通过全局变量的方式共享，但这又使得程序模块间耦合严重，扩展和维护都很困难。C++中可以通过具有文件作用域的全局对象来实现数据和函数的共享，这实际上也属于全局变量的范畴。

3.类的数据成员

类中封装了数据成员和函数成员。其中数据成员可以被同一个类中的所有函数成员访问，这样就实现了数据的共享。但是我们又可以通过设置访问控制权限，把这种共享限制到类的范围之内，这样类的数据成员对外界来说是隐藏的。也就达到了共享与隐藏的辩证结合。总起来说，要达成这样的效果需要把类的数据成员都设置为私有类型。

鸡啄米再给出一个类的数据成员共享的例子：

```
#include<iostream>
using namespace std;
class Example
{
public:
          void Init();
         void Out();
private:
          int global;
};
void Example::Init()
{
         global=2;
}
void Example::Out()
```

```
{
        cout<<global<<endl;
}
int _tmain(int argc, _TCHAR* argv[])
{
        Example   MyEmp;
        MyEmp.Init();
        MyEmp.Out();
         return 0;
}
```

我们可以看出，类的数据成员 global 可以被类的函数成员 Init 和 Out 共享，但是在类外是不可以访问 global 的，只能通过接口 Init 和 Out 访问，实现了共享访问的权限控制，弥补了局部变量和全局变量的不足。这里的数据成员 global 发生类型等结构变化时，只会影响访问它的函数成员 Init 和 Out，对程序其他地方没有影响，如果有修改的必要我们只需要修改 Init 和 Out 的实现就可以了。这样程序模块间的耦合性就大大降低，提高了程序的可扩展性和可维护性，能够大大提高软件开发效率。

关于数据和函数的局部变量、全局变量和类的数据成员的共享鸡啄米就讲到这里了，后面会接着讲剩下的类的静态成员和友元的共享，谢谢大家。

鸡啄米：C++编程入门系列之二十一（C++程序设计必知：类的静态成员）

鸡啄米在上一讲数据和函数中讲到，函数之间共享数据也就是此函数访问彼函数的数据主要是通过局部变量、全局变量、类的数据成员、类的静态成员及友元实现的，前三个已经讲过了，这一讲鸡啄米来讲讲静态成员。静态成员包括静态数据成员和静态函数成员。

1.静态数据成员

之前讲到的类的数据成员都是一个对象一个拷贝，每个对象都有自己的数据成员的值，但有时候我们需要让某个类的所有对象共享某个数据成员，比如我们有一个学生类 CStudent，其中有学号、姓名等数据成员，如果我们想要对学生人数进行统计，那么这个数据应该放在哪里呢？放在类外的某个地方的话数据的隐藏性就不好了。可能有朋友就说了，可以在类中增加一个数据成员存放学生人数，但这样就有两个坏处，第一，该类的每个对象都将有一个这个数据成员的副本，浪费内存；第二，学生人数可能在不同的对象中有不同的数，不一致了。这个时候就需要静态数据成员了。

静态数据成员存放的是类的所有对象的某个共同特征的数据，对于每个对象而言，该数据都是相同的，在内存中只存在一份。这与类的一般数据成员不同，一般数据成员会在每个对象中都有一个拷贝，每个拷贝的值可能都不一样。静态数据成员由类的所有对象共同维护和使用，这样就实现了类的对象间的数据共享。

声明静态数据成员的方式与一般数据成员不同的是，其前面要加 static 关键字，比如 static int x;。静态数据成员具有静态生存期。它可以通过类名或对象名访问。用类名直接访问的方式为：类名::标识符。在类的声明中仅对静态数据成员进行引用性说明，就是带 static 的声明，必须在其文件作用域的其他地方进行定义性说明，也可以进行初始化，如果不显式初始化的话会将此静态数据成员初始化为 0。

对于静态数据成员鸡啄米给大家一个例程：

```
#include <iostream>
using namespace std;
class CStudent        // 学生类的声明
{
public:
        CStudent(int nID)   { m_nID = nID; m_nCount++; }      // 构造函数
        CStudent(CStudent &s);                                // 拷贝构造函数
        int GetID(){ return m_nID;}
        void GetCount()     { cout<<" 学生人数："<<m_nCount<<endl; } // 输出静态数据成员
private:
        int m_nID;
        static int m_nCount;                                  // 静态数据成员的引用性说明
};
CStudent::CStudent(CStudent &s)
{
        m_nID = s.m_nID;
        m_nCount ++;
}
int CStudent::m_nCount=0;                                     // 静态数据成员的定义性说明及初始化
int main()
{
        CStudent A(6);          // 定义对象 A
        cout<<"学生 A,"<<A.GetID();
        A.GetCount();           // 输出此时学生个数
        CStudent B(A);          // 定义对象 B，并用 A 初始化 B
        cout<<"学生 B,"<<B.GetID(); // 输出此时学生个数
        B.GetCount();
        return 0;
}
```

当然学生的学号不可能一样，这里只是举个例子。类 CStudent 的静态数据成员 m_nCount 用来给学生人数计数，定义一个新的学生对象，它的值就加 1。这里初始化的时候要注意，必须用类名来引用，还有就是此数据成员的访问控制属性，这里 m_nCount 声明为私有类型，初始化时可以直接访问，但是在主函数内就不能直接访问了。

在对象 A 声明时，调用构造函数，m_nCount 加 1，对象 B 声明时，调用拷贝构造函数，m_nCount 又加 1，两次都访问的 A 和 B 共同维护的静态数据成员，这样就实现了对象间的数据共享。上面的运行结果是：

学生 A,6 学生人数：1

学生 B,6 学生人数：2

2.静态函数成员

上面的学生类中，GetCount 函数用来输出静态数据成员 m_nCount，我们想输出 m_nCount 时就要通过学生类的某个对象调用 GetCount 函数来实现。但是，在定义任何对象之前，m_nCount 是有初始值的，那我们怎样输出这个初始值呢？因为没有定义任何对象，就没法通过对象调用 GetCount。m_nCount 是私有的，也不能用类名直接引用。因为 m_nCount 是类的所有对象共有的，那我们希望不通过对象直接用类名调用函数来显示 m_nCount，这就要用到静态成员函数了。

静态成员函数跟静态数据成员一样，也是由类的所有对象所共有，由他们共同维护和使用。声明时前面也要加 static 关键字，比如，static void fun(void);。我们可以通过类名或对象名调用公有类型的静态成员函数，而非静态成员函数只能由对象名调用。

静态成员函数可以访问该类的静态数据成员和其他静态成员函数，如果要访问非静态数据成员，则必须将对象作为参数传递进去，然后通过传进去的对象名来访问。鸡啄米给大家举个例子：

```
class A
{
public:
        static void f(A a);
private:
        int x;
};
void A::f(A a)
{
        cout<<x;     //对 x 的引用是错误的
        cout<<a.x;   //正确
}
```

所以静态成员函数访问非静态成员很麻烦，它一般用来访问全局变量和同一个类中的静态数据成员。

鸡啄米把上面学生类的 GetCount 函数改成静态成员函数大家看下：

```
#include <iostream>
using namespace std;
class CStudent        // 学生类的声明
{
public:
        CStudent(int nID)   { m_nID = nID; m_nCount++; }      // 构造函数
        CStudent(CStudent &s);                                // 拷贝构造函数
        int GetID()             { return m_nID;}
        static void GetCount(){ cout<<" 学生人数："<<m_nCount<<endl; } // 静态成员函数，输出静态数据成员
private:
        int m_nID;
       static int m_nCount;                                   // 静态数据成员的引用性说明
};
CStudent::CStudent(CStudent &s)
{
        m_nID = s.m_nID;
        m_nCount ++;
}
int CStudent::m_nCount=0;                                     // 静态数据成员的定义性说明及初始化
int main()
{
        CStudent A(6);                  // 定义对象 A
        cout<<"学生 A,"<<A.GetID();
        A.GetCount();                   // 输出此时学生个数
        CStudent B(A);                  // 定义对象 B，并用 A 初始化 B
        cout<<"学生 B,"<<B.GetID(); // 输出此时学生个数
        CStudent::GetCount();
        return 0;
}
```

大家可以比较的看下两个例子，学生类的声明只有一点不同，就是 GetCount 前加了 static 将其改为了静态成员函数，主函数中就可以通过类名或对象名调用静态成员函数，就像上面的 A.GetCount()和 CStudent::GetCount()，大家也可以把 A.GetCount()改成 CStudent::GetCount()，或者把 CStudent::GetCount()改成 B.GetCount()试下，运行结果应该是一样的。

静态成员的好处就是内存中只有一份拷贝，可以直接通过类名访问。鸡啄米今天就讲到这了，下一讲给大家讲讲友元。

鸡啄米：C++编程入门系列之二十二（C++程序设计必知：友元）

上一讲中鸡啄米讲了类的静态成员，这一讲来讲下最后一种共享数据方式--友元。友元是一种破坏类的封装性获取类的隐藏数据的方式。

如果有以下程序：

```
class A
{
public:
        int Getx()  { return x; }
private:
        int x;
};
class B
{
public:
        void Set(int y);
private:
        A a;
};
```

上面就是类的组合的形式。类 B 中包含一个类 A 的对象作为数据成员。但是类 B 的成员函数不能直接访问类 A 的私有数据成员 x。比如下面类 B 的成员函数 Set 的实现是不正确的：

```
void B::Set(int y)
{
      a.x = y;
}
```

那么怎样实现 B 的成员函数直接访问 A 的数据成员呢？C++提供了友元机制。

通过友元的方式，某个普通函数或者类的成员函数可以访问某个类中的私有数据，这样就等于在类的封装的很好的外衣上剪开了一个小洞，外界可以通过这个小洞访问到一些类内部的数据。友元提供了一种不同类或对象的成员函数之间、类的成员函数与普通函数之间共享数据的机制。它破坏了类的封装性和类数据的隐藏性，但是又给我们进行软件开发提供了很多方便，在我们实地进行软件开发的时候可以自己在共享和封装之间平衡一下，决定是否选择使用友元，鸡啄米认为，原则上尽量少使用或不使用友元，除非确实很大的提高了开发效率。

在一个类中声明友元的方式是，用关键字 friend 把普通函数、其他类的成员函数或其他类声明为此类的友元，用 friend 声明的元素就可以访问此类的私有数据。**如果友元是普通函数或是类的成员函数就叫做友元函数，如果友元是一个类则叫做友元类，友元类的所有成员函数均为友元函数。**

1.友元函数

友元函数是在类的声明中用关键字 friend 修饰的普通函数或者其他类的成员函数。友元函数虽不是本类的成员函数，但在它的函数体中可以访问该类对象的私有成员和保护成员。鸡啄米给大家看个例子：

```
#include <iostream>
#include <math.h>
using namespace std;
class Data //Data 类声明
{
public:     //外部接口
        Data(int xx=0) { x=xx; }
        int GetX() { return x; }
        friend int Add(Data &a, Data &b);
private: //私有数据成员
        int x;
};
int Add(Data &a, Data &b)
{
        return a.x + b.x;
}
int main()
{
        Data a(1);
        Data b(2);
        int sum = Add(a, b);
        cout<<"The sum is "<<sum<<endl;
        return 0;
}
```

程序运行结果是：The sum is 3

在 Data 类中声明友元函数 Add 时只给出了友元函数原型，友元函数 Add 的实现在类 Data 外。我们看到，在 Add 函数体中直接通过对象名访问了 Data 类的私有成员 x，这就是友元的强大作用。我们在类外用一般方式访问 x 的话必须通过公共接口 GetX 来实现，若要访问的私有成员很多或者要访问的地方很多就要多次调用函数，对于我们写代码和程序运行都有效率上的损失，但是用友元也有个很大的缺点就是，如果私有数据 x 发生结构性的变化，那么友元函数就要做相应的修改，所有调用友元函数的地方可能也要修改，这就降低了开发效率，所以是否要使用友元可以自己权衡。

上面例子中的友元函数是普通函数，这个函数也可以是其他类的成员函数，用法类似。——Mike:声明格式、定义格式？

2.友元类

类也可以声明为另一个类的友元，就像友元函数那样，这个作为另一个类的友元的类就叫做友元类。如果一个类 B 是类 A 的友元类，则类 B 的所有成员函数都是类 A 的友元函数，都能访问类 A 的私有成员和保护成员。友元类的声明形式为：

```
class A
{
    ...
    friend class B;   // 将类 B 声明为类 A 的友元类
    ...
}
```

上面声明中，类 B 是类 A 的友元类，B 的成员函数可以访问类 A 的对象的私有成员和保护成员。鸡啄米再让大家看看友元类的使用：

```
class A
{
public:
    void Display()  { cout<<x<<endl;}
    friend class B;
private:
    int x;
}
class B
{
public:
    void Set(int i);
    void Display();
private:
    A a;
};
void B::Set(int i)
{
    a.x=i;        // 因为类 B 是类 A 的友元类，所以类 B 的成员函数可以访问类 A 对象的私有成员
}
void B::Display()
{
```

```
    a.Display();
}
```

鸡啄米提醒大家，**友元关系不能传递**，如果类 B 是类 A 的友元，类 C 又是类 B 的友元，类 C 和类 A 如果没有声明则没有友元关系，另外，**友元关系是单向的**，如果类 B 是类 A 的友元，类 B 的成员函数可以访问类 A 对象的私有成员和保护成员，但是类 A 的成员函数不能访问类 B 对象的私有成员和保护成员。

友元的知识大概就这些了，大家在使用的时候要综合考虑封装和共享之间的平衡，鸡啄米认为还是要尽量少使用友元。

鸡啄米：C++编程入门系列之二十三（C++程序设计必知：常引用、常对象和对象的常成员）

数据的封装实现了数据的隐藏，让数据更安全，但是前面讲到的通过局部变量、全局变量、类的数据成员、类的静态成员及友元实现了数据的共享，这样又降低了数据的安全性。有些数据是需要共享而又不能被改变的，那么这时候我们就要将其声明为常量。

就像前面讲到的简单数据类型的符号常量，我们也可以用 const 声明对象，叫做常对象。鸡啄米这一讲就给大家讲讲常引用、常对象和对象的常成员，另外还有常数组和常指针，这两个概念以后鸡啄米会讲。

1.常引用

用 const 声明的引用就是常引用。**常引用所引用的对象不能被更改。**我们经常见到的是常引用作为函数的形参，这样不会发生对实参的误修改。常引用的声明形式为：const 类型说明符 &引用名。鸡啄米给大家看个常引用作为函数形参的例子：

```
#include<iostream>
using namespace std;
void show(const double& r);
int main()
{
    double d(9.5);
    show(d);
    return 0;
}
void show(const double& r)
//常引用作形参，在函数中不能更新 r 所引用的对象。
{
   cout<<r<<endl;
}
```

2.常对象

所谓常对象，是指数据成员在它的生存期内不会被改变。**定义常对象时必须对其进行初始化，并且不能改变其数据成员的值。**常对象的声明形式为：类名 const 对象名 或者 const 类名 对象名。常对象的例子如下：

```
class A
{
public:
        A(int i,int j) {x=i; y=j;}
         ...
private:
       int x,y;
};
A const a(6,8); //a 是常对象，不能被更新
```

如果程序中出现对常对象的数据成员修改的语句，编译器会报错。一般修改对象的数据成员有两种途径，一种是通过对象名访问公有数据成员并修改其值，而常对象的数据成员是不能被修改的；另一种是类的成员函数修改数据成员的值，而常对象不能调用普通的成员函数。可是这样的话，常对象就只剩数据，没有对外的接口了，这就需要为常对象专门定义的常成员函数了。

3.类的常成员函数

类中用 const 声明的成员函数就是常成员函数。常成员函数的声明形式为：类型说明符 函数名(参数表) const;。

鸡啄米要提醒大家注意几点：a.常成员函数在**声明和实现时都要带 const 关键字**；b.常成员函数**不能修改（可以访问）对象的数据成员，也不能访问类中没有用 const 声明的非常成员函数**；c.**常对象只能调用它的常成员函数，不能调用其他的普通成员函数**；d.**const 关键字可以被用于参与对重载函数的区分**，比如，如果有两个这样声明的函数：void fun(); void fun() const;，则它们是重载函数。

```
#include<iostream>
 using namespace std;
 class R
{
public:
        R(int r1, int r2)   { R1=r1; R2=r2; }
        void print();
        void print() const;
 private:
        int R1,R2;
};
 void R::print()
```

```
{
        cout<<R1<<":"<<R2<<endl;
}
void R::print() const
{
        cout<<R1<<";"<<R2<<endl;
}
int main()
{
        R a(5,4);
        a.print(); //调用 void print()
        const R b(20,52);
        b.print(); //调用 void print() const
        return 0;
}
```

上面的 R 类中声明了两个同名函数 print，第二个是常成员函数。在 main 函数中定义了两个对象 a 和 b，b 是常对象，通过 a 调用的是没有用 const 声明的函数，而通过 b 调用的是用 const 声明的常成员函数。

4.类的常数据成员

类的数据成员也可以是常量和常引用，用 const 声明的数据成员就是常数据成员。**在任何函数中都不能对常数据成员赋值。构造函数对常数据成员初始化，只能通过初始化列表。**鸡啄米给大家一个常数据成员的例子：

```
#include<iostream>
using namespace std;
class A
{
public:
        A(int i);
        void print();
        const int& r;
private:
        const int a;
        static const int b; //静态常数据成员
};
const int A::b=20;
A::A(int i):a(i),r(a) {}
void A::print()
{
```

```
        cout<<a<<":"<<b<<":"<<r<<endl;
}
int main()
{
        //建立对象 a 和 b，并以 50 和 10 作为初值，分别调用构造函数，通过构造函数的初始化列表给对象的常数据成员赋初值
        A a1(50),a2(10);
        a1.print();
        a2.print();
        return 0;
}
```

此程序的运行结果是：

```
50:20:50
10:20:10
```

对于常引用、常对象和对象的常成员，鸡啄米就讲到这里了，代码比较多，大家试验一下吧。谢谢大家。

鸡啄米：C++编程入门系列之二十四（C++程序设计必知：多文件结构和编译预处理命令）

鸡啄米上一讲给大家讲了常引用、常对象和对象的常成员，今天给大家讲下编程入门知识--多文件结构和编译预处理命令。

一.C++程序的多文件结构

之前鸡啄米给大家看了很多比较完整的 C++程序的例子，大家可能发现了，它们的结构基本上可以分为三个部分：类的声明、类的成员函数的实现和主函数。因为代码比较少，所以可以把它们写在一个文件中，但是我们实际进行软件开发时，程序会比较复杂，代码量比较大，

一个程序按结构至少可以划分为三个文件：**类的声明文件（*.h 文件）、类的实现文件（*.cpp 文件）和主函数文件（使用到类的文件）**，如果程序更复杂，我们会为每个类单独建一个声明文件和一个实现文件。这样我们要修改某个类时就直接找到它的文件修改即可，不需要其他的文件改动。

鸡啄米在第十九讲中讲生存期时有个时钟类的例子，现在鸡啄米给大家看下将那个程序按照上面说的结构分到三个文件里：

```
// 文件 1:Clock 类的声明，可以起名为 Clock.h
#include <iostream>
using namespace std;
```

```
class Clock //时钟类声明
{
public: //外部接口
        Clock();
        void SetTime(int NewH, int NewM, int NewS);   //三个形参均具有函数原型作用域
        void ShowTime();
       ~Clock(){}
private: //私有数据成员
        int Hour,Minute,Second;
};
// 文件 2：Clock 类的实现，可以起名为 Clock.cpp
#include "Clock.h"
//时钟类成员函数实现
Clock::Clock() //构造函数
{
        Hour=0;
        Minute=0;
        Second=0;
}
void Clock::SetTime(int NewH,int NewM,int NewS)
{
        Hour=NewH;
        Minute=NewM;
        Second=NewS;
}
void Clock::ShowTime()
{
       cout<<Hour<<":"<<Minute<<":"<<Second<<endl;
}
// 文件 3：主函数，可以起名为 main.cpp
#include "Clock.h"
//声明全局对象 g_Clock，具有文件作用域，静态生存期
Clock g_Clock;
int main() //主函数
{
        cout<<"文件作用域的时钟类对象:"<<endl;
        //引用具有文件作用域的对象：
        g_Clock.ShowTime();
        g_Clock.SetTime(10,20,30);
```

```
    Clock  myClock(g_Clock);    //声明具有块作用域的对象 myClock，并通过默认拷贝构造函数用 g_Clock 初始化 myClock
    cout<<"块作用域的时钟类对象:"<<endl;
    myClock.ShowTime(); //引用具有块作用域的对象
    return  0;
}
```

在 vs2010 中如何生成这三个文件呢？我们可以点菜单中 Project->Add Class，在弹出的对话框中选择 c++ class，然后由弹出个对话框，在 class name 处填上类名点 finish 就可以了，这样.h 文件和.cpp 文件会自动生成，我们也可以点 Project->Add New Item，在弹出的对话框中选择 Header File(.h)或 C++ File(.cpp)来生成.h 文件或.cpp 文件。

Clock.cpp 和 main.cpp 都使用#include "Clock.h"把类 Clock 的头文件 Clock.h 包含进来。#include 指令的作用就是将#include 后面的文件嵌入到当前源文件该点处，被嵌入的文件可以是.h 文件也可以是.cpp 文件。如果不包含 Clock.h，Clock.cpp 和 main.cpp 就不知道 Clock 类的声明形式，就无法使用此类，所以所有使用此类的文件都应该包含声明它的头文件。关于#include 指令下面鸡啄米会讲。

上面的程序在编译时，由 Clock.cpp 和 Clock.h 编译生成 Clock.obj，由 main.cpp 和 Clock.h 编译生成 main.obj，然后就是链接过程，Clock.obj 和 main.obj 链接生成 main.exe 可执行文件。如果我们只修改了类的实现文件，那么只需重新编译 Clock.cpp 并链接就可以，别的文件不用管，这样就提高了效率。在 Windows 系统中的 C++程序用工程来管理多文件结构，而 Unix 系统一般用 make 工具管理，如果大家从事 Unix 系统软件开发，就需要自己写 make 文件。

二.编译预处理程序

编译器在编译源程序以前，要由预处理程序对源程序文件进行预处理。预处理程序提供了一些编译预处理指令和预处理操作符。预处理指令都要由“#”开头，每个预处理指令必须单独占一行，而且不能用分号结束，可以出现在程序文件中的任何位置。

1.#include 指令

#include 指令也叫文件包含指令，用来将另一个源文件的内容嵌入到当前源文件该点处。其实我们一般就用此指令来包含头文件。#include 指令有两种写法：

```
#include  <文件名>
```

使用这种写法时，会在 C++安装目录的 include 子目录下寻找<>中标明的文件，通常叫做按标准方式搜索。

#include "文件名"

使用这种写法时，会先在当前目录也就是当前工程的目录中寻找""中标明的文件，若没有找到，则按标准方式搜索。

2.#define 和#undef 指令

如果你学过 C 语言，就会知道用#define 可以定义符号常量，比如，#define PI 3.14 这条指令定义了一个符号常量 PI，它的值是 3.14。C++也可以这样定义符号常量，但一般更常用的是在声明时用 const 关键字修饰。C 语言还用#define 定义参数宏，来实现简单的函数运算，比如，#define add(x,y) (x+y) 这条指令说明如果我们用到 add(1,2)则预处理后就会用(1+2)代替，C++中一般用内联函数来实现。

#undef 用来删除由#define 定义的宏，使其不再起作用。

3.条件编译指令

用条件编译指令可以实现某些代码在满足一定条件时才会参与编译，这样我们可以利用条件编译指令将同一个程序在不同的编译条件下生成不同的目标代码。例如，我们可以在调试程序时加入一些调试语句，用条件编译指令控制只有在 debug 模式下这些调试语句才参与编译，而在 release 模式下不参与编译。

条件编译指令有 5 种形式：

a.第一种形式：

```
#if  常量表达式
      程序正文       //当“ 常量表达式”非零时本程序段参与编译
#endif
```

b.第二种形式：

```
#if   常量表达式
      程序正文 1      //当“ 常量表达式”非零时本程序段参与编译
#else
      程序正文 2      //当“ 常量表达式”为零时本程序段参与编译
#endif
```

c.第三种形式：

```
#if 常量表达式 1
      程序正文 1       //当“ 常量表达式 1”非零时本程序段参与编译
elif 常量表达式 2
      程序正文 2      //当“常量表达式 1”为零、“ 常量表达式 2”非零时本程序段参与编
```

译

...

elif 常量表达式 n

程序正文 n //当“常量表达式 1”、...、“常量表达式 n-1”均为零、“ 常量表达式 n”非零时本程序段参与编译

#else

程序正文 n+1 //其他情况下本程序段参与编译

#endif

d.第四种形式：

#ifdef 标识符

程序段 1

#else

程序段 2

#endif

如果“标识符”经#defined 定义过，且未经 undef 删除，则编译程序段 1，否则编译程序段 2。

e.第五种形式：

#ifndef 标识符

程序段 1

#else

程序段 2

#endif

如果“标识符”未被定义过，则编译程序段 1，否则编译程序段 2。

4.**define 操作符**

define 是预处理操作符，不是指令，所以不能用#开头。使用形式为：define(标识符）。如果括号里的标识符用#define 定义过，并且没有用#undef 删除，则 define(标识符)为非 0，否则为 0。可以这样使用：

#if !define(HEAD_H)

#define HEAD_H

我们在包含头文件时，有时多次重复包含同一个头文件，比如下面这种情况：

// main.cpp 文件

#include "file1.h"

#include "file2.h"

int main()

{

```
        …
}
// file1.h 文件
#include "head.h"
 …
// file2.h 文件
#include "head.h"
 …
// head.h 文件
 ...
class A
{
      ...
}
 ...
```

main.cpp 包含了 file1.h 文件，file1.h 又包含了 head.h 文件，main.cpp 还包含了 file2.h 文件，file2.h 也包含了 head.h 文件，那么 main.cpp 就包含了两次 head.h 文件，在编译时就会报错，说 head.h 中的类 A 重复定义了。这时我们可以在被重复包含的文件 head.h 中使用条件编译指令，用一个唯一的标识符来标识 head.h 文件是否已经编译过，如果已经编译过则不会重复编译了。鸡啄米给大家改写下上面的 head.h 文件：

```
// head.h 文件
#ifndef HEAD_H
 #define HEAD_H
 ...
class A
{
     ...
}
...
#endif
```

在这个改好的 head.h 文件中，上来会先判断 HEAD_H 是否被定义过，如果没有被定义过，则 head.h 文件还没参与过编译，就编译此文件中的源代码，同时定义 HEAD_H，标记 head.h 文件已经参与过编译。如果 HEAD_H 已经被定义过，则说明此文件已经参与过编译，编译器会跳过本文件左右内容编译其他部分，类 A 也不会有重复定义的错误了。

鸡啄米今天给大家讲的内容比较多，大家慢慢看吧，其实并不难。编程入门学习中难免有问题，如果有什么问题大家可以在鸡啄米博客上留言讨论，谢谢大家。

第六部分：数组、指针和字符串

鸡啄米：C++编程入门系列之二十五（数组、指针和字符串：数组的声明和使用）

鸡啄米已经给大家讲了几种基本的控制结构、函数以及类与对象的知识，对于一般问题这些都可以解决了，但是我们在软件开发中难免要用到大型点的数据，相互之间有一定联系的数据，比如数据类型相似，这个时候我们就需要通过比较合理的数据结构来组织数据，达到一种高效的软件开发过程。

鸡啄米给大家讲讲数组。**数组是一些相同类型的对象的集合，这些对象具有一定的顺序关系，在逻辑上是连续的，在内存中的存储上也是连续的。**这些组成数组的对象叫做这个数组的元素。数组元素用数组名加带方括号的下标表示，比如，a[2]。数组元素具有相同的类型，可以是除 void 之外的任意一种类型。

一.数组的声明

数组跟结构体、类等一样也是自定义数据类型，所以使用以前也需要先进行声明。声明数组的时候要明确以下三点：1.确定数组名称；2.确定数组元素的类型；3.确定数组的维数及每维的大小。

数组的声明形式为：数据类型 标识符[常量表达式 1][常量表达式 2]...;。

数据类型规定了数据元素的类型，可以是整型、浮点型等基本数据类型，还可以是结构体、类等自定义数据类型。

标识符指定了数组的名称，记住，**数组名是一个常量，代表着数组元素集合在内存存储的起始地址，不能进行自增自减运算。**

“常量表达式 1”、“常量表达式 2”、...叫做下标表达式，必须是正整数。这些下标确定了数组元素的个数及每个元素在数组中的位置。一个数组可能有多个下标，下标的个数就是数组的维数，有 n 个下标的数组叫做 n 维数组。而每个下标表达式表示该维的下标个数也就是该维的元素个数。数组元素的总个数是各个下标表达式的乘积。

鸡啄米还要跟大家强调下标上界和下标下界的概念，**数组每一维的下标的下界是 0，上界是该下标所在维的元素个数减 1。**

例如：int a[5];表示 a 是一个一维整型数组，有 5 个元素，下标下界是 0，上界是 5-1=4，所以这 5 个元素为 a[0]...a[4]，可以用来存放 5 个整数序列。

int b[2][3];表示 b 是一个二维整型数组，其中第一维有 2 个下标（0,1），第二维有 3 个下标（0,1,2）。数组元素的个数是 6，可以用来存放 2 行 3 列的整数表格。对于数组 b，第一个元素是 b[0][0]，最后一个元素是 b[1][2]，至于它们的存储顺序鸡啄米下一讲会讲到。

二.数组的使用

我们在使用数组时，只能分别访问数组的各个元素。访问哪个元素是由数组名和下标来决定的。

数组元素的引用形式为：数组名[下标表达式 1][下标表达式 2]...;。

这里下标表达式的个数要看数组的维数，n 维数组就需要 n 个下标表达式。

实际上，数组的每一个元素都能看成是一个同类型的变量来使用，对整型变量的操作同样也可以用于整型数组的每个元素，对类的对象的操作也同样可用于类类型的数组的每个元素。鸡啄米提醒大家，在使用数组时要注意：1.引用数组元素时的下标表达式可以是任意合法的算术表达式，但是结果必须是整型数据；2.引用数组元素时的下标表达式不能超过声明数组时规定的上下界，否则会出现数组越界的问题。

鸡啄米给大家一个数组声明和使用的例子：

```
#include <iostream>
using namespace std;
int main()
{
    int a[5];
    int b[5];
    int i;
    for(i=0; i<5; i++)
    {
        a[i] = i*2;
        b[5-i-1] = a[i];
    }
    for(i=0; i<5; i++)
    {
        cout << "a[" << i << "]=" << a[i];
        cout << "  b[" << i << "]=" << b[i] << endl;
    }
    return 0;
}
```

上面的程序中声明了两个一维整型数组 a 和 b，都有 5 个元素，用 for 循环对它们进行赋值，在引用 b 的元素时使用了算术表达式作为下标。for 循环中把 0,2,4,6,8 分别赋值给 a[0]...a[4]，而 b 的各元素恰好是 a 中各元素的逆序排列，即 8,6,4,2,0。在这个程序中如果我们用了表达式 a[5]或 b[5]或者下标更大，就会有数组越界的错误。这是编程入门时非常容易犯的错误，鸡啄米希望大家一定注意。

如果没有数组，上面的 a[5]和 b[5]就需要一共定义 10 个变量，输出时也要写 10 条语句，如果有更复杂的运算那程序的复杂度可想而知，有了数组我们就可以用循环结构来简单的处理这类问题。

关于数组的声明和使用鸡啄米就先讲到这里了。鸡啄米博客欢迎大家来讨论交流。谢谢大家。

鸡啄米：C++编程入门系列之二十六（数组、指针和字符串：数组的存储与初始化、对象数组、数组作为函数参数）

hello，大家好，鸡啄米上一讲讲了数组的声明和使用，今天给大家讲下数组的存储与初始化、对象数组和数组作为函数参数的知识。

一.数组的存储

数组在内存中是一组连续的内存单元，也就是说数组元素是连续存储的。数组名是数组所占内存的首地址。

一维数组是按照下标的顺序存储的，而对多维数组就复杂些，以一定的约定顺序将多维数组存储在连续的内存单元中很重要。因为要对数组赋初值、函数间的数组数据传递等都需要先知道数组元素和存储位置的对应关系。

一维数组的元素是按照下标从小到大的顺序存在内存中的，例如，int a[3]在内存中的存储顺序是：a[0] a[1] a[2]。

对于二维数组元素，第一个下标叫做行标，第二个下标叫做列标。例如，数组 int a[2][3]相当于一个两行三列的矩阵：

a[0][0] a[0][1] a[0][2]
a[1][0] a[1][1] a[1][2]

上面的 a[0][0]，行标为 0，列标也为 0，表示矩阵第一行第一个元素，a[1][2]，行标为 1，列标为 2，表示矩阵第二行第三个元素。二维数组在内存中是按行存储的，也就是先存第一行，再存第二行...。每行中的元素按照列标从小到大的顺序存储。这种存储方式叫做行优先存储。上面例子中的二维数组 a 在内存中的存储顺序是：a[0][0] a[0][1] a[0][2] a[1][0] a[1][1] a[1][2]。

对于多维数组，其存储方式与二维数组类似。可以把多维数组的下标看作一个计数器。多维数组右边的下标为低位，左边的为高位，每一位都在各自的上界与下界之间变化。当某一位超过上界则向左进一位，该位及右边各位就恢复为下界。最右边一维的下标变化最快，最左边的变化最慢。鸡啄米再提醒大家注意，下界都是 0，某一维的上界是声明时该维的下标表达式值减 1。例如，数组 int a[2][2][2]在内存中的存储顺序是：a[0][0][0] a[0][0][1] a[0][1][0] a[0][1][1] a[1][0][0] a[1][0][1] a[1][1][0] a[1][1][1]。

实际上我们在软件开发时三维及三维以上的数组很少用到，用的最多的是一维数组。

二.数组的初始化

在我们声明数组时对部分或全部元素赋值就是数组的初始化。这里鸡啄米分开给大家讲简单数组类型的数组和对象数组，简单数组类型数组的初始化就是给数组元素赋初值，而对象数组的每个元素都是某个类的对象，它的初始化就是调用对象的构造函数。对象数组下面单独讲。

我们可以在声明数组时就给出每个元素的初值，例如：int a[2] = { 2,3 };这条语句声明了一个有 2 个元素的 int 类型的数组，a[0]的初值是 2，a[1]的初值是 3。

如果我们在声明数组时每个元素都给出初始值的话，就可以不说明元素个数，例如：int a[] = { 2,3 };和上面的那个数组声明语句等价。

我们也可以只对数组的前面一部分元素赋初值，例如，int a[3] = { 1,2 };这时数组元素的个数就必须给出，经过这样声明后，a[0]为 1，a[1]为 2，a[3]默认为 0，也就是后面没有赋初值的元素默认为 0。

多维数组的初始化也满足上面讲到的规则。另外，如果我们对数组初始化时给出了全部元素的初值，则第一位的元素个数可以不用显式说明，例如：int a[2][2] = { 1,2,1,2 };等价于 int a[][2] = { 1,2,1,2 };。多维数组还可以按照第一维的下标分组，用括号把每一组括起来。二维数组的话可以用大括号将每组括起来，例如：int a[2][2] = { {1,2},{1,2}};与上面的语句等价，通过将每组元素括起来我们更能直观的分开每行数据。

三.对象数组

当数组的元素是某个类的对象时此数组就是对象数组。声明一维对象数组的形式为：类名 数组名[下标表达式];。跟前面说过的基本数据类型的数组一样，使用对象数组也只能引用单个的数组元素，而每个数组元素都是对象，利用这个对象又可以引用它的公有成员，引用形式为：数组名[下标].成员名。

对象数组在初始化时每个对象元素都会调用其构造函数。如果初始化时数组元素显式给出初始值就会调用带形参的构造函数，如果没有显式指定初始值则调用默认构造函数。例如，A b[2] = { A(2,3) };会先调用带形参的构造函数初始化 b[0]，再调用默认构造函数初始化 b[1]。

四.数组作为函数参数

函数的参数可以是数组元素也可以是数组名。数组元素作为函数参数时跟同类型的变量作函数参数效果一样。

数组名作为函数参数时，实参和形参都须是数组名，并且数组类型要一样。**此时传递的是数组的首地址**，也就是说形参数组的首地址跟实参是一样的，后面的元素根据其在内存中的顺序进行对应，对应的元素的内存地址相同，所以实参数组的元素个数应该等于或多于形参数组的元素个数。如果在函数内对数组元素值改变，则主调函数中实参数组的相应元素也会改变。

鸡啄米给大家一个数组作为函数参数的例子：主函数中初始化一个矩阵并将每个元素都输出，然后调用子函数，分别计算每一行的元素之和，将和直接存放在每行的第一个元素中，返回主函数之后输出各行元素的和。

```
#include <iostream>
using namespace std;
void RowSum(int A[][4], int nrow)
{
    int sum;
    for (int i = 0; i < nrow; i++)
    {
        sum = 0;
        for(int j = 0; j < 4; j++)
            sum += A[i][j];
        cout << "Sum of row " << i << " is " << sum << endl;
        A[i][0]=sum;
    }
}
int main()
{
    int Table[3][4] = {{1,2,3,4},{2,3,4,5},{3,4,5,6}};
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 4; j++)
            cout << Table[i][j] << "   ";
        cout << endl;
    }
    RowSum(Table,3);
    for (int i = 0; i < 3; i++)
        cout << Table[i][0] << endl;
    return 0;
}
```

此程序的运行结果是：

```
1  2  3  4
2  3  4  5
3  4  5  6
Sum of row 0 is 10
Sum of row 1 is 14
Sum of row 2 is 18
10
 14
18
```

鸡啄米对数组的存储与初始化、对象数组和数组作为函数参数的内容就讲到这里，如果有什么问题可以在鸡啄米博客上留言讨论。谢谢大家的支持。

鸡啄米：C++编程入门系列之二十七（数组、指针和字符串：指针变量的声明、地址相关运算--“*”和“&”）

前面鸡啄米讲的是数组，从这一讲开始讲指针的概念及相关知识。学过 C 语言的应该对指针不陌生了，但是指针在软件开发中确实是个很重要的元素，建议大家都再好好看看吧。

C++从 C 语言继承了指针的概念。指针继续为 C++软件开发人员提供编程的方便。可能很多人认为 C 和 C++有指针太麻烦了，而且很容易因为指针的使用出现非法地址访问的问题，但是鸡啄米想说的是，指针给我们带来的更多的是对数据的操作和组织灵活和方便，至于容易出问题这方面，只要大家细心点，积累些经验就会很好的利用指针了。编程入门的时候大家先学好打好基础吧。

1.内存空间如何访问

我们使用的内存的基本存储单位是字节，一个字节由 8 个二进制位组成。每个字节都会按照一定的规则编号，这个编号就是该字节存储单元的地址。计算机就是利用这种编号也就是字节存储单元的地址来定位内存进行数据读写的。

我们在访问内存中的数据时，有两种方式，一种是通过变量名，一种是通过地址。我们声明的每个变量都是占据内存空间的，前面也讲到了，char 型占 1 个字节，float 型占 4 个字节。而内存空间的分配是在什么时候呢？全局变量、static 静态变量等静态生存期变量在程序开始运行之前就被分配了内存空间，动态生存期的变量是在变量声明时分配内存空间的，然后变量名也代表了分配的空间。但是有时候没有变量名可用或者很不方便，例如，在动态内存分配时就没有变量名可以用，这个鸡啄米会在后面讲；调用函数传递很多参数时，就不可能一个一个的通过变量名传递，而是需要用地址传递。

鸡啄米给大家形象的说下存储地址：每个学生住一个宿舍，学生相当于变量，宿舍相当于存储单元，我们可以用学生的名字来找到他，这就是变量名的方式，也可以通过宿舍号找到他，这就是通过地址访问的方式。

C++中的指针类型就是用来存放上面讲的内存单元地址的变量类型。

2.指针变量的声明

指针跟 int 等一样也是一种数据类型，**指针类型的变量就是指针变量，指针变量存放的是内存单元的地址。**

以前我们都是通过变量名访问变量，是一种直接的方式，而通过指针访问变量是间接的。例如，张三住在哪个宿舍不知道，可以在宿舍名单上第一行看到，那么宿舍名单的第一行就是指针变量，第一行上写的内容就是指针变量中存放的内容，而张三就是指针所指向的对象值。

指针同样是先声明再使用，声明的形式是：数据类型 *标识符;。

我们注意到中间有个“*”，它表示声明的是一个指针变量。“*”前面的“数据类型”表示的是指针所指向的变量或对象的类型，可以是任意类型，也就是说在声明的该指针所指向的内存单元中可以存放此数据类型的数据。而指针本身在 32 位系统上是 unsigned long 类型的，在 64 位系统上是 unsigned _int64 类型的，这里可能大家糊涂了，怎么又出来一个类型？结合上面张三的例子跟大家解释下，声明中的“数据类型”指定的是张三的类型，指针本身的类型指的是宿舍名单第一行的类型，清楚了吧。

例如，int *p;声明了一个指针变量 p，它指向 int 类型的数据，用来存放 int 型数据的地址。

3.地址相关运算--“*”和“&”

“*”和“&”可能大家已经见过了，作为二元操作符时，“*”表示相乘，“&”表示位运算里的与运算。在上面的指针变量的声明中，“*”用来表示声明的是一个指针变量。“&”在变量声明语句中出现在变量名左边时表示声明的是引用。而现在鸡啄米要跟大家讲的是它们的另外一种用法--地址运算。

“*”在地址运算中叫做指针运算符，表示指针指向的变量的值，是一元操作符，例如，**如果 p 是一个 int 型的指针，则*p 表示 p 指向的 int 型数据的值**。“&”在地址运算中叫做取地址运算符，也是一个一元操作符，用来获取一个对象的地址，例如，**有一个变量 i，&i 就表示变量 i 的存储单元地址**。

鸡啄米提醒大家注意一下“*”和“&”出现在声明语句中和在执行语句中意义是不一样的。

“*”出现在声明语句中被声明的变量名之前时，表示声明的是指针，例如，int *p;。**“*”出现在声明语句的初值表达式中或者执行语句中时，表示指针指向的对象的内容**，例如，int i=*p; cout<<*p;。**“&”出现在声明语句中被声明的变量名之前时，表示声明的是引用**，例如，int &rf;。**“&”出现在声明语句的初值表达式中或者执行语句中时，表示取对象的地址**，例如：

```
int  x;
int  y;
int  *p1=&x;
int  *p2;
p2=&y;
```

鸡啄米今天讲了指针的声明及“*”和“&”的用法，属于指针方面最基础的内容，希望大家掌握好。有什么问题在鸡啄米博客留言讨论。谢谢大家的关注。

鸡啄米：C++编程入门系列之二十八（数组、指针和字符串：指针的赋值和指针运算）

上一讲鸡啄米介绍了指针的基本概念和地址运算，今天为大家带来指针赋值和指针运算的内容。

一.指针的赋值

当我们声明了一个指针之后，这个指针变量中没有存储任何确定的地址值，而是一个随机整数。也就是它指向的地址是不确定的，有可能它指向了系统的重要数据，这时候我们如果修改了它指向地址的值可能会引起想象不到的系统问题。所以指针声明以后要先赋值才可以引用。

跟鸡啄米前面讲的其他类型的变量一样，给指针变量赋值也有两种方式：

1.在声明指针时对其进行初始化，也就是赋一个初值，初始化形式为：数据类型 *指针名=初始地址值;。

2.在声明指针后，单独给它赋值，赋值的语法形式为：指针名=地址;。

如果我们像方式 1 中使用变量地址为指针初始化，或者像方式 2 中将变量地址赋给指针，此变量必须在此之前声明过，并且这个变量的类型要和指针类型一样。也可以将一个已经赋值的指针赋值给另一个指针，让多个指针指向相同的变量。

鸡啄米给大家一个用两种方式为指针赋值的例子：

```
int  a[5];        // 声明一个整型数组
int  *p1=a;       // 声明指针 p1，并用数组首地址 a 来初始化 p1
int  *p2;         // 声明指针 p2
p2=a;             // 将数组首地址 a 赋值给指针 p2
```

鸡啄米说过，**数组名就是数组的首地址，所以它可以用来给指针赋值**。

另外，鸡啄米要强调的是，**赋给指针变量的值必须是地址常量（比如数组名）或地址变量，但一定不能是非 0 整数。**给指针赋值为 0 时表示该指针是一个空指针，它不指向任何地址，比如，int *p=0;。为什么会把指针声明为空指针呢？大家想想，我们在声明一个指针时没有给它赋值，这时它是一个随机的值，在给它赋确定的地址值之前如果我们使用了它，就可能会访问到重要的内存地址并破坏此地址的数据，造成严重后果，所以我们在软件开发中一般先将指针设为空指针。

给大家一个指针的声明、赋值和使用的例子：

```
#include<iostream>
using namespace std;
int main()
{
    int *p; //声明 int 型指针 p
    int i; //声明 int 型数 i
    p=&i; //取 i 的地址赋给 p
    i=6; //int 型数赋初值
    cout<<"Output int i="<<i<<endl;//输出 int 型数的值
    cout<<"Output int pointer i="<<*p<<endl;//输出 int 型指针所指地址的内容
    return 0;
}
```

程序运行结果是：

```
Output int i=6
Output int pointer i=6
```

上面的程序首先声明了 int 型指针 p，又声明了 int 型变量 i，然后取 i 的地址赋给指针 p，再给 i 赋值 6，因为 i 等于 6，所以第一个 cout 语句输出 Output int i=6。*p 表示 p 指向的变量的值，因为 p 存储的是 i 的地址，所以 p 指向 i，*p 的值也是 6，第二个 cout 语句输出 Output int pointer i=6。

鸡啄米再给大家强调几点：

1.我们可以声明指向常量的指针，这时候指针本身的值可以改变，也就是指针可以指向其他对象，但是我们不能通过指针改变它指向的值。例如：

```
const char *name="Tom"; //指向常量的指针
char s[]="Lili";
name=s;                    //正确，name 本身的值可以改变
*name='a';                 //编译时指出错误，不能通过 name 修改指向的对象
```

2.我们还可以声明指针常量，这时候指针本身的值不能改变，例如：

```
int  a=1;
int  b=2;
 int  *const  p=&a;    // 声明指针常量 p
 p=&b;              // 错误，不能改变指针常量 p 的值
```

3.我们进行指针赋值时可以将某个指针的值赋给相同类型的另一个指针。但是有一种特殊类型的指针，可以用任意类型对象的地址为之赋值，这就是 void 类型指针。我们在使用 void 类型的指针访问数据时需要进行强制类型转换。鸡啄米给大家一个 void 指针的例子：

```
 void  *p;
 int  *p1;
 int  a;
 void  main()
 {
        p=&a;        // void 类型的指针 p 指向整型变量 a
        p1=(int*)p;   // 用强制类型转换的方式将 void 指针 p 的指针赋给 int 型指针 p1
 }
```

二.指针运算

鸡啄米前面说过，指针也是一种数据类型，也可以参与一些运算。指针能够进行的运算有赋值运算、算术运算和关系运算。赋值运算鸡啄米上面刚讲过了。下面介绍算术运算和关系运算。

指针可以和整数进行加减运算，但是跟一般的加减运算不同，指针的加减运算跟指针的类型密切相关，例如，char 型指针 p 和整数 3，p+3 表示 p 所指地址后面第 3 个字符的地址，p-3 就表示 p 所指地址前面第 3 个字符的地址。给大家一个更直接的例子，有整型数组 int a[5]，p 指向数组首地址 a，则 p+3 表示首地址后第 3 个整数的地址，即 a[3]。

“指针++”和“指针--”表示指针所指地址的下一个或前一个数据的地址。

指针的算术运算一般都是在操作数组的时候进行，因为只有声明数组才可以得到连续的内存空间。如果我们对一个单独的变量地址进行加或减的算术运算，然后改变新地址的值，则可能会破坏该地址的数据，所以在对指针进行算术运算时要明确运算的结果是程序中分配可用的地址。

指针的关系运算说的是相同类型的指针之间的关系运算。不同类型的指针之间的关系运算没有任何意义，指针和非 0 整数的关系运算也没有任何意义，但是指针可以和 0 进行关系运算，后面鸡啄米会讲。两个相同类型的指针相等，表示这两个指针指向同一个地址。

今天讲的仍然是很重要的内容，鸡啄米希望大家掌握好，关键是理解，这些以后在软件开发中会经常用到。有什么问题可以在鸡啄米博客留言讨论。

鸡啄米：C++编程入门系列之二十九（数组、指针和字符串：指向数组元素的指针和指针数组）

鸡啄米这一讲继续给大家讲指针的知识，包括将指针用于数组元素的处理和指针数组。

一.指向指针元素的指针

前面说过，数组是有一定顺序关系的数组元素的集合，数组元素在内存中的存储地址是连续的。指针的加减运算特别适合处理这种存储在连续内存空间内的相同类型的数据，我们可以用指针方便的来处理数组元素的运算。

假设有一个声明：int a[10];，声明了一个一维整型数组，有 10 个元素。数组名 a 代表了数组 a 的首地址，它是一个指针常量，也就是不能改变，所以，注意，不能进行自增自减运算。首地址就是数组第一个元素的地址，所以 a 和&a[0]是完全等价的。a 中的 10 个元素是连续存放在内存中的，所以用数组名结合简单的算术运算就可以访问数组的元素。a 是数组第一个元素的地址，那么第 i+1 个元素也就是下标为 i 的元素的地址就是 a+i，第 i+1 个元素就是*(a+i)。例如，*a 就是 a[0]，*(a+7)就是 a[7]。

鸡啄米结合前面跟这一讲的内容给大家三种访问数组元素的方式的例子：

1.使用数组名和下标访问的方式

```
int main()
{
    int a[5];
    int i;
    for(i=0; i<5; i++)
        cin>>a[i];
    cout<<endl;
    for(i=0; i<5; i++)
        cout<<a[i]<<"  ";
    return 0;
}
```

2.使用数组名和指针运算访问的方式

```
int main()
{
    int a[5];
    int i;
    for(i=0; i<5; i++)
        cin>>a[i];
    cout<<endl;
```

```
    for(i=0; i<5; i++)
        cout<<*(a+i)<<"  ";
    return 0;
}
```

3.使用指针变量访问的方式

```
int main()
{
    int a[5];
    int *p;
    int i;
    for(i=0; i<5; i++)
        cin>>a[i];
    cout<<endl;
    for(p=a; p<(a+5); p++)
        cout<<*p<<"  ";
    return 0;
}
```

这几个例子很简单，鸡啄米就是让大家了解下几种访问数组元素的方式，大家在不同的情况下可以灵活运用。上面 3 个程序都是让用户先输入 5 个元素存入数组 a，然后顺序输出。

二.指针数组

前面鸡啄米讲到的数组中的元素或者是基本类型的变量或者是类的对象，同理，数组元素也可以是指针变量，而且每个指针变量的类型是相同的，这样的数组就是指针数组。

声明一维指针数组的语法形式是：数据类型 *数组名[下标表达式];。

数据类型指明指针数组元素的类型，就是确定每个元素指针指向什么类型的数据。数组名也是这个指针数组的首地址。下标表达式指明数组元素的个数。

比如，int *p[5];声明了一个整型指针数组，有 5 个元素。其中每个元素都指向整型数据。指针数组的每个元素都是指针，所以也需要先赋值才可以引用。

鸡啄米再给大家一个用指针数组输出两个一般数组的数据的例子：

```
#include <iostream>
using namespace std;
int main()
{
```

```
    int  array1[]={1,2,3};//声明一个数组
    int  array2[]={2,4,6};//声明另一个数组
    int  *p[2];      //声明整型指针数组
    p[0]=array1;        //初始化指针数组元素
    p[1]=array2;
    //输出两个数组的全部元素
    for(int  i=0;i<2;i++)     //对指针数组元素循环
    {
        for(int  j=0;j<3;j++)//对每个一般数组循环
        {
            cout<<p[i][j]<<"  ";
        }
        cout<<endl;
    }
    return  0;
}
```

上面的程序中先声明了两个整型数组 array1 和 array2，然后又声明了一个整型指针数组 p，使 p 的两个元素分别指向数组 array1 和 array2。p[0]就指向 array1，那么 p[0][0]就是 array1[0]，p[0][1]就是 array1[1]，p[0][2]就是 array1[2]，p[1]同理。这样利用指针数组的元素就可以访问数组 array1 和 array2 的数据并输出。程序运行结果是：

```
1  2  3
2  4  6
```

二维数组在内存中是按照行优先的方式顺序存放的，所以二维数组我们可以理解为一维指针数组，每个指针元素分别指向二维数组的一行，这个一维指针数组的元素个数就是二维数组的行数。

例如有一个二维数组：int array[2][3]={{1,2,3},{2,4,6}};。这是一个两行三列的二维数组，它就相当于一个一维整型指针数组，指针数组的元素个数就是二维数组 array 的行数 2，等价指针数组的两个元素可以表示为 array[0]和 array[1]，分别代表了二维数组 array 第 0 行和第 1 行的首地址，而 array 的每一行就相当于有 3 个元素的一维整形数组。

大家来看一个二维数组输出的例子：

```
#include  <iostream>
using  namespace  std;
int  main()
{
    int  array[2][3]={{1,2,3},{2,4,6}};
    for(int  i=0;i<2;i++)
```

```
    {
        for(int  j=0;j<3;j++)
        {
            cout<<*(*(array+i)+j)<<"  ";
            //也可以是  cout<<array2[i][j]<<"  ";
        }
        cout<<endl;
    }
    return  0;
}
```

程序运行结果是：

```
1  2  3
2  4  6
```

用数组指针访问二维数组元素的方式是：*(*(array+i)+j)。怎样理解这个表达式呢？array 是它的等价指针数组的首地址，那么 array+i 就是第 i（基数是 0）个指针元素，联系上面讲的，它指向的是二维数组的第 i 行的首地址，*(array+i)则是二维数组第 i 行的首地址，*(array+i)+j 就是二维数组第 i 行第 j（基数是 0）个元素的地址，所以*(*(array+i)+j)就是二维数组 array 第 i 行的第 j 个元素的值了。

指针数组可能有点难理解，大家多看几遍，思考思考，其实还是很清楚的。鸡啄米这一讲就讲这么多了，有问题欢迎到鸡啄米博客留言交流。

鸡啄米：C++编程入门系列之三十（数组、指针和字符串：指针用作函数参数、指针型函数和函数指针）

前一讲鸡啄米给大家讲了指向数组元素的指针和指针数组，这一讲给大家讲讲指针作函数参数的方法和作用，何为指针型的函数，以及指向函数的指针的概念和使用方法。编程入门时指针的内容是难点，望大家细心研究。

一.指针用作函数参数

以前我们学过的函数参数要么是基本数据类型的变量，要么是类的对象，又或者是数组名，前几讲学到的指针同样可以用作函数参数。指针作函数形参时，我们调用此函数将实参值传递给形参后，实参和形参指针变量将指向相同的内存地址，那么在被调函数中对形参指针所指向的对象的改变会一样影响主调函数中实参指针指向的对象。

指针用作函数参数用什么作用呢？其一，使形参指针和实参指针指向相同的内存地址，在被调函数中可以使用主调函数中的数据并可以改变主调函数中的数据，达到数据双向传递的效果。当然，前面讲过的引用也可以实现相同的作用。其二，用指针作函数参数传递数据

可以减少参数传递的开销，引用当然也可以实现这些。其三，可以通过指向函数的指针来传递函数代码的首地址。

指针和引用很多时候作用是一样的，引用相对指针来说可读性更好，但有时还是需要使用指针。

鸡啄米把之前讲引用调用时的例程给大家修改下说明指针怎样用作函数参数：

```
#include<iostream>
using namespace std;
void Swap(int *a, int *b);
int _tmain(int argc, _TCHAR* argv[])
{
	int x=5, y=10;
	cout<<"x="<<x<<"    y="<<y<<endl;
	Swap(&x, &y);
	cout<<"x="<<x<<"    y="<<y<<endl;
	return 0;
}
void Swap(int *a, int *b)
{
	int t;
	t=*a;
	*a=*b;
	*b=t;
}
```

这里的 Swap 函数使用了指针作为参数，我们在调用时传入的是 x 和 y 的地址，分别传递给了 a 和 b，然后在 Swap 内部改变了 a 和 b 指向的变量的值，而实际上它们指向的就是 x 和 y，所以 x 和 y 的值发生了同样的改变。程序运行结果依然是：

```
 x=5    y=10
x=10    y=5
```

二.指针型函数

函数都有自己的类型，除 void 类型的函数外都有自己的返回值。函数的返回值也可以是指针。返回值为指针类型的函数就是指针型函数。普通函数只能返回一个变量或对象，但指针型的函数可以在函数调用结束时将大量数据从被调函数返回到主调函数中，这就是它的好处。**注意不要返回局部变量的地址，因为出了被调函数局部变量就释放了，返回的地址中存放的内容也是无效的了。**

声明指针型函数的语法形式是：

```
数据类型 *函数名(参数表)
{
        函数体
}
```

数据类型指明了函数返回的指针的类型，“*”和函数名说明这是一个指针型函数，参数表是函数的形参列表。

三.函数指针

我们知道，程序运行时，数据会占用内存空间，实际上程序执行时代码也会调入内存，也会占用内存空间。函数名就是函数代码占用内存空间的首地址。函数指针就是用于存放函数代码首地址的变量。

我们也可以使用函数指针来调用函数，它和函数名实现的作用是一样的。声明方式也类似，也需要指明函数的返回值、形参列表，声明的语法形式是：

```
数据类型 (*函数指针名)(形参列表);
```

数据类型指明了函数指针所指向函数的返回值类型，函数指针名给出函数指针的名称，形参列表则说明了函数指针所指函数的形参类型和形参个数。

函数指针和一般的指针一样也要在使用之前先赋值，让它指向一个函数代码的首地址。为函数指针赋值的语法形式是：

```
函数指针名 = 函数名;
```

被指向的函数需要是声明过的已存在的，和函数指针必须具有相同的返回值类型、形参个数和形参类型。给函数指针赋值之后我们就可以通过函数指针名来调用它指向的函数了。

鸡啄米给大家一个函数指针的简单程序的例子：

```
#include <iostream>
using namespace std;
void show(int x)
{
      cout << x << endl;
}
 void (*show_pointer)(int);      // 声明一个 void 类型的函数指针

int main()
{
```

```
    int a = 10;
    show_pointer = show;        // 函数指针 show_pointer 指向 show
    show_pointer(a);            // 函数指针调用
    return 0;
}
```

上面例子中声明了一个 void 类型的函数指针：void (*show_pointer)(int);。我们通过给此函数指针赋值让其指向 void 类型的函数 show，然后用函数指针实现了对函数的调用，输出了 a 的值。程序运行结果是：10。

今天所讲内容中，指针作函数参数最常用，希望大家能够理解并掌握好。指针型函数和函数指针是两个比较容易混淆的概念，但实际意义又相去甚远，大家可以在很多软件开发公司的面试题中见到。有任何问题可以到鸡啄米博客留言交流。谢谢大家。

鸡啄米：C++编程入门系列之三十一（数组、指针和字符串：对象指针）

上一讲鸡啄米给大家讲的是指针用作函数参数、指针型函数和函数指针的知识，今天鸡啄米把指针方面的内容结个尾，讲讲对象指针。

一.什么是对象指针

每个变量都占有一定的内存空间，对象同样也需要占用内存空间。对象有数据成员和函数成员两种成员，但是实际上只有对象的数据成员会占用内存，函数成员则不会。我们可以通过对象名引用对象，也可以通过对象地址访问对象。对象指针就是存储对象地址的变量。声明对象指针的方式与一般的指针类似：

类名 *对象指针名;

使用对象名可以方便的访问对象的公有成员，同样使用对象指针也可以很容易的访问对象的公有成员。用对象指针访问对象的公有成员的形式为：

对象指针名->公有成员名;

鸡啄米让大家看一个简单的对象指针的例子：

```
#include <iostream>
using namespace std;
class CStudent
{
public:
    CStudent(int nAge=15)  { m_nAge = nAge; }     // 构造函数
    int GetAge()           { return m_nAge; }     // 内联函数，返回 m_nAge
```

```
private:
        int m_nAge;        // 私有数据
};
int main()
{
        CStudent student(17);      // 声明对象 student 并对其初始化
        CStudent *ptr;             // 声明对象指针
        ptr = &student;            // 初始化对象指针
        cout << student.GetAge() << endl;   // 通过对象名访问对象的成员
        cout << ptr->GetAge() << endl;      // 通过对象指针访问对象的成员
        return 0;
}
```

跟一般指针一样，对象指针在使用之前也必须先赋值，因为它必须先明确指向一个对象才能正常使用，否则可能会由于被赋与了随机值而有可能访问到重要地址破坏系统数据。上面的程序运行结果是：

```
17
17
```

二.this 指针

this 指针实际上就隐含于类的成员函数中，它指向成员函数正在操作的对象。构造函数和析构函数也不例外，也同样隐含了 this 指针。它也是一个指针，只是有点特殊。

比如上面的 CStudent 类的成员函数 GetAge 中的语句：return m_nAge;对于编译器来说相当于执行 return this->m_nAge;。为什么要有这个 this 指针呢？因为在执行这条语句时编译器需要知道返回的 m_nAge 到底属于哪一个对象，编译器事先已经把对象地址赋给 this 指针，调用成员函数操作数据成员时就隐含使用了这个 this 指针，这样就确定了访问的 m_nAge 属于哪个对象。

一般在软件开发中不会直接用到 this 指针访问数据成员，有时需要将此对象作为参数传递给某个函数时会使用 this 指针，亦或是写程序时忘记某个函数名，输入 this->后 vs2010 会提示出所有成员函数。

三.指向类的非静态成员的指针

类的成员都是变量、对象、函数等，我们同样可以定义存放它们的地址的指针，从而使指针指向对象的成员，通过指针就可以访问对象的成员了。但是通过这种指向成员的指针一样也只能访问公有成员。

声明指向非静态成员的指针的语法形式为：

类型说明符 类名::*指针名; // 声明指向公有数据成员的指针
类型说明符 (类名::*指针名)(形参表); // 声明指向公有成员函数的指针

指向成员的指针也要先声明再赋值，然后才能引用。给指向成员的指针赋值就是要说明此指针指向类的哪一个成员。

为指向数据成员的指针赋值的语法形式为：

指针名 = &类名::数据成员名;

在前面的地址相关运算中，鸡啄米讲到，用“&”运算符可以取到变量的地址，将其赋值给某个指针后就可以通过这个指针访问变量了。但是对于指向类的数据成员的指针就不同了，在类声明时并不会为类的数据成员分配内存空间，所以给指向数据成员的指针赋值只是确定了此指针指向哪个数据成员，同时也在指针中存放了该数据成员相对于类的起始地址的地址偏移量。这时通过赋值后的指针不能访问到具体的数据成员。

在声明了类的对象后就会为对象分配内存，这样根据对象的起始地址和指向数据成员的指针中存放的偏移量就可以访问到对象的数据成员了。通过对象和指向数据成员的指针访问公有数据成员的语法形式为：

对象名.*指向数据成员的指针名
或者 对象指针名->*指向数据成员的指针名

为指向成员函数的指针赋值的语法形式为：

指针名 = &类名::函数成员名;

在上面的形式为成员函数指针赋值以后还不能直接用此指针调用成员函数，必须先声明了类的对象，再通过对象和指向非静态成员函数的指针调用成员函数。调用的语法形式为：

(对象名.*指向成员函数的指针名)(形参表)
或者 (对象指针名->*指向成员函数的指针名)(形参表)

在为成员函数指针赋值时，还有用对象和成员函数指针调用成员函数时，我们都要注意成员函数指针的返回值类型和形参表一定要和指向的成员函数一致。

鸡啄米再以上面对象指针中的例子为基础修改下，让大家看看访问对象公有成员函数的几种方式：

```
int main()
{
    CStudent student(17);      // 声明对象 student 并对其初始化
    CStudent *ptr;             // 声明对象指针
    int (CStudent::*pGetAge)();// 声明指向成员函数 GetAge 的指针
```

```
    pGetAge = &CStudent::GetAge;// 为 pGetAge 赋值
    ptr = &student;           // 初始化对象指针
    cout << student.GetAge() << endl;  // 通过对象名访问成员函数
    cout << ptr->GetAge() << endl;     // 通过对象指针访问对象的成员函数
    cout << (student.*pGetAge)() << endl; // 通过成员函数指针访问成员函数
    return 0;
}
```

上面的例子中，先声明了一个 CStudent 类的对象 student 并初始化，又分别声明了一个指向对象 student 的指针 ptr 和指向成员函数 GetAge 的指针并各自初始化后，分别通过对象名、对象指针和指向成员函数的指针这三种方式访问公有成员函数 GetAge。程序的运行结果是：

```
17
17
17
```

四.指向类的静态成员的指针

上面说了类的非静态成员的指针的概念和使用方法，对于类的静态成员，我们可以用普通指针存放它的地址，通过普通指针访问它。

鸡啄米再把 CStudent 类的例子修改下，说明怎样通过指向类的静态数据成员的指针访问静态数据成员：

```
#include <iostream>
using namespace std;
class CStudent
{
public:
    CStudent(int nAge=15)  { m_nAge = nAge; m_nCount++; }     // 构造函数
    CStudent(CStudent &stu);                    // 拷贝构造函数
    int GetAge()          { return m_nAge; }    // 内联函数，返回 m_nAge
    static int m_nCount;  // 静态数据成员声明
private:
    int m_nAge;         // 私有数据
};
CStudent::CStudent(CStudent &stu)
{
    m_nAge = stu.m_nAge;
    m_nCount++;
}
```

```
int CStudent::m_nCount = 0;   // 静态数据成员初始化
int main()
{
        int *pCount = &CStudent::m_nCount;  // 声明一个 int 型的指针，指向静态数据成员 m_nCount
        CStudent student1(17);          // 声明对象 student1 并对其初始化
        cout << "student1:" << student1.GetAge() << endl;
        cout << "student id=" << *pCount << endl;     // 通过指针访问静态数据成员
        CStudent student2(student1);              // 声明对象 student2 并用 student1 初始化它
        cout << "student2:" << student2.GetAge() << endl;
        cout << "student id=" << *pCount << endl;     // 通过指针访问静态数据成员
        return 0;
}
```

程序运行结果是：

```
student1:17
student id=1
student2:17
student id=2
```

这一讲的内容就这些了，根据鸡啄米的经验，指向类的非静态成员的指针和指向类的静态成员的指针其实并不常用，大家可以重点看对象指针和 this 指针。如果有任何问题可以到鸡啄米博客留言讨论，希望我们在交流中一块提高。

鸡啄米：C++编程入门系列之三十二（数组、指针和字符串：动态内存分配和释放）

上一讲中鸡啄米讲的是对象指针，指针的相关内容就讲完了。今天鸡啄米给大家讲解下 C++编程入门时必须要掌握的又一个重点内容--动态内存分配。

动态内存分配最常见的就是用来分配一个某类型的数组。我们可以使用数组来处理大量的数据，但实际上很多情况下我们并不知道此数组会有多少个元素，所以在定义数组时到底定义多大的数组就要仔细考虑下了。比如，我们要对输入的若干个数进行分析，得出所有的正数存到一个数组里以备他用，这个正数数组应该定义多大呢？如果大了可能会造成内存的浪费，如果小了可能会出现数组越界的情况。这种情况下比较理想的是判断全部数据里有多少个正数就定义多大的数组，既不浪费内存也不会出现数组越界的问题，这就需要用到动态内存分配了。

动态内存分配指的就是在程序运行过程中根据实际情况动态分配适量的内存，使用完以后再释放。动态内存分配时申请和释放的内存单元叫做堆对象。申请和释放的过程一般叫做建立和删除。

一.C++动态内存分配运算符：new 和 delete

C++中 new 运算符用来动态分配内存，也可以称为动态创建堆对象。使用 new 运算符动态分配内存的语法形式为：

new 类型名(初值列表);

此语句用于申请分配一个内存空间，此内存空间存放由“类型名”表示的类型的数据，并用初值列表中列出的值初始化。该语句会返回一个由“类型名”表示的类型的指针，如果分配失败则返回 NULL。

我们动态建立一个普通变量时，初始化就是为其赋值。例如：

```
int *p;
p = new int(10);
```

上面的语句动态分配了用来存放一个整数数据的内存空间，同时把初始值 10 存入该内存空间中，也就是 p 指向的整形数据是 10，最后把分配内存的首地址赋值给 p。

如果我们动态建立的是一个类的对象，那么会根据初值列表调用该类合适的构造函数。

delete 运算符用来删除用 new 动态建立的堆对象，即释放 new 动态分配的堆内存。使用 delete 运算符的语法形式为：

delete 指针名;

如果被删除的是普通变量，则会直接释放动态分配的内存。如果被删除的是对象，则该对象的析构函数被调用。这里要注意一下，用 new 动态分配的内存只能用 delete 释放一次，如果释放第二次会出现错误。

鸡啄米写一个动态创建对象的例子：

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

class CStudent
{
public:
    CStudent()          { cout<<"Default constructor called."<<endl; }      //
```

无参数的构造函数

```
        CStudent(int nAge)    { m_nAge = nAge; cout<<"Constructor called."<<endl; }  // 带参数的构造函数
        ~CStudent()           { cout<<"Destructor called."<<endl;
 }                    // 析构函数
        int GetAge()          { return m_nAge; }      // 内联函数，返回 m_nAge
    private:
        int m_nAge;       // 私有数据
    };
    int main()
    {
        CStudent *ptr = new CStudent;       // 动态创建对象，没有给出初值列表，所以调用无参数的构造函数
        delete ptr;                         // 删除对象，自动调用析构函数
        ptr = new CStudent(16);             // 动态创建对象，给出了初值，所以调用带参数的构造函数
        delete ptr;                         // 删除对象，自动调用析构函数
        return 0;
    }
```

程序运行结果是：

```
Default constructor called.
Destructor called.
Constructor called.
Destructor called.
```

用运算符 new 也可以建立数组类型的堆对象，动态创建一维数组的语法形式为：

```
new 类型名[下标表达式];
```

上面的下标表达式指明了数组的元素个数。如果动态分配内存成功，会返回一个指向分配内存首地址的指针，如果分配失败则返回空指针 NULL。这里要注意的是，为数组动态分配内存时不能指定数组元素的初值。

用 new 动态建立的数组，也可以用 delete 删除，但是指针名前要加“[]”。delete 数组的语法形式为：

```
delete []指针名;
```

鸡啄米再给大家写一个动态创建对象数组的简单例子：

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

class CStudent
{
public:
    CStudent()              { cout<<"Default constructor called."<<endl; }          //
无参数的构造函数
    CStudent(int nAge)      { m_nAge = nAge; cout<<"Constructor called."<<en
dl; }   // 带参数的构造函数
    ~CStudent()             { cout<<"Destructor called."<<endl;
 }                          // 析构函数
    int GetAge()            { return m_nAge; }      // 内联函数，返回 m_nAge
private:
    int m_nAge;         // 私有数据
};
int main()
{
    CStudent *ptr = new CStudent[2];      // 动态创建对象数组，为每个对象元素
调用无参数的构造函数
    delete []ptr;                       // 删除对象数组，自动为每个对象调用析构函数
    return 0;
}
```

程序运行结果是：

```
Default constructor called.
Default constructor called.
Destructor called.
Destructor called.
```

用 new 动态创建多维数组的语法形式为：

new 类型名 T[下标表达式 1][下标表达式 2]...;

上面的下标表达式 1 可以是任何结果为正整数的表达式，而其他下标表达式必须是结果为正整数的常量表达式。这个语句如果执行成功则返回指向分配内存首地址的指针，但是这个指针不是 T 类型的指针，而是指向 T 类型数组的指针。

例如：int *p = new int[2][3];是错误的，因为这里 new 返回的是指向一个有 3 个元素的一维 int 型数组的指针，而 p 是一个指向 int 型数据的指针。这里大家可能有两个疑问：1.不是分配了一个二维数组吗，怎么返回的是指向一维数组的指针？2.怎样声明指向数组的指针？

鸡啄米为大家解答：1.我们可以把多维数组看成是由除第一维外其他维数据组成的一维数组，此一维数组的元素个数就是第一个下标表达式的值，比如上面的 int[2][3]可以看成是有 2 个元素的一维数组，每个元素又含有 3 个整数。new 创建多维数组返回的是指向一维数组第一个元素（也是数组）的指针，对于 new int[2][3]来说第一个元素实际上是二维数组的第一行，对于 new int[2][3][4]第一个元素实际上是一个 3x4 的二维数组。

2.声明指向数组的指针的形式是：

类型名 (*指针名)[下标表达式 1][下标表达式 2]...;

动态创建上面的二维数组的正确写法如：

```
int (*p)[3];          // 声明一个指向数组的指针
p = new int[2][3];    // 动态创建二维数组
```

可以按照下面的形式访问此二维数组的元素：

```
for (int i=0; i<2; i++)
{
    for (int j=0; j<3; j++)
    {
        *(*(p+i)+j) = 0;    // 通过指针访问数组元素
    }
}
```

二.继承自 C 语言的动态内存分配与释放的函数

C++语言除了可以使用 new 和 delete 运算符进行动态内存管理外，还继承了 C 语言的动态内存管理函数。

1.动态内存分配函数

void *malloc(size);参数 size 是要分配的字节数。函数执行成功时返回 void 类型的指针，执行失败时返回空指针 NULL。

2.动态内存释放函数

void free(void *memblock);参数 memblock 是指向要释放的内存的指针。此函数没有返回值。

关于动态内存分配的内容鸡啄米就讲这么多了，软件开发中会经常用到，所以大家在编程入门的时候多多思考，掌握的扎实些，以后会减少很多出错的机会。有问题欢迎来鸡啄米博客交流，或者给我发邮件沟通。

鸡啄米：C++编程入门系列之三十三（数组、指针和字符串：用字符数组存放和处理字符串）

上一讲鸡啄米讲了动态内存分配和释放的内容，今天要讲的是字符串，是编程入门必须要掌握牢的内容之一。

在编程系列之四（数据类型）中鸡啄米说过，字符串是用双引号括起来的字符序列。比如，"China"。字符串常量会在字符序列末尾添加'\0'作为结尾标记。字符串在内存中按照串中字符的排列顺序存放，并在末尾会添加'\0'作为结尾标记。对于 ASCII 码来说每个字符占一个字节，而对 Unicode 码来说每个字符占两个字节。本教程只讲 ASCII 编码的字符和字符串。

C++和 C 一样，基本数据类型的变量中没有字符串变量，那怎样处理字符串的存储和操作呢？C 语言中用字符数组存放字符串，操作同一般数组类似。C++语言继承了这种方式。除此之外，标准 C++库中还定义了 string 类。这一讲主要说下怎样通过字符数组的方式存储和操作字符串，string 类的相关内容下一讲会讲到。

一.字符数组的声明和使用

字符数组的声明和使用方法和其他类型的数组是一样的。声明字符数组的语法形式为：

```
char 字符数组名[下标表达式 1][下标表达式 2]...;
```

鸡啄米给大家一个有关字符数组声明和使用的简单例子：

```
#include <iostream>
using namespace std;
int main()
{
    char str[10] = {'I', ' ','l','o','v','e',' ','y','o','u' };    // 声明和初始化一维字符数组 str
    int i;
    for (i=0; i<10; i++)
    {
        cout << str[i];
    }
    cout << endl;
    return 0;
}
```

程序运行结果：

```
I love you
```

二.字符串用字符数组存放和处理

上面的例子中定义了字符数组并逐个处理数组中的元素，但存储的并不是字符串，因为结尾没有结束标记'\0'。如果我们在对字符数组进行初始化赋值时，末尾添加了'\0'就构成了字符串，但是用来存放字符串的数组的元素个数应该大于等于字符串的长度加 1。为字符数组初始化赋值时，初值可以是用逗号分隔的字符常量或 ASCII 码，也可以是字符串常量。对于字符串常量，结尾不用显式添加'\0'，‘\0’是隐式包含的。

鸡啄米根据上述几种为字符数组初始化赋值的方式，分别列举几个例子：

```
char str[5] = {108,111,118,101,0};          // 以逗号分隔的ASCII码为字符数组初始化赋值
char str[5] = {'l','o','v','e','\0'};          // 以逗号分隔的字符常量为字符数组初始化赋值
char str[5] = "love";                          // 以字符串常量为字符数组初始化赋值
```

字符串用字符数组存放时，我们可以按照上面的例程中逐个字符处理和输出，还可以将整个字符串一次性输入或者输出。例如：

```
一次性输入：char str[5];        cin >> str;
一次性输出：char str[5]="love";   cout << str;
```

将字符串一次性输入或输出时我们要注意：1.对于字符串结尾标记'\0'，输出字符串不会输出。2.输入多个字符串时需要用空格分隔，若要输入单个字符串则不能有空格，否则会被认为是多个字符串。3.输出字符串时，输出参数是字符数组名，遇到'\0'时输出结束。

鸡啄米举例说明下以上三点：

1.char str[5]="love"; cout << str;。字符串"love"的结尾隐含'\0'，输出时只会输出"love"而不会输出'\0'。

2.char str1[5],str2[5],str3[5]; cin >> str1 >> str2 >> str3;程序执行时输入"I love you"，则字符串 str1、str2 和 str3 分别被赋值"I"、"love"和"you"。如果改为 char str[11]; cin >> str;程序执行时输入"I love you"，则 str 被赋值为"I"。因为'I'后输入了空格，被认为是多个字符串，str 只用空格前的子字符串赋值。

3.输出字符串时 cout 参数只写字符数组名就可以了，比如 cout << str 就可以输出 str 字符串，遇到'\0'时输出结
束。

我们可以使用库中的字符串处理函数来处理字符串，比如，strcat 用来连接两个字符串，strcpy 用来拷贝字符串，strcmp 用来进行字符串的比较，strlen 用来计算字符串的长度。使

用这些函数之前需要先包含头文件 string.h。至于这些函数的详细说明和用法可以在 vs2010 的 msdn 中查询。

鸡啄米今天就讲到这里了，主要讲了用字符数组存储和操作字符串的方法，下一讲再讲标准 C++库中的 string 类。大家有问题可以来鸡啄米博客留言交流。

鸡啄米 :C++编程入门系列之三十四(数组、指针和字符串 :string 类)

在编程入门系列上一讲中鸡啄米说过，存放和处理字符串有两种方式，一种是用字符数组，另一种是用 C++标准类 string。今天鸡啄米就大概讲下 string 类及其用法。

C++从 C 语言继承了用字符数组存放字符串的方式，并且可以使用像 strcpy 等系统函数来处理字符串，但是这种方式大家应该也感觉不是很方便。数据与处理数据的函数分离开也不符合面向对象的设计思想。为此 C++提供了处理字符串的 string 类，其中封装了字符串的一些属性和操作字符串的函数。

string 类位于 C++标准库中。C++从 C 语言中继承了一些系统函数，另外又自己定义了一些模板和类，这些模板和类逻辑上分为六种类型：输入/输出类、容器类与 ADT（抽象数据类型）、存储管理类、算法、错误处理和运行环境支持。这些大家了解下就可以了。

C++标准库中的类的声明分别处于不同的头文件中，我们要使用这些类时就要包含相应的头文件。但是不同的编译器头文件也不同，VC++ 4.1 以前的版本中头文件名的形式为“*.h”，VC++ 4.2 版本开始就使用符合 ANSI 标准的标准 C++库，可以在不同的编译器间移植，头文件就不再有.h 扩展名了，比如，原来的 iostream.h 文件的新版本文件名为 iostream，但 C++标准库还是保留了来自 C 语言的 18 个带有.h 扩展名的头文件。我们在软件开发中，可以使用以前的带有.h 扩展名的头文件也可以使用新版本的头文件，编译器会自动判断链接哪一个库。但是注意，不能两种文件混着包含，比如，包含了头文件 iostream.h，又包含了新版本头文件 string，这种情况是不允许的。

在包含了标准 C++库中不带.h 扩展名的头文件后，必须加入指定名字空间的语句。例如：

```
#include <iostream>     // 包含 iostream 头文件
using namespace std;    // 指定名字空间 std
```

至于名字空间，鸡啄米在博客文章中会讲。这里大家只要记住这种规定就可以了。

string 类封装了字符串的属性和操作，这些属性包括：字符序列、字符序列的长度、一个字符的大小等等，还提供了一些对字符串的操作：查找、分配、连接和追加等。

鸡啄米给大家介绍下 string 类的部分成员函数，这里写的函数原型跟头文件中的不完全一样，是简化版。具体详细信息可以查看 msdn。

1.string 类的构造函数

string(); // 默认构造函数，创建一个长度为 0 的字符串

string(const string &ths); // 拷贝构造函数

string(const string &ths, unsigned int pos, unsigned int n); // 从字符串 ths 的位置 pos（第 pos+1 个字符）开始取 n 个字符，用来初始化 string 类的对象。提示：第 i 个字符的位置为 i-1

string(const char *s, unsigned int n); // 用指针 s 指向的字符串中的前 n 个字符初始化 string 类的对象

string(unsigned int n, char c); // 用 n 个重复的字符 c 来初始化 string 类的对象

2.string 类的一些常用成员函数

string 类有很多成员函数，鸡啄米就不一一讲了，这里只讲几个常用的成员函数。且称成员函数所属对象为“本对象”，其中存放的字符串为“本字符串”。

string append(const char *s); // 将指针 s 指向的字符串添加到本字符串尾

string assign(const char *s); // 讲指针 s 指向的字符串赋值给本对象

string& insert(unsigned int p0, const char *s); // 讲 s 指向的字符串插入到本字符串的位置 p0 之前

string substr(unsigned int pos, unsigned int n) const; // 从本字符串的位置 pos 开始取 n 个字符构成新的字符串对象返回

unsigned int find(const basic_string &str) const; // 在本字符串中查找 str 并返回第一次出现的位置

unsigned int length() const; // 返回本字符串的长度

void swap(string &str); // 交换本字符串与 str 的内容

int compare(const string &str) const; // 比较本字符串与字符串 str 的大小。若两个字符串相等则返回 0，若本字符串小于 str，则返回负数，若本字符串大于 str，则返回正数。

两个字符串 str1 和 str2 的比较有几点规则：a.若 str1 和 str2 的长度相同，并且字符都相同，则 str1 与 str2 相等。b.若 str1 和 str2 的字符不完全相同，就比较第一对不相同的字符的 ASCII 码，ASCII 码较小者所属的字符串就是较小的字符串。c.若 str1 的长度 n1 小于 str2 的长度 n2，并且 str1 与 str2 前 n1 个字符完全相同，则 str1 小于 str2。

3.string 类的操作符

操作符	举例	备注
+	str1+str2	将字符串 str1 和 str2 连接成一个字符串
=	str1=str2	将 str2 赋值给 str1
+=	str1+=str2	相当于 str1=str1+str2
==	str1==str2	判断 str1 与 str2 是否相等
!=	str1!=str2	判断 str1 与 str2 是否不相等
<	str1<str2	判断 str1 是否小于 str2
<=	str1<=str2	判断 str1 是否小于等于 str2
>	str1>str2	判断 str1 是否大于 str2

>=	str1>=str2	判断 str1 是否大于等于 str2
[]	str[i]	引用字符串 str 中位置为 i 的字符

鸡啄米最后给大家一个 string 类应用的例子：

```
#include <string>
#include <iostream>
using namespace std ;
void ShowBoolStr(int n)
{
      cout<<(n? "True": "False")<<endl;
}
int main()
{
       string str1="www.jizhuomi.com";
       string str2="http://jizhuomi.com";
        char s1[ ]="/software";
       char s2[ ]="/internet";
       cout<<"str1 为 "<<str1<<endl;
       cout<<"str2 为 "<<str2<<endl;
        cout<<"s1 为 "<<s1<<endl;
        cout<<"s2 为 "<<s2<<endl;
        cout<<"str1 的长度:"<<str1.length()<<endl;
      cout<<"str1<s1 的结果为 ";
       ShowBoolStr(str1<s1);
       cout<<"str2>=s2 的结果为 ";
       ShowBoolStr(str2>=s2);
        str1+=s1;
       cout<<"str1=str1+s1:"<<str1<<endl;
       cout<<"str1 的长度:"<<str1.length()<<endl;
       return 0;
}
```

程序运行结果为：

```
str1 为 www.jizhuomi.com
str2 为 http://jizhuomi.com
s1 为 /software
s2 为 /internet
str1 的长度:16
str1<s1 的结果为 False
str2>=s2 的结果为 True
```

str1=str1+s1:www.jizhuomi.com/software
str1 的长度:25

关于 string 类及其用法鸡啄米就讲到这了。如果以后大家从事 MFC 开发，可能 sting 类会很少用到，而主要用 MFC 中的 CString 类，但操作类似。希望大家以此对标准 C++库中的类有初步的认识。有问题可以到鸡啄米博客留言探讨。跟您的交流将促使我们共同进步。

第七部分：继承与派生

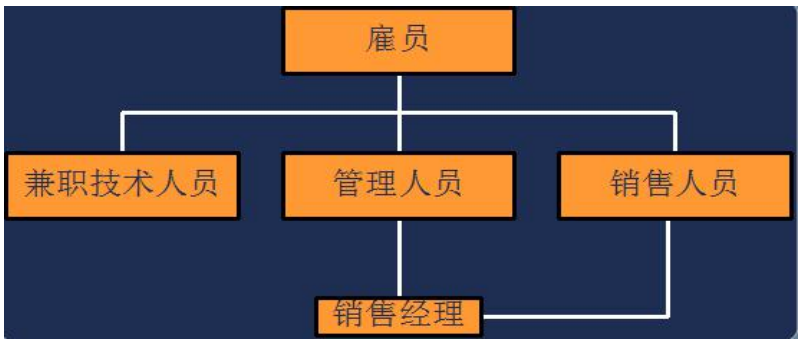
鸡啄米：C++编程入门系列之三十五（继承与派生：概念介绍与派生类的声明）

前面关于数组、指针和字符串的内容鸡啄米讲完了。现在开始鸡啄米将会介绍 C++编程入门的一个很重要的知识点--继承与派生。

一.继承与派生的概念

在之前的教程中我们学习了类的抽象性、封装性以及数据的共享等以后，我们就可以对于现实中的问题进行抽象和处理了。但是面向对象设计中代码的复用性和扩展性还没有体现出来。对于某个问题前人已经有了既有成果，我们怎样不做重复性劳动而直接运用？在问题有了新的发展以后我们怎样快速高效的修改或扩展现有的程序？这些都可以通过类的继承与派生来解决。

类是对现实中事物的抽象，类的继承和派生的层次结构则是对自然界中事物分类、分析的过程在程序设计中的体现。下图说明了某个公司雇员的派生关系。位于最高层的雇员其抽象程度最高，是最具一般性的概念。最下层抽象程度最低，最具体。从上层到下层是具体化的过程，从下层到上层是抽象话的过程。面向对象设计中上层与下层是基类与派生类的关系。



我们看到上图中此公司的雇员有三类：兼职技术人员、管理人员和销售人员。每个雇员都有姓名、级别和薪水等信息。每种雇员都可以升级，但升级方式不同。他们的月薪计算方式也不同，兼职技术人员应按实际工作小时数领取月薪，管理人员领取固定月薪，而销售人员是根据当月销售额领取提成。

这三类雇员的升级方式和月薪的计算方法等不同，所以不能用同一个类来描述，需要有三个类来分别抽象三类雇员。但这三个类中又有很多数据成员是一样的，例如姓名、级别和薪水等，函数成员也有很多相同的，只是可能实现方法不同，例如升级函数和计算月薪函数等。

我们应该先描述所有雇员的共性，再分别描述每类雇员。分别描述时应先说明他是雇员，然后描述他特有的属性和处理方法。这种描述方法在面向对象设计中就是类的继承与派生。对雇员共性进行描述就形成了基类，而对每类雇员的特性的描述可以通过从基类派生出子类来实现。

类的继承就是新类由已经存在的类获得已有特性，类的派生则是由已经存在的类产生新类的过程。这两个概念是两个相对的方向上的。

由已有类产生新类时，新类会拥有已有类的所有特性，然后又加入了自己独有的新特性。**已有类叫做基类或者父类，产生的新类叫做派生类或者子类。**派生类同样又可以作为基类派生新的子类，这样就形成了类的层次结构。

二.派生类的声明

派生类声明的语法形式为：

```
class 派生类名 : 继承方式 1 基类名 1, 继承方式 2 基类名 2, ... 继承方式 n 基类名 n
{
        派生类成员的声明;
}
```

例如，类 Parent1 和 Parent2 是已经声明过的类，类 Child 是由这两个类派生出的子类，则声明 Child 类的基本形式为：

```
class Child : public Parent1, private Parent2
{
public:
        Child();
        ~Child();
}
```

“基类名”（Parent1 和 Parent2）是已有类的名称，“派生类名”（Child）是从已有类产生的新类的名称。从上面的例子可以看到，**一个派生类可以有多个基类，这叫做多继承。**这种情况下派生类就同时具有多个基类的特性。**一个派生类如果只有一个基类，则叫做单继承。**我们在软件开发中用的最多的就是单继承，也就是一个派生类只有一个基类。同样，一个基类也可以产生多个派生类，这比多继承更常见。

基类产生派生类，派生类又可以作为基类再派生它自己的派生类，任何基类又可以产生多个派生类，这样就形成了一个类族。**直接派生出某个类的基类叫做这个类的直接基类，**

基类的基类或者更高层的基类叫做派生类的间接基类。例如，类 A 派生出类 B，类 B 派生出类 C，则类 A 是类 B 的直接基类，类 B 是类 C 的直接基类，而类 A 是类 C 的间接基类。

我们在声明派生类时，除了要指明基类，还要指定继承方式。**继承方式限定了派生类访问从基类继承来的成员的方式，指出了派生类成员或类外的对象对从基类继承来的成员的访问权限。**每个“继承方式”只限定紧随其后的基类。继承方式有：公有继承、保护继承和私有继承，分别对应的关键字是 public、protected 和 private。如果没有显式指定继承方式，则默认为私有继承。上面的例子中 Child 类继承 Parent1 类是公有继承，继承 Parent2 类是私有继承。关于继承方式的详细说明鸡啄米后面很快会讲。

派生类声明语法形式中的派生类成员指除了原封不动从基类中继承来的成员以外，修改的基类成员或者新增加的成员。派生类成员是对基类的扩展。

从基类继承产生派生类实现了对代码的复用，派生类中修改的基类成员或新增加的成员实现了对代码的扩展。这样继承与派生使得我们减少了重复性劳动，提高了软件开发效率，维护和扩展程序更容易。

鸡啄米今天主要讲了继承与派生的概念和派生类的声明方式，大家是不是对面向对象设计有了些更深入的认识了？如果有什么问题欢迎大家留言讨论。

鸡啄米：C++编程入门系列之三十六（继承与派生：派生类从基类继承的过程）

上一讲鸡啄米讲了继承与派生的概念和派生类的声明，继承与派生是编程入门的重要内容，是以后进阶的基础。这一讲是关于派生类从基类继承的过程。

在使用上一讲中讲到的声明方式声明派生类之后，再实现派生类的成员函数，这样派生类就算完整了。

根据前面所讲，**派生类从基类继承的过程可以分为三个步骤：吸收基类成员、修改基类成员和添加新成员。**吸收基类成员就是代码复用的过程，修改基类成员和添加新成员实现的是对原有代码的扩展，而代码的复用和扩展是继承与派生的主要目的。

鸡啄米基于上一讲中雇员的例子，写一个雇员类和其派生类-销售人员类。这里只给出部分声明，大家先了解下派生过程。

```
class  employee                  // 雇员类
{
public:
        employee();              // 构造函数
        ~employee();             // 析构函数
        void  promote(int);      // 升级函数
        void  getSalary();       // 计算工资
```

```
protected:
        char  *m_szName;          // 雇员姓名
         int    m_nGrade;         // 级别
        float  m_fSalary;         // 工资
};
class  salesman  :  public  employee
{
public:
        salesman();
        ~salesman();
        void  getSalary();        // 计算工资
private:
        float  m_fProportion;     // 提成比例
        float  m_fSalesSum;       // 当月总销售额
};
```

鸡啄米以上面的代码为例详细讲下三个步骤。

一.吸收基类成员

派生类从基类继承时首先就是吸收基类成员，将基类成员中除了构造函数和析构函数外的所有其他成员全部接收。这里要注意，基类的构造函数和析构函数都不能被派生类继承。上例中，employee 类除构造函数和析构函数外的所有成员：promote(int)函数、getSalary()函数，以及数据成员 m_szName、m_nGrade 和 m_fSalary，都被派生类 salesman 继承过来。

二、修改基类成员

派生类修改基类成员的方式有两种，一种是通过设置派生类声明中的继承方式，来改变从基类继承的成员的访问属性，这种方式下一讲会讲到；第二种是通过在派生类中声明和基类中数据或函数同名的成员，覆盖基类的相应数据或函数。**一旦我们在派生类中声明了一个和基类某个成员同名的成员，那么派生类这个成员就会覆盖外层的同名成员。这叫做同名覆盖。**需要注意的是，要实现函数覆盖不只要函数同名，函数形参表也要相同，如果函数形参表不同只有名字相同则属于前面所说的重载。

上例中，salesman 类的 getSalary 函数覆盖了 employee 类的同名函数。比如，我们定义了一个 salesman 类的对象 A，则 A.getSalary()调用的是 salesman 类中的 getSalary 函数而不是基类 employee 中的。

三、添加新成员

代码的扩展是继承与派生的主要目的之一，而添加新成员是实现派生类在基类基础上扩展的关键。上例中，派生类 salesman 就添加了两个新数据成员 m_fProportion 和 m_fSa

lesSum。可见，能够添加新成员还是很方便的，我们在软件开发中可以根据实际需要为派生类添加新的数据成员或函数成员。

上面说过，派生类不能从基类继承构造函数和析构函数。但是派生类同样需要有初始化和清理，所以我们需要为派生类添加新的构造函数和析构函数，就像上例中，派生类 salesman 中就添加了新的构造函数 salesman()和新的析构函数~salesman()。

鸡啄米后面会根据这三个步骤加以详细讲解。吸收基类成员是伴随着派生类的声明完成的，这个就不需要讲了；通过派生类的继承方式改变基类成员的访问属性下一讲会讲到；关于添加新成员，主要讲为派生类添加构造函数和析构函数的方式，添加一般新成员和类声明时类似，可以参考下编程入门系列之十三中类的声明。

鸡啄米今天所讲内容不算多，希望大家好好理解消化。有什么编程入门问题仍然欢迎到鸡啄米博客留言交流。

鸡啄米：C++编程入门系列之三十七（继承与派生：派生类对基类成员的访问控制之公有继承）

在编程入门系列上一讲中鸡啄米讲了派生类从基类继承的过程，其中提到了通过继承方式可以控制对从基类继承的成员的访问属性，这一讲主要讲解公有继承方式怎样控制访问属性。

前面说过，派生类从基类继承时有三个步骤，第一个步骤是吸收基类成员，吸收了基类中除构造函数和析构函数之外的所有数据成员和函数成员，第二个步骤就是修改基类成员，包括修改对基类成员的访问属性和覆盖基类成员，第三个步骤是添加新成员。第二个步骤中修改对基类成员的访问属性可以通过派生类的继承方式控制。

类的成员有 public（公有）、protected（保护）和 private（私有）三种访问属性。**类的继承方式也有 public（公有继承）、protected（保护继承）和 private（私有继承）三种。**派生类通过不同的继承方式可以获得对基类成员的不同的访问属性。派生类对基类成员的访问主要有两种，**一种是派生类的新增成员对继承的基类成员的访问，另一种是派生类的对象对继承的基类成员的访问。**下面会详细讲。

鸡啄米这里插一句本应属于“类成员的访问控制”中的话，**通过对象访问类的成员属于外部访问，只能访问类的公有成员。**

派生类的继承方式为 public，即公有继承时，对基类中的公有成员和保护成员的访问属性都不变，而对基类的私有成员则不能访问。具体说，就是基类的公有成员和保护成员被继承到派生类中以后同样成为派生类的公有成员和保护成员，派生类中新增成员对他们可以直接访问，派生类的对象只能访问继承的基类公有成员。但是派生类的新增成员和派生类的对象都不能访问基类的私有成员。

鸡啄米给大家写一个公有继承的简单例子：

```
#include <iostream>
using namespace std;
class Base          // 基类 Base 的声明
{
```

```
public:              // 公有成员函数
        void SetTwo(int a, int b) { x=a; y=b; }
        int GetX()   { return x; }
        int GetY()   { return y; }
private:             // 私有数据成员
        int x;
        int y;
};
class Child : public Base    // 派生类的声明，继承方式为公有继承
{
public:                  // 新增公有成员函数
      void SetThree(int a, int b, int c) { SetTwo(a, b); z=c; }
      int GetZ()   { return z; }
private:                 // 新增私有数据成员
       int z;
};
int main()
{
      Child child;          // 声明 Child 类的对象
      child.SetThree(1, 2, 3); // 设置派生类的数据
      cout << "The data of child:"<<endl;
      cout << child.GetX() << "," << child.GetY() << "," << child.GetZ() << endl;
      return 0;
}
```

程序运行结果是：

```
The data of child:
1,2,3
```

上面的程序声明了一个基类 Base，又声明了 Base 类的派生类 Child，最后是主函数部分。派生类 Child 从基类 Base 中继承了除构造函数和析构函数外的所有数据成员和函数成员，这些再加上派生类 Child 的新增成员就组成了 Child 类的全部。类 Child 的继承方式为公有继承，基类 Base 的所有公有成员在派生类 Child 中的访问属性不变，都可以直接访问，所以 Child 类的 SetThree 函数可以直接调用 Base 类的 SetTwo 函数。基类公有成员 SetTwo、GetX 和 GetY 都变成了 Child 类外部接口的一部分。但是上面说过，派生类不能访问基类的私有成员，所以 Child 类不能访问 Base 类的 x 和 y。

主函数中首先定义了派生类 Child 的对象 child，然后通过对象 child 调用了派生类 Child 的新增公有函数 SetThree 和 GetZ，还调用了从基类 Base 继承的公有成员函数 GetX 和 GetY。

通过这个例子，派生类对基类成员的两种访问方式：派生类的新增成员对继承的基类成员的访问和通过派生类的对象对继承的基类成员的访问，都讲到了。

因为这部分访问属性控制的内容比较重要而且需要慢慢理解，鸡啄米后面再单独讲保护继承和私有继承，大家慢慢领会。有问题欢迎到鸡啄米博客留言讨论。

鸡啄米：C++编程入门系列之三十八（继承与派生：派生类对基类成员的访问控制之保护继承与私有继承）

鸡啄米在编程入门系列上一节中讲了公有继承方式的访问属性，这一节接着讲剩下的两种继承方式：保护继承和私有继承。

一.保护继承

在保护继承方式中，基类的公有成员和保护成员被派生类继承后变成派生类的保护成员，而基类的私有成员在派生类中不能访问。因为基类的公有成员和保护成员在派生类中都成了保护成员，所以派生类的新增成员可以直接访问基类的公有成员和保护成员，而派生类的对象不能访问它们，上一讲鸡啄米说过，**类的对象也是处于类外的，不能访问类的保护成员。**对基类的私有成员，派生类的新增成员函数和派生类对象都不能访问。

通过上述保护继承的讲解，相信大家对类的保护成员的访问属性有更加深刻的理解了。假设 A 类是基类，B 类是从 A 类继承的派生类，A 类中有保护成员，则对派生类 B 来说，A 类中的保护成员和公有成员的访问权限是一样的。而对 A 类的对象的使用者来说，A 类中的保护成员和私有成员都一样不能访问。可见类中的保护成员可以被派生类访问，但是不能被类的外部对象（包括该类的对象、一般函数、其他类等）访问。我们可以利用保护成员的这个特性，在软件开发中充分考虑数据隐藏和共享的结合，很好的实现代码的复用性和扩展性。

鸡啄米举个简单的例子讨论下保护成员的访问属性。

```
class Base
{
protected:
        int x;          // 基类的保护成员
};
int main()
{
        Base base;
        base.x = 0;     // 编译报错
        return 0;
}
```

这段代码在编译的时候会报错，错误就出在通过对象 base 访问保护成员 x 时，就像上面讲的，对 Base 类的对象 base 的使用者来说，Base 类中的保护成员 x 和私有成员的访问特性是一样的，所以对象 base 不能访问 x，这样跟使用私有成员一样通过保护成员实现了数据的隐藏。

```
class  Base
{
protected:
        int  x;          //  基类的保护成员
};
class  Child  :  public  Base
{
public:
        void  InitX();
};
void  Child::InitX()
{
        x  =  0;
}
```

对上面的派生类 Child 来说，基类 Base 中的保护成员 x 和公有成员的访问权限一样，所以 Child 类的成员函数 InitX 可以访问 Base 类的保护成员 x。

二.私有继承

在私有继承方式中，基类的公有成员和保护成员被派生类继承后变成派生类的私有成员，而基类的私有成员在派生类中不能访问。派生类的新增成员可以直接访问基类的公有成员和保护成员，但是在类的外部通过派生类的对象不能访问它们。而派生类的成员和派生类的对象都不能访问基类的私有成员。

讲到这里，我们看到不管是保护继承还是私有继承，在派生类中成员的访问特性都是一样的，都是基类的公有和保护成员可以访问，私有成员不能访问。但是派生类作为基类继续派生新类时，两种继承方式就有差别了。例如，A 类派生出 B 类，B 类又派生出 C 类，如果 B 类是以保护继承方式从 A 类继承的，则 A 类的公有成员和保护成员都成为 B 类的保护成员，再由 B 类派生出 C 类时，原来 A 类的公有成员和保护成员间接继承到 C 类中，成为 C 类的保护成员或者私有成员（C 类从 B 类公有继承或保护继承时为前者，私有继承时为后者），所以 C 类的成员可以间接访问 A 类的公有成员和保护成员。但是如果 B 类是以私有继承方式从 A 类继承的，则 A 类的公有成员和保护成员都成为 B 类的私有成员，A 类的私有成员不能在 B 类中访问，B 类再派生出 C 类时，原来 A 类的所有成员都不能在 C 类中访问。

由以上分析得出，私有继承使得基类的成员在其派生类后续的派生中不能再被访问，中止了基类成员继续向下派生，这对代码的复用性没有好处，所以一般很少使用私有继承方式。

鸡啄米将上一讲中的例子由公有继承改为私有继承，继而更形象的说明私有继承的特性。

```
#include <iostream>
using namespace std;
class Base          // 基类 Base 的声明
{
public:             // 公有成员函数
    void SetTwo(int a, int b) { x=a; y=b; }
    int GetX()  { return x; }
    int GetY()  { return y; }
private:            // 私有数据成员
    int x;
    int y;
};
class Child : private Base   // 派生类的声明，继承方式为私有继承
{
public:                 // 新增公有成员函数
    void SetThree(int a, int b, int c) { SetTwo(a, b); z=c; }
    int GetX()  { return Base::GetX(); }
    int GetY()  { return Base::GetY(); }
    int GetZ()  { return z; }
private:                // 新增私有数据成员
    int z;
};
int main()
{
    Child child;        // 声明 Child 类的对象
    child.SetThree(1, 2, 3); // 设置派生类的数据
    cout << "The data of child:"<<endl;
    cout << child.GetX() << "," << child.GetY() << "," << child.GetZ() << endl;
    return 0;
}
```

程序运行结果：

```
The data of child:
1,2,3
```

Child 类从 Base 类中私有继承，Base 类中的公有成员 SetTwo()、GetX()和 GetY()成为 Child 类的私有成员，在 Child 类中可以直接访问它们，例如 Child 类的成员函数 SetThree()中直接调用了 Base 类的公有成员函数 SetTwo()。Base 类的私有成员 x 和 y 在 Child 类中不能访问。在外部通过 Child 类的对象不能访问 Base 类的任何成员，因为 Base 类的公有成员成为 Child 类的私有成员，Base 类的私有成员在 Child 类中不能访问。那么 Base 类的作为外部接口的公有成员 SetTwo()、GetX()和 GetY()都被派生类 Child 隐藏起来，外部不能通过 Child 类的对象直接调用。

如果我们希望派生类也提供跟基类中一样的外部接口怎么办呢？我们可以在派生类中重新定义重名的成员。上面的 Child 类就重新定义了公有成员函数 GetX()和 GetY()，函数体则只有一个调用基类函数的语句，照搬了基类函数的功能。因为派生类中重新定义的成员函数的作用域位于基类中同名函数的作用域范围的内部，根据前面可见性中讲的同名覆盖原则，调用时会调用派生类的函数。通过这种方式可以对继承的函数进行修改和扩展，在软件开发中经常会用到这种方法。

大家可能发现，main 函数的函数体跟上一讲例子中的完全相同，但实际上在程序执行的时候是不同的，这里调用的函数 GetX()和 GetY()都是派生类 Child 的函数，由于是私有继承，基类 Base 中的同名函数都不能通过 Child 类的对象访问。

这个例子跟上一讲的例子相比，Base 类和主函数 main 的函数体都没有修改，鸡啄米只修改了派生类 Child 的声明，但 Child 类的外部接口没有改变。到此大家有没有认识到面向对象设计封装性的优越性？我们可以根据需要调整类的内部数据结构，但只要保持外部接口不变，那我们做的类的内部调整对外部就是透明的，不会影响到程序的其他部分。这充分体现了面向对象设计的可维护性和可扩展性。

关于派生类以各种继承方式派生后对基类成员的访问控制就讲到这里了，几种继承方式中最常用的就是公有继承了，但保护继承和私有继承大家也有必要学习一下，对于提升内功很有帮助。有问题到鸡啄米博客留言交流吧。

鸡啄米：C++编程入门系列之三十九（继承与派生：派生类的构造函数）

派生类通过公有继承、保护继承和私有继承控制对基类成员的访问属性在鸡啄米编程入门的前两节讲完了。今天讲讲派生类的构造函数。

鸡啄米前面说过，基类的构造函数和析构函数派生类是不能继承的。如果派生类需要对新成员初始化或者进行特定的清理工作，就需要自己定义构造函数和析构函数了。从基类继承的成员的初始化仍可通过基类的构造函数来完成。

我们使用派生类的对象以前需要对它的数据成员进行初始化赋值。派生类的数据成员包括从基类继承来的数据成员和派生类新增的数据成员，还可能包括其他类的对象作为其数据成员，包括其他类的对象时实际上还间接包括了这些对象的数据成员。那么我们**对派生类初始化时就需要对基类的数据成员、派生类新增数据成员和内嵌的其他类对象的数据成员**

进行初始化。由于不能继承基类的构造函数，派生类就必须增加自己的构造函数。派生类的构造函数需要做的工作有，使用传递给派生类的参数，调用基类的构造函数和内嵌对象成员的构造函数来初始化它们的数据成员，再添加新语句初始化派生类新成员。派生类构造函数的语法形式为：

派生类名::派生类名(参数表):基类名 1(参数表 1),...基类名 m(参数名 m),

内嵌对象名(内嵌对象参数表 1),...,内嵌对象名 n(内嵌对象参数表 n)

{

初始化派生类新成员的语句;

}

鸡啄米先讲下上面的语法形式吧。派生类的构造函数名同样也要与类名相同。构造函数参数表要给出初始化基类数据成员、新增数据成员和内嵌对象的数据成员的所有参数。在给出所有这些参数以后就要像上面那样指明所有要初始化的基类名及其参数表，还有内嵌对象名及其参数表。各个基类名和内嵌对象名可以以任何的顺序排列。

可能大家觉得此语法形式太过复杂，但其实构造函数也不是必须写这么复杂。假设一个类继承自多个基类，即为多继承时，对于那些构造函数有参数的基类就必须显式给出基类名及其参数表，而对于那些使用默认构造函数的基类没有必要给出基类名及其参数表，同理内嵌对象的构造函数若有参数也必须给出内嵌对象名及其参数表，若使用默认构造函数也没有必要给出内嵌对象名及其参数表。试想，如果派生类只有一个基类，而且有默认构造函数，没有内嵌对象或者可以使用其他公有函数成员初始化内嵌对象，那么后面的基类名和内嵌对象名就都不需要了，是不是形式就很简单了？其实我们最常用的比这种最简单的情况复杂不了多少。

鸡啄米再次强调，**基类的构造函数若有参数，则派生类必须定义构造函数，将传入的参数再传递给基类的构造函数，对基类进行初始化**。若基类没有定义构造函数，则派生类也可以不定义构造函数，都使用默认构造函数，对于派生类的新增数据成员可以通过其他的公有函数成员来初始化。而如果基类同时定义了默认构造函数和带参数的构造函数，那么在派生类的构造函数中可以给出基类名及其参数表，也可以不显式给出。

构造派生类的对象调用构造函数时的处理顺序是：1.首先调用基类的构造函数，若有多个基类，调用顺序按照它们在派生类声明时从左到右出现的顺序；2.如果有内嵌对象成员，则调用内嵌对象成员的构造函数，若为多个内嵌对象，则按照它们在派生类中声明的顺序调用，如果无内嵌对象则跳过这一步；3.调用派生类构造函数中的语句。

这里需要说明的是，**基类和内嵌对象成员的构造函数的调用顺序和它们在派生类构造函数中出现的顺序无关**。

鸡啄米这里给一个例子说明下派生类的构造函数。

派生类 Child 有三个基类 Base1、Base2 和 Base3，继承方式都是公有继承，Base1 只有一个默认构造函数，Base2 类和 Base3 类都只有一个有参数的构造函数，Child 类有三个内嵌对象成员，分别是 Base1 类、Base2 类和 Base3 类的对象。

```
#include <iostream>
using namespace std;
class Base1          // 基类 Base1，只有默认构造函数
{
public:
        Base1()        { cout<<"Base1 construct"<<endl; }
};
class Base2          // 基类 Base2，只有带参数的构造函数
{
public:
        Base2(int x)    { cout<<"Base2 construct "<<x<<endl; }
};
class Base3          // 基类 Base3，只有带参数的构造函数
{
public:
        Base3(int y)    { cout<<"Base3 construct "<<y<<endl; }
};
class Child : public Base2, public Base1, public Base3   // 派生类 Child
{
public:
        Child(int i,int j,int k,int m):Base2(i),b3(j),b2(k),Base3(m)    { }
private:            // 派生类的内嵌对象成员
        Base1 b1;
        Base2 b2;
        Base3 b3;
};
int main()
{
        Child child(3,4,5,6);
        return 0;
}
```

由于上面的类 Base2 和类 Base3 都有带参数的构造函数，所以派生类 Child 必须定义带参数的构造函数。Child 类构造函数的参数表给出了基类 Base2 和 Base3、内嵌对象成员 b2 和 b3 需要的所有参数。因为基类 Base1 只有默认构造函数，没有参数，所以类 Base1 和 Base1 类的对象成员 b1 都不需要列在派生类的构造函数中。Child 类的构造函数中基类 Base2 和 Base3 以及对象成员 b2 和 b3 可以按照任意的顺序排列。

主函数 main 中声明了派生类 Child 的对象 child，构造对象 child 时会调用 Child 类的构造函数，执行时先调用基类的构造函数，调用顺序按照它们在派生类声明时从左到右出现的顺序，即按照先 Base2 再 Base1 后 Base3 的顺序，调用基类构造函数后就调用内嵌对象成

员的构造函数，按照它们在派生类中声明的顺序调用，即按照先 b1 再 b2 后 b3 的顺序。基类 Base1 和 Base1 类的对象成员 b1 没有显式列出，执行调用 Base1 类的默认构造函数。

程序运行结果为：

```
Base2  construct  3
Base1  construct
Base3  construct  6
Base1  construct
Base2  construct  5
Base3  construct  4
```

此运行结果也完全验证了鸡啄米上面的分析。

关于派生类的构造函数鸡啄米就先讲到这了，下一节会讲派生类的析构函数。有问题欢迎到鸡啄米博客讨论。

鸡啄米：C++编程入门系列之四十（继承与派生：派生类的析构函数）

鸡啄米在 C++编程入门系列之三十九中讲了派生类的构造函数，提到构造函数就不能不说析构函数，这一节主要讲讲派生类析构函数的相关内容。

派生类的析构函数也是在派生类对象释放的时候进行清理工作的。前面说过，派生类无法继承基类的析构函数，所以如果需要的话就要自己定义析构函数。派生类析构函数的定义方式与一般类的析构函数是一样的，也是没有返回类型，没有参数，所以比构造函数需要注意的东西少多了。

派生类的析构函数一般只需要在其函数体中清理新增成员就可以了，对于继承的基类成员和派生类内嵌对象成员的清理，则一般由系统自动调用基类和对象成员的析构函数来完成。这个执行过程的顺序正好和派生类构造函数相反：1.执行析构函数语句清理派生类的新增成员；2.调用内嵌对象成员所属类的析构函数清理派生类内嵌对象成员，各个对象成员的清理顺序与其在构造函数中的构造顺序相反；3.调用基类的析构函数清理继承的基类成员，如果是多继承则各个基类的清理顺序也与其在构造函数中的构造顺序相反。总起来一句话，**析构函数执行时所有成员或对象的清理顺序与构造函数的构造顺序刚好完全相反**。

上一讲的例程中所有类都没有定义析构函数，这时系统会为每个类生成默认析构函数，由他们完成清理工作。

鸡啄米讲上一讲的例子稍加改造，为每个基类显示定义析构函数，然后大家看下程序的执行顺序。

```
#include <iostream>
using namespace std;
class Base1          // 基类 Base1
{
public:
        Base1()       { cout<<"Base1 construct"<<endl; }      // Base1 的构造函数
        ~Base1()      { cout<<"Base1 destruct"<<endl; }       // Base1 的析构函数
};
class Base2          // 基类 Base2
{
public:
        Base2(int x)   { cout<<"Base2 construct "<<x<<endl; }   // Base2 的构造函
数
        ~Base2()      { cout<<"Base2 destruct"<<endl; }       // Base2 的析构函数
};
class Base3          // 基类 Base3
{
public:
        Base3(int y)   { cout<<"Base3 construct "<<y<<endl; }   // Base3 的构造函
数
        ~Base3()      { cout<<"Base3 destruct"<<endl;}        // Base3 的析构函数
};
class Child : public Base2, public Base1, public Base3   // 派生类 Child
{
public:
        Child(int i,int j,int k,int m):Base2(i),b3(j),b2(k),Base3(m)   { }
private:             // 派生类的内嵌对象成员
        Base1 b1;
        Base2 b2;
        Base3 b3;
};
int main()
{
        Child child(3,4,5,6);
        return 0;
}
```

三个基类 Base1、Base2 和 Base3 都添加了析构函数，派生类 Child 没有添加，系统会为派生类生成默认析构函数。主函数的函数体没有变。程序执行时会先调用 Child 类的构造函数构造 child 对象，然后调用 Child 类的默认析构函数完成清理工作。构造函数的执行

过程上一讲已经讲过了，Child 类的默认析构函数会依次调用内嵌对象成员的析构函数和基类的析构函数，执行顺序和构造函数完全相反。

上面程序的运行结果是：

```
Base2  construct  3
Base1  construct
Base3  construct  6
Base1  construct
Base2  construct  5
Base3  construct  4
Base3  destruct
Base2  destruct
Base1  destruct
Base3  destruct
Base1  destruct
Base2  destruct
```

从程序的执行结果来看，和我们前面的分析完全一致，析构函数的执行顺序与构造函数完全相反。

程序虽然就简单一小段，但执行过程还需要大家认真思考，分析透彻。这些都是以后进阶的基础。关于派生类的构造函数和析构函数鸡啄米就讲完了，有问题欢迎到鸡啄米博客交流。

鸡啄米：C++编程入门系列之四十一（继承与派生：作用域分辨符）

鸡啄米在编程入门系列的前两节中讲了派生类的构造函数和析构函数，现在来讲讲通过作用域分辨符访问派生类成员的问题。

访问派生类的成员有可见性和唯一性两个问题，我们只能访问到具有唯一性的可见成员。

先说说可见性的问题。前面 C++编程入门系列之十八的作用域和可见性中讲过，如果有两个以上具有包含关系的作用域，外层作用域中的标识符在内层作用域中没有同名标识符则它在内层作用域中也是可见的，但是如果在内层作用域中存在同名标识符则外层的标识符被屏蔽，这也叫做同名覆盖。派生类在继承时，基类的成员和派生类的新增成员都有类作用域，但是是两个具有包含关系的作用域，派生类作用域位于内层。如果在派生类中存在一个和基类某数据成员同名的数据成员，或者和基类某成员函数的名称和参数表都相同的成员函数，则派生类中的新成员就覆盖了基类成员，不管是在派生类内还是在派生类外部只能通过

成员名访问到派生类的成员，而访问不到基类成员。如果需要在派生类中访问基类中的同名成员怎么办呢？

我们可以通过基类名和作用域分辨符来访问基类中的同名成员。作用域分辨符就是“::”，在派生类内部访问基类同名成员的语法形式是：

```
基类名::数据成员名;          // 数据成员
基类名::函数成员名(参数表);   // 函数成员
```

如果是在派生类外通过派生类对象访问的话，前面还要加上“派生类对象名.”：

```
派生类对象名.基类名::数据成员名;                // 数据成员
派生类对象名.基类名::函数成员名(参数表);   // 函数成员
```

这里的基类名就限定了后面的成员属于哪个类。

如果是多继承的话，要考虑派生类的基类有没有共同基类的问题。鸡啄米先讲多个基类之间没有继承关系也没有共同基类的情况。这种情况下**如果多个基类具有同名成员，派生类也新增了同名成员，则派生类成员会覆盖所有基类中的同名成员。**通过成员名只能访问到派生类的成员，要访问各基类的同名成员就需要使用作用域分辨符。而如果派生类中不存在同名成员，访问多个基类的同名成员也需要使用作用域分辨符，因为从不同基类继承过来的同名成员具有相同的作用域，通过成员名无法唯一标识成员，所以也需要作用域分辨符来分辨。

鸡啄米给大家一个多继承情况下同名覆盖的例子：

派生类 Child 由基类 Base1 和 Base2 公有继承而来，两个基类有同名数据成员 x 和同名函数成员 show，派生类 Child 又新增了同名的数据成员 x 和同名的函数成员 show，这样派生类 Child 中就一共有 6 个成员，三个同名的数据成员和三个同名的函数成员。

```
#include <iostream>
using namespace std;
class Base1                 // 基类 Base1 的声明
{
public:
        int x;
        void show()       { cout<<"x of Base1: "<<x<<endl; }
};
class Base2                 // 基类 Base2 的声明
{
public:
        int x;
        void show()       { cout<<"x of Base2: "<<x<<endl; }
};
class Child : public Base1, public Base2    // 派生类 Child 的声明
{
```

```
public:
    int x;
    void show()     { cout<<"x of Child: "<<x<<endl; }
};
int main()
{
    Child child;
    child.x = 5;          // 访问派生类数据成员
    child.show();         // 调用派生类函数成员
    child.Base1::x = 7;   // 使用作用域分辨符访问基类 Base1 的数据成员
    child.Base1::show();  // 使用作用域分辨符调用基类 Base1 的函数成员
    child.Base2::x = 8;   // 使用作用域分辨符访问基类 Base2 的数据成员
    child.Base2::show();  // 使用作用域分辨符访问基类 Base2 的函数成员
    return 0;
}
```

程序运行结果：

```
x of Child: 5
x of Base1: 7
x of Base2: 8
```

主函数 main 中声明了派生类 Child 的对象 child，因为同名覆盖，所以通过成员名只能访问派生类 Child 的成员，要访问基类 Base1 和 Base2 的同名成员就需要像上面那样使用作用域分辨符访问。如果在派生类 Child 的成员函数 show 中访问基类 Base1 的同名成员，比如 x，则可以将 Child 的 show 函数修改为：void show() { cout<<"x of Child: "<<Base1::x<<endl; }。

如果上例中的派生类 Child 中没有定义与基类成员同名的成员，则通过成员名就访问不到任何成员，因为继承的 Base1 和 Base2 的同名成员具有相同的作用域，系统无法唯一标识它们。如果要访问基类 Base1 或 Base2 的同名成员就需要使用作用域分辨符。

将上例中派生类 Child 的新增同名成员去掉，改为：

```
class Child : public Base1, public Base2
{
};
```

程序其余部分不变，则主函数 main 中的语句 child.x = 5;和 child.show();就会编译报错，因为这两个标识符具有二义性，系统无法唯一标识它们，不知道该访问哪个成员。只能通过作用域分辨符来访问。

上面对于多继承的讨论都是假设多个基类之间没有继承关系也没有共同基类的情况，而**如果派生类的全部或者部分基类有共同的基类，也就是说派生类的这些基类是从同一个基**

类派生出的，那么派生类的这些直接基类从上一级基类继承的成员都具有相同的名称，即都是同名成员，要访问它们就必须通过直接基类限定，使用作用域分辨符访问。

上面说的可能有些抽象，鸡啄米再给出个程序例子来说明这种情况吧。我们先声明一个基类 Base0，Base0 中有数据成员 x 和函数成员 show，再声明类 Base1 和 Base2，它们都由 Base0 公有继承而来，最后从 Base1 和 Base2 共同派生出类 Child。这时 Base0 的成员经过到 Base1 和 Base2 再到 Child 的两次派生过程，出现在 Child 类中时，实际上 Base0 的数据成员 x 已经是两个不同的成员，只是名称相同但是在内存中是两份，函数成员 show 也是两个不同的成员，只是名称相同但是函数体可能不同。这就需要使用作用域分辨符访问了，但是不能用基类 Base0 来限定，因为这样还是不能说明成员是从 Base1 还是 Base2 继承而来，所以必须使用直接基类 Base1 或者 Base2 来限定，达到唯一标识成员的目的。程序例子如下：

```
#include <iostream>
using namespace std;
class Base0                    // 基类 Base0 的声明
{
public:
        int x;
        void show()     { cout<<"x of Base0: "<<x<<endl; }
};
class Base1 : public Base0     // 由 Base0 派生的类 Base1 的声明
{
};
class Base2 : public Base0     // 由 Base0 派生的类 Base2 的声明
{
};
class Child : public Base1, public Base2
{
};
int main()
{
      Child child;
      child.Base1::x = 3;      // 通过直接基类 Base1 限定成员
      child.Base1::show();
      child.Base2::x = 5;      // 通过直接基类 Base2 限定成员
      child.Base2::show();
      return 0;
}
```

程序运行结果：

x of Base0: 3
x of Base0: 5

上面的主函数 main 中定义了派生类 Child 的对象 child，如果只通过成员名访问成员 x 和 show，系统就不能确定访问哪个 x 和哪个 show，这就需要使用直接基类 Base1 或者 Base2 和作用域分辨符来访问它们。上面鸡啄米说了，数据成员 x 在内存中有两份拷贝，可以存放不同的数值，但是一般我们只需要一个这样的拷贝，那多出来的那个就是对内存的浪费。解决这个问题就需要后面讲的虚基类技术。

作用域分辨符的使用方法和作用就讲这么多了，鸡啄米觉得这些在编程入门的时候还是有些难度的，因为有点抽象，大家慢慢消化吧。有问题还是欢迎到鸡啄米博客讨论。

鸡啄米：C++编程入门系列之四十二（继承与派生：虚基类及其派生类的构造函数）

鸡啄米在编程入门系列的上一讲中讲了作用域分辨符。今天主要讲解虚基类及其派生类的构造函数。

1.虚基类的概念及用法

上一讲中说过，如果派生类的全部或者部分基类有共同的基类，那么派生类的这些直接基类从上一级基类继承的成员都具有相同的名称，定义了派生类的对象后，同名数据成员就会在内存中有多份拷贝，同名函数也会有多个映射。访问这些同名成员时，为了唯一标识它们可以使用上一讲中的作用域分辨符，也可以使用虚基类技术。**我们将派生类直接基类的共同基类声明为虚基类后，派生类从不同的直接基类继承来的同名数据成员在内存中就会只有一份拷贝，同名函数也会只有一个映射，这样不仅实现了唯一标识同名成员，而且也节省了内存空间，**可见虚基类技术是很实用的。

在派生类声明时除继承方式外还使用关键字 virtual 限定基类的话，此基类就是虚基类。虚基类声明的语法形式为：

class 派生类名:virtual 继承方式 基类名

这里关键字 virtual 跟继承方式一样，只限定紧跟在它后面的基类。比如，声明了类 A 为虚基类，类 B 为 A 的派生类，类 C 也是 A 的派生类，类 D 是由类 B 和 C 共同继承而来，则类 B 和类 C 从 A 继承的同名数据成员在类 D 的对象中只有一份拷贝，同名函数成员也只有一个函数体。

鸡啄米讲上一讲中的第二个例子做下修改，将 Base0 声明为虚基类来说明下虚基类的用法：我们先声明一个基类 Base0，Base0 中有数据成员 x 和函数成员 show，再声明类 Base1 和 Base2，它们都由 Base0 公有继承而来，与上一讲中不同的是派生时声明 Base0 为虚基类，最后从 Base1 和 Base2 共同派生出类 Child。这时 Base0 的成员经过到 Base1

和 Base2 再到 Child 的两次派生过程，出现在 Child 类中时，数据成员 x 在内存中也只有一份拷贝，函数成员 show 也只有一个映射。

```
#include <iostream>
using namespace std;
class Base0                    // 基类 Base0 的声明
{
public:
        int x;
        void show()     { cout<<"x of Base0: "<<x<<endl; }
};
class Base1 : virtual public Base0     // Base0 为虚基类，公有派生 Base1 类
{
};
class Base2 : virtual public Base0     // Base0 为虚基类，公有派生 Base2 类
{
};
class Child : public Base1, public Base2
{
};
int main()
{
      Child child;
      child.x = 5;
      child.show();
      return 0;
}
```

程序运行结果为：

```
x of Base0: 5
```

大家可以看到，声明虚基类只需要在它的派生类声明时使用关键字 virtual 修饰。

我们对作用域分辨符和虚基类技术进行对比分析可知，使用作用域分辨符唯一标识同名成员时，派生类中有同名成员的多个拷贝，可以存放不同的数据，进行不同的操作，而使用虚基类时派生类的同名成员只有一份拷贝，更节省内存。我们在软件开发中可以根据实际情况自己做出选择。

2.虚基类派生类的构造函数

上面例子中各个类都没有定义构造函数，而是使用的默认构造函数。如果虚基类定义了带参数表的非默认构造函数，没有定义默认形式的构造函数，那么情况会有些复杂。因为由虚基类直接或间接继承的所有派生类，都必须在构造函数的成员初始化列表中给出对虚基类成员的初始化。鸡啄米这里再讲上面的例子做进一步修改，为虚基类添加带参数表的构造函数，那么整个程序就要改成以下形式：

```
#include <iostream>
using namespace std;
class Base0                    // 基类 Base0 的声明
{
public:
        Base0(int y)     { x=y; }
        int x;
        void show()      { cout<<"x of Base0: "<<x<<endl; }
};
class Base1 : virtual public Base0     // Base0 为虚基类，公有派生 Base1 类
{
public:
        Base1(int y):Base0(y)    { }
};
class Base2 : virtual public Base0     // Base0 为虚基类，公有派生 Base2 类
{
public:
        Base2(int y):Base0(y)    { }
};
class Child : public Base1, public Base2
{
public:
        Child(int y):Base0(y),Base1(y),Base2(y)   { }
};
int main()
{
      Child child(3);
      child.show();
      return 0;
}
```

程序运行结果为：

```
x of Base0: 3
```

主函数中定义了派生类 Child 的对象 child，在构造对象 child 时调用了 child 的构造函数，其初始化列表中不只调用了虚基类 Base0 的构造函数对从它继承的成员 x 进行初始化，而且还调用了基类 Base1 和 Base2 的构造函数 Base1()和 Base2()，而 Base1()和 Base2()的初始化列表中又有对虚基类 Base0 成员 x 的初始化。这么说，从虚基类 Base0 继承来的成员 x 初始化了三次，其实不然，因为编译器在遇到这种情况时会进行特殊处理：**如果构造的对象中有从虚基类继承来的成员，那么虚基类成员的初始化由而且只由最远派生类的构造函数调用虚基类的构造函数来完成。**最远派生类就是声明对象时指定的类，上面例子中构造对象 child 时，类 Child 就是最远派生类。**除了最远派生类，它的其他基类对虚基类构造函数的调用会被忽略。**上例中就只会由 Child 类的构造函数调用虚基类 Base0 的构造函数完成成员 x 的初始化，而 Child 类的基类 Base1 和 Base2 对虚基类 Base0 构造函数的调用会被忽略。

鸡啄米对虚基类及其构造函数的内容就讲到这里了。有任何编程入门问题欢迎到鸡啄米博客留言讨论。

鸡啄米：C++编程入门系列之四十三（继承与派生：赋值兼容规则）

上一节鸡啄米讲了虚基类及其派生类的构造函数，本节来讲讲赋值兼容规则。

前面说过，派生类如果是从基类公有继承的，则它会包含基类中除构造函数和析构函数外的所有成员，基类的公有成员也成为派生类的公有成员，又因为对象只能访问类的公有成员，所以基类对象具有的功能，派生类对象都有。这样就引出了赋值兼容规则。

赋值兼容规则就是指在基类对象可以使用的地方都可以用公有派生类对象来代替。注意必须是公有派生类。赋值兼容规则中的代替有三种方式。鸡啄米通过一个例子分别说明。

假设有基类 Base，类 Child 是 Base 的公有派生类，base 为 Base 类的对象，pBase 为 Base 类指针，child 为 Child 类的对象。代码如下：

```
class Base
{
    ...
};
class Child : public Base
{
    ...
};
Base base, *pBase;
Child child;
```

那么根据赋值兼容规则，可以使用类 Base 对象的地方都可以使用类 Child 的对象来代替。这里的代替有三种：

1.派生类的对象可以赋值给基类的对象。也就是将派生类对象从基类继承的成员的值分别赋值给基类对象相应的成员。例如：

```
base = child;
```

2.派生类对象的地址可以赋值给基类类型的指针。例如：

```
pBase = &child;
```

3.派生类对象可以用来初始化基类的引用。例如：

```
Base &b = child;
```

因为有了赋值兼容规则，有了上述三种赋值方式，所以函数的参数中有基类对象或者基类指针又或者基类引用时，我们可以直接传入派生类对象或者派生类对象的地址作为实参来执行相同的操作。这样的好处是什么呢？那就是我们想对基类及派生类的对象做相同的操作时，只要定义一个函数就行了，它的参数为基类对象或者基类指针也或者是基类引用。这样就大大提高了软件开发的效率。

公有派生类对象可以代替基类对象使用，但是我们只能使用它从基类继承的成员，而无法使用它的新添成员。

鸡啄米给大家举个例子说明下赋值兼容规则：

类 Base 为基类，类 Child0 为 Base 的公有派生类，类 Child1 为类 Child0 的公有派生类。三个类中都定义了成员函数 show()。

```
#include <iostream>
using namespace std;
class Base          // 基类 Base 的声明
{
public:
        void show()   { cout << "Base::show()" << endl; }     // 公有成员函数 sho
w
};
class Child0 : public Base     // 类 Base 的公有派生类 Child0 的声明
{
public:
        void show()   { cout << "Child0::show()" << endl; }    // 公有成员函数 sho
w
};
class Child1 : public Child0   // 类 Child0 的公有派生类 Child1 的声明
{
public:
```

```
        void show()   { cout << "Child1::show()" << endl; }   // 公有成员函数 show
    };
    void CallShow(Base *pBase)    // 一般函数，参数为基类指针
    {
        pBase->show();
    }
    int main()
    {
        Base base;              // 声明 Base 类的对象
        Base *pBase;            // 声明 Base 类的指针
        Child0 ch0;             // 声明 Child0 类的对象
        Child1 ch1;             // 声明 Child1 类的对象
        pBase = &base;          // 将 Base 类对象 base 的地址赋值给 Base 类指针 pBase
        CallShow(pBase);
        pBase = &ch0;           // 将 Child0 类对象 ch0 的地址赋值给 Base 类指针 pBase
        CallShow(pBase);
        pBase = &ch1;           // 将 Child1 类对象 ch1 的地址赋值给 Base 类指针 pBase
        CallShow(pBase);
        return 0;
    }
```

程序运行结果为：

```
Base::show()
Base::show()
Base::show()
```

我们首先定义了一个函数 CallShow，其参数 pBase 为基类 Base 类型的指针，根据赋值兼容规则，我们可以用公有派生类对象的地址为基类指针赋值，那么 CallShow 函数就可以处理这个类族的所有对象。在主函数中我们就分别把基类对象 base 的地址、派生类对象 ch0 的地址和派生类对象 ch1 的地址赋值给基类指针 pBase，然后将 pBase 作为实参调用 CallShow，在 CallShow 中调用了成员函数 show。

但是，根据上面所讲，将派生类对象的地址赋值给 pBase 以后，通过 pBase 只能访问派生类从基类继承的成员。所以即使指针 pBase 指向的是派生类对象 ch0 或者 ch1，在 CallShow 中通过 pBase 也只能调用从基类 Base 继承的成员函数 show，而不会调用 Child0 类或者 Child1 类的成员函数 show。因此主函数中三次调用 CallShow 函数，都是访问的基类 Base 的成员函数 show，输出都是 Base::show()。

这时我们深切的感受到，即使派生类对象代替了基类对象，它也只能产生基类的功能，自己的新功能无法体现。要想在代替以后同样能够实现自己的功能，就要用到面向对象设计的另一个特性--多态性。而本节学的赋值兼容规则是多态性的基础。

鸡啄米最后提醒大家，只有在编程入门的时候把赋值兼容规则搞明白，才能更好的学习多态性。今天就讲到这了。有问题欢迎到鸡啄米博客交流。

第八部分：多态性

鸡啄米：C++编程入门系列之四十四（多态性：多态的概念和类型）

上一节讲完赋值兼容规则后，编程入门中继承与派生的部分就讲完了。今天开始讲解面向对象设计中的多态性。

鸡啄米先介绍一个概念：消息。消息在C++编程中指的是对类的成员函数的调用。**多态就是指相同的消息被不同类型的对象接收会引起不同的操作**，直接点讲，就是在不同的情况下调用同名函数时，可能实际调用的并不是同一个函数。

以“+”运算符为例，“+”可以实现整型变量之间、浮点型变量之间的加法运算，也可以实现不同类型变量之间的加法运算，例如整型变量和浮点型变量相加，这时需要先将整型变量转换为浮点变量再进行加法运算。同样是加法运算，参与运算的变量类型不同时，进行加法运算的方式也不同。这就是多态的典型例子。

1.多态的类型

多态性有四种类型：重载多态、强制多态、参数多态和包含多态。

前两种可以统称为专用多态。之前鸡啄米讲过普通函数的重载和类的成员函数的重载，它们都属于重载多态。上面说的整型变量和浮点型变量相加时，需要先把整型变量强制转换为浮点型再进行加法运算，这就是强制多态。从概念上讲，强制多态就是将一个变量的类型进行转换，以满足一个函数运算的要求。

后两种统称为通用多态。前面讲过的类模板将类型参数化，设定了确定的类型才可以实例化。由类模板实例化得到的所有类都有相同的操作，但是被操作对象的类型不同，这就是参数多态。包含多态是指类族中不同类的同名成员函数实现的操作不相同。包含多态一般通过虚函数来实现。虚函数在我们进行软件开发设计时会经常用到，初学者在有了一定经验后会越来越多的使用虚函数。

鸡啄米后面会主要讲重载多态的运算符重载和包含多态的虚函数。

2.多态的实现

从多态实现的阶段不同来分类可以分为编译时的多态和运行时的多态。

编译时的多态是指在编译的过程中就确定了具体调用同名函数中的哪个函数，而运行时的多态则是在程序运行过程中才动态的确定调用的具体函数。这种确定调用同名函数的哪个

函数的过程就叫做联编或者绑定。鸡啄米一般称其为绑定。绑定实际上就是确定某个标识符对应的存储地址的过程。**按照绑定发生的阶段的不同可以分为：静态绑定和动态绑定。**静态绑定就对应着编译时的多态，动态绑定对应运行时的多态。

如果绑定过程发生在编译链接阶段，则称为静态绑定。在编译链接过程中，编译器根据类型匹配等特征确定某个同名标识究竟调用哪一段程序代码，也就是确定通过某个同名函数到底调用哪个函数体。1 中的四种多态中有三种需要静态绑定：重载多态、强制多态和参数多态。

而**如果绑定过程发生在程序运行阶段，则成为动态绑定。**在编译链接过程中无法确定调用的具体函数，就要等到程序运行时动态确定。包含多态就需要使用动态绑定实现。

因为重载、强制和参数多态有关的内容鸡啄米前面都讲过，可能大家对它们的理解还不算困难。但是包含多态可能很难理解，但是没关系，后面鸡啄米会细讲包含多态的虚函数，大家会明白包含多态的强大。最后仍然是欢迎大家到鸡啄米博客交流。

鸡啄米：C++编程入门系列之四十五（多态性：运算符重载的概念和规则）

上一节鸡啄米讲了多态性的概念和类型，多态有四种类型：重载多态、强制多态、参数多态和包含多态。这一节主要介绍重载多态中的运算符重载。

一.运算符重载的概念

为什么我们需要运算符重载？因为自定义数据类型有时也需要使用运算符进行某些运算，比如加法运算，但是预定义的运算符的操作数只能是基本数据类型，所以自定义数据类型的运算需要进行运算符重载。例如，有日期类 Date 声明如下：

```
class Date
{
public:
        Date(int nYear, int nMonth, int nDay) { m_nYear=nYear; m_nMonth=nMonth; m_nDay=nDay; } // 构造函数
        void show();      // 显示日期
private:
        int m_nYear;
        int m_nMonth;
        int m_nDay;
};
```

假设我们声明了两个 Date 类的对象：Date date1(2011, 11, 1), date2(2012, 1, 6);。然后需要计算 date1 和 date2 所表示日期差多少天，也就是进行减法运算，最简单的就是用运算符“-”，但是如果直接写 date2-date1，编译器会报错，因为编译器不知道怎样进行此减法运算。这就需要我们自己写程序来说明在对 Date 类对象进行“-”运算时，具体做哪些处理，也就是需要进行运算符重载。

总结一下，**运算符重载就是为预定义的一些运算符增加新的意义，使其因操作数类型的不同而产生不同的操作。**运算符重载实际上属于函数重载，因为在运算符重载中，不是运算符表达式而是调用运算符函数，操作数变成了运算符函数的参数，运算符函数的参数不同时调用不同的函数。这些与函数重载如出一辙。

二.运算符重载的规则

运算符重载的使用有如下规则：

1.运算符重载是为了让自定义数据类型能够使用预定义运算符，对预定义运算符进行重定义，但**一般重定义的功能与原运算符的功能相似，运算符重载的参数个数与原运算符的操作数个数相同，而且至少有一个参数属于自定义数据类型。**

2.运算符重载后其优先级和结合性都与原运算符相同。

3.除了类属关系运算符“.”、成员指针运算符“.*”、作用域分辨符“::”、sizeof 运算符和条件运算符“?:”这五种运算符外，其余 C++运算符都能重载，而且只有 C++中已有的运算符可以重载。

“.”和“.*”不能重载是为了保证其功能不被改变，sizeof 运算符和作用域分辨符的操作数不是一般的表达式，而是类型，所以也不能重载。

运算符重载后能作用于类的对象的话，最容易想到的重载形式是类的成员函数，其次就是类的友元函数。

运算符重载为类的成员函数时的声明形式为：

```
函数类型 operator 运算符(参数表)
{
    函数体;
}
```

函数类型是运算符重载的返回值类型。operator 是声明和定义运算符重载时的关键字。运算符就是需要重载的运算符，比如“+”或“-”，但不能是“.”、“.*”、“::”、sizeof 或“?:”。参数表列出重载运算符的参数及类型，这里当重载运算符不是后置“++”或“--”时，参数的个数比原运算符的操作数个数少一个，因为类的对象调用运算符重载成员函数时，自己的数据可以直接访问，不需要在参数表中传递，所以参数表中就不必列出该对象本身了。

运算符重载为类的友元函数时的声明形式为：

```
friend 函数类型 operator 运算符(参数表)
{
    函数体;
}
```

与上面运算符重载为类的成员函数时不同的是，在函数类型前需要加关键字 friend。另外，运算符重载友元函数访问类的对象的数据时，必须通过类的对象名访问，所以此友元函数的所有参数都需要进行传递，参数个数与原运算符的操作数个数相同。

大家以后在软件开发中用了运算符重载后，会体会到复杂类型数据也能进行加减运算的方便的。这让我们的程序书写更简单，可读性更高，更易维护，最终提高软件开发效率。

今天就先讲到这了，主要是让大家了解下运算符重载的概念，及运算符重载的规则。至于运算符如何重载为类的成员函数和类的友元函数，后面会细讲。欢迎大家在鸡啄米博客留言，相互学习。

鸡啄米：C++编程入门系列之四十六（多态性：运算符重载为类的成员函数）

上一节中鸡啄米讲到了运算符重载的概念和规则，运算符可以重载为类的成员函数或友元函数，这一节就来讲讲运算符怎样重载为类的成员函数。

运算符重载为类的成员函数后就可以像其他成员函数一样访问本类的数据成员了。在类的外部通过类的对象，可以像原运算符的使用方式那样使用重载的运算符，比如，“+”运算符被重载为类 A 的成员函数后，A 的对象 a 和其他对象 b 就可以这样进行加法运算：a+b。

重载的运算符可能是双目运算符也可能是单目运算符。

如果是双目运算符，比如“+”和“-”，则一个操作数是使用此运算符的对象本身，另一个操作数使用运算符重载函数传递进来的对象。假设有双目运算符 U，a 为类 A 的对象，另有某类也可以是 A 类的对象 b，我们想实现 a U b 这样的运算，就可以把 U 重载为类 A 的成员函数，此**函数只有一个形参，形参的类型为对象 b 的类型**。这样进行 a U b 的运算就相当于函数调用：a.operator U(b)。

如果是单目运算符，比如“++”和“--”，操作数就是此对象本身，**重载函数不需要传递参数**，只是后置单目运算符语法上规定有一个形式上的参数，以区别于前置单目运算符。

假设有前置单目运算符 U，如前置“++”，a 为类 A 的对象，我们想实现 U a 这样的运算，也可以把 U 重载为类 A 的成员函数，此函数没有形参。这样 U a 表达式就相当于函数调用：a.operator U()。

假设有后置单目运算符 U，如后置“--”，a 为类 A 的对象，我们想实现 a U 这样的运算，同样可以把 U 重载为类 A 的成员函数，但此函数需要有一个整型的形参。重载后 a U 表达式就相当于函数调用：a.operator U(0)。

这里鸡啄米需要强调下，**前置单目运算符重载和后置单目运算符重载在语法形式上的区别就是前者重载函数没有形参，而后者重载函数有一个整型形参，此形参对函数体没有任何影响，这只是语法上的规定，仅仅是为了区分前置和后置**。

鸡啄米给大家两个程序例子，分别演示双目运算符和单目运算符的使用。

第一个例子：时间值的加法，比如 2 个小时 20 分钟加 3 个小时 30 分钟，应该是 5 个小时 50 分钟，运算规则就是小时数相加，分钟数相加，如果分钟数的和超过 60 分钟则小时数再加 1，分钟数减 60。双目运算符“+”需要重载为时间值类的成员函数，此函数只有一个形参，类型也是时间值类的对象。

```
#include <iostream>
using namespace std;
class CTimeSpan
{
public:
        CTimeSpan(int nHours=0, int nMins=0);      // 构造函数
        CTimeSpan operator +(CTimeSpan ts);        // 运算符“+”重载为成员函数
        int GetHours()      { return m_nHours; }   // 获取小时数
        int GetMins()       { return m_nMins; }    // 获取分钟数
        void Show();                               // 显示时间值
private:
        int m_nHours;       // 小时数
        int m_nMins;        // 分钟数
};
CTimeSpan::CTimeSpan(int nHours, int nMins)        // 构造函数的实现
{
       nHours += nMins/60;
        nMins %= 60;
        m_nHours = nHours;
        m_nMins = nMins;
}
CTimeSpan CTimeSpan::operator +(CTimeSpan ts)    // 重载运算符函数实现
{
        int nNewHours;
        int nNewMins;
        nNewHours = m_nHours + ts.GetHours();
        nNewMins = m_nMins + ts.GetMins();
        nNewHours += nNewMins/60;
        nNewMins %= 60;
        return CTimeSpan(nNewHours, nNewMins);
}
void CTimeSpan::Show()
{
```

```
        cout << m_nHours << "小时" << m_nMins << "分钟" << endl;
}
int main()
{
        CTimeSpan timeSpan1(2, 50);
        CTimeSpan timeSpan2(3, 30);
        CTimeSpan timeSum;
        timeSum = timeSpan1 + timeSpan2;
        cout << "timeSpan1: ";
        timeSpan1.Show();
        cout << "timeSpan2: ";
        timeSpan2.Show();
        timeSum = timeSpan1 + timeSpan2;
        cout << "timeSum=timeSpan1+timeSpan2: ";
        timeSum.Show();
        return 0;
}
```

程序运行结果：

```
timeSpan1: 2 小时 50 分钟
timeSpan2: 3 小时 30 分钟
timeSum=timeSpan1+timeSpan2: 6 小时 20 分钟
```

我们可以看出，运算符重载成员函数跟一般的成员函数类似，只是使用了关键字 operator。使用重载运算符的方式与原运算符相同。运算符作用于整型、浮点型和 CTimeSpan 等不同的对象会发生不同的操作行为，这就是多态性。

注意，重载“+”的函数中，语句 return CTimeSpan(nNewHours, nNewMins);看似是对 CTimeSpan 构造函数的调用，实则不然，这是构造一个临时对象并将它返回到主函数中。

第二个例子：时钟类的例子。前置“++”和后置“++”重载为时钟类的成员函数。前置“++”重载函数没有形参，后置“++”重载函数有一个整型形参。

```
#include<iostream>
using namespace std;
class Clock //时钟类声明
{
public: //外部接口
        Clock(int NewH=0, int NewM=0, int NewS=0);
        void ShowTime();
        Clock& operator ++(); //前置单目运算符重载
        Clock operator ++(int); //后置单目运算符重载
```

```
 private: //私有数据成员
        int Hour,Minute,Second;
 };
 Clock::Clock(int NewH, int NewM, int NewS)
{
        if (0<=NewH && NewH<24 && 0<=NewM && NewM<60 && 0<= New
S && NewS<60)
        {
                Hour = NewH;
                Minute = NewM;
                Second = NewS;
        }
        else
                cout << "错误的时间！" << endl;
}
 void Clock::ShowTime()
 {
        cout << Hour << ":" << Minute << ":" << Second << endl;
 }
 Clock& Clock::operator ++() //前置单目运算符重载函数
{
        Second++;
        if(Second>=60)
        {
                Second=Second-60;
                Minute++;
                if(Minute>=60)
                {
                        Minute=Minute-60;
                        Hour++;
                        Hour=Hour%24;
                }
        }
        return *this;
 }
 //后置单目运算符重载
 Clock Clock::operator ++(int)      //注意形参表中的整型参数
 {
        Clock old=*this;
        ++(*this);
```

```
        return old;
}
int main()
{
        Clock myClock(23,59,59);
        cout<<"初始时间 myClock:";
        myClock.ShowTime();
        cout<<"myClock++:";
        (myClock++).ShowTime();
        cout<<"++myClock:";
        (++myClock).ShowTime();
        return 0;
}
```

程序运行结果：

```
初始时间 myClock:23:59:59
myClock++:23:59:59
++myClock:0:0:1
```

因为后置单目运算符重载函数中的整型形参没有实际意义，只是为了区分前置和后置，所以参数表中只给出类型就行了，参数名写不写都可以。

运算符重载部分需要大家认真理解，可以在 VS2010 上运行上面两个例子，然后自己举一反三，写几个简单例子试验下其他运算符。

鸡啄米就讲到这了，有问题欢迎在鸡啄米博客交流。谢谢大家。

鸡啄米：C++编程入门系列之四十七（多态性：运算符重载为类的友元函数）

鸡啄米在上一节中讲了运算符重载为类的成员函数的方式和规则，这一节接着讲运算符重载的另一种方式--运算符重载为类的友元函数。

在编程入门系列之二十二--友元中，鸡啄米讲到过，友元函数通过类的对象可以访问类的公有、保护和私有成员，也就是类的所有成员友元函数都能访问到。所以运算符重载为类的友元函数以后也可以访问类的所有成员。

与运算符重载为成员函数时不同的是，重载的友元函数不属于任何类，运算符的操作数都需要通过函数的形参表传递。操作数在形参表中从左到右出现的顺序就是用运算符写表达式时操作数的顺序。

这里也分双目运算符和单目运算符两种情况讨论运算符重载为友元函数的具体方式。

如果有双目运算符 U，它的其中一个操作数是类 A 的对象 a，那么运算符 U 就可以重载为类 A 的友元函数，此友元函数的两个参数中，一个是类 A 的对象，另一个是其他对象，也可以是类 A 的对象。这样双目运算符重载为类的友元函数后，假设运算符的两一个操作数是对象 b，则表达式 a U b 就相当于调用函数 operator U(a, b)。

下面再讨论单目运算符的重载。**如果有前置单目运算符 U，比如前置“--”，a 为类 A 的对象，我们想实现 U a 这样的运算，就可以把 U 重载为类 A 的友元函数，此友元函数只有一个形参，为类 A 的对象，**重载后表达式 U a 相当于调用函数 operator U(a)。**如果是后置单目运算符 U，如后置“++”，a 还是类 A 的对象，那么要实现 a U 这样的运算，也可以把 U 重载为类 A 的友元函数，此时友元函数就需要有两个形参，一个是类 A 的对象，另一个是整型形参，**此整型形参没有实际意义，与上一节后置单目运算符重载为成员函数时的整型形参一样，只是为了区分前置运算符和后置运算符的重载。重载后表达式 a U 就相当于调用函数 operator U(a, 0)。

鸡啄米将上一节中第一个例子中的运算符重载改为友元函数，再简单介绍下要实现的功能：时间值的加法，比如 2 个小时 20 分钟加 3 个小时 30 分钟，应该是 5 个小时 50 分钟，运算规则就是小时数相加，分钟数相加，如果分钟数的和超过 60 分钟则小时数再加 1，分钟数减 60。双目运算符“+”需要重载为时间值类的友元函数，此函数有两个形参，类型都是时间值类的对象。

```
#include <iostream>
using namespace std;
class CTimeSpan
{
public:
        CTimeSpan(int nHours=0, int nMins=0);     // 构造函数
        friend CTimeSpan operator +(CTimeSpan ts1, CTimeSpan ts2); // 运算符“+”重载为成员函数
        int GetHours()     { return m_nHours; }   // 获取小时数
        int GetMins()      { return m_nMins; }    // 获取分钟数
        void Show();                              // 显示时间值
private:
        int m_nHours;      // 小时数
        int m_nMins;       // 分钟数
};
CTimeSpan::CTimeSpan(int nHours, int nMins)       // 构造函数的实现
{
        nHours += nMins/60;
        nMins %= 60;
        m_nHours = nHours;
        m_nMins = nMins;
}
```

```
void CTimeSpan::Show()
{
	cout << m_nHours << "小时" << m_nMins << "分钟" << endl;
}
CTimeSpan operator +(CTimeSpan ts1, CTimeSpan ts2) // 重载运算符函数实现
{
	int nNewHours;
	int nNewMins;
	nNewHours = ts1.m_nHours + ts2.m_nHours;
	nNewMins = ts1.m_nMins + ts2.m_nMins;
	nNewHours += nNewMins/60;
	nNewMins %= 60;
	return CTimeSpan(nNewHours, nNewMins);
}
int main()
{
	CTimeSpan timeSpan1(2, 50);
	CTimeSpan timeSpan2(3, 30);
	CTimeSpan timeSum;
	timeSum = timeSpan1 + timeSpan2;
	cout << "timeSpan1: ";
	timeSpan1.Show();
	cout << "timeSpan2: ";
	timeSpan2.Show();
	timeSum = timeSpan1 + timeSpan2;
	cout << "timeSum=timeSpan1+timeSpan2: ";
	timeSum.Show();
	return 0;
}
```

程序运行结果：

```
timeSpan1: 2 小时 50 分钟
timeSpan2: 3 小时 30 分钟
timeSum=timeSpan1+timeSpan2: 6 小时 20 分钟
```

这个程序的主函数 main 与上一节例子的 main 函数完全相同，程序运行结果也一样。区别就是加法运算符重载为 CTimeSpan 类的友元函数而不是成员函数，我们看到运算符重载函数有两个形参 ts1 和 ts2，通过这两个参数将需要进行运算的操作数传递进去，而在此函数中也能够访问类 CTimeSpan 的私有成员 m_nHours 和 m_nMins。

这两节中鸡啄米给出的例子仅介绍了几个简单运算符的重载例子，对于其他如“%”、“>>”等运算符的重载可能有些不同，但是只要大家真正理解了重载方法，相信也很容易掌握其他运算符的重载。

好了，运算符重载的内容就讲完了。如果有问题欢迎到鸡啄米博客交流讨论。

鸡啄米：C++编程入门系列之四十八（多态性：虚函数）

春节假期大家过的快乐吗？因为是假期鸡啄米就没有更新编程入门教程，估计这期间学习的人也很少吧。

节前最后一讲鸡啄米讲的是运算符重载为类的友员函数。今天讲的是属于包含多态的虚函数。

一.虚函数的意义

在讲赋值兼容规则时鸡啄米给出了一个程序例子，其中包含类 Base、Child0 和 Child1。在程序运行结果中我们看到，main 函数中 Base 类型的指针 pBase，分别指向 Base、Child0 和 Child1 类的对象时调用的 show 函数都是基类 Base 的 show 函数。因为基类类型的指针指向派生类对象时，通过此指针只能访问从基类继承来的同名成员。这些在 C++编程入门系列之四十三--赋值兼容规则中已经分析过了。

但是如果我们希望通过指向派生类对象的基类指针，访问派生类中的同名成员该怎么办呢？这就要用到虚函数了。我们在基类中将某个函数声明为虚函数，就可以通过指向派生类对象的基类指针访问派生类中的同名成员了。这样使用某基类指针指向不同派生类的不同对象时，就可以发生不同的行为，也就实现了运行时的多态（编译时并不知道调用的是哪个类的成员）。

虚函数是动态绑定的基础。记住，**虚函数是非静态的成员函数**，一定不能是静态（static）的成员函数。虚函数在以后我们进行软件架构设计时会起到很关键的作用。编程入门时可能不会有这方面的意识，等熟练到一定程度你就会发现虚函数的强大。

二.一般虚函数成员的声明和使用

一般的虚函数声明形式为：

```
virtual  函数类型  函数名(形参表)
{
        函数体
}
```

虚函数就是在类的声明中用关键字 virtual 限定的成员函数。以上声明形式是成员函数的实现也在类的声明中的情况。**如果成员函数的实现在类的声明外给出时，则虚函数的声明只能出现在类的成员函数声明中，而不能在成员函数实现时出现，**简而言之，只能在此成员函数的声明前加 virtual 修饰，而不能在它的实现前加。

总结下运行时多态的几个条件：1.类之间要满足赋值兼容规则；2.要声明虚函数；3.通过类的对象的指针、引用访问虚函数或者通过类的成员函数调用虚函数。下面举例说明下，大家通过这个例子来对照下这几个条件。

此例是由赋值兼容规则中的例子改进的。将基类中的函数 show 声明为虚函数，程序其他部分不做任何修改。

```
#include <iostream>
using namespace std;
class Base          // 基类 Base 的声明
{
public:
        virtual void show()   { cout << "Base::show()" << endl; }     // 虚成员函数 show
};
class Child0 : public Base     // 类 Base 的公有派生类 Child0 的声明
{
public:
        void show()   { cout << "Child0::show()" << endl; }   // 虚成员函数 show
};
class Child1 : public Child0   // 类 Child0 的公有派生类 Child1 的声明
{
public:
        void show()   { cout << "Child1::show()" << endl; }   // 虚成员函数 show
};
void CallShow(Base *pBase)     // 一般函数，参数为基类指针
{
       pBase->show();
}
int main()
{
      Base base;              // 声明 Base 类的对象
      Base *pBase;            // 声明 Base 类的指针
      Child0 ch0;             // 声明 Child0 类的对象
      Child1 ch1;             // 声明 Child1 类的对象
      pBase = &base;          // 将 Base 类对象 base 的地址赋值给 Base 类指针 pBase
      CallShow(pBase);
      pBase = &ch0;           // 将 Child0 类对象 ch0 的地址赋值给 Base 类指针 pBase
      CallShow(pBase);
```

```
    pBase = &ch1;          // 将 Child1 类对象 ch1 的地址赋值给 Base 类指针 pBas
e
    CallShow(pBase);
    return 0;
}
```

程序运行结果：

```
Base::show()
 Child0::show()
Child1::show()
```

我们可以看出，仅仅是在 Base 类中的 show 函数前加了 virtual 的修饰，运行结果就差了很多，这正是虚函数的美丽所在。

例程中，类 Base、Child0 和 Child1 属于同一个类族，而且 Child0 是由 Base 公有派生的，Child1 是从 Child0 公有派生的，所以满足赋值兼容规则，这就符合了运行时多态的第一个条件。基类 Base 的函数 show 声明为了虚函数，这是第二个条件。在 CallShow 函数中通过对象指针 pBase 来访问虚函数 show，这又满足了第三个条件。这个动态绑定过程在运行时完成，实现了运行时的多态。这样通过基类指针就可以访问指向的不同派生类的对象的成员，这在软件开发中不仅使代码整齐简洁，而且也大大提高了开发效率。

基类的成员函数声明为虚函数以后，派生类中的同名函数可以加 virtual 修饰也可以不加。

三.虚析构函数

大家可能奇怪为什么不先讲虚构造函数，很简单，因为**不能声明虚构造函数，而可以声明虚析构函数。**

多态是指不同的对象接收了同样的消息而导致完全不同的行为，它是针对对象而言的，虚函数是运行时多态的基础，当然也是针对对象的，而构造函数是在对象生成之前调用的，即运行构造函数时还不存在对象，那么虚构造函数也就没有意义了。

析构函数用于在类的对象消亡时做一些清理工作，我们在基类中将析构函数声明为虚函数后，其所有派生类的析构函数也都是虚函数，使用指针引用时可以动态绑定，实现运行时多态，通过基类类型的指针就可以调用派生类的析构函数对派生类的对象做清理工作。

前面讲过，析构函数没有返回值类型，没有参数表，所以虚析构函数的声明也比较简单，形式如下：

```
virtual ~类名();
```

虚函数的内容就先讲到这里了，抽象度比较高。建议大家经常回来复习，以便更熟练的掌握，并服务于以后的程序架构的设计。谢谢大家，有问题还是欢迎到鸡啄米博客来交流。

鸡啄米：C++编程入门系列之四十九（多态性：纯虚函数和抽象类）

上一讲中鸡啄米讲了多态性中的重要概念，虚函数。本节主要讲解另一个软件开发中经常用到的多态概念--抽象类。

抽象类可以为某个类族提供统一的操作接口。外部可以透明的使用抽象类的统一接口，而不需要知道到底是调用的抽象类的哪个派生类的成员函数。其实这些也可以通过在基类中定义虚函数来实现，但抽象类跟一般类不同的是，它使用纯虚函数，不需要定义纯虚函数的实现，而且抽象类不能实例化，即不能定义抽象类的对象，只能从它继承出非抽象派生类再实例化。

抽象类的定义就是，含有纯虚函数的类。纯虚函数跟一般的虚函数有很大不同。我们下面来了解下纯虚函数。

一.纯虚函数

即使有的虚函数在基类中不需要做任何工作，我们也要写出一个空的函数体，这时这个函数体没有什么意义，重要的是此虚函数的原型声明。C++为我们提供了纯虚函数，让我们在这种情况下不用写函数实现，只给出函数原型作为整个类族的统一接口就可以了，函数的实现可以在派生类中给出。

纯虚函数是在基类中声明的，声明形式为：

```
virtual 函数类型 函数名(参数表) = 0;
```

大家可以看到，纯虚函数的声明形式与一般虚函数类似，只是最后加了个“=0”。纯虚函数这样声明以后，在基类中就不再给出它的实现了，各个派生类可以根据自己的功能需要定义其实现。

二.抽象类

上面说到，抽象类就是含有纯虚函数的类。**抽象类可以为某个类族定义统一的接口，接口的具体实现是在派生类中给出。**这种实现就具有多态特性。

这里要注意的是，**抽象类的派生类如果没有实现所有的纯虚函数，只给出了部分纯虚函数的实现，那么这个派生类仍然是抽象类，仍然不能实例化，只有给出了全部纯虚函数的实现，派生类才不再是抽象类并且才可以实例化。**

我们不能声明抽象类的对象，使用抽象类一般是通过声明抽象类的指针或引用，将指针或引用指向派生类的对象，访问派生类的成员。

鸡啄米在这里把上一讲中的例程进一步修改，将基类 Base 中的 show 函数声明为纯虚函数，这时 Base 类就是一个抽象类，我们不能声明 Base 类的对象了，但是可以声明 Bas

e 类的指针，指向派生类 Child0 和 Child1 的对象，通过此指针访问派生类的虚函数。程序如下：

```
#include <iostream>
using namespace std;
class Base          // 抽象类 Base 的声明
{
public:
    virtual void show() = 0;      // 纯虚函数成员 show
};
class Child0 : public Base     // 类 Base 的公有派生类 Child0 的声明
{
public:
    void show()   { cout << "Child0::show()" << endl; }   // 虚成员函数 show
};
class Child1 : public Child0   // 类 Child0 的公有派生类 Child1 的声明
{
public:
    void show()   { cout << "Child1::show()" << endl; }   // 虚成员函数 show
};
void CallShow(Base *pBase)     // 一般函数，参数为基类指针
{
    pBase->show();
}
int main()
{
    Base *pBase;            // 声明 Base 类的指针
    Child0 ch0;             // 声明 Child0 类的对象
    Child1 ch1;             // 声明 Child1 类的对象
    pBase = &ch0;           // 将 Child0 类对象 ch0 的地址赋值给 Base 类指针 pBase
    CallShow(pBase);
    pBase = &ch1;           // 将 Child1 类对象 ch1 的地址赋值给 Base 类指针 pBase
    CallShow(pBase);
    return 0;
}
```

程序运行结果为：

Child0::show()
Child1::show()

这里派生类 Child0 和 Child1 的虚函数 show 并没有使用关键字 virtual 显式说明，因为 Child0 和 Child1 中的虚函数和基类 Base 中的纯虚函数名称一样，参数和返回值都相同，编译器会自动识别其为虚函数。

上面的程序中，基类 Base 是抽象类，为整个类族提供了统一的外部接口。派生类 Child0 中给出了全部纯虚函数的实现（其实只有一个纯虚函数--show），因此不再是抽象类，可以声明它的对象。Child0 的派生类 Child1 当然也不是抽象类。根据赋值兼容规则，基类 Base 的指针可以指向派生类 Child0 和 Child1 的对象，通过此指针可以访问派生类的成员，这样就实现了多态。

抽象类确实是比较抽象，作为编程入门学习者可能还不能灵活运用。但当有了些经验以后，你就会意识到抽象类的重要性，它几乎是程序设计中必不可少的。鸡啄米相信大家只要认真的学，多写写程序，很快就会用的得心应手的。最后依然欢迎大家有问题在鸡啄米博客留言。谢谢大家。

第九部分：异常处理

鸡啄米：C++编程入门系列之五十（异常处理）

关于面向对象设计的重要特性--多态性到上一节纯虚函数和抽象类就讲完了。这一讲是本 C++编程入门教程的最后一讲--异常处理。

我们开发的软件一般按照正常的流程操作时运行不会出问题，但是用户不一定会根据软件工程师的想法来操作软件，而且往往随机性很大，另外，软件的运行环境也会改变，例如硬盘空间不足、文件被移走，这些都可能会导致软件出现异常，甚至崩溃。所以我们进行软件开发时要充分考虑异常的捕捉和处理。

一.异常处理的基本思想

进行异常处理的目标是，使软件具有容错能力，在出现运行环境或者异常操作等问题时，程序能够继续往下运行，必要时弹出提示信息。

软件开发中往往每个函数都有自己的分工，很多出现错误的函数都不会处理错误，而是产生一个异常让调用者捕捉和处理。如果调用者也不能处理此异常，则异常就会被继续向上级调用者传递，这个传递过程会一直持续到异常能被处理为止。如果程序最终没能处理这个异常，那么它就会被传递给 C++的库函数 terminate，然后 terminate 会调用 abort 函数终止程序。

二.C++异常处理的语法

异常处理机制是靠 try、throw 和 catch 语句实现的。

throw 语句的形式为：

throw 表达式

try 块的语法形式为：

```
try
{
        复合语句
}
catch(异常类型的声明)
{
        复合语句
}
catch(异常类型的声明)
{
        复合语句
}
...
```

先说说 throw 语句，当某段程序有了不能处理的异常时，就可以用“throw 表达式”的方式传递这个异常给调用者。这里 throw 后的表达式在语法上与 return 后的表达式类似。

再来看 try 块的 try 子句，子句后括号里的复合语句就是被监测的程序段。如果某段程序或者调用的某个函数可能会产生异常，就把它放到 try 后。当 try 子句后的程序段发生异常时，程序段中 throw 就会抛出这个异常。

最后来看 try 的 catch 子句，catch 子句后括号里的异常类型的声明，在语法上与函数的形参类似，可以是某个类型（包括类）的值也可以是引用，它指明了此 catch 子句用来处理何种类型的异常。当 try 子句中的异常抛出后，每个 catch 子句会被依次检查，哪个 catch 子句的异常类型的声明与抛出异常的类型一致，就由哪个 catch 子句来处理此异常。**catch 后异常类型的声明部分可以是一个省略号，形式如：catch(...)，这种形式的 catch 子句可以处理任何类型的异常，它只能放到 try 块所有其他 catch 语句之后。**

可见，**如果 try 监测的某段程序多个地方需要抛出异常，那么 throw 后应该跟不同类型的表达式来区分，而不应该只通过不同的值区分。**

讲了异常处理的语法形式，据此再来说说异常处理的执行过程：

1.程序正常执行到 try 块的 try 子句，然后执行 try 子句后的复合语句，也就是被监测的程序段。

2.如果 try 子句后的程序段正常执行了，没有发生任何异常，那么此 try 块的所有 catch 子句将不被执行，程序直接跳转到整个 try 块（包括 try 子句和所有 catch 子句）后继续执行。

3.如果 try 子句后的程序段或者此程序段中的任何调用函数发生了异常，并通过 throw 抛出了这个异常，则此 try 块的所有 catch 子句会按其出现的顺序被检查。若没有找到匹配的处理程序则继续检查外层的 try 块。如果一直找不到则此过程会继续到最外层的 try 块被检查。这里有两种结果：a.找到了匹配的处理程序，则相应 catch 子句捕捉异常，将异常对象拷贝给 catch 的参数，如果此参数是引用则它指向异常对象。catch 的参数被这样初始化以后，此 catch 子句对应的 try 子句后的程序段中，从开头到异常抛出位置之间构造的所有对象进行析构，析构顺序与构造顺序相反。然后 catch 处理程序被执行，最后程序跳转到 try 块之后的语句执行。b.始终没有找到匹配的处理程序，则运行 C++库函数 terminate，而 terminate 函数调用 abort 函数终止程序。

鸡啄米给大家举个异常处理的例子，大家知道我们只能对非负实数求平方根，若是负数就应该处理此异常。例程如下：

```
#include <iostream>
#include "math.h"
using namespace std;
double GetSqrt(double x);       // 求平方根的函数的原型声明
int main()
{
    try
    {
        // 由于求平方根运算有可能出现对负数运算的异常，所以放到 try 块中
        cout << "9.0 的平方根是 " << GetSqrt(9.0) << endl;
        cout << "-1.0 的平方根是 " << GetSqrt(-1.0) << endl;
        cout << "16.0 的平方根是 " << GetSqrt(16.0) << endl;
    }
    catch(double y)    // 捕捉 double 型异常
    {
        cout << "发生对负数" << y << "求平方根的异常。" << endl;
    }
    cout << "程序继续运行完毕。" << endl;
    return 0;
}
double GetSqrt(double x)
{
    if (x < 0)
        throw x;    // 如果 x 为负数，则抛出一个 double 型异常
```

```
    return sqrt(x);
}
```

程序运行结果为：

```
9.0 的平方根是 3
发生对负数-1 求平方根的异常。
程序继续运行完毕。
```

根据结果可以看出，程序在运行 cout << "-1.0 的平方根是 " << GetSqrt(-1.0) << endl;时 GetSqrt 函数抛出异常，异常被 main 函数中 catch 子句捕捉，输出信息后，程序跳转到 main 函数最后一句输出"程序继续运行完毕。"。而 try 子句后的 cout << "16.0 的平方根是 " << GetSqrt(16.0) << endl;没有被执行。这是因为**异常抛出后会按照 catch 子句出现的顺序依次检查，当找到匹配的 catch 处理程序时后面的所有 catch 子句就被忽略。**根据这个原理，若 catch(...)放到前面则其后的所有 catch 子句就不会被检查，因此它只能放到 try 块的最后。

其实很多情况下 catch 子句的处理程序并不需要访问异常对象，只需要声明异常的类型就够了，例如上面程序中的 catch 子句就可以改成

```
catch(double)    // 捕捉 double 型异常
{
    cout << "发生对负数求平方根的异常。" << endl;
}
```

当然如果需要访问异常对象就要给出参数名，就像上面程序中的 catch(double y)。

三.异常接口声明

我们可以在函数的声明中给出它可能会抛出的所有异常类型。例如：

```
void func()  throw(X, Y);
```

上面的语句表明函数 func 能够抛出也只能抛出 X、Y 及它们子类型的异常。

若函数的声明中没有给出任何异常接口声明，则此函数可能抛出任何类型的异常。例如：

```
void func();
```

如果函数不抛出任何类型的异常，则可以这样声明:

```
void func()  throw();
```

这一讲的内容比较多，希望大家好好掌握，多多练习。这是本教程最后一讲，鸡啄米希望大家学完以后能够实践和应用起来，最终成功加入到C++开发队伍中来。大家有问题可以经常回鸡啄米博客来交流讨论。