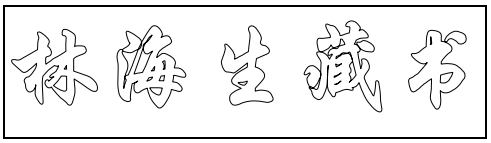
白话 C++



☆ ToYang 网络文化出版社 ☆

2008 年 12 月第 1 版 2008 年 12 月第 1 次印刷

定价：99.80 元

目　　录

第十三章 函数(二)

某勤奋好学之大款看到这段教程，沉思片刻，转头对床上的“二奶”说：“终于明白你和街 上‘鸡’的区别了”。

第十四章 程序的文件结构

程序是由什么组成的？学习到今天,我们至少有两个答案.....今天我们又有一个新的回答。

第十五章 存储类型、作用域、可见性和生存期

局部静态变量的特殊之处：尽管出了函数的作用域之后，变量已经不可见，并且也失去了作用。但是，它仍然存在着！

第十六章 数组(一)

虽说是嚷嚷着要回到童年，可以看到输出结果后，我怎么觉得它有些“少儿不宜”呢？如果你有女朋友，把这个程序寄给她吧。她一定会很开心。

第十七章 数组(二)

系主任在使用你的这个程序时，十个指头一定在不停地颤抖……

第十八章 数组(三) （选修）

知道的人明白我是在说“快速排序”，不知道的人还当我是在说小布和老萨扔板砖哪？

第十九章 指针(一) 基本概念

有一天我们去人民路 108 号，今天纸条写着的地址是：“美眉街 8 号”，于是我们兴冲冲地去了……

第二十章 指针(二) 为指针分配和释放内存

昨天来时，美眉还住在这里一座别致小阁楼里，今日故地重游，这里竟成废墟一片

第二十一章 指针(三) 实例演练与提高

简单变量是一间房屋…数组是房间数固定的一排房子…指针呢…它不是实际房子，而是设计纸上的房子。

第二十二章 结构

说了这么多，大家不要被“面向对象”吓坏了。今天我们所要学习的面向对象的设计方法，很简单：把同属于一个整体的“数据”，归成一个新的类型去考虑，而不是分割成每一部分。

第二十三章 类（一）封装

从有了“类”开始，C++的世界越来越有趣了。前面说类就像一个家，家里有成员（数据或函数），现在，我们还要讲“访问”类的成员……想像有个类叫“美女”。

第一章 程序漫谈

1.1 硬件、软件、程序

对于计算机，也许你是老鸟，也许你是菜鸟……

但不管怎样，如果你此时此地你要学习编程，那么你应该多多少少知道点什么叫硬件什么叫软件——反正我不管你懂不懂，为了面子，我很不乐意你问我什么叫硬件什么叫软件——我做在这凳子半个小时了，一直想不出如何给二者下个定义。

美国一个电脑神童说：“凡是摔到地上会坏的就是硬件。”我深感不妥，众所周知，如果把硬盘摔到地上，那么硬盘坏了，里头的那些数据——都是软件——也一样地坏得让你我心疼。

倘若按字面上理解，那就更加的矛盾重重：硬盘硬是硬件；软盘软也是硬件。

还一种说法是：看得见摸得着的为硬件，看不见摸不着的为软件。刚觉得它说得不错，但马上我就发觉了它的破绽：我现在用的 Word2000，它就在屏幕上，界面美观，操作方便……

无奈之下，我搬出金山词霸，它说：“硬件：计算机及其它直接参与数据运算或信息交流的物理设备”。挺好。硬件就是设备。平常我们生活中的各种设备，洗衣机，冰箱，电视，还有螺丝刀，钳子，都是硬件。

软件呢？“软件：控制计算机硬件功能及其运行的指令、例行程序和符号语言”。指令、程序和符号语言是什么且不说，至少我们得知：软件是用来控制硬件的运行的。

这就好办点。我们可以打比方：譬如汽车，其本身自然是硬件，但关于驾驶车的那一套技术，及有关交通规则，我们可称为软件，因为后者控制了前者的运行方式。

（不传之秘笈：如果你英语一般，学习编程时，别忘装上金山词霸。并且装上后立即上网升级）

现在来谈“指令、程序、和符号语言”。我想交通方面的“软件”确实就是这些东西。我不会驾车，但曾多次看到警察在我上班坐的班车前用指头一指，就令我们的司机脸色发青。之后，一套既定的处罚程序被执行。很快，听说我们的司机又在学习那些用来表示“单行”、“只许右拐”、“不许停车”、“禁鸣”等奇奇怪怪的符号语言了……

事实上，说软件看不见摸不着其实也正确。因为它们是思想，精神，规则，逻辑。本身是抽象的，确实不可触及。但软件总是要有载体来存放，要有表达或表现方式，这些使得它们变得形象具体起来。在此意义上，说软件是摔在地上坏不了的东西，也相当行得通，神童毕竟是神童。

最后，什么是程序？我决定斗胆来给它下个定义：

程序是一组按照一定的逻辑的进行组合的指令。

因此，在以后的学习过程上，很多时候我们会觉得程序就是指令；同样很多时候，我们觉得程序就是逻辑。

当然，更多的时候，我们并不区分程序和软件二者。也许前者更趋于抽象，而后者趋于具体。比如我们在写那些表达我们的思想逻辑时，我们喜欢说“写程序”；而当程序完成，可以待价而沽时，我们称它为软件。

1.2 计算机语言

程序用计算机语言写成。编程的实质就是你用计算机语言来表达你解决问题的逻辑。

那么，什么叫计算机语言？

先不必去解释。因为，计算机是机器，机器不是生物，它怎么能有语言？小猫小狗有语言我尚可相信，机器也有语言，还要我们去学习，这似乎有渎人类之尊严。

如果我不把这个结解开，可能部分特别在意人类尊严的学生对学习编程从此产生心理障碍，无法继续学习……

狭义上，我们讲的语言，汉语英语广东话，它是语言，有声音。小鸟之间吱吱喳喳，大抵也是语言。但其实语言二字虽都带口，却不是说非得有声才称为语言：哑语无声，但它也是语言。广义上讲，语言是沟通、交流的一种手段。基于此，我们认为所有的机器或工具，也就包括计算机，都有它们自己的语言。比如锤子，它的语言是敲打；比如螺丝刀，它的语言是拧，如果你非要拧锤子，非要敲打螺丝刀，那么结果就像你用法语和广东佬交谈，用粤语和法国佬说话一样莫名其妙。

一般地，越复杂的机器，人类与其沟通的语言也越复杂。譬如汽车，你想驾驭它，你就必须去驾校参加学习。想一想，开车的时候，我们的确是在和车进行沟通。如果你俩之间的沟通出现差错——你心里右转，手却一个劲向左转方向盘，向机器发出了错误的命令——这将多么可怕！

至此，我们的心理障碍可以消除了。小猫小狗有语言是因为它们聪明，而机器有语言却是因为它们的笨：它们笨，没办法像动物一样可以通过培训来理会人类的意愿，所以，让人类来反过来为它们定一套沟通的规则，然后人自己去学会这些语言，从而可以方便控制机器。

再所以，我们推论，凡是机器语言都是笨笨的语言。机器语言可以分低级语言和高级语言，但无论何者，都是笨得可爱——学得越多你就会越发现它的笨和可爱。另外，当我说越复杂的机器，其语言也越复杂时，我用“一般地”加以修饰。这是因为，发明和发展机器的智者们会为机器制造出越来越高级的语言，这些高级语言，最终越来越接近人类的自然语言。就像计算机，我们有信心相信，终有一天，它能听懂我们的语言——这就是流传在程序员中的一个梦。当程序员熬红眼敲打数万行代码时，他们便会想起这个梦：闭上双眼，伸腰，对 PC 说：“BEGIN……”；深呼吸一次，然后说：“END”。张眼时发现计算机已完成了所有工作……

（不传秘笈：程序员必备之工具：日产乐敦牌眼药水一瓶，用于预防角膜炎；韩产 777 牌指甲刀一枚，用于铰除因击键刺激而疯长的指甲；国产肛泰若干，治疗因长期坐姿不当而导致的痔疮）

回到计算机。它是机器，也是人类有史以来，继发明使用火、电、电子这些改善人类生活的工具后，最为重要，最为先进，最为广泛使用的工具。它的机器语言之复杂程度可想而知，已经复杂到必须成为一门大学的专业课程。然而别忘了我们前面的结论，语言只是沟通的手段。在这个意义上，当你用鼠标或键盘在计算机上进行输入时，只要你输入的是正确操作，我们都认为你在使用计算机语言，因为你确实是在用一种特定的方式或动作，进行和计算机的交流。

当然，这里的课程并不特意教你任何有关计算机的基本操作。计算机的基本操作主要是指如何使用计算机内已有的软件产品，比如 Windows 本身（操作系统是软件，称为系统软件）；比如办公系统 MS Office 或 WPS Office（这些实现工作生活中具体应用需求的称为应用软件）；比如游戏（一种特定的，只拿来玩的应用，称为游戏软件）。但我们不同，我们学的是如何编写软件。也就是说，我们将是发明人，设计师，创造者；而他们（到今天仍拒不学习编程的家伙）都只是使用者。（我突然有些不安：这么说其实纯属煽情。公平地讲，任何人的任凭创造都是别人的劳动成果之上，任何人也都在创造自己的杰作）

程序（或软件）是用计算机语言写出来的。

- 写一个程序，大致是这么一个过程：
- 人有一个问题或需求，想用计算机解决……

- 人想出解决问题或实现需求的思路……
- 人将思路抽象成数学方法和逻辑表达或某种流程的模式……
- 程序员将数学方法，逻辑表达中的数据和流程用计算机语言表达，称为代码……

用计算机高级语言写成的代码被语言的实现工具（VC，VB，Delphi，或 C++ Builder）转换成计算机的最低级机器语言。这就完成了人与机器在程序制定上的最后沟通。

可见，你的思路是先用人类自己的语言思考，然后用一门计算机语言写成代码，最终，需要一个语言工具来将它转换成机器可以理解的机器语言。我们要学的就是一门承上启下的计算机语言。这样语言有很多：BASIC，Pascal ,C, C++, Java, C#……我们学 C 和 C++。它是使用最多的语言。有关 C，C++ 的更多特点，我们将在下一节谈到。

尽管你完全可以直接用最低级的计算机语言——机器语言——来写代码，那样就不需要语言工具了，但在这里你要弄清了，我们不是教机器语言。下一节，你会明白用机器直接能懂的语言——不妨称之为原始的机器语言——写软件，在今天是多么的不现实。

1.3 语言和实现语言的工具

1.3.1 机器语言

你知道香蕉叫什么吗？就叫香蕉？叫 banana?

错，都错。

香蕉叫“牙牙”。

这是一个 baby 的语言，一个婴儿还没学会人类的主要语言，所以面对喜欢的东西总是发出咿咿呀呀的声音，也许你听不懂，但这是她的语言。符合小孩特点的语言。

计算机的机器语言也一样，必须符合计算机的硬件特点。而痛苦就在这里，越符合机器的特点，同时也就越不符合人类的特点。

计算机，全称电子计算机，20 世纪 40 年代，无线电技术和无线电工业的的发展为电子计算机的研制准备了物质基础。1943 年~1946 年美国宾夕法尼亚大学研制的电子数字积分和计算机 ENIAC（Electroic Numerical Integrator And Computer）是世界上第一台电子计算机。ENIAC 计算机共用了 18000 多个电子管，15000 个继电器，占地 170 m2 ……

这是计算机的始祖，一堆电子管。随后，电子计算机进入第二时期，小巧的晶体管取代了电子管；再后，集成电路又取代晶体管，电子计算机进入第三时期。

但无论是哪一时期(以后也许不是)，计算机始终采用电子器件作为其基本器件，因此，电子器件的特点，就是计算机的特点。

为什么使要电子？为什么木头不能做计算机——还真别说不能，您也应该知道，最早出现的用于计算的机器，真是木头的。你用过计算尺吗？算了，这玩意儿太简单。以前有人用木头作成齿轮，经过设计，当表示个位数的齿轮转动一圈时，就会带动表示十位数上的齿轮转动 1 格。以此原理，只要你转动转轴，木头机器就会算出 123+456 = 579……

电子元件没有齿轮，但它们的特点是它们有两种很稳定的状态：导电或不导电，假如用不通电时表示 0，通电时表示 1，再通过集成电路实现进位的机制。于是，计数功能就有了基础。我们用图表示：

我们生活中常用的数逢十进一，称为 10 进制。而计算机，由于其电子元件的特点，它是二进制数。

这里简单地对比一下这两种进制造成的区别，以帮助你更容易看明白上图。

十进制数：最低位称为个位，高一位称为十位，再高一位称为百位。为什么这样称呼？因为在个位上，0 表示 0，1 表示 1，2 表示 2，3 表示 3……；在十位上，0 表示 0，1 表示 10，2 表示 20，3 表示 30……总之，每高一位长十倍，为十进制。

二进制数：最低位仍可称为个位，但这里称为 1 位。1 位上，0 表示 0，1 表示 1，2 呢？没有 2，因为逢 2 就得进 1（后面同）。高一位称为 2 位，0 表示 0，1 表示 2，再高一位称为 4 位，0 表示 0，1 表示 4。可以看出，每高一位长 2 倍，为二进制。

现在看上面的图，00，01，10，是三个二进制数。根据上面的进位方法，你可以算出它位分别表示十进制数的 0，1，2 来吗？如果你算得出来，不错，值得表扬。算不出来，别急，我来告诉你。首先，当你面对二进制数时，先要扳过来它们从低到高（从右到左）的位依次不再是个位十位百位，而是：1 位，2 位和 4 位。

00：都是 0，所以它就是 0；

01：2 位为 0，1 位为 1，表示 0 个 2 和 1 个 1，所以是 1；

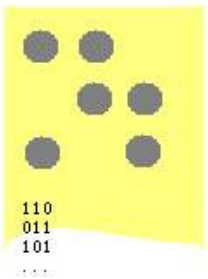
10：2 位为 1，1 位为 0，表示 1 个 2 和 0 个 1，所以是 2。

计算机的机器语言正是由这些 0 和 1 组成。事实上，计算机里的所有数据，无论是一个程序，一篇文稿，一张照片，一首 MP3，最终都是 0 和 1。

世界就是这样奇妙。万事万物五彩缤纷，但进了计算机，却只是个 0 和 1 的组合。不由得你会想起道教的古老玄机：“无极生太极，太极生两仪，两仪生四象，四象生八卦，八卦生十六爻”

严重跑题。

机器语言尽是 0 和 1，于是我们可以想像当时（还没有其它语言时）的程序员是如何编写程序的。他们写程序不用坐在计算机前，而在家里或什么地方，拿笔在纸上画圈，一圈两圈三圈（感觉有点象阿 Q？），圈够了就给专门的打孔小姐照着在纸带上打成孔。最后这些纸带被计算机“吃”进去并读懂，然后执行。来看一眼侏罗纪的程序吧：



（如用孔表示 1，则左图表示三行数 110,011,101）

面对这样的“程序”你是否表示狐疑？别以为我瞎说，也许你的电脑很先进，是 P4 吧？但在你的电脑上，仍有那种程序历史遗迹：软驱是也。如果有软驱，那你应该能找到一张软盘。知道软盘有写保护吗？仔细看看那个写保护的开关——就是一个方孔——打开，告诉软驱本张软盘不能进行写操作，关闭，告诉软驱本张软盘可以进行写操作。

1.3.2 汇编语言

前面说机器语言尽是 0 和 1，那么是不是可以随便写一串 0 和 1 就算是程序呢？不是。就像汉语是由汉字组成，可我要是说下面这一串汉字：　　天爱我京门北安

你觉得我是在说人话吗？

机器也有自己固定词汇，在机器语言里，称为机器指令，程序是由指令及数据组成。这些指令是一些固定的 0 和 1 的组合（不同产商不同型号的机器，其指令又有不同）。作为程序员，就得将这些指令一次次正确地用 0 和 1 拼写出来。

你决不会将“我爱北京天安门”说成上面的话，但你极有可能将 10101101 写成 10010101，对不？所以很自然地，出现了用符号来表示这些固定的二进制指令的语言，这就是汇编语言。

下面是一段我从 C++ Builder 的 CPU 调试窗口摘出的代码，它实现的功能是：

已知 b 等于 1; c 等于 2; 然后计算 b + c 值，并将该值赋给 a 。

把这段代码的机器语言(左)和汇编语言（右）进行对照，你可看出二者各自特点。

```
10001010 01010101 11000100 mov edx,[ebp-0x3c]
00000011 01010101 11000000 add edx,[ebp-0x40]
10001001 01010101 11001000 mov [ebp-0x38],edx
```

汇编语言仅是机器语言的一种助记符，没有本质的区别，所以很多时候，我们把二者等同视之。

无论是机器还是汇编语言，都让人看了头痛，好在我们并不去学它们。

1.3.3 高级语言

汇编语言和机器语言虽然很难记难写，但它们的代码效率高，占用内存少，这相当符合当时计算机的存储器昂贵，处理器功能有限等硬件特点。

众所周知，计算机的硬件发展飞速，功能越来越强大。一方面，它有能力，人们也要求它能处理越来越复杂或庞大数据量的计算功能，机器/汇编语言已经无法胜任实现这些需求；另一方面，硬件的发展和关键元件价格的降低，使得程序员不需要在程序的降低内存占用，运算时间上花太多的精力， 这样，各门高级语言便接二连三地出现了。

那么，高级语言“高级”在何处呢？前面我们说过，一门计算机语言“越符合机器的特点，同时也就越不符合人类的特点” 。人类总是希望凡事能舒服点就舒服点，于是某一天，先知先觉们一声怒吼“是该到让计算机语言接近人类的时候了！”从此冒出了 Pascal,冒出了 C， C++， BASIC 语言等数百种高级语言，现在又有 Java, C#等等。高级语言高级在哪里呢？就高级在它总是尽量接近“高级动物”的自然语言和思维方式。

那么多高级语言，我们为什么挑了 C++呢？

向来头痛这种问题，其实无论是 Pascal, BASIC,还是 C++，甚至 C 和 C++相比，每一种语言都有极大的相通之处，又都有各自的独到之处。我大致鼓吹一下 C++吧。

首先，这是全世界用得最多的计算机程序语言。

其次，C/C++语言既有高级语言的优点，又在很多方面保留了低级语言速度快，可进行很多具有可直接映射硬件结构的操作的长处，我们无时不用的 Windows 等操作系统，就是用 C 和汇编写成。事实上很多人称它为中级语言，这样两头俱备的语言，当然值得学。

再次，C 语言本身，只有 32 个关键字（固定词），C++也只是进行了有限的扩展。另外，C/C++语言是众多语言中最简洁，紧凑，灵活的语言，学得易，用得爽！

再再次，由于前面所说的，C/C++是中级语言，它所生成的机器代码自然更接近于直接用汇编所写（和其它语言相比），所以，同样的程序，用 C/C++写，代码小却跑得快。

再再再次，C/C++语言是少见的，不专属于（因为版权或龚断）某一家公司或组织，你完全不必担心你只是学了某门某户的 C 或 C++，到别的地儿全玩不转。C/C++有美国国家标准协会制定语言标准。事实上，你就是到 UNIX,Linux 上写点 C/C++代码，也能跑。

最后，一门语言再好，如果没有实现工具（语言本身只是一种规范，必须有编译器可以实现它），那么也不成了屠龙之技，C/C++，你可能用微软的 VC，也可以用我们推荐的 Borland C++ Builder，二者都相当优秀。

1.3.4 语言实现工具

1.3.4.1 C++ Builder 的基本功能

我们用高级语言写程序，我们很得意，因为高级语言比较接近人类的语言，我们用起来得心应手，所以我们当然得意。但我们更得意的一定是让程序代码赶快变成可执行文件。

无论是在写代码的过程，还是最后要编译成可执行文件，都需要有一个工具存在。这一具一般成为编程集成环境（IDE）。之所以称为集成，是因为从写代码到最后软件的出炉，我们需要它的地方实在太多了。这里列出其中最重要的功能项。

1、 代码编辑 方便的代码编辑功能。尽管你可以使用记事本、Word 或其它任何文本编辑器来写代码，但除非特殊需要，否则那将是极为低效的方法。相反，现在的编程集成环境，都相当的智能，举例如代码自动功能，可以很多情况下自动完成我们所需的代码，既准确还迅速。Borland 公司出品的编程集成环境不仅有常见的关键字高亮等功能，还支持代码模板，支持键盘宏，同样支持高级的脚本插件功能。

2、 界面设计 可视化的程序界面设计功能。你所要产生的窗口，在设计期间就真实地出现，包括字体，颜色，定位，比如，你不仅可以插入 falsh 的动画，而且无需运行，就直接可以在你的界面上看到该动画的演播，这是别的编程环境不能做到的。

3、 程序编译 这更是编程工具的主要功能。前面已讲过，我们写的代码，在成为机器能懂的可执行程序时，必须通过编译。

4、 程序调试 如何尽量减少你的程序的 BUG？没有编程集成环境都提供的强大调试功能，我们做的程序将毫无质量保证。

5、 代码优化 Borland 提供的编译器不仅在编译速度一直在美国屡获大奖，而且其代码自动优化功能一直领先对手几近一个时代。使用编程集成环境，我们可以轻松获得更快更优的最终可执行程序文件。

6、 辅助程序安装 程序的安装已属于另外一种工具的范畴，但我们仍可以通过编程集成环境来决定最终生成单一可执行文件，还是带有其它动态库。如果是后者，我们还可以通过集成环境来检查程序运行时调用了哪些动态库文件。

C++ Builder 提供的功能远不止我上面所说的，并不是因为我嘴笨，而是我认为对一个工具，你只有动手使用，才会真正了解它。

C++是一门语言，而 Borland C++ Builder 则是语言实现工具。作为一个编程工具，CB 提供以上功能正是份内之事。在这个意义上，你可以认为 CB 是 Word2000,而 C++则是英语或汉语。正如我们用英语或汉语在 Word2000 上写出优美文章，编程可说为：我们用 C++在 CB 上编写出优美的程序。

1.3.4.2 VCL vs. MFC

在作为一种编程工具的意义上，我们认为 C++ Builder 和你也许常听的 VC (Visual C++)没有什么本质的区别。就像 Word2000 和 WPS2000 在本质都是字处理软件。但现在我们要从另外一个角度讨论 C++ Builder 这个编程工具。

这个角度，就是“封装”——面向对象编程思想中的最重要也是最基础的概念。

一个要学习编程的人，可能从 C 开始学起。学 C 时，我们没有接触那些挺玄的概念，到了 C++，一切就来了，什么面向对象，什么封装、继承、多态……于是我们兴奋起来，努力去理解，掌握，运用这些概念所代表的技术，在掌握这些别人暂时未理能的概念之后而颇有成就感……。现在我要问的是，为什么要有这些概念？这些技术？正确回答这个问题，不仅有助于我们今后对编程语言各种概念的学习，而且它能让我们避免成为新技术的奴隶——这一切也许听起来有些形而上，不过我想通过以下讨论，至少可以回答一个很现实的问题：为什么要选 C++ Builder？而不是我们更常听的 VC？这是我碰到的编程初学者较疑惑的问题之一。

如果人类长有翅膀，那么飞机大抵永远不会被发明。飞机的发明，是为了弥补人类自己不能飞翔的缺陷。不能说所有的技术都是这样，但 C++对于 C 的发展，完全是为弥补程序员脑力的不足。一个在校生在学会 C 后，往往并没有机会用 C 去实践一个大中型的项目，体会不到在一个庞大软件工程中，非面向对象语言的短处，所以在之后学习 C++的过程中，也就很难真正体会到面向对象语言的长处。简短一点说：不知道 C 的短处，就不懂 C++的长处。相反，倒是很快就发现 C++的缺点：它的代码效率多数情

况下都要比 C 低不少。

前面我们说过低级语言与高级语言的对比。C++语言也正是从语法结构，语言功能上来限定或实现一门编程语言更加接近人在现实生活中的思维习惯，从而达到减轻人的记忆和判断上的负担。这其中最佳的方法之一就是所谓的“封装”。

关于封装，初学阶段最直观的比喻就是抽屉。抽屉将各种**对象**分门别类地进行存储。譬如中药房，上千种的中药被上千个贴有标签的抽屉“封装”起来，这一充满艺术性的“封装”，使一个一点不懂中药的人也可以去当抓药师，相反，如果没有这些“封装”呢？

上一段中我们把对象两字设为黑体，你应该在理解封装的同时明白什么叫对象了：对象就是东西，就是物品，就是事物。不信你把上段中的“对象”替换为“东西”或“物品”，或“事物”是不是读得更顺点？总而言之，面向对象的编程方法其实就是说，让编程的思路尽量符合人们生活中事务处理中已经掌握的科学方法。它并不是牛顿发现地心引力一样的科学探索成果，它只是把已有的科学运用到计算机编程。

“封装”中草药的方法也许只有一种，而程序所要解决的问题，比区分中草药要复杂得多，所以，如何进行常用数据，行为的封装，也就仁者见仁智者见智。

VC 的 MFC 和 CB 的 VCL 都是基于（但不限于）对 Windows API（应用程序接口函数）的封装。为什么要对 API 进行封装？这就是回到了我们前面说过的，为什么有了 C 又会有 C++的问题。因为操作系统是用 C 和汇编写成的，它获得到操作系统必须的代码效率，但对应用程序开发者而言，它失去了易用性。所以微软和 Borland 都使用高级语言对之进行封装工作。二者谁进行得更好呢？

VC 的封装类库称为 MFC，它是一种很低阶的封装，它并没有按照人类的思维习惯来得新组织重新解释 Windows 对象（指 Windows 编程中所需的数据，处理，机制，接口）， 纯粹是 API 一对一的翻版。这个的封装工作带来代码封装所固有的代码效率降低的副作用，却没有给使用者带来任何方便。如果你是编程初学者，而你身边又有 VC 高手，那么你一定要多多向他学习请教，因为一个真正的 VC 编程高手，其同时一定也是一个深刻理解 Windows 内核机制（消息循环，内存管理，多任务实现，资源使用等），熟悉 Windows 各种常用 API 函数等等的高手。相反，如何一个人对这一些知之不多，而自称为 VC 高手。你放心，参加了笔者下一级的 Windows 编程学习，只需 2 星期，你就会明白那种只会用 VC 的向导写程序的人是什么样的“高手”了 。

C++ Builder 对封装库称为 VCL(带 VC 字样，可别以为它是 Visual C++,其实它是:Visual Component Library，即：可视控件库)。

要想成为 Windows 编程高手，最终一定要绕过各种封装，理解 Windows 对象。但作为一个初学者，我们必须挑选一个好的封装。下面我们举字体（Font）作为例子，将三者：没有封装过的 Windows 字体 API、封装过的 MFC 字体对象、封装过的 VCL 字体对象。做一个对比。为了保证不会有偏倚和差错，有关前二者的代码，都是笔者从 MSDN（微软提供的帮助文档）中直接拷贝出来。

Window API	Windows API 创建指定样式字体： HFONT CreateFont(int nHeight, // height of font int nWidth, // average character width int nEscapement, // angle of escapement int nOrientation, // base-line orientation angle int fnWeight, // font weight DWORD fdwItalic, // italic attribute option DWORD fdwUnderline, // underline attribute option DWORD fdwStrikeOut, // strikeout attribute option DWORD fdwCharSet, // character set identifier DWORD fdwOutputPrecision, // output precision

	DWORD fdwClipPrecision, // clipping precision DWORD fdwQuality, // output quality DWORD fdwPitchAndFamily, // pitch and family LPCTSTR lpszFace // typeface name);
MFC （Visual C++）	将 HFONT 封装为 CFont BOOL CFont::CreateFont （ int nHeight, int nWidth, int nEscapement, int nOrientation, int nWeight, BYTE bItalic, BYTE bUnderline, BYTE cStrikeOut, BYTE nCharSet, BYTE nOutPrecision, BYTE nClipPrecision, BYTE nQuality, BYTE nPitchAndFamily, LPCTSTR lpszFacename);

VCL （C++ Builder）	将 HFONT 封装为 TFont1
	要设置字体名，高度，尺寸等使用以下代码：
	Font->Name = “宋体”; //设置为宋体
	Font->Size = 24; //设置尺寸为 24 号 2
	将字体的粗，斜，下划线，删除线再封装为 TFontStyle 属性：
	Font->Style = Font->Style << fsBold << fsUnderlien; //字体增加粗体和下划线属性。
	对于字体不常用的旋转等属性，不进行封装，可直接调用 API 函数设置 TFont 的 Handle 属性。

比较表中第一行和第二行：前者是原始的 API，后者是 VC 精心的封装成果。可惜二者几近雷同。既然你要封装，你就是要让它变得面向对象，易记易用；一模一样的照抄一遍，然后改改参数的名字，意义何在？如你是想维持代码的效率，那么在繁杂度一样的情况下，为什么我不直接使用效率更高的 API 函数呢？

倘若说，MFC 的“封装”纯粹是一种多余，那或许也还可以接受。然而 MFC 偏偏还要在这种冗余的封装上建立自己的应用程序架构。和前面的“封装”一样，MFC 建立应用架构的出发点也是良好的，为了方便 Windows 程序员编程的难度。结果却更糟糕。有问题的架构犯了类库或接口提供者的大忌：“有协议编程”。

什么叫“有协议编程”？我们先来讲“无协议编程”。所谓“无协议编程”是指接口的提供者在提供接口时，同时也提供接口的使用约定。这一套约定应该在接口所要提供的功能上广泛适用，而无须再有种种特殊的例外。这样的接口显然非常适于使用。打个比方就如交通规则。“红灯停绿灯”行应该是普遍适用的规则，如果在红灯停绿灯行还附加以下例外条款：

1、 单号并且在阴天下东行的路面上红灯行绿灯停；

2、 轿车在乘客人数为奇数的情况下见红灯允许直行但不能右拐；

3、 日间 12 点到 14 点区间并且街心交警为女性时，红灯停或行临时取决于女警的身高是否高于 1 米 70……

尽管这些例外都是一些不常见的情况，但想像一下你自己是这种制度下生存的司机吧。学习编程，如果挑错了我们每天都要面对的封装类库。就将永远都在努力处理这些无任何意义的问题。和司机不同的是你倒不会有多难受——一个人陷在泥潭中久了，往往就会认为自己挣扎的动作优美得堪称艺术。

MFC 的 CWnd 提供了对 Windows 最基本的窗口元素的封装，其中对创建窗口的函数的封装为：

未封装的 API：

```
HWND CreateWindow(
LPCTSTR lpClassName, // registered class name
LPCTSTR lpWindowName, // window name
DWORD dwStyle, // window style
int x, // horizontal position of window
int y, // vertical position of window
int nWidth, // window width
int nHeight, // window height
HWND hWndParent, // handle to parent or owner window
HMENU hMenu, // menu handle or child identifier
HINSTANCE hInstance, // handle to application instance
LPVOID lpParam // window-creation data
);
```

使用这个 API 函数，我们可以创建各种窗口。

CWnd 封装的函数：

```
virtual BOOL CWnd::Create (
  LPCTSTR lpszClassName,
  LPCTSTR lpszWindowName,
  DWORD dwStyle,
  const RECT& rect,
  CWnd* pParentWnd,
  UINT nID,
  CCreateContext* pContext = NULL
  );
```

不用我说，你也能看出这仍然是个改改参数的蹩脚的封装。我们不去管它，现在我们关心的是：CWnd::Create 对 CreateWindow 进行了封装，可是这一封装的结果是：原来 CreateWindow 能实现的一些事情，在 CWnd::Create 里突然成了例外。是的，为了适应 CWnd 在 MFC 架构中所处的角色，程序员在涉及 CWnd 时必须记忆这样一条例外：

“CWnd 的 Create 用于创建窗口的实际元素，但其中参数 dwStyle 不能包含有窗口风格中 WS_POPUP(弹出式窗口)，如果你要建立一个带有该风格的窗口，请使用 CreateEx……”

我仍然要说 VC 也是一个很优秀的编程工具，但对于不想浪费无谓精力的编程初学者，我个人建议使用 Borland C++ Builder，因为它实现真正的对象封装，从而，你可以节省不低于 80%的时间来学习编程本质的知识——就是我们常说的数据结构与算法，这些东西最终决定你的编程能力。

这就是 C++ Builder 提供我们的最重要的东西：

VCL 类库：一个好的底层类库，让我们从学习编程最初时刻就自然而然地学会使用面向对象的方法来

写程序。它大大降低了我们入门门槛的高度，却又让我们一开始就站在比别人高的位置看待有程序有关的问题；

组件技术：组件技术代表了当今编程技术的主要方向，其设计思想与 MS 力推的 Active 控件如出一辙，拥有相同的先进性。只有借助组件技术，我们才有可能从一个初学者，迅速到达可实际工作的编程工作者；另一方面，如果作为组件的提供者，我们可以编写组件的过程中迅速提高自己的编程能力。

C++ Builder 还提供了许多其它先进技术，如事件委拖等等，归根到底都通过封装让 Windows 编程原本需要长期积累的才有可能掌握的知识变得直观易懂。如果你刚刚开始学习编程，或者学习较长时间仍没有重大突破。或许使用 C++ Builder 结合本课程系列，是个不错的选择。

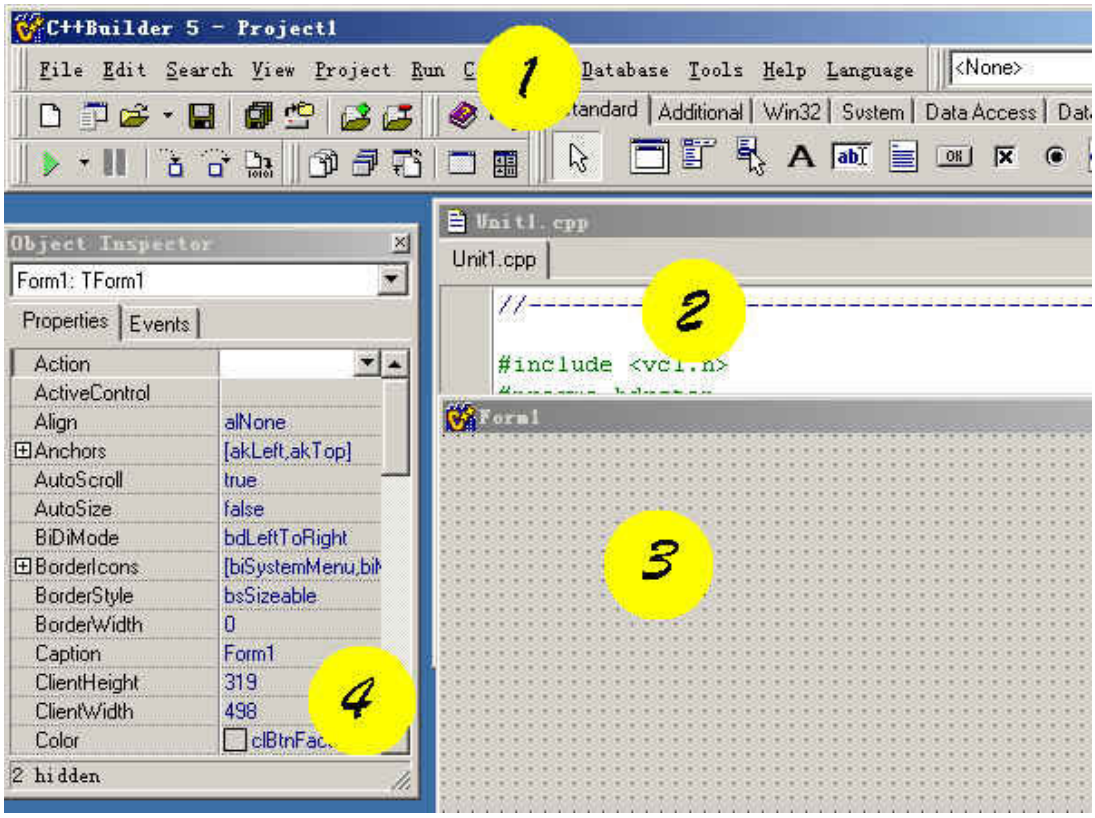
（附：微软最近推出的 C#相信会对上述 MFC 的不足做一个收拾，它对 C++的扩展与约束与 Borland C++ Builder 对 C++的扩展与约束惊人的相似。如果你乐意，我也真的很建议你在学完 C++ Builder 后，继续学习 C#）

第二章 编程环境

在第一章，我们从一较高的高度上谈论——是的，我们只是在“谈论”计算机语言。直到最后一节，我们才相对具体地说到了编程工具。这就好比是和一个初识的女孩子海阔天空地聊了一番国际国内形势后，临走时我们问了她一句：“你有 e-mail 吗？”

2.1 CB 界面

启动 Borland C++ Builder 后，你会发现它的窗口和我们常见的一体化窗口的应用程序有所不同，CB 的各子窗口并没有集成在一个主应用窗口中，而是分散为独立的子窗口。



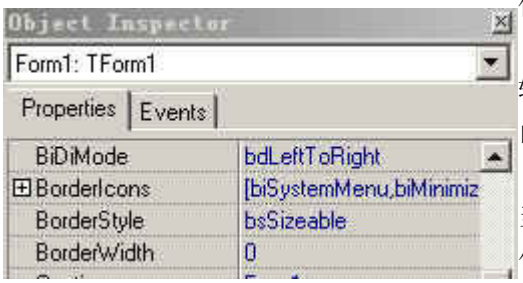
（笔者的桌面分辨率为 1024*728，但为了不让图的尺寸太大，我特意将各窗口拉得很小）

如果你打开 CB 后出现在界面和上图有所不同，可以通过 File 菜单，再选 New Application,建一新的

工程，则上图标注的 2 到 4 的窗口应该出现。如果仍然有缺，请用鼠标点上图的标注为 1 所示窗口的标题栏（以确保为活动状态），然后反复按 F11，F12，可以在以上窗口来回切换。

下面我们来了解这四个窗口：

标注为 1 的窗口：这是我们比较熟悉的应用程序主窗口，虽然它看上去是一个长条，不过其上有主菜单(MainMenu)，工具栏(Toolbar)，和别的软件一样。不一样的是这一“长条”的右下部分的多页式工具条。事实上它并不是寻常意义上的工具条，因为其上的每一工具按钮并不提供执行某一命令的功能。我们称它为控件栏。控件是 CB 提供的了先进的编程思想的体现之一，本章后面我们会初步学习如何使用控件。



标注为 2 的窗口：相信你一眼就可以看出这是一个文件编辑的窗口。我们称之为代码编辑窗口，或简称为代码窗口。由名及义，**这是我们写程序代码的地方**。

标注为 3 的窗口称它为 C++ Builder 软件的窗口并不妥当。事实上，**它是我们自己要写的软件的窗口**。Windows 操作系统的应用软件，譬如 Word2000,譬如 WPS Office,或者简单如画笔，记事本，总是会有一或多个窗口。这是 Windows 应用软件的典型特征（Windows 操作系统也因此称为 Windows）。尽管也可以写没有窗口的应用程序，但大多数情况下我们的程序至少需要一个窗口，所以 CB 在创建新工程时，总是默认为我们生成一个主窗口，这就是标注为 3 的窗口——在程序运行前，我们称它为设计表单(Form)，在程序运行之后，它就是我们程序的窗口。我们的程序需要有几个不同的窗口，就可在设计期间生成同样多个类似标注为 3 的设计表单。

（我对”表单”这个词总是无法产生具体的概念，可是不仅 C++ Builder，还有 Delphi—CB 的“姐姐”，以及 Visual Basic——微软的得意之作，包括.net 计划中的 C#等快速应用程序设计系统，都使用 Form 这个词来称呼**设计期间的窗口**。所以我还是统一口径叫表单。但不管怎样，如果你在我的课程中偶尔发现设计窗口这样的说法，不用猜测，你尽管认定我就是在说表单。一个原则：只在设计期间，我们才有可能称它为表单，当窗口运行了，那就是窗口了，我们从不叫一个运行着的窗口为表单）。

标注为 4 的窗口，标题写着“Object Inspector”直译对象检视器。事实上 CB 在调试程序时还有一个“Debug Inspector”，我倒觉得让后者叫前者的名字更合适。因为这里 Object，也就对象，可不是我们以前说过面向对象的对象。它其实是用来查看，设置当前放在设计表单上的某个/些控件的属性值和事件值的工具。后面我们会用到它，控件，也称组件——但有些 CB 的书区分这两者，认为前者是后者的某一特定子集——就是窗口 1 右下的控件栏上的控件，至于控件的属性，稍后我们使用时，你就会了解。为了直观，我决定称 4 号窗口为控件属性检视器，或者属性检视器，或者属性窗口，总之离不了属性二字。属性检视器有两页：Properties（属性）和 Events(事件)。

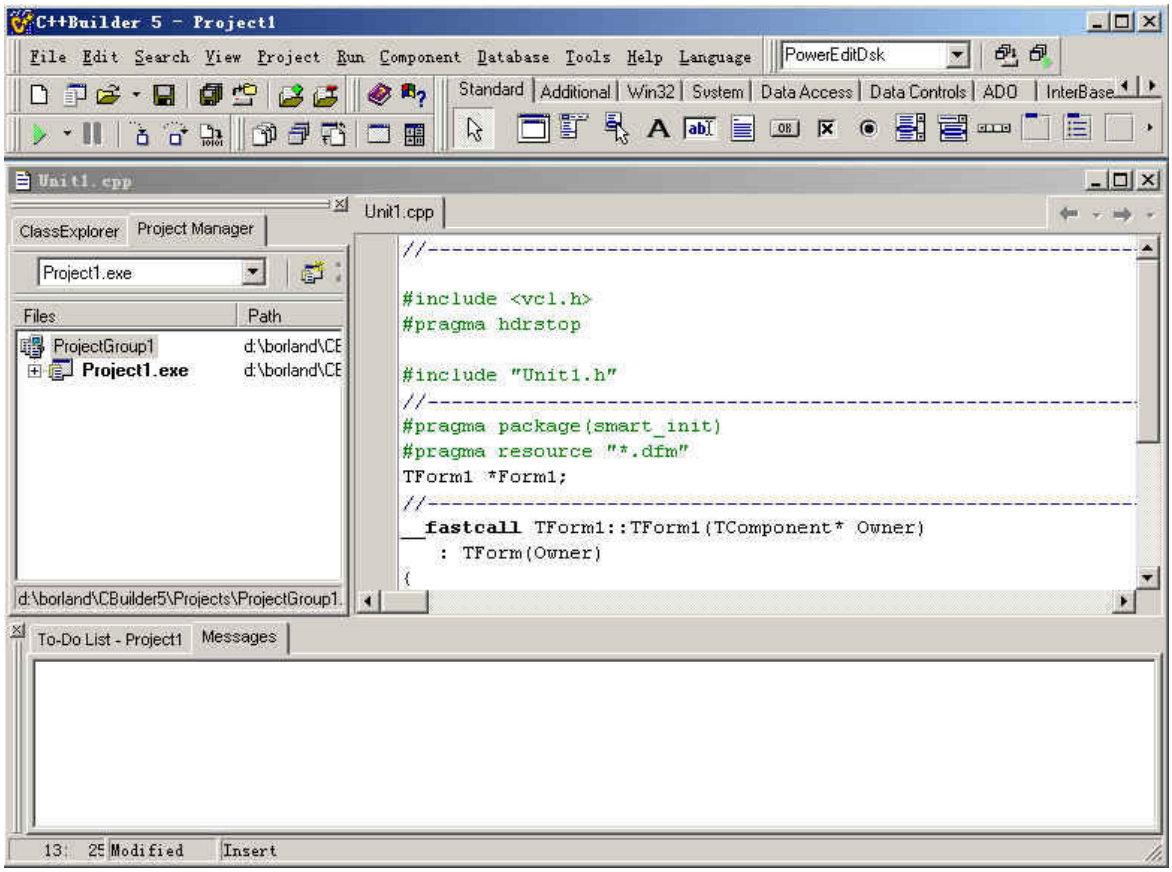
参照左边的图,这里顺便再统一一下口径,如图中 Properties 和 Events;我们称它为多页(Properties 为第一页，Events 为第二页)。

关于分散窗口（苹果机早期的应用程序风格）的得失，我们不想在此讨论。需要说明的是，CB 的的各常用子窗口都提供 Dockable，因此如果你喜欢集成式的窗口，大可通过鼠标拖拖放放来定制自己的集成窗口。

Dockable 是指：拖动窗口 A，当经由窗口 B 的某一边缘地带时，窗口 A 可以成为窗口 B 上的子窗口而停靠在窗口 B 的某一角落。在 CB 里，不仅角落可以停靠，当位置为窗口 B 的中心时，窗口 B 还能以多页的方式加入窗口 B。

大多数软件或许会在退出时保存住最后的窗口位置大小等设置，CB 则提供你随时保存，调用各套桌面设置，比如编写代码时的桌面，调试时的桌面等。

以下就是笔者常用的，用于编写代码时 CB 桌面设置之一，它被我存盘为“PowerEditDsk”：



这套桌面集成了类专家(ClassExplorer)，工程管理(Project Manager)及消息窗口，任务列表(To-Do List)等窗口于代码编辑窗口内。

鉴于如果各位的桌面设置不统一会造成课程讲解上的一些困难，另外还有一个不是理由的理由：CB 有关桌面的设置有烦人的 BUG，所以我们的课程使用 CB 默认的桌面设置，也是文前标有 1、2、3、4 的那张图中所示的窗口位置。

2.2 Win 版 Hello，World

编写自己的第一个程序，并且用字幕打出“世界你好！”——这是“很早很早”以前一本 C 程序教材的作者的发明——后来据说成了经典……不过很多人说这太过时了。不管怎样，我认为这作为我们初次使用 CB 的教学例子仍然很合适。

很多人可能感到有点突然。当 CB 慢吞吞启动后，一堆界面元素无论如何都让人感到这不是一个轻量级的人物；我们什么编程基础都还没有掌握，能驾驭好这个巨人吗？其实这就是 CB 的强大所在：具有高度的弹性，你可以用它编写很小的程序，也可以集合数十个程序员，用它编写大型软件。

牛刀小试开始。

2.2.1 一个空白工程

这是 Windows 的天下，尽管在《白话 C++》的学习中，Windows 编程并非重点，但我们还是选择了 Windows 作为我们认识 CB 的起点。

从主菜单中选择 File | New Application

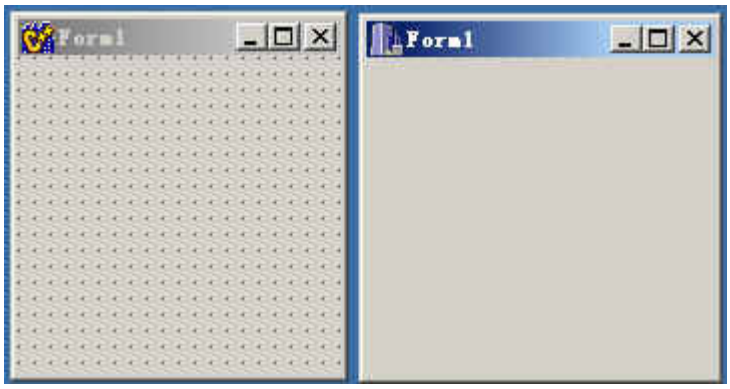
(约定：在谈菜单命令时我们约定用这种格式：File | New Application 表示如上图的实际操作。)

执行 New Application 之后，CB 为我们新一个空白工程。所谓空白工程是指：绝大都数 Windows 程序所共同拥有的，必须的一个框架。再说白点，就是你每写一个新程序，都需要的一堆完全一样的代码，也就是说在多数情况，这是一步机械的工作。既然是“机械”的工作，当然由机器来完成最合适。

很多年前，笔者 VC 和 BC 都未流行的时，做了一个月“机械工”后，和许多那时的程序员一样，笔者很快尝试自己对这部分的代码进行封装。后来笔者又建议单位购买一套中国人写的窗口对象类库；再后来，笔者可始用 BC，用 VC，至今天，对于大家能有机会直接使用堪称最好封装的 VCL 学习编程，笔者能说的一句话就是：珍惜你的机会，珍惜你的 20 元钱。

空白工程带有一个名为“Form1”的表单（上一节图中标有“3”的窗口）。这就是程序运行时的主窗口。验证一下你就能明白：

请选菜单：Run | Run 或者按 F9 键。空白工程——当然也是一个完整的工程——被编译成程序，最后自动运行，出现一个标题为“Form1”的窗口。怎么和设计时的那个表单一模一样呢？当然，所见即所得嘛。不一样的地方也有——设计表单上有一些用于定位的小点，而运行后的窗口没有这些。



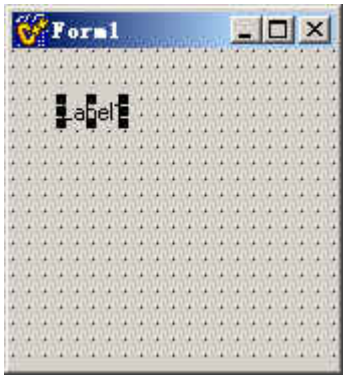
左图为表单，右图为窗口，表单（Form）指设计时的窗口，窗口(Window)指运行时的表单。

关闭该窗口，继续我们的 Hello,World 的工作。

2.2.2 最简单的 Hello World 工程

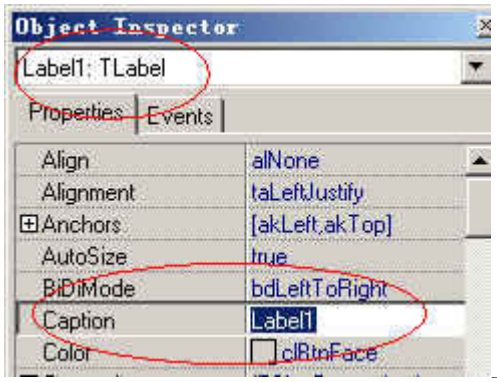
请从 C++ Builder 的主窗口（上一节中标有“1”的窗口）右下部的控件栏中找到如图所示的 Label 控件，同时记住：Label 控件在 Standard 页中。点击代表该控件的图标按钮：

（图中画有字母“A”的图标按钮即为 Label，这个控件用来显示一些简单的文字内容。）鼠标摁下该按钮后（按下后按钮的形状如上图），将鼠标挪取表单 Form1 上随便位置点左键，一个 Label 控件被放在表单上：

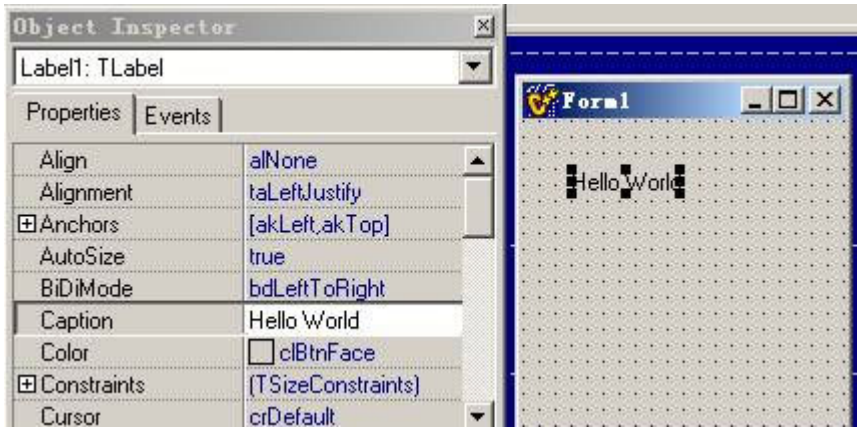


用鼠标再点一下 Label1，确保它为如上图中的选中状态（带有八个黑点块）。我们要通过修改属性，让它显示为“Hello，World”。

主菜单：View | Object Inspector（或者按 F11 键），出现 Object Inspector 窗口，也就是上一节中标有“4”的窗口，我们称为控件属性检视器，通过它，我们可检查并设置当前选中的控件的属性（包括事件）。 由于刚才选中了 Label(如果不是，请重复用鼠标再点一下表单上的 Label1)。所以 Object Inspector 窗口的上部应显示如下：



上面的 Label:TLabel 表示当前属性窗口显示的是 Label1 的属性。同时我们注意下在的 Caption 属性为 Label1,（注意，Name 的属性也为 Label1,千万别混了）。Caption 意为“标题”，它决定一个 Label 显示的内容。你应该很明白怎么做了，在图中所示的 Caption 属性右边的编辑框内，将 Label1 改为 Hello,World。眼尖的学员一定发现，在改的同时，表单上的 Label1 如我们所愿，显示为“Hello,World”了。笔者我眼力不钝，但还是没有看到结果，原因是一个有一个窗口挡住了 Form1。按一下 F2,Form1 跳到前面。



再按 F9,只见屏幕一闪，第一个我们参与设计的程序闪亮登场！

很好，我们已经向这个世界打声招呼。工作似乎有点成果。我们插播一段关于如何存盘的说明。 在CB中保存程序，总体上和在字处理里保存一个文档是一样的操作，比如你一般都要给新文档命名，并且找到一个合适位置保存。这里讲的是特别之处。

其一、建议大家为本课程中所讲的例子程序准备一个统一的文件夹，然后以这个文件夹为父文件夹，再为每一个例子各建一个子文件夹（除非是教程中特别指出需要将两个例子工程放于一处）。

其二、在字处理中（如MS Word），一个文档就是一个文档，而在CB的程序工程中，一个工程除了工程自身的文件以外，还包含其它配套文件。这些文件中，有些可以在保存其它文件时自动保存，现在需要我们手工保存的是工程文件和代码文件。

保存文件是相当频繁的事，我们几乎不用菜单，而总是使用快捷工具按钮(Toolbar)或者键盘，尤其是后者。这里我只讲一个按钮： 。这个有一叠软盘的图标表示Save All（保存全部）,现在请点它（或按Ctrl + Shift +S）。将弹出两次的存盘对话框，第一次要求存盘的是代码文件，CB默认的文件名为Unit1；我就采用了这个默认名，必竟这只是一个小小的工程。随后是工程文件要求存盘，我一样采用了默认的Project2.bpr的名字。（正因为我们经常图省事，直接使用默认名，所以如果不将不同的工程存在各自的文件夹内，重名的文件名就将令事情很糟糕。） 最后是一些提醒：提醒一：CB默认的存盘位置是在其安装目录下的Projects文件夹。建议不要采用它，而是自己在外面新创建一个文件夹。提醒二：当程序复杂时，程序运行甚至编译时就有可能造成死机。所以在写程序时常常存盘是一种非常良好且必要的习惯。

除了这些，存盘是再熟悉不过的事了，在File菜单下有不少和存盘有关的命令。有空大家自个儿看看。

2.3 DOS 版 Hello，World

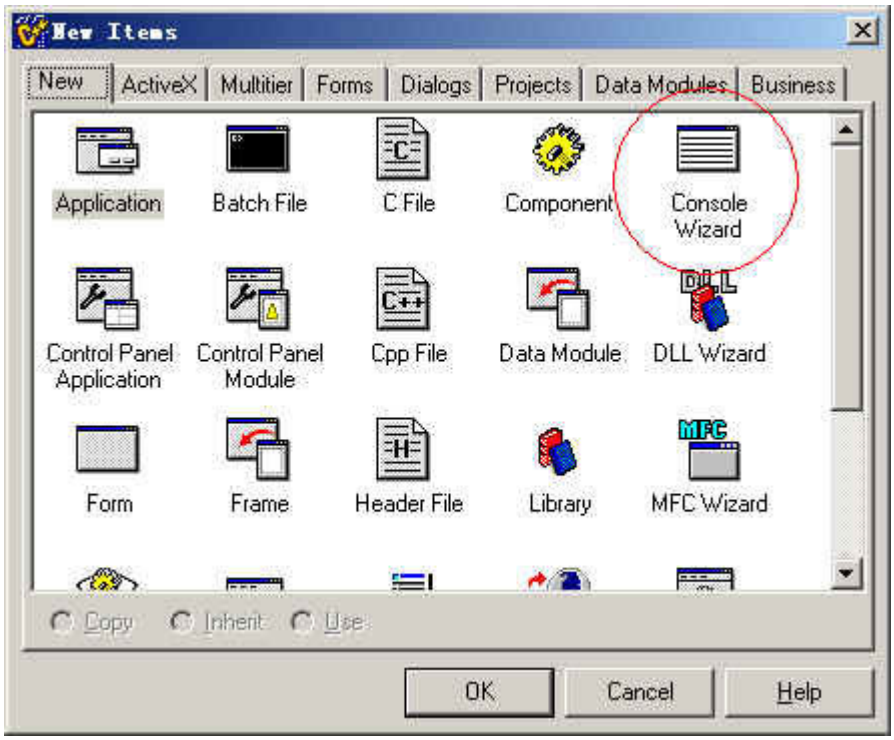
我们可以——并且往往是忍不住就将时间花在——把Windows版本的Hello Word的工程做得很漂亮。不过现在不是时候。在本部教程内，我们要集中全部精力，扎扎实实地打下C++ 语言的基础。这是以后我们深入学习任何编程要点，无论是Windows编程还是你想学Linux编程，也无论我们今后在实际工作中侧重于通讯或数据库或工控；甚至无论我们以后是否用C++这门语言编程——只要有C++根基，你就会比别人更容易掌握各种编程新知。

传言也并非都是假的。关于C++晦涩难懂之类的传言便很现实。另一方面，Windows编程的重点几乎全部都在于如何和Windows操作系统打交道，而和语言本身关系——它要求你有很好的C语言基础。出于这样的考虑及实际运行中遇到的困难，笔者放弃曾有的，“以语言为中心，同时学会Windows编程基础”的理想。 写DOS风格的程序想起来是令人沮丧的——没有华丽的窗口界面，也不再使用那些功能强大的控件。但正因为DOS程序的朴素，它可以使我们在更好地将重心放在C++语言本身。等到学习第二部教程时，我们才有足够的底气对Windows世界说一声：我来了！

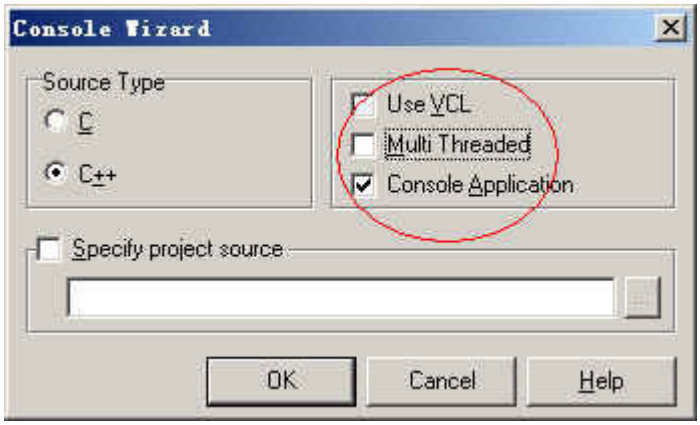
2.3.1 一个空白的控制台工程

如果你的 CB 停留在前面 Windows 版 Hello World 工程中。确保退出 Hello World 的窗口。然后选主菜单：File | Close All。关闭 Windows 版 Hello World 工程。

菜单：File | New，出现一个 New Items 对话框，等待我们选择要新建什么。



默认的选中项是图中 New 页内的第一个：Application。如果选它，确认后的工程就是上一节我们创建的空的工程。但现在我们要建立一个 DOS 风格的程序。请选上图加有红圈的 Console Wizard (控制台程序向导)。 确认后出现向导的第一步：



请您确认将对话框中的相关选项设置成和上一图一致。然后点 OK。

没有表单，只有代码编辑框窗口（第一节的“2”号窗口）内中代码为：

```
//---------------------------------------------------------------------------
#pragma hdrstop
//---------------------------------------------------------------------------
#pragma argsused
int main(int argc, char* argv[])
{
return 0;
}
//---------------------------------------------------------------------------
```

这是 C 语言的主函数，或者称为入口函数。程序从这个地方作为起来开始执行。以后会有更多讲解。同样，这也是一个程序框架，一个空白的 DOS 程序框架。让我们选菜单中 Run ｜ Run 或直接按 F9 键，看看结果和前面的 Windows 空白程序会有什么不同？

眼前出现一个黑色的窗口，然后就一晃而过回到了 C++ Builder 的界面。 黑色的窗口，在不同版本的 Windows 中有不同的叫法和不同的实现机制，但都是 Windows 操作系统兼容 DOS 的方式。这种 DOS 方式在术语上称为“控制台/ console”。如果你不明白什么是 DOS，请通过其它途径另外学习。

2.3.2 用控制台输出“Hello world”

我们需要加入三行代码才能实现 DOS 版的 Hello World。以下黑体部分为你需要在代码中加入的内容：*（约定：在代码中使用黑体，用来表示您应该输入的部分）*

```
//---------------------------------------------------------------------------
#include <stdio.h>
#pragma hdrstop
//---------------------------------------------------------------------------
#pragma argsused
int main(int argc, char* argv[])
{
printf("Hello,world!");
getchar();
return 0;
}
//---------------------------------------------------------------------------
```

这是一个很短的程序，我们稍微做点解说。

如果是一个大程序，代码一多，自己都会看晕，所以就需要在代码加一些注释，用以解说某行或某段代码的用途，或者用以让代码显得清晰。C++中，常用双斜杠 //一直到该行结束的内容来代表注释。上面的代码中，有三行：

```
//---------------------------------------------------------------------------
```

这正是用来从视觉上分隔代码的注释。注释对程序的运行没有影响，只用来给人看。当编译器编译进，它会过滤掉到所有注释内容。

#pragma hdrstop 和 #pragma argsused 两行为编译预处理命令，当编译器在编译本段代码，预处理命令会影响编译的某些行为。

#include<stdio.h> 是我们新加的一行。stdio.h 是一个文件的名字，扩展名是.h，h 是 head 的意思，所以被称为头文件。这个文件在哪里呢？在你的机器里，在 C++ Builder 安装后的文件夹内的某个子文件夹内。在那个文件夹内，C++ Builder 为我们提供了上千个头文件。头文件起什么作用呢？我们做个比喻，设想你家很富，但不幸四周的邻居都穷。今天李四推门而入，四下搜索一番，借走一把钳子，明天张三不请自来，抬走一台电视……天天有人直接进入你家，这显示是危险的。于是你在门口贴了各式清单，其一为五金工具，其二为家用电器清单……当别人需要某种东西时，先从清单上找一找是否存在，如果有，则在清单上作个登记……

Borland C++ Builder 为我们提供了函数库（RTL /runtime library），类库(VCL)；而 Windows 操作系统(Windows 用 C 写成)，则向我们提供了 API 函数库；库“储存”大量实用的解决各种问题，实现各种功能的“工具”，它们都通过头文件列出清单。

当我们写上#include <stdio.h> 这一行时，我们是在向编译器说明：本文件中的程序需要用到 stdio.h 头文件中所列的某些函数。include 正是包含之意。

printf("Hello,world!");

getchar();

现在看这两句。printft 和 getchar 正是两个在 stdio.h 中列出的函数。如果没有#include <stdio.h> 这一行，编译器便会报错说不认识这两个符号。换句话说：include 语句让编译知道我们要用的库函数在哪一个库里。我们会在下一节“简单程序调试”中做相应试验。

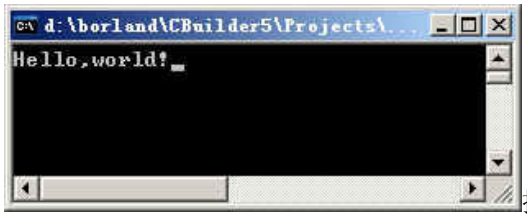
print 是打印之意，而 f 则为 format，二者结合意为“有格式地打印”。我们的 Hello World 不需要任何格式。至于 getchar，从字面上理解是“得到字符”？事实上就是程序会在此处停下来，等待用户输入字符，直到输入一个回车符，程序继续执行。在我们的这个程序里，继续执行的结果是碰上这一行：

return 0;

return 是返回。当主函数 main 返回时，整个程序就结束了。所以，要这个 getchar（）的目的很明显：让程序在结束之前可以等我们一下（因为我们需要看一眼输出的“Hello World”）。

printf 是输出(output)内容,而 getchar()则是等待你输入,从这一点看,我们也可以理解前面 include 的为什么是 stdio.h 了：std 是英文标准的前三个字符，而 I 和 O 分别是 input 和 output 的首字母。

说了很长，但程序运行结果却很简单，这是按 F9 后的运行结果：



按回车，结束。结束后，别忘了保存我们的第二个程序。

2.4 简单程序调试

2.4.1 编译期错误与运行期错误

如果我在上面的程序输出时，打出的字幕不是“Hello,world”，而是“Hello,word”。你一定会说“哎，哥们，程序错了！”。于是，我就要去代码中查找有关输出的那一句，一看，哟，真少写了一个‘l’。将这一错改正，重新运行，检查结果。

这就是一个调试过程，发现错误；查找出错原因；改正；再运行。当然，由于很多时候我们也不是非要等到发现错误了才去检查，有时我们会主动一步步去事先主动检查是否有错。必竟 BUG 总是常见的。程序错误（BUG）表现上各式各样，但若是从其“发作”的时间上分，则可分为“编译期错误”和“运行期错误”。如上面说的错误，属于后者，因为它对编译过程并不产生什么阻碍，编译可以顺得通过。错误在运行中表现出来。（按时间分还有一种为“链接期错误”，这里不说）

我们来故意制造一个编译期的 BUG。

这是上一节的代码：

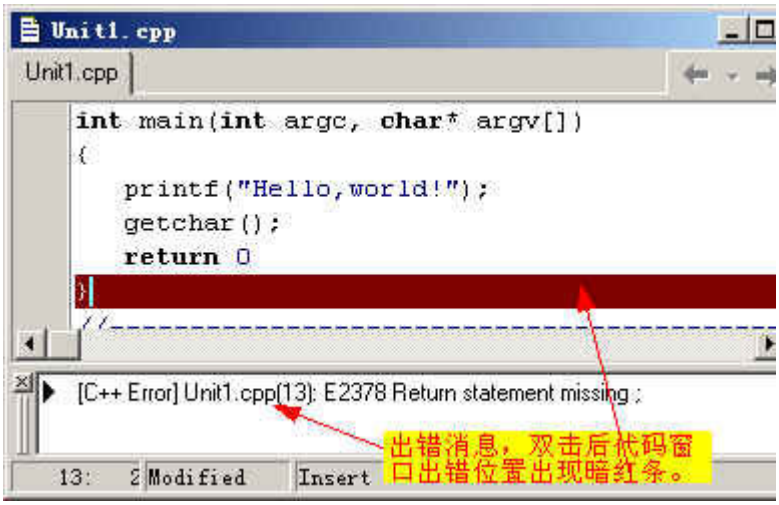
```
int main(int argc, char* argv[])
{
    printf("Hello,world!");     getchar();     return 0;
}
```

现在我们故意将最后一句代码： return 0; 行末的分号删除，结果为：

```
int main(int argc, char* argv[])
{
printf("Hello,world!");          getchar();     return 0 ;
}
```

*（约定：我们用**粗斜体**表示修改过的代码）*

现在按 F9，程序并没有跑起来。但在代码编器窗口下面，出现一消息框（如果你没有发现，请在代码窗口中点鼠标右键，将出现右键菜单，选：Message View）



编译过程是一个相当复杂的过程，在编译之前，代码会被做很多格式上的转换，另一边，人写代码出错的原因也五花八门，所以，想让编译器为判断到底你犯了什么错事实上是很难的。但 CB 的编译出错消息竭尽全力让我们找到出错原因：

[C++ Error]：错误类型。这是在告诉我们，这是一个 C++语言语法方面的错误。编译器说对了，因为我们少写一个分号，确实是犯了语法错误。

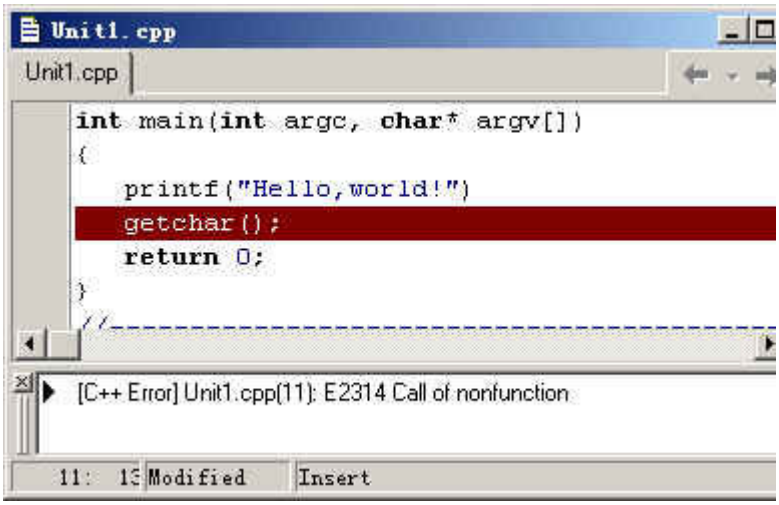
Unit.cpp(13)：错误位置。说是在 Unit.cpp 这个文件内的第 13 行。文件显然是说对了，但行数却稍有偏差，少了分号的那行其实是第 12 行。（从状态栏可以看到当前光标位置的行列数。

E2378：错误编号。CB 对各错误消息加进行了编号。

Return statement missing; 错误消息。很好，这个消息 100%正确：return 语句丢了分号。

并不是每回都能幸运得到这样准确的错误报告消息。比如同样是去掉分号，但这回把 return 0 那一句的分号补上，去掉它

printf(“Hello world!”)这一行的分号，编译出错消息是这样：



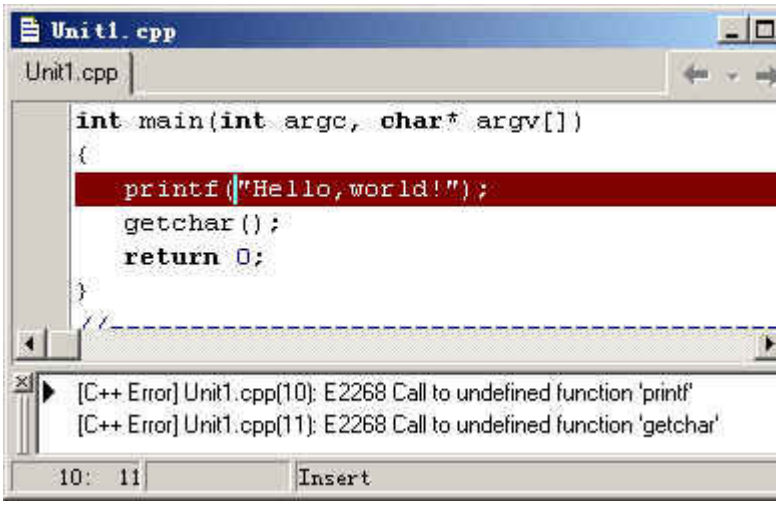
错误消息说我们调用了一个“非函数”，为什么呢？答案是因为没有了分号，编译将两句连成一句：printf(

“Helo,world!”)getchar();这样，它就把 printf(“Helo,world!”)getchar 整个儿认为一个是一个函数名，于是就不认识它了。

下面我们做一个实验，结果报错也同样为“Call of nonfunction”，但原因不是因为我们写错了函数名，而是因为我们使用了库函数，**但没有通过 include 语句来告诉编译器：这些函数在哪个库里。**

```
//---------------------------------------------------------------------------
//#include <stdio.h>
#pragma hdrstop
//---------------------------------------------------------------------------
#pragma argsused
int main(int argc, char* argv[])
{
printf("Hello,world!");
getchar();
return 0;
}
//---------------------------------------------------------------------------
```

代码中，我们故意有 // 将#include <stdio.h> 从一行代码摇身一变成为一行注释。前面说过注释是仅给人看的，编译看不到这一行，相当于你删除了这一行。现在编译这一行，错误如下：



两行同样类型的错误消息。编译一下子不认识 printf 和 getchar 两个函数了。

2.4.2 学会使用帮助文档

经过上一节，同学可能会问三个问题：

第一，“老师，你怎么知道有 printf 和 getchar 这个两个函数呢？”

回答：两个途径：常用的函数会在课程里讲到。其它函数则根据需要在各上文档中查找”。

第二，你怎么会知道 printf 是用来输出而 getchar 则可以用于等待输入？

回答和上面一致，从课程和文档中获得。

第三，就算知道了 printf 这个函数，我又怎么才能知道它在 stdio 这个头文件对应的库里？

回答还是一样。

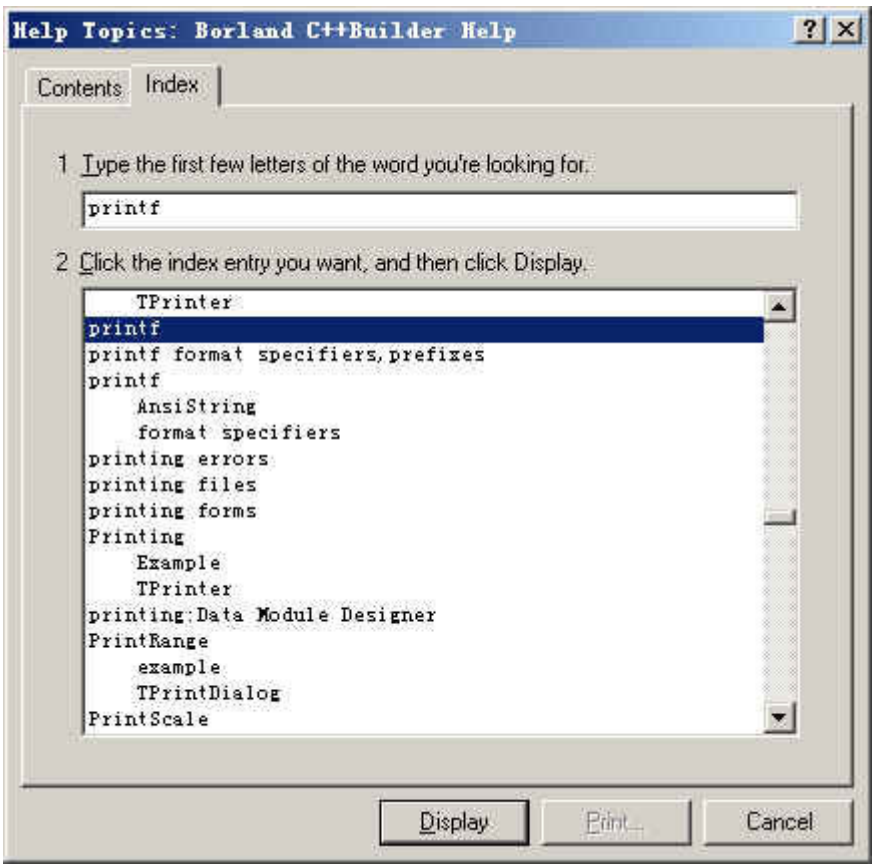
学会使用文档看来很重要。文档有多种，比如有专门的函数大全之类的书，不过现在的文档是帮助文档。初学者在调试程序时，很多人由于英文不佳，于是根本不去动帮助文档。其实当你带些具体的目的去看帮助文档，你会发现就算英文不好，你可以从中获得帮助，解决大部分程序问题。

我们用一节的那个 BUG 来说明问题。假如你用了 printf 和 getchar 这两个函数，但你不知它们包含

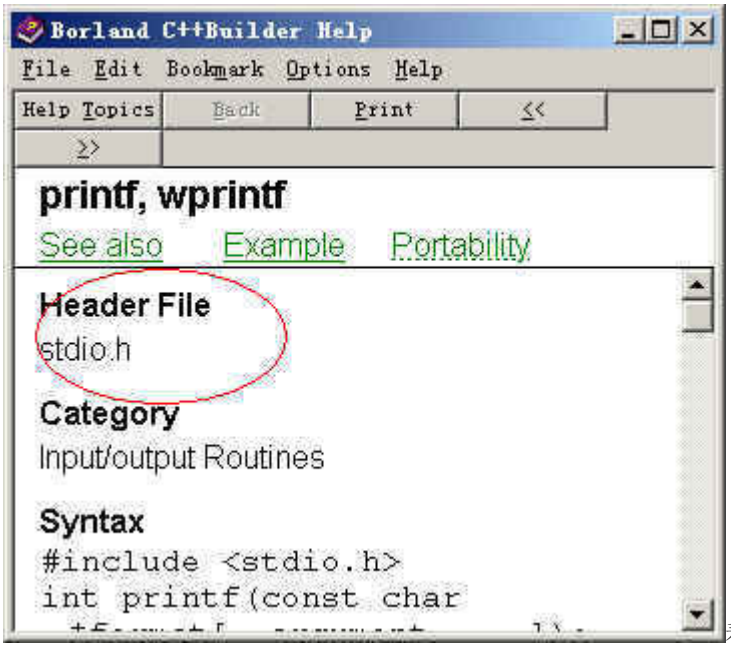
在哪个库里（也就是说你不知道在#include 之后应该写”stdio.h”）。

菜单：Help | C++ Builder Help

出现 Windows 标准的帮助索引窗口,我们在索引内键入：printf。



如此，我们找到了有关 printf 的帮助：



看，红圈内的内容清楚地说明了什么？

（提高英语水平对学习编程的帮助是很大的。如果你需要学习英语，就不妨听我做一句广告：

英语学万句，开口如有神——点击此处查看我们的最新英语学习产品《句神英语 2002》介绍）

在广告的喧哗声中，我们结束了第二章的学习。

第三章 计算机原理

“原理”一词，似乎总是代表艰深难度，另外还多少有些“太过理论，脱离实际”的意味。对于计算机，它的原理还偏向于硬件。基于这些，我们需要明白，在真正开始学习编程之前，我们有何必要学习一些计算机原理呢？

在大学里，计算机系有专门的一门《计算机原理》课，计算机原理的内容并不出现在计算机语言的课中。或许是这个原因，许多面向社会人员（非专业人士）的计算机编程书籍同样不讲计算机原理。这容易造成学习者也许上手很快，但学到一定程度后就难以有较深入的发展。我想，这是因为“底气不足”。

如上面提到的，计算机原理是完整的一门课，我们此处只用一章的时间阐述，所以我们必须讲最本质的原理。

3.1 电唱机、电话、电脑——谈谈模拟信号

要了解电脑的原理，不妨举一些我们早已熟悉的其它电器的例子，看看它们的原理。

假如你第一次看到一台会能说会唱的电脑，你可能会感到新奇，但事实上，别忘了，从 CD 机到带式的随机听，它们都没有生命却“记下”然后“说出”声音。事实上它们运行的原理，在本质和电脑完全一致，只不过由于我们太过熟悉，所以就认为它们没有什么了不起。其实，你能说出电唱机为什么能唱吗？

原始的电唱机会发唱的原理：

一张盘，表面涂一层石蜡。取一根针，针尖正好接触蜡面，针上顶一张薄膜。让帕瓦罗蒂在不远处冲着这张膜唱《我的太阳》。另有一人在老帕高歌时均速地旋转蜡盘。于是，歌声高低不同，薄膜向下压的幅度也不同，针在蜡上刻的深浅便不同，这样，就将人的声音最终以蜡上划痕的深浅记录下来。将蜡盘固化，在一套反方向的装置上：盘转，顶针上下高底不同地拉动一张膜，那张膜就会有模有样地唱《我的太阳》了。

再来看看电话的原理：

话筒内有一堆碳粉，碳粉内埋一导线，碳粉盖一张膜。同样，当你对话筒大喊大叫时，膜对碳粉造成忽紧忽松的压力，碳粉之间时紧时松，引起其电阻的大小变化，最终忽大忽小的电流传到对话的听筒。听筒内有一电磁铁随电脑大小而磁性不同，它对埋有金属丝薄膜时吸时放，薄膜便发出了你的声音。

你我都已经永远地失去了发明的电唱机或电话的机会了……伟大的先驱们是那么的聪明，懂得将一种不便于存储，不便于传播的信号转换为另一种便于存储，便于传播的信号，从而有了伟大的发明。

电脑（计算机）要管理各种信息，首先它必须能存储，传载这些信息，所以，在这一点上，它的本质和电唱机或电话没有区别，必须实现各种信息以某种方法，转换为另一种信息。

这就是计算机的第一条本质原理：将各类信息以某种信号进行存储。

好极了，现在，我们已经是了解计算机本质之一的人，从今天起，当我们再听到有菜鸟在说硬盘时，你大可对其怒喝：“成天就知道‘酷鱼大脚西部数据’！不管是什么牌子，也不管是硬是软是光，它们都是计算机将外界信息以磁或者激光信号的形式进行存储的介质，懂不？看你真像个中关村的电脑贩

子！”（如果对方体积俨然在你的 1.5 倍以上，最后一句可以不说）

3.2 数字信号

“数字”——digit。

无论我说中文还是英文的，除了刚巧学完 0~9 的小学低年级学生外，20 和 21 世纪内出生的人都知道我是在说一个时髦词。

这个“数字”所代表的，是“模拟”的反面。

电视，我们要数字的（尽管据说在中国并没有数字信号的电视节目）；

手机，当然是数字的，前阵子中国电信已彻底地向采用模拟信号的蜂窝说 bye-bye；

相机，数码的。（数码就是数字）

空调，数控的（空调吹不出数字的风，但它说，我这风是在数字信号的控制下吹出来的，当然就白里透红与众不同）。

“数字的”就这么好吗？电脑也是数字的吗？

前面我们说唱机，电话的原理时，你可以看到，在一种信号转换为另一种信号时，采用的方法是进行“模拟”。比如用针在蜡上刻的深浅来模拟声音的高低。尽管新的信号记载原来不空易存储的信号（声音），但这种新信号本身也是不稳定，不精确，比如针的不同，或蜡的质量不同，但会造成虽然同样是老帕在同一时刻唱的歌，不同的盘最后播出的声音却不同。类似的，笔者便常常在电话这头将丈母娘的声音听成是丈母娘女儿的声音而下不了台。

和许多伟大发明一样，当初计算机的发明的需明恐怕也是为了战争。现代计算机更是广泛用于卫星发射，飞机导航等不允许出错的领域。其实，就算是仅仅用于让你给女朋友或男朋友发一封情义绵绵的 e-mail，只怕你也不希望它出什么差错。

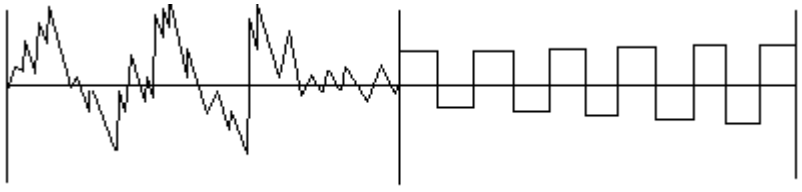
尽管模拟的手段是一种信号转换为另一信号时几乎是不可避免的最初方法，但我们要求有进一步的转换来或得可以精确复制，从而更利于存储，传播的信号。

当你将一首存在磁带式唱片上歌不停地录到别的唱片时，歌声会越来越变调。同样操作进行在数字信号的 CD 唱片，歌声却完全一致。

模拟信号转换成数字信号和其反方向的转换是如何实现，不是一章的文字能说清。也不是我们学习的重点。我们只需记住以下内容：

1、首先：模拟信号英文为：analog signals 缩写 AS;数字信号为：digital signals/DS;因此，前者到后者的转换称为 AD 转换，另一方向则称为 DA 转换。在各种计算机系统（特别是工业控制）中，AD 转换往往是采集数据的初始端，而 DA 则是播放数据的最终端。

2、第二就更简单。以下是两张示意图，分别为模拟和数字信号的波形图。你需要能认出二者。



像心电图一样的锯齿波是模拟信号典型的波样，比如我们的声音。而方形波则为数字信号。

（笔者用画笔毛糙地画成，不像样之处尚望各位包涵。）

面对左图的锯齿波。我们也就明白了为什么人生和生活会那么复杂。因为人类的原始的现实的生活中，各种信号几乎都是模拟的。不仅仅是声音，不仅仅是我们的心跳的规律，更包括我们的感情，都是“模拟信号”。你看“情绪化”一词又总是用来代表不稳定的，非理智的……和生活相比，编程世界是那么

的简单。它 0 就是就 0，1 就是 1，正所谓爱恨分明……似乎有些跑题，不过接来的另一段“面对”倒很重要，请认真看。

面对方形波，回忆第一章我们说过的二进制：二进制数只用 0、1 两个数字。方形波最适合于二进制表示。凸起的用 1 表示，凹下的用 0 表示。所以现在你可明白：在计算机的世界里，所有信息最终都被数字化为 0 和 1，这是一种最彻底的数字化。譬如在我们已熟悉的数据存储方面：硬盘和软盘都为磁盘，它通过有盘上某一点有无磁性来表示 1 或 0；而光盘，它盘面上会有连续的凸起颗粒，和间隔的平面。当激光头的射线进入前者时，光被散射，这些连续的点用于表示一个 0，当光线照在平面区时，光被反射，则可表示 1。（如下图）

（由于这种数字化的实现大多数通过电子电路，所以，一些场合数字化也被称为电子化）

至此，让我们再重复一遍在第一章我们说是严重跑题的那段话：

世界就是这样奇妙。万事万物五彩缤纷，但进了计算机，却只是个 0 和 1 的组合。不由得你会想起道教的古老玄机：“无极生太极，太极生两仪，两仪生四象，四象生八卦，八卦生十六爻”。

3.3 I/O 设备，存储器、处理器

计算机系统中，硬盘、软盘/软驱、光盘/光驱等称为存储器。它们用来存储信息，这些信息在它们的内部统统以 0 和 1 表示。

如果只有这些，那计算机就只能叫数据仓库了。有了数据还需要处理数据的能力。

数据和处理。依笔者的理论，这是程序的全部，计算机的全部，也是世界的全部。反过来说，正因为整个世界都可以用数据和处理来表达，所以最终程序才有可能实现对现实问题的解决。

当我们在写程序时，就是在用程序来表达这个世界。当然，由于很多数据没法实现“数字化”，所以它们永远无法用程序来表达，比如人类的感情，或许永远只能是“模拟信号”。(基于此，所有科幻片中关于有一天计算机会突然具有自己的意识，并开始以人类为敌的设想，我们可以断定它也就只能是幻想——永远都是。)

计算机系统中，键盘，鼠标，扫描仪，数码相机等，可称为输入(Input)设备。

显示器，打印机，称为输出(Output)设备。

这二者统称为输入输出设备，也就是计算机英文中常见的那个缩写：IO 或 I/O。

输入设备用来做什么？用来向计算机输入信息，这过程便有非常多的“模拟->数字”转换器。而输出设备，则将**处理后**的的信息以合适的格式输出（一般是为了输出给人看）。下面我们用我们最熟悉不过的鼠标来说明。

第一，鼠标的输入端是什么呢？

答：是我们手里握着的那个类似老鼠的东西。

第二，鼠标的输入端要处理的是什么信息呢（换一种问法是：鼠标要输入什么信息）？

答：是我们胳膊肘的来回挪动（这里暂不说单击，双击等）。可别说胳膊肘的动作不是信息——那样说可真外行——包括挪动的方向，距离，速度等。

第三，胳膊肘的动作是“模拟信号”还是“数字信号”呢？

答：只要你的手是肉长的，那么就只能是模拟信号。

第四，胳膊肘的动作是如何被采集，又如何传输入进电脑，又如何被处理，又如何变成一个光标在屏幕上跑来跑去，有时候还会变成一只小手……

答：@ # ￥ % # ？！

关于鼠标的具体工作过程已经不是我能回答的了，也不是我们要学习的内容。不过如你很穷，和我一样用的是 10 来块的机械式鼠标，那么恭喜你，你可以亲自“解剖”一下鼠标，观察鼠标里头的“模数转换器”。方法是把鼠标背过来，揭掉合格证（提醒，揭掉后你的鼠标可能无法保修了），拧掉螺丝，打开上盖，会发现内有滚轮，水平向滚轴，垂直向滚轴，辅助压轮各一，组成一套采集设置，看看你就明白它们是如何配合工作，完成采集你胳膊肘的挪动的信息了。

最后在合上盖时，顺便将滚轴上的积泥刮掉，它们严重影响数据采集的精度。

说完存储设备和 I/O 设备，重要人物也要该出场了。它就是电脑的心脏：CPU。

CPU 何许人也？Central Processor Unit。即：中央处理器。中央并不是说它正好在机箱内正中间，而是说它是核心人物，其实你显卡声卡等也有芯片在处理一些数据。但大都数数据，比如鼠标采集到信息后，便需要送到 CPU 中进行复杂的计算，最终才能输出。

CPU 便是这样一个人物，它要处理几乎所有计算系统中的数据。它的重要性得就像是大脑之于人体。把它说成是心脏真是个混淆视听的比喻。

CPU 又是如何处理数据的呢？大千世界中的数据（当我们偏向于专业时，我们就将信息说成是数据）各类各样，极其复杂；同样的，对种种数据的处理也相当复杂。比如给你一个苹果你的处理是吃掉，而女友把她的手给你时她的意思是要你牵着，如果你把后者等同于前者进行同样的处理……后果……

Intel 或 AMD 生产的 CPU 如何先进，终究是个东西，怎能自己决定如何处理各种数据呢？

有数据：钢板拴着一个螺丝钉，

有处理数据的能力：工具箱中一把螺丝刀。

一只狗和一只猪从二者前面走过，它们不知用后者把前者拧下。因为马克思说了，只有人类才会制造和利用工具。

CPU 也只是一个工具。尽管它有处理各数据的能力，但必须由人来控制它：什么时候，什么方法，计算什么样的数据。这样的工具并不仅有 CPU，早在你我童年时爱不释手的，会自己摇摆走路的玩具小鸭内，那个发条就是这种工具。通过既定的设计，发条具有把人拧紧的能量存储，然后释放，一点点控制其它齿轮，小鸭的脚，最终让玩具小鸭如人所愿地走。

发条处理数据的动作很简单，只须一点展开就行。但是如果没有人事先将其拧紧，它一样动不了。CPU 要处理的数据复杂，处理的方法更复杂。同样，必须有**人**事先将**计算机处理数据的方法**存储在上述的存储器上，在要开始处理时，装上这些方法，然后开始执行。

一切重要概念至此呼之欲出：

人不是普通的人，是伟大的程序员（当然当然，各行各业除了中国电信以外的从事者都很伟大）；

计算机处理数据的方法，便是：程序！程序！！程序！！

《超级解霸》是什么？是音频视频播放软件，但归根是处理多媒体数据的程序。《金山毒霸》是什么？是杀毒软件，但归根是对付病毒数据的程序；《句神英语 2002》是什么？是英语教育软件，但归根是辅助你自学英语口语的程序。Windows 或 Linux 是什么？是操作系统软件，不过其实它们也是程序：管理所有其它程序的程序。

下一节，我们讲计算机内存，内存也是存储器，但它是一种特殊的存储器。

3.4 内存

有外存吗？软盘，光盘这些放在外面的存储器就是外存啊——真不明白这算是什么知识，但各种考试似乎兴趣这些。硬盘有可以放里头的也有可以放外头的，不知怎么算。

对内存要弄明白的第一件是事是：为什么要有内存？听我做一个不负任何责任的预言：10 年之内，高

速硬盘和高速外部总线的发展，但得计算机可在硬盘上固定划块分区作为内存。如此这般，以后关机时不用使用 Win2000 或 XP 的休眠功能，更不必像普通关机那样听硬盘卡卡响半天，一个关机命下，机器迅速关掉，妙哉。

程序和数据平常存储在硬盘等存储器上，不管你开机或关机了，它们都是存在的，不会丢失。硬盘可以存储的东西很多，但其传输数据的速度较慢。所以需要运行程序或打开数据时，这些数据必须从硬盘等存储器上先传到另一种容量小但速度快得多的存储器，之后才送入 CPU 进行执行处理。这中间的存储器就是内存。

无论何种存储器，软盘、硬盘、光盘或者内存，都有地址。因为它们要存储数据，所以就必须按一定的单位的数据分配一个地址。有了地址，程序才能找到这些数据。这很好理解，想想你们家为什么要有门牌号即可。

学习编程，必须对内存的地址有一个透彻的理解。我们编程中的每一行代码，代码中用到的每个数据，都需要在内存上有其映射地址。当然，我们并不需要掌握内存是如何进行编址，那是计算机系中的另外一门课：操作系统的事了。

下面，我将旧课程中的有关内存的一段内存拷到这里。

3.4.1 内存地址

“你叫丁小明吧？”

“是的。”

“噢，你记得你的姓名，那么请告诉我你把你的姓名记在你的脑海中的哪一个位置呢？在你记着你的姓名信息的下一个位置，你记着什么？上一个位置呢？”

“啊！这我不知道。”

“你骗我！既然你记着你的姓名，你怎么会不知道把它记在哪儿呢？”计算机生气说。

是啊，依靠现在的科技力量，我们无法得知自己把一个数据记在脑海里的哪个脑细胞里。这也是人的记忆状态无法数据化的原因。计算机就不一样的，我们说过，它是什么都数字化了。所以它知道自己把一个数据，一条命令记到了内存中的哪个（些）位置。每一个位置都有编号，就像编了门牌号一样。如果让计算机在内存里记住“丁小明”这个名字，可以示意为：

丁		小		明	
1000H	1001H	1002H	1003H	1004H	1005H

在第一行中，每一格表示一段内存，而格子里的内容是这段内容记下的数据；第二行中每一格内数字就是对应的内存的地址。至于为什么数字后面跟了一个字母 H ，那是为了表示这是一个 16 进制的数。什么是 16 进制的数，大家现在可以不管。只要把它想成和上一节我们所讲的 2 进制一样即可：长有 16 个指头的人订出来的数。从 0 一直数到 15，到了 16 才往高位进 1。

可能有人会啄磨：为什么一个“丁”字（“小”“明”两字也一样）占用两个内存地址呢？这是因为汉字在一个地址（位置）里呆不下，必须放在**连续**的两个地址空间内。

那么，什么东西可以放在单独的一个内存地址里呢？像英文的里字母，比如’A’，像阿拉伯数字：比如’1’,可以，而且就是放在一个内存地址里。假设有一字符串“ABC”，被记在内存里，可示意为(这次我们假设从内存地址 2000H 处记起)：

A	B	C
2000H	2001H	2002H

现在我们提几个问题：

计算机记住”丁”字的内存地址是多少？ 答案是：1000H。请见上图

在计算机记住”丁”字的内存地址后移两个最小内存地址单位，计算机记住的是哪个字：答案是：”

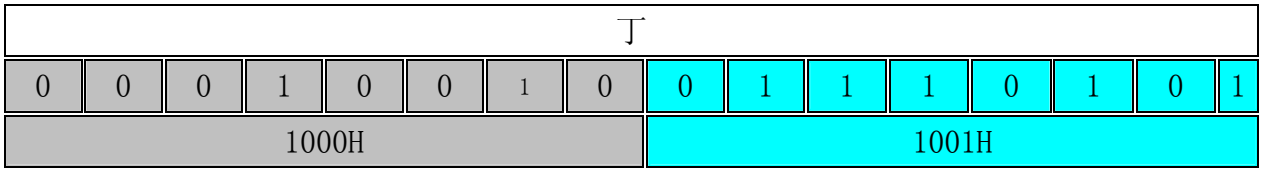
小”。因为 1000H+2=1002H。

请同学自己对图 1.3 作类似的问答。

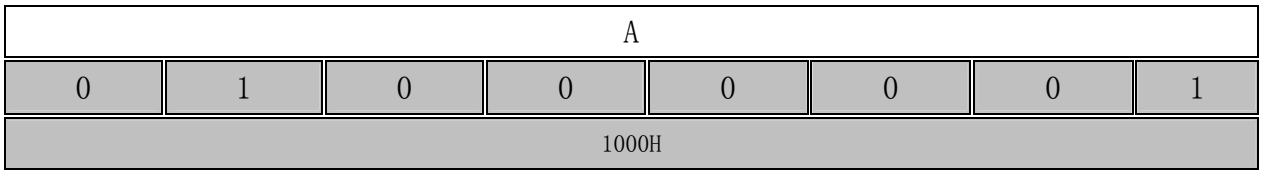
如果不再往下讲一点点，可能会使关于内存地址的这一讲的有些内容和第 1 章中有关 2 进制语言的内容看起有矛盾。

我们一直在说，在计算机中，所有信息都被数字化为 2 进制的 0、1，所以，“丁小明”这个名字被也应该是一串：0001 0010 0111 0101……,可是在中图所画出的，计算机内存里记的，仅是“丁小明”三个字啊。

下面是解释，我们只举一个字”丁”讲解。我们假设在那一串里的 0001 0010 0111 0101 对应的是“丁”字，那么有：



让我们把字母’A’对应的图也画出来：



在上面的两个图中：

第一行分别是“丁”和“A”，它是给人看的。

第二行则是一串的的 0 和 1，这才是计算机内存中实际存储的数据。

第三行是内存的地址。并不是每个 0 和 1 所占的位置都被编上地址。而是每 8 个才拥有一个地址。

关于第三行，你可以这样理解，门牌号是一个家庭分配一个，每家每户内还有客厅卧室，这些就没有地址了。

可见：

‘丁’的确是由一串 0、1 组成的。更确切地，从图上可以看出‘丁’是由 16 **位** 0 和 1 组成。这 16 数都存放在 2 个内存地址里。

‘A’也一样，它是由 8 **位** 0、1 组成的。占 1 个内存地址。

位： 一个 0 或 1 称为一位（bit）；

字节：连续八位称为一个字节(Byte)；字节是计算机中可单独处理的最小单位。

用上的两个单位来表达上面的图,便是:

汉字如“丁”，在内存中，占两个字节(Byte)，共 16 位(Bit)。

英文字母如 ‘A’在内存中，占 1 个字节，8 位。

（这里说的是内存，其实在其它存储器中，所占空间是一致的）

公式 ：1 字节(Byte) = 8 位(bit)

3.4.2 虚拟内存

程序和数据必须装入内存，这就必须面对内存不足的问题。这一问题有许多解决措施，其中很重要的措施就是使用虚拟内存。而所谓的虚拟内存，其实就是硬盘。

打开一个 Word 写文章，再打开几个 IE 浏览网页；后台还有 FlashGet 在默默地为你下载网上文件。很快你就会觉得的机器反应变得慢了。为什么？因为程序本身和程序所使用的数据太大，物理内存（真实内存）已经不足，系统采用了大量的硬盘空间来模拟内存。上节说过，硬盘虽大，但其（传输、查找）速度比物理内存慢一个数量级，所以整个系统速度就变慢了。

在 DOS 的编程时代，程序必须自己实现虚拟内存，或者采用第三方程序。而在 Windows 时代，虚拟内存机制由操作系统来实现。所以在本节，我们只需知道，虽然理论上程序在运行时必须装载入内存，但这内存并不一定全是真正的内存，很大一部分，其实是在使用虚拟内存。当然，在 Windows 下，程序员无须去考虑自己的程序什么时候使用物理内存，什么时候使用虚拟内存。

操作系统设置了最大可以使用多少虚拟内存？现在正在使用多少虚拟内存，这一些都有工具可以查看到。现在让我们用最方便的方法来看看第一个问题。

Windows95/98/Me：

请在您的电脑桌面上找到“我的电脑”图标，点击鼠标右键，在弹出的右键菜单里选“属性”，然后在弹出的“系统属性”对话框里选“性能”这一页，就可以看到右下角的“虚拟内存”按钮，点击后便可看到 Windows 对我们所讲的“虚拟内存”的设置。

Windows XP/2000:

在开始菜单中找到“我的电脑”，点击鼠标右键，在弹出的右键菜单里选“属性”，然后在弹出的“系统属性”对话框里选“高级”选项页，点其中“性能”组内的“设置”按钮。出现“性能选项”对话框。选“高级”选项页，底部有“虚拟内存”组。你可以从中看到有多少 MB 的硬盘空间允许 Windows 拿来当虚拟内存。如何想修改或查看更多信息，可点“更改”按钮。建议采用系统默认值，不要修改。

第四章 数据类型

4.1 这是个有类型的世界

问大家一个问题：

现实生活，有哪些信息可以用计算机可以管理呢？

职工、学员、客户、工资、原材料、产品、商品……现实中实现用计算机管理的信息已是无数。

职工又有什么信息呢？

职工有姓名、性别、出生年月、家庭住址、电话，婚否、工龄、工种、工资、等等。

这两个问题你可能回答得不错，现在，考验你前几章有没有认真学习的时刻到了，请看下面这个问题：

所有的这些信息，在计算机里都是以什么样的数据形式来表达呢？请自觉闭上眼睛，想一想。再看以下的各种回答。

“二进数”，正确。

“已数字化的数据”，也算正确。

“0 和 1”，正确。

“机器语言”，正确，你还记得第一章的内容啊，不错。

本章需要继续的一个问题就由此开始。所有的信息都用机器语言——那些 0 和 1——表达，那你我编写程序岂不很难？

这是第一章的问题，你还记得在第一章关于本问题回答吗？那就是：机器语言不好记，那就用高级语言。高级语言高级在哪里呢？第一章也有答案：就高级在它尽量向“高级动物”的思维习惯做了一些接近。当然，只能说尽量，它必须仍然保持符合机器的绝大部分特点，否则，大家就不要学**计算机**语言了。

数据类型，就是计算机语言向人类语言靠近时，走出的第一步，很重要的一步。（机器语言或汇编语里，没有数据类型一说。）

人类的世界，是有类型的世界。

树木花草，归一类：植物；

猪狗猫羊，动物；

金银铜铁，金属；

你我他她，人类。（不要告诉我你不是人类，在这虚拟的网络的世界……）

上一章笔者“鼓吹”过一个观点：**整个世界都可以用数据和处理来表达。**基于此，整个世界就是一个程序；而万物是世界的数据。如果你找一个人，对他说：“你等于一只猪”，他一定暴跳如雷。为什么呢？嘻嘻，学了这一章，我们就可以从程序的角度来解释了：人和猪不是一类型，不适于做赋值操作。待以后我们学了 C++的“类”，则又有更好回答：上帝创造世界是，没有为人“类”实现参数为猪“类”的拷贝构造函数，或等号重载函数。

（想和上帝做同行？快做个程序员。）

4.2 数据类型基本概念

4.2.1 理解数据类型

“数据类型”就是这么一个很好理解的概念。我们的重点是了解在计算机编程世界中，有哪些基本的数据类型？在人类世界里，数据类型那就多了，把人类的对万物划分类型的方法照搬入计算机世界，显然不可能。怎么办呢？方法就是：抽象。

计算机先哲们为我们做了这一切。其中，最重要类型，也称为 C/C++语言的**基本数据类型**，只有两个：“数值”和“字符”。

第一是“数值类型”。这样，在职工的信息中，譬如年纪，工龄、工资就有了归属。你现在需要暂时不将目光从屏幕前移开，想一想“数值类型”是一个多好的抽象结果！无论进行任何信息管理，离开了“数”，还能管理什么？

第二是“字符类型”。像职工姓名或家庭住址，这些由字符类型的数据组成。你可能想不到的还有，职工的电话号码 010-1234567，这也是由字符类型数据组成。0、1、2、3 不是数字吗？应该属于上面的“数值”类型吧？嗯，你听我说：现实生活中，数字被广泛地应用在两种不同范畴：其一是那些典型的，需要进行计算的场合。比如鸡蛋 1 斤 2.3 元；其二则那些只用来表示符号的范畴。比如电话号码，比如车牌号。把两个电话号码进行相加或相减的操作是没有意义的。基于数字的两种完全不一样的使用范畴，所以在被抽象到计算机程序语言时，数字被分到“数值”和“字符”两种类型中。但字符类型并不只有阿拉伯数字，键盘上的**字**母**符**号，都属于**字符**类型。

在 C/C++中，“字符类型”其实也可归入数值类型。在某些情况下，它仍然会被用来参与计算。比如在计算机中，字符‘A’加 1 后，会得到下一个字符‘B’，这种操作符合我们的习惯。

4.2.2 理解整型和实型

数值类型又被分为“整型”和“实型”。整型就是不带小数位的数，而实型则是指带小数位的数，也称为“浮点数”。我们在生活中一般并不做如此区分。譬如说鸡蛋的价格罢，今儿便宜了，一斤 2 块整，但这不防碍明儿价涨了，我们说成：涨了，2.30 元一斤。在编程时可不能这样，你必须事先考虑到鸡蛋的价格是必须带小数的，就算现在是 2 块整，也得将价格定为实型，否则，涨成 2 块 3 时，计算机会把那 0.3 元给丢了，还是变成 2 元整。

你会问为什么不直接就定一个实型就好呢？实型不就包含了整型的值吗（如 2.0 等于 2 元）？好！问得好，我不喜欢书上说什么就记什么，一个“为什么”也不去想的学习方法。由于能力的限制，大多数人如你我，都不能为这个世界发明奉献什么新技术，我们只能作为技术的“接受者”而活在这个世上。然而正因为只能作为接受者，我们就更应该在学习任何技术前，先学会从一定的高度来询问：为什么会有这个技术（这个技术为什么会出现）？我现在需要这个技术吗（必竟学习需要占用时间）？这么做并非是一种消极态度，相反，它的积极意义在于：经过这种思考，你不仅可以把有限的时间花在值的学习

上，而且只有弄明白了一样技术（或技术的概念）产生的需要，我们才有可能真正理解它，从而比别人学得更深刻更本质。

一个问题牵出一堆废话，回答却很简单：因为在计算机里，对整数的运算要远远快于对浮点数的运算，所以，在人类的思维习惯与计算机特点之间，必须做一个折衷。

折衷的结果就是虽然都是数值类型，但程序员在程序时，必须事先想好，哪些数据是必须带小数计算的，那些数据定为实型，比如工资，没有哪个员工会认为老板可以将自己工资中的小数位忽略不计；而那些可以不用，或者可以不计较小数的数值，则被建议设计成整型，比如人的年纪，虽然可以，但我们很少需要有类似 1.6 岁这种说法。想想，假如一位女士说自己 28 岁，而你却立刻纠正她：“不，准确地说，您已经 28.7 岁了”……结局会怎样呢？为了那 0.7 岁，不仅你的程序会跑得慢，而且会倍受女人的白眼杀伤。

想想其实也很自然：工资必须设计成实型，而年龄虽然不是必须，但建议设计成整型。

4.2.3 理解数值的范围

然而事情并未就此结束。

顾及计算机的运行速度，人（程序员）做出了妥协，必须面对“整型”或“实型”的考虑。另一方面，和速度同样重要的是计算机的空间的考虑。它让程序员必须再做一步妥协。

小时候你一定有过这样的经历：你说 100，另一个小伙伴就说 101。你说 1000，他就说 1001，你说 10000，他就说：那我 10001。总之他就是要比你大 1。这种游戏不会有分出胜负的结局，只会让你郁闷为什么就不能有个最大数让你说了以后，那家伙就再了无法往上加 1 了！现在你学编程，儿时的“妄想”终于在计算机的世界中实现了。在计算机世界中，你可以说一个数，当别人再住这个数加 1 时，真不幸，计算机会告诉他说：加 1 是加 1 了，可是结果变成 0，甚至是负数。

这就是计算机的空间问题：任何一个量，都有一个大的上限，和小的下限，出了这个范围（比上限还大，比下限还小），就会称为**溢出**。这是一种物理的现实，也是一种人为的规定。为什么要这样规定？原因是计算机的存储器，特别是其中很重要的内存（见上章），其可存储的数据多少总是有限度。（而且，同样大小数，2 进制的表达形式比 10 进制长得多），如果允许编程像生活中一样任何一个数都可以很大很大，也就是这个量表达起来将很长很长，那么存储器的空间就会很快用完！（无穷大就不用说了，因为不可能有无穷大的存储器）。

就是这样，数值类型在被划分为整型和实型后，将根据所**占用的空间**而被继续划分为几种不同类型。而我们，在考虑工资必须设计成带小数的实型后，必须再面对一个设计上的考虑，工资这个量在程序里要占多大空间？依据其实很简单，就看你的单位最高月薪是多少，凭此找到一个合适的类型。比如月薪不超过 1 万元，那我们选择一个刚刚可以包含 10000 的数据类型。

两种基础类型：数值类型和字符类型，前者说了很多，现在我们也来对字符类型做一些附加说明。字符类型要比数值类型简单得多：它不能带小数，所以没有整型实型之说。它占用 1 个字节，已经是计算机能单独处理的最小空间单位，所以也不存在继续分为不同空间长度的问题。因此，我们将以它为例，详细说明有关数据类型的一些基本而重要的概念。

第 1、由于计算机和编程都是老外的发明，而老外生活中常用的的字符并不多——主要是阿拉伯数字、英文字母、标点符号等——所以字符的宽度被定为 1 个**字节**（如果忘了什么叫字节，请看上章）。

1 字节 = 8 位，所以它能表示的最大数当然是 8 位都是 1（既然 2 进制的数只能是 0 或 1,如果是我们常见的 10 进制，那就 8 位都为 9，这样说，你该懂了？）。

1 字节的二进制数中，最大的数：11111111。

这个数的大小是多少呢？让我们来把它转换为十进制数。

无论是什么进制，都是左边是高位，右边是低位。第一章中我们说过，10 进制数的最低位（个）位的权值是 10^0，第二位是 10^1，第三位是 10^2……，用小学课本上的说法就是：个位上的数表示几个 1，十位上的数表示向个 10，百位上的数表示几个 100……

二进制数则是：第 1 位数表示几个 1 （2^0），第 2 位数表示几个 2（2^1），第 3 位数表示几个 4（2^2），第 4 位数表示向个 8(2^3)……

在 C/C++中，很多计数的习惯都是从 0 开始，所以，在你看明白上面那行内容后，让我们立刻改口换成下面的说法，以后我们只用这种说法：

二进制数：第 **0** 位数表示几个 1 （2^0），第 **1** 位数表示几个 2（2^1），第 **3** 位数表示几个 4（2^2），第 **4** 位数表示向个 8(2^3)……

按照这种说法，我们可以发现，从右向左数，第 n 位数的权值 = 2 的 n 次方。

二进制各位权值的计算方法： 第 n 位权值 = 2^n

下表详细地表示 2 进制数：11111111 是如何逐位计算，累加得到 10 进制的值：

第几位	7	6	5	4	3	2	1	0	合计
权值	2^7=128	2^6=64	2^5=32	2^4=16	2^3=8	2^2=4	2^1=2	2^0=1	
2 进制	1	1	1	1	1	1	1	1	
10 进制	128	64	32	16	8	4	2	1	255

上表表示了这么一个计算过程(*表示乘号)：

$1 * 2^7 + 1 * 2^6 + 1 * 2^5 + 1 * 2^4 + 1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1* 2^0 = 255$

（顺便说一句，如果你忘了 2^0 等于多少有点迟疑，请复习一下初中的数学知识：任何数的 0 次方都等于 1）

结果是：

11111111(b) = 255 (d)

（为不了互相混淆，我们在书中常用(b)来表示前面的数是 2 进制的，而(d)则表示该数是 10 进制数。同样地，另有 8 进制数用(o)表示，16 进制用(h)表示。不过记住了，这只是在书中使用，在程序中，另有一套表示方法。）

以前我们知道 1 个字节有 8 位，现在通过计算,我们又得知：1 个字节可以表达的最大的数是 255，也就是说表示 0~255 这 256 个数。

那么两个字节（双字节数）呢？双字节共 16 位。 1111111111111111，这个数并不大，但长得有点眼晕，从现在起，我们要学会这样来表达二制数：

1111 1111 1111 1111,即每 4 位隔一空格。

双字节数最大值为：

$1 * 2^{15} + 1 *2^{14} + 1* 2^{13} + 1 * 2^{12} + 1 * 2^{11} + 1 * 2^{10} + \cdots\cdots + 1 * 2^2 + 1 * 2^1 + 1* 2^0 = 65535$

很自然，我们可以想到，一种数据类型允许的最大值，和它的位数有关。具体的计算方法方法是，如果它有 n 位，那么最大值就是：

n 位二进制数的最大值：$1 * 2^{(n-1)} + 1 * 2^{(n-2)} + ... + 1 * 2^0$

任何一种基本数据类型，都有其范围。比如字符类型，它的最大值是 255,那么，当一个数在其类型的范围已经是最大值时，如果再往上加 1，就会照成“溢出”。

其实，有限定的范围的数量，并不只在计算机中出现。钟表就是一个例子。10 点再加 1 点是 11 点，再加 1 点是 12 点，可是再加 1 点，就又回到 1 点。再如汽车的行程表，假设最多只能显示 99999 公里，当达到最高值后继续行驶，行程表就会显示为 00000 公里。

4.2.4 理解有符号数和无符号数

回头看上一节，我们所讲的数都是正数。同样是年纪和工资，前者不需要有负值，但后者可能需要——至少所有的老板都这样认为。

那么，负数在计算机中如何表示呢？

这一点，你可能听过两种不同的回答。

一种是教科书，它会告诉你：计算机用“补码”表示负数。可是有关“补码”的概念一说就得一节课，这一些我们需要在第 6 章中用一章的篇幅讲 2 进制的一切。再者，用“补码”表示负数，其实一种公式，公式的作用在于告诉你，想得问题的答案，应该如何计算。却并没有告诉你为什么用这个公式就可以和答案？

另一种是一些程序员告诉你的：用二进制数的最高位表示符号，最高位是 0，表示正数，最高位是 1，表示负数。这种说法本身没错，可是如果没有下文，那么它就是错的。至少它不能解释，为什么字符类型的-1 用二进制表示是“1111 1111”(16 进制为 FF)；而不是我们更能理解的“1000 0001”。（为什么说后者更好理解呢？因为既然说最高位是 1 时表示负数，那 1000 0001 不是正好是-1 吗？）。

让我们从头说起。

1、你自己决定是否需要有正负。

就像我们必须决定某个量使用整数还是实数，使用多大的范围数一样，我们必须自己决定某个量是否需要正负。如果这个量不会有负值，那么我们可以定它为带正负的类型。

在计算机中，可以区分正负的类型，称为有符类型，无正负的类型（只有正值），称为无符类型。

数值类型分为整型或实型，其中整型又分为无符类型或有符类型，而实型则只有符类型。

字符类型也分为有符和无符类型。

比如有两个量，年龄和库存，我们可以定前者为无符的字符类型，后者定为有符的整数类型。

2、使用二制数中的最高位表示正负。

首先得知道最高位是哪一位？1 个字节的类型，如字符类型，最高位是第 7 位，2 个字节的数，最高位是第 15 位，4 个字节的数，最高位是第 31 位。不同长度的数值类型，其最高位也就不同，但总是最左边的那位（如下示意）。字符类型固定是 1 个字节，所以最高位总是第 7 位。

(红色为最高位)

单字节数： 1111 1111

双字节数： 1111 1111 1111 1111

四字节数： 1111 1111 1111 1111 1111 1111 1111 1111

当我们指定一个数量是无符号类型时，那么其最高位的 1 或 0，和其它位一样，用来表示该数的大小。

当我们指定一个数量是无符号类型时，此时，最高数称为“符号位”。为 1 时，表示该数为负值，为 0 时表示为正值。

3、无符号数和有符号数的范围区别。

无符号数中，所有的位都用于直接表示该值的大小。有符号数中最高位用于表示正负，所以，当为正值时，该数的最大值就会变小。我们举一个字节的数值对比：

无符号数： 1111 1111 值：255 $1*2^7+1*2^6+1*2^5+1*2^4+1*2^3+1*2^2+1*2^1+1*2^0$

有符号数： 0111 1111 值：127 $1*2^6+1*2^5+1*2^4+1*2^3+1*2^2+1*2^1+1*2^0$

同样是一个字节，无符号数的最大值是 255，而有符号数的最大值是 127。原因是有符号数中的最高位被挪去表示符号了。并且，我们知道，最高位的权值也是最高的(对于 1 字节数来说是 2 的 7 次方=128)，所以仅仅少于一位，最大值一下子减半。

不过，有符号数的长处是它可以表示负数。因此，虽然它的在最大值缩水了，却在负值的方向出现了伸展。我们仍一个字节的数值对比：

无符号数： 0 ---------- 255

有符号数： -128 ---------- 0 ---------- 127

同样是一个字节，无符号的最小值是 0 ，而有符号数的最小值是-128。所以二者能表达的不同的数值的个数都一样是 256 个。只不过前者表达的是 0 到 255 这 256 个数，后者表达的是-128 到+127 这 256 个数。

一个有符号的数据类型的最小值是如何计算出来的呢？

有符号的数据类型的最大值的计算方法完全和无符号一样，只不过它少了一个最高位（见第 3 点）。

但在负值范围内，数值的计算方法**不能直接使用** $1* 2^6 + 1* 2^5$ 的公式进行转换。在计算机中，负数除为最高位为 1 以外，还采用**补码**形式进行表达。所以在计算其值前，需要对补码进行还原。这些内容我们将在第六章中的二进制知识中统一学习。

这里，先直观地看一眼补码的形式：

以我们原有的数学经验，在 10 进制中：1 表示正 1，而加上负号：-1 表示和 1 相对的负值。

那么，我们会很容易认为在 2 进制中（1 个字节）： 0000 0001 表示正 1，则高位为 1 后：1000 0001 应该表示-1。

然而，事实上计算机中的规定有些相反，请看下表：

二进制值（1 字节）	十进制值
1000 0000	-128
1000 0001	-127
1000 0010	-126
1000 0011	-125
...	...
1111 1110	-2
1111 1111	-1

首先我们看到，从-1 到-128，其二进制的最高位都是 1（表中标为红色），正如我们前面的学。

然后我们有些奇怪地发现，1000 0000 并没有拿来表示 -0；而 1000 0001 也不是拿来直观地表示-1。事实上，-1 用 1111 1111 来表示。

怎么理解这个问题呢？先得问一句是-1 大还是-128 大？

当然是 -1 大。-1 是最大的负整数。以此对应，计算机中无论是字符类型，或者是整数类型，也无论这个整数是几个字节。它都用全 1 来表示 -1。比如一个字节的数值中：1111 1111 表示-1，那么，1111 1111 - 1 是什么呢？和现实中的计算结果完全一致。1111 1111 - 1 = 1111 1110，而 1111 1110 就是-2。这样一直减下去，当减到只剩最高位用于表示符号的 1 以外，其它低位全为 0 时，就是最小的负值了，在一字节中，最小的负值是 1000 0000，也就是-128。

我们以-1 为例，来看看不同字节数的整数中，如何表达-1 这个数：

字节数	二进制值	十进制值
单字节数	1111 1111	-1
双字节数	1111 1111 1111 1111	-1
四字节数	1111 1111 1111 1111 1111 1111 1111 1111	-1

可能有同学这时会混了：为什么 1111 1111 有时表示 255，有时又表示-1？所以我再强调一下本节前面所说的第 2 点：你自己决定一个数是有符号还是无符号的。写程序时，指定一个量是有符号的，那么当这个量的二进制各位上都是 1 时，它表示的数就是-1；相反，如果事选声明这个量是无符号的，此时它表示的就是该量允许的最大值，对于一个字节的数来说，最大值就是 255。

这一节课，看似罗嗦，但我希望每位没有编程基础，或者以前对进制，对负值、补码、反码等概念，对数据类型理解不透彻的学员，都能多花时间反复阅读，直到看懂得文中的每一张图表的意思为止。如果有困难，请发信到问答处的专门信箱：wenda@bcbschool.com （为了方便我的查阅，请无关课程的问答或其它来信，不要发到这个邮址，谢谢）。

4.3 字符集和保留字

4.3.1 字符集

字符集和保留并不专属于“数据类型”的基础知识。它是一门语言最基础的东西。就像字母 A-Z 对于英语的作用一样。我把它放到这里，更多的是因为这是我们第一次要碰到它，在下一节，马上就要用了。幸好， 它的难度和学会 26 个字母差多少。

每种语言都使用-组字符来构造有意义的语句，组成 C++程序的，最终的是以下这些字符（空格这个字符不好表示，就直接写上“空格”两字了，以后同）：

26 个字母的大小写形式：ABCDEFGHIJKLMNOPQRSTUVWXYZ,abcdefghijklmnopqrst；

10 个阿拉伯数字：0123456789；

其它符号：+ - * / = , . _ : ; ? \ ″ ’ ~ | ! # % & ( ) [ ] { } ^ 〈 〉（空格）

其它符号，包括汉字，则可能作为程序中字符串的内容，用于显示等。

最后，C/C++语言是区分大小的语言，也就是说 ABC 和 abc 并不相同。这一点我们将在下一章特别强调。

4.3.2 保留字

保留字也称关键字。它是预先定义好的标识符，这些标识符必须保留给 C++语言自身专用。因为它们用来在编译过程中表示特殊的含义。比如，我们想定义一个量为整数类型，那么 C++就必须有一个词来表示什么是整数类型，这个词就是一个关键字。

C，C++主要的关键字，我们在章末附表列出。下面先列出本章要用的关键字。

char ：字符类型

int ： 整型

float ：单精度实型（浮点型）

double ： 双精度实型

unsigned : 无符号类型

signed : 有符号类型

bool : 布尔类型

true : 布尔类型的真值

false : 布尔类型的假值

void ： 无类型

sizeof : 取得指定类型的所占用的范围

typedef : 为某种类型取一别名

4.4 基本数据类型

前面讲的一些有关数据类型的各种基本概念，下面是数据类型这一课真正开始的时候。如果在其中你有什么概念不能理解，最好的方法就是首先回头看本章前面的内容。

4.4.1 字符型和各种整型、实型

类型标识符	类型说明	长度（字节）	范围	备注
char	字符型	1	-128 ~ 127	-2^7 ~ (2^7-1)
unsigned char	无符字符型	1	0 ~ 255	0 ~ (2^8-1)
short int	短整型	2	-32768 ~ 32767	2^{-15} ~ $(2^{15}-1)$
unsigned short int	无符短整型	2	0 ~ 65535	0 ~ $(2^{16}-1)$
int	整型	4	-2147483648 ~ 2147483647	-2^{31} ~ $(2^{31}-1)$
unsigned int	无符整型	4	0 ~ 4294967295	0 ~ $(2^{32}-1)$
float	实型（单精度）	4	$1.18*10^{-38}$ ~ $3.40*10^{38}$	7 位有效位
double	实型（双精度）	8	$2.23*10^{-308}$ ~ $1.79*10^{308}$	15 位有效位
long double	实型（长双精度）	10	$3.37*10^{-4932}$ ~ $1.18*10^{4932}$	19 位有效位

unsigned 用于修饰 int 和 char 类型。它使 int 或 char 类型成为无符号类型。

signed 是 unsigned 反义词，如 signed int 表示有符号类型，不过 signed 可以省略，所以上面列出 char,short int,int 都是有符号类型。

有 short int（短整型）,所以也就有对应 long int （长整型）。long int 用于表示 4 个字节（32 位）的整数。但是在我们现在普通使用的 32 位计算机中，int 默认就是 4 个字节，所以 long 也是可以省略的。

（较早几年，也就是 Windows 3.1/DOS 流行的时候，那时的机器及操作系统都是 16 位的，这种情况下，int 默认是 16 位的。此时，如果想实现 32 位整数，就必须定义为这样的类型：long int）。

在浮点数方面，我们最常用的将是 double。它的精度适合于我们日常中的各种运算。当然，float 的精度也在很多情况下也是符合要求的。

4.4.2 布尔型(bool)和无类型(void)

除字符型，整型，实型以外，布尔型和无类型也是较常用的两种数据类型。

布尔型(bool)

布尔类型是 C++的内容，C 语言没有这一类型。

布尔类型的数据只有两种值：true（真） 或 false（假）。

什么时候使用布尔型呢？

履历表中一般有“婚否”这一项，婚否这种数据就适于用真或假来表示。性别男女，有时也会用布尔值表示，（一般程序都不约而同地把男性设置“真”，女性设置为“假”。）

无类型(void)

这个类型比较怪“无”类型。是的，没有类型的类型。或者我们这样认为比较好接受：在不需要明确指定类型的时候，我们可能使用 void 来表示。

4.4.3 为数据类型起别名：typedef

用法： typedef 原类型名 类型的别名;

为什么要给现成的数据类型起别名？当然也是为了迁就我们为人类。就像我们给人家起绰号一样，形象好记，不易混淆。

比如，我们在编程中需要使用一些年龄数据，应该使用整型(int)类型。另我们还使用到身高的数据，

由于单位采用“厘米”，所以也可能使用 int 类型。两种数据属于同一数据类型，但所代表的内容却不容混淆。我们可以使用 typedef 来临时为 int 取两个别名：

```
typedef int AGE;
typedef int STATURE;
```

通过以上两行话（行末都需要以分号结束），我们获得了两种新的数据类型名称。它们的一切属性都和其原名 int 的数据类型完全一致。只是名字变得很有意义一点而已。

这里为了说明问题举一简单的例子，事实上例子的情况并不值得使用 typedef 来取别名。typedef 常用来为一些复杂的类型取一简单且意义明了的别名。比如定义函数的指针。在 C++ Builder 的控件编写中，更是有非常频繁地用 typedef 来为那些定义复杂的事件函数（类成员函数的指针）取别名。

4.5 sizeof 的使用

用法： sizeof(数据类型)

sizeof(变量)

C/C++提供了关键字 sizeof，用于在程序中测试某一数据类型的占用多少字节。sizeof 有两种用法，效果一样，由于我们还没有学变量，所以我们先讲第一种。

sizeof 是一种计算，计算的对象是指定的一种数据类型，计算的结果是该数据类型占用的字节数目。如：

sizeof(char) = 1；也就是说 char 类型占用 1 个字节。

sizeof(int) = 4;也就是说 int 类型占用 4 个字节。

4.5.1 sizeof 例程

下面我们来完成一个实例。这个例子使用 sizeof 将我们已学的所有数据类型的名称，占用字节数列出。

首先，新建一个控制台工程（详细图解请见：第 2 章第 3 节 DOS 版 Hello World ）

步骤：

1、选择 主菜单 File | New;

2、New Items 对话框，选 New 页中的 Console Wizard，然后点 OK;

3、Console Wizard 对话框，Source Type 中选中 C++ ,右边的分组中**只选中** Console Application 。点 OK

完成以上步骤后，代码编辑窗口自动新建并打开 Unit1.cpp,其代码如下：

```
//---------------------------------------------------------------------------
#pragma hdrstop
//---------------------------------------------------------------------------
#pragma argsused
int main(int argc, char* argv[])
{
   return 0;
}
//---------------------------------------------------------------------------
```

请保存整个工程，包括工程文件名(Project1.bpr)和一个 CPP 文件（Unit1.cpp），请参见第二章。不过为了工程名字有点意义，我决定将它另存为 SizeOf.bpr。

按 F9，编译并运行这个空白工程，会发现一人 DOS 窗口一闪而过，这我们在第二章就知道了。

请添加以下代码中的粗体部分：

```
//---------------------------------------------------------------------------
#include <iostream.h>
#pragma hdrstop
//---------------------------------------------------------------------------
#pragma argsused
int main(int argc, char* argv[])
{
   getchar();
   return 0;
}
//---------------------------------------------------------------------------
```

在第二章中我们的加过类似的代码，目的一样：让程序执行以后不是自动结束，而是在我们敲一个回车键后，才结束，因为我们需要查看程序的执行结果。以后我们所有使用控制台形式的程序，都将需要这两行代码，不过以后不会特别指出了。如你还是有些模糊这两话的作用，最好的办法是现在再按一次 F9 键，运行这个程序。

下面开始我们的 sizeof 用法初试。

我们这程序目标是输出所有我们已学的数据类型：整数（含 unsigned,short 的修饰等），实型（含各精度），字符，bool 等数据类型占用的字节数。但我们事实上还没有学过 C/C++的任何语法，所以我想我们还是先来一个就一句话的，确保成功以后，再完成所有的语句。

请添加以下代码中的粗体部分：

为了省定版面，我只抄出需要变化的 main 那一部分，之前的代码由于不再需要变化，就不抄到这里了，省略号只用来示意前面还有代码，你别把……也输入代码中啊！

```
……
int main(int argc, char* argv[])
{
   cout << "Size of char is : " << sizeof(char) << endl;
   getchar();
   return 0;
}
//---------------------------------------------------------------------------
```

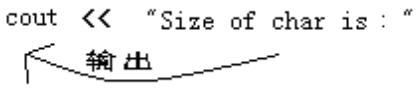
这次输入的是这一行：

```
cout << "Size of char is : " << sizeof(char) << endl;
```

这是一行标准的 C++ 语句。cout 在 C++里表示“标准输出设置”。曾经是每个学习 C++的人必学的内容。在 Windows 的编程里，它已是昨日黄花，我想诸位就不用在花什么时间去学习它了，只要简单地弄明白它的基本用法就是。

cout 和之后的 << 合用。 后者是两小于号，你可别老眼昏眼看作是中文的书名号《 。<< 在 C++里属于操作符重载，我们以后在学习时，还会在不同的场合里遇到见和它的另一位老兄：>> 。理解并且迅速记下 << 的作用是看它的箭头方向：

<< 的指向左边，它的意思是把它右边的东西给，或者是接到右边。现在，它的左边是 cout,cout 这里就是屏幕。给屏幕东西，专业的说法就是：输出到屏幕。



图示为：将 "Size of char is : " 输出到屏幕，当然，一对双引号是不会被输出的，因为它们用来表示这是一句话。

cout 不仅可以输出一句话（字符串），而且可以用来输出各种 C/C++基本类型的数据。并且，通过 <<，它可以一个接一个地输出所需内容。在上面的那行代码中，它首先输出："Size of char is : "；然后是输出 sizeof(char)的计算结果。最后输出 endl。 endl 表示换一新行(回车换行)。

理解了 cout 部分，那一行代码中我们惟一不明白就是行末的那个分号了。是啊你看到了没，所有 C/C++语句行都在语句后有一个';'。当然，和其它 C++语句的主要组成部分一样，它必须是英文符号。如果你打入一个中文的符号。现在改还来得及。并且从此刻起就记住了，所有 C++语句中用到标点符号，除非是指定就要用中文作为输出等参数，否则，分号，逗号，冒号，感叹号，问号等等，全都必须是英文的。

现在，你可以把放心地把那句话原样照敲入你的代码中了。不过，聪明的学员已经完成“Ctrl + C"/"Ctrl + V"大法，直接粘贴了。不过，我想，如果你真的以前还没写过 C/C++代码，还是动手输入。

按 F9 编译，运行。结果如图：

"Size of char is : 1 "中，最后的那个 1 就是 sizeof(char)的计算结果。

至于那个 endl 呢？你看，光标不是在下一行的行首闪烁吗？如果你把代码改成这样：

cout << "Size of char is : " << sizeof(char);

那么，输出结果几乎一模一样，除了光标改成在行末。 我们之所以需要换行，是因为后面还要输出。

你可能喜欢用中文来输出？好吧。

cout << "char(字符)占用的字节数： " << sizeof(char) << endl;

结果是：

我默认不采用中文的原因只是因为学生容易在切换中英输入法的关节出错，结果在不该使用中文的时候输入了一些中文字符，造成无谓的错误，白白浪费学习时间。

再接再厉，我们输出所有语句。包含我们用 typedef 自定义了数据类型的别名。

……

```
int main(int argc, char* argv[])
{
   typedef int AGE;  //为 int 取一个别名
   //cout << "char(字符)占用的字节数： " << sizeof(char) << endl;
   cout << "Size of char is : " << sizeof(char) << endl;
   cout << "Size of unsigned char is : " << sizeof(unsigned char) << endl;
```

```
    cout << "Size of short int is : " << sizeof(short int) << endl;
    cout << "Size of unsigned short int is : " << sizeof(unsigned short int) << endl;
    cout << "Size of int is : " << sizeof(int) << endl;
    cout << "Size of unsigned int is : " << sizeof(unsigned int) << endl;
    cout << "Size of long int is : " << sizeof(long int) << endl;
    cout << "Size of unsigned long int is : " << sizeof(unsigned long int) << endl;
    cout << "Size of AGE is : " << sizeof(AGE) << endl;
    cout << "Size of float is : " << sizeof(float) << endl;
    cout << "Size of double is : " << sizeof(double) << endl;
    cout << "Size of long double is : " << sizeof(long double) << endl;
    cout << "Size of bool is : " << sizeof(bool) << endl;
    getchar();
    return 0;
}
```

为了看着清楚，我加了一些空行。

我很乐意承认，这回我也当了一次聪明人。写完成上面的代码过程中，一直在用“Ctrl + C”和“Ctrl + V”。关键是复制完之后要及时改变需要变的地方。

另外，检查是否改错的一个方法就是利用 CBC 的代码窗口对关键字变色的特征。sizeof 本身和 sizeof()括号中的数据类型关键字都会变色。如果你发现有一处不是这样，那你一定是输入错了。比如你把 sizeof(char)写成 sizeof(cahr)。那么，cahr 不会变色。

我们漏了一个类型：void。void 是无类型。sizeof 无法对无类型取其字节数。这是 void 的特殊性。当作为“无类型”意义时，它字大小是 0 字节，如函数的返回值。当作为“未确定”的意义时，它的大小也是未确定的。当然，这也是 sizeof 的一知识点：它的计算对像必须确定的类型。

确保输入无误。F9。结果如下：

“这一章好长。这一夏好热！要收多少个学员的 20 元钱，才能省下钱买空调呢？”我把键盘往前一推，长吁一口气，幽幽地想。

最终我决定到楼下买根冰棍。

第五章 变量和常量

5.1 从类型到变量

5.1.1 公孙龙的“白马非马”

故事是春秋时的公孙龙先生说的。

城门上告示：“马匹不得入城”。公孙龙同志骑白马而来，遭拒入。公孙龙一脸正色：“告示上写的是‘马’，而我骑的是‘白马’，难道 ‘马’等于 ‘白马’吗？”

守门士兵觉得白马还真不是马，于是放行。

依公孙龙先生的理论认为：如果白马是马，黑马也是马，那么岂不白马等于黑马，所以，不能说白马是马。“白马非马”是中国哲学史上的一桩公案。不过，若是我们从程序的角度上说，可以认为：马在这里表示一种类型，而白马，黑马它们的类型都是马。

白马，黑马具有相同的“数据类型”，但它们都**相对独立的个体**。从这点说，别说有白黑之分，就算同样是白马，这一匹和哪一匹白马，也是相对独立的个体。

在程序中，“类型”和“变量”的关系正是“马”和“白马”的关系。

如果 C 或 C++有这种数据类型： Horse,那么，定义一匹“白马”的变量应该这样：

```
Horse AWhiteHorse;
```

以后我们说不定真的有机会自己定义 Horse，不过现在，我们在上一章的学的数据类型是：

char,int,bool 等等。

假设我们需发使用一个有关年龄的变量，在 C 或 C++中是这样定义的：

```
int age;
```

现在让我们来事先建立一个空的工程，随着本章课程的进展，我们需要不断地在工程中加入零星的代码，及时实践。

仍然是一个空的控件台程序。方法是……以前我们讲过，忘了就看前面章节吧。

代码文件 Unit1.cpp 中，手工加入以下的黑体部分：

```
//---------------------------------------------------------------------------
#include <iostream.h>
#pragma hdrstop
//---------------------------------------------------------------------------
#pragma argsused
int main(int argc, char* argv[])]
{
  getchar();
  return 0;
}
//---------------------------------------------------------------------------
```

5.1.2 定义变量

语法：

数据类型 变量名;

“张三”既可以指张三这个人，也可以是张三的名字。同样，上面的“变量名”，其实就是变量本身。

举上一节的例子：

```
int age;
```

其中，int 是数据类型（整型），而 age 是变量名，更多的时候，我们就说是变量 age。最后是一人分号。它表示定义一变量在 C 或 C++里一句完整的语句。因为 C++的语言总是以分号结束。

如果要声明一个字符类型变量：

```
char letter;
```

声明一个 bool 类型的变量：

```
bool do_u_love_me;
```

其它类型，除了 **void 不能直接定义一个变量**以外，格式都是一样的。

```
void avoid; //错！void 类型无法直接定义一个变量。
```

有时同一时候同一数据类型需要多个变量，此时可以分别定义，也可以一起定义：

```
int a;
int b;
int c;
```

下面采用一起定义，会更省事：

```
int a,b,c;
```

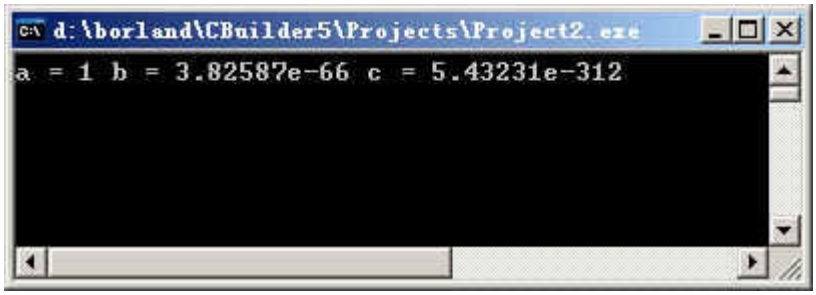
一起定义多个同类型变量的方法是：在不同变量之间以逗号（,）分隔，最后仍以分号（;）结束。

让我们来试试变量定义，另外，我们还要趁此机会，看看一个变量在仅仅进行定义之后，它的值会是什么。

继续上一小节的代码。仍然是加入黑体部分，当然 // 及其后面的内容是注释，你可以不输入。

```
......
int main(int argc, char* argv[])]
{
 ////////////////定义变量/////////////////////////////////////////////////
  //以下定义三个变量：a,b,c
  int a;
  double b,c;
  //a，b，c 仅仅被定义，它的值会是什么？我们用 cout 输出三个变量：
  cout << "a = " << a << " b = " << b << " c = " << c << endl;
  getchar();
  return 0;
}
```

最好先保存代码文件和工程文件。然后按 F9 运行。以下是笔者机器得到结果。



a 是 1，b 和 c 都像天文数字？嗯，从这里我们学到一个 C，C++编程极其重要知识：**未初始化的变量，它的值将是未确定的**。所谓“未初始化”，就是指这个变量只有定义代码，而没有赋值。

（立即重复执行这段代码，得到结果可能都一样，但这并不说明这些值就是可以确定不变。）

5.1.3 如何为变量命名

C/C++的变量的名字中只能有以下字符：大小写字母、阿拉伯数字（但不能做为开头）、下划线 _。

汉字不能成为变量名。

不过，就算允许，又有谁会这么累呢，用汉字作变量名？

不能或不要

不能取名为 C、C++的保留字。

如：

```
int char; //不行
```

这是不被允许的。char 是一个保留字。我们不能再拿来做变量。

不能超过 250 个字符。

这在 BCB 里有规定。是一个可以调整的值。

不能以数字开头

```
int 100;      //不行
int 100ren;  //不行
```

不能夹有空格

```
bool do you love me; //不行
```

你可以用下划线代替空格：

```
bool do_you_love_me; //OK
```

不能在同一作用范围内有同名变量 （仅 C++）

比如：

```
int abc;
int abcd;
int abc;  //不行
```

在 C 里， 上面重复定义变量 abc 是被允许的。

关于作用范围,我们以后会讲到。

不要和 C、C++中已有的全局变量，函数，类型名取相同的名字。

```
double sin;
```

这不合适。因为 C 库为我们提供了计算正弦值的函数，名字就叫 sin;

不要太长。

是的，250 个字符其实太长了。如果有个变量名长达 30 个字母，我看不仅你自己写得累，别人看着也会觉得是在读外国小说，主人公的名字太长，烦人。

不要太短

这个我们放到后面说。

以上几点中，凡标为“**不能**”，意味如果你违反了，你的程序便会在编译时出错。而“**不要**”则仅属建议内容。真要这么做，程序也没错。

另外，我们还建议要为每个变量取一个有意义的名字。比如 NumberOfStudents,这个变量一看就明白是为“学生的人数”定义的。而如果定义为 aaa,cc,之类，就容易混淆。当然，有一些约定成俗的用法，如字母 i,j，等常用来作循环流程中的计数变量。再者，有意义的名字并不是指一定要把变量所要代表的意思用英文句子写出，一般可以用简写，如 NumOfStudent，它同样意义明了，但更简短。而且，如果我们英文一般，那么有时也可以使用拼音。这里就不举例了，因为笔者连拼音都很次。

前面说到取名不要太短，说的就是避免像 aaa，cc 之类的图输入方便，但毫无意义，不可读的变量命名习惯。

(你很快就会在教程中发现，笔者似乎自己违反了这个规定，用一些 a,b,c 作为变量名。这不能说是笔者的错。因为会有些时候变量的意义并不重要)

最后，**C,C++是区要大小写的语言**，所以变量 Num 和变量 num 完全是两个变量。大家在定义，使用变量要注意到这一点。

关于变量的命名，我们不做实践。下面附加说说编程中命名的一些风格。

附：关于命名变量的几种风格。

较早以前，现在仍流行于 UNIX、Linux 编程界，程序员喜欢用全部小写的单词来命名变量，如果两个单词，比如 my car，常用的命名方法有两种：my_car 或 myCar。my_car 自然看起清楚，不过输入频繁地下划线是件累事(根据指法，下划线必须使用小指头按)。后一种方法被称为“驼峰表示法”，原因是大写字母看起来想凸起的驼峰。

之所以不使用 MyCar,原因是 C/C++允许程序自定义数据类型，所以有必要从一个名字上很快地区分它是变量或是数据类型。方法是让自定义的数据类型都用大写开头。比如前面的说的“马”是程序员自定的数据类型，那么如果采用这里的命名规则，则应取名为：Horse，而“一匹白马”是变量，所以取名为：aWhiteHorse。

```
Horse aWhiteHorse;
```

在 C++ Builder 里，并没有限制大家如何为变量取名。所以为了仍可以很明显的做到上述的区分，CB 的方法是对用户自定义的数据类型在前头加一个字母 T(Type 的首字母)。仍如 Horse,则改名为:THorse。前面我们写 Windows 版的 hello world 时，使用了一个 Label 控件，其实，检查代码你会发现，它的类名就叫：TLabel。

最后还有一种方法是匈牙利标记法（Hungarian notation）。该法要求在每人变量的前面加上若干个用于表示该变量的数据类型的小写字母。如 iMyCar 表示这个变量是整型(i 表示 int);而倘若是 cMyCar，

则表示这个变量是 char 类型。该法经过一段时间的训练熟悉以后，会带来一些好处。问题是如果对自定义的数据类型也按这种方法进行，就不是经过训练就能熟悉了。比如 hoWhite,这个名字中的 ho 表示“马”，真有点强人所难。举上实际存在的例子，在 Windows API 中，如果你看到：

LPCITEMIDLIST pidlRoot;

想要一眼看明白 pidRoot,必须的要求是你很明白 ITEMIDLIST 是什么玩意儿了。

是的，Windows 的 API 使用的是最后一种方法。在大多数情况下，它的变量的名字都看上去怪怪的。

在本部教程中，我们在正式程序中，最常使用的方法是简单的“驼峰”法。

5.1.4 如何初始化变量

5.1.4.1 什么时候需要给变量初始化？

int a;

声明了一个整型变量 a。但这变量的值是多少？a 等于 0 吗？还是等于 100？我们都不知道。“不知道”的意思是：a 有值，但这个值的大小是随机的，所以我们无法确定。

无法确定一个变量值是常有的事，比如我们要求用户输入他的年龄。我们需要声明一个整型变量来存储用户年龄，但在用户输 入之前，我们是无法确认它的值。

但有些时候，我们可以事先给一个变量初始时的值。同样是年龄的问题，虽然我们不知道用户到底几岁，但我们知道他不可能是 0 ，所以我们把年龄一开始设为 0。为什么要这样？用户有时不小心忘了输入年龄（就像我们在网上填表一样），我们就可以检查年龄是否为 0 来发现了。另外一种相反的用法是，我们发现大都数用户是 8 岁（比如一个小学生入学登记表），这时我们初始化年龄变量为 8，目的是为了方便用户了。

那么，如果为一个变量赋值呢？

答案就像我们初中的代数：设 x = 10, y = 100。用等号。请记住：现实生活中，等号（=）有两个意义，但在 C/C++里，＝ 只用来给一个变量赋值。

5.1.4.2 初始化变量的两个时机

1. 在定义时初始化变量

```
int a = 0;
```

通过一个等号，我们让 a 的值等于 0；

同时定义多个变量时也一样：

```
int a = 0, b= 1;
```

当然也可以根据需要，只给部分变量初始化。

```
int a = 0, b;
```

或：

```
int a,b = 1;
```

2. 在定义以后赋值

```
int a;
a = 100;
```

5.1.4.3 通过计算得到初始值

给变量赋值，除了给一个直接的值以外，还可以通过计算获得。如：

```
int a = -3 + 200 - 5;
```

或者如：

```
int a = 9;
int b = 3;
int c = a / b;
```

(/ 表示除号)

现在来试试给变量赋值的几种方法。

```
......
int main(int argc, char* argv[])]
{
 ////////////////定义变量/////////////////////////////////////////////////
 //以下定义三个变量：a,b,c
  int a;
  double b,c;

  //用 cout 输出三个变量：
  cout << "a = " << a << " b = " << b << " c = " << c << endl;

////////////////初始化变量/////////////////////////////////////////////////
  int d = 0;
  float e =  1.234, f;
  f = 567.8 + 0.2;
  cout << "d = " << d << " e = " << e << " f = " << f << endl;
  getchar();
  return 0;
}
```

5.1.4.4 变量的取值范围

1). 变量允许取值范围 ＝ 变量数据类型的范围

关于赋值的最后一个需要注意的是：变量的取值范围。

变量的取值范围？变量是有类型的，变量的允许的取值范围等同于其数据类型的范围。比如一个整型数，那么它的取值范围就是：-2147483648 ~ 2147483647。而如果是一个字符类型，它的取值就不能是-129。

以下是示例：

```
int  a = 3000000000; //错误!
char b = -129; //错误!
```

我们来写代码实践一下。由于 char 类型计算机在输出将直接输出字符，我们不好观察结果，所以我们选择了 int 类型。

```
......
int main(int argc, char* argv[])]
{
```

```
    ////////////////定义变量/////////////////////////////////////////////////
    //以下定义三个变量：a,b,c
    int a;
    double b,c;
    //用 cout 输出三个变量：
    cout << "a = " << a << " b = " << b << " c = " << c << endl;
    ////////////////初始化变量/////////////////////////////////////////////////
    int d = 0;
    float e =  1.234, f;
    f = 567.8 + 0.2;
    cout << "d = " << d << " e = " << e << " f = " << f << endl;
////////////////变量值越界/////////////////////////////////////////////////
    int g = 3000000000; //给 g 赋值超过 int 允许的范围，所以 g 的值不可能如愿等于 3000000000
    cout << "g = " << g <<endl;
    getchar();
    return 0;
}
```

g 的值已经溢出，它的值究竟是多少？自己按 F9 运行，看结果吧。

2). bool 类型的特殊

至于 bool 类型，事实上它的内存空间范围和 char 一样是一字节的（因为计算机能直接处理的最小内存单位是字节），所以按说它也能存 256 个不同的值，但作为 C++的一个规定，强制给它两个值：false,true。那么 false 是什么呢？其实它是 0。而 true 则是非 0 的数。也就是说，值 0 对应为 false,而所有非 0 的数对应成 true。

所以，你这样给 bool 类型的变量赋值自然没错：

```
bool do_u_love_me = false; //噢，你不爱我。
```

或

```
bool do_u_love_me = true;
```

但你也可以这样给来：

```
bool do_u_love_me = 0;
```

或

```
bool do_u_love_me = 1;
```

当然，我们建议使用第一种方法，那是“正规”的方法。

在 C 里，并没有 bool 这个数据类型，所以 C 程序员采用的方法是自定义：

```
typedef char BOOL;
```

如果你在某些书上碰到 BOOL，你就可以明白它就是我们学的 bool.

false 和 true 到底是什么？“假”或“真”，很简单啊。噢，我不是问二者的意思，C++之所以要加入 bool 类型，目的也是为了让程序员在处理那些只用于表示“真或假”、“是或否”等数据时，可以用直观的 false 和 true 来表示。而在计算机内部，二者仍然是数。让我们写几行代码，揭开 bool 类型的“假面”。

```
......
int main(int argc, char* argv[])]
{
```

```
 ////////////////定义变量/////////////////////////////////////////////////
 //以下定义三个变量：a,b,c
  int a;   double b,c;
  //用 cout 输出三个变量：
  cout << "a = " << a << " b = " << b << " c = " << c << endl;
////////////////初始化变量/////////////////////////////////////////////////
  int d = 0;
  float e =  1.234, f;
  f = 567.8 + 0.2;
  cout << "d = " << d << " e = " << e << " f = " << f << endl;
////////////////变量值越界/////////////////////////////////////////////////
  int g = 3000000000; //给 g 赋值超过 int 允许的范围，所以 g 的值不可能如愿等于 3000000000
  cout << "g = " << g <<endl;
////////////////bool 类型的"真面目"/////////////////////////////////////////
 bool h = false ,i = true;
  cout << "h = " << h << " i = " << i << endl;
  getchar();
  return 0;
}
```

运行后……原来，false 是 0，true 是 1。

3). char 类型的特殊

char 的范围是 -128 ~ 127

unsigned char 的范围是 0 ~ 255

那么按照前面的说法，我们可以为这样为一个字符类型的变量赋值：

```
char c = 120;
unsigned char uc = 250;
```

这样看来，所谓的“字符”类型，似乎除了取值范围小一点以外，和整型也没有什么区别。这句话的确没错。对于 C、C++来说，字符类型完全可以当成一个整数来对待。

事实上，所有信息在计算机里，都是使用数字来表达。英文字母 'A' 在计算机里表示为 65; 字母 'B' 表示为 66。所有你在键盘可以看到的字符，如大小写英文字母，阿拉伯数字符号，标点符号都可以有一个相应的数值表示。

但要让我们记住 65 就是'A',而 33 就 '!'等 255 个对应关系，显然很折磨人，所以，计算机高级语言允许我们直接为字符类型变量这样赋值：

```
char  c = 'A';
char  d = '!'; //英文感叹号
char  e = '.'; //英文句号
char  f = ' '; //空格
```

即：将所要得到的字符用单引号括住。（引号''是英文状态下的，千万不要使用中文符号）

另外，对于一个数值类型，如果它等于 120，那么输出时显示的 120，如果是一个字符类型，输出却是 120 对应的字符。也就是说：

```
int  k = 120;
char j = 120;
```

二者虽然值都为 120，但输出 j 时，计算机并不显示 120 这个值，而是 120 对应的字符。试试看！

为了不让教程中的代码重复占用版面，省略号省略掉的代码要多点了……

```
......
//////////////////char 类型//////////////////////////////////////////////////
int k = 120;
char j = 120;
cout << "k(int) = " << k << " j(char) = " << j << endl;
getchar();
......
```

输出结果，k 当然是 120，但 j，原来 120 对应的字母是 'x'。写的是 120，输出的却是 x，很不直观对不？所以，除非我们故意要和自己或者其他看代码的人玩“密码”，否则，还是直接想要什么字符，就写哪个字符吧。

```
//////////////////char 类型//////////////////////////////////////////////////
int k = 120;
char j = 120;
cout << "k(int) = " << k << " j(char) = " << j << endl;
char l = 'A';
char m = l + 1;
cout << "l = " << l << " m = " << m << endl;
getchar();
......
```

输出结果，l 为 'A',而 m 为 'B',想一想，为什么？学完后面内容就有答案。

单引号本身也是一个字符，如何表示单引号呢？是否用 ''' 来表示？看明白下面的常用字符 ASCII 码表以后再说。

（ASCII 是指：American Standard Code for Information Interchange，美国信息交换标准码。）

值	符号	值	符号	值	符号
0	空字符	44	,	91	[
32	空格	45	-	92	\
33	!	46	.	93	]
34	"	47	/	94	^
35	#	48 ~ 57	0 ~ 9	95	-
36	$	58	:	96	`
37	%	59	;	97 ~ 122	a ~ z
38	&	60	<	123	{
39	'	61	=	124	|
40	(	62	>	125	}
41	)	63	?	126	~
42	*	64	@	127	DEL (Delete 键)
43	+	65 — 90	A ~ Z		

（其中，0~31 都是一些不可见的字符，所以这里只列出值为 0 的字符，值为 0 的字符称为空字符，输出该字符时，计算机不会有任何反应。我们以后会学习 0 字符的特殊作用。）

4)．转义符的使用

根据前面的说法，单引号应该表达为：

char c = ’’’;

但这是错误的。C、C++不认识 ’’’，因为它容易引起歧义。

另外，有一些字符属于不可见的字符，无法直接写出，比如键盘上大大的回车键，在输入文档时，用它可以输入一个回车换行，显然我们不能这样在 C/C++里表示一个回车换行：

char c = ’
’

在第一个’和第二个’之间夹了一个换行，这样的表示方法不仅不方便，C 和 C++也不认。

类似这样的问题还有制表符（键盘上的 Tab 键）等等。

解决的方法是使用**转义符.**C/C++使用反斜杠’\’作为转义符。如：

’\’’ ： 表示单引号；

’\″’ ： 表示双引号；

’\n’ ： 表示换行（n ： line)；

下面列出常用的 C、C++特殊字符：

字符	数值	意义
’\a’	7	响铃（输出该字符时，屏幕无显示，但喇叭发音）
’\n’	10	换行（n: line）
’\t’	9	制表符（横向跳格）
’\r’	13	回车（return）
’\\’	92	输出转义符 ’/’ 本身
’\″’	34	双引号
’\’’	39	单引号

这里顺便解释一下“回车换行”是什么，尽管我们对这个词耳熟得很。

“回车换行”是“回车”加“换行”。

换行好理解，但什么叫“回车”呢？它和“换行”又有什么关系？

原来，“回车换行”的概念源于早先的打字机。类似于现在打印机中有一个打印头，这个打印头平常停在打印机内的某一端。在打印一行时，则需要向外移动，打印一行结束后，打印头需要回到原来位置。由于打印头在英文中用“车”来表示，所以这个动作就称为“回车”，用金山词霸的中的解释就是：“将打印或显示位置移到同行起始位置的运动。”

所以对于打印机，假设有两行字，两行之间若光有“回车”，那么这两行字将重叠在一起(对于控制台程序的屏幕，则将回到行首)。如果光有“换行”，则第二行将不从起始位置打起，样子如下：

这是第一行
　　这是第二行。

只有既输出“回车”又输出“换行”，才是我们常见的换行结果。当然，对于当今的大都软件，往往都把单独的回车或换行直接解释于二者的结合。

转义符的另外一种用法是直接接数值。但必须采用 8 进制或 16 进制。这里暂不讲解。

如果需要使用数值表示，最直接的还是使用类似： c = 120; 的方法。比如要让变量 c 的值为单引号，我们至少可以有以下 2 种方法：

char c = ’\’’; //使用转义符

```
char c = 39;   //直接赋给字符的 ASCII 的值。
```

转义符的内容，看上去怪怪的？不过，多用几次我们就会明白。

```
//////////////////char 类型/////////////////////////////////////////////////////
int k = 120;
char j = 120;
cout << "k(int) = " << k << " j(char) = " << j << endl;
char l = 'A';
char m = l + 1;
cout << "l = " << l << " m = " << m << endl;
//////////////////转义符/////////////////////////////////////////////////////////
cout << "TAB:" << '\t' << "AA" << endl;
cout << "换行:" << '\n' << "AA" << endl;
cout << "回车:" << '\r' << "AA" << endl;
cout << "听到 BEEP 声了吗？" << '\a' << endl;
cout << '\'' << endl;
cout << '\"' << endl;
cout << '\\' << endl;
getchar();
......
```

在执行之前，有必要稍作解释。

首先那是“AA"做什么用。因为制表符、回车、换行等特殊字符，其输出效果是改变光标位置，所以我们需要一些上下文来显示出光标位置改变效果，这里就随便写个“AA”了事。

然后是在 cout 语句中，平常我们叫是使用双引号输出一行话，但如果当前要输出只是一个字符，我们也可以使用单引号。

至于所谓“BEEP”声，你可别抱太多期望，它只是计算机内置的小喇叭短促一个声音，听起来并不美妙。

现在来看结果(请只关心转义符部分)：

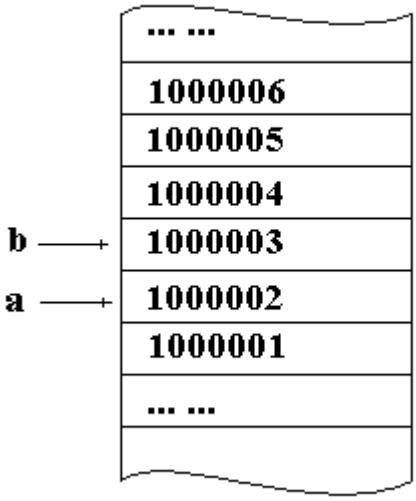
关于输出结果的几点说明：

1、需要注意的是 '\t' 在控制台窗口的输出效果，如果前面的字符长度已超过一个制表位，那么后面的第一个'\t'将是没有效用的。（要理解这一点，你可以将代码中“TAB”加长一点，如"TABTAB"）。

2、“AA 车” 的输出结果是怎么来的呢？请大家考虑考虑。

试验程序在这里结束。

5.2 变量与内存地址



前面讲到“白马、黑马”时，我们说一匹白马和一匹黑马具有共同的数据类型“马”，但二者是相对独立的个体。现在我们以共熟悉的“人”来继续这个话题，最终引出变量与内存地址的关系。

张三和李四的数据类型都是“人类”。但张三和李四显然是独立的量：张三吃了两块蛋糕，李四不可能因此就觉和肚子饱了；而李四在下班的路上捡到一个钱包，虽然正好是张三的，两人似乎通过钱包有了点关系，但谁得谁失仍然不容混淆。

这一切都很好理解。张三和李四之所以是不同的个体，根本原因在于两人有着不同的肉身。如果是一对连体婴儿，虽然也是两个人，但当左边的婴儿被蚊子咬一口时，右边婴儿是否也会觉得痒，就不好说了。

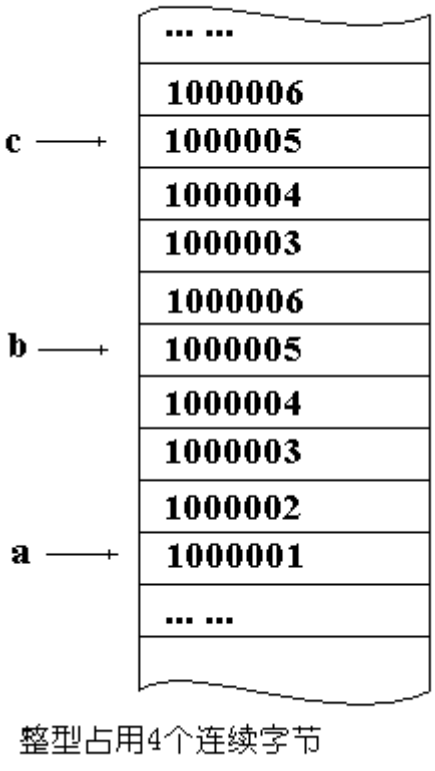
现在我们困难的是，如何理解两个不同的变量，也是互相独立的呢？

答案就在“内存地址”，“内存地址”就是变量的肉身。不同的变量，拥有不同的内存地址。譬如：

```
char a;
char b;
```

上面有两个字符类型的变量 a 和 b，a 拥有自己的内存地址，b 也拥有自己的内存地址，二者绝不相同。而 a、b 只不过分别是那两个内存地址的“名字”，恰如“张三、李四”。

让我们看图解：



整型占用4个连续字节

看，内存就像是开宾馆的。不过这有宾馆有点怪。首先它每一个“房间”的大小都是一个**字节**(因此，计算机能单独处理的最小内存单位为字节)。它的门牌号也不叫房号，而是叫**内存地址**。

在左图中，“房客”，变量 a 住在内存地址为 1000002 的内存中，而变量 b 则住在它的隔壁，地址为 100003 的内存中。另外，如果你足够细心，你还会发现:**内存地址由下往上,从小到大排列**。

变量的内存地址是在程序运行时，才由操作系统决定。这就好像我们住宾馆。我们预定一个房间，但房间号由宾馆根据情况决定，

我们可以改变变量的值，但变量的地址我们无法改变。对照宾馆一说，就是我们订了房间，可以不去住，还可以决定让谁去住在那个房间里。（当然，现实生活中宾馆可能不会允许你这么做）。

在前面图示的例子中，a、b 是字符(char)类型。各占用 1 个字节。如果是 int 类型，那么应该占 4 个字节。这 4 个字节必须是连续的。让我们再来看一个例子：

```
int a;
int b;
char c;
```

这回，我们声明了两个 int 类型和一个 char 类型的变量。同时和上面一样，我们事实上是假设了这三个变量被依次序分配在相邻的内存地址上（真实情况下，这和其它因素，如指定的字节对齐方式等有关）。从右图中可以看到整型变量 a 占用了 1000001～100004 这 4 个字节。

在我们已学习的数据类型中，long double 占用 10 个字节，是占用内存空间最大的一种数据类型。以后我们学习数组，或者用户自定数据类型，则可能要求占用相当大的，并且同样必须是连续的空间。因此，如果操作系统仅仅通过简单的“按需分配”的原则进行内存管理，内存很快就会宣告不足。事实上，操作系统的内存管理相当复杂。幸好，一个普通的程序员并不要求去了解这些内幕。更多的有关内存管理的知识，我们会在下一部课程中学习。但是本章中有关内存的内容却相当重要。

让我们来看看我们学了什么：

1、不同的变量，存入在不同的内存地址，所以变量之间相互独立。

2、变量的数据类型决定了变量占用连续的多少个字节。

3、变量的内存地址在程序运行时得以确定。变量的内存地址不能改变。

除了这些以外，我们现在还要增加几点：

现在，我们可以明白，为什么需要变量，显然，这又是一个讨好人类的做法。在汇编和机器语言，就是只对内存进行直接操作。但你也看到了，内存地址是一堆长长的数，不好记忆；另外，管理内存复杂易错，让程序员直接管理内存显示不可能。而通过变量，不仅让内存地址有了直观易记的名字，而且程序员不用直接对内存操作，何乐而不为呢？事实上，这是所有高级语言赖于实现基础。

既然变量只不过是内存地址的名称，所以：

4、对变量的操作，等同于对变量所在地址的内存操作。

第五点是反过来说：

5、对指定内存地址的内存操作，等同对相应变量的操作。

尽管这简直就是在重复。但这一条却是我们今后理解 C、C++语言相对于其它很多高级语言的，最灵活也最难学的“指针”概念的基石。

5.3 常量

说完变量，我们来学常量。

看一段代码片段。省略号表示可能会有的其它操作。

```
int a = 3;
....
a = 100;
```

代码中，a 是变量。一开始我们初始化为 3。后来出于什么需要，我们又将它赋值为 100。a 的值也就从 3 变成了 100。

代码中，3 和 100 就是一种常量。像这种直接在代码写出大小的量，称为**立即数**，也称为**常数**，而常数是常量的一种。

常量的定义：常数，或代表固定不变值的名字。

5.3.1 几种数据类型常数的表达方法

5.3.1.1 整型常数的表达

用 10 进制表示，当然是最常用也是最直观的了。如：7,356,-90,等等。 C,C++语言还允许我们使用 8 进制或 16 进制表示。这里且不讲。至于 2 进制形式，虽然它是计算机中最终的表示方法，但它太长，而且完全可以使用 16 进制或 8 进制方便地表达，所以计算机语言不提供用 2 进制表达常数的方法。

有时，你也会看到一些老的代码中，在一些整型常后面加一个大写或小写的 L 字母。如：989L 这是什么意思呢？原来，一个常数如果其范围允许，那么计算机默认将其认为是 int 类型的，那么要让计算机把某个数认为是 long int 类型，就可以在其后面加 L 或 l。不过，这在以前的 16 位机器才有意义了。现在，我们的机器都是 32 位，long int 和 int 完全一样，都是占用 4 个字节，所以，我们没有必要这样用了。

5.3.1.2 实型常数的表达

实型常数一般只用 10 进制表示。比如 123.45，或 .123。后者是 0.123 的简写。不过我个人认为，少写一个 0 的代价是很容看错。

实型数还可以使用科学计数法,或曰指数形式，如：123e4、或 123E4 都表示 123 * 10^4,即 1230000。

我们学过的实数数据类型有：float,double,long double。在 C++中，默认的常数类型是 double。比如你写：

1.234;

那么，C++按 double 类型为这个数分配内存，也就是说为它分配 8 个字节。如果要改变这一点，可以通过加后缀字母来实现。

加 f 或 F，指定为 float 类型。

加 l 或 L, 指定为 double 类型。

以下示例：

```
12.3f  //float 类型
12.3   //默认类型（double）
12.3L  //long double 类型
12.3e400  //long double 类型，因为值已不在 double 类型的取值范围内
```

5.3.1.3 字符常量的表达

关于字符的表示方法，我们已经在 5.1.3.4 节中的第 3 点讲过。这里简单重复。

字符常量用单引号括起来的一个字符，如：'a','b','c',' ','A'。等。

可以用转义符来表示一些无法打出的符号，如 '\n','\r','\t'。

这里补充一点：值为 0 的字符，称为空字符，或零字符。它用 '\0' 表示。注意，它和阿拉伯数字字符 '0'完全是两个数。（请大家查一前面常用字符 ASCII 值表中，后者的值是多少）

5.3.1.4 字符串常量

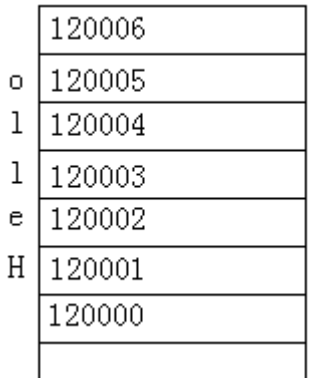
字符串由字符组成。在C / C++语言中,字符串的是由一对双引号括起的来的字符序列。如：

```
"Hello, world!"
"How do you do?"
"Hello"
```

上面 3 行都是字符串常量。注意，双引号是英文字符。

字符串是由一个个字符排列而成，所以在 C/C++中，字符串在内存中的存储方式，我想你可以想象得出。每个字符占用一个字节，一个接一个排列。但是，双引号本身并不存储，因为双引号是为了表达方便，并不是实际内容。下面是示意图：

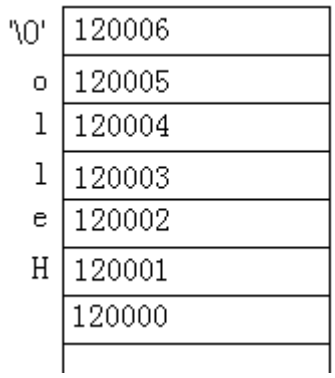
（为了简单，我们选了例子中最短的字符串）



不过，字符串的存储工作还有一点点事情需要做。举一个直观的例子。如果在上图中的 Hello 后，内存地址为 120006 正好还存放一个字符：‘o’。那么，程序就会把 Hello 和 o 连起来认作是一个字符串“Helloo”。为什么呢？

前面我们讲字符类型，整型，等变量或常量时，根据数据类型的不同，它们都有已知的，固定的大小。字符串是由字符组成的，虽然每个字符的大小固定 1 个字节，但字符串的大小却是可变的。所以必须有一个方法来表示一个字符串在哪里结束。

空字符（值为 0 字符）担起这个责任。因为空字符没有任何输出效果，所以正常的字符串中是不会出现空字符的。因此，用它来表示一个字符串的结束，再合适不过了。以下是带有字符串在真正的内存存储示意图。



记住，空字符用 ’\0’ 表示。

有两个字符串：

"Hello"

"What"

假设二者在内存中存储的位置正好是连续的，那么内存示意就为：

（为了结束版本，我这里横向表示，并且不再写出仅用于示意的内存地址）

H	e	l	l	o	\0	W	h	a	t	\0

从表中，我们可以看出空字符 ’\0’ 是如何起到标志一个字符结束的作用。

5.3.2 用宏表示常数

假如我们要写一个有关圆的种种计算的程序，那么∏(3.14159)值会被濒繁用到。我们显然没有理由去改∏的值，所以应该将它当成一个常量对待，那么，我们是否就不得不一遍一遍地写 3.14159 这一长

串的数呢？

必须有个偷懒的方法，并且要提倡这个偷懒，因为多次写 3.14159，难免哪次就写错了。

这就用到了宏。宏不仅可以用来代替常数值，还可以用来代替表达式，甚至是代码段。（宏的功能很强大，但也容易出错，所以其利弊大小颇有争议。）今天我们只谈其中代替常数值的功能。

宏的语法为：

#define 宏名称 宏值

比如要代替前面说到的∏值，应为：

#define PAI 3.14159

注意，宏定义不是 C 或 C++严格意义上的语句，所以其行末不用加分号结束。

宏名称的取名规则和变量名一样，所以我们这里用 PAI 来表示∏，因为 C、C++不能直接使用∏字符。有了上面的语句，我们在程序中凡是要用到 3.14159 的地方都可以使用 PAI 这个宏来取代。

作为一种建议和一种广大程序员共同的习惯，宏名称经常使用全部大写的字母。

假设原来有一段代码：

```
double zc = 2 * 3.14159 * R; //求圆周长，其中 R 是代表半径的变量
double mj = 3.14159 * R * R; //求圆面积
```

在定义了宏 PAI 以后，我们就可以这样使用：

```
#define PAI 3.14159
double = 2 * PAI * R; //求圆周长，其中 R 是代表半径的变量
double = PAI * R * R; //求圆面积
```

用宏来取代常数，好处是：

1)让代码更简洁明了

当然，这有赖于你为宏取一个适当的名字。一般来说，宏的名字更要注重有明确直观的意义，有时宁可让它长点。

2)方便代码维护

就如前面说的 3.14159。哪天你发现这个∏值精度不够，想改为 3.1415926，那么你只修改一处宏，而不是修改代码中的所有宏。

原来的宏：

```
#define PAI 3.14159
```

修改后的宏：

```
#define PAI 3.1415926
```

对宏的处理，在编译过程中称为“预处理”。也就是说在正式编译前，编译器必须先将代码出现的宏，用其相应的宏值替换，这个过程有点你我在文字处理软件中的查找替换。完成预处理后，所有原来的“PAI”都成了立即数 3.1415926。所以在代码中使用宏表达常数，归根结底还是使用了立即数，并没有明确指定这个量的类型。这容易带来一些问题，所以 C++使用另一更稳妥的方法来代替宏的这一功能。

5.3.3 常量定义

常量定义的格式为：

```
const 数据类型 常量名 = 常量值;
```

相比变量定义的格式，常量定义必须以 const 开始，另外，常量必须在定义的同时，完成赋值。

```
const float PAI = 3.1415926;
```

const 的作用就是指明这个量（PAI)是常量，而非变量。
常量必须一开始就指定一个值，然后，在以后的代码中，我们不允许改变 PAI 的值，比如：

```
const float PAI = 3.14159;
double zc = 2 * PAI * R;
PAI = 3.1415926;          //错误！，PAI 不能再修改。
double mj = PAI * R * R;
```

如果一个常量是整型，可以省略指出数据类型，如：
const k = 100;
相当于
const int k = 100;

反过来说，如果不指定数据类型，则编译器将它当成整型。比如：
const k = 1.234;
虽然你想让 k 等于一个实型数，然而，最终 k 的值其实是 1。因为编译器把它当成整型常量。
我们建议在定义变量时，明确指出类型，不管它是整型或其它类型。

```
const int i = 100;
const double di = 100.0;
```

5.3.4 枚举常量

5.3.4.1 为什么需要枚举类型

生活中很多信息，在计算机中都适于用数值来表示，比如，从星期一到星期天，我们可以用数字来表示。在西方，洋人认为星期天是一周的开始，按照这种说法，我们定星期天为 0，而星期一到六分别用 1 到 6 表示。

现在，有一行代码，它表达今天是周 3：

```
int today = 3;
```

很多时候，我们可以认为这已经是比较直观的代码了，不过可能在 6 个月以后，我们初看到这行代码，会在心里想：是说今天是周 3 呢，还是说今天是 3 号？。其实我们可以做到更直观，并且方法很多。

第一种是使用宏定义：

```
#define SUNDAY 0
#define MONDAY 1
#define TUESDAY 2
#define WEDNESDAY 3
#define THURSDAY 4
#define FRIDAY 5
#define SATURDAY 6

int today = WEDNESDAY;
```

第二种是使用常量定义：

```
const int SUNDAY = 0;
const int MONDAY  = 1;
const int TUESDAY = 2;
const int WEDNESDAY = 3;
const int THURSDAY = 4;
const int FRIDAY = 5;
const int SATURDAY = 6;

int today = WEDNESDAY;
```

第三种方法就是使用枚举。

枚举也是我们将学习的第二种用户自定义数据类型的方法。上回我们学过 typedef。typedef 通过为原有的数据类型取一别名来获得一新的数据类型。枚举类型的原有数据只能是 int 或 char 类型（有些编译器则只支持 int,如 VC）。 枚举类型是在整数的取值范围中，列出需要的个值作为新数据类型的取值范围。

这就像 bool 类型，其实它是 char 类型，但它只需要 0 或 1 来代表 false 或 true。

这里的例子中，我们用整型来表示周一到周日。整型的取值范围是多少来的？反正很大，可我们只需 0 到 6，并且我们希望这 7 个值可以有另外一种叫法，以直观地表示星期几。

5.3.4.2 定义枚举类型的基本语法

```
enum 枚举类型名 {枚举值 1，枚举值 2，…… };
```

enum ： 是定义枚举类型的关键字。
枚举类型名 ：我们要自定义的新的数据类型的名字。
枚举值 ：可能的个值。

比如：

```
enum Week {SUNDAY,MONDAY,TUESDAY,WEDNESDAY,THURSDAY,FRIDAY,SATURDAY};
```

这就定义了一个新的数据类型：Week。

Week 数据类型来源于 int 类型（默认）。

Week 类型的数据只能有 7 种取值，它们是：SUNDAY,MONDAY,TUESDAY……SATURDAY。

其中 SUNDAY = 0,MONDAY = 1……SATURDAY = 6 。也就是说，第 1 个枚举值代表 0 ，第 2 个枚举值代表 1 ，这样依次递增 1 。

不过，也可以在定义时，直接指定某个或某些枚举值的数值。比如，对于中国人，可能对于用 0 表示星期日不是很好接受，不如用 7 来表示星期天。这样我们需要的个值就是 1,2,3,4,5,6,7。可以这样定义：

```
enum Week {MONDAY = 1,TUESDAY,WEDNESDAY,THURSDAY,FRIDAY,SATURDAY,SUNDAY};
```

我们希望星期一仍然从 1 开始，枚举类型默认枚举值从 0 开始，所以我们直接指定 MONDAY 等于 1 ，这样，TUESDAY 就将等于 2 ，直接到 SUNDAY 等于 7 。

枚举值，我们就称为格举常量，因为它一经定义以后，就不可再改变，以下用法是错误的！

```
TUESDAY = 10; //错误！我们不能改变一个枚举值的数值。
```

用枚举常量来完成表达今天是星期三：

```
Week today = TUESDAY;
```

再举一例。在计算机中，颜色值也是使用整数来表示的。比如红色用 255 表示，绿色用 65280 表示，而黄色则用 65535 表示。（在自然界中，红 ＋ 绿 ＝ 黄，在计算机中也一样，你注意到了吗?）

假设我们在交警队工作，需要写一个有关红绿灯的程序，我们可以这样定义一个颜色的枚举类型：

```
enum TLightColor { Red = 255, Green = 65280, Yellow = Red + Green };
```

5.3.4.3 关于枚举常量的输出效果

bool 类型，其实就是个地道的枚举类型。我们已经在前面程序中试过它的输出效果：false 输出 0，true 输出 1。假设我们有一个 TLightColor 类型的变量：

```
TLightColor redLamp = Red;
```

然后我们输出 redLamp：

```
cout << redLamp << endl;
```

很多学员可能会猜出，输出结果是 “255” 。 显然他们比笔者聪明。当时我学习枚举类型，就天真地认为屏幕会打出“Red”。（呵呵，别嘲笑我，其实，有时候，比起聪明来，天真更难得……）

我不在课程里给出如何定义，如何使用，如何输出枚举常量的例程了。我想，学到今天，你应该有兴趣，也有能力自己新建一个控制台工程，然后自己写代码实现这一切了。

别发愣，歇上 10 分钟，你是要再看看本章课程，还是启动 CB，写个例程，自己决定吧。

第六章　二进制、八进制、十六进制

这是一节“前不着村后不着店”的课。不同进制之间的转换纯粹是数学上的计算。不过，你不必担心会有么复杂，无非是乘或除的计算。

生活中其实很多地方的计数方法都多少有点不同进制的影子。

比如我们最常用的 10 进制，其实起源于人有 10 个指头。如果我们的祖先始终没有摆脱手脚不分的境况，我想我们现在一定是在使用 20 进制。

至于二进制……没有袜子称为 0 只袜子，有一只袜子称为 1 只袜子，但若有两袜子，则我们常说的是：1 双袜子。

生活中还有：七进制，比如星期。十六进制，比如小时或“一打”，六十进制，比如分钟或角度……

6.1 为什么需要八进制和十六进制?

编程中，我们常用的还是 10 进制……必竟 C/C++是高级语言。

比如：

```
int a = 100,b = 99;
```

不过，由于数据在计算机中的表示，最终以二进制的形式存在，所以有时候使用二进制，可以更直观地解决问题。

但，二进制数太长了。比如 int 类型占用 4 个字节，32 位。比如 100，用 int 类型的二进制数表达将是：

```
0000 0000 0000 0000 0110 0100
```

面对这么长的数进行思考或操作，没有人会喜欢。因此，C,C++ 没有提供在代码直接写二进制数的方法。

用 16 进制或 8 进制可以解决这个问题。因为，**进制越大，数的表达长度也就越短**。不过，为什么偏偏是 16 或 8 进制，而不其它的，诸如 9 或 20 进制呢？

2、8、16，分别是 2 的 1 次方，3 次方，4 次方。这一点使得三种进制之间可以非常直接地互相转换。8 进制或 16 进制缩短了二进制数，但保持了二进制数的表达特点。在下面的关于进制转换的课程中，你可以发现这一点。

6.2 二、八、十六进制数转换到十进制数

6.2.1 二进制数转换为十进制数

二进制数第 0 位的权值是 2 的 0 次方，第 1 位的权值是 2 的 1 次方……
所以，设有一个二进制数：0110 0100，转换为 10 进制为：
下面是竖式：

0110 0100 换算成 十进制

第 0 位 0 * 2^0 = 0
第 1 位 0 * 2^1 = 0
第 2 位 1 * 2^2 = 4
第 3 位 0 * 2^3 = 0
第 4 位 0 * 2^4 = 0
第 5 位 1 * 2^5 = 32
第 6 位 1 * 2^6 = 64
第 7 位 0 * 2^7 = 0 ＋

100

用横式计算为：
0 * 2^0 + 0 * 2^1 + 1 * 2^2 + 1 * 2^3 + 0 * 2^4 + 1 * 2^5 + 1 * 2^6 + 0 * 2^7 = 100
0 乘以多少都是 0，所以我们也可以直接跳过值为 0 的位：
1 * 2^2 + 1 * 2^3 + 1 * 2^5 + 1 * 2^6 = 100

6.2.2 八进制数转换为十进制数

八进制就是逢 8 进 1。
八进制数采用 0～7 这八数来表达一个数。
八进制数第 0 位的权值为 8 的 0 次方，第 1 位权值为 8 的 1 次方，第 2 位权值为 8 的 2 次方……
所以，设有一个八进制数：1507，转换为十进制为：
用竖式表示：

1507 换算成十进制。

第 0 位 $7 * 8^0 = 7$

第 1 位 $0 * 8^1 = 0$

第 2 位 $5 * 8^2 = 320$

第 3 位 $1 * 8^3 = 512$ ＋

839

同样，我们也可以用横式直接计算：

$7 * 8^0 + 0 * 8^1 + 5 * 8^2 + 1 * 8^3 = 839$

结果是，八进制数 1507 转换成十进制数为 839

6.2.3 八进制数的表达方法

C,C++语言中，如何表达一个八进制数呢？如果这个数是 876,我们可以断定它不是八进制数，因为八进制数中不可能出 7 以上的阿拉伯数字。但如果这个数是 123、是 567，或 12345670，那么它是八进制数还是 10 进制数，都有可能。

所以,C,C++规定，**一个数如果要指明它采用八进制，必须在它前面加上一个 0**，如：123 是十进制，但 0123 则表示采用八进制。这就是八进制数在 C、C++中的表达方法。

由于 C 和 C++都没有提供二进制数的表达方法，所以，这里所学的八进制是我们学习的，CtC++语言的数值表达的第二种进制法。

现在，对于同样一个数，比如是 100，我们在代码中可以用平常的 10 进制表达，例如在变量初始化时：

```
int a = 100;
```

我们也可以这样写：

```
int a = 0144; //0144 是八进制的 100；一个 10 进制数如何转成 8 进制，我们后面会学到。
```

千万记住，用八进制表达时，你不能少了最前的那个 0。否则计算机会通通当成 10 进制。不过，有一个地方使用八进制数时，却不能使用加 0，那就是我们前面学的用于表达字符的“转义符”表达法。

6.2.4 八进制数在转义符中的使用

我们学过用一个转义符'\'加上一个特殊字母来表示某个字符的方法，如：'\n'表示换行(line)，而'\t'表示 Tab 字符，'\''则表示单引号。今天我们又学习了一种使用转义符的方法：转义符'\'后面接一个八进制数，用于表示 ASCII 码等于该值的字符。

比如，查一下第 5 章中的 ASCII 码表，我们找到问号字符（?)的 ASCII 值是 63，那么我们可以把它转换为八进值：77，然后用 '\77'来表示'?'。由于是八进制，所以本应写成 '\077'，但因为 C,C++规定不允许使用斜杠加 10 进制数来表示字符，所以这里的 0 可以不写。

事实上我们很少在实际编程中非要用转义符加八进制数来表示一个字符，所以，6.2.4 小节的内容，大家仅仅了解就行。

6.2.5 十六进制数转换成十进制数

2 进制，用两个阿拉伯数字：0、1；

8 进制，用八个阿拉伯数字：0、1、2、3、4、5、6、7；

10 进制，用十个阿拉伯数字：0 到 9；

16 进制，用十六个阿拉伯数字……等等，阿拉伯人或说是印度人，只发明了 10 个数字啊？

16 进制就是逢 16 进 1，但我们只有 0~9 这十个数字，所以我们**用 A，B，C，D，E，F 这五个字母来分别表示 10，11，12，13，14，15**。字母不区分大小写。

十六进制数的第 0 位的权值为 16 的 0 次方，第 1 位的权值为 16 的 1 次方，第 2 位的权值为 16 的 2 次方……

所以，在第 N（N 从 0 开始）位上，如果是是数 X （X 大于等于 0，并且 X 小于等于 15，即：F）表示的大小为 X * 16 的 N 次方。

假设有一个十六进数 2AF5，那么如何换算成 10 进制呢？

用竖式计算：

2AF5 换算成 10 进制:

第 0 位： 5 * 16^0 = 5
第 1 位： F * 16^1 = 240
第 2 位： A * 16^2 = 2560
第 3 位： 2 * 16^3 = 8192 ＋

10997

直接计算就是：
5 * 16^0 + F * 16^1 + A * 16^2 + 2 * 16^3 = 10997
(别忘了，在上面的计算中，A 表示 10，而 F 表示 15)

现在可以看出，所有进制换算成 10 进制，关键在于各自的权值不同。

假设有人问你，十进数 1234 为什么是 一千二百三十四？你尽可以给他这么一个算式：
1234 = 1 * 10^3 + 2 * 10^2 + 3 * 10^1 + 4 * 10^0

6.2.6 十六进制数的表达方法

如果不使用特殊的书写形式，16 进制数也会和 10 进制相混。随便一个数：9876，就看不出它是 16 进制或 10 进制。

C，C++规定，**16 进制数必须以 0x 开头**。比如 0x1 表示一个 16 进制数。而 1 则表示一个十进制。另外如：0xff,0xFF,0X102A,等等。其中的 x 也也不区分大小写。(注意：0x 中的 0 是数字 0，而不是字母 O)

以下是一些用法示例：

```
int a = 0x100F;
int b = 0x70 + a;
```

至此，我们学完了所有进制：10 进制，8 进制，16 进制数的表达方式。最后一点很重要，C/C++中，10 进制数有正负之分，比如 12 表示正 12，而-12 表示负 12，；但**8 进制和 16 进制只能用达无符号的正整数**，如果你在代码中里：-078，或者写：-0xF2,C,C++并不把它当成一个负数。

6.2.7 十六进制数在转义符中的使用

转义符也可以接一个 16 进制数来表示一个字符。如在 6.2.4 小节中说的 ’?’ 字符，可以有以下表达方式：

```
’?’       //直接输入字符
’\77’     //用八进制，此时可以省略开头的 0
’\0x3F’ //用十六进制
```

同样，这一小节只用于了解。除了空字符用八进制数 ’\0’ 表示以外，我们很少用后两种方法表示一个字符。

6.3 十进制数转换到二、八、十六进制数

6.3.1 10 进制数转换为 2 进制数

给你一个十进制，比如：6，如果将它转换成二进制数呢？

10 进制数转换成二进制数，这是一个连续除 2 的过程：
把要转换的数，除以 2，得到商和余数，
将商继续除以 2，直到商为 0。最后将所有余数倒序排列，得到数就是转换结果。

听起来有些糊涂？我们结合例子来说明。比如要转换 6 为二进制数。

“把要转换的数，除以 2，得到商和余数”。
那么：
要转换的数是 6， 6 ÷ 2，得到**商是 3，余数是 0**。 （不要告诉我你不会计算 6÷3！）

“将商继续除以 2,直到商为 0……”
现在商是 3，还不是 0，所以继续除以 2。
那就： 3 ÷ 2, 得到**商是 1,余数是 1**。

“将商继续除以 2，直到商为 0……”
现在商是 1，还不是 0，所以继续除以 2。
那就： 1 ÷ 2, 得到**商是 0，余数是 1** （拿笔纸算一下，1÷2 是不是商 0 余 1!）

“将商继续除以 2，直到商为 0……最后将所有余数倒序排列”
好极！现在商已经是 0。
我们三次计算依次得到余数分别是：0、1、1，将所有余数倒序排列，那就是：110 了！

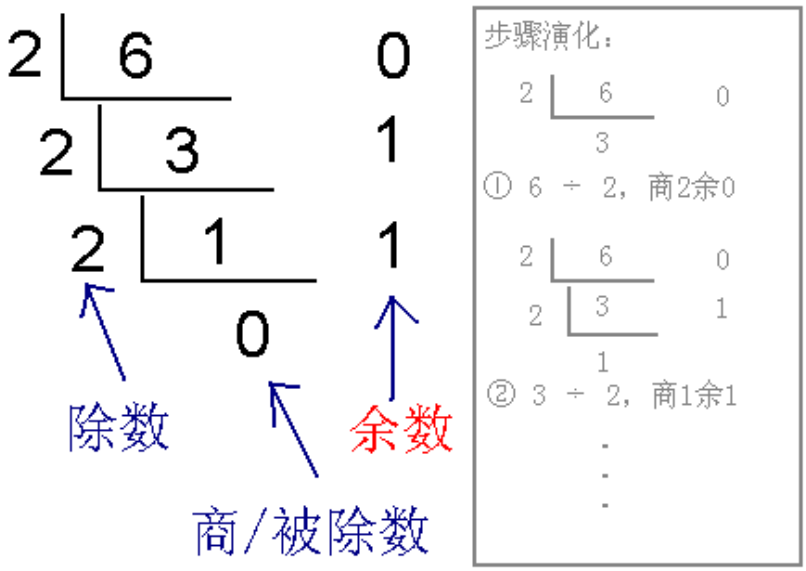
6 转换成二进制，结果是 110。

把上面的一段改成用表格来表示，则为：

被除数	计算过程	商	余数
6	6/2	3	0
3	3/2	1	1
1	1/2	0	1

（在计算机中，÷用 / 来表示）

如果是在考试时，我们要画这样表还是有点费时间，所更常见的换算过程是使用下图的连除：



（图：1）

请大家对照图，表，及文字说明，并且自己拿笔计算一遍如何将 6 转换为二进制数。

说了半天，我们的转换结果对吗？二进制数 110 是 6 吗？你已经学会如何将二进制数转换成 10 进制数了，所以请现在就计算一下 110 换成 10 进制是否就是 6。

6.3.2 10 进制数转换为 8、16 进制数

非常开心，10 进制数转换成 8 进制的方法，和转换为 2 进制的方法类似，惟一变化：除数由 2 变成 8。

来看一个例子，如何将十进制数 120 转换成八进制数。

用表格表示：

被除数	计算过程	商	余数
120	120/8	15	0
15	15/8	1	7
1	1/8	0	1

120 转换为 8 进制，结果为：170。

非常非常开心，10 进制数转换成 16 进制的方法，和转换为 2 进制的方法类似，惟一变化：除数由 2 变成 16。

同样是 120，转换成 16 进制则为：

被除数	计算过程	商	余数
120	120/16	7	8
7	7/16	0	7

120 转换为 16 进制，结果为：78。

请拿笔纸，采用（图：1）的形式，演算上面两个表的过程。

6.4 二、十六进制数互相转换

二进制和十六进制的互相转换比较重要。不过这二者的转换却不用计算，每个 C，C++程序员都能做到看见二进制数，直接就能转换为十六进制数，反之亦然。

我们也一样，只要学完这一小节，就能做到。

首先我们来看一个二进制数：1111，它是多少呢？

你可能还要这样计算：$1 * 2^0 + 1 * 2^1 + 1 * 2^2 + 1 * 2^3 = 1 * 1 + 1 * 2 + 1 * 4 + 1 * 8 = 15$。

然而，由于 1111 才 4 位，所以我们必须直接记住它每一位的权值，并且是从高位往低位记，：8、4、2、1。即，最高位的权值为 $2^3 = 8$，然后依次是 $2^2 = 4$，$^{21}=2$，$2^0 = 1$。

记住 8421，对于任意一个 4 位的二进制数，我们都可以很快算出它对应的 10 进制值。

下面列出四位二进制数 xxxx 所有可能的值（中间略过部分）

仅 4 位的 2 进制数　快速计算方法　十进制值　　十六进值

1111　　= 8 + 4 + 2 + 1　= 15　　F

1110　　= 8 + 4 + 2 + 0　= 14　　E

1101　　= 8 + 4 + 0 + 1　= 13　　D

1100　　= 8 + 4 + 0 + 0　= 12　　C

1011　　= 8 + 4 + 0 + 1　= 11　　B

1010　　= 8 + 0 + 2 + 0　= 10　　A

1001　　= 8 + 0 + 0 + 1　= 10　　9

....

0001　　= 0 + 0 + 0 + 1　= 1　　1

0000　　= 0 + 0 + 0 + 0　= 0　　0

二进制数要转换为十六进制，就是以 4 位一段，分别转换为十六进制。

如(上行为二制数，下面为对应的十六进制)：

1111 1101 ， 1010 0101 ， 1001 1011

F　D　，　A　5　，　9　B

反过来，当我们看到 FD 时，如何迅速将它转换为二进制数呢？

先转换 F：

看到 F，我们需知道它是 15（可能你还不熟悉 A～F 这五个数），然后 15 如何用 8421 凑呢？应该是 8 + 4 + 2 + 1，所以四位全为 1 ：1111。

接着转换 D：

看到 D，知道它是 13，13 如何用 8421 凑呢？应该是：8 + 2 + 1,即：1011。

所以,FD 转换为二进制数，为： 1111 1011

由于十六进制转换成二进制相当直接，所以，我们需要将一个十进制数转换成 2 进制数时，也可以先转换成 16 进制，然后再转换成 2 进制。

比如，十进制数 1234 转换成二制数，如果要一直除以 2，直接得到 2 进制数，需要计算较多次数。所以我们可以先除以 16，得到 16 进制数:

被除数	计算过程	商	余数
1234	1234/16	77	2
77	77/16	4	13 (D)
4	4/16	0	4

结果 16 进制为： 0x4D2

然后我们可直接写出 0x4D2 的二进制形式： 0100 1011 0010。

其中对映关系为：

0100 -- 4

1011 -- D

0010 -- 2

同样，如果一个二进制数很长，我们需要将它转换成 10 进制数时，除了前面学过的方法是，我们还可以先将这个二进制转换成 16 进制，然后再转换为 10 进制。

下面举例一个 int 类型的二进制数：

01101101 11100101 10101111 00011011

我们按四位一组转换为 16 进制： 6D E5 AF 1B

6.5 原码、反码、补码

结束了各种进制的转换，我们来谈谈另一个话题：原码、反码、补码。

我们已经知道计算机中，所有数据最终都是使用二进制数表达。

我们也已经学会如何将一个 10 进制数如何转换为二进制数。

不过，我们仍然没有学习一个负数如何用二进制表达。

比如，假设有一 int 类型的数，值为 5，那么，我们知道它在计算机中表示为：

00000000 00000000 00000000 00000101

5 转换成二制是 101，不过 int 类型的数占用 4 字节（32 位），所以前面填了一堆 0。
现在想知道，-5 在计算机中如何表示？

在计算机中，负数以其正值的补码形式表达。
什么叫补码呢？这得从原码，反码说起。

原码：一个整数，按照绝对值大小转换成的二进制数，称为原码。
比如 00000000 00000000 00000000 00000101 是 5 的 原码。

反码：将二进制数按位取反，所得的新二进制数称为原二进制数的反码。
取反操作指：原为 1，得 0；原为 0，得 1。（1 变 0; 0 变 1）
比如：将 00000000 00000000 00000000 00000101 每一位取反，得 11111111 11111111 11111111 11111010。
称：11111111 11111111 11111111 11111010 是 00000000 00000000 00000000 00000101 的反码。
反码是相互的，所以也可称：
11111111 11111111 11111111 11111010 和 00000000 00000000 00000000 00000101 互为反码。

补码：反码加 1 称为补码。
也就是说，要得到一个数的补码，先得到反码，然后将反码加上 1，所得数称为补码。
比如：00000000 00000000 00000000 00000101 的反码是：11111111 11111111 11111111 11111010。
那么，补码为：
11111111 11111111 11111111 11111010 + 1 = 11111111 11111111 11111111 11111011

所以，-5 在计算机中表达为：11111111 11111111 11111111 11111011。转换为十六进制：0xFFFFFFFB。

再举一例，我们来看整数-1 在计算机中如何表示。
假设这也是一个 int 类型，那么：

1、先取 1 的原码：00000000 00000000 00000000 00000001
2、得反码：　　11111111 11111111 11111111 11111110
3、得补码：　　11111111 11111111 11111111 11111111

可见，－1 在计算机里用二进制表达就是全 1。16 进制为：0xFFFFFF。

一切都是纸上说的……说－1 在计算机里表达为 0xFFFFFF，我能不能亲眼看一看呢？当然可以。利用 C++ Builder 的调试功能，我们可以看到每个变量的 16 进制值。

6.6 通过调试查看变量的值

下面我们来动手完成一个小小的实验，通过调试，观察变量的值。
我们在代码中声明两个 int 变量，并分别初始化为 5 和- 5 。然后我们通过Ｃ Ｂ提供的调试手段，可以查看到程序运行时，这两个变量的十进制值和十六进制值。
首先新建一个控制台工程。加入以下黑体部分（就一行）：

```
//---------------------------------------------------------------------------
#pragma hdrstop
//---------------------------------------------------------------------------
#pragma argsused
int main(int argc, char* argv[])
{
int aaaa = 5, bbbbb = -5;
return 0;
}
//---------------------------------------------------------------------------
```

没有我们熟悉的的那一行：

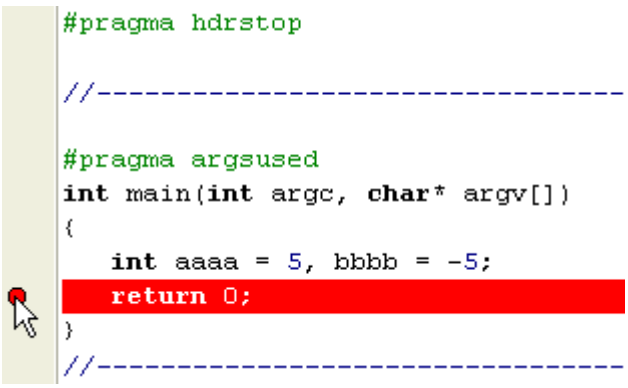
```
getchar();
```

所以，如果全速运行这个程序，将只是ＤＯＳ窗口一闪而过。不过今天我们将通过设置**断点**，来使用程序在我们需要的地儿停下来。

设置断点：最常用的调试方法之一，使用程序在运行时，暂停在某一代码位置，

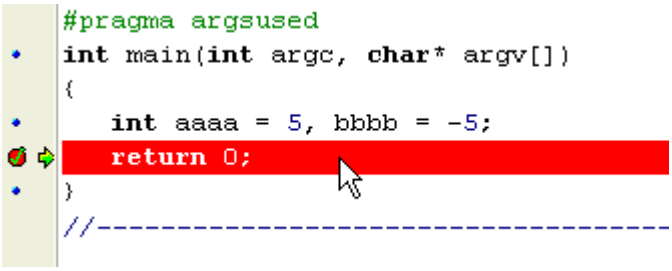
在ＣＢ里，设置断点的方法是在某一行代码上按Ｆ５或在行首栏内单击鼠标。

如下图：



在上图中，我们在 return 0;这一行上设置断点。断点所在行将被ＣＢ以红色显示。

接着,运行程序(F9),程序将在断点处停下来。



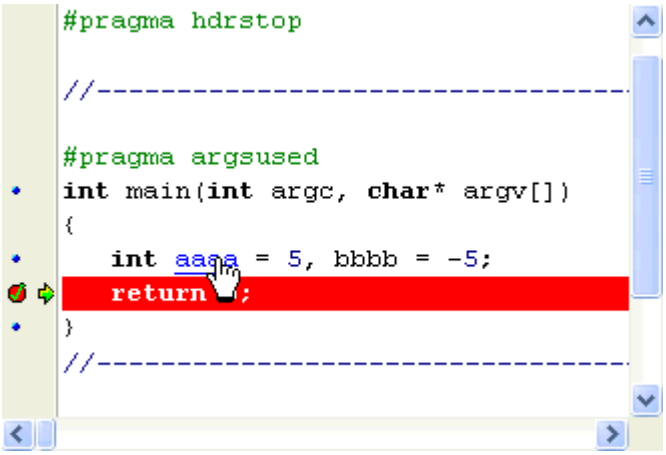
(请注意两张图的不同，前面的图是运行之前，后面这张是运行中，左边的箭头表示运行运行到哪一行)

当程序停在断点的时，我们可以观察当前代码片段内，可见的变量。观察变量的方法很多种，这里我们学习使用 Debug Inspector (调试期检视)，来全面观察一个变量。

以下是调出观察某一变量的 Debug Inspector 窗口的方法：

先确保代码窗口是活动窗口。(用鼠标点一下代码窗口)

按下 Ctrl 键，然后将鼠标挪到变量 aaaa 上面，你会发现代码中的 aaaa 变蓝，并且出现下划线，效果如网页中的超链接，而鼠标也变成了小手状：



点击鼠标，将出现变量 aaaa 的检视窗口：

(笔者使用的操作系统为 WindowsXP,窗口的外观与 Win9X 有所不同)

从该窗口，我可以看到：

aaaa :变量名

int :变量的数据类型

0012FF88:变量的内存地址，请参看 5.2 变量与内存地址；地址总是使用十六进制表达

5 ： 这是变量的值，即 aaaa = 5;

0x00000005 :同样是变量的值，但采用 16 进制表示。因为是 int 类型，所以占用 4 字节。

首先先关闭前面的用于观察变量 aaaa 的 Debug Inspector 窗口。

现在，我们用同样的方法来观察变量 bbbb,它的值为-5,负数在计算机中使用补码表示。

正如我们所想，-5 的补码为：0xFFFFFFFB。

再按一次 F9，程序将从断点继续运行，然后结束。

6.7 本章小结

很难学的一章？

来看看我们主要学了什么：

1)我们学会了如何将二、八、十六进制数转换为十进制数。

三种转换方法是一样的，都是使用乘法。

2)我们学会了如何将十进制数转换为二、八、十六进制数。

方法也都一样，采用除法。

3)我们学会了如何快速的地互换二进制数和十六进制数。

要诀就在于对二进制数按四位一组地转换成十六进制数。

在学习十六进制数后，我们会在很多地方采用十六进制数来替代二进制数。

4)我们学习了原码、反码、补码。

把原码的 0 变 1，1 变 0，就得到反码。要得到补码，则先得反码，然后加 1。

以前我们只知道正整数在计算机里是如何表达，现在我们还知道负数在计算机里使用其绝对值的补码表达。

比如，－5 在计算机中如何表达？回答是：5 的补码。

5)最后我们在上机实验中，这会了如何设置断点，如何调出 Debug Inspector 窗口观察变量。

以后我们会学到更多的调试方法。

第七章 运算符、表达式、语句

我们已经学会如何用变量来表达数据，比如我们现在要写一个《工资管理系统》……我看到很多学员本来昏昏欲睡的眼睛刷地放出了光芒：“老师，我们现在就能写《工资管理系统》系统了吗？”

回答：不能，我们刚刚学了点基础而已。不过，至少我们可以猜想，要写一个工资管理系统，总得懂得如何表达“工资”这个信息吧？还有像职工的年纪，职工人数等等，都得用C / C＋＋语言来表达，这些我们现在都会啊！

```
double gongZi;  //工资
int    nianLin; //年龄
int    zhiGongRenShu; //职工人数
```

看看，上面那些定义变量的代码，你都看得懂，写得出的吧？我们还是颇有成就感的。

本章，我们将学习如何对数据进行运算。

7.1 算术运算符

7.1.1 加减乘除

先来学习最基本也最常用的加减乘除等运算。在C＋＋中，加减乘除分别使用字符 ’+’、’-’、’*’、’/’作为运算符号。

加、减、乘的操作没有什么需要特别说明之处，和生活中的相关运算完全一样，如：

```
int a = 1 + 2 - 3 * 4;
```

得到的结果：a 等于 -9。当然，乘号使用*表示，这你得记。

除运算除了使用反余杠’/’表示以外，很重要的一点是别忘了，对于整数类型，或字符类型的数据进行除运算时，小数部分将被截掉，因为整型类型的数据不能保存小数部分，如：

```
int a = 5 / 2;
```

得到结果：a 等于 2，而不是 2.5。

注意：可能大家会以为，之所以 5 /2 结果是 2,是因为我们让一个整型变量 a 等于它，其实原因并不

是因为 a 是 int 类型，请看：

```
float a = 5 / 2;
```

虽然 a 现在被声明为实型，但执行这句程序，a 的值仍然是 2。事实上，精度丢失是在计算机计算 5/2 时就发生了。所以，准确的写法是：

```
float a = 5.0 / 2;
```

或者：

```
float a = 5 / 2.0;
```

或者：

```
float a = 5.0 / 2.0;
```

也就是说，只有除数或被除数至少需要有一个是明确指定为实型，除运算才能得到小数部分。这里我们也更明确类似于 5 和 5.0 在计算机中的区别：虽然数值大小一样，但加了 5.0 被当成实型数对待，而 5 则被当成整型数。

7.1.2 求模运算

除了 + - * / 以外，% 操作也是 C++常用的操作符。% 并不是进行“百分比”的运算。在 C 和 C++里，% 进行求余数运算，求余数也称“求模”，以下是求余操作的例子：

```
int a = 5 % 2;
```

结果是，a 等于 1,即：5 除以 2，余数为 1 。

7.1.3 赋值运算

差点忘了，我们已经很熟悉的等号：= ，C,C++称为赋值操作。看看例子，是不是很熟悉：

```
int a = 10;
```

再如：

```
int b;
b = a;
```

或：

```
int c = 12 * 2;
```

在Ｃ＋＋中，可以使用连等操作：

```
int a ,b;
a = b = 100;
```

结果是，a 和 b 都为 100。

7.1.4 自运算

先来看一个计算机编程中常有的语句例子：

```
int a = 10;
a = a + 1;
```

上面的代码执行后，结果 a 的值是 11。可能不是很理解 a = a + 1;这种运算。

首先可能会认为， a 和 a + 1 怎么会相等呢？这可是个严重错误，要知道，在 C,C++里，'=' 就是表示赋值操作，至于表示左右两值“相等” 的另有符号。因此，a = a + 1，所完成的工作就是：先计算出 a + 1 的值，然后将该值赋给 a 。

假设我们的存款原为 a ，现在存入 1 元，那么新的存款就等于旧存款加上 1 元钱，用编程语言表达，就是 a = a +1;

在 C，C＋＋中，这样的自加操作可以有另一种表达，并且用这一种表达，计算机的运算速度比较快。

a = a +1;的另一种运算速度较快的表达方法：

a += 1;

+= 被定义为一种新的操作符（因此+和=要连着，中间不能有空格）。它实现的操作就是将其左边的量在自身的基础上加上右边表达式的值。比如：，假设 a 原来的值为 10,那么：

a += 2;

执行这一句后，a 的值为 12,即 a = 10 + 2;

同样的，减，乘，除，求余都有这种操作符：-= 、 *= 、/= 、%= 等。我们以后学习到的另外一些运算符，也有同样的这种对应运算。举一些例子：

假设在分别进行以下操作之前，a 原值都为 10。

a -= 2;

执行后， a 值为 8；(a = 10 - 2)

a *= 2;

执行后， a 值为 20；(a = 10 * 2)

a /= 2;

执行后，a 值为 5；(a = 10 / 2)

a %= 2;

执行后，a 值为 0；(a = 10 % 2)

C，C＋＋提供这些操作符，目的仅仅是为了提高相应操作的运算速度。为什么 a += 2;会比 a = a + 2;运算得快呢？从编译的角度上看，是因为前者可以生成更短小的汇编代码。

C，C＋＋提供这些别的语言没有的操作符，可以供我们写出优化的代码。

在某些特殊情况下，优化还可以继续。请看下一小节。

7.1.5 ++ 和 -- 运算

当运算是自加或自减 1 的时候，C，C＋＋提供了更为优化的运算操作符：++,--。

设整型变量 a，原值为 10。我们已经知道，要实现对其加 1 ，可以有以下两种写法：

方法 1： a = a +1;

方法 2： a += 1;

我们还知道方法 2 比方法 1 好。现在还有方法 3 ，并且是最好的方法。

++a，或者：a++;

也就是说，在只自加 1 的情况下，代码 a++ 或 ++a 可以生成最优化的汇编代码。

同样，自减 1 操作也有对应的操作符：--a　或　a--;

设 a 原值为 10，则执行 --a 或者 a--后， a 的值都为 9 。

现在来谈谈 ++a 和 a++　有什么区别。

在Ｃ，Ｃ＋＋语言里，++a　和 --a　称为前置运算(prefix)，而 a++ 和 a--称为后置运算(postfix)。

如果仅仅是进行前置工或后置运算，那么结果是相同的，这我们已经在前面谈过，我们以++为例：设 a 为 1 0 ，则无论是　++a 或 a++，执行结果都是让 a 递增 1 ，成为 11。

但在有其它运算的复杂表达式中，前置++运算过程是：先加 1 ，然后将**已加 1** 的变量参以其它运算。后置++的运算过程是：先用**未加 1** 的变量参以其它运算，然后再将该变量加 1 。

听起来有些绕，我们举些例子看，还是变量 a ，原来值为 10：

例子 1 ：

```
int b = ++a; //前置＋＋
```

运算结果： a 的值为 11， b 的值也为 11。

计算过程解析：

先计算 ++a,结果 a 值为 11;

然后再计算 b = a;结果 b 值也为 11。

例子 2 ：

```
int b = a++;  //后置＋＋
```

运算结果： a 的值为 11，但 b 的值为 10。

计算过程解析：

先计算 b = a;因此，b 的值是未加 1 之前的 a，所以为 10;

然后再计算 a++, a 的值为 11。

再举一复杂点的表达式：

```
int a = 10;
int c = 5;

int b = a++ + c;
```

执行这些代码，b 值为 15

倘若换成：

```
int a = 10;
int c = 5;
int b = ++a + c;
```

执行这些代码，b 值为 16;

想一想，为什么？

上在举的是++的例子，对于--操作，其计算顺序的规定一样。

++和--的确能加快运算速度，但它们在前置和后置运算上的微小区别，却很空易让你的代码变得不清晰。更为不好的是，不同的编译器可能会对比有不同的解释，比如 VC 和 BC/CB 会对同一代码会有不同的

编译结果，造成代码的运行结果也不一样，这是我们应该尽量避免的。所以我们建议在代码尽量不要依赖于前置和后置运算的区别。（尽管它会让你的代码看上去很象“高手”所写）。

7.2 算术类型转换

7.2.1 隐式类型转换

类型转换在C,C++中也属于一种运算。

前面我们举过一个例子：

```
float a = 5 / 2 ;
```

还记得a的计算结果吗？上式中，a将得到的值是2。因为在除式 5/2 中，5和2都是整数，所以计算机按整数类型进行了除运算，结果所有的小数位都被丢失了。

我们列出了三种可以解决小数位丢失的方法：

```
方法1： float a = 5.0 / 2;
方法2： float a = 5 / 2.0;
方法3： float a = 5.0 / 2.0;
```

最后一种方法好理解，5.0和2.0都明确指定为实型(double)，所以计算结果可以保存小数位。而像第一种：被除数5.0被指定的为实型，但除数2仍然为整型，这两种数据类型的精度不一样，这时，计算机将按哪个类型作为标准呢？

C++遇到两种不同数据类型的数值进行运算时，会将某个数做适当的类型转换，然后再进行转换。转换总是朝表达能力列强的方向进行，并且转换总是逐个运算符进行的。

以下是转换的两条方向线：

```
char/unsigned char --> short/unsigned short --> int/unsigned int --> double --> long double
                                                    float          --> double --> long double
```

像上面的 a = 5 / 2。计算机先计算5/2,由于 5,2一样是整型，所以计算机不作转换，算出其结果为2，然后赋值给a，因此，就算a是float类型，但仍然只能得到2的结果，而不是2.5。

而 a = 5.0 / 2。计算机在计算 5.0 / 2 时，发现5.0是实型(带小数点)，而2是整型，二者不一，所以将2先自动转换成double数,然后现和5.0进行除运算。

这个转换过程，在程序运行时自动进行，称为隐式转换。

隐式（自动）转换尽量使用我们的程序更加合理，但有时它并不能完全符合我们的要求。比如：

```
int b = 5, c = 2;
float a = b / c;
```

由于除号两边的操作数：b、c都是有明确类型的变量。这时，既不会有隐式转换进行，我们也不能通过加 '.0' 来改变其中某个数的数据类型：

```
float a = b.0 / c.0; //这种写法是错误的，不可能实现。
```

这种情况下，我们需要显式（强制）类型转换。

7.2.2 显式类型转换

显式类型转换也称为强制类型转换。它的语法形式有两种：
形式 1 ：(类型名) 变量或数值
形式 2： 类型名(变量或数值)

实际例子如：
```
int b = 5, c = 2;
float a = (float)b / c;
```
或者：
```
float a = float(b) / c;
```
两种写法都可以将变量 b 强制转换为 float 类型。
不过，在要转换的数据类型带有修饰符时，则必须使用第一种型式。比如：(unsigned int) a;
其实，两边都加上括号有时更清晰：(unsigned int) (a);

7.3 关系运算

“关系”运算？听上去很费解。
计算机系一师哥师妹正在处朋友，某晚两人在校园林荫处正在谈情说爱，突然冒出一校监：“说！你俩什么关系？”
果然不愧为计算机系的一对小情侣，以下是他们的回答：
男：“我比她高！”
女：“我比他瘦。”
男：“我比她壮！”
女：“我比他美。”
校监：“我倒！”

所谓的关系运算，在 C,C++语言里，就是比较运算。
算术运算所得的结果是数值，而**关系运算所得的结果为逻辑值，也称布尔值**。即我们以前所学的 bool 类型允许的值：真或假。真用 true 表示，假用 false 表示。
关系操作符有：
== (比较左右值是否相等)
> (比较左值是否大于右值)
>= (比较左值是否大于或等于右值，也称为不小于)
< (比较左值是否小于右值)
<= (比较左值是否小于或等于右值，也称为不大于)
!= (比较左右值是否不相等)

比较是否相等，使用两个连写的等号表示。因此 == 和 = 是两个不同的概念，后者指赋值运算。
C,C++的不等于用 != 表示，而不是我们更常见的 <>，请大家注意。
下面举一些例子。

int a = 10;
int b = 9;
则:
a == b+1 运算结果: true;
a == b 运算结果: false;
a > b 运算结果: true
a >= b 运算结果: true;
b > a 运算结果: false;
a >= b+1 运算结果: true;
a <= b+1 运算结果: true;
a != b; 运算结果：true;

7.4 逻辑运算

逻辑运算有三个操作符：
! （非，取逻辑反，NOT）
&& (逻辑与，并且，AND)
|| (逻辑或，或者，OR)

该到考验你是否有资质学习编程的时候了……请回答以下三个问题：

1、真 并且 真，结果是真是假？
2、真 并且 假，结果是真是假？
3、真 或者 假，结果是真是假？

三道题的答案分别是：真，假，真。
&& 对应于题目的的“并且”,而 || 对应 “或者”
当我们需要说，只有条件 A 和条件 B 都成立……用 C,C++表达即是： A && B。
当我们需要说，只要条件 A 或者条件中成立……，这时表达为： A || B。

如果你是一个女学员，追求者太多让你烦，那么你就可以开出如下条件：

告示：
凡追我者，请先看以下程序代码（如看懂，请先点击此处到‘没有弯路，编程摇篮’报名进修），判断自己是否符合条件，条件符合者，方可得以见本小姐一面：
a = 富和盖茨有一比;
b = 帅胜德华留三分;
见面条件： a && b;
和比尔盖茨一样富**并且**和刘德华一样帅，这样的人有吗？不管你嫁得出去退不出去，现在你必须牢牢记住：
表达式：条件 1 && 条件 2，其运算结果要为真(true)，必须条件 1 和条件 2 都为真。换言之，二者

中有一个为假，那么整体条件就为假。

那么，如果换成或者呢？我们来看看：
a = 富和盖茨有一比;
b = 帅胜德华留三分;
见面条件： a || b;
事情发生了巨大的变化。现在，有个丑八怪，但却和比尔盖茨一样富，你就得会会他老人家。另外有一穷光蛋，却貌如潘安（假设潘安比德仔帅点），按照条件，你也得去见见。因为你列的条件是：a 或者 b，只要其一成立，总体条件即成立。

最后我们来说说“！”。感叹号在C,C++拿来表达“相反”或“非、不”等意思。它的运行很简单：原来为真，加感叹号后为假，原来为假，加感叹号后为真。

下表列出三种逻辑操作符的使用方法：

符号	意思	例子
&&	并且 (and)	条件 1 && 条件 2
||	或者 (or)	条件 1 || 条件 2
!	非 （not）	！条件

下面列出了 &&（与） 操作的所有可能条件及结果：

真 && 真 = 真
真 && 假 = 假
假 && 假 = 假

下面列出了 || （或）操作的所有可能条件及结果

真 || 真 = 真
真 || 假 = 真
假 || 假 = 假

以下则为 ！ 操作：
!真 = 假
!假 = 真

尽管课程列出了所有可能性，尽管看上去也就这几话，但大家一定要从骨子里头理解这些，不能管背住来解决问题。当我们写程序时，就会发现天天在和一堆的真假判断打交道，如果这些最基本的逻辑操作你不理解，那么就会给解决复杂的逻辑问题带来麻烦。

7.5 表达式

变量、常量、各种运算符等组成表达式，用于表达一个计算过程。
比如写一个计算圆面积，我们可以如下表达：

area = PAI * R * R;
其中，R 是某一定义的变量，表示半径，而 PAI 我们事先定义的一个值为 3.14 的宏。

PAI * R * R; 是一个表达式；area = PAI * R * R;也是一个表达式。表达式组成了 C,C++语句，而语句组成 C,C++的程序。

简单的如：3 + 2 ，也是一个表达式。

表达式是操作符、操作数和标点符号组成的序列，用于表达一个计算过程。

对表达式的计算，需要考虑其各计算部分的运算优先级，其中最熟悉莫过于我们小学就学过的“括号优先，先乘除后加减”。
下面列出我们已学过的运算符的优先级：

按优先级高低排列的运算符：

级别	运算符	说明
1	( )	括号
2	! +(正号) -(负号) ++ -- sizeof	+,-在这里不是加减，而是指正负号
3	* / %	乘，除，求模（取余）
4	+ -	加减
5	== !=	等于 不等于判断
6	&&	逻辑与
7	||	逻辑或
8	= += *= /= %=	赋值，自运算操作符

7.6 语句

C++语句和表达式并没有来格的区分。一人表达式，加上一个分号后，可以直接形成语句:

3 + 2;

计算机可以执行该语句，但它并不改变程序的运行逻辑。就像我们说话时说了一句废话。当一些表达式组合起来，完成某一相对完整的功能后，再加一个分号表示结束，这就组成一条语句。如：

a = 3 +2;

看，这是一行赋值语句，它改变了 a 的值。

当然，语句也可以直接是一个分号，称为空语句：

```
;
```

除非为了调试方例，否则写一句空语句纯属多余。

我们已经学习过两种常用的语句类型，其一为变量定义语句，其二为赋值语句。

变量定义语句完成指定变量的定义。

```
int a ,b,c;
```

赋值语句则实现为指定变量获得指定值的操作。

```
a = 20;
b = 10 * 2 / 3;
c = 2 * (a + b);
```

如上面所示，赋值时，右值（等号右边的值），可以是一简单的常数或变量，也可以是一个表达式。

在 C,C++中，赋值语句可以使用连等：

```
a = b = 10;
```

执行这一语句时，b 先等于 10，然后 a 等于 b 的值，结果 a 和 b 都是 10。

当然，变量定义语句也可以和赋值语结合,即我们以前学的，在定义变量时同时初始化。

```
int a = 10,b = 20;
int c = a * b;
```

不过，你不能在定义变量时同时使用连等来实现初始化：

```
int a = b = 10; //错误。
```

编译器会报错说，b 还没定义。

有时候，连续的多句语句属于同一控制范围，这时，我们用一对花括号将这些语句括起：

```
{
  int a = 100;
  int b = a * 20;
  ....
}
```

花括号内的内容，称为**复合语句**。

正是一行行语句组成 C,C++程序，结果本章时，我们可以自豪地宣布：我们已经一脚迈入了 C++大门的门槛！

第八章 顺序流程

你喝一杯水，一般是这样：
1， 往杯里倒满开水；
2， 等开水冷却；
3， 往嘴里倒。
从这个生活的例子中，你可以想到，完成事情，总是要有顺序的，并且执行顺序往往还很需讲究。譬如喝水的例子，如果你把第 2 步和第 3 步调序，结局可能会很难受；而如果你想把第 1 步放到最后去执行，大概你将永远也喝不了水。
程序是用来解决现实生活的问题的，所以流程在程序中同样重要。我们已经学习过语句，当我们写下一行行代码时，这些代码必须按照一定次序被执行，这就是程序的流程。

我们先为熟悉最简单的流程：顺序流程。

8.1 顺序流程

笔直的长安街，东西走向，长达 40 公里。
顺序流程就像一条笔直的，没有分叉的路。程序执行完第一行，然后第二行、第三行……。我们这一节课用两个例子来熟悉什么叫顺序流程。
电脑，原称计算机（computer）。你的电脑是奔 III、奔 IV，或其它什么，据说它每秒钟能计算几百几千万次云云，反正就是很厉害。现在，既然我们学编程了，是该到我们亲手出道让计算机为我们算算的时候了。
一道很简单的题，加法。我们准备做两个版本。先来 DOS 版本。

8.1.1 加法计算器（DOS 版）

来生成一个空白的控制台工程，很熟悉了吧？黑体是你要加入的，它们的作用你知道。

```
//---------------------------------------------------------------------------
#include <iostream.h>
#pragma hdrstop
//---------------------------------------------------------------------------
#pragma argsused
int main(int argc, char* argv[])
{
getchar();
return 0;
}
//---------------------------------------------------------------------------
```

现在，我们来加入实现加法计算的代码：

```
int main(int argc, char* argv[])
{
int js1,js2; //加数 1，加数 2
int he;      //和

cout << "请输入第 1 个加数：" << endl;
cin >> js1;

cout << "请输入第 2 个加数：" << endl;
cin >> js2;

he = js1 + js2;   // 和 = 加数 1 + 加数 2
cout << js1 << " + " << js2 << " = " << he << endl;

getchar();
return 0;
}
//------------------------------------------------------------------------
```

上面的代码，你应该都能看懂。不过我还是解释一下（你不妨在输入后按一下 F9，看一下运行结果，可以更有助于理解）。

这段代码从功能上分，可以分为三个部分。

```
int js1,js2; //加数 1，加数 2
int he;      //和
```

这是第一部分，两行代码定义了三个变量：加数 1、加数 2 及和。至于 “//加数 1，加数 2 ”双斜杠到行末的代码，那是注释。也就是写给程序员自己看的“程序说明”。你可以不理它。

第二部分为输入部分，用来输入

```
cout << "请输入第 1 个加数：" << endl;
cin >> js1;

cout << "请输入第 2 个加数：" << endl;
cin >> js2;
```

cout 输出一行提示，告诉用户（现在就是我们自己）做什么。而 cin 则将用户的输入存到变量中。如：

```
cin > js1;
```

这一行执行时，会等待用户输入一个数，直到用户回车后（别忘了，回车结束输入），用户输入的数值会被自动赋值给变量 js1；js1 是“加数 1”的拼音首字母，而非你更熟悉的“奸商 1”或“句神 1 :(”。

最后一部分实现计算和输出。

```
he = js1 + js2;     // 和 = 加数 1 + 加数 2
cout << js1 << " + " << js2 << " = " << he << endl;
```

这段程序的核心代码就行：

```
he = js1 + js2;     // 和 = 加数 1 + 加数 2
```

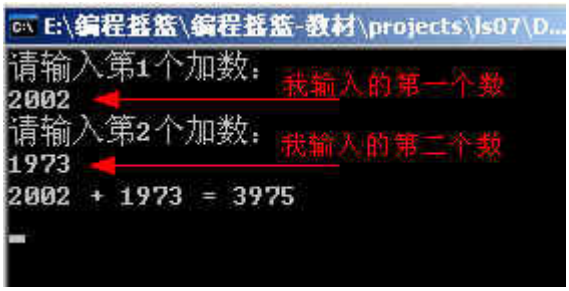
它实现将 js1 和 js2 相加，并附值给 he。

最后一句 cout 将结果输出，你可以只写成这样：

```
cout << he;
```

这样写也把计算结果输出了，但可能会被人说成“用户界面不友好”噢。

来看看我们程序运行时的某种结果。我决定让它计算：2002 + 1973。你要让它算什么？你自已试吧。



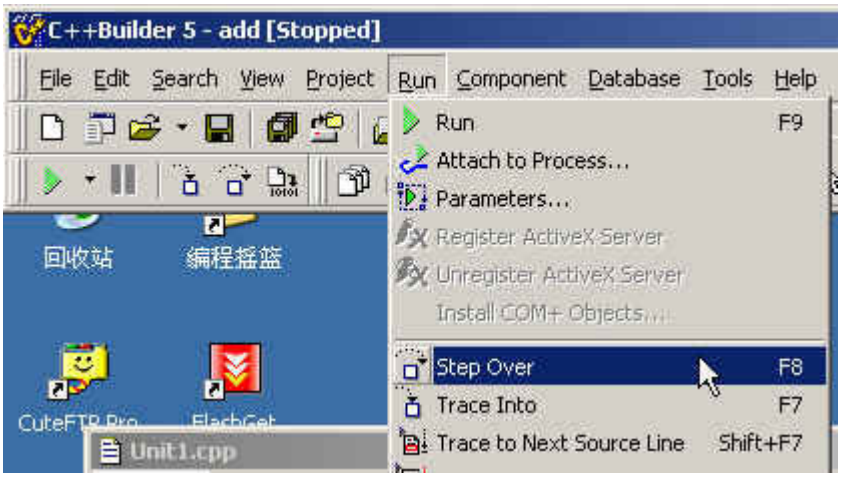
虽然是个很不起眼的小程序，虽然只是一道小学低年级的算术题，可是必竟我们亲手证明了我们的爱机具有计算能力。得意 5 秒钟，我们来继续我们的课程。我将通过**单步运行**来亲眼程序是如何一步一步地按顺序运行的。

如果你在还在运行着程序，回车键关掉那个黑色窗口。

切换回 C++ Builder 的代码窗口。

单步运行是一种最必要的调试方法（其它众多调试方法几乎都基于该方法），它可以让程序按代码行一步步运行。

在 CB 中，通过按 F7 或 F8 键，可以实现单步运行一个完整的代码行（不一定是物理上的一行代码）。F7 和 F8 的功能区别现暂不必理。本章中，我们使用 F8。对应的功能菜单为：



从菜单中看到，F8 对应的功能名称为：Step Over （单步越过）。

按下 F8 后， 程序开始运行，但并没有直接出现结果窗口。相反，代码窗口出现了变化：

```
//---------------------------------------------------------

#pragma argsused
int main(int argc, char* argv[])
{
    int js1,js2;
    int he;

    cout << "请输入第1个加数：" << endl;
    cin >> js1;

    cout << "请输入第2个加数：" << endl;
    cin >> js2;

    he = js1 + js2;      // 和 = 加数1 + 加数2

    cout << js1 << " + " << js2 << " = " << he << endl;

    getchar();
    return 0;
}
//---------------------------------------------------------
```

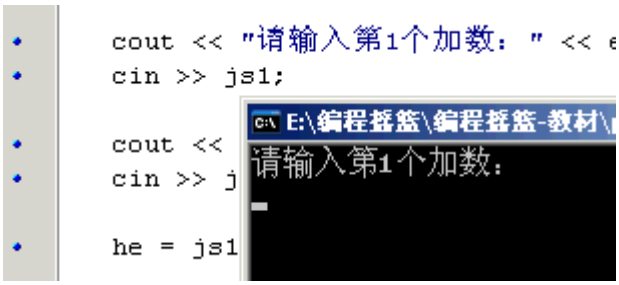
左边栏上的 • 表示该行是一可以单步中断的代码行，而 ➪ 则指示了当前正要运行的代码行，注意，是正要运行，而不是正在运行。

现在让我们再按一次 F8（在代码窗口里），可以看到程序往下走到第二个可中断行。

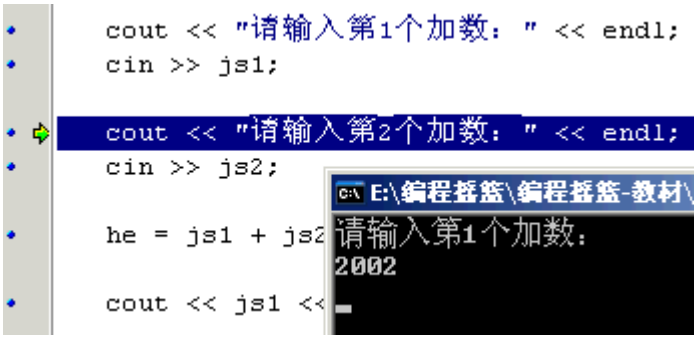
```
    int js1,js2;
    int he;

    cout << "请输入第1个加数：" << endl;
    cin >> js1;

    cout << "请输入第2个加数：" << endl;
```

再按一次 F8，程序又往下走了一行（为了方便观察，我将输出窗口拉近了一并抓下图）。

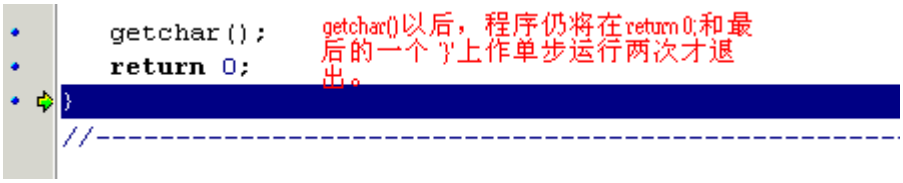
输出窗口（DOS 窗口）有出输出，并且有光标闪烁，但你可以试着在输出窗口里敲敲键盘，会发现你并不能在这一步输入加数 1，因为此时程序运行到 cin >> js1，但并未执行这一行。只有我们再按一下 F8（记得在 CB 的代码窗口里，而不是在 DOS 窗口里），程序将要求并允许我们输入加数 1。如果你还没有再按一次 F8，现在按一下。结果如下：



我们发现代码窗口里暂时没有了，因为**控制权已暂时移交到我们程序**，它现在可以输入了。我们输入 2002，然后回车，很快又出现在代码窗口里，并且，已经越过 cin >> js1 这一行。



接下来是提示输入第二个加数，大同小异。只要你记得当控制权转到程序时，你需要切换到输出窗口输入第二个数。 注意了：最后当程序运行：getchar()，需要我们输入一个回车时，控制权也会移到程序。由于是在单步运行，所以当我们在输出窗口最后敲一个回车时，程序同样不会直接运行到结束。所以你仍得到 CB 的代码窗口里按 F8。



getchar() 以后，程序仍将在 return 0; 和最后的一个 ‘}’ 上作单步运行两次才退出。

如果你懒得这样一步步运行到结束，想结束单步运行，可以直接按 F9，程序将恢复全速运行，直接运行到结束。

就这样，我们一步一步地运行完这个程序。我们学会了如何通过 F8 来单步运行程序，我们也理解了什么叫顺序流程。这个程序中每一个可中断点（事实上也是可执行点），在程序运行时，被依照其先后次序，一一执行。

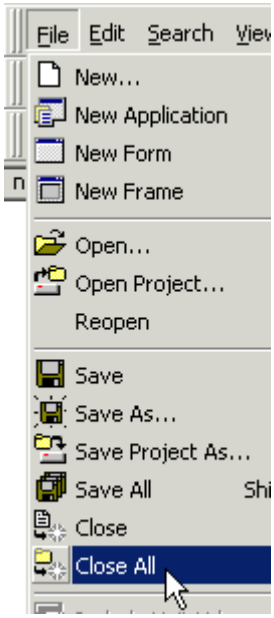
如果你还是不理解什么叫“顺序”，那就从椅子上站起来，然后双眼直直地瞄住一面墙（挑结实一点的），然后直直地走过去，听到“砰”的一声后，你一定会开窍。

（一般不传之秘笈：当程序员觉得被眼前的代码弄得头晕脑胀的时候，大多数人都会起来走走，下楼抽颗烟，或者只是坐坐，吹吹风……最佳方法是挪到不远处某漂亮的女测试员的桌边胡说八道几句，再挪回电脑前，往往发现问题的答案很简单！）

8.1.2 加法计算器（WIN 版）

我们一直在写 DOS 下的程序（控制台程序）。趁今天的课程知识点不多，我们来写一个正宗的 Windows 应用程序。它还是实现一个加法计算器。

不要做得太顺手了！听我说，如果刚才那个 DOS 版你还没保存，请先 Shift + Ctrl + S 保存全部。接着，最好执行一下这个菜单命令：File | Close All。如图：



然后，我们重复一下如何建立一个空白的 Windows 应用程序工程，我们在以前的课程中曾经做过一次，在作业里也考过一次。

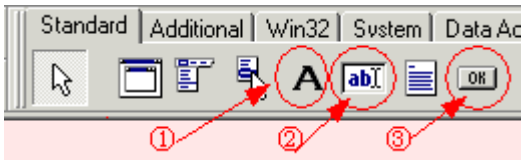
其实很简单。菜单： File | New Applicaton（如果是 CB6,可能略有不同）。也可以按工具栏上的这个图标：

然后选择：Application。

新的空白工程建成以后。我们需要来设计程序界面——毕竟，这是一个有图形用户界面（GUI）的 Windows 程序。

其实很简单，我们需要两个编辑框，让用户输入两个加数。再来一个编辑框用于显示和。还要一个按钮，当用户按下时，加法运算才开始。当然还需要一些标签用于显示提示文字。

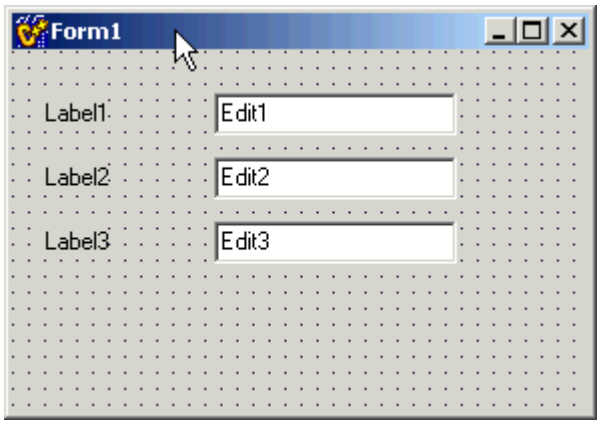
下面先把要用到三种控件在控件栏位置说一下：



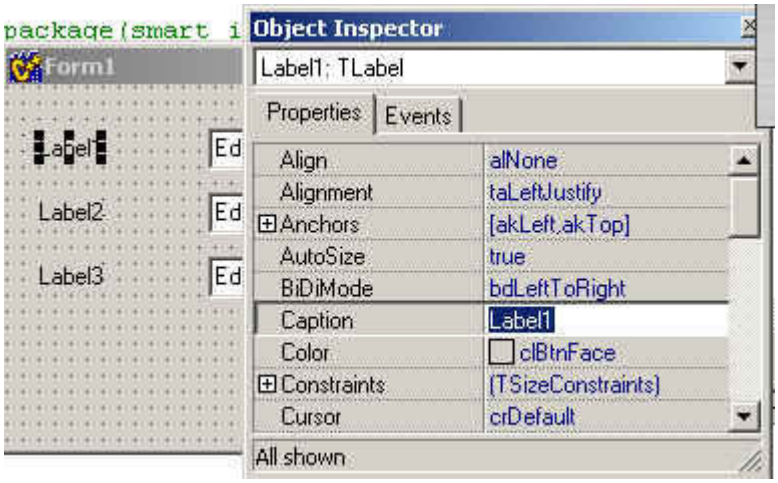
这三个控件都在控件栏的 Standard 页上。上图中，

①：标签控件（TLabel)；②：编辑框控件（TEdit)；③：按钮控件（TButton)。

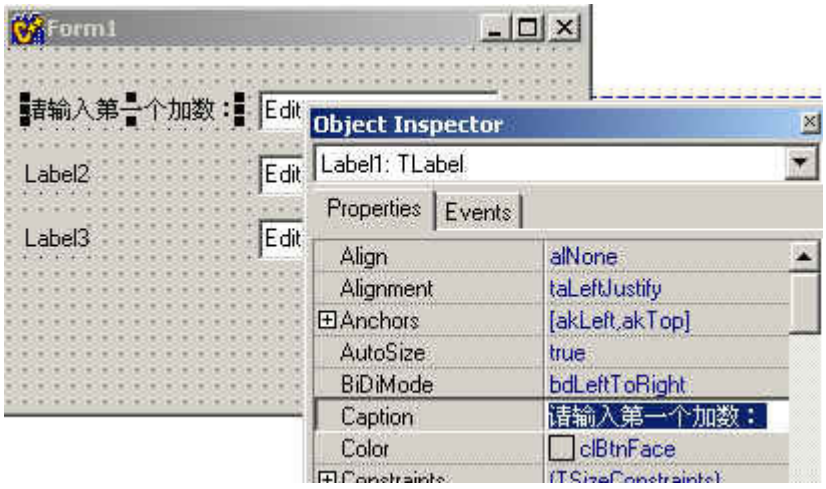
先在表单上放三个标签，三个编辑框：（在控件栏上按下控件图标，然后在表单上单击）



然后，用鼠标单击表单上的 Label1,如果你看不到下面这个窗口（属性检视器），请按 F11。



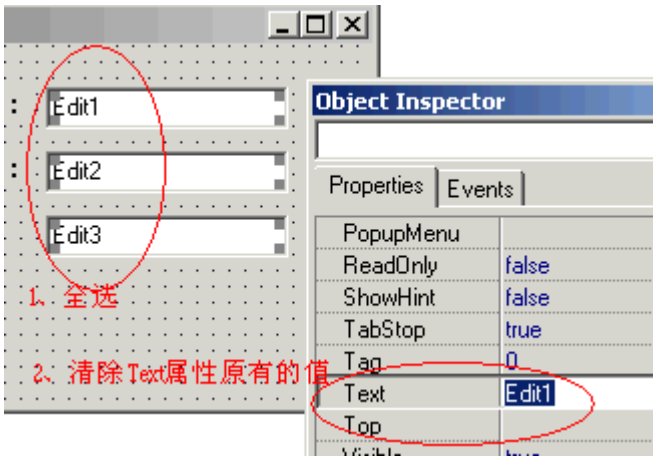
将其 Caption(标题）属性，由原来默认的 Label1 改为：“请输入第一个加数：”如图：



同样的方法，修改 Label2,Label3 的标题，分别为：“请输入第二个加数”和“和”。

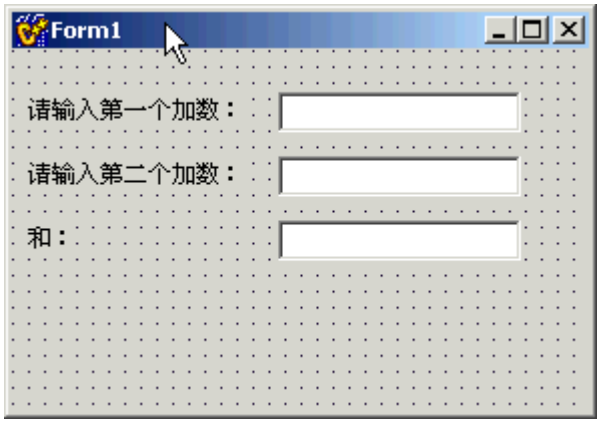
三个编辑框中默认的文本“Edit1”、“Edit2”、“Edit3”我们都不要。编辑框中的文字属于 TEdit 的 Text 属性。让我们一起将它们清除：

首先拉动鼠标，一起选中三个编辑框：

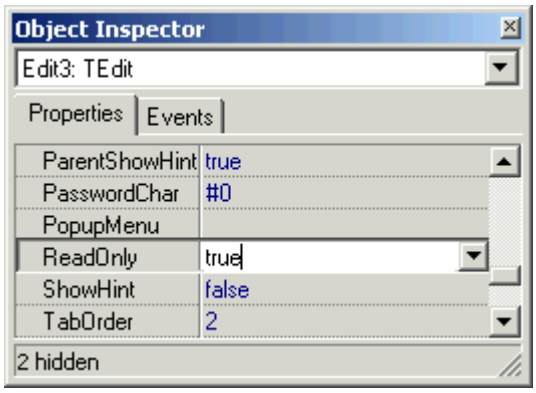


将上图中 Text 属性的值：Edit1 清空。

现在，设计结果如图：

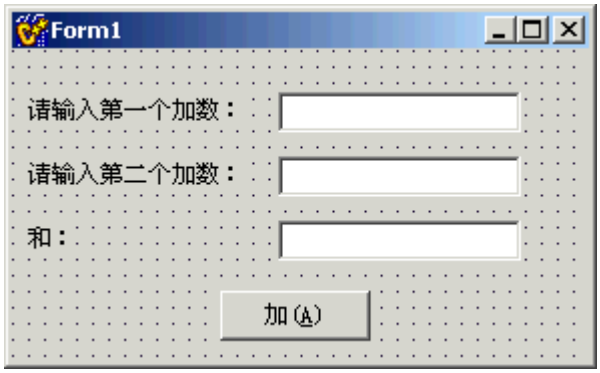


前两个编辑框我们准备让用户输入加数，最后一个编辑框要用来显示相加结果（和），所以最后一个编辑框不能让用户修改。编辑框（TEdit）有一个属性名为 ReadOnly（只读），当它被设置为真（true）时，编辑框的内容不能手工修改。



如上图，将 Edit3 的 ReadOnly 属性改为 true(原来默认为 false)。修改 ReadOnly 的结果并不能直接在设计期间看出。

最后，让我们再加上一个按钮(TButton)，并改变其 Caption 属性为：“加(&A)”



界面设计完成。下面开始代码设计。

双击 “加”按钮，CB 将自动切换到代码窗口，并且自动生成以下代码：

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
 }
```

请加入以下黑体部分的代码：

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
   int js1,js2;
   int he;
   js1 = Edit1->Text.ToIntDef(0);
   js2 = Edit2->Text.ToIntDef(0);
   he = js1 + js2;
   Edit3->Text = IntToStr(he);
}
//---------------------------------------------------------------------------
```

js1 = Edit1->Text.ToIntDef(0);用来得到用户输入到 Edit1 里的数值，你现在可能看不懂，可以不必理会。只要你能看出 js1 = ...这是一句赋值语句。

保存，运行这个程序。我们可以反复输入不同数让程序运算。当然，每次运算前，需正确输入合适的加数，然后按那个按钮。如果你没有输入加数，或者输入的是非法的字符，如："ABC"、"-2-3"等无法转换为数值的内容，则该加数将被当成 0。

以下是运行时的一个界面：

你还可以试着在第三个编辑框里敲敲键盘，会发现的确无法改变其内容。

很开心的一章：我们终于能做些有点意义的小程序了。从身边找一个会电脑，但没有学习编程的家伙，我们大可用这个程序小小的炫耀一番。

不过，现在你也应该能理解，为什么我们在学习 C,C++语言时，我们为什么大多会采用 DOS 下程序来作为例子，因为若使用 Windows 程序，它的界面设计很容易让我们分心。并且，我们还必须面对如 Edit2->Text.ToIntDef(0)这些后面才学到的内容。

第九章 条件分支语句

"to be or not be"?

这个问题深深地困扰着哈姆雷特。他必须在“生存还是毁灭”之间做出一个选择，这是一个困难的选择。

在你的人生中，您曾经面对什么选择？

“学编程还是不学编程”？

“学 CB 还是学 VC”？

选择哪一个，最终总要是要决定，不同的是每个人在作出选择时所要面对的不同条件。前一章我们讲“顺序流程”就好像长安街一样笔直的，从头走到尾；这一章我们要讲的“条件分支流程”，就像是在道路上遇到了分叉，是直行还是右拐？全看程序走到分叉时所碰上的条件。

9.1 if...else 语句

if，中文意思“如果”……

痞子蔡说：“如果把整个太平洋的水倒出，也浇不熄我对你爱情的火”。 多么充满感情的话！当然，这一切仅仅是因为你还没有学过编程。如果你学了编程，学了本章，你就会明白这是一句多么冰凉的条件分支语句，正如它后面的那句所揭示的一切：“整个太平洋的水全部倒得出吗？不行。所以我并不爱你”。来看看 if...else 语句在 C,C++语言中如何使用。

if...else 语法格式：

```
if (条件)
{
  分支一
}
else
{
  分支二
}
```

其中，“条件”由表达式如何，典型的如关系表达式（忘了什么叫“关系表达式”？回头找找讲小情侣遇校监那章）。“分支一”和“分支二”表示当条件成立时和不成立分别要执行语句。用一句表达，

就是：**如果**条件成立（也称条件为真），那么程序执行分支一中的语句，**否则**（条件为假）程序执行分支二的语句。这就是 if...else 语句。

举一生活中的例子：

```
if（我中了这一期体彩的500万大奖）
{
  我买房；
  我买车；
  我去欧洲旅游；
}
else
{
  我要修理自行车；
  我买下一期体彩，
  我继续烧香。
}
```

上面的例子中：

```
{
  我买房；
  我买车；
  我去欧洲旅游；
}
```

这是第一个分支，而：

```
{
  我要修理自行车；
  我买下一期体彩，
  我继续烧香。
}
```

这是第二个分支。到底我将执行哪个分支，依赖于条件：“我中了这一期体彩的500万大奖”是否成立。在梦中，我经常很开心地执行第一个分支，因为那个条件只能在梦中成立。

来一段真实的程序：

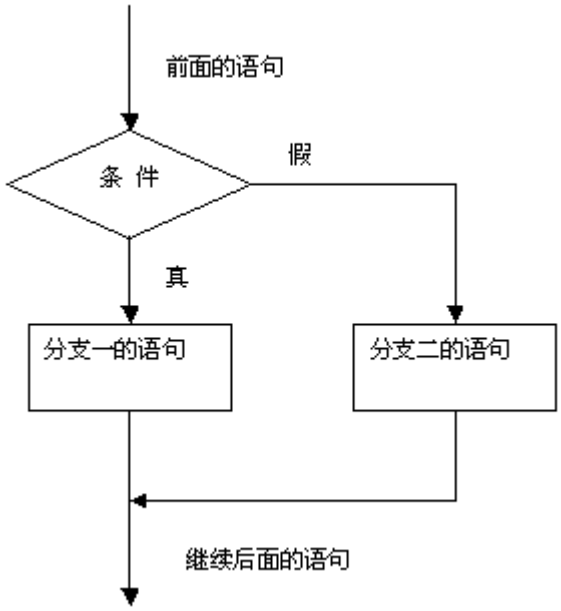
设a,b,c为已定义的三个int变量，有以下代码片段：

```
if (a > b)
{
  c = 1;
}
else
```

```
{
  c = 0;
}
```

若 a 值为 2，b 值为 1，那么执行上面代码,c 的值将为 1;相反，如果 a 值为 1，b 值为 1，由于条件：a > b 不成立，所以程序将执行 else 后面一对 { } 中的代码，结果将是 c 值为 0。

用流程图可以直观在表达程序的执行可能的方向。我们来看 if...else... 的流程图：



箭头表示了程序可能的走向，当遇到条件（菱形）时，根据条件成立的真假，程序将作出选择,是走分支一还是分支二。但无论经过哪个分支，最后都将同样继续后面的代码。

上机题目一：用户输入一个整数，请写一程序，通过 if...else... 判断该数是偶数或奇数。

解题：被 2 整除的整数叫偶数，什么叫整除？就是除了以后余数为 0 啊。还得我们学过一个操作符是用来求两数相除的余数吗？（如果不记得，先自觉到墙角站 10 分钟，然后找第七章复习）。

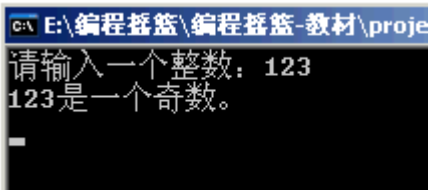
% 操作符用来求两数相除的余数，比如 5 % 2 得到余数 1。那么一个数，比如 a，是否偶数，条件就是 (a % 2) == 0。（对 == 也看着也有点生疏？20 分钟！然后还找第七章）。

打 CB，然后新建一个空白控制台工程。在代码中加下以下黑体部分：

```
//---------------------------------------------------------------------------
#include <iostream.h>
#pragma hdrstop
//---------------------------------------------------------------------------
#pragma argsused
int main(int argc, char* argv[])
{
  int num;
```

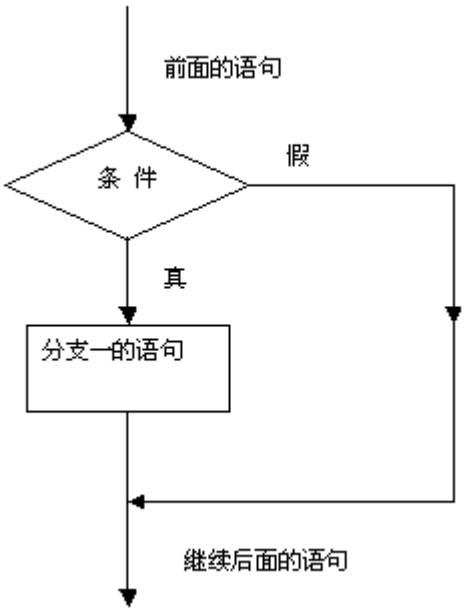
```
    cout << "请输入一个整数：";

    cin >> num;

    if((num % 2) == 0)
       cout << num << "是一个偶数。" << endl;
    else
       cout << num << "是一个奇数。" << endl;

    getchar();
    return 0;
}
//-------------------------------------------------------------------------
```

按 F9 编译并运行程序后，笔者输入 123，以下是屏幕显示结果：



9.2 if... 语句

if...else...中的 else(否则)并不是必须的，因为有时候，当指定条件成立时，我们执行某些动作，否则，我们不执行那些动作。用流程图表示就是：



对比 if...else...的流程图，我们发现 if...语句中，当条件不成立时，将绕过分支一的语句，直接执行后面的代码。

if...语句格式：

```
if（条件）
{
   分支一
}
```

来看这个例子，然后做出你的判断：

```
if（我每天赚１００万）
{
   我每天都将１００成中的９０万分给你。
}
```

看完上面的“程序”，你觉得我是个大方的人吗？看起来是噢，但问题是我永远也无法每天赚１００万，所以关于“我每天都将１００成中的９０万分给你”　整个是在穷开心呵。同样的道理，你自己分析痞子蔡的那句话吧。

上机题目二：用户输入一个字符，用程序判断是否为小写字母，如果是，请输出“您输入的字符是小写字母”。

解题：如何判断一个字符是小写字母？让我们查一下第五章中的 ASCII 码表。在表里头，小写字母（ a ~ z)对应的 ASCII 值为：

97 ~ 122	a ~ z

可见，所有小写字母的值是连续的。那么，判断一个字符是否为小写字母，就看它的值是否大于等于97，并且小于等 122。

假设一个字符变量：　a;要判断它是否“大于等于 97，并且小于等 122”－－

你**不能**这样写：

```
if （ 97 <= a <= 122) //错误!
```

你应该使用 && 来表达"并且"：

```
if (a >= 97 &&  a <= 122)
```

当然，更直观，更不易出错的写法，是直接使用字符，而不是使用字符的ＡＳＣＩＩ值：

```
if (a >= 'a' && a <= 'z')
```

新建一个控制台空白工程。然后在代码中输入以下黑体部分：

```
//---------------------------------------------------------------------------
#include <iostream.h>
#pragma hdrstop
//---------------------------------------------------------------------------
#pragma argsused
```

```
int main(int argc, char* argv[])
{
  char a;
  cout << "请输入一个字符：" ;
  cin >> a;
  if(a >= 'a' && a <= 'z')
     cout << a << "是一个小写字母。" << endl;

  getchar();
  return 0;
}
//--------------------------------------------------------------------------
```

9.3 ？： 表达式

？： 表达式 我们称为 问号冒号表达式。

用 if...else... 和 if...语句，已经可以很好地实现所有条件分支的代码。不过 C 是一门追求简捷高效的语言，它提供的 ？：表达式 来在某种情况下代替 if...else...，起来让代码更简捷的作用。

来看看 if...else...语句在什么情况下可以简化。

首先来看原型：

```
if (条件)
{
  分支一
}
else
{
  分支二
}
```

我们知道，分支一或分支二一般都是一组（多行）语句，用来分别实现条件是否成立时的动作。由于是一组（多行）语句，所以我们有一对{}括在外面，用于形成复合语句。不过，有时候，分支中的语句比较简单，用一句话就可以实现。比如我们所举的例子：

```
if (a > b)
{
  c = 1;
}
else
{
  c = 0;
}
```

在这个例子中，分支一、二都分别只有一条语句。**对于只有一条语句的情况，我们可以省略{}**（除了

在特殊的上下文中外，其它情况下都可以省略。以后我们学习的其它流程控制也一样)，如：

```
if (a > b)
   c = 1;
else
   c = 0;
```

看，没有花括号的代码，感觉多紧凑。不过，对于上面的代码，我们还可以继续简化，那就是使用 ?: 语句。

```
c = (a > b)? 1 : 0;
```

就一行话，多简捷！语句中的问号问的是什么？问的是 a 是否大于 b？如果是，则得到值 1，否则，得到值 0。

? : 表达式格式语法：

(条件) ? 值 1 : 值 2

举一例子：
设有 int 变量 a,b，二者均已初始化。请得到二者中的较大值。

方法是：

```
int c = (a  > b) ?  a : b;
```

是的，就这么简单，执行上面代码，c 将得到 a 和 b 中的较大值。

我们称 (a > b) ? a : b 为一个表达式，而不称它为完整的一个语句（尽管它的确也可以作一个单独的语句存在）。一般地，? : 表达式拿来作为等号的右值，用于通过条件判断确定一个值后，该值被赋予等号的左值。因此，并不是所有的 if..else...语句都可以用 : ? 来代替，只有那些两个分支都只是用来计算一个同一类型的值，然后赋予同一变量的条件分支语句，才适合。

上机题目三：两数取大

新建一个控制台空白工程。然后在代码中输入以下黑体部分：

```
//---------------------------------------------------------------------------
#include <iostream.h>
#pragma hdrstop
//---------------------------------------------------------------------------
#pragma argsused
int main(int argc, char* argv[])
{
  int a,b,c;
```

```
    cout << "请输入两个整数（用空格分开）：" ;
    cin >> a >> b;

    c = (a > b)? a : b;

    cout << c << "大" << endl;

    getchar();
    return 0;
}
//-------------------------------------------------------------------------
```

下面是屏幕输出的结果。我输入的两个数是 102 和 134。

9.4 多级 if...else...语句

不要害怕为什么一个条件分支就有这么多种语句。多级 if..else..语句——只是我这么叫它，其实它完全是由 if..else..语句组成，就好像楼梯，一个阶的楼梯是楼梯，100 个阶的楼梯也是楼梯。

多级 if..else...语法格式：

```
if(条件 1)
{
   分支一
}
else if(条件 2)
{
   分支二
}
else if(条件 3)
{
   分支三
}
else
{
   分支四
```

}

格式中举出四个分支，实际可以只有三个，两个，也可以有更多个。请看下在这个例子。

让我们想像这么一幕情景剧——

时间：XXXX 年 2 月 14 日；地点：某校园内小公园；

人物：女生一，男生一。

（男生送给女生一束环瑰）

女生（高兴地）：多美啊——多少钱买的！

男生：猜。

女生（心里想：如果多于 100 元我就亲他一口）：超过 100 元吗？

男生：NO。

女生：（心里想：如果多于 50 元我就许他亲我一口）：那，不低于 50 元吧？

男生：NO。

女生：（心里想：如果多于 10 元就跟他说声谢谢吧）：那是不低于 10 元了？

男生：NO。

女生：（不再有任何想法，一把丢掉鲜花）：呸！

看明白了吗？

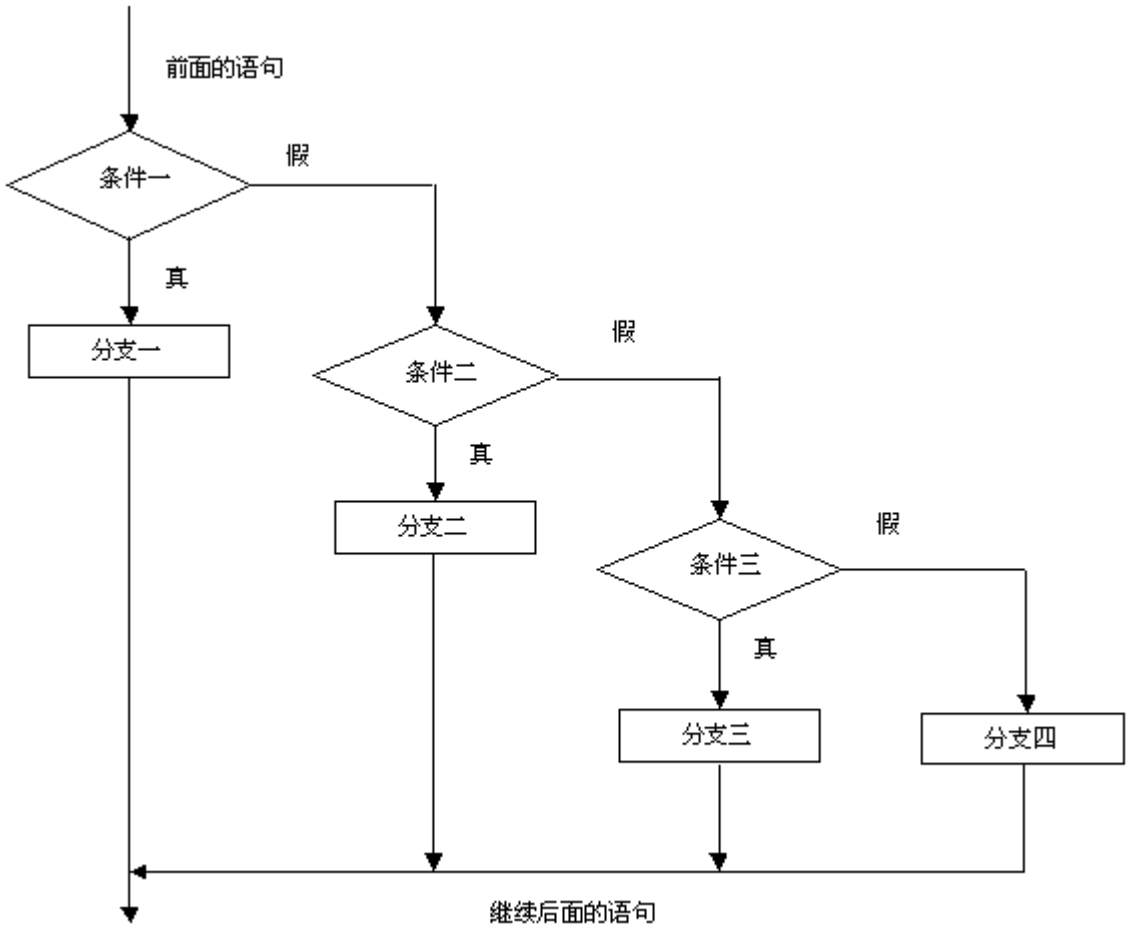
“看明白了……”一个小男生眼泪汪汪地站起来，回答：“我是看明白了，现在的女生没一个是好东西！”#·$&*%!@ 啊?? 我是说，大家看出其中的多级条件分支了吗？

写成程序，便是：

```
int price = 玫瑰花价钱;

if (price > 100)
{
     小女生要亲小男生一口;
}
else if(price > 50)
{
     小女生准备让小男生亲一口;
}
else if (price > 10)
{
    小女生对小男生说声“谢谢”;
}
else
{
    小女生扔掉花，说：“呸！”;
}
```

如果花价大于 1 0 0 元，那么女生亲男生一口；否则呢？否则就再判断花价是否大于 5 0 元，如果还是不大，那不再来判断花价是否大于 1 0 元，如果还是不大，最后无条件地执行“小女生扔掉花……”这一句。

看一下多级 if...else...语句的流程图：



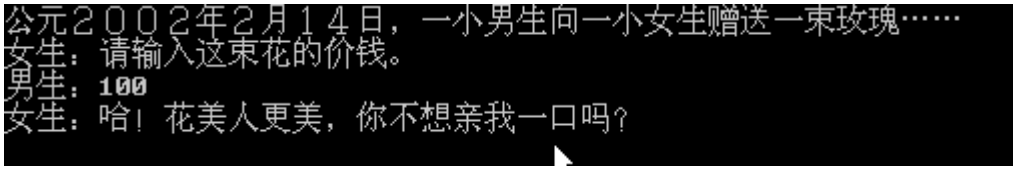
上机题目四：送花待遇，要求根据上面的情景剧，写一程序，用户输入花价后，程序根据不同的花价输出相应的待遇。

新建一控制台工程。

```
//---------------------------------------------------------------------------
#include <iostream.h>
#pragma hdrstop
//---------------------------------------------------------------------------
#pragma argsused
int main(int argc, char* argv[])
{
  int flowerPrice;

  cout << "公元２００２年２月１４日，一小男生向一小女生赠送一束玫瑰……" << endl;
```

```
    cout << "女生：请输入这束花的价钱。" << endl;
    cout << "男生：";
    cin  >> flowerPrice;

    if(flowerPrice > 100)
       cout << "女生：哇！我太喜欢这花了，让我亲你一口以示谢意！" << endl;
    else if (flowerPrice > 50)
       cout << "女生：哈！花美人更美，你不想亲我一口吗？" << endl;
    else if (flowerPrice > 10)
       cout << "女生：谢谢！" << endl;
    else
       cout << "女生：什么破花，大头鬼才会喜欢。" << endl;

    getchar();

    return 0;
}
//-------------------------------------------------------------------------
```

以下是运行结果的一种,我实在买不起１００元以上的花：



9.5 switch 语句

多级 if...else...显然是为了那些可能需要进行多级判断才能做出选择的情况。如前面的花价。如果正好是大于１００元，那么只需判断一次，女生就可以做出决定，但如果是７元钱，那就必须经过“是否大于１００？是否大于５０？是否大于１０？”三次判断。

Ｃ为了简化这种多级判断，又提供了 switch 语句。

switch 语句的格式：

```
switch ( 整型或字符型变量 )
{
   case 变量可能值１ :
        分支一;
        break;
   case 变量可能值２ :
        分支二;
        break;
```

```
    case 变量可能值3 ：
        分支三;
        break;
    ...
    default :
        最后分支;
}
```

在 switch 的语法里，我们要学到4个关键字：switch、case 、break、default。

在 switch（ 变量 ） 这一行里，变量只能是整型或字符型。程序先读出这个变量的值，然后在各个"case"里查找哪个值和这个变量相等，如果相等，就算条件成立，程序执行相应的分支，直到碰上break或者switch语句结束。

说起来有点抽象。我们结合一个实例看看。

实例模拟一个网上调查。它要求网友输入数字以选择自己是如何到达当前网页。然后，程序根据网友这一输入，打出不同的结果。

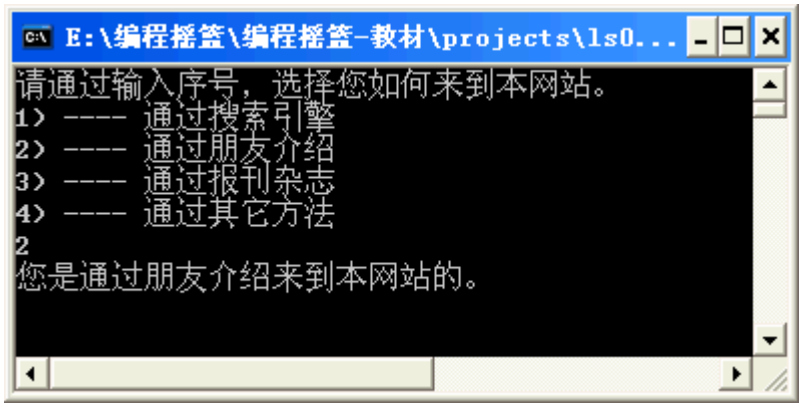
上机题目五：模拟网络调查，要求输出不同的选项，供网友选择（通过简单地输入数字），程序根据网友的选择，输出相应不同的结果。

请新建一控制台空白工程，然后在代码里输入以下黑体部分。

```
//---------------------------------------------------------------------------
#include <iostream.h>
#pragma hdrstop
//---------------------------------------------------------------------------
#pragma argsused
int main(int argc, char* argv[])
{
  int fromWay;

  cout << "请通过输入序号，选择您如何来到本网站。" << endl;
  cout << "1) ---- 通过搜索引擎" << endl;
  cout << "2) ---- 通过朋友介绍" << endl;
  cout << "3) ---- 通过报刊杂志" << endl;
  cout << "4) ---- 通过其它方法" << endl;

  cin >> fromWay;

  switch (fromWay)
  {
    case 1 :
```

```
        cout << "您是通过搜索引擎来到本网站的。" << endl;
        break;
    case 2 :
        cout << "您是通过朋友介绍来到本网站的。" << endl;
        break;
    case 3 :
        cout << "您是通过报刊杂志来到本网站的。" << endl;
        break;
    case 4 :
        cout << "您是通过其它方法来到本网站的。" << endl;
        break;
    default :
        cout << "错误的选择！请输入 1 ～ 4 的数字做出选择。" << endl;
  }

  getchar();

   return 0;
}
```

以下是我运行的一个结果，我输入的是 2 。



对照输入结果，我们讲一讲这段代码，其中主要目的是要弄明白 switch 语句的用法。

首先，int fromWay 定义了一个整型变量，准备用来存储用户输入的选择。

```
cout << "请通过输入序号，选择您如何来到本网站。" << endl;
cout << "1) ---- 通过搜索引擎" << endl;
cout << "2) ---- 通过朋友介绍" << endl;
cout << "3) ---- 通过报刊杂志" << endl;
cout << "4) ---- 通过其它方法" << endl;
```

这些语句输出提示和选择项，结果如上图。

cin >> fromWay; 这一句则负责等待用户输入，并且将用户的输入存储到 fromWay。

接下来程序遇上了 switch 语句：

```
switch (fromWay)
```

程序将根据 fromWay 值，在下面的各个 case 中找到匹配的值。 本例中 switch 带有四个 case，值分别是 1 、 2 、 3 、 4 。在结果图中，由于我输入的是 2 ，所以程序进入下面这个 case:

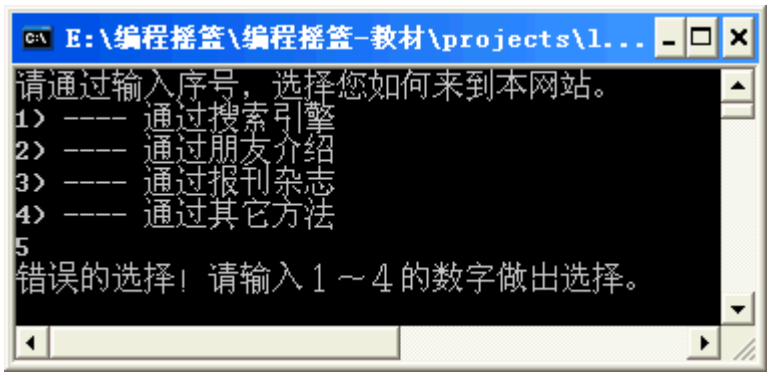
```
case 2 :
    cout << "您是通过朋友介绍来到本网站的。" << endl;
    break;
```

程序打出"您是通过朋友介绍来到本网站的。" 这行内容。最后遇上 **break;**于是跳出整个 switch 语句。

同样的道理，如果用户输入的是其它的数，如 1 或 3 ，则会进入相应的 case 1 或 case 3 分支。

但如果用户输入的数在所有的 case 里都找不到时，会怎么样？比如，用户输入 5 。这种情况下，default 分支将起作用，所有 case 里的值都不匹配时，switch 进入 default 分支。如果连 default 也没有，那么程序在 switch 语句里什么也不做，直接完成 switch 语句。

我们来看一下如果用户不按所列的情况输入，而造成的结果：



了解一下 switch,case,break，default 的意思，对理解前面的一切也会有帮助，它们分别是：开关，情况，中断，默认（值）。那么用一句话套起来的说法就是：根据开关值的不同，执行不同的情况，直到遇上中断；如果所有的情况都不符合开关值，那么就执行默认的分支。

最后说一下关于 switch 中非常重要的几个注意点。

第一、switch （ 整型或字符型变量 ） 中，**变量的类型**如文中所标，**只能是整型和字符类型。**它们包含 int,char。当然无符类型或不同的长度整型(unsigned int,short,unsigned char)等都可以。另外，枚举类型(enum)内部也是由整型或字符类型实现。所以也可以。实型（浮点型）数就不行，如：

```
float a = 0.123;
switch(a) //错误！a 不是整型或字符类型变量。
{
  ....
}
```

第二、case 之后可以是直接的常量数值，如例中的 1 、 2 、 3 、 4 ，也可以是一个使用常量计算式，

如 2 +2 等，但不能是变量或带有变量的表达式，如 a * 2 等。当然也不能是实型数，如 4.1，或 2.0 / 2 等。

```
switch(formWay)
{
   case 2-1 :  //正确
     ...
   case a-2 :  //错误
     ...
   case 2.0 :  //错误
     ...
}
```

另外，在 case 与常量值之后，需要一个冒号，请注意不要疏忽。

第三、break 的作用。

break 使得程序在执行完选中的分支后，可以跳出整个 switch 语句（即跳到 switch 接的一对 {} 之后），完成 switch。如果没有这个 break，程序将在继续前进到下一分支，直到遇到后面的 break 或者 switch 完成。

比如，假设现在程序进入 case 1: 中的分支，但 case 1 的分支这回没有加 break:

```
case 1 :
    cout << "您是通过搜索引擎来到本网站的。" << endl;
case 2 :
    cout << "您是通过朋友介绍来到本网站的。" << endl;
```

那么，程序在输出 "您是通过搜索引擎来到本网站的。" 之后，会继续输出 case 2 中的 "您是通过朋友介绍来到本网站的。" 。

请大家将前面实例中的代码片段改为如下(红色部分，即将所有的 break 都通过加//使之无效。)：

```
    ...
    case 1 :
       cout << "您是通过搜索引擎来到本网站的。" << endl;
       //break;
    case 2 :
       cout << "您是通过朋友介绍来到本网站的。" << endl;
       //break;
    case 3 :
       cout << "您是通过报刊杂志来到本网站的。" << endl;
       //break;
    case 4 :
       cout << "您是通过其它方法来到本网站的。" << endl;
       //break;
    default :
       cout << "错误的选择！请输入 1 ～ 4 的数字做出选择。" << endl;
    ...
```

运行后，结果会是如何？请大家动手试试，然后在作业中回答我。

第四、default 是可选中，前面我们已经说过它的用处，及如果没有 default，程序在找不到匹配的 case 分支后，将在 switch 语句范围内不做什么事，直接完成 switch。大家也可以在实例中将 default 的代码注释掉，然后试运行，并且在选择时输入 5 。

```
...
//default :
   //cout << "错误的选择！请输入 1 ～ 4 的数字做出选择。" << endl;
...
```

第五、必要时，可在各个 case 中使用{}来明确产生独立的复合语句。

前面我们在讲 if...语句和其它流程控制语句时，都使用{}来产生复合语句：

```
if (条件)
{
  分支一;
}
```

除非在分支中的语句正好只有一句，这里可以不需要花括号{}。但在 switch 的各个 case 语句里，我们在语法格式上就没有标出要使用 { } ，请看：

```
switch ( 整型或字符型变量 )
{
   case 变量可能值 1 ：
        分支一;
        break;
   case 变量可能值 2 ：
   ....
}
```

一般教科书上只是说 case 分支可以不使用 { } ，但这里我想提醒大家，并不是任何情况下 case 分支都可以不加{}，比如你想在某个 case 里定义一个变量：

```
switch (formWay)
{
   case 1 :
     int a=2;  //错误。由于 case 不明确的范围，编译器无法在此处定义一个变量。
     ...
   case 2 :
     ...
}
```

在这种情况下，加上 { } 可以解决问题。

```
switch (formWay)
{
   case 1 :
```

```
    {
      int a=2;  //正确，变量 a 被明确限定在当前 { } 范围内。
      ...
    }
    case 2 :
      ...
}
```

由于本注意点涉及到变量的作用范围，所以你如果看得不是很明白，可以暂时放过。

第六、switch 并不能代替所有有 if..else...语句。这一点你应该知道了，因为前面已说过，它在对变量做判断时，只能对整型或字符型的变量做判断。另外，switch 也只能做“值是否相等”的判断。你不能在 case 里写条件：

```
switch (i)
{
  case (i >= 32 && i<=48) //错误！  case 里只能写变量的可能值，不能写条件。
     ...
}
```

在这种情况下，你只能通过 if...else 来实现。

9.6 小结

这一节课我们学了所有条件分支的流程控制语句。在编程的时候，我们选择什么语句呢？嗯，这是一个问题。幸好，这个问题不像丹麦王子哈姆雷特面对的那样困难。你所要做的就是**多看几遍本章课程、多做几次本章的各个短小简单的例程。**从基础做起。然后，我们会安排一个完整的，适合我们现在水准的小项目进行实战。

也许你害怕做选择、也许你正在为难于某个选择、也许你一直为某个错误的选择而后悔……
但如果生活真的没有了选择……或者
如果我们所要经历一切选择都只能由别人做出决定……
这样的生活肯定不精彩。
编程也一样。学好条件选择的流程，你就能写一些精彩的程序！
所以庆贺一下吧，就在这一章的内容，我们学完了所有的条件分支的流程控制语句！要知道，无论是什么编程语言，也都只有两大流程控制！
多加努力！下一章《循环语句》见！

第十章　循环语句

循环就是反复。

生活中，需要反复的事情很多。譬如你我的整个人生，就是一个反复，反复每一天的生活，直到死，幸好，我们每天的生活并不完全一个样。

10.1 while 循环

语法形式：

```
while(条件)
{
  需要循环执行的语句;
}
```

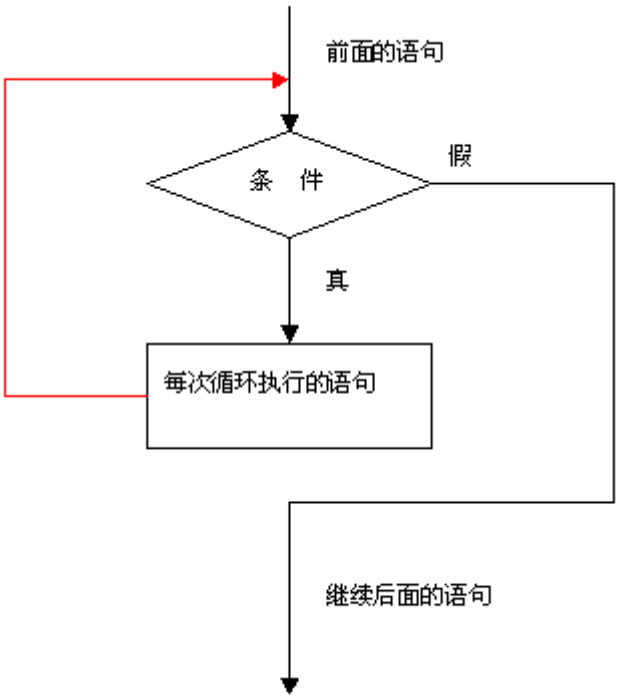
while 是“当”的意思。

请首先和 if 语句作一个比较：

```
if(条件)
{
  条件成立时执行的语句;
}
```

二者除了关键字不一样以外，结构完全一样。但一定要注意，在条件成立时，if 语句仅仅执行一遍，而 while 语句则将反复执行，直到条件不再成立。

请看 while 循环的流程图：



程序从“前面的语句”开始执行，然后进行条件判断，如果条件成立，则执行一次“每次循环执行的语句”，再后请特别注意红色部分，这是我们碰上的，第一次会往后走流程：红线就像汽车拐弯，掉头到条件处（并不包括前面的语句），然后再进行下一次的条件判断……直到某一次判断时条件不成立了，程序“继续后面的语句”。

我们用 while 的语法套用生活中的实际例子，可以直观地看出 while 的用法。

假设有一个爱哭的小娃娃，有一天她要求父母给买一条小红裙，可惜父母不同意，于是她就开始一个循环：

```
while ( 父母不给买小红裙)
{
    我哭;
}
```

这段“代码”的意思是：当“父母不给买小红裙”，那么，小女孩就一遍一遍地哭。

这就是我们和循环流程的第一个遭遇战。所举的例子看似直观：“小孩一遍遍地哭，直到父母给买裙”，但真正要用程序的语言来正确地表达出来，需要很多方面要考虑到，必竟，程序是严谨的。

首先，**一个合适的判断是否继续的条件相当重要。**小女孩要继续哭，仅仅“父母不给买小红裙”，这显示不符合事实，想想我们小时候，再会哭，最终也有累的时候，所以，要想继续哭，我们的条件有两个：“父母不给买小红裙”并且“我还没有哭累”。

```
while ( 父母不给买小红裙 && 我还没有哭累)
{
    我哭;
}
```

其次，大多数情况下，**条件需要被恰当地改变。**小女孩在不停地哭，那么她如何知道父母是否买了红裙呢？所以，她不能只顾哭，还得在哭的间隙观察大人是否同意买裙。至于是否哭累，我们假设小女孩

有一个疲劳度，每哭一次疲劳度加 1，当疲劳度到达 200 时，可怜的小女孩累了……

```
while(父母不给买小红裙 && 疲劳度 < 200)
{
  我哭;
  我偷看爸妈是否同意买裙;
  疲劳度++;
}
```

例一： 用 while 语句实现求从 1 到 100 的累加和。

求 1＋2 的和，我们可以写 a = 1 + 2;求 1 加到 100，我们当然可以这样写 a = 1 + 2 + 3 + ... 100.不过这样写显然太累人了，要从 1 写到 100 啊！所以聪明如高斯的你，当然也知道这样写：a = (1+100) * 50;这确实是个在任何时候都值得称赞的，又快又简的方法，只是今天我们想让计算机累一点，老老实实地从 1 加到 100。首先用我们先学的 while 式的循环。

请同学们打开 CB，然后新建一空白的控制台程序，在 main()函数体加入下面黑体部分代码。然后按 F9 运行。查看运行结果以加深印象。

```
//---------------------------------------------------------------------------
#include <iostream.h>
#pragma hdrstop
//---------------------------------------------------------------------------
#pragma argsused
int main(int argc, char* argv[])
{
int sum = 0; //变量 sum 将用于存储累加和，将它初始化为 0，这很重要。
int i = 1; //i 是每次要加的数，它从 1 开始。

while ( i<= 100)
{
   sum += i;
   i++;
}

//输出累加结果：
cout << "1 到 100 的累加和为：" << sum << endl;
getchar();
}
```

sum 初始为 0，然后在每一遍的循环里，它都加上 i，而，i 则每次都在被加后，增加 1。最终，i 递增到 101，超过 100 了，这个循环也就完成了任务。

运行上面程序，输出结果为：

1 到 100 累加和为为：5050

例二：用 while 循环实现简单的统计功能

统计功能在各行业里都经常用到，比如学校学生成绩总分的统计，商店中每日销售额的统计等。下面我们实现一个学生成绩的统计。

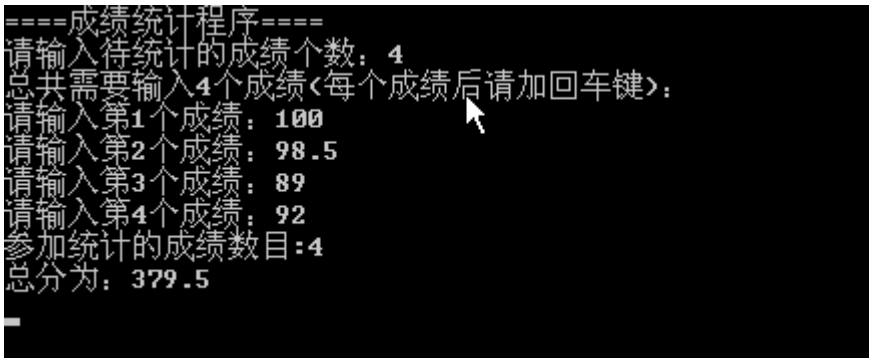
由于成绩中包含有 80.5 这样的需要小数的部分，所以我们使用实数类型。

保存，然后关闭上面的工程，然后再新建一个控制台工程。在主函数 main 内加入以下黑体部分的代码：

```
//---------------------------------------------------------------------------
#include <iostream.h>
#pragma hdrstop
//---------------------------------------------------------------------------
#pragma argsused
int main(int argc, char* argv[])
{
   float sum,score;
   int num; //num 用于存储有几个成绩需要统计。
   int i;   //i 用于计数

   //初始化：
   sum = 0;
   i = 1;

   cout << "====成绩统计程序====" << endl;
   //用户需事先输入成绩总数：
   cout << "请输入待统计的成绩个数：";
   cin >> num;
   cout << "总共需要输入"<< num << "个成绩(每个成绩后请加回车键)：" << endl;

   while ( i <= num)
   {
      cout << "请输入第" << i << "个成绩：";
      cin >> score;
      sum += score;
      i++;
   }

   //输出统计结果:
   cout << "参加统计的成绩数目:" << num << endl;
   cout << "总分为：" << sum << endl;
```

```
    getchar();
}
//---------------------------------------------------------------------------
```

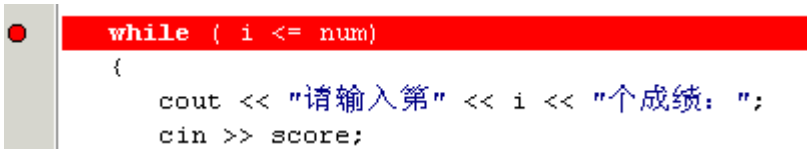
以下是运行结果，我输入 4 个成绩参加统计：



回车结束上面的程序。稍作休息。

为了更直观地了解循环流程，现在我们来跟踪这段程序中的 while 循环。

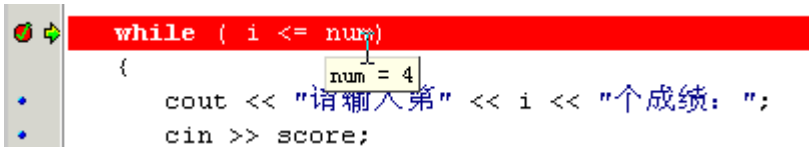
1、首先在循环开始处设置断点（F5 功能）：



2、按 F9 运行程序，在 DOS 窗口提示“请输入待统计的成绩个数：”时，输入 4，并回车。

3、程序将在一瞬间运行到第一步设置的断点所在行。即 while(...)这一行。

此时请鼠标挪到 i 上，稍等片刻，出现提示 “i=1”,同样的方法可以观察 num 的值。

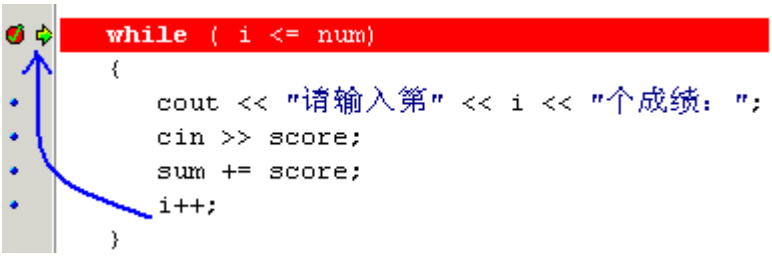


可见第一遍循环时，i = 1，num = 4，条件：i <= num 显然成立，循环得以继续。

4、按 F8，程序往下运行一行，接着再按 F8，程序要求输入一个成绩，请切换到 DOS 窗口，随便输入一个数，并回车。

回车后，程序运行到下图中蓝底的一行：

5、之后，连续按 F8，你将发现程序“回头”运行到 while(...)这一行。此时，i=2, i <= num 条件仍然成立，如果您想再跟踪一遍循环，请继续按 F8，如果想结束跟踪，在断点行上再按一次 F5 以取消断点，然后按 F9，程序恢复全速运行。



（程序往回走，回到 while 行）

10.2 do ... while 循环

语法形式：

```
do
{
    需要循环执行的语句;
}
while(条件);
```

和 while 循环最明显的区别，就是 do...while 循环中，判断是否继续循环的条件，放在后面。也就是说，就算是条件一开始就不成立，循环也要被执行一次。请比较以下两段代码，前者使用 while 循环，后者使用 do...while 循环。

代码段一：

```
int a = 0;
while( a > 0 )
{
  a--;
}
```

变量 a 初始值为 0,条件 a > 0 显然不成立。所以循环体内的 a--;语句未被执行。
本段代码执行后，变量 a 值仍为 0;

代码段二：

```
int a = 0;
do
{
  a--;
}
while( a > 0 );
```

尽管循环执行前，条件 a > 0 一样不成立，但由于程序在运行到 do...时，并不先判断条件，而是直接先运行一遍循环体内的语句：a--。于是 a 的值成为 -1，然后，程序才判断 a > 0 ，发现条件不成立，循环结束。

do..while 中的条件和 while 循环中的条件一样是：“允许继续循环的条件”，而不是“结束循环的条件”，这和 Pascal 语言中的 do...until 正好相反，学习过 Pascal(Delphi)的学员可得注意。

以笔者的经验，do..while 循环用得并不多，大多数的循环用 while...来实现会更直观。下面我们仅简单地将 1 到 100 的连加程序转换为用 do...while 实现：

```
int sum =0;
int i=1;

do
{
   sum += i;
   i++;
}
while(i<=100);
```

例三：用 do...while 实现可以多次统计的程序。

在例二中，我们做了一个统计程序。假如一个学生有三门成绩，如语文，数学，英语要统计总分，例二的程序可以方便地使用，但如果要连续统计一个班级每个学生的这三门成绩，我们就得不断地运行例二的成绩，这显然不方便。

一个同学的三门成绩需要一层循环，要不断统计多个同学各自的成绩，就需要再套上一层循环。请看下面例子中，如何在原来的 while...循环上再加一层 do...while 循环。

程序的思路是：统计完一遍后，就问一句是否要继续统计新同学的成绩，如果用户输入字母 Y 或 y，表示需要统计一下位，否则，程序结束循环。

这个程序是在例二的基础上进行功能改进，以下粗体部分为新加的代码。

```
//---------------------------------------------------------------------------
#include <iostream.h>
#pragma hdrstop
```

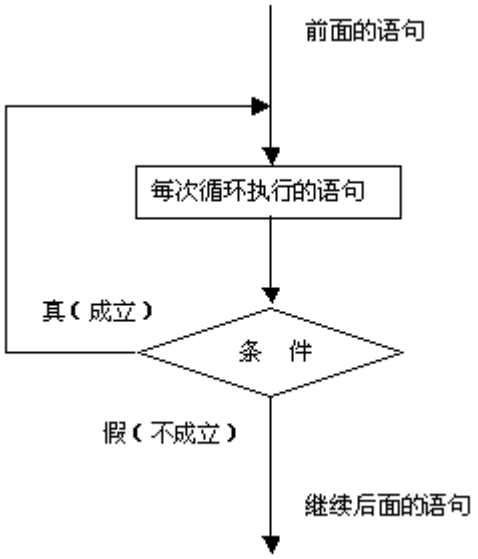
```
//---------------------------------------------------------------------------
#pragma argsused
int main(int argc, char* argv[])
{
   float sum,score;
   int num; //num 用于存储有几个成绩需要统计。
   int i;   //i 用于计数

   char c; //用来接收用户输入的字母

   do
   {
     //初始化：
      sum = 0;
      i = 1;

      cout << "====成绩统计程序====" << endl;
      //用户需事先输入成绩总数：
      cout << "请输入待统计的成绩个数：";
      cin >> num;
      cout << "总共需要输入"<< num << "个成绩(每个成绩后请加回车键)：" << endl;

      while ( i <= num)
      {
         cout << "请输入第" << i << "个成绩：";
         cin >> score;
         sum += score;
         i++;
      }

      //输出统计结果:
      cout << "参加统计的成绩数目:" << num << endl;
      cout << "总分为：" << sum << endl;

      //提问是否继续统计：
      cout <<"是否开始新的统计？(Y/N)?";
      cin  >> c;
     }
     while( c == 'y' || c == 'Y');
}
//---------------------------------------------------------------------------
```

程序完成一次统计以后，会提问“是否开始新的统计”，用户输入一个字母，存到变量 c，然后程序在 do...while 的条件里检查 c 是否等于‘Y’或‘y’。如果不等于，就结束循环。

由于程序在统计之后有一个提问的时间，所以，原来的 getchar()就不再需要了。

在这个例子，外层循环使用 do...while 是最好的选择，因为，用户运行本程序，多数情况下，他至少想统计一次。

最后我们来看 do...while 循环的流程图，请和 while 的流程图对比。



10.3 for 循环

for 循环里在C，C＋＋里用得最多，也是最灵活的循环语句。要学好它，需要从已经学过的 while 循环的身上，“挖掘”出有关循环流程的要素，这些要素隐藏在 while,或 do...while 的背后，但它将直接体现在 for 循环的结构上。

10.3.1 循环条件三要素

学习了两种循环，我们来挖掘一下循环流程中的“条件三要素”。

第一、条件一般需要进行一定的初始化操作。

请看我们用 while 循环实现 1 到 100 累加的代码：

```
int sum = 0; //变量 sum 将用于存储累加和，将它初始化为 0，这很重要。
int i = 1;   //i 是每次要加的数，它从 1 开始。

while ( i<= 100)
{
   sum += i;
   i++;
```

```
}
```

这段代码中，循环的条件是 i <= 100;因此，一开始，i 肯定需要一个确定的值。前面的：

int i = 0;这一行代码，在声明变量 i 的同时，也为 i 赋了初始值：1。这样，条件 i <= 100 得以成立（因为 i 为 1，所以 i <= 100 当然成立）。

第二、循环需要有结束的机会。

程序中最忌“死循环”。所谓的“死循环”就是指该循环条件永远为真，并且，没有另外的跳出循环的机会（后面将学到）。比如：

```
//一段死循环的例子：
while ( 2 > 1 )
{
  cout << "死循环" <<endl;
}
```

执行这段代码，你会发现程序停不下来了。原因就是它的循环条件 2 > 1 永远为 true。所以，一个最后可以变成不成立条件在大多数情况下是必需的。比如在 while 的那个例子：

while (i<= 100)

条件 i <= 100 ，由于 i 在循环中被改变，所以它至少在理论上有可能造成 i <= 100 不成立。

第三、在循环中改变循环条件的成立因素

这一条和第二条互相配套。

比如这段代码：

```
int i=1;
while ( i<= 100)
{
   sum += i;
}
```

同样是一段可怕的“死循环”。因为 i 没有被改变的机会，其值永远为 1 ，从而循环条件 i<=100 也就永远为真。所以在循环中最后一句(下面加粗部分)，不可遗忘。

```
while ( i<= 100)
{
   sum += i;
   i++;
}
```

当然，在这段程序里，i++除了起改变条件成立因素以外，同时也起上 sum 不断加递增的数，从而实现得到累加和。

说完这一些，我们来看Ｃ，Ｃ＋＋中最灵活循环结构：for 循环。

10.3.2 三要素在 for 循环结构上体现

for 循环的语法：

```
for(条件初始化;条件;条件改变)
{
  需要循环执行的语句;
}
```

可见，for 的结构中，不仅提供了的“条件”的位置，同时也提供了条件初始化，和条件改变的位置。这三者虽然在同一行上，但并不是依次连接地执行。

条件初始化的表达式首先被执行（并且只被执行一次）；
然后程序检查条件是否成立，如果成立就执行循环体中的语句，否则直接结束循环。
执行完一遍循环以后，程序执行“条件改变”语句。

1 到 100 整数累加的程序，改为 for 循环写，是最合适的了：

```
int sum = 0;
int i;

for( i=1; i <= 100;i++)
{
  sum += i;
}
```

程序先执行条件初始化语句：i=1;
然后立即判断条件 i <= 100 吗？显示，此时该条件成立；
于是程序执行循环体内的语句，此时只有一句： sum += i;
然后，执行改变条件因子的语句： i++; 此时，i 值变为 2;
程序再次判断条件 i <= 100 ?，依然成立，于是开始第二遍循环……

变量 i 可以初始化条件时才临时声明：

```
for(int i = 1;i <= 100;i++) ……
```

for 语句的复合结构，使得程序变得简捷。比如上面的例子中，原来 while 或者 do...while 结构中，循环体内必须两句语句，现在只需一句（即：i++这一句被移到 for 的特定位置上），这样，我们可以去除花括号：

```
for(int i=0;i <= 100;i++)
   sum += 100;
```

当然，如果在其它情况下，for 的循环体内仍需有多行语句时，｛｝仍是不可避免的。事实上，就算现在这种情况，我也建议大家使用花括号。这样可以让程序的结构看上去更清晰。

在本例中，如果非要讲究简捷，我们还可以将循环体内的那惟一的一行移到“条件改变”的位置：

```
for(int i=1; i<=100;sum += i,i++);
```

sum += i 和 i++之间用逗号分开。而在 for 后面的()行末，则直接跟上分号，表示 for 不必再执行其它的语句。

考虑到后置++的特性（在完成表达式的求值后，才进行加 1 操作），也可以将 sum += i 和 i++合为一句：

```
for(int i=1;i<=100;sum += i++);
```

以上讲了 for 语句极尽合并之技巧，以求一个简捷。反过来，for 语句也可以向 while 或 do...while 语句一样拆开写：

```
int i = 1;
for(; i <= 100;)
{
  sum += i;
  i++;
}
```

看，条件初始化语句被移出 for 的结构，而条件改变语句则被当成一行普通语句，直接加入到循环体内。而在相应的位置上，只留下分号，用于表示空语句（请注意这一行中有 2 个分号,分别在 i<=100 前后）：

```
for (; i <= 100;)
```

如上行所示，for 循环结构中的“条件初始”和“条件的改变”表达式都被省略，在这种情况下 for 和 while 或 do...while 循环完全一样。比如求 1 ～ 1 0 0 累加和：

```
int i=1,sum=0;
for(;i<=100;)
{
  sum += i;
  i++;
}
```

```
int i=1,sum=0;
while(i<=100)
{
  sum += i;
  i++;
}
```

下面分析几个实例：(用于分析的实例不提供上机的完整代码，请同学们自行创建空白工程，然后加入需要代码，确保每个实例都可运行，这是初学者逐步熟练的必经之路……信不信由你。打开 CB 吧)。

题一：用 for 循环在屏幕上逐行输出数字： 1 ～200。

分析：这需要一个变量，其值从 1 变到 2 0 0 ，并且每变一次新值，就用 cout 语句在屏幕上输出其值。

答案：

```
for(int i=1;i<=200;i++)
   cout << i << endl;
```

由于循环中执行的动作只有一句，所以我们省略了 { } 。

题二： 6 能被 1 、 2 、 3 、 6 整除，这些数称为 6 的因子，请循环列出 3 6 的所有因子。

分析： 因子？忘了吗？求 3 6 的因子，就是求 1 ～ 3 6 中哪些整数可以整除 3 6 。我们学过 % 操作符，它用于求两数相除的余数。所以整除不整除，只要看余数是否为 0 即可。

答案：

```
for(int i=1;i<=36;i++)
{
   if(36 % i == 0)   //余数为 0 ，说明整除
      cout << i << " ";  //多输出一个空格，用于做两数之间的间隔
}
```

如果运行程序，得到结果应是：

```
1  2  3  4  6  9  12  18  36
```

在这道题中，我们也看到了两种流程的结合：for 循环流程和 if 条件分支流程。复杂问题的解决，往往就是条件流程和循环流程的种种组合，下面要讲的多层循环也是这些组合中一种。

10.4 多层循环

有些问题需要多层循环嵌套才能解决。前面可以多次统计的程序，就使用了两层循环。外层的 do...while 实现重复统计，内层的 while 循环实现一次统计。

继续分析一些题目：

题三： 输出以下内容，要求使用两种方法，第一种方法采用单层循环，第二种方法采用双层循环。

```
123
456
789
```

方法一：

分析： 单层循环的思路是：从 1 输出到 9 ，并且，每当输出三个数字时，多输出一个换行符。

答案：

```
for(int i=1;i<=9;i++)
{
  cout << i;

  if( i % 3 == 0) //又一次用到“求余”操作。
     cout << endl;
}
```

方法二：

分析：双层循环的思路是：输出三行，每行输出三个数字。

答案：

```
for(int i=1;i<=3;i++)
{
    for(int j=i;j<=i+3;j++)
    {
        cout << j;
    }

    cout << endl;
}
```

代码中，内层的 for 用于输出每一行的数字，而外层的 for 则在每一行输出完成后,输出一个换行符,用于换行.需要另加注意的是,内层循环的条件初始化,和外层循环有关。即这一句：int j=i; 正是。每次开始内层循环时，j 的值都将从当前 i 的值开始。

这道题似乎让人留恋于用单层循环解决一切，因为看上去用双层循环并不是很直观？

题四：请用输出以下内容：

```
1
12
123
1234
12345
123456
1234567
12345678
123456789
```

题目刚出，只见一同学噼噼啪啪开始输入代码，并且很快在屏幕上输出正确的内容，他的答案是：

```
cout << "1" << endl;
cout << "12" << endl;
cout << "123" << endl;
cout << "1234" << endl;
cout << "12345" << endl;
cout << "123456" << endl;
cout << "1234567" << endl;
cout << "12345678" << endl;
cout << "123456789" << endl;
```

如果谁在有关循环的作业中交上类似这样的答案，我准备给他一个“鸭蛋”呵。

这道题目，除非跟自己过不去，否则没有人会非要硬去用一层循环来实现（用一层循环不如用上面的“答案”，笨是笨点，却最为直观）。本题使用双层循环来实现实为最佳方法。

分析：外层循环用于控制输出 9 行；内层循环用于输出每行的数字。每一行都是从 1 开始，但第一行输出 1 个数字，第二行输出 2 个，第三行输出 3 个……

答案：

```
for(int i=1; i<=9; i++)
{
    for(int j=1; j<=i; j++)
    {
       cout << j;
    }

   cout << endl;
}
```

看，在这道中，内层循环的条件初始化和外层循环无关，但循环条件判断却和外层的 i 有关了(j<=i)。当然，这并不是必要条件，但内层循环的条件初始化或条件判断，和外层循环的某些因素有关，这是很多多层循环的解决问题的关键！继续一个经典的题目。

题五：请输出以下九九口诀表：

```
1*1=1
1*2=2 2*2=4
1*3=3 2*3=6 3*3=9
1*4=4 2*4=8 3*4=12 4*4=16
1*5=5 2*5=10 3*5=15 4*5=20 5*5=25
1*6=6 2*6=12 3*6=18 4*6=24 5*6=30 6*6=36
1*7=7 2*7=14 3*7=21 4*7=28 5*7=35 6*7=42 7*7=49
1*8=8 2*8=16 3*8=24 4*8=32 5*8=40 6*8=24 7*8=56 8*8=64
1*9=9 2*9=18 3*9=27 4*9=36 5*9=45 6*9=36 7*9=63 8*9=72 9*9=81
```

分析：你可以看出，本题和题四有很大的类似，都是要输出一个“三角形”（严格说是梯形？），所以解题思路也大致一样：输出九行。你可能会说，输出九列也可以吧？的确可以，不过由于我们现在使用控制台（DOS)窗口来输出，所以被限定只能由上到下一行一行输出，而不能由左到右一列列输出。

既然是按行输出，我们来看每一行的内容有什么特点：

每一行输出的内容都是：i*j=k,其中 k 是积,由 i 和 j 决定,所以我们可以不必看:

第 1 行：1*1，只有一个，看不出有什么特点。

第 2 行：1*2 2*2 ……1,2 分别乘以 2

第 3 行：1*3 2*3 3*3 ……1,2,3 分别乘以 3

所以，各行的内容的规律是：设当前为第 line 行，则输出 n*line ,n 为 1 到 line.

```
for(int line=1; line <= 9;line++)
```

```
{
   for(int n=1; n<=line;line++)
   {
      cout << n << '*' << line << '=' << n * line <<' ';
   }

  cout << endl;
}
```

把 line 换成 i，n 换成 j，本题的循环控制部分的代码和题四完全一样。

理解以上各分析题，最好的方法是实际上机，然后运行，并且最好自己尝试按 F8 来一步步运行。所以再说以遍，本章内容可以宣告结束，但如果你没有上机操作这些循环题，就不能说你学好本章。

10.5 小结及一点问题

学习了三种循环流程：while,do...while,for。

while 在每一遍循环开始时检查条件是否成立，如果一开始条件就不成立，则循环中的语句将一次也没有执行。

do...while 的特殊之处在于，第一遍循环前，并不检查条件，所以，就算条件根本就不成立，循环也将并执行一次，如：

```
do
{
  ...
}
while(false);
```

条件是“false”？这也是条件吗？是的，这也是条件，一个摆明了就是不成立的条件，常见的还有：while(0)。0 可以看 false。相反的，摆明是真的条件是：while(true)或 while(1)等。

for 的特殊之处在于，它除了条件判断以外，还明确地留出了条件初始化，条件变化的位置。所以，有关计数的循环最适于用 for 来实现，请参看课程中用 for 实现 1 到 100 累加的实例。

至于 for 的各种“变体”，只须了解，以后用得熟了，再去卖弄不迟 :）。

循环就象绕圈子。比如，体育课，跑 1200 米，跑道一圈 400 米，所以我们要做的事就是一边跑一边在心里计数（当然要已数，否则老师万一少计一圈，我们可就玩完了），当计数到 3 圈时，“循环”结束。

在这个例子，条件初始化就将计数设为 0；循环的条件则是计数小于 3；条件变化则是指在每跑完一圈时，心里将计数加 1；至于循环的动作，自然就是跑步了……

用 while 实现：

```
int js = 0; //js 取意“计数”，而不是“奸商”，也不是“句神”

while( js < 3)
{
  跑一圈……;
  js++;
}
```

用 do...while 实现：

```
int js = 0;

do
{
  跑一圈……
  js++;
}
while( js <3);
```

用 for 实现（最适合了）

```
for(int i=0;i<3;i++)
{
   跑一圈……;
}
```

如果，我在跑步时不幸由于体力不支而晕倒……怎么办？下一章见！

第十一章　流程控制拾遗与混合训练

说“拾遗”，可能你会以为本章的内容不是重点？那可不是，流程控制的内容并不多，却支撑着所有程序的框架！所所有有关流程的内容都是基础加重点。只是本章中继续讲到一些关键字可以改变流程，但并不独自构成完整流程结构。

另外，作为流程控制内容的结束章节，我们于最后安排了一些各流程混合使用的训练。

11.1 break

11.1.1 break 的作用与用法

循环就象绕圈子。比如，体育课，跑 1200 米，跑道一圈 400 米，所以我们要做的事就是一边跑一边在心里计数（当然要已数，否则老师万一少计一圈，我们可就玩完了），当计数到 3 圈时，“循环”结束。

如果，我在跑步时不幸由于体力不支而晕倒……怎么办？

有两种办法，一种是在判断是否继续循环的条件中加入新增条件的判断：

假设原来的循环表达为：

```
while(已跑完的圈数 < 3)
{
  跑一圈……;
}
```

那么，加上附加条件后，循环表达为：

```
while(已跑完的圈数 <3 && 我还跑得好好的） //&& 就是"并且"，没忘吧？
{
  跑一圈……
}
```

第二种方法是在循环中使用条件分支，在指定的条件成立时，中途跳出循环，用于实现跳出的关键字为：break。

```
while(已跑的圈数 < 3 )
{
  跑一圈……;

  if(我身体感觉不妙）
     break;
}
```

在循环中，每跑完一圈，都检查一下自己是否感觉不妙，如果是，则程序执行 break,直接跳出 while，而不管此时圈数是否到达 3 圈。

还记得“小女孩买裙子”的故事吗？那时候，我们将“父母不给买小红裙 && 我还没有哭累”作为循环继续的条件，如果使用 break，则可以写成这样：

```
while(父母不给买小红裙）
{
  我哭;

  if(我哭累了）
    break;
}
```

在循环中，“我”每哭一次，都想想是否累了，如果是，则程序执行 break，直接跳出 while，而不管此时爸妈是否已经买了我的裙。

通过这两个例子，你应该注意到了，如果要用 break，则 if 的条件（也就是要执行 break 分支的条件），正好是把原来放在循环判断中的条件反正过来。比如，原来是判断“我还跑得好好的”，现在则是判断“我身体感觉不妙”；原来是判断“我还没有哭累”，现在是判断“我哭累了”。

一句话，原来是判断“是否继续循环”，现在是判断“是否跳出循环”……

再来看那个“可以多次统计”的统计程序。看看是否也能把它改成使用 break 来结束循环。

为了节省篇幅同时也是为了突出重点，我们将其中用于实现一次统计的代码，用一句**伪代码**来实现。（什么叫伪代码？我们用得很经常啊，就是那些用自然语言写的“代码”，这些代码当然无法在计算机上运行，它们只是要方便地表达实际代码要实现的功能）。

```
int main(int argc, char* argv[])
```

```
{
   实现统计一个学员的成绩；  //伪代码，详细代码请见上章相关部分

   do
   {
      //提问是否继续统计：
      cout <<"是否开始新的统计？(Y/N)?";
      cin  >> c;
    }
    while( c == 'y' || c == 'Y');
}
```

改成用 break;

```
int main(int argc, char* argv[])
{
   实现统计一个学员的成绩；  //伪代码，详细代码请见上章相关部分

   do
   {
      //提问是否继续统计：
      cout <<"是否开始新的统计？(Y/N)?";
      cin  >> c;

     //如果用户输出的不是字母 Y，说明他不想继续统计了，我们需要中断循环。
     if( c != 'y' && c != 'Y')
         break;
    }
    while (true);
}
```

请首先 while(true)部分，其条件直接写上真(true),表明这是一个无条件的循环（即，循环将无条地一直持续下去），这岂不犯了程序界的武林大岂：成了一个“死循环”？其实，相信你已明白，在循环体内，有一个 break 的分支在呢，当判断用户输入的字母既不是小写的 y，也不是大写的 Y，break 就起它能起的作用了。

三个例子，都是从循环判断的条件摘出一部分或全部（最后一个例子），然后循环体中，采用一个 if 判断，结束 break 来跳出循环。可能你会问：为什么要 break 呢？直接用原来的方法，在 while 处判断条件不是很好吗？

break 的长处在于，它可以在循环体内的任意位置进行判断。

继续上一例。假设我们出于慎重，想在用户按入 N 时，再问他一句是否真的退出统计，则此时显示出了 break 的方便：

```
int main(int argc, char* argv[])
```

```
{
   实现统计一个学员的成绩；  //伪代码，详细代码请见上章相关部分

   do
   {
      //提问是否继续统计：
      cout <<"是否开始新的统计？(Y/N)?";
      cin  >> c;

      //如果用户输出的不是字母 Y，说明他不想继续统计了，我们需要中断循环。
      if( c != 'y' && c != 'Y')
     {
          //出于慎重起见，我们要再问一句用户是否真的不统计了？
          cout << "您真的不想继续计算了？(Y:真的结束 / N:继续统计)";
          cin >> c;

          //这回，如果用户输入 Y,表明他真的不统计了：
          if( c == 'Y' || c == 'y')
               break;
     }
     }
    while (true);
}
```

在上面例子，由于用户的两次输入我们都采用变量 c (char 类型)接收。但如果第一次输入 字母‘Y’时，循环需继续，但如果用户是在第二次输入‘Y’，则表示是真的不统计了，循环却必须结束；所以，此时 while 无法仅凭 c 的值来做出正确判断，但，采用 break，正如上面代码，我们在合适的位置安排一个 break，从而直观地实现了。

当然，这里仅为了讲学方便而举此例，如果你真的在程序中为了一个“是否继续统计”而问了用户两遍，可能会被用户骂做“神经质”。不过，如果是删除某些重要数据（直接删除，不可恢复的情况），多问一次就选得很重要了。（比如句神英语删除用户操作就会在最后多问一句“真的要说再见吗？我们会想你的……”）

再举一例，看我们前面关于跑步的例子：

```
while(已跑的圈数 < 3 )
{
  跑一圈……;

  if(我身体感觉不妙)
     break;
```

```
}
```

这段代码有点问题，因为判断“我身体感觉不妙”是在跑完一圈之后……很可能我在某一圈刚开始跑时就觉得肚子剧痛，极可能是得阑尾炎啊！按照这段程序，我只有坚持跑完一圈后，才能 break 了……

要完美解决这个问题，我们将在本章再后讲到，现在先采用一个“通融”的办法，我们允许你每跑 100 米就检查一次吧：

```
while(已跑完图数 < 3)
{
  跑第 1 个 100 米;

  if(我身体感觉不妙)
     break;

  跑第 2 个 100 米;

  if(我身体感觉不妙)
     break;

  跑第 3 个 100 米;

  if(我身体感觉不妙)
     break;

  跑第 4 个 100 米;

  if(我身体感觉不妙)
     break;
}
```

代码中，我们将 1 圈拆为 4 个 1 0 0 米，每跑完 1 / 4 ，我们就检查一次是否身体不对。看明白这个例子，我想你对 break 的用途和用法，可以算是理解了。

11.1.2 break 的一个“高级用法”

本小节不是很适于没有多少实际编程经历的初学者，所以初学者可以跳过,以后再回头阅读。当然，所谓的“高级用法”的确是应该加对引号的，所谈的内容只是一个高手们常用小小技巧。

使用 do...break...while 简化多级条件判断的结构。

如果你写过不少代码，那么一定会不时遇到类似下的情况：

假设要找到文件 A，复制该文件为 B；然后打开 B 文件，然后往 B 文件内写入一些内容；最后在写入成功后，我们需要再进行一些相关操作。

在此过程，遇到以下情况时将放弃后续的操作，认为是操作失败：

１、如果 A 文件不存在；

2、如果B文件已经存在,并且询问用户是否覆盖时，用户回答“不”；
3、无法复制出B文件；
4、无法打开B文件；
5、无法写入B文件；
6、无法正常关闭B文件。

用伪代码写该段程序为：

```
if( A文件存在  )
{
    执行A文件的相关操作;
    if( B文件不存在 || 用户允许覆盖原有B文件)
    {
        复制A文件为B文件;
        if(复制文件成功)
        {
             打开B文件;
              if(打开文件成功)
             {
                写入文件;
                  if(写入成功)
                  {
                    关闭B文件;
                    if(关闭成功)
                      {
                       执行其它必须在一切成功后进行的操作。
                           ……
                       }
                  }
              }
        }
    }
}
```

可能有些操作和判断可以同时处理，但这个程序的繁琐仍然不可避免，而现实中程序的复杂性往往要远过于此例。从语法上看，这个例子没有任何错误，但它的一层套一层的条件判断却让人难以书写，阅读，调试，在复杂的情况就容易造成人为的错误(比如最马虎的，花括号匹配不对等……)。

同样一段代码“程序老鸟”是这样写的：

```
do
{
   if(A文件不存在)
      break;
   执行A文件的相关操作;
```

```
    if(B文件存在  && 用户不允许覆盖)
       break;

    复制A文件为B文件;
    if(复制不成功)
       break;

    打开B文件;
    if(打开B文件不成功)
       break;

    写入文件;
    if(写入文件不成功)
       break;

    关闭B文件;
    if(关闭不成功)
       break;

     执行其它必须在一切成功后进行的操作。
     ……
}
while(false);
```

看，代码是不是“直”了很多？这里用了 do..while，可是根本不是为了循环，而是为了使用它的 break 功能。每当有操作不成功，就直接用 break 跳出循环。所以循环条件总是一个“永假” false。

在一个程序中，这种结构相当的多，为了更加一步淡化 while 的原来的循环用途，我们非常值得在代码加入两个共用的宏：

```
#define  BEG_DOWHILE do {
#define  END_DOWHILE  } while(false);
```

这里举的是 do...while 结构，在某些情况下，可以使用 while...来实现类似功能。

11.1.3 break 在 for 循环中的一点注意

前面举的例子都是 do...while 或 while，break 在 for 循环也一个样。请看下面例题：

例一：从 1 开始累加，每次递增 1，请问累加到哪个数，累加和超过２０００？请输出该数，及当时的累加和。

分析：和求 1～100 的累加和类似，只是在发现累加和已经超过 2000 时，就输出当前累加的数，然后结束循环。

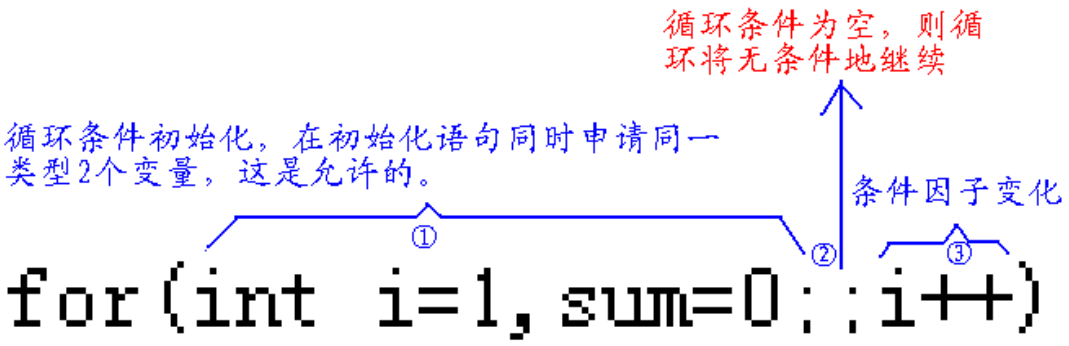
```
for(int i=1,sum=0;;i++)
```

```
{
    sum += i;
    if(sum > 2000)
    {
        cout << i << "," << sum << endl;
        break;
    }
}
```

输出结果为：

```
63, 2016
```

关于这段例子，需要注意三点：1、循环条件初始的位置，我们同时声明两个变量；2、没有循环条件。为了解这两点注意，请看下面放大图：



最后一点注意是关于 break 和“条件因子变化”的注意。我们知道，for 每执行一遍循环体后，都将执行一次“条件因子变化”语句（见上图③）。现在需要注意的是：

在 for 循环中，执行 break 后，“条件因子变化”语句同样被跳过，没有被执行循环就被中断。

（完整代码请见 lz1.bpr）

至此，break 在 while,do...while,for 中的用法我们都已见过。不过，你还记得吗，我们最早学到 break 是在哪里？在讲条件分支语句中 switch 里。如果你有点忘了那里的 break 是起什么作用，现在就去看看吧。

11.1.4 多层循环中的 break

break 只能跳出当前层的循环，所以，如果有多层循环，则在内层的 break 跳出循环后，外层的循环还将继续。

前面说跑步的例子，一圈 400 米，我们每跑 100 检查一下是否肚子疼什么的，如果疼得利害就 break，不跑了。这和现实不符，我们应该每跑一步就检查一次是否肚子疼最合理。

一圈得分成几步呢？显然不能再像上面分成四次检查那样写代码了。我们加一层循环，也就是把跑一

圈的工作用一个循环来实现：

```
while(一圈未结束)
{
  跑一步;
}
```

然后，我们在每跑完一步时加入一个判断：

```
while(一圈未完)
{
  跑一步;

  if(我身体感觉不妙)
     break;
}
```

把这跑一圈的代码加入外层循环：

```
while(已跑完图数 < 3)
{
    while(一圈未完)
    {
       跑一步;

       if(我身体感觉不妙)
           break;
     }
}
```

外层的 while 用于负责一圈一圈循环跑完三圈，内层的 while 用于负责一步一步地循环跑完一圈，同时负责每跑一步就检查是否身体不妙，若身体不舒服，就跳出循环，不跑了。看起来代码很完美，其实 BUG 已经产生：问题就在那个 break。当“我身体感觉不妙”后，程序遇上 break,跳出内层 while，落入外层的 while,外层的循环可没有被 break,所以程序将继续外层的循环。假如你跑第一圈跑了一半时肚子疼，按照这段程序逻辑，那好这第一圈剩下的一半你可以不用跑了，但后面的两圈你还得继续。

解决的第一种方法是：

```
while(已跑完图数 < 3)
{
    while(一圈未完)
    {
       跑一步;
```

```
        if(我身体感觉不妙)
            break;
    }

    if(我身体感觉不妙)
        break;
}
```

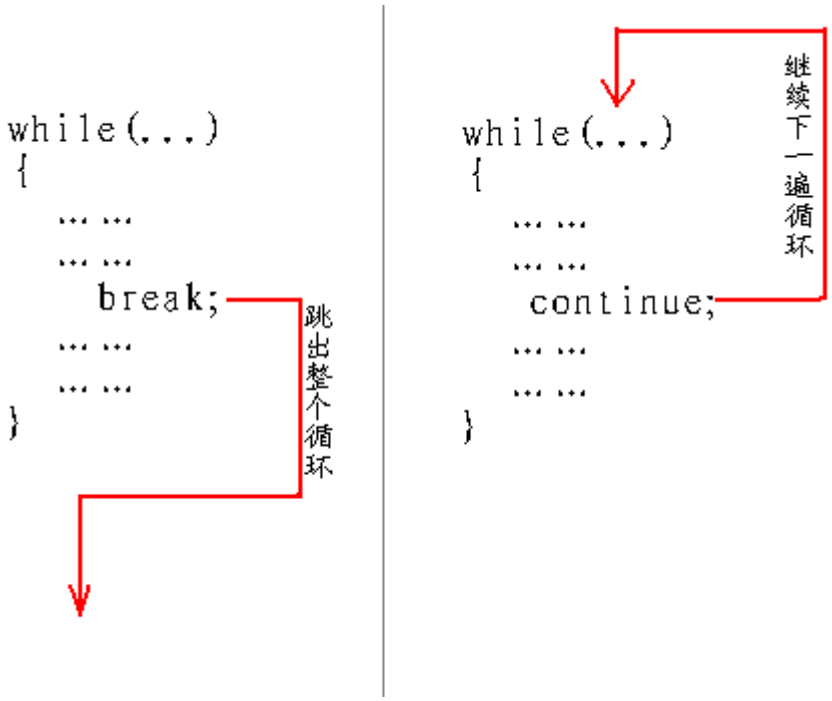
我们在外层也进行了一次判断，这样当然就可保证从内层跳出来以后，外层的循环也被跳出。但在内层已经做过一次“感觉”的情况下，外层还要重新“感觉”一次，这种代码让人不爽，所以我们可以加一个变量，用于记住现在的身体状态：

```
bool needBreak = false; //是否需要跳出循环

while(已跑完图数 < 3)
{
    while(一圈未完)
    {
        跑一步;

        if(我身体感觉不妙)
        {
            needBreak = true; //做一标志，需要 break;
            break;
    }

    if (needBreak)
        break;
}
```

虽然本人的课程并不是绕什么圈子，可是在这有关 break 的长篇累牍的文字中，想必各位现在头脑里只有一个词，只有一个念头：break。想 break？后面的课程就不要啦？不可能，我们还是 continue 吧。

11.2 continue

continue 汉意为继续。它的作用及用法和 break 类似。重要区别在于，当前循环遇到 break，是直接结束循环，而若遇上 continue,则是停步当前这一遍循环，然后直接**尝试**下一遍循环。我把“尝试”加粗以引起注意，为什么要注意原因后面再说，请先看下面关于 break 和 continue 的对比：



continue 并不结束整个循环，而仅仅是中断的这一遍循环，然后跳到循环条件处，继续下一遍的循环。当然，如果跳到循环条件处，发现条件已不成立，那么循环也将结束，所以我们称为：**尝试**下一遍循环。

例二：求整数 1～100 的累加值，但要求跳过所有个位为 3 的数。

分析：在循环中加一个判断，如果是该数个位是 3，就跳过该数不加。

如何判断一个 1 到 100 中，哪些整数的个位是 3 呢？还是 % ，将一个 2 位以内的正整数，除以 10 以后，余数是 3，就说明这个数的个位为 3。

比如： 23 ,除以 10，商 2，余数 3。这里我们不需要商，所以用求余（也称为求模）运算：23 % 10 = 3。

```
int sum = 0;

for(int i = 1; i<=100;i++)
{
   if( i % 10 == 3)
      continue;

   sum += i;
}

cout << sum << endl;
```

（完整代码请见 lz2.bpr）

和 break 正相反：

在 for 循环中，执行 continue 后，“条件因子变化”语句没有被跳过，将被执行一次，然后再尝试

循环的下一遍。

在上例中，当条件 i %10 ==3 成立时，continue 被执行，于是，程序首先执行 i++;然后执行 i <= 100 的判断。如果将该段程序改成 while，正确答案为：

```
int sum = 0;
int i = 1;

while(i <= 100)
{
   if( i % 10 == 3)
   {
      i++;
      continue;
   }

   sum += i;
   i++;
}

cout << sum << endl;
```

请注意程序中的两句"i++;"，缺一不可。

11.3 goto

臭名昭著的 goto 出场了。

goto 的汉义为“转到”，在计算机语言里，它的完整名称为：“无条件跳转语句”。几乎所有高级语言都会劝你尽量不要使用它 goto。因为它会破坏程序的模块性，严重降低一段程序的可读性。若是老外写的书，则比喻使用大量 goto 的代码：“像意大利面条”。嗯，其实北京的杂酱面也很缠绕……可惜没有走向世界。

goto 的用法是，首先要在代码中某处加上一个位标（也称标号），然后在代码中的需处，加上 goto，并写让要跳转到位标。比如你在第三行代码加一个位标：A ： ，然后可以在第 10 行写上一个 goto A，程序执行到该行时，就将跳到第三行。

加位标的方法是在一空行加上位标的名称，命名规则和变量一样，但最后要加上一冒号“：”。
例如：

```
int i = 1;
```

```
A  :

cout << i << endl;
i++;

if(i <= 10)
  goto A;

... ...
```

goto 虽然号称“无条件跳转”，事实上倒是有些条件限制。主要是三条。

1、goto 只能在当前的同一程序段内跳转；

2、goto 可以从循环内跳转到循环外的代码，但不能从循环外的代码跳到循环内；

3、在有 goto 的跳转范围内，不能再使用 C++允许的临时变量声明。

好了，其实笔者写程序近 10 年，惟一用到 goto 的地方就是：将一段简单的程序故意用 goto 写得面目全非，以期能让破解程序的人因为眼晕而放弃功击……一句老话：如果没有什么特殊理由，不要在程序里使用 goto。

11.4 流程控制强化训练

这一节将提供一系列的有关流程控制的实例，程序中知识点全部不超过你学到本章时的水平。有些程序需要一些数学上的小小技巧。

所有实例是从建立一个新的控制台工程开始。所以一些由Ｃ Ｂ自动生成代码我将不写出。你应该知道如何将它们写成一个完整的控制台程序。如果实在有困难也不要紧，各个实例的完整我都已经提供下载。（但只有付费报名学员可以通过课程下载器下载，解密）。

最后，在例子中会出现不少常见的编程技巧，可能在前面的课程中没有直接讲到，我会对这些技巧进行解说。

11.4.1 求绝对值

例三：用户输入一整数，请用程序输出绝对值

分析：

1、本例演示了一个最简单的流程控制：if...

2、同时你可以学到如何求一个数的绝对值，很简单;另外，看一个数是否为负数，就是看它是否小于 0，这也很简单。

3、另外，本例使用一个 while(true)来无限循环，你可以不断地输入，如果要中止程序，请按 Ctrl+C,这是由操作系统提供的，DOS 窗口下中止程序的热键。(因此，本例也无须在最后加 getchar();这行代码)

答案:

```
#include <iostream.h>

int main(int argc, char* argv[])
{
    int num;

    while (true)
    {
        cout << "求绝对值的程序" << endl;
        cout << "要中止运行请按 Ctrl + C " << endl;

        cout << "==============" << endl;
        cout << "请输入一个整数：";
        cin  >>  num;

        //正数和 0 的绝对值是本身，负数的绝对值为其相反数
        if(num < 0)
           num = -num;

        cout << "绝对值为：" << num << endl << endl;  //输出两个换行，仅是为了美观
    }
}
```

11.4.2 判断用户输入字符的类型

例四：用户输入一字符，请判断该字符是：大写字母，小写字母，数字字符，其它字符。

分析：

1、本题主要演示多级 if..else...

2、在 ASCII 表中，题中所提的 前 3 类字符，其 ASSCII 值都各自连续（换句话说就是：所有的大写字母都是连续的，所有在小写字母也是连续的……）。基于这一点，你容易看明白代码中为判断字符类型的方法。

3、本解答也采用了循环，所以也不用加 getchar()这行代码。关于循环中条件判断方法比较特殊，请见代码后的说明。

```
#include <iostream.h>
#include <conio.h>

int main(int argc, char* argv[])
{
    char ch;
    cout << "请输入一个字符：" << endl;

    while( (ch = getche()) != '\r' )
    {
         cout << endl; //加一个换行，仅为了输出美观
```

```
            if( ch >= 'A' && ch <= 'Z')
               cout << ch << "是一个大写字母。" << endl;
            else if ( ch >= 'a' && ch <= 'z')
               cout << ch << "是一个小写字母。" << endl;
            else if( ch >= '0' && ch <= '9')
               cout << ch << "是一个数字字符。"  << endl;
            else
               cout << ch << "是一个其它的字符。" << endl;
      }
   }
```

这段代码中，我们用到了 getche()库函数，它的声明包含在 conio.h 文件中，所以本例程除了"#include <iostream.h>"以外，还另需 "#include <conio.h>"。

getche() 和我们常用的 getchar()同样是接收用户从键盘输入的一个字符，但 getchar()在用户输入字符后，用户还需要敲一下回车键才能完成输入；而 getche()则在用户敲入一个字符后，立即完成。本例中，我们希望如果用户敲一个回车键，则程序自动结束（见下面解析），所以我们采用 getche()函数。

现在来看 while 的循环条件：

```
while( (ch = getche()) != '\r' )
```

这行代码依次完成下面两件事：

首先是： ch = getche() ，它等待用户敲入一字符，然后将该字符存储在 ch 变量。

然后是判断条件： （……） != '\r' 。 程序判断 ch 是否不等于 '\r', '\r' 即回车(return)字符。也就是看用户输入的字符是否为回车键，如果不是，则循环继续，如果是，则循环结束。

记住，在C和C＋＋里 一个赋值表达式： A = B, 本身也有值，值就是完成赋值后的A。在上例中，A 是 ch,B 是 getche()。在C， C＋＋里，几乎所有表达式本身都有值，比如： 1 ＋ 2 的值是 3；而表达式 a = 3 的值为 3 。

理解这段代码，最好的方式就是在C B中运行它。至于我们所要练习的多级 if...else 在例中的表现，我不再多说，你需要自己看懂它。

最后解释一下 conio.h, 其中 con 即我们总说的控制台，io 则和 iostream 中的 io 一样，指：input/output。

11.4.3 等腰三角形图形的输出

例五：请输出以下图形：

```
      *
     ***
    *****
   *******
  *********
```

分析：

新手刚看这道词可能觉得无从下手，其实，如果把图形改成一个矩形：

```
*********
*********
*********
*********
*********
```

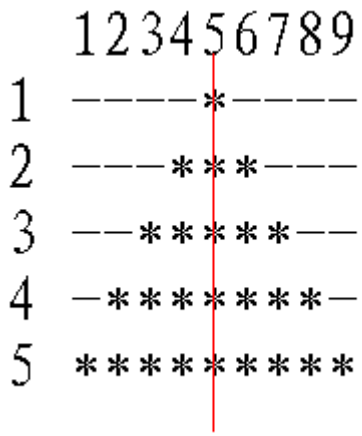
那么就很好解决了：输出 5 行，其中每行都输出 9 个* 。

```
for(int i=0;i<5;i++)
{
  for(int j=0;j<9;j++)
  {
    cout << '*';
  }
}
```

对于三角形，程序仍然是这个结构：需要两层循环。同样是要输出 5 行，所以外层循环不变；不同的地方在于每一行输出的内容。其实三角形同样是输出一个矩形，只不过有些地方要打空格，有些地方要打*，以下我们用“-”表示空格，则三角形实为：

```
----*----
---***---
--*****--
-*******-
*********
```

所以，问题的重点在于：在每一行中，哪些地方要输出空格，哪些地方要输出星号？如果我们行和列都从 1 开始编号，如图：



仔细观察我们发现，哪一列要打星，哪一列要打空格，主要和该列与第 5 列（红线所在列）的距离有关：

第　1　行：　只有第 5 列本身打星，第 5 列和第 5 列（自身）的距离是 0

第　2　行：　除了第 5 列以外，增加第 4 、 6 列， 4 和 6 与 5 的距离都为 1

第　3　行：　增加了 3 、 7 两列要打星， 3 ， 7 两列和 5 的距离都为 2

……

行了，规律就是：在第 n 行内，凡是和第 5 列的距离小于 n 的列，都要打星，其余的列打空格。
下面代码中，row 表示当前行，col 表示当前列。

答案：

```
#include <iostream.h>
int main(int argc, char* argv[])
{
   for(int row=1;row<=5;row++)
  {
      for(int col=1;col<=9;col++)
      {
         if( col-5 > -row && col-5 < row)
             cout << "*";
         else
             cout << " ";
      }

  cout << endl;
 }

getchar();

return 0;
}
```

以下是输出结果：

在本例中，为了保持大家日常生活的习惯，我对行，列的编号均从 1 开始，其实，C，C＋＋程序员更习惯于编程从 0 开始，即原来的第 1 行现在称为第 0 行，第 1 列称为第 0 列，则相关代码如下(黑体部分为改动处)：

```
 for(int row=0;row<5;row++)
  {
      for(int col=0;col<9;col++)
      {
         if( col-4 >= -row && col-4 <= row)
             cout << "*";
         else
             cout << " ";
      }
```

```
  cout << endl;
}
```

学会从 0 开始索引的思想方法，这也是大家所要注意的，否则在阅读别人代码时会比较困难。

11.4.4 输出正弦曲线图

例六：请在ＤＯＳ窗口输出正弦曲线图

分析：

１、还记得初中代数学的正弦函数吧？

y = sin (x);

当 x 从 0 到 2 π 变化时， y 的值在 -1 和 +1 之间变化。

我们现在的任务就是随着 x （位置）的变化，在 y 的位置上打一个点即可。

２、Ｃ为我们提供了 sin 的库函数。只要我们给它 x 的值，它就能计算出相应的 y 值。sin(x)函数包含在头文件 main.h 里。

３、为了方便，我们将“竖”着输出曲线，即 x 的值由上而下增长，而 y 值则在左右“摇摆”。并且，如果 y 值为负数的话，那么将输出到屏幕的最左边外面，所以我们将 y 值统一加上一值，用于向右偏到合适的位置。至于要加多大的值，和第 4 点有关。

４、和前面输出“等腰三角形”类似。假如我们需要在屏幕的某一行最右边（行末）打出一个点，我们的方法是在前边连续地打满空格。正弦值在 -1 到 1 之间，我们不可能打零点几个空格，所以，需要正弦值放大一定的倍数。

答案：

```
int main(int argc, char* argv[])
{
   #define PI 3.14159

   int scale = 30; //放大倍数
   double X,Y;

   for(float X = 0.0; X <= 2 * PI; X += 0.1)
   {
       // 乘上 scale 是为了放大Y值，而加上 scale 则是为了向右边偏移
       // 以保证所有的点都不会跑出屏幕左边。
       Y = sin(X) * scale + scale;

       //前面打空格
       for(int dx = 0;dx<Y;dx++)
            cout << ' ';

       cout << '.' << endl;
    }

   getchar();
```

```
    return 0;
}
```

完整的代码请查代码文件。由于输出画面太长，所以这里不显示结果图。

11.4.5 标准体重计算程序

尽管类似输出“九九口诀表”、“等腰三角形”，“正弦曲线”这些题目对锻练大家的编程思维颇为有益，但可能很多人都不会喜欢这种题。

嗯，这很多人当中，就有我自己一个。所以，我们来点有趣的题目吧。

街上有一种电子称，你站上去一量身高体重，它就会告诉你的身材是否为标准体重。可是有一天我兴冲冲地往上一站，那机器竟怪里怪气地说：“本仪器不适于非洲儿童……”。害得我当众狼狈而逃！

回去后，我痛下血本，每天大鱼大肉，如此月余，自认为横了一点，很想再测测这回该是哪一洲的儿童。然而由于上次的经历已经对我的造成了极大的心灵伤害，以致于我上街看见那种电子称就腿软。只好自购小站秤一台，米尺一条，不过，如何计算是否标准体重呢？嗯，就是这节课的“标准体重计算程序”了。

计算标准（理想）体重的方法是从网上搜到的：

“最近军事科学院还推出一种计算中国人理想体重的方法:
北方人理想体重 = 〔身高 cm—150〕x0. 6 十 50(kg)
南方人理想体重 = 〔身高 cm—150〕x0. 6 十 48(kg)”

（原文请见：三九健康网）

考虑到女性一般要比男性轻，所以如果是女性，我们还需要将标准体重减去 2 公斤。

可见，要计算一个人的标准体重，必须知道是男人女人，是北方人还是南方人,及他的身高。

用户还必须输入他的现实体重，这样，在程序计算出标准体重之后，我们计算实际体重在标准体重百分之几的范围之内，作出不同判断。

代码请见例程文件，加有详细的注解。本题事实上没有什么复杂算法，所以比前面的题都要简单——尽管代码看上去长多了。

试用时请注意：程序需要输入数字时，如果不小心输入字母并回车，将引起死循环，这是 cin 的问题所致，请按 Ctrl + C 强行退出即可。

在课程的最后，测一下自己的体重与“理想体重”的差距，是个不错的选择……测好了？能告诉我程序对你的身材所作的评价吗？

第十二章　函数（一）

函数是C语言的一个重点和难点，我们此次将连续两章进行讲解。本章重点在于彻底理解函数的作用，学会调用函数，学会自己编写函数。

秉承我们“以人为本:)”的学习方法，我们学习函数第一件事就是问话：干嘛让我学习函数？反过来说就是：函数能为一个程序员做些什么？

12.1 函数的引入

家里地板脏了怎么办？

拿起扫帚,自个儿扫呗。当然，在扫之前要对地板上的各种“脏”东西定好数据类型，针对不同的“数据类型”，我们需要进行不同的处理，比如是废纸，则无情地扫到垃圾桶；但若是在地上发现一张百元大钞，则应该脉脉含情地捡起放在胸口：“你让我找得好苦”。

在扫地的过程中，当然也无处不在使用“流程控制”。比如家里有三间房子，则应该是一个循环。而每一间房子的打扫过程也是一个循环过程：从某个角落的地板开始，向另一个角落前进，不断地重复扫把的动作。中间当然还需进行条件判断：比如前面所说的对地面脏物的判断，再如：if （这一小块地面不脏），则 continue　到下一块地面……

我们学了“数据类型、常量、变量”，所以我们有了表达问题中各种数据的能力；

我们还学了“流程控制”，所以我们还会针对各个问题，用正确的流程组合解决问题的步骤，从而形成解决问题的方法。

看起来我们已经拥有了从根本上解决任何问题的能力。但－－

家里电视坏了怎么办？

呃？这个，我不是学电器专业的。我只会看电视，我不会修理电视。

这时候我们的办法是：打一个电话请专业的修理师上门修理。

还有很多问题的解决办法都是和修电视类似，即：我们自己没有这个能力，但我们可以**调用**一个具备这一能力的人来进行。

函数在程序中就相当于：具备某些功能的一段相对独立的，可以被调用的代码。是的，函数也就是一段代码，代码也就是我们前面的学的“变量，常量，流程控制”等写成的一行行语句。这些语句以一种约定形式存在着，等待我们去调用它。

其实我们已经用过函数了：给你一个数：2.678,能帮我们求出它的正弦值吗？想起来了吗？我们在上一章中学过 sin()函数。

一段用以被调用的代码，这是函数的本质。当然，使用函数在程序中还有许多其它的作用，但我们将从这个最关键的地方讲起：怎样调用一个函数？

12.2　学会调用函数

这一节的任务是通过学会如何调用一个函数，从使用者的角度来了解函数各个重要知识点。从而，也为下一节学习如何写一个函数打下基础。我们相信这样的安排是科学的，因为在生活中，我们也往往是先是一个“使用者”，然后才是一个“
创造者”或“提供者”。

12.2.1 哪些函数可调用？

在学会如何调用函数之前，不妨先看看有哪些现成的函数可以调用。

12.2.1.1 库函数

C++ Builder 提供了数百个库函数。之所以称为“库”函数，是因为这些函数被集中在一个或几个文件里，这些文件就像存放函数仓库，当我们需要时，程序就可以从“库”中调用。

库文件又分为两种形式：

第一种是把不同的函数分门别类地放在不同的文件里。比如和数学计算有关的，放到一个文件，和 I / O 操作有关的，放到另一个文件。这样做的结果是：文件很多，但每个文件都比较小。这种库我们称为“静态库”。

使用静态库的好处是：当我们的程序调用到某一库的函数是，C++ Builder 可以将这个库文件直接和我们的程序“合并”到一起。这样，我们提供给用户程序时，只需要提供一个可执行文件（比如叫：A.exe)。用户得到这个程序时，不用安装其它文件，就可以运行了。

使用静态库的坏处是：假如你需要向用户提供两个可执行文件，比如 A.exe 和 B.exe,两个文件可能都用到同一库文件，所 以同一个库函数既被“合并”入 A.exe,也被合并入 B.exe,造成了事实上的空间浪费。另外，虽然说每人静态库的文件都比较小，但如果一个程序“合并”了不少库文件，那么这个程序的可执行文件体积仍然不可避免地变得比较大。

和静态库相对，另外一种库称为“动态库”。它的做法是：把所有函数不管三七二十一，都放在一个文件里。这样做的结果：库文件只有一个，但体积很大。

使用动态库的坏处是：动态库不允许“合并”到你的程序中－－显然也不适于合并，因为动态库太大了。所以若你使用动态库，在发布你的应用程序时，你必须向你的用户提供动态库文件。

使用动态库的好处在于：如果你向用户提供的是一套程序，比如有 A.exe,B.exe,C.exe...，那么这些可执行文件都可以使用同一个动态库，所以尽管你需额外提供一个很大的动态库，但你的各个应用程序却都很小。当然，采用动态库发布程序时，一般来说你还需要向用户提供一个安装程序，很多动态库要

被安装到 Windows 目录的 system 或 system32 子目录下。

什么时候使用静态库，什么时候使用动态库？当你只是写一个小小应用程序时，显然大多数人喜欢只提供一个单独.exe 文件。比如情人节到了，你觉得通过网络向你的 girlfriend 发一个电子贺卡太俗（前几年还很风雅呵：），同时也不能突显你作为一个程序员的实力－－风水轮流转啊，前年搞网络的人还笑话程序员是“传统工业”－－所以你决定用 C++ Builder 写一个电子贺卡，这时你可不能用动态库啊，否则挤爆了女友的信箱，嘿嘿，这个情人节就有你好受的了……

相反，一个稍大点软件系统，你就应该采用动态库。大的如整个 Windows 操作系统，就彻头彻尾是使用动态库；再如一整套 MS Office,还有 WPS，这些都是。一般地说（不绝对），那些提供了安装程序的软件，都是使用动态库的。总之，使用动态库是专业程序的做法。

（又有人举手打断我的课程，说我们什么时候才能自己写个电子贺卡？回答是下一部教程《白话 Windows 编程》，顺便说说，下部教程很贵很贵的－－吓你的：）

不管使用动态或静态的库，写程序时都是一样的。只有在最后要链接程序时，我们通过ＣＢ设置不同的选项即可。嗯？我说到了“链接”(link)这个词？对了，它就是我们一直加引号的“合并”一词的专业说法。你可以把前面课程上所有的“合并”一词替换为链接，并且不用加引号了。

现在我们来看看 CB 主要提供哪些类别的库函数（以下内容仅供了解）：

1、分类判断函数：

这类函数主要用对判断一个字符是什么类型的。就像我们上一章做的“判断用户输入字符的类型”的例子。不使用函数，我们可以这样的条件判断一个字符是否为小写字母：

```
if ( ch >= 'a' && ch <= 'z' )
     cout << ch << "是一个小写字母。" << endl;
```

我们也可以直接使用相关的库函数 islower：

```
if ( islower(ch) )
     cout << ch << "是一个小写字母。" << endl;
```

2、控制台输入输出函数：

像我们总是使用的 getchar(),及 getche();这两个函数用来接受用户在控制台程序中的按键输入。另外还有不和输入输出函数。当然，在输出方面，我们几乎都采用 cout 来往屏幕输出内容。cin, cout 这是Ｃ＋＋的方法，如果写Ｃ程序（而不是Ｃ＋＋），则输出更常用的是 printf();比如：

```
printf("Hello world!");
```

这行代码在屏幕上打出一行："Hello world!"。

除了教学上，或其它一些特殊要求，我们几乎不写控制台式的程序了，我们最终目标是写 Windows 下的ＧＵＩ（图形用户界面）程序，而这些控制台输入输出函数，都不能用在ＧＵＩ程序中。所以，当课程例中用到的某个控制台库函数，我会临时解释一下，其它的，大家就不必花时间了。

3、转换函数：

这类函数完成各种数据类型之间的转换，比如把字符串“123”转换数字 123，或把小写字母转换为大写字母等等。

5、目录管理函数：

目录就是我们现在常说的“文件夹”啦。这些函数可以建立，删除，切换文件夹。一般地，我们已经不再使用，转而使用 Windows 提供的相关函数。请参看下面的 Windows API 函数说明。

6、数学函数：

例如我们前面说的 sin()函数，其它的各种三角函数，还有求整，求绝对值，求随机数，求对数等。

这些函数大都枯燥无味，其中的随机函数倒是有趣点。很多游戏程序都要使用到它。这里粗略讲讲。

什么叫随机？大白话说就是：一件事情的结果有几种相同概率的可能。比如你扔一个硬币到地上，可能是正面，也可能是反面朝上，两种可能的概率都是 50%。但如果你要考虑硬币还有“立”着在地上的可能，那么这种可能就不属于随机的范畴了。下面的程序随机生成一个 0～99 的数,然后要求你输入一个 0～99 之间的数，如果这你输入的和它生成的数相等（概率为 1%），就表示你中奖了。

```
//虽然属于数学类函数，但随机函数其实放在标准库(stdlib)里：
#include <stdlib.h>
#include <iostream.h>

int main(int argc, char* argv[])
{
   //在第一次调用随机数之前，要调用一次这个函数,
   //这个函数称为“随机种子函数”
   randomize();
   //随机函数：random(int n)的用法：
   //随机返回一个 0～ (n-1) 之间的整数，
   //如： int x = random(100),则 x 值将是 0 到 99 之间的一个数。

   int x = random(100);
   int y;
   cout << "请输入一个 0～99 的整数：";
   cin  >> y;
   if( x == y) //可能性为 1%
      cout << "恭喜！您中奖了！" << endl;
   else
     cout << "谢谢使用。" << endl;
}
```

7、字符串函数：

我们在学习字符串时将用到。

8、内存管理管理函数：

我们在学习内存管理时将用到。

9、杂七杂八的其它函数。

这个且不说。

12.2.1.2 操作系统的 API 函数

大家总该知道什么叫操作系统吧？Windows 就是一套操作系统，另外如 UNIX,Linux 也是，当然我们最常用的是前者。操作系统有两个主要任务：

第一是给普通用户提供一套界面，比如桌面啦，任务条啦，及任务条上的开始按钮，桌面上的图标；还有资源管理器等等。这一些我们都称为“用户界面”。它的作用是让用户“用”这台电脑。因此我们也可以称它为用户与电脑之间的“接口”。

第二就是给我们这些程序员的接口，我们所写的程序是运行在操作系统上，就必须和操作系统有着千丝万缕的关系。比如我们想在屏幕上显示一个窗口，那么我们所做的事是“请求操作系统为我们在屏幕

上画一个窗口”，同样在有了窗口后，我们想在窗口上画一条直线，那么也是“请求操作系统在座标(2,1)-(100,200)之间画一条直线”。

那么，这些“请求”是如何实现的呢？其实也是调用函数，调用操作系统为我们准备的各种函数。这些函数同样是放在库文件里，当然，由于这些库文件是操作系统提供的，每一台装有相同操作系统的电脑都有这些库，所以它不用安装，所以它当然采用了动态库的形式。 对于我们正在用的 Windows，这些库一般都放在 Windows 的安装目录:Windows,主要是 Windows\System 或 System32 下。那里有一堆的.dll，其中有不少文件就是操作系统的动态库文件。

我们写的程序，一般称为“应用程序”(Application Program),所以 Windows 为我们提供的库函数也就称为“应用程序接口”(Application Program Interface)，缩写即：API。

在本部教程，我们主要学习 C++语言本身，只有学好 C、C++语言，才有可能学会用 C、C++语言来和操作系统打交道。要知道所有在 API 函数都声明为 C 语言的形式，这是因为，Windows 本身也是主要用 C 语言写成的。结论是：学习 C、C++语言非常重要，并且，如果想在操作系统上写程序，那么学习 C、C++当然最合算！

12.2.1.3 VCL 库函数

VCL 意为：可视化控件库（Visual Component Library），事事都直接和 Windows 的 API 打交道，编程效率将非常的低。主要表现两个方面：第一，由于使用 API 编程是非可视化的，我们将不得不花费非常冗长的时间在处理界面显示的事务上，而界面显示其实不是我们程序的主要逻辑。第二，有关显示等工作的大量代码事实上有很大的相似性，大量重复。我们要么仍受每写一个程序就重复写一堆千篇一律的代码，要么像早期的 Windows 程序员一样自己动手写一套的类库用来“包装”这段代码，以求每次可以得重复利用。但这是件庞大而灵活的工作，显然我们不值得这样做，事实上也不具备这样的能力。笔者在 Windows3.1 下写程序时，曾经购买过国人高手写的一套这种类库，事实上钱花得不值。很快笔者转向了当时 Borland 提供的类库：OWL 和微软的 MFC。

VCL 提供的也主要是类库，我们暂未学到“类”的概念，所以这时且不详谈。

12.2.2 调用者必须能“找”得到被调用者

调用函数前提之一：调用者必须能看到被调用者。

一个“者”字，可能让你以为这里说的是“人”，其实不是，这里说的调用者指的是当前程序，而被调用者当然是“将被调用”的函数。

不过，确实，这里拿人来比喻是再合适不过了。

就拿前面说的“找电视修理工”的例子来说：

要修电视，显然要能找到电视修理工。这个道理很明显。

所以本小节的重点其实是：程序如何才能找到要调用的函数呢？

有三种方法：

第一种、将被调用的函数写在当前代码前面

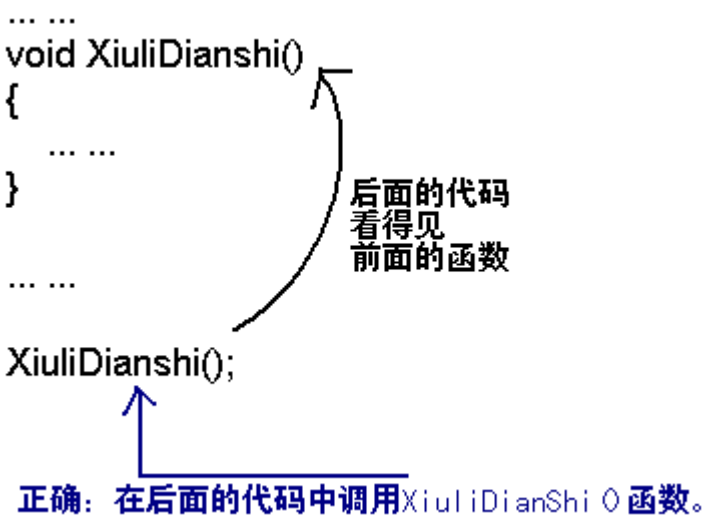
修理工正在我家喝茶呢！是啊，我有个朋友是干这活的，有一天他来我家串门，而我家电视正好坏了。

下面我先写一个函数，这个函数的大部分代码我没有写出来－－根本写不出来。我只是要用它表示一个叫“修理电视”的功能。

```
//本函数实现“修理电视”
void XiuliDianshi()
{
   ......
}
```

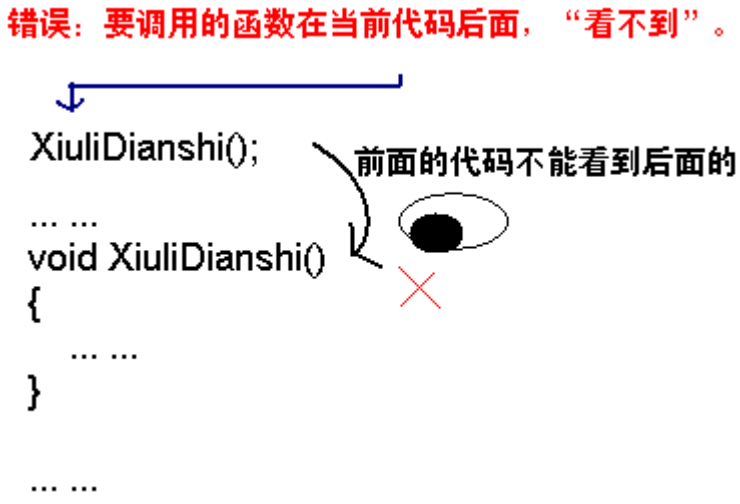
尽管我们稍后才能学如何自己写函数，但你现在要记住了，上面那几行代码就是一个函数，它的函数名为：XiaoliDianshi，意为“修理电视”。

好！有了“修理电视”的函数了，如何调用它呢？下图表示的是正确的情况：



当我们写程序要调用一个函数，而这个函数位于我们现在在写的代码前面时，我们就可以直接调用它，这就像修理工就在我们家里一样。注意这里的前面并非仅限于“跟前”，如果你的代码很多行，这个函数在“很前面”，也不妨碍我们调用它。

要注意的是另一面：当函数在我们的代码后面时，代码就“看”不见这个函数了。下面即为这种错误情况：



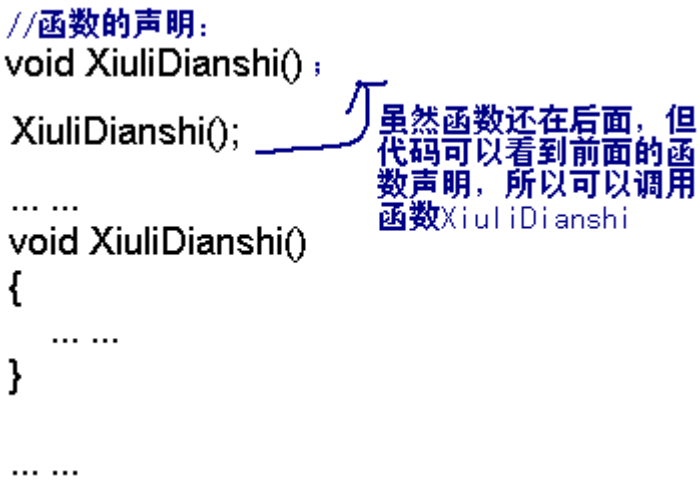
第二种、将被调用的函数声明写在当前代码前面

修理工不在我家，不过，他曾经留给我一张名片，名片上写着：“张三，电视修理工，Tel:1234567，住址：……”。所以我们也能知道他会修理电视，并且知道他的电话和住址，这样就不愁找不到他了，

对不？

函数也可以有名片，在程序中我们称为函数的“声明”。

下面的代码演示了什么叫函数的“声明”，及它所起的作用：



第三种：使用头文件

当我们手里有了电视修理工的名片，有了冰箱修理工的名片，有了电脑修理工的名片……名片多了，我们可以将名片整理到一个名片夹。这样做至少有两个好处：

其一：便于管理。家里任何电器坏了，只需找“家用电器修理工名片”的名片夹即可。

其二：便于多人共用，比如隔壁家想找一个电视修理工，只需上你家借名片夹即可。

C，C++中，类似“名片夹”功能的文件，称为“头文件”。头文件的扩展名为 .h（head）。头文件是放置函数声明的好地方。如何写函数声明下面再说，现在要明白，“函数声明”就是给编译器看的函数说明，或曰函数的“自我介绍”。至于为什么叫“头”文件呢？是因为它总是要在程序代码文件的开头。就你我们在交谈时，开头总是大家各作一番介绍一样。（该说法未经证明，仅供参考：）

说千道万，不如先简单地看一眼真实的头文件吧。

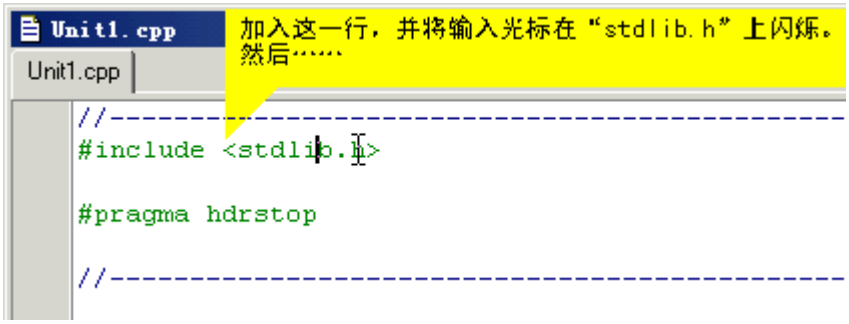
启动 C++ Builder。然后新建一个控制台应用工程。在 CB6 里，新建控制台工程在 File｜New｜Others 去找，别忘了。

（CB6 启动为什么这么慢啊！我且先上趟洗手间）

然后在代码窗口里，加上一行：

```
#include <stdlib.h>
```

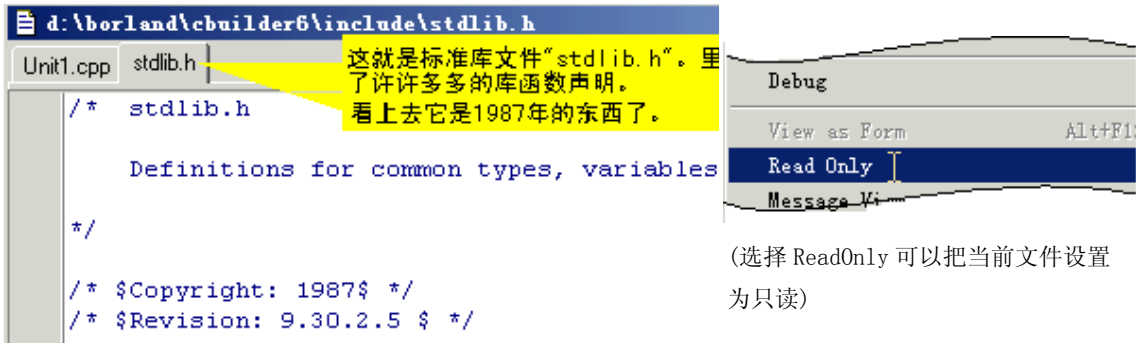
并且用鼠标在这一行点一下，现在代码窗口里的内空看起来如下：



确保输入光标在单词“stdlib.h” 上面闪烁！现在按 **Ctrl + 回车**,CB 将打开光标所在处的文件。

（如果你出现的是一个文件打开对话框，那有两点可能，其一是你没有把光标移到指定的单词上，另一可能是你安装 CB 时没有选择“Full”模式的安装，造成 CB 没有安装源文件。）

以下就是打开的 stdlib.h 头文件：



(选择 ReadOnly 可以把当前文件设置为只读)

打开的文件是 C++ Builder 工程师为我们所写的头文件，请注意千万不要有意无意地改动它！为了保险起见，通过右键菜单，选择 Read Only 将当前文件设置为只读(如上面右图)。请大家将这当作一条准则来执行：不管出于什么原因打开 CB 提供的源文件，立即将其设置为只读。

好，我们说过“只看一眼”的。关于头文件,在讲完函数以后，还会专门讲到头文件在工程中应用。现在重复头文件的目的：

函数可以统一在一个头文件中声明，代码中需要使用这些函数，只需通过“include”语句包含这个头文件，就可以让编译器找到函数。

用一句大白话讲就是：要想用函数？请包含它所在的名片夹（头文件）。

函数的“声明”有时被称为函数的“原型”，比如在讲到编译过程时。当我们阅读其它文章时，如果看到“函数原型”一说，希望大家也能明白。

12.2.3 调用者必须传递给被调用者正确的参数

现在，我能找到修理工，而且他已经到我家。

“电视呢？”他说。

“就是它”我指着家里的苏泊尔高压锅，“劳驾，把它修修，最近它总漏气。”

“可是，我好象是来修理电视的？”

“知道，现在你先修高压锅。”

“好吧，我试试……先用电笔试试它哪里短路。”

显然我这是在胡搅蛮缠。电视修理工要开始干活，就得给他电视。给他一只高压锅他不能开工。

函数也一样，函数的目的是实现某个特定功能，当我们调用它时，我们一般需要给它一些数据，这些数据可能是让它直接处理，也可能是辅助它实现具体的功能。

当然有些函数不需要任何外部数据，它就能完成任务。这也很好理解，修理工修理电视是得有台电视，但叫一位歌手到家里随便哼几句歌，你就不用给他什么。

关键一句话：**函数要不要外部传给它数据，要什么类型的数据，要多少数据，由函数本身决定，而非调用者决定**。本例中，电视修理工需要一台电视，这是他决定的，不能由请他的人决定。

传给函数的数据，我们称为“参数”，英文为：parameter。

基于此，我们发现所写的 XiuliDianshi()函数有很大的不足，那就是它没有参数。现在我们假设有一种数据类型为“电视机”，嗯,就假设这种数据类型叫作：TDianshi。

加入参数的 XiuliDianShi()函数变为：

```
XiuliDianshi (TDianshi ds)
{
}
```

看一个实际的例子。上一章我们曾经学过 sin()函数。现在我们来看看 sin()函数的声明。看看它声明需要什么参数。

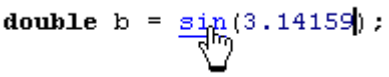
关闭刚才的工程，CB 会问你是否存盘，统统不存（如果你要存，就存到别的什么地方去，不要存在 CB 默认的目录下）。然后重新创建一个空白的控制台工程。在代码窗口里加入以下两行黑体代码：

```
//---------------------------------------------------------------------------
//包含“数学库函数”的头文件，因为 sin()函数的声明在这个头文件里：
#include <math.h>
#pragma hdrstop
//---------------------------------------------------------------------------
#pragma argsused
int main(int argc, char* argv[])
{
   double b = sin(3.14159);
   return 0;
}
//---------------------------------------------------------------------------
```

并不需要编译及运行这个程序。因为我们只是想找到 sin()函数的声明。

本来，我们可以通过老办法来找到 sin 函数声明。按 Ctrl+回车键打开 math.h 文件，然后通过 Ctrl+F 打开查找对话框，找到 sin 函数。不过 CB 为我们提供了一种更方便的查找函数声明的方法，有点像我们在网页点击链接：

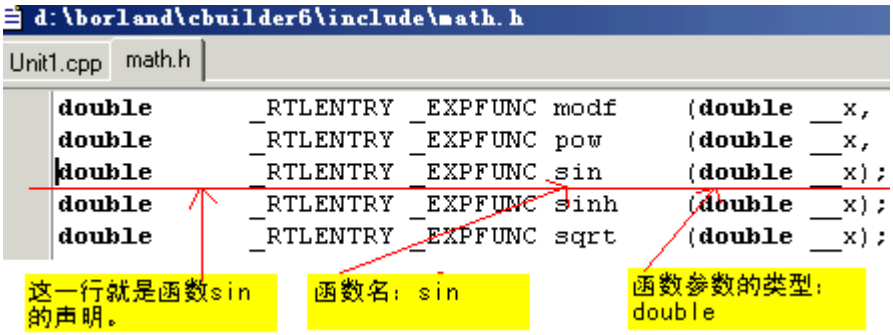
请按住 Ctrl 键不放，然后将鼠标移到代码中的 sin 处，注意要准确在移到 sin 字母身上，发现什么？呵，sin 出现了超链接效果：



点一下，CB 将自动打开 math.h 头文件，并且跳转到 sin 函数的声明处。

（以上操作的成功依赖于你正确地照我说的，在代码中加入#include<math.h>这一行，当然你在安装

CB 时也必须选择了安装源代码。最后，成功打动后，记得将 math.h 文件设置为只读。）



从图中我们看到，sin 函数的参数只有一个：__x,类型要求是 double（双精度浮点数,如果你忘了，复习第四章）。

所以，当我们调用 sin 函数来求正弦值时，我们最好应该给它一个 double 类型的数，如：

```
double x = 3.1415926 * 2;
double y = sin(x);
```

当然， 我们传给它一个整数:

```
double y = sin(0);
```

或者，传给它一个单精度浮点数：

```
float x = 3.14;
double y = sin(x);
```

这些都是可以的。这并不违反“参数类型由函数本身决定，不能由调用者决定”的原则。因为在第七章第二节讲算术类型转换时，我们知道一个整数，单精度浮点数，都可以隐式地转换为双精度浮点数。并且属于安全的类型转换，即转换过程中，数据的精度不会丢失。（反过来。一个 double 类型转换为 int 类型，就是不安全的转换。比如 3.14159 转换为整型，就成了 3。）

有些函数并不需要参数，比如，我们用了许多次的控制台函数：getchar();。这个函数要做的事就是：等待用户输入一个字符并回车。前面讲数学函数时，举的随机数例子。要想让程序能够产生真正的随机数，需要让程序事先做一些准备。所以我们调用 randmize()函数。这个函数也没有参数。因为我们调它的目的，无非是：喂，告诉你，我一会儿可能要用到随机数，你做好准备吧。

12.2.4 如何得到函数的运行结果

函数总是要实现一定的功能，所以我们也可以认为函数执行起来就像是在做一件事。

做一件事一般会有个结果，当然，只是“一般会有”。有些事情真的会有结果吗？嗯？看来，这句话勾起某些同学一些旧事，他们陷入了深深的，似乎很痛苦的回忆之中……对此，为师我表示最大的理解，并有一言相送：“并非是一件事情不会有结果，只是，有时候，我们并不需要结果……”。

写函数的人就是这样的啊。函数需要什么参数，由写函数的人决定，函数返回什么结果，也由他们决定。如果他们认定这个函数不需要什么结果，那么这个函数就将写成返回 void 类型。void 是“无类型”之意，这就相当于这个函数没有返回结果。

举修理电视的例子来说，我们认为它至少应该返回一个 bool 值，即真或假。真表示电视修好了，假表示电视修不好。

```
bool XiuliDianshi(TDianshi ds);
```

然后，我们如何得知结果呢？

```
bool jg = XiuliDianshi(ds);
```

看，我们也声明了一个 bool 变量，然后让它等于这个函数，这就可以得到函数的返回值。

来看一个实例，仍然是 sin 函数。

double x = 3.1415926;
double y = sin(x);
y 值将是一个非常接近 0 的值。

getchar();是一个不需要参数的函数，但它有返回值。它返回用户输入的字符（事实上它返回的是整型）。所以我们可以这样用：
char c = getchar();
c 将得到用户输入的字符。
而另一个例子： randomize()函数，则赤条条地来，赤条条地走，潇洒得很。根本就不打算返回什么。连到底准备成功了吗？都不返回－－因为它认定自己一定会执行成功。

还需说明的是，有时函数是有返回值，但我们并不在意。还是 getchar();我们不是一直在使用它来“暂时”停止程序，以期能看到 DOS 窗口上的输出结果吗？这时，用户输入什么键我们都不在意。所以我们总这么写：
getchar();
就完事，并没让谁去等于谁。

最后一点针对学过 PHP，JavaScript,Perl 等脚本语言的学员：在 C,C++里，一个函数返回值的类型，必须是确定的。不像脚本语言中的函数，可以返回不定类型的结果。

12.2.5 调用库函数的实例

实例一：使用库函数创建或删除文件夹。
（本例子中删除的文件夹将无法恢复！请大家操作时小心。）

在本实例里，我们将“大胆地”在 C 盘根目录下创建指定的目录（文件夹），然后再把它删除。
使用到两个函数：
1、mkdir("文件夹名称")
参数是一个字符串，即指定的文件夹名称。
返回值比较特殊：整数：0 表地成功，-1 表示失败：比如那个文件夹已经存在，或者，你想让它一次创建多级目录，如：C:\abc\123,而 C:\abc 并不存在。
2、_rmdir("文件夹名称")
参数是一个字符串，即指定的文件夹名称。
返回同样是 0 或-1。删除一个文件夹比较容易失败：比如文件夹内还有文件或其它子文件夹，比如该文件夹正好是当前文件夹，另外你也不能删除一个根目录，比如你想删除："c:\" ！！！（想删除整个 C 盘？病毒？）

两个函数都在“dir.h”文件里声明，所以我们需要 include 它。

下面是完整的代码：

```
//---------------------------------------------------------------------------
#include <dir.h>
#include <iostream.h>
#pragma hdrstop
//---------------------------------------------------------------------------

#pragma argsused
```

```
int main(int argc, char* argv[])
{
   char path[50];
   char ch;

   do
   {
      //让用户选择操作项：
      cout << "0、退出本程序" << endl;
      cout << "1、创建文件夹" << endl;
      cout << "2、删除文件夹" << endl;
      cout << "请选择：";
      cin >> ch;

      //如果输入字符'0'，则结束循环以退出:
      //请注意 break 在这里的用法：
      if(ch == '0')
      {
         break;
      }

      //如果输入的既不是 1，也不是 2，要求重新输入,
      //请注意 continue 在这里的用法:
      if(ch != '1' && ch != '2')
      {
          cout << "输入有误，请重新选择！" << endl;
          continue;
       }

       //不管是创建还是删除，总得要用户输入文件夹名称：
       cout << "请输入文件夹的绝对路径：" ;
       cin >> path;

       //先定义一个 bool 变量，用来判断操作是否成功:
       bool ok;
       //现在需要区分用户想做什么了：
       if(ch == '1') //创建文件夹：
       {
           ok = (0 == mkdir(path)); //若 mkdir 返回结果等于 0，表示操作成功
       }
       else //否则就是要删除了！
       {
           ok = (0 == _rmdir(path)); //同样，_rmdir 也是返回 0 时表示成功
        }
```

```
            //给出结论：
            if(ok)
            {
                cout << "恭喜！操作成功。" << endl;
            }
            else
            {
                cout <<"抱歉，操作失败，请检查您的输入。" << endl;
            }
        }
        while(true);

        return 0;
    }
    //-------------------------------------------------------------------------
```

代码里头有一个 do...while 循环，一个 continue,和 break;另有几个 if...else，这些相信你可以边运行程序，边看明白其间的逻辑。惟一陌生的是最开头的一句：

```
char path[50];
```

这里涉及到了“数组”的知识。针对本例，你可以这样理解：

```
char ch;
```

这一行我们能看懂，定义了一个字符类型的变量，ch。ch 变量的空间是 1 个字节，能存储一个字符，因此你可以用它存储诸如：'A','2','H'等，但现在我们要输入的是："c:\abcd"这么一句话，所以变量 ch 无法胜任。C,C++提供了数组，我们可以通过定义数组来存储同一类型的多个数据。如：

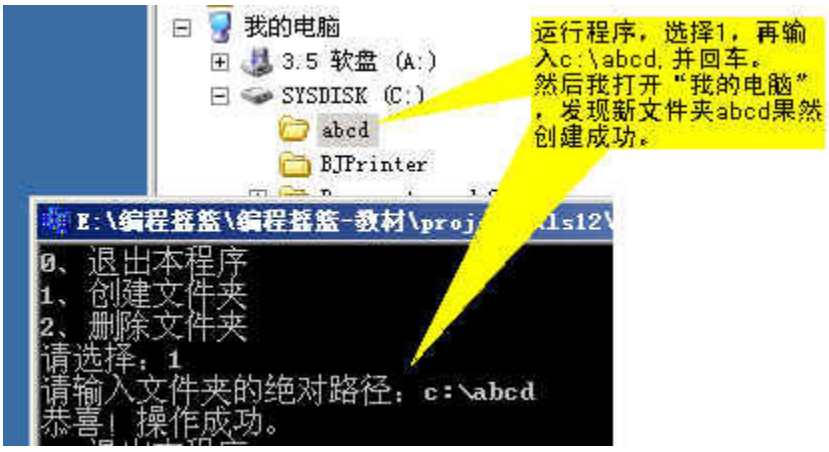
```
char path[50];
```

本行定义了 path 这个数组，它可以存储 50 个 char 类型的数据。

注意，path 只能存储最多 50 个字符。所以在运行本例时，不要输入太长的文件夹名称。

另外，Windows 对新建文件夹的名称有一些特殊的要求，所以如果文件夹名称含了一些非法字符，操作将失败。

以下是我运行的一个结果：



12.3 自定义函数

学会如何调用别人的函数，现在我们来学习如何自己写一个函数。首先迅速看看函数的格式：

12.3.1 函数的格式

定义一个函数的语法是：

```
返回类型  函数名(函数参数定义)
 {
  函数体

  return 结果;
}
```

其中：

1、返回类型： 指数据类型，如：int ,float,double, bool char ,void 等等。表示所返回结果的类型。如果是 void 则表示该函数没有结果返回。

2、函数名：命名规则和变量命名一样。注意要能够表达出正确的意义。如果说一个变量命名重在说明它“是什么”的话，则一个函数重在说明它要“做什么”。比如一个函数要实现两数相加，则可以命名为：AddTwoNum,这样有助于阅读。

3、函数参数定义：关于参数的作用，我们前面已说。现在看它的格式：

```
int AddTwoNum(int a,int b);
```

函数参数的定义有点类似定义变量，先写参数的数据类型，上例中是 int,然后再写参数名。下面是不同之处：

3.1 多个参数之间用逗号隔开，而不是分号。最后一个变量之后则不需要符号。 请对比：

普通变量定义：

```
int a;  //<--以分号结束
int b;
```

函数中参数定义：

```
(int a, int b ) //以逗号分隔，最后不必以分号结束
```

3.2 两个或多个参数类型相同时，并不能同时声明，请对比：

普通变量定义：

```
int a,b; //多个类型相同的变量可以一起定义。
```

函数中参数定义：

```
AddTwoNum(int a, b) //这是错误的。
```

4、函数体：函数体用一对｛｝包括。里面就是函数用以实现功能的代码。

5、return 结果：return 语句其实属于函数体。由于它的重要性，所以单独列出来讲。“return”即“返回”，用来实现返回一个结果。“结果”是一个表达式。记住：当函数体内的代码执行到 return 语句时，函数即告结束，如果后面还有代码，则后面的代码不被执行。依靠流程控制，函数体里可以有多个 return 语句。当然，对于不需要返回结果的函数，可以不写 return 语句，或者写不带结果的 return 语句。这些后面我们都将有例了演解。return 返回的结果，类型必须和前面“返回类型”一致。

一个最简单的例子，也比一堆说明文字来得直观，下面我写一个函数，用于实现两个整数，返回相加的和。这当然是一个愚不可及的函数，两数相加直接用+就得，写什么函数啊？

```
//愚不可及的函数：实现两数相加
//参数：a：加数 1，b：加数 2；
//返回：相加的和
int AddTwoNum(int a, int b)
```

```
{
   return a + b;
}
```

例子中，谁是“返回类型”，谁是“函数名”？谁是“参数定义”？哪些行是“函数体”？这些你都得自己看明白。这里只想指出：这是个极简单的函数，它的函数体内只有一行代码：即 return a+b;语句，直接返回了 a＋b 的结果。

最后说明一点：C，C＋＋中，不允许一个函数定义在另一个函数体内。

```
void A()
{
   void B()  //错误：函数B定义在函数A体内。
   {
     ....
   }

   ...
}
```

如上代码中，函数 B“长”在函数 A 体内，这不允许。不过有些语，如 Pascal 则允许这样定义函数。

12.3.2 自定义函数实例

下面我们将动手写几个函数，并实现对这些函数的调用。从中我们也将进一步理解函数的作用。

12.3.2.1 小写字母转换为大写字母的函数

实例二：自定义小写字母到大写字母的转换函数。

尽管这个功能很可能已经有某个库函数实现了，但像这种小事，我们不妨自己动手。

之所以需要这个函数，缘于最近我们写程序时，经常用到循环，而循环是否结束，则有赖我们向用户提一个问题，然后判断用户的输入；如果用户输入字母 Y 或 y，则表示继续，否则表示退出。

每次我们都是这样判断的：

```
if(ch == 'Y' || ch == 'y')
{
    ...
}
```

平常我们的键盘一般都是在小写状态，因为用户有可能不小心碰到键盘的“Caps Lock”，造成他所输入的任何字母都是大写的－－尽管键盘上有个大小写状态指示灯，但有谁会去那么注意呢？所以如果你的程序仅仅判断用户是否输入‘y’字母，那么这个用户敲了一个‘Y’，结果程序却“很意外”的结束了？显然这会让用户很小瞧你：才三行程序就有 BUG。

（一般不传之秘笈：用户就像女友一样，需要“哄”：有时你发现软件中存在一项潜在的，系统级的严重 BUG，你自己惊出一身冷；但在用户那里，他们却纠缠你立即改进某个界面上的小小细节，否则就要抛弃这个软件－－就像你的女友，天天和你吃萝卜秧子没有意见，但情人节那天忘了送花，她就对你失

望透了。）

言归正传！现在问题，我讨厌每回写那行条件都既要判断大写又要判断小写。解决方法是，在判断之前，把用户输入的字母统统转换为大写！

下面是将用户输入字符转换为大写的函数。要点是：

1、用户输入的字符不一定是小写字母，说不定已经是大写了，甚至可能根本就不是字母。所以在转换之前需要判断是否为小写字母。

2、小写字母‘a’的 ASCII 值比大写字母‘A’大 32，这可以从第五章的 ASCII 码表中查到。不过我不喜欢查表，所以最简单的方法就是直接减出二者的差距。所有字母的大小之间的差距都一样。这是我们得以转换大小写字母的前提。

```
//函数：小写字母转换为大写字母。
//参数：待转换的字母，可以不为小写字母；
//返回：如果是小写字母，返回对应的大写字母，否则原样不动返回。

char LowerToUpper(char ch)
{
   //判断是否为小写字母：
   if(ch >= 'a' && ch <= 'z')
   {
      ch -= ('a' - 'A'); //相当于 ch -= 32; 或 ch = ch - 32;
   }

   //返回：
   return ch;
}
```

这个函数也再次提醒我们，在 ASCII 表里，大写字母的值其实比小写字母小。所以，小写字母转换为大写，用的是“减”。小写字母减去 32，就摇身一变成了大写。

现在，有了这个函数，假设我们再遇上要判断用户输入是’y’或‘n’的情况，我们就方便多了。

作为一种经历，我们此次采用将函数放在要调用的代码之前。

```
//-------------------------------------------------------------------------
//函数：小写字母转换为大写字母。
//参数：待转换的字母，可以不为小写字母；
//返回：如果是小写字母，返回对应的大写字母，否则原样不动返回。

char LowerToUpper(char ch)
{
   //判断是否为小写字母：
   if(ch >= 'a' && ch <= 'z')
   {
      ch -= ('a' - 'A'); //相当于 ch -= 32; 或 ch = ch - 32;
```

```
    }

    //返回：
    return ch;
}
//---------------------------------------------------------------------------
int main(int argc, char* argv[])
{
    char ch;
    do
    {
        cout << "继续吗？(Y/N)";
        cin >> ch;

        //调用函数，将可能小写字母转换为大写：
        ch = LowerToUpper(ch);
    }
    while(ch == 'Y');

    return 0;
 }
//---------------------------------------------------------------------------
```

完整的代码见相应例子文件。例子只是为了演示如何自己定义函数，并调用。运行时它问一句“继续吗？”你若输入大写或小写的‘y’字母，就继续问，否则结束循环。

函数的返回值也可以直接拿来使用。上面代码中的 do...while 循环也可以改写的这样：

```
    do
    {
        cout << "继续吗？(Y/N)";
        cin >> ch;
    }
    while(LowerToUpper(ch) == 'Y');
```

功能完全一样，但看上去更简洁。请大家进行对比，并理解后面的写法。

本例中的“小写转换大写”的函数，虽然我们已经成功实现，但我们并没有将它的声明放到某个头文件，所以，如果在别的代码文件中，想使用这个函数，还是不方便。确实，我们很有必要为这个函数写一个头文件，在讲完函数后，我们将去做这件事。

实例二代表了一种函数的使用需求：我们将一些很多代码都要使用的某个功能，用一个函数实现。这样，每次需要该功能时，我们只需调用函数即可。这是函数的一个非常重要的功能：代码重用。通过函数，不仅仅是让你少敲了很多代码，而且它让整个程序易于维护：如果发现一某个功能实现有误，需要改正或改进，我们现在只需修改实现该功能的函数。如果没有函数？那将是不可想像的。

但是，只有那些一直要使用到的代码，才有必要写成函数吗？并不是这样。有些代码就算我们可能只

用一次，但也很有必要写在函数。请看下例。

12.3.2.2 使用函数改写“统计程序”

实例三：使用函数改写第十章“可连续使用的统计程序”。

我们先把第十章的例子拷过来（只拷其中的 main()函数部分）：

```
int main(int argc, char* argv[])
{
   float sum,score;
   int num; //num 用于存储有几个成绩需要统计。
   int i;   //i 用于计数

   char c; //用来接收用户输入的字母

   do
   {
     //初始化：
      sum = 0;
      i = 1;

      cout << "====成绩统计程序====" << endl;
      //用户需事先输入成绩总数：
      cout << "请输入待统计的成绩个数：";
      cin >> num;
      cout << "总共需要输入"<< num << "个成绩(每个成绩后请加回车键)：" << endl;

      while ( i <= num)
      {
         cout << "请输入第" << i << "个成绩：";
         cin >> score;
         sum += score;
         i++;
      }

      //输出统计结果:
      cout << "参加统计的成绩数目:" << num << endl;
      cout << "总分为：" << sum << endl;

      //提问是否继续统计：
      cout <<"是否开始新的统计？(Y/N)?";
      cin  >> c;
    }
    while( c == 'y' || c == 'Y');
}
//-------------------------------------------------------------------------
```

我们将要对这段代码所作的改进是：将其中完成一次统计功能的代码，写入到一个单独的函数。

```
//函数：实现一个学员的成绩统计：
//参数：无
//返回：无
void ScoreTotal()
{
   float sum,score;
   int num; //num 用于存储有几个成绩需要统计。
   int i; //i 用于计数

   sum = 0;
   i = 1;

   cout << "====成绩统计程序(Ver 3.0)====" << endl;
   //用户需事先输入成绩总数：
   cout << "请输入待统计的成绩个数：";
   cin >> num;
   cout << "总共需要输入"<< num << "个成绩(每个成绩后请加回车键)：" << endl;

   while ( i <= num)
   {
      cout << "请输入第" << i << "个成绩：";
      cin >> score;
      sum += score;
      i++;
   }

   //输出统计结果:
   cout << "参加统计的成绩数目:" << num << endl;
   cout << "总分为：" << sum << endl;
 }
 //-------------------------------------------------------------------------
```

我只是将一些代码从在原来的位置抽出来，然后放到ScoreTotal()函数体内。接下来，请看原来的main()函数内的代码变成什么：

```
//-------------------------------------------------------------------------
int main(int argc, char* argv[])
 {
   char c;

   do
   {
      //调用函数实现一次统计：
      ScoreTotal();
```

```
        //提问是否继续统计：
        cout <<"是否开始新的统计？(Y/N)?";
        cin >> c;
      }
      while(c == 'Y' || c == 'y');
   }
//--------------------------------------------------------------------------
```

看，当实现统计一次的功能的代码交由 ScoreTotal()处理之后，这里的代码就清晰多了。

函数的另一重要作用：通过将相对独立的功能代码写成独立的函数，从而使整体程序增加可读性，同样有益于代码维护。这称为“模块化”的编程思想。“模块化”的思想并不与Ｃ＋＋后面提倡的“面向对象”的编程思想相抵触。而函数正是Ｃ，Ｃ＋＋中实现“模块化”的基石。

实例三的演变过程也向我们展示了一种编写程序的风格：当一个函数中的代码看上去很长时，你就应该去检查这段代码，看看中间是否有哪些逻辑是可以独立成另外一个函数？在本例子中，main()函数中套了两层循环，但这两种循环相互间没有多大逻辑上的联系：内层用于实现一次完整的统计功能，外层则只负责是否需要继续下一次的统计。所以，把内层循环实现的功能独立“摘”出去，这是一个非常好的选择。

我们阅读ＶＣＬ的源代码时（用 Pascal 实现），发现尽管ＶＣＬ是一套庞大的类库，但其内部实现仍保持了相当好的简约风格，很少有代码超过２０００行的函数。这的确可以作为我们今后编写软件的楷模。

本例的完整请见相关例子文件。其中我还把前例的 LowerToUpper()函数也加入使用。

12.3.2.3 求多种平面形状的面积

实例四：写一程序，实现求长方形，三角形，圆形，梯形的面积，要求各种形状分别用一个函数处理。

程序大致的流程是：

首先提问用户要求什么形状态的面积？然后根据用户的输入，使用一个 switch 语句区分处理，分别调用相应的函数。求不同形状态的面积，需要用户输入不同的数据，基于本程序的结构，我们认为将这些操作也封装到各函数比较合适。

先请看 main()函数如何写：

```
int main(int argc, char* argv[])
  {
          char ch;
    do
    {
          cout << "面积函数" <<endl;

          cout    <<"0、退出 "<< endl //<--没有分号！用一个 cout 输出多行，只是为了省事
                  << "1、长方形" << endl
                        << "2、三角形" << endl
                  << "3、圆形" << endl
                  << "4、梯形" << endl;  //<--有分号
```

```
            cin >> ch;

            if(ch == '0')
                    break;

            switch(ch)
            {
                    case '1' : AreaOfRect(); break; //长方形
                    case '2' : AreaOfTriangle(); break; //三解形
                    case '3' : AreaOfRound(); break; //圆形
                    case '4' : AreaOfTrape(); break; //梯形

                    default :
                            cout <<"输入有误，请在 0 ～ 4 之间选择。" << endl;
            }
    }
    while(true);
}
```

函数 main()的任务很清晰：负责用户可以连续求面积，这通过一个 do...while 实现，同时负责让用户选择每次要计算面积的形状，这通过一个 switch 实现。而具体的，每一个平面图形的面积计算，都通过三个自定义的函数来实现。尽管我们还没有真正实现（编写）这三个函数，但这并不影响我们对程序整体架构的考虑。

当我们学会如何编写函数的时候，我们就必须开始有意识地考虑程序架构的问题。如果说变量，表达式等是程序大厦的沙子，水泥；而语句是砖头钢筋的话，那么函数将是墙，栋梁。仅仅学会写函数是不够的，还需要学习如何把一个大的程序分划为不同的功能模块，然后考虑这些模块之间的关系，最终又是如何组合为完整系统。

实例四的目的在于向我们演示：当你写一个程序时，有时候你不必去考虑一些小函数的具体实现，相反，你就当它们已经实现了一样，然后把精力先集中在程序总体架构上。

这种写程序的方法，我们称为“由上而下”型,它有助于我们把握程序主脉，可以及时发现一个程序中潜在的重要问题，从而使我们避免在开发中后期才发现一些致命问题的危险；同时也避免我们过早地在一些程序上的枝节深入，最终却发现这些枝节完全不必要。

不过，当程序很庞大时，想一次性理清整个程序的脉胳是不可能的，很多同样是重要的方向性修改都必须在对具体的事情有了分析后，才能做出准确的调整。另外，采用“由上而下”的开发方法时，有时也会遇上开发到后期，发现某些枝节的难度大大超过来原来的预估，需要占用过多开工期，甚至可能根本无法实现的危险。所以，我们还得介绍反方向的方法“由下而上”法。

采用“由下而上”时，我们会事先将各个需要，或者只是可能需要的细小功能模块实现出来，然后再由这些模块逐步组合成一个完整系统。采用由下而上的方法所写的代码还易于测试，因为这种代码不会过早地与其它代码建立关系，所以可以独立地进行测试，确保无误后，再于此基础上继续伸展。

一个小实例子引出这个大话题，有些远了，只是希望学习我的教程学员，能比其它途径学习编程的人，多那么一点“前瞻”能力。

最后，我给出 AreaOfRect()函数的完整代码。另外几个函数，有劳各位自己在实例的源代码添加完整。

```
void AreaOfRect()
{
```

```
    int x,y;

    cout << "请输入长方形的长:";
    cin >> x;
    cout << "请输入长方形的宽:";
    cin >> y;

    cout <<"面积为：" << (x * y) << endl;
}
```

12.4 主函数

Ｃ，Ｃ＋＋被称为“函数式”的编程语言。意指用这门语言写成的程序，几乎都由函数组成，程序运行时，不是在这个函数执行，就是在那个函数内执行。整个程序的结构看上去类似：Ａ函数调用Ｂ函数，Ｂ函数又调用Ｃ函数，而Ｃ函数则可能调用了Ｄ函数后又继续调用Ｅ函数……甚至一个函数还可以调用自身，比如Ａ函数调用Ａ函数，或Ａ调用Ｂ，而Ｂ又反过来调用Ａ等等……

问题是最开始运行的，是哪个函数？

最开始运行的那个函数，称为主函数。主函数在控制台或ＤＯＳ应用程序中。为 main()函数。在标准的 Windows 应用程序中，则为名为 WinMain()。

12.4.1 ＤＯＳ程序的主函数

控制台应用程序的主函数：main()我们已经很“熟悉”了，每回写程序都要用到它，只是我们没有专门讲到它。现在回头看看：

```
int main(int argc, char* argv[])
 {
    ……
    return 0;
}
```

ｍａｉｎ函数的返回值数据类型为 int,参数定义：int argc,char* argv[]的具体含义我们暂不用关心，只需知道，ＤＯＳ程序或控制台程序中，程序运行时的入口处就是 main()函数。

12.4.2 Windows 程序的主函数

我们先来创建一个 Windows 应用程序。请注意看课程,不要轻车熟路地生成一个“控制台”工程。

请打开ＣＢ，（如果你正打开着ＣＢ，请先关闭原来的工程），然后选择主菜单 File | New | Application,如果是ＣＢ５，选择 File | New Application。

下一步请选择主菜单 Project | View Source，该命令将让ＣＢ在代码窗口中打开工程源文件，主函数 WinMain 正是在该文件中。请你在工程源文件（默认文件名：Project1.cpp)中找到 WinMain()。

```
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
```

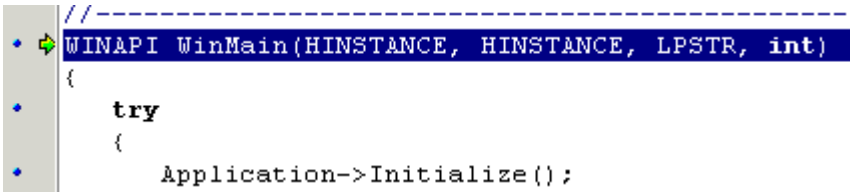
这行代码看上去很复杂，但无变不离其宗，你现在尽可以从位置上判断：函数名无疑是 WinMain()，而 WINAPI 估计是“返回类型”，至于"HINSTANCE, HINSTANCE, LPSTR, int"则必是参数定义。尽管还有些细节需要确定，但我们现在能够看懂这些就已经是９５％掌握了学习的重点。其它的且先放过。

由于现在我们很少采用 Windows 程序来做来实例，所以有必要验证一番，WinMain 是否真的是 Windows

应用程序运行时的第一个函数？

还记得 F8 或 F7 吗？（有个女生站起，声音响亮：我记得 F4!!!没听说要扩编为 F8 啊？）

在调试程序时，Ｆ 8 或Ｆ 7 键可以让程序单步运行，现在我们就来按一次Ｆ 8，看看程序“迈”出的第一步是否就是 WinMain()函数？请在Ｃ Ｂ里按Ｆ 8 。



程序先编译一番，然后便如上图直接运行到 WinMain()这一行。我们这一章的任务也就完成了。按Ｆ 9 让程序恢复全速运行。然后关闭Ｃ Ｂ（不用存盘）。

12.5 小结

函数的声明起什么作用？

函数的参数起什么作用？

return　语句起什么作用？

大致说说动态库与静态库各自的优缺点？

函数带来的哪两个主要用处？

不看课程，你能自己写出小写字母转换大写的函数吗？

什么叫“由上而下”的编程方法，什么叫“由下而上”的编程方法？

什么叫主函数函数？ＤＯＳ程序和 Windows 的主函数一样吗？

第十三章　函数（二）

上一章我们讲了函数最基本的知识，即如何函数调用一个函数，和如何写一个函数。这一章我们的任务是：重点加深学习函数的返回值和函数的参数；另外我们还将选修函数的递归调用。

通过对这两个知识点的深化学习，我们对函数的理解会更深。

13.1 函数的返回值

有关函数的返回值，将涉及到函数的这些知识点：函数的类型，return,及如何得到函数的返回类型。

13.1.1 函数的返回类型

函数的类型，其实是函数返回值的类型。请看例子：

```
//实现两个整数相加的函数：
int AddTwoNum(int a,int b)
{
   return a + b;
}
```

上面标为红色的 int 即为函数 AddTwoNum 的类型，普通的说法是“函数 AddTwoNum 的返回类型是整型”。也就是说函数 AddTwoNum 只能返回整型的值。我们看代码：

```
return a + b;
```

返回了 a + b，其中 a 和 b 都是整型，二者相加也是整型。所以这个函数的返回类型正确。下面看一个错误的实例：

```
int AddTwoNum(float a,float b)
{
   return a + b;
}
```

尽管从逻辑上看，这段代码也没有错误，同样可以实现两个数相加，但我们认为它是有错的代码。因为函数 AddTwoNum()的类型仍然规定为 int 类型，但函数体中的代码，却试图返回的却是 float 类型。为什么说返回的是 float 类型呢？因为请注意，现在 a,b 都是 float 类型了。

不仅这段代码有错，下面的代码也同样错误：

```
int AddTwoNum(int a, int b)
{
   float c = a + b;
   return c;
}
```

要注意，写类似上面的代码，编译器会放行，并不认为错误。那是因为编译器将一个 float 类型强制转换为 int 类型，这就会造成精度丢失。比如调用：AddTwoNum(1.2, 2.4),得到结果为 3，而不 3.6。

13.1.2 return 语句

return 语句只在函数内使用。它起到让函数停止运行，然后返回一个值的作用。我们通过一个特殊的对比，可以看到 return 的第一个作用：让函数停止运行.

代码一

```
void OutputSomething()
{
   cout << "第 1 行" << endl;
   cout << "第 2 行" << endl;
   cout << "第 3 行" << endl;
}

OutputSomething();
```

代码二

```
void OutputSomething()
{
   cout << "第 1 行" << endl;
   return;
   cout << "第 2 行" << endl;
   cout << "第 3 行" << endl;
}

OutputSomething();
```

输出结果：

```
第 1 行
第 2 行
第 3 行
```

```
第 1 行
```

为什么代码二只输出了一行？原因正是因为代码中标成红色的 return;当函数执行到 return;后，函数就结束了。后面的代码等于白写。这里只是为了突出 return 的作用才故意这样写。

一个函数没有 return;语句，也可以自然地结束，比如上面的代码一，当在屏幕上打印完第三行后，函数体内的代码也没了，所以函数自然就结束了，为什么还要 return 语句呢？

结合流程控制语句和 return 语句，我们可以控制一个函数在合适的位置返回，并可返回合适的值。

下面的函数实现返回二数中的较大者：

```
int max(int a, int b)
{
   if(a > b)
      return a;
   return b;
}
```

这个函数有两个 return;但并不是说它会返回两次。而是根据条件来执行不同的返回。执行以下面代码来调用上面的函数：

```
int c = max(10,7);
```

得到的结果将是 c 等于 1 0 。

这个例子也演示了 return 后面可以接一个表达式，然后将该表达式的值返回。

请大家想一想调用 max(10,7)时，max 函数中哪一行的 return 语句起作用了？想不出来也没关系，我们下一节将通过调试，看最终走的是哪一行。

关于 return 的最后几句话是：

1 、有些函数确实可以不需要 return，自然结束即可，如上面的 OutputSomething();

2 、有些人习惯为 return 的返回值加一对(),如： return (a); 这样写和 return a;完全一样。当然，在某些特殊的情况下，一对()是必要的。

3 、一个函数是 void 类型时，return 不能接返回，这时 return 仅起结束函数的作用。

4 、记得 return 接的是一个表达式，可以是一个立即数，一个变量，一个计算式，前面我们就看到 return a+b;的例子。 return 甚至也可以接一个函数。

13.1.3 跟踪函数

结合本小节的最后一个例子，我们来学习如何跟踪一个函数。同时我们也将直观地看到 return 的作用。

我们一直说 F7,F8 都是单步运行，下面的例子，二者的区别就表现出来了。

Ｆ 7 和Ｆ 8 都是单步跟踪，让代码一行行运行，但如果代码调用一个函数，那么，按 F8 将使调试器直接完成该函数的调用，而按下Ｆ 7 ，调试器将进入该函数的内部代码。

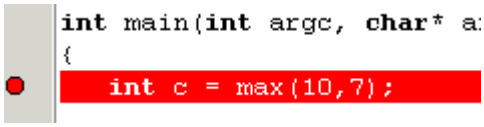
例一：调试函数

新建一个控制台的工程，然后加入以下黑体部分的代码。

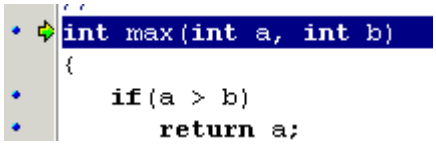
```
//------------------------------------------------------------------------
```

```
#include <iostream.h>
#pragma hdrstop
//---------------------------------------------------------------------------
int max(int a, int b)
{
   if(a > b)
      return a;
   return b;
}
//---------------------------------------------------------------------------
#pragma argsused
int main(int argc, char* argv[])
{
    int c = max(10,7);
    cout << c << endl;

    getchar();
    return 0;
}
//---------------------------------------------------------------------------
```

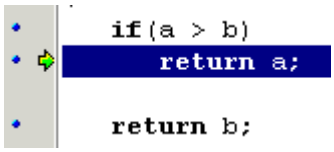
并在图中所示行加上断点：



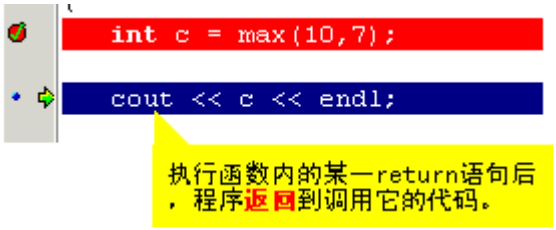
现在按F 9运行，程序在断点处停下，然后请看准了F7键，按下，发现在我们跟进了max()函数：



现在继续按F 7或F 8键，程序走到：if(a > b) 这一行，然后请将鼠标分别移到该行的a和b上，稍停片刻出现的浮动提示将显示a或b的当前值，相信你会明白很多，至少，你能知道程序下一步何去何从。看一看你想对了没有，再按一次F8或F7:



现在程序即将运行return语句！再按一次F7或F8,再想一想程序将何去何从？



现在，你若是再说不明白 return 在函数内的作用，就有点过份了吧？

下面大家照我说的继续做两次实验：

第一，重做这个实验，但在第一次需要按Ｆ７处，改为按Ｆ８，看看事情起了什么变化？

第二，将代码中的： int c = max(10,7); 改为： int c = max(7,10); 然后重做这个实验，看看这回走的是哪一行的 return 语句？

现在我们明白，在跟踪的过程，如果当前代码调用了一个函数，而你想进入这个函数跟踪，最直接的方法就是在调用函数的代码处按Ｆ７。

而关于Ｆ８与Ｆ７键所起的作用，名称为：Ｆ７单步进入（）；Ｆ８单步越过。它们分别对应于主菜单上 Run 下的 Trace Into 和 Step Over。如果当前行代码并没有调用函数，则二者的功能完全一样。以后我们讲到无区分的单步跟踪时，将只说按Ｆ８。

如果你需要频繁地跟踪某一函数，这种方法并不方便，你可以在设置断点，直接设置在那个函数的定义代码上。然后按Ｆ９，程序即可直接停在断点。比如，你可以将断点设置在这一行：int max(int a, int b)。

13.2 函数的参数

讲完函数的类型及返回值，我们来看函数另一重点及难点：参数。

不知上一章关于“修理工需要一台电视”的比喻，有没有在一定程度上帮助大家理解参数对于一个函数的作用。那时我们只是要大家从概念上明白：参数是调用者给出去的，被调用进接来过使用的数据，就像我们给修理工一台电视，供也修理一样。你只有明白了这点，才可以在下面继续学习参数的具体知识点；否则你应该复习上一章。

13.2.1 形参和实参

隔壁小李开了家服装店，在往服装上贴价钱标签时，小李突发奇想：他决定直接往衣服上贴上和价钱相应的钞票！比如这条裤子值１００元，小李就往贴一张百元大钞……

尽管我们无法否认小李是一个非常具有创意的人，但听说最近他一直在招保安，才１０来平方的小店里，挤满了一堆保安，不干别的事，就盯着那些贴在衣服上的钱。

这当然只是一个笑话……

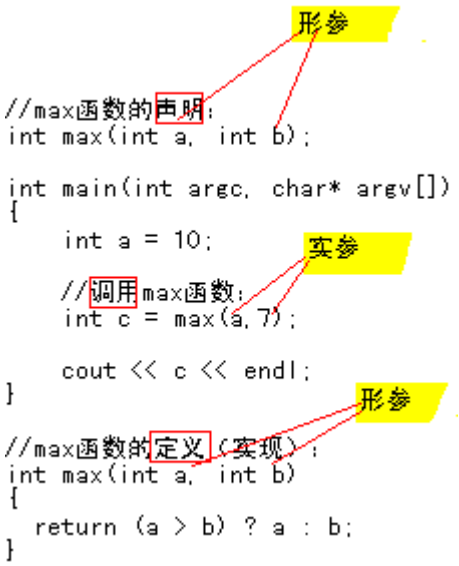
上一章我们说过，一个函数它需要什么类型的参数，需要多少参数，这都由函数本身决定。这就像一

件商品需要多少钱，是由商人来决定，商人认为某件衣服需要 100 元，他就往上面贴个写着"￥100"价钱的标签，而一张真正百元钞票。

函数也一样，它需要什么样的参数？这需要在声明和定义时写好。

所以，所谓的形参就是：函数在声明或定义时，所写出的参数定义。

比如有这么一段代码：



由于形参只是形式上参数，所以在声明一个函数时，你甚至可以不写形参的变量名：

如**声明**某个函数：

```
int max(int a, int b);
```

你完全可以这么写：

```
int max(int ,int );
```

现在再说**实参：实际调用时，传给函数的实际参数，是为实参。**请参看上图有关“调用”函数时的参数。

13.2.2 参数的传递方式

参数是调用函数的代码，传给函数的数据，在Ｃ，Ｃ＋＋中，参数有两种传递方式：传值方式和传址方式。这两个名词分别指：传递“参数的值”和传递“参数的地址”。

13.2.2.1 传值方式

如果你是第一次学习程序语言，下面代码的执行结果可能让你出乎意料。

例二：传值方式演示

```
//定义函数 F:
void F(int a)
{
   a = 10; //函数 F 只做一件事：让参数 a 等于 10
}

int main(int argc, char* argv[])
{
   //初始化 a 值为 0 ：
   int a = 0;

   //调用函数 F:
   F(a);

   //输出 a 的结果：
   cout << a << endl;
}
```

想一想，屏幕上打出的 a 的值是什么？　如果你猜“10”，呵呵，恭喜你猜**错**了。猜错了是好事，它可以加深你对本节内容的记忆。

如果你不服，立即打开 C B 将上面代码付诸实际，然后查看结果，那就更是好事！

正确答案是： a 打出来将是“ 0 ”。

代码不是明明将参数 a 传给函数 F 了吗？

而函数 F 不是明明让传来的 a 等于 10 了吗？

为什么打出的 a 却还是原来的值： 0 呢？

这就是 C， C＋＋传递参数的方法之一：值传递。它是程序中最常见的传递参数的方法。

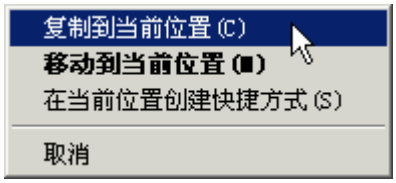
传值方式：向函数传递参数时，先复制一份参数，然后才将复制品传给参数。这样，函数中所有对参数的操作，就只是在使用复制品。不会对改变传递前的参数本身。

我曾经说过（不是在课程说的），你要学习编程，可以没有任何编程基础，但你至少会要对普通的电脑的操作很熟练。是该考一考你的电脑操作水平了。

以下是某用户在电脑上的操作过程，请仔细阅读，然后回答问题。

操作一：用户在C盘上找一个文本文件；

操作二：用户使用鼠标右键拖动该文件到D盘，松开后，出现右键菜单，用户选择“复制到当前位置”，如图：



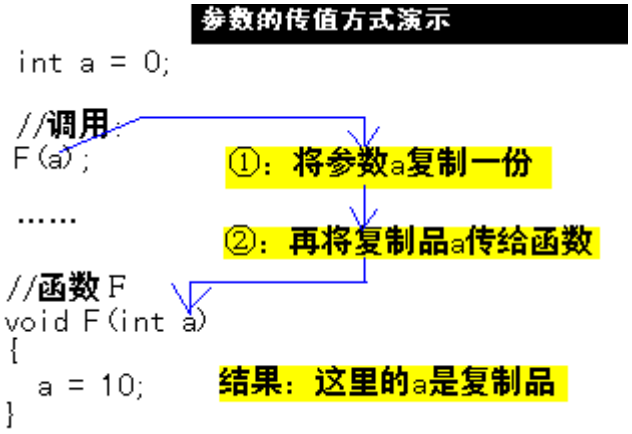
操作三：用户双击打开复制到D盘的该文件，进行编辑，最后存盘。

请问：C盘上的文件内容是否在该过程受到修改？

答案：不会，因为D盘文件仅是C盘文件的复制品，修改一个复制文件不会造成原文件也受到改动。

前面关于a值为什么不会变，道理和此相同： a被复制了一份。然后才传递给函数F();

请看参数传值方式的图解：



13.2.2.2 传址方式

即：传递参数地址

“地址”？是啊，我们第三次说到它。请大家先复习一下以前的内容：

第三章：3.4.1 内存地址
第五章：5.2 变量与内存地址

地址传递和值传递正好相反，这种方式可以将**参数本身**传给函数。从而，函数对参数的操作，将直接改变实参的值。

那么，如何才能指定让某个函数的某个参数采用“地址传送”的方式呢？方法很简单，只要在定义时函数时，在需要使用地址传送的参数名之前，加上符号：&。

如：

```
void F(int &a) //在形参 a 之前加一个 &
{
  a = 10;
}
```

笔者我更习惯于把 & 贴在参数的类型后面：

```
void F(int& a)  //把&贴在类型之后，也可以
{
  a = 10;
}
```

两种书写格式在作用上没有区别。

现在让我们用一模一样的代码调用函数 F：

例三：地址传递演示：

```
int main(int argc, char* argv[])
{
   //初始化 a 值为 0 ：
   int a = 0;

   //调用函数 F:
   F(a);

   //输出 a 的结果：
   cout << a << endl;
}
```

输出结果， a 的值真的被函数 F （）改为 1 0 了。

通过这个例子我们发现， C＋＋中，函数的参数采用“值”或“地址”传递，区别仅仅在于函数的定义中，参数是否加了＆符号。在调用函数时，代码没有任何区别。如此产生一个不便之处：我们仅仅通过看调用处的代码，将不好确定某一函数中有哪些参数使用地址传递？我们不得不回头找这个函数的定义或声明。C＋＋很多地方被反对它的人指责，这是其一。C＃语言改进了这一点，要求在调用时也明确指明调用方式。

比如，假设有一函数：

```
int foo(int a ,int &b, int c);
```

……

在代码某处调用该函数：

```
int i,j,k;
```

```
int r = foo(i,j,k);
```

如果你看不到前面函数的声明，那么你在读后面的代码时，可能比较难以想起其中的 j 是采用传址方式。

当然，我们没有必要因此就放弃C＋＋这门强大的语言。如果的确需要让阅读代码的人知道某个地方采用了地址传送，可以加上注释，也可以使用我们以后将学的指针作为参数来解决就是。

关于地址传送方式的作用，及如何实现地址传送，我们已明白。剩下来需要弄明白的是，“地址传送”是如何实现的？

首先，为什么叫“地址”传送？如果你完成了前面指出的复习任务。那么你应该明白了变量与地址的关系，这里我从根本上重述一次：

程序中，数据存放在内存里；
内存按照一定规则被编号，这些号就称为内存地址，简称地址；
内存地址很长，所以高级语言实现了用变量代表内存地址；
所以，一个变量就是一个内存地址。

因此，这里“地址传送”中的“地址”，指的就是变量的地址。那么参数（实参）是变量吗？

参数可以是变量，也可以不是变量。我们先来说是的情况。比如前面的例子：

```
……
int a = 0;
F(a);  //正确地调用函数F：参数a是一个变量
……
```

如果面上面例子中，我们直接传给F函数0，可以吗？

```
……
F(0); //错误地调用函数F： 0是一个常数，不是变量。
……
```

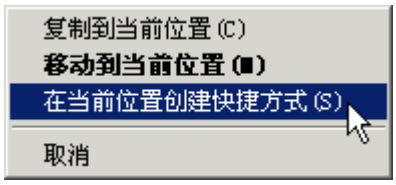
错误原因是：因为函数F()的参数采用“地址传递”方式，所以它需要得到参数的地址，而0是一个常数，无法得到它地址。
得出第一个结论：在**调用函数时，凡是采用“传址方式”的参数，不能将常数作为该参数。**

如果你在程序中违返了这一规定，不要紧，编译器不会放过这一错误。下面让我们来理解为什么传递变量地址可以起到让函数修改参数。这也好有一比，我们再来考一次“电脑操作知识”。

以下是某用户在电脑上的操作过程，请仔细阅读，然后回答问题。

操作一：用户在C盘上找一个文本文件；
操作二：用户使用鼠标右键拖动该文件到D盘，松开后，出现右键菜单，用户选择“在当前位置创建

方式”，如图：

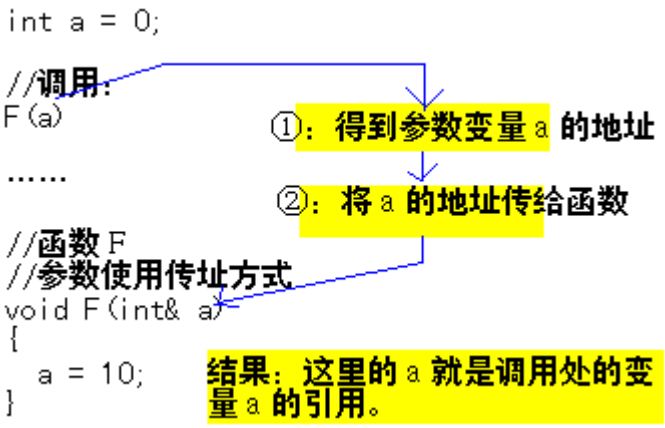


操作三：用户双击打开在D盘创建的该快捷方式，然后进行编辑，最后存盘。

请问：C盘上的文件内容是否在该过程受到修改？

答案：C盘的文件并改变了，因为D盘上的快捷方式，正是C盘上文件的一个“引用”，双击该快捷方式，正是打开了C盘的文件。

“地址传递”类似于此，将地址传送给函数，函数对该地址的内容操作，相当于对实参本身的操作。

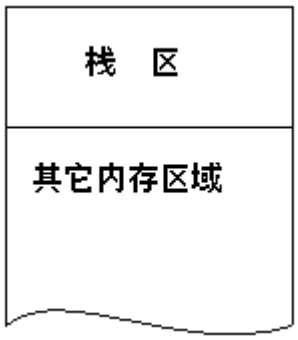


13.2.3 参数的传递过程（选修）

刚讲完“参数的传递方式”，又讲“参数的传递过程”，不禁让人有点发懵：方式和过程有何区别？中学时我对前桌的女生“有意思”，想给人家传递点信息，是往她家打个电话呢？还是来个“小纸条”？这就是“传递方式”的不同。我选择了后者。至于传递过程：刚开始时我把纸条裹在她的头发里，下课时假装关心地“喂，你的头发里掉了张纸……”。后来大家熟了，上课时我轻轻动一下她的后背，她就会不自在，然后在一个合适时机，自动把手别过来取走桌沿的纸条……这就是传递过程的不同吧？

（以上故事纯属虚构）

程序是在内存里运行的。所以无论参数以哪一种方式传递，都是在内存中“传来传去”。在一个程序运行时，程序会专门为参数开辟一个内存空间，称为“栈”。栈所在内存空间位于整个程序所占内存的顶部（为了直观，我们将地址较小的内存画在示意图顶部，如果依照内存地址由下而上递增规则，则栈区应该在底部），如图：



当程序需要传递参数时，将一个个参数“压入”栈区内存的底部，然后，函数再从栈区一个个读出参数。

如果一个函数需要返回值，那么调用者首先需要在栈区留出一个大小正好可以存储返回值的内存空间，然后再执行参数的入栈操作。

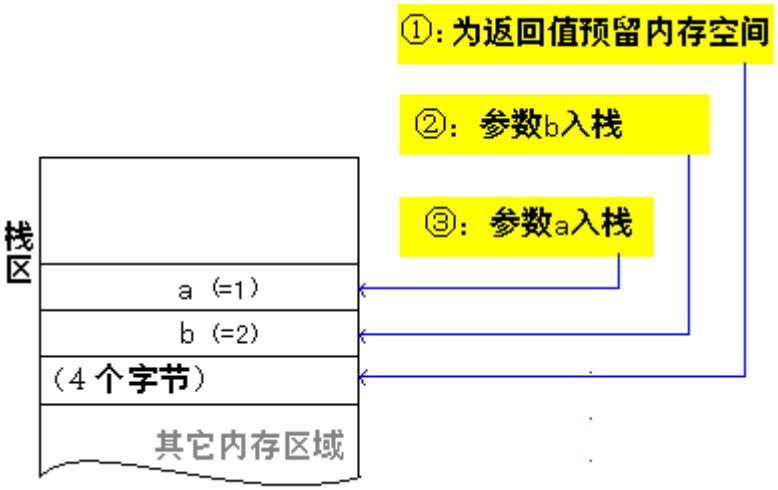
假设有一函数：

```
int AddTwoNum(int n1, int n2);
```

然后在代码某处调用：

```
....
int a = 1;
int b = 2;

int c = AddTwoNum(a,b);
```

当执行上面黑体部分，即调用函数的动作发生时，栈区出现下面的操作：



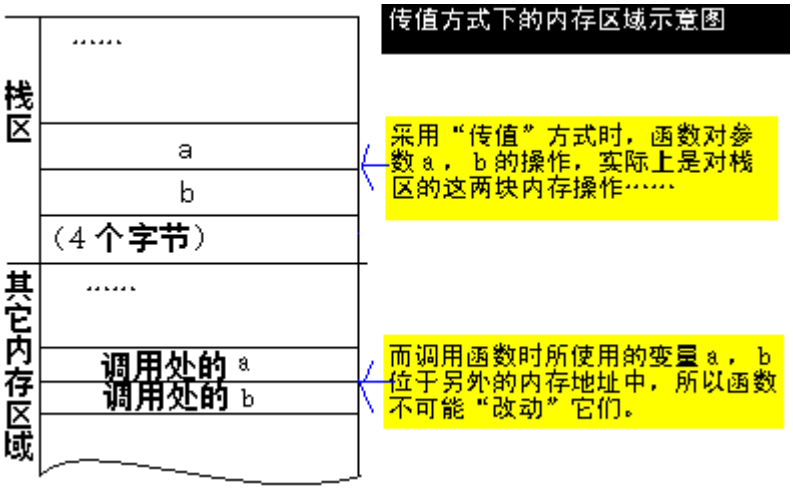
图中标明为返回值预留的空间大小是 4 个字节，当然不是每个函数都这个大小。它由函数返回值的数据类型决定，本函数 AddTwoNum 返回值是 int 类型，所以为 4 个字节。其它的 a ， b 参数也是 int 类型，所以同样各占 4 字节大小的内存空间。

至于参数是 a 还是 b 先入栈，这依编译器而定，大都数编译器采用“从右到左的次序”将参数一个个

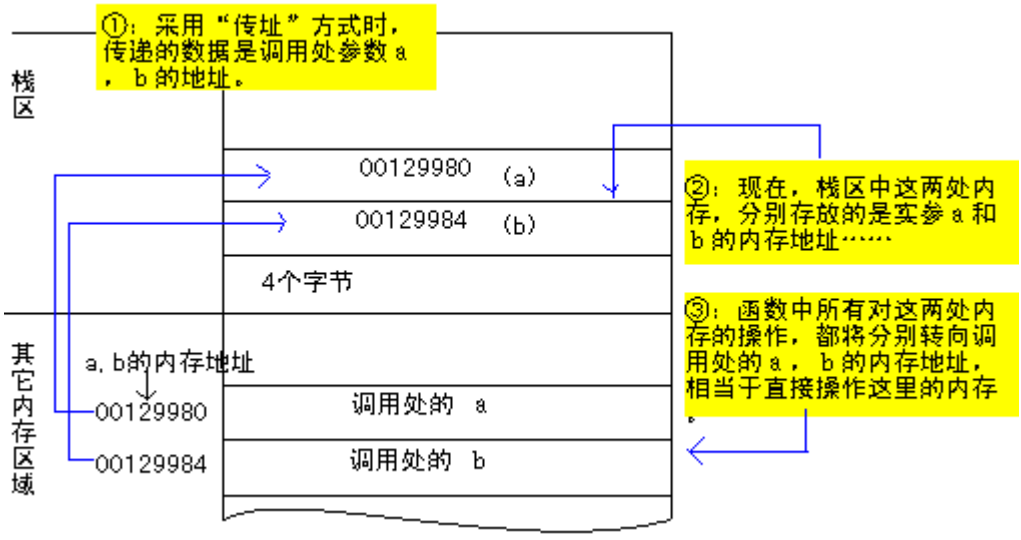
压入。所以本示意图，参数 b 被先“压”入在底部，然后才是 a 。

这样就完成了参数的入栈过程。根据前面讲的不同“传递方式”，被实际压入栈的数据也就不同。

一、如果是“传值”，则栈中的 a ， b 就是“复制品”，对二者的操作，仅仅是改变此处栈区的内存，和调用处的实参： a ， b 毫不关联：



二、而在“传址”方式时，编译器会将调用处的 a ， b 的内存地址写入栈区，并且将函数中所有对该栈区内存的操作，都转向调用处 a ， b 的内存地址。请看：



看起来二的图比一要复杂得多。其实实质的区别并不多。

你看：

实参 a ， 值为 1 ， 内存地址为：00129980

实参 b ， 值为 2 ， 内存地址为：00129984

在一图中，传给函数的是 a ， b 的值，即 1 ， 2 ；

在二图中，传给函数的是 a ， b 的地址，即：00129980，00129984。

这就是二者的本质区别。

“参数的传递过程”说到最后，还是和“参数的传递方式”纠缠在一起。我个人认为，在刚开始学习Ｃ＋＋时，并不需要－－或者甚至就是最好不要－－去太纠缠语言内部实现的机制，而重在于运用。下面我们就来举一个使用“传址”方式的例子。

题目是：写一函数，实现将两个整型变量的值互换。

比如有变量：int a = 1, b =2;我们要求将它作为所写函数的参数，执行函数后， a ， b 值互换。即 a 为 2 ， b 为 1 。

交换两个变量的值，这也是一个经典题目。并且在实际运用中，使用得非常广泛。事实上很多算法都需要用到它。

幸好实现它也非常的简单和直观。典型的方法是使用“第三者”你可能感到不解：交换两个变量的值，就让这两个变量自个互换就得了，比如小明有个苹果，小光有个梨子，两人你给我给你就好了啊，要小兵来做什么？

呵，你看吧：

```
int a = 1, b = 2;

//不要“第三者”的交换（失败）
a = b;
b = a;
```

好好看看，好好想想吧。当执行交换的第一句：a = b;时，看去工作得不错， a 的值确实由 1 变成了 2 。然后再下去呢？等轮到 b 想要得到 a 的值时，a 现在和 b 其实相等，都是 2 ，结果 b=a;后，b 的值还是 2.没变。

只好让“第三者”插足了……反正程序没有婚姻法。

```
int a = 1, b = 2;
int c ; //“第三者”

//交换开始：
c = a;
a = b;
b = c;
```

好了，代码你自己琢磨吧。下面把这些代码写入函数，我命名为 Swap;

例四：两数交换。

```
void Swap(int& a, int& b)
{
   int c = a;
   a = b;
   b = c;
}
```

```
int main(int argc, char* argv[])
{

   int a, b;
   cout << "请输入整数 a：" ;
   cin >> a;

   cout << "请输入整数 b":";
   cint >> b;

   cout << "交换之前，a = " << a << " b = " << b << endl;

   Swap(a,b);
   cout << "交换之后，a = " << a << " b = " << b << endl;

   getch(); //getchar 会自动“吃”到我们输入 b 以后的回车符，所以改为 getch(),记得前面有
#include <conio.h>

   return 0;
}
```

完整程序请见下载的课程实例。

13.2.4 参数默认值

在手头的某本C＋＋教材里，有关本节内容的第一句话是：“参数默认值也称默认参数值”。对着这话我愣了半天才算明白。所以在后面课程里，有些地方我说“参数默认值”有些地方我又会胡里胡涂说成“默认参数值”。你可不别像我一样去“研究”二者的区别呵。个人认为，从词法角度上看，“参数默认值”更准确些。

C++支持在定义或声明函数时，设置某些参数的默认值，这一点C不允许。

比如我们为卖萝卜的大娘的写一个计价函数。这个函数需要三个参数：顾客交多钱？买多少斤萝卜？及萝卜的单价。返回值则是大娘应该找多少钱。例如，顾客交了 100 元，他买 5 斤萝卜，单价是 1 .00 元/斤。那么函数就会计算并返回 95，表示应该找给顾客 95 元钱。

```
//函数定义如下：
float GiveChange(float money, float count, float price)
{
   return money - count * price; //找钱 = 已付款 - 数量 * 单价
}
```

当我们在程序中需要使用该函数时，我们大致是这样调用：

```
float change = GiveChange(100,5,1);
```

看上去一切很完美－－确实也很完美。不过Ｃ＋＋允许我们锦上添花。并且不是一朵只为了好看的“花”。

现实情况是，萝卜的价钱是一个比较稳定的数－－当然并不是不会变，在时出现亚洲经济风暴，萝卜价还是发变－－总之是会变，但很少变。

碰上这种情况，我们每回在调用函数时都写上最后一个参数，就有些亏了，这时，我们可以使用“参数的默认值”。

```
//首先，函数的定义做一点改动：
float GiveChage(float money, float count, float price = 1.0)
{
   .....
}
```

看到变化了吗？并不是指函数体内我打了省略号，而是在函数参数列表中，最后一个参数的定义变为：float price = 1.0。这就默认参数值，我们指定价格默认为 1 元。

然后如何使用呢？

以后在代码中，当需要计算找钱时，如果价钱没有变，我们就可以这样调用：

```
float change = GiveChange(100,5);  //没有传递最后一个参数。
```

是的，我没有写最后一参数：价钱是多少？但编译器发现这一点时，会自动为我填上默认的１.0。

如果在代码的个别地方，大娘想改一改价钱，比如某天笔者成了顾客，大娘决定按 1 斤 2 毛钱把萝卜卖给我：

我给大娘 5 毛钱，买 2 斤：

```
float changeForBCBSchool = GiveChange(0.5, 2 ,0.2); //你一样可以继续带参数
```

我想，这个实例很直观，但必须承认这个例子并没有体现出参数默认值的种种优点。不过不管如何，你现在应该对参数的默认值有感性认识。

下面学习有关参数默认值的具体规定。

１、必须从**最右边**开始，然后**连续**地设置默认值。

如果理解这句话？

首先我们看关键词“最右边”。也就是说假如一个函数有多个参数，那么你必须从最后一个参数开始设置默认值。

如：

```
void foo(int a, int b, bool c);
```

那么，下面的设置是正确的：

```
void foo(int a, int b, bool c = false); //ok,c 是最后一个参数
//而，下面是错误的：
void foo(int a, int b = 0, bool c);    //fail,b 不是最后一参数
```

然后我们看“连续”。也就是说，从最右边开始，你可以连续地向左设置多个参数的默认值，而不能跳过其中几个:

如：

下面的的设置是正确的：

```
void foo(int a, int b=0, bool c = false); //ok ,连续地设置 c ， b 的默认值
```

同样，这也是正确的：

```
void foo(int a=100, int b=0, bool c = false); //ok ,连续地设置 c ， b , a 的默认值
//而，这样设置是错误的：
void foo(int a=100, int b, bool c = false); //fale,不行，你不能跳过中间的 b 。
```

2、如果在函数的声明里设置了参数默认值，那么就不参在函数的定义中再次设置默认值。

函数的“声明”和“定义”你可能又有些胡涂了。好，就趁此再复习一次。

所谓的“定义”，也称为“实现”，它是函数完整的代码，如：

```
//函数定义如下(函数定义也称函数的实现)：
float GiveChange(float money, float count, float price)
{
   return money - count * price; //找钱 = 已付款 - 数量 * 单价
}
```

而函数的“声明”，则是我们上一章不断在说的函数的“名片”，它用于列出函数的格式，函数的声明包含函数的“返回类型，函数名，参数列表”，惟一和函数定义不一样的，就是它没有实现部分，而是直接以一分号结束,如：

```
//声明一个函数：
float GiveChange(float money, float count, float price); //<---注意，直接以分号结束。
```

现在和参数默认值有关的是，如果你在函数声明里设置了默认值，那就不用，也不能在函数定义处再设置一次。

如，下面代码正确：

———————————————————————————————————————

```
//定义:
float GiveChange(float money, float count, float price)
{
   return money - count * price; //找钱 = 已付款 - 数量 * 单价
}
```

```
//声明：
float GiveChange(float money, float count, float price = 1.0);
————————————————————————————————————
```

而下面的代码有误：

```
//定义:
float GiveChange(float money, float count, float price = 1.0)
{
   return money - count * price; //找钱 = 已付款 - 数量 * 单价
}

//声明：
float GiveChange(float money, float count, float price = 1.0);
————————————————————————————————————
```

3、默认值可以最常见的常数，或全局变量，全局常量，甚至可以是一个函数的调用。

关于题中的“全局”，我们还没有学习，这时理解就是在程序运行区别稳定存在的变量或常量。下面举一个让我们比较狐疑的，使用函数作来参数默认的例子：

```
//某个返回 double 的函数：
double  func1();
double  func2(double a, double b = func1()); //func1()的执行结果将被用做 b 的默认值。
```

13.3 函数重载

重，重复也。载者，承载也。

“重复”一词不用解释，“承载”不妨说白一点，认为就是“承负”。

函数的“重载”，意为可以对多个功能类似的函数使用相同的函数名。

13.3.1 重载的目的

有这个需要吗？不同的函数取相同的名字？这不会造成混乱？在现实生活中，我们可一点也不喜欢身边有哪两个人同名。

当然有这个必要。“函数名重载”是C＋＋对C的一种改进（因此C也不支持重载）。

想一想那个求“二数较大者”的 max 函数吧。如果不支持函数名重载，那么就会有以下不便：

```
int max(int a, int b);
```

这是前面我们写的，用以实现两数中较大者的函数。比如你传给它 20，21,那么，它将很好地工作，返回 21。现在，我们想求 20.5,和 21.7 两个实数中较大者？对不起，max 函数要求参数必须为 int 类型，所以传给它 20.5,21.7:

```
float larger = max(20.5,21.7);
```

编译器不会让这行代码通过。它会报错说“参数不匹配”。

好吧，我们只好为实数类型的比较也写一个参数，但C语言不允函数重名，所以我们只好另起一个名字：

```
float maxf(float a, float b);
```

你可能会就，那就不要 int 版的 max,只要这个 float 版本的：

```
float max(float a, float b);
```

因为，实数版本的完全可以处理整数。说得没错，但这不是一个好办法，其一我们已知道实数和整数相比，它有计算慢，占用空间大的毛病；其二，float 版本的 max 函数，其返回值必然也是 float 类型，如果你用它来比较两个整数：

```
int larger = max(1, 2);
```

编译器将不断警告你，“你把一个 float 类型的值赋值一个 int 类型的变量”。编译器这是好心，它担心你丢失精度，但这会让我们很烦，我们不得不用强制类型转换来屏蔽这条警告消息：

```
int larger = (int) max(1,2);
```

这样的代码的确不是好代码。

好吧，就算你能容忍这一切，下一问题是，我想写了一个求 3 个整数谁最大的函数。这回你没有理由因为要写三个参数的版本，就把两个参数的版本扔了。只好还是换名：

```
int max_3(int a, int b, int c);
```

看着 max_3 这个函数名字，我不禁想起前几天在 yahoo 申请免费电子信箱，我想叫 nanyu@yahoo.com.cn ,它却坚持建议我改为：nanyu1794@yahoo.com.cn （１７９４？一去就死？），折腾了我两个半小时，我才找到一个可以不带一串数字，又让我能接受点的呢称。

结论是：不允许重名的世界真的有些烦。C＋＋看到了这一点，所以，它允许函数在某些条件的限制下重名。这就是函数重载。

前面有关 max()的问题，现在可以这样解决：

```
//整数版的 max()
int max(int a, int b);

//单精度实数版的 max()
float max(float a, float b);

//双精度实数版的 max();
double max(double a, double b);

//甚至，如果你真的有这个需要，你还可以来一个这种版本的 max();
double max(int a, double b);

//接下来是三个参数的版本：
int max(int a, int b, int c);
double max(double a, double b, double c);
```

上面林林总总的求最大值函数，名字都叫 max();好处显而易见：对于实现同一类功能的函数，只记一个名字，总比要记一堆名字要来得舒服。

13.3.2 函数重载的规则

有一个问题，那么多 max 函数，当我们要调用其中某一个时，编译器能知道我们到底在调用哪一个吗？如何让编译器区分出我们代码中所调用的函数是哪一个ｍａｘ，这需要有两个规则。

实现函数重载的规则一：同名函数的参数必须不同，不同之处可以是**参数的类型**或**参数的个数**。

如果你写想两个同名函数：

错误一：

```
int max(int a, int b);
int max(int c, int d);
```

看上去这两个函数有些不同，但别忘了，形参只是形式，事实上两个声明都可以写成：

```
void max(int, int);
```

所以记住：仅仅参数名不一样，不能重载函数。

错误二：

```
float max(int a, int b);
int max(int a, int b);
```

两个函数不同之处在返回类型，对不起，Ｃ＋＋没有实现通过返回值类型的不同而区分同名函数的功能。

所以记住：仅仅返回值不一样，不能重载函数。

正因为函数的重载机制和函数的参数息息相关，所以我们才把它紧放在“函数参数”后面。但函数重载并不能因此就归属于“参数”的变化之一，以后我们会学习不依赖于参数的重载机制。

实现函数重载的规则二：参数类型的匹配程度，决定使用哪一个同名函数的次序。

若有这三个重载函数：

```
１）int max(int a, int b);
２）float max(float a, int b);
３）double max(double a, double b);
```

现在我这样调用：

```
int larger = max(1, 2);
```

被调用的将是第１）个函数。因为参数１，２是 int 类型。

而：

```
double larger = max(1.0, 2);
```

被调用的将是第……注意了！是第３）个函数。为什么？

首先它不能是第１）个，因为虽然参数 2 是 int 类型，但 1.0 却不是 int 类型，如果匹配第１）函数，

编译器认为会有丢失精度之危险。

然后，你可能忘了，一个带小数的常数，例如 1.0,在编译器里，默认为比较保险的 double 类型（编译器总是害怕丢失精度）。

最后，关于这两个规则，都是在同名的函数参数个数也相同的情况下需要考虑，如果参数个数不一样：

```
int max(int a, int b);
int max(int a, int b ,int c);
```

当然就没有什么好限制了，编译器不会傻到连两个和三个都区分不出，除非……

实现函数重载的附加规则：有时候你必须附加考虑参数的默认值对函数重载的影响。

比如：

```
int max(int a, int b);
int max(int a, int b ,int c  = 0);
```

此例中，函数重载将失败，因为你在第二个 max 函数中设置了一个有默认值的参数，这将造成编译器对下面的代码到底调用了哪一个 max 感到迷惑。不要骂编译器笨，你自己说吧，该调用哪个？

```
int c = max(1, 2);
```

没法断定。所以你应该理解、接受、牢记这条附加规则。

事实上影响函数重载的还有其它规则，但我们学习这些就够了。

13.3.3 参数默认值与函数重载的实例

例五：参数默认值、函数重载的实例

有关默认值和函数重载的例子，前面都已讲得很多。这里的实例仅为了方便大家学习。请用ＣＢ打开下载的配套例子工程。所用的就是上面提到例子，希望大家自己动手分别写一个默认值和重载的例子。

13.4 inline 函数

从某种角度上讲，inline 对程序影响几乎可以当成是一种编译选项（事实上它也可以由编译选项实现）。

13.4.1 什么叫 inline 函数？

inline（小心，不是 online），翻译成“内联”或“内嵌”。意指：当编译器发现某段代码在调用一个内联函数时，它不是去调用该函数，而是将该函数的代码，整段插入到当前位置。

这样做的好处是省去了调用的过程，加快程序运行速度。（函数的调用过程，由于有前面所说的参数入栈等操作，所以总要多占用一些时间）。

这样做的不好处：由于每当代码调用到内联函数，就需要在调用处直接插入一段该函数的代码，所以程序的体积将增大。

拿生活现象比喻，就像电视坏了，通过电话找修理工来，你会嫌慢，于是干脆在家里养了一个修理工。这样当然是快了，不过，修理工住在你家可就要占地儿了。

(某勤奋好学之大款看到这段教程，沉思片刻，转头对床上的“二奶”说：

“终于明白你和街上‘鸡’的区别了”。

“什么区别？”

“你是内联型。”)

内联函数并不是必须的，它只是为了提高速度而进行的一种修饰。要修饰一个函数为内联型，使用如下格式：

inline 函数的声明或定义

简单一句话，在函数声明或定义前加一个 inline 修饰符。

```
inline int max(int a, int b)
{
  return (a>b)? a : b;
}
```

13.4.2 inline 函数的规则

规则一、一个函数可以自己调用自己，称为递归调用（后面讲到），含有递归调用的函数不能设置为 inline；

规则二、使用了复杂流程控制语句：循环语句和 switch 语句，无法设置为 inline；

规则三、由于 inline 增加体积的特性，所以建议 inline 函数内的代码应很短小。最好不超过 5 行。

规则四、inline 仅做为一种“请求”，特定的情况下，编译器将不理会 inline 关键字，而强制让函数成为普通函数。出现这种情况，编译器会给出警告消息。

规则五、在你调用一个内联函数之前，这个函数一定要在之前有声明或已定义为 inline,如果在前面声明为普通函数，而在调用代码后面才定义为一个 inline 函数，程序可以通过编译，但该函数没有实现 inline。

比如下面代码片段：

```
//函数一开始没有被声明为 inline:
void foo();

//然后就有代码调用它：
foo();

//在调用后才有定义函数为 inline:
inline void foo()
{
  ......
}
```

代码是的 foo()函数最终没有实现 inline;

规则六、为了调试方便，在程序处于调试阶段时，所有内联函数都不被实现。

最后是笔者的一点“建议”：如果你真的发觉你的程序跑得很慢了，99.9%的原因在于你不合理甚至是错误的设计，而和你用不用 inline 无关。所以，其实，inline 根本不是本章的重点。

所以，有关 inline 还会带来的一些其它困扰，我决定先不说了。

13.5 函数的递归调用（选修）

第 4 次从洗手间里走出来。在一周前拟写有关函数的章节时，我就将递归调用的内容放到了最后。

函数递归调用很重要，但它确实不适于初学者在刚刚接触函数的时候学习。

13.5.1 递归和递归的危险

递归调用是解决某类特殊问题的好方法。但在现实生活中很难找到类似的比照。有一个广为流传的故事，倒是可以看出点“递归”的样子。

“从前有座山，山里有座庙，庙里有个老和尚，老和尚对小和尚说故事：从前有座山……”。

在讲述故事的过程中，又嵌套讲述了故事本身。这是上面那个故事的好玩之处。

一个函数可以直接或间接地调用自已，这就叫做“递归调用”。

Ｃ，Ｃ＋＋语言不允许在函数的内部定义一个子函数，即它无法从函数的结构上实现嵌套，而递归调用的实际上是一种嵌套调用的过程，所以Ｃ，Ｃ＋＋并不是实现递归调用的最好语言。但只要我们合理运用，Ｃ，Ｃ＋＋还是很容易实现递归调用这一语言特性。

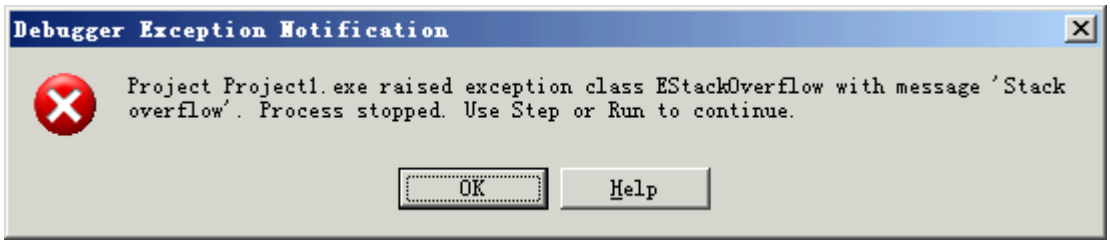
先看一个最直接的递归调用：

有一函数 F();

```
void F()
{
  F();
}
```

这个函数和“老和尚讲故事”是否很象？在函数 F()内，又调用了函数 F()。

这样会造成什么结果？当然也和那个故事一样，没完没了。所以上面的代码是一段“必死”的程序。不信你把电脑上该存盘的存盘了，然后建个控制台工程，填入那段代码，在主函数 main()里调用 F()。看看结果会怎样？WinNT,2k,XP 可能好点，98,ME 就不好说了……反正我不负责。出于“燃烧自己，照亮别人”的理念，我在自己的ＸＰ＋ＣＢ６上试了一把，下面是先后出现的两个报错框：



这是C B 6 的调试器“侦察”到有重大错误将要发生，提前出来的一个警告。我点O K，然后无厌无悔地再按下一次 F9，程序出现真正的报错框：

这是程序抛出的一个异常，EStackOverflow 这么看：E 字母表示这是一个错误(Error)，Stack 正是我们前面讲函数调用过程的“栈”，Overflow 意为“溢出”。整个 StasckOverflow 意思就：栈溢出啦！

“栈溢出”是什么意思你不懂？拿个杯子往里倒水，一直倒，直到杯子满了还倒，水就会从杯子里溢出了。栈是用来往里“压入”函数的参数或返回值的，当你无限次地，一层嵌套一层地调用函数时，栈内存空间就会不够用，于是发生“栈溢出”。

（必须解释一下,本例中，void F()函数既没有返回值也没有参数，为什么还会发生栈溢出？事实上，调用函数时，需要压入栈中的，不仅仅是二者，还有某些寄存器的值，在术语称为“现场保护”。正因为C，C＋＋使用了在调用时将一些关键数值“压入”栈，以后再“弹出”栈来实现函数调用，所以C，C＋＋语言能够实现递归。）

这就是我们学习递归函数时，第一个要学会的知识：

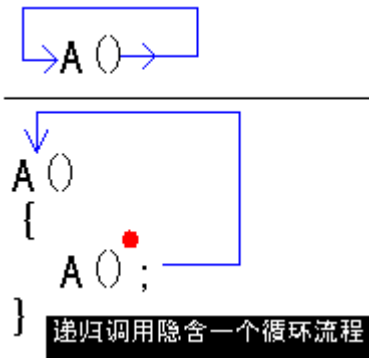
逻辑上无法自动停止的递归调用，将引起程序死循环，并且，很快造成栈溢出。

怎样才能让程序在逻辑上实现递归的自动停止呢？这除了要使用到我们前面辛辛苦苦学习的流程控制语句以后，还要掌握递归调用所引起的流程变化。

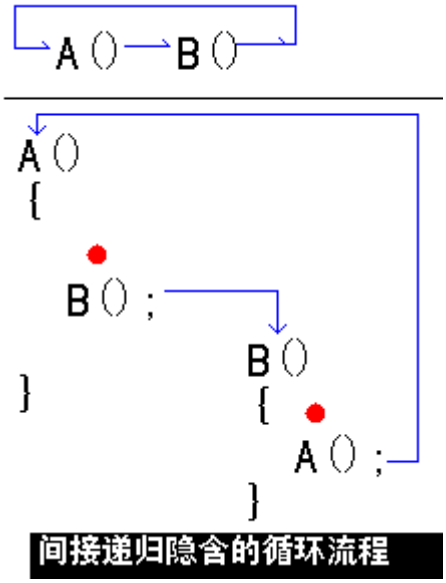
13.5.2 递归调用背后隐藏的循环流程

递归引起什么流程变化？前面的黑体字已经给出答案：“循环”。自己调用自己，当然就是一个循环，并且如果不辅于我们前面所学的 if...语句来控制什么时候可以继续调用自身，什么时候必须结束，那么这个循环就一定是一个死循环。

如图：



递归调用还可间接形成：比如 A() 调用 B()；B() 又调用 A()；虽然复杂点，但实质上仍是一个循环流程：



在这个循环之里，函数之间的调用都是系统实现，因此要想“打断”这个循环，我们只有一处“要害”可以下手：在调用会引起递归的函数之前，做一个条件分支判断，如果条件不成立，则不调用该函数。图中以红点表示。

现在你明白了吗？一个合理的递归函数，一定是一个逻辑上类似于这样的函数定义：

```
void F()
{
  ……
  if(……)  //先判断某个条件是否成立
  {
     F();  //然后才调用自身
  }
  ……
```

```
}
```

在武侠小说里，知道了敌人的“要害”，就几乎掌握了必胜的机会；然而，“递归调用”并不是我们的敌人。我们不是要“除掉”它，相反我们利用它。所以尽管我们知道了它的要害，事情还要解决。更重要的是要知道：什么时候该打断它的循环？什么时候让它继续循环？

这当然和具体要解决问题有关。所以这一项能力有赖于大家以后自己在解决问题不断成长。就像我们前面的讲的流程控制，就那么几章，但大家今后却要拿它们在程序里解决无数的问题。

（有些同学开始合上课本准备下课）程序的各种流程最终目的是要合适地处理数据，而中间数据的变化又将影响流程的走向。在函数的递归调用过程中，最最重要的数据变化，就是参数。因此，大多数递归函数，最终依靠参数的变化来决定是否继续。（另外一个依靠是改变函数外的变量）。

所以我们必要彻底明了参数在递归调用的过程中如何变化。

13.5.3 参数在递归调用过程中的变化

我们将通过一个模拟过程来观察参数的变化。

这里是一个递归函数：

```
void F(int a)
{
  F(a+1);
}
```

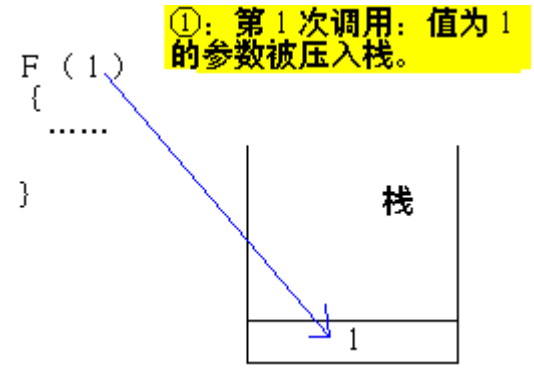
和前面例子有些重要区别，函数 F()带了一个参数，并且，在函数体内调用自身时，我们传给它当前参数加 1 的值，作为新的参数。

红色部分的话你不能简单看过，要看懂。

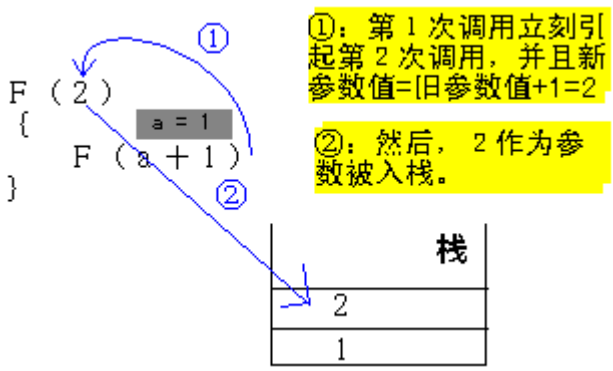
现在，假设我们在代码中以 1 为初始参数，第一次调用 F ()：

```
F(1);
```

现在，参数是 1 ，依照我们前面“参数传递过程”的知识，我们知道 1 被“压入”栈，如图：



F()被第 1 次调用后，马上它就调用了自身，但这时的参数是 a+1,a 就是原参数值，为 1 ，所以新参数值应为 2 。随着 F 函数的第二次调用，新参数值也被入栈：



再往下模拟过程一致。第三次调用 F （）时，参数变成 3 ，依然被压入栈，然后是第四次……递归背后的循环在一次次地继续，而参数 a 则在一遍遍的循环中不断变化。

由于本函数仍然没有做结束递归调用的判断，所以最后的最后：栈溢出。

要对这个函数加入结束递归调用的逻辑判断是非常容易的。假设我们要求参数变到 1 0 （不含 1 0 ）时，就结束，那么代码如：

```
void F(int a)
{
   if( a < 10)
     F(a+1);
}
```

终于有了一个安全的递归调用例子了。不过它似乎什么也没有做，我们加一句输出代码，然后让它做我们有关递归的第一个实例吧。

13.5.4 一个安全的递归调用函数实例

例六：用递归实现连续输出整数 1 到 9 。

```
//递归调用的函数：
void F(int a)
{
  if( a < 10)
  {
    cout << a;
    F(a+1);
  }
}

//然后这样调用：

F(1);
```

完整的代码请见下载的相应例子。输出将是：

123456789

请大家自行模拟本题函数的调用过程。

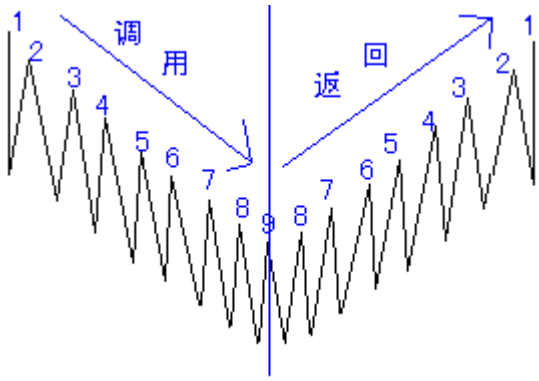
13.5.5 递归函数的返回

这里并不是要讲递归函数的返回值。

天气还不是很冷，你能把身上的衣服脱光一下吗？当初你穿衣服时，一定是先穿上最里层的衣服，然后穿上第二层的，再穿上第三层。现在让你脱衣服，你就得先脱外层，再脱稍里一层，然后才是最内层。

函数的递归调用，和穿衣脱衣类似，不过内外相反而已。开始调用时，它是外层调内层，内层调更内一层。等到最内层由于条件不允许，必须结束了，这下可好，最内层结束了，它就会回到稍外一层，稍外一层再结束时，退到再稍外一层，层层退出，直到最外层结束。

如果用调用折线图来表示前例，则为：



本小节不是讲递归函数的返回值，而是讲递归函数的返回次序。前面听说要脱衣服而跑掉或跑来的同学，可以各回原位了。

做为本小节的一个例子，我只给实例的代码，请大家考虑会是什么输出结果。考虑并不单指托着腮做思考状（你以为你是大卫？）。另外，我相信有很多同学有小聪明，他们凭感觉就可以猜出结果。聪明很好，但千万别因为聪明而在不知不觉中失去动手调试程序的动力。

代码其实只是在上例中再加上一行。

例七：递归函数的返回次序：

```
//递归调用的函数：
void F(int a)
{
  if( a < 10)
```

```
  {
    cout << a;
    F(a+1);
    cout << a;
  }
}

//然后这样调用：
F(1);
```

完整代码见下载的例子。

13.6 小结

我们讲了函数是如何通过 return 返回一个值，这个的值的类型就是函数类型。
讲完 return 之后，我们还“深入函数”内部进行跟踪，其实就是一个按Ｆ７还是Ｆ８的问题。

关于参数，我们讲了什么叫形参，函数在声明或定义处的参数，称为形参，它实际上位于栈内的某个内存地址。至于实参，就是调用时函数时所用的参数，它不在栈内存区里。
一句话：一个参数在未被“压入”栈内，就称为实参，并“压入”栈内了，就成了形参。
呵，如果你刚才没有选修“参数的传递过程”这一节，可能现在有些后悔了。

Ｃ＋＋支持的设置参数默认值比较好玩，大家在实际编程中可能会常用。你知道“从右到左，连续”这两个词和默认值的关系吧。

函数的重载就更有意思了，不过记住了，只对“功能类似”的函数进行重载吧，把一些功能互异的函数全叫成同名，那么效果就适得其反。另外，由于重载规则的限制，有时候与其在为排列同名函数的参数表头痛，不如还是恢复老办法，直接另取个函数名就是。学了一项技术，千万不要有非要用上的想法。是改变名字还是改变参数，怎样方便怎样来。

inline 函数？估计很多同学什么也没记住，只记了关于“大款”的讲话，哎，失败的教育。

最后是函数的递归调用，没什么好说的。我认为学习编程很重要的一点是学会看别人的代码（当然，那个人应该水平比你高）。大家自己尝试一下写几个安全的递归调用，然后有机会看别人如何用递归解决实际问题。至于本章的课程，希望你常回头看。

后面的课程，将不断地用到函数了。大家多动手写一些有关这两章的练习程序。

第十四章　程序的文件结构

程序是由什么组成的？学习到今天,我们至少有两个答案:

第 1,程序由代码语句组成。正是一行行的代码，组成了一个完整的程序。

第 2，程序由函数组成。一个个函数之间的互相调用，最终构建出一个完整的程序。

今天我们又有一个新的回答：“程序由文件组成”。

程序为什么需要使用多个文件？

一个小的程序，可以只写一个源文件,但程序稍微一大,就需要将其中不同的逻辑实现放到不同的源文件.对于需要多人一起开发的软件,自然更需要多个源文件。

14.1 源文件和头文件

和别的一些语言不同，C，C++的代码文件有“头文件”和“代码文件”之分。二者合起来我们称为单元(Unit)文件。

扩展名为 .c 或 .cpp 的文件，主要用以实现程序的各种功能，我们称为代码文件。

扩展名为 .h 的文件，称为头文件。在头文件里主要写一些函数、数据（包括数据类型的定义）、等的声明，这样可以在多个.c 或.cpp 文件内**共享**这些函数、数据。第 12 章我们提过到头文件的功能。说它可以起到函数“名片夹”的作用。

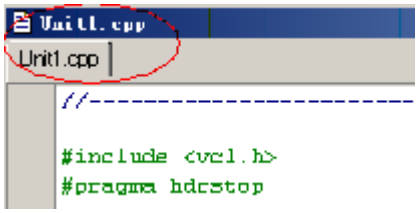
大都数时候，源文件和头文件是对应出现的，比如有一个 A.cpp 的源文件，就会有一个 A.h 的头文件。这种情况在我们写应用程序时，更是常见。所以 C++ Builder 对此进行了强化。比如，它支持在同名源文件和头文件之间通过热键来回切换。在 CB6.0 里，编辑器打开对应的源文件和头文件时，将显示为同一页下的两个子页。

我们来实际动手看看源文件与头文件在 CB 里的对应关系。

运行 C++ Builder 6 或 5。

这一次我们需要一个空白的 Windows 工程。很有可能，在你打开 CB 时，它就自动为你打开了一个工程。为了不出错，我们还是亲自建一个。CB6 请使用主菜单：File | New | Application；而 CB5 则使用：File | New Application **新建一个 Windows 空白工程**。 如果在这过程中 CB 出现是否存盘的询问，请回答不存盘。

找到“代码窗口”。如果你看到的是一个叫"Form1"的表单，请按 F12,“代码窗口”将跑到前面。它的标题应该是默认的"Unit1.cpp"。正是当前代码文件的文件名。如下图：



对于 CB6，还可以看到在该窗口的底部有这样一个分页：

源文件：Unit1.cpp 和头文件：Unit1.h 并列着，我们可以方便地选择。至于 "Diagram"，称为“图解”。这是一个给这个源文件加配套注解，及表单上各控件的依赖关系的地方。如果是一个开发小组在进行共同开发，严格地要求每个成员为每个单元文件写上“Diagram”，可以更好地实现程序员之间的沟通。

CB5 没有这些，不过下面的热键操作两个版本均一样的，要求大家记住。

按 Ctrl + F6 可以在源文件和头文件之间来回切换。请大家试试。这个简单的操作将在我们今后的编程过程中高频率地使用。

14.2 如何创建多个单元文件

前面我们在“Windows 应用程序工程”中看到了头文件与源文件的匹配关系，在“控制台”的工程中，也同样存在。不过由于控制台经常只用来写一些小小的程序，所以往往只需一个源文件即可。由于只有一个源文件，所以也就不存在函数、数据在多个文件之间“共享”的需要，因此边头文件也就可以不提供。

那么，是不是只有在程序很大，或者只有在有很多人同时开发一个软件时，才需要多个源文件呢？

这就好像你家里只有两本书：《红楼梦》和《格林童话》，是把它们放在同一个抽屉里呢？还是分开放到两个抽屉里？我觉得后者是比较好的选择。因为我们常常希望家里看《格林童话》的人，最好不要去看《红楼梦》。

程序也一样，最好把不同的逻辑实现，放到不同的源文件中。

下面我们做一个实例。例子的代码我们都已经学过。目标是实现一个可以求统计值和平均值的程序。

根据我们现在所学的情况，我把这个工程中的代码分为三个源代码：

其一：主程序。就是 main()函数所在的代码。这个源文件实现总的流程。我将该文件取为 main.cpp。

其二：计算总和及计算平均值的代码。这个源文件负责用户计算过程，也包括每个过程所需输入输出。该文件将被存盘为 mainfunc.cpp。 意为主要功能。

其三： assifunc.cpp。表示辅助功能函数所在代码。它只提供一个函数：将用户输入的大写或小写的字母'Y'或'N' 确保转换为大写。这个函数将 main()主函数内,判断用户是否继续时用到。

新 CB 新建一个控制台程序（如果你还开着上个程序，先选 File | Close All 关闭它）。CB 会自动生成第一个文件，不过现在的名字为“Unit1.cpp”。

接下来是一项新工作，我们来添加两人新的单元文件，即上面说的“其二”和“其三”。

CB6 ：File | New | Unit；CB5：File | New Unit。

请进行两次以上操作，CB 将为我们生成新的两个单元文件：Unit2.cpp 和 Unit3.cpp。大家可以再试试 Ctrl＋F6。(注意，第一个单元文件：Unit1.cpp 没有配套的.h 文件，所以不要在该文件里尝试 Ctrl＋F6)。

然后选择 File | Save All。全部存盘，最好不要存在 CB 默认的目录下。记得按以下关系重命名：

Unit1.cpp 存盘为 main.cpp；

Unit2.cpp 存盘为 mainfunc.cpp；

Unit3.cpp 存盘为 assifunc.cpp；

至于总的工程，随你便，我图方便，还是叫:Project1.bpr。

(现在我们第一次在一个工程中用到多个源文件。所以你得学会如何快速打开一个工程中某个源文件——当然，现在这三个文件都已经打开着，不过假设你有点事关闭 CB，我们不希望下回打开这个工程时，你“找”不到第 2 和第 3 个文件了——请点击 CB 工具栏上的这个图标：  ，将出现源文件列表对话框，如左图)

接下来讲在这三个文件中，我们分别写些什么？大多数代码我们都已经在前面学过，所以我对代码的功能不作太多的解释。我们的重点是：三个源文件之间的代码如何实现“沟通”。

第一个文件：main.cpp 用来实现程序的主流程。

在 main.cpp 中的 main()函数内，我们加入代码。

```
#include <iostream.h>
... ...
int main(int argc, char* argv[])
{
   char ch;
   int count; //求总和或平均值时，需要输入的成绩个数

   do
   {
      cout << "1)求总和" << endl;
      cout << "2)求平均" << endl;

      cout << "请选择(1 或 2)";
```

```
        cin >> ch;

        //输入有误，重输：
        if(ch != '1' && ch != '2')
        {
           cout << "输入有误，请重新输入!" << endl;
           continue;
        }

        cout << "请输入个数:";
        cin >> count;

        //根据用户的选择，调用不同函数：
        switch(ch)
        {
           case '1' :
              CalcTotal(count); //需要调用的函数之一
              break;
           case '2' :
              CalcAverage(count); //需要调用的函数之一
              break;
        }

        //是否继续：
        cout << "是否继续? (y/n)";
        cin >> ch;

        //确保转换为大写：
        ch = ToUpper(ch); //需要调用的函数之一
     }
     while(ch == 'Y');

     return 0;
  }
```

代码中，红色部分的注释表明，主函数 main()需要调用到三个自定义函数。但现在我们一个也没有定义。和往常把所有的函数定义在同一个代码文件中不同，今天我们需要把它们分开到不同的代码文件。

第二个文件：mainfunc.cpp 存放和计算有关的两个过程(函数)。

先看：CalcTotal()和 CalcAverage()。这两个函数我们将在 mainfunc.cpp 文件内定义。你可能又忘了“定义”这个术语？呵，就是“实现”，更白点，就是在 mainfunc.cpp 文件内“写”这两个函数。

下面是 mainfunc.cpp 的内容。在我们输入以下代码时，mainfunc.cpp 已经有了一些必要的内容，下面的代码，除了“#include ..”一行在文件最首外，其它均在原有内容之后添加。

```
#include <iostream.h> //在文件最首行
```

```
... ...
//--------------------------------------------------------------------------
//求总和的过程
//参数：n 用户需要输入的个数
void CalcTotal(int n)
{
   int num;
   int sum = 0;

   for(int i=0;i<n;i++)
   {
      cout << "请输入第" << i+1 <<"个整数：";
      cin >> num;

      sum += num;
   }

   cout << "总和为：" << sum << endl;
}
//--------------------------------------------------------------------------
//求平均值的过程
//参数：n 用户需要输入的个数
void CalcAverage(int n)
{
   int num;
   int sum = 0;
   float ave;
   for(int i=0;i<n;i++)
   {
      cout << "请输入第" << i+1 <<"个整数：";
      cin >> num;

      sum += num;
   }
   //注意不要除 0 出错：
   if( n >=0 )
   {
      ave = (float)sum / n;
      cout << "平均值：" << ave << endl;
   }
   else
   {
      cout << "个数为 0，不能求平均。" << endl;
   }
}
//--------------------------------------------------------------------------
```

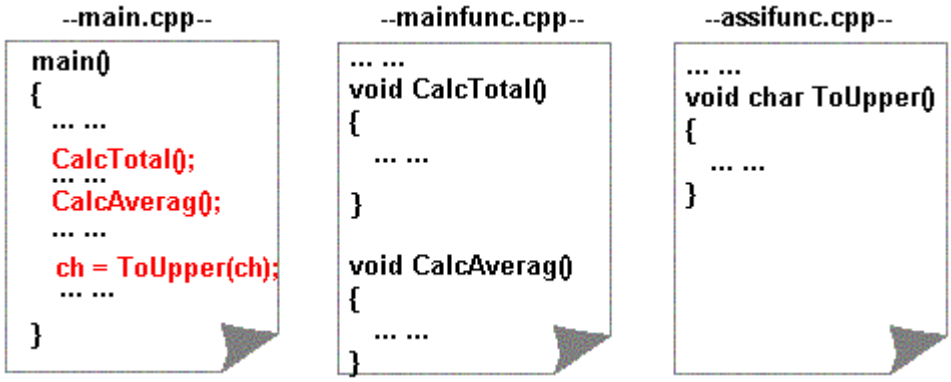
第三个文件：assifunc.cpp 用以存放辅助作用的函数，现在只有一个。

现在还差一个函数：ToUpper()。这个函数用来将用户输入的某个小写字母转换为大写。当然，如果用户输入的不是小写字母，那就不用转换。和上面的两个函数不同，它需要返回值。

我们把 ToUpper()函数单独放在 assifunc.cpp 里。同样，下面的代码加在该文件中原有的代码之后。不过本文件不需要 include <iostream.h>，因为没有用到 cin,cout 等。

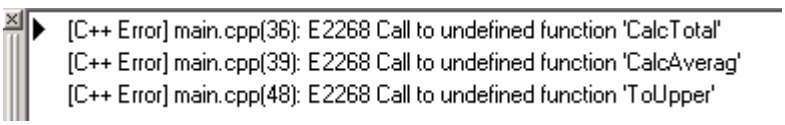
```
//小写字母转换为大写
//参数： c 待转换的字符
//返回值： 转换后的字符，如果原字符不是小写字母，则为原字符
char ToUpper(char c)
{
   int ca = 'A' - 'a'; //大写字母和小写字母之间差距多少?
   if(c >= 'a' && c <= 'z')
      c += ca;

   return c;
}
```

至此，所有自定义函数都已完成**定义**（实现），而三个文件的主要内容也以确定。让我们看看示意图：



main.cpp 中的 main()函数调用了三个函数。回忆我们学习过的“如何调用函数”的知识，当前代码在调用一个函数时，必须能“看到”这个函数。尽管 CalcTotal()、CalcAverage()、ToUpper()三个函数所在文件都在同一工程里，但是在 main.cpp 里的代码，还是看不到它们。想一想我们以前说的“请修理工”的比喻。现在情况是：在你所住的小区，甚至就是同一楼道里，就有一个电视修理工，但可惜你们互不认识，所以当你电视坏了，想“调用”一个修理工时，你还是找不到修理工。哎！要是有它的名片就好了。

让我们试试看，按 Ctrl + F9，编辑该工程。出错！



正好是三个错。分别告诉我们调用了三个没有定义的函数（Call to undefined function ...）。

（如果你出现的是一堆错，那有可能是你没有在前两个文件内最首行写：

“#include <iostream.h>”

或者是你有些代码输入有误。）

如何消除这三个错？两种方法。

第一种方法就是以前我们在讲“如何调用函数”的时候所说的，直接在调用直接声明要调用的函数。这里写出代码，算做是一次复习，然后我们将讲该方法不好之处。

在 main.cpp 的 main()函数之前加入如下三行函数声明：

```
void CalcTotal(int n);
void CalcAverage(int n);
char ToUpper(char c);

int main(int argc, char* argv[])
{
   ... ...
}
```

(上例中,尽管你可以将三行函数声明写在 main()函数体内,但不建议这样做)。

如果你一切输入正确的话，现在按 Ctrl + F9 或 F9 将可以完成编译或运行。

对于现在这个工程，这种方法确实也不能指责它有何不利之处。问题在于，如果我们还有其它文件中代码需要调用到这三个函数，我们就不得不在其它文件中也一一写上这三行声明。所以另一种方法是：把源文件中需要对外“共享”的函数声明统一写到某个头文件，然后凡是需要用到的其它文件，直接使用“#include”语句来包含该头文件，从而获得这些函数声明。

14.3 如何写头文件

在 CB 中，如果你通过上小节的方法新建个单元文件，那么 CB 将自动同时生成源文件和头文件。其实在 CB 里，源文件和头文件合称为单元文件，它们有相同的文件名，而扩展名一者为.cpp，另一为.h。

14.3.1 在头文件内加入函数声明

头文件：mainfunc.h

CalcTotal()和 CalcAverage()函数定义在 mainfunc.cpp 文件里，所以它们的声明最好写在对应的头文件 mainfunc.h 内。

下面我们就来看如何在头文件 mainfunc.h 内增加函数声明。

一开始，头文件内有以下这些代码。另外，我增加了一行用于标明我们新加的代码应写在哪里。

```
//---------------------------------------------------------------------------

#ifndef mainfuncH
#define mainfuncH
```

```
//---------------------------------------------------------------------------
/* ！！！头文件中，我们新增的代码必须写在此处！！！  */
#endif
```

和源文件中新增代码添加在最后不一样，头文件中新加代码 必须在#endif 之前插入。所以本例中，加完函数声明的代码应如其下所示。（另外，CB 总是在头文件的第二行留了一行空白行，我不知道它这是有意还是无意。总之这里是我们写本文件总体注释的好地方。记住，头文件像名片，用于让别人看，很有必要写得详细点）

```
//---------------------------------------------------------------------------
//主要操作函数
#ifndef mainfuncH
#define mainfuncH
//---------------------------------------------------------------------------
//计算总和：
void CalcTotal(int n);
//计算平均值：
void CalcAverage(int n);
//---------------------------------------------------------------------------
#endif
```

这就是“在头文件中声明函数”的整个过程。下面是另外一个头文件。

头文件：mainfunc.h

```
//---------------------------------------------------------------------------
//辅助操作函数
#ifndef assifuncH
#define assifuncH
//---------------------------------------------------------------------------
//将字符转换成大写
char ToUpper(char c);
#endif
```

今天我们学的是如何在头文件内声明函数，以后我们需要在头文件内声明变量，或者定义新的数据类型，它们都一样需要在上述的#endif 之前加入。

14.3.2 最常见的预编译语句

现在来解释这三行话：

```
#ifndef mainfuncH
#define mainfuncH

#endif
```

中间的 #define mainfuncH 我们有点脸熟。在第五章《变量与常量》中，我们讲过用宏表示常数。语法为：

#define 宏名称 宏值

比如，定义一个∏值：

#define PAI 3.14159

这里我们学的是宏定义的另一种用法：仅仅定义一个宏，不需要给出它的值，语法为：

#define 宏名称

比如：#define mainfuncH

定义了一个宏：mainfuncH。如果你无法理解“宏”这个词，不妨就当把它解释成“记号”。即编译器通过该语句，做了一个记号，记号名称为：mainfucH。

这么做的作用是什么呢？我们继续看上下文。

#ifndef 中， if 是“如果”，n 是 no,即“还没有”，def 是 define，即“定义”，那么：

#ifndef mainfuncH 意为：“如果还没有定义 mainfuncH 这个宏”，那么……

那么之后做什么呢？就是一直到 #endif 之间的语句。

总的再来看一遍：

mainfunc.h 中的主要内容：

```
#ifndef mainfuncH
#define mainfuncH

void CalcTotal(int n);
void CalcAverage(int n);

#endif
```

当编译第一次编译 mainfunc.h 文件时，宏 mainfuncH 还没有定义，因些，编译器通过对 #define mainfuncH 的编译而产生了宏 mainfuncH。当编译器第二次编译到 mainfunc.h 文件时，宏 mainfuncH 已经存在，所以该头文件被直接跳过，不会重复处理该头文件中内容，比如上面的两个函数声明。

你可能会问两个问题：第一，为什么编译器可能多次编译到同一个头文件？第二，为什么源文件，比如 mainfunc.cpp 就不需要用到#ifndef... #endif?

这两个问题只要回答了其中一个，另一个也就自然消解。

这是由头文件特性所决定的。头文件是用来被别人包含(include)的。谁都可以指定要包含某一头文件，这样就可能造成对该头文件的重复包含。

假设有头文件 head.h。如果 A 文件包含了 head.h，而 B 文件也包含了 head.h，那么编译器不会在编译 A 和编译 B 时，都要对该头文件尝试编译一次。

另外，头文件本身也可以包含另一个头文件，这种情况下，各文件之间互相嵌套包含的情况就更多了。

源文件(.c 或.cpp)尽管可以，但一般不被用来被别的文件包含，所以不需要在源文件中加这些语句。当然，如果需要，你也可以源文件中使用 #ifndef...#endif。

每生成一个头文件，包括在重命名它时，CB 会为我们取好该头文件中，上述的宏名称，它取该头文件的全小写文件名，加上一个大写的‘H’字母，比如： "mainfuncH"。请大家不要改变该宏的名称，以免出错。

除了 #ifndef ... #endif 语句外，还有它的相反逻辑的语句：
#ifdef ... #endif 它是在如果**有定义**某个宏，那么，编译将继续其后的语句。

另外就像有 if 语句，还有 if...else...语句一样，有 #ifdef ... #endif，也就还有这个语句：

```
#ifdef
... ...
#else
... ...
#endif
```

不过这些都和我们这里的头文件相关不大，我们暂时不讲。最后我们来解释一个名词“预编译”。
编译器在编译代码时，至少需要两遍的编译处理，其中第一次，就是专门用于处理所有以 #开头的语句，如上述的#ifndef...#endif、#define 等等。这一遍处理，我们称为**预编译**。

14.4 如何使用头文件

事实上我们经常在使用头文件。不过，以前我们一直在使用别人的头文件，今天是第一次使用我们自己的写的头件。
以前，我们几乎每个例子，包括今天的例子中，都需要在源文件的顶部写上一行：

```
#include <iostream.h>
```

或者：

```
#include <stdio.h>
```

iostream.h 和 stdio.h 都是 CB 提供给我们的头文件。这些头文件随 CB 安装时，被**保存在固定的文件夹内**。
今天的例子中，main.cpp 需要使用到在 mainfunc.h 和 assifunc.h。这是我们自己写的头文件，它们**保存在我们自定的文件夹中**。
包含自己写的头文件，和包含 CB 提供的头文件并无多大区别。
请在 main.cpp 代码顶部,加入以下黑体部分：

```
#include <iostream.h>
#include "mainfunc.h"
#include "assifunc.h"
//------------------------------------------------------------------------
```

二者的细小区别是，包含 CB 提供的头文件时，用尖括号<>；而包含我们自己的头文件时，使用双引号“”。CB 据此判断如何找到指定的头文件。<>相当于告诉 CB，这是你自己提供的头文件，到你安装时的头文件目录下找去吧，而“”则是告诉 CB，是这我自己写的头文件，请首先到我当前工程所在目录下查找，如果找不到，再到别的可能的头文件目录下找这个文件。(别的还有什么目录可能存放当前工程的头文件呢？稍后会讲。)

现在，我们让 main.cpp 包含了它想要的头文件，头文件内有它所需函数的正确声明，那么 main.cpp 中原来的这三行就多余了：

~~void CalcTotal(int n);~~

~~void CalcAverage(int n);~~

~~char ToUpper(char c);~~

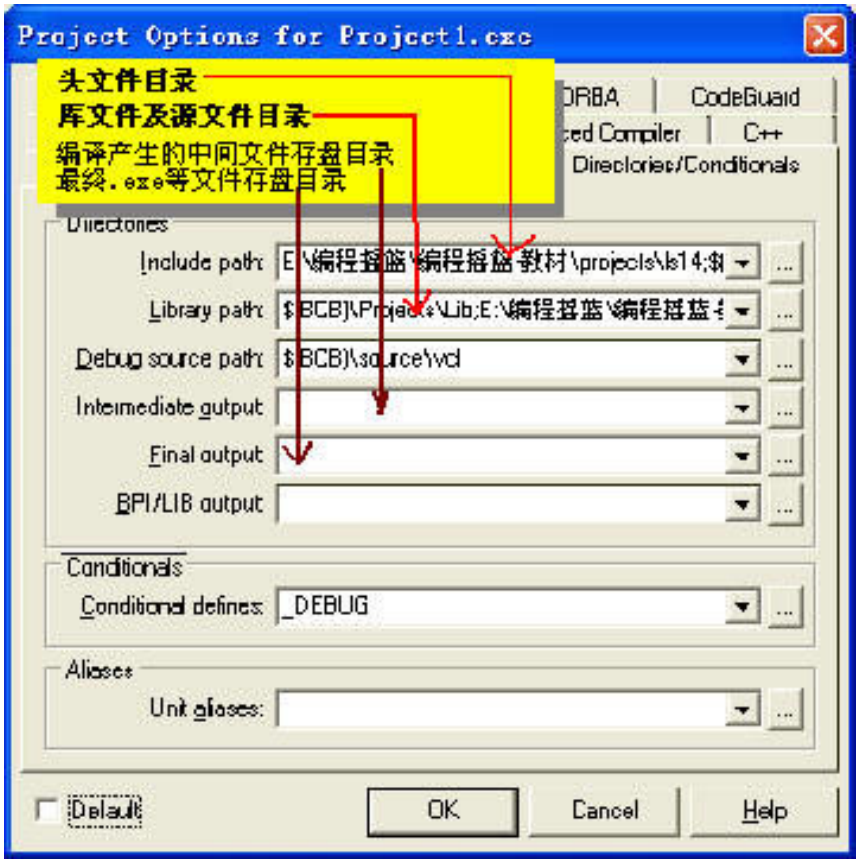
请删除。 然后，按 F9,程序正确编译，然后运行。这里我们不关心它的运行结果。

现在来看一眼在 CB 中如何设定某一工程的头文件目录。

必须先说清楚，在相当长的一段时间内，我们并不需要去进行此设置。对于 CB 提供的头文件，它们固定就在 CB 安装时自动存储的某些目录下，你只要记得包含这些头文件时，使用<>即可。对于我们自己写的头文件，我们都把它们和工程文件存放在同一目录下，暂时还没有什么理由需要把某个或某些头文件“扔”在别的目录下。所以，记住在包含自己的头文件时，对使用“”即可。

首先保证当前 CB 正打开着上面的那个例子工程。

然后，主菜单： Project | Options 或按 Ctrl + Shift + F11,打开“工程设置(Project Options)”对话框，并切换到“目录与条件(Directories/Conditionals)”页：



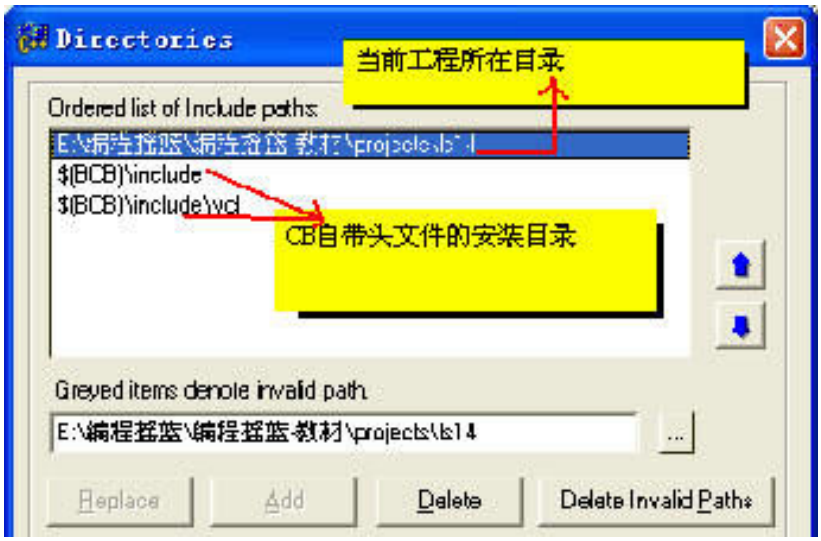
图中有关目录的设置共六行，我们说其中常用的四行。

最主要的，当然是今天所说的“头文件目录”。当 CB 编译时，当它遇到这样一行：

#include "xxxx.h"

那么，它必须找到文件 xxxx.h。如果，你写的是绝对路径：#include "c:\abc\123\xxxx.h"，那自然没有查找这一说，不过我们不会喜欢这样写程序，因为我们不希望源代换个位置就得一一去改那些绝对路径。事实上我们不可能把头文件到处放，总是固定那么几个目录，绝大多数就一个：所有源文件和头文件都在当前工程所在目录下。这里可以添加，删除，修改一些目录位置，CB 将按本设置中的目录次序去查找头文件。

请点击“头文件目录”右边，带 "..."的小按钮。出来一个新的对话框：



（$BCB） 表示 Borland C++Builder 的安装目录。

在这里，你可以修改(Replace)，增加(Add)，删除(Delete)，调整次序(向上和向下的蓝箭头)各个头文件目录。CB6 还提供了对无效目录的判断，如果列表中列出的某个目录实际上并不存在对应的文件夹，则将以灰色显示，并且可以用"Delete Invalid Paths"按钮全部删除。

我们什么也不用做。点 Cancel， 放弃就是。

其它目录的设定，操作完全一样。

关于在工程中如何使用头文件，我们就说这些了。

14.5 变量在多个源文件之间的使用

前面讲的是，通过在头文件中声明函数，可以达到让这个函数被其它文件共用的作用。同样地，变量也可以在多个源文件之间“共享”。下面我们就要讲，如何通过**声明变量**，以达到让其它文件共用同一个变量的目的。

14.5.1 变量声明

先说说“声明变量”。好像以前的课程只教过我们：定义变量，定义函数，声明函数，没有讲过“声明变量”啊？

我们很早就学过如何定义一个变量。（5.1.2）

比如：

```
//定义一个整型变量：
int age;

//然后，在后面的某处代码中使用这个变量：
... ...
age = 18;
cout << age << endl;
... ...
```

但是，我们没有遇到过如何声明一个变量。这是因为，定义一个变量的同时，也就声明了一个变量；绝大多数的时候，我们都是可以需要某个变量时，直接定义它。

今天的情况有点不一样。我们需要在某个源文件中定义一个变量，然后，在另外一个源文件中使用这个变量。

仍以前面 age 变量为例：

```
//我们在 A.cpp 文件中定义了这个变量：
int age;

//然后，在 B.cpp 文件中要使用这个变量：
age = 18;
cout << age << endl;
```

问题就出来了：在编译 B.cpp 文件时，编译器会说：“age 这个变量没有定义啊？”——当编译器在编译 B.cpp 时，它并不懂得去 A.cpp 里去找有关 age 的定义。
那么，能不能在 B.cpp 里再定义一次 age 变量呢?

```
//A.cpp 文件中：
int age;

//B.cpp 文件中：
int age;
age = 18;
cout << age << endl;
```

这样，单独编译 A.cpp，或 B.cpp，都可以通过。但一旦要编译整个工程，编译器又会报错：“怎么有两个 age 变量的定义啊”？

不要嘲笑编译器为什么这么笨笨。C,C++是一门严谨的的计算机语言，我们不能指望编译器会“智能”地猜测程序员的企图。

解决方法是，仅在一处**定义变量**，别的代码需要用到该变量，但无法看到前面的定义时，则改为“声明变量”。

声明变量的语法：

extern 数据类型 变量名

和定义变量的语法相比，多了前面的 **extern** 这个关键字。

extern 意为“外来的”・・・它的作用在于告诉编译器：有这个变量，它可能不存在当前的文件中，但它肯定要存在于工程中的某一个源文件中。

这就好像：微软公司在北京招人，微软的报名方法是：在北京的应聘者必须当天去面试，而外地应聘者则通过发 e-mail 先报名，然后以后再去面试。 在 C,C++里，不处于当前源文件中的变量被称为外部变量。比喻中，发 e-mail 就相当于外部变量在某一个源中写个声明。声明什么呢？就是声明“我存在啊！虽然我现在不在这里,但是我真的存在！”

上例中，正确的代码应该这样写：

```
//A.cpp 文件中：
int age;

//B.cpp 文件中：
extern int age;
age = 18;
cout << age << endl;
```

变量 age 是在 A.cpp 文件里定义的，当 B.cpp 文件要使用它时，必须先声明。这就是我们讲半天课的核心。

(有些教材并不认为 extern int age; 是在声明一个变量，它们把这也称为是“定义变量”的一种，只不过它是定义了一个名部变量。我认为这样认为不好，一来它造成了学习者认为“变量可以重复定义”的错误认为，二来它也造成了不统一，函数有“定义”和“声明”两种形式，而变量都没有“声明”。

可能你会说，现在也不统一啊？函数声明没有“extern″，而变量却需要？呵呵，其实恰恰相反。函数声明本来也是需要一个“extern”的，比如：

```
extern void CalcTotal(int n);
```

你在代码里这样完全正确！只不过由于函数声明和函数定义的格式区别很大，（声明没有函数体，定义则必须有函数体），所以这个 extern 就算不写，也可以让编译器认出来它是一个“声明”。结果就规定可以不写″extern″了。

而变量呢？

```
extern int age;     //这是声明
int age;            //这是定义
```

你看看，不写"extern"可不行！就靠它来区分是定义还是声明了。
如此而已。）

14.5.2 多个文件中共享变量的实例

做一个最简单的例子。新建一个控制台工程。然后再加一个单元文件。把工程存盘为 Project1.bpr，把两个源文件分别存盘为 Unit1.cpp、Unit2.cpp （即，都采用默认文件名）。

程序内容是：在 Unit1.cpp 内定义一个变量，即：int age,并且，要求用户输入。在 Unit2.cpp 里，写一函数，OutputAgeText(),它根据 age 的值， 输出一些文本。

请问，变量 age 在哪里定义？又在哪里声明？

定义指定是在 Unit1.cpp 文件里，而声明，则可以在 Unit2.cpp 内直接声明（如上例中的红色代码），也可以是在头文件 Unit1.h 里声明，然后在 Unit2.cpp 内使用 include 来包含 Unit1.h。 事实让，声明也可以放在 Unit2.h 内。只要能让 Unit2.cpp“看到”这个声明即可。这一点和函数的声明一个道理。

我们采用放在 Unit2.cpp 中的方法，该方法所需代码如下：

//Unit1.cpp 内的主要代码：

```
#include <iostream.h>
#include <conio.h>
#pragma hdrstop
#include "Unit2.h"
... ...
//---------------------------------------------------------------------------
int age; //全局变量，年龄
#pragma argsused
int main(int argc, char* argv[])
{
   cout << "请输入您的年龄：" ;
   cin >> age;

   //调用 Unit2.cpp 中的函数，该函数根据 age，作出相应输出
   OutAgeText();

   getch();
   return 0;
}
//---------------------------------------------------------------------------
```

```
//Unit2.cpp 中的主要代码：
#include <iostream.h>
... ...
extern int age;  //需要 Unit1.cpp 内定义的变量

//报名参加“没有弯路”的学员各行业，年龄段也各处不同，在此，我们用这个函数作为共勉！
void OutAgeText()
{
   if(age < 15)
      cout  << "计算机要从娃娃抓起！" << endl;
   else if(age < 25)
      cout << "青春年华，正是学习编程的黄金时代！" << endl;
   else if(age < 35)
      cout << "学习编程需要热情，更需要理性！我和您一样，也在这个年龄段！"<< endl;
   else if(age < 45)
      cout <<"活到老，学到老！何况您还未老。杀毒王王江民，不也在这个时候才开始学习电脑吗？
" << endl;
   else
      cout <<  "前辈，只要您像学书法一样潜心学编程！您一定会有收获！" << endl;
}
//---------------------------------------------------------------------------


//Unit2.h 的主要代码：

//声明 OutAgeText()函数，供 Unit1.cpp 使用
void OutAgeText();
//---------------------------------------------------------------------------
```

请大家完成这个工程，直到能正确运行。

现在我们得到一个印象：当我们定义了一个函数或变量之后，似乎所有的源代码文件中都可以使用它，只要你在使用之前写一下相应的声明。

这样会不会带来麻烦了？想象一下，你在 A 文件定义了一个变量： int i， 那么以后你在别的文件里就不能再定义这个变量了！原因前面已经说过，编译器（或链接器）会说有两个变量重名。函数也一样，尽管它有重载机制，便那也只能是有限制地允许函数重名。

事实上，上例中的 int age 是一个全局变量。关于“全局”的解释，需要引起 C,C++程序的另一话题：作用范围。这是下一章的内容。在那一章里，我们将看到，大部分变量只在它一定的作用范围内“生存”，不同的作用范围的变量就可以毫无障碍地重名了。

休息休息（该点眼药水了・・・），然后学习本章附加一节。

14.6 附：如何单独生成一个头文件

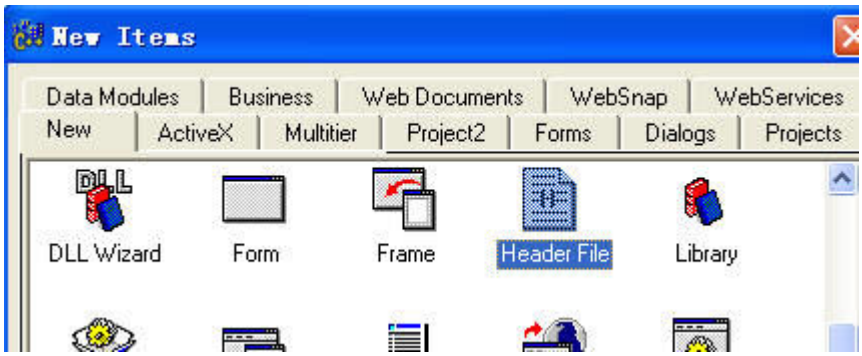
在 14.5.2 试一试在多个文件中共享变量 小节中，我们说，变量的声明可以像函数声明一样放在某个头文件中，然后使用者通过 include 语句包含该头文件，从而获得变量的声明。

如果你想自己再写一次那个例子（建议），并且改用上述的方法，那么你可能会发现一个小小的故障：Unit1.cpp 竟然没有配套的头文件？

CB 在生成一个空白的控制台工程时，会自动主函数(main())文件，即默认文件名为：Unit1.cpp 的源文件，但它没有为该文件生成对应 Unit1.h。
这是因为作为控制台程序，一般都是小程序，一个源文件即可全部解决。

现在，作为一个课程例子，我们要来讲讲如何为 Unit1.cpp 生成一个头文件。

CB6 中选择菜单：New | Other,CB5 中选择 New...，或直接点“  ”，出现 New Item...对话框,选择 New 页内的 Header File:



确认后，CB 生成头文件： File1.h，其内容为空白，我们需要手工加入以下内容：

```
#ifndef unit1H
#define unit1H

#endif
```

然后，选择主菜单 File | Save As... 将其另存为： Unit1.h。 完后成，在 Unit1.cpp 或 Unit1.h 内，按 Ctrl + F6,你就可以看到二者之间的切换了。

下一章见！

第十五章 存储类型、作用域、可见性和生存期

上一章我们讲了“程序的文件结构”。主要涉及到一个问题，即：A 文件中定义的某个变量，如果在 B 文件也能使用它。其间我们学到一个新关键字：**extern**，它用来声明一个变量，并且指明这是一个“外来的”的变量。如果你对我说的这些感到陌生，那么你先复习一下上一章。

这一章，我们正是要从 extern 说起。

15.1 存储类型

存储类型分“外部存储”和“静态存储”两种。

15.1.1 外部存储

外部存储类型使用 **extern** 关键字表示。

上一章我们其实一直在用外部存储类型的变量。

一个全局变量或函数，如果你需要在其它源文件中可以共用到，那么你必须将它声明为“外部存储类型”。这其实就是上一章我们所讲的内容。这里再举个例子，简要复述一次。

在 A.cpp 文件中，有一个全局变量 a,和一个函数： func();

```
//A.cpp 文件：
...
```

```
int a;
void func()
{
  ...
}
...
```

我们希望在 B.cpp 或更多其它文件可以使用到变量 a 和函数 func(),必须在**“合适的位置”**声明二者：

```
//B.cpp 文件：
...
extern int a;         //a 由另一源文件(A.cpp)定义
extern void func();  //func 由另一源文件(A.cpp)定义

a = 100;
func();
...
```

这里例子中，**“合适的位置”**是在 B.cpp 文件里。其它合适的位置，比如在头文件里的例子，请复习上一章。

另外一点需要得强调一次：**函数的定义默认就是外部的,**所以上面 func()之前的 extern 也可以省略。

在使用 extern 声明全局变量或函数时，一定要注意：所声明的变量或函数必须在，且仅在一个源文件中实现定义。

如果你的程序声明了一个外部变量，但却没有在任何源文件中定义它，程序将可以通编译，但无法链接通过。下面是该错误类型的一个例子，大家请打开 CB，将下面代码写入完整的一个控制台工程。

错误一、只有声明，没有定义：

1、用 CB 新建一个空白控制台工程，CB 将自动生成 Unit1.cpp。加入以下黑体部分：

```
extern void func2();
int main(int argc, char* argv[])
{
   func2();
   return 0;
}
```

2、新建一个单元文件(菜单：New | Unit)：Unit2.cpp。在 Unit2.cpp 后面加入：

```
extern int a;         //a 由另一源文件(A.cpp)定义
extern void func();  //func 由另一源文件(A.cpp)定义

void func2()
```

```
{
   a = 100;
   func();
}
```

现在按 Ctrl + F9，将出现以下错误：

[Linker Error] 表明这是一个“链接”错误。两个错误分别是说变量 a 和函数 func()没有定义。

（你可能奇怪为什么错误消息里，变量'a'的名字变成了'_a'？这是编译器遵循某些标准，在编译结果上对变量名做了一些改变，我们不必理会）

请大家想一想，并试一试，如何解决这两个链接错误。

错误二、有声明，但重复定义

1、用 CB 新建一个空白控制台工程，CB 将自动生成 Unit1.cpp。加入以下黑体部分：

```
extern void func2();

int a;            // <--全局变量 a 在此定义了一次
void func()       // <--函数 func()在此定义了一次
{
  a = 20;
}

int main(int argc, char* argv[])
{
   func2();
   return 0;
}
```

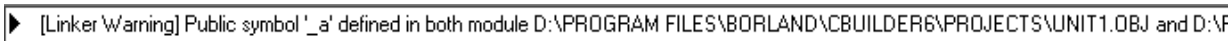
2、和错误一的第 2 步完全一样：

```
extern int a;         //a 由另一源文件(A.cpp)定义
extern void func();  //func 由另一源文件(A.cpp)定义

void func2()
{
   a = 100;
   func();
}
```

3、再新建一个单元文件：Unit3.cpp，在文件后加入：

```
int a;          // <--全局变量 a 在此又定义了一次
void func()     // <--函数 func()在此又定义了一次
{
  a = 20;
}
```

现在编译这个含有三个单元文件的工程。这回答的是一个链接“**警告/Warning**”：

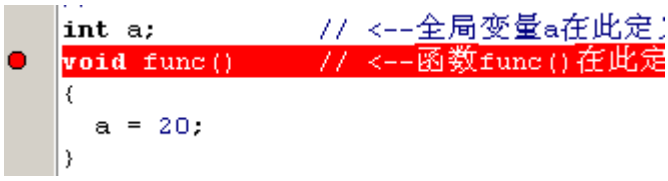


警告很长，无非是说全局变量 ’a’ 在两个模块内重复定义。对了，func()函数我们不是也重复定义了吗？为什么没有得到警告？这是因为 CB 对重复定义的函数，将只取其一，然后自动抛弃所有重复项。下面的操作可以看到这一结果。

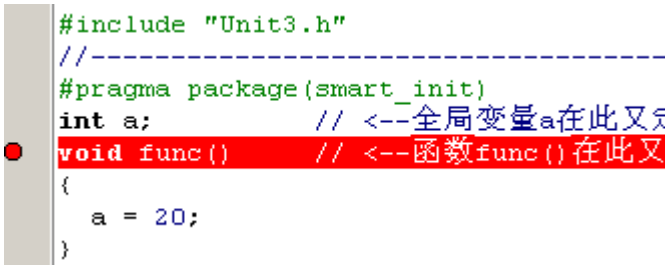
既然是错误类型只是“警告”，那就是说我们可以硬下心肠不管，继续运行。我们现在来看看，两个 func()函数，CB 到底用了哪一个？

这里需要在运行前，在两个 func()的函数定义处，都设置断点：

第一个断点：在 Unit1.cpp 文件里：

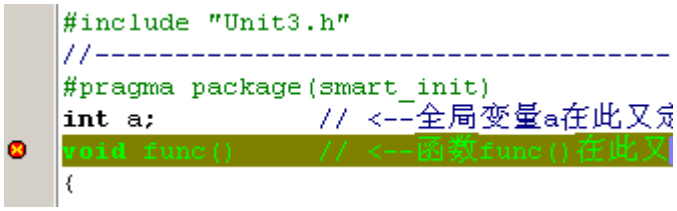


第二个断点：在 Unit3.cpp 文件里：



然后按 F9，运行，我们看到断点停在 Unit1.cpp 中的 func()定义上：

而另一处：Unit3.cpp 里的断点，变“黄”（无效断点）了：



之所以成为无效断点，有两种原因：

其一是某些代码，比如单纯的变量声明：int a; 或如宏定义等，这些代码在编译后成为程序的初始化部分，无需运行。

其二是某些无用，或可优化的代码中编译过程被丢弃。

这里正是第二种情况。

尽管变量或函数重复定义似乎并不造成“致命”错误，但我们同样需要严加注意，消除所有这类错误。请大家对本例进行改错。

15.1.2 静态存储类型

静态存储类型使用 static 关键字表示。

static 关键限定其所修饰的全局变量或函数只能在当前源文件中使用。

反过来说，如果我们确定某个全局变量仅仅是在当前源文件中使用，我们可以限定它为静态存储类型。

static 的使用格式 ：

static 变量定义或函数定义

如：

```
static int a;
static void func();
```

举一个例子，下面的代码可以正确编译、运行：

Unit1.cpp 文件：

```
...
extern int a;
int main(int argc, char* argv[])
{
   a = 100;

   return 0;
}
```

Unit2.cpp 文件：

```
...
int a;
```

说明：在 Unit1.cpp 文件中用到了外部变量：a, a 在 Unit2.cpp 文件内定义。

现在，我们要限定 Unit2.cpp 里的变量 a 只能在 Unit2.cpp 内可以使用：

Unit2.cpp 文件：

```
...
static int a;
```

我们为 a 的定义加了一个修饰：static。现在再编译，编译器提示一个链接错误，我们在本章前面说过的：“变量 a 没有定义”：

静态函数的例子类似：

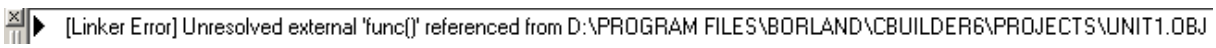
Unit1.cpp 文件：

```
...
void func();
int main(int argc, char* argv[])
{
   func();
   return 0;
}
```

Unit2.cpp 文件：

```
int i;
static void func()
{
  i = 100;
}
```

按 Ctrl+F9，得到以下链接错误：



又是两个制造错误例子，不要偷懒，务必亲手制造出这两个错误，并且再改正后，才继续看下面的课程。千万不要仅满足于“看得懂”就不动手。那样绝对不可能学会编程。

static 还有一种用法,称为函数局部静态变量，作用和这里的“全局静态”关系不大，我们在后面的“生

存期”中会讲到。

由于静态变量或静态函数只在当前文件（定义它的文件）中有效，所以我们完全可以在多个文件中，定义两个或多个同名的静态变量或函数。

比如在 A 文件和 B 文件中分别定义两个静态变量 a：

A 文件中：
```
static int a;
```

B 文件中：
```
static int a;
```

这两个变量完全独立，之间没有任何关系，占用各自的内存地址。你在 A 文件中改 a 的值，不会影响 B 文件中那个 a 的值。

15.2 作用域和可见性

作用域和可见性可以说是对一个问题的两种角度的思考。

“域”，就是范围；而“作用”，应理解为“起作用”，也可称为“有效”。所以作用域就是讲一个变量或函数在代码中起作用的范围，或者说，一个变量或函数的“有效范围”。打个比方，就像枪发出的子弹，有一定的射程，出了这个射程，就是出了子弹的“有效”范围,这颗子弹就失去了作用。

代码中的变量或函数，有的可以在整个程序中的所有范围内起作用，这称为“全局”的变量或函数。而有的只能在一定的范围内起作用，称为“局部”变量。

15.2.1 局部作用域

我们在 5.1.3 “如何为变量命名”这一小节中讲到：**“不能**在同一作用范围内有同名变量”。因此，下面的代码是错误的：

```
...
int a;    //第一次定义 a
int b;
b = 2*a;
int a;   //错误：又定义了一次 a
...
```

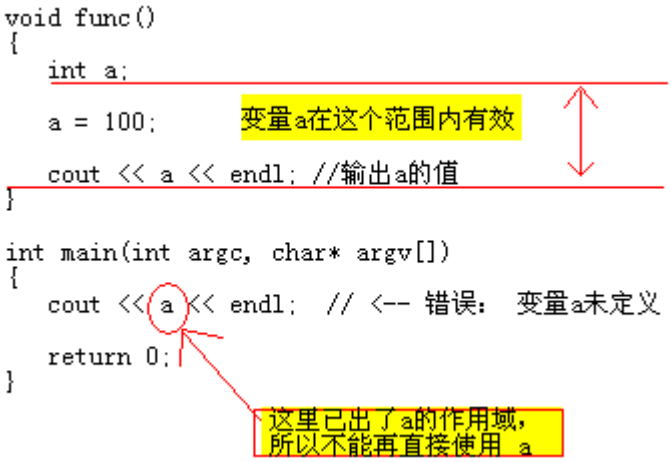
那么，在什么情况下，变量属于不同的作用范围呢？我们这里说的是第一种：**一对{}括起来的代码范围，**

属于一个局部作用域。如果这个局部作用域包含更小的子作用域，那么子作用域的具有较高的优先级。在一个局部作用域内，变量或函数从其声明或定义的位置开始，一直作用到该作用域结束为止。

例一：变量只在其作用域内有效

```
void func()
{
   int a;

   a = 100;

   cout << a << endl; //输出 a 的值
}

int main(int argc, char* argv[])
{
   cout << a << endl;  // <-- 错误： 变量 a 未定义

   return 0;
}
```

说明：在函数 func()中，我们定义了变量 a，但这个变量的“作用域”在 } 之前停止。所以，出了花括号以后，变量 a 就不存在了。请看图示：



结论：在局部作用域内定义的变量，其有效范围从它定义的行开始，一直到该局部作用域结束。
在局部作用域内定义的变量，称为“局部变量”。

上例中的局部作用域是一个函数。其它什么地方我们还能用到{}呢？很多，所有使用到复合语句的地方，比如：

```
//if 语句
```

```
if( i> j)
{
    int a;
    ... ...
}
```

上面的 a 是一个局部变量，处在的 if 语句所带的那对 {} 之内。

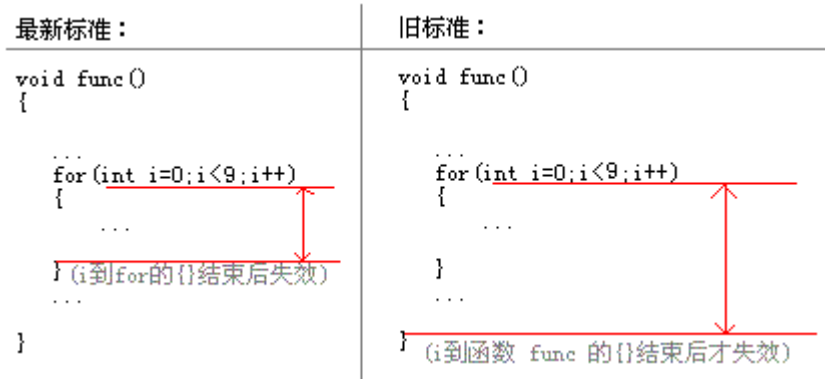
```
//for 语句：
for(int i=0;i<100;i++)
{
   int a;
   ... ...
}
```

上面的 a 也是一个局部变量。处在 for 语句带的{}之内。

for 语句涉及局部作用域时，有一点需要特别注意：上面代码中，变量 i 的作用域是什么？

根据最新的 ANSI C++ 规定，在 for 的初始语句中声明的变量，其作用范围是从它定义的位置开始，一直到 for 所带语句的作用域结束。而原来老的标准是出了 for 语句仍然有效，直到 for 语句外层的局部作用域结束。请看对比：

假设有一 for 语句，它的外层是一个函数。新老标准规定的不同作用域对比如下：



如果按照旧标准，下面的代码将有错，但对新标准，则是正确的，请大家考虑为什么：

```
void func()
{
   for(int i=0;i<9;i++)
   {
     cout << i << endl;
   }

  for(int i=9;i>0;i--) //<-- 在这一行，旧标准的编译器将报错，为什么?
  {
    cout << i << endl;
  }
}
```

Borland C++ Builder 对新旧标准都可支持，只需通过工程中的编译设置来设置采用何种标准。默认总是采用新标准。记住：如果你在代码中偶尔有需要旧标准要求的效果，你只需把代码码写成这样：

```
int i;
for(i=0;i<9;i++)
{
   ...
}
```

这时候，i 的作用域就将从其定义行开始，一直越过整个 for 语句。

其它还有不少能用到复合语句（一对{}所括起的语句组）的流程控制语句，如 do..while 等。请复习以前相关课程。

其实，就算没有流程控制语句，我们也可以根据需要，在代码中直接加上一对{}，人为地制造一个“局部作用域”。比如在某个函数中：

```
void func()
{
   int a = 100;
   cout << a << endl;

   {
      int a = 200;
      cout << a << endl;
   }

   cout << a << endl;
}
```

代码中红色部分即是我们制造的一个局部作用域。执行该函数，将有如下输出：

```
100
200
100
```

你能理解吗？

15.2.2 全局作用域 和 域操作符

如果一个变量声明或定义不在任何局部作用域之内，该变量称为**全局变量**。同样，一个函数声明不处于任何局部作用域内，则该函数是全局函数。

一个全局变量从它声明或定义的行起，将一起直接作用到源文件的结束。

请看下例：

```
//设有文件 Unit1.cpp,内定义一个全局变量：
```

```
int a = 100;

void func()
{
  cout << a << endl;
}
```

输出：

```
100
```

我们今天还要学习到一个新的操作符，域操作符 “::”。域操作符也称“名字空间操作符”，由于我们还没学到“名字空间”，所以这里重点在于它在全局作用域上的使用方法。

:: 域操作符，它要求编译器将其所修饰的变量或函数看成全局的。反过来说，当编译器遇到一个使用:: 修饰的变量或函数时，编译器仅从全局的范围内查找该变量的定义。

下面讲到作用域的嵌套时，你可以进一步理解全局作用域如何起作用，同时，下例也是我们实例演示如何使用域作用符::的好地方。

15.2.3 作用域嵌套及可见性

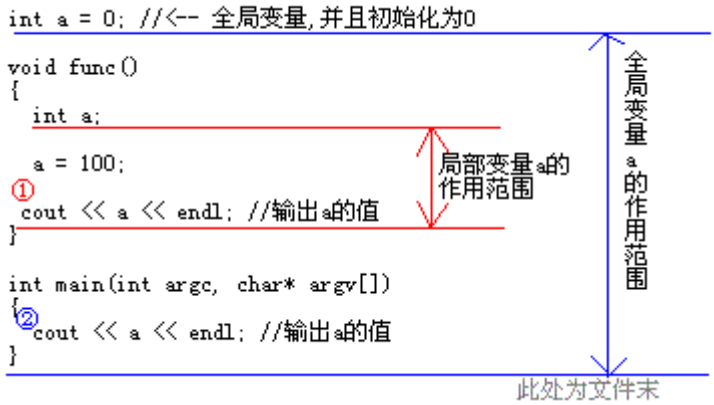
例二：嵌套的两个作用域

在例一的基础上，我增加一个全局变量：

```
int a = 0; //<-- 全局变量,并且初始化为 0

void func()
{
  int a;

  a = 100;

  cout << a << endl; //输出 a 的值
}

int main(int argc, char* argv[])
{
  cout << a << endl; //输出 a 的值
}
```

我们在 5.1.3 “如何为变量命名”这一小节中讲到： “不能在同一作用范围内有同名变量”。 上面的

代码中，定义了两个 a，但并不违反这一规则。因为二者处于不同的作用范围内。下图标明了两个 a 的不同作用范围：



从图示中看到：两个变量 a：1 个为全局变量，一个为局部变量。前者的作用域包含了后者的作用域。这称为作用域的嵌套。

如果在多层的作用域里，有变量同名，那么内层的变量起作用，而外层的同名变量暂时失去作用。比如在上例中，当代码执行到①处时，所输出的是函数 func()内的 a。而代码②处，输出的是全局变量 a。

这就引出一个“可见性”这个词，当内层的变量和外层的变量同名时，在内层里，外层的变量暂时地失去了可见性。

不过，如果外层是全局作用域，那么我们可以使用::操作符来让它在内层有同名变量的情况下，仍然可见。

```
int a = 0;
void func()
{
  int a;
  a = 100;

  cout << a << endl; //输出内层的 a;
  cout << ::a << endl;  //输出全局的 a。
}
```

最后请大家把本节中讲到例子，都在 CB 上实例际演练一下。

15.3 生存期

一个变量为什么有会不同的作用域？其中一种最常见的原因就是它有一定的生存期。什么叫生存期？就像人一样，在活着的时候，可以“起作用”，死了以后，就不存在了，一了百了。

那么，在什么情况下一个变量是“活”着，又在什么情况下它是“死”了，或“不存在”了呢？

大家知道，变量是要占用内存的。比哪一个 int 类型的变量占用 4 个字节的内存，或一个 char 类型的变量占用 1 个字节的内存。如果这个变量还占用着内存，那么我们就认为它是“活着”，即，它存在着。而一

个变量释放了它所占用的内存，我们就认为它“死了”，“不存在”了。

有哪个同学能告诉我，在我们的教程中，我这是第几次讲到“变量和内存”的关系？呵，我也记不得了。不管怎样，这里又是一次——我们必须从整体上讲一讲：一个程序在内存中如何存放？

15.3.1 程序的内存分区

先从程序上看“生”和“死”。

用 CB 编译出一个可执行文件(.exe)，它被存放在磁盘上。当它没有运行时，我们认为它是“死”的。而当我们双击它，让它“跑”起来时，我们认为它是“活”的，有了“生命”。等我们关闭它，或它自行运行结束，它又回到了“死”的状态下。在这个过程里。

程序运行时，它会从操作系统那里分得一块内存。然后程序就会把这些内存（严格讲是内存的地址）进行划分，哪里到哪里用来作什么。这有点像我们从老板那里领来 2000 大洋，其中 1000 无要交月租，500 元做生活费……真惨。

那么，程序有哪些需要入占用内存呢？

首先，代码需要一个空间来存放。因此，到手的内存首先要分出一块放代码的地方，称为代码区。剩下的是数据。根据不同需要，存放数据有区域有三种：数据区，栈区，堆区。为什么存放数据的内存需要分成三个区域？这个我先不说，先来说说数据（变量等）被放入不同的区内，将遇上什么样不同的命运。

第一、放入数据区的数据。

生存期：这些数据的命运最好。它们拥有和程序一样长的生存期。程序运行时，它们就被分配了内存，然后就死死占着，直到程序结束。

谁负责生死：这些数据如何产生，如何释放，都是程序自动完成的，我们程序员不用去费心为产生或释放这些变量写代码。

占用内存的大小：这些数据都必须有已知，且固定的大小，比如一个 int 变量，大小是 4 个字节，一个 char 类型，大小是 1 个字节。为什么必须这样？因为如果这个数据可以占用的大小是未定的，那么，程序就不可能为自动分配内存。

初始化：就是这个变量最开始的值是什么？放在数据区里的数据，可以是程序员用代码初始化，比如：

```
int a = 100;
```

这样，a 的值按你意思去办，并初始化为 100；但如果你没有写初始的代码，如：

```
int a;
```

那么，数据区内的数据将被初始化为全是 0。

第二、放入堆区的数据。

生存期：堆内的数据什么时候“生（分配内存）”，什么时候“死（释放内存）”，由程序员决定。

谁负责生死：当然就是程序员了。C++里，有专门的函数或操作符来为堆里的变量分配或释放内存。程序员通过写这些代码来在需要时，让某个堆里的变量“生”，不需要时，让它“死”。

占用内存的大小：堆里的数据占用的内存可以是固定的，也可以是可变的。这就是 C,C++里最强大也最难学的内容：“指针”所要做事。

初始化：由程序员完成。如果程序员不给它初始值，则它的值是未定的。

由于程序员掌握着堆区内的数据的“生死大权”，并且决定着该数据占用多少内存。所以在写程序时，必须特别注意这些数据。一不小心就会出错。比如一个数据还没有分配内存呢，你就要使用它，就会出错。更

常见的是，一个数据，你为它分配了内存，可是却始终没有为替它释放内存，那样就会造成“内存泄漏”。就算你的程序都退出了，这个数据依然可能“阴魂不散”地占用着内存。

第三、放入栈区的数据。

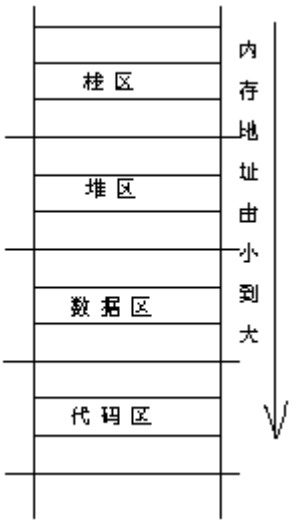
生存期：对比前面的两种，数据区里数据具有永久的生存期，而堆里的数据的生存期算是“临时”的。需要了，程序员写代码产生；不需要了，又由程序员写代码释放。在程序员，临时才需要变量非常多，如果每个变量都由程序员来负责产生、释放，那程序员岂不很累？并且很危险（万一忘了释放哪个大块头的家伙....）。所以，必须有一种机制可以让程序自己来产生和释放某些临时变量。所以，放入堆区的数据是只有程序员才能决定的何时需要，何时不需的临时数据，而栈区数据则是编译器就能决定是否需要的临时数据。当然，要想让编译器能知道数据什么时候需要，什么时候不需要，就必须做一种约定。这正是我们现在讲的“生存期”的语法内容。

谁负责生死：程序（和数据区的一样）。

占用内存的大小：固定大小（和数据区的一样）。

初始化：由程序员完成。如果程序员不给它初始值，则它的值是未定的(和堆区的一样)。

下面是三个区加上代码区的分布示意图：



现在，我们也比较好回答前面的问题：“为什么存放数据的内存需要分成三个区域”?原因正在于程序所要用到的数据具有不同的生存期要求，所以编译器把它们分别放到不同空间，好方便实现要求。

生存期和作用域的关系是：如果一个变量已经没有了生存期，那么自然它也就没了有作用域。但反过来，如果一个变量出了它的作用域，它并不一定就失去了生存期。典型的如函数内的静态数据，下面会讲到。

15.3.2 动态生存期

就是放在“堆区”的数据。这些数据是在程序运行到某一处时，由程序员写的代码动态产生；后面又由程序员写的代码进行释放。我们现在还没有学习如何为变量（指针变量）分配和释放的内存的知识。

15.3.3 局部生存期

这里的局部和前面讲“局部作用域”一致，都是指“**一对{}括起来的代码范围**”。

请看下面代码，并思考问题：

```
//从前，有一个函数……
void func()
{
   //函数内，有一个局部变量……
   int a;

   cout << a << endl;

   a = 100;
}

//看清楚了，上面输出 a 的值的语句， 位于给 a 赋值之前！
//然后，下面的代码是两次调用这个函数：
...
func();
func();
...
```

第一次调用，我们知道屏幕肯定是要输出一个莫名其妙的数，因未初始化的局部变量，其值是不定的。我们以前讲变量时，就做过实例。现在，这里的变量 a 被输出后，我们让赋于它 100。再接下来，我们又调用了一次函数 func();请问这回输出的值，是 100 呢？或者仍然是莫名其妙的数？

大家打开 CB，把这个例子做做。注意，动手生成一个空白的控制台工程后，调用 func()的那两行代码，要放到主函数 main()内，形如：

```
……
int main(int argc, char* argv[])
{
    func();
    func();

    ……
}
```

正确答案应该是：“仍然是莫名其妙的数”。尽管在第一次调用时 func()时，局部变量最后被赋值为 100；但很可惜，出了函数这个作用域，a 立即就死掉了……第二次再调用函数 func()时，那个像个 a 投胎转世的婴儿，一切又重新开始……它又是一个没有被赋值的的变量了。

请大家把本例中的变量 a 改为全局变量，并且在函数 func()的定义之前定义。再试一试。

15.3.4 静态生存期

就是放在“数据区”里的数据。程序一运行时，它们就开始存在；程序结束后，它们自动消亡。

这里讲的“静态”，和前面的“静态存储类型”不是一个意思。（老师，我忘了什么叫“静态存储类型”？呵，这有可能，本章的内容互相都有些关联和相似，大家多看几遍本章，最主要是课程让你动手的地儿，你就动手做，正所谓“该出手时就出手手……”）。

“静态存储类型”是指：一个全局变量，它被加上 static 之后，就只能在本文件内使用，别的文件不能通过加 extern 的声明来使用它。

“静态生存期“是指：一个变量，它仅仅产生和消亡一次（即在程序运行时产生，在程序退时消亡），而不像“动态生存期”或“局部生存期”那样可以生生死死，不断“投胎转世”。

下面的代码演示了“静态生存期”和“局部生存期”变量的不同。请你看完以后，回答问题。

```
#include <iostream.h>
//定义，声明一个全局变量：
int a;
//声明一个函数,定义在后面
void func();

int main(int argc, char* argv[])
{
   int b = 100;

   a = 10;
   cout << a << end;
   cout << b << end;

   //调用函数 func：
   func();
}

//func()的定义：
void func()
{
   cout << a << endl;
   cout << b << endl;
}
```

哪里有错呢？请大家想想，试试。

15.3.5 局部静态变量

//从前，有一个函数……

```
void func()
{
   //函数内，有一个局部变量……
   int a;

   cout << a << endl;

   a = 100;
}

//调用两次：
func();
func();
```

同样是这个例子，我们只是要把 int a 之前加上一个 static 关键字：

```
void func()
{
   //函数内，有一个局部静态变量……
   static int a;

   cout << a << endl;

   a = 100;
}
...
func();
func();
...
```

我们要问的也是同样一个问题：第二次调用 func()后，输出的 a 值是多少？

这回答案是：输出的值是 100。

这就是局部静态变量的特殊之处：尽管出了函数的作用域之后，变量已经不可见，并且也失去了作用。但是，它仍然存在着！并且保留着它最后的值。因此，它也是静态生存期。它也只在程序结束之后，才失去生存期。

上面讲的是局部静态变量“死”的问题，它也只“死”一次，对应地，显然它也只能“生”一次。

```
void func()
{
   static int a = 30;  //在定义时，同时初始化该局部静态变量为 30。

   cout << a << endl;

   a = 100;
}
...
```

```
func();
func();
...
```

这回要问的是第一次调用 func()时，输出的是什么？第二次呢？

答案：第一次输出 30，第二次输出 100，以后若有第三次，第四次，也是输出 100。这就是说，初始化：static int = 30；这一句，仅被执行一次！

好，假如代码是这样子呢?

```
void func()
{
    static int a;
   a = 30;  //改成不是在定义时同时初始化

   cout << a << endl;

   a = 100;
}
...
func();
func();
...
```

请回答我，这回，两次调用 func()分别输出什么？

有关内存的堆、栈内容，最近我曾在 CSDN 上做过回答。大家如有兴趣，不妨去看看。(别忘了，我叫“nanyu”)。要学习编程，CSDN 是不错的地方。大家有空常去(最好注册个 ID)。我因为太忙，去不了几次。

http://expert.csdn.net/Expert/topic/1255/1255577.xml?temp=.1724665

15.4 对前 15 章的一点小结

一般教材用 6 或 7 章的内容，我们扩展成 15 章。大家可能嫌我讲得太慢。这我承认。哎，我的事情太多了。特别是最近这一段时间，先是笔者得了肾结石，剧痛了一夜最后住入医院，接着是我的宝宝发高烧，我在医院里守了一宿，未料到我的爱人接着也 得上医院……可恨的是电信局里不知哪个家伙一时兴起，来个什么“改线”，改来改去也不知改错了什么，我的 ASDL 就一断数天……幸好我天天打电话免费（不，是倒贴电话费 啊）为他们培训什么叫 ADSL。（其实我也不懂，不过我总得旁敲侧击地暗示他们，“你们是不是动了什么啊？好好想想？再想想？”，哎，总的来说，他们的客户服务态度还是很好的……）。

不管怎样，在此我向所有付费报名的学员致歉。

这 15 章的内容，属于 C,C++的基础知识，其中有些更是基础中基础。从第 16 章（数组）开始，就开始中级或高级的内容了。这些新的内容都有一个特点：都和内存地址有着千丝万缕的关系。所以大家有时间抓紧把前面的都复习中，其中犹其是要把我讲到的，有关“变量和内存”关系，全部重新消化一遍。

明天就是冬至了。圣诞节即到，在此我向大家问个节日快乐，并感谢早早给我发来贺卡的几位同学。

更重要的是春节即将到来，但愿我们能在春节前学完 C++。然后一起向更为精彩的 Windows 编程世界出发。

第十六章　数组（一）

16.1 引子 -- “小王”成绩管理系统

小王老师参加了“编程摇篮”的学习，这一天他的系主任请他写程序。
系主任提的第一个要求是：
用户输入 6 个班级的各自的学生英语成绩总分，要求程序输出成绩总分和平均分。
“这简单了！”小王心想，“在前面的课程里，早就有过成绩统计的例子了嘛！改改就行。”

16.1.1 “小王成绩管理系统 Ver1.0”

功能：求六个班级的成绩总分及平均分 （为了方便起见，下面的所有成绩数据都使用整型，不考虑小数）

```
......
int cj,zcj=0,pjcj=0;  //各班成绩，总成绩，平均成绩
```

```
for(int i=0; i<6;i++)
{
  cout << "请输入第" << i+1 << "班的分数:";
  cin  >> cj;

  zcj += cj;  //累加总成绩
}

//求平均:
pjcj = zcj / 6;

cout << "总成绩:" << zcj << endl;
cout << "平均成绩:" << pjcj << endl;
......
```

小王迅速测试了一遍，没有问题，任务胜利完成！小王的形像立刻在系主任的眼里高大起来……不过，客户的需求总是在不断进步的！系主任立即提出第二个要求：

必须加入查询功能，比如说，用户输入 1，则程序输出一班的成绩，输入 2，则输出二班的成绩，以此类推。

“这可不好办了！南郁老师没有教我这个啊……”小王心里很着急，“要让用户可以查询，那至少我得让程序把这 6 个成绩记下。”小王来回看了好几遍 for 循环，也没有想出如何在 for 循环里记下用户输入的成绩。他决定放弃了……等等！不就 6 个班级吗？我不用循环 总行吧，在讲循环流程的那一章里，不是举了一个最笨的办法吗？（见 10.4 ）

（尽管这样写程序，按南郁老师说只能得到“鸭蛋”，可以桌子那边传来系主任殷切的目光……）

一阵“噼噼叭叭”，小王删除了前面所有代码，可以实现查询的新代码如下：

16.1.2 “倒退的 2.0”

功能：求六个班级的成绩总分及平均分，并可实现对各班成绩查询。

```
......
//定义六个变量，用于存储六个班级的成绩:
int cj1,cj2,cj3,cj4,classSocre5,cj6;

//老师说下面的方法很“笨”，不过没办法了，只能这样...
//(反复用 Ctrl+C,Ctrl+V 还是很方便的)
cout << "请输入第 1 班的成绩:" << endl;
cin >> cj1;

cout << "请输入第 2 班的成绩:" << endl;
cin >> cj2;

cout << "请输入第 3 班的成绩:" << endl;
cin >> cj3;
```

```
cout << "请输入第 4 班的成绩：" << endl;
cin >> cj4;

cout << "请输入第 5 班的成绩：" << endl;
cin >> cj5;

cout << "请输入第 6 班的成绩：" << endl;
cin >> cj6;

//求总成绩和平均成绩:
int  zcj = (cj1+cj2+cj3+cj4+cj5+cj6);
int  pjcj = zcj / 6;

cout << "总成绩：" << zcj << endl;
cout << "平均成绩：" << pjcj << endl;

//下面是新功能：查询：

char c;

do
{
  cout << "请输入要查询的班级次序(1~6,0:退出)" <<endl;
  cin >> c;

  //用 switch 实现对用户输入的判断及分流：
  switch(c)
  {
      case '1' : cout << cj1 << endl; break;
      case '2' : cout << cj2 << endl; break;
      case '3' : cout << cj3 << endl; break;
      case '4' : cout << cj4 << endl; break;
      case '5' : cout << cj5 << endl; break;
      case '6' : cout << cj6 << endl; break;
      //其它的，输出出错消息：
      default :  cout << "您的输入有误！" << endl;
  }
}
while(c != '0') ; //用户输入 '0'，表示退出
......
```

手有点酸，不过毕竟实现了查询功能。小王把写好的程序拿到系主任那里一运行，主任的眼睛透过厚厚的眼镜片闪闪发光……并且开始打听“没有弯路，编程摇篮”的网址。然后他说：“小王，你这个程序太好了！我们完全有必要把它发扬光大！你这就把程序稍作修改，扩展到整个学校的 5000 个学生的成绩都可以查

询……”

“5000 个学生？我岂不要定义 5000 个成绩变量……”小王眼前一黑，倒在地上。

16.2 数组的定义及使用

请从下面的实例中讲解，数组变量与我们以前学过的普通变量在定义和使用上有什么区别。

16.2.1 实例演示

处理 5000 个学员的成绩，是否需要定义 5000 个变量？

现实生活中有大量的数据类似于此，它们**有相同的类型,需要的处理的方法也一致**。 为了实现对这些数据统一表达和处理，C，C++提供了“数组”这一数据结构。

数据类型 数组变量名[个数常量];

和普通变量定义的语法：数据类型 变量名；相比，主要是多了一个“[个数]”。

定义数组时，方括号内的数值指明该数组包含几个元素；使用数组时，方括号内的数值表示：使用的是第几个元素。为了直观，我假设可以用汉字来命名变量：

int 抽屉[100];

这里定义了一个“抽屉数组”，这个数组包含了 100 个抽屉。

而在使用时：

int a = 抽屉[10];

这里的 10 表示我们要使用第 10 个抽屉（其实是第 11 个，见后）。

还是来看小王老师的学生成绩管理系统吧。因为主任说要管理 5000 个学生成绩，那么，我们就得定义一个成绩数组，包含 5000 个元素。

```
int  cj[5000];
```

这一行我们就相当于定义了 5000 个 int 类型的变量。这 5000 个变量有个统一的名字：cj。为了区分出每个变量，还需要使用“[下标]”的格式指明。下标从 0 开始，一直到 4999。比如，第一变量是： cj[0],第二个为：cj[1]……最后一个是 cj[4999]。

强调：**在 C,C++中，数组的下标从 0 开始**。因此，我们定义了数组 int cj[5000]；得到的是从 cj[0] ~ cj[4999]的变量，而不是 cj[1] ~ cj[5000]。

比如，让第 100 个学员的成绩为 80：

cj[99] = 80;

为什么是 99 而不是 100？因为数组的下标从 0 开始。

再比如，让变量 a 等于第 100 个学员的成绩：

int a = cj[99];

有了“数组”这个新武器，要解决前面全校成绩查询问题，一下子变得简单了。

```
......
//定义一个含 5000 个元素的数组,用于存入 5000 个学生的成绩：
int cj[5000];
```

```
//总成绩，平均成绩：
int zcj=0, pjcj;

//仍然可以用我们熟悉的循环来实现输入：
for(int i=0; i<5000; i++)  //i 从 0 开始，到第 49999(含）个结束。
{
   cout << "请输入第" << i+1 << "学员的成绩:";
   cin >> cj[i];       //输入数组中第 i 个元素

   //不断累加总成绩：
   zcj += cj[i];
}

//平均成绩：
pjcj = zcj / 5000;

//输出：
cout << "总成绩：" << zcj << endl;
cout << "平均成绩：" << pjcj << endl;

//下面实现查询：
int i;

do
{
   cout << "请输入您要查询的学生次序号(1 ~ " << 5000 << "):" ;
   cin >> i;

   if( i >= 1 && i <= 5000)
   {
      cout << cj[i-1] << endl; //问：为什么索引是 i-1，而不是 i ？
   }
   else if( i != 0)
   {
      cout << "您的输入有误！" << endl;
   }
}
while(i != 0);  //用户输入数字 0，表示结束。
......
```

代码用了两个循环——是的，**有了数组，你会发现循环流程将用得越来越多**——第一个 for 循环用以输入 5000 个成绩。第二个 do...while 循环用以查询。用户可以输入 1 到 5000 之内的数字（和现实生活习惯一致）。

下面要复习“宏”的知识。如果学校多了一个学生，那么我们就得将上面所有以斜体标出的“5000”改为5001。这是一件让人觉得无趣，且很容易出错的工作。另外，现在我们要调试这段程序，难道你真的有耐心输入 5000 个成绩？在调试时，我们可能希望只输入 5 个成绩。

这时我们可以使用“宏”。下面的代码中，先是定义了一个宏：MAX_CJ_COUNT,而后所有立即数 5000 都用它来表示。当我们需要改变人数时，只需更改 MAX_CJ_COUNT 的宏定义。

```
//定义一个含 5000 个元素的数组,用于存入 5000 个学生的成绩：
#define MAX_CJ_COUNT  5000 //定义一个宏，方便修改为实际学生人数。
int cj[MAX_CJ_COUNT];
```

以下代码略。有关宏的用处，请复习：5.3.2 用宏表示常数

除非确实可以一次决定某数组的大小，否则，使用一个意义明确的宏来定义数组的大小，总是一个不错的主意。

16.2.2 个数定义必须是常量

再一看眼数组定义的语法：

数据类型 数组变量名[个数常量];

注意“常量”两字，这说明，个数必须是一个可以事先决定的值，并且该值不能被改变。

比如用立即数：

```
int arr[5000];
```

或者用宏：

```
#define MAX_CJ_COUNT  5000
int arr[MAX_CJ_COUNT];
```

或者用常量：

```
const int max_cj_count = 5000;
int arr[max_cj_count];
```

就是不能用变量：

```
int max_cj_count =  5000;
int arr[max_cj_count];   // error! 不能用变量指定数组的大小。
```

为什么呢？因为数组占用的内存空间大小必须在程序编译时决定，并且一旦决定了，就不能再改变。所以只能用常量（不变的量）来指明数组的大小。当然，这是指在数据区或栈区分配内存（和程序有一样的全存期），如果是在堆区，则可以动态地分配数组的大小，这些我们在指针一章里讲。

有关常量和变量的区别，如果有所不清，请前面章节内容。

16.3 如何给数组中的元素赋值

如果把单个变量看成是“游兵散勇”的话，那么数组对应的是“集团”。集团的“兵”就是我们前面说的数组的元素。这些“兵”不再有单独的名字，而是统一使用编号来区别，这个编号，我们称为“下标”。

在和数组打交道时，我们需要分清:是对整个数组操作，还是对数组中的单个元素进行操作。

16.3.1 在定义数组时初始化。

普通变量可以在定义时同时赋初值：

```
int a = 100;
```

也可以在定义以后才赋值：

```
int a;
a = 100;
```

对于数组变量，则只能在定义时，对**整个数组**赋初值：

数据类型 数组变量名[个数] = {元素 1 初值,元素 2 初值,};

即，将初值用一对 {} （ 花括号 ）括起来，相邻的值之间用逗号分隔。

比如：

```
int arr[10] = {9,8,7,6,5,4,3,2,1,0};
```

上面定义一了个数组 arr,共 10 个元素。初始值为从 9 到 0。即，执行上面代码以后，arr[0]值为 10，arr[1]值为 9……arr[9]值为 0。

在初始化赋值是时，注意所给值的个数不能超过数组的大小，比如：

```
int arr[10] = {10,9,8,7,6,5,4,3,2,1,0};  //错误！越界了
```

你定义了数组为 10 个元素，可是你却赋给它 11 个值，这称为数组越界：你在宾馆里预定了 10 间房，你却让 11 个人去住，多出的那个人挤在哪里呢？编译器遇上这类问题时，很明智地停了下来。

不过，你可以给它少于定义个数的初值：

```
int arr[10] = {9,8,7,5};                    //允许
```

你定义了数组为 10 个元素，但你可以给它少于 10 个的初始值。那些没有得到初始值的元素值是多少呢？对于全局变量来说，将被编译器初始化为 0，对于局部变量而言，则是未定义的值(不可预测的值)。如果只定义数组，没有对其进行任何初始化，同样适于本情况。所有元素的初值均依本数组是全局或局部变量而定，为 0 或未定义值。

可以跳过一两个元素不初始化吗？如：

```
int arr[10] =  {9,,7,,6};  //跳过中间的某些元素，C：OK；C++: Error。
```

因为我们主要学习 C++，所以认为**跳过数组中某些元素的初始化赋值是错误的**。

也就是说，你尽可以先预定下多个房间，然后先只派部分人去住。不过 C 说：允许你们跳着房号住，而 C++则要求依次住下，把空房留在后面。

你还可以不指定数组元素个数，直接通过对其初始化来让编译器得到它的实际个数：

```
int arr[] = {9,8,7};  //元素个数为： 3
```

即，**在未指定义大小时，则初始值的个数就是数组元素的个数**。

不过，你**不能既不指定数组元素个数，也不进行初始化：**

```
int arr[];   //Error，  到底是几个元素？
```

这也没有什么不好理解的，去几个人，就开几个房间。让宾馆老板自己去数人头，我们不必非得自己报人数——前提是，房客一个一个的都得事先现身。

16.3.2 在定义之后为元素赋值

很多时候，我们无法在定义数组的同时就知道数组中各元素的值，所以，这时需要在定义以后各数组中的各个元素赋值。记住，此时只能对**单个元素**进行直接操作。这和普通变量不一样，下面的代码是错误的：

```
int arr[5];
...
arr[5] = {1,2,3,4,5}; //错，在编译器看来，arr[5]是数组 arr 的第 6 个元素。
//或
arr = {1,2,3,4,5};   //错，仍然不行。
```

这一点和普通变量不一样。也就是说，对数组整体的初始化，只能在定义时实行。

大都数情况，我们这样改变数组中某个元素的值：

```
int arr[5];

arr[0] = 95;
arr[1] = 89;
arr[2] = 100;
...
```

前面关于成绩管理的例子中，已经有过如何改变数组元素值的代码：cin >> cj[i]; 这一句将用户输入的成绩赋给数组 cj 中的第 i 个元素（i 从 0 计起）。

两个数组可以相互赋值吗？答案也是不行：

```
int arr1[5] = {1,2,3,4,5};
int arr2[5];

arr2 = arr2; //不行！整个数组之间不能直接相互赋值。
```

你可能很不理解为什么上面的代码会出错，不过在学习后面有关指针及内存地址的更多知识以后，你会明白。现在我们只能告诉你，以我们的所学，可以方便地用一个循环来 实现将一个数组内的值全部赋值给另一个数组，这也称为数组间的拷贝。

```
for(int i=0;i<5;i++)
{
    arr2[i] = arr1[i]; //正确，数组元素之间可以相互赋值。
}
```

当然，这样做可一定要注意：当两个数组的元素个数不一致时，不要越界访问：

```
int arr1[5] = {1,2,3,4,5};
int arr2[6];

for(int i = 0;i<6;i++)  // i 从 0 到 5
```

```
{
    arr2[i] = arr1[i];
}
```

arr2 有 6 个元素，而 arr1 只有 5 个。当循环执行到 i 为 5 时，访问 arr1[5]将造成越界。

（越界可以直观地理解为“越出边界“，即越出数组的边界,访问到了数组以外的内存，其内容将是未知的。）

16.4 控制台下如何输入和输出数组

输入和输出属于程序设计中，用户界面这一方面的内容。在 Windows 图形界面和控制台或 DOS 的字符界面下会完全不同。

16.4.1 输入

先说输入：

以前要让用户输入个某个变量的值，很简单，比如让用户输入一个学成成绩（整型）：

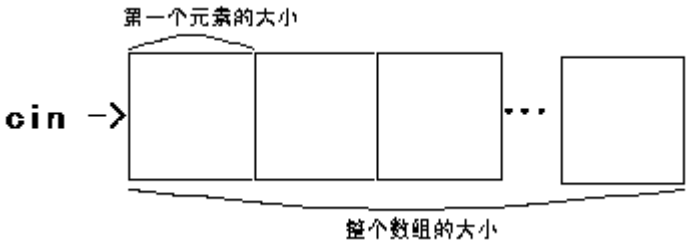
```
int cj;
cin >> cj;
```

现在，cj 是一个数组： int cj[5];

那么要输入这 5 个成绩，是否可以直接写：cin >> cj; ？

你可能猜到了：不行。不信可以试试（建议：不管你信不信都请试试）。

为什么不行？原因是当 cin 碰上 cj 时，它只知道 cj 是整型数组，但它不知道这个数组多大。你可能又要问，为什么 cin 能知道 cj 是一个数组，却不能知道这个数组有多大呢？这又涉及到变量与内存关系。我们先来看这个图：



（图：当 cin 遭遇数组,cin 只“看”到数组中的第一个元素）

数组中的各个元素总是连续地存入在内存里。所以在 C，C++里,为了效率，数组变量的传递，总是只传递第一个元素的内存地址，（后面的元素自然紧跟着）。所以，当 cin 得到 cj 这个数组时，它只看到第一个元素，它知道第一个元素的大小，却并不知道整个数组的大小，也不知道到底有多少个元素在这个数组里。这就有比你是一个窗口的售票员，你知道外面来买票的是一个队伍，但你不知这个队伍到底有多少。当然，C、C++也提供“卖团体票“的函数，但那就需要向这个函数额外传递一个数组的大小的参数。

说了这么长关于输入的问题，其实解决方法我们早于用过，来一个循环流程，一个一个输入即可：

```
int arr[5];
```

```
for(int i=0;i<5;i++)
{
   cin >> arr[i];
}
```

16.4.2 输出

再说输出。

可想而已，想一次性输出一个数组中的各元素的值，也是不可能的：

```
int arr[5] = {1,2,3,4,5};

cout << arr; //并不能输出 arr 内 5 个元素的内容。
```

你不能就此放过这个问题，我一直建议要多动手，多试试，这次就可以看出作用。当你试着在 C++ Builder 内写上面的代码，然后编译，你会发现编译器并没有报错。程序也可以运行。

请照着下面的代码“试试”：

```
#include <iostream.h>
int main(int argc, char* argv[])
{
    int arr[5] = {1,2,3,4,5};

    cout << arr; //提问：输出一个数组，结果会是什么？

    cin.get();  //从本章起，不再使用 getchar();改为本行
}
```

这运行结果截图： 

输出了一个怪怪的数值。这个数值正是数组 arr 在内存中的“门牌号”——地址。并且，数组的地址其实也正是数组中第一个元素的地址。关于数组与内存地址的关系。我们在下一章详细讲解。这里要关心的正事是如何输出数组中的每个元素的值，代码如下：

```
int arr[5];
for(int i=0;i<5;i++)
{
   cout << arr[i];
}
```

同输入一样，还是对元素操作。

16.4.3 数组输入输出练习

现在请大家打开 CB，新建一控制台工程。实现以下要求。

接受用户的键盘输入 10 个整数，然后输出这 10 个数，每行输出 1 个数。

提示：

1、别忘了文件开始位置加入本行：#include <iostream.h>

2、由于 Windows 操作系统的问题，使得其控制台环境下，getch()或 cin.get()在前面有用户输入数值并且回车的情况下，将直接接受回车字符，导致起不到“暂停”的作用。解决方法为：使用两行：

```
cin.get();
cin.get();
```

或：

```
getchar();
getchar();
```

从本章开始，我们将改为使用 cin.get()来起暂停作用。请大家注意。

16.5 数组的尺寸

这里的尺寸说的是数组占用内存空间的大小，即某个数组占用了多少个字节。

以前我们说一个 int 型数据占用 4 个字节，那么一个有 n 个元素的整型数组占用了多少个字节？几乎可直接猜到：4 * n 个字节。而一个有 n 个元素的字符型或 bool 型数组，则应占用 1 * n 个字节，因为它们单个元素的大小均为 1 字节。

我们用 sizeof 测量某个变量占用的字节数：

```
int a;
int size = sizeof(a);
```

size 将得到 4。

同样，我们可以这样来得到数组的尺寸：

```
int arr[10];
int size = sizeof(arr);
```

size 将得到 40。

你可能会问，为什么前面的 cin 不知道 arr 的尺寸，而 sizeof 却可以得知？因为 sizeof 事实上是在编译阶段就进行计算。

得知数组的尺寸的一个最大的用处，就是可以在程序里通过计算得到这个数组包含几个元素。这相当于一道小学算术题：

已知某个数组占用 40 个字节的空间，并且知道该数组单个元素占用 4 个字节的空间，请问这个数组包含多少个元素？

看：

```
int arr[] = {1,2,3,4,5,6,7,8,9,0};
//计算 arr 数组内有多少个元素：
int count = sizeof(arr) / sizeof(arr[0]);
```

count 将得到 10，即 arr 数组中元素的个数。

其中 sizeof(arr) 得到 整个数组占用的字节数，sizeof(arr[0]) 得到单个元素占用的字节数。数组的基本概念就是每一个元素的大小是并且必须一致的（你可以推想以后我们会学到内中元素可以大小不一的数据结构），为什么我们要取第 1 个（下标为 0）元素？看看下例：

```
int arr[] = {0};
int count = sizeof(arr) / sizeof(arr[0]);
```

明白了吗？我们或许不知道或 arr 有几个元素，但有一点，并且只有一点可以确定：只要是数组，那么它就**至少**会有一个元素。可以有 int arr[1];但不允许有：int arr[0]。

最后，对于简单的数据类型，sizeof 还可以对其类型进行求值：

```
int a;
int size = sizeof(a); size 将得到 4
```

而直接对 int 操作：

```
int size = sizeof(int); size 也将得到 4.
```

数组没有和后者对应的做法。虽然数组也可以有不同类型：整型数组，实型（浮点型）数组，布尔型数组，字符型数组，但数组的大小不仅和它的类型有关系，而且和它含有几个元素有关系。以公式表达：

数组的大小 = 数组类型的大小 * 元素个数。

或者：

sizeof(数组) = sizeof(数组类型) * 元素个数

或者：

sizeof(数组) = sizeof(数组[0]) * 元素个数

16.6 字符数组

字符数组具有一些特殊性，主要在三方面：1、初始化，2、输入输出、3、结束字符。大家需要注意了。

16.6.1 字符数组通常用于显示

前面我们用整型表示成绩，所以数组：int cj[5000];是一个整数数组，或整型数组。

其它的数据类型，如果浮点型：float、布尔型：bool 或者字符型 char 均可以有相关的数组，如：

```
int age[] {25,32,29,40};
float money[] = {1000.50,2000,2870.95};
bool  married[] = {false,true};
char name[] = {'M','i','k','e'};
```

最后一行定义了一个字符数组，用来存储一个英文人名：“Mike”。相对于其它数据类型的数组而言，我们将更有需要将字符数组输出。比如上面的 name,如何将“Mike”输出到电脑屏幕呢？依据上一小节的知识，应该是：

```
char name[] = {'M','i','k','e'};
for(int i=0; i<sizeof(name) / sizeof(char); i++)
   cout << name[i];
```

上面代码的屏幕输出是： Mike

假设有一个整型数组：

```
int age[] {25,32,29,40};
```

当我们要输出它时，大抵应是希望输出如：

25 32 29 40 或者加上逗号： 25,32,29,40
或者干脆换行：

```
25
32
29
40
```

但对于字符串数组，我们更多地希望它如上面“Mike”一样连续输出。C，C++ 设计者们看到了这一点，所以针对字符数组，有以下特殊做法。说是特殊，是指其它类型的数组没有，对于人的习惯来说，它倒是非常“自然”的做法。

16.6.2 字符数组初始化

首先是初始化部分，字符数组允许这样实现：

```
char name[] = "Mike";
```

对于中国人来说，这是一个“救命”的做法，我们不用去“拆”汉字了——还记得吗？一个汉字占两个字节，即一个汉字其实是由两个字符组成的。

```
char myname[] = "南郁";
```

当然，明确指定元素个数也不违法：

```
char name[5] = "Mike";
```

为什么我指定 name 的元素个数为 5？ 这是一件更重要的问题。后面会说到。

另外，如果你习惯于给数组加花括号，也可以：

```
char name[] = {"Mike"};
```

当以后讲到二维数组，花括号就是必须的了。

16.6.3 字符数组的输入

然后是输入：

```
char name[4];
cout << "请输入您的姓名，不要超过 4 个英文字符或 2 个汉字！" << endl;
cin >> name;
```

这是运行的结果：

```
请输入你的姓名，不要超过4个字符或2个汉字!
nanyu
```

每一个(正直 && 认真学习)的学员都会指出老师的自相矛盾：前面刚刚说，当“cin 遭遇数组，cin 只‘看’到数组中的第一个元素”，即：cin 不知道 name 的大小！

事实上 cin 仍然不知道 name 的大小。所以上段代码一开始就“请求”用户只输入 4 个以内的字符。用户当然不会那么听话！如果他输入 5 个，或更多字符，会发生什么糟糕的事情？如果你试试，你可能会发现一切正常，但其实性质严重的错误确实存在。就像一个小孩偷了家里的东西，他倒不会因此被派出所抓走（因为大多数家庭会选择内部处理），但错误的严重性值每个父母重视。

有没有办法让 cin 指接受指定个数的字符？而不管具有“破坏欲望”的用户输入多长？

（本来， cin, cout 这一类仅在 DOS 或控制台下或 UNIX 或 Linux 下的字符界面下才能用的东东，都不是我们的学习目标。记住，我们只是为了学习 C，C++基础知识上的方便，才染指。在讲如何实现前述要求之前，我们来插一个“广告”吧 ：）放松放松。

想像南郁老师是电视里的那个漂亮的幼儿园大班女阿姨……

大家——想像时光倒流，回到无忧无虑的童年时代，脸上露出天真无邪的笑容——为了配合场景，还得想像人手一张花花绿绿的盗版 Windows XP 安装盘，用左手拿，举着，摆好 POS。

好了，本阿姨问：“我们的目标是~~~”。

大家齐声回答：“Windows 编程！！”

呵呵，回答“没有弯路”也许更像一点。 Windows 编程的课程即将出笼。。。一开始将都是免费的，暂不收费。但是若要为了“没有弯路“，《白话 C++ 》一定要学完学好。所以你若没有 C++基础，建议不要放弃而跑到去学习《白话 Windows 编程》。）

（下面这一段是选学内容）

cin.getline 的用法：

getline 是一个函数，它可以接受用户的输入的字符，直到已达指定个数，或者用户输入了特定的字符。

它的函数声明形式（函数原型）如下：

```
istream& getline(char line[], int size, char endchar = '\n');
```

不用管它的返回类型，来关心它的三个参数：

char line[]： 就是一个字符数组，用户输入的内容将存入在该数组内。

int size : 最多接受几个字符？用户超过 size 的输入都将不被接受。

char endchar :当用户输入 endchar 指定的字符时，自动结束。默认是回车符。

结合后两个参数，getline 可以方便地实现： 用户最多输入指定个数的字符，如果超过，则仅指定个数的前面字符有效，如果没有超过，则用户可以通过回车来结束输入。

```
char name[4];
cin.getline(name,4,'\n');
```

由于 endchar 默认已经是 '\n'，所以后面那行也可以写成：

```
cin.getline(name,4);
```

如果你想输入三个汉字，把 name 改成 6 即可。作为配合，你可以用一个宏来指定 name 的元素个数，也可以用前面的 sizeof 的方法：

```
char name[6];
```

```
cin.getine(name,sizeof(name));
```

16.6.4 字符数组的输出

和输入对应，字符数组也允许不用循环，直接用 cout 输出。

```
char name[] = "Mike";
cout << name;
```

上面代码的屏幕输出是： Mike 。效果和前面使用 for 循环的代码完全一样。

同样的问题是，cout 怎么会知道 name 内有四个字符呢？
为了加深你对这个问题的理解，先来看下面的程序：

```
char name[] = {'M','i','k','e'};  //我们用“老”的方法初始
cout << name;
```

运行结果是： 

在“Mike”后面，接了一个怪字符。为什么？因为 cout 不知道 name 中有几个字符，结果出输出过头了。你的机器上可能“Mike”之后不是这个怪字符，它是随机不定的。

或许这样的结果还是不足于让你“感动”。下面这个例子绝对值得你去一试。我写完整一点：

```
#include <iostream.h>

char name[] = {'M','i','k','e'};
char othername[] = "Tom";

int main(int argc, char* argv[])
{
  cout << name << endl;
  cin.get();

  return 0;
}
```

运行结果是： 

。

在内存里，由于数组：othername 紧挨着 name 后面，而 cout 在输出 name 时，它并不知道 name 中有几个元素，所以把 othername 的内容也当成是 name 的，一并输出了。这应该是一个明明白白的错误表现了。

注意，如果你想把 name 和 othername 的定义放在 main 函数内，那为了达到上面的效果，你应把二者的次序倒过来：

```
int main(int argc, char* argv[])
{
```

```
    char othername[] = "Tom";        //改成它先定义。
    char name[] = {'M','i','k','e'};

    cout << name << endl;
    cin.get();

    return 0;
}
```

这是因为函数内的局部变量是建立在栈内存，而栈内存是“倒立”的。

如何让 cout 知道 它要输出的字符数组有几个字符？并不是依靠 cout 提供对应于 getline()的函数，而是由 C++ 内建机制，提供一个间接方法来保证程序有办法断定一个字符数组的结束。这就是：为字符数组添加一个空字符为结束。

先来看答案：

```
char name[] = "Mike";
cout << name;
```

就是这么简单，奥妙在于初始化 name 时，请采用这种“特殊”而又“自然”的方法：直接给出用双引号括起来的字符串。当你用这种方法初始时，编译器在编译过程中，将偷偷地为在这个字符数组后面加上一个元素。这个元素就是空字符。

（什么叫空字符？也称“零字符”就是值为 0 的字符，表示为：'\0' 如果忘了，请复习第五章。）

只有使用 "Mike" 这种方法，编译器才会做这件附加的工作。采用 {'M','i','k','e'}这种方法时，空字符不会被添加。

添加了空字符，cout 就能知道 name 有多长吗？其实它仍然不知，但它不管，它只需一个一个字符地输出，直到碰上空字符，即：'\0'时，它就认为这个字符串结束了。

在 char name[] = "Mike"; 中，编译器真的往 name 后加一个空字符了吗？

我们用两个办法来验证。

第一、通过调试工具观察

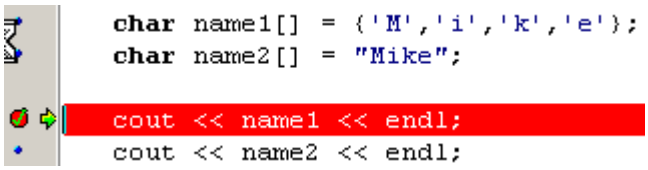
眼见为实。字符数组最后的空字符仅用于表示当前字符串结束了，本身并不输出（即使输出，也是一个不占位置的字符，看不到）。但是当程序运行时，它在内存是的的确确占了一个字节，我们可以通过调试来查看。

新建一个控制台工程，添加以下黑体部分代码：

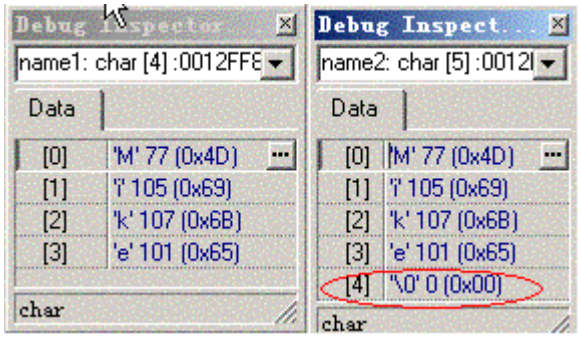
```
#include <iostream.h>
...
int main(int argc, char* argv[])
{
```

```
    char name1[] = {'M','i','k','e'};
    char name2[] = "Mike";

    cout << name1 << endl;
    cout << name2 << endl;

    return 0;
}
```

在其中的某行 cout << ... 上设置断点,然后运行，程序停在断点上，如下图：



将输入鼠标分别在移到代码中的 name1 和 name2 上，按 Ctrl 键，并用鼠标点击，将出现 Debug Inspector 窗口（调试查看器）。



可以清楚看到，name2 多了一个元素，并且值正是:'\0'。它就是编译器自动加入字符串结束标起：空字符。

第二、既然加了一个字符，那么，name[] 的大小就应该是 5，而非 4。为了加深记忆，我们用程序验证。

把上面的相关代码做点改动：

```
char name1[] = {'M','i','k','e'};
char name2[] = "Mike";

cout << sizeof(name1) << endl;
cout << sizeof(name2) << endl;
```

结果：

```
4
5
```

正如我们所想，第一种初始化方法，会让 name1 的尺寸多出一个字节，那个字节用于存入空字符。

由于引出一个重要的注意事项：**指定字符数组的大小时，请注意为空字符预留一个字节。**

下面是一个反例:

```
char name[4] = "Mike";
cout << name;
```

由于你强制限定 name 为 4 个元素，所以编译器爱莫能助，无法为它加上空字符作为结尾。当 cout 仍然无法正确地输出。

正确的是：

```
char name[5] = "Mike";
```

最不会犯错的不写：

```
char name[] = "Mike";
```

反过来，当采用 {'M','i','k','e'} 的形式初始化时，我们可以手工添加一个空字符：

```
char name [] = {'M','i','k','e', '\0'};
cout << name;
```

由于有了'\0'作为结束，所以 cout 可以正确地输出 name。

请回答，下面的代码运行后输出应为？

```
char name [] = {'M','i','k','\0','e'};
cout << name;
```

16.7 数组应用实例

我们将举几个例子，例子从简单到复杂。要求大家每个都照课程做一遍。我们不给出源代码。
以下例子均需要包含 #include <iostream.h>。若另需其它头文件，我们在例子中指出。

16.7.1 “小王成绩管理系统新版”

请自定义一个成绩数组(这回使用 float 类型），并求出其总分和平均分

```
//为了不重复太多内容，我这里直接用一个现成的数组
//你可以把它改回通过用户输入的方法得到成绩。
float cj[] = {85.5,90,100,70,60,45.5,89};

float pjcj,zcj = 0;

//成绩个数：
int cjCount = sizeof(cj) / sizeof(cj[0]);

for(int i=0;i< cjCount; i++)
{
   zcj += cj[i];
}
```

```
pjcj = zcj / cjCount;

cout << "总分:" << zcj << endl;
cout << "平均：" << pjcj << endl;

cin.get();
```

在作业中，我们要求大家继续完善本例，让它可以变得专业：
1、可以根据平均分，评判本次试卷的出题质量。（每年教委内部都要对高考试卷进行类似的评析）
2、学员输入自己的座号，可以输出他的成线和计算机对他的评价。
3、可以求最高分和最低分。

16.7.2 “!dnalroB evol I”

请自定义一字符数组，先将其输出，然后逆序输出（不包括空字符）

```
char str[] = "I love Borland!";

//正常输出：
cout << str << endl;

//字符串长度：
int len = sizeof(str) - 1; //问：为什么要减 1?

//逆序输出：
for(int i=len-1; i>=0; i--) //问：为什么 i 从 len-1 开始?
{
  cout << str[i];
}
```

这是输出结果：

请在完成本例后，继续尝试这一要求：
如果要求使用这样的 for 循环：

```
for(int i=0; i<len; i++)
{
   //?????
}
```

仍然要实现逆序输出,请问循环体的那行代码该如何写？这也是我们本章一道作业。

16.7.3 “数组中的玄机”

这个有点意思!

有一整型数组:

```
int flags [] = {
   0,0,0,0,1,1,3,4,6,5,3,3,5,5,6,4,3,2,2,-1,
   0,0,0,0,0,2,0,0,0,6,0,0,0,0,6,0,0,1,-1,
   0,0,0,0,7,0,0,0,1,10,0,13,0,10,2,0,0,0,8,-1,
   0,0,0,0,8,0,0,0,0,1,0,0,0,2,0,0,0,0,7,-1,
   0,0,0,0,13,4,6,9,9,13,4,14,4,13,9,9,6,4,13,-1,
   0,0,7,3,11,12,4,12,4,12,96,9,12,4,12,4,12,11,3,8,-1,
   0,0,8,13,2,2,0,0,0,0,4,12,4,0,0,0,0,1,1,13,7,-1,
   0,0,0,0,12,9,0,0,15,0,15,0,15,0,0,0,12,-1,
   0,0,0,0,4,12,9,0,0,16,17,18,19,0,9,12,4,-1,
   0,0,0,0,7,0,4,12,9,0,0,0,0,0,0,9,12,4,0,8,-1,
   0,0,0,13,8,0,0,8,4,12,9,0,9,12,4,7,0,0,7,13,-1,
   0,0,1,1,1,13,13,13,2,0,4,12,4,0,1,13,13,13,2,-1,
   };
```

请根据该数组中各元素的值依次做如下输出:

0 => 空格
1 => (
2 =>)
3 => ` 在数字键 1 的左边
4 => ’ 单引号
5 => ″ 双引号
6 => - 连字符
7 => /
8 => \
9 => .
10 => o 小写字母 o
11 => ; 分号
12 => #
13 => _ 下划线
14 => =
15 => ☆、16 => 我、17 => 爱、18 => 你
19 => !

注意除了 15,16,17,18 为汉字以外，其它均为英文状态下输入字符。而当输出汉字时，记得使用 ″″，因为一个汉字相当于个字符，所以应算字符串。另外：单引号、双引号、反斜杠 需要使用转义字符，详见下面代码。

看看输出结果是什么？其实这是我们小时候的游戏啦。一张看似乱七八糟的图，旁边写着：“请小朋友把图中带点的小格子涂上红颜色，看看图中隐藏着什么？”——呵呵，童年真好。快打开 CB，让我们一起回到童年。

这是代码：

```
int flags [] = {
   /* 请: 从上面拷贝数组元素的值*/
   };

int flagCount = sizeof(flags) / sizeof(flags[0]);

for(int i=0;i<flagCount;i++)
{
   switch(flags[i])
   {
      case 0 : cout << ' '; break;   //空格
      case 1 : cout << '('; break;
      case 2 : cout << ')'; break;
      case 3 : cout << '`'; break;   //数字键 1 左边的字符
      case 4 : cout << '\''; break;  //单引号
      case 5 : cout << '\"'; break;  //双引号
      case 6 : cout << '-'; break;   //连字符
      case 7 : cout << '/'; break;
      case 8 : cout << '\\'; break;  //反斜框
      case 9 : cout << '.'; break;
      case 10 : cout << 'o'; break;  //小写字母 o
      case 11 : cout << ';'; break;
      case 12 : cout << '#'; break;
      case 13 : cout << '_'; break;  //下划线
      case 14 : cout << '='; break;
      case 15 : cout << "☆"; break; //汉字字符注意要用双引号 ""
      case 16 : cout << "我"; break;
      case 17 : cout << "爱"; break;
      case 18 : cout << "你"; break;
      case 19 : cout << '!'; break;
      case -1 : cout << endl; break; //换行
   }
}
```

输出：略
(本例全部源代码请查看课程例子包)

虽说是嚷嚷着要回到童年，可以看到输出结果后，我怎么觉得它有些“少儿不宜”呢？如果你有女朋友，把这个程序寄给她吧。她一定会很开心。

如果不依靠数组事先定义好的图形数据，要纯粹使用流程控制来输出上面的图形，将会很困难。本例演示了“流程”+“数据”之间的配合，从面更直观地实现程序的目的。

16.7.4 “猜奖 Ver 1.0”

定义一含 10 元素的整型数组，通过随机函数，让每个元素值为随机的一个 2 位数（10~99）。然后要求用户输入一个数，如果该数在数组内，则打印出该数及其所在的数组下标，有几个打几个。 最后奖 10 个随机数输出到屏幕。

分析：简要地说，这是一个在数组中查找某数问题。

随机数用 random() 函数来实现，它的原型（或声明）为：

```
int random(int  num);
```

（所谓随机值，即值的大小不定，就像体育彩票开号时的“在气箱里飘浮的乒乓球”，或者麻将开局时往桌上一扔的那个家伙，都可以得出的一个事先不能确定大小的数 值）。

random 可以返回一个随机整数，这个整数的范围在 0 ~ num-1。

所以若有 int a = random(100); 则 a 是 0~99 之间的一个数。记住了，是 0~num-1，而不是 1~num。

题目要求是随机的一个 2 位数（10~99），所以应这么写： random(89)+ 10 。

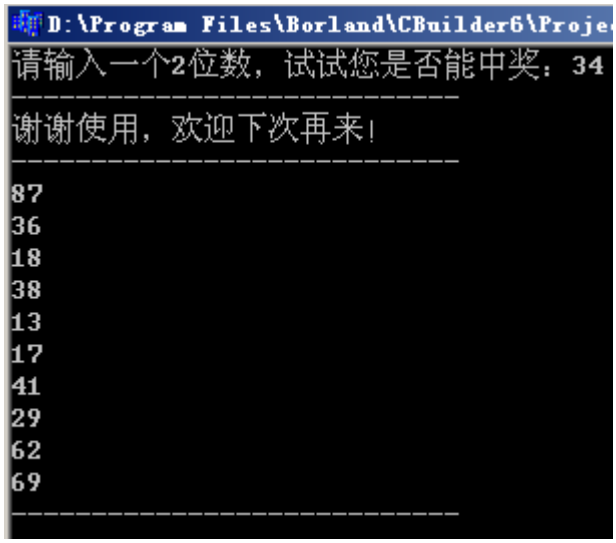
另外，为了让电脑能准确地得到随机数，必须在之前至少执行过一次另一个函数：随机种子函数：randomize();它没有参数没有返回值。只需拿来用就是。当然，这两个函数的声明包含在 stdlib.h 文件内。所以我们得包含这个头文件。

再一次碰到随机函数，我就不这么详细它们了。

```
#include <iostream.h> //怕你忘了，还是写上吧
#include <stdlib.h>
...
int main(int argc, char* argv[])
{
   #define MAX_NUM 10

   int num[MAX_NUM];
   //随机种子：
   randomize();

   //得到随机数：
   for(int i=0;i<MAX_NUM;i++)
      num[i] = random(89) + 10;

   //用户输入一个数：
   int a;
   cout << "请输入一个 2 位数，试试您是否能中奖：";
   cin >> a;

   //在随机数组中查找 a：
   int count = 0; //找到的次数
   for(int i=0;i<MAX_NUM;i++)
```

```
    {
        if(a == num[i])
        {
            count++;
            cout << i << "==>" << num[i] << endl;
        }
    }

    //输出一条横线，仅为美观:
    char line[] = "----------------------------";
    cout << line << endl;

    //找到 0 次，说明没有猜中：
    if(count == 0)
        cout << "谢谢使用，欢迎下次再来！" << endl;
    else
        cout << "恭喜！您中了" << count << "次" << endl;

    //最后显示所有随机数：
    cout << line << endl;
    for(int i=0; i< MAX_NUM; i++)
        cout << num[i] << endl;

    cout << line << endl;

    //由于前面有输入，所以需要两行 get();
    cin.get();
    cin.get();
    return 0;
}
```

这是我的一次运行结果：



运气不佳，我连猜 20 次也没中一次，本来想写完课程去买体彩的，想想还是算了，500 万可要比这个还难得多！这个例子我们就称为“猜奖 V1.0”吧，以后让我们慢慢升级到中奖概率和体彩一样的版本。

本题中奖的机率是多少呢？ 用户有 90 种数值可选（10~89），而每次可能中奖的数值为 10 个。所以中奖机率为 10 / 90 。计算百分比为： 10 * 100 / 90 = 11.11 %。

在作业中，我们布置一条题目，要求大家改变本例代码，使它可以验证中奖机率。

16.7.5 求最值

最值指最大值和最小值。我们可以写程序，经过一遍循环，同时求出这两个值。

为了不让事情一下复杂化，所我们先来求最大值，然后再加入求最小值的代码（这也是写程序中极其常见的一种技巧）。

我不写思路，请大家自已分析代码。

```
#define MAX_NUM 10
int num[MAX_NUM] = {45,34,56,78,12,56,70,67,2,76};


int maxValue; //最大值

maxValue = num[0]; //假设最大值是第一个元素。

//然后让它和第二个、第三个……一个个比下去：
for(int i = 1; i<MAX_NUM; i++)
{
   //如果后面有个元素，比当前“假设”的最大值还要大，那就把最大值换作是它:
   if ( num[i] > maxValue)
       maxValue = num[i];
}

cout << "最大值：" <<maxValue;

... ...
```

现在我们加入最小值的查找：

```
... ...
int maxValue,minValue; //最大值、最小值

maxValue = minValue = num[0]; //假设最大值和最小值都是第一个元素。

//然后让它们和第二个、第三个……一个个比下去：
for(int i = 1; i<MAX_NUM; i++)
{
```

```
    //如果后面有个元素，比当前临时的最大值还要大，那就把最大值换作是它:
    if (num[i] > maxValue)
        maxValue = num[i];

    //如果后面有个元素，比当前临时的最小值还要小，那就把最小值换作是它:
    if (num[i] < minValue)
        minValue = num[i];
}

cout << "最大值：" <<maxValue << endl;
cout << "最小值：" <<minValue << endl;

......
```

一点优化。

在 for() 循环中，每次都要作两个比较：1、比较 num[i] 是否比 maxValue 还大，1、比较 num[i] 是否比 minValue 还小。仔细一想：如果 num[i] 比 最大值还要大，那它就不可能比 最小值还小，对不？ 反过来说，如果 num[i]想比 minValue 还小，那么一定是在它不比 maxValue 大的情况下。

代码优化为：

```
.......
for(int i = 1; i<MAX_NUM; i++)
{
    if (num[i] > maxValue) //如果后面有个元素，比当前临时的最大值还要大，那就把最大值换作是它:
        maxValue = num[i];
    else if (num[i] < minValue) //如果后面有个元素，比当前临时的最小值还要小，那就把最小值换作是它:
        minValue = num[i];
}
.......
```

16.8 小结：我们又迈出了重要的一步

奢谈我们又如何迈出如何重要的一步是没有意义的。在小结本章之前，你应该把本章所有能上机演练的代码都调通一遍。其实这章的例子都有很意思，对不——当初我们学了“流程控制”时，我们说我们的例子会精彩一点，而当我们开始了“数组”的学习，我们的例子会更有意思。为什么？

有个说法：“程序 = 算法 + 数据结构”。我们可以把写程序理解为雕刻，要想有好作品，除了你的手艺要高以外，同样重要的是要有一个好的原材，如果没有，正所谓“朽木不可雕也”。在这个角度上看，世间所有创作皆同此理。

另外，我们也可以把“算法”称为“对动作的表达”；而把“数据结构”称为“对数据的表达”。

因此——说点题外话——当你想表达世间最真挚的感情时，最好作最本质的“动作”和“数据”表达。比如，真爱一个人，就对他说“我爱你”；而不要说成“我非常爱你”，除非同样的爱你还有好几份，只不过这一份是“非常的”，而另一份可能是“不那么非常的”，加了比较爱就俗了——更不要学腔“我好爱好爱你啊~~~~~”，这不仅“俗”，而且是“媚俗”。

所以程序员应该这样对他的爱人表达：“真爱的算法，只能用在惟一的数据结构上，那就是：你；来吧，让我们结婚吧，我相信我的算法加上你的数据结构，一定会演算出世间最美好的爱情……”

说点正经话。 类似： int i; 这样 C++基本的数据，谈不上有什么数据结构：只有一个数据，也就没有组合；没有组合，又何来结构呢？如果非要说是一种结构，那么就称它们为“单一结构”吧。当然，在 C++中，最小可处理的内存单位为字节。一个整型由 4 个字节组成，从这点上看似乎单一的整数变量也有结构，不过个这个结构其实由计算机内部决定并实现，不在我们操作范围。

而数组，则是我们接触到的第一种具有“复合结构”（相对于前面的“单一结构”）的数据。记住，在这里学习，你首先要把数组理解为一种“数据结构”，然后才能像其它教科书一样把它理解为“数据类型”。

不算“单一结构”。则数组是最简单的数据结构。它的特点是：

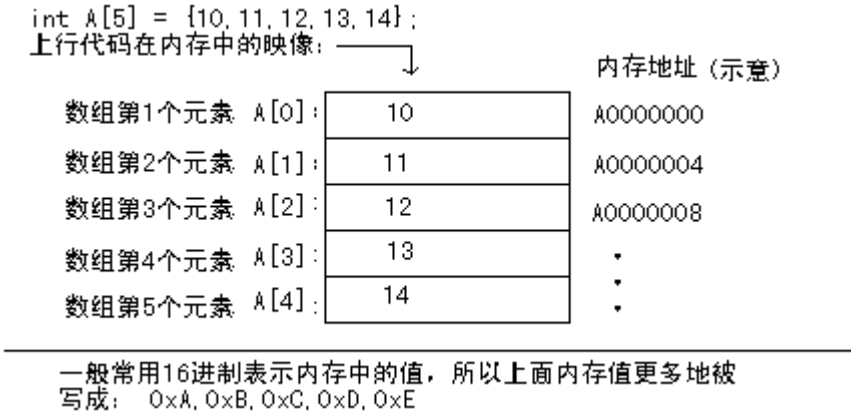
1、各元素在内存中的大小都一样，因为所有元素只能是同一种数据类型；

2、各元素在内存中整齐划一地连续排列。

正因为有这些特点，我们才可以方便通过数组的下标来操作数组中指定元素。下标从 0 开始，则下标为 n 的元素，就是数组中第 n+1 个元素。想想看，一群人排成一条整齐的队伍，我们就可以方便地说第 7 个是谁，第 100 个又是谁；如果这群人一盘散沙地凑在一起，那谁是第 7 个，谁又是第 100 个呢？

这就叫做：**“数据结构决定了对数据的存取方法”**。

先看看这个数组内存示意图吧，下一章我们会先专门讲数组在内存是如何排列的。



从图中你看出了数组中各元素在内存位置的关系了吗：1、大小一致；2、连续排列。比“队伍”更好联想是一排抽屉。

下一章我们在数组内存结构的基础上，讲字符串、二维数组、字符串数组等。

第十七章 数组（二）

17.1 数组与内存

变量需要占用内存空间，内存空间有地址。不同数据类型的变量，可能占用不同的内存大小及有不同的内存结构。

以前我们所学都称为“简单数据类型”，如：int,char,float,double,bool。像 char,bool,只占用一个字节，所以我们不去管它的的“结构”，其余如 int,float,double 占用多个字节，但比较简单，适当的时候我们会去探讨 4 个字节是如何组成一个整数。

后来我们学习了数组。数组变量占用内存的大小是不定的，因为不同的数组变量除了类型可以不同，还可以拥有不同个数的元素，这两点都影响它的大小。

因此，数组是我们第一个要着力研究它的结构的数据类型。和后面我们还要学习的更多数据类型相比，数组的结构还是相当简单的。简单就简单在它的各个元素大小一致，整整齐齐地排列。

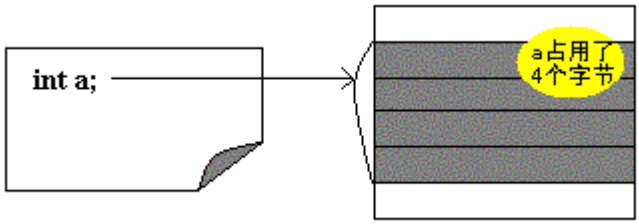
17.1.1 数组的内存结构

变量需要占用内存空间，内存空间有地址。

声明一个整型变量

```
int a;
```

系统会为该变量申请相应大小的空间，一个 int 类型的变量时，需要占用 4 个字节的空间，如下图：



也就是说，一个 int 类型的变量，它的内存结构就是 “4 个连续的字节”。

当我们声明一个数组：int arr[100];

我们可以想像，arr 数组在内存中占用了 100 * sizeof(int) 个字节。

现在请大家打开 Windows 的画笔程序，家画一个数组的内存结构示意图。

17.1.2 数组的内存地址

一个 int 类型变量，占用 4 个字节的内存，其中第一个字节的位置，我们称为该变量的内存地址。
同样，一个数组变量，占用一段连续的内存，其中第一个字节的位置，我们称为该数组变量的内存地址。

还记得 & 这个符号吗？通过它我们可以得到指定变量的内存地址。

```
int a;
cout << &a << endl;
```

& 称为“取址符”。如果你有点记不清，可以查看以前的课程。

本章第一个需要你特别注意的内容来了：
查看数组变量的地址，不需要使用 & 。下面的话是一个原因也是一个结论，你必须记住。

C，C++语言中，对数组变量的操作，就相当于直接对该数组变量的地址的操作。

因此，想要查看一个数组变量的地址，代码为：

```
int arr[10];
cout << arr << endl;  //注意，arr 之前无需 & 。
```

现在，请大家打开 CB,然后将上面代码写成完整的一个控制台程序，看看输出结果。

17.1.3 数组元素的内存地址

一个数组变量包含多个连续的元素，每一个元素都是一个普通变量。因此，对就像对待普通变量一样可以通过&来取得地址：

```
//查看数组中第一个元素的地址：
int arr[10];
cout << &arr[0] << endl;
```

例一：

现在，请大家在 CB 里继续上一小节的代码，要求：用一个 for 循环，输出数组 arr 中每一个元素的地址。如果你已完成，现在来看我的答案。

```
#include <iostream.h>
...
int arr[10];

for(int i=0; i<10; i++)
    cout << &arr[i] << endl;
...
cin.get();
```

我们把它和前面输出数组地址的例子结合起来，然后观察输出结果。

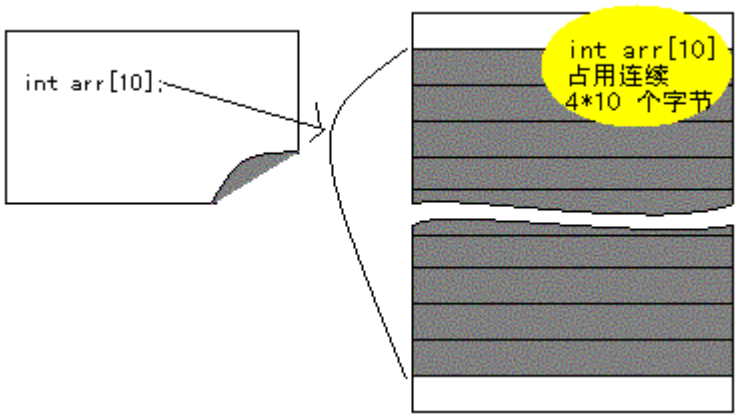
```
...
int arr[10];

//输出数组的地址:
cout << "数组 arr 的地址: " << arr << endl;

//输出每个元素的地址：
for(int i=0; i<10; i++)
    cout << "元素 arr[" <<i <<"]的地址：" << &arr[i] << endl;

...
```

输出结果：

第一个要注意的的是头两行告诉我们，整个数组变量 arr 的地址，和第一个元素 arr[0]，二者的地址完全一样。

事实上，数组和元素，是对同一段内存的两种不同的表达。把这一段内存当成一个整体变量，就是数组，把这段内存分成大小相同的许多小段，就是一个个数组元素。

请参看下图：



(分开一段段看是一个个元素，整体看称为一个数组,但二者对应的是同一段内存)

第二个要注意的，大家算算相邻的两个元素之间地址差多少？比如 &arr[1] - &arr[0] = 1245028 - 1245024 = 4 个字节。这 4 字节，就是每个数组元素的大小。当然，这里是 int 类型，所以是 4 字节，如果是一个 char 或 bool 类型的数组，则每个元素的大小是 1。

根据这两点，我来提几个问题：

1、如果知道某个 int 类型数组的地址是 1245024，请问下标为 5 的元素的地址是多少？
2、如果知道某个 char 类型的数组，其下标为 4 的元素地址为：1012349,请问下标为 2 的元素地址是多少？

由于可通过 sizeof() 操作来取得各类型数据的大小，所以我们可以假设有一数组：

```
T arr[N];   //类型为 T，元素个数为 N。
```

存在：

```
&arr[n] = arr + sizeof(T) * n ;  (0 <= n < N)
```

或者：

```
&arr[n] = arr + sizeof(arr[0]) * n;  (0 <= n < N)
```

17.1.4 数组访问越界

上一章我们说过“越界”。由于这一问题的重要性，我们需要专门再说一回。

越界？越谁的界？当然是内存。一个变量存放在内存里，你想读的是这个变量，结果却读过头了，很可能读到了另一个变量的头上。这就造成了越界。有点像你回家时，走过了头，一头撞入邻居家……后果自付。

数组这家伙，大小不定！所以，最容易让程序员走过头。

我们通过数组的下标来得到数组内指定索引的元素。这称作对数组的访问。

如果一个数组定义为有 n 个元素，那么，对这 n 个元素（0 到 n-1）的访问都合法，如果对这 n 个元素之外的访问，就是非法的，称为“越界”。

比如，定义一个数组：

```
int arr[10];
```

那么，我们可以访问 arr[0] ~ arr[9] 这 10 个元素。如果你把下标指定为 10 ，比如：

```
int a = arr[10]; //访问了第 11 个元素。
```

这就造成了数组访问越界。

访问越界会出现什么结果了？

首先，它**并不会**造成编译错误！ 就是说，C,C++的编译器并不判断和指出你的代码“访问越界”了。这将很可怕，也就是说一个明明是错误的东西，就这样“顺利”地通过了编译，就这样不知不觉地，一个 BUG，“埋伏”在你的程序里。

更可怕的是，数组访问越界在运行时，它的表现是不定的，有时似乎什么事也没有，程序一直运行（当然，某些错误结果已造成）；有时，则是程序一下子崩溃。

不要埋怨编译器不能事先发现这个错误，事实上从理论上编译过程就不可能发现这类错误。也不要认为：“我很聪明，我不会犯这种错误的，明明前面定义了 10 个元素，我不可能在后面写出访问第 11 个元素的代码！”。

请看下面的代码：

```
int arr[10];

for(int i=1; i<=10; i++)
{
   cout << arr[i];
}
```

它就越界了，你看出原因了吗？

再说上一章的成绩查询。我们让用户输入学生编号，然后查该学生的成绩。如果代码是这样：

```
int cj[100];
...

//让用户输入学生编号，设现实中学生编号由 1 开始：
cout << "请输入学生编号（在 1~100 之间）："
int i;
cin >> i;

//输出对应学生的成绩：
cout << cj[i-1];
```

这段代码看上去没有什么逻辑错误啊。可是，某些用户会造成它出错。听话的用户会乖乖地输入 1 到 100 之间数字。而调皮的用户呢？可能会输入 101，甚至是-1 —— 我向来就是这种用户 ——这样程序就会去尝试输出：cj[100] 或 cj[-2]。

解决方法是什么？这里有一个简单，只要多写几个字：

```
...
cout << "请输入学生编号（在 1~100 之间 如果不输入这个范围之内数，计算机将爆炸！）："
int i;
cin >> i;
...
```

系主任在使用你的这个程序时，十个指头一定在不停地颤抖……

理智的作法还是让我们程序员来负起这个责任吧，我们需要在输出时，做一个判断，发现用户输入了不在编号范围之内的数，则不输出。正确答案请看上章。

为什么数组访问越界会造成莫名其妙的错误？ 前面一节我们讲到数组占用了一段连续的内存空间。然后，我们可以通过指定数组下标来访问这块内存里的不同位置。因此，当你的下标过大时，访问到的内存，就不再是这个数组“份内”的内存。你访问的，将是其它变量的内存了。 前面不是说数组就像一排的宿舍吗？假设有 5 间，你住在第 2 间；如果你晚上喝多了，回来时进错了房间，只要你进的还是这 5 间，那倒不会有大事，可是若是你“越界”了。竟然一头撞入第 6 间……这第 6 间会是什么？很可能它是走廊的尽头，结果你一头掉下楼，这在生活中是不幸，可对于程序倒是好事了，因为错误很直接（类似直接死机），你很容易发现。可是，如果第 6 间是？？据我所知，第 6 间可能是小便处，也可能是女生宿舍。

17.2 二维数组

事实要开始变得复杂。

生活中，有很多事物，仅仅用一维数组，将无法恰当地被表示。还是说学生成绩管理吧。一个班级 30 个学员，你把他们编成 1 到 30 号，这很好。但现在有两个班级要管理怎么办？人家每个班级都自有自的编号，

比如一班学生编是 1~30；二班的学生也是 1~30。你说，不行，要进行计算机管理，你们两班学员的编号要混在一起，从 1 号编到 60 号。

另外一种情况，仍然只有一个班级 30 人。但这回他们站到了操场，他们要做广播体操，排成 5 行 6 列。这时所有老师都不管学员的编号了，老师会这样喊：“第 2 排第 4 个同学，就说你啦！踢错脚了！”。假设我们的校长大人要坐在校长室里，通过一个装有监视器的电脑查看全校学员做广播体操，这时，我们也需要一个多维数组。

17.2.1 二维数组基本语法

语法：定义一个二维数组。

数据类型　数组名[第二维大小][第一维大小];

举例：

```
int arr[5][6];  //注意，以分号结束。
```

这就是操场上那个“5 行 6 列的学生阵”。当然，哪个是行哪个列凭你的习惯。如果数人头时，喜欢一列一列地数，那你也可以当成它是“5 列 6 行”——台湾人好像有这怪僻——我们还是把它看成 5 行 6 列吧。

现在：

第一排第一个学员是哪个？答：arr[0][0]；
第二排第三个学员是？答：arr[1][2]；

也不并不困难，对不？惟一别扭的其实还是那个老问题：现实上很多东西都是从 1 开始计数，而在 C 里，总是要从 0 开始计数。

接下来，校长说，第一排的全体做得很好啊，他们的广播体操得分全部加上 5 分！程序如何写？答：

```
for(int col=0; col<6; col++)
{
   arr[0][col] += 5;
}
```

对了，这里我没有用 i 来作循环的增量，而是用 col。因为 col 在英语里表示“列”，这样更直观对不？下面要用到行，则用 row。

广播操做到“跳跃运动”了，校长大人在办公室蹦了两下，感觉自己青春依旧，大为开心，决定给所有学员都加 1 分，程序如何写？答：

```
for(int row = 0; row < 5; row++)
{
    for(int col = 0; col < 6; col++)
```

```
    {
        arr[row][col] += 1;
    }
}
```

看明白了吗？在二维数组，要确定一个元素，必须使用两个下标。

另外，这个例子也演示了如何遍历一个二维数组：使用双层循环。第一层循环让 row 从 0 到 4，用于遍历每一行；col 从 0 到 5，遍历每一行中的每一列。

(遍历：访问某一集合中的每一个元素的过程)

大家把这两个程序都实际试一试.

17.2.2 二维数组初始化

一维数组可以定义时初始化：

```
int arr[] = {0,1,2,3,4};
```

二维数组也可以：

```
int arr[5][6] =
{
     { 0,  1,  2,  3,  4,  5},
     {10,11,12,13,14,15},
     {20,21,22,23,24,25},
     {30,31,32,33,34,35},
     {40,41,42,43,44,45},
};  //注意，同样以分号结束
```

初始化二维数组使用了两层{},内层初始化第一维，每个内层之间用逗号分隔。

例二：

我们可以把这个数组通过双层循环输出：

```
for(int row = 0; row < 5; row++)
{
    for(int col = 0; col < 6; col++)
    {
        cout << arr[row][col] << endl;
    }
}
```

这段代码会把二维数组 arr 中的所有元素（5*6=30 个），一行一个地，一古脑地输出，并不适于我们了解它的二维结构。我们在输出上做些修饰：

```
for(int row = 0; row < 5; row++)
{
    cout << "第" << row + 1 << "行: "

    for(int col = 0; col < 6; col++)
    {
        cout << arr[row][col] << ","; //同一行的元素用逗号分开
    }

   cout << endl; //换行
}
```

请大家分别上机试验这两段代码，对比输出结果，明白二维数组中各元素次序。下面是完整程序中，后一段代码的输出：

```
按任意键继续...
第1行: 0,1,2,3,4,5,
第2行: 10,11,12,13,14,15,
第3行: 20,21,22,23,24,25,
第4行: 30,31,32,33,34,35,
第5行: 40,41,42,43,44,45,
```

现在说初始化时，如何省略指定二维数组的大小。

回忆一维数组的情况：

int arr[] = {0,1,2,3,4};

代码中没有明显地指出 arr 的大小，但编译器将根据我们对该数组初始化数据，倒推出该数组大小应为 5。

那么，二维数组是否也可以不指定大小呢？比如：

```
int arr[][] =
{
   {1,2,3},
   {4,5,6}
}; //ERROR!
```

答案是：对了一半……所以还是错，这样定义一个二维数组，编译器不会放过。正确的作法是：

必须指定第二维的大小，而可以不指定第二维的大小，如：

```
int arr[][3] =
{
   {1,2,3},
   {4,5,6}
};
```

编译器可以根据初始化元素的个数，及低维的尺寸，来推算出第二维大小应为：6 / 3 = 2。但是，很可

惜，你不能反过来这样：

```
int arr[2][] =
{
   {1,2,3},
   {4,5,6}
};  //ERROR! 不能不指定低维尺寸。
```

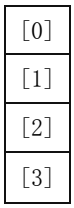
事实上，低维的花括号是写给人看的，只要指定低维的尺寸，编译器甚至允许你这么初始化一个二维数组：

```
int arr[][3] = {1,2,3,4,5,6};  //看上去像在初始一维数组？其实是二维的。
```

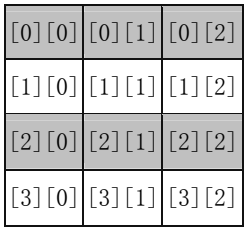
看上去像在初始一维数组？其实是二维的。 为什么可以这样？我们下面来说说二维数组的内存结构。

17.2.3 二维数组的内存结构

从逻辑上讲，一维数组像一个队列，二维数组像一个方阵，或平面：



一维数组是“一长串”



这是一个二维数组，4 行 3 列，像一个 4*3 的**平面**

一维数组的逻辑结构和它在内存里的实际位置相当一致的。但到了二维数组，我们应该想，在内存真的是排成一个“平面”吗？这不可能。内存是一种物理设备，它的地址排列是固定的线性结构，它不可能因为我们写程序中定义了一个二维数组，就把自己的某一段地址空间重新排成一个“平面”。后面我们还要学更高维数组，比如三维数组。三维数组的逻辑结构像一个立方体。你家里有“魔方”吗？拿出来看看，你就会明白内存更不可能把自己排出一个立方体。

结论是：内存必须仍然用直线的结构，来表达一个二维数组。

比如有一个二维数组：

```
char arr[3][2] =  //一个 3 行 2 列的二维数组。
{
   {'1','2'},
   {'3','4'},
   {'5','6'}
};
```

它的内存结构应为：

下标	内存地址 （仅用示意）	元素值
[0][0]	100001	'1'
[0][1]	100002	'2'
[1][0]	100003	'3'
[1][1]	100004	'4'
[2][0]	100005	'5'
[2][1]	100006	'6'

（二维数组 char arr[3][2] 的内存结构:每个元素之间的地址仍然连续）

也就是说，二维数组中的所有元素，存放在内存里时，它们的内存地址仍然是连续的。假如另有一个一维数组：

```
char arr[6] = {'1','2','3','4','5','6'}; //一维数组
```

这个一维数组的内存结构：

下标	内存地址 （仅用示意）	元素值
[0]	100001	'1'
[1]	100002	'2'
[2]	100003	'3'
[3]	100004	'4'
[4]	100005	'5'
[5]	100006	'6'

（一维数组 char arr[3][2] 的内存结构）

你猜到我想说什么了吗？请对比这两个表：一个有 2*3 或 3*2 的二维数组，和一个有 6 个元素的同类型一维数组，它们的内存结构完全一样。所以前面我们说如此定义并初始化一个二维数组：

```
int arr[][3] = {1,2,3,4,5,6};
```

也是正确的，只是对于程序员来说有些不直观，但编译器看到的都一样：都是那段同内存中的数据。不一样的是前面的语法。对于一维数组：

```
int arr[] = {1,2,3,4,5,6};
```

红色部分告诉编译器，这是一个一维数组。对于二维数组：

```
int arr[][3] = {1,2,3,4,5,6};
```

红色部分告诉编译器，这是一个二维数组，并且低维尺寸为 3 个，也就是要按每 3 个元素分出一行。C++的语法规定，编译器首先查看低维大小，所以我们若没有指明低维大小，则编译器立即报错，停止干活。因此，定义：

```
int arr[2][] = {1,2,3,4,5,6}; 是一种错误。
```

17.2.4 二维数组的内存地址

了解了二维数组的内存结构，我们再来说说几个关于二维数组地址问题，会有些绕，但并不难。嗯，先来

做一个智力测试。

以下图形中包含几个三角形?

正确答案是：3 个。想必没有人答不出。我们要说的是 ：这三个三角形中，两个小三角和一个大三角重叠着，因此若计算面积，则面积并非三个三角形的和，而是两个小三角或一个大三角的面积。

这个问题我们在一维数组时已经碰到过：一个数组本身可称为一个变量，而它包含的各个元素也都是一个个变量，但它们占用的内存是重叠的。

二维数组本身也是一个变量，并且也直接代表该数组的地址，我们要得到一个二维数组变量的地址，同样不需要取址符：&。

```
int arr[2][3] = {1,2,3,4,5,6};

//输出整个二维数组的地址。
cout << arr;
```

同样，我们也可以得到每个元素的地址，不过需要使用取址符：

```
//输出第一个元素（第 0 行第 0 列）的地址：
cout << &arr[0][0] << endl;
//输出第 2 行第 3 列的元素地址：
cout << &arr[1][2] << endl;
```

除此之外，我们还可以按“行”来输出元素地址,不需要使用取址符：

```
//输出第一行元素的起始地址：
cout << arr[0] << endl;
//输出第二行元素的起始地址：
cout << arr[1] << endl;
```

1、整个二维数组的开始地址 arr
2、第0行数组的开始地址： arr[0];
3、第1个元素的地址： &arr[0][0]

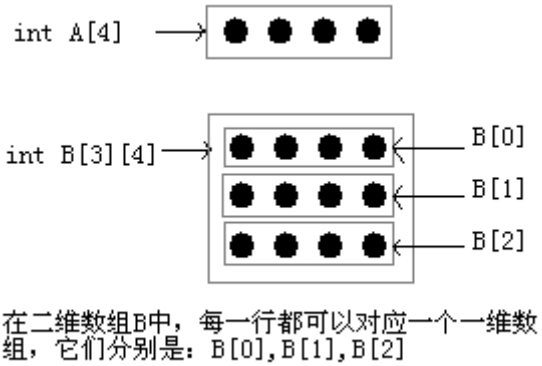
1、第1行数组的开始地址：arr[1]
2、元素 arr[1][0]的地址：

1、第2行数组的开始地址：arr[2]
2、元素 arr[2][0]的地址：

[0][0]	100001	1
[0][1]	100002	2
[1][0]	100003	3
[1][1]	100004	4
[2][0]	100005	5
[2][1]	100006	6

上图表明：arr, arr[0], &arr[0][0] 都指向了同一内存地址。即： arr == arr[0] == &arr[0][0]。
另外： arr[1] == &arr[1][0] 及 arr[2] == &arr[2][0]。

我们可以有这些推论： **二维数组中的每一行，相当于一个一维数组。**或者说，一维数组是由多个简单变量组成，而二维数组是由多个一维数组组成。
示意图：



（二维数组包含一维数组）

例子：
int arr[2][3];
则： arr[i][j] 相当于一个普通 int 变量。而 arr[i] 相当于一个一维数组。

现在，我还是来提问两个问题：

问题一：
有一数组 char arr[3][4];
已知 arr 中第一个元素(arr[0][0])的地址为：10000,请问 &arr[2][1] 的值为？

解答：先要知道 arr[1][1]是数组 arr 中的第几个元素？ 数组 arr 共 3 行，每行 4 列，而 arr[2][1]是位于第 3 行第 2 列，所以它是第： 2 * 4 + 2 = 10，即第 10 个元素。
这样就计算出来，第 1 个元素地址是 10000，则第 10 个元素地址： 10000 + (10 - 1) * sizeof(char) = 10009。

问题二：
如果上题中的数组为： int arr[3][4];其余不变，请问该如何计算？
答案：10000 + (10 - 1) * sizeof(int) = 10036。

17.3 二维数组实例

是不是前面的内容让你有些发晕。知识重在应用。我们还是来多操练几个二维数组的例子吧。但是，等用得多了，用得熟了，我希望大家回头再看前面的那些内容。

17.3.1 用二维数组做字模

例三：字模程序。

手机屏幕是如何显示英文字母或汉字的？这个小程序将要从原理上模拟这个过程。

手机屏幕采用的字体称为“点阵”字体。所以“点阵”，就是用一个个小点，通过“布阵”，组成一个字形。而这些点阵数据，就是一个二维数组中的元素。不同的手机，点阵的大小也不同。如果不支持中文，则最小只需 7*7；但若是要支持汉字，则应不小于 9*9，否则许多汉字会缺横少竖。采用大点阵字体，则手机屏幕要么是面积更大，要么是分辨率更高（同一面积内可以显示更多点）；并且手机的内部存储器也要更多。由于汉字数量众多，不像英文主要只有 26 个字母；所以支持汉字的手机，比只能显示英文字手机，其所需存储器自然要多出一个很大的数量级。

下面举例英文字母“A″的点阵，为了看的方便，我们用*来代替小黑点，并且打上了表格。我们使用最小的 7*7 点阵：

			*			
		*		*		
	*	*	*	*	*	
*						*

对于这样一个点阵，对应一个二维数组为：

```
int A[7][7] =
{
   {0,0,0,0,0,0,0},
   {0,0,0,1,0,0,0},
   {0,0,1,0,1,0,0},
   {0,1,1,1,1,1,0},
   {1,0,0,0,0,0,1},
   {0,0,0,0,0,0,0},
   {0,0,0,0,0,0,0},
};
```

程序要在屏幕上打出“A″时，则只需遍历该数组，然后在元素值为 0 的地方，打出空格，在元素值为 1 的地方，打出小点即可。当然，在我们的模拟程序里，我们打出星号。

所有这些数组，都需要事先写到手机的固定存储器中，这些数据就称为“字模”。

对于“A″的例子，打印时的代码如下：

```
for(int row = 0;row < 7; row++)
{
    for(int col = 0; col < 7; col++)
    {
```

```
            if(A[row][col] == 0)
               cout << ' ';
            else
               cout << '*';
        }

      //别忘了换行:
      cout << endl;
   }
```

结果如:

大家小时候有没刻过印章？哎！大概 80 年代出生的人是不会有过这种游戏了。印章分“阴文”和“阳文”。如果把上面的程序稍做修改，即在元素值为 0 的地方打出“*”，而在元素值为 1 的地方打出空格，那么输出结果就是“阴文”了。大家不妨试试。

例四： 躺着的“A”。

同样使用例三的二维数组数据:

```
int A[7][7] =
{
   {0,0,0,0,0,0,0},
   {0,0,0,1,0,0,0},
   {0,0,1,0,1,0,0},
   {0,1,1,1,1,1,0},
   {1,0,0,0,0,0,1},
   {0,0,0,0,0,0,0},
   {0,0,0,0,0,0,0},
};
```

请改动例三的程序,但不允许改变数组 A 的元素值，使之打印出一个躺着的“A”。下面是输出结果:

请大家想想如何实现，具体代码请见课程配套代码源文件。

17.3.2 二维数组在班级管理程序中应用

例五：多个班级的成绩管理

以前我们做过单个班级，或整个学校的成绩，那时都是将所有学生进行统一编号。事实上，不同的班级需要各自独立的编号。

比如初一年段有 4 个班级，每个班级最多 40 人。那么很直观地，该成绩数据对应于这样一个二维数组：

```
int cj[4][40];
```

在这里，数组的高维（第二维）和低维（第一维）具备了现实意义。例中，4 代表 4 个班级，40 代表每个班级中最多有 40 个学生。因此低维代表班级中学生的编号，高维代表班级的编号。这和现实的逻辑是对应的：现实中，我们也认为班级的编号应当比学员的编号高一级，对不？你向别人介绍说：“我是 2 班的 24 号”，而不是“我是 24 号的 2 班”。

一个可以管理多个班级的学生成绩的程序，涉及到方方面面，本例的重点仅在于：请掌握如何将现实中的具有高低维信息，用二维数组表达出来。并不是所有具有二维信息的现实数据，都需要明确地区分高低维，比如一个长方形，究竟”长“算”高维还是“宽”算高维？这就无所谓了。但另外一些二维数据，注意到它们的高低维区分，可以更直观地写出程序。

闲话少说，现在问，2 班 24 号的成绩是哪个？你应该回答： cj[1][23]; ——最后一次提醒，C++中的数组下标从 0 开始，无论是一维二维还是更多维的数组，所以 2 班 24 号对应的是下标是 1 和 23。

我们要实现以下管理：

1、录入成绩，用户输入班级编号，然后输入该班每个学员的成绩；
2、清空成绩，用户输入班级编号，程序将该班学员的成绩都置为 0；
3、输出成绩，用户输入班级编号，程序将该班学员成绩按每行 10 个，输出到屏幕；
4、查询成绩，用户输入班级编号和学员编号，程序在屏幕上打出该学员成绩。
5、统计成绩，用户输入班级编号，程序输出该班级合计成绩和平均成绩。
0、退出。

看上去，这是一个稍微大点的程序了。

我们已经学过函数，所以上面的四个功能都各用一个函数来实现。另外，“让用户输入班级编号”等动作，我们也分别写成独立的函数。

四个功能中，都需要对各班级学员成绩进行处理，所以我们定义一个全局的二维数组。

下面我们一步一步实现。

第一步：定义全局二维数组，加入基本的头文件。

第一步的操作结果应是这样(黑色部分为需要加入的代码)：

```
//---------------------------------------------------------------------------
//支持多班级的成绩管理系统
#pragma hdrstop
#include <iostream.h>
//---------------------------------------------------------------------------
```

```
#define CLASS_COUNT 4              //4 个班级
#define CLASS_STUDENT_COUNT 40     //每班最多 40 个学员

//定义一个全局二维数组变量，用于存储多班成绩：
int cj[CLASS_COUNT][CLASS_STUDENT_COUNT];  //提示：全局变量会被自动初始化为全 0。
                                           //所以一开始该 cj 数组中每个成绩均为 0。

#pragma argsused
int main(int argc, char* argv[])
{
   return 0;
}
//---------------------------------------------------------------------------
```

第二步：加入让用户选择功能的函数。

我们大致按从上而下方法来写这个程序。该程序首先要做的是让用户选择待执行功能。下面的函数实现这个界面。

```
//函数：提供一个界面，让用户选择功能：
//返回值：1~5，  待执行的功能，0：退出程序
int SelectFunc()
{
   int selected;

   do
   {
      cout << "请选择：(0~5)"   << endl;

      cout << "1、录入成绩" << endl
           << "2、清空成绩" << endl
           << "3、输出成绩" << endl
           << "4、查询成绩" << endl
           << "5、统计成绩" << endl
           << "0、退出" << endl;

     cin >> selected;
  }
  while(selected < 0 || selected > 5); //如果用户输入 0~5 范围之外的数字，则重复输入。

  return selected;
}
```

函数首先输入 1 到 5 项功能，及 0:用于退出。注意我们用了一个 do...while 循环，循环继续的条件是用户输入有误。do...while 流程用于这类目的，我们已经不是第一次了。

函数最后返回 selected 的值。
这个函数代码最好放在下面位置：

```
......
int cj[CLASS_COUNT][CLASS_STUDENT_COUNT];

/*
   <<<< 前面 SelectFunc（）函数的实现代码加在此处。
*/

#pragma argsused
int main(int argc, char* argv[])
......
```

为了验证一下我们关于该函数是否能正常工作，我们可以先把它在 main()函数内用一下：

```
int main(int argc, char* argv[])
{
    SelectFunc();
}
```

按 Ctrl + F9 编译，如果有误，只现在就参照本课程或源代码改正。如果一切顺利，则你会发现这个函数工作得很好。那么，删掉用于调试的：

```
SelectFunc();
```

这一行，我们开始下一个函数。

第三步：辅助函数：用户输入班级编号等。

“录入、清空、统计”成绩之间，都需要用户输入班级编号。而“查询成绩”则要求用户输入班级编号及学号，所以这一步我们来实现这两个函数。

```
//用户输入班级编号：
int SelectClass()
{
   int classNumber; //班级编号
   do
   {
        cout << "请输入班级编号：(1~4)";
        cin >> classNumber;
   }
    while(classNumber < 1 || classNumber > CLASS_COUNT);
   return classNumber - 1;
}
```

SelectClass 和 SelectFunc 中的流程完全一样。为了适应普通用户的习惯，我们让他们输入 1~4，而不是 0~3,所以最后需要将 classNumber 减 1 后再返回。

另外一个函数是用户在选择“成绩查询”时，我们需要他输入班级编号和学生学号。前面的 SelectClass() 已经实现班级选级，我们只需再写一个选级学号的函数 SelectStudent 即可。并且 SelectStudent 和 SelectClass 除了提示信息不一样，没什么不同的。我们不写在此。

第四步：录入、清空、查询、统计成绩功能的一一实现。

```
//录入成绩
//参数 classNumber: 班级编号
void InputScore(int classNumber)
{
   /*
      一个班级最多 40 个学员，但也可以少于 40 个，所以我们规定，当用户输入-1 时，表示已经输入完毕。
  */

  //判断 classNumber 是否在合法的范围内：
  if(classNumber < 0 || classNumber >= CLASS_COUNT)
     return;

  //提示字串：
  cout << "请输入" << classNumber + 1 << "班的学生成绩。" << endl;
  cout << "输入-1 表示结束。" << endl;

  for(int i=0; i < CLASS_STUDENT_COUNT; i++)
  {
     cout << "请输入" << i+1 << "号学员成绩：";
     cin >> cj[classNumber][i];   //cj 是全局变量，所以这里可以直接用。

     //判断是否为-1，若是，跳出循环：
     if( -1 == cj[classNumber][i])
         break;
  }
}
//-------------------------------------------------
//清空成绩：
void ClearScore(int classNumber)
{
  //判断 classNumber 是否在合法的范围内：
  if(classNumber < 0 || classNumber >= CLASS_COUNT)
     return;

  for(int i=0; i < CLASS_STUDENT_COUNT; i++)
  {
      cj[classNumber][i] = 0;
  }

  cout << classNumber + 1 << "班学生成绩清空完毕" << endl;
```

```
}
//-----------------------------------------------------
//输出成绩：
void OutputScore(int classNumber)
{
   //判断 classNumber 是否在合法的范围内：
   if(classNumber < 0 || classNumber >= CLASS_COUNT)
      return;

   cout << "============================" << endl;
   cout << classNumber + 1 << "班成绩" << endl;

   /*
       有两点要注意：
       1、要求每行输出 5 个成绩。
       2、每个班级并不一定是 40 个成绩，所以只要遇到-1，则停止输出。当然，如果该班
         成绩尚未录入，则输出的是 40 个 0。
   */

   for(int i = 0; i < CLASS_STUDENT_COUNT; i++)
   {
       if(i % 5 == 0)  //因为每行输出 5 个，所以 i 被 5 整除，表示是一新行
          cout << endl;

      if(-1 == cj[classNumber][i]) //遇到成绩为-1...
         break;

      cout << cj[classNumber][i] << ",";
   }
}
//-----------------------------------------------------
//查询成绩：
void FindScore(int classNumber, int studentNumber)
{
  //判断 classNumber 是否在合法的范围内：
  if(classNumber < 0 || classNumber >= CLASS_COUNT)
     return;

  //判断学生编号是否在合法范围内：
  if(studentNumber < 0 || studentNumber >= CLASS_STUDENT_COUNT)
     return;

  cout << classNumber + 1 << "班，" << studentNumber + 1 << "号成绩："<<
          cj[classNumber][studentNumber] << endl;
}
```

```
//-------------------------------------------------
//统计成绩：
void TotalScore(int classNumber)
{
  //判断 classNumber 是否在合法的范围内：
  if(classNumber < 0 || classNumber >= CLASS_COUNT)
     return;

  int totalScore = 0; //总分
  int scoreCount = 0; //个数

   //同样要注意遇到-1 结束。
   for(int i = 0; i < CLASS_STUDENT_COUNT; i++)
   {
       if(cj[classNumber][i] != -1)
       {
           totalScore += cj[classNumber][i];
           scoreCount++;
       }
       else
       {
          break;
       }
   }

   //还要注意，如果第一个成绩就是-1，则个数为 0，此时无法求平均值（因为除数不能为 0）
   if(scoreCount == 0)
   {
      cout << "该班学员个数为 0" << endl;
      return;
   }

   cout << "总分：" << totalScore << "平均：" << totalScore / scoreCount << endl;
}
```

第五步：完成主函数(总体流程)

在主函数内，将上面的主要函数放到合适的流程里。（仅从这一步看，我们的开发过程又有点像是“由下而上”法了：写好了各函数，最后组织起来。事实上，几乎所有大软件的开发，都是“由上而下”与“由下而上”结合）。

```
int main(int argc, char* argv[])
{
   int selected;
   do
   {
```

```
//用户选择要执行的功能：
selected = SelectFunc();

//如果选择 0，则退出：
if(selected == 0)
    break;

//根据 selected 来执行相应功能:
switch(selected)
{
    //1、输入成绩：
    case 1 :
    {
         int classNumber = SelectClass();
         InputScore(classNumber); //两行代码可以合为：InputScore(SelectClass());
         break;
    }
    //2、清空成绩：
    case 2 :
    {
         int classNumber = SelectClass();
         ClearScore(classNumber);
         break;
    }
    //3、输出成绩：
    case 3 :
    {
         int classNumber = SelectClass();
         OutputScore(classNumber);
    }
    //4、查询成绩：
    case 4 :
    {
         int classNumber = SelectClass();
         int studentNumber = SelectStudent();
         FindScore(classNumber,studentNumber);
         //以上三行也可合为：FindScore(SelectClass(),SelectStudent());
         break;
    }
    //5、统计成绩：
    case 5 :
    {
         int classNumber = SelectClass();
         TotalScore(classNumber);
         break;
```

```
            }
        }
    }
  while(true); //一直循环，直到前面用户输入 0 跳出。
}
```

一点题外话。代码中的注释也说明了，像：

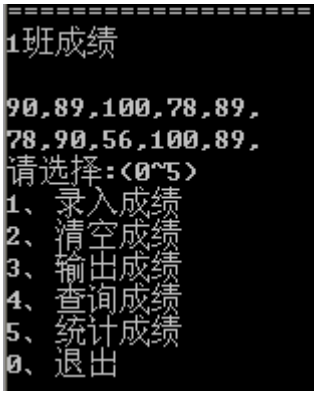
```
int classNumber = SelectClass();
int studentNumber = SelectStudent();
FindScore(classNumber,studentNumber);
```

在 C,或 C++里，可以直接用一行来表示：

```
FindScore(SelectClass(),SelectStudent());
```

这也是一个熟练的 C 或 C++程序员常做的事。大家现在就把这种写法写到例子中试试，并且理解。随着我们练习的代码量的不断增多，类似这样的很多 简洁的写法，我们都会用上，如果你不写，等我们一用上，你容易感到困惑。

OK! 似乎是突然来了一个大程序？把它调通吧，下面是我运行这个程序的输出界面：



仔细地想一想，我们至少还有一个重要功能没有实现，那就是排序。呵呵，关于排序，我们需要一整章来讲它。下面，还是说说如何数组的一些其它事情吧。如果你觉得有些累，就休息 30 分钟。

17.4 三维和更多维数组

一维和二维是最常用的数组。到了三维就用得少了。四维或更高维，几乎不使用。我们这里不多讲，仅举一些三维数组的实例。大家通过二维数组知识，就可看懂。

17.4.1 多维数组的定义与初始化

```
//单单定义一个三维数组：
int arr[3][4][2];

//如果是在定义的同时还初始化：
int arr[3][4][2] =
```

```
    {
    {
      {1,2},
      {3,4},
      {5,6},
      {7,8}
    },
    {
      {11,12},
      {13,14},
      {15,16},
      {17,18}
    },
    {
      {21,22},
      {23,24},
      {25,26},
      {27,28}
    }
    };
```

要看懂上面的初始化，关键在于找出:
哪里体现了最低维大小： 2?
哪里体现了第二维的大小：4?
哪里体现了最高维大小：3?
我加了彩色帮你寻找。

如果你看懂，那就这样吧。有一天我们需要使用三维数组。那时再说。我很相信你现在其实也会用一个三维数组，无非是：

```
cout << arr[2][1][0] << endl;
```

初始化时，可以不省略最高维的大小，其它低维的大小则必须指明。

```
int arr[][4][2] =
{
   ......
}
```

下面我举一个简单的例子。

17.4.2 多维数组的定义与初始化

没错，还是成绩管理。但我们仅要用一些代码来示意，让大家更实在地理解三维及更高维数组：
前面我们的成绩已经可以实现多个班级的同时管理。如果再进一步，你想实现多个年段的成绩管理怎么办？那就再来一维吧.

下面我们示例可以管初中三个年段的成绩管理：

```
#define GRADE_COUNT 3  //年段总数：3
#define CLASS_COUNT 4  //每个年段允许的最多班级数目
#define STUDENT_COUNT 40 //每个班级允许的最多学员人数
int cj [GRADE_COUNT][CLASS_COUNT][STUDENT_COUNT];
```

好！我们先插播一下说明。从这行定义，我们就应该学会高低维与现实数据的如何对应。看，在生活中，年段，班级，学号按层次分，正好是高、中、低；而三者在数组中也正是分别占用了高中低三维。这是很自然而然的做法。

现在，我们要想得到初三年段，2 班，20 号学员的成绩，如何办？

```
//让 a 为初三年段，2 班，20 号学员的成绩：
int a = cj[2][1][19];
修改呢？
cj[2][1][19] = 78;
```

取得与设置三维数组元素的操作，就是这样而已。如果想清空每个成绩，则循环相应地变成三层：

```
//nd : 年段, bj : 班级,  xh : 学号
for(int nd = 0; nd < GRADE_COUNT; nd++)
{
    for(int bj = 0; bj < CLASS_COUNT; bj++)
    {
        for(int xh = 0; xh < STUDENT_COUNT; xh++)
        {
            cj[nd][bj][xh] = 0;
        }
    }
}
```

哈哈，以我们现在才能，当初年段长投向我们的目光已经不算什么了，校长大人完全应该让我们当个教务长啊。你真的很想？那就试试把前面的那个“二维版”的成绩系统改写为“三维”版？？

改还是不改？要改可真累！算了，把初一初二初三的成绩混在一个数组里管理，其实是一个很糟的做法，并不实用，对不？。我们这里只是想让大家看到三维数组可以解决什么样的问题。

再高一维的呢？好，还是成绩管理系统。你以为我这回想做一个“跨校”的成绩管理？当然不是。一个学员只有一个成绩吗？不是啊。我们再定义一个学员可以有最多 6 个成绩：

```
......
#define SCORE_COUNT 6
int cj [GRADE_COUNT][CLASS_COUNT][STUDENT_COUNT][SCORE_COUNT];
......
```

17.5 数组作为函数的参数

要学习这一章，首先确保你没有忘记“函数的参数”是什么？如果您有些模糊，就先复习函数的两章。

17.5.1 数组参数默认是传址方式

数组作为函数的参数，难点和重点都在于这两点：

1、理解函数参数两种传递方式：传值、传址之间区别。

2、数组变量本身就是内存地址。

这两点我们都已讲过，但此时是我们复习——或者说进一步理解这两点时候。

现来说第一点：传值、传址的区别。如果你连什么叫“函数参数”都没有印象，那你现在需要的不是复习，而是“补习”。请回头看第 13 章。

现在我再举一个例子，来解释当把一个参数传给函数时，使用“传值”方式和使用“传址”有何区别：

首先假设科技发达，可以通过克隆复制出一个和你一模一样的人。

接着是个美妙的故事：有个阿拉伯公主将要嫁给你。

再来就是两种情况。

第一种情况：

先把你克隆，然后公主嫁给“你”（那个复制品）。

在这种情况下，请问：当公主和复制的人深情相吻时，不知你有何感觉？当然是没有什么感觉，尽管那个复制品和你长得一模一样，但是他的一切行为都和你无关，若一天他不幸被惹恼了国王，被砍头，不要紧，你还活着。

这种情况对应的是函数参数传递方式的第一种“传值”:传的是一个复制品，虽然值完全一样，但并不是实参本身。

第二情况：

被送到阿拉伯王宫的人就是你本人！这回，嘿嘿，和公主亲密相吻的感觉你可尽情享受，但若被砍头，则在这世界上消失的也是你。

这就是“传址”的情况：传给函数的是实参的内存地址，我们知道，变量在计算机里就是一个内存的地址，反过来，传一个内存地址，也就起到传送实参变量本身的作用。

一句话：传值方式下，传的只是实参的复制品（值一样）；传址方式下，传的是实参本身。

接下来，回顾一个简单变量的作为参数的例子，同时也是检查你是否理解第一点的时候了。

```
void Func1(int a)
{
   a = 100;
}
//------------------------
void Func2(int& a)
{
   a = 200;
}
//------------------------
int main(int argc, char* argv[])
```

```
{
   int c = 0;

   Func1(c);
   cout << c << endl;

   Func2(c);
   cout << c << endl;
}
```

请向上面 main()运行以后，屏幕上输出的两个数是多少？请先回答该问题，然后上机实验。如果您答错了，或者你知道自己只是“蒙”对了，你需要去复读第 12、13 章讲函数的内容。

从上面的代码中我们也看到了，“传值”方式下，函数的形参没有“&”；“传址”的方式下，形参前有一个“&”。这是二者语法上的区别。但是在下面，这将会有一点变化。

有关第 1 点的新知识来了：在 C,C++中，如果函数的参数是数组，则该参数固定为传址方式。

例：

```
void Func(int arr[5])
{
  ...
}
```

Func 函数的参数是： int arr[5]。 这是第一次接触使用数组作为参数。它表示在调用 Func 时，需要给这个函数一个大小为 5 的整型数组。

在这个参数里，我们没有看到“&”。似乎这应该是一个“传值”方式的参数，但错了，对数组作为参数，则固定是以传址方式将数组本身传给函数，而不是传数组的复制品。

为什么要有这样一个例外？首先是出于效率方面的考虑。复制数组需要的时间可能和复制一个简单变量没有区别：比如这个复制就只有一个元素： int arr[1]; 但如果这个数组是 1000 个，或 50000 个元素，则需要较长的时间，对于 C,C++这门追求高效的语言，太不合算。

接着从第二点上说：“数组本身就是内存地址”，也正好说明了这一点，数组作为函数的参数，传的是“地址”，并且不需要加‘&’符号来标明它是一个传址方式的参数，因为，“数组本身就是内存地址”。

请看下面的举例：

```
void Func(int arr[5])
{
   for(int i=0;i<5;i++)
     arr[i] = i;
}

int main(int argc, char* argv[])
{
   int a[5];
   Func(a);
   for(int i=0; i<5;i++)
    count << a[i] << ',';
```

```
}
```

输出将是 “0,1,2,3,4,”。这证明数组 a 传给 Func 之后，的的确确被 Func 在函数内部修改了，并且改的是 a 本身，而不是 a 的复制品。

17.5.2 可以不指定元素个数

我们定义一个数组变量时，需要告诉编译器该数组的大小（直接或间接地指定）。但在声明一个函数的数组参数时，可以不指定大小。

在
声明一个函数时：

```
void Func(int arr[]);
```

及在定义它时：

```
void Func(int arr[])
{
  ...
}
```

上面中的参数：int arr[]。没有指定数组 arr 的大小。这样做的好处是该函数原来只能处理大小固定是 5 的数组，现在则可以处理任意大小的整型数组。

当然，对于一个不知大小的数组，我们处理起来会胆战心惊，因为一不小心就会越界。一般的做法是再加一个参数，用于在运行时指定该数组的实际大小：

```
void Func(int arr[], int size)
{
    for(int i=0;i<size;i++)
       arr[i] = i;
}
```

现在这个函数可以处理任意大小的数组，很方便。

```
int a[5],b[10],c[100];

Func(a,5);
Func(b,10);
Func(c,100);
```

你还可以根据需要，指定一个比数组实际大小要小的 size 值。比如我们只想让 Func 函数处理 c 数组中的前 50 个元素：

```
Func(c,50);
```

说完“数组参数可以不指定大小”这一规定的好处，我们再来说这一规定的技术原理。其实说这是一项“规定”，其实说法不合理。只有那些解释性的语言（如 BAISC)才会有各种规定，对于 C++这样一门既灵活又严谨的，纯编译型的语言，当它的语法规定下来后，就会自然而然地产生一些特性——是语言自身实现的特性，

而不是人为规定。

前面说数组做为函数参数，使用的是“传址”方式。由于传递的是数组的地址，而不是数组的所有元素，所以函数可以不知道该数组的实际大小。

假设有这么一些代码片段：

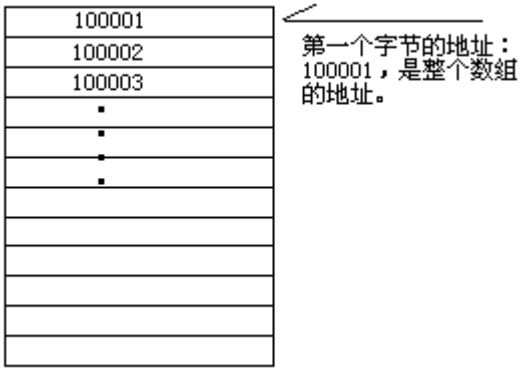
```
void Func(int arr[])
{
    ....
};

...
int a[3];
Func(a);
```

前面是函数 Func 的实现，并没有执行动作。我们来看后面两句。

代码　　说明　　内存示意图

```
int a[5];
```

在内存里申请了一个大小为 3 的整型数组：（字节数： 3 * sizeof(int) = 12）



（10001...表示该字节内存的地址）

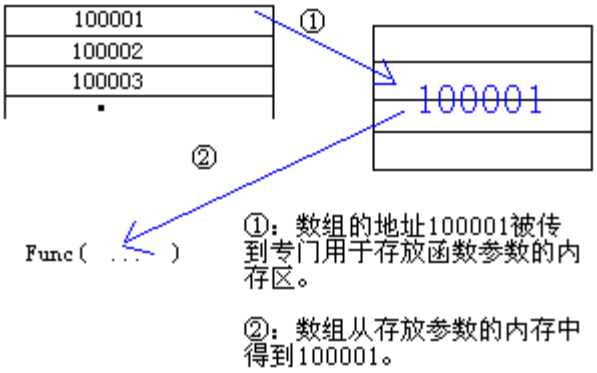
接下来，程序以数组 a 作为参数。调用函数：Func。

由于是“传址方式”，所以函数 Func 其实只得到了一个地址值：10001，至于这 10001 后面跟了多少个字节，或跟了多少个整型元素？Func 无从得知，既然不知，也就无法做出限制。所以，换一段代码，你传给它一个大小数 300 的函数，它也能接受。

代码　　说明　　内存示意图

```
Func(a)
```

以数组 a 为参数，调用函数 Func()：



17.5.3 数组作为函数参数的上机实例

在“支持多个班级的成绩管理系统”那个例子，我们写了不少函数，但没有哪一个函数用到数组参数。

这是因为，作为程序中要用到的惟一一个数组数据：； int cj[CLASS_COUNT][CLASS_STUDENT_COUNT]，它被我们定义为一**全局**变量。全局变量不在任意函数内，所以不专属哪个函数，所有的函数都可以在函数中使用，所以我们没有必要通过参数来传递。

“成绩管理系统”的第二个功能：“清空成绩”，是一个将某一数组中的所有元素清 0 的过程。我们针对这一功能，首先，让我们自己写一个相应的函数。

```
//函数：将一个整型数组中的指定元素值全部清 0：
void ZeroIntegerArray(int arr[],int size)
{
   for(int i=0;i<size;i++)
       arr[i] = 0;
}
```

解释一下函数名字：Zero:归 0，Integer：整数，Array：数组。

接着，我们让这个函数用在上面的“成绩管理”中的“清空成绩”。
原来的清空功能是这样实现的：

```
//-----------------------------------------------------
//清空成绩：
void ClearScore(int classNumber)
{
  //判断 classNumber 是否在合法的范围内：
  if(classNumber < 0 || classNumber >= CLASS_COUNT)
     return;

  for(int i=0; i < CLASS_STUDENT_COUNT; i++)
  {
      cj[classNumber][i] = 0;
  }

  cout << classNumber + 1 << "班学生成绩清空完毕" << endl;
}
//-----------------------------------------------------
```

你可能发现了，被清空的数组是一个二维数组：cj[][]，而我们的 ZeroIntegerArray(int arr[] ...)需要的参数是一维数组。

其实，在 ClearScore()函数中，被清空的只是指定那个班级的所学员成绩，而不是所有班级的所有学员成绩。我们说过，二维数组是由多个一维数组组成。请理解以下代码：

```
//清空成绩：
void ClearScore(int classNumber)
{
  //判断 classNumber 是否在合法的范围内：
```

```
    if(classNumber < 0 || classNumber >= CLASS_COUNT)
        return;

    ZeroIntegerArray(cj[classNumber],CLASS_STUDENT_COUNT);

    cout << classNumber + 1 << "班学生成绩清空完毕" << endl;
}
//-----------------------------------------------------
```

例子中的，ZeroIntegerArray()数组，清的是一个二维数组中的某一行，实参是：cj[classNumber]，如果对它有疑问，请回头看本章“二维数组包含一维数组”的图及相关内容。

17.5.4 二维及更多维数组作为函数参数

函数参数也可以是二维及及更高维数组。但必须指定除最高维以后的各维大小。这一点和初始化时，可以省略不写最高维大小的规则一致：

```
//定义一个使用二维数组作为参数
void Func(int arr[][5])  //第二维的大小可以不指定
{
  ...
}

//定义一个使用三维数组作为参数
void Func(int arr[][2][5])  //第三维的大小可以不指定
{
  ...
}
```

17.5.5 函数的返回值类型不能是数组

最后特别指出一点：函数的返回值不能是数组类型：

```
int[5] Func();  //ERROR!
```

本意是想让函数返回一个大小为 5 的数组，但实际语法行不通。

由于数组作为参数时，使用的是传址方式，所以一个数组参数，可以直接得到它在函数内被修改的结果，无须通过函数返回。另外，后面我们将学习“指针”，则通过返回指针来达到返回数组的同等功能。

17.6 sizeof 用在数组上

还记得 sizeof 吧？它可以求一个变量某数据类型占用多少个字节。比如，sizeof(int)得到 4，因为 int 类型占用 4 个字节。或者：

```
int i;
char c;
```

```
cout << sizeof(i) << "," << sizeof(c);
```

将输出 4 和 1。

sizeof 用在数组上，有两个要点：

1、可以用通过它来计算一个数组的元素个数。

2、当数组是函数的参数时，sizeof 对它不可用。

17.6.1 用 sizeof 自动计算元素个数

sizeof 对于一个数组变量，则会得到整个数组所有元素占用字节之和：

```
int arr[5];

cout << sizeof(arr);
```

屏幕上显示：20。因为 5 个元素，每个元素都是 int 类型，各占用 4 字节，总和：4 * 5 = 20。

由这个可以算出某一数组含多少个元素：sizeof(arr) / sizeof(arr[0]) = 20 / 4 = 5。从而得知 arr 数组中有 5 个元素。

这些我们都已经知道，但下面的事情有些例外。

17.6.2 sizeof 对数组参数不可用

对于一个从函数参数传递过来的数组，sizeof 无法得到该数组实际占用字节数。

这句话什么意思呢？看看这个例子：

```
void Func(int arr[])
{
   cout << sizeof(arr) << endl;
}

int main(int argc, char* argv[])
{
    int b[20];

    Func(b);

    system("pause");

    return 0;
}
```

程序将输出：4，而实际上，b 占用的字节数应为：4 * 20 = 80;大家可以在调用 Func(b)之前加一行：

```
cout << sizeof(b) << endl;
Func(b);
```

这样更可以发现，sizeof 直接对 **b** 取值，得到的是正确的大小，而将 b 传给 Func 后，在 Func 内部对该参数取值，则只能得到:4。

为什么？

这正是我们前面说的，数组作为参数传给函数时，采用的是传址方式，固定只送了数组的地址过去，而用于存放数组的地址，仅需 4 个字节即可。再打比方吧，如果你想你现在居住的三房二厅的房子送给我，你有两种办法：第一是把整个房子用倚天屠龙刀从楼幢里切出，然后买一巨大的信封（意指占用很大内存），寄给我（意指将整个实际数组传给函数）；另一种方法是将你的家钥匙用个小信封寄过来。

整来整去，我们一直在复习函数参数的内容啊？不过有什么办法呢？当涉及到把数组作为参数，就不得不直面对“传址”的理解。归结为一首儿歌吧：

“传值”传复制
“传址”传地址
数组当参数
固定传地址

这一章到此结束，内容很多，在老师不讲课的期间，你应该如何做才能学透本章呢？

请自己给自己出一些“一维或二维数组”的题目，如何出不了，就做作业吧。

第十八章 数组（三） ---- 数组的最值与排序

什么叫程序？随着我们学习的不断进展，这个问题的答案不断有新的表述。

今天，我们学过了“流程”，也学过了“数据类型”。

“流程”表达某种动作或操作的过程；“数据”表达现实生活的事物。因此，程序自然可以表达为“通过流程控制，来对数据进行正确的处理”。其实这一句话，也可以用两个字来代替“算法”。

事实上有一个著名的公式，说：程序 = 数据结构 + 算法。

要想真正理解什么叫算法，最好的办法还是从我们的现实生活入手。

最常见的例子，就是给整理扑克牌了。给你一付打乱的扑克牌，然后让你把它们整理，就是让你排序。结果是：前四张是：黑桃 A，红心 A，草花 A、方块 A，然后是 2，3……老 K，最后是大小王两张。

这个过程使用的是“排序”算法。

更简单的，给你 3 张牌，让你找出其中最大的一张，这也需要一种算法。称为“求最值”。

你会说，这也算“算法”，3 张牌往桌子上一摆，我“一眼”就能找出哪一张最大啊，我的大脑好像没有进行过任何计算。呵呵，这样说可就不对了。你把这三张牌往一头猪前面摆，摆上三年它也找不出哪一张是最大的。这可以证明，我们的大脑的确进行了一定的演算。

一套相同的算法，其实是连续的一段“流程控制”。可以用在不同的数据上。比如排序算法，我们可以用于整理扑克，也可以用于排出学员成绩的名次，而不这两样数据的数据结构是什么。但是一套算法在实现时，针对不同数据结构，有不同的实现。

这一章主要就是讲两种算法在数组上的实现，这两种算法是:“求最值”、“排序”。

18.1 求数组中的最大值

数组含有许多元素，这些元素如果是可以比较大小的，那就常常需要一种计算，求出这些元素中的最大值或最小值。求最值的算法应用在方方面面，比如：如何找出一条街上你喜欢的那某裙子最便宜卖的那家店。比如当早上第四节下课铃敲响后，如何找出从教室到食堂最近的一条路等等。

18.1.1 基本思路与实现

我想大家都知道了，一到要讲实例，我举的例子就是“成绩管理”。“烦不烦呢？”我看到有些同学使劲

撇嘴。可不能烦啊，上一章的成绩管理中，“求成绩第一名”和“成绩排序”这样重要的功能还没实现呢。本章的作业就是它们了。

比如有这么一个数组，用于存储几个学生成绩。现在老师想找出其中的第一名。

```
int cj[] = {80,67,76,87,78};
```

我们还是一眼“找”出了结果：87。但如果不是 5 个成绩，而是 5 万个成绩呢（比如首钢的工人进行考试的结果）？我们就不能一眼看出，而是不断地从一个个成绩里搜寻那个最大值。不管是 5 万还是 5 个，其实算法是一样的。

冰心老奶奶举了个例子：同样是从动物园回来，有的小学生写出让你如临其境的作文，而有的小学生则像是没有去过动物园一样，写得干巴巴的。

在把你的解决问题的思路转化为程序代码的过程中，显然第一步应该做是你能够用自然语言清楚地，准确地表达出你的思路。有些人能做好这一点，而有些人则表达得相当困难，仿佛他不会解决问题。

当然这是一个双向锻炼的过程，如果你原来在这方面不擅长，跟着我在这里学习编程，慢慢的你会发现自己不仅学会也写程序，而且学会了如何表达自己的想法、思路、情感……很多人说学习编程是一件快乐的事，很多人沉迷于编程，其中的一点奥妙，他们都不肯“泄密”，我泄密了。

言归正传。大家提起精神来！

求最大值是一个“比较”的过程。我们就说 5 个数的情况，看看如何找出 5 个数中的最大值：

2、3、1、4、0

为了方便表达，我们用 N 来表示最大值。

1、首先假设第一个数就是最大值，则 N = 2;
2、把 N 和第二个数比较，发现 3 比 N 大，于是让 N = 3;
3、把 N 和第三个数比较，发现 1 不比 N 大，于是 N 不变。
4、把 N 和第四个数比较，发现 4 比 N 大，于是让 N = 4;
5、把 N 和第五个数比较，发现 0 不比 N 大，于是 N 不变;

求五个数的最大值，我们用了五行话表达，如果求 100 个数的最值呢？要比较 99 次，岂不是要写 100 行？按照它的表达，我们写成的代码是：

```
int n[5] = {2,3,1,4,0};
int N = n[0];
if(N > n[1])
  N = n[1];
if(N > n[2])
  N = n[2];
if(N > n[3])
  N = n[3];
if(N > n[4])
  N = n[4];
```

这可不叫“算法”。所以前面的表达并没有说出真正的算法。我们要改进它。

1、首先假设第一个数就是最大值，则 N = 2;
2、把 N 和下一个数比较，如果下一个数比 N 大，则让 N 等于该数;
3、重复第二步，直到没有下一个数。

明白了吗？算法就是这样而来的。第一，这三行话可以适用于无论多少个数求最大值的情况，这是你的算法是否正确的一个必要条件，如果你的算法表达的长短依赖于具体数据的个数，那么你的算法不是通用的算法，不管是否能解决问题。第二，我们在表达中看到了“如果”，看到“重复”，很好，“如果”就是“分支流程”，就是 if 或 switch；而“重复”就是“循环流程”，是 for 或 while 或 do...while。

```
int n[5] = {2,3,1,4,0};
int N = n[0];

for( int i = 1; i < 5; i++)
{
   if(n[i] > N)
     N = n[i];
}
```

循环从数组下标 1 开始，因为从算法的表述中，我们也看到了，N 一开始就等于数组中的第一个数，而后和“下一个数”开始比较。
我们可以把代码改良，以让它方便于应用在任何个数的元素上。

```
int n[] = {2,3,1,4,0};

int N = n[0];

int count = sizeof(n) / sizeof(n[0]);

for( int i = 1; i < count; i++)
{
   if (n[i] > N)
     N = n[i];
}
```

18.1.2 实例

要求： 1、不使用数组，实现让用户输入 10 个数，然后输出其中最大值。
2、同 1，但要求使用数组。

既然是两个小题，我们就分别写两个函数吧。

```
//不使用数组的例子:
void max1()
{
  cout << "请输入 10 个数（每个数输入后加回车)" << endl;
```

```
  int N,n;

  cout << "第 1 个数：" :
  cin >> N;

  for(int i = 1; i<10; i++)
  {
      cout << "第" << i+1 << "个数:" ;
      cin >> n;

      if( n > N)
         N = n;
  }

  cout << "最大值为：" << N << endl;

  system("PAUSE");  //让控制台系统暂停。相当于我们以前的 cin.get()或 getchar();
}

//使用数组的例子：
void max2()
{
   cout << "请输入 10 个数（每个数输入后加回车)" << endl;

   int n[10];
   int N;

   for(int i = 0; i<10; i++)
   {
      cout << "第" << i+1 << "个数：";
      cin >> n[i];
   }

   N = n[0];
   for(int i=1; i<10; i++)
   {
     if(n[i] > N)
       N = n[i];
   }

  cout << "最大值为：" << N << endl;

  system("PAUSE");  //让控制台系统暂停。
}
```

这样就完成了求最大值实例，如果是要求求最小值呢？改动仅在于那个 if 判断条件：

```
……
N = n[0]; //一开始假设第一个元素就是最小值

for(……)
{
  if (n[i] < N) //如果有元素比我们假设的最小值还小，那就让最小值等于它吧
      N = n[i];
}
……
```

这套题目我没有提供实际代码，大家找开 CB 自己完成吧。重要的是，在调通程序之后，认真地比较两种处理方法之间的异同。

结论应该是：“算法的抽象逻辑是一样的，只是用在于不同的数据结构上，会有不同的实现”。前者只使用简单的数据类型，所以它不得不在一边输入的情况下，一边求最大值；而后者采用了数组，所以可以从容地先完成输入工作，然后再求最大值。

当算法经较复杂时，采用良好的数据结构的重要性就开始体现，比如下面的排序，我们必须使用数组或其它更复杂的数据。否则就实现不了。

18.2 将数组元素排序

排序，一个经典教学课程。

排序，一个在超高频的实用算法。

第一点是说，我们必须去学。第二点是说，像这样一个实用算法以，事实上 C,C++肯定都为我们写好了，以库函数等形式提供给我们使用，而且，这些写好的代码，肯定是最优秀的实现。

可是我们还是要学，而且是从最笨“冒泡算法”学起。所谓的最笨，是指效率差的。

学习的原因：1、前面说了，为了锻炼我们的逻辑思维。2、为了在某些时候，我们可以对排过程做更多的控制。

18.2.1 现实算法与程序算法的不同

大家都是这么整理扑克牌：把 54 张摊开放在桌面，然后不断地调整各张牌的位置，并把已经有序的牌放到另外一个位置。

生活中的各种算法一般不用考虑“内存”的问题。比如上面的问题，54 牌每一张都要占用一点桌面，这算是固定需要的内存，而在“腾挪”各张牌，使之渐渐变得有序的过程中，还需要开辟新的空间，包括手里抓着的牌，即手心也算是一个内存。

程序排序，要求既要占用内存少，又要速度快。这是衡量一个算法是否优秀的两个基本点。

若是应用到人整理牌这一例子，则除了实现将 54 张牌按次序（牌值和牌花）排好以外，还需另有要求：

1、除了 54 张牌一开始占用的桌面，及你的一个手心以外，你在整理的过程中，不能让牌再占用新的桌面空间。

2、要求“比较两张牌大小”“交换两张的位置”等过程都尽量地少。

你可以拿出家里的扑克牌，现在就开始按上面的要求进行手工排序。也可以下载网站上的“扑克排序”的程序，通过它来模拟手工排序：鼠标点击某一张牌，该牌将移到当前的空位上。（正工学员下载课程包中已含该程序）

18.2.2 冒泡排序

“冒泡”是什么意思？湖底有时会冒出一个气泡，气泡刚在湖底时，是很小的，在向上浮的过程中，才一点地慢慢变大。学过高中的物理的人，应该不难解释这一现象。冒泡排序的过程有点类似这个过程，每前进一步，值就大一点。

排序当然有两个方向，一种是从小排到大，一种是从大排到小。大多数教科书里都讲第一种，我们也如此。这样一来，冒泡排序法就改为“沉泡法”了，较大值一点点跑到数组中的末尾。

一般教科书里也会说，冒泡排序法是人们最熟悉，及最直观的排序法，我可不这样认为。或许老外在生活中用的是这种最笨的排序法？我猜想，大家在生活中 99%使用后面要讲的“选择”排序法。

冒泡排序是这么一个过程(从小到大)：

1、比较相邻的两个元素，如果后面的比前面小，就对调二者。反复比较，到最后两个元素。结果，最大值就跑到了最末位置。
2、反复第一步，直到所有较大值都跑到靠后的位置。

看一眼例子：

2，5，1，4，3

第一遍：
• 比较第一对相邻元素：2，5，发现后面的 5 并不比 2 小，所以不做处理。 序列保持不变：2，5，1，4，3
• 继续比较后两对元素：5，1，发现后面的 1 比前面的 5 小，所以对调二者。现在，序列变为：2，1，5，4，3
• 继续比较后两对元素：5，4……对调，于是：2，1，4，5，3
• 继续比较后两对元素：5，3……对调，于是：2，1，4，3，5 <----- OK,现在最大值 5 跑到最尾处了。

大泡泡“5”浮出来了，但前面的 2,1,4,3,还是没有排好，没事，再来一遍，不过，由于最后一个元素肯定是最大值了，所以我们这回只排到倒数第二个即可。

第二遍：
• 比较第一对相邻元素：2，1，发现 1 比 2 小，所以对调：1,2,4,3,5
• 继续比较后两对元素：2，4，不用处理，因为后面的数比较大。序列还是：1,2,4,3,5
• 继续 4，3，对调：1,2,3,4,5。

前面说，5 不用再参加比较了。现在的序列是 1，2，3，4，5。接下来，我们再来一遍：

第三遍：

• 比较第一对相邻元素：1，2：不用对调。

……等等……

有人说，现在已经是 1，2，3，4，5 了，完全是排好序了啊，何必再来进行呢？我们确实是看出前面 1，2，3 也井然有序了，但对于程序来说，它**只能明确地知道自己已经排好了两个数**：4，5，并不知道的 1，2，3 凑巧也排好了。所以它必须再排两次，直到确认把 3 和 2 都已推到合适的位置上。最后剩一个数是 1，因为只有一个数，没得比，所以这才宣告排序结束。

那么到底要排几遍？看一看前面的“第一遍”、“第二遍”的过程你可发现，每进行一遍，可以明确地将一个当前的最大值推到末尾，所以如果排 Count 个数，则应排 Count 遍。当然，最后一遍是空走，因为仅剩一个元素，没得比较。

下面就动手写冒泡排序法的函数。写成函数是因为我们希望这个排序法可处理任意个元素的数组。

```
//冒泡排序(从小到大)：
//num: 要接受排序的数组
//count : 该数组的元素个数
void bubble(int num[],int count)
{
    int tmp;

    //要排 Count 个数，则应排 Count 遍：
    for (int i = 0; i < count; i++)
    {
       for(int j = 0; j < count - i - 1; j++)
       {
           //比较相邻的两个数：
           if(num[j+1] < num[j])
           {
               //对调两个数，需要有"第三者"参以
               tmp = num[j+1];
               num[j+1] = num[j];
               num[j] = tmp;
           }
       }
    }
}
```

注意在内层循环中 j 的结束值是 count - i - 1。 要理解这段代码，明白为什么结束在 count - i -1？如果你忘了如何在 CB 进行代码调试，如果设置断点，如何单步运行，如何观察变量的值，那么你需要“严重”复习前面有关“调试”的章节；如果你一直是高度着每一章的程序到现在，那么你可以继续下面的内容。

排序函数写出一个了，如何调试这个函数？在 CB 里新建一空白控制台程序，然后在主函数里，让我们写一些代码来调用这个函数，并且观察排序结果。

```
#include <iostream.h>
```

```
……
void bubble(int num[],int count)
{
  ……
}

int main() //我实在有些懒得写 main 里两个参数，反正它们暂时对我们都没有用，
           //反正 CB 会为你自动生成，所以从此刻起，我不写了,除非有必要。
{
    int values[] = {2,5,1,4,3};
    int count = sizeof(values[]) / sizeof(values[0]);
    bubble(value,sizeof);
}
```

你要做的工作是单步跟踪上面的代码，看看整个流程是不是像我前面不厌其烦的写的“第一遍第二遍第三遍”所描述的。

完成上面的工作了吗？全部过程下来，只花 20 分钟应该算是速度快或者不认真的了（天知道你是哪一种？天知道你到底是不是没有完成就来骗我？）。现在让这个程序有点输出。我们加一个小小的函数：

```
//输出数组的元素值
//num :待输出的数组
//count:元素个数
void printArray(int num[],int count)
{
  for(int i = 0; i < count; i++)
  {
     count << num[i] << ",";
  }

  cout << endl;
}
```

把这个函数加到 main()函数头之前，然后我们用它来输出：

```
int main() //我实在有些懒得写 main 里两个参数，反正它们暂时对我们都没有用，
           //反正 CB 会为你自动生成，所以从此刻起，我不写了,除非有必要。
{
    int values[] = {2,5,1,4,3};
    int count = sizeof(values[]) / sizeof(values[0]);

    cout << "排序之前：" << endl;
    printArray(values,count);

    //冒泡排序：
    bubble(value,sizeof);
```

```
    cout << "排序之后:" << endl;
    printArray(values,count);

    system("PAUSE");
}
```

后面要讲的其它排序法也将用这个 printArray()来作输出。

冒泡排序是效率最差劲的方法（速度慢），不过若论起不另外占用内存，则它当属第一。在交换元素中使用了一个临时变量（第三者），还有两个循环因子 i 和 j，这些都属于必须品，其它的它一个变量也没多占。

我们现在讲讲如何避免数据其实已经排好，程序仍然空转的的局面。

首先要肯定一点，至少一遍的空转是不可避免的，这包括让人来排，因为你要发现结果已是 1，2，3，4，5 了，你也是用眼睛从头到尾抄了一遍（如果你视力不好，说不定还要扫两遍呢）。

接下来一点，我们来看看除了第一遍空转，后面的是否可以避免。冒泡排序法的空转意味着什么？因为算法是拿相邻的两个比较，一发现次序不合“从小到大”的目的（小的在大的后头），就进行对调。所以如果这个对调一次也没有进行，那就说明后面的元素必然是已经完全有序了，可以放心地结束。

让我们来加个变量，用于标志刚刚完成的这一遍是否空转，如果是空转，就让代码跳出循环。

```
//冒泡排序(从小到大，且加了空转判断)：
void bubble(int num[],int count)
{
    int tmp;
    bool swapped;  //有交换吗?

    //要排 Count 个数，则应排 Count 遍：
    for (int i = 0; i < count; i++)
    {
       //第一遍开始之前，我们都假充本遍可能没有交换（空转）：
       swapped = false;

       for(int j = 0; j < count - i - 1; j++)
       {
           //比较相邻的两个数：
           if(num[j+1] < num[j])  //后面的数小于前面的数
           {
               swapped = true;  //还是有交换

               //对调两个数，需要有"第三者"参以
               tmp = num[j+1];
               num[j+1] = num[j];
               num[j] = tmp;
           }
```

```
        }

        if (!swapped)
           break;
    }
}
```

加了 swapped 标志，这个算法也快不了多少，甚至会慢也有可能。冒泡排序还有一些其它的改进的可能，但同样作用不大，并且会让其丧失仅有优点“代码简单直观”。所以我个人认为真有需要使用冒泡排序时，仅用最原汁原味的“泡”就好。必竟，你选择了冒泡算法，就说明你对本次排序的速度并无多高的要求。

对于 n 个元素，原汁原味的“冒泡排序”算法要做的比较次数是固定的： （n - 1）* n/2 次的比较。
交换次数呢？如果一开始就是排好序的数据，则交换次数为 0。
一般情况下为 3 * (n - 1) * n / 4;最惨时（逆序）为 3 * (n - 1) * n / 2。

冒完泡以后——情不自禁看一眼窗台罐头瓶里那只胖金鱼——让我们开始中国人最直观的选择排序法吧。对了，补一句，如果你看到有人在说“上推排序”，那么你要知道，“上推排序”是“冒泡排序”的另一种叫法，惟一的区别是：它不会让我们联想到金鱼。

18.2.3 选择排序

本章前头我们讲了“求最值”的算法，包括最大值和最小值。其实，有了求最值的算法，排序不也完成了一半？想像一下桌子上摊开着牌，第一次我们从中换挑出大王放在手上，第二次我们挑出小王，然后是黑桃老 K……黑桃 Q，如此下去直到小 A，手中的牌不也就已经排好次序了？

每次从中选出最大值或最小值，依此排成序，这就是选择排序法的过程描述。

不过，上述的过程有一点不合要求。我们说过手中只能过一张牌。因此，在程序实现时，我们找出一个最大值之后，就要把它放到数组中最末。那数组中最末位置原来的值？当然是把它放到最大值原来所在位置了。
为了再稍稍直观点，我们改为：每次找的是最小值，找出后改为放到数组前头。

```
//选择排序（从小到大）
void select(int num[], int count)
{
    int tmp;
    int minIndex; //用于记住最小值的下标

    for (int i = 0; i < count; i++)
    {
        minIndex = i; //每次都假设 i 所在位置的元素是最小的
        for (int j = i + 1; j < count; j++) //j 从 i+1 开始，i 之前的都已排好,而 i 是本次的第一个元素
        {
            if (num[minIndex] < num[j])
               minIndex = j;
```

```
            }

            //把当前最小元素和前面第 i 个元素交换：
            if(minIndex != i)
            {
                tmp = num[i];
                num[i] = num[minIndex];
                num[minIndex] = tmp;
            }
        }
    }
```

同样是两层循环，内层循环非常类似于前面讲的求最值的的方法，重要的区别在于求最小值时，可以直接用 N 记下最小值，而我们这里是记下最小值元素的下标（位置）。最后把这个位置上的元素值和前面第 i 个元素交换。这就完成把挑出的最小值放到前面的过程。

关于如何调试，如何输出，和“冒泡”那一节一样。大家一会儿再动手吧。我先在纸上简要模拟一番，这样大家调试起来会更加心中有数。

2，5，1，4，3

第一遍：找出最小值 1（下标为 2），将它和第一个元素（下标为 0）进行交换，结果： 1,5,2,4,3
第二遍：找出最小值 2（下标为 2），将它和第二个元素进行交换，结果：1,2,5,4,3
第三遍：1,2,3,4,5
同样，我们发现现在已经排好了，但程序的循环过程还得继续。只是后面将是白忙活，什么也没变。最后一遍是剩下一个 1，没得比较，外层循环结束，选择排序完成。
但是，由于选择排序中内层循环完成的工作仅是找出其中的最小值，如果它空转了，只是意味着这些剩下元素中中的第一个元素正好就是最小值，并不意味剩下的元素已经有序。所以我们也不就费心去加什么空转标志了。

同冒泡排序一样，选择排序的外层循环要进行 n-1 次，而内层为 n / 2 次，所以总比较次数为： (n-1) * n / 2。
交换次数最好时为： 3 * (n-1)，最坏时为 n^2 /4 + 3 *(n-1)。

18.2.4 快速排序 （选修）

排序的算法还有不少。譬如“插入排序法”和“希尔排序法”。前者有点像我们抓牌，抓到新牌，往手中已有牌的合适位置插入，最终牌都到手时，也排好序。，后者是以它的创造者的名字命名。它们都不是最快的算法。我们不去说它了。还是来说说（一般说来是）号称最快的算法吧。

“最快”是有代价的。一个是其算法复杂，不直观，根本不是人脑所擅长的思维方式，因为它要求使用“递归”，我想就算是爱因斯坦在整理扑克时，估计也不爱用这种算法。

快速排序的基本思想是分割排序对象。先选择一个值，作为“分割值”。将所有大于该值的元素放到数组

的一头，而所有小于该值的元素，放到数组的另一头。这样就把数组按这个分割值划为两段。接下来的事情是对这两段分别再进行前述的操作（找分割值，再划两段）……就这样一划二、二划四、四划八进行下去。直到最每一段都只剩一个元素，排序完成。

在分段的过程中，每一个数组又是如何被归到某一段中去呢？采用的也是巧妙的交换方法。

假设我们仍然是要进行“从小到大”的排序，那么当有了一个分割值以后，就应该把比分割值大的数扔到数组后头，而比分割值小的数扔到数组前头。在快速排序法中，这个扔的过程是一种“对扔”。即：先找好前面的有哪个数需要扔到后面；再找好后面有哪个数需要扔到前头。两个数都找好了，就把这两个数互相“扔”过去，其实还是交换两个数。知道的人明白我是在说“快速排序”，不知道的人还当我是在说小布和老萨扔板砖哪？

所以，每一遍的分割过程是：
1、指定一个“分割值”。
2、从当前分段的数组前头开始往后找，找到下一个大于分割值的元素（准备扔到后头去）；
3、从当前分段的数组后头开始往前找，找到下一个小于分割值的元素（准备扔到前头去）；
4、交换 2，3 步找到两个元素
5、反复执行 2，3，4，步；直到两端都已找不到要扔的元素。

这样，就把数组在逻辑上分为两段，前头的所有小于分割值是一段，后头所有大于分割值的是段，程序接下来递归调用快速排序的函数，分别把这两段都再次进行分割。

函数的递归调用也是我们曾经的“选修课”，如果你有些遗忘，可以回头加以复习。

每次的分割值是什么并没有太死的限定，但得在当前段数组元素最小值和最大值（含二者）之间，否则，比如元素是： 5，4，3，而分割值取的是 6，就会分不出两段了。我们下面做法比较通用：就取当前段最中间那个元素的值，比如 5，4，3 中的 4。

我们按照书写顺序，把数组前端称为“左端”，后端称为“右”端。下面的代码中,left 和 L 用来表示数组前端，而 right 和 R 表示后端。

```
void quick(int arr, int left, int right)
{
   int tmp;
   int L = left, R = right;
   int V;  //分割值。

   V = arr[ (L + R) / 2];  //分割值取中间那个元素的值。

   do
   {
      //找前端下一个小于分割值的元素：
      while (L < right && arr[L] < V)
          L++;
```

```
        //找后端下一个大于分割值的元素：
        while (R > left && arr[R] > V)
           R++;

        if (L > R) //跑出当前段了，结束本段的“互扔”过程
          break;

        //开始互换，但 L == R 的情况说明是同一个元素，不用交换。
        if ( L < R)
        {
            tmp = arr[L];
            arr[L] = arr[R];
            arr[R] = tmp;
        }

        L++;
        R--;
    }
    while (true);

    //前面还可以分段，继续划分
    //由于前面是在 L > R 情况下 break 出循环，所以 R 此时已经比 L 靠前,所以拿它
    //来和是前头的 left 比较，以确定前面的元素是否超过 1 个。
    if (left < R)
        quick(left, R);   //递归调用 quick()

    //后面还可以分段，继续划分
    //同理，L 此时其实比较靠后。
    if (L > right)
        quick(L, right);  //递归调用 quick()
}
```

快速排序的比较和交换的次数分别为：n * log(n) 和 n * log(n) / 6。远远少于前面的两种排序方法。

18.3 小结

必须承认，在我要写上面的这些排序法算法时，我需要小心翼翼地地进行，并多次测试。 我已经将上述三种排序算加入到“扑克牌排序”的程序中，你下载以后，可以通过菜单项“扑克”下找它三者，运行后程序将演示将用各种算法来排序 1 到 K 十三张牌。不管是哪一种算法，排序 13 个数，都是雷霆之速，所以我加了恐怖的延时，这样你看清扑克牌的交换过程。注意：每当交换两张牌时，届面上的第 14 个空白位置就起到交换中“第三者”的作用。

不过，对于快速排序的演示，你可能还是只能看到前一张后一张的牌在对换，而来不及预想出下一次该换哪两张牌。必竟这一算法复杂得很。

我写这些代码需要小心翼翼，说明两点：

第一点，说明像 quick 这样精妙的算法，确实在理解和记忆上都比较难，而我的水平很一般。

第二点，说明我已经很长时间没有写排序法算法了，虽然我一直在用排序算法，那我的用的代码是从哪里来的？结论是：C，C++库提供的，及 CB 在其它各种场合提供的现成排序算法很好用，“排序？我们一们在用它，代码现成速度又快，效果还真不错！”

现在的 quick 排序算法还很多，连 Windows 的 API 也为我们提供了一份。不过无论是哪个现成的函数，我们现在都用不了，因为这个函数需要一个函数指针做为参数。而我们还没有学到指针（指针啊指针~!@#%$^）

（哎！最近我的手得了什么炎， 打字相当困难，只能边说边划再让人敲键盘了，原想 8 号推出这一课程，可一看电脑，得，9 号凌晨 0 点 1 分。）

关于这一章，怎么说呢？算法是值得大家慢慢推敲的一章，因此，本章是值得花工夫的，但如果你觉得有困难，也是正常的。关于在于一个“慢慢”，我说的“慢慢”就是：“长期地，连续地，一点点来”——“就是：循-序-渐-进-啦！”。

第十九章 指针一 基本概念

19.1 指针是什么？

当我们说“人”这个词时，有时指的是“人类”如：“人是一种会笑的动物”，有时则指个体：比如“张三这人”。

“指针”这个要念也一样，它可以意指“指针类型”，也可以代表某个具体的“指针变量”。下面我们重点要讲的是：“什么叫指针变量”。

所以这一小节的题目其实应是：“指针变量是什么？”

“指针变量”是什么？诚如其名，它首先是一个变量。

变量最重要的特性是什么？那就是它可以存储一个值。比如：

下面是三行代码，行号是我为了表达方便而特意加上的，并不在实际代码中出现。

```
(1)    int a;
(2)    a = 100;
(3)    a = 200;
```

第（1）行定义了一个变量：a。现在它存储的那个值的大小是不定的，因为我们还没有赋给它具体的值呢。

到了（2）行执行以后，a 就存了一个值，这个值多大？答：100。

这里举的是一个整型变量，但其实无论是什么类型的变量，都是用来存值的。

并且，变量的值可以在以后改变大小，比如第（3）行，a 中存放的值成了 200。

回忆完普通变量的特性，现在说“指针变量”。

指针也是变量，所以也是用于存储一个值。重点是，它存储的这个值，意义有点特别。

指针存储的不是普通的一个值，而是另外一个变量的地址。

一句话：**指针是一种用于储存“另外一个变量的地址”的变量**。或者拆成两句：指针是一个变量，它的值是另外一个变量的地址。

这就是指针变量与其它的变量的同与不同：的都是一个变量，都用来存储一个值；但，指针存放的是另外一个变量的地址。

可以这样打个比方：

有一间房子，它的地址是：人民路 108 号。这个房子相当于一个变量。那么：

一、如果它是普通变量，则房子里可能今天住的是张三，明天住的是李四。张三，李四就是这个变量的值。通过访问这间房子，我们可以直接找到张三或李四。

二、如果它是一个指针变量，则房子里不住具体的人，而是放一张纸条，上面写：“南京东路 77 号”。

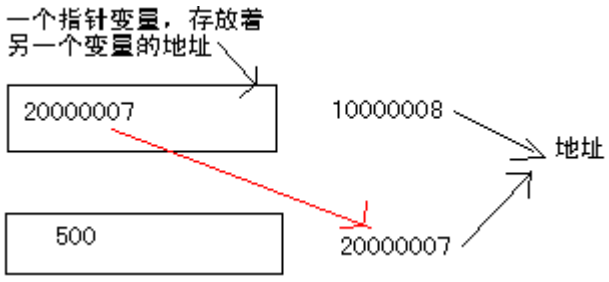
“南京东路 77 号”是一个什么东西？是一个地址。

通过该地址，我们继续找，结果在“南京东路 77 号”里找到张三。

变量的存储的值可以改变，指针变量的值同样可以变更：

过一天，我们再去访问这个房子，纸条变了“珠海路 309 号”，通过它，我们找到的是另一个人。

图解：



能够说出图中两个“20000007”的关系吗？

本质就这点不同，同样是变量，同样要占用一定大小的内存空间，不同的是普通变量在那个内存里，存储了一个具体的值；而指针变量存放的是另一个变量的地址。

不过，什么叫“变量的地址”？只要你是一直跟着我们的课程学习到这里，相信你能明白什么叫变量的“地址”。否则，您可以需要从第一章从头看起了。

说到这里，我们其实一直在说的是“指针变量”，而不是“指针类型”。**指针也需要类型，它所存储的那个变量类型，就称为指针的类型。**继续前面的比方，有一天我们去人民路 108 号，今天纸条写着的地址是：“美眉街 8 号”，于是我们兴冲冲地去了……结果“美眉街 8 号”里住着一头猪！是不是很失落——我们以为是“人类”，并且是“美眉”，未料却关着一头“猪类”？！

计算机虽然不懂得什么叫“失落”，但为了它的准确性，我们需要事先定义好一个指针到底是放的是什么类型的变量。这个类型通常也当作是该指针的类型。

“指针变量中存储某变量的地址”这种说法是不是有些绕？所以有一个简短的说法：“指针**指向**某一变量”。

这种说法的缺陷是不能像第一种说法好样道出指针的本质。但它确实方便。下面我们将不断的使用这两种说法，大家需要知道它们具有相同意义。

19.2 指针的大小

指针的大小是问：一个指针变量占用多少内存空间？

分析：既然指针只是要存储另一个变量的**地址**，。注意，是存放一变量的地址，而不是存放一个变量本身，所以，不管指针指向什么类型的变量，它的大小总是固定的：只要能放得下一个**地址**就行！（这是一间只有烟盒大小的“房间”，因为它只需要入一张与着地址的纸条）。

存放一个地址需要几个字节？答案是和一个 int 类型的大小相同：4 字节。

所以,若有：

int* pInt;

char* pChar;

bool* pBool;

float* pFloat;

double* pDouble;

则: sizeof(pInt)、sizeof(pChar)、sizeof(pBool)、sizeof(pFloat)、sizeof(pDouble)的值全部为：4。

（你敢拆电脑吗？拆开电脑，认得硬盘数据线吗？仔细数数那扁宽的数据线由几条细线组成？答案：32 条，正是 4 * 8）。

我们这一章有很多上机实验。这就算是第一个,只是我提供了代码:请写一个程序,验证上面关于 sizeof(T *)的结论。在写程序之前，务必要先参考一下“数据结构”这一章中 sizeof 的例子。

19.3 定义一个指针变量

数据类型* 变量名;

或：

数据类型 *变量名;

和申请一个普通变量相比，只是在数据类型后面多了一个星号。比如：

int* p;

星号也可以靠在变量名前面，如：

int *p;

要同时定义多个相同类型的指针，则为：

int *p1, *p2;

注意，每个变量之前都必须有 * 。

19.4 初始化指针变量

是变量就存在一个初始化的问题。一个不能确定是住着什么人的房间，总是有点恐怖。

19.4.1 指向不明的指针

我先定义一个整型指针：

int* p;

现在，p 是一个指针，int 规定它只能存放**整型变量**的地址,而不是其它如字符型，布尔型等等。

我们称：p 是一个整型指针。

不过，现在 p 指向哪里 （即：p 存储的是哪个变量的地址 ）？

变量在没有赋值之前，其值不定的。对于指针变量，值不定可以表述为：指向不明。

重点来了！ 一个指向不明的指针，是一个危险的家伙。很多软件有 BUG，其最后的原因，就在这里。

来看看下而的“恐怖片”：

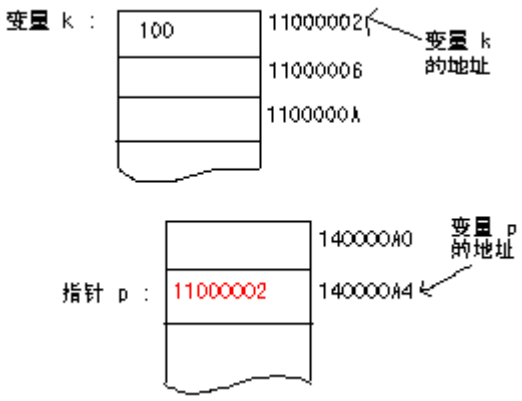
你来到一间阴森森的房间，这房间里有一张纸条	--->	打开火折，但见纸条内容：“XXX 街 3K 号”	--->	你前往这纸条的神秘地址……	--->	“XXX 街 3K 号”里住着一千年老妖！你……
程序访问了一个没有初始化的指针： int* p;		p 的内存是随机的一个数，比如： 0x3FF0073D		程序随即访问内存地址： 0x3FF0073D		0x3FF0073D 是哪里的内存？说不定正好是 Windows 老大要用的内存，你竟敢访问！ Windows 一生气，蓝屏。

既然没有赋值的指针这么危险，下面来看看如何给指针赋值。

19.4.2 给指针变量赋值

```
(1)         int k = 100;
(2)         int* p;
(3)         p = &k;    //p 取得变量 k 的地址 （也称：p 指向 k）
```

第 1 行定义一个整型变量 k。
第 2 行定义了一个整型指针 p。
而第 3 行，“指针 p 存储了变量 k 的地址”。短的说法是“p 指向了 k”。
执行了上面三行代码后，结果是： p 指向 k。我们来看具体的内存示意图。



p 指向 k 示意图

上图中，红色数字代表什么？红数字 11000002 是 变量 k 的内存地址。

指针初始化的结果，就是让它存储下另一个变量的地址。简称指向另一个变量。

下面说说三种常见的，给指针赋值的操作：

一、用 & 取得普通变量的地址。
要想让指针指向某个普通变量，需要通过 & 来得到该普通变量的地址。

```
int k;
int* p = &k;
```

二、指针之间的赋值。

两个指针之间也可以相互赋值，此时不需要通过 & 来取址。

```
int k;
int* p1
int* p2;
p1 = &k;  //p1 先指向 k
p2 = p1;  //然后 ，p2 也指向 k。
```

注意，p2 = p1 的结果是：让 p2 也指向“p1 所指的内容”，而不是让 p2 指向“p1 本身”。

上面的情况可以改为直接在定义时就赋值：

```
int k;
int* p1 = &k;
int* p2 = p1;
```

三、让指针指向数组

数组本身就是地址，所以对数组变量也不用通过 & 来取址。

```
char name[] = "NanYu";
char* p = name;  //不用取址符 &
```

或者：

```
int arr[3] = {100,99,80};
int* p = arr;
```

四、让指针指向一个新地址

前而的赋值方法都是让指针指向一个已有的内存空间的地址。比如：int* p = &k; 指针变量 p 存放的是已有的变量 k 的地址。其实指针也可以指向一个新开辟的内存空间，这一内存空间不归属于其它变量。

在 C++中，常用一个关键字：**new** 来为指针开辟一段**新**空间。比如：

```
int* p = new int;
```

现在，指针变量 p 存储着一个内存地址，该内存地址确实存在——它是由 new 操作符申请而得。可以这样认为，new 是一位特权人物，不通过它，指针只能指向已有的“房间”；而使用了它，则可以要求系统为指针“新开辟一片空地，然后建上新房间”。

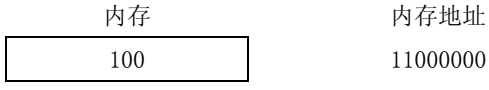
有特权的人不仅仅是“new”，还有几个系统定义的函数，及 Windows 提供的函数，都可以实现“向系统要空间”的功能。我们将在后面专门的 章节详细讲解。

19.5 地址解析 及其操作符 *

* 在 C,C++语言里除了起“乘号”的作用以外，前面我们还刚学了它可以在定义变量时，表明某个变量是属于“指针类型”。现在，则要讲到它还能起“地址解析”的作用。

什么叫“地址解析”？假设有一 int 类型变量 k：

```
int k = 100;
```



方框是变量 k 所占用的内存。100 是该内存中存放的**值**。而 11000000 则是该内存的地址。

“地址解析”就是 **地址->值** 的解释过程。即：通过地址 11000000 得到位于地址的变量。
可见“地址解析（*）” 和 “取址（&）” 正好是一对相反的操作。

这好有一比：地址解析符 * 是个邮递员，他要根据信封上的地址，找到实际的房子。而 取址符 & 则是当初到你家抄下门牌号的人。

看一下实际应用例子：

```
int k = 100;
int* p = &k;
int m = *p;

cout << m << endl;
```

执行以上代码，屏幕上将输出 100。
实际上也可以这样写，以取得相同结果：

```
int k = 100;
int* p = &k;

cout << *p << endl;
```

直接输出 *p 即可。*p 的值就是 100.

明白了吗？ p = &k 让 p 得到 k 的地址，而 *p 则得到 k 的值。
下面就考一考你 & 和 * 的作用。

```
int k = 100;
cout << *&k << endl;
```

将会输出什么？

通过地址解析得到变量，不仅仅可以“得知”该变量的值，还可直接修改该变量的值。

```
int k = 100;
int* p = &k; //p 指向 k

*p = -100;
cout << k << endl;
```

屏幕上将输出 -100。

19.6 上机实验一 指向普通变量的指针和指针之间的互相赋值

实验一有多个步骤。

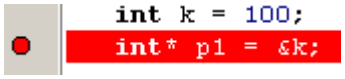
第一步、将上面的代码片段变成程序。请大家打开 CB，新建空白控制台工程，然后输入以下代码。

```
1) int k = 100;
2) int* p1 = &k;

3) cout << *p1 << endl;

4) system("PAUSE");
```

(行前的编号是为下面讲课方便，你当然不能将它们输入到 CB 中去。)

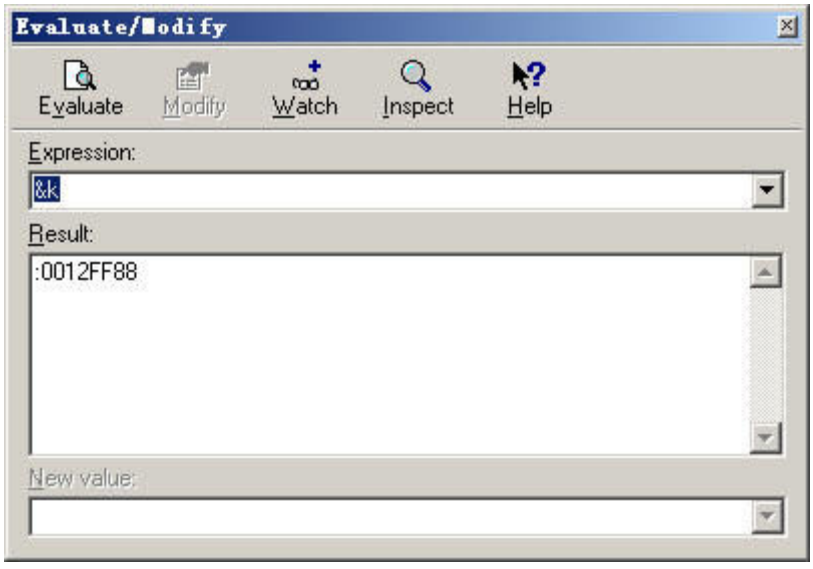
运行将查看结果。然后按任意键退出该程序。下面我们要亲眼看看，指针变量 p1，是否真的存放着普通变量 k 的地址。

把断点设在第 2 行上。(输入光标移到第二行，然后按 F5)



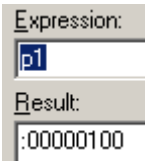
（设置断点）

按 F9 运行，程序停在断点上，然后按 Ctrl + F7 调出"Evaluate/Modify" 对话框。该对话框可以在程序运行时观察或强行修改某个变量的值。我们这里仅是想看看变量 k 的地址。所以如图输入 &k ，回车后，Result（结果）栏出现：0012FF88。这就是 k 的地址。



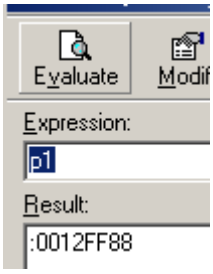
(&k 的值就是 k 的内存地址)

接下来我们该看 p1 的值。看它是否等于 0012FF88。删除 &k ，改为 p1，回车，怎么回事？CB 竟然很不配合地显示 p1 的值为 00000100 ？呵，这是因为当前断点那一行还没有运行呢，p1 现在的值是随机的，这验证好我们前面说的“指向不明”。



(还没有初始化的指针 p1，随随便便指向一个莫名的地址)

关掉该对话框，然后按 F8 再运行一步。再按 Ctrl + F7 调出上面窗口，输入 p1,回车，这回显示的的值正合我们的推想。



（p1 的值，就是 k 的地址，即：p1 等于 &k）

第二步、基于前一题，再加上一个指针变量，叫 p2。

```
1) int k = 100;
2) int* p1 = &k;
3) int* p2 = p;

4) cout << *p1 << endl;
5) cout << *p2 << endl;
```

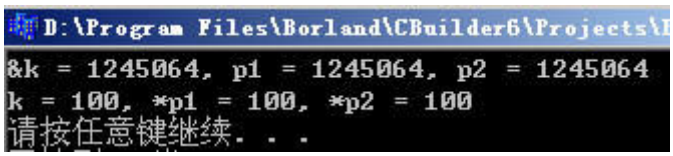
编译并运行，观察结果应发现，*p 和 *p2 值相等，为什么？因为二者指向同一变量：k。

第三步、请像第一步一样，观察，k 的地址，及 p1， p2 的值。看三者是否相等。

最后,将后面的两行输出删除，改为以下两行代码。第一行输出 k 的地址、p1、p2 的值。
第二行 输出 k 的值、*p1、*p2 值。

```
//cout << *p1 << endl;
//cout << *p2 << endl;

cout << "&k = " << &k << ", p1 = " << p1 << ", p2 = " << p2 <<endl;
cout << "k = " << k << ", *p1 = " << *p1 << ", *p2 = " << *p2 <<endl;
```

运行后结果如下：



（1245064 是十进制的，它等于十六进制的 0x0012FF88）

19.7 上机实验二：改变指针所指变量的值，改变指针的指向

尽管在上面的例子中修修改改也能改成本例，不过枝节太多的代码不会混淆了我们的目的。这次我们重点在于“改变”。

第一步、 通过指针，改变其所指向的变量的值。（好绕啊！代码却很简单）

```
int k = 100;

int* p = &k;

//先输出一开始的 k 和*p 的值(用逗号分开)：
cout << k << "," << *p << endl;

//现在直接改变 k 值：
k = 200;

//输出此时的二者的值：
cout << k << "," << *p << endl;

//然后通过指针来改变 k 值：
*p = 300;

//输出此时的二者的值：
cout << k << "," << *p << endl;

system("PAUSE");
```

输出将是：

可见，当 p 指向 k 以后，修改 *p 的值完全等同于直接修改 k 值。

第二步、改变指针的指向

所谓的“改变指向”，其实就是“改变指针变量中存储的值（另一个变量的地址）”。我们一开始说的，两种不同的说法而已。

在前面的代码最后，我们加上以下代码：

```
...
int m = 1000;

//现在 p 改为指向变量 m ：
p = &m;

 k = 400;
cout << k << "," << m << "," << *p << endl;
```

```
*p = 2000;
cout << k << "," << m << "," << *p << endl;

system("PAUSE");
```

屏幕输出是：

当 p 改为指向 m 以后，之前指向的 k 便再也和它没有什么关系了。改变 k 值不会再影响 p；而改变 p 值，则影响到 m 值而非 k 值。

19.8 指针的加减操作

整型变量可以加减，求和，求差：

```
int a = 100;
int b = 99;
int c = a - b;
```

而指针，由于它存的是一个内存地址，那么我们可以想到，对两个指针进行求和，求差，是没有意义的。想想，把你家的门牌号（206）和对面的的门牌号（207）相加（413），得到的数有什么意义吗？

那么，指针的加减指什么呢？主要是指移动（而不是联通：（）。比如，你家是 206，那么，你的下一家，应该是 206 + 1 = 207，而上一家则是 206 - 1 = 205。你会说，我们这里的门牌号不是这样编的。但不管如何，大致应当是一个等差关系，或其它规律。都可以通过不断加上一个数，来得到下一家。

19.8.1 指向数组的指针

现在，来说说指针指向一个数组的情况。

```
int arr[] = {1,2,3,4,5}; //一个数组

int* parr; //一个指针。

parr = arr; //没有 ‘&’ ？对啊，对数组就是不用取址符。

cout << *parr << endl;  //输出 *parr
```

先猜想一下，输出结果是什么？

最“直觉”的想法是：parr 指向一个数组，那么输出时，自然是输出数组中的所有元素了。所以答案应该是：“12345”了？

不过，我想，学过前面的数组，我们就能知道这种想法错误。

正确答案是输出数组中的第一个元素： 1 。

接下来，如果是这样输出呢？

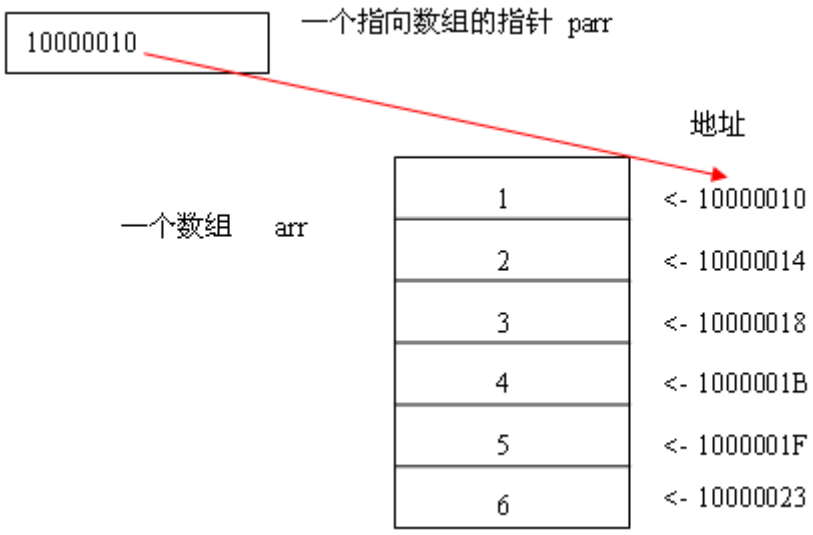
```
parr = arr;
cout << parr << endl;
```

答案是输出了 arr 的地址。就等同于输出 arr 。
cout << arr << endl; 的作用

在这里，难点是要记住，数组变量本身就是地址。所以有：

1、想让指针变量存储一个数组的地址（想让指针变量指向一个数组）时，不用取址符。
2、解析一个指向数组的指针，得到的是数组的第一个元素。

我们来看示意图：



（指向数组的指针）

尽管数组中的每一个元素都有自己的地址，然而一个指向数组的指针，它仍然只是存储数组中第一个元素的地址。复习数组的章节，我们知道，数组中第一个元素的地址，就是数组的地址。即上图中的 10000010。

事实上，如果我们想故意让代码很难理解，则对于这一句：

```
parr = arr;
```
可以改为：
```
parr = &arr[0];
```

本来嘛， arr 和 &arr[0] 的值就相等，我们在数组的章节已经学过。你若不信，可以输出一个：
```
cout << arr << "," << &arr[0] << endl;
```

19.8.2 上机实验三：指向数组的指针

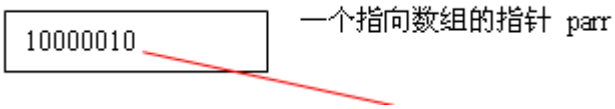
```
int arr[2] = {1,2};
int* p = &arr[0];

//输出：指向数组的指针，数组的地址，数组中第一个元素的地址。
cout << p << "," << arr << "," << &arr[0] << endl;

system("PAUSE");
```

结果是：

（指向数组的指针，数组的地址，数组中第一个元素的地址。）

19.8.3 偏移指针

请看前图的这一部分：



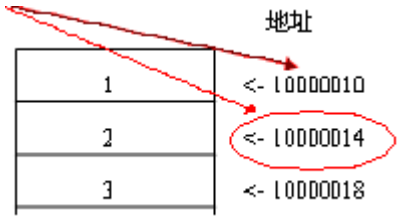
parr 中存的值为 “10000010”。

指针可以进行加减操作。假设我现在再定义一个指针：

```
int* parr2;

parr2 = parr + 1;
```

现在问,parr2 的值是多少？有两种答案。一种说，parr 存的值是 10000010 ,加 1 后,自然应为 10000011 了。这看似自然的答案又是错误了。

正确答案：10000014。继续看前图另一小部分：



（加 1 后，指针指向了下一个元素）

加 1 后，指针指向了下一个元素。由于这是一个 int 类型的数组，每个元素的大小是 4 个字节。所以第

二个元素的地址是 10000014。

重点 & 易错点：对指针 进行加 1 操作，得到的是下一个元素的地址，而不是原有地址值直接加 1。

知到了如何“加”，也就知道了如何“减”。减以后，得到的是上一个元素的大小。

所以，一个类型为 T 的指针的移动，以 sizeof(T) 为移动单位。

比如：
int* pInt; 移动单位为 sizeof(int) 。即：4。而 char* pChar; 移动单位为 sizeof(char)。即 1。

试举一例：

19.8.4 上机实验四：指针的最小移动单位

```
int arr[6] = {101,102,103,104,105,106};
int* pI = arr;

cout << "pI 是一个指向整型数组的指针，移动单位:4 字节" << endl;

for (int i = 0; i < 6; i++)
   cout << "pI + " << i << " ----> " << pI + i << ", *(pI + i) = "  << *(pI + i) << endl;

cout << "-------------------------------------" << endl;

//接下 来是一个指向 char 类型数组的指针：
char str[4] = {'a','b','c','d'}

char*  pC = str;

cout << "pC 是一个指向字符数组的指针，移动单位:1 字节" << endl;
for (int i=0; i < 4; i++)
    cout << "pC + " << i << " ----> " << (int)(pC + i) << ", *(pC + i) = "  << *(pC + i) << endl;

system("PAUSE");
```

输出结果：

(指针的最小移动单位)

每一行中，程序先输出指针加上偏移量以后的值（地址），比如：1245024、1245028；然后输出偏移后指针指向的值，比如 101，102。

查看移动前后指针存储的地址，我们就可以计算出移动单位。1245028 - 1245024 = 4 （byte）。

现在，我们回头再来看这道题：

有以下代码，设 arr 的地址是 10000010，请问最终 指针变量 part2 的值是多少？

```
int arr[] = {1,2,3,4};
int* parr1 = arr;
int* parr2;
int* parr2 = parr + 1;
```

答案： 10000010 + sizeof (int) = 10000014。

这就是对指针加减操作的规则：假设某指针类型为 T* ，则该指针的最小移动单位为： sizeof(T)。

即，若有：

T* p;

则 p + n = p + sizeof(T) * n; 及： p - n = p - sizeof(T) * n;

19.8.5 指针的 += 、 -= 、++、 -- 操作

C、C++ 除了“传统”的 +,- 操作以外，还提供了如题的四种加减操作。这些对于指针同样适用。

```
int arr[] = {1,2,3,4,5,6,7,8,9,10};
int* parr1 = arr;

parr1 += 1; //向后称动一个单位
```

parr1 += 1； 的结果，相当于： parr1 = parr1 + 2;

我们再前面的例子是： parr2 = parr1 + 2; 所以计算结果赋值给 parr2,所以 parr2 指向了 parr1 所指的下一个元素的位置。parr1 本身仍然指在第一个元素。

但对于 parr1 += 1 或 parr1 = parr1 + 2; 则改变的是 parr1 自身的值。

现在，如果来加一句：

```
parr1 -= 1; //向前移动一个单位
```
则 parr1 又回到 arr 数组的开始位置。

也可以直接移动 2 个或更多个单位：

```
parr1 = arr;
parr1 += 2;  //parr1 现在指向：元素 3
parr1 -= 2;  //parr1 现在又回到了 元素 1 上面
```

++ 和 -- 操作 运算结果等同于 += 1 和 -= 1。

```
//后置：
parr1++;
parr1--;

//前置：
++parr1;
--parr1;
```

前置与后置的区别，请复习我们第一次讲 ++ 和 --的章节内容。这时仅举一例，用两段代码来对比，请大家思考，并且最好把它写成实际程序运行。

19.8.6 上机实验五：指针的前置++与后置++的区别

```
//代码片段一：
int arr[] = {1,2,3,4,5,6,7,8,9,10};
int* parr1 = arr;

int A = *parr1++;
int B = *parr1;

cout << "A = " << A << endl;
cout << "B = " << B << endl;
```

输出结果：

```
A = 1;
B = 2;
```

代码片段二：

```
int arr[] = {1,2,3,4,5,6,7,8,9,10};
int* parr1 = arr;

int A = *++parr1;
int B = *parr1;

cout << "A = " << A << endl;
cout << "B = " << B << endl;
```

输出结果：

```
A = 2;
B = 2;
```

19.8.7 * （地址解析符）与 ++ 的优先级

从上例中我们可以看到。当 * （作为地址解析符） 和 ++ 同时作用在指针时，不管是前置还是++，都要比*有更高的优先级。比如代码中的：

```
int A = *parr++;
```

我们来一个反证：假设*的优先级比++高，那么，应先计算：

*parr 结果为： 1 （第一个元素）
然后计算 1++ ，结果为：2。

但实验发现结果为 1，这个 1 又是如何来的呢？有点复杂。
首先，++优先计算，所以应先计算：parr++ 。
结果是 parr 指向了下一个元素：2。因为这是后置++，所以，它必须返回自己计算之前的值；所以，在改变 parr 之前，编译程序会生成一个临时变量，计算原先 parr 的值。我们假设为 old_parr 。下面是第二步操作：
A = *old_parr。
由于 old_parr 是 parr 原来的值，指向第一个元素，所以 A 得到值： 1 。

可见，后置 ++ 或 后置-- 操作，需要系统生成一个临时变量。
如果这个变量占用的内存空间很小(比如指针类型总是只有 4 字节)，则该操作带来的，对程序速度的负面影响可以不计，如果变量很大，并且多次操作。则应在可能的情况下，尽量使用前置++或前置--操作。

你自然会问，前置++就不会产生临时变量吗？我们来试试。

```
int A = *++parr;
```

同样，++优先级大于*，所以先计算：++parr。
结果 parr 指向下一个元素。因为这是前置++，所以，它只需要返回的，正是自己计算之后的值。下一步是：
A = *parr;
由于 parr 此时已完成++操作，指向下一个元素。所以 A 得到值： 2 。

19.8.8 上机实验六：指针的 ++与--操作

```
int arr [] = {1,2,3,4,5};

int* parr = arr;

//前进 ++：
for (int i=0; i < 5; i++) //如果为了优化，你可以写成： ++i :)
{
   cout << *parr << endl;
   parr++;   //如果为了优化，你可以写成：++parr :D

   /*
     上面两句你还可以写成一句：
     cout << *parr++ << endl;  //这里，你可不能为了优化写成： *++parr.
  */
}

//后退 --：
for (int i=0; i < 5; i++)
{
   parr--;
   cout << *parr << endl;
}
```

输出结果：

（指针的++与--）

19.9 小结

指针是什么？不过也是一个变量，只是存储的是另一个变量的内容地址。
指针有多大？总是 4 字节。

如何定义指针？ 多了一个*。
如何为指针赋值？全靠一个个&，除非对方已经是地址（如对方也是指针或是一个数组）。
如何得到指针所指的变量？如何通过指针改变所指变量？地址解析符：*。
如何移动指针？加加减减。

第二十章 指针 二 为指针分配和释放空间

20.1 理解指针的两种“改变”

普通变量（非指针，简单类型变量）只能改变值：

```
1) int a = 100;
2) ...
3) a = 200;
```

第 1 行代码，声明 int 类型变量 a，并且初始化 a 的值为 100。
到第 3 行代码，变量 a 的值被改变成 200。

对于非指针的简单变量，能被程序改变的，只有这一种。而指针变量，似乎有两种改变。

20.1.1 改变指针的值

这一点和普通变量一致。但要特别注意，“改变指针的值”引起的结果是什么？其实就是“改变指针的指向”。

因为，指针的值是某个变量的地址。假如指针 P 原来的值是 A 变量的地址，现在改为 B 变量的地址。我们

称为：“P 由指向 A 改为指向 B”。这就是指针的第一种改变。

以下是示例代码：

```
int* P;
int A,B;

1) P = &A;
2) ...
3) P = &B;
```

1）行代码中，P 的值为 &A，即 P 指向变量 A。
到 3）行代码中，P 的值变为&B，即改为指向变量 B。

下面讲：指针的第二种改变。通过指针，改变指针所指的变量的值。

20.1.2 改变指针所指的变量的值

复习前一章，我们知道通过 * （地址解析符），可以得到、或改变指针所指的变量的值。

```
int* P;
int A = 100;

P = &A;
*P = 200;

cout << A << endl;
```

代码中加粗的那一行：*P = 200; ，其作用完全等同于：A = 200;
所以，最后一行输出的结果是 200。

这就是指针的第二种改变：所指变量的值，被改变了。

20.1.3 两种改变？一种改变？

两种改变的意义不同：
改变一：改变指针本身的值（改变指向）。
改变二：改变指针指向的变量的值。

从代码上看：
第一种改变，P = &A; 左值（等号左边的值）是变量本身，右值则是一个地址。
而第二种改变，*P = 200; 左值通过星号对 P 操作，来取得 P 指向的变量；右值是普通的值。

理解,区分对指针的两种改变，才能学会如何使用指针。

请思考：上一章讲的“指针的加减操作”，是对指针的哪一种改变？

最后需要说明，严格意义上，指针仍然只有一种改变，即改变指针本身的值。改变指针指向的变量，应视为对另一变量的改变，只不过在代码上，它通过指针来进行，而不是直接对另一变量进行操作。

为指针分配、释放内存空间

之前，我们给指针下的定义是“指针是一个变量，它存放的值是另一个变量的地址”。
比如：

```
int a;
int* p = &a;
```

看，a 就是“另一个变量”，p 指向了 a。
我们知道，变量总是要占用一定的内存空间，比如上面的 a，就占用了 4 个字节（sizeof(int)）。这四个字节属于谁？当然属于变量 a，而不是 p。

现在要讲的是：也可以单独为指针分配一段新的内存空间。这一段内容不属于某个变量。

20.2 C++ 方式的内存分配与释放 new 和 delete

在内存管理上，C++ 和 C 有着完全不同的两套方案。当然，C++的总是同时兼容 C。C 的那一套方案在 C++里同样可行。
我们首先看看纯 C++的那一套： new 和 delete。
new ，从字面上看意思为 “新”；而 delete 字面意思为“删除”。二者在 C++中内存管理中大致的功能，应是一个为“新建”，一个为“删除”。

20.2.1 new

new 是 c++ 的一个关键字。被当作像 +、-、* 、/ 一样的操作符。它的操作结果是在申请到一段指定数据类型大小的内存。

语法：

指针变量 = **new** 数据类型;

new 将做三件事：

1、主动计算指定数据类型需要的内存空间大小；
2、返回正确的指针类型；
3、在分配内存的一，将按照语法规则，初始化所分配的内存。

这是什么意思呢？看看例子吧：

```
int* p;
p = new int;
```

和以往不一样，p 这回不再“寄人篱下”，并不是指向某个已存在的变量，而是直接指向一段由 new 分配

而来的新内存空间。

“p 指向一段由 new 分配而来的新内存空间” 这句话等同于：
“new 分配一段新的内存空间，然后将该内存空间的地址存入到变量 p 中。”
所以，最终 p 中仍然是存储了一个变量的地址，只是，这是一个“无名”变量。

指向原有的某个变量，和指向一段新分配的内存空间，有什么区别呢？
“原有的变量”，可以比喻成指向一间原有的，并且有主的房间。而“新分配的内存空间”，则像是一个“临时建筑物”。我们必须在不用它的时候，主动将它拆迁。拆迁的工作由 delete 来完成。

当指针变量通过 new ，而得到一个内存地址后，我们就可以像以前的所说的，通过该指针，通过*号，而对该内存地址（一个无名的变量），进行操作。
如：

```
int* p = new int;
*p = 100;
cout << *p << endl;
```

屏幕将输出 100。

20.2.2 在 new 时初始化内存的值

new 也可以在申请内存空间时，直接设置该段内存里要放点什么.

语法：

指针变量 = **new** 数据类型**(初值)**;

这样，上例可以改为：

```
int* p = new int(100);
cout << *p << endl;
```

如果你申请的是字符类型的空间,并且想初始化为‘A’：

```
char* pchar = new char('A');
```

20.2.3 delete

语法：
delete 指针变量;

delete 将释放指定指针所指向的内存空间。

举例：

```
int* p;
p = new int;

*p = 100;
cout << *p << endl;

delete p;

system("PAUSE");
```

注意，当一个指针接受 delete 操作后，它就又成了一个“指向不明”的指针。尽管我们可以猜测它还是指向“原来的房子”，然而，事实上，那座“房子”已经被 delete “拆迁”掉了。

20.2.4 实验： new 和 delete

很简单的例子。
第一步：
首先，在 CB 新建一个控制台程序。然后把上一小节的代码放到 main()函数内。运行。结果如下：

(new 和 delete)

按任意键退出后，保存工程(Ctrl + Shift + S)。

第二步：
接下来我们来观察指针变量被 delete 之后，所指向的内存会是什么。但，这是一件犯了 C、C++编程大忌的事：访问一个已经 delete 的指针的值。如果你最近运气很差，你的 CB 可能会被强行退出。所以，你明白我们为什么要先存盘了，对不？

在前面的代码中，加入以下加粗加红的一行(同时，你也应注意我的加的注释)：

```
int* p;
p = new int;

*p = 100;
cout << *p << endl;

delete p;     //p 所指向的内存空间已经被释放

cout << *p << endl;  //我们故意去访问此时 p 所指的内存

system("PAUSE");
```

运行结果：

（访问 delete 之后的指针）

44244844？？在你的机器可能不是这个数，但一定同样是怪怪的值。 原来是好端端的 100，现在却成了 44244844。不要问我这是为什么？昨天来时，美眉还住在这里一座别致小阁楼里，今日故地重游，这里竟成废墟一片，依稀只见破墙上尚有：**“拆！——城建局”**的字样？！

new 是管建房的，而 delete 就一个字：拆！

请大家自行在 CB 上完成本实验。我没有提供本题的实际工程。

20.2.5 new 和 delete 的关系

如果只有“建房”而没有“拆房”，那么程序就会占用内存越来越多。所以，当使用 new 为某个指针分配出内存空间后，一定要记得在不需要再使用时，用 delete 删除。下面是一个例子。演示 new 和 delete 的对应使用。

```
//建屋和入住：
1) int* p = new int(100);

//使用：
2) cout << *p << endl;

//拆：
3) delete p;
```

看，第 1 句，申请了 4 字节的内存空间，同时存入值为 100 的整数。
第 2 句，在屏幕上输出入住者的值（100）。
第 3 句，释放内存（这 4 字节被系统收回准备做其它用途）。入住者呢？自然消失了。

前面举的例子都是在 new 一个 int 类型，其它类型也一样：

```
char* a = new char('A');
cout << *a << endl;
*a = 'B';
cout << *a << end;
delete a;

bool* b = new bool;
*b = true;

if (*b)
   cout << "true" << endl;
```

```
else
  cout << "fale" << endl;
```

但是这些都是简单数据类型，如果要分配数组一样的连续空间，则需要使另一对武器。

20.3 new [] 和 delete []

new / delete 用于分配和释放单个变量的空间，而 new [] / delete[] 则用于分配连续多个变量的存间。

20.3.1 new[] / delete[] 基本用法

new [] 语法:

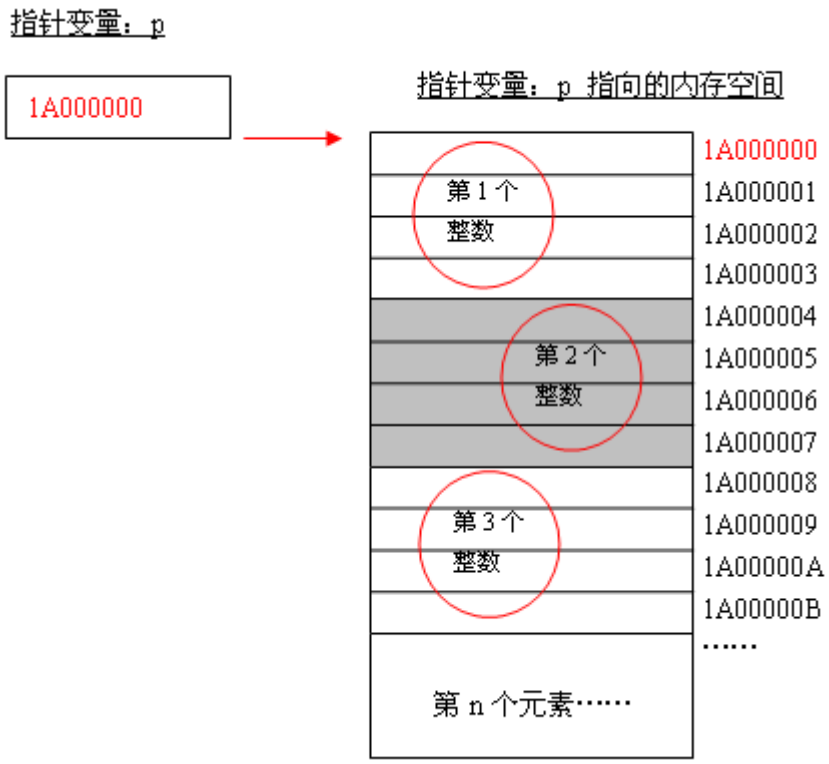
指针变量 = **new** 数据类型**[元素个数]**

语法实例：

```
int* p = new int[20];
```

首先，你需要迅速回想一下，如果是 int* p = new int(20); 那么该是什么作用？否则你很容易在事后把二者混了。

实例中，用 new 申请分配了 20 个连续的整数所需的空间，即：20 * sizeof(int) = 80 个字节。
图示为：



(指针变量 p 指向一段连续的内存空间)

new int 只是分配了一个整数的内存空间，而 new int[N]却分配了 N 个整数的连续空间。看来，new[] 比 new “威力更猛”，所以，我们同样得记得：用 new [] 分配出空间，当不在需要时，必须及时调用 delete [] 来释放。

delete [] 语法：

delete [] 指针变量;

如：

```
//分配了可以存放 1000 个 int 的连续内存空间：
int* p = new int[1000];

//然后使用这些空间：
……

//最后不需要了，及时释放：
delete [] p;
```

20.3.2 new []/ delete[] 示例

在 Windows XP 、Windows NT 或 Windows 2000 中，按 Ctrl + Alt + Del (其它操作系统，如 Windows98/Me 等千万不要按些组合键，否则电脑将重启)。可以调出 Windows 任务管理器，其中要以看出当前粗略的的内存使用量。下面我们结合该工具，写一个程序，先分配 100M 的内存，再释放。

这是程序代码的抓图：

```
int main(int argc, char* argv[])
{
    cout << "现在，我什么也没有做，按任意键我开始吃内存。" << endl;
    system("PAUSE");

    unsigned char *p = new unsigned char [1024 * 1024 * 100];

    cout << "哇，好饱啊。我吃掉了整整100兆的内存！" << endl;

    system("PAUSE");

    delete [] p;

    cout << "现在，我吐还100兆的内存。" << endl;
    system("PAUSE");

    return 0;
}
//--------------------------------------------------------------------
```

各步运行结果：

程序显示	任务管理器抓图

第一步：分配内存之前 现在，我什么也没有做，按任意键我开始吃内存。 请按任意键继续. . .	 （任务管理显示我的机器使用了 207 兆的内存）
第二步：分配了 100 兆的内存 哇，好饱啊。我吃掉了整整100兆的内存！ 请按任意键继续. . .	 （多出了 100M）
第三步：又释放出这 100 兆 现在，我吐还100兆的内存。 请按任意键继续. . .	 （回到 207 兆）

注意：使用 new 得来的空间，必须用 delete 来释放；使用 new [] 得来的空间，必须用 delete [] 来释放。彼此之间不能混用。

用 new [] 分配出连续空间后，指针变量“指向”该空间的首地址。

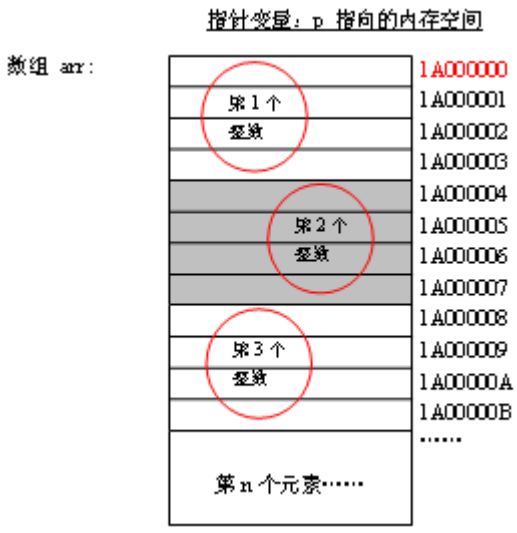
20.3.3 详解指向连续空间的指针

在 通过 new [] 指向连续空间以后，p 就变得和一个一维数组很是类似。我们先来复习一下数组相关知识。

假设是这么一个数组：

```
int arr[20];
```

则 arr 的内存示意图为（为了不太占用版面我缩小了一点）：



（数组 arr 的内存示意）

和指针变量相比， 数组没有一个单独的内存空间而存放其内存地址。即：指针变量 p 是一个独立的变量，只不过它的值指向另一段连续的内存空间；而数组 arr，本身代表的就是一段连续空间。

如果拿房间来比喻。指针和数组都是存放地址。只不过，指针是你口袋里的那本通讯录上写着的地址，你可以随时改变它的内容，甚至擦除。而数组是你家门楣上钉着的地址，你家原来是“复兴路甲 108 号”，你绝对不能趁月黑天高，把它涂改为“唐宁街 10 号”。

数组是“实”的地址，不能改变。当你和定义一个数组，则这个数组就得根据它在内存中的位置，得到一个地址，如上图中的“0x1A000000”。只要这个数组存在，那么它终生的地址就是这个值。

指针是一个“虚”的地址，可以改变地址的值。当你定义一个指针变量，这个变量占用 4 个字节的内存，你可以往这 4 字节的内存写入任意一个值，该值被当成一个内存地址。比如，你可以写入上面的“0x1A000000”，此时，指针 p 指向第一个元素。也可以改为“0x1A000003”，此时，指针 p 指向第二个元素。

所以，当 p 通过 new [] 指向一段连续空间的结果是，p 是一个指向数组的指针，而*p 是它所指的数组。

我们来看实例，首先看二者的类似之处。下面左边代码使用数组，右边代码使用指针。

数组	指针 (通过 new [] 所得)
//定义: int arr[20];	//定义： int* p = new int[20];

//让第一个元素值为 100：
arr[0] = 100;
	//让第一个元素值为 100：
	p[0] = 100;

//让后面 19 个元素值分别为其前一元素加 50：
for (int i = 1; i < 20; i++)
{
arr[i] = arr[i-1] + 50;
}
	//让后面 19 个元素值分别为其前一元素加 50：
	for (int i = 1; i < 20; i++)
	{
	p[i] = p[i-1] + 50;
	}

//输出所有元素：
for (int i = 0; i < 20; i++)
{
cout << arr[i] << endl;
}
	//输出所有元素：
	for (int i = 0; i < 20; i++)
	{
	cout << p[i] << endl;
	}

//也可以不用[]，而通过+号来得到指定元素：
//当然，对于数组，更常用的还是 [] 操作符。
cout << *(arr + 0) << endl; //*(arr+0) 等于 *arr
cout << *(arr + 1) << endl;
cout << *(arr + 1) << endl;
 	
输出结果：
100
	//也可以不用[]，而通过+号来得到指定元素：
	//其实，对于指针，这样的+及-操作用得还要多点。
	cout << *(p + 0) << endl; //*(p + 0) 等于 *p
	cout << *(p + 1) << endl;
	cout << *(p + 1) << endl;
	
	输出结果：
	100

150 200	150 200

当指针变量 P 通过 new [] 指向一连续的内存空间：

1、p[N] 得到第 N 个元素 （0 <= N < 元素个数）;2、*(p + N) 同样得到第 N 个元素 （0 <= N < 元素个数）

如 p[0] 或 *(p + 0) 得到内存空间第 0 个元素；

把上面右边代码中的大部分 p 替换为 arr，则和左边代码变得一模一样。

下面再来比较二者的不同。

数组	指针

//定义并且初始化：
int arr[20] = {0,1,2,3,4,5,6,7,8,9,0,……,19};
	//定义、并且生成空间，但不能直接初始空间的内容：
	int* p = new int[20] {0,1,2,3,4 ……} // 错!
	
	//只得通过循环一个个设置：
	for (int i = 0; i < 20; i++)
	{
	  p[i] = i;
	}

//不能通过对数组本身 + 或 - 来改变数组的位置：
arr = arr + 1; // 错!
cout << *arr << endl;
 	
arr++; // 错!
cout << *arr << endl;
 	
arr--; // 错!
cout << *arr << endl;
 	
输出结果：
无，因为程序有语法错误，通不过编译。
	//可以通过 + 或 - 操作直接改变指针：
	p = p + 1;
	cout << *p << endl;
	
	p++;
	cout << *p << endl;
	
	p--;
	cout << *p << endl;
	
	输出结果：
	1
	2
	1

//释放空间：
//数组所带的空间由系统自动分配及回收
//无须也无法由程序来直接释放。
	//释放空间：
	//指向连续空间的指针，必须使用 delete [] 来释放
	delete [] p;

关于指针本身的 + 和 - 操作，请复习上一章相关内容。

接下来的问题也很重要。

20.4 delete/delete[] 的两个注意点

指针通过 new 或 new[] ，向系统“申请”得到一段内存空间，我们说过，这段内存空间必须在不需要将它释放了。有点像人类社会的终极目标“共产主义”下的“按需分配”。需要了就申请，不需要了，则主动归还。

现在问题就在于这个“主动归还”。当然，指针并不存在什么思想觉悟方面的问题，说光想申请不想归还。真正的问题是，指针在某些方面的表现似乎有些像“花心大萝卜”。请看下面代码，演示令人心酸的一幕。

```
/*
  初始化 p  ----- p 的新婚
  通过 new ，将一段新建的内存“嫁给”指针 p
  这一段分配的内存，就是 p 的原配夫妻
*/
int* p = new int[100];



/*
   使用 p  ----- 恩爱相处
   N 多年恩爱相处，此处略去不表
*/
……

/*
   p 改变指向 ---- 分手
*/

int girl [100];   //第三者出现
p = girl;         //p 就这样指向 girl



/*
   delete [] p ----  落幕前的灾难

   终于有一天，p 老了，上帝选择在这一时刻
   惩罚他
*/

delete [] p;
```

扣除注释，上面只有 4 行代码。这 4 行代码完全符合程序世界的宪法：语法。也就是说对它们进行编译，编译器会认为它们毫无错误，轻松放行。

但在灾难在 delete [] p 时发生。
我们原意是要释放 p 最初通过 new int[100]而得到的内存空间，但事实上，p 那时已经指向 girl[100]

了。结果，第一、最初的空间并没有被释放。第二、girl[100] 本由系统自行释放，现在我们却要强行释放它。

20.4.1 一个指针被删除时，应指向最初的地址

当一个指针通过 +,- 等操作而改变了指向；那么在释放之前，应确保其回到原来的指向。

比如：

```
int* p = new int[3];

*p = 1;
cout << *p << endl;

p++;    //p 的指向改变了，指向了下一元素
*p = 2;
cout << *p << endl;

//错误的释放：
delete [] p;
```

在 delete [] p 时，p 指向的是第二个元素，结果该释放将产生错位：第一个元素没有被释放，而在最后多删除了一个元素。相当你盖房时盖的是前 3 间，可以在拆房时，漏了头一间，从第二间开始拆起，结果把不是你盖的第 4 房间倒给一并拆了。

如何消除这一严重错误呢？
第一种方法是把指针正确地"倒"回原始位置：

```
p--;
delete [] p;
```

但当我们的指针指向变化很多次时，在释放前要保证一步不错地一一退回，会比较困难。所以另一方法是在最初时“备份”一份。在释放时，直接释放该指针即可。

```
int* p = new int[3];
int* pbak = *p;    //备份

//移动 p
……

//释放：
delete [] pbak;
```

由于 pbak 正是指向 p 最初分配后的地址，我们删除 pbak,就是删除 p 最初的指向。此时我们不能再删除一次 p。这也就引出 new / delete 及 new[] / delete[] 在本章的最后一个问题。

20.4.2 已释放的空间，不可重复释放

第一种情况，错了最直接：

```
int* p = new int(71);
cout << *p << endl;

delete p; //OK!
delete p; //ERROR! 重复删除 p
```

当然，如果同一指针在 delete 之后，又通过 new 或 new[] 分配了一次内存，则需要再删除一次：

```
int* p = new int(71);
cout << *p << endl;

delete p; //OK!
...
p = new int(81);
delete p; //OK!
...

p = new int[10];
for (int i=0; i<10; i++)
  *p = i;

...
delete [] p; //OK!
```

上面代码中，共计三次对 p 进行 delete 或 delete[]，但不属于重复删除。因为每次 delete 都对应一次新的 new。

我们下面所说的例子，均指一次 delete 之后，没有再次 new，而重复进行 delete。

第二种情况，重复删除同一指向的多个指针

```
int* p1 = new int(71);
int* p2 = p1;   //p2 和 p1 现在指向同一内存地址

cout << *p1 << endl;
cout << *p2 << endl;

delete p1;  //OK
delete p2;  //ERROR! p2 所指的内存，已通过 delete p1 而被释放，不可再 delete 一次。
```

同样的问题，如果你先删除了 p2，则同样不可再删除 p1。

```
...
delete p2; //OK
delete p1; //ERROR
```

第三种情况，删除指向某一普通变量的指针

```
int a = 100;
int* p = &a;
delete p;  //ERROR
```

p 不是通过 new 得到新的内存空间，而是直接指向固定变量：a。所以删除 p 等同要强行剥夺 a 的固有空间,会导致出错。

20.5 C 方式的内存管理

new/delete 只在 C++里得到支持。在 C 里，内存管理是通过专门的函数来实现。另外，为了兼容各种编程语言，操作系统提供的接口通常是 C 语言写成的函数声明 （Windows 本身也由 C 和汇编语言写成）。这样，我们就不得不同时学习 C 的内存管理函数。

20.5.1 分配内存 malloc 函数

需要包含头文件：

```
#include <alloc.h>
```

或

```
#include <stdlib.h>
```

函数声明(函数原型)：

```
void *malloc(int size);
```

说明：malloc 向系统申请分配指定 size 个字节的内存空间。返回类型是 void* 类型。void* 表示未确定类型的指针。C,C++规定，void* 类型可以强制转换为任何其它类型的指针。

从函数声明上可以看出。malloc 和 new 至少有两个不同：new 返回指定类型的指针，并且可以自动计算所需要大小。比如：

```
int *p;

p = new int; //返回类型为 int* 类型(整数型指针)，分配大小为 sizeof(int);
```

或：

```
int* parr;

parr = new int [100];  //返回类型为 int* 类型(整数型指针)，分配大小为 sizeof(int) * 100;
```

而 malloc 则必须由我们计算要字节数，并且在返回后强行转换为实际类型的指针。

```
int* p;

p = (int *)  malloc (sizeof(int));
```

第一、malloc 函数返回的是 void * 类型，如果你写成：p = malloc (sizeof(int)); 则程序无法通过编译，报错：“不能将 void* 赋值给 int * 类型变量”。所以必须通过 (int *) 来将强制转换。

第二、函数的实参为 sizeof(int) ，用于指明一个整型数据需要的大小。如果你写成：

```
int* p = (int *) malloc (1);
```

代码也能通过编译，但事实上只分配了 1 个字节大小的内存空间，当你往里头存入一个整数，就会有 3 个字节无家可归，而直接“住进邻居家”！造成的结果是后面的内存中原有数据内容全部被清空。

malloc 也可以达到 new [] 的效果，申请出一段连续的内存，方法无非是指定你所需要内存大小。

比如想分配 100 个 int 类型的空间：

```
int* p = (int *) malloc ( sizeof(int) * 100 ); //分配可以放得下 100 个整数的内存空间。
```

另外有一点不能直接看出的区别是，malloc 只管分配内存，并不能对所得的内存进行初始化，所以得到的一片新内存中，其值将是随机的。

除了分配及最后释放的方法不一样以外，通过 malloc 或 new 得到指针，在其它操作上保持一致。

20.5.2 释放内存 free 函数

需要包含头文件(和 malloc 一样)：

```
#include <alloc.h>
```

或

```
#include <stdlib.h>
```

函数声明：

```
void free(void *block);
```

即： void free(指针变量);

之所以把形参中的指针声明为 void* ，是因为 free 必须可以释放任意类型的指针，而任意类型的指针都可以转换为 void *。

举例：

```
int* p = (int *) malloc(4);

*p = 100;
```

```
free(p); //释放 p 所指的内存空间
```

或者：

```
int* p = (int *) malloc ( sizeof(int) * 100 ); //分配可以放得下 100 个整数的内存空间。
……
free(p);
```

free 不管你的指针指向多大的空间,均可以正确地进行释放，这一点释放比 delete/delete [] 要方便。不过，必须注意，如果你在分配指针时，用的是 new 或 new[]，那么抱歉，当你在释放内存时，你并不能图方便而使用 free 来释放。反过来，你用 malloc 分配的内存，也不能用 delete/delete[] 来释放。一句话，new/delete、new[]/delete[]、malloc/free 三对均需配套使用，不可混用！

```
int* p = new int[100];
... ...
free(p);  //ERROR! p 是由 new 所得。
```

这也是我们必须学习 malloc 与 free 的重要理由之一，有时候，我们调用操作系统的函数（Windows API）时，会遇到由我们的程序来分配内存，API 函数来释放内存；或 API 函数来分配内存，而我们的程序来负责释放，这时，必须用 malloc 或 free 来进行相应的工作。

当然，保证所说的内存分配与释放方式不匹配的错误发生，Windows API 函数也提供了一套专门的内存管理函数给程序员,为了不在这一章里放太多相混的内容，我们在 Windows 编程的课程再讲相关内容。

最后还有一个函数，也是我们要学习 C 方式的内存管理函数的原因。

20.5.3 重调空间的大小： realloc 函数

需要包含头文件(和 malloc 一样)：　　#include <alloc.h>　或　#include <stdlib.h>

函数声明：

```
void *realloc(void *block, int size);
```

block 是指向要扩张或缩小的内存空间的指针。size 指定新的大小。

realloc 可以对给定的指针所指的空间进行扩大或者缩小。size 是新的目标大小。比如，原来空间大小是 40 个字节，现在可以将 size 指定为 60，这样就扩张了 20 个字节；或者，将 size 指定为 20，则等于将空间缩小了 20 个字节。

无论是扩张或是缩小，原有内存的中内容将保持不变。当然，对于缩小，则被缩小的那一部分的内容会丢失。

举例：

```
//先用 malloc 分配一指针
int* p = (int *) malloc (sizeof(int) * 10);  //可以存放 10 个整数
```

```
……
//现在，由于些某原因，我们需要向 p 所指的空间中存放 15 个整数
//原来的空间不够了：
p = (int *) realloc (p, sizeof(int) *15); //空间扩张了 (15 - 10) * sizeof(int) = 20 个字节

……

//接下来，我们决定将 p 所指内存空间紧缩为 5 个整数的大小：
p = (int *) realloc (p, sizeof(int) * 5); //缩小了 (15 - 5) * sizeof(int) = 40 个字节

……

free (p);
```

这么看起来，realloc 有点像是施工队对一个已建的房屋进行改修：可以将房间后面再扩建几间，也可以拆掉几间。不管是扩还是拆，屋里原来的东西并不改变。

不过，这里要特别提醒一点：这个施工队有时会做这种事：1、在一块新的空地上新建一座指定大小的房屋；2、接着，将原来屋子里的东西原样照搬到新屋；3、拆掉原来的屋子。

这是什么指意呢？

realloc 并不保证调整后的内存空间和原来的内存空间保持同一内存地址。相反，realloc 返回的指针很可能指向一个新的地址。

所以，在代码中，我们必须将 realloc 返回的值，重新赋值给 p ：

```
p = (int *) realloc (p, sizeof(int) *15);
```

甚至，你可以传一个空指针（0）给 realloc ，则此时 realloc 作用完全相当于 malloc。

```
int* p = (int *) realloc (0,sizeof(int) * 10);  //分配一个全新的内存空间，
```

这一行，作用完全等同于：

```
int* p = (int *) malloc(sizeof(int) * 10);
```

20.5.4 malloc、realloc、free 的例子

打开 CB6，新建一空白控制台工程。

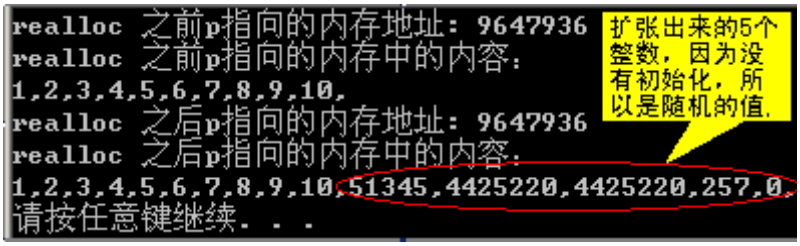
第一步：在 Unit1.cpp 中的最前面，加入引用 alloc.h 等头文件的代码：

```
……
#pragma hdrstop
#include <alloc.h> //三个函数的声明都这个头文件里
#include <iostream.h>
……
```

第二步：将以下代码加入主函数 main 中间：

```
int* p = (int *) malloc (sizeof(int) * 10);
cout << "realloc 之前 p 指向的内存地址: " << p << endl;
for (int i=0; i<10; i++)
{
   p[i] = i + 1;
}
cout << "realloc 之前 p 指向的内存中的内容：" << endl;
for (int i=0; i<10; i++)
{
   cout << p[i] << ",";
}
cout << endl;
p = (int *) realloc (p, sizeof(int) * 15);

cout << "realloc 之后 p 指向的内存地址: " << p << endl;

cout << "realloc 之后 p 指向的内存中的内容：" << endl;
for (int i=0; i<15; i++)
{
   cout << p[i] << ",";
}
cout << endl;
free (p);
system("PAUSE");
```

运行结果：



(malloc, realloc, free 上机题)

从图中我们看到 realloc 前后的 p 指向的内存地址同样是 9647936 。但记住，这并不是结论，真正的结论我们已经说过“realloc **并不保证**返回和原来一样的地址。”所谓的“并不保证”的意思是：“我尽力去做了，但仍然有可能做不到。”。

无论调用几次 realloc,最后我们只需一次 free。

第二十一章 指针 三 实例演练与提高

21.1 简单变量、数组、指针

学习的知识点越来越多了……

刚开始会觉得很兴奋啊，学得越多越好嘛。可慢慢的就会感到压力了，各种知识点在头脑里混在一起，每个都变得模糊了。

其实，每个知识点都有它存在，或出现的理由，只要我们多做对比，就会发现学习的知识点越多，反倒越容易理解每个知识点本质。

比如说，简单变量、数组、指针，三者都是 C++中用于表达数据的工具，但在表达能力上，又各有不同。

如果用建筑上的房间来比喻：

简单变量是一间房屋。优点是占用空间少，建筑时间短，缺点是一间房子只适于住一个人；

数组是房间数固定的一排房子，每个房子里头同样只住一人，但由于它有多间，所以适于多人居住，优点是可以统一管理多人，缺点一来是占用空间大，二来房间数一旦确定，就不能改变了。先头盖了 10 间，如果如果来了 11 个人，就有一人住不下，如果来了 9 个人，就有一间浪费。

指针呢……它不是实际房子，而是设计纸上的房子。因此，它首先有一个特点：如果你想让指针存储数据，那一定得先为它分配内存。这就像光有一张设计蓝图是解决不了四代同堂的问题的，重要的是你还得根据这张蓝图，去找块地皮盖好房子。指针的优点是可以临时决定要盖多少间房子。

下面我们回顾一个例子，以理解三者的不同用处。

21.2 小王成绩管理系统 V2.0 的问题

先回顾一下该程序的升级过程，今天我们将对它做出两种不同方向的改进。

21.2.1 软件升级历史

V1.0 ： 本版成绩管理系统实现让计算机自动统计 6 个班级的成绩总分和平均分。

V2.0 ： 经过改进，本版可以实现多达 5000 个学生的成绩进行求总分和平均分，并且可以支持用户输入序号，查询任意一个学生的成绩！

在第一版，小王正在学习“循环流程”。通过在每次循环中，让用户输入一个成绩，然后保存在一个简单变量里，并累加到另一个简单变量，最终计算出总分和平均分。

第二版，由于段长要求不仅可以统计 5000 个学员的成绩，而且应实现成绩查询功能，这就要求程序必须同时记下 5000 个学生成绩。小王先是想用 5000 个简单变量来记下成绩——这显然太不实际了，后来学到数组，用数组轻松解决了这个问题，因为数组正是为“同时存储多个相同类型的变量”这一问题来设计的。

然而，第二版存在的不足也是显然易见的。那就是，它固定只能处理最多 5000 个学员的成绩。假想，这个软件要推广到全市 300 个学校，每个学校的学生总数都是不一样的，更惨的是每一年，一个学校的学生个数总是会有变化。难道就让我们的王老师时不时地改它的程序？

在没有指针时，惟一办法就是，浪费一点，比如定义数组元素个数为 1 万。目的是宁可浪费一点，也尽量不要出现不够的情况。显然，本办法只能算是一个无奈之举。难道就没有一个办法，即可以适应某个山区小学只有 30 名学员也情况，又可以轻松对付某大学高达 2 万名学员的情况？

锣声响起，锵锵锵……指针出场了。

指针是如何完成这一历史使命？带着问题，我们来学习下面的内容。

我们会在学习新内容之中，同时有选择地做一些旧知识点的复习工作。但如果你仍看不懂下面的一些代码，那得全面复习前两章的指针内容；或者，如果你连 for 都有些陌生，那你得重温一下小王成绩管理系统的前两个版本。

21.3 指针的最常用用法

21.3.1 分配内存

如何为指针分配和释放内存，上一章的内容中讲到了 C++独用的 new/delete、new[] / delete[] 和 C 使用的 malloc, realloc/ free 方法。如果你忘了，请先复习。我们这里使用 C++的方法演练。

new 只能为我们分配一个简单变量的内存，就是说 new 只盖了一间房子。new [] 才能为我们盖出一排的房子。

例子：

```
int* p; //定义一个整型指针
```

p = new [10]; //new [] 为我们分配出 10 个 int 大小的内存。（盖了 10 间房，每间住一个整数）

21.3.2 访问指针指向的内存

前面：p = new [10]; 为我们分配了 10 个 int，那么，我们该如何设置和访问这 10 个整数的值呢？这一点完全和数组一致，我们来看数组是如何操作：

```
int a[10]; //以数组方式来定义 10 个 int

//让第 1 个整数的值为 100：
a[0] = 100;
//让第 2 个整数的值为 80：
a[1] = 80;
```

指针的操作方式如下：

```
int* p = new int[10]; //定义 1 个整型指针，并为它分配出 10 个 int 的空间

//让第 1 个整数的值为 100：
p[0] = 100;
//让第 2 个整数的值为 80：
p[1] = 80;
```

对比以上两段代码，你可以发现，对指针分配出的元素操作，完全和对数组的元素操作一致。不过，指针还有另一种对其元素的操作方法：

```
int* p = new int[10]; //定义 1 个整型指针，并为它分配出 10 个 int 的空间

//让第 1 个整数的值为 100：
*（p+0） = 100;
//让第 2 个整数的值为 80：
*（p+1） = 80;
```

请大家自己对比，并理解。如果觉得困难，请复习第 19 章关于*的用法，和指针偏移部分的内容。

21.4 小王成绩管理系统 V3.0

3.0 版的最重要的改进就是：用户可以事先指定本校的学生总数。

请仔细看好。

```
//定义一个指针,用于存入未知个数学生的成绩：
int* pCj;

//总成绩，平均成绩：
int zcj=0, pjcj;

//首先，要求用户输入本校学生总数:
int xszs; //学生总数
cout << "请输入本校学生总数:";
cin >> xszs;

//万一有调皮用户输入不合法的总数，我们就不处理
if (xszs <= 0)
{
    cout << "喂，你想耍我啊？竟然输入一个是 0 或负数的学生总数.我不干了！" << endl;
    return -1; //退出
}

pCj = new int[xszs];

//仍然可以用我们熟悉的循环来实现输入：
for(int i=0; i < xszs; i++)
{
   cout << "请输入第" << i+1 << "学员的成绩:";
   cin >> pCj[i];      //输入数组中第 i 个元素

   //不断累加总成绩：
   zcj += pCj[i];
}

//平均成绩：
pjcj = zcj / xszs;

//输出：
cout << "总成绩：" << zcj << endl;
cout << "平均成绩：" << pjcj << endl;

//下面实现查询：
int i;

do
```

```
{
    cout << "请输入您要查询的学生次序号(1 ~ " << xszs << "):" ;
    cin >> i;

    if( i >= 1 && i <= xszs)
    {
        cout << cj[i-1] << endl; //问：为什么索引是 i-1，而不是 i ？
    }
    else if( i != 0)
    {
        cout << "您的输入有误！" << endl;
    }
}
while(i != 0);  //用户输入数字 0，表示结束。

//最后，要释放刚才分配出的内存：
delete [] pCj;
......
```

请大家现在就动手，实现小王成绩管理 3.0 版。这是本章的第一个重点。通过该程序，你应该可以记住什么叫“动态分配内存”。

21.5 字符串指针

21.5.1 为字符串分配指定大小的空间

有必要的话，你应复习一下第 16 章之第 6 节：字符数组。

假设有个老外叫 "Mike"，以前我们用字符数组来保存，需要指定是 5 个字符大小的数组：

```
char name[5] = "Mike";
```

"Mike"长 4 个字符，为什么要 5 个字符的空间来保存？ 这是因为计算机还需要为字符串最后多保存一个零字符：'\0'。用来表示字符串结束了。

在学了指针以后，我们可以用字符串指针来表达一个人的姓名：

```
char* pname = "Mike";
```

此时，由系统自动为 pname 分配 5 个字符的位置，并初始化为 "Mike"。最后一个位置仍然是零字符：‘\0’。

采用字符串的好处，同样前面所说的，可以在程序中临时决定它的大小（长度）。
比如：

```
char* pname;
pname = new char[9]; //临时分配 9 个字符的大小。
```

除了要记得额外为字符串的结束符'\0'分配一个位置以外，字符串指针并没有和其它指定有太多的不同。

既然讲到字符串，我们就顺带讲几个常用的字符串操作函数

21.5.2 字符串常用函数

字符串操作函数的声明都包含在该头文件：<string.h>

21.5.2.1 字符串比较

```
int strcmp(const char *s1, const char *s2);
```

比较 s1 和 s2 两个字符串，返回看哪个字符串比较大。对于字母，该比较区分大小写
返回值：

```
< 0   :  s1 < s2;
  0   :  s1 == s2;
> 0   :  s1 > s2;
```

```
int strcmpi(const char *s1, const char *s2);
```

该函数类似于上一函数，只是对于字母，它不区分大小写，比如它认为'A'和'a' 是相等的。

要说两个字符串相等不相等，还好理解，比如： "Borland" 和 "Borlanb" 显然不相等。不过，字符串之间还有大小之分吗？
对于字母，采用 ASCII 值来一个个比较。谁先出现一个 ASCII 值比较大的字母，谁就是大者。比如："ABCD" 比 "AACD"大。
如果一直相等，但有长短不一，那就长的大。比如：“ABCD” 比 “ABC”。
记住了，由于在 ASCII 表里，小写字母比大写字母靠后，所以小写的反倒比大写的大。比如："aBCD"比"ABCD"大啊。

我这里写个例子，看如何比较字符串：

```
#include <string.h>
#include <iostream.h>
...

int reu = strcmp ("ABCD", "AACD");
```

```
if (reu > 0)
   cout << "没错， ABCD > AACD" << endl;
else
   cout << "搞错了吧？" << endl;
```

请大家照此例，分别比较 "ABCD" 和 "ABC" 、 "aBCD" 和 "ABCD"。
如果你对如何用 C++ Builder 建立一个控制台下的工程，请复习第二章第 3 节。

前面说的是英文字母，对于汉字字符串的比较，大小是如何确定的呢？
对于常用汉字，Windows 按其拼音进行排序，比如“啊”是最小的，排在最前面，而“坐”之类的，则比较大，排在后面。
对于非常用的汉字，则按笔划来排序。有关常用不常的划分，是国家管的事，我们就不多说了。

我一直在网上叫“南郁”，大家可以拿你的名字和我做一下 strcmp，看看谁的名字比较大。（友情提醒：名字大没有什么好处，相反，名字大了，在各种场合里，一般是排名靠后的……）

21.5.2.2 字符串复制

```
char *strcpy(char *dest, const char *src);
```

该函数用于将字符串 src 的内容，复制给 字符串 dest。 注意，一定要保证 dest 有足够的空间。
该函数最后返回 dest.

比如:

```
char name1[10];
char* name2 = "张三";

strcpy (name1, name2);
```

现在 name1 的内容也是“张三”。

21.6 指针数组

学过数组，指针，二者结合起来，指什么？

21.6.1 什么叫“指针数组”？

一个数组用来存放整型数，我们就叫它 整型数组或整数数组；
一个数组用来存入字符，就叫字符数组；
同样，一个数组用来存入指针，那就叫指针数组。

比如：
int* p; 这只是一个指针.
而
int* p[10]; 这是一个数组，里头存放了 10 个指针。

请大家区分：
int* p = new int[10];
和
int* pa[10];
前者，是一个指针，并且该指针分配了 10 个元素的空间。
而后者，则是一个指针数组，用于存放 10 个指针(pa[0],pa[1]...pa[9]都是指针)，这 10 个指针都可以分配 10 个元素（也可以不是 10 个，比如是 8 个或 11 个）。

仍然以建筑来比喻：

int* p = new int[10];
p 是 一张（是的一张而已）图纸，new int[10] 是 我们根据它建了 10 间房子。

int* pa[10];
pa[10] 是 10 张图纸。至于这 10 图纸上各准备建多少间房子，我们暂未定下。

我们可以通过一个循环，来为 pa[10]中的每个指针分配 8 个元素的空间。

```
for (int i=0; i < 10; ++i)
{
   pa[i] = new int [8]; //为每个指针都分配 8 个 int 元素空间。
}
```

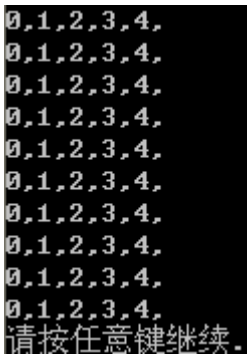
21.6.2 指针数组实例一

请打 CB,并新建一个控制台工程（记得在出现的对话框中选中“C++选项”）。
然后输入以下代码（部分代码是 CB 自动生成的，你不必加入）：

```
//---------------------------------------------------------------------------
#pragma argsused
int main(int argc, char* argv[])
{
   //定义一个指针数组，可以存放 10 个 int 型指针
   int *p[10];

   //循环，为每个指针各分配空间。
   for(int i=0; i<10; ++i)
   {
```

```
        p[i] = new int [5];   //分配 5 个元素的空间

        //然后为当前指针中每个元素赋值：
        for (int j = 0; j<5; ++j)
           p[i][j] =  j;
     }

    //输出每个指针中每个元素（用了两个“每”，所以需要两层循环）
    for(int i=0; i<10; ++i)
    {
        for (int j = 0; j<5; ++j)
           cout << p[i][j] << ",";
       cout << endl;
    }

    //重要！！！最后也要分别释放每个指针
    for(int i=0; i<10; ++i)
      delete [] p[i];

    system("Pause");
    return 0;
  }
  //---------------------------------------------------------------------------
```

本例的输出为：



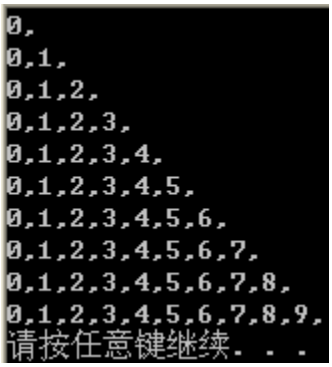
21.6.3 指针数组实例二

本例对上例做一些小小改动：

```
//---------------------------------------------------------------------------
#pragma argsused
int main(int argc, char* argv[])
{
   //定义一个指针数组，可以存放 10 个 int 型指针
```

```
    int *p[10];

    //循环，为每个指针各分配空间。
    for(int i=0; i<10; ++i)
    {
       p[i] = new int [i+1];   //分配 i+1 个元素的空间

       //然后为当前指针中每个元素赋值：
       for (int j = 0; j<i+1; ++j)
          p[i][j] = j;
    }

   //输出每个指针中每个元素
   for(int i=0; i<10; ++i)
   {
       for (int j = 0; j<i+1; ++j)
          cout << p[i][j] << ",";
      cout << endl;
   }

   //重要！！！最后也要分别释放每个指针
   for(int i=0; i<10; ++i)
     delete [] p[i];

   system("Pause");
   return 0;
}
//-------------------------------------------------------------------------
```

本例的输出为：



21.6.4 字符串指针数组

假设我们想在程序中加入某个班级的花名册。让我们来想想如何实现。

由于一个人名由多个字符组成，所以人名就是一个数组。而全班人名，就是数组的数组，因此可以用二维

数组来实现。

假设本班只有 3 个学员，每个学员的人名最长不超过 4 个汉字,每个汉字占 2 个字符，加上最后 1 个固定的结束符，共 9 个字符来表示一个人名。

```
char names[3][9] =
{
    {"郭靖"},
    {"李小龙"},
    {"施瓦辛格"},
};
```

这是个不错的解决方案。惟一稍有一点不足的是，我们为每个学员分配长 9 个字符，其实像 1 号郭靖，他只需 5 个字符就足矣。但我们使用“二维数组”的方案无法为不同长度的姓名分配不同的空间。

这时候，就可以用字符串指针数组了！“锵锵锵”，我们让字符串指针数组的方案出场。

```
char* names[3] =
{
    {"郭靖"},
    {"李小龙"},
    {"施瓦辛格"},
};
```

变化并不多，但是学杂费空间的问题却得到了解决。更为美妙的是，就算现在新来一位大侠叫“无敌鸳鸯腿”，我们也可以从容处理：

```
char* names[4] =
{
    {"郭靖"},
    {"李小龙"},
    {"施瓦辛格"},
    {"无敌鸳鸯腿"},
};
```

接下来，我们来对“小王成绩管理 V2.0”系统做另一种改进。

此次改进并不要求可以动态输入学校学生的总数。相反，作来一种“定制版”，我们希望专门为某个学校加入花名册功能,希望在录入成绩时，可以增加显示该学生的姓名。

假设这个学校叫“精武文武学校”，并暂定本期学员也只有上述那 4 位。

"精武馆定制版的小王成绩管理系统"如何实现？呵，我不写了，大家写吧。题目条件就是上面的黑体部分。最终录入界面应类似于：

快点开始做吧，把你的作业发在 BBS 里，让我评评你的成绩是多少 :)))) ？

第二十二章　结构

大家好。课程拖了好久，大家急，我也急。今天是周末，一早去医院体检，被女医生扎了一针，胳膊上留下一个针眼。不禁想起一个真实的故事。一个我的同行（程序员），和我差不多瘦。有一年夏天到南方出差，住在旅馆里，一个晚上没睡好！为什么？因为蚊子又太多了啊。几夜没睡好这可真够倒霉的啦。谁知祸不单行。上了火车他困啊！卧铺上一歪他就睡着了，那只胳膊瘦瘦的，从睡铺上垂下来，露出被蚊子们叮的密密麻麻的红点。才睡不久，就被乘警叫醒带走——下铺的乘客去举报了，说他是吸毒青年。

兄弟们，当程序员很苦！不过苦得值。当然身体要弄好。

22.1 面向对象的启蒙

我们以前学习了各种数据类型的变量。先来复习一下。

变量做什么用？程序用来变量来存储数据，用它来表达真实世界中的事物。

比如：假设我最近开了一家幼儿园，园里有一群小娃娃。娃娃的家长们把孩子交给我们之后，都要求我们要时时关心孩子们的“健康成长”，因此我们不得不为每个孩子建一个入园档案。档案记载每个孩子的一些数据。

```
//娃娃的名字：
char xingMing[11];  //姓名最长 5 个汉字，占用 10 字节，多留一个字节用于存放'\0'

//娃娃的身高：
int shenGao; //单位 cm

//体重:
float tiZhong; //单位 公斤
```

我们知道，世界的万事万物，去除一些无用的修饰，可以表达为“数据”和“操作”。比如：我吃饭，“我”

和“饭”是数据，而“吃”是一种动作，一种操作。对应到编程语言，就是“数据”和“流程”。那么，当我们写程序来解决某一现实问题时，应该先考虑的“数据”，还是“流程”呢？多数问题，应该先去考虑“数据”。也就是说，当遇上一个现实问题，我们应先去“抽取”这个问题的数据，考虑如在程序中表达，设计，定义这些数据。然后再去设计流程。

以我们上面的幼儿园管理的例子，我们现在已经差不多抓出相关的数据。不过，上面所做的，只是“一个孩子”的数据，幼儿园里的孩子当然不会只有一个。还好，我们学过数组，不是吗？ 所以，我们将上面的变量定义改一改。

先定义一个宏，假设园里有 30 名宝宝。

```
#define BAOBAO_GESHU 30  //30 个宝宝

//30 个宝宝，要 30 个名字：
char xingMing[BAOBAO_GESHU][11]; //忘了二维数组？呵呵。复习一下。

//30 个宝宝，30 个身高：
int shenGao[BAOBAO_GESHU];

//30 个宝宝，30 个体重：
float tiZhong[BAOBAO_GESHU];
```

假设我们的程序具备打印功能。这样每一天放学时，我们都在宝宝走出校门前，将他或她称量一番，得出体重，身高。记录到我们的程序，然后程序再打印出一张小纸条。贴在宝宝后脑勺，供宝宝妈妈参考……哈哈，我们可以把这个小程序卖给本市的 300 家幼儿园，每份卖 400 元，300 家就是 400＊300＝120000 元……流口水中……

擦干口水回过神，来开始我们的今天最重要的学习：面向对象的启蒙。
什么叫面向对象，我且不解释。不结合实例子，解释也没有用啊。

一个人，有眼睛、鼻子、嘴、头发、四肢。也就是说，“人”是一种“数据”，而“鼻子”，“嘴”，“头发”等也各自是一种数据，彼此之间具备不同的“数据类型”。但是在多数情况下，“人”是一种不可再分的整体（医院里负责解剖的人所做的事不在我们所说的多数情况之内）。扯到编程上而来 ，当我们想用程序管理 30 个人时，我们应该是定义一个数组，存储 30 个人，而不是分开来定义成：30 个眼睛[2]、30 个鼻子、30 个头发[1000]，30 个四肢。

回到幼儿园程序。每个宝宝都有身高、体重、姓名这些属性；宝宝也应作为一个整体，而不是像上面那样分开来定义。

这就是面向对象的第一个启蒙：面向对象，是为了让我们把程序写得更“自然而然”。越是支持面向对象的编程语言，我们就越能以接近人类自然逻辑的思路来设计程序；而越不支持面向对象的编程语言，也许它初看起来似乎很浅显易用，但当用它来解决实际问题时，程序员就不得不受限于这门语言特有的解决问题的思路。

说完面向对象的好处，我们必须马上来做几个问题的“纠偏”。

第一、面向对象并不代表解决问题的最高效率。

确实地这样的。“面向对象”被提出，是因为某些问题实在太庞大太复杂了，人类的写程序解决这些问题时，一会儿就胡涂了，晕了，搞错了，于是希望有一种方法来让程序员不会那么快在一堆代码前精神崩溃。这才提出了“面向对象”。所以在我们第一次接触到这个概念时，请先记住，和我们前面所讲的一样，比哪为什么要有变量，为什么要有数据类型：编程语言设计的不断改进，都是为了迁就人类的“容易犯错”或“天生懒惰”。否则，如果只是要追求最高效率，如果人类有着机器般的脑，编程语言根本不需要有 C，C++，JAVA，C#什么的，甚至连汇编也不需要，只需要一个机器语言就可以。“面向对象”的编程思想迁就我们什么呢？它迁就人类总是喜欢以比较自然的方式来解决问题。

先来考虑，“自然而然”的事，不一定是最高效率。这很多，比如，路口的红绿灯制度，显然不是最高效率。最高效率其实应该是闯红灯。你会说，闯红灯会出车祸，出车祸不就堵车？造成效率低了？哦，其实我是要说：如果每个司机，行人都闯红灯，并且能保证不撞到其它车或行人，那么路口上车辆穿行的效率肯定最高。可惜，驾驶员做不到这一点，所以只好有红绿灯。

第二、虽然说面向对象是“很自然的事”，但我们仍然要花力去学习。

古人老子说:“道法自然”。那什么叫“自然”啊?

这里的自然也是有规定的，并不是人类的所有行为都称为“自然”的，也不一定是你习惯的行为就自然，你不习惯的行为就不自然。比如人家都觉得“饭前便后要洗手”，可若你偏要认为这种行为太不自然，太做作，那我们也没有办法。

另外，人类解决现实生活中，有时也要追求点效率。比如，酒家承办婚礼，要准备 10 桌一样的酒席。每一桌上面都有这道菜那道汤的。我们可以把完整的一桌酒菜看成一个“整体”。但大厨们可不这样认为，我猜他们在准备时，一定是先把某道菜一块儿做好 10 桌的份量，然后做下一道菜，而不是我们认为的，先办好一桌，再办下一桌。对于我们，一桌一桌菜是自然的，而对做的人来说，一道一道菜才是自然的。

如何设计一个面向对象的程序，并且保证一定的高效率，这是一门无止境的科学，我们需要不断地学习。面象对象源于生活，而高于生活。

说了这么多，大家不要被“面向对象”吓坏了。今天我们所要学习的面向对象的设计方法，很简单：**把同属于一个整体的“数据”，归成一个新的类型**去考虑，而不是分割成每一部分。

22.2 结构/struct 的定义

“结构”提供一种能力，允许程序员把多个数据类型，归属成一个完整的，新的数据类型。

以上面的幼儿园管理程序为例，我们可以定义出这样一个**新的**数据类型：

```
struct BaoBao
{
    char xingMing[11];
    int shenGao;
```

```
    float tiZhong;
};//<----注意，以分号结束定义
```

现在，我们有了一种新的数据类型，叫 BaoBao 结构。该数据类型由三个变量的定义组成。分别是 xingMing[10], shenGao, tiZhong。这三个组成变量，我们称为该结构的“**成员变量**”。

既然 BaoBao 是一种数据类型，我们就可以定义一个该类型的变量：

```
BaoBao daBao; //daBao 是一个“BaoBao”类型的变量。
```

这个过程，和我们定义一个整型变量，并无多大区别:

```
int a;
```

我们记得不同数据类型的变量，有着不同的大小（占用内存的大小）。
比如：一个 bool 或一个 char 类型的变量，占用一个字节，而一个 int 类型则占用 4 个字节的内存；
后来我们又学习数组，数组的大小除了和它的数据类型有关外，还有数组元素的个数有关。比如：char xingMing[11]占用 11 个字节，而 int txZhong[30]占用 4 * 30 个字节
最后，前面的几章内，我们又学习了指针。指针类型固定占用 4 个字节。这是因为不管什么类型的指针，都是要用来存储一个内存地址，而在 32 位的计算机上，内存地址的大小固定为 4 字节（8 * 4 = 32 位）。

这一切我们可以用 sizeof 来验证：

```
int s1 = sizeof(char);
int s2 = sizeof(int);
int s3 = sizeof(char *);

int arr[10];
int s4 = sizeof(arr);
```

上面的程序执行后，s1 到 s4 分别为：1,4,4,40。

现在我们来问，我们自定义的 BaoBao 数据类型，它的大小是多少？换句话，也可以问：做出如下定义之后：

```
BaoBao daBao;
```

daBao 这个变量“吃”掉了多少字节的内存呢？

我们再看一次 BaoBao 这个结构的定义:

```
struct BaoBao
{
```

```
    char xingMing[11];
    int shenGao;
    float tiZhong;
};
```

直观上猜测，BaoBao 这个结构由三个成员变量组成，那么它的大小应该就是这三个成员变量大小之和。这个猜测颇有道理，如果它是正确的话，那么，sizeof(BaoBao) 应等于 11 + 4 + 4 = 19;

让我们打开 CB6，然后新建一个控制工程。然后在 Unit1.cpp 里增加一些代码，使之看来如下(黑体部分为需要增加的代码)：

```
//---------------------------------------------------------------------------
#pragma hdrstop
#include <iostream.h>

//---------------------------------------------------------------------------
struct BaoBao
{
   char xingMing[11];
   int shenGao;
   float tiZhong;
};

#pragma argsused
int main(int argc, char* argv[])
{
   int size = sizeof(BaoBao);

   cout << "SizeOf struct BaoBao = " << size << endl;

   system("PAUSE");

   return 0;
}
//---------------------------------------------------------------------------
```

按 F9 后看到运行结果：

（sizeof BaoBao）

奇怪，BaoBao 结构的大小，竟然是 20。比我们猜测的 19，多出了一个字节？

事情是这样的。就像我们去小店买东西，假设有一天我们要去赶飞机，走之前去小店买了点食品，总价 19 元，店老板没有 1 元钱，于是我们为了能快点出发，就直接给他 20 元，告诉他不用找零钱了。

为了效率，编译器也会有类似的动作。这称为结构的“字节对齐”，当然，这个对齐方法要比我们的 19 取整到 20 复杂一点。关于编译器是通过什么规则来把一个结构进行扩展，我们留在本章的增补课程中。这里只需记住，对于一个结构的大小，一定要用 sizeof 才能得到它的实际大小。当然，可以肯定的是，**结构的大小一定是大于或等于其所有成员变量的大小之和**。

现在我们知道，定义了一个 BaoBao 的变量（daBao），就会在吃掉 20 个字节的内存。接下来我们来看，如何使用 daBao 这个变量呢？

22.3 . 操作符

通过点操作符，我们可以得以一个结构中的成员变量。
请看例子：

```
BaoBao daBao; //定义一个 BaoBao 类型的变量，变量名为 daBao;

daBao.shenGao = 83; //“大宝”的身高是 83 公分
daBao.tiZhong = 14; //体重是 14 公斤

//字符串用 strcpy 函数来设置
strcpy(daBao.xingMing, "大宝");
```

.操作符，可以理解为“的”。不是吗？ daBao.shenGao ，就读成 “大宝的身高”，多么的“自然而然”啊！今天我们已经摸到“面向对象”编程世界的门环了。

22.4 -> 操作符

结构变量也可以是一个指针：

```
BaoBao* pDaBao; //定义一个 指向“BaoBao”结构的指针
```

考一下，pDaBao 占用几个字节的内存，如果你回答是：20，那真该自己绝食一顿。：（

指针的大小总是只有 4 个字节。指向“结构”的指针也如此。我们可以通过 new 操作符来为一个指针分配实际内存：

```
pDaBao = new BaoBao;
```

这一点，和我们定义一个 int 型指针，然后为它分配内存的操作一致：

```
int* pInt = new int;
```

pInt 分配后，指向一块大小为 sizeof(int)的内存，而 pDaBao 分配后，指向一个大小为 sizeof(BaoBao)，的内存。

对于指向结构的指针变量，要取得相应结构内的成员变量，必须通过以下语法：

```
(*pDaBao).xingMing;
(*pDaBao).shenGao;
```

从语法上分析，你必须先复习一下《指针》章节中，关于*的用法与意义。*pDaBao 得到指针实际向的结构变量，然后再进行点操作符，得到该结构变量内的某一成员。

不过上面的写法显然很繁琐。简化方法是使用 -> (可读作箭头操作符，或指向操作符，或者就叫“减大于”吧)：

```
pDaBao->xingMing;
pDaBao->shenGao;
```

也就是说，如果一个变量是结构变量，那么它可以直接用.来取得它的成员。而如果变量是一个结构的指针，那么可以请用 ->来得到它的成员。

顺便说一句，为一个指向结构的指针变量分配的内存，当不再需要时，同样需要记得释放：

```
//定义指针，并分配内存：
BaoBao *pDaBao = new BaoBao;

//为各成员赋值：
pDaBao->shenGao = 171;
pDaBao->tiZhong = 13.5;
strcpy(pDaBao->xingMing,"大宝");

//输出：
cout << pDaBao->xingMing << "的身高为: " << pDaBao->shenGao << endl;
cout << pDaBao->xingMing << "的体重为: " << pDaBao->tiZhong << endl;

//释放：
delete pDaBao;
```

22.5 结构实例

例子中，我们要求老师输入 5 个宝宝的数据，然后程序将这 5 个宝宝的情况打在屏幕上：

打开 CB，新建一个控制台工程。

因为需要输入输出，所以先在 Uint1.cpp 中加下以下黑体部分：

```
#pragma hdrstop
#include <iostream.h>
```

然后是定义 BaoBao 结构：

```
//---------------------------------------------------------------------------
struct BaoBao
{
    char xingMing[11];
    int shenGao;
    float tiZhong;
};  //分号结束结构的定义
```

为了方便演示，我们只让有 5 个宝宝,在上面的结构定义代码后面加一行：

```
#define BAOBAO_GESHU 5  //宏定义之后没有分号
```

然后，我们需定义一个 BaoBao 数组变量：

```
int main(int argc, char* argv[])
{
   BaoBao baoBao[BAOBAO_GESHU];

    return 0;
}
```

程序一开始时，我们要求幼儿园老师输入这五个宝宝的数据：

```
int main(int argc, char* argv[])
{
   BaoBao baoBao[BAOBAO_GESHU];

   for (int i = 0; i < BAOBAO_GESHU; ++i)
   {
      cout << "请输入第 " << i + 1 << "个宝宝的数据" << endl;
```

```
        cout << "姓名:";
        cin >> baoBao[i].xingMing;

        cout << "身高:";
        cin >> baoBao[i].shenGao;

        cout << "体重:";
        cin >> baoBao[i].tiZhong;
    }

    return 0;
}
```

运行上述程序，我们就可以进行输入了。注意，名字不可以超过 5 个汉字，否则程序将可能发生不可预料的错误。

最后，再一个循环，我们将 5 个宝宝的数据输出：

```
int main(int argc, char* argv[])
{
    BaoBao baoBao[BAOBAO_GESHU];

    for (int i = 0; i < BAOBAO_GESHU; ++i)
    {
        cout << "请输入第 " << i + 1 << "个宝宝的数据." << endl;

        cout << "姓名:";
        cin >> baoBao[i].xingMing;

        cout << "身高:";
        cin >> baoBao[i].shenGao;

        cout << "体重:";
        cin >> baoBao[i].tiZhong;
    }

    for (int i = 0; i < BAOBAO_GESHU; ++i)
    {
        cout << "第 " << i + 1 << "个宝宝的数据如下" << endl;

        cout << "姓名:" << baoBao[i].xingMing << endl;

        cout << "身高:" << baoBao[i].shenGao << endl;
```

```
        cout << "体重:" << baoBao[i].tiZhong << endl;
    }

    system("Pause");

    return 0;
}
```

运行结果略。

22.6 结构与函数

int foo(int a); 是一个函数，其中 a 是参数，返回值是一个整数。
假充 foo 的实现如下：

```
int foo(int a)
{
    return a * 2;
}
```

那么， 代码:

int b = foo (100); 将使 b 值为 200。 请参看专门函数专门章节。本章中，我们将分别讲如何在参数及返回值中使用结构变量。

22.6.1 结构变量作为函数参数

函数的参数有两种传递方法，一种是传值，一种是传址。如果您觉得有些陌生了，那一定要复习一下函数参数的相关章节。

我们先来定义一个结构，这个结构用于表达一个四边形(SiFangXing)：

```
struct SiFangXing
{
    int l1,l2,w1,w2;  //四方形的四条边
};
```

现在我们要做两道题：
第一道：写一个函数，对所给的四方形求周长。
第二道：写一个函数，将所给的四方形长宽各增加一倍。
请大家在下面小节中，注意两个函数的参数形递有何不同，并思考各自的目的。

22.6.1.1 结构变量以传值的方式传递

第一道：写一个函数，对所给的四方形求周长。

```
//求周长：
int QiuZhouChang(SiFangXing sfx)
{
    return sfx.l1 + sfx.l2 + sfx.w1 + sfx.w2; //四边之和即是周长
}
```

22.6.1.2 结构变量以传址的方式传递

第二道：写一个函数，将所给的四方形长宽各增加一倍。

```
//加倍长宽
void JiaBeiChangKuan(SiFangXing& sfx)
{
   //各边都*2;
   sfx.l1 *= 2;
   sfx.l2 *= 2;
   sfx.w1 *= 2;
   sfx.w2 *= 2;
}
```

22.6.1.3 结构变量以常量传址方式传递

在 22.6.1 例中，由于我们并不需要在求周长的函数内改变所传入的四方形，所以我们采用“传值”方式。传值将复制一份实参，然后把“复制品”传给函数。这样就有了一个问题。以前我们写的函数的参数，多是像 int,char 之类的简单变量，它们占用的内存并不多，复制也快。但今天我们学习结构，结构往往由多个成员变量组成，占用内存较大，如果复制一份，就会显得浪费。并且，占用内存大的结构，在复制起来，比较占用时间。真是时间和空间双重浪费。

怎么办呢？首先想到的是，那就改成传址啊——

```
int QiuZhouChang(SiFangXing& sfx)
{
    return sfx.l1 + sfx.l2 + sfx.w1 + sfx.w2; //四边之和即是周长
}
```

采用传址方式后，传递给参数的，其实是实参的地址。而地址的大小只需要 4 个字节，相当于传一个整形数，又小又快。双重浪费的问题算是解决。但是这却带来了一个“隐患”。如果写 QiuZhouChang 函数的人一不小心，在该函数内修改了传入的参数，就会造成程序难以查找的错误：

```
int QiuZhouChang(SiFangXing& sfx)
{
   int zc = sfx.l1 + sfx.l2 + sfx.w1 + sfx.w2; //四边之和即是周长
```

```
    sfx.l1 *= 2;  //当时写这个函数的程序员发高烧，所以有了这行代码 :)

    return zc;
}
```

上面函数显然是错误，对一个四方形求周长的函数，怎么可以莫名地修改人家四方形的边长呢？并且由于参数我们在前面改为用“传址”方式，所以当一个四方形求完周长后，它的边长 1 竟然长了一倍长……

```
...
SiFangXing sfxA; //四方形 A
sfxA.l1 = sfxA.l2 = 10; //长 10
sfxA.w1 = sfxA.w2 = 5;  //宽 5
//求周长之前，输出四边长：
cout << "四边长：" << sfxA.l1 << "," << sfxA.l2 << ","
       << sfxA.w1 << "," << sfxA.w2 << endl;
int zhouChang = QiuZhouChang(sfxA); //求周长
cout << "周  长:" << zhouChang << endl;
//求周长之后，再输出四边长：
cout << "四边长：" << sfxA.l1 << "," << sfxA.l2 << ","
       << sfxA.w1 << "," << sfxA.w2 << endl;
...
```

上面的代码输出结果是：

```
四边长：10,10,5,5
周  长：30
四边长：20,10,5,5
```

你可能会说，我绝不会在发高烧时写代码，但要知道，如果一个函数体内的代码足够复杂，那么每个人都有可能在不发高烧的情况下，也写出愚蠢的代码来。C++提供了一种方法，让我们可更好的避免此类错误代码。这就是我们所说的“**常量传址**”。

```
//传址常量形式的参数
int QiuZhouChang (const SiFangXing& sfx)
{
    return sfx.l1 + sfx.l2 + sfx.w1 + sfx.w2;
}
```

1、上述参数形式中，参数 sfx 将以传址的方式来传递。传址方式避免了参数复制品造成的内存与速度的问题。符号“**&**”标明了它是使用传址。

2、**const** 修饰符 则指明 sfx 将被当作常量对待，该语法规定你不能在当前函数内修改这个参数——不管你是否发高烧。

请在 CB 时试着写下面的高烧代码：

```
int QiuZhouChang(const SiFangXing& sfx)
{
    int zc = sfx.l1 + sfx.l2 + sfx.w1 + sfx.w2; //四边之和即是周长
    sfx.l1 *= 2;  //编译时，该行会报错
    return zc;
}
```

编译器将拒绝通过上述代码。

趁热打铁，我们再来一个“常量传址”例子，就是上面的输出四方形各边长的代码：

```
//输出指定四方形的四边长：
void ShuChuSiBianChang(const SiFangXing& sfx)
{
   cout << "四边长: " << sfx.l1 << "," << sfx.l2 << "," << sfx.w1 << "," << sfx.w2 << endl;
}
```

像类似上述的两个函数参数，你可以拒绝使用 “&” 来指定使用传址；你也可以拒绝使用 const 来限制它是一个常量。你的代码可以工作，但它们效率不好；并且，你的代码越来越多时，你犯错误的机率很可能急剧地升高，直到整个程序乱成一团。

程序员要养成良好习惯。否则除了上述的问题外，当你与其他具备良好习惯的程序员一起写程序时，你会发现你的代码将无法和别人的代码很好地衔接，甚至不“兼容”。

C++ 和其它语言相比，就是它提供了很多强大的语法功能，但它并不强制你使用。其它的语言，有的是提供了同样的语法功能，并强制你使用；而有些则缺少必要的语法。之所以 C++是现在这个样子，有它的历史原因，比如说它必须兼容 C 语言。

22.6.1.4 兼容 C：使用指针传递结构变量

C 当然也只持最普通的“传值”方式：

```
int QiuZhouChang (SiFangXing sfx)
{
   return sfx.l1 + sfx.l2 + sfx.w1 + sfx.w2;
}
```

但前所言，“传值”有双浪费之弊。我们坚决反倒。微软的程序员也坚决反对——可以 Windows 中很大一部分就是拿 C 写的。而 C 既不支持使用“&”来实现传址方式，也不支持 const 修饰为“常量”。怎么办？答案可能很出乎你的意料：没办法。象写操作系统这类程序，“效率”永远是重中之重，所以只要冒着程序有层出不穷的 BUG 的危险，只考虑如效率了。

在 C 语言时，要实现传址，方法就是用指针。C++ 兼容这一点。

```
int QiuZhouChang(SiFangXing* psfx)
{
   return psfx->l1 + psfx->l2 + psfx->w1 + psfx->w2;  // . 换成了 ->,因为 psf 是指针
}
```

调用时，如果实参不是一个指针，就使用 & 来取址：

```
...
SiFangXing* sfxA;
sfxA.l1 = sfxA.l2 = 10;
sfxA.w1 = sfxA.w2 = 5;

int zc = QiuZhouChang(&sfxA); //& 用于得到 sfxA 地址
```

传指针就是传地址。效率问题解决了，但“高烧”代码编译器不会指出：

```
int QiuZhouChang(SiFangXing* psfx)
{
   int zc = psfx->l1 + psfx->l2 + psfx->w1 + psfx->w2;  // . 换成了 ->,因为 psf 是指针
   psfx->l1 *= 2;
```

```
    return zc;
}
```

编译器会认为，或许这就是你原来想要的。事实上，编译器读不懂英语，更不懂得中国拼音，它，怎么知道你是“QiuZhouChang”函数是要“求周长”呢？：P

22.6.2 函数返回值是结构类型

函数的返回值也可以一个结构变量。

让我们来实现这么一个功能：把四方形 A 和四方形 B 相加，得到四方形 C。相加的方法是长+长，宽+宽。

```
SiFangXing AddSiFangXing (const SiFangXing& aSfx, const SiFangXing& bSfx)
{
    SiFangXing cSfx;
    cSfx.l1 = aSfx.l1 + bSfx.l1;
    cSfx.w1 = aSfx.w1 + bSfx.w1;
    cSfx.l2 = aSfx.l2 + bSfx.l2;
    cSfx.w2 = aSfx.w2 + bSfx.w2;
    return cSfx;
}
```

调用样例为：

```
...
SiFangXing a,b,c;
a.l1 = a.l2 = 10;
a.w1 = a.w2 = 5;
b.l1 = b.l2 = 20;
b.w1 = b.w2 = 18;
  c = AddSiFangXing ( a, b );
  ShuChuSiBianChang(c); //输出，见 22.6.3 节
...
```

上这代码结果为：

```
四边长： 30,30,23,23
```

22.7 作业

一、本章中的所有例程，请都在 CB 上演练一番。

二、写一个求四方形面积的函数。

三、请写一个将四方形长和宽对调的函数。

四、请写一个函数，输入两个四方形，返回其中面积较大者。

五、请定义一个“圆”的结构。并写相应的三个函数实现：1、求圆周长，2、求圆面积，3、让指定的圆周长增加一倍。

第二十三章　类 / class （一） 封装

23.1 从“我吃饭”开始

我吃饭……

其中，“我”是一个变量，“我”的类型是“人类”；

“吃”是一个函数。

“饭”也是一个变量，它的类型是“食物”。这里用于做函数“吃”的参数。

“我吃饭”！这是一种面向对象的思想的表达。其中的对象是“我”。以我为中心。我因为是人类，所以具备“吃”这种能力。如果说“桌子吃饭”，那么编译器会报错，因为桌子属于家具类，而家具不提供“吃”的函数。

C++是一种具备面向对象能力的编程语言，所以，用 C++来表达“我吃饭”这样一件事时，它的代码风格贴近这种人类的自然语言，即：我.吃（饭）；“我”是一个对象，“吃”是“我”所属对象（人类）的一个函数，而“饭”是函数参数。

换成 C 语言，因为它不具备面向对象的设计思想，所以，它只能说成：“吃（我，饭）”。“吃”是函数，“我”和“饭”是两个参数。没有人规则一定要把“我”作为第一个参数，你尽可写成“吃（饭，我）”。二者比较，面向对象的最基本的好处或许您已经有所体会：自然，从而不容易出错。

23.2 从“结构”到“类”

上一章我们学习了结构（struct）。（强烈建议你暂时放下新课程，重温一下 struct）。

结构让我们具备了把多种相同或不同的类型，组成一种新类型的能力。

比如上一章讲的“宝宝 / BaoBao”这一结构，它的组合为：

```
struct BaoBao
{
    char xingMing[11]; //用字符数组，来存储姓名
    int shenGao; //身高，整型
    float tiZhong; //体重，实型
};
```

通过 struct，我们通过将简单数据类型（int,float,bool……）或简单数据类型的数组、指针，组合成一个新的**数据类型**。由此，我们在用程序表达复杂的现实世界时，更接近了一步。但是别忘了，我们说过世界是由“数据”和“动作”组成的。光能定义出各种数据类型，还只是编程世界的一半。你可能会说，我们有函数啊，函数不是可以表达“动作”？

没错，比如说，宝宝肯定有“吃”的动作，所以我来声明一个“吃”的函数。为了直观，我们的函数命名为“Chi”。并且我们假充有一种数据类型叫“饭”，同样用拼音 Fan 表示，首字母大写，而小写的 fan 用来做形参。

```
void Chi(Fan fan) ;
```

乍一看，感觉这个函数这样声明也就对了。“吃饭”函数嘛，有“吃”又有“饭”……可仔细一想，谁吃饭啊？这个光有吃的动作和吃的东西，和我们前面的“宝宝”数据类型有何关系？所以，显然不够，需要再加一个参数，用于传一个“要吃饭”的宝宝进去。因此函数变成：

```
void Chi(BaoBao bb, Fan fan);
```

得，“吃我饭”的表达出来了。过程化的编程，其设计重在关心“过程”，比如：“如何吃”，但这个世界要求我们完整地关心“谁？如何吃？吃什么？”。事实上，同一样一个“吃”，宝宝吃的动作，和一个大男人吃的动作；或者，吃饭还是吃奶的动作？怕是完全不一样。如果真写，就不得不写很多版本的“吃”这一函数：

假设用 DNR 表大男人(晕，好像大女人也可以？)：

```
void DNRChi(DNR dnr, PingGuo pg); //吃函数版本一：大男人吃苹果
void DNRChi(DNR dnr，Fan fan); //吃函数版本二：大男人吃饭
void BaoBaoChi(BaoBao bb, Nai nai); //吃函数版本三：宝宝吃奶
void BaoBaoChi(BaoBao bb, Fan fan); //吃函数版本四：宝宝吃饭
……
```

这样的函数还可以有很多。函数多或许并没有错，必竟所要表达事物本来就复杂。然而问题是我们如何去理解，区分，记忆这些函数呢？仅靠函数名和函数参数的不同吗？在超市付款时，看过收款员拉开过放钱的小抽屉吗？拉开一看，都是钱，但 10 元的 5 元，100 元的及硬币分门别类地放好……聪明的你一定会提出：如果能把函数也归类就好了……这就有了“类”，英文称为：class。

请仔细看，下面示例的 class 定义里，加入了一个函数：

```
class BaoBao
{
    char xingMing[11]; //用字符数组，来存储姓名
    int shenGao; //身高，整型
    float tiZhong; //体重，实型
      void Chi(Fan fan); //加入“吃”的函数。
};
```

这算是一个“突变”——我们一直说着的“动作”与“数据”从这里开始合二为一，表面看来或许不过如此：无非是在类的定义里，同时可以包括数据及函数。然而却由此开启了“面向对象”世界之门。如果你喜欢武侠，那你可以把它看成一门语言打通了任督二脉……

类的数据定义里，出现函数，那么，这个函数的**声明**它占用类的大小吗？

来看两个数据定义，前者是 struct，后者是 class。前者没有包括函数，后者包括一个函数。其余的数据定义完全一样。

struct SBaoBao	class CBaoBao
{	{
char xingMing[11];	char xingMing[11];
int shenGao;	int shenGao;

```
    float tiZhou;
};
```

```
    float tiZhou;

    void Chi(int a); //参数可不能用 Fan 了
};
```

然后，我们来做个比较：

[略]

23.3 类的成员数据与成员函数

成员？长这么大，肯定填写过什么“家庭成员”的样的表格。我们进行的数据定义或函数定义，它们之所以前面有没加个“成员”的修饰，是因为它们都没有一个家，哎，谁不想有个家呢？在 C#和 Java 里，所有数据及函数都必须有个家才可以存在，而在 C++里，允许数据或函数可以没有家（不属于某个类）；也可以允许有个家（属于某个类）。

```
  class CBaoBao
{
   //下面就是 CBaoCBao 类的成员数据：
   char xingMing[11];
   int shenGao;
   float tiZhou;
     //而这个是 CBaoBao 类的成员函数：
   void Chi(int a);
};
```

类的成员数据和成员函数，都称为类的成员（像是一句废话？）。

23.3.1 成员数据初始化的疑问

我们以前常有这样的代码：

```
  int a = 100; //定义一个变量，同时给它赋值为 100.
```

可是，你一定一定要明白了，当我们在定义一个类或一个结构时，我们只是在“**组装**”一个新的数据类型。而并没有实际定义一个变量，所以 C＋＋不允许在定义一个类的内部，对它的成员数据赋值。下面的代码是错误的：

```
  //在定义一个类时，试图初始化它的成员数据！不行！
class CBaoBao
{
    int shenGao = 70; //不行！
    ...
};
```

那么，当我们定义某一个类的具体变量时，这个变量里的成员数据初始值是多少？理论上，它们将是随机的值，也就是不能预定的值。

```
  ...
CBaoCBao baobao1;
```

...

定义完 baobao1 之后，bao1bao1 里的 shenGao 是多少？不知道！

那么，如何给某个类变量的成员数据一个初始的值（即默认的值）？下章我们会讲到。

23.3.2 成员函数的实现

我们可以在类里头加了函数（声明了类的成员函数），比如在前面的 BaoBao 类加了函数：Chi(...)；可是我们还没有实现这个函数呢。

一个类的成员函数一般在类的外部定义，但要注意它和普通函数定义时的区别。

```
//定义，或称为实现 CBaoBao 的成员函数：Chi:
void CBaoBao::Chi(int a) //和普通函数区别：类的成员函数定义时，必须加上类名和::
{
  //...和普通函数一样，这里是函数体
}//和普通函数一样，这里没有分号
```

类成员函数也可以直接在类的体内定义，但此时就不必写类名加::了。

```
class CBaoBao
{
  //下面就是 CBaoCBao 类的成员数据：
  char xingMing[11];
  int shenGao;
  float tiZhou;
    //而这个是 CBaoBao 类的成员函数：
  //并且直接定义：
  void Chi(int a) //没有分号
  {
     //函数体...
  } //没有分号了
};
```

不过，直接在类体内的定义的成员函数，将被默认当作 inline(内联）函数。关于内联函数，大家可以找一找前面函数章节。

23.4 封装

从有了“类”开始，C++的世界越来越有趣了。前面说类就像一个家，家里有成员（数据或函数），现在，我们还要讲“访问”类的成员……想像有个类叫“美女”。

```
class MeiNu //美女类!
{
  int XW;  //胸围
  int YW;  //我就不说了噢 :)
  int TW;  //我还是不说了噢 :))
};
```

我看到部分学员在想入非非了，这可不行。请打开 CB，新建一个“控制台/Console”工程。然后把上

面新建工程后默认出现的 Unit1.cpp 中的 main()函数之上。（偷偷说一声，后面的章节里，我们学习 C++的也可开始慢慢有在 Windows 下的工程了！因为我们学习类了嘛，任督二脉都打通了，当然得来点更有意思的……）

Unit1.cpp

```
#pragma argsused

class MeiNu //美女类!
{
    int XW;  //胸围
    int YW;  //我就不说了噢 :)
    int TW;  //我还是不说了噢 :))
};

int main(int argc, char* argv[])
{
    return 0;
}
//---------------------------------------
```

（在 Unit1.cpp 里加入 MeiNu 类）

然后……当然是定义一个美女了！我们就在 main 函数里定义了，我不贴图了，你们对着课程，自己往 CB 里添代码。

```
int main(int argc, char* argv[])
{
    MeiNu zhaoWei; //美女赵?

    return 0;
}
```

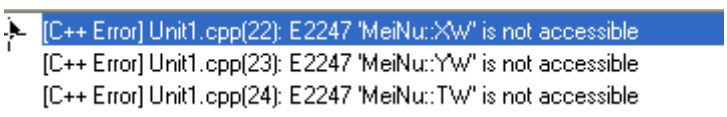
现在开始有些为难了，赵薇的三围是多少？甭说她了，一般地说，通常美女的三围是多少啊？上网查一下吧……

哈哈，终于找到了，不过是三版女朗乔丹的。郁闷中……88 了赵薇。现在代码为：

```
...
MeiNu jordan; //now is 乔丹!

jordan.XW = 34;
jordan.YW = 24;
jordan.TW = 34;
...
```

按 Ctrl+Shift+S，保存 Unit1.cpp 和工程，工程我就命令为 MeiNuPrj.bpr 了。

然后按 Ctrl + F9.试图编译一下！可是，可是，编译好像说：对不起，乔丹的胸围无法访问…（如图）



（not accessible 就是 “无法接触到”，或“无法访问” ）

我们想给乔丹设置一下三围，可是 C++编译器竟义正辞严地拒绝了！这是怎么回事？

因为，类/class 对它的成员（数据或函数），有“保护”机制。不允许“外人”随便访问到它的成员。**这也就是传说中“面向对象”的三大基石之一“封装性”**。

［略］

23.4.1 私有成员/private member

［略］

从 private:开始，后面本类的成员数据或函数，都将是私有的，除非我们又加了一个新的访问等级限制关键字。

23.4.2 保护成员/protected member

保护成员也不能在类的外部直接访问，但可以在该类的子类（或称为派生类）中访问。所谓子类或派生类，我们后面的章节才会讲到。大致的意思，先不妨认为是，你们家的东西，外人不能用，但你儿子或儿媳（他有自己的家）可以用……这是不合适的比喻。只是为了感性理解一下私有和保护的一种区别。

23.4.3 公有成员/public member

［略］

23.4.4 “封装”的作用

说着说着，这问题就来了。为什么要用 private 或 protected 来保护类的成员啊？大家都是公有的，都可以直接访问，多方便啊？这问题如果反过来问，就是面向对象三大基石之一“封装”有什么好处？

封装的好处，两点，并且两点相辅相成。

［略］

23.5 作业

［略］