C++从零开始（一）

——何谓编程

引言

曾经有些人问我问题，问得都是一些很基础的问题，但这些人却已经能使用 VC 编一个对话框界面来进行必要的操作或者是文档/视界面来实时接收端口数据并动态显示曲线（还使用了多线程技术），却连那些基础的问题都不清楚，并且最严重的后果就是导致编写出拙劣的代码（虽然是多线程，但真不敢恭维），不清楚类的含义，混杂使用各种可用的技术来达到目的（连用异常代替选择语句都弄出来了），代码逻辑混乱，感觉就和金山快译的翻译效果一样。

我认为任何事情，基础都是最重要的，并且在做完我自定的最后一个项目后我就不再做编程的工作，守着这些经验也没什么意义，在此就用本系列说说我对编程的理解，帮助对电脑编程感兴趣的人快速入门（不过也许并不会想象地那么快）。由于我从没正经看完过一本 C++的书（都是零碎偶尔翻翻的），并且本系列并不是教条式地将那些该死的名词及其解释罗列一遍，而是希望读者能够理解编程，而不是学会一门语言（即不止会英翻汉，还会汉翻英）。整个系列全用我自己的理解来写的，并无参考其他教材（在一些基础概念上还是参考了 MSDN），所以本系列中的内容可能有和经典教材不一致的地方，但它们的本质应该还是一样的，只是角度不同而已。本系列不会仔细讲解 C++的每个关键字（有些并不重要），毕竟目的不是 C++语言参考，而是编程入门。如果本系列文章中有未提及的内容，还请参考 MSDN 中的 C++语言参考（看完本系列文章后应该有能力做这件事了），而本系列给出的内容均是以 VC 编译器为基础，基于 32 位 Windows 操作系统的。

何谓程序

程序，即过程的顺序，准确地说应该是顺序排列的多个过程，其是方法的描述。比如吃菜，先用筷子夹起菜，再用筷子将菜送入嘴中，最后咀嚼并吞下。其中的夹、送、咀嚼和吞下就被称作命令，而菜则是资源，其状态（如形状、位置等）随着命令的执行而不断发生变化。上面就是吃菜这个方法的描述，也就是吃菜的程序。

任何方法都是为了改变某些资源的状态而存在，因此任何方法的描述，也就是程序，也都一定有命令这个东西以及其所作用的资源。命令是由程序的执行者来实现的，比如上面的吃菜，其中的夹、送等都是由吃菜的人来实现的，而资源则一定是执行者可以改变的东西，而命令只是告诉执行者如何改变而已。

电脑程序和上面一样，是方法的描述，而这些方法就是人期望电脑能做的事（注意不是电脑要做的事，这经常一直混淆着许多人），当人需要做这些事时，人再给出某些资源以期电脑能对其做正确的改变。如计算圆周率的程序，其只是方法的描述，本身是不能发生任何效用的，直到它被执行，人为给定它一块内存（关于内存，请参考《C++从零开始（三）》），告诉它计算结果的精度及计算结果的存放位置后，其才改变人为给定的这块内存的状态以表现出计算结果。

因此，对于电脑程序，命令就是 CPU 的指令，而执行者也就由于是 CPU 的指令而必须是 CPU 了，而最后的资源则就是 CPU 可以改变其状态的内存（当然不止，如端口等，不过

一般应用程序都大量使用内存罢了）。所以，电脑程序就是电脑如何改变给定资源（一般是内存，也可以是其他硬件资源）的描述，注意是描述，本身没有任何意义，除非被执行。

何谓编程

编程就是编写程序，即制订方法。为什么要有方法？方法是为了说明。而之所以要有说明就有很多原因了，但电脑编程的根本原因是因为语言不同，且不仅不同，连概念都不相通。

人类的语言五花八门，但都可以通过翻译得到正解，因为人类生存在同一个四维物理空间中，具有相同或类似的感知。而电脑程序执行时的 CPU 所能感受到的空间和物理空间严重不同，所以是不可能将电脑程序翻译成人类语言的描述的。这很重要，其导致了大部分程序员编写出的拙劣代码，因为人想的和电脑想的没有共性，所以他们在编写程序时就随机地无目的地编写，进而导致了拙劣却可以执行的代码。

电脑的语言就是 CPU 的指令，因为 CPU 就这一个感知途径（准确地说还有内存定位、中断响应等感知途径），不像人类还能有肢体语言，所以电脑编程就是将人类语言书写的方法翻译成相应的电脑语言，是一个翻译过程。这完全不同于一般的翻译，由于前面的红字，所以是不可能翻译的。

既然不可能翻译，那电脑编程到底是干甚？考虑一个木匠，我是客人。我对木匠说我要一把摇椅，躺着很舒服的那种。然后木匠开始刨木头，按照一个特殊的曲线制作摇椅下面的曲木以保证我摇的时候重心始终不变以感觉很舒服。这里我编了个简单的程序，只有一条指令——做一把摇着很舒服的摇椅。而木匠则将我的程序翻译成了刨木头、设计特定的曲木等一系列我看不懂的程序。之所以会这样，在这里就是因为我生活的空间和木工（是木工工艺，不是木匠）没有共性。这里木匠就相当于电脑程序员兼 CPU（因为最后由木匠来制作摇椅），而木匠的手艺就是 CPU 的指令定义，而木匠就将我的程序翻译成了木工的一些规程，由木匠通过其手艺来实现这些规程，也就是执行程序。

上面由于我生活的空间和木工（指木工工艺，不是工人）没有共性，所以是不可能翻译的，但上面翻译成功了，实际是没有翻译的。在木工眼中，那个摇椅只是一些直木和曲木的拼接而已，因为木工空间中根本没有摇椅的概念，只是我要把那堆木头当作摇椅，进而使用。如果我把那堆木头当作凶器，则它就是凶器，不是什么摇椅了。

“废话加荒谬加放屁！”，也许你会这么大叫，但电脑编程就是这么一回事。CPU 只能感知指令和改变内存的状态（不考虑其他的硬件资源及响应），如果我们编写了一个计算圆周率的程序，给出了一块内存，并执行，完成后就看见电脑的屏幕显示正确的结果。但一定注意，这里电脑实际只是将一些内存的数值复制、加减、乘除而已，电脑并不知道那是圆周率，而如果执行程序的人不把它说成是圆周率那么那个结果也就不是圆周率了，可能是一个随机数或其他什么的，只是运气极好地和圆周率惊人地相似。

上面的东西我将其称为语义，即语言的意义，其不仅仅可应用在电脑编程方面，实际上许多技术，如机械、电子、数学等都有自己的语言，而那些设计师则负责将客户的简单程序翻译成相应语言描述的程序。作为一个程序员是极其有必要了解到语义的重要性的（我在我的另一篇文章《语义的需要》中对代码级的语义做过较详细的阐述，有兴趣可以参考之），在后续的文章中我还将提到语义以及其对编程的影响，如果你还没有理解编程是什么意思，随着后续文章的阅读应该能够越来越明了。

电脑编程的基础知识——编译器和连接器

我从没见过（不过应该有）任何一本 C++教材有讲过何谓编译器（Compiler）及连接器（Linker）（倒是在很老的 C 教材中见过），现在都通过一个类似 VC 这样的编程环境隐藏了大量东西，将这些封装起来。在此，对它们的理解是非常重要的，本系列后面将大量运用到这两个词汇，其决定了能否理解如声明、定义、外部变量、头文件等非常重要的关键。

前面已经说明了电脑编程就是一个“翻译”过程，要把用户的程序翻译成 CPU 指令，其实也就是机器代码。所谓的机器代码就是用 CPU 指令书写的程序，被称作低级语言。而程序员的工作就是编写出机器代码。由于机器代码完全是一些数字组成（CPU 感知的一切都是数字，即使是指令，也只是 1 代表加法、2 代表减法这一类的数字和工作的映射），人要记住 1 是代表加法、2 是代表减法将比较困难，并且还要记住第 3 块内存中放的是圆周率，而第 4 块内存中放的是有效位数。所以发明了汇编语言，用一些符号表示加法而不再用 1 了，如用 ADD 表示加法等。

由于使用了汇编语言，人更容易记住了，但是电脑无法理解（其只知道 1 是加法，不知道 ADD 是加法，因为电脑只能看见数字），所以必须有个东西将汇编代码翻译成机器代码，也就是所谓的编译器。即编译器是将一种语言翻译成另一种语言的程序。

即使使用了汇编语言，但由于其几乎只是将 CPU 指令中的数字映射成符号以帮助记忆而已，还是使用的电脑的思考方式进行思考的，不够接近人类的思考习惯，故而出现了纷繁复杂的各种电脑编程语言，如：PASCAL、BASIC、C 等，其被称作高级语言，因为比较接近人的思考模式（尤其 C++的类的概念的推出），而汇编语言则被称作低级语言（C 曾被称作高级的低级语言），因为它们不是很符合人类的思考模式，人类书写起来比较困难。由于 CPU 同样不认识这些 PASCAL、BASIC 等语言定义的符号，所以也同样必须有一个编译器把这些语言编写的代码转成机器代码。对于这里将要讲到的 C++语言，则是 C++语言编译器（以后的编译器均指 C++语言编译器）。

因此，这里所谓的编译器就是将我们书写的 C++源代码转换成机器代码。由于编译器执行一个转换过程，所以其可以对我们编写的代码进行一些优化，也就是说其相当于是一个 CPU 指令程序员，将我们提供的程序翻译成机器代码，不过它的工作要简单一些了，因为从人类的思考方式转成电脑的思考方式这一过程已经由程序员完成了，而编译器只是进行翻译罢了（最多进行一些优化）。

还有一种编译器被称作翻译器（Translator），其和编译器的区别就是其是动态的而编译器是静态的。如前面的 BASIC 的编译器在早期版本就被称为翻译器，因为其是在运行时期即时进行翻译工作的，而不像编译器一次性将所有代码翻成机器代码。对于这里的“动态”、“静态”和“运行时期”等名词，不用刻意去理解它，随着后续文章的阅读就会了解了。

编译器把编译后（即翻译好的）的代码以一定格式（对于 VC，就是 COFF 通用对象文件格式，扩展名为.obj）存放在文件中，然后再由连接器将编译好的机器代码按一定格式（在 Windows 操作系统下就是 Portable Executable File Format——PE 文件格式）存储在文件中，以便以后操作系统执行程序时能按照那个格式找到应该执行的第一条指令或其他东西，如资源等。至于为什么中间还要加一个连接器以及其它细节，在后续文章中将会进一步说明。

也许你还不能了解到上面两个概念的重要性，但在后续的文章中，你将会发现它们是如此的重要以至于完全有必要在这唠叨一番。

C++从零开始（二）

C++从零开始（二）

——何谓表达式

本篇是此系列的开头，在学英语时，第一时间学的是字母，其是英语的基础。同样，在C++中，所有的代码都是通过标识符（Identifier）、表达式（Expression）和语句（Statement）及一些必要的符号（如大括号等）组成，在此先说明何谓标识符。

标识符

标识符是一个字母序列，由大小写英文字母、下划线及数字组成，用于标识。标识就是标出并识别，也就是名字。其可以作为后面将提到的变量或者函数或者类等的名字，也就是说用来标识某个特定的变量或者函数或者类等C++中的元素。

比如：abc就是一个合法的标识符，即abc可以作为变量、函数等元素的名字，但并不代表abc就是某个变量或函数的名字，而所谓的合法就是任何一个标识符都必须不能以数字开头，只能包括大小写英文字母、下划线及数字，不能有其它符号，如,!^等，并且不能与C++关键字相同。也就是我们在给一个变量或函数起名字的时候，必须将起的名字看作是一个标识符，并进而必须满足上面提出的要求。如12ab_C就不是一个合法的标识符，因此我们不能给某个变量或函数起12ab_C这样的名字；ab_12C就是合法的标识符，因此可以被用作变量或函数的名字。

前面提到关键字，在后续的语句及一些声明修饰符的介绍中将发现，C++提供了一些特殊的标识符作为语句的名字，用以标识某一特定语句，如if、while等；或者提供一些修饰符用以修饰变量、函数等元素以实现语义或给编译器及连接器提供一些特定信息以进行优化、查错等操作，如extern、static等。因此在命名变量或函数或其他元素时，不能使用if、extern等这种C++关键字作为名字，否则将导致编译器无法确认是一个变量（或函数或其它C++元素）还是一条语句，进而无法编译。

如果要让某个标识符是特定变量或函数或类的名字，就需要使用声明，在后续的文章中再具体说明。

数字

C++作为电脑编程语言，电脑是处理数字的，因此C++中的基础东西就是数字。C++中提供两种数字：整型数和浮点数，也就是整数和小数。但由于电脑实际并不是想象中的数字化的（详情参见《C++从零开始（三）》中的类型一节），所以整型数又分成了有符号和无符号整型数，而浮点数则由精度的区别而分成单精度和双精度浮点数，同样的整型数也根据长度分成长整型和短整型。

要在C++代码中表示一个数字，直接书写数字即可，如：123、34.23、-34.34等。由于电脑并非以数字为基础而导致了前面数字的分类，为了在代码中表现出来，C++提供了一系列的后缀进行表示，如下：

u或U　表示数字是无符号整型数，如：123u，但并不说明是长整型还是短整型

l或L　表示数字是长整型数，如：123l；而123ul就是无符号长整型数；而34.4l就是长双精度浮点数，等效于双精度浮点数

i64或I64　表示数字是长长整型数，其是为64位操作系统定义的，长度比长整型数长。如：

43i64
f 或 F　表示数字是单精度浮点数，如：12.3f
e 或 E　表示数字的次幂，如：34.4e-2 就是 0.344；0.2544e3f 表示一个单精度浮点数，值为 254.4

当什么后缀都没写时，则根据有无小数点及位数来决定其具体类型，如：123 表示的是有符号整型数，而 12341434 则是有符号长整型数；而 34.43 表示双精度浮点数。

为什么要搞这么多事出来，还分什么有符号无符号之类的？这全是因为电脑并非基于数字的，而是基于状态的，详情在下篇中将详细说明。

作为科学计算，可能经常会碰到使用非十进制数字，如 16 进制、8 进制等，C++也为此提供了一些前缀以进行支持。

在数字前面加上 0x 或 0X 表示这个数字是 16 进制表示的，如：0xF3Fa、0x11cF。而在前面加一个 0 则表示这个数字是用 8 进制表示的，如：0347，变为十进制数就为 231。但 16 进制和 8 进制都不能用于表示浮点数，只能表示整型数，即 0x34.343 是错误的。

字符串

C++除了提供数字这种最基础的表示方式外，还提供了字符及字符串。这完全只是出于方便编写程序而提供的，C++作为电脑语言，根本没有提供字符串的必要性。不过由于人对电脑的基本要求就是显示结果，而字符和字符串都由于是人易读的符号而被用于显示结果，所以 C++专门提供了对字符串的支持。

前面说过，电脑只认识数字，而字符就是文字符号，是一种图形符号。为了使电脑能够处理符号，必须通过某种方式将符号变成数字，在电脑中这通过在符号和数字之间建立一个映射来实现，也就是一个表格。表格有两列，一列就是我们欲显示的图形符号，而另一列就是一个数字，通过这么一张表就可以在图形符号和数字之间建立映射。现在已经定义出一标准表，称为 ASCII 码表，几乎所有的电脑硬件都支持这个转换表以将数字变成符号进而显示计算结果。

有了上面的表，当想说明结果为“A”时，就查 ASCII 码表，得到“A”这个图形符号对应的数字是 65，然后就告诉电脑输出序号为 65 的字符，最后屏幕上显示“A”。

这明显地繁杂得异常，为此 C++就提供了字符和字符串。当我们想得到某一个图形符号的 ASCII 码表的序号时，只需通过单引号将那个字符括起来即可，如：'A'，其效果和 65 是一样的。当要使用不止一个字符时，则用双引号将多个字符括起来，也就是所谓的字符串了，如："ABC"。因此字符串就是多个字符连起来而已。但根据前面的说明易发现，字符串也需要映射成数字，但它的映射就不像字符那么简单可以通过查表就搞定的，对于此，将在后续文章中对数组作过介绍后再说明。

操作符

电脑的基本是数字，那么电脑的所有操作都是改变数字，因此很正常地 C++提供了操作数字的一些基本操作，称作操作符（Operator），如：+ - * / 等。任何操作符都要返回一个数字，称为操作符的返回值，因此操作符就是操作数字并返回数字的符号。作为一般性地分类，按操作符同时作用的数字个数分为一元、二元和三元操作符。

一元操作符有：

+　其后接数字，原封不动地返回后接的数字。如：　+4.4f 的返回值是 4.4；+-9.3f 的返回值是-9.3。完全是出于语义的需要，如表示此数为正数。

-　其后接数字，将后接的数字的符号取反。如：　-34.4f 的返回值是-34.4；-(-54)的返回值是 54。用于表示负数。

!　其后接数字，逻辑取反后接的数字。逻辑值就是“真”或“假”，为了用数字表示逻辑值，在 C++中规定，非零值即为逻辑真，而零则为逻辑假。因此 3、43.4、'A'都表示逻辑真，而 0 则表示逻辑假。逻辑值被应用于后续的判断及循环语句中。而逻辑取反就是先判断“!”后面接的数字是逻辑真还是逻辑假，然后再将相应值取反。如：

!5 的返回值是 0，因为先由 5 非零而知是逻辑真，然后取反得逻辑假，故最后返回 0。

!!345.4 的返回值是 1，先因 345.4 非零得逻辑真，取反后得逻辑假，再取反得逻辑真。虽然只要非零就是逻辑真，但作为编译器返回的逻辑真，其一律使用 1 来代表逻辑真。

~　其后接数字，取反后接的数字。取反是逻辑中定义的操作，不能应用于数字。为了对数字应用取反操作，电脑中将数字用二进制表示，然后对数字的每一位进行取反操作（因为二进制数的每一位都只能为 1 或 0，正好符合逻辑的真和假）。如~123 的返回值就为-124。先将 123 转成二进制数 01111011，然后各位取反得 10000100，最后得-124。

这里的问题就是为什么是 8 位而不是 16 位二进制数。因为 123 小于 128，被定位为 char 类型，故为 8 位（关于 char 是什么将下篇介绍）。如果是~123ul，则返回值为 4294967172。

为什么要有数字取反这个操作？因为 CPU 提供了这样的指令。并且其还有着很不错且很重要的应用，后面将介绍。

关于其他的一元操作符将在后续文章中陆续提到（但不一定全部提到）。

二元操作符有：

+

-

*

/

%　其前后各接一数字，返回两数字之和、差、积、商、余数。如：

34+4.4f 的返回值是 38.4；3+-9.3f 的返回值是-6.3。

34-4 的返回值是 30；5-234 的返回值是-229。

3*2 的返回值是 6；10/3 的返回值是 3。

10%3 的返回值是 1；20%7 的返回值是 6。

&&

||　其前后各接一逻辑值，返回两逻辑值之“与”运算逻辑值和“或”运算逻辑值。如：

'A'&&34.3f 的返回值是逻辑真，为 1；34&&0 的返回值是逻辑假，为 0。

0||'B'的返回值是逻辑真，为 1；0||0 的返回值是逻辑假，为 0。

&

|

^　其前后各接一数字，返回两数字之“与”运算、“或”运算、“异或”运算值。如前面所说，先将两侧的数字转成二进制数，然后对各位进行与、或、异或操作。如：

4&6 的返回值是 4，4 转为 00000100，6 转为 00000110 各位相与得，00000100，为 4。

4|6 的返回值是 6，4 转为 00000100，6 转为 00000110 各位相或得，00000110，为 6。

4^6 的返回值是 2，4 转为 00000100，6 转为 00000110 各位相异或得，00000010，为 2。

>

<

==

>=

<=

!= 其前后各接一数字，根据两数字是否大于、小于、等于、大于等于、小于等于及不等于而返回相应的逻辑值。如：

34>34 的返回值是 0，为逻辑假；32<345 的返回值为 1，为逻辑真。

23>=23 和 23>=14 的返回值都是 1，为逻辑真；54<=4 的返回值为 0，为逻辑假。

56==6 的返回值是 0，为逻辑假；45==45 的返回值是 1，为逻辑真。

5!=5 的返回值是 0，为逻辑假；5!=35 的返回值是真，为逻辑真。

>>

<< 其前后各接一数字，将左侧数字右移或左移右侧数字指定的位数。与前面的 ~、&、| 等操作一样，之所以要提供左移、右移操作主要是因为 CPU 提供了这些指令，主要用于编一些基于二进制数的算法。

<<将左侧的数字转成二进制数，然后将各位向左移动右侧数值的位数，如：4，转为 00000100，左移 2 位，则变成 00010000，得 16。

>>与<<一样，只不过是向右移动罢了。如：6，转为 00000110，右移 1 位，变成 00000011，得 3。如果移 2 位，则有一位超出，将截断，则 6>>2 的返回值就是 00000001，为 1。

左移和右移有什么用？用于一些基于二进制数的算法，不过还可以顺便作为一个简单的优化手段。考虑十进制数 3524，我们将它左移 2 位，变成 352400，比原数扩大了 100 倍，准确的说应该是扩大了 10 的 2 次方倍。如果将 3524 右移 2 位，变成 35，相当于原数除以 100 的商。

同样，前面 4>>2，等效于 4/4 的商；32>>3 相当于 32/8，即相当于 32 除以 2 的 3 次方的商。而 4<<2 等效于 4*4，相当于 4 乘以 2 的 2 次方。因此左移和右移相当于乘法和除法，只不过只能是乘或除相应进制数的次方罢了，但它的运行速度却远远高于乘法和除法，因此说它是一种简单的优化手段。

, 其前后各接一数字，简单的返回其右侧的数字。如：

34.45f,54 的返回值是 54；-324,4545f 的返回值是 4545f。

那它到底有什么用？用于将多个数字整和成一个数字，在《C++从零开始（四）》中将进一步说明。

关于其他的二元操作符将在后续文章中陆续提到（但不一定全部提到）。

三元操作符只有一个，为?:，其格式为：<数字 1>?<数字 2>:<数字 3>。它的返回值为：如果<数字 1>是逻辑真，返回<数字 2>，否则返回<数字 3>。如：

34?4:2 的返回值就是 4，因为 34 非零，为逻辑真，返回 4。而 0?4:2 的返回值就是 2，因为 0 为逻辑假，返回 2。

表达式

你应该发现前面的荒谬之处了——12>435 返回值为 0，那为什么不直接写 0 还吃饱了撑了写个 12>435 在那？这就是表达式的意义了。

前面说“>”的前后各接一数字，但是操作符是操作数字并返回数字的符号，因为它返回数字，因此可以放在上面说的任何一个要求接数字的地方，也就形成了所谓的表达式。如：23*54/45>34 的返回值就是 0，因为 23*54 的返回值为 1242；然后又将 1242 作为“/”的左接数字，得到新的返回值 27.6；最后将 27.6 作为“>”的左接数字进而得到返回值 0，为逻辑假。

因此表达式就是由一系列返回数字的东西和操作符组合而成的一段代码，其由于是由操作符组成的，故一定返回值。而前面说的“返回数字的东西”则可以是另一个表达式，或者一个变量，或者一个具有返回值的函数，或者具有数字类型操作符重载的类的对象等，反正只要是能返回一个数字的东西。如果对于何谓变量、函数、类等这些名词感到陌生，不需要去管它们，在后继的文章中将会一一说明。

因此 34 也是一个表达式，其返回值为 34，只不过是没有操作符的表达式罢了（在后面将会了解到 34 其实是一种操作符）。故表达式的概念其实是很广的，只要有返回值的东西就可以称为表达式。

由于表达式里有很多操作符，执行操作符的顺序依赖于操作符的优先级，就和数学中的一样，*、/的优先级大于+、-，而+、-又大于>、<等逻辑操作符。不用去刻意记住操作符的优先级，当不能确定操作符的执行顺序时，可以使用小括号来进行指定。如：

((1+2)*3)+3)/4 的返回值为 3，而 1+2*3+3/4 的返回值为 7。注意 3/4 为 0，因为 3/4 的商是 0。当希望进行浮点数除法或乘法时，只需让操作数中的某一个为浮点数即可，如：3/4.0 的返回值为 0.75。

& | ^ ~等的应用

C++从零开始（三）

——何谓变量

本篇说明内容是 C++中的关键，基本大部分人对于这些内容都是昏的，但这些内容又是编程的基础中的基础，必须详细说明。

数字表示

数学中，数只有数值大小的不同，绝不会有数值占用空间的区别，即数学中的数是逻辑上的一个概念，但电脑不是。考虑算盘，每个算盘上有很多列算子，每列都分成上下两排算子。上排算子有 2 个，每个代表 5，下排算子有 4 个，每个代表 1（这并不重要）。因此算盘上的每列共有 6 个算子，每列共可以表示 0 到 14 这 15 个数字（因为上排算子的可能状态有 0 到 2 个算子有效，而下排算子则可能有 0 到 4 个算子有效，故为 3×5=15 种组合方式）。

上面的重点就是算盘的每列并没有表示 0 到 14 这 15 个数字，而是每列有 15 种状态，因此被人利用来表示数字而已（这很重要）。由于算盘的每列有 15 个状态，因此用两列算子就可以有 15×15=225 个状态，因此可以表示 0 到 224。阿拉伯数字的每一位有 0 到 9 这 10 个图形符号，用两个阿拉伯数字图形符号时就能有 10×10=100 个状态，因此可以表示 0 到 99 这 100 个数。

这里的算盘其实就是一个基于 15 进制的记数器（可以通过维持一列算子的状态来记录一位数字），它的一列算子就相当于一位阿拉伯数字，每列有 15 种状态，故能表示从 0 到 14 这 15 个数字，超出 14 后就必须通过进位来要求另一列算子的加入以表示数字。电脑与此一样，其并不是数字计算机，而是电子计算机，电脑中通过一根线的电位高低来表示数字。一根线中的电位规定只有两种状态——高电位和低电位，因此电脑的数字表示形式是二进制

的。

和上面的算盘一样，一根电线只有两个状态，当要表示超出 1 的数字时，就必须进位来要求另一根线的加入以表示数字。所谓的 32 位电脑就是提供了 32 根线（被称作数据总线）来表示数据，因此就有 2 的 32 次方那么多种状态。而 16 根线就能表示 2 的 16 次方那么多种状态。

所以，电脑并不是基于二进制数，而是基于状态的变化，只不过这个状态可以使用二进制数表示出来而已。即电脑并不认识二进制数，这是下面“类型”一节的基础。

内存

内存就是电脑中能记录数字的硬件，但其存储速度很快（与硬盘等低速存储设备比较），又不能较长时间保存数据，所以经常被用做草稿纸，记录一些临时信息。

前面已经说过，32 位计算机的数字是通过 32 根线上的电位状态的组合来表示的，因此内存能记录数字，也就是能维持 32 根线上各自的电位状态（就好象算盘的算子拨动后就不会改变位置，除非再次拨动它）。不过依旧考虑上面的算盘，假如一个算盘上有 15 列算子，则一个算盘能表示 15 的 15 次方个状态，是很大的数字，但经常实际是不会用到变化那么大的数字的，因此让一个算盘只有两列算子，则只能表示 225 个状态，当数字超出时就使用另一个或多个算盘来一起表示。

上面不管是 2 列算子还是 15 列算子，都是算盘的粒度，粒度分得过大造成不必要的浪费（很多列算子都不使用），太小又很麻烦（需要多个算盘）。电脑与此一样。2 的 32 次方可表示的数字很大，一般都不会用到，如果直接以 32 位存储在内存中势必造成相当大的资源浪费。于是如上，规定内存的粒度为 8 位二进制数，称为一个内存单元，而其大小称为一个字节（Byte）。就是说，内存存储数字，至少都会记录 8 根线上的电位状态，也就是 2 的 8 次方共 256 种状态。所以如果一个 32 位的二进制数要存储在内存中，就需要占据 4 个内存单元，也就是 4 个字节的内存空间。

我们在纸上写字，是通过肉眼判断出字在纸上的相对横坐标和纵坐标以查找到要看的字或要写字的位置。同样，由于内存就相当于草稿纸，因此也需要某种定位方式来定位，在电脑中，就是通过一个数字来定位的。这就和旅馆的房间号一样，内存单元就相当于房间（假定每个房间只能住一个人），而前面说的那个数字就相当于房间号。为了向某块内存中写入数据（就是使用某块内存来记录数据总线上的电位状态），就必须知道这块内存对应的数字，而这个数字就被称为地址。而通过给定的地址找到对应的内存单元就称为寻址。

因此地址就是一个数字，用以唯一标识某一特定内存单元。此数字一般是 32 位长的二进制数，也就可以表示 4G 个状态，也就是说一般的 32 位电脑都具有 4G 的内存空间寻址能力，即电脑最多装 4G 的内存，如果电脑有超过 4G 的内存，此时就需要增加地址的长度，如用 40 位长的二进制数来表示。

类型

在本系列最开头时已经说明了何谓编程，而刚才更进一步说明了电脑其实连数字都不认识，只是状态的记录，而所谓的加法也只是人为设计那个加法器以使得两个状态经过加法器的处理而生成的状态正好和数学上的加法的结果一样而已。这一切的一切都只说明一点：电脑所做的工作是什么，全视使用的人以为是什么。

因此为了利用电脑那很快的“计算”能力（实际是状态的变换能力），人为规定了如何解释那些状态。为了方便其间，对于前面提出的电位的状态，我们使用 1 位二进制数来表示，则上面提出的状态就可以使用一个二进制数来表示，而所谓的“如何解释那些状态”就变成了如何解释一个二进制数。

C++是高级语言，为了帮助解释那些二进制数，提供了类型这个概念。类型就是人为制订的如何解释内存中的二进制数的协议。C++提供了下面的一些标准类型定义。

signed char　表示所指向的内存中的数字使用补码形式，表示的数字为-128 到+127，长度为 1 个字节

unsigned char　表示所指向的内存中的数字使用原码形式，表示的数字为 0 到 255，长度为 1 个字节

signed short　表示所指向的内存中的数字使用补码形式，表示的数字为－32768 到+32767，长度为 2 个字节

unsigned short　表示所指向的内存中的数字使用原码形式，表示的数字为 0 到 65535，长度为 2 个字节

signed long　表示所指向的内存中的数字使用补码形式，表示的数字为-2147483648 到+2147483647，长度为 4 个字节

unsigned long　表示所指向的内存中的数字使用原码形式，表示的数字为 0 到 4294967295，长度为 4 个字节

signed int

表示所指向的内存中的数字使用补码形式，表示的数字则视编译器。如果编译器编译时被指明编译为在 16 位操作系统上运行，则等同于 signed short；如果是编译为 32 位的，则等同于 signed long；如果是编译为在 64 位操作系统上运行，则为 8 个字节长，而范围则如上一样可以自行推算出来。

unsigned int　表示所指向的内存中的数字使用原码形式，其余和 signed int 一样，表示的是无符号数。

bool　表示所指向的内存中的数字为逻辑值，取值为 false 或 true。长度为 1 个字节。

float　表示所指向的内存按 IEEE 标准进行解释，为 real*4，占用 4 字节内存空间，等同于上篇中提到的单精度浮点数。

double　表示所指向的内存按 IEEE 标准进行解释，为 real*8，可表示数的精度较 float 高，占用 8 字节内存空间，等同于上篇提到的双精度浮点数。

long double　表示所指向的内存按 IEEE 标准进行解释，为 real*10，可表示数的精度较 double 高，但在为 32 位 Windows 操作系统编写程序时，仍占用 8 字节内存空间，等效于 double，只是如果 CPU 支持此类浮点类型则还是可以进行这个精度的计算。

标准类型不止上面的几个，后面还会陆续提到。

上面的长度为 2 个字节也就是将两个连续的内存单元中的数字取出并合并在一起以表示一个数字，这和前面说的一个算盘表示不了的数字，就进位以加入另一个算盘帮助表示是同样的道理。

上面的 signed 关键字是可以去掉的，即 char 等同于 signed char，用以简化代码的编写。但也仅限于 signed，如果是 unsigned char，则在使用时依旧必须是 unsigned char。

现在应该已经了解上篇中为什么数字还要分什么有符号无符号、长整型短整型之类的了，而上面的 short、char 等也都只是长度不同，这就由程序员自己根据可能出现的数字变化幅度来进行选用了。

类型只是对内存中的数字的解释，但上面的类型看起来相对简单了点，且语义并不是很强，即没有什么特殊意思。为此，C++提供了自定义类型，也就是后继文章中将要说明的结

构、类等。

变量

在本系列的第一篇中已经说过，电脑编程的绝大部分工作就是操作内存，而上面说了，为了操作内存，需要使用地址来标识要操作的内存块的首地址（上面的 long 表示连续的 4 个字节内存，其第一个内存单元的地址称作这连续 4 个字节内存块的首地址）。为此我们在编写程序时必须记下地址。

做 5+2/3-5*2 的计算，先计算出 2/3 的值，写在草稿纸上，接着算出 5*2 的值，又写在草稿纸上。为了接下来的加法和减法运算，必须能够知道草稿纸上的两个数字哪个是 2/3 的值哪个是 5*2 的值。人就是通过记忆那两个数在纸上的位置来记忆的，而电脑就是通过地址来标识的。但电脑只会做加减乘除，不会去主动记那些 2/3、5*2 的中间值的位置，也就是地址。因此程序员必须完成这个工作，将那两个地址记下来。

问题就是这里只有两个值，也许好记一些，但如果多了，人是很难记住哪个地址对应哪个值的，但人对符号比对数字要敏感得多，即人很容易记下一个名字而不是一个数字。为此，程序员就自己写了一个表，表有两列，一列是“2/3 的值”，一列是对应的地址。如果式子稍微复杂点，那么那个表可能就有个二三十行，而每写一行代码就要去翻查相应的地址，如果来个几万行代码那是人都不能忍受。

C++作为高级语言，很正常地提供了上面问题的解决之道，就是由编译器来帮程序员维护那个表，要查的时候是编译器去查，这也就是变量的功能。

变量是一个映射元素。上面提到的表由编译器维护，而表中的每一行都是这个表的一个元素（也称记录）。表有三列：变量名、对应地址和相应类型。变量名是一个标识符，因此其命名规则完全按照上一篇所说的来。当要对某块内存写入数据时，程序员使用相应的变量名进行内存的标识，而表中的对应地址就记录了这个地址，进而将程序员给出的变量名，一个标识符，映射成一个地址，因此变量是一个映射元素。而相应类型则告诉编译器应该如何解释此地址所指向的内存，是 2 个连续字节还是 4 个？是原码记录还是补码？而变量所对应的地址所标识的内存的内容叫做此变量的值。

有如下的变量解释：“可变的量，其相当于一个盒子，数字就装在盒子里，而变量名就写在盒子外面，这样电脑就知道我们要处理哪一个盒子，且不同的盒子装不同的东西，装字符串的盒子就不能装数字。”上面就是我第一次学习编程时，书上写的（是 BASIC 语言）。对于初学者也许很容易理解，也不能说错，但是造成的误解将导致以后的程序编写地千疮百孔。

上面的解释隐含了一个意思——变量是一块内存。这是严重错误的！如果变量是一块内存，那么 C++中著名的引用类型将被弃置荒野。变量实际并不是一块内存，只是一个映射元素，这是致关重要的。

内存的种类

前面已经说了内存是什么及其用处，但内存是不能随便使用的，因为操作系统自己也要使用内存，而且现在的操作系统正常情况下都是多任务操作系统，即可同时执行多个程序，即使只有一个 CPU。因此如果不对内存访问加以节制，可能会破坏另一个程序的运作。比如我在纸上写了 2/3 的值，而你未经我同意且未通知我就将那个值擦掉，并写上 5*2 的值，结果我后面的所有计算也就出错了。

因此为了使用一块内存，需要向操作系统申请，由操作系统统一管理所有程序使用的内存。所以为了记录一个 long 类型的数字，先向操作系统申请一块连续的 4 字节长的内存空间，然后操作系统就会在内存中查看，看是否还有连续的 4 个字节长的内存，如果找到，则返回此 4 字节内存的首地址，然后编译器编译的指令将其记录在前面提到的变量表中，最后就可以用它记录一些临时计算结果了。

上面的过程称为要求操作系统分配一块内存。这看起来很不错，但是如果只为了 4 个字节就要求操作系统搜索一下内存状况，那么如果需要 100 个临时数据，就要求操作系统分配内存 100 次，很明显地效率低下（无谓的 99 次查看内存状况）。因此 C++发现了这个问题，并且操作系统也提出了相应的解决方法，最后提出了如下的解决之道。

栈（Stack） 任何程序执行前，预先分配一固定长度的内存空间，这块内存空间被称作栈（这种说法并不准确，但由于实际涉及到线程，在此为了不将问题复杂化才这样说明），也被叫做堆栈。那么在要求一个 4 字节内存时，实际是在这个已分配好的内存空间中获取内存，即内存的维护工作由程序员自己来做，即程序员自己判断可以使用哪些内存，而不是操作系统，直到已分配的内存用完。

很明显，上面的工作是由编译器来做的，不用程序员操心，因此就程序员的角度来看什么事情都没发生，还是需要像原来那样向操作系统申请内存，然后再使用。

但工作只是从操作系统变到程序自己而已，要维护内存，依然要耗费 CPU 的时间，不过要简单多了，因为不用标记一块内存是否有人使用，而专门记录一个地址。此地址以上的内存空间就是有人正在使用的，而此地址以下的内存空间就是无人使用的。之所以是以下的空间为无人使用而不是以上，是当此地址减小到 0 时就可以知道堆栈溢出了（如果你已经有些基础，请不要把 0 认为是虚拟内存地址，关于虚拟内存将会在《C++从零开始（十八）》中进行说明,这里如此解释只是为了方便理解）。而且 CPU 还专门对此法提供了支持，给出了两条指令，转成汇编语言就是 push 和 pop，表示压栈和出栈，分别减小和增大那个地址。

而最重要的好处就是由于程序一开始执行时就已经分配了一大块连续内存，用一个变量记录这块连续内存的首地址，然后程序中所有用到的，程序员以为是向操作系统分配的内存都可以通过那个首地址加上相应偏移来得到正确位置，而这很明显地由编译器做了。因此实际上等同于在编译时期（即编译器编译程序的时候）就已经分配了内存（注意，实际编译时期是不能分配内存的，因为分配内存是指程序运行时向操作系统申请内存，而这里由于使用堆栈，则编译器将生成一些指令，以使得程序一开始就向操作系统申请内存，如果失败则立刻退出，而如果不退出就表示那些内存已经分配到了，进而代码中使用首地址加偏移来使用内存也就是有效的），但坏处也就是只能在编译时期分配内存。

堆（Heap） 上面的工作是编译器做的，即程序员并不参与堆栈的维护。但上面已经说了，堆栈相当于在编译时期分配内存，因此一旦计算好某块内存的偏移，则这块内存就只能那么大，不能变化了（如果变化会导致其他内存块的偏移错误）。比如要求客户输入定单数据，可能有 10 份定单，也可能有 100 份定单，如果一开始就定好了内存大小，则可能造成不必要的浪费，又或者内存不够。

为了解决上面的问题，C++提供了另一个途径，即允许程序员有两种向操作系统申请内存的方式。前一种就是在栈上分配，申请的内存大小固定不变。后一种是在堆上分配，申请的内存大小可以在运行的时候变化，不是固定不变的。

那么什么叫堆？在 Windows 操作系统下，由操作系统分配的内存就叫做堆，而栈可以认为是在程序开始时就分配的堆（这并不准确，但为了不复杂化问题，故如此说明）。因此在堆上就可以分配大小变化的内存块，因为是运行时期即时分配的内存，而不是编译时期已计算好大小的内存块。

变量的定义

上面说了那么多，你可能看得很晕，毕竟连一个实例都没有，全是文字，下面就来帮助加深对上面的理解。

定义一个变量，就是向上面说的由编译器维护的变量表中添加元素，其语法如下：

long a;

先写变量的类型，然后一个或多个空格或制表符（\t）或其它间隔符，接着变量的名字，最后用分号结束。要同时定义多个变量，则各变量间使用逗号隔开，如下：

long a, b, c; unsigned short e, a_34c;

上面是两条变量定义语句，各语句间用分号隔开，而各同类型变量间用逗号隔开。而前面的式子 5+2/3-5*2，则如下书写。

long a = 2/3, b = 5*2; long c = 5 + a – b;

可以不用再去记那烦人的地址了，只需记着 a、b 这种简单的标识符。当然，上面的式子不一定非要那么写，也可以写成：long c = 5 + 2 / 3 – 5 * 2; 而那些 a、b 等中间变量编译器会自动生成并使用（实际中编译器由于优化的原因将直接计算出结果，而不会生成实际的计算代码）。

下面就是问题的关键，定义变量就是添加一个映射。前面已经说了，这个映射是将变量名和一个地址关联，因此在定义一个变量时，编译器为了能将变量名和某个地址对应起来，帮程序员在前面提到的栈上分配了一块内存，大小就视这个变量类型的大小。如上面的 a、b、c 的大小都是 4 个字节，而 e、a_34c 的大小都是 2 个字节。

假设编译器分配的栈在一开始时的地址是 1000，并假设变量 a 所对应的地址是 1000-56，则 b 所对应的地址就是 1000-60，而 c 所对应的就是 1000-64，e 对应的是 1000-66，a_34c 是 1000-68。如果这时 b 突然不想是 4 字节了，而希望是 8 字节，则后续的 c、e、a_34c 都将由于还是原来的偏移位置而使用了错误的内存，这也就是为什么栈上分配的内存必须是固定大小。

考虑前面说的红色文字：“变量实际并不是一块内存，只是一个映射元素”。可是只要定义一个变量，就会相应地得到一块内存，为什么不说变量就是一块内存？上面定义变量时之所以会分配一块内存是因为变量是一个映射元素，需要一个对应地址，因此才在栈上分配了一块内存，并将其地址记录到变量表中。但是变量是可以有别名的，即另一个名字。这个说法是不准确的，应该是变量所对应的内存块有另一个名字，而不止是这个变量的名字。

为什么要有别名？这是语义的需要，表示既是什么又是什么。比如一块内存，里面记录了老板的信息，因此起名为 Boss，但是老板又是另一家公司的行政经理，故变量名应该为 Manager，而在程序中有段代码是老板的公司相关的，而另一段是老板所在公司相关的，在这两段程序中都要使用到老板的信息，那到底是使用 Boss 还是 Manager？其实使用什么都不会对最终生成的机器代码产生什么影响，但此处出于语义的需要就应该使用别名，以期从代码上表现出所编写程序的意思。

在 C++中，为了支持变量别名，提供了引用变量这个概念。要定义一个引用变量，在定义变量时，在变量名的前面加一个“&”，如下书写：

long a; long &a1 = a, &a2 = a, &a3 = a2;

上面的 a1、a2、a3 都是 a 所对应的内存块的别名。这里在定义变量 a 时就在栈上分配了一块 4 字节内存，而在定义 a1 时却没有分配任何内存，直接将变量 a 所映射的地址作为变量 a1 的映射地址，进而形成对定义 a 时所分配的内存的别名。因此上面的 Boss 和 Manager，应该如下（其中 Person 是一个结构或类或其他什么自定义类型，这将在后继的文章中陆续

说明）：

Person Boss; Person &Manager = Boss;

由于变量一旦定义就不能改变（指前面说的变量表里的内容，不是变量的值），直到其被删除，所以上面在定义引用变量的时候必须给出欲别名的变量以初始化前面的变量表，否则编译器编译时将报错。

现在应该就更能理解前面关于变量的红字的意思了。并不是每个变量定义时都会分配内存空间的。而关于如何在堆上分配内存，将在介绍完指针后予以说明，并进而说明上一篇遗留下来的关于字符串的问题。

C++从零开始（四）

——赋值操作符

本篇是《C++从零开始（二）》的延续，说明《C++从零开始（二）》中遗留下来的关于表达式的内容，并为下篇指针的运用做一点铺垫。虽然上篇已经说明了变量是什么，但对于变量最关键的东西却由于篇幅限制而没有说明，下面先说明如何访问内存。

赋值语句

前面已经说明，要访问内存，就需要相应的地址以表明访问哪块内存，而变量是一个映射，因此变量名就相当于一个地址。对于内存的操作，在一般情况下就只有读取内存中的数值和将数值写入内存（不考虑分配和释放内存），在 C++中，为了将一数值写入某变量对应的地址所标识的内存中（出于简便，以后称变量 a 对应的地址为变量 a 的地址，而直接称变量 a 的地址所标识的内存为变量 a），只需先书写变量名，后接“=”，再接欲写入的数字（关于数字，请参考《C++从零开始（二）》）以及分号。如下：

a = 10.0f; b = 34;

由于接的是数字，因此就可以接表达式并由编译器生成计算相应表达式所需的代码，也就可如下：

c = a / b * 120.4f;

上句编译器将会生成进行除法和乘法计算的 CPU 指令，在计算完毕后（也就是求得表达式 a / b * 120.4f 的值了后），也会同时生成将计算结果放到变量 c 中去的 CPU 指令，这就是语句的基本作用（对于语句，在《C++从零开始（六）》中会详细说明）。

上面在书写赋值语句时，应该确保此语句之前已经将使用到的变量定义过，这样编译器才能在生成赋值用的 CPU 指令时查找到相应变量的地址，进而完成 CPU 指令的生成。如上面的 a 和 b，就需要在书写上面语句前先书写类似下面的变量定义：

float a; long b;

直接书写变量名也是一条语句，其导致编译器生成一条读取相应变量的内容的语句。即可以如下书写：

a;

上面将生成一条读取内存的语句，即使从内存中读出来的数字没有任何应用（当然，如果编译器开了优化选项，则上面的语句将不会生成任何代码）。从这一点以及上面的 c = a / b * 120.4f;语句中，都可以看出一点——变量是可以返回数字的。而变量返回的数字就是按照

变量的类型来解释变量对应内存中的内容所得到的数字。这句话也许不是那么容易理解，在看过后面的类型转换一节后应该就可以理解了。

因此为了将数据写入一块内存，使用赋值语句（即等号）；要读取一块内存，书写标识内存的变量名。所以就可以这样书写：a = a + 3;

假设 a 原来的值为 1，则上面的赋值语句将 a 的值取出来，加上 3，得到结果 4，将 4 再写入 a 中去。由于 C++使用“=”来代表赋值语句，很容易使人和数学中的等号混淆起来，这点应注意。

而如上的 float a;语句，当还未对变量进行任何赋值操作时，a 的值是什么？上帝才知道。当时的 a 的内容是什么（对于 VC 编译器，在开启了调试选项时，将会用 0xCCCCCCCC 填充这些未初始化内存），就用 IEEE 的 real*4 格式来解释它并得到相应的一个数字，也就是 a 的值。因此应在变量定义的时候就进行赋值（但是会有性能上的影响，不过很小），以初始化变量而防止出现莫名其妙的值，如：float a = 0.0f;。

赋值操作符

上面的 a = a + 3;的意思就是让 a 的值增加 3。在 C++中，对于这种情况给出了一种简写方案，即前面的语句可以写成：a += 3;。应当注意这两条语句从逻辑上讲都是使变量 a 的值增 3，但是它们实际是有区别的，后者可以被编译成优化的代码，因为其意思是使某一块内存的值增加一定数量，而前者是将一个数字写入到某块内存中。所以如果可能，应尽量使用后者，即 a += 3;。这种语句可以让编译器进行一定的优化（但由于现在的编译器都非常智能，能够发现 a = a + 3;是对一块内存的增值操作而不是一块内存的赋值操作，因此上面两条语句实际上可以认为完全相同，仅仅只具有简写的功能了）。

对于上面的情况，也可以应用在减法、乘法等二元非逻辑操作符（不是逻辑值操作符，即不能 a &&= 3;）上，如：a *= 3; a -= 4; a |= 34; a >>= 3;等。

除了上面的简写外，C++还提供了一种简写方式，即 a++;，其逻辑上等同于 a += 1;。同上，在电脑编程中，加一和减一是经常用到的，因此 CPU 专门提供了两条指令来进行加一和减一操作（转成汇编语言就是 Inc 和 Dec），但速度比直接通过加法或减法指令来执行要快得多。为此 C++中也就提供了“++”和“—”操作符来对应 Inc 和 Dec。所以 a++;虽然逻辑上和 a = a + 1;等效，实际由于编译器可能做出的优化处理而不同，但还是如上，由于编译器的智能化，其是有可能看出 a = a + 1;可以编译成 Inc 指令进而即使没有使用 a++;却也依然可以得到优化的代码，这样 a++;将只剩下简写的意义而已。

应当注意一点，a = 3;这句语句也将返回一个数字，也就是在 a 被赋完值后 a 的值。由于其可以返回数字，按照《C++从零开始（二）》中所说，“=”就属于操作符，也就可以如下书写：

c = 4 + (a = 3);

之所以打括号是因为“=”的优先级较“+”低，而更常见和正常的应用是：c = a = 3;

应该注意上面并不是将 c 和 a 赋值为 3，而是在 a 被赋值为 3 后再将 a 赋值给 c，虽然最后结果和 c、a 都赋值为 3 是一样的，但不应该这样理解。由于 a++;表示的就是 a += 1;就是 a = a + 1;，因此 a++;也将返回一个数字。也由于这个原因，C++又提供了另一个简写方式，++a;。

假设 a 为 1，则 a++;将先返回 a 的值，1，然后再将 a 的值加一；而++a;先将 a 的值加一，再返回 a 的值，2。而 a—和—a 也是如此，只不过是减一罢了。

上面的变量 a 按照最上面的变量定义，是 float 类型的变量，对它使用++操作符并不能

得到预想的优化，因为 float 类型是浮点类型，其是使用 IEEE 的 real*4 格式来表示数字的，而不是二进制原码或补码，而前面提到的 Inc 和 Dec 指令都是出于二进制的表示优点来进行快速增一和减一，所以如果对浮点类型的变量运用“++”操作符，将完全只是简写，没有任何的优化效果（当然，如果 CPU 提供了新的指令集，如 MMX 等，以对 real*4 格式进行快速增一和减一操作，且编译器支持相应指令集，则还是可以产生优化效果的）。

赋值操作符的返回值

在进一步了解++a 和 a++的区别前，先来了解何谓操作符的计算（Evaluate）。操作符就是将给定的数字做一些处理，然后返回一个数字。而操作符的计算也就是执行操作符的处理，并返回值。前面已经知道，操作符是个符号，其一侧或两侧都可以接数字，也就是再接其他操作符，而又由于赋值操作符也属于一种操作符，因此操作符的执行顺序变得相当重要。

对于 a + b + c，将先执行 a + b，再执行(a + b) + c 的操作。你可能觉得没什么，那么如下，假设 a 之前为 1：

c = (a *= 2) + (a += 3);

上句执行后 a 为 5。而 c = (a += 3) + (a *= 2);执行后，a 就是 8 了。那么 c 呢？结果可能会大大的出乎你的意料。前者的 c 为 10，而后者的 c 为 16。

上面其实是一个障眼法，其中的“+”没有任何意义，即之所以会从左向右执行并不是因为“+”的缘故，而是因为(a *= 2)和(a += 3)的优先级相同，而按照“()”的计算顺序，是从左向右来计算的。但为什么 c 的值不是预想的 2 + 5 和 4 + 8 呢？因为赋值操作符的返回值的关系。

赋值操作符返回的数字不是变量的值，而是变量对应的地址。这很重要。前面说过，光写一个变量名就会返回相应变量的值，那是因为变量是一个映射，变量名就等同于一个地址。C++中将数字看作一个很特殊的操作符，即任何一个数字都是一个操作符。而地址就和长整型、单精度浮点数这类一样，是数字的一种类型。当一个数字是地址类型时，作为操作符，其没有要操作的数字，仅仅返回将此数字看作地址而标识的内存中的内容（用这个地址的类型来解释）。地址可以通过多种途径得到，如上面光写一个变量名就可以得到其对应的地址，而得到的地址的类型也就是相应的变量的类型。如果这句话不能理解，在看过下面的类型转换一节后应该就能了解了。

所以前面的 c = (a += 3) + (a *= 2);，由于“()”的参与改变了优先级而先执行了两个赋值操作符，然后两个赋值操作符都返回 a 的地址，然后计算“+”的值，分别计算两边的数字——a 的地址（a 的地址也是一个操作符），也就是已经执行过两次赋值操作的 a 的值，得 8，故最后的 c 为 16。而另一个也由于同样的原因使得 c 为 10。

现在考虑操作符的计算顺序。当同时出现了几个优先级相同的操作符时，不同的操作符具有不同的计算顺序。前面的“()”以及“-”、“*”等这类二元操作符的计算顺序都是从左向右计算，而“!”、负号“-”等前面介绍过的一元操作符都是从右向左计算的，如：!-!!a;，假设 a 为 3。先计算从左朝右数第三个“!”的值，导致计算 a 的地址的值，得 3；然后逻辑取反得 0，接着再计算第二个“!”的值，逻辑取反后得 1，再计算负号“-”的值，得-1，最后计算第一个“!”的值，得 0。

赋值操作符都是从右向左计算的，除了后缀“++”和后缀“—”（即上面的 a++和 a--）。因此上面的 c = a = 3;，因为两个“=”优先级相同，从右向左计算，先计算 a = 3 的值，返回 a 对应的地址，然后计算返回的地址而得到值 3，再计算 c = (a = 3)，将 3 写入 c。而不是从左向右计算，即先计算 c = a，返回 c 的地址，然后再计算第二个“=”，将 3 写入 c，这样 a

就没有被赋值而出现问题。又：

a = 1; c = 2; c *= a += 4;

由于“*=”和“+=”的优先级相同，从右向左计算先计算 a += 4，得 a 为 5，然后返回 a 的地址，再计算 a 的地址得 a 的值 5，计算“*=”以使得 c 的值为 10。

因此按照前面所说，++a 将返回 a 的地址，而 a++也因为是赋值操作符而必须返回一个地址，但很明显地不能是 a 的地址了，因此编译器将编写代码以从栈中分配一块和 a 同样大小的内存，并将 a 的值复制到这块临时内存中，然后返回这块临时内存的地址。由于这块临时内存是因为编译器的需要而分配的，与程序员完全没有关系，因此程序员是不应该也不能写这块临时内存的(因为编译器负责编译代码，如果程序员欲访问这块内存，编译器将报错)，但可以读取它的值，这也是返回地址的主要目的。所以如下的语句没有问题：

(++a) = a += 34;

但(a++) = a += 34;就会在编译时报错，因为 a++返回的地址所标识的内存只能由编译器负责处理，程序员只能获得其值而已。

a++的意思是先返回 a 的值，也就是上面说的临时内存的地址，然后再将变量的值加一。如果同时出现多个 a++，那么每个 a++都需要分配一块临时内存(注意前面 c = (a += 3) + (a *= 2);的说明)，那么将有点糟糕，而且 a++的意思是先返回 a 的值，那么到底是什么时候的 a 的值呢？在 VC 中，当表达式中出现后缀“++”或后缀“—”时，只分配一块临时内存，然后所有的后缀“++”或后缀“—”都返回这个临时内存的地址，然后在所有的可以计算的其他操作符的值计算完毕后，再将对应变量的值写入到临时内存中，计算表达式的值，最后将对应变量的值加一或减一。

因此：a = 1; c = (a++) + (a++);执行后，c 的值为 2，而 a 的值为 3。而如下：

a = 1; b = 1; c = (++a) + (a++) + (b *= a++) + (a *= 2) + (a *= a++);

执行时，先分配临时内存，然后由于 5 个“()”，其计算顺序是从左向右，

计算++a 的值，返回增一后的 a 的地址，a 的值为 2

计算 a++的值，返回临时内存的地址，a 的值仍为 2

计算 b *= a++中的 a++，返回临时内存的地址，a 的值仍为 2

计算 b *= a++中的“*=”，将 a 的值写入临时内存，计算得 b 的值为 2，返回 b 的地址

计算 a *= 2 的值，返回 a 的地址，a 的值为 4

计算 a *= a++中的 a++，返回临时内存的地址，a 的值仍为 4

计算 a *= a++中的“*=”，将 a 的值写入临时内存，返回 a 的地址，a 的值为 16

计算剩下的“+”，为了进行计算，将 a 的值写入临时内存，得值 16 + 16 + 2 + 16 + 16 为 66，写入 c 中

计算三个 a++欠下的加一，a 最后变为 19。

上面说了那么多，无非只是想告诫你——在表达式中运用赋值操作符是不被推崇的。因为其不符合平常的数学表达式的习惯，且计算顺序很容易搞混。如果有多个“++”操作符，最好还是将表达式分开，否则很容易导致错误的计算顺序而计算错误。并且导致计算顺序混乱的还不止上面的 a++就完了，为了让你更加地重视前面的红字，下面将介绍更令人火大的东西，如果你已经同意上面的红字，则下面这一节完全可以跳过，其对编程来讲可以认为根本没有任何意义（要不是为了写这篇文章，我都不知道它的存在）。

序列点（Sequence Point）和附加效果（Side Effect）

在计算 c = a++时，当 c 的值计算（Evaluate）出来时，a 的值也增加了一，a 的值加一就

是计算前面表达式的附加效果。有什么问题？它可能影响表达式的计算结果。

对于 a = 0; b = 1; (a *= 2) && (b += 2);，由于两个“()”优先级相同，从左向右计算，计算“*=”而返回 a 的地址，再计算“+=”而返回 b 的地址，最后由于 a 的值为 0 而返回逻辑假。很正常，但效率低了点。

如果“&&”左边的数字已经是 0 了，则不再需要计算右边的式子。同样，如果“||”左边的数字已经非零了，也不需要再计算右边的数字。因为“&&”和“||”都是数学上的，数学上不管先计算加号左边的值还是右边的值，结果都不会改变，因此“&&”和“||”才会做刚才的解释。这也是 C++保证的，既满足数学的定义，又能提供优化的途径（“&&”和“||”右边的数字不用计算了）。

因此上面的式子就会被解释成——如果 a 在自乘了 2 后的值为 0，则 b 就不用再自增 2 了。这很明显地违背了我们的初衷，认为 b 无论如何都会被自增 2 的。但是 C++却这样保证，不仅仅是因为数学的定义，还由于代码生成的优化。但是按照操作符的优先级进行计算，上面的 b += 2 依旧会被执行的（这也正是我们会书写上面代码的原因）。为了实现当 a 为 0 时 b += 2 不会被计算，C++提出了序列点的概念。

序列点是一些特殊位置，由 C++强行定义（C++并未给出序列点的定义，因此不同的编译器可能给出不同的序列点定义，VC 是按照 C 语言定义的序列点）。当在进行操作符的计算时，如果遇到序列点，则序列点处的值必须被优先计算，以保证一些特殊用途，如上面的保证当 a 为 0 时不计算 b += 2，并且序列点相关的操作符（如前面的“&&”和“||”）也将被计算完毕，然后才恢复正常的计算。

“&&”的左边数字的计算就是一个序列点，而“||”的左边数字的计算也是。C++定义了多个序列点，包括条件语句、函数参数等条件下的表达式计算，在此，不需要具体了解有哪些序列点，只需要知道由于序列点的存在而可能导致赋值操作符的计算出乎意料。下面就来分析一个例子：

a = 0; b = 1; (a *= 2) && (b += ++a);

按照优先级的顺序，编译器发现要先计算 a *= 2，再计算++a，接着“+=”，最后计算“&&”。然后编译器发现这个计算过程中，出现了“&&”左边的数字这个序列点，其要保证被优先计算，这样就有可能不用计算 b += ++a 了。所以编译器先计算“&&”的数字，通过上面的计算过程，编译器发现就要计算 a *= 2 才能得到“&&”左边的数字，因此将先计算 a *= 2，返回 a 的地址，然后计算“&&”左边的数字，得 a 的值为 0，因此就不计算 b += ++a 了。而不是最开始想象的由于优先级的关系先将 a 加一后再进行 a 的计算，以返回 1。所以上面计算完毕后，a 为 0，b 为 1，返回 0，表示逻辑假。

因此序列点的出现是为了保证一些特殊规则的出现，如上面的“&&”和“||”。再考虑“,”操作符，其操作是计算两边的值，然后返回右边的数字，即：a, b + 3 将返回 b + 3 的值，但是 a 依旧会被计算。由于“,”的优先级是最低的（但高于前面提到的“数字”操作符），因此如果 a = 3, 4;，那么 a 将为 3 而不是 4，因为先计算“=”，返回 a 的地址后再计算“,”。又：

a = 1; b = 0; b = (a += 2) + ((a *= 2, b = a - 1) && (c = a));

由于“&&”左边数字是一个序列点，因此先计算 a *= 2, b 的值，但根据“,”的返回值定义，其只返回右边的数字，因此不计算 a *= 2 而直接计算 b = a – 1 得 0，“&&”就返回了，但是 a *= 2 就没有被计算而导致 a 的值依旧为 1，这违背了“,”的定义。为了消除这一点（当然可能还有其他应用“,”的情况），C++也将“,”的左边数字定为了序列点，即一定会优先执行“,”左边的数字以保证“,”的定义——计算两边的数字。所以上面就由于“,”左边数字这个序列点而导致 a *= 2 被优先执行，并导致 b 为 1，因此由于“&&”是序列点且其左边数字非零而必须计算完右边数字后才恢复正常优先级，而计算 c = a，得 2，最后才

恢复正常优先级顺序，执行 a += 2 和“+”。结果就 a 为 4，c 为 2，b 为 5。

所以前面的 a = 3, 4;其实就应该是编译器先发现“,”这个序列点，而发现要计算“,”左边的值，必须先计算出 a = 3，因此才先计算 a = 3 以至于感觉序列点好像没有发生作用。下面的式子请自行分析，执行后 a 为 4，但如果将其中的“,”换成“&&”，a 为 2。

a = 1; b = (a *= 2) + ((a *= 3), (a -= 2));

如果上面你看得很晕，没关系，因为上面的内容根本可以认为毫无意义，写在这里也只是为了进一步向你证明，在表达式中运用赋值运算符是不好的，即使它可能让你写出看起来简练的语句，但它也使代码的可维护性降低。

类型转换

在《C++从零开始（二）》中说过，数字可以是浮点数或是整型数或其他，也就是说数字是具有类型的。注意《C++从零开始（三）》中对类型的解释，类型只是说明如何解释状态，而在前面已经说过，出于方便，使用二进制数来表示状态，因此可以说类型是用于告诉编译器如何解释二进制数的。

所以，一个长整型数字是告诉编译器将得到的二进制数表示的状态按照二进制补码的格式来解释以得到一个数值，而一个单精度浮点数就是告诉编译器将得到的二进制数表示的状态按照 IEEE 的 real*4 的格式来解释以得到一个是小数的数值。很明显，同样的二进制数表示的状态，按照不同的类型进行解释将得到不同的数值，那么编译器如何知道应该使用什么类型来进行二进制数的解释？

前面已经说过，数字是一种很特殊的操作符，其没有操作数，仅仅返回由其类型而定的二进制数表示的状态（以后为了方便，将“二进制数表示的状态”称作“二进制数”）。而操作符就是执行指令并返回数字，因此所有的操作符到最后一定执行的是返回一个二进制数。这点很重要，对于后面指针的理解有着重要的意义。

先看 15;，这是一条语句，因为 15 是一个数字。所以 15 被认为是 char 类型的数字（因为其小于 128，没超出 char 的表示范围），将返回一个 8 位长的二进制数，此二进制数按照补码格式编写，为 00001111。

再看 15.0f，同上，其由于接了“f”这个后缀而被认为是 float 类型的数字，将返回一个 32 位长的二进制数，此二进制数按照 IEEE 的 real*4 格式编写，为 10000010111000000000000000000000。

虽然上面 15 和 15.0f 的数值相等，但由于是不同的类型导致了使用不同的格式来表示，甚至连表示用的二进制数的长度都不相同。因此如果书写 15.0f == 15;将返回 0，表示逻辑假。但实际却返回 1，为什么？

上面既然 15 和 15.0f 被表示成完全不同的两个二进制数，但我们又认为 15 和 15.0f 是相等的，但它们的二进制表示不同，怎么办？将表示 15.0f 的二进制数用 IEEE 的 real*4 格式解释出 15 这个数值，然后再将其按 8 位二进制补码格式编写出二进制数，再与原来的表示 15 的二进制数比较。

为了实现上面的操作，C++提供了类型转换操作符——“()”。其看起来和括号操作符一样，但是格式不同：(<类型名>)<数字>或<类型名>(<数字>)。

上面类型转换操作符的<类型名>不是数字，因此其将不会被操作，而是作为一个参数来控制其如何操作后面的<数字>。<类型名>是一个标识符，其唯一标识一个类型，如 char、float 等。类型转换操作符的返回值就如其名字所示，将<数字>按照<类型名>标识的类型来解释，返回类型是<类型名>的数字。因此，上面的例子我们就需要如下编写：15 == (char)15.0f;，

现在其就可以返回 1，表示逻辑真了。但是即使不写(char)，前面的语句也返回 1。这是编译器出于方便的缘故而帮我们在 15 前添加了(float)，所以依然返回 1。这被称作隐式类型转换，在后面说明类的时候，还将提到它。

某个类型可以完全代替另一个类型时，编译器就会进行上面的隐式类型转换，自动添加类型转换操作符。如：char 只能表示-128 到 127 的整数，而 float 很明显地能够表示这些数字，因此编译器进行了隐式类型转换。应当注意，这个隐式转换是由操作符要求的，即前面的“==”要求两面的数字类型一致，结果发现两边不同，结果编译器将 char 转成 float，然后再执行“==”的操作。注意：在这种情况下，编译器总是将较差的类型（如前面的 char）转成较好的类型（如前面的 float），以保证不会发生数值截断问题。如：-41 == 3543;，左边是 char，右边是 short，由于 short 相对于 char 来显得更优（short 能完全替代 char），故实际为：(short)-41 == 3543;，返回 0。而如果是-41 == (char)3543;，由于 char 不能表示 3543，则 3543 以补码转成二进制数 0000110111010111，然后取其低 8 位，而导致高 8 位的 00001101 被丢弃，此被称为截断。结果(char)3543 的返回值就是类型为 char 的二进制数 11010111，为-41,结果-41 == (char)3543;的返回值将为 1，表示逻辑真，很明显地错误。因此前面的 15 == 15.0f;实际将为(float)15 == 15.0f;（注意这里说 15 被编译器解释为 char 类型并不准确，更多的编译器是将它解释成 int 类型）。

注意前面之所以会朝好的方向发展（即 char 转成 float），完全是因为“==”的缘故，其要求这么做。下面考虑“=”：short b = 3543; char a = b;。因为 b 的值是 short 类型，而“=”的要求就是一定要将“=”右边的数字转成和左边一样，这样才能进行正确的内存的写入（简单地将右边数字返回的二进制数复制到左边的地址所表示的内存中）。因此 a 将为-41。但是上面是编译器按照“=”的要求自行进行了隐式转换，可能是由于程序员的疏忽而没有发现这个错误（以为 b 的值一定在-128 到 127 的范围内），因此编译器将对上面的情况给出一个警告，说 b 的值可能被截断。为了消除编译器的疑虑，如下：char a = (char)b;。这样称为显示类型转换，其告诉编译器——“我知道可能发生数据截断，但是我保证不会截断”。因此编译器将不再发出警告。但是如下：char a = (char)3543;，由于编译器可以肯定 3543 一定会被截断而导致错误的返回值，因此编译器将给出警告，说明 3543 将被截断，而不管前面的类型转换操作符是否存在。

现在应该可以推出——15 + 15.0f;返回的是一个 float 类型的数字。因此如果如下：char a = 15 + 15.0f;，编译器将发出警告，说数据可能被截断。因此改成如下：char a = (char)15 + 15.0f;，但类型转换操作符“()”的优先级比“+”高，结果就是 15 先被转换为 char 然后再由于“+”的要求而被隐式转成 float，最后返回 float 给“=”而导致编译器依旧发出警告。为此，就需要提高“+”的优先级，如下：char a = (char)(15 + 15.0f);就没事了（或 char(15 + 15.0f)），其表示我保证 15 + 15.0f 不会导致数据截断。

应该注意类型转换操作符“()”和前缀“++”、“!”、负号“-”等的优先级一样，并且是从右向左计算的，因此(char)-34;将会先计算-34 的值，然后再计算(char)的值，这也正好符合人的习惯。

下篇将针对数字这个特殊操作符而提出一系列的东西，因此如果理解了数字的意思，那么指针将很容易理解。

C++从零开始（五）

——何谓指针

本篇说明C++中的重中又重的关键——指针类型，并说明两个很有意义的概念——静态和动态。

数组

前面说了在 C++中是通过变量来对内存进行访问的，但根据前面的说明，C++中只能通过变量来操作内存，也就是说要操作某块内存，就必须先将这块内存的首地址和一个变量名绑定起来，这是很糟糕的。比如有100块内存用以记录100个工人的工资，现在要将每个工人的工资增加5%，为了知道各个工人增加了后的工资为多少，就定义一个变量 float a1;，用其记录第1个工人的工资，然后执行语句a1 += a1 * 0.05f;，则a1里就是增加后的工资。由于是100个工人，所以就必须有100个变量，分别记录100个工资。因此上面的赋值语句就需要有100条，每条仅仅变量名不一样。

上面需要手工重复书写变量定义语句float a1;100遍（每次变一个变量名），无谓的工作。因此想到一次向操作系统申请100*4=400个字节的连续内存，那么要给第i个工人修改工资，只需从首地址开始加上4*i个字节就行了（因为float占用4个字节）。

为了提供这个功能，C++提出了一种类型——数组。数组即一组数字，其中的各个数字称作相应数组的元素，各元素的大小一定相等（因为数组中的元素是靠固定的偏移来标识的），即数组表示一组相同类型的数字，其在内存中一定是连续存放的。在定义变量时，要表示某个变量是数组类型时，在变量名的后面加上方括号，在方括号中指明欲申请的数组元素个数，以分号结束。因此上面的记录100个工资的变量，即可如下定义成数组类型的变量：

float a[100];

上面定义了一个变量a，分配了100*4=400个字节的连续内存（因为一个float元素占用4个字节），然后将其首地址和变量名a相绑定。而变量a的类型就被称作具有100个float类型元素的数组。即将如下解释变量a所对应内存中的内容（类型就是如何解释内存的内容）：a 所对应的地址标识的内存是一块连续内存的首地址，这块连续内存的大小刚好能容纳下100个float类型的数字。

因此可以将前面的float b;这种定义看成是定义了一个元素的float数组变量b。而为了能够访问数组中的某个元素，在变量名后接方括号，方括号中放一数字，数字必须是非浮点数，即使用二进制原码或补码进行表示的数字。如a[5 + 3] += 32;就是数组变量a的第5 + 3个元素的值增加32。又：

long c = 23; float b = a[(c – 3) / 5] + 10, d = a[c – 23];

上面的b的值就为数组变量a的第4个元素的值加10，而d的值就为数组变量a的第0个元素的值。即C++的数组中的元素是以0为基本序号来记数的，即a[0]实际代表的是数组变量a中的第一个元素的值，而之所以是0，表示a所对应的地址加上0*4后得到的地址就为第一个元素的地址。

应该注意不能这样写：long a[0];，定义0个元素的数组是无意义的，编译器将报错，不过在结构或类或联合中符合某些规则后可以这样写，那是C语言时代提出的一种实现结构类型的长度可变的技术，在《C++从零开始（九）》中将说明。

还应注意上面在定义数组时不能在方括号内写变量，即long b = 10; float a[b];是错误的，因为编译此代码时，无法知道变量b的值为多少，进而无法分配内存。可是前面明明已经写了b = 10;，为什么还说不知道b的值？那是因为无法知道b所对应的地址是多少。因为编译器编译时只是将b和一个偏移进行了绑定，并不是真正的地址，即b所对应的可能是Base - 54，而其中的Base就是在程序一开始执行时动态向操作系统申请的大块内存的尾地址，因

为其可能变化，故无法得知 b 实际对应的地址（实际在 Windows 平台下，由于虚拟地址空间的运用，是可以得到实际对应的虚拟地址，但依旧不是实际地址，故无法编译时期知道某变量的值）。

但是编译器仍然可以根据前面的 long b = 10;而推出 Base - 54 的值为 10 啊？重点就是编译器看到 long b = 10;时，只是知道要生成一条指令，此指令将 10 放入 Base - 54 的内存中，其它将不再过问（也没必要过问），故即使才写了 long b = 10;编译器也无法得知 b 的值。

上面说数组是一种类型，其实并不准确，实际应为——数组是一种类型修饰符，其定义了一种类型修饰规则。关于类型修饰符，后面将详述。

字符串

在《C++从零开始（二）》中已经说过，要查某个字符对应的 ASCII 码，通过在这个字符的两侧加上单引号，如'A'就等同于 65。而要表示多个字符时，就使用双引号括起来，如："ABC"。而为了记录字符，就需要记录下其对应的 ASCII 码，而 ASCII 码的数值在-128 到 127 以内，因此使用一个 char 变量就可以记录一个 ASCII 码，而为了记录"ABC"，就很正常地使用一个 char 的数组来记录。如下：

```
char a = 'A'; char b[10]; b[0] = 'A'; b[1] = 'B'; b[2] = 'C';
```

上面 a 的值为 65，b[0]的值为 65，b[1]为 66，b[2]为 67。因为 b 为一个 10 元素的数组，在这其记录了一个 3 个字符长度的字符串，但是当得到 b 的地址时，如何知道其第几个元素才是有效的字符？如上面的 b[4]就没有赋值，那如何知道 b[4]不应该被解释为字符？可以如下，从第 0 个元素开始依次检查每个 char 元素的值，直到遇到某个 char 元素的值为 0（因为 ASCII 码表中 0 没有对应的字符），则其前面的所有的元素都认为是应该用 ASCII 码表来解释的字符。故还应 b[3] = 0;以表示字符串的结束。

上面的规则被广泛运用，C 运行时期库中提供的所有有关字符串的操作都是基于上面的规则来解释字符串的（关于 C 运行时期库，可参考《C++从零开始（十九）》）。但上面为了记录一个字符串，显得烦琐了点，字符串有多长就需要写几个赋值语句，而且还需要将末尾的元素赋值为 0，如果搞忘则问题严重。对于此，C++强制提供了一种简写方式，如下：

```
char b[10] = "ABC";
```

上面就等效于前面所做的所有工作，其中的"ABC"是一个地址类型的数字（准确的说是一初始化表达式，在《C++从零开始（九）》中说明），其类型为 char[4]，即一个 4 个元素的 char 数组，多了一个末尾元素用于放 0 来标识字符串的结束。应当注意，由于 b 为 char[10]，而"ABC"返回的是 char[4]，类型并不匹配，需要隐式类型转换，但实际没有进行转换，而是做了一系列的赋值操作（就如前面所做的工作），这是 C++硬性规定的，称为初始化，且仅仅对于数组定义时进行初始化有效，即如下是错误的：

```
char b[10]; b = "ABC";
```

而即使是 char b[4]; b = "ABC";也依旧错误，因为 b 的类型是数组，表示的是多个元素，而对多个元素赋值是未定义的，即：float d[4]; float dd[4] = d;也是错误的，因为没定义 d 中的元素是依次顺序放到 dd 中的相应各元素，还是倒序放到，所以是不能对一个数组类型的变量进行赋值的。

由于现在字符的增多（原来只用英文字母，现在需要能表示中文、日文等多种字符），原来使用 char 类型来表示字符，最多也只能表示 255 种字符（0 用来表示字符串结束），所以出现了所谓的多字节字符串（MultiByte），用这种表示方式记录的文本文件称为是 MBCS 格式的，而原来使用 char 类型进行表示的字符串称为单字节字符串（SingleByte），用这种表

示方式记录的文本文件称为是 ANSI 格式的。

由于 char 类型可以表示负数，则当从字符串中提取字符时，如果所得元素的数值是负的，则将此元素和下一个 char 元素联合起来形成一 short 类型的数字，再按照 Unicode 编码规则（一种编码规则，等同于前面提过的 ASCII 码表）来解释这个 short 类型的数字以得到相应的字符。

而上面的"ABC"返回的就是以多字节格式表示的字符串，因为没有汉字或特殊符号，故好象是用单字节格式表示的，但如果：char b[10] = "AB 汉 C";，则 b[2]为-70，b[5]为 0，而不是想象的由于 4 个字符故 b[4]为 0，因为“汉”这个字符占用了两个字节。

上面的多字节格式的坏处是每个字符的长度不固定，如果想取字符串中的第 3 个字符的值，则必须从头开始依次检查每个元素的值而不能是 3 乘上某个固定长度，降低了字符串的处理速度，且在显示字符串时由于需要比较检查当前字符的值是否小于零而降低效率，故又推出了第三种字符表示格式：宽字节字符串（WideChar），用这种表示方式记录的文本文件称为是 Unicode 格式的。其与多字节的区别就是不管这个字符是否能够用 ASCII 表示出来，都用一个 short 类型的数字来表示，即每个字符的长度固定为 2 字节，C++对此提供了支持。

short b[10] = L"AB 汉 C";

在双引号的前面加上“L”（必须是大写的，不能小写）即告诉编译器此双引号内的字符要使用 Unicode 格式来编码，故上面的 b 数组就是使用 Unicode 来记录字符串的。同样，也有：short c = L'A';，其中的 c 为 65。

如果上面看得不是很明白，不要紧，在以后举出的例子中将会逐渐了解字符串的使用的。

静态和动态

上面依然没有解决根本问题——C++依旧只能通过变量这个映射元素来访问内存，在访问某块内存前，一定要先建立相应的映射，即定义变量。有什么坏处？让我们先来了解静态和动态是什么意思。

收银员开发票，手动，则每次开发票时，都用已经印好的发票联给客人开发票，发票联上只印了 4 个格子用以记录商品的名称，当客人一次买的商品超过 4 种以上时，就必须开两张或多张发票。这里发票联上的格子的数量就被称作静态的，即无论任何时候任何客人买东西，开发票时发票联上都印着 4 个记录商品名称用的格子。

超市的收银员开发票，将商品名称及数量等输入电脑，然后即时打印出一张发票给客人，则不同的客人，打印出的发票的长度可能不同（有的客人买得多而有的少），此时发票的长度就称为动态的，即不同时间不同客人买东西，开出的发票长度可能不同。

程序无论执行多少遍，在申请内存时总是申请固定大小的内存，则称此内存是静态分配的。前面提出的定义变量时，编译器帮我们从栈上分配的内存就属于静态分配。每次执行程序，根据用户输入的不同而可能申请不同大小的内存时，则称此内存是动态分配的，后面说的从堆上分配就属于动态分配。

很明显，动态比静态的效率高（发票长度的利用率高），但要求更高——需要电脑和打印机，且需要收银员的素质较高（能操作电脑），而静态的要求就较低，只需要已经印好的发票联，且也只需收银员会写字即可。

同样，静态分配的内存利用率不高或运用不够灵活，但代码容易编写且运行速度较快；动态分配的内存利用率高，不过编写代码时要复杂些，需自己处理内存的管理（分配和释放）且由于这种管理的介入而运行速度较慢并代码长度增加。

静态和动态的意义不仅仅如此，其有很多的深化，如硬编码和软编码、紧耦合和松耦合，都是静态和动态的深化。

地址

前面说过“地址就是一个数字，用以唯一标识某一特定内存单元”，而后又说“而地址就和长整型、单精度浮点数这类一样，是数字的一种类型”，那地址既是数字又是数字的类型？不是有点矛盾吗？如下：

浮点数是一种数——小数——又是一种数字类型。即前面的前者是地址实际中的运用，而后者是由于电脑只认识状态，但是给出的状态要如何处理就必须通过类型来说明，所以地址这种类型就是用来告诉编译器以内存单元的标识来处理对应的状态。

指针

已经了解到动态分配内存和静态分配内存的不同，现在要记录用户输入的定单数据，用户一次输入的定单数量不定，故选择在堆上分配内存。假设现在根据用户的输入，需申请1M 的内存以对用户输入的数据进行临时记录，则为了操作这 1M 的连续内存，需记录其首地址，但又由于此内存是动态分配的，即其不是由编译器分配（而是程序的代码动态分配的），故未能建立一变量来映射此首地址，因此必须自己来记录此首地址。

因为任何一个地址都是 4 个字节长的二进制数（对 32 位操作系统），故静态分配一块 4 字节内存来记录此首地址。检查前面，可以将首地址这个数据存在 unsigned long 类型的变量 a 中，然后为了读取此 1M 内存中的第 4 个字节处的 4 字节长内存的内容，通过将 a 的值加上 4 即可获得相应的地址，然后取出其后连续的 4 个字节内存的内容。但是如何编写取某地址对应内存的内容的代码呢？前面说了，只要返回地址类型的数字，由于是地址类型，则其会自动取相应内容的。但如果直接写：a + 4，由于 a 是 unsigned long，则 a + 4 返回的是 unsigned long 类型，不是地址类型，怎么办？

C++对此提出了一个操作符——“*”，叫做取内容操作符（实际这个叫法并不准确）。其和乘号操作符一样，但是它只在右侧接数字，即*(a + 4)。此表达式返回的就是把 a 的值加上 4 后的 unsigned long 数字转成地址类型的数字。但是有个问题：a + 4 所表示的内存的内容如何解释？即取 1 个字节还是 2 个字节？以什么格式来解释取出的内容？如果自己编写汇编代码，这就不是问题了，但现在是编译器代我们编写汇编代码，因此必须通过一种手段告诉编译器如何解释给定的地址所对内存的内容。

C++对此提出了指针，其和上面的数组一样，是一种类型修饰符。在定义变量时，在变量名的前面加上“*”即表示相应变量是指针类型（就如在变量名后接“[]”表示相应变量是数组类型一样），其大小固定为 4 字节。如：

```
unsigned long *pA;
```

上面 pA 就是一个指针变量，其大小因为是为 32 位操作系统编写代码故为 4 字节，当*pA;时，先计算 pA 的值，就是返回从 pA 所对应地址的内存开始，取连续 4 个字节的内容，然后计算“*”，将刚取到的内容转成 unsigned long 的地址类型的数字，接着计算此地址类型的数字，返回以原码格式解释其内容而得到一个 unsigned long 的数字，最后计算这个 unsigned long 的数字而返回以原码格式解释它而得的二进制数。

也就是说，某个地址的类型为指针时，表示此地址对应的内存中的内容，应该被编译器

解释成一个地址。

因为变量就是地址的映射，每个变量都有个对应的地址，为此 C++又提供了一个操作符来取某个变量的地址——“&”，称作取地址操作符。其与“数字与”操作符一样，不过它总是在右侧接数字（而不是两侧接数字）。

“&”的右侧只能接地址类型的数字，它的计算（Evaluate）就是将右侧的地址类型的数字简单的类型转换成指针类型并进而返回一个指针类型的数字，正好和取内容操作符“*”相反。

上面正常情况下应该会让你很晕，下面释疑。

unsigned long a = 10, b, *pA; pA = &a; b = *pA; (*pA)++;

上面的第一句通过“*pA”定义了一个指针类型的变量 pA，即编译器帮我们在栈上分配了一块 4 字节的内存，并将首地址和 pA 绑定（即形成映射）。然后“&a”由于 a 是一个变量，等同于地址，所以“&a”进行计算，返回一个类型为 unsigned long*（即 unsigned long 的指针）的数字。

应该注意上面返回的数字虽然是指针类型，但是其值和 a 对应的地址相同，但为什么不直接说是 unsigned long 的地址的数字，而又多一个指针类型在其中搅和？因为指针类型的数字是直接返回其二进制数值，而地址类型的数字是返回其二进制数值对应的内存的内容。因此假设上面的变量 a 所对应的地址为 2000，则 a;将返回 10，而&a;将返回 2000。

看下指针类型的返回值是什么。当书写 pA;时，返回 pA 对应的地址（按照上面的假设就应该是 2008），计算此地址的值，返回数字 2000（因为已经 pA = &a;），其类型是 unsigned long*，然后对这个 unsigned long*的数字进行计算，直接返回 2000 所对应的二进制数（注意前面红字的内容）。

再来看取内容操作符“*”，其右接的数字类型是指针类型或数组类型，它的计算就是将此指针类型的数字直接转换成地址类型的数字而已（因为指针类型的数字和地址类型的数字在数值上是相同的，仅仅计算规则不同）。所以：

b = *pA;

返回 pA 对应的地址，计算此地址的值，返回类型为 unsigned long*的数字 2000，然后“*pA”返回类型 unsigned long 的地址类型的数字 2000，然后计算此地址类型的数字的值，返回 10，然后就只是简单地赋值操作了。同理，对于++(*pA)（由于“*”的优先级低于前缀++，所以加“()”），先计算“*pA”而返回 unsigned long 的地址类型的数字 2000，然后计算前缀++，最后返回 unsigned long 的地址类型的数字 2000。

如果你还是未能理解地址类型和指针类型的区别，希望下面这句能够有用：地址类型的数字是在编译时期给编译器用的，指针类型的数字是在运行时期给代码用的。如果还是不甚理解，在看过后面的类型修饰符一节后希望能有所帮助。

在堆上分配内存

前面已经说过，所谓的在堆上分配就是运行时期向操作系统申请内存，而要向操作系统申请内存，不同的操作系统提供了不同的接口，具有不同的申请内存的方式，而这主要通过需调用的函数原型不同来表现（关于函数原型，可参考《C++从零开始（七）》）。由于 C++是一门语言，不应该是操作系统相关的，所以 C++提供了一个统一的申请内存的接口，即 new 操作符。如下：

unsigned long *pA = new unsigned long; *pA = 10;

unsigned long *pB = new unsigned long[*pA];

上面就申请了两块内存，pA 所指的内存（即 pA 的值所对应的内存）是 4 字节大小，而 pB 所指的内存是 4*10=40 字节大小。应该注意，由于 new 是一个操作符，其结构为 new <类型名>[<整型数字>]。它返回指针类型的数字，其中的<类型名>指明了什么样的指针类型，而后面方括号的作用和定义数组时一样，用于指明元素的个数，但其返回的并不是数组类型，而是指针类型。

应该注意上面的 new 操作符是向操作系统申请内存，并不是分配内存，即其是有可能失败的。当内存不足或其他原因时，new 有可能返回数值为 0 的指针类型的数字以表示内存分配失败。即可如下检测内存是否分配成功。

```
unsigned long *pA = new unsigned long[10000];
if( !pA )
    // 内存失败！做相应的工作
```

上面的 if 是判断语句，下篇将介绍。如果 pA 为 0，则!pA 的逻辑取反就是非零，故为逻辑真，进而执行相应的工作。

只要分配了内存就需要释放内存，这虽然不是必须的，但是作为程序员，它是一个良好习惯（资源是有限的）。为了释放内存，使用 delete 操作符，如下：

```
delete pA; delete[] pB;
```

注意 delete 操作符并不返回任何数字，但是其仍被称作操作符，看起来它应该被叫做语句更加合适，但为了满足其依旧是操作符的特性，C++提供了一种很特殊的数字类型——void。其表示无，即什么都不是，这在《C++从零开始（七）》中将详细说明。因此 delete 其实是要返回数字的，只不过返回的数字类型为 void 罢了。

注意上面对 pA 和 pB 的释放不同，因为 pA 按照最开始的书写，是 new unsigned long 返回的，而 pB 是 new unsigned long[*pA]返回的。所以需要在释放 pB 时在 delete 的后面加上“[]”以表示释放的是数组，不过在 VC 中，不管前者还是后者，都能正确释放内存，无需“[]”的介入以帮助编译器来正确释放内存，因为以 Windows 为平台而开发程序的 VC 是按照 Windows 操作系统的方式来进行内存分配的，而 Windows 操作系统在释放内存时，无需知道欲释放的内存块的长度，因为其已经在内部记录下来（这种说法并不准确，实际应是 C 运行时期库干了这些事，但其又是依赖于操作系统来干的，即其实是有两层对内存管理的包装，在此不表）。

类型修饰符（type-specifier）

类型修饰符，即对类型起修饰作用的符号，在定义变量时用于进一步指明如何操作变量对应的内存。因为一些通用操作方式，即这种操作方式对每种类型都适用，故将它们单独分离出来以方便代码的编写，就好像水果。吃苹果的果肉、吃梨的果肉，不吃苹果的皮、不吃梨的皮。这里苹果和梨都是水果的种类，相当于类型，而“XXX 的果肉”、“XXX 的皮”就是用于修饰苹果或梨这种类型用的，以生成一种新的类型——苹果的果肉、梨的皮，其就相当于类型修饰符。

本文所介绍的数组和指针都是类型修饰符，之前提过的引用变量的“&”也是类型修饰符，在《C++从零开始（七）》中将再提出几种类型修饰符，到时也将一同说明声明和定义这两个重要概念，并提出声明修饰符（decl-specifier）。

类型修饰符只在定义变量时起作用，如前面的 unsigned long a, b[10], *pA = &a, &rA = a;。这里就使用了上面的三个类型修饰符——“[]”、“*”和“&”。上面的 unsigned long 暂且叫作原类型，表示未被类型修饰符修饰以前的类型。下面分别说明这三个类型修饰符的作用。

数组修饰符“[]”——其总是接在变量名的后面，方括号中间放一整型数 c 以指明数组元素的个数，以表示当前类型为原类型 c 个元素连续存放，长度为原类型的长度乘以 c。因此 long a[10];就表示 a 的类型是 10 个 long 类型元素连续存放，长度为 10*4=40 字节。而 long a[10][4];就表示 a 是 10 个 long[4]类型的元素连续存放，其长度为 10*(4*4)=160 字节。

相信已经发现，由于可以接多个“[]”，因此就有了计算顺序的关系，为什么不是 4 个 long[10]类型的元素连续存放而是倒过来？类型修饰符的修饰顺序是从左向右进行计算的，但当出现重复的类型修饰符时，同类修饰符之间是从右向左计算以符合人们的习惯。故 short *a[10];表示的是 10 个类型为 short*的元素连续存放，长度为 10*4=40 字节，而 short *b[4][10];表示 4 个类型为 short*[10]的元素连续存放，长度为 4*40=160 字节。

指针修饰符“*”——其总是接在变量名的前面，表示当前类型为原类型的指针。故：

short a = 10, *pA = &a, **ppA = &pA;

注意这里的 ppA 被称作多级指针，即其类型为 short 的指针的指针，也就是 short**。而 short **ppA = &pA;的意思就是计算 pA 的地址的值，得一类型为 short*的地址类型的数字，然后“&”操作符将此数字转成 short*的指针类型的数字，最后赋值给变量 ppA。

如果上面很昏，不用去细想，只要注意类型匹配就可以了，下面简要说明一下：假设 a 的地址为 2000，则 pA 的地址为 2002，ppA 的地址为 2006。

对于 pA = &a;。先计算“&a”的值，因为 a 等同于地址，则“&”发挥作用，直接将 a 的地址这个数字转成 short*类型并返回，然后赋值给 pA，则 pA 的值为 2000。

对于 ppA = &pA;。先计算“&pA”的值，因为 pA 等同于地址，则“&”发挥作用，直接将 pA 的地址这个数字转成 short**类型（因为 pA 已经是 short*的类型了）并返回，然后赋值给 ppA，则 ppA 的值为 2002。

引用修饰符“&”——其总是接在变量名的前面，表示此变量不用分配内存以和其绑定，而在说明类型时，则不能有它，下面说明。由于表示相应变量不用分配内存以生成映射，故其不像上述两种类型修饰符，可以多次重复书写，因为没有意义。且其一定在“*”修饰符的右边，即可以 short **&b = ppA;但不能 short *&*b;或 short &**b;因为按照从左到右的修饰符计算顺序，short*&*表示 short 的指针的引用的指针，引用只是告知编译器不要为变量在栈上分配内存，实际与类型无关，故引用的指针是无意义的。而 short&**则表示 short 的引用的指针的指针，同上，依旧无意义。同样 long &a[40];也是错误的，因为其表示分配一块可连续存放类型为 long 的引用的 40 个元素的内存，引用只是告知编译器一些类型无关信息的一种手段，无法作为类型的一种而被实例化（关于实例化，请参看《C++从零开始（十）》）。

应该注意引用并不是类型（但出于方便，经常都将 long 的引用称作一种类型），而 long **&rppA = &pA;将是错误的，因为上句表示的是不要给变量 rppA 分配内存，直接使用“=”后面的地址作为其对应的地址，而&pA 返回的并不是地址类型的数字，而是指针类型，故编译器将报类型不匹配的错误。但是即使 long **&rppA = pA;也同样失败，因为 long*和 long**是不同的，不过由于类型的匹配，下面是可以的（其中的 rpA2 很令人疑惑，将在《C++从零开始（七）》中说明）：

long a = 10, *pA = &a, **ppA = &pA, *&rpA1 = *ppA, *&rpA2 = *(ppA + 1);

类型修饰符和原类型组合在一起以形成新的类型，如 long*&、short *[34]等，都是新的类型，应注意前面 new 操作符中的<类型名>要求写入类型名称，则也可以写上前面的 long*等，即：

long **ppA = new long*[45];

即动态分配一块 4*45=180 字节的连续内存空间，并将首地址返回给 ppA。同样也就可以：

long ***pppA = new long**[2];

而 long *(*pA)[10] = new long*[20][10];

也许看起来很奇怪，其中的 pA 的类型为 long *(*)[10]，表示是一个有 10 个 long*元素的数组的指针，而分配的内存的长度为(4*10)*20=800 字节。因为数组修饰符“[]”只能放在变量名后面，而类型修饰符又总是从左朝右计算，则想说明是一个 10 个 long 元素的数组的指针就不行，因为放在左侧的“*”总是较右侧的“[]”先进行类型修饰。故 C++提出上面的语法，即将变量名用括号括起来，表示里面的类型最后修饰，故：long *(a)[10];等同于 long *a[10];，而 long *(&aa)[10] = a;也才能够正确，否则按照前面的规则，使用 long *&aa[10] = a;将报错（前面已说明原因）。而 long *(*pA)[10] = &a;也就能很正常地表示我们需要的类型了。因此还可以 long *(*&rpA)[10] = pA;以及 long *(**ppA)[10] = &pA;。

限于篇幅，还有部分关于指针的讨论将放到《C++从零开始（七）》中说明，如果本文看得很晕，后面在举例时将会尽量说明指针的用途及用法，希望能有所帮助。

C++从零开始（六）

——何谓语句

前面已经说过程序就是方法的描述，而方法的描述无外乎就是动作加动作的宾语，而这里的动作在 C++中就是通过语句来表现的，而动作的宾语，也就是能够被操作的资源，但非常可惜地 C++语言本身只支持一种资源——内存。由于电脑实际可以操作不止内存这一种资源，导致 C++语言实际并不能作为底层硬件程序的编写语言（即使是 C 语言也不能），不过各编译器厂商都提供了自己的嵌入式汇编语句功能（也可能没提供或提供其它的附加语法以使得可以操作硬件），对于 VC，通过使用__asm 语句即可实现在 C++代码中加入汇编代码来操作其他类型的硬件资源。对于此语句，本系列不做说明。

语句就是动作，C++中共有两种语句：单句和复合语句。复合语句是用一对大括号括起来，以在需要的地方同时放入多条单句，如：{ long a = 10; a += 34; }。而单句都是以“;”结尾的，但也可能由于在末尾要插入单句的地方用复合语句代替了而用“}”结尾，如：if(a) { a--; a++; }。应注意大括号后就不用再写“;”了，因为其不是单句。

方法就是怎么做，而怎么做就是在什么样的情况下以什么样的顺序做什么样的动作。因为 C++中能操作的资源只有内存，故动作也就很简单的只是关于内存内容的运算和赋值取值等，也就是前面说过的表达式。而对于“什么样的顺序”，C++强行规定只能从上朝下，从左朝右来执行单句或复合语句（不要和前面关于表达式的计算顺序搞混了，那只是在一个单句中的规则）。而最后对于“什么样的情况”，即进行条件的判断。为了不同情况下能执行不同的代码，C++定义了跳转语句来实现，其是基于 CPU 的运行规则来实现的，下面先来看 CPU 是如何执行机器代码的。

机器代码的运行方式

前面已经说过，C++中的所有代码到最后都要变成 CPU 能够认识的机器代码，而机器代码由于是方法的描述也就包含了动作和动作的宾语（也可能不带宾语），即机器指令和内存地址或其他硬件资源的标识，并且全部都是用二进制数表示的。很正常，这些代表机器代码的二进制数出于效率的考虑在执行时要放到内存中（实际也可以放在硬盘或其他存储设备中），则很正常地每个机器指令都能有一个地址和其相对应。

CPU 内带一种功能和内存一样的用于暂时记录二进制数的硬件，称作寄存器，其读取速度较内存要快很多，但大小就小许多了。为了加快读取速度，寄存器被去掉了寻址电路进而一个寄存器只能存放 1 个 32 位的二进制数（对于 32 位电脑）。而 CPU 就使用其中的一个寄存器来记录当前欲运行的机器指令的位置，在此称它为指令寄存器。

CPU 运行时，就取出指令寄存器的值，进而找到相应的内存，读取 1 个字节的内容，查看此 8 位二进制数对应的机器指令是什么，进而做相应的动作。由于不同的指令可能有不同数量的参数（即前面说的动作的宾语）需要，如乘法指令要两个参数以将它们乘起来，而取反操作只需要一个参数的参与。并且两个 8 位二进制数的乘法和两个 16 位二进制数的乘法也不相同，故不同的指令带不同的参数而形成的机器代码的长度可能不同。每次 CPU 执行完某条机器代码后，就将指令寄存器的内容加上此机器代码的长度以使指令寄存器指向下一条机器代码，进而重复上面的过程以实现程序的运行（这只是简单地说明，实际由于各种技术的加入，如高速缓冲等，实际的运行过程要比这复杂得多）。

语句的分类

在 C++中，语句总共有 6 种：声明语句、定义语句、表达式语句、指令语句、预编译语句和注释语句。其中的声明语句下篇说明，预编译语句将在《C++从零开始（十六）》中说明，而定义语句就是前面已经见过的定义变量，后面还将说明定义函数、结构等。表达式语句则就是一个表达式直接接一个“;”，如：34;、a = 34;等，以依靠操作符的计算功能的定义而生成相应的关于内存值操作的代码。注释语句就是用于注释代码的语句，即写来给人看的，不是给编译器看的。最后的指令语句就是含有下面所述关键字的语句，即它们的用处不是操作内存，而是实现前面说的“什么样的情况”。

这里的声明语句、预编译语句和注释语句都不会转换成机器代码，即这三种语句不是为了操作电脑，而是其他用途，以后将详述。而定义语句也不一定会生成机器代码，只有表达式语句和指令语句一定会生成代码（不考虑编译器的优化功能）。

还应注意可以写空语句，即;或{}，它们不会生成任何代码，其作用仅仅只是为了保证语法上的正确，后面将看到这一点。下面说明注释语句和指令语句——跳转语句、判断语句和循环语句（实际不止这些，由于异常和模板技术的引入而增加了一些语句，将分别在说明异常和模板时说明）。

注释语句——//、/**/

注释，即用于解释的标注，即一些文字信息，用以向看源代码的人解释这段代码什么意思，因为人的认知空间和电脑的完全不同，这在以后说明如何编程时会具体讨论。要书写一段话用以注释，用“/*”和“*/”将这段话括起来，如下：

```
long a = 1;
a += 1;   /* a 放的是人的个数，让人的个数加一 */
b *= a;   /* b 放的是人均花费，得到总的花费 */
```

上面就分别针对 a += 1;和 b *= a;写了两条注释语句以说明各自的语义（因为只要会 C++都知道它们是一个变量的自增一和另一个变量的自乘 a，但不知道意义）。上面的麻烦之处就是需要写“/*”和“*/”，有点麻烦，故 C++又提供了另一种注释语句——“//”：

```
long a = 1;
```

a += 1; // a 放的是人的个数，让人的个数加一

b *= a; // b 放的是人均花费，得到总的花费

上面和前面等效，其中的“//”表示从它开始，这一行后面的所有字符均看成注释，编译器将不予理会，即

long a = 1; a += 1; // a 放的是人的个数，让人的个数加一 b *= a;

其中的 b *= a;将不会被编译，因为前面的“//”已经告诉编译器，从“//”开始，这一行后面的所有字符均是注释，故编译器不会编译 b *= a;。但如果

long a = 1; a += 1; /* a 放的是人的个数，让人的个数加一 */ b *= a;

这样编译器依旧会编译 b *= a;，因为“/*”和“*/”括起来的才是注释。

应该注意注释语句并不是语句，其不以“;”结束，其只是另一种语法以提供注释功能，就好象以后将要说明的预编译语句一样，都不是语句，都不以“;”结束，既不是单句也不是复合语句，只是出于习惯的原因依旧将它们称作语句。

跳转语句——goto

前面已经说明，源代码（在此指用 C++编写的代码）中的语句依次地转变成用长度不同的二进制数表示的机器代码，然后顺序放在内存中（这种说法不准确）。如下面这段代码：

long a = 1; // 假设长度为 5 字节，地址为 3000

a += 1; // 则其地址为 3005，假设长度为 4 字节

b *= a; // 则其地址为 3009，假设长度为 6 字节

上面的 3000、3005 和 3009 就表示上面 3 条语句在内存中的位置，而所谓的跳转语句，也就是将上面的 3000、3005 等语句的地址放到前面提过的指令寄存器中以使得 CPU 开始从给定的位置执行以表现出执行顺序的改变。因此，就必须有一种手段来表现语句的地址，C++对此给出了标号（Label）。

写一标识符，后接“:”即建立了一映射，将此标识符和其所在位置的地址绑定了起来，如下：

long a = 1; // 假设长度为 5 字节，地址为 3000

P1:

a += 1; // 则其地址为 3005，假设长度为 4 字节

P2:

b *= a; // 则其地址为 3009，假设长度为 6 字节

goto P2;

上面的 P1 和 P2 就是标号，其值分别为 3005 和 3009，而最后的 goto 就是跳转语句，其格式为 goto <标号>;。此语句非常简单，先通过“:”定义了一个标号，然后在编写 goto 时使用不同的标号就能跳到不同的位置。

应该注意上面故意让 P1 和 P2 定义时独占一行，其实也可以不用，即：

long a = 1; P1: a += 1; P2: b *= a; goto P2;

因此看起来“P1:”和“P2:”好象是单独的一条定义语句，应该注意，准确地说它们应该是语句修饰符，作用是定义标号，并不是语句，即这样是错误的：

long a = 1; P1: { a += 1; P2: b *= a; P3: } goto P2;

上面的 P3:将报错，因为其没有修饰任何语句。还应注意其中的 P1 仍然是 3005，即“{}”仅仅只是其复合的作用，实际并不产生代码进而不影响语句的地址。

判断语句——if else、switch

if else　前面说过了，为了实现“什么样的情况”做“什么样的动作”，故 C++非常正常地提供了条件判断语句以实现条件的不同而执行不同的代码。if else 的格式为：

if(<数字>)<语句 1>else<语句 2>　或者　if(<数字>)<语句 1>

```
long a = 0, b = 1;
P1:
    a++;
    b *= a;
    if( a < 10 )
        goto P1;
    long c = b;
```

上面的代码就表示只有当 a 的值小于 10 时，才跳转到 P1 以重复执行，最后的效果就是 c 的值为 10 的阶乘。

上面的<数字>表示可以在“if”后的括号中放一数字，即表达式，而当此数字的值非零时，即逻辑真，程序跳转以执行<语句 1>，如果为零，即逻辑假，则执行<语句 2>。即也可如此：if(a － 10) goto P1;，其表示当 a － 10 不为零时才执行 goto P1;。这和前面的效果一样，虽然最后 c 仍然是 10 的阶乘，但意义不同，代码的可读性下降，除非出于效率的考虑，不推荐如此书写代码。

而<语句 1>和<语句 2>由于是语句，也就可以放任何是语句的东西，因此也可以这样：

```
if( a ) long c;
```

上面可谓吃饱了撑了，在此只是为了说明<语句 1>实际可以放任何是语句的东西，但由于前面已经说过，标号的定义以及注释语句和预编译语句其实都不是语句，因此下面试图当 a 非零时，定义标号 P2 和当 a 为零时书写注释“错误!”的意图是错误的：

```
if( a ) P2:          或者      if( !a )   // 错误!
a++;                                  a++;
```

但编译器不会报错，因为前者实际是当 a 非零时，将 a 自增一；后者实际是当 a 为零时，将 a 自增一。还应注意，由于复合语句也是语句，因此：

```
if( a ){ long c = 0; c++; }
```

由于使用了复合语句，因此这个判断语句并不是以“;”结尾，但它依旧是一个单句，即：

```
if( a )
    if( a < 10 ) { long c = 0; c++; }
else
    b *= a;
```

上面虽然看起来很复杂，但依旧是一个单句，应该注意当写了一个“else”时，编译器向上寻找最近的一个“if”以和其匹配，因此上面的“else”是和“if(a < 10)”匹配的，而不是由于上面那样的缩进书写而和“if(a)”匹配，因此 b *= a;只有在 a 大于等于 10 的时候才执行，而不是想象的 a 为零的时候。

还应注意前面书写的 if(a) long c;。这里的意思并不是如果 a 非零，就定义变量 c，这里涉及到作用域的问题，将在下篇说明。

switch　这个语句的定义或多或少地是因为实现的原因而不是和“if else”一样由于逻辑的原因。先来看它的格式：switch(<整型数字>)<语句>。

上面的<整型数字>和 if 语句一样，只要是一个数字就可以了，但不同地必须是整型数字（后面说明原因）。然后其后的<语句>与前相同，只要是语句就可以。在<语句>中，应该使用这样的形式：case <整型常数 1>:。它在它所对应的位置定义了一个标号，即前面 goto 语句使用的东西，表示如果<整型数字>和<整型常数 1>相等，程序就跳转到“case <整型常数 1>:”所标识的位置，否则接着执行后续的语句。

```
long a, b = 3;
switch( a + 3 )
case 2: case 3: a++;
b *= a;
```

上面就表示如果 a + 3 等于 2 或 3，就跳到 a++;的地址，进而执行 a++，否则接着执行后面的语句 b *= a;。这看起来很荒谬，有什么用？一条语句当然没意义，为了能够标识多条语句，必须使用复合语句，即如下：

```
long a, b = 3;
switch( a + 3 )
{
    b = 0;
case 2:
    a++;        // 假设地址为 3003
case 3:
    a--;        // 假设地址为 3004
    break;
case 1:
    a *= a;    // 假设地址为 3006
}
b *= a;         // 假设地址为 3010
```

应该注意上面的“2:”、“3:”、“1:”在这里看着都是整型的数字，但实际应该把它们理解为标号。因此，上面检查 a + 3 的值，如果等于 1，就跳到“1:”标识的地址，即 3006；如果为 2，则跳转到 3003 的地方执行代码；如果为 3，则跳到 3004 的位置继续执行。而上面的 break;语句是特定的，其放在 switch 后接的语句中表示打断，使程序跳转到 switch 以后，对于上面就是 3010 以执行 b *= a;。即还可如此：

```
switch( a ) if( a ) break;
```

由于是跳到相应位置，因此如果 a 为-1，则将执行 a++;，然后执行 a--;，再执行 break;而跳到 3010 地址处执行 b *= a;。并且，上面的 b = 0;将永远不会被执行。

switch 表示的是针对某个变量的值，其不同的取值将导致执行不同的语句，非常适合实现状态的选择。比如用 1 表示安全，2 表示有点危险，3 表示比较危险而 4 表示非常危险，通过书写一个 switch 语句就能根据某个怪物当前的状态来决定其应该做“逃跑”还是“攻击”或其他的行动以实现游戏中的人工智能。那不是很奇怪吗？上面的 switch 通过 if 语句也可以实现，为什么要专门提供一个 switch 语句？如果只是为了简写，那为什么不顺便提供多一些类似这种逻辑方案的简写，而仅仅只提供了一个分支选择的简写和后面将说的循环的简写？因为其是出于一种优化技术而提出的，就好象后面的循环语句一样，它们对逻辑的贡献都可以通过 if 语句来实现（毕竟逻辑就是判断），而它们的提出一定程度都是基于某种优化技术，不过后面的循环语句简写的成分要大一些。

我们给出一个数组，数组的每个元素都是 4 个字节大小，则对于上面的 switch 语句，如下：

unsigned long Addr[3]; Addr[0] = 3006; Addr[1] = 3003; Addr[2] = 3004;

而对于 switch(a + 3)，则使用类似的语句就可以代替：goto Addr[a + 3 － 1];

上面就是 switch 的真面目，应注意上面的 goto 的写法是错误的，这也正是为什么会有 switch 语句。编译器为我们构建一个存储地址的数组，这个数组的每个元素都是一个地址，其表示的是某条语句的地址，这样，通过不同的偏移即可实现跳转到不同的位置以执行不同的语句进而表现出状态的选择。

现在应该了解为什么上面必须是<整型数字>了，因为这些数字将用于数组的下标或者是偏移，因此必须是整数。而<整型常数 1>必须是常数，因为其由编译时期告诉编译器它现在所在位置应放在地址数组的第几个元素中。

了解了 switch 的实现后，以后在书写 switch 时，应尽量将各 case 后接的整型常数或其倍数靠拢以减小需生成的数组的大小，而无需管常数的大小。即 case 1000、case1001、case 1002 和 case 2、case 4、case 6 都只用 3 个元素大小的数组，而 case 0、case 100、case 101 就需要 102 个元素大小的数组。应该注意，现在的编译器都很智能，当发现如刚才的后者这种只有 3 个分支却要 102 个元素大小的数组时，编译器是有可能使用重复的 if 语句来代替上面数组的生成。

switch 还提供了一个关键字——default。如下：

```
long a, b = 3;
switch( a + 3 )
{
case 2:
    a++;
    break;
case 3:
    a += 3;
    break;
default:
    a--;
}
b *= a;
```

上面的“default:”表示当 a + 3 不为 2 且不为 3 时，则执行 a--;，即 default 表示缺省的状况，但也可以没有，则将直接执行 switch 后的语句，因此这是可以的：switch(a){}或 switch(a);，只不过毫无意义罢了。

循环语句——for、while、do while

刚刚已经说明，循环语句的提供主要是出于简写目的，因为循环是方法描述中用得最多的，且算法并不复杂，进而对编译器的开发难度不是增加太多。

for　其格式为 for(<数字 1>;<数字 2>;<数字 3>)<语句>。其中的<语句>同上，即可接单句也可接复合语句。而<数字 1>、<数字 2>和<数字 3>由于是数字，就是表达式，进而可以做表达式语句能做的所有的工作——操作符的计算。for 语句的意思是先计算<数字 1>，相当于初始化工作，然后计算<数字 2>。如果<数字 2>的值为零，表示逻辑假，则退出循环，执行 for 后面的语句，否则执行<语句>，然后计算<数字 3>，相当于每次循环的例行公事，接着再计算<数字 2>，并重复。上面的<语句>一般被称作循环体。

上面的设计是一种面向过程的设计思想，将循环体看作是一个过程，则这个过程的初始化（<数字 1>）和必定执行（<数字 3>）都表现出来。一个简单的循环，如下：

```
long a, b;
for( a = 1, b = 1; a <= 10; a++ )
    b *= a;
```

上面执行完后 b 是 10 的阶乘，和前面在说明 if 语句时举的例子相比，其要简单地多，并且可读性更好——a = 1, b = 1 是初始化操作，每次循环都将 a 加一，这些信息是 goto 和 if 语句表现不出来的。由于前面一再强调的语句和数字的概念，因此可以如下：

```
long a, b = 1;
for( ; b < 100; )
    for( a = 1, b = 1; a; ++a, ++b )
        if( b *= a )
            switch( a = b )
            {
            case 1:
                a++; break;
            case 2:
                for( b = 10; b; b-- )
                {
                    a += b * b;
                    case 3: a *= a;
                }
                break;
            }
```

上面看着很混乱，注意“case 3:”在“case 2:”后的一个 for 语句的循环体中，也就是说，当 a = b 返回 1 时，跳到 a++;处，并由于 break;的缘故而执行 switch 后的语句，也就是 if 后的语句，也就是第二个 for 语句的++a, ++b。当返回 2 时，跳到第三个 for 语句处开始执行，循环完后同样由 break;而继续后面的执行。当返回 3 时，跳到 a *= a;处执行，然后计算 b--，接着计算 b 的值，检查是否非零，然后重复循环直到 b 的值为零，然后继续以后的执行。上面的代码并没什么意义，在这里是故意写成这么混乱以进一步说明前面提过的语句和数字的概念，如果真正执行，大致看过去也很容易知道将是一个死循环，即永远循环无法退出的循环。

还应注意 C++提出了一种特殊语法，即上面的<数字 1>可以不是数字，而是一变量定义语句，即可如此：for(long a = 1, b = 1; a < 10; ++a, ++b);。其中就定义了变量 a 和 b。但是也只能接变量定义语句，而结构定义、类定义及函数定义语句将不能写在这里。这个语法的提出是更进一步地将 for 语句定义为记数式循环的过程，这里的变量定义语句就是用于定义此循环中充当计数器的变量（上面的 a）以实现循环固定次数。

最后还应注意上面写的<数字 1>、<数字 2>和<数字 3>都是可选的，即可以：for(;;);。

while　其格式为 while(<数字>)<语句>，其中的<数字>和<语句>都同上，意思很明显，当<数字>非零时，执行<语句>，否则执行 while 后面的语句，这里的<语句>被称作循环体。

do while　其格式为 do<语句>while(<数字>);。注意，在 while 后接了“;”以表示这个单句的结束。其中的<数字>和<语句>都同上，意思很明显，当<数字>非零时，执行<语句>，否则执行 while 后面的语句，这里的<语句>被称作循环体。

为什么 C++要提供上面的三种循环语句？简写是一重要目的，但更重要的是可以提供一定的优化。for 被设计成用于固定次数的循环，而 while 和 do while 都是用于条件决定的循环。对于前者，编译器就可以将前面提过的用于记数的变量映射成寄存器以优化速度，而后者就要视编译器的智能程度来决定是否能生成优化代码了。

while 和 do while 的主要区别就是前者的循环体不一定会被执行，而后者的循环体一定至少会被执行一次。而出于简写的目的，C++又提出了 continue 和 break 语句。如下：

```
for( long i = 0; i < 10; i++ )
{
    if( !( i % 3 ) )
        continue;
    if( !( i % 7 ) )
        break;
    // 其他语句
}
```

上面当 i 的值能被 3 整除时，就不执行后面的“其他语句”，而是直接计算 i++，再计算 i < 10 以决定是否继续循环。即 continue 就是终止当前这次循环的执行，开始下一次的循环。上面当 i 的值能被 7 整除时，就不执行后面的“其他语句”，而是跳出循环体，执行 for 后的语句。即 break 就是终止循环的运行，立即跳出循环体。如下：

```
while( --i )
{
    if( i == 10 )
        continue;
    if( i > 20 )
        break;
    // 其他语句
}
a = i;
```

```
do
{
    if( i == 10 )
        continue;
    if( i > 20 )
        break;
    // 其他语句
}while( --i );
a = i;
```

上面的 continue;执行时都将立即计算—i 以判断是否继续循环，而 break;执行时都将立即退出循环体进而执行后继的 a = i;。

还应注意嵌套问题，即前面说过的 else 在寻找配对的 if 时，总是找最近的一个 if，这里依旧。

```
long a = 0;
P1:
for( long i = a; i < 10; i++ )
    for( long j = 0; j < 10; j++ )
    {
        if( !( j % 3 ) )
            continue;
        if( !( j % 7 ) )
            break;
        if( i * j )
        {
            a = i * j;
            goto P1;
```

```
        }
        // 其他语句
    }
```

上面的 continue;执行后，将立即计算 j++，而 break;执行后，将退出第二个循环（即 j 的循环），进而执行 i++，然后继续由 i < 10 来决定是否继续循环。当 goto P1;执行时，程序跳到上面的 P1 处，即执行 long i = a;，进而重新开始 i 的循环。

上面那样书写 goto 语句是不被推荐的，因为其破坏了循环，不符合人的思维习惯。在此只是要说明，for 或 while、do while 等都不是循环，只是它们各自的用处最后表现出来好象是循环，实际只是程序执行位置的变化。应清楚语句的实现，这样才能清楚地了解各种语句的实际作用，进而明确他人写的代码的意思。而对于自己书写代码，了解语句的实现，将有助于进行一定的优化。但当你写出即精简又执行效率高的程序时，保持其良好的可读性是一个程序员的素养，应尽量培养自己书写可读性高的代码的习惯。

上面的 long j = 0 在第一个循环的循环体内，被多次执行岂不是要多次定义？这属于变量的作用域的问题，下篇将说明。

本篇的内容应该是很简单的，重点只是应该理解源代码编译成机器指令后，在执行时也放在内存中，故每条语句都对应着一个地址，而通过跳转语句即可改变程序的运行顺序。下篇将对此提出一系列的概念，并说明声明和定义的区别。

C++从零开始（七）

——何谓函数

本篇之前的内容都是基础中的基础，理论上只需前面所说的内容即可编写出几乎任何只操作内存的程序，也就是本篇以后说明的内容都可以使用之前的内容自己实现，只不过相对要麻烦和复杂许多罢了。

本篇开始要比较深入地讨论 C++提出的很有意义的功能，它们大多数和前面的 switch 语句一样，是一种技术的实现，但更为重要的是提供了语义的概念。所以，本篇开始将主要从它们提供的语义这方面来说明各自的用途，而不像之前通过实现原理来说明（不过还是会说明一下实现原理的）。为了能清楚说明这些功能，要求读者现在至少能使用 VC 来编译并生成一段程序，因为后续的许多例子都最好是能实际编译并观察执行结果以加深理解（尤其是声明和类型这两个概念）。为此，如果你现在还不会使用 VC 或其他编译器来进行编译代码，请先参看其他资料以了解如何使用 VC 进行编译。为了后续例子的说明，下面先说明一些预备知识。

预备知识

写出了 C++代码，要如何让编译器编译？在文本文件中书写 C++代码，然后将文本文件的文件名作为编译器的输入参数传递给编译器，即叫编译器编译给定文件名所对应的文件。在 VC 中，这些由 VC 这个编程环境（也就是一个软件，提供诸多方便软件开发的功能）帮我们做了，其通过项目（Project）来统一管理书写有 C/C++代码的源文件。为了让 VC 能了解到哪些文件是源文件（因为还可能有资源文件等其他类型文件），在用文本编辑器书写了 C++代码后，将其保存为扩展名为.c 或.cpp（C Plus Plus）的文本文件，前者表示是 C 代码，

而后者表示 C++代码，则缺省情况下，VC 就能根据不同的源文件而使用不同的编译语法来编译源文件。

前篇说过，C++中的每条语句都是从上朝下执行，每条语句都对应着一个地址，那么在源文件中的第一条语句对应的地址就是 0 吗？当然不是，和在栈上分配内存一样，只能得到相对偏移值，实际的物理地址由于不同的操作系统将会有各自不同的处理，如在 Windows 下，代码甚至可以没有物理地址，且代码对应的物理地址还能随时变化。

当要编写一个稍微正常点的程序时，就会发现一个源文件一般是不够的，需要使用多个源文件来写代码。而各源文件之间要如何连接起来？对此 C++规定，凡是生成代码的语句都要放在函数中，而不能直接写在文本文件中。关于函数后面马上说明，现在只需知道函数相当于一个外壳，它通过一对“{}”将代码括起来，进而就将代码分成了一段一段，且每一段代码都由函数名这个项目内唯一的标识符来标识，因此要连接各段代码，只用通过函数名即可，后面说明。前面说的“生成代码”指的是表达式语句和指令语句，虽然定义语句也可能生成代码，但由于其代码生成的特殊性，是可以直接写在源文件内（在《C++从零开始（十）》中说明），即不用被一对“{}”括起来。

程序一开始要从哪里执行？C++强行规定，应该在源文件中定义一个名为 main 的函数，而代码就从这个函数处开始运行。应该注意由于 C++是由编译器实现的，而它的这个规定非常的牵强，因此纵多的编译器都又自行提供了另外的程序入口点定义语法（程序入口点即最开始执行的函数），如 VC，为了编写 DLL 文件，就不应有 main 函数；为了编写基于 Win32 的程序，就应该使用 WinMain 而不是 main；而 VC 实际提供了更加灵活的手段，实际可以让程序从任何一个函数开始执行，而不一定非得是前面的 WinMain、main 等，这在《C++从零开始（十九）》中说明。

对于后面的说明，应知道程序从 main 函数开始运行，如下：

```
long a; void main(){ short b; b++; } long c;
```

上面实际先执行的是 long a;和 long c;，不过不用在意，实际有意义的语句是从 short b; 开始的。

函数（Function）

机器手焊接轿车车架上的焊接点，给出焊接点的三维坐标，机器手就通过控制各关节的马达来使焊枪移到准确的位置。这里控制焊枪移动的程序一旦编好，以后要求机器手焊接车架上的 200 个点，就可以简单地给出 200 个点的坐标，然后调用前面已经编好的移动程序 200 次就行了，而不用再对每次移动重复编写代码。上面的移动程序就可以用一个函数来表示。

函数是一个映射元素。其和变量一样，将一个标识符（即函数名）和一个地址关联起来，且也有一类型和其关联，称作函数的返回类型。函数和变量不同的就是函数关联的地址一定是代码的地址，就好像前面说明的标号一样，但和标号不同的就是，C++将函数定义为一种类型，而标号则只是纯粹的二进制数，即函数名对应的地址可以被类型修饰符修饰以使得编译器能生成正确的代码来帮助程序员书实现上面的功能。

由于定义函数时编译器并不会分配内存，因此引用修饰符“&”不再其作用，同样，由数组修饰符“[]”的定义也能知道其不能作用于函数上面，只有留下的指针修饰符“*”可以，因为函数名对应的是某种函数类型的地址类型的数字。

前面移动程序之所以能被不同地调用 200 次，是因为其写得很灵活，能根据不同的情况（不同位置的点）来改变自己的运行效果。为了向移动程序传递用于说明情况的信息（即点

的坐标），必须有东西来完成这件事，在 C++中，这使用参数来实现，并对于此，C++专门提供了一种类型修饰符——函数修饰符“()”。在说明函数修饰符之前，让我们先来了解何谓抽象声明符（Abstract Declarator）。

声明一个变量 long a;（这看起来和定义变量一样，后面将说明它们的区别），其中的 long 是类型，用于修饰此变量名 a 所对应的地址。将声明变量时（即前面的写法）的变量名去掉后剩下的东西称作抽象声明符。比如：long *a, &b = *a, c[10], (*d)[10];，则变量 a、b、c、d 所对应的声明修饰符分别是 long*、long&、long[10]、long(*)[10]。

函数修饰符接在函数名的后面，括号内接零个或多个抽象声明符以表示参数的类型，中间用“,”隔开。而参数就是一些内存（分别由参数名映射），用于传递一些必要的信息给函数名对应的地址处的代码以实现相应的功能。声明一个函数如下：

long *ABC(long*, long&, long[10], long(*)[10]);

上面就声明了一个函数 ABC，其类型为 long*(long*, long&, long[10], long(*)[10])，表示欲执行此函数对应地址处开始的代码，需要顺序提供 4 个参数，类型如上，返回值类型为 long*。上面 ABC 的类型其实就是一个抽象声明符，因此也可如下：

long AB(long*(long*, long&, long[10], long(*)[10]), short, long&);

对于前面的移动程序，就可类似如下声明它：

void Move(float x, float y, float z);

上面在书写声明修饰符时又加上了参数名，以表示对应参数的映射。不过由于这里是函数的声明，上述参数名实际不产生任何映射，因为这是函数的声明，不是定义（关于声明，后面将说明）。而这里写上参数名是一种语义的体现，表示第一、二、三个参数分别代表 X、Y、Z 坐标值。

上面的返回类型为 void，前面提过，void 是 C++提供的一种特殊数字类型，其仅仅只是为了保障语法的严密性而已，即任何函数执行后都要返回一个数字（后面将说明），而对于不用返回数字的函数，则可以定义返回类型为 void，这样就可以保证语法的严密性。应当注意，任何类型的数字都可以转换成 void 类型，即可以(void)(234);或 void(a);。

注意上面函数修饰符中可以一个抽象修饰符都没有，即 void ABC();。它等效于 void ABC(void);，表示 ABC 这个函数没有参数且不返回值。则它们的抽象声明符为 void()或 void(void)，进而可以如下：

long* ABC(long*(), long(), long[10]);

由函数修饰符的意义即可看出其和引用修饰符一样，不能重复修饰类型，即不能 void A()(long);，这是无意义的。同样，由于类型修饰符从左朝右的修饰顺序，也就很正常地有：void(*pA)()。假设这里是一个变量定义语句（也可以看成是一声明语句，后面说明），则表示要求编译器在栈上分配一块 4 字节的空间，将此地址和 pA 映射起来，其类型为没有参数，返回值类型为 void 的函数的指针。有什么用？以后将说明。

函数定义

下面先看下函数定义，对于前面的机器手控制程序，可如下书写：

```
void Move( float x, float y, float z )
{
    float temp;
    // 根据 x、y、z 的值来移动焊枪
}
```

```
int main()
{
    float x[200], y[200], z[200];
    // 将 200 个点的坐标放到数组 x、y 和 z 中
    for( unsigned i = 0; i < 200; i++ )
        Move( x[ i ], y[ i ], z[ i ] );
    return 0;
}
```

上面定义了一个函数 Move，其对应的地址为定义语句 float temp;所在的地址，但实际由于编译器要帮我们生成一些附加代码（称作函数前缀——Prolog，在《C++从零开始（十五）》中说明）以获得参数的值或其他工作（如异常的处理等），因此 Move 将对应在较 float temp;之前的某个地址。Move 后接的类型修饰符较之前有点变化，只是把变量名加上以使其不是抽象声明符而已，其作用就是让编译器生成一映射，将加上的变量名和传递相应信息的内存的地址绑定起来，也就形成了所谓的参数。也由于此原因，就能如此书写：void Move(float x, float, float z) { }。由于没有给第二个参数绑定变量名，因此将无法使用第二个参数，以后将举例说明这样的意义。

函数的定义就和前面的函数的声明一样，只不过必须紧接其后书写一个复合语句（必须是复合语句，即用“{}”括起来的语句），此复合语句的地址将和此函数名绑定，但由于前面提到的函数前缀，函数名实际对应的地址在复合语句的地址的前面。

为了调用给定函数，C++提供了函数操作符“()”，其前面接函数类型的数字，而中间根据相应函数的参数类型和个数，放相应类型的数字和个数，因此上面的 Move(x[i], y[i], z[i]);就是使用了函数操作符，用 x[i]、y[i]、z[i]的值作为参数，并记录下当前所在位置的地址，跳转到 Move 所对应的地址继续执行，当从 Move 返回时，根据之前记录的位置跳转到函数调用处的地方，继续后继代码的执行。

函数操作符由于是操作符，因此也要返回数字，也就是函数的返回值，即可以如下：

```
float AB( float x ) { return x * x; } int main() { float c = AB( 10 ); return 0; }
```

先定义了函数 AB，其返回 float 类型的数字，其中的 return 语句就是用于指明函数的返回值，其后接的数字就必须是对应函数的返回值类型，而当返回类型为 void 时，可直接书写 return;。因此上面的 c 的值为 100，函数操作符返回的值为 AB 函数中的表达式 x * x 返回的数字，而 AB(10)将 10 作为 AB 函数的参数 x 的值，故 x * x 返回 100。

由于之前也说明了函数可以有指针，将函数和变量对比，则直接书写函数名，如：AB;。上面将返回 AB 对应的地址类型的数字，然后计算此地址类型数字，应该是以函数类型解释相应地址对应的内存的内容，考虑函数的意义，将发现这是毫无意义的，因此其不做任何事，直接返回此地址类型的数字对应的二进制数，也就相当于前面说的指针类型。因此也就可以如下：

```
int main() { float (*pAB)( float ) = AB; float c = ( *pAB )( 10 ); return 0; }
```

上面就定义了一个指针 pAB，其类型为 float(*)(float)，一开始将 AB 对应的地址赋值给它。为什么没有写成 pAB = &AB;而是 pAB = AB;？因为前面已经说了，函数类型的地址类型的数字，将不做任何事，其效果和指针类型的数字一样，因此 pAB = AB;没有问题，而 pAB = &AB;就更没有问题了。可以认为函数类型的地址类型的数字编译器会隐式转换成指针类型的数字，因此既可以(*pAB)(10);，也能(*AB)(10);，因为后者编译器进行了隐式类型转换。

由于函数操作符中接的是数字，因此也可以 float c = AB(AB(10));，即 c 为 10000。还应注意函数操作符让编译器生成一些代码来传递参数的值和跳转到相应的地址去继续执行代码，因此如下是可以的：

long AB(long x) { if(x > 1) return x * AB(x - 1); else return 1; }

上面表示当参数 x 的值大于 1 时，将 x - 1 返回的数字作为参数，然后跳转到 AB 对应的地址处，也就是 if(x > 1)所对应的地址，重复运行。因此如果 long c = AB(5);，则 c 为 5 的阶乘。上面如果不能理解，将在后面说明异常的时候详细说明函数是如何实现的，以及所谓的堆栈溢出问题。

现在应该了解 main 函数的意义了，其只是建立一个映射，好让连接器制定程序的入口地址，即 main 函数对应的地址。上面函数 Move 在函数 main 之前定义，如果将 Move 的定义移到 main 的下面，上面将发生错误，说函数 Move 没定义过，为什么？因为编译器只从上朝下进行编译，且只编译一次。那上面的问题怎么办？后面说明。

重载函数

前面的移动函数，如果只想移动 X 和 Y 坐标，为了不移动 Z 坐标，就必须如下再编写一个函数：

void Move2(float x, float y);

它为了不和前面的 Move 函数的名字冲突而改成 Move2，但 Move2 也表示移动，却非要变一个名字，这严重地影响语义。为了更好的从源代码上表现出语义，即这段代码的意义，C++提出了重载函数的概念。

重载函数表示函数名字一样，但参数类型及个数不同的多个函数。如下：

void Move(float x, float y, float z) { }和 void Move(float x, float y) { }

上面就定义了两个重载函数，虽然函数名相同，但实际为两个函数，函数名相同表示它们具有同样的语义——移动焊枪的程序，只是移动方式不同，前者在三维空间中移动，后者在一水平面上移动。当 Move(12, 43);时就调用后者，而 Move(23, 5, 12);时就调用前者。不过必须是参数的不同，不能是返回值的不同，即如下将会报错：

float Move(float x, float y) { return 0; }和 void Move(float a, float b) { }

上面虽然返回值不同，但编译器依旧认为上面定义的函数是同一个，则将说函数重复定义。为什么？因为在书写函数操作符时，函数的返回值类型不能保证获得，即 float a = Move(1, 2);虽然可以推出应该是前者，但也可以 Move(1, 2);，这样将无法得知应该使用哪个函数，因此不行。还应注意上面的参数名字虽然不同，但都是一样的，参数名字只是表示在那个函数的作用域内其映射的地址，后面将说明。改成如下就没有问题：

float Move(float x, float y) { return 0; }和 void Move(float a, float b, float c) { }

还应注意下面的问题：

float Move(float x, char y); float Move(float a, short b); Move(10, 270);

上面编译器将报错，因为这里的 270 在计算函数操作符时将被认为是 int，即整型，它即可以转成 char，也可以转成 short，结果编译器将无法判断应是哪一个函数。为此，应该 Move(10, (char)270);。

声明和定义

声明是告诉编译器一些信息，以协助编译器进行语法分析，避免编译器报错。而定义是告诉编译器生成一些代码，并且这些代码将由连接器使用。即：声明是给编译器用的，定义是给连接器用的。这个说明显得很模糊，为什么非要弄个声明和定义在这搅和？那都是因为

C++同意将程序拆成几段分别书写在不同文件中以及上面提到的编译器只从上朝下编译且对每个文件仅编译一次。

编译器编译程序时，只会一个一个源文件编译，并分别生成相应的中间文件（对 VC 就是.obj 文件），然后再由连接器统一将所有的中间文件连接形成一个可执行文件。问题就是编译器在编译 a.cpp 文件时，发现定义语句而定义了变量 a 和 b，但在编译 b.cpp 时，发现使用 a 和 b 的代码，如 a++;，则编译器将报错。为什么？如果不报错，说因为 a.cpp 中已经定义了，那么先编译 b.cpp 再编译 a.cpp 将如何？如果源文件的编译顺序是特定的，将大大降低编译的灵活性，因此 C++也就规定：编译 a.cpp 时定义的所有东西（变量、函数等）在编译 b.cpp 时将全部不算数，就和没编译过 a.cpp 一样。那么 b.cpp 要使用 a.cpp 中定义的变量怎么办？为此，C++提出了声明这个概念。

因此变量声明 long a;就是告诉编译器已经有这么个变量，其名字为 a，其类型为 long，其对应的地址不知道，但可以先作个记号，即在后续代码中所有用到这个变量的地方做上记号，以告知连接器在连接时，先在所有的中间文件里寻找是否有个叫 a 的变量，其地址是多少，然后再修改所有作了记号的地方，将 a 对应的地址放进去。这样就实现了这个文件使用另一个文件中定义的变量。

所以声明 long a;就是要告诉编译器已经有这么个变量 a，因此后续代码中用到 a 时，不要报错说 a 未定义。函数也是如此，但是有个问题就是函数声明和函数定义很容易区别，因为函数定义后一定接一复合语句，但是变量定义和变量声明就一模一样，那么编译器将如何识别变量定义和变量声明？编译器遇到 long a;时，统一将其认为是变量定义，为了能标识变量声明，可借助 C++提出的修饰符 extern。

修饰符就是声明或定义语句中使用的用以修饰此声明或定义来向编译器提供一定的信息，其总是接在声明或定义语句的前面或后面，如：

extern long a, *pA, &ra;

上面就声明（不是定义）了三个变量 a、pA 和 ra。因为 extern 表示外部的意思，因此上面就被认为是告诉编译器有三个外部的变量，为 a、pA 和 ra，故被认为是声明语句，所以上面将不分配任何内存。同样，对于函数，它也是一样的：

extern void ABC(long);　或　extern long AB(short b);

上面的 extern 等同于不写，因为编译器根据最后的“;”就可以判断出来上面是函数声明，而且提供的“外部”这个信息对于函数来说没有意义，编译器将不予理会。extern 实际还指定其后修饰的标识符的修饰方式，实际应为 extern"C"或 extern"C++"，分别表示按照 C 语言风格和 C++语言风格来解析声明的标识符。

C++是强类型语言，即其要求很严格的类型匹配原则，进而才能实现前面说的函数重载功能。即之所以能几个同名函数实现重载，是因为它们实际并不同名，而由各自的参数类型及个数进行了修饰而变得不同。如 void ABC(), *ABC(long), ABC(long, short);，在 VC 中，其各自名字将分别被变成“?ABC@@YAXXZ”、“?ABC@@YAPAXJ@Z”、“?ABC@@YAXJF@Z”。而 extern long a, *pA, &ra;声明的三个变量的名字也发生相应的变化，分别为“?a@@3JA”、“?pA@@3PAJA”、“?ra@@3AAJA”。上面称作 C++语言风格的标识符修饰（不同的编译器修饰格式可能不同），而 C 语言风格的标识符修饰就只是简单的在标识符前加上“_”即可（不同的编译器的 C 风格修饰一定相同）。如：extern"C" long a, *pA, &ra;就变成_a、_pA、_ra。而上面的 extern"C" void ABC(), *ABC(long), ABC(long, short);将报错，因为使用 C 风格，都只是在函数名前加一下划线，则将产生 3 个相同的符号（Symbol），错误。

为什么不能有相同的符号？为什么要改变标识符？不仅因为前面的函数重载。符号和标识符不同，符号可以由任意字符组成，它是编译器和连接器之间沟通的手段，而标识符只是在 C++语言级上提供的一种标识手段。而之所以要改变一下标识符而不直接将标识符作为符

号使用是因为编译器自己内部和连接器之间还有一些信息需要传递，这些信息就需要符号来标识，由于可能用户写的标识符正好和编译器内部自己用的符号相同而产生冲突，所以都要在程序员定义的标识符上面修改后再用作符号。既然符号是什么字符都可以，那为什么编译器不让自己内部定的符号使用标识符不能使用的字符，如前面 VC 使用的“?”，那不就行了？因为有些 C/C++编译器及连接器沟通用的符号并不是什么字符都可以，也必须是一个标识符，所以前面的 C 语言风格才统一加上“_”的前缀以区分程序员定义的符号和编译器内部的符号。即上面能使用“?”来作为符号是 VC 才这样，也许其它的编译器并不支持，但其它的编译器一定支持加了“_”前缀的标识符。这样可以联合使用多方代码，以在更大范围上实现代码重用，在《C++从零开始（十八）》中将对此详细说明。

当书写 extern void ABC(long);时，是 extern"C"还是 extern"C++"？在 VC 中，如果上句代码所在源文件的扩展名为.cpp 以表示是 C++源代码，则将解释成后者。如果是.c，则将解释成前者。不过在 VC 中还可以通过修改项目选项来改变上面的默认设置。而 extern long a;也和上面是同样的。

因此如下：

```
extern"C++" void ABC(), *ABC( long ), ABC( long, short );
int main(){ ABC(); }
```

上面第一句就告诉编译器后续代码可能要用到这个三个函数，叫编译器不要报错。假设上面程序放在一个 VC 项目下的 a.cpp 中，编译 a.cpp 将不会出现任何错误。但当连接时，编译器就会说符号“?ABC@@YAXXZ”没找到，因为这个项目只包含了一个文件，连接也就只连接相应的 a.obj 以及其他的一些必要库文件（后续文章将会说明）。连接器在它所能连接的所有对象文件（a.obj）以及库文件中查找符号“?ABC@@YAXXZ”对应的地址是什么，不过都没找到，故报错。换句话说就是 main 函数使用了在 a.cpp 以外定义的函数 void ABC();，但没找到这个函数的定义。应注意，如果写成 int main() { void (*pA) = ABC; }依旧会报错，因为 ABC 就相当于一个地址，这里又要求计算此地址的值（即使并不使用 pA），故同样报错。

为了消除上面的错误，就应该定义函数 void ABC();，既可以在 a.cpp 中，如 main 函数的后面，也可以重新生成一个.cpp 文件，加入到项目中，在那个.cpp 文件中定义函数 ABC。因此如下即可：

```
extern"C++" void ABC(), *ABC( long ), ABC( long, short );
int main(){ ABC(); } void ABC(){}
```

如果你认为自己已经了解了声明和定义的区别，并且清楚了声明的意思，那我打赌有 50%的可能性你并没有真正理解声明的含义，这里出于篇幅限制，将在《C++从零开始（十）》中说明声明的真正含义，如果你是有些 C/C++编程经验的人，到时给出的样例应该有 50%的可能性会令你大吃一惊。

调用规则

调用规则指函数的参数如何传递，返回值如何传递，以及上述的函数名标识符如何修饰。其并不属于语言级的内容，因为其表示编译器如何实现函数，而关于如何实现，各编译器都有自己的处理方式。在 VC 中，其定义了三个类型修饰符用以告知编译器如何实现函数，分别为：__cdecl、__stdcall 和__fastcall。三种各有不同的参数、函数返回值传递方式及函数名修饰方式，后面说明异常时，在说明了函数的具体实现方式后再一一解释。由于它们是类型修饰符，则可如下修饰函数：

```
void *__stdcall ABC( long ), __fastcall DE(), *( __stdcall *pAB )( long ) = &ABC;
```

```
void ( __fastcall *pDE )() = DE;
```

变量的作用域

前面定义函数 Move 时，就说 void Move(float a, float b);和 void Move(float x, float y); 是一样的，即变量名 a 和 b 在这没什么意义。这也就是说变量 a、b 的作用范围只限制在前面的 Move 的函数体（即函数定义时的复合语句）内，同样 x 和 y 的有效范围也只在后面的 Move 的函数体内。这被称作变量的作用域。

```
//////a.cpp//////
long e = 10;
void main()
{
    short a = 10;
    e++;
    {
        long e = 2;
        e++;
        a++;
    }
    e++;
}
```

上面的第一个 e 的有效范围是整个 a.cpp 文件内，而 a 的有效范围是 main 函数内，而 main 函数中的 e 的有效范围则是括着它的那对“{}”以内。即上面到最后执行完 e++;后，long e = 2;定义的变量 e 已经不在了，也就是被释放了。而 long e = 10;定义的 e 的值为 12，a 的值为 11。

也就是说“{}”可以一层层嵌套包含，没一层“{}”就产生了一个作用域，在这对“{}”中定义的变量只在这对“{}”中有效，出了这对“{}”就无效了，等同于没定义过。

为什么要这样弄？那是为了更好的体现出语义。一层“{}”就表示一个阶段，在执行这个阶段时可能会需要到和前面的阶段具有相同语义的变量，如排序。还有某些变量只在某一阶段有用，过了这个阶段就没有意义了，下面举个例子：

```
float a[10];
// 赋值数组 a
for( unsigned i = 0; i < 10; i++ )
    for( unsigned j = 0; j < 10; j++ )
        if( a[ i ] < a[ j ] )
        {
            float temp = a[ i ];
            a[ i ] = a[ j ];
            a[ j ] = temp;
        }
```

上面的 temp 被称作临时变量，其作用域就只在 if(a[i] < a[j])后的大括号内，因为那表示一个阶段，程序已经进入交换数组元素的阶段，而只有在交换元素时 temp 在有意义，用于辅助元素的交换。如果一开始就定义了 temp，则表示 temp 在数组元素寻找期间也有效，

这从语义上说是不对的，虽然一开始就定义对结果不会产生任何影响，但应不断地询问自己——这句代码能不能不要？这句代码的意义是什么？不过由于作用域的关系而可能产生性能影响，这在《C++从零开始（十）》中说明。

下篇将举例说明如何已知算法而写出 C++代码，帮助读者做到程序员的最基本的要求——给得出算法，拿得出代码

C++从零开始（八）

——C++样例一

前篇说明了函数的部分实现方式，但并没有说明函数这个语法的语义，即函数有什么用及为什么被使用。对于此，本篇及后续会零散提到一些，在《C++从零开始（十二）》中再较详细地说明。本文只是就程序员的基本要求——拿得出算法，给得出代码——给出一些样例，以说明如何从算法编写出 C++代码，并说明多个基础且重要的编程概念（即独立于编程语言而存在的概念）。

由算法得出代码

本系列一开头就说明了何谓程序，并说明由于 CPU 的世界和人们存在的客观物理世界的不兼容而导致根本不能将人编写的程序（也就是算法）翻译成 CPU 指令，但为了能够翻译，就必须让人觉得 CPU 世界中的某些东西是人以为的算法所描述的某些东西。如电脑屏幕上显示的图片，通过显示器对不同象素显示不同颜色而让人以为那是一幅图片，而电脑只知道那是一系列数字，每个数字代表了一个象素的颜色值而已。

为了实现上面的“让人觉得是”，得到算法后要做的的第一步就是找出算法中要操作的资源。前面已经说过，任何程序都是描述如何操作资源的，而 C++语言本身只能操作内存的值这一种资源，因此编程要做的第一步就是将算法中操作的东西映射成内存的值。由于内存单元的值以及内存单元地址的连续性都可以通过二进制数表示出来，因此要做的第一步就是把算法中操作的东西用数字表示出来。

上面做的第一步就相当于数学建模——用数学语言将问题表述出来，而这里只不过是用数字把被操作的资源表述出来罢了（应注意数字和数的区别，数字在 C++中是一种操作符，其有相关的类型，由于最后对它进行计算得到的还是二进制数故使用数字进行表示而不是二进制数，以增强语义）。接着第二步就是将算法中对资源的所有操作都映射成语句或函数。

用数学语言对算法进行表述时，比如将每 10 分钟到车站等车的人的数量映射为一随机变量，也就前述的第一步。随后定此随机变量服从泊松分布，也就是上面的第二步。到站等车的人的数量是被操作的资源，而给出的算法是每隔 10 分种改变这个资源，将它的值变成按给定参数的泊松函数分布的一随机值。

在 C++中，前面已经将资源映射成了数字，接着就要将对资源的操作映射成对数字的操作。C++中能操作数字的就只有操作符，也就是将算法中对资源的所有操作都映射成表达式语句。

当上面都完成了，则算法中剩下的就只有执行顺序了，而执行顺序在 C++中就是从上朝下书写，而当需要逻辑判断的介入而改变执行顺序时，就使用前面的 if 和 goto 语句（不过后者也可以通过 if 后接的语句来实现，这样可以减少 goto 语句的使用，因为 goto 的语义是

跳转而不是“所以就”），并可考虑是否能够使用循环语句以简化代码。即第三步为将执行流程用语句表示出来。

而前面第二步之所以还说可映射成函数，即可能某个操作比较复杂，还带有逻辑的意味，不能直接找到对应的操作符，这时就只好利用万能的函数操作符，对这个操作重复刚才上面的三个步骤以将此操作映射成多条语句（通过 if 等语句将逻辑信息表现出来），而将这些语句定义为一函数，供函数操作符使用以表示那个操作。

上面如果未明不要紧，后面有两个例子，都将分别说明各自是如何进行上述步骤的。

排序

给出三张卡片，上面随便写了三个整数。有三个盒子，分别标号为 1、2 和 3。将三张卡片随机放到 1、2、3 这三个盒子中，现在要求排序以使得 1、2、3 三个盒子中装的整数是由小到大的顺序。

给出一最简单的算法：称 1、2、3 盒子中放的卡片上的整数分别为第一、二、三个数，则先将第一个数和第二个数比较，如果前者大则两个盒子内的卡片交换；再将第一个和第三个比较，如果前者大则交换，这样就保证第一个数是最小的。然后将第二个数和第三个数比较，如果前者大则交换，至此排序完成。

第一步：算法中操作的资源是装在盒子中的卡片，为了将此卡片映射成数字，就注意算法中的卡片和卡片之前有什么不同。算法中区分不同卡片的唯一方法就是卡片上写的整数，因此在这里就使用一个 long 类型的数字来表示一个卡片。

算法中有三张卡片，故用三个数字来表示。前面已经说过，数字是装在内存中的，不是变量中的，变量只不过是映射地址而已。在这里需要三个 long 类型数字，可以借用定义变量时编译器自动在栈上分配的内存来记录这些数字，故可以如此定义三个变量 long a1, a2, a3; 来记录三个数字，也就相当于装三张卡片的三个盒子。

第二步：算法中的操作就是对卡片上的整数的比较和交换。前者很简单，使用逻辑操作符就可以实现（因为正好将卡片上的整数映射成变量 a1、a2 和 a3 中记录的数字）。后者是交换两个盒子中的卡片，可以先将一卡片从一盒子中取出来，放在桌子上或其他地方。然后将另一盒子中的卡片取出来放在刚才空出来的盒子。最后将先取出来的卡片放进刚空出来的盒子。前面说的“桌子上或其他地方”是用来存放取出的卡片，C++中只有内存能够存放数字，因此上面就必须再分配一临时内存来临时记录取出的数字。

第三步：操作和资源都已经映射好了，算法中有如果的就用 if 替换，由什么重复多少次的就用 for 替换，有什么重复直到怎样的就用 while 或 do while 替换，如上照着算法映射过来就完了，如下：

```
void main()
{
    long a1 = 34, a2 = 23, a3 = 12;
    if( a1 > a2 )
    {
        long temp = a1;
        a1 = a2;
        a2 = temp;
    }
    if( a1 > a3 )
```

```
    {
        long temp = a1;
        a1 = a3;
        a3 = temp;
    }
    if( a2 > a3 )
    {
        long temp = a2;
        a2 = a3;
        a3 = temp;
    }
}
```

上面就在每个 if 后面的复合语句中定义了一个临时变量 temp 以借助编译器的静态分配内存功能来提供临时存放卡片的内存。上面的元素交换并没有按照前面所说映射成函数，是因为在这里其只有三条语句且容易理解。如果要将交换操作定义为一函数，则应如下：

```
void Swap( long *p1, long *p2 )
{
    long temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}
void main()
{
    long a1 = 34, a2 = 23, a3 = 12;
    if( a1 > a2 )
        Swap( &a1, &a2 );
    if( a1 > a3 )
        Swap( &a1, &a3 );
    if( a2 > a3 )
        Swap( &a2, &a3 );
}
```

```
void Swap( long &r1, long &r2 )
    {
      long temp = r1;
       r1 = r2;
       r2 = temp;
    }
  void main()
    {
long a1 = 34, a2 = 23, a3 = 12;
     if( a1 > a2 )
          Swap( a1, a2 );
     if( a1 > a3 )
          Swap( a1, a3 );
     if( a2 > a3 )
          Swap( a2, a3 );
    }
```

先看左侧的程序。上面定义了函数来表示给定盒子之间的交换操作，注意参数类型使用了 long*，这里指针表示引用（应注意指针不仅可以表示引用，还可有其它的语义，以后会提到）。

什么是引用？注意这里不是指 C++提出的那个引用变量，引用表示一个连接关系。比如你有手机，则手机号码就是“和你通话”的引用，即只要有你的手机号码，就能够实现“和你通话”。

再比如 Windows 操作系统提供的快捷方式，其就是一个“对某文件执行操作”的引用，它可以指向某个文件，通过双击此快捷方式的图标就能够对其所指的文件进行“执行”操作（可能是用某软件打开这个文件或是直接执行此文件等），但如果删除此快捷方式却并不会删除其所指向的文件，因为它只是“对某文件执行操作”的引用。

人的名字就是对“某人进行标识”的引用，即说某人考上大学通过说那个人的名字则大家就可以知道具体是哪个人。同样，变量也是引用，它是某块内存的引用，因为其映射了地址，而内存块可以通过地址来被唯一表明其存在，不仅仅是标识。注意其和前面的名字不同，

因为任何对内存块的操作，只要知道内存块的首地址就可以了，而要和某人面对面讲话或吃饭，只知道他的名字是不够的。

应注意对某个东西的引用可以不止一个，如人就可以有多个名字，变量也都有引用变量，手机号码也可以不止一个。

注意上面引入了函数来表示交换，进而导致了盒子也就成了资源，因此必须将盒子映射成数字。而前面又将盒子里装的卡片映射成了 long 类型的数字，由于“装”这个操作，因此可以想到使用能够标识装某个代表卡片的数字的内存块来作为盒子映射的数字类型，也就是内存块的首地址，也就是 long*类型（注意不是地址类型，因为地址类型的数字并不返回记录它的内存的地址）。所以上面的函数参数类型为 long*。

下面看右侧的程序。参数类型变成 long&，和指针一样，依旧表示引用，但注意它们的不同。后者表示它是一个别名，即它是一个映射，映射的地址是记录作为参数的数字的地址，也就是说它要求调用此函数时，给出的作为参数的数字一定是有地址的数字。所谓的“有地址的数字”表示此数字是程序员创建的，不是编译器由于临时原因而生成的临时内存的地址，如 Swap(a1++, a2);就要报错。之前已经说明，因为 a1++返回的地址是编译器内部定的，就程序逻辑而言，其是不存在的，而 Swap(++a1, a2);就是正确的。Swap(1 + 3, 34);依旧要报错，因为记录 1 + 3 返回的数字的内存是编译器内部分配的，就程序逻辑上来说，它们并没有被程序员用某块内存记录起来，也就不会有内存。

一个很简单的判定规则就是调用时给的参数类型如果是地址类型的数字，则可以，否则不行。

还应注意上面是 long&类型，表示所修饰的变量不分配内存，也就是编译器要静态地将参数 r1、r2 映射的地址定下来，对于 Swap(a1, a2);就分别是 a1 和 a2 的地址，但对于 Swap(a2, a3);就变成 a2 和 a3 的地址了，这样是无法一次就将 r1、r2 映射的地址定下来，即 r1、r2 映射的地址在程序运行时是变化的，也就不能且无法编译时静态一次确定。

为了实现上面的要求，编译器实际将会在栈上分配内存，然后将地址传递到函数，再编写代码以使得好像动态绑定了 r1、r2 的地址。这实际和将参数类型定为 long*是一样的效果，即上面的 Swap(long&, long&);和 Swap(long*, long*);是一样的，只是语法书写上不同，内部是相同的，连语义都相同，均表示引用（虽然指针不仅仅只带有引用的语义）。即函数参数类型为引用类型时，依旧会分配内存以传递参数的地址，即等效于指针类型为参数。

商人过河问题

3 个商人带着 3 个仆人过河，过河的工具只有一艘小船，只能同时载两个人过河，包括划船的人。在河的任何一边，只要仆人的数量超过商人的数量，仆人就会联合起来将商人杀死并抢夺其财物，问应如何设计过河顺序才能让所有人都安全地过到河的另一边。

给出最弱却万能的算法——枚举法。坐船过河及划船回来的可能方案为一个仆人、一个商人或两个商人、两个仆人及一个商人一个仆人。

故每次从上述的五种方案中选择一个划过河去，然后检查河岸两侧的人数，看是否会发生仆人杀死商人，如果两边都不会，则再从上述的五个方案中选择一个让人把船划回来，然后再检查是否会发生仆人杀死商人，如果没有就又重新从五个方案中选一个划过河，如上重复直到所有人都过河了。

上面在选方案时除了保证商人不被杀死，还要保证此方案运行（即过河或划回来）后，两岸的人数布局从来都没有出现过，否则就形成无限循环，且必须合理，即没有负数。如果有一次的方案选择失败，则退回去重新选另一个方案再试。如果所有方案都失败，则再退回

到更上一次的方案选择。如果一直退到第一次的方案选择，并且已没有可选的方案，则说明上题无解。

上面的算法又提出了两个基本又重要的概念——层次及容器。下面先说明容器。

容器即装东西的东西，而 C++中操作的东西只有数字，因此容器就是装数字的东西，也就是内存。容器就平常的理解是能装多个东西，即能装多个数字。这很简单，使用之前的数组的概念就行了。但如果一个盒子能装很多苹果，那它一定占很大的体积，即不管装了一个苹果还是两个苹果，那盒子都要占半立方米的体积。数组就好像盒子，不管装一个元素还是两个元素，它都是 long[10]的类型而要占 40 个字节。

容器是用来装东西的，那么要取出容器中装的东西，就必须有种手段标识容器中装的东西，对于数组，这个东西就是数组的下标，如 long a[10]; a[3];就取出了第四个元素的值。由于有了标识，则还要有一种手段以表示哪些标识是有效的，如上面的 a 数组，只前面两个元素记录了数字，但是却 a[3];，得到的将是错误的值，因为只有 a[0]和 a[1]是有意义的。

因此上面的用数组作容器有很多的问题，但它非常简单，并能体现各元素之间的顺序关系，如元素被排序后的数组。但为了适应复杂算法，必须还要其他容器的支持，如链表、树、队列等。它们一般也被称做集合，都是用于管理多个元素用的，并各自给出了如何从众多的元素中快速找到给定标识所对应的元素，而且都能在各元素间形成一种关系，如后面将要提到的层次关系、前面数组的顺序关系等。关于那些容器的具体实现方式，请参考其他资料，在此不表。

上面算法中提到“两岸的人数布局从来都没有出现过”，为了实现这点，就需要将其中的资源——人数布局映射为数字，并且还要将曾经出现过的所有人数布局全部记录下来，也就是用一个容器记录下来，由于还未说明结构等概念，故在此使用数组来实现这个容器。上面还提到从已有的方案中选择一个，则可选的方案也是一个容器，同上，依旧使用一数组来实现。

层次，即关系，如希望小学的三年 2 班的 XXX、中国的四川的成都的 XXX 等，都表现出一种层次关系，这种层次关系是多个元素之间的关系，因此就可以通过找一个容器，那个容器的各元素间已经是层次关系，则这个容器就代表了一种层次关系。树这种容器就是专门对此而设计的。

上面算法中提到的“再退回到更上一次的方案选择”，也就是说第一次过河选择了一个商人一个仆人的方案，接着选择了一个商人回来的方案，此时如果选择两个仆人过河的方案将是错误的，则将重新选择过河的方案。再假设此时所有过河的方案都失败了，则只有再向后退以重新选择回来的方案，如选择一个仆人回来。对于此，由于这里只要求退回到上一次的状态，也就是人数布局及选择的方案，则可以将这些统一放在容器中，而它们各自都只依靠顺序关系，即第二次过河的方案一定在第一次过河的方案成功的前提下才可能考虑，因此使用数组这个带有顺序关系的容器即可。

第一步：上面算法的资源有两个：坐船的方案和两岸的人数布局。坐船的方案最多五种，在此使用一个 char 类型的数字来映射它，即此 8 位二进制数的前 4 位用补码格式来解释得到的数字代表仆人的数量，后 4 位则代表商人的数量。因此一个商人和一个仆人就是(1 << 4) | 1。两岸的人数布局，即两岸的商人数和仆人数，由于总共才 3+3=6 个人，这都可以使用 char 类型的数字就能映射，但只能映射一个人数，而两岸的人数实际共有 4 个（左右两岸的商人数和仆人数），则这里使用一个 char[4]来实现（实际最好是使用结构来映射而不是 char[4]，下篇说明）。如 char a[4];表示一人数布局，则 a[0]表示河岸左侧的商人数，a[1]表示左侧的仆人数，a[2]表示河岸右侧的商人数，a[3]表示右侧的仆人数。

注意前面说的容器，在此为了装可选的坐船方案故应有一容器，使用数组，如 char sln[5];。在此还需要记录已用的坐船方案，由于数组的元素具备顺序关系，所以不用再生成一容器，

直接使用一 char 数字记录一下标，当此数字为 3 时，表示 sln[0]、sln[1]和 sln[2]都已经用过且都失败了，当前可用的为 sln[3]和 sln[4]。同样，为了装已成功的坐船方案作用后的人数布局及当时所选的方案，就需要两个容器，在此使用数组（实际应该链表）char oldLayout[4][200], cur[200];。oldLayout 就是记录已成功的方案的容器，其大小为 200，表示假定在 200 次内，一定就已经得出结果了，否则就会因为超出数组上限而可能发生内存访问违规，而为什么是可能在《C++从零开始（十五）》中说明。

前面说过数组这种容器无法确定里面的有效元素，必须依靠外界来确定，对此，使用一 unsigned char curSln;来记录 oldLayout 和 cur 中的有效元素的个数。规定当 curSln 为 3 时，表示 oldLayout[0～3][0]、oldLayout[0～3][1]和 oldLayout[0～3][2]都有效，同样 cur[0]、cur[1]和 cur[2]都有效，而之后的如 cur[3]等都无效。

第二步：操作有：执行过河方案、执行回来方案、检查方案是否成功、退回到上一次方案选择、是否所有人都过河、判断人数布局是否相同。如下：

前两个操作：将当前的左岸人数减去相应的方案定的人数，而右岸则加上人数。要表现当前左岸人数，可以用 oldLayout[0][curSln]和 oldLayout[1][curSln]表示，而相应方案的人数则为(sln[cur[curSln]] & 0xF0) >> 4 和 sln[cur[curSln]] & 0xF。由于这两个操作非常类似，只是一个是加则另一个就是减，故将其定义为函数，则为了在函数中能操作 oldLayout、curSln 等变量，就需要将这些变量定义为全局变量。

检查是否成功：即看是否

oldLayout[1][curSln] > oldLayout[0][curSln] && oldLayout[0][curSln]以及是否

oldLayout[3][curSln] > oldLayout[2][curSln] && oldLayout[2][curSln]

并且保证各自不为负数以及没有和原来的方案冲突。检查是否和原有方案相同就是枚举所有原由方案以和当前方案比较，由于比较复杂，在此将其定义为函数，通过返回 bool 类型来表示是否冲突。

退回上一次方案或到下一个方案的选择，只用 curSln--或 curSln++即可。而是否所有人都过河，则只用 oldLayout[0～1][curSln]都为 0 而 oldLayout[2～3][curSln]都为 3。而判断人数布局是否相同，则只用相应各元素是否相等即可。

第三步：下面剩下的就没什么东西了，只需要按照算法说的顺序，将刚才的各操作拼凑起来，并注意“重复直到所有人都过河了”转成 do while 即可。如下：

```
#include <stdio.h>
// 分别表示一个商人、一个仆人、两个商人、两个仆人、一个商人一个仆人
char sln[5] = { ( 1 << 4 ), 1, ( 2 << 4 ), 2, ( 1 << 4 ) | 1 };
unsigned char curSln = 1;
char oldLayout[4][200], cur[200];

void DoSolution( char b )
{
    unsigned long oldSln = curSln - 1;   // 临时变量，出于效率
    oldLayout[0][ curSln ] =
        oldLayout[0][ oldSln ] - b * ( ( sln[ cur[ curSln ] ] & 0xF0 ) >> 4 );
    oldLayout[1][ curSln ] =
        oldLayout[1][ oldSln ] - b * ( sln[ cur[ curSln ] ] & 0xF );
    oldLayout[2][ curSln ] =
        oldLayout[2][ oldSln ] + b * ( ( sln[ cur[ curSln ] ] & 0xF0 ) >> 4 );
```

```
        oldLayout[3][ curSln ] =
            oldLayout[3][ oldSln ] + b * ( sln[ cur[ curSln ] ] & 0xF );
}
bool BeRepeated( char b )
{
        for( unsigned long i = 0; i < curSln; i++ )
            if( oldLayout[0][ curSln ] == oldLayout[0][ i ] &&
                oldLayout[1][ curSln ] == oldLayout[1][ i ] &&
                oldLayout[2][ curSln ] == oldLayout[2][ i ] &&
                oldLayout[3][ curSln ] == oldLayout[3][ i ] &&
                ( ( i & 1 ) ? 1 : -1 ) == b )   // 保证过河后的方案之间比较，回来后的方案之间比
较
                                                // i&1 等效于 i%2，i&7 等效于 i%8，i&63
等效于 i%64
                return true;
        return false;
}
void main()
{
        char b = 1;
        oldLayout[0][0] = oldLayout[1][0] = 3;
        cur[0] = oldLayout[2][0] = oldLayout[3][0] = 0;
        for( unsigned char i = 0; i < 200; i++ )   // 初始化每次选择方案时的初始化方案为 sln[0]
            cur[ i ] = 0;                          // 由于 cur 是全局变量，在 VC 中，其已经
被赋值为 0
                                                   // 原因涉及到数据节，在此不表
        do
        {
            DoSolution( b );
            if( ( oldLayout[1][ curSln ] > oldLayout[0][ curSln ] && oldLayout[0][ curSln ] ) ||
                ( oldLayout[3][ curSln ] > oldLayout[2][ curSln ] && oldLayout[2][ curSln ] ) ||
                oldLayout[0][ curSln ] < 0 || oldLayout[1][ curSln ] < 0 ||
                oldLayout[2][ curSln ] < 0 || oldLayout[3][ curSln ] < 0 ||
                BeRepeated( b ) )
            {
            // 重新选择本次的方案
P:
                cur[ curSln ]++;
                if( cur[ curSln ] > 4 )
                {
                    b = -b;
                    cur[ curSln ] = 0;
                    curSln--;
                    if( !curSln )
```

```
                    break;    // 此题无解
                goto P;    // 重新检查以保证 cur[ curSln ]的有效性
            }
            continue;
        }
        b = -b;
        curSln++;
    }
    while( !( oldLayout[0][ curSln - 1 ] == 0 && oldLayout[1][ curSln - 1 ] == 0 &&
              oldLayout[2][ curSln - 1 ] == 3 && oldLayout[3][ curSln - 1 ] == 3 ) );

    for( i = 0; i < curSln; i++ )
        printf( "%d    %d\t %d    %d\n",
                oldLayout[0][ i ],
                oldLayout[1][ i ],
                oldLayout[2][ i ],
                oldLayout[3][ i ] );
}
```

上面数组 sln[5]的初始化方式下篇介绍。上面的预编译指令#include 将在《C++从零开始（十）》中说明，这里可以不用管它。上面使用的函数 printf 的用法，请参考其它资料，这里它只是将变量的值输出在屏幕上而已。

前面说此法是枚举法，其基本上属于万能方法，依靠 CPU 的计算能力来实现，一般情况下程序员第一时间就会想到这样的算法。它的缺点就是效率极其低下，大量的 CPU 资源都浪费在无谓的计算上，因此也是产生瓶颈的大多数原因。由于它的万能，编程时很容易将思维陷在其中，如求和 1 到 100，一般就写成如下：

```
for( unsigned long i = 1, s = 0; i <= 100; i++ ) s += i;
```

但更应该注意到还可 unsigned long s = (1 + 100) * 100 / 2;，不要被枚举的万能占据了头脑。

上面的人数布局映射成一结构是最好的，映射成 char[4]所表现的语义不够强，代码可读性较差。下篇说明结构，并展示类型的意义——如何解释内存的值。

C++从零开始（九）

——何谓结构

前篇已经说明编程时，拿到算法后该干的第一件事就是把资源映射成数字，而前面也说过“类型就是人为制订的如何解释内存中的二进制数的协议”，也就是说一个数字对应着一块内存（可能 4 字节，也可能 20 字节），而这个数字的类型则是附加信息，以告诉编译器当发现有对那块内存的操作语句（即某种操作符）时，要如何编写机器指令以实现那个操作。比如两个 char 类型的数字进行加法操作符操作，编译器编译出来的机器指令就和两个 long 类型的数字进行加法操作的不一样，也就是所谓的“如何解释内存中的二进制数的协议”。由于解释协议的不同，导致每个类型必须有一个唯一的标识符以示区别，这正好可以提供强烈的语义。

typedef

提供语义就是要尽可能地在代码上体现出这句或这段代码在人类世界中的意义，比如前篇定义的过河方案，使用一 char 类型来表示，然后定义了一数组 char sln[5]以期从变量名上体现出这是方案。但很明显，看代码的人不一定就能看出 sln 是 solution 的缩写并进而了解这个变量的意义。但更重要的是这里有点本末倒置，就好像这个东西是红苹果，然后知道这个东西是苹果，但它也可能是玩具、CD 或其它，即需要体现的语义是应该由类型来体现的，而不是变量名。即 char 无法体现需要的语义。

对此，C++提供了很有意义的一个语句——类型定义语句。其格式为 typedef <源类型名> <标识符>;。其中的<源类型名>表示已存在的类型名称，如 char、unsigned long 等。而<标识符>就是程序员随便起的一个名字，符合标识符规则，用以体现语义。对于上面的过河方案，则可以如下：

typedef char Solution; Solution sln[5];

上面其实是给类型 char 起了一个别名 Solution，然后使用 Solution 来定义 sln 以更好地体现语义来增加代码的可读性。而前篇将两岸的人数分布映射成 char[4]，为了增强语义，则可以如下：

typedef char PersonLayout[4]; PersonLayout oldLayout[200];

注意上面是 typedef char PersonLayout[4];而不是 typedef char[4] PersonLayout;，因为数组修饰符“[]”是接在被定义或被声明的标识符的后面的，而指针修饰符“*”是接在前面的，所以可以 typedef char *ABC[4];但不能 typedef char [4]ABC*;，因为类型修饰符在定义或声明语句中是有固定位置的。

上面就比 char oldLayout[200][4];有更好的语义体现，不过由于为了体现语义而将类型名或变量名增长，是否会降低编程速度？如果编多了，将会发现编程的大量时间不是花在敲代码上，而是调试上。因此不要忌讳书写长的变量名或类型名，比如在 Win32 的 Security SDK 中，就提供了下面的一个函数名：

BOOL ConvertSecurityDescriptorToStringSecurityDescriptor(…);

很明显，此函数用于将安全描述符这种类型转换成文字形式以方便人们查看安全描述符中的信息。

应注意 typedef 不仅仅只是给类型起了个别名，还创建了一个原类型。当书写 char* a, b; 时，a 的类型为 char*，b 为 char，而不是想象的 char*。因为“*”在这里是类型修饰符，其是独立于声明或定义的标识符的，否则对于 char a[4], b;，难道说 b 是 char[4]？那严重不符合人们的习惯。上面的 char 就被称作原类型。为了让 char*为原类型，则可以：typedef char *PCHAR; PCHAR a, b, *c[4];。其中的 a 和 b 都是 char*，而 c 是 char**[4]，所以这样也就没有问题：char **pA = &a;。

结构

再次考虑前篇为什么要将人数布局映射成 char[4]，因为一个人数可以用一个 char 就表示，而人数布局有四个人数，所以使用 char[4]。即使用 char[4]是希望只定义一个变量就代表了一个人数分布，编译器就一次性在栈上分配 4 个字节的空间，并且每个字节都各自代表一个人数。所以为了表现河岸左侧的商人数，就必须写 a[0]，而左侧的仆人数就必须 a[1]。

坏处很明显，从 a[0]无法看出它表示的是左岸的商人数，即这个映射意义（左岸的商人数映射为内存块中第一个字节的内容以补码格式解释）无法从代码上体现出来，降低了代码的可读性。

上面其实是对内存布局的需要，即内存块中的各字节二进制数如何解释。为此，C++提出了类型定义符“{}”。它就是一对大括号，专用在定义或声明语句中，以定义出一种类型，称作自定义类型。即 C++原始缺省提供的类型不能满足要求时，可自定义内存布局。其格式为：<类型关键字> <名字> { <声明语句> …}。<类型关键字>只有三个：struct、class 和 union。而所谓的结构就是在<类型关键字>为 struct 时用类型定义符定义的原类型，它的类型名为<名字>，其表示后面大括号中写的多条声明语句，所定义的变量之间是串行关系（后面说明），如下：

struct ABC { long a, *b; double c[2], d; } a, *b = &a;

上面是一个变量定义语句，对于 a，表示要求编译器在栈上分配一块 4+4+8*2+8=32 字节长的连续内存块，然后将首地址和 a 绑定，其类型为结构型的自定义类型（简称结构）ABC。对于 b，要求编译器分配一块 4 字节长的内存块，将首地址和 b 绑定，其类型为结构 ABC 的指针。

上面定义变量 a 和 b 时，在定义语句中通过书写类型定义符“{}”定义了结构 ABC，则以后就可以如下使用类型名 ABC 来定义变量，而无需每次都那样，即：

ABC &c = a, d[2];

现在来具体看清上面的意思。首先，前面语句定义了 6 个映射元素，其中 a 和 b 分别映射着两个内存地址。而大括号中的四个变量声明也生成了四个变量，各自的名字分别为 ABC::a、ABC::b、ABC::c、ABC::d；各自映射的是 0、4、8 和 24；各自的类型分别为 long ABC::、long* ABC::、double (ABC::) [2]、double ABC::，表示是偏移。其中的 ABC::表示一种层次关系，表示“ABC 的”，即 ABC::a 表示结构 ABC 中定义的变量 a。应注意，由于 C++是强类型语言，它将 ABC::也定义为类型修饰符，进而导致出现 long* ABC::这样的类型，表示它所修饰的标识符是自定义类型 ABC 的成员，称作偏移类型，而这种类型的数字不能被单独使用（后面说明）。由于这里出现的类型不是函数，故其映射的不是内存的地址，而是一偏移值（下篇说明）。与之前不同了，类型为偏移类型的（即如上的类型）数字是不能计算的，因为偏移是一相对概念，没有给出基准是无法产生任何意义的，即不能：ABC::a; ABC::c[1];。其中后者更是严重的错误，因为数组操作符“[]”要求前面接的是数组或指针类型，而这里的 ABC::c 是 double 的数组类型的结构 ABC 中的偏移，并不是数组类型。

注意上面的偏移 0、4、8、24 正好等同于 a、b、c、d 顺次安放在内存中所形成的偏移，这也正是 struct 这个关键字的修饰作用，也就是前面所谓的各定义的变量之间是串行关系。

为什么要给偏移制订映射？即为什么将 a 映射成偏移 0 字节，b 映射成偏移 4 字节？因为可以给偏移添加语义。前面的“左岸的商人数映射为内存块中第一个字节的内容以补码格式解释”其实就是给定内存块的首地址偏移 0 字节。而现在给出一个标识符和其绑定，则可以将这个标识符起名为 LeftTrader 来表现其语义。

由于上面定义的变量都是偏移类型，根本没有分配内存以和它们建立映射，它们也就很正常地不能是引用类型，即 struct AB{ long a, &b; };将是错误的。还应注意上面的类型 double (ABC::)[2]，类型修饰符“ABC::”被用括号括起来，因为按照从左到右来解读类型操作符的规则，“ABC::”实际应该最后被解读，但其必须放在标识符的左边，就和指针修饰符“*”一样，所以必须使用括号将其括住，以表示其最后才起修饰作用。故也就有：double (*ABCD::)[2]、double (**ABCD::)[2]，各如下定义：

struct ABCD { double (*pD)[2]; double (**ppD)[2]; };

但应注意，“ABCD::”并不能直接使用，即 double (*ABCD:: pD)[2];是错误的，要定义偏

移类型的变量，必须通过类型定义符“{}”来自定义类型。还应注意 C++也允许这样的类型 double (*ABCD::*)[2]，其被称作成员指针，即类型为 double (*ABCD::)[2]的指针，也就是可以如下：

double (**ABCD::*pPPD)[2] = &ABC::ppD, (**ABCD::**ppPPD)[2] = &pPPD;

上面很奇怪，回想什么叫指针类型。只有地址类型的数字才能有指针类型，表示不计算那个地址类型的数字，而直接返回其二进制表示，也就是地址。对于变量，地址就是它映射的数字，而指针就表示直接返回其映射的数字，因此&ABCD::ppD 返回的数字其实就是偏移值，也就是 4。

为了应用上面的偏移类型，C++给出了一对操作符——成员操作符“.”和“->”。前者两边接数字，左边接自定义类型的地址类型的数字，而右边接相应自定义类型的偏移类型的数字，返回偏移类型中给出的类型的地址类型的数字，比如：a.ABC::d;。左边的 a 的类型是 ABC，右边的 ABC::d 的类型是 double ABC::，则 a.ABC::d 返回的数字是 double 的地址类型的数字，因此可以这样：a.ABC::d = 10.0;。假设 a 对应的地址是 3000，则 a.ABC::d 返回的地址为 3000+24=3024，类型为 double，这也就是为什么 ABC::d 被叫做偏移类型。由于“.”左边接的结构类型应和右边的结构类型相同，因此上面的 ABC::可以省略，即 a.d = 10.0;。而对于“->”，和“.”一样，只不过左边接的数字是指针类型罢了，即 b->c[1] = 10.0;。应注意 b->c[1]实际是(b->c)[1]，而不是 b->(c[1])，因为后者是对偏移类型运用“[]”，是错误的。

还应注意由于右边接偏移类型的数字，所以可以如下：

double (ABC::*pA)[2] = &ABC::c, (ABC::**ppA)[2] = &pA;

(b->**ppA)[1] = 10.0; (a.*pA)[0] = 1.0;

上面之所以要加括号是因为数组操作符“[]”的优先级较“*”高，但为什么不是 b->(**ppA)[1]而是(b->**ppA)[1]？前者是错误的。应注意括号操作符“()”并不是改变计算优先级，而是它也作为一个操作符，其优先级被定得很高罢了，而它的计算就是计算括号内的数字。之前也说明了偏移类型是不能计算的，即 ABC::c;将错误，而刚才的前者由于“()”的加入而导致要求计算偏移类型的数字，故编译器将报错。

还应该注意，成员指针是偏移类型的指针，即装的是偏移，则可以程序运行时期得到偏移，而前面通过 ABC::a 这种形式得到的是编译时期，由编译器帮忙映射的偏移，只能实现静态的偏移，而利用成员指针则可以实现动态的偏移。不过其实只需将成员定义成数组或指针类型照样可以实现动态偏移，不过就和前篇没有使用结构照样映射了人数布局一样，欠缺语义而代码可读性较低。成员指针的提出，通过变量名，就可以表现出丰富的语义，以增强代码的可读性。现在，可以将最开始说的人数布局定义如下：

struct PersonLayout{ char LeftTrader, LeftServitor, RightTrader, RightServitor; };

PersonLayout oldLayout[200], b;

因此，为了表示 b 这个人数分布中的左侧商人数，只需 b.LeftTrader;，右侧的仆人数，只需 b.RightServitor;。因为 PersonLayout::LeftTrader 记录了偏移值和偏移后应以什么样的类型来解释内存，故上面就可以实现原来的 b[0]和 b[3]。很明显，前者的可读性远远地高于后者，因为前者通过变量名（b 和 PersonLayout::LeftTrader）和成员操作符“.”表现了大量的语义——b 的左边的商人数。

注意 PersonLayout::LeftTrader 被称作结构 PersonLayout 的成员变量，而前面的 ABC::d 则是 ABC 的成员变量，这种叫法说明结构定义了一种层次关系，也才有所谓的成员操作符。既然有成员变量，那也有成员函数，这在下篇介绍。

前篇在映射过河方案时将其映射为 char，其中的前 4 位表示仆人数，后 4 位表示商人数。对于这种使用长度小于 1 个字节的用法，C++专门提供了一种语法以支持这种情况，如下：

struct Solution { ServitorCount : 4; unsigned TraderCount : 4; } sln[5];

由于是基于二进制数的位（Bit）来进行操作，只准使用两种类型来表示数字，原码解释数字或补码解释数字。对于上面，ServitorCount 就是补码解释，而 TraderCount 就是原码解释，各自的长度都为 4 位，而此时 Solution::ServitorCount 中依旧记录的是偏移，不过不再以字节为单位，而是位为单位。并且由于其没有类型，故也就没有成员指针了。即前篇的 (sln[cur[curSln]] & 0xF0) >> 4 等效于 sln[cur[curSln]].TraderCount，而 sln[cur[curSln]] & 0xF0 等效于 sln[cur[curSln]].ServitorCount，较之前具有了更好的可读性。

应该注意，由于 struct AB { long a, b; };也是一条语句，并且是一条声明语句（因为不生成代码），但就其意义上来看，更通常的叫法把它称为定义语句，表示是类型定义语句，但按照不生成代码的规则来判断，其依旧是声明语句，并进而可以放在类型定义符“{}”中，即：

struct ABC{ struct DB { long a, *b[2]; }; long c; DB a; };

上面的结构 DB 就定义在结构 ABC 的声明语句中，则上面就定义了四个变量，类型均为偏移类型，变量名依次为：ABC::DB::a、ABC::DB::b、ABC::c、ABC::a；类型依次为 long ABC::DB::、long* (ABC::DB::)[2]、long ABC::、ABC::DB；映射的数值依次为 0、4、0、4。这里称结构 DB 嵌套在结构 ABC 中，其体现出一种层次关系，实际中这经常被使用以表现特定的语义。欲用结构 DB 定义一个变量，则 ABC::DB a;。同样也就有 long* (ABC::DB::*pB)[2] = &ABC::DB::b; ABC c; c.a.a = 10; *(c.a.b[0]) = 20;。应注意 ABC::DB::表示“ABC 的 DB 的”而不是“DB 的 ABC 的”，因为这里是重复的类型修饰符，是从右到左进行修饰的。

前面在定义结构时，都指明了一个类型名，如前面的 ABC、ABCD 等，但应该注意类型名不是必须的，即可以 struct { long a; double b; } a; a.a = 10; a.b = 34.32;。这里就定义了一个变量，其类型是一结构类型，不过这个结构类型没有标识符和其关联，以至于无法对其运用类型匹配等比较，如下：

struct { long a; double b; } a, &b = a, *c = &a; struct { long a; double b; } *d = &a;

上面的 a、b、c 都没有问题，因为使用同一个类型来定义的，即使这个类型没有标识符和其映射，但 d 将会报错，即使后写的结构的定义式和前面的一模一样，但仍然不是同一个，只是长得像罢了。那这有什么用？后面说明。

最后还应该注意，当在复合语句中书写前面的声明语句以定义结构时，之前所说的变量作用域也同样适用，即在某复合语句中定义的结构，出了这个复合语句，它就被删除，等于没定义。如下：

```
void ABC()
{
    struct AB { long a, b; };
    AB d; d.b = 10;
}
void main()
{
    {
        struct AB{ long a, b, e; };
        AB c; c.e = 23;
    }
    AB a;   // 将报错，说 AB 未定义，但其他没有任何问题
}
```

初始化

初始化就是之前在定义变量的同时，就给在栈上分配的内存赋值，如：long a = 10;。当定义的变量的类型有表示多个元素时，如数组类型、上面的结构类型时，就需要给出多个数字。对此，C++专门给出了一种语法，使用一对大括号将欲赋的值括起来后，整体作为一个数字赋给数组或结构，如下：

```
struct ABC { long a, b; float c, d[3]; };
ABC a = { 1, 2, 43.4f, { 213.0f, 3.4f, 12.4f } };
```

上面就给出了为变量 a 初始化的语法，大括号将各元素括起来，而各元素之间用“,”隔开。应注意 ABC::d 是数组类型，其对应的初始化用的数字也必须用大括号括起来，因此出现上面的嵌套大括号。现在应该了解到“{}”只是用来构造一个具有多个元素的数字而已，因此也可以有 long a = { 34 };，这里“{}”就等同于没有。还应注意，C++同意给出的大括号中的数字个数少于相应自定义类型或数组的元素个数，即：ABC a = { 1, 2, 34 }, b = { 23, { 34 }, 65, { 23, 43 } }, c = { 1, 2, { 3, { 4, 5, 6 } } };

上面的 a.d[0]、a.d[1]、a.d[2]都为 0，而只有 b.d[2]才为 0，但 c 将会报错，因为嵌套的第一个大括号将{ 4, 5, 6 }也括了起来，表示 c.c 将被一个具有两个元素的数字赋值，但 c.c 的类型是 float，只对应一个元素，编译器将说初始化项目过多。而之前的 a 和 b 未赋值的元素都将被赋值为 0，但应注意并不是数值上的 0，而是简单地将未赋值的内存的值用 0 填充，再通过那些补码原码之类的格式解释成数值后恰好为 0 而已，并不是赋值 0 这个数字。

应注意，C++同意这样的语法：long a[] = { 34, 34, 23 };。这里在定义 a 时并没有给出元素个数，而是由编译器检查赋值用的大括号包的元素个数，由其来决定数组的个数，因此上面的 a 的类型为 long[3]。当数组时，如：long a[3][2] = { { 1, 2 }, { 3, 4 }, { 5, 6 } };。因为每个元素又是需要多个元素的数字，就和前面的 ABC::d 一样。再回想类型修饰符的修饰顺序，是从左到右，但当是重复类型修饰符时，就倒过来从右到左，因此上面就应该是三个 long[2]，而不是两个 long[3]，因此这样将错误：long a[3][2] = { { 1, 2, 3 }, { 4, 5, 6 } };。

还应注意，C++不止提供了上面的“{}”这一种初始化方式，对于字符串，其专门提供如：char a[] = "ABC";。这里 a 的类型就为 char[4]，因为字符串"ABC"需要占 4 个字节的内存空间。除了这两种初始化方式外，C++还提供了一种函数式的初始化函数，下篇介绍。

类型的运用

```
char a = -34; unsigned char b = ( unsigned char )a;
```

上面的 b 等于 222，将-34 按照补码格式写成二进制数 11011110，然后将这个二进制数用原码格式解释，得数值 222。继续：

```
float a = 5.6f; unsigned long b = ( unsigned long )a;
```

这回 b 等于 5。为什么？不是应该将 5.6 按照 IEEE 的 real*4 的格式写成二进制数 0X40B33333（这里用十六进制表示），然后将这个二进制数用原码格式解释而得数值 1085485875 吗？因为类型转换是语义上的类型转换，而不是类型变换。

两个类型是否能够转换，要视编译器是否定义了这两个类型之间的转换规则。如 char 和 unsigned char，之所以前面那样转换是因为编译器把 char 转 unsigned char 定义成了那样，同样 float 转 unsigned long 被编译器定义成了取整而不是四舍五入。

为什么要有类型转换？有什么意义？的确，像上面那样的转换，毫无意义，仅仅只是为了满足语法的严密性而已，不过由于 C++定义了指针类型的转换，而且定义得非常地好，以

至于有非常重要的意义。

char a = -34; unsigned char b = *(unsigned char*)(&a);

上面的结果和之前的一样，b 为 222，不过是通过将 char*转成 unsigned char*，然后再用 unsigned char 来解释对应的内存而得到 222，而不是按照编译器的规定来转换的，即使结果一样。因此：

float a = 5.6f; unsigned long b = *(unsigned long*)(&a);

上面的 b 为 1085485875，也就是之前以为的结果。这里将 a 的地址所对应的内存用 unsigned long 定义的规则来解释，得到的结果放在 b 中，这体现了类型就是如何解释内存中的内容。上面之所以能实现，是因为 C++规定所有的指针类型之间的转换，数字的数值没有变化，只有类型变化（但由于类的继承关系也是可能会改变，下篇说明），因此上面才说 b 的值是用 unsigned long 来解释 a 对应的内存的内容所得的结果。因此，前篇在比较 oldLayout[curSln][0～3]和 oldLayout[i][0～3]时写了四个“==”以比较了四次 char 的数字，由于这四个 char 数字是连续存放的，因此也可如下只比较一次 long 数字即可，将节约多余的三次比较时间。

(long)&oldLayout[curSln] == *(long*)&oldLayout[i]

上面只是一种优化手段而已，对于语义还是没有多大意义，不过由于有了自定义类型，因此：

struct AB { long a1; long a2; }; struct ABC { char a, b; short c; long d; };

AB a = { 53213, 32542 }; ABC *pA = (ABC*)&a;

char aa = pA->a, bb = pA->b, cc = pA->c; long dd = pA->d;

pA->a = 1; pA->b = 2; pA->c = 3; pA->d = 4;

long aa1 = a.a1, aa2 = a.a2;

上面执行后，aa、bb、cc、dd 的值依次为-35、-49、0、32542，而 aa1 和 aa2 的值分别为 197121 和 4。相信只要稍微想下就应该能理解为什么没有修改 a.a1 和 a.a2，结果它们的值却变了，因为变量只不过是个映射而已，而前面就是利用指针 pA 以结构 ABC 来解释并操作 a 所对应的内存的内容。

因此，利用自定义类型和指针转换，就可以实现以什么样的规则来看待某块内存的内容。有什么用？传递给某函数一块内存的引用（利用指针类型或引用类型），此函数还另有一个参数，比如是 long 类型。当此 long 类型的参数为 1 时，表示传过去的是一张定单；为 2 时，表示传过去的是一张发货单；为 3 时表示是一张收款单。如果再配上下面说明的枚举类型，则可以编写出语义非常完善的代码。

应注意由于指针是可以随便转换的，也就有如下的代码，实际并没什么意义，在这只为加深对成员指针的理解：

long AB::*p = (long AB::*)(&ABC::b); a.a1 = a.a2 = 0; a.*p = 0XAB1234CD;

上面执行后，a.a1 为 305450240,a.a2 为 171，转成十六进制分别为 0X1234CD00 和 0X000000AB。

枚举

上面欲说明 1 时为定单，2 时为发货单而 3 时为收款单，则可以利用 switch 或 if 语句来进行判断，但是语句从代码上将看见类似 type == 1 或 type == 2 之类，无法表现出语义。C++专门为此提供了枚举类型。

枚举类型的格式和前面的自定义类型很像，但意义完全不同，如下：

```
enum AB { LEFT, RIGHT = 2, UP = 4, DOWN = 3 }; AB a = LEFT;
switch( a )
{
    case LEFT:;   // 做与左相应的事
    case UP:;     // 做与上相应的事
}
```

枚举也要用“{}”括住一些标识符，不过这些标识符即不映射内存地址也不映射偏移，而是映射整数，而为什么是整数，那是因为没有映射浮点数的必要，后面说明。上面的 RIGHT 就等同于 2，注意是等同于 2，相当于给 2 起了个名字，因此可以 long b = LEFT; double c = UP; char d = RIGHT;。但注意上面的变量 a，它的类型为 AB，即枚举类型，其解释规则等同于 int，即编译成在 16 位操作系统上运行时，长度为 2 个字节，编译成在 32 位操作系统上运行时为 4 个字节，但和 int 是属于不同的类型，而前面的赋值操作之所以能没有问题，可以认为编译器会将枚举类型隐式转换成 int 类型，进而上面没有错误。但倒过来就不行了，因为变量 a 的类型是 AB，则它的值必须是上面列出的四个标识符中的一个，而 a = b;则由于 b 为 long 类型，如果为 10，那么将无法映射上面的四个标识符中的一个，所以不行。

注意上面的 LEFT 没有写“=”，此时将会从其前面的一个标识符的值自增一，由于它是第一个，而 C++规定为 0，故 LEFT 的值为 0。还应注意上面映射的数字可以重复，即：

```
enum AB { LEFT, RIGHT, UP = 5, DOWN, TOP = 5, BOTTOM };
```

上面的各标识符依次映射的数值为 0、1、5、6、5、6。因此，最开始说的问题就可以如下处理：

```
enum OperationType { ORDER = 1, INVOICE, CHECKOUT };
```

而那个参数的类型就可以为 OperationType，这样所表现的语义就远远地超出原来的代码，可读性高了许多。因此，当将某些人类世界的概念映射成数字时，发现它们的区别不表现在数字上，比如吃饭、睡觉、玩表示一个人的状态，现在为了映射人这个概念为数字，也需要将人的状态这个概念映射成数字，但很明显地没有什么方便的映射规则。这时就强行说 1 代表吃饭，2 代表睡觉，3 代表玩，此时就可以使用将 1、2、3 定义成枚举以表现语义，这也就是为什么枚举只定义为整数，因为没有定义成浮点数的必要性。

联合

前面说过类型定义符的前面可以接 struct、class 和 union，当接 union 时就表示是联合型自定义类型（简称联合），它和 struct 的区别就是后者是串行分布来定义成员变量，而前者是并行分布。如下：

```
union AB { long a1, a2, a3; float b1, b2, b3; }; AB a;
```

变量 a 的长度为 4 个字节，而不是想象的 6*4=24 个字节，而联合 AB 中定义的 6 个变量映射的偏移都为 0。因此 a.a1 = 10;执行后，a.a1、a.a2、a.a3 的值都为 10，而 a.b1 的值为多少，就用 IEEE 的 real*4 格式来解释相应内存的内容，该多少是多少。

也就是说，最开始的利用指针来解释不同内存的内容，现在可以利用联合就完成了，因此上面的代码搬到下面，变为：

```
union AB
{
    struct { long a1; long a2; };
    struct { char a, b; short c; long d; };
```

```
};
AB a = { 53213, 32542 };
char aa = a.a, bb = a.b, cc = a.c; long dd = a.d;
a.a = 1; a.b = 2; a.c = 3; a.d = 4;
long aa1 = a.a1, aa2 = a.a2;
```

结果不变，但代码要简单，只用定义一个自定义类型了，而且没有指针变量的运用，代码的语义变得明显多了。

注意上面定义联合 AB 时在其中又定义了两个结构，但都没有赋名字，这是 C++的特殊用法。当在类型定义符的中间使用类型定义符时，如果没有给类型定义符定义的类型绑定标识符，则依旧定义那些偏移类型的变量，不过这些变量就变成上层自定义类型的成员变量，因此这时“{}”等同于没有，唯一的意义就是通过前面的 struct 或 class 或 union 来指明变量的分布方式。因此可以如下：

```
struct AB
{
    struct { long a1, a2; };
    char a, b;
    union { float b1; double b2; struct { long b3; float b4; char b5; }; };
    short c;
};
```

上面的自定义类型 AB 的成员变量就有 a1、a2、a、b、b1、b2、b3、b4、b5、c，各自对应的偏移值依次为 0、4、8、9、10、10、10、14、18、19，类型 AB 的总长度为 21 字节。某类型的长度表示如果用这个类型定义了一个变量，则编译器应该在栈上分配多大的连续空间，C++为此专门提供了一个操作符 sizeof，其右侧接数字或类型名，当接数字时，就返回那个数字的类型需要占的内存空间的大小，而接类型名时，就返回那个类型名所标识的类型需要占的内存空间的大小。

因此 long a = sizeof(AB); AB d; long b = sizeof d;执行后，a 和 b 的值都为 40。怎么是 40？不应该为 21 吗？而之前的各成员变量对应的偏移也依次实际为 0、4、8、9、16、16、16、20、24、32。为什么？这就是所谓的数据对齐。

CPU 有某些指令，需要处理多个数据，则各数据间的间隔必须是 4 字节或 8 字节或 16 字节（视不同的指令而有不同的间隔），这被称作数据对齐。当各个数据间的间隔不符合要求时，CPU 就必须做附加的工作以对齐数据，效率将下降。并且 CPU 并不直接从内存中读取东西，而要经一个高速缓冲（CPU 内建的一个存取速度比内存更快的硬件）缓冲一下，而此缓冲的大小肯定是 2 的次方，但又比较小，因此自定义类型的大小最好能是 2 的次方的倍数，以便高效率的利用高速缓冲。

在自定义类型时，一个成员变量的偏移值一定是它所属的类型的长度的倍数，即上面的 a 和 b 的偏移必须是 1 的倍数，而 c 的偏移必须是 2 的倍数，b1 的偏移必须是 4 的倍数。但 b2 的偏移必须是 8 的倍数，而 b1 和 b2 由于前面的 union 而导致是并行布局，因此 b1 的偏移必须和 b2 及 b3 的相同，因此上面的 b1、b2、b3 的偏移变成了 8 的倍数 16，而不是想象的 10。

而一个自定义类型的长度必须是其成员变量中长度最长的那个成员变量的长度的倍数，因此 struct { long b3; float b4; char b5; };的长度是 4 的倍数，也就是 12。而上面的无名联合的成员变量中，只有 double b2;的长度最长，为 8 个字节，所以它的长度为 16，并进而导致 c 的偏移为 b1 的偏移加 16，故为 32。由于结构 AB 中的成员变量只有 b2 的长度最长，为 8，故 AB 的长度必须是 8 的倍数 40。因此在定义结构时应尽量将成员和其长度对应起来，如下：

struct ABC1 { char a, b; char d; long c; };

struct ABC2 { char a, b; long c; char d; };

ABC1 的长度为 8 个字节，而 ABC2 的长度为 12 个字节，其中 ABC1::c 和 ABC2::c 映射的偏移都为 4。

应注意上面说的规则一般都可以通过编译选项而进行一定的改变，不同的编译器将给出不同的修改方式，在此不表。

本篇说明了如何使用类型定义符“{}”来定义自定义类型，说明了两种自定义类型，实际还有许多自定义类型的内容未说明，将在下篇介绍，即后面介绍的类及类相关的内容都可应用在联合和结构上，因为它们都是自定义类型。

C++从零开始（十）

——何谓类

前篇说明了结构只不过是定义了内存布局而已，提到类型定义符前还可以书写 class，即类型的自定义类型（简称类），它和结构根本没有区别（仅有一点小小的区别，下篇说明），而之所以还要提供一个 class，实际是由于 C++是从 C 扩展而成，其中的 class 是 C++自己提出的一个很重要的概念，只是为了与 C 语言兼容而保留了 struct 这个关键字。不过通过前面括号中所说的小小区别也足以看出 C++的设计者为结构和类定义的不同语义，下篇说明。

暂时可以先认为类较结构的长足进步就是多了成员函数这个概念（虽然结构也可以有成员函数），在了解成员函数之前，先来看一种语义需求。

操作与资源

程序主要是由操作和被操作的资源组成，操作的执行者就是 CPU，这很正常，但有时候的确存在一些需要，需要表现是某个资源操作了另一个资源（暂时称作操作者），比如游戏中，经常出现的就是要映射怪物攻击了玩家。之所以需要操作者，一般是因为这个操作也需要修改操作者或利用操作者记录的一些信息来完成操作，比如怪物的攻击力来决定玩家被攻击后的状态。这种语义就表现为操作者具有某些功能。为了实现上面的语义，如原来所说进行映射，先映射怪物和玩家分别为结构，如下：

struct Monster { float Life; float Attack; float Defend; };

struct Player { float Life; float Attack; float Defend; };

上面的攻击操作就可以映射为 void MonsterAttackPlayer(Monster &mon, Player &pla);。注意这里期望通过函数名来表现操作者，但和前篇说的将过河方案起名为 sln 一样，属于一种本末倒置，因为这个语义应该由类型来表现，而不是函数名。为此，C++提供了成员函数的概念。

成员函数

与之前一样，在类型定义符中书写函数的声明语句将定义出成员函数，如下：

struct ABC { long a; void AB(long); };

上面就定义了一个映射元素——第一个变量 ABC::a，类型为 long ABC::;以及声明了一个

映射元素——第二个函数 ABC::AB，类型为 void (ABC::)(long)。类型修饰符 ABC::在此修饰了函数 ABC::AB，表示其为函数类型的偏移类型，即是一相对值。但由于是函数，意义和变量不同，即其依旧映射的是内存中的地址（代码的地址），但由于是偏移类型，也就是相对的，即是不完整的，因此不能对它应用函数操作符，如：ABC::AB(10);。这里将错误，因为 ABC::AB 是相对的，其相对的东西不是如成员变量那样是个内存地址，而是一个结构指针类型的参数，参数名一定为 this，这是强行定义的，后面说明。

注意由于其名字为 ABC::AB，而上面仅仅是对其进行了声明，要定义它，仍和之前的函数定义一样，如下：

void ABC::AB(long d) { this->a = d; }

应注意上面函数的名字为 ABC::AB，但和前篇说的成员变量一样，不能直接书写 long ABC::a;，也就不能直接如上书写函数的定义语句（至少函数名为 ABC::AB 就不符合标识符规则），而必须要通过类型定义符“{}”先定义自定义类型，然后再书写，这会在后面说明声明时详细阐述。

注意上面使用了 this 这个关键字，其类型为 ABC*，由编译器自动生成，即上面的函数定义实际等同于 void ABC::AB(ABC *this, long d) { this->a = d; }。而之所以要省略 this 参数的声明而由编译器来代劳是为了在代码上体现出前面提到的语义（即成员的意义），这也是为什么称 ABC::AB 是函数类型的偏移类型，它是相对于这个 this 参数而言的，如何相对，如下：

ABC a, b, c; a.ABC::AB(10); b.ABC::AB(12); c.AB(14);

上面利用成员操作符调用 ABC::AB，注意执行后，a.a、b.a 和 c.a 的值分别为 10、12 和 14，即三次调用 ABC::AB，但通过成员操作符而导致三次的 this 参数的值并不相同，并进而得以修改三个 ABC 变量的成员变量 a。注意上面书写 a.ABC::AB(10);，和成员变量一样，由于左右类型必须对应，因此也可 a.AB(10);。还应注意上面在定义 ABC::AB 时，在函数体内书写 this->a = d;，同上，由于类型必须对应的关系，即 this 必须是相应自定义类型的指针，所以也可省略 this->的书写，进而有 void ABC::AB(long d) { a = d; }。

注意这里成员操作符的作用，其不再如成员变量时返回相应成员变量类型的数字，而是返回一函数类型的数字，但不同的就是这个函数类型是无法用语法表示出来的，即 C++并没有提供任何关键字或类型修饰符来表现这个返回的类型（VC 内部提供了__thiscall 这个类型修饰符进行表示，不过写代码时依旧不能使用，只是编译器内部使用）。也就是说，当成员操作符右侧接的是函数类型的偏移类型的数字时，返回一个函数类型的数字（表示其可被施以函数操作符），函数的类型为偏移类型中给出的类型，但这个类型无法表现。即 a.AB 将返回一个数字，这个数字是函数类型，在 VC 内部其类型为 void (__thiscall ABC::)(long)，但这个类型在 C++中是非法的。

C++并没有提供类似__thiscall 这样的关键字以修饰类型，因为这个类型是要求编译器遇到函数操作符和成员操作符时，如 a.AB(10);，要将成员操作符左侧的地址作为函数调用的第一个参数传进去，然后再传函数操作符中给出的其余各参数。即这个类型是针对同时出现函数操作符和成员操作符这一特定情况，给编译器提供一些信息以生成正确的代码，而不用于修饰数字（修饰数字就要求能应付所有情况）。即类型是用于修饰数字的，而这个类型不能修饰数字，因此 C++并未提供类似__thiscall 的关键字。

和之前一样，由于 ABC::AB 映射的是一个地址，而不是一个偏移值，因此可以 ABC::AB; 但不能 ABC::a;，因为后者是偏移值。根据类型匹配，很容易就知道也可有：

void (ABC::*p)(long) = ABC::AB;或 void (ABC::*p)(long) = &ABC::AB;

进而就有：void (ABC::**pP)(long) = &p; (c.**pP)(10.0f);。之所以加括号是因为函数操作符的优先级较“*”高。再回想前篇说过指针类型的转换只是类型变化，数值不变（下篇说明数值变化的情况），因此可以有如下代码，这段代码毫无意义，在此仅为加深对成员函

数的理解。

```
struct ABC { long a; void AB( long ); };
void ABC::AB( long d )
{
	this->a = d;
}
struct AB
{
	short a, b;
	void ABCD( short tem1, short tem2 );
	void ABC( long tem );
};
void AB::ABCD( short tem1, short tem2 )
{
	a = tem1; b = tem2;
}
void AB::ABC( long tem )
{
	a = short( tem / 10 );
	b = short( tem - tem / 10 );
}
void main()
{
	ABC a, b, c; AB d;
	( c.*( void ( ABC::* )( long ) )&AB::ABC )( 43 );
	( b.*( void ( ABC::* )( long ) )&AB::ABCD )( 0XABCDEF12 );
	( d.*( void ( AB::* )( short, short ) )ABC::AB )( 0XABCD, 0XEF12 );
}
```

上面执行后，c.a 为 0X00270004，b.a 为 0X0000EF12，d.a 为 0XABCD，d.b 为 0XFFFF。对于 c 的函数调用，由于 AB::ABC 映射的地址被直接转换类型进而直接被使用，因此程序将跳到 AB::ABC 处的 a = short(tem / 10);开始执行，而参数 tem 映射的是传递参数的内存的首地址，并进而用 long 类型解释而得到 tem 为 43，然后执行。注意 b = short(tem - tem / 10);实际是 this->b = short(tem - tem / 10);，而 this 的值为 c 对应的地址，但在这里被认为是 AB* 类型（因为在函数 AB::ABC 的函数体内），所以才能 this->b 正常（ABC 结构中没有 b 这个成员变量），而 b 的偏移为 2，所以上句执行完后将结果 39 存放到 c 的地址加 2 所对应的内存，并且以 short 类型解释而得到的 16 位的二进制数存放。对于 a = short(tem / 10);也做同样事情，故最后得 c.a 的值为 0X0027004（十进制 39 转成十六进制为 0X27）。

同样，对于 b 的调用，程序将跳到 AB::ABCD，但生成的 b 的调用代码时，将参数 0XABCDEF12 按照参数类型为 long 的格式记录在传递参数的内存中，然后跳到 AB::ABCD。但编译 AB::ABCD 时又按照参数为两个 short 类型来映射参数 tem1 和 tem2 对应的地址，因此容易想到 tem1 的值将为 0XEF12，tem2 的值为 0XABCD，但实际并非如此。参数如何传递由之前说的函数调用规则决定，函数调用的具体实现细节在《C++从零开始（十五）》中说明，这里只需了解到成员函数映射的仍然是地址，而它的类型决定了如何使用它，后面说明。

声明的含义

前面已经解释过声明是什么意思，在此由于成员函数的定义规则这种新的定义语法，必须重新考虑声明的意思。注意一点，前面将一个函数的定义放到 main 函数定义的前面就可以不用再声明那个函数了；同样如果定义了某个变量，就不用再声明那个变量了。这也就是说定义语句具有声明的功能，但上面成员函数的定义语句却不具有声明的功能，下面来了解声明的真正意思。

声明是要求编译器产生映射元素的语句。所谓的映射元素，就是前面介绍过的变量及函数，都只有 3 栏（或 3 个字段）：类型栏、名字栏和地址栏（成员变量类型的这一栏就放偏移值）。即编译器每当看到声明语句，就生成一个映射元素，并且将对应的地址栏空着，然后留下一些信息以告诉连接器——此.obj 文件（编译器编译源文件后生成的文件，对于 VC 是.obj 文件）需要一些符号，将这些符号找到后再修改并完善此.obj 文件，最后连接。

回想之前说过的符号的意思，它就是一字符串，用于编译器和连接器之间的通信。注意符号没有类型，因为连接器只是负责查找符号并完善（因为有些映射元素的地址栏还是空的）中间文件（对于 VC 就是.obj 文件），不进行语法分析，也就没有什么类型。

定义是要求编译器填充前面声明没有书写的地址栏。也就是说某变量对应的地址，只有在其定义时才知道。因此实际的在栈上分配内存等工作都是由变量的定义完成的，所以才有声明的变量并不分配内存。但应注意一个重点，定义是生成映射元素需要的地址，因此定义也就说明了它生成的是哪个映射元素的地址，而如果此时编译器的映射表（即之前说的编译器内部用于记录映射元素的变量表、函数表等）中没有那个映射元素，即还没有相应元素的声明出现过，那么编译器将报错。

但前面只写一个变量或函数定义语句，它照样正常并没有报错啊？实际很简单，只需要将声明和定义看成是一种语句，只不过是向编译器提供的信息不同罢了。如：void ABC(float); 和 void ABC(float){}，编译器对它们相同看待。前者给出了函数的类型及类型名，因此编译器就只填写映射元素中的名字和类型两栏。由于其后只接了个“;”，没有给出此函数映射的代码，因此编译器无法填写地址栏。而后者，给出了函数名、所属类型以及映射的代码（空的复合语句），因此编译器得到了所有要填写的信息进而将三栏的信息都填上了，结果就表现出定义语句完成了声明的功能。

对于变量，如 long a;。同上，这里给出了类型和名字，因此编译器填写了类型和名字两栏。但变量对应的是栈上的某块内存的首地址，这个首地址无法从代码上表现出来（前面函数就通过在函数声明的后面写复合语句来表现相应函数对应的代码所在的地址），而必须由编译器内部通过计算获得，因此才硬性规定上面那样的书写算作变量的定义，而要变量的声明就需要在前面加 extern。即上面那样将导致编译器进行内部计算进而得出相应的地址而填写了映射元素的所有信息。

上面难免显得故弄玄虚，那都是因为自定义类型的出现。考虑成员变量的定义，如：

struct ABC { long a, b; double c; };

上面给出了类型——long ABC::、long ABC::和 double ABC::；给出了名字——ABC::a、ABC::b 和 ABC::c；给出了地址（即偏移）——0、4 和 8，因为是结构型自定义类型，故由此语句就可以得出各成员变量的偏移。上面得出三个信息，即可以填写映射元素的所有信息，所以上面可以算作定义语句。对于成员函数，如下：

struct ABC { void AB(float); };

上面给出了类型——void (ABC::)(float)；给出了名字——ABC::AB。不过由于没有给出地址，因此无法填写映射元素的所有信息，故上面是成员函数 ABC::AB 的声明。按照前面说

法，只要给出地址就可以了，而无需去管它是定义还是声明，因此也就可以这样：

```
struct ABC { void AB( float ){} };
```

上面给出类型和名字的同时，给出了地址，因此将可以完全填写映射元素的所有信息，是定义。上面的用法有其特殊性，后面说明。注意，如果这时再在后面写 ABC::AB 的定义语句，即如下，将错误：

```
struct ABC { void AB( float ){} };
void ABC::AB( float ) {}
```

上面将报错，原因很简单，因为后者只是定义，它只提供了 ABC::AB 对应的地址这一个信息，但映射元素中的地址栏已经填写了，故编译器将说重复定义。再单独看成员函数的定义，它给出了类型 void (ABC::)(float)，给出了名字 ABC::AB，也给出了地址，但为什么说它只给出了地址这一信息？首先，名字 ABC::AB 是不符合标识符规则的，而类型修饰符 ABC:: 必须通过类型定义符“{}”才能够加上去，这在前面已多次说明。因此上面给出的信息是：给出了一个地址，这个地址是类型为 void (ABC::)(float)，名字为 ABC::AB 的映射元素的地址。结果编译器就查找这样的映射元素，如果有，则填写相应的地址栏，否则报错，即只写一个 void ABC::AB(float){}是错误的，在其前面必须先通过类型定义符“{}”声明相应的映射元素。这也就是前面说的定义仅仅填充地址栏，并不生成映射元素。

声明的作用

定义的作用很明显了，有意义的映射（名字对地址）就是它来做，但声明有什么用？它只是生成类型对名字，为什么非得要类型对名字？它只是告诉编译器不要发出错误说变量或函数未定义？任何东西都有其存在的意义，先看下面这段代码。

```
extern"C" long ABC( long a, long b );
void main(){ long c = ABC( 10, 20 ); }
```

假设上面代码在 a.cpp 中书写，编译生成文件 a.obj，没有问题。但按照之前的说明，连接时将错误，因为找不到符号_ABC。因为名字_ABC 对应的地址栏还空着。接着在 VC 中为 a.cpp 所在工程添加一个新的源文件 b.cpp，如下书写代码。

```
extern"C" float ABC( float a ){ return a; }
```

编译并连接，现在没任何问题了，但相信你已经看出问题了——函数 ABC 的声明和定义的类型不匹配，却连接成功了？

注意上面关于连接的说明，连接时没有类型，只管符号。上面用 extern"C"使得 a.obj 要求_ABC 的符号，而 b.cpp 提供_ABC 的符号，剩余的就只是连接器将 b.obj 中_ABC 对应的地址放到 a.obj 以完善 a.obj，最后连接 a.obj 和 b.obj。

那么上面什么结果，由于需要考虑函数的实现细节，这在《C++从零开始（十五）》中再说明，而这里只要注意到一件事：编译器即使没有地址也依旧可以生成代码以实现函数操作符的功能——函数调用。之所以能这样就是因为声明时一定必须同时给出类型和名字，因为类型告诉编译器，当某个操作符涉及到某个映射元素时，如何生成代码来实现这个操作符的功能。也就是说，两个 char 类型的数字乘法和两个 long 类型的数字乘法编译生成的代码不同；对 long ABC(long);的函数调用代码和 void ABC(float)的不同。即，操作符作用的数字类型的不同将导致编译器生成的代码不同。

那么上面为什么要将 ABC 的定义放到 b.cpp 中？因为各源文件之间的编译是独立的，如果放在 a.cpp，编译器就会发现已经有这么个映射元素，但类型却不匹配，将报错。而放到 b.cpp 中，使得由连接器来完善 a.obj，到时将没有类型的存在，只管符号。下面继续。

```
struct ABC { long a, b; void AB( long tem1, long tem2 ); void ABCD(); };
void main(){ ABC a; a.AB( 10, 20 ); }
```

由上面的说法，这里虽然没有给出 ABC::AB 的定义，但仍能编译成功，没有任何问题。仍假设上面代码在 a.cpp 中，然后添加 b.cpp，在其中书写下面的代码。

```
struct ABC { float b, a; void AB( long tem1, long tem2 ); long ABCD( float ); };
void ABC::AB( long tem1, long tem2 ){ a = tem1; b = tem2; }
```

这里定义了函数 ABC::AB，注意如之前所说，由于这里的函数定义仅仅只是定义，所以必须在其前面书写类型定义符“{}”以让编译器生成映射元素。但更应该注意这里将成员变量的位置换了，这样 b 就映射的是 0 而 a 映射的是 4 了，并且还将 a、b 的类型换成了 float，更和 a.cpp 中的定义大相径庭。但没有任何问题，编译连接成功，a.AB(10,20);执行后 a.a 为 0X41A00000，a.b 为 0X41200000，而*(float*)&a.a 为 20，*(flaot*)&a.b 为 10。

为什么？因为编译器只在当前编译的那个源文件中遵循类型匹配，而编译另一个源文件时，编译其他源文件所生成的映射元素全部无效。因此声明将类型和名字绑定起来，而名字就代表了其所关联的类型的地址类型的数字，而后继代码中所有操作这个数字的操作符的编译生成都将受这个数字的类型的影响。即声明是告诉编译器如何生成代码的，其不仅仅只是个语法上说明变量或函数的语句，它是不可或缺的。

还应注意上面两个文件中的 ABC::ABCD 成员函数的声明不同，而且整个工程中（即 a.cpp 和 b.cpp 中）都没有 ABC::ABCD 的定义，却仍能编译连接成功，因为声明并不是告诉编译器已经有什么东西了，而是如何生成代码。

头文件

上面已经说明，如果有个自定义类型 ABC，在 a.cpp、b.cpp 和 c.cpp 中都要使用它，则必须在 a.cpp、b.cpp 和 c.cpp 中，各自使用 ABC 之前用类型定义符“{}”重新定义一遍这个自定义类型。如果不小心如上面那样在 a.cpp 和 b.cpp 中写的定义不一样，则将产生很难查找的错误。为此，C++提供了一个预编译指令来帮忙。

预编译指令就是在编译之前执行的指令，它由预编译器来解释执行。预编译器是另一个程序，一般情况，编译器厂商都将其合并进了 C++编译器而只提供一个程序。在此说明预编译指令中的包含指令——#include，其格式为#include <文件名>。应注意预编译指令都必须单独占一行，而<文件名>就是一个用双引号或尖括号括起来的文件名，如：#include "abc.c"、#include "C:\abc.dsw"或#include <C:\abc.exe>。它的作用很简单，就是将引号或尖括号中书写的文件名对应的文件以 ANSI 格式或 MBCS 格式（关于这两个格式可参考《C++从零开始（五）》）解释，并将内容原封不动地替换到#include 所在的位置，比如下面是文件 abc 的内容。

```
struct ABC { long a, b; void AB( long tem1, long tem2 ); };
```

则前面的 a.cpp 可改为：

```
#include "abc"
void main() { ABC a; a.AB( 10, 20 ); }
```

而 b.cpp 可改为：

```
#include "abc"
void ABC::AB( long tem1, long tem2 ){ a = tem1; b = tem2; }
```

这时，就不会出现类似上面那样在 b.cpp 中将自定义类型 ABC 的定义写错了而导致错误的结果（a.a 为 0X41A00000，a.b 为 0X41200000），进而 a.AB(10, 20);执行后，a.a 为 10，a.b 为 20。

注意这里使用的是双引号来括住文件名的，它表示当括住的只是一个文件名或相对路径而没有给出全路径时，如上面的 abc，则先搜索此时被编译的源文件所在的目录，然后搜索编译器自定的包含目录（如：C:\Program Files\Microsoft Visual Studio .NET 2003\Vc7\include 等），里面一般都放着编译器自带的 SDK 的头文件（关于 SDK，将在《C++从零开始（十八）》中说明），如果仍没有找到，则报错（注意，一般编译器都提供了一些选项以使得除了上述的目录外，还可以再搜索指定的目录，不同的编译器设定方式不同，在此不表）。

如果是用尖括号括起来，则表示先搜索编译器自定的包含目录，再源文件所在目录。为什么要不同？只是为了防止自己起的文件名正好和编译器的包含目录下的文件重名而发生冲突，因为一旦找到文件，将不再搜索后继目录。

所以，一般的 C++代码中，如果要用到某个自定义类型，都将那个自定义类型的定义分别装在两个文件中，对于上面结构 ABC，则应该生成两个文件，分别为 ABC.h 和 ABC.cpp，其中的 ABC.h 被称作头文件，而 ABC.cpp 则称作源文件。头文件里放的是声明，而源文件中放的是定义，则 ABC.h 的内容就和前面的 abc 一样，而 ABC.cpp 的内容就和 b.cpp 一样。然后每当工程中某个源文件里要使用结构 ABC 时，就在那个源文件的开头包含 ABC.h，这样就相当于将结构 ABC 的所有相关声明都带进了那个文件的编译，比如前面的 a.cpp 就通过在开头包含 abc 以声明了结构 ABC。

为什么还要生成一个 ABC.cpp？如果将 ABC::AB 的定义语句也放到 ABC.h 中，则 a.cpp 要使用 ABC，c.cpp 也要使用 ABC，所以 a.cpp 包含 ABC.h，由于里面的 ABC::AB 的定义，生成一个符号?AB@ABC@@QAEXJJ@Z（对于 VC）；同样 c.cpp 的编译也要生成这个符号，然后连接时，由于出现两个相同的符号，连接器无法确定使用哪一个，报错。因此专门定义一个 ABC.cpp，将函数 ABC::AB 的定义放到 ABC.obj 中，这样将只有一个符号生成，连接时也就不再报错。

注意上面的 struct ABC { void AB(float){} };。如果将这个放在 ABC.h 中，由于在类型定义符中就已经将函数 ABC::AB 的定义给出，则将会同上，出现两个相同的符号，然后连接失败。为了避开这个问题，C++规定如上在类型定义符中直接书写函数定义而定义的函数是 inline 函数，出于篇幅，下篇介绍。

成员的意义

上面从语法的角度说明了成员函数的意思，如果很昏，不要紧，实现不能理解并不代表就不能运用，而程序员重要的是对语言的运用能力而不是语言的了解程度（虽然后者也很重要）。下面说明成员的语义。

本文一开头提出了一种语义——某种资源具有的功能，而 C++的自定义类型再加上成员操作符“.”和“->”的运用，从代码上很容易的就表现出一种语义——从属关系。如：a.b、c.d 分别表示 a 的 b 和 c 的 d。某种资源具有的功能要映射到 C++中，就应该将这种资源映射成一自定义类型，而它所具有的功能就映射成此自定义类型的成员函数，如最开始提到的怪物和玩家，则如下：

```
struct Player { float Life; float Attack; float Defend; };
struct Monster { float Life; float Attack; float Defend; void AttackPlayer( Player &pla ); };
Player player; Monster a; a.AttackPlayer( player );
```

上面的语义就非常明显，代码执行的操作是怪物 a 攻击玩家 player，而 player.Life 就代表玩家 player 的生命值。假设如下书写 Monster::AttackPlayer 的定义：

```
void Monster::AttackPlayer( Player &pla )
```

```
{
    pla.Life -= Attack - pla.Defend;
}
```

上面的语义非常明显：某怪物攻击玩家的方法就是将被攻击的玩家的生命值减去自己的攻击力减被攻击的玩家的防御力的值。语义非常清晰，代码的可读性好。而如原来的写法：

```
void MonsterAttackPlayer( Monster &mon, Player &pla )
{
    pla.Life -= mon.Attack - pla.Defend;
}
```

则代码表现的语义：怪物攻击玩家是个操作，此操作需要操作两个资源，分别为怪物类型和玩家类型。这个语义就没表现出我们本来打算表现的想法，而是怪物的攻击功能的另一种解释（关于这点，将在《C++从零开始（十二）》中详细阐述），其更适合表现收银工作。比如收银台实现的是收钱的工作，客户在柜台买了东西，由营业员开出单据，然后客户将单据拿到收银台交钱。这里收银台的工作就需要操作两个资源——钱和单据，这时就应该将收钱这个工作映射为如上的函数而不是成员函数，因为在这个算法中，收银台没有被映射成自定义类型的必要性，即我们对收银的工作由谁做不关心，只关心它如何做。

至此介绍完了自定义类型的一半内容，通过这些内容已经可以编写出能体现较复杂语义的代码了，下篇将说明自定义类型的后半内容，它们的提出根本可以认为就是语义的需要，所以下篇将从剩余内容是如何体现语义的来说明，不过依旧要说明各自是如何实现的。

C++从零开始（十一）上篇

C++从零开始（十一）上篇

——类的相关知识

前面已经介绍了自定义类型的成员变量和成员函数的概念，并给出它们各自的语义，本文继续说明自定义类型剩下的内容，并说明各自的语义。

权限

成员函数的提供，使得自定义类型的语义从资源提升到了具有功能的资源。什么叫具有功能的资源？比如要把收音机映射为数字，需要映射的操作有调整收音机的频率以接收不同的电台；调整收音机的音量；打开和关闭收音机以防止电力的损耗。为此，收音机应映射为结构，类似下面：

```
struct Radiogram
{
    double Frequency;   /* 频率 */   void TurnFreq( double value );     // 改变频率
    float  Volume;      /* 音量 */   void TurnVolume( float value );   // 改变音量
    float  Power;       /* 电力 */   void TurnOnOff( bool bOn );         // 开关
    bool   bPowerOn;    // 是否开启
};
```

上面的 Radiogram::Frequency、Radiogram::Volume 和 Radiogram::Power 由于定义为了结

构 Radiogram 的成员，因此它们的语义分别为某收音机的频率、某收音机的音量和某收音机的电力。而其余的三个成员函数的语义也同样分别为改变某收音机的频率、改变某收音机的音量和打开或关闭某收音机的电源。注意这面的“某”，表示具体是哪个收音机的还不知道，只有通过成员操作符将左边的一个具体的收音机和它们结合时才知道是哪个收音机的，这也是为什么它们被称作偏移类型。这一点在下一篇将详细说明。

注意问题：为什么要将刚才的三个操作映射为结构 Radiogram 的成员函数？因为收音机具有这样的功能？那么对于选西瓜、切西瓜和吃西瓜，难道要定义一个结构，然后给它定义三个选、切、吃的成员函数？？不是很荒谬吗？前者的三个操作是对结构的成员变量而言，而后者是对结构本身而言的。那么改成吃快餐，吃快餐的汉堡包、吃快餐的薯条和喝快餐的可乐。如果这里的两个吃和一个喝的操作变成了快餐的成员函数，表示是快餐的功能？！这其实是编程思想的问题，而这里其实就是所谓的面向对象编程思想，它虽然是很不错的思想，但并不一定是合适的，下篇将详细讨论。

上面我们之所以称收音机的换台是功能，是因为实际中我们自己是无法直接改变收音机的频率，必须通过旋转选台的那个旋钮来改变接收的频率，同样，调音量也是通过调节音量旋钮来实现的，而由于开机而导致的电力下降也不是我们直接导致，而是间接通过收听电台而导致的。因此上面的 Radiogram::Power、Radiogram::Frequency 等成员变量都具有一个特殊特性——外界，这台收音机以外的东西是无法改变它们的。为此，C++提供了一个语法来实现这种语义。在类型定义符中，给出这样的格式：<权限>:。这里的<权限>为 public、protected 和 private 中的一个，分别称作公共的、保护的和私有的，如下：

```
class Radiogram
{
protected: double m_Frequency; float m_Volume; float m_Power;
private:    bool    m_bPowerOn;
public:     void TurnFreq( double ); void TurnVolume( float ); void TurnOnOff( bool );
};
```

可以发现，它和之前的标号的定义格式相同，但并不是语句修饰符，即可以 struct ABC{ private: };。这里不用非要在 private:后面接语句，因为它不是语句修饰符。从它开始，直到下一个这样的语法，之间所有的声明和定义而产生的成员变量或成员函数都带有了它所代表的语义。比如上面的类 Radiogram，其中的 Radiogram::m_Frequency、Radiogram::m_Volume 和 Radiogram::m_Power 是保护的成员变量，Radiogram::m_bPowerOn 是私有的成员变量，而剩下的三个成员函数都是公共的成员函数。注意上面的语法是可以重复的，如：struct ABC { public: public: long a; private: float b; public: char d; };。

什么意思？很简单，公共的成员外界可以访问，保护的成员外界不能访问，私有的成员外界及子类不能访问。关于子类后面说明。先看公共的。对于上面，如下将报错：

```
Radiogram a; a.m_Frequency = 23.0; a.m_Power = 1.0f; a.m_bPowerOn = true;
```

因为上面对 a 的三次操作都使用了 a 的保护或私有成员，编译器将报错，因为这两种成员外界是不能访问的。而 a.TurnFreq(10);就没有任何问题，因为成员函数 Radiogram::TurnFreq 是公共成员，外界可以访问。那么什么叫外界？对于某个自定义类型，此自定义类型的成员函数的函数体内以外的一切能写代码的地方都称作外界。因此，对于上面的 Radiogram，只有它的三个成员函数的函数体内可以访问它的成员变量。即下面的代码将没有问题。

```
void Radiogram::TurnFreq( double value ) { m_Frequency += value; }
```

因为 m_Frequency 被使用的地方是在 Radiogram::TurnFreq 的函数体内，不属于外界。

为什么要这样？表现最开始说的语义。首先，上面将成员定义成 public 或 private 对于最终生成的代码没有任何影响。然后，我之前说的调节接收频率是通过调节收音机里面的共

谐电容的容量来实现的，这个电容的容量人必须借助元件才能做到，而将接收频率映射成数字后，由于是数字，则 CPU 就能修改。如果直接 a.m_Frequency += 10;进行修改，就代码上的意义，其就为：执行这个方法的人将收音机的接收频率增加 10KHz，这有违我们的客观世界，与前面的语义不合。因此将其作为语法的一种提供，由编译器来进行审查，可以让我们编写出更加符合我们所生活的世界的语义的代码。

应注意可以 union ABC { long a; private: short b; };。这里的 ABC::a 之前没有任何修饰，那它是 public 还是 protected？相信从前面举的那么多例子也已经看出，应该是 public，这也是为什么我之前一直使用 struct 和 union 来定义自定义类型，否则之前的例子都将报错。而前篇说过结构和类只有一点很小的区别，那就是当成员没有进行修饰时，对于类，那个成员将是 private 而不是 public，即如下将错误。

class ABC { long a; private: short b; }; ABC a; a.a = 13;

ABC::a 由于前面的 class 而被看作 private。就从这点，可以看出结构用于映射资源（可被直接使用的资源），而类用于映射具有功能的资源。下篇将详细讨论它们在语义上的差别。

构造和析构

了解了上面所提的东西，很明显就有下面的疑问：

struct ABC { private: long a, b; }; ABC a = { 10, 20 };

上面的初始化赋值变量 a 还正确吗？当然错误，否则在语法上这就算一个漏洞了（外界可以借此修改不能修改的成员）。但有些时候的确又需要进行初始化以保证一些逻辑关系，为此 C++提出了构造和析构的概念，分别对应于初始化和扫尾工作。在了解这个之前，让我们先看下什么叫实例（Instance）。

实例是个抽象概念，表示一个客观存在，其和下篇将介绍的“世界”这个概念联系紧密。比如：“这是桌子”和“这个桌子”，前者的“桌子”是种类，后者的“桌子”是实例。这里有 10 只羊，则称这里有 10 个羊的实例，而羊只是一种类型。可以简单地将实例认为是客观世界的物体，人类出于方便而给各种物体分了类，因此给出电视机的说明并没有给出电视机的实例，而拿出一台电视机就是给出了一个电视机的实例。同样，程序的代码写出来了意义不大，只有当它被执行时，我们称那个程序的一个实例正在运行。如果在它还未执行完时又要求操作系统执行了它，则对于多任务操作系统，就可以称那个程序的两个实例正在被执行，如同时点开两个 Word 文件查看，则有两个 Word 程序的实例在运行。

在 C++中，能被操作的只有数字，一个数字就是一个实例（这在下篇的说明中就可以看出），更一般的，称标识记录数字的内存的地址为一个实例，也就是称变量为一个实例，而对应的类型就是上面说的物体的种类。比如：long a, *pA = &a, &ra = a;，这里就生成了两个实例，一个是 long 的实例，一个是 long*的实例（注意由于 ra 是 long&所以并未生成实例，但 ra 仍然是一个实例）。同样，对于一个自定义类型，如：Radiogram ab, c[3];，则称生成了四个 Radiogram 的实例。

对于自定义类型的实例，当其被生成时，将调用相应的构造函数；当其被销毁时，将调用相应的析构函数。谁来调用？编译器负责帮我们编写必要的代码以实现相应构造和析构的调用。构造函数的原型（即函数名对应的类型，如 float AB(double, char);的原型是 float(double, char)）的格式为：直接将自定义类型的类型名作为函数名，没有返回值类型，参数则随便。对于析构函数，名字为相应类型名的前面加符号“~”，没有返回值类型，必须没有参数。如下：

struct ABC { ABC(); ABC(long, long); ~ABC(); bool Do(long); long a, count; float *pF; };

```
ABC::ABC() { a = 1; count = 0; pF = 0; }
ABC::ABC( long tem1, long tem2 ) { a = tem1; count = tem2; pF = new float[ count ]; }
ABC::~ABC() { delete[] pF; }
bool ABC::Do( long cou )
{
	float *p = new float[ cou ];
	if( !p )
		return false;
	delete[] pF;
	pF = p;
	count = cou;
	return true;
}
extern ABC g_ABC;
void main(){ ABC a, &r = a; a.Do( 10 ); { ABC b( 10, 30 ); } ABC *p = new ABC[10]; delete[] p; }
ABC g_a( 10, 34 ), g_p = new ABC[5];
```

上面的结构 ABC 就定义了两个构造函数(注意是两个重载函数),名字都为 ABC::ABC(实际将由编译器转成不同的符号以供连接之用)。也定义了一个析构函数(注意只能定义一个,因为其必须没有参数,也就无法进行重载了),名字为 ABC::~ABC。

再看 main 函数,先通过 ABC a;定义了一个变量,因为要在栈上分配一块内存,即创建了一个数字(创建装数字的内存也就导致创建了数字,因为内存不能不装数字),进而创建了一个 ABC 的实例,进而调用 ABC 的构造函数。由于这里没有给出参数(后面说明),因此调用了 ABC::ABC(),进而 a.a 为 1,a.pF 和 a.count 都为 0。接着定义了变量 r,但由于它是 ABC&,所以并没有在栈上分配内存,进而没有创建实例而没有调用 ABC::ABC。接着调用 a.Do,分配了一块内存并把首地址放在 a.pF 中。

注意上面变量 b 的定义,其使用了之前提到的函数式初始化方式。它通过函数调用的格式调用了 ABC 的构造函数 ABC::ABC(long, long)以初始化 ABC 的实例 b。因此 b.a 为 10,b.count 为 30,b.pF 为一内存块的首地址。但要注意这种初始化方式和之前提到的“{}”方式的不同,前者是进行了一次函数调用来初始化,而后者是编译器来初始化(通过生成必要的代码)。由于不调用函数,所以速度要稍快些(关于函数的开销在《C++从零开始(十五)》中说明)。还应注意不能 ABC b = { 1, 0, 0 };,因为结构 ABC 已经定义了两个构造函数,则它只能使用函数式初始化方式初始化了,不能再通过“{}”方式初始化了。

上面的 b 在一对大括号内,回想前面提过的变量的作用域,因此当程序运行到 ABC *p = new ABC[10];时,变量 b 已经消失了(超出了其作用域),即其所分配的内存语法上已经释放了(实际由于是在栈上,其并没有被释放),进而调用 ABC 的析构函数,将 b 在 ABC::ABC(long, long)中分配的内存释放掉以实现扫尾功能。

对于通过 new 在堆上分配的内存,由于是 new ABC[10],因此将创建 10 个 ABC 的实例,进而为每一个实例调用一次 ABC::ABC(),注意这里无法调用 ABC::ABC(long, long),因为 new 操作符一次性就分配了 10 个实例所需要的内存空间,C++并没有提供语法(比如使用“{}”)来实现对一次性分配的 10 个实例进行初始化。接着调用了 delete[] p;,这释放刚分配的内存,即销毁了 10 个实例,因此将调用 ABC 的析构函数 10 次以进行 10 次扫尾工作。

注意上面声明了全局变量 g_ABC,由于是声明,并不是定义,没有分配内存,因此未产生实例,故不调用 ABC 的构造函数,而 g_a 由于是全局变量,C++保证全局变量的构造函数在开始执行 main 函数之前就调用,所有全局变量的析构函数在执行完 main 函数之后才调

用（这一点是编译器来实现的，在《C++从零开始（十九）》中将进一步讨论）。因此 g_a.ABC(10, 34)的调用是在 a.ABC()之前，即使它的位置在 a 的定义语句的后面。而全局变量 g_p 的初始化的数字是通过 new 操作符的计算得来，结果将在堆上分配内存，进而生成 5 个 ABC 实例而调用了 ABC::ABC()5 次，由于是在初始化 g_p 的时候进行分配的，因此这 5 次调用也在 a.ABC()之前。由于 g_p 仅仅只是记录首地址，而要释放这 5 个实例就必须调用 delete（不一定，也可不调用 delete 依旧释放 new 返回的内存，在《C++从零开始（十九）》中说明），但上面并没有调用，因此直到程序结束都将不会调用那 5 个实例的析构函数，那将怎样？后面说明异常时再讨论所谓的内存泄露问题。

因此构造的意思就是刚分配了一块内存，还未初始化，则这块内存被称作原始数据（Raw Data），前面说过数字都必须映射成算法中的资源，则就存在数字的有效性。比如映射人的年龄，则这个数字就不能是负数，因为没有意义。所以当得到原始数据后，就应该先通过构造函数的调用以保证相应实例具有正确的意义。而析构函数就表示进行扫尾工作，就像上面，在某实例运作的期间（即操作此实例的代码被执行的时期）动态分配了一些内存，则应确保其被正确释放。再或者这个实例和其他实例有关系，因确保解除关系（因为这个实例即将被销毁），如链表的某个结点用类映射，则这个结点被删除时应在其析构函数中解除它与其它结点的关系。

派生和继承

上面我们定义了类 Radiogram 来映射收音机，如果又需要映射数字式收音机，它和收音机一样，即收音机具有的东西它都具有，不过多了自动搜台、存储台、选台和删除台的功能。这里提出了一个类型体系，即一个实例如果是数字式收音机，那它一定也是收音机，即是收音机的一个实例。比如苹果和梨都是水果，则苹果和梨的实例一定也是水果的实例。这里提出三个类型：水果、苹果和梨。其中称水果是苹果的父类（父类型），苹果是水果的子类（子类型）。同样，水果也是梨的父类，梨是水果的子类。这种类型体系是很有意义的，因为人类就是用这种方式来认知世界的，它非常符合人类的思考习惯，因此 C++又提出了一种特殊语法来对这种语义提供支持。

在定义自定义类型时，在类型名的后面接一“:”，然后接 public 或 protected 或 private，接着再写父类的类型名，最后就是类型定义符“{}”及相关书写，如下：

```
class DigitalRadiogram : public Radiogram
{
protected:    double m_Stations[10];
public:       void SearchStation();                    void SaveStation( unsigned long );
                void SelectStation( unsigned long ); void EraseStation( unsigned long );
};
```

上面就将 Radiogram 定义为了 DigitalRadiogram 的父类，DigitalRadiogram 定义成了 Radiogram 的子类，被称作类 Radiogram 派生了类 DigitalRadiogram，类 DigitalRadiogram 继承了类 Radiogram。

上面生成了 5 个映射元素，就是上面的 4 个成员函数和 1 个成员变量，但实际不止。由于是从 Radiogram 派生，因此还将生成 7 个映射，就是类 Radiogram 的 7 个成员，但名字变化了，全变成 DigitalRadiogram::修饰，而不是原来的 Radiogram::修饰，但是类型却不变化。比如其中一个映射元素的名字就为 DigitalRadiogram::m_bPowerOn，类型为 bool Radiogram::，映射的偏移值没变，依旧为 16。同样也有映射元素 DigitalRadiogram::TurnFreq，类型为 void

(Radiogram::)(double)，映射的地址依旧没变，为 Radiogram::TurnFreq 所对应的地址。因此就可以如下：

```
void DigitalRadiogram::SaveStation( unsigned long index )
{
    if( index >= 10 ) return;
    m_Station[ index ] = m_Frequency; m_bPowerOn = true;
}
DigitalRadiogram a; a.TurnFreq( 10 ); a.SaveStation( 3 );
```

上面虽然没有声明 DigitalRadiogram::TurnFreq，但依旧可以调用它，因为它是从 Radiogram 派生来的。注意由于 a.TurnFreq(10); 没有书写全名，因此实际是 a.DigitalRadiogram::TurnFreq(10);，因为成员操作符左边的数字类型是 DigitalRadiogram。如果 DigitalRadiogram 不从 Radiogram 派生，则不会生成上面说的 7 个映射，结果 a.TurnFreq(10); 将错误。

注意上面的 SaveStation 中，直接书写了 m_Frequency，其等同于 this->m_Frequency，由于 this 是 DigitalRadiogram*（因为在 DigitalRadiogram::SaveStation 的函数体内），所以实际为 this->DigitalRadiogram::m_Frequency，也因此，如果不是派生自 Radiogram，则上面将报错。并且由类型匹配，很容易知道：void (Radiogram::*p)(double) = DigitalRadiogram::TurnFreq;。虽然这里是 DigitalRadiogram::TurnFreq，但它的类型是 void (Radiogram::)(double)。

应注意在 SaveStation 中使用了 m_bPowerOn，这个在 Radiogram 中被定义成私有成员，也就是说子类也没权访问，而 SaveStation 是其子类的成员函数，因此上面将报错，权限不够。

上面通过派生而生成的 7 个映射元素各自的权限是什么？先看上面的派生代码：

```
class DigitalRadiogram : public Radiogram {…};
```

这里由于使用 public，被称作 DigitalRadiogram 从 Radiogram 公共继承，如果改成 protected 则称作保护继承，如果是 private 就是私有继承。有什么区别？通过公共继承而生成的映射元素（指从 Radiogram 派生而生成的 7 个映射元素），各自的权限属性不变化，即上面的 DigitalRadiogram::m_Frequency 对类 DigitalRadiogram 来说依旧是 protected，而 DigitalRadiogram::m_bPowerOn 也依旧是 private。保护继承则所有的公共成员均变成保护成员，其它不变。即如果保护继承，DigitalRadiogram::TurnFreq 对于 DigitalRadiogram 来说将为 protected。私有继承则将所有的父类成员均变成对于子类来说是 private。因此上面如果私有继承，则 DigitalRadiogram::TurnFreq 对于 DigitalRadiogram 来说是 private 的。

上面可以看得很简单，即不管是什么继承，其指定了一个权限，父类中凡是高于这个权限的映射元素，都要将各自的权限降低到这个权限（注意是对子类来说），然后再继承给子类。上面一直强调“对于子类来说”，什么意思？如下：

```
struct A { long a; protected: long b; private: long c; };
struct B : protected A { void AB(); };
struct C : private B { void ABC(); };
void B::AB() { b = 10; c = 10; }
void C::ABC() { a = 10; b = 10; c = 10; AB(); }
A a; B b; C c; a.a = 10; b.a = 10; b.AB(); c.AB();
```

上面的 B 的定义等同于 struct B { protected: long a, b; private: long c; public: void AB(); };。

上面的 C 的定义等同于 struct C { private: long a, b, c; void AB(); public: void ABC(); };

因此，B::AB 中的 b = 10;没有问题，但 c = 10;有问题， 因为编译器看出 B::c 是从父类继

承生成的，而它对于父类来说是私有成员，因此子类无权访问，错误。接着看 C::ABC，a = 10; 和 b = 10;都没问题，因为它们对于 B 来说都是保护成员，但 c = 10;将错误，因为 C::c 对于父类 B 来说是私有成员，没有权限，失败。接着 AB();，因为 C::AB 对于父类 B 来说是公共成员，没有问题。

接着是 a.a = 10;，没问题；b.a = 10;，错误，因为 B::a 是 B 的保护成员；b.AB();，没有问题；c.AB();，错误，因为 C::AB 是 C 的私有成员。应注意一点：public、protected 和 private 并不是类型修饰符，只是在语法上提供了一些信息，而继承所得的成员的类型都不会变化，不管它保护继承还是公共继承，权限起作用的地方是需要运用成员的地方，与类型没有关系。什么叫运用成员的地方？如下：

long (A::*p) = &A::a; p = &A::b;

void (B::*pB)() = B::AB; void (C::*pC)() = C::ABC; pC = C::AB;

上面对变量 p 的初始化操作没有问题，这里就运用了 A::a。但是在 p = &A::b;时，由于运用了 A::b，则编译器就要检查代码所处的地方，发现对于 A 来说属于外界，因此报错，权限不够。同样下面对 pB 的赋值没有问题，但 pC = C::AB;就错误。而对于 b.a = 10;，这里由于成员操作符而运用了类 B 的成员 B::a，所以在这里进行权限检查，并进而发现权限不够而报错。

好，那为什么要搞得这么复杂？弄什么保护、私有和公共继承？首先回想前面说的为什么要提供继承，因为想从代码上体现类型体系，说明一个实例如果是一个子类的实例，则它也一定是一个父类的实例，即可以按照父类的定义来操作它。虽然这也可以通过之前说的转换指针类型来实现，但前者能直接从代码上表现出类型继承的语义（即子类从父类派生而来），而后者只能说明用不同的类型来看待同一个实例。

那为什么要给继承加上权限？表示这个类不想外界或它的子类以它的父类的姿态来看待它。比如鸡可以被食用，但做成标本的鸡就不能被食用。因此子类“鸡的标本”在继承时就应该保护继承父类“鸡”，以表示不准外界（但准许其派生类）将它看作是鸡。它已经不再是鸡，但它实际是由鸡转变过来的。因此私有和保护继承实际很适合表现动物的进化关系。比如人是猴子进化来的，但人不是猴子。这里人就应该使用私有继承，因为并不希望外界和人的子类——黑种人、黄种人、白种人等——能够把父类“人”看作是猴子。而公共继承就表示外界和子类可以将子类的实例看成父类的实例。如下：

```
struct A { long a, b; };
struct AB : private A { long c; void ABCD(); };
struct ABB : public AB { void AAA(); };
struct AC : public A { long c; void ABCD(); };
void ABC( A *a ) { a->a = 10; a->b = 20; }
void main() { AB b; ABC( &b ); AC c; ABC( &c ); }
void AB::ABCD() { AB b; ABC( &b ); }
void AC::ABCD() { AB b; ABC( &b ); }
void ABB::AAA() { AB b; ABC( &b ); }
```

上面的类 AC 是公共继承，因此其实例 c 在执行 ABC(&c);时将由编译器进行隐式类型转换，这是一个很奇特的特性，本文的下篇将说明。但类 AB 是私有继承，因此在 ABC(&b); 时编译器不会进行隐式类型转换，将报错，类型不匹配。对于此只需 ABC((A*)&b);以显示进行类型转换就没问题了。

注意前面的红字，私有继承表示外界和它的子类都不可以用父类的姿态来看待它，因此在 ABB::AAA 中，这是 AB 的子类，因此这里的 ABC(&b);将报错。在 AC::ABCD 中，这里对于 AB 来说是外界，报错。在 AB::ABCD 中，这里是自身，即不是子类也不是外界，所以 ABC(&b);

将没有问题。如果将 AB 换成保护继承，则在 ABB::AAA 中的 ABC(&b);将不再错误。

关于本文及本文下篇所讨论的语义，在《C++从零开始（十二）》中会专门提出一个概念以给出一种方案来指导如何设计类及各类的关系。由于篇幅限制，本文分成了上下两篇，剩下的内容在本文下篇说明。

C++从零开始（十二）

C++从零开始（十二）

——何谓面向对象编程思想

前面已经说明了 C++中最重要的概念——类，并且介绍了大部分和类相关的知识，至此，已经可以开始做些编程方面比较高级的应用——设计程序，而不再只是将算法变成代码。要说明如何设计程序，有必要先了解何谓编程思想。

编程思想

编程，即编写程序，而之前已经说过，程序就是方法的描述，那么编程就是编写方法的描述。我知道如何到人民公园，然后我就编写了到人民公园的方法的描述——从市中心开始向东走 400 米再向右转走 200 米就是。接着另一个人也知道如何去，但他编的程序却是——从市中心沿人民东路走过两个交叉口，在第三个交叉口处右转，直走就能在右手方看到。很明显，两个程序不同，但最后走的路线是相同的，即大家的方法相同，但描述不同。

所谓的编程思想，就是如何编程，即编写程序的方法。那么之前在《C++从零开始（八）》中说的编程的三个步骤其实就是一种编程思想。这也是为什么不同的人对同一算法编写出的程序不同（指程序逻辑，不是简单的变量或函数名不同），不同的人的编程思想不同。

如果多编或多看一些程序，就会发现编程思想是很重要的。好的编程思想编出的程序条理分明，可维护性高；差的编程思想编出的程序晦涩难懂，可维护性低。注意，这里是从程序的易读性来比较的，实际出于效率，是会使用不符合人脑思维习惯的编程思想，进而导致代码的难于维护，但为了效率还是会经常在程序的瓶颈位置使用被优化了的代码（这种代码一般使用汇编语言编写，算法则很大程度上是数学上的优化，丢弃了大部分其在人类世界中的意义）。

本系列一直坚持并推荐这么一个编程思想——一切均按照语义来编写。而语义是语言的意义，之前说它是代码在人类世界中的意义。比如桌子，映射成一个结构，有桌脚数、颜色等成员变量，那么为什么没有质量、材料、价格、生产日期等成员？对此有必要说明一下“人类世界”的含义。

世界

我们生活在一个四维的客观物理世界中，游戏中的怪物生活在游戏定义的游戏世界中，白雪公主生活在一个童话世界中。什么叫世界？世界即规则的集合。比如客观世界中，力可作用于有质量的物体上，并进而按照运动学定律改变物体的速度；电荷异性相吸同性相斥；能量守恒等，这些都是对客观世界这个规则的集合中的某些规则的描述。注意它们都只是规

则的描述，不是规则，就好像程序是方法的描述，但不是方法。即方法和规则都是抽象的逻辑概念，各自通过程序和论调来表现。程序就是我们要编写的，而论调就是一门理论，如概率论、运动学、流体力学等。而前面所说的游戏世界，是因为游戏也是一系列规则，关于这点，我在我写的另一篇文章《游戏论》中做了详细的阐述，如果还未理解世界的概念，《游戏论》中关于何谓游戏的阐述希望能有所帮助。同样，童话世界也是由一系列的规则组成。如白雪公主能吃东西，能睡觉，并且能因为吃了毒苹果而中毒；魔镜能回答问题等。

那么就算了解了世界这个概念又怎样？有什么用？前面说了本系列是推荐按照语义来编写程序，即知道了算法后按照《C++从零开始（八）》中说的三步来编写程序。而算法是基于某些规则的，如给出 1 到 100 求和的算法是(1+100)*100/2，这里就暗示已经有那么些规则说明什么是加减乘除，什么是求和。即一个算法一定是就一个世界来说的，它在另一个世界可能毫无意义。因为算法就是方法，是由之前说的命令和被操作的资源组成，而命令和资源就是由世界来定义的。

前面说根据算法写代码，其实是先制订了一个世界来做为算法展示的平台。如之前的商人过河，其就在如下的一个规则集上表现的：有一只能坐两人的船可以载人过河；有三个商人和三个仆人在河的一边；河的任意一边仆人数多过商人数商人就会被杀。这是对过河问题所基于的世界的严重不准确的描述，但在这过于抽象并没什么好处，只用注意：上面的商人和仆人不是现实世界中的商人和仆人，他们不能吃饭不能睡觉不能讲话，甚至连走路都不会，唯一会的是通过坐船过河来改变自身的位置。当某一位置（即河的某一岸）的仆人的实例多于商人的实例时（且商人的实例至少有一个），则称商人被杀。上面的描述暂且称为商人仆人论，它是对过河问题所基于的世界的一个描述。

另一个人却不像上面那样看待问题。河有两个岸，每个河岸总对应着两个数字——商人数和仆人数。有一个途径能按照某个规则改变河岸对应的两个数字（就是坐只能坐两人的船过河），而当任何一个河岸所对应的仆人数多于商人数时（且商人数不为零），则称商人被杀。此人没有定义商人和仆人这么两个概念，而只定义了一个概念——河岸，此概念具有两个属性——商人数和仆人数。这是另一个论调，暂且称为河岸论。

什么意思？上面就是对商人过河问题所基于的世界的两个不同论调。注意上面论调不同，但描述的都是同一个世界，就好像动力学和量子力学，都是对客观世界物体之间作用规则的描述，但大相径庭。算法总是基于一个世界，但更准确点的说法应是算法总是基于一个世界的描述，而所谓的设计程序就是编写算法所基于的世界描述，即论调，而论调其实就是问题的描述。

现在考虑前面说的商人仆人论和河岸论，它们都是同一世界的描述，但前者提出两个名词性概念——商人和仆人，各自具有“位置”这个状态和“坐船”这么一个功能以及“商人被杀”这个动词性概念；后者提出一个名词性概念——河岸，具有“商人数”和“仆人数”两个属性和“商人被杀”及“坐船”两个动词性概念。在此，说后者比前者好，因为后者定义的名词性概念更少（即名词性概念比动词性概念更容易增加架构的复杂性，因为其代表了世界中东西的种类，种类越繁多则世界越复杂，越难以实现），虽说不一定更容易理解，但结构更简单。

易发现，所有的论调都可以只由“名词性概念”和“动词性概念”组成，其中前者在数学中就是数、实数、复数等，后者是加减乘除、求导等，它们都被称作定义。在《游戏论》中，我将前者称为类，而类的实例就是方法中被操作的的资源，后者称为命令。而在方法中，前者是资源的类型，后者是操作的类型。一个论调，提出的概念越少，结构就越简单，也就越好。但应注意，就电脑编程来说，由于电脑并不是抽象的概念，而是存在效率因素的，因此基于前述的好的论调的算法而编出的代码的运行效率并不一定高。

因此，所谓的程序设计，就是设计算法所基于的论调，而好的程序设计，就是相应的论

调设计得好。但前面说了，效率并不一定高，对此，一般仅在代码的瓶颈位置另外设计，而程序的整体架构依旧按照之前的设计。随着程序的日趋庞大，清晰简明的程序架构越显重要，而要保持程序架构的简明，就应设计好的论调；要保持架构的清晰，就应按照语义来编写代码。下面，介绍如此风靡的面向对象编程思想来帮助设计程序。

何谓对象

要说明面向对象，首先应了解何谓对象。对象就是前述的“名词性概念”的实现，即一个实例。如商人仆人论中有商人和仆人两个“名词性概念”，其有三个商人和三个仆人，则称有六个对象，分别是三个商人的实例和三个仆人的实例。应注意对象和实例的区别，其实它们没有区别，如果非要说区别，可以认为对象能够没有状态，但实例一定有状态。

那么什么叫状态？还是先来看看什么叫属性。桌子有个属性叫颜色，这张桌子是红色的而那张是绿的。人有个状态叫脸色，这个人的脸色红润而那个的惨白。都是颜色，但一个是属性一个是状态，什么区别？如果把桌子和人都映射成类，那么桌子的颜色和人的脸色都应映射成相应类的成员变量，而两者的区别就是桌子的实例在主要运作过程中颜色都不变化，其主要用于读；人的脸色在人的实例的主要运作过程中可能变化，其主要用于写。什么叫运作过程？类映射的是资源，资源可以具有功能，即成员函数，当一个实例的功能执行时，就是这个实例的运作过程。

桌子有个功能是“放东西”，当调用这个成员函数时，其中会读取颜色这个属性的值以判断放在桌子上的东西的颜色是否和桌子的颜色搭配协调。人有个功能是“泡澡”，其可以使相应实例的脸色从惨白向红润转变。但桌子也有个功能是“改变颜色”，调用它可以改变桌子的颜色。按照前面所说，颜色是属性，应该被读，但这里却在实例的运作过程中对它进行了写操作。注意前面说的是“主要运作过程”，即桌子的目的是用来“放东西”，不是“改变颜色”。如果桌子这个概念在其相应世界中主要是用来改变其颜色而不是放东西，此时桌子只不过是一个能记录颜色值的容器，而这时桌子的颜色就是状态，不是属性了。

有何意义？属性和状态都映射为成员变量，从代码上是看不出它们的区别的，但它们的语义是有严重区别的。属性是用来配置实例而状态是用来表现实例。在面向对象编程思想中，只是简单地说对象是具有属性和功能（也被称作方法）的实例，这在编写的程序所基于的世界比较复杂时显得非常地孱弱，而且就是对“属性”的错误理解，再加上“封装”这个词汇的席卷，导致出现大量的荒谬代码，后面说明。

属性和状态的差别导致出现所谓的无状态对象（在 MTS——Microsoft Transaction Server 中提出，称作 Stateless Component，无状态组件），这正是对象和实例的差别——对象是实现，因此可以是一个抽象概念的实现；实例是实际存在，不能是抽象概念的实现。这在 C++ 代码上就表现为没有成员变量的类和有成员变量的类。如下：

```
struct Search { virtual int search( int*, int, int ); };
Search a, b; int c[3] = { 10, 20, 5 }; a.search( c, 3, 20 );
```

这里就生成两个对象 a 和 b，它们都是抽象概念——搜索功能的对象。注意结构 Search 没有成员变量，因为不需要，那么 a 和 b 的长度是多少？由于可能出现下面的情况，一般的编译器都将上面的 a 和 b 的长度定为一个字节，进而&a 就不等于&b。

```
struct BSearch : public Search { int search( int*, int, int ); };
Search *p; BSearch d; p = &a; p = &b; p = &d; p->search( c, 3, 5 );
```

注意从代码上依旧可以称上面生成了 Search 的两个实例 a 和 b，BSearch 的一个实例 d（即使实际上它们根本不存在，逻辑上大小为零），这也就是为什么之前说它和对象没有区

别，仅仅有概念上的微小差别。

应注意前面提到的无状态对象并不是说没有成员变量的类的实例，只是没状态，并不代表没有属性。如前面的 BSearch 可能有个属性 m_MaxSearchTimes 以表示折半搜索时如果搜索 m_MaxSearchTimes 那么多次仍没找到，则 BSearch::search 返回没找到。虽然这里 BSearch 有了成员变量，但就逻辑上它还是一个抽象概念。由于属性和状态的实现相同（都通过成员变量），因此要实现无状态对象需要一些特殊手段，由于与本系列无关，在此不表。

前面的 a 和 b 有区别吗？为什么要有两个实例？“搜索功能”按照之前说的语义不是更应该映射为函数？为什么要映射成没有成员变量的类？上面的用法在 STL（Standard Template Library——标准模版库）中被使用，做了一些变形，称作函数类，是一种编程技巧。但的确有这种语义——查找功能有三个参数：查找条件、查找位置（即欲搜索的容器或集合）、查找前排序容器或集合的方法。这里传递函数指针不是刚刚好吗（实际并不刚刚好，指针的语义是引用，在这并不很准确）？这就是所谓的面向对象编程思想。

面向对象编程思想

前面已说明设计程序就是编写程序欲解决的问题的描述，也就是编写论调。而论调可以只用“名词性概念”和“动词性概念”表现出来，对象又正好是“名词性概念”的实现，而利用前面说的没有成员变量的类来映射“动词性概念”就可以将其转换为对象。因此，一个世界，可以完全由对象组成，而将算法所基于的世界只用对象表现出来，再进行后续代码的编写，这种编程方法就被称作面向对象的编程思想。

注意，先设计算法应基于的世界，再全部用对象将其表述出来，然后再设计算法，最后映射为代码。但前面在编写商人过河问题时是直接给出算法的，并没有设计世界啊？其实由于那个问题的过于简单，我直接下意识地设计了世界，并且用前面所说的河岸论来描述它。应注意世界的设计完全依赖于问题，而准确地说，前面我并没有设计世界，而是设计了河岸论来描述问题。

接着，由于对象就是实例，因此以对象来描述世界在 C++中就是设计类，通过类的实例来组合表现世界。但应注意，面向对象是以对象来描述世界，但也描述算法，因为算法也会提出一些需要被映射的概念，如前面商人过河问题的算法中的过河方案。但切记，当描述算法时操作了描述世界时定义的类，则一定要保持那个类的设计，不要因为算法中对那个类的实例的操作过于复杂而将那部分算法映射为这个类的一个成员函数，因为这严重遮蔽了算法的实现，破坏了程序的架构。如一个算法是让汽车原地不停打转，需要复杂的操作，那么难道给汽车加一个功能，让它能原地不停地打转？！这是在设计类的时候经常犯的错误，也由于这个原因，一个面向对象编写的代码并不是想象的只由类组成，其也可能由于将算法中的某些操作映射成函数而有大量的全局函数。请记住：设计类时，如果是映射世界里的概念，不要考虑算法，只以这个世界为边界来设计它，不要因为算法里的某个需要而给它加上错误的成员。

因此，将“名词性概念”映射成类，“名词性概念”的属性和状态映射为成员变量，“名词性概念”的功能映射为成员函数。那么“动词性概念”怎么办？映射成没有成员变量的类？前面也看见，由于过于别扭，实际中这种做法并不常见（STL 中也只是将其作为一种技巧），故经常是将它映射为函数，虽然这有背于面向对象的思想，但要易于理解得多，进而程序的架构要简明得多。

随着面向对象编程思想的问世，一种全新的设计方式诞生了。由于它是如此的好以至于广为流传，但理解的错误导致错误的思想遍地而生，更糟糕的就是本末倒置，将这个设计方

式称作面向对象的编程思想，它的名字就是封装。

封装

先来看现在在各类 VC 教程中关于对象的讲解中经常能看见的如下的一个类的设计。

```
class Person
{ private: char m_Name[20]; unsigned long m_Age; bool m_Sex;
   public:    const char* GetName() const;    void SetName( const char* );
                unsigned long GetAge() const; void SetAge( unsigned long );
                bool GetSex() const;                void SetSex( bool );
};
```

上面将成员变量全部定义为 private，然后又提供三对 Get/Set 函数来存取上面的三个成员变量（因为它们是 private，外界不能直接存取），这三对函数都是 public 的，为什么要这样？那些教材将此称作封装，是对类 Person 的内部内存布局的封装，这样外界就不知道其在内存上是如何布局的并进而可以保证内存的有效性（只由类自身操作其实例）。

首先要确认上面设计的荒谬性，它是正宗的“有门没锁”毫无意义。接着再看所谓的对内存布局的封装。回想在《C++从零开始（十）》中说的为什么每个要使用类的源文件的开头要包含相应的头文件。假设上面是在 Person.h 中的声明，然后在 b.cpp 中要使用类 Person，本来要#include "Person.h"，现在替换成下面：

```
class Person
{ public: char m_Name[20]; unsigned long m_Age; bool m_Sex;
   public: const char* GetName() const;    void SetName( const char* );
              unsigned long GetAge() const; void SetAge( unsigned long );
              bool GetSex() const;                void SetSex( bool );
};
```

然后在 b.cpp 中照常使用类 Person，如下：

```
Person a, b; a.m_Age = 20; b.GetSex();
```

这里就直接使用了 Person::m_Age 了，就算不做这样蹩脚的动作，依旧#include "Person.h"，如下：

```
struct PERSON { char m_Name[20]; unsigned long m_Age; bool m_Sex; };
Person a, b; PERSON *pP = ( PERSON* )&a; pP->m_Age = 40;
```

上面依旧直接修改了 Person 的实例 a 的成员 Person::m_Age，如何能隐藏内存布局？！请回想声明的作用，类的内存布局是编译器生成对象时必须的，根本不能对任何使用对象的代码隐藏有关对象实现的任何东西，否则编译器无法编译相应的代码。

那么从语义上来看。Person 映射的不是真实世界中的人的概念，应该是存放某个数据库中的某个记录人员信息的表中的记录的缓冲区，那么缓冲区应该具备那三对 Get/Set 所代表的功能吗？缓冲区是缓冲数据用的，缓冲后被其它操作使用，就好像箱子，只是放东西用。故上面的三对 Get/Set 没有存在的必要，而三个成员变量则不能是 private。当然，如果 Person 映射的并不是缓冲区，而在其它的世界中具备像上面那样表现的语义，则像上面那样定义就没有问题，但如果是因为对内存布局的封装而那样定义类则是大错特错的。

上面错误的根本在于没有理解何谓封装。为了说明封装，先看下 MFC（Microsoft Foundation Class Library——微软功能类库，一个定义了许多类的库文件，其中的绝大部分类是封装设计。关于库文件在说明 SDK 时阐述）中的类 CFile 的定义。从名字就可看出它映射

的是操作系统中文件的概念，但它却有这样的成员函数——CFile::Open、CFile::Close、CFile::Read、CFile::Write，有什么问题？这四个成员函数映射的都是对文件的操作而不是文件所具备的功能，分别为打开文件、关闭文件、从文件读数据、向文件写数据。这不是和前面说的成员函数的语义相背吗？上面四个操作有个共性，都是施加于文件这个资源上的操作，可以将它们叫做“被功能”，如文件具有“被打开”的功能，具有“被读取”的功能，但应注意它们实际并不是文件的功能。

按照原来的说法，应该将文件映射为一个结构，如 FILE，然后上面的四个操作应映射成四个函数，再利用名字空间的功能，如下：

```
namespace OFILE
{
    bool Open( FILE&, … );    bool Close( FILE&, … );
    bool Read( FILE&, … );    bool Write( FILE&, … );
}
```

上面的名字空间 OFILE 表示里面的四个函数都是对文件的操作，但四个函数都带有一个 FILE&的参数。回想非静态成员函数都有个隐藏的参数 this，因此，一个了不起的想法诞生了。

将所有对某种资源的操作的集合看成是一种资源，把它映射成一个类，则这个类的对象就是对某个对象的操作，此法被称作封装，而那个类被称作包装类或封装类。很明显，包装类映射的是“对某种资源的操作”，是一抽象概念，即包装类的对象都是无状态对象（指逻辑上应该是无状态对象，但如果多个操作间有联系，则还是可能有状态的，但此时它的语义也相应地有些变化。如多一个 CFile::Flush 成员函数，用于刷新缓冲区内容，则此时就至少有一个状态——缓冲区，还可有一个状态记录是否已经调用过 CFile::Write，没有则不用刷新）。

现在应能了解封装的含义了。将对某种资源的操作封装成一个类，此包装类映射的不是世界中定义的某一“名词性概念”，而是世界的“动词性概念”或算法中“对某一概念的操作”这个人为定出来的抽象概念。由于包装类是对某种资源的操作的封装，则包装类对象一定有个属性指明被操作的对象，对于 MFC 中的 CFile，就是 CFile::m_hFile 成员变量（类型为 HANDLE），其在包装类对象的主要运作过程（前面的 CFile::Read 和 CFile::Write）中被读。

有什么好处？封装提供了一种手段以将世界中的部分“动词性概念”转换成对象，使得程序的架构更加简单（多条“动词性概念”变成一个“名词性概念”，减少了“动词性概念”的数量），更趋于面向对象的编程思想。

但应区别开包装类对象和被包装的对象。包装类对象只是个外壳，而被包装的对象一定是个具有状态的对象，因为操作就是改变资源的状态。对于 CFile，CFile 的实例是包装类对象，其保持着一个对被包装对象——文件内核对象（Windows 操作系统中定义的一种资源，用 HANDLE 的实例表征）——的引用，放在 CFile::m_hFile 中。因此，包装类对象是独立于被包装对象的。即 CFile a;，此时 a.m_hFile 的值为 0 或-1，表示其引用的对象是无效的，因此如果 a.Read(…);将失败，因为操作施加的资源是无效的。对此，就应先调用 a.Open(…);以将 a 和一特定的文件内核对象绑定起来，而调用 a.Close(…);将解除绑定。注意 CFile::Close 调用后只是解除了绑定，并不代表 a 已经被销毁了，因为 a 映射的并不是文件内核对象，而是对文件内核对象操作的包装类对象。

如果仔细想想，就会发现，老虎能够吃兔子，兔子能够被吃，那这里应该是老虎有个功能是“吃兔子”还是多个兔子的包装类来封装“吃兔子”的操作？这其实不存在任何问题，“老虎吃兔子”和“兔子被吃”完全是两个不同的操作，前者涉及两种资源，后者只涉及一种资源，因此可以同时实现两者，具体应视各自在相应世界中的语义。如果对于真实世界，则可以简略地说老虎有个“吃”的功能，可以吃“肉”，而动物从“肉”和“自主能动性”多重继承，兔子再从动物继承。这里有个类叫“自主能动性”，指动物具有意识，能够自己

动作，这在C++中的表现就是有成员函数的类，表示有功能可以被操作，但收音机也具有调台等功能，难道说收音机也能自己动？！这就是世界的意义——运转。

方法——世界的驱动方式

算法就是方法，前面已说过其由操作和被操作的资源组成，即资源的类型和操作的类型。方法指出如何使用世界中定义出的各种操作，但并不执行。由前面的阐述，世界可以只由对象组成，当对象产生后，世界中所有对象的状态和属性，即成员变量，的一份拷贝，称作世界的状态的一份快照，而世界的状态的变化称作世界的运转。世界的状态就是世界中所有对象的状态和属性，要改变它，就是要执行世界定义的操作，但只能通过方法指出如何执行它以改变世界的状态，进而驱动世界，即使世界定义的操作被执行才能驱动世界。

上面越说越远了，感觉虚得很，有什么意义？考虑为什么要提出世界这个概念。世界是我们欲编程解决的问题所基于的规则集合体，而设计程序就是设计描述世界的论调，然后在这个论调上设计算法，编写出代码，执行代码，得到结果。“得到结果”？！什么是结果？即世界最终状态中的某一部分，如求圆周率的值。这其实是目的，但值得注意的是目的不止这种。代码执行的过程往往是另一种目的，如将数据保存到某个文件中；将文件打开编辑再保存等，这种目的并不关心世界的状态最后如何。而世界的状态的变化过程，也就是世界的运转则是另一种非常流行的目的——电子游戏。

不管什么样的目的，都需要改变世界的状态，即要驱动世界运转，也就必须使世界定义的操作被执行，而这只能通过方法来实现。因此在设计算法时，也就决定了驱动世界的方式。

对于上面的第一种目的，由于是要看世界的最终状态，因此一直连续执行操作到最后。在《C++从零开始（八）》中给出的商人过河的例子就在通过算法得到结果后直接调用 printf 打印出结果并结束。对于第二种目的，由于要的是执行的过程，因此也可直接连续执行操作到完。但这种目的往往要求由用户决定何时执行且执行不止一段代码，如文件打开后，直到用户给出命令（通过键盘或鼠标或其它输入方式）后才进行编辑操作，且用户可能随机地执行不同的编辑操作，最后也由用户决定是否保存文件。这种世界的运转完全由用户控制的世界驱动方式称为用户驱动方式。这里的算法仅仅是如何打开、保存文件，如何编辑数据，但由于决定是用户驱动方式，则算法就必须修改以实现这种驱动方式。再看第三种目的，要的是世界的状态的变化，则前面两种都可以。但很明显，第一种变化过程不能持续，连续执行完就完；第二种由用户驱动，则太麻烦。因此往往都会有个循环，在游戏编程中一般称其为主循环，每次循环都按照一定的规则改变世界中部分对象的状态，此称作循环驱动方式。每次循环，都会被改变状态的对象就被称作具有自主能动性，如前面提到的动物的实例。除此以外的就不称作具有自主能动性，如前面提到的收音机的实例。同样，游戏中的算法依旧不会涉及到上面提到的循环驱动方式，因此必须修改算法以实现循环驱动方式。

上面将那么多只为了说明一点，已经不能再如《C++从零开始（八）》中说的那几步来编写程序了，下面给出一个方法。

1. 当得到一个问题，应同时得到这个问题的算法（程序员并不是科学家），或由客户给出或由于过于简单而直接得出。

2. 由问题抽象设计出它的描述，即前面说的论调，也就是所谓的程序设计。

3. 将之前给出的算法用刚设计出的论调进行描述，并完善这个论调（因为算法可能带入一些原来世界中并不存在的概念）。

4. 由需要决定使用何种世界驱动方式，并实现以完善算法和论调（世界的驱动方式也可能带入一些原来并不存在的概念）。

5. 继续《C++从零开始（八）》中提出的三步。

这里给出的步骤只是一般性的，当程序对应的世界过于复杂时，上面的第 2 步将还需细分。先设计程序的大框架；再设计接口；最后决定接口的具体运用。关于接口会在《C++从零开始（十八）》中说明，而这里提到的关于接口的设计方式并不用管它，它只是使设计简单化，并不是必须的。由于其与本系列无关，在此不表。

本文提出了多个抽象的概念，例子较少，且只说了以面向对象思想设计类时应注意的一些问题，提到世界定义的概念越少越好，但并没说如何决定有哪些概念。对此，其实从前面关于老虎吃兔子的问题中就可看出，因为问题是现实世界中的问题，因此只需简单地映射现实世界中的概念，并去掉或简化一些次要概念（其实这就是语义在另一方面的体现）。在下篇，将针对本文提出的那五个程序编写步骤给出一个基于对象设计的简单样例以大致说明如何编写面向对象的程序。