C 语言非常道

前　　言

毋庸置疑，C 是非常流行的编程语言。正是因为流行，和它有关的图书之多，可以用多如牛毛、汗牛充栋来形容。

既然都已经这么多了，那我为什么还要再来一本，给牛增加负担呢？原因很简单：想看看是否能用一种和别人不同的方法来把 C 语言讲清楚。当然，自负是人类的共性，这本书是否真的能把 C 语言讲清楚，还得靠读者来检验。

C 语言难学吗？来自这个行业的声音始终自相矛盾。一方面，很多过来人声称 C 语言其实很简单；另一方面，很多初学者觉得很难，不得其门而入。仅就语法而言，C 语言确实比较简单。但是，这种简单性使得很多人对它的掌握只停留在似是而非的表面上。似是而非的学习一开始很轻松，但你走不远。实际上，即使是声称已经掌握了这门编程语言的人，对很多语法要素的认识和理解也是错误的，在书写稍微复杂一些的代码时，也发现自己突然变得糊涂起来。

那么，学习 C 语言的诀窍在哪里呢？

首先，掌握它的类型系统并学会以类型的观点来构造和解析程序中的代码，这样你就不会迷路。如果你没有掌握 C 语言的类型系统，不会从类型的角度来分析一个表达式，说明你并没有掌握 C 语言。

其次，你要了解 C 语言在整个计算机系统中的位置，知道它和操作系统或者硬件之间的关系；尤其是要理解库和 C 语言的关系，要明白是库拓展了 C 语言的实用性。除了 C 语言本身的简洁、优美和强大的表达能力外，C 标准库和其他形形色色的库也是 C 语言变得流行并威力无穷的重要因素。

写一本编程语言的通俗入门教材，最痛苦的莫过于你不能一下子展现事物的全貌和众多细节。尽管你知道它，也渴望表达，然而毫不客气地说，读者们并不需要它。读者不了解原委，没有耐心，记不住，而且恐惧。为此，本书力求在以下几个方面做一些突破：

首先，C 语言的知识点是网状的，是互相牵扯和交叉的，如果不加梳理，随着阅读的深入，读者不理解的概念和术语将越来越多，从而产生挫败感。为了克服这一问题，我们把它变成线性的，还没讲到的内容一概不提，没讲过的概念一概不用；讲过了，有印象了，掌握了，再用来解释新的知识。

其次，对于一本 C 语言的图书或者教材来说，最怕的是陷于细节而无法自拔，这往往会使读者成为语法机器而不能领略 C 语言的全貌，不知道 C 语言到底有什么实际的用处，更不知道哪些知识才是最重要的。

因此，在内容的组织上，这本书的宗旨是先观其大略，而不是一上来就究其细节。如果只是陈列各种语法元素及其细节，我们可能需要写几百上千页，各种示例和习题堆砌其中。学生学完了，习题也做完了，还是莫名其妙，不知道 C 到底在整个计算机体系结构中处于什么位置，它和操作系统的关系是什么，也不知道为什么别人的程序可以播放音乐、处理图片、过滤网络数据包，而自己的程序只能打印哪些学生的成绩高于 60 分。

第三，考虑到类型系统的重要性，从本书一开始就逐渐强化类型的知识和基于类型的语法分析。这是掌握 C 语言的关键，不可等闲视之。

第四，多数教材和图书都从一个令初学者抓狂的语句

```
printf ("hello world");
```

开始，其理由是初学者可以马上看到“成果”，增加他们的学习兴趣。然而，除非是面向有编程经验的读者，否则这样做可能弊多利少。一方面，printf 只是一个普通的输入输出函数，而且并不是 C 语言的组成部分，但初学者可能会先入为主地认为它就是 C 语言里的大梁；另一方面，这个函数并不是它表面上看起来的那样简单，实际上涉及多个知识点和

概念，而且无法在一本书的开始部分完全展开。对于初学者来说，从一开始就将他们引入一个迷局可能并不值得。

考虑到这些，本书一反常态，将输入输出留到第六章单独讲述。在此之前，我们用调试软件来跟踪程序的执行过程并观察执行结果。事实上，程序调试是非常重要的技能，所以本书这样安排应该是科学的。

第五，本书有一章是专门介绍 WINDOWS 编程的，虽然如走马观花一样简单，也会令某些读者质疑，毕竟 C 语言无关具体的硬件和操作系统平台。之所以这样安排，主要的目的是让读者领略 C 语言是如何在具体的平台上发挥作用的，以及库在这个过程中所起到的作用和扮演的角色，并从一个侧面解答 C 语言到底有什么用的问题。尽管 C 语言不依赖于平台，但用 C 语言写出来的程序却需要在具体的平台上执行。

第六，本书引入了很多概念和术语，但在正文中夹杂这些术语的英文拼写可能会对部分读者造成阅读障碍。考虑到这一点，我们在每一章的前面用思维导图单独列出，这样做的另一个好处是可以让读者清楚地知道本章中都讲了哪些内容。

第七，这本书开篇没有讲 C 语言的由来、历史、优点和应用领域，通常来说，这是一本 C 语言教材的格式化组成部分。但考虑到别的书都已经讲了，网络上也到处都有，所以我就没必要再啰唆了，请大家不要见怪。

最后，这不是一本类似于辞典或者语法参考手册之类的书，内容的组织具有渐进和逐步展开的特点，应该从第一章开始顺序阅读。在学习这门编程语言之前，必须先了解计算机的工作原理，有使用计算机的经验。对于大学新生来说，我并不担心这一点，学校自有他们的教学计划和进度安排；对于自学这门编程语言的人来说，这是需要注意的。

纸质书的内容承载力有限，为了帮助大家更好地使用本书和理解书中的内容，我会准备一些辅助的学习资料，比如导读和习题解析之类的文档，它们都存放在我的个人网站上，网站的地址就在下面。当然，如果你有什么意见和建议，也可以在网站上留言，或者通过下面的电子邮件地址与我联系。

在即将出版之前，编辑同学希望我能在前言里提一提我以前写过的书。说白了，就是要做做广告。我当时就大义凛然地一口回绝：“此事决不可为！谦虚谨慎乃做人之根本，休想让朕把写过《穿越计算机的迷雾》和《x86 汇编语言：从实模式到保护模式》这两本书的事说出来！”

王晓波和李双圆参与了本书的写作，我们在此共同祝愿读者们阅读愉快，早日通过本书掌握 C 语言的精髓。

李忠

2019 年 1 月 6 日于长春

网站：http://www.lizhongc.com/

电邮：leechung@126.com

目　　录

第 4 章　指针不是指南针

第 5 章　准备显示累加结果

第 6 章　输入和输出

第 7 章　字符集和字符编码

第 8 章 欢迎来到类型之家

第 9 章 作用域、链接、线程和存储期

第 10 章 WINDOWS 编程基础


从1加到100
存储器（storage）
存储区(region of storage)
引入变量(variable)的概念
值(value)
翻译器(translator)
翻译(translate)
将写的程序文本翻译为处理器可识别和执行的机器指令
声明(declaration)
类型指定符(type specifier)
int
关键字(keyword)
标识符(identifier)
引入空白字符(white character)的概念
标准整数类型(standard integer type)
标准有符号整数类型
signed char、signed short int、signed int、signed long int和signed long long int
标准无符号整数类型
unsigned char、unsigned short int、unsigned int、unsigned long int、unsigned long long int和_Bool
语句(statement)
表达式语句(expression statement)
循环语句(iteration statement)
while语句(while statement)
什么是循环体(loop body)
引入复合语句(composite statement)
表达式(expression)
运算符(operator)
引入运算符<=
关系运算符
关系表达式
引入运算符=
赋值运算符
赋值表达式
引入运算符+
加性运算符
加性表达式
运算符的优先级和结合性
操作数(operand)
子表达式(subexpression)
左值(lvalue)
左值转换(lvalue conversion)
强调表达式有值
常量(constant)
常量表达式(constant expression)
表达式的值是运算符的值或者结果
引入值计算(value computation)的概念
什么是控制表达式(controlling expression)
括住的表达式(parenthesized expression)
基本表达式(primary expression)
引入副作用(side effect)的概念
源文件(source file)
函数(function)
参数(parameter)
关键字void意味着无参数
返回值和返回类型(return type)
返回类型为void意味着不返回任何值
函数体(function body)
什么是函数声明(function declaration)
必须有一个main函数
return语句(return statement)

第 1 章　从 1 加到 100

今次番[1]打开这本书，讲的是一门语言，一门给计算机编程用的语言，这就是大名鼎鼎的 C 语言。

客观地讲，C 语言很容易掌握，但它也不是一点门槛都没有。如果你之前根本没接触过计算机，不懂得它的工作原理，对存储器、处理器（俗称 CPU）、二进制、十六进制，以及计算机如何存储和处理数字（整数和小数）还不了解，那就该先放下本书去补补课再来。

如果你已经学过这些内容，那么我们现在就可以开始了。对于初学者来说，用 C 语言编程就像第一次吃螃蟹，全是硬壳，好吃的肉在哪个地方都不知道，根本无从下手。

所以，我们首先需要了解一下 C 程序的大体结构，然后呢，还要知道使用 C 语言编程的过程和步骤。在这个学习过程中，还要了解 C 的编程环境——这当然不是指你在哪个房间里编程，这间房子装修得怎么样，而是指你的 C 语言程序在什么样的电脑环境中编写，以及它最终在什么样的电脑环境中运行。

所谓“编程”，重点在于“编”，也就是编写。编写程序要在纸上，或者电脑的文字处理器中进行，比如 Windows 记事本就是最简单的文字处理器。下面，我们就来看看，要用 C 语言编写程序，需要在纸上或文字处理器中写些什么。

1.1 如何从 1 加到 100

计算机语言是用来解决实际问题的，这里有一个很普通的例子：计算从 1 到 100 的整数累加和。现在让我们看一看，如果用 C 语言来编程解决这个数学问题，应该怎么做。

数学家高斯小时候的做法是 101 乘以 50，因为他发现 1+100=101，2+99=101，共有 50 对这样的组合。现在的计算机还缺乏这种自主的思考和分析能力，因此，和高斯的灵机一动不同，我们只能通过编写程序，让计算机老老实实地从 1 加到 100。办法很笨，但是计算机的速度比高斯的大脑和手要快得多得多。

在本书中，存储器特指处理器内部的寄存器、高速缓存或处理器可以直接访问的存储芯片（也就是我们通常所说的内存），除非明确指出，存储器并不包括外部的辅助存储器（比如硬盘和 U 盘）。存储器由字节单元组成，存储器里的一个字节单元，或者，多个连续的字节单元合起来形成一个更大的单元，称为存储区。

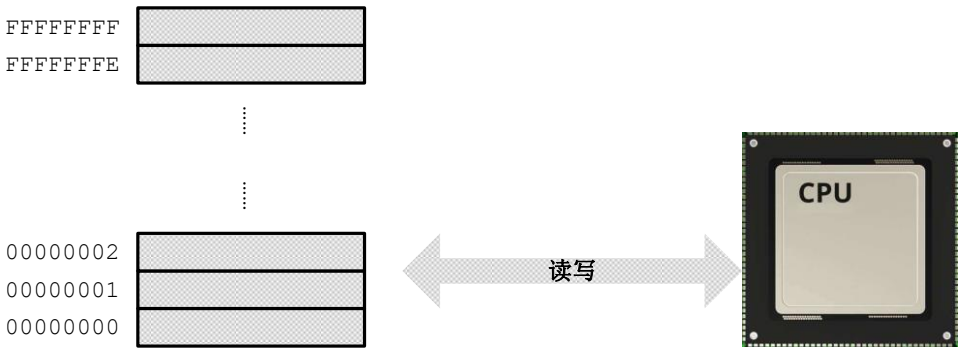


Fig.1-1 内存的组织和处理器

如图 1-1 所示，内存储器由字节单元组成，所有字节单元都按顺序编号，每个字节单元的编号就是它的地址。图中的地址是用十六进制表示的，第一个字节单元的编号是 0；第二个字节单元的编号是 1，其他依次类推。显然，图中的这个内存储器很大，其最后一个字节单元的地址是 FFFFFFFF。

处理器通过地址线和数据线同内存储器相连，如果是按字节访问，则每个地址对应一个

[1] 这回，今几个，这次。

字节单元；如果是按长度不同的字来访问，则一个地址将对应 2 个、4 个或 8 个连续的字节单元。内存储器里存放了数据和机器指令，处理器可以从内存储器里取得指令加以执行，也可以读取内存储器里的数据，或者把数据写入内存储器。

我们知道，处理器可以读写存储区，可以用存储区里的内容进行算术或者逻辑操作，可以自动执行程序指令，这都是处理器可以完成的基本动作。因为处理器只会做有限的基本动作，因此，所谓的编程就是组合这些基本的动作，以解决复杂的问题。

比如说，处理器可以从存储区读取数字，也可以把数字写入存储区，还可以做加法和减法，所以，我们可以把这 100 个数字放到存储区里，第 1 个存储区放数字 1，第 2 个存储区放数字 2……第 100 个存储区放数字 100。然后，编写程序，让处理器自动地按顺序读这些存储区，并把存储区里的数字累加起来。

老实说，这样做没问题，但是很愚蠢，既体现不出计算机的价值，也给编程这门职业抹黑。所以得有巧妙的办法，既节省人力，也能让计算机自动进行操作。最终，我们认为可以让计算机自动按以下步骤来解决问题：

1. 在存储器里找两个空闲的存储区，第一个存放数值 1，第二个存放数值 0；
2. 将两个存储区里的数值相加，结果放回第二个存储区；
3. 将第一个存储区的数值取出，加 1 后再存回；
4. 如果第一个存储区的数值小于等于 100 则转到步骤 2；否则停止。

注意，上述过程不需要人工干预即能自动执行，当它最终停止后，第二个存储区里的内容就是从 1 加到 100 的结果。现在的问题是，如何写一个程序来命令计算机自动重复地做上述工作呢？嗯，这就要依靠程序设计了，也就是写程序，或者叫编程。

我们是说人话的，而计算机只能依靠电信号组成的机器指令工作。早先的时候，只有理解机器指令的程序员才能做编程工作，此时的编程是人类用机器的语言说话。

机器语言是 0 和 1 的组合，处理器执行起来干脆利落，快如闪电——不，比闪电还要快无数倍，但对人类来说非常抽象难懂，外行很难快速入门，就算是内行用起来也是烦琐得要命。在这种情况下，最自然的想法就是创造高级一点的编程语言。高级语言的终极目标是让编程像我们平时写文章一样，用自然语言进行，但目前来说还远无法实现，所以我们只能退而求其次，发明一些不那么“高级”的高级语言，这些高级语言比机器语言好懂多了。

用高级语言写的程序只是一些文本，我们能看懂，但计算机看不懂，处理器无法执行，因为那些东西并不是处理器可以识别的机器指令。所以，我们还必须用一个特殊的软件程序，称为翻译器，来将我们写的程序文本翻译成处理器可以执行的机器指令。翻译器也是人类写出来的程序，可以想象，第一个翻译器是用机器语言编写的。

翻译器不是人的大脑，它无法理解人类的自然语言。你可以将翻译器看成学校里用来自动识别答题卡的阅读机，它只能识别具有固定格式的内容。因此，在现阶段用高级语言编程就像在书写一篇具有固定格式的文章。每种高级语言都具有自己的格式，这种格式被称为那种高级语言的语法。现在，你应该明白 C 语言就是高级语言的一种，学习 C 语言，说白了就是为了掌握 C 语言的语法。

写程序和程序的执行既有关系，又很不同。程序在实际执行时，处理器执行这些机器指令，计算机按照程序的指示做各种动作。但是，程序在编写的时候，这一切都还没有发生，还仅仅是在描述那个执行过程。

就拿分配存储区这件事来说，这需要你亲自做出明确的指示。电脑虽然可以做很多事，但它不是你肚子里的蛔虫，你不跟它说，它是不知道你想要什么的。所以，你的 C 语言程序不但要指明整个累加过程如何进行，还得明确地告诉电脑在存储器里分配两个存储区。

在 C 语言里，要求分配存储区的事情是靠声明来完成的。“声明”的意思就是“告诉”、

“宣布”或“通知”，它用来指明某个东西是什么。例如：

```
int obj;
```

高级语言的特点是屏蔽了底层的细节，使你不需要关心这两个存储区在存储器中的具体位置，况且它们被安排到哪里，只有程序运行的时候才能确定下来。但为了能够访问得到它们，还是得有个凭据，就像人的名字。在 C 语言里，使用一个符号来代表、表示，或者说指示那个存储区，这里的 obj 就是一个存储区的名字，或者说它代表着一个存储区。

使用符号当然可以摆脱烦人的存储器位置，但是仅有符号还不够。首先，存储器是按字节划分的，字节是存储器的最小可寻址单位。但是，一个字节能表示的数的大小有限，存储一个数可能需要好几个连续的字节才行。所以问题来了，你这个符号是代表着一个字节的存储区呢，还是几个连续的字节？

除了存储区的大小，还有一个内容的解释问题。当你使用这个符号来访问它对应的存储区时，如何解释它里面的内容呢？

如图 1-2 所示，这是一个字节的存储区。字节的长度缺乏标准定义，但绝大多数计算机系统都支持将它定义为 8 个比特，所以我们的这个存储区也是由 8 个比特组成。

由图中可知，这个存储区的内容为二进制序列“11111111”。但是，它的含义是什么呢？是个小数？整数？还是其他什么东西？

如果它是一个整数，那么，它可能是一个无符号整数，也可能是一个有符号整数。有关这一点，相信大家在学习 C 语言这门课之前都已经有所了解。

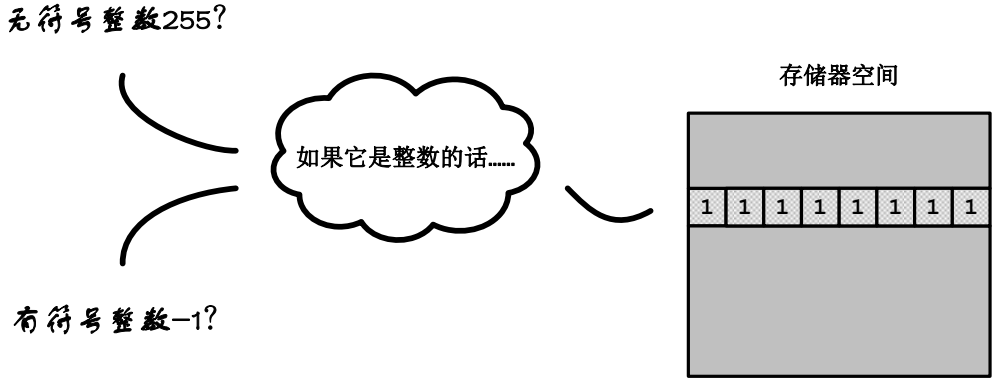


Fig.1-2 存储区的内容可以有多种解释

如果是一个无符号整数，那么，这 8 个比特全都用于表示数值，所以这个二进制序列所对应的十进制数字为 255。

如果是一个有符号整数，那么，在这 8 个比特中，有 1 个用来表示正负，另外 7 个比特才用来表示数值。假定这里采用的是对 2 的补码来表示负数，则这个二进制序列所对应的十进制数字为-1。

以上说的，是我们在读一个存储区时必须考虑的问题；当我们往一个符号所指示的存储区里写数字时，由于无符号数和有符号数的表示方法不同，=所以也同样有这样的问题。

显然，为了分配一个存储区，仅仅声明一个符号是不够的。为此，C 语言要求程序员在声明一个符号时，必须指定它的类型。类型决定了该符号所指示的存储区只能用来读写哪些种类的数据，实际上也就决定了数据以什么样的比特序列存在。

除此之外，C 语言的类型还用来决定存储区的大小。数是无限的，无论存储区有多大，占多少个字节，有些数它依然表示不了，容纳不下；另一方面，如果处理的数都很小，而你又分配了一个特别大的存储区，显然是很浪费的。特别是在计算机发展的早期，存储器的容量很小，在存储空间的利用上用“锱铢必较”来形容毫不过分。

所以，对于上述声明，“int”是类型指定符，用于指定 obj 的类型。在 C 语言里，整

数类型有好多种，而 int 是其中之一。该声明的完整意思是“声明一个符号 obj，它的类型是 int”。实际上，符号是没有类型的，而该符号所指示的存储区也没有类型，它只是用来存储电荷、容纳二进制序列的空间。所以，完整的表述应该是“声明一个符号 obj，该符号所指示的存储区用来容纳 int 类型的数据，要用 int 类型来访问（读和写）”。

一旦把存储区和类型关联起来，那么，就等于约定以后只用这种类型来写入或者读出这个存储区，而且存储区的大小也就确定了。如果你用另一种截然不同的类型来读取或者写入该存储区，将无法保证结果的正确性，也无法预期程序的行为，因为存储区的长度不同，而且存储区的内容用不同的类型来解释将得到不同的值。

每个声明里都会有一些具有固定拼写的部分，在这里是“int”和分号“;”。“int”是 C 语言里的关键字，关键字就是那些具有固定拼写的单词，在 C 语言里有特定的含义和用途，其中的一个功能就是充当线索。因为机器不是人，它没有智慧，所以只能依靠关键字来分析你敲入的内容是什么意思。比如在这里，当翻译器看到有一行的开始部分是“int”，它就知道这应当理解为一个声明，并根据声明的语法继续分析该声明的剩余部分。

和很多别的计算机语言不同，C 语言是区分大小写的，所以关键字也是大小写敏感的，不能将 int 写成 Int 或者 INT，等等。

当然，声明里还有一些程序员可以自主决定的部分，比如这里的“obj”，这在 C 语言里称为标识符。你可以将“obj”改成“i”、“x”和“object”等，都没问题，但你不能使用和关键字相同的符号，所以也不允许出现这样的声明：

```
int int;
```

简直乱套，这像什么话！要知道，关键字是 C 语言语法专属的部分，有固定的意义和用途，不能和标识符冲突。

标识符的拼写可以自由决定，但并不是完全没有限制，而且它也是区分大小写的。比较常规的拼写是使用下划线、26 个英文小写字母、26 个英文大写字母，以及 0 到 9 这十个数字字符，但是不能以数字字符打头。所以，_Myid、store、no001、id_ab 是合法标识符的例子，而 21century、pg dn、go-to、~num 和 cc*^w 都是非法标识符的例子（分别是因为以数字字符打头、中间有空格、使用了“-”、“~”、“*”和“^”这些不允许的字符）。

标识符的最大长度原则上没有限制，唯一的限制来自你所使用的翻译软件，这是一套软件包，用来将你编写的源程序翻译成可执行程序。世界上存在着多种不同的翻译软件，由不同的人和机构编写，他们在标识符长度的问题上并不统一，你认为 31 个字符足够，我认为起码得 128 个。 翻译软件在发行时会提供帮助文档，告诉你如何使用该软件，而且在文档中会给出所允许的标识符长度。如果你无法获取这些信息，那么就请记住，将标识符的长度控制在不超过 31 个字符的范围内一定是安全的，这个长度是 C 语言对翻译软件的最低要求。

到目前为止，我们一直在使用“存储区”这个词，但是用起来不方便。程序在运行时总是要读写数据的——可能是一个非常小的整数，也可能是一个人的完整履历资料。不管它是什么，是一个数字，还是一整块数据，都要在存储器中分配空间来读取和写入。我们先前称之为存储区，但更经常的叫法是“变量”。

要将 C 源程序翻译为可执行程序，需要一套翻译软件，但翻译软件需要根据 C 语言的语法规则来工作，而程序员也需要知道如何用 C 语言写程序，这就需要一个标准化的文档和依据。C 语言刚刚诞生时，它的作者写了一本书，这本书就是事实上的标准。然后，因为这本书太过于简略，很多细节没说清，于是各个翻译软件只能自由发挥，各搞一套。

在这种情况下，C 语言的标准化工作就提上了日程。1989 年，国际标准化组织推出了第一个 C 语言的国际标准 ISO/IEC 9899:1989，简称 C89；1999 年，又推出第二个 C 语言的国际标准 ISO/IEC 9899:1999，简称 C99；最新的一版是 2011 年推出的 C 语言

标准 ISO/IEC 9899：2011，简称 C11。之所以标准一直在更新，是因为很多组织和厂商希望加入新的语言特性。另外，旧标准里有一些不完善的地方也需要加以补充和修改。

在 C 标准文档的正文里，不使用“变量”一词，而代之以“对象”。当然了，这不是谈恋爱时所找的对象，也不是有些面向对象的编程语言（例如 C++和 JAVA）里的对象，不要混为一谈。标准文档之所以避免使用“变量”一词，是因为它是一个已经被滥用，但还缺乏标准定义的词。绝大多数教材根本不加解释就用，有的则解释得非常笼统。

在本书中，我们约定，“变量”的含义和 C 标准文档里的“对象”是等同的，都是指一个存储区，可用来保存值。“值”是计算机操作和加工的对象，是存储在计算机中的、用特定类型来解释的、精度意义上的内容。

一旦我们按照上面的方法声明了符号 obj，这个符号就与它所对应的变量之间建立了关联，如图 1-3 所示。变量只有在程序真正运行时才会分配，但现在还只是在编程阶段，但我们完全可以纸上谈兵，在编写程序时就假定已经有了这个变量，并对它进行并非真实的读写操作，就像它们已经存在一样，这很方便，不是吗？

不过，obj 毕竟不同于它所指示的变量，它仅仅是符号，而变量是在程序运行时才会确定下来的存储区。所以，严格地说，我们只是在编写程序时用一个符号来指示或者表示一个尚不存在的变量，即“obj 所指示的变量”或者“obj 所代表的变量”。有时候，为了方便起见，会直接说“变量 obj”，但你要明白实际上是怎么一回事。

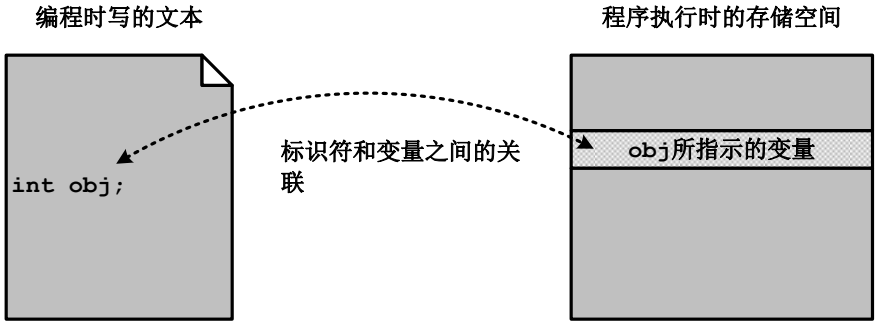


Fig.1-3 标识符和变量之间的对应关系

C 语言对程序的格式并不在意，只要各语法成分能够互相区分开来就行。例如，类型指定符 int 和标识符 obj 一定要用空白字符分开，但标识符 obj 和后面的分号“;”可以连写，因为标识符不能由分号组成，所以能够被识别为不同的东西，但是用空白字符分开也没问题。空白字符包括空格、换行符、制表符（对应于键盘上的 TAB 键），等等。所以你可以这样重写前面的声明：

```
int
obj
;
```

在这里，分隔字符是换行符。为了告诉翻译器，一个声明已经结束，后面的内容不再是当前声明的一部分，这里需要一个结尾标志。对，就是分号“;”。另外，要注意类型指定符 int 和标识符 obj 的相对位置，int 应该在前面。

1.1.1 标准整数类型

回到开头，继续讨论如何计算从 1 加到 100。因为我们需要两个变量，一个存放初始数值“1”，另一个存放每次累加的结果，所以需要声明两个符号（标识符）。假定这两个符号分别是 n 和 sum，则它们可以这样声明：

```
int n;
int sum;
```

问题是，尽管我们已经知道关键字“int”是 C 语言的类型指定符，代表的是一种整数类型，但它的数字表示范围是多少呢？这种类型的变量能容纳从 1 加到 100 的结果吗？

在 C 语言里内置了多种整数类型，它们有固定的名称，也明确地定义了最小的、可以保证的取值范围，称为标准整数类型。

我们已经知道，在计算机内部，整数的存储和运算有两套方法，一种是不考虑它们的符号，数字在存储时没有符号信息，所有比特都用来表示数字的大小；另一种则是将数字区分为正负，数字在存储时带有符号信息，要用 1 个比特来表示符号，其他剩余的比特才真正用于表示数字的大小。因为这个原因，标准整数类型又分为标准有符号整数类型和标准无符号整数类型。

在 C 语言里，标准有符号整数类型包括 signed char、signed short int、signed int、signed long int 和 signed long long int。

在所有 C 语言可以适用的计算机上，signed char 类型的变量要占用 1 个字节的存储空间。尽管“字节”的叫法很普遍，但它的比特长度却从来没有标准定义。对于 C 语言来说，一个字节至少要包含 8 个比特，不能再少了。所以，signed char 类型可以表示的数据范围起码是-127 ~ +127，或者说是$-(2^7-1)$ ~ $+(2^7-1)$。

之所以是 2 的 7 次方而不是 8 次方，是因为不包括符号位。另外，这里给出的只是最小范围，在有些机器上，1 个字节被定义为具有 9 个或者更多的比特（这样的计算机可能很少，但不是没有），在这种机器上，signed char 类型可表示的数据范围会比-127 更小，比+127 更大。

顾名思义，signed short int 经常被叫作“短整型”，这个类型的名字也可以简单地写成 signed short、short int 或者直接写成 short，该类型可以表示的数据范围起码是-32767 ~ +32767，也即$-(2^{15}-1)$~$+(2^{15}-1)$。当然，这只是最低限度，取决于你的计算机，short 类型可表示的数据范围可以比-32767 更小，比+32767 更大。

signed int 可简单地写作 int，或者直接写成 signed，该类型可表示的数据范围起码是-32767 ~ +32767，也即$-(2^{15}-1)$ ~ $+(2^{15}-1)$。当然，这只是最低限度，取决于你的计算机，short 类型可表示的数据范围可以比-32767 更小，比+32767 更大。

signed long int 可简写为 signed long、long int 或者 long，它可以表示的数据范围起码得是-2147483647 ~ +2147483647，也即$-(2^{31}-1)$ ~ $+(2^{31}-1)$。当然，这只是最低限度，取决于你的计算机，long 类型可表示的数据范围可以比-2147483647 更小，比+2147483647 更大。

signed long long int 可简写为 signed long long、long long int 或者直接写成 long long，它可以表示的数据范围起码得是-9223372036854775807 ~ +9223372036854775807，也即$-(2^{63}-1)$ ~ $+(2^{63}-1)$。当然，这只是最低限度，取决于你的计算机，long long 类型可表示的数据范围可以比-9223372036854775807 更小，比+9223372036854775807 更大。

你一定会问，为什么这些整数类型可以表示的数值范围不能固定下来，而仅仅是保证一个最基本的取值范围？这是因为不同的计算机系统具有不同的字长，C 语言的发明者希望整数类型可以弹性地适应具体的计算机系统。比如说 int 类型，如果你还在用老旧的 16 位计算机，你只能用它来表示-32767 ~ +32767 之间的数值；如果你用的是 32 位计算机，它所表示的数据范围可扩大到-2147483647 ~ +2147483647 之间。在没有引入能表示更大整数的类型之前，这样做的好处是显而易见的。

如果仅仅是在同一种类型的计算机上编写、翻译和运行你的 C 语言程序，你完全可以无视标准的限制，自由使用计算机硬件和翻译器支持的取值范围。但是，你要想让自己的程序能够跑在不同的机器上，就必须遵守这个最低限制，必要时可以使用表示范围更大的其他

整数类型。

既然有标准有符号整数类型，那自然也有标准无符号整数类型。在 C 语言里，标准无符号整数类型包括 unsigned char、unsigned short int、unsigned int、unsigned long int 和 unsigned long long int，它们与相对应的有符号整数类型相比，占用的存储空间相同。所有无符号整数类型可表示的最小值是 0，最大值因具体类型而异。

unsigned char 类型可表示的最大值起码是 255（2^8-1）。取决于字节的长度，可以比这个值更大。

unsigned short int 可简写为 unsigned short，该类型可表示的最大值起码是 65535（2^{16}-1）。实际的取值范围可以比它更大，但不能再小。

unsigned int 可简写为 unsigned，该类型的最大值起码是 65535（2^{16}-1），实际的取值范围可以比 65535 更大，但不能再小。

unsigned long int 可简写为 unsigned long，该类型的最大值起码是 4294967295（2^{32}-1）。实际的取值范围可以比它更大，但不能再小。

unsigned long long int 可简写为 unsigned long long，该类型的最大值起码是 18446744073709551615（2^{64}-1）。实际的取值范围可以比它更大，但不能再小。

标准无符号整数类型里还有一个_Bool，它是最近才引入的，以前没有，而且没有对应的有符号类型。它只能用来表示两个数字：0 和 1。

我们已经知道从 1 加到 100 的结果是 5050，这个数值，除了_Bool、signed char 和 unsigned char，其他标准整数类型都足以表示。但是，为了讲解的连贯性，我们使用取值范围最宽的 unsigned long long int 类型来声明前面的 n 和 sum：

```
unsigned long long int n;
unsigned long long int sum;
```

这样声明当然没有任何问题，但有点啰唆，毕竟 C 语言允许我们一次性声明多个符号，就像这样：

```
unsigned long long int n, sum;
```

很明显地，如果要一次性声明多个标识符，类型指定符出现一次即可，但各个标识符之间必须用逗号“,”分开。

练习 1.1

1. C 语言的标准整数类型包括哪两大类？这两大类又各自包括哪些具体的类型？
2. 以下 C 语言的声明中，正确的是（ ）

   (A)char c;　(B)signed char c　(C)c:char;　(D)c char;
3. 当我们说“变量 m”的时候，实际上说的是“标识符 m 所指示的变量”，对吗？若变量 m 的类型是 signed long int，请写出它的声明。
4. 在 C 语言中，“int”属于（　　）

   A.类型指定符　B.关键字　C.所有声明的一部分　D.整数类型
5. 在以下声明中，类型指定符是（　　）；标识符是（　　）；关键字是（　　）；（　　）指示变量。

   ```
   int speed, width, height;
   ```
6. C 语言中所指的变量可以位于（　　）

   A.内存　B.寄存器　C.硬盘　D.U 盘
7. 变量是计算机存储器中的（　　）

   A.存储区　B.数据　C.电信号　D.字节

1.2 相加过程的实现

以上，我们已经声明了两个标识符 n 和 sum，它们各自指示或者说代表一个变量，但这两个变量只有在程序运行时才会从存储器里分配。

因为我们是要从 1 加到 100，所以，变量 n 的初始存储值应当为 1（以后在此基础上递增即可），而变量 sum 的初始存储值应当为 0。既然我们是在纸上谈兵，那么，尽管我们还是在写程序，变量还没有分配，也可以在程序中表达这样的意图：在程序运行的时候修改变量 n 和 sum 的存储值，让它们分别为 1 和 0。

这当然是可以的，在 C 语言里，这种事情需要用语句来完成。在生活中，语句是一个能够表达完整意思的句子；C 语言借用了这个术语，用来描述程序在实际执行时应当完成的动作。例如，往变量里存储一个数值，这就是一个动作。为了向变量 n 写数值 1，往变量 sum 写数值 0，可以使用下面两条语句：

```
n = 1;
sum = 0;
```

注意，你数学学得再好，也不可以把它们理解为“n 等于 1”和“sum 等于 0”，因为这并不是在解方程，而 n 和 sum 也不是未知数。

实际上，如图 1-4 所示，第一条语句的意思是“在程序实际运行时，往变量 n 里写入数值 1”；第二条语句的意思是“在程序实际运行时，往变量 sum 里写入数值 0”。

注意，这两条语句仅仅是在用文本来“表达”一个动作，只有当程序真正在电脑中运行时，这个动作才可能实实在在地执行。用文本描述程序在运行时将要实际执行的动作，这就是编程的本质。

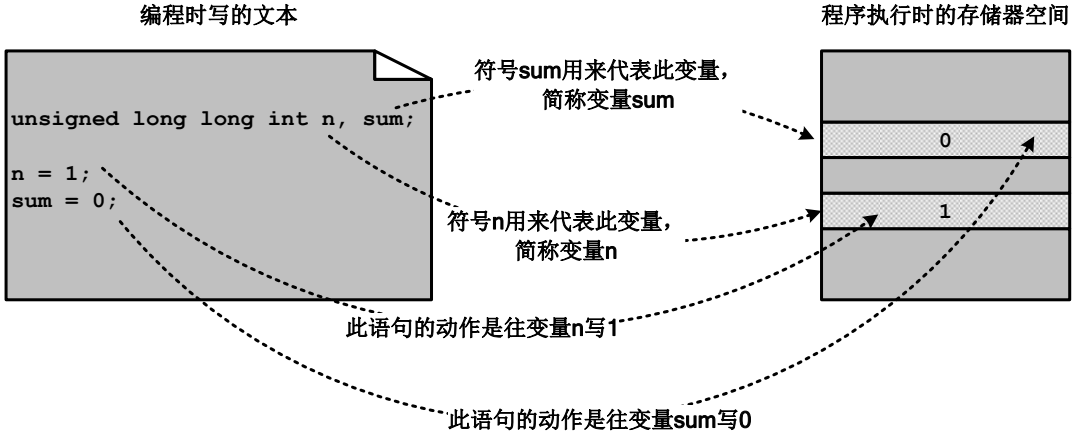


Fig.1-4 语句用于描述程序运行时的动作

问题在于，语句是如何构建的呢？符号“=”是什么意思？而“n = 1”又是什么意思呢？为什么它的后面要加个“;”呢？

在 C 语言里，语句用于指示一个在程序执行时应当完成的动作——可以是算术或者逻辑计算动作，也可以是改变程序执行流程的动作，比如有选择地执行程序的某一部分，或者重复执行程序的某一部分。完成不同的动作需要使用不同的语句，就像写文章时要用到陈述句、疑问句和感叹句一样。

生活中，文章里的语句虽然是一个整体，但它是由字、词组成。在 C 语言里，语句也是一个粗粒度的语法单位，也是由不同的成分组成。如果语句指示的是算术或者逻辑计算动作，那么，它应当由表达式和一个分号“;”组成，称为表达式语句，表达式决定了执行的是什么样的算术和逻辑计算。

典型地，上面那两条语句就是表达式语句，如果去掉这两条语句末尾的分号“;”，剩

下的部分，也就是“n = 1”和“sum = 0”，就是表达式。

表达式是 C 语言的重要语法成分，用来表达不同的计算意图，是由运算符及其操作数组成的序列，例如 1 + 2。在表达式里，运算符指明了要进行何种运算和操作，而操作数则是运算符操作的对象。

因此，对于 1 + 2 这个例子，“+”是运算符，而 1 和 2 是操作数，这个表达式所表达的意图是把 1 和 2 加起来，得到 3 这个结果。

回到表达式 n = 1 和 sum = 0，这里的“=”也是运算符，而 n、1、sum 和 0 都是操作数。不要以为 0 和 1 这类的数字才是操作数，这样理解太片面了。操作数可以是数字，但也可以是指示某个实体的符号，比如代表变量的标识符。

运算符“=”不是数学课里的“等于”，它的真实意思是保存、存储或者“赋予”，这在编程语言里叫作赋值，所以该运算符称为赋值运算符。实际上，应该把它换成“<-”才更形象更直观，但是没办法，C 语言的发明者选择了“=”，我们只能接受这个现实。

运算符“=”需要一左一右两个操作数，而且左操作数必须代表一个变量，右操作数必须计算出一个数值。该运算符所指定的操作是将右边那个操作数的值赋给（存储到）左边那个操作数所指示和代表的变量。正是因为这个，像 n = 1 和 sum = 0 这样的表达式称为赋值表达式。

尽管每个表达式都已经清楚地“表达”了自己所要进行的运算和操作，但是很遗憾，它们无法独立存在于程序中，而只能是某些声明和语句的组成部分。因此，如果一个语句是由表达式和末尾的分号“;”组成，则此语句称为表达式语句。但是反过来，C 语言里的语句种类很多，并不都是表达式语句，很快你就会看到各种各样的语句。

大的、复杂的表达式都是由小的、简单的表达式组成的，组成大表达式的小表达式称为子表达式。比如表达式 n = 1 里的 n 和 1 既是运算符=的操作数，又是子表达式；再比如表达式 sum = 0 里的 sum 和 0 既是运算符=的操作数，也是子表达式。显然，很多表达式并不包含运算符。尽管我们说表达式是运算符和操作数组成的序列，但是如果一个表达式里没有运算符也不是什么大不了的事。

说了这么多，不能再说了，再说你就记不住了，甚至会糊涂。为了不让你糊涂，我们帮你理一理，总结一下刚才都讲了什么：

- ✓ 在 C 语言里，语句用于指定程序运行时应当执行的动作；
- ✓ 有多种类型的语句，表达式语句是其中的一类；
- ✓ 表达式语句由表达式和末尾的分号“;”组成；
- ✓ 表达式是由子表达式通过运算符连接而成，它们被视为运算符的操作数；
- ✓ 有各种不同类型的表达式，赋值表达式是其中的一类。

不管你是在纸上写程序，还是在文字处理器中写程序，现在，我们的程序清单中已经有了以下这些内容：

```
unsigned long long int n, sum;
n = 1;
sum = 0;
```

这段代码声明了标识符 n 和 sum 并分别用两条语句给运算符=的左操作数 n 和 sum 赋值，它们用来指定当程序实际运行时应当完成的任务和动作：在内存储器里分配变量 n 和 sum，然后把数值 1 存储到变量 n；把数值 0 存储到变量 sum。

既然已经完成了声明和赋值的编码工作，接下来的编程任务就是描述从 1 加到 100 的过程了。怎么加呢？该不会是写 100 行语句，结结实实地从 1 加到 100 吧？如果是这样的话，我们何必大费周章地写程序来做这件事。

既然是编写程序来做这件事，那肯定得有既省事，效率又高的方法，C 语言必定要准备

这样的方法供我们使用。

说得不错，我们知道，语句用于指定程序运行时应当执行的动作，如果是一个表达式语句，那么，它所指定的动作实际上是表达式所描述的算术或者逻辑运算。但是我们已经讲过，语句的动作未必都是表达式所描述的操作，有些动作是表达式“表达”不了的，比如改变程序的执行流程，这些并不是算术或者逻辑运算。举个例子来说，在我们现有的程序里，是先执行语句

```
n = 1;
```

再执行语句

```
sum = 0;
```

而且不会掉转头去重复执行。如果想要改变程序的执行流程，比如重复执行某些语句，这就不是表达式语句所能胜任的了。

好在 C 语言为我们提供了好多种不同类型的语句，比如循环语句，我们可以用循环语句来做这件事。循环语句又可以继续细分为好几种，我们先来介绍 while 语句。如果使用 while 语句，从 1 加到 100 的写法可以是这样的：

```
while (n <= 100)
{
    sum = sum + n;
    n = n + 1;
}
```

看样子 while 语句很复杂呀，但实际上并非如此。while 语句有固定的格式，其语法结构为

while (*表达式*) *语句*

在 C 语言里，写程序就像造句和填字游戏。在这里，正常字体的部分意味着它是固定不变的成分，例如“while”、“(”和“)”，它们具有固定的拼写和相对位置；斜体意味着它只是一个用来占位子的名称，又叫占位符，需要用具体的内容来填充和取代，比如这里的“表达式”和“语句”，它们需要用具体的表达式和语句来代替。显然，while 语句在语法组成上是迭代的，它本身是语句，但还要由其他语句组成（甚至可能是另一个 while 语句）。

英语单词“while”的意思是“当……的时候”，所以，C 语言用它来表示当某个条件成立的时候，重复做某些事。而且呢，因为条件会发生变化，所以每次重复做之前，都要重新检查一下。

在这里，所谓的“条件”是指表达式 n <= 100 的运算结果。根据直觉，循环的前提条件是变量 n 的值小于等于 100，因为我们都认识这个小于等于号“<=”。说得不错，正是如此。

在这里，“<=”是运算符，用于比较两个数值的大小关系，表示“小于等于”，在写这个运算符时，“<”和“=”必须连写而不得分开。

每个表达式所定义的运算和操作都应当合乎逻辑。在赋值表达式 n = 1 中，运算符=的左操作数（子表达式）n 代表一个程序运行时的变量，这是合乎逻辑的，因为只有变量才能容纳一个值。如果是 3 = 1，这就不合法了，你不能把一个数值保存到另一个数值中去。

运算符<=需要一左一右两个操作数，问题是，左操作数 n 代表一个变量，而右操作数 100 是一个数值，变量和数值怎么能比较大小呢？不要着急，且听我慢慢道来。

1.2.1 左值和左值转换

表达式在 C 语言里的用途不单单是描述算术或者逻辑运算，实际上具有多种作用，有的用于指示或者说代表一个程序运行时的变量，例如在表达式 n = 1 和 sum = 0 中，表

达式 n 和 sum 就各自指示或者说代表一个变量。

原则上，指示一个变量的表达式称为左值。因此，在表达式 n = 1、sum = 0 及 n <= 100 中，子表达式 n 和 sum 都是左值。

既然左值也是表达式，那我们直接用“表达式”好了，为什么还要发明一个新的术语呢？原因很简单，很多运算符需要它的操作数是一个代表变量的表达式，例如运算符=要求它的左操作数必须是一个代表变量的表达式。使用术语“左值”可以使我们的描述变得更简洁，例如，“运算符=的左操作数必须是一个左值”。

因为左值代表程序运行时的一个变量，所以它被视为变量的“定位器”。可想而知，运算符=的左操作数必须是一个左值。有鉴于此，要判断一个表达式是否为左值，可依据它是否能够位于运算符=的左侧，而据说这也是它为什么被称为“左值”的原因。

不过，左值并不一定非得位于赋值运算符的左边，实际上，它可以位于表达式的任何位置，因为左值的定义仅仅强调它代表着程序运行时的变量。

表达式 n = 1 是把数值 1 赋给左值 n，在这里，n 维持它左值的身份不变，因为它必须代表一个变量。但是在表达式 n <= 100 里，n 不能再保持它原来的身份，因为我们不能说“用左值 n 和数值 100 做比较”，左值是代表变量的表达式，不是数值，只有数值才能和数值比较。

除非另有指定，如果一个运算符的操作数是个左值，则将它替换为该左值所代表的那个变量的存储值，这称为左值转换。在 C 语言里，左值转换是非常重要的概念。

来看表达式 n <= 100，基于上述规定，因运算符<=的操作数 n 是一个左值，所以必须将它替换为它所代表的那个变量（变量 n）的存储值。这样一来，运算符<=的左右操作数现在都是数值，可以进行比较操作。

在 C 语言里，左值转换是非常普遍的，但也有少数运算符的操作数例外，例如在表达式 n = 1 和 sum = 0 中，左值 n 和 sum 就不存在左值转换而保持它原来的左值属性。因为按照 C 语言的规定，如果一个左值是赋值运算符的左操作数，则不发生左值转换。想想看，如果发生了左值转换，则将出现把一个数值保存到另一个数值的情况，这是荒谬的。至于其他不发生左值转换的特殊情况，我们将在遇到的时候再予以说明。

练习 1.2：

1. 什么是左值？如果声明了一个变量 m，则对于表达式 m = 3，m 是左值吗？3 是左值吗？为什么？
2. 选择题：表达式 num = 26 的意思是（ ），它在程序运行时执行的动作是（ ）。
   (a) 将数值 26 写入变量 num　(b) 将数值 26 赋给左值 num　(c) 变量 num 存储的内容是 26
3. 什么是左值转换？为什么要进行左值转换？

1.2.2 表达式的值

C 语言提供了很多运算符，可组成多种多样的表达式，这些表达式描述了运算符如何作用于操作数并得到一个什么样的结果。例如运算符<=属于关系运算符，它需要一左一右两个操作数，并组成关系表达式。这两个操作数是左值的，要先进行左值转换。

表达式的作用是算术或者逻辑运算，既然是运算，必然得算出一个结果来。因此，每个表达式都可以计算出一个值。对表达式 n <= 100 来说，它由子表达式 n 和 100 组成，子表达式 n 要计算一个值，这个值是左值转换后的值；子表达式 100 也要计算一个值，这个值是数值意义上的 100。

你可能觉得奇怪，这里的 100 本来就是个数值啊，还用得着计算吗？事实上，尽管你一眼就看出它是个数字，但翻译器并不认得它，还需要经过分析和转换。在你编写程序的时

候，你输入的只是三个代表数字的字符“1”“0”“0”，它们组成了符号“100”。在程序翻译期间，翻译器会将它识别为代表整数的常量，并将它从符号转换为真正的数字。换句话说，对常量值的计算是在程序翻译期间完成的，而不是在执行的时候。

所有程序的任务和功能都不外乎是操作数字、加工文本，数字和文本的内容可能是来自变量，但也可能在程序中直接给出，就像当前程序中的 0、1 和 100。这些在程序中直接给出的数字和文本在翻译和执行期间不会改变，也没有什么办法改变，故称之为常量。

当一个常量出现在表达式应该出现的地方时，它也是常量表达式。确切地说，常量表达式是指那些值为常量的表达式。所以 100 是常量表达式，201 也是常量表达式，而 705+201 也是常量表达式，因为它的值在程序翻译期间计算，其结果为 906，也是常量，因为两个常量相加的结果总是常量。

不单单是 n 和 100 需要计算出数值，表达式 n <= 100 作为一个整体，同样要计算出一个结果（值）。对于关系表达式来说，如果相应的关系成立，则表达式的值是 1，否则表达式的值为 0。具体到这个表达式，如果左值 n 经左传转换后的值的确小于或者等于 100，则该表达式的值是 1；否则，结果是 0。

而对于 while 来说，它正需要这个结果。圆括号内的表达式用于控制 while 语句如何执行，是终止循环呢，还是继续下一轮循环，称为控制表达式。while 语句并不关心括号内的表达式是什么，它只关心该表达式的值。

如图 1-5 所示，在第一次进入 while 语句时，以及每次执行完循环体之后，要先计算控制表达式的值。这里所谓的循环体，是指组成 while 语句的“语句”。如果控制表达式的结果（值）是 0，就退出 while 语句；否则，如果控制表达式的结果（值）不是 0，就执行循环体。

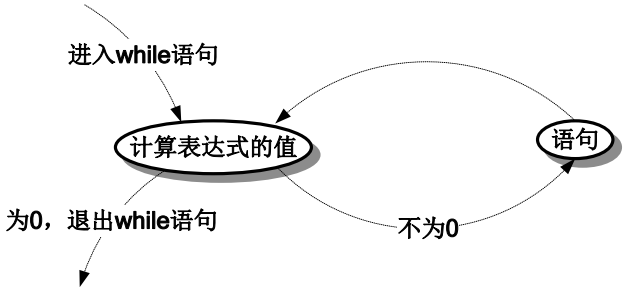


Fig.1-5 while 语句的执行过程

在这里，组成 while 语句的“语句”，也就是循环体，看起来很奇怪，因为它带有一对花括号（注意，这里的缩进并不是必须的，而仅仅是为了看起来直观一些而做的排版）：

```
{
    sum = sum + n;
    n = n + 1;
}
```

组成 while 语句的那个语句称为循环体，是每次循环都要执行的。就语法上来说，循环体是直接位于圆括号“)”之后的那个语句。如果没有花括号，则 while 语句将会是这个样子（同样，这里的缩进是为了看起来直观一些而做的排版）：

```
while (n <= 100)
    sum = sum + n;
n = n + 1;
```

在这种情况下，循环体仅仅是由语句

```
sum = sum + n;
```

组成，而并不包括

```
n = n + 1;
```

也就是说，如果没有花括号，则只有第一条语句是 while 语句的组成部分，而第二条语句不是。也就是说，只有语句

```
sum = sum + n;
```

才会循环执行，而语句

```
n = n + 1;
```

并不会，该语句只在退出 while 语句之后才开始执行。

但是，如果循环体的工作需要多条语句才能完成，那怎么办呢？好在 C 语言为我们准备了另一种语句——复合语句。

复合语句是由一对花括号“{”和“}”，以及可选地，位于这对花括号中间的声明和语句组成。之所以是“可选”，是因为复合语句可以只是一对花括号而中间为空，比如下面这个复合语句，它什么也不做：

```
{}
```

空的复合语句似乎没有什么用，但实际上它很有用。往后你就会看到，有些地方从语法上来说需要一个复合语句，但我们又没什么事可做，只能使之为空。

现在，让我们看看这个 while 语句的循环体都干了些什么。先来看第一个，它实际上是一个表达式语句，由表达式 sum = sum + n 和分号“;”组成。对于刚接触编程的人来说，这令人十分困惑，他们很自然地把这当成一个方程式，并自作聪明地认为 n 的值理应是 0，不然的话两边怎么会相等呢？

然而实际上，这是表达式，而不是方程。同时，正如前面所说，符号“=”并非等号或者等于，而是赋值运算符。这里还出现了另一个运算符“+”，它的功能倒是符合数学里的本义，就是相加的意思，用于把它两边的值加起来求和。

1.2.3 运算符的优先级

表达式 sum = sum + n 是令人迷惑的，因为这里涉及两个运算符，而且中间那个子表达式 sum 夹在这两个运算符之间，它到底是哪个运算符的操作数呢？它究竟跟谁是一伙的呢？

这就涉及运算符的优先级了。运算符的优先级决定了谁才能优先与它旁边的操作数结合，也决定了表达式的类别。因为 C 语言规定运算符“+”的优先级比“=”高，所以中间的那个 sum 是“+”的操作数，而不是“=”的操作数。在这种情况下，运算符“=”的操作数只能是 sum 和 sum + n 的值，所以表达式 sum = sum + n 本质上是一个赋值表达式，其作用是将表达式 sum + n 的值赋予左值 sum。

二元运算符+和-称为加性运算符，“二元运算符”的意思是需要两个操作数。加性运算符用于组成加性表达式，例如 103 + 266 或者 500 - 78。

加性运算符的功能是将左右两个操作数的值相加或者相减，从而得出一个结果。对于上述子表达式 sum + n 来说，sum 和 n 都是左值，两个左值是无法相加的，所以都必须进行左值转换，转换为它们所代表的那个变量的存储值。

相加的结果作为运算符=的右操作数，被赋给左值 sum，这将改变它所代表的那个变量的存储值。如果变量 n 的原值是 1，变量 sum 的原值是 0，则语句

```
sum = sum + n;
```

执行后，变量 n 的值仍旧是 1，而变量 sum 的值则从原来的 0 变为 1。

再来看表达式 n = n + 1，基于相同的原因，运算符+的操作数是 n 和 1，而运算符=

的操作数则是 n 和子表达式 n + 1 的值。

对于子表达式 n + 1 来说，n 是左值，无法用一个左值和 1 相加，所以 n 要进行左值转换，转换为它所指示的变量的存储值。然后，相加的结果作为运算符=的右操作数，被赋给左值 n，这将改变它所代表的那个变量的存储值。如果变量 n 的原值是 1，则语句

```
n = n + 1;
```

执行后，变量 n 的值是 2。

现在，让我们来看一下这个 while 语句的工作过程。为了方便阅读起见，这里列出程序的全部内容（到目前为止）：

```
unsigned long long int n, sum;

n = 1;
sum = 0;

while (n <= 100)
{
    sum = sum + n;
    n = n + 1;
}
```

注意，这里面的空行和缩进并不是必须的，而仅仅是为了方便阅读做的排版，对程序的功能和将来的运行无任何影响。

当 while 语句执行的时候，先要计算表达式 n <= 100 的值，但这又要先计算表达式 n 的值（左值转换），也就是用变量 n 的存储值来代替这里的表达式 n。

接着，用表达式 n 的值和 100 进行比较。因为刚开始的时候，n 的存储值为 1，这个小于等于的关系成立，表达式 n <= 100 的值为 1，所以要执行循环体（复合语句）。

先是执行

```
sum = sum + n;
```

刚开始的时候，变量 sum 的存储值为 0，所以表达式 sum + n 的值为 1。紧接着，这个 1 被写入变量 sum。

接着执行语句

```
n = n + 1;
```

第一次执行的时候，变量 n 的存储值为 1，所以表达式 n + 1 的值为 2，并被写入变量 n，使得它的存储值变为 2。

至此，while 语句的循环体执行完毕。注意，每当程序的执行到达循环体尾部，都将再次回到 while 语句的起始处，重新判断循环条件，也就是重新计算控制表达式 n <= 100 的值。此时，变量 n 的存储值为 2，依然符合小于等于 100 的条件，再次执行循环体。

后面的执行过程都大同小异，每次循环后，变量 n 的存储值都比前一次大 1，而变量 sum 的存储值都会在原先的基础上和变量 n 的值累加，这和我们用手工做是一样的。

最后，变量 n 的存储值会递增到 101，此时，表达式 n <= 100 的值为 0，不再执行 while 的循环体，而是退出 while 语句，继续往后执行。当然，后面的内容尚未给出，但很快就会揭晓。

1.2.4 运算符的结合性

我们说过，如果需要的话，每个表达式都可以计算出一个值。表达式 n <= 100 可以计算出一个值，如果变量 n 的值的确小于等于 100，则该表达式的值为 1，否则为 0；表达

式 sum + n 也可以计算出一个值，这个值是变量 sum 和变量 n 的值相加的结果。

即使是一个赋值表达式，例如表达式 n = 1，也可以计算出一个值，该表达式的值是变量 n 被赋值之后的新值。因为这个原因，我们可以写出这样的表达式语句：

```
sum = n = 0;
```

首先我要说明的是，这个语句里的表达式 sum = n = 0 没有一点问题，是个合法的表达式。棘手的是，这里只有两个运算符，还一模一样，原先的优先级规则不管用了，你不知道中间的 n 到底是哪个=的操作数。

如果运算符的优先级不同，还能分出个高下先后，如果优先级相同，这就得定出个章程来。最好的办法就是按顺序轮流挑选操作数。比如，可以先从最左边的运算符来，由它先挑操作数，然后依次是右边的运算符，这称为“从左往右结合”；或者，也可以反过来，先让最右边的运算符来挑操作数，然后依次是左边的运算符，这称为“从右往左结合”。

这样安排，大家都服气，C 语言就是这么干的。那么，到底是从左往右结合呢，还是从右往左结合？这还不一定。在 C 语言里，有些运算符是从左往右结合的，而有的则是从右往左结合，这称为运算符的结合性。

运算符=是从右往左结合的。所以，在刚才那个例子中，n 和 0 是右边那个运算符=的操作数，左边那个运算符=的操作数则是 sum 和 n = 0 的值。

因此，这个表达式的计算过程是这样的：先计算表达式 n = 0 的值，这个值是变量 n 被赋值之后的新值；接着，这个值又被赋给变量 sum。这个表达式语句执行结束后，变量 sum 和变量 n 的值相同，都是 0。原则上，表达式 sum = n = 0 在整体上也要计算出一个值，但这个值没有什么用。

在 C 语言里，运算符的作用是对操作数进行计算和加工，所以，表达式的值也是运算符“运算”和“加工”的结果。从这个意义上说，**表达式的值也被认为是运算符的值或者运算符的结果**。例如表达式 1 + 2 的结果是 3，我们就认为这个 3 是运算符+的结果，或者说是运算符+的值。再比如，表达式 n = 1 的值也是运算符=的结果；表达式 n <= 100 的值也是运算符<=的结果；表达式 sum + n 的值也是运算符+的结果；而表达式 sum = sum + n 呢，因为这是一个赋值表达式，故该表达式的值是运算符=的结果。

计算表达式的值，被称为表达式的值计算，简称值计算。C 语言规定，**操作数的值计算必须先于运算符的值计算**。比如对于表达式 n <= 100，因为运算符<=的操作数是 n 和 100，所以是先计算操作数 n 的值，也就是先进行左值转换，然后才开始计算运算符<=的值。

再比如表达式 sum = sum + n，因为它是一个赋值表达式，运算符=的操作数是 sum 和 sum + n 的值，所以必须先计算子表达式 sum + n 的值；而对于表达式 sum + n 来说，因为 sum 和 n 是运算符+的操作数，所以必须先要对 sum 和 n 进行左值转换，然后才开始计算运算符+的值。然而，子表达式 sum 和 n 的值谁先计算却没有规定。

每个运算符都有它的优先级和结合性，但是，任由它们自然结合可能会带来麻烦。比如在上学的时候，要计算 5 加上 6 乘以 2 的结果，列出的算式为 5 + 6 × 2。但是，如果我们是希望先将 5 和 6 相加，结果再乘以 2，怎么办呢？老师会教我们使用括号来改变计算顺序：（5 + 6） × 2。

同样，如果我们希望 C 语言里的表达式能够打破运算符固有的优先级和结合性，也得使用括号。这里有个实际的例子，在我们现有的程序中，为了给变量 n 和 sum 赋值，使用了两条表达式语句：

```
n = 1;
sum = 0;
```

但是，我们现在要用一条语句来完成这两条语句的工作，该怎么写呢？答案是将这两条语句替换为下面这条表达式语句：

```
n = (sum = 0) + 1;
```

用一对圆括号括住的表达式，连同这对圆括号一起，被称为括住的表达式。括住的表达式属于 C 语言里的基本表达式，而基本表达式是其他表达式的基本构件。用作表达式的标识符、常量等，都是基本表达式。例如，在表达式 n = (sum = 0) + 1 里，n、sum、0、1 和(sum = 0)都是基本表达式。

在这里，作为一个基本表达式，括住的表达式(sum = 0)要独立地进行计算，并得到一个值，这个值是运算符+的左操作数。显然，这条语句的意思是将常量 0 赋给变量 sum，然后，用表达式 sum = 0 的值与常量 1 相加，结果再赋给左值 n。当这条语句执行之后，变量 sum 的值为 0，变量 n 的值为 1。

截至目前，我们已经知道表达式有多种作用。比如，它可以指示一个变量，也可以计算出一个值。就表达式的作用而言，是将运算符施加于操作数并计算出结果。但是你也看到了，很多表达式不但会计算出一个值，还会改变变量的存储值（甚至改变文件的内容）。例如表达式 n = 1 就改变了变量 n 的存储值。

这样的效果更像表达式在值计算过程中的一个副产品、一个额外的结果，故称之为副作用。因此，我们说表达式可以指示一个变量，也可以计算出一个值，还可以发起一个副作用，这都是表达式的作用。

练习 1.3：

1．给定表达式 a = b + c，请判断下面的说法是否正确：

(a) 运算符+的结果也是表达式 b + c 的值。( )

(b) 运算符=的值也是表达式 a = b + c 的值。( )

(c) 运算符+的优先级比=高。( )

(d) 这是一个赋值表达式。( )

(e) 要计算运算符+的值，必须先计算操作数 b 和 c 的值。( )

(f) 先计算 b 的值，再计算 c 的值，然后计算运算符+的值。( )

2．若 n 是一个 int 类型的变量，且其值为 0，则以下（ ）是表达式；（ ）是语句；表达式 n = 1 + n 的值为（ ）。

(A)n + (B)n + 1 (C)n = (D)n; (E)n + n (F)0 (G)1

3．表达式语句由（ ）和（ ）组成。

4．while 语句由关键字（ ）、位于圆括号中的（ ）、作为循环体的（ ）组成。

(A)分号 (B)while (C)While (D)表达式 (E)语句

5．若 m 是一个 int 类型的变量，则以下（ ）是语句。

(A)m (B)m; (C) m + 1; (D)m = m + 1 + m;

1.3 源文件

和几十年前不同，我们现在的人编写程序，都是面对着显示器，敲着键盘。编程就像和电脑说话，要通过文字符号来表达我们的意图。

实际上，编程更像在电脑上写文章。在电脑上写文章需要打开一个文本编辑器，而为了输入和修改源代码，你同样需要这个东西。

电脑上的软件系统有它自己的生态。单纯的硬件系统是无法工作的，只有软件才能将它们驱动起来，我们编写程序就是为了生成软件。在原始社会，人类只有最简单的工具，但他们可以用简单的工具制造更复杂的工具，就这样工具越来越多，越来越先进。

与此类似，最期的程序员写程序很麻烦，需要使用开关、纸带，但随着软件的积累，开始有了操作系统（例如我们现在常用的 WINDOWS 和 LINUX），同时也产生了很多需要运行

在操作系统上的各种软件程序，包括各种各样的文本编辑工具，既可以让公司文员用来写文章排版，也可以让程序员用来写程序。

可以将程序的文本保存在电脑里，比如保存在硬盘或者 U 盘上。保存的时候，当然得起一个名字，文本编辑器将创建一个以该名字命名的文本文件，该文件包含了你输入的程序文本。

包含 C 程序文本（源代码）的文件称为源文件。按习惯，C 源文件都是以.c 作为扩展名，但这并不是必要的，C 语言并未规定源文件的命名方法，这不是它该管的事。

1.3.1 函数

在前面，我们已经讲过如何编写从 1 加到 100 的代码，如果要将这些内容保存为源文件的话，那么，在很多初学者看来，该文件的内容应该是这样的：

```
unsigned long long int n, sum;

n = 1;
sum = 0;

while (n <= 100)
{
    sum = sum + n;
    n = n + 1;
}
```

作为我们的第一个 C 语言程序，这么编写似乎是很自然的，符合直觉。你看，源文件的内容不就应该是把要做的事一一交代清楚吗？！

但是很遗憾，C 语言有它自己的法则，它的创造者坚持认为语句不能是源文件的直接组成部分，而只能出现在函数体内，就像这样：

```
int main (void)
{
    unsigned long long int n, sum;

    n = 1;
    sum = 0;

    while (n <= 100)
    {
        sum = sum + n;
        n = n + 1;
    }

    return 0;
}
```

不管是在哪里，都会有很多重复性的劳动在等着我们，只不过每次做的时候，初始条件不同。一个比较简单的例子是老师让我们计算圆形的面积，每次求解圆面积的时候，过程都一样，只不过老师给出的半径或者直径不同。

计算机程序也是如此，可以将重复使用的代码组织在一起，形成一个独立的代码块以便

重复使用。这种手段是如此重要、如此有用，以至于所有微处理器都提供了相应的机器指令来调用这种代码块，而所有的编程语言也同样都会支持这种机制和做法。在处理器、机器语言和汇编语言中，这种可重复使用的代码块叫过程或者例程，而在 C 语言里称为函数。

每个函数都有自己的名字，用于指示（代表）那个代码块。如果没有名字，你就无法通过指名道姓来使用它。在上面的这个例子中，main 就是函数的名字，我们可以简单地称之为“main 函数”，或者“函数 main”。

很多函数需要从外部接受一些数据，比如圆的半径、从 1 加到几的“几”，等等，这称为参数，参数在函数名后面的一对圆括号中指定。如果函数不接受任何参数，则括号中的内容应当为关键字“void”。显然，这个 main 函数就不接受任何参数。

你可以把函数看成一支小分队，小分队接受调动出去执行任务，任务完成后还可以带点什么东西回来。类似地，对函数的使用称为函数调用，很多函数可能还要返回数据给它的调用者，但每个函数只能有一个返回值。返回值是有类型的，这个类型需要在函数名字的左侧指定。显然，这个 main 函数返回一个 int 类型的值。为了描述方便，我们把函数返回值的类型称为函数的返回类型。

有些函数只做一些内部操作而并不返回任何值，在这种情况下，它仅仅是简单地返回到调用者。如果一个函数不返回任何值，则函数名左侧的类型应指定为关键字“void”。

函数是可以重复使用的代码块，因此，一个完整的函数还必须包含可执行的代码，称为函数体。如以上代码所示，函数体只能是一个复合语句，由一对花括号“{”和“}”，以及位于花括号里的声明和语句组成。

因此，一个完整的函数包括函数名、参数声明、返回类型声明和函数体。类似于在程序中声明一个变量，在程序中编写一个函数实际上也是声明了一个函数，称为函数声明。

1.3.2 return 语句

函数是可重复使用的代码块，函数在执行完毕后可以返回到它的调用者那里，这个返回动作可以用 return 语句来完成，其语法形式为

return *表达式*可选 ;

return 语句由关键字“return”开始，后面的表达式是可选的。如果函数的返回类型是 void，也就是不返回任何值，则关键字“return”后面不能有表达式，而必须直接跟着一个分号“;”，例如

```
return ;
```

实际上，对于这种不返回值的函数来说，它甚至可以没有 return 语句。在这种情况下，当程序的执行到达组成函数体的右花括号“}”时，自动返回到它的调用者。

相反地，如果函数被声明为具有返回值，则关键字“return”后面必须跟着一个表达式，该表达式为函数的调用者提供返回值。如果函数的返回类型不是 void 且没有 return 语句，则函数返回时，将返回无法确定的随机值。

在上面的代码中，因为 main 函数的返回类型是 int，故它要用 return 语句返回 0 值。

1.3.3 main 函数

函数的名字也是一个标识符，想起什么名字由你决定，是你的自由。但是，如果你的程序要在操作系统里运行（就像所有 WINDOWS 程序和 LINUX 程序一样），源文件中就必须有一个叫作 main 的函数。

操作系统不但为用户操作计算机提供方便，允许我们用键盘或者鼠标来管理计算机、运行各种程序，还管理着各种各样的程序和硬件资源。当我们要运行一个程序（比如一个电子

游戏）时，它要负责把程序从外部存储设备（比如硬盘或者 U 盘）调入内存，并把控制权交给程序。当程序执行结束之后，还要清理程序所占用的资源。

为了能够让操作系统识别、管理程序，控制程序的运行，源文件在翻译之后所生成的机器指令里不但含有与源文件内容相对应的部分，还附加了一些额外的代码，用来初始化程序的运行环境，并在程序结束时做一些清理工作。

因此，如图 1-6 所示，当翻译之后的程序开始执行时，是由操作系统把控制转移到程序的初始化部分，于是处理器就开始取这里的指令并加以执行。这些指令用于完成例行的初始化工作，就像你刚搬进新家，要先布置布置，为过日子做一些准备工作。然后，初始化代码将调用 main 函数。当 main 函数执行完毕，又返回到调用点之后完成清理工作，最后返回操作系统。

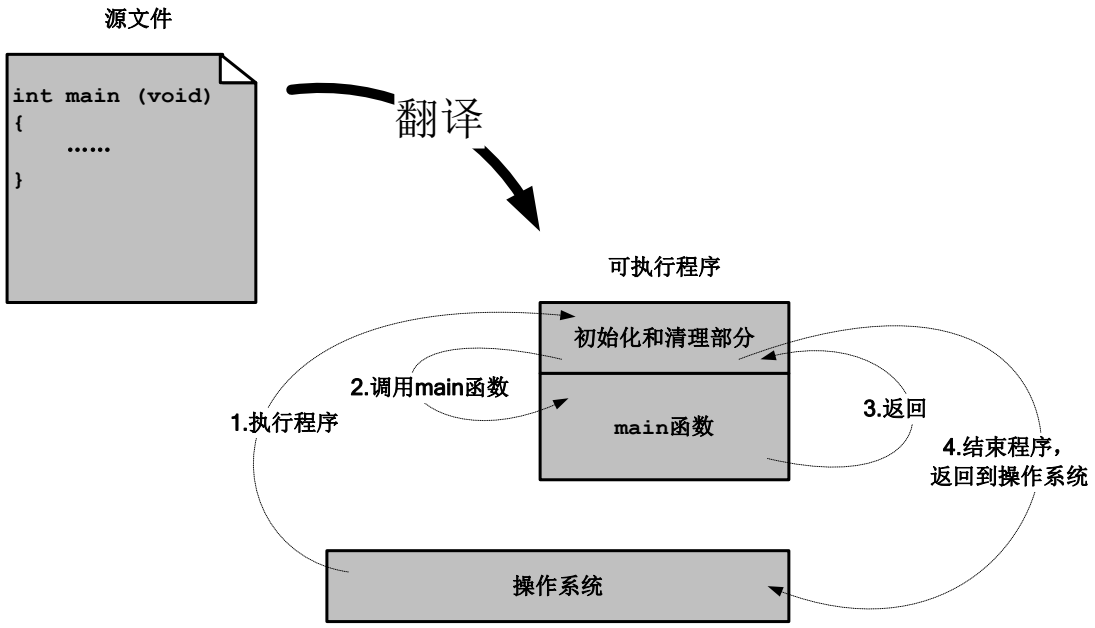


Fig.1-6 源文件的翻译和程序的执行过程

相反，如果翻译后的程序不需要依赖操作系统就能运行，比如你所写的程序本身就是一个操作系统，或者你制造了一个简单的机器人，需要一个软件来控制它，这就不需要操作系统，你的程序直接运行在处理器上，直接控制机器人的各个部件。在这种情况下，程序里不必非得有一个名字为 main 的函数。

尽管“main”不是 C 语言里的关键字，但它的拼写是固定的。很多没有经过严格训练的手容易把它打成“mian”或者“Main”，一定要注意避免。

C 语言规定，函数 main 的返回类型应当是 int 或者与 int 等同的类型，所以它必须返回一个整数值，这个值通常用于指示程序的执行成功与否。按照惯例，返回 0 代表程序正常结束，非零值表示其他含义。在上述 main 函数里，我们是直接返回一个 0 值：

```
return 0;
```

与其他函数不同，从 C99（ISO/IEC 9899:1999）开始规定，如果 main 函数里没有 return 语句，则程序的执行到达组成函数体的右花括号“}”时自动返回，并默认返回数值 0。

函数是将返回值传递给它的调用者。然而，和其他函数不同，main 函数是将值返回到操作系统。

本章的内容到这里就结束了，在下一章里，我们将学习如何将这个程序翻译为可执行程序并观察执行结果。


程序的翻译、执行和调试
C实现(C Implementation)
诊断消息(diagnostic message)
GCC和CLANG
程序的调试
GDB (GNU debuger)
调试命令：b,r,p,n,l,c,q
集成开发环境(IDE：Integrated Development Environment)
程序注释(comments)
多行注释/*.....*/
单行注释// (C99)
函数调用(function call)
形式参数(形参：parameter)
实际参数(实参：argument)
函数指示符(function designator)
函数调用表达式(function call expression)
函数调用运算符()
变量的生存期(life time)
函数定义(function definition)
无函数体的声明中，形参的名字可以省略
函数原型(function prototype)

第 2 章　程序的翻译、执行和调试

经过上一章的学习，我们已经写出了一个 C 语言程序。对于那些第一次学习 C 语言的人来说，这也可能是他们人生中的第一个 C 语言程序，可喜可贺。

现在，不管你用的是 WINDOWS 操作系统，还是 LINUX 操作系统，抑或是别的什么操作系统，都请启动一个你惯用的文本编辑器，输入下面这些程序代码（实际上这是我们在上一章里的成果，为本章阅读方便起见，再次列出），然后保存为源文件 c0201.c。

```
int main (void)
{
    unsigned long long int n, sum;

    n = 1;
    sum = 0;

    while (n <= 100)
    {
        sum = sum + n;
        n = n + 1;
    }

    return 0;
}
```

在本章里，我们的任务是将这个源文件翻译成可执行程序（文件），然后执行它。在这个过程中，我们将学习如何在不同的平台上完成这些操作，以及程序调试的方法和步骤。

2.1 C 实现

源文件里的内容是一些文本，或者说是符号的堆积，这些文本程序员倒是看得明白，但它们并不是机器指令，处理器不认得，所以不能直接提交给处理器执行。因此，我们还必须有一个翻译软件，来把源文件转换为可执行程序。

C 语言的本质是一套语法规则，而一个 C 程序则是按照这些语法规则而编写的文本符号的集合（源文件）。一个翻译软件则反过来，按照既定的语法规则来分析源文件的内容，将它们翻译成最终的机器指令。从这个意义上说，一套完整的翻译软件实际上是 C 语言本身的一个实现，一个现实的化身，简称 C 实现。

历史上，第一个 C 语言翻译程序当然出自该编程语言的发明者之手。1972 年，美国贝尔实验室的 Brian Kernighan 和 Dennis Ritchie 发明了 C 语言，并且不难想象，也正是他们创造了世界上第一个 C 程序翻译软件，或者说实现了 C。

在之后的岁月里，有很多人、很多公司也致力于编写 C 实现，所以现在有大量的 C 实现可用。在本书中，我们仅推荐 GCC 和 CLANG。

和人们熟知的 LINUX 一样，GCC 也是开放源代码的自由软件，而且一直在不断地推出新的版本。GCC 最早的意思是 GNU C Compiler，也即“GNU C 编译器”。GNU 是 1983 年发起的一个软件开源运动，目标是创建一套完全自由的操作系统。它的发起者还成立了自由软件基金会，此时的 GCC 就是一个开源且免费的 C 实现。

到后来，GCC 不单单可以翻译 C 程序，还包括 C++、Objective-C、Fortran、Ada 和 Go，等等，所以 GCC 的意思则变成了 GNU Compiler Collection，也就是“GNU 编

译器集合”。

LLVM CLANG 是 APPLE 公司赞助和主推的项目，现阶段，与 GCC 相比它的翻译速度更快，诊断消息也更加详尽、清晰和明确，这有助于我们更快地找到出错的原因。

问题在于，C 语言之父写的 C 实现还不够用吗？为什么还会有那么多人和企业在不断地编写 C 实现呢？

我们知道，不同的计算机系统有不同的处理器，运行着不同的操作系统。不同的处理器有不同的机器指令集，翻译一个程序，实际上是将源文件的内容转换为那种处理器可识别的机器指令。C 实现是一套软件程序，得有人为那种处理器编写这套软件。正是因为 C 语言广受欢迎，才会有那么多人针对各种不同的处理器编写了大量的 C 实现。

另一方面，C 实现本质上是一套软件，它和别的程序没有什么区别，唯一的区别是别的程序用来听歌、玩游戏、聊天和购物，而 C 实现则用来把源文件变成可执行程序。和别的程序一样，C 实现也要运行在操作系统上。

我们平时在计算机上做事情都要依赖操作系统，取决于个人的习惯和喜好，它可能是 UNIX、LINUX、WINDOWS 或者其他操作系统。操作系统为我们使用计算机提供了一个最基本的平台或者说环境，让你更方便地在计算机上完成各种操作。比如说在 WINDOWS 下，你可以看到有哪些程序和文档，也可以通过键盘和鼠标打开文档、启动程序。

不同的操作系统在外观、操作方式，乃至内部的运作机制上都有差别。运行在操作系统上的可执行程序并不是一个仅仅包含了机器指令的文件，它还夹带了很多操作系统的私货。如果没有这些私货，操作系统将不知道这个程序该如何加载到内存，也不知道怎么去执行它。换句话说，针对不同操作系统而编写和翻译的程序，具有不同的格式，在文件的结构和内容上是不一样的。针对 WINDOWS 编写的程序，无法在 LINUX 上运行，反之亦然，因为它们不能识别对方的某些格式。

所以，运行在不同操作系统上的 C 实现，它本身的可执行文件格式要符合那个操作系统的要求。不过用不着担心，因为 C 是简洁、灵活、高效的计算机语言，经久不衰，深受欢迎，对绝大多数操作系统来说，都有人编写能在它上面工作的 C 实现。

基本以上叙述，我们就很容易理解一个事实：不同的计算机系统具有不同的处理器和操作系统，因而需要有各自不同的 C 实现。比如，GCC 有 LINUX 下的版本和 WINDOWS 下的版本，CLANG 也是如此。

GCC 诞生于 LINUX 操作系统，它们彼此之间的支持和包容性是最好的。而 CLANG 呢，也能运行在 LINUX 上。GCC 的官方网站是 https://gcc.gnu.org/，CLANG 的官方网站是 http://clang.llvm.org/，LINUX 用户可以访问上述链接来了解更多信息并下载和安装它们。

考虑到 WINDOWS 的流行性和用户数量，下面再来说说 WINDOWS 上的情况。WINDOWS 的大部分代码都是用 C 语言写的，但是从那以后，该操作系统的所有者 Microsoft 公司不愿在 C 语言上投入过多的精力。微软公司曾经开发过一款非常流行的产品 Microsoft Visual C++，简称 MSVC 或者 VC，它的 6.0 版本，也即 VC6，直到现在还被国内的大量用户作为学习 C 语言的主要工具。问题在于，它是一个 C++的实现而不是 C 实现，而且它产生于 1998 年，所以充其量仅支持 C 语言的第一个标准 C89，毕竟在名义上 C++是 C 的超集，并且兼容后者。要知道，从那之后 C 语言又经历了两次标准化，分别是 1999 年的 C99 和 2011 年的 C11，经过这两次标准化之后，C 语言已经发生了很大的变化。换句话说，使用 VC6 学习 C 语言并不是明智的选择。

Microsoft 公司现今的编程工具是 Visual Studio 系列，但依然只是针对 C++、C#和 F#等编程语言，而不是 C。

令人稍感快慰的是有很多人依然在努力将 C 移植到 WINDOWS 平台上，所以我们现在可

以使用 GCC 和 CLANG 的移植版本。GCC 在 WINDOWS 操作系统上的移植版本在最早的时候被称为 MinGW，意思是“针对 WINDOWS 的最小化 GNU”。GNU 是 LINUX 上的软件项目，由于平台之间的差异，没有办法完全移植到 WINDOWS 上，只能是移植一个可用的基本部分，这就是所谓的“最小化 GNU”。

当 GCC 更新的时候，MinGW 也会做相应的版本升级，但由于各种原因，这个维护工作在后期变得乏力，而且不能支持 64 位应用程序的开发。于是有好事者在此基础上创建出一个新的分支来，称为 MinGW_W64，该分支可用于生成 32 位或者 64 位的 WINDOWS 应用程序，而且更新较快。

你可以通过 http://mingw-w64.org/了解 MinGW_W64 的更多信息；下载和安装的网络链接是 https://sourceforge.net/projects/mingw-w64/files/。

对于初学者来说，学习 C 语言固然不易，但下载、安装和配置 C 实现却是第一个拦路虎。本书建议使用 GCC 和 CLANG 来完成后面的学习过程，但是，软件随时都在升级，印在书上的详细步骤很快就会过时。因此，考虑到本书的主旨，本书不会详细地介绍它们的下载和安装过程，完整的教程会放在网站上（网站地址在前言的末尾）中供大家参考。

2.2 程序的翻译和执行

假定已经正确安装了 GCC，那么现在的任务就是用它来翻译源文件（程序），但在此之前假定你已经创建了源文件 c0201.c。

以 LINUX 为例，如图 2-1 所示，先用 ls 查看当前目录下的文件，以确保源文件 c0201.c 是存在的；然后，我们用 gcc c0201.c 来翻译这个源文件，如果源文件的内容没有问题（比如拼写错误），则翻译过程会悄无声息地完成。否则，将会在屏幕上出现一些消息（文本），告诉你哪里有什么性质的问题，这称为诊断消息。

GCC 在 UNIX 和 LINUX 上的默认输出是 a.out，所以在翻译后用 ls 查看当前目录下的文件，会发现多了一个 a.out。此时，可以用./a.out 来运行这个可执行文件。

GCC 允许我们使用翻译选项-o 来指定生成的可执行文件名，所以如图中所示，我们可以使用 gcc c0201.c -o c0201.out 来指定要生成的可执行文件为 c0201.out。

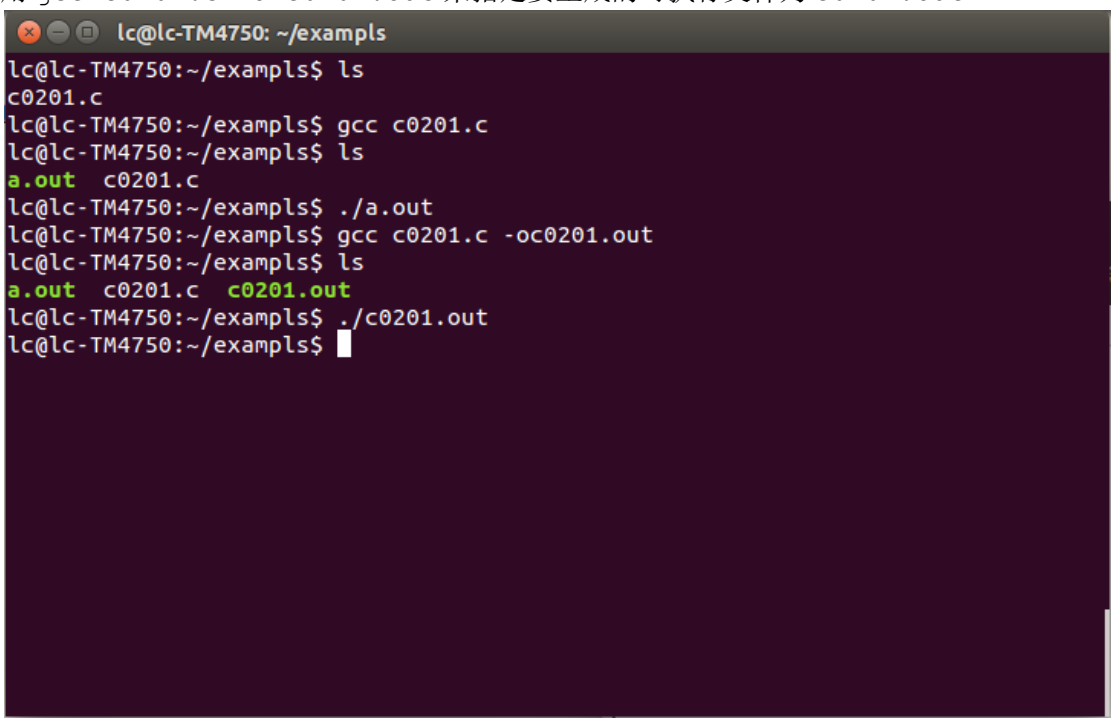


Fig.2-1 LINUX 平台上的翻译和执行过程

再以 WINDOWS 为例，如图 2-2 所示，先用 dir 查看当前目录下的文件，以确保源文件 c0201.c 是存在的；然后，我们用 gcc c0201.c 来翻译这个源文件。

GCC 在 WINDOWS 上的默认输出是 a.exe，所以在翻译后用 dir 查看当前目录下的文件，会发现多了一个 a.exe。此时，可以直接在命令行输入这个可执行文件的名字来运行它。同样地，可使用翻译选项-o 来指定要生成的可执行文件名。

非常明显地，翻译后生成的可执行文件在运行时不会显示累加的结果，这是因为结果保存在变量 sum 中，除非你通过某种方法在屏幕上显示这个变量的内容，否则，它不可能自动显示出来。那么，怎样才能将这个结果显示出来呢？

这是一个既简单又复杂的问题，说它简单，是因为只需要添加两行代码就能解决；说它复杂，是因为要想解释清楚这两行代码的功能，以及其背后的原理，不是三言两语就能说清的，这需要对 C 语言有更多的了解才行。现在就讲这些内容，我怕你会头晕。

所以，本书的前六章以实现文本的打印输出为主线来组织，在完成这一任务的过程中介绍 C 语言及其他相关知识。在实现最终的打印输出之前，我们将用另外的方法来观察程序的执行过程和执行结果。

Fig.2-2 WINDOWS 平台上的翻译和执行过程

2.3 程序的调试

对于计算机系统来说，在屏幕或者其他设备上呈现内容、送出数据，这称为输出；通过键盘、鼠标或者其他设备得到内容和数据，这称为输入。但是，C 语言并没有输入输出的能力，这并不是该语言的组成部分。

要输入或者输出内容，需要借助于操作系统，由它代理，但这并不是一件简单的事情。在最终实现这一功能之前，我们将通过另一种方法来观察程序的计算结果，那就是用调试器来调试一个程序。调试器是一个软件程序，它允许我们以各种方式控制程序的执行，例如单步执行每一条语句，并随时观察执行结果。对于有经验的程序员来说，调试器是必不可少的工具，因为我们很少会写出完全正确的程序，即使它非常简单。在这种情况下，就需要通过调试器来查找产生错误的原因。

在本书中，我们推荐的调试器是 GDB，和 GCC 一样也是 GNU 开源项目的一部分。它的功能十分强大，但在这里无法一一展示，只能小试牛刀。下面以 WINDOWS 平台为例，简单地演示一下调试一个可执行文件的过程。

首先，我们要用 GCC 来翻译 c0201.c，翻译的时候使用选项“-g”，它的目的是向翻译后的可执行程序中添加包括源代码、符号表在内的调试信息，这些额外的内容将有助于 GDB 更好地完成调试工作：

```
D:\examples>gcc c0201.c -o c0201.exe -g
```

这里，以正常字体显示的内容是 WINDOWS 命令行的固有部分，通常为提示符或者调试器的输出内容；以粗体显示的部分是我们输入的命令，下同。

接下来是启动 gdb 并调试刚生成的程序 c0201.exe：

```
D:\examples>gdb c0201.exe -silent
Reading symbols from c0201.exe...done.
```

选项-silent 用于屏蔽 gdb 的前导信息，否则它会先在屏幕上打印一堆免责条款。启动 gdb 后，它输出的信息表明已经读入了 c0201.exe 的符号表。接下来，gdb 会显示自己的提示符“(gdb)”，提示并等待你输入调试命令。

调试一个程序的时候，应该在我们关注的地方，或者在故障点的前边设置一个断点，让程序执行到这里停下来，这样我们就可以慢慢地用别的调试命令进行观察。在 gdb 中，设置断点的方法很多，包括在指定的内存地址处设置断点、在源代码的某一行设置断点，或者在某个函数的入口处设置断点，等等。设置断点的命令是“b”或者“break”，在这里我们是将 main 函数的入口处作为断点：

```
(gdb) b main
Breakpoint 1 at 0x40155d: file c0201.c, line 5.
```

b 命令在执行后返回了断点的具体信息，也就是说，断点（main 函数的入口位置）的内存地址为 0x40155d，对应于源文件的第 5 行（也就是说，main 函数位于源文件的第 5 行）。因此，如果我们用内存地址的方式来设置这个断点，则可以是

```
b * 0x40155d
```

星号“*”意味着是以内存地址作为断点的。或者，如果用源代码行的形式设置这个断点，则可以是

```
b 5
```

一旦设置了断点，下一步就是用“r”或者“run”命令执行被调试的程序，执行后会自动在第一个断点处停下来：

```
(gdb) r
Starting program: D:\examples\c0201.exe
[New Thread 1500.0x1e34]
[New Thread 1500.0x2fb8]

Thread 1 hit Breakpoint 1, main () at c0201.c:5
5           n = 1;
```

在运行了被调试的程序后，GDB 的输出信息显示程序已经启动，下一个将要执行的语句是第 5 行的“n = 1;”。

注意，这条语句并没有执行，而仅仅是告诉你，再继续执行程序的话，执行的语句会是它。

在当前位置，变量 n 和 sum 已经分配，但并没有开始赋值。此时，这两个变量的值会是多少呢？我们可以使用“p”或者“print”命令来分别显示：

```
(gdb) p n
$1 = 16
(gdb) p sum
$2 = 11671024
```

GDB 的 p 命令用于打印一个表达式的值，在这里是表达式 n 和 sum。GDB 先计算表达式的值，并把它保存在一个存储区中，存储区的名字用“$”外加数字来表示，并且这个数字会随着调试过程的进行而不断递增（这意味着存储区也是不断开辟的）。以上，第一个 p 命令执行后，GDB 的回应是$1 = 16，意思是表达式 n 的值保存在$1 中，其内容为 16。

注意，在你的计算机上，变量 n 和 sum 的当前值可能和这里显示的不同。这很好理解，内存是反复使用的，当一个程序终止后，它占用的内存会分配给其他程序使用；当一个变量不再使用后，它占用的内存也会重新分配，并成为另一个变量。因为变量 n 和 sum 刚刚分配，还没有往里面保存任何数值，故它们的内容是随机的，是其他程序或者变量用过的垃圾值。

顺便说一下，既然$1 是 GDB 用于保存计算结果的内部存储区的名字，那么我们也可以用 p 命令来打印它：

```
(gdb) p $1
$3 = 16
```

下面，我们将通过单步执行程序，来看一看变量 n 和 sum 赋值后的值。调试命令“n”或者“next”用于继续执行源文件中的下一行。

```
(gdb) n
6           sum = 0;
```

执行“n”命令后，实际执行的是第 5 行“n = 1;”，GDB 显示下一个即将执行的源代码行，也就是第 6 行的“sum = 0;”。

因为此时已经往变量 n 写入了 1，所以我们可继续用 p 命令来观察它现在的存储值：

```
(gdb) p n
$4 = 1
```

显然，经赋值后，变量 n 的值已经变成 1。

继续执行下一条指令，实际执行的是第 6 行“sum = 0”。执行后，GDB 停下并显示下一条即将执行的源代码行，也即第 8 行的“while (n <= 100)”，第 7 行为空行，所以直接跳过了：

```
(gdb) n
8           while (n <= 100)
```

刚才执行的语句是往变量 sum 保存数值 0，故我们可以再次用 p 命令来观察变量 sum 现在的存储值，可发现它已经变成 0：

```
(gdb) p sum
$5 = 0
```

继续用 n 命令执行下一个源代码行，则将计算 while 语句的控制表达式，并根据该表达式的值决定是否进入循环体，执行后 GDB 显示下一条即将执行的源代码行是第 10 行：

```
(gdb) n
10              sum = sum + n;
```

进入循环体之后，我们想再看看变量 n 和 sum 的当前值。但这次使用 p 命令的方法不一样，这次是用花括号将表达式 n 和 sum 围住以形成一个集合。GDB 允许用这种方式来一次性地打印多个表达式的值：

```
(gdb) p {n, sum}
```

```
$6 = {1, 0}
```

显然，变量 n 和 sum 此时的值依然分别为 1 和 0。继续用 n 命令执行第 10 行，执行后 GDB 停留在即将执行的第 11 行：

```
(gdb) n
11              n = n + 1;
```

注意，第 10 行已经执行完毕，但第 11 行还没有执行。猜猜看，变量 n 和 sum 此时的值是多少？猜测之后，用 p 命令看看结果是否如你所想：

```
(gdb) p {n, sum}
$7 = {1, 1}
```

继续用 n 命令执行下一个源代码行，这将执行第 11 行的“n = n + 1;”，执行后控制又回到了循环的起始处，也即第 8 行：

```
(gdb) n
8           while (n <= 100)
```

此时，变量 n 和 sum 的值各自会是多少？使用 p 命令打印一下就知道了：

```
(gdb) p {n, sum}
$8 = {2, 1}
```

因为现在处于一个循环体内，如果继续用 n 命令往下执行，则其过程与前面相比大同小异。前面已经循环过一次，本次循环完整的调试过程如下：

```
(gdb) n
10              sum = sum + n;
(gdb) n
11              n = n + 1;
(gdb) n
8           while (n <= 100)
(gdb) p {n, sum}
$9 = {3, 3}
```

显然，第二次循环过后，变量 n 的值为 3，变量 sum 的值也是 3。你可能已经发现了，我们现在进退维谷：如果继续用 n 命令执行，则将陷入循环，直到变量 n 的值等于 101。

好在这也算不上什么大的问题，我们可以在循环语句的后面设置断点，然后命令程序一直执行，直至到达这个断点。为了搞清楚 while 语句的下一条语句的行号，我们需要列出源文件的内容，这需要使用“l”或者“list”命令：

```
(gdb) l
3           unsigned long long int n, sum;
4
5           n = 1;
6           sum = 0;
7
8           while (n <= 100)
9           {
10              sum = sum + n;
11              n = n + 1;
12          }
```

l 命令默认每次显示 10 行源代码，但我们关心的那一行显然还没有出来。为此，可继续使用 l 命令来显示后面的行：

```
(gdb) l
13
14          return 0;
15      }
```

好了，我们已经知道 while 语句之后是 return 语句，它的行号是 14，现在就可以用 b 命令设置一个新的断点：

```
(gdb) b 14
Breakpoint 2 at 0x401583: file c0201.c, line 14.
```

现在，可以用一个新的命令“c”或者“continue”来持续执行程序，直至遇到断点或者程序结束。因为已经设置断点，故程序将持续执行，在第 14 行处停下：

```
(gdb) c
Continuing.

Thread 1 hit Breakpoint 2, main () at c0201.c:14
14          return 0;
```

非常好，既然已经退出了 while 循环，说明累加过程已经成功结束，变量 sum 的值就是累加结果。我们来看看它到底是多少：

```
(gdb) p {n, sum}
$10 = {101, 5050}
```

显然，变量 n 的当前值是 101，变量 sum 的当前值是 5050，和高斯同学的结果一模一样。

本次调试即将结束，我们可以先用 c 命令让程序“跑完全程”，然后再用“q”或者“quit”结束本次调试工作，这将使得调试器 GDB 结束运行并返回到操作系统：

```
(gdb) c
Continuing.
[Inferior 1 (process 1500) exited normally]
(gdb) q

D:\examples>
```

即使是对于一个经验非常丰富的程序员来说，在编写程序的时候也避免不了出错。程序中的语法错误通常可以在翻译阶段就能被诊断出来，但逻辑错误却很难被发现和纠正，比如在解决问题时使用了错误或者不完备的方法。在这种情况下，调试器可能是唯一的救命稻草。通过设置适当的断点，你可以观察结果并和预期的结果进行比较以缩小问题代码的范围，并最终发现问题所在。

GDB 是非常强大的工具，它的用法可以写一本厚厚的书，上述调试过程虽然只能说是蜻蜓点水、走马观花，但对于本书后续的讲解来说应该足够了。

练习 2.1：

1. 你可曾想过如何检验关系运算符<=的结果（或者说关系表达式的值）？很简单，将该表达式的值保存到一个变量，在调试器里设置断点并检查变量的值就可以办到。在下面的程序里，第一条语句是将表达式 5 <= 6 的值写入变量 m；第二次是将表达式 33 <= 32 的值写入变量 m。请编辑、保存、翻译并调试这个程序，在第一条语句那里设置断点，然后使用 n 命令和 p 命令观察变量 m 的值如何变化。注意：关系运算符的优先级高于赋值运算符。

   ```
   int main (void)
   ```

```
{
    int m;
    m = 5 <= 6;
    m = 33 <= 32;

    return 0;
}
```

2. 在上一章里我们曾经说过，语句

```
n = 1;
sum = 0;
```

可以合并为一条语句：

```
n = (sum = 0) + 1;
```

请修改源文件 c0201.c，将那两条语句替换为这条语句。然后，翻译并调试新生成的可执行文件，观察这条语句执行后变量 n 和 sum 的值是多少。

2.3 集成开发环境

最早，程序设计的各个阶段都要用不同的工具软件来进行处理，比如要先用文本编辑器来创建源文件，然后用 C 实现来翻译程序并生成可执行文件。如果程序的执行不正确，还要用调试器来分析产生问题的原因，然后从头再来。在这个过程中，程序员必须在几种软件之间来回切换进行操作。

为了方便操作，后来出现了一种“容器”性质的软件，它提供文本创建、编辑、保存、翻译、运行和调试命令，当程序员选择这些命令时，它将调用相应的软件来完成这些操作。这相当于提供了一种方便、友好、简易和一致的操作环境，称为集成开发环境。

集成开发环境是一个软件工具，“集成”表明了它是多种功能的合体，这些功能都在同一个界面下完成。比如，它有一个内置的文本编辑器，允许你创建、编辑、保存源文件；它可以调用翻译器来翻译你的源程序；它可以让你在不用离开该集成环境的情况下就能运行翻译之后的可执行程序并看到结果；如果在翻译的过程中发现了错误，它还可以为你显示这些诊断消息，包括出错的位置和错误原因。最后，它还可以调用调试器来调试你的程序，显示程序运行时的状态，这将有助于你更快地分析和定位程序中的各种问题，而且这一切工作都在同一个集成的、整体的界面环境中完成。

一个集成开发环境的例子是 Code::Blocks，它是一个专为 C、C++和 Fortran 语言定制的集成开发环境，而且是一个开放源代码的自由软件，既有 LINUX 发行版，也有 WINDOWS 发行版，你可以通过网络链接 http://www.codeblocks.org/来了解、下载和安装它。

之所以介绍 Code::Blocks 是因为它很简单，图 2-3 显示了它工作时的界面，源文件的创建、编辑、保存、翻译、运行和调试功能可以通过窗口顶部的菜单和工具栏完成，源文件的编辑工作在窗口的编辑区进行。从图中还可以看出 Code::Blocks 当前正处于调试状态，断点设置在 return 语句所在的那一行，程序已经执行到这里，而且正在等待进一步的指示。悬浮的小窗口内部显示了我们要观察的变量 n 和 sum，它们的当前值分别为 101 和 5050。

很多集成开发环境并不包括翻译器和调试器，因为它们并不是一个集成开发环境的必要组成部分。在这种情况下，你可以单独安装翻译器和调试器软件，然后使它们和集成开发环境建立关联，使得集成开发环境能够调用它们。不过，很多集成开发环境是为特定的程序设计语言而生，会绑定默认的翻译器和调试器，例如微软公司的 Visual Studio 系列。

对 WINDOWS 用户来说，Code::Blocks 提供了好几个版本。有的版本带有翻译器 GCC 和调试器 GDB，但我们建议下载安装不带 GCC 和 GDB 的版本，因为它所包含的 GCC 并不是 MinGW_W64，而是原先的 MinGW，而且版本很老。因为不带有 GCC 和 GDB，所以在安装完之后还必须加以配置才能工作。我并不建议使用集成开发环境来完成本书的学习，但如果你想体验一下也是可以的。关于如何安装和配置 Code::Blocks，可以通过网络搜索相应的教程，在我的网站（参见本书前言的最后）里也会有所提及。

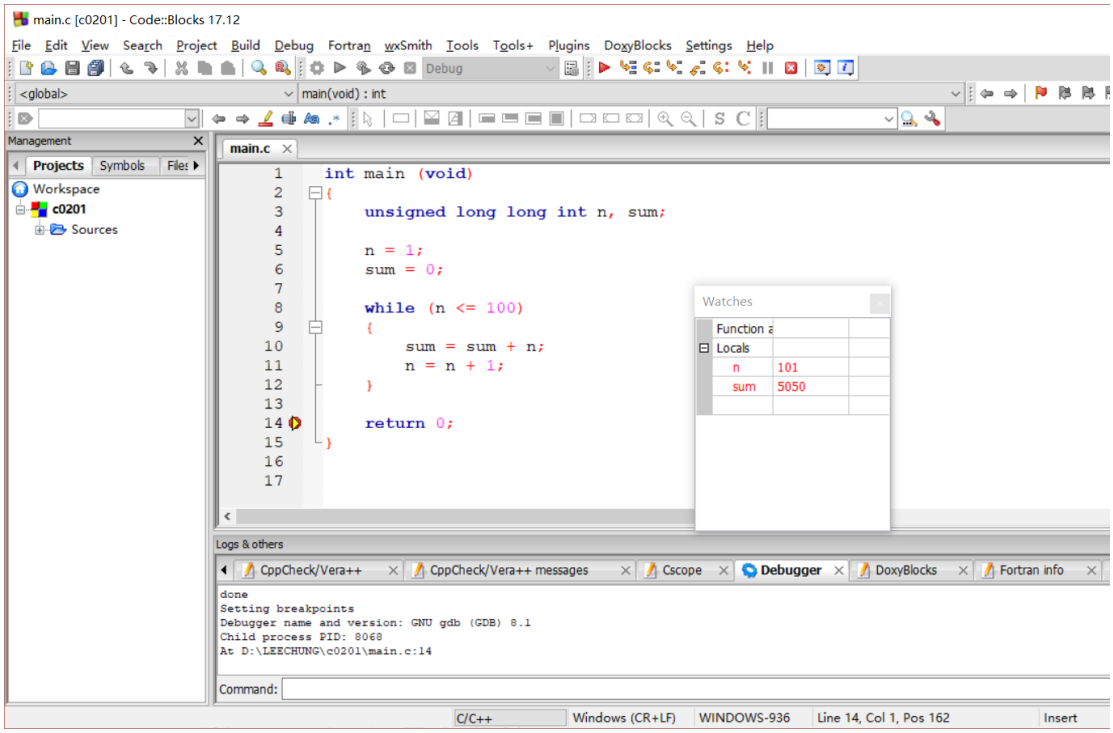


Fig.2-3

2.4 执行环境

我们知道，不同的处理器具有不同的机器指令集，为一种处理器编写的程序在另一种不同的处理器上无法识别和执行，这就解释了为什么用机器语言和汇编语言编写的程序不能运行在另一种截然不同的计算机上。

不过 C 是高级语言，用它编写的程序更像是在用人类的自然语言描述做什么事情、怎么做，而并不直接对应于任何机器指令，所以在翻译一个 C 程序时，需要指明处理器类型或者计算机架构，这样才能有针对性地翻译成机器指令的序列，这也解释了为什么人们经常说 C 是可移植的语言。例如，为了使用英特尔奔腾 4 处理器的机器指令来生成可执行文件，可以用下面的方法来翻译源文件，-march 选项用于指定一个处理器类型：

```
gcc –march=pentium4 c0201.c
```

然而在我们日常的应用中，这个选项通常是不需要的，C 实现会自动应用一个适当的处理器类型。这是因为从本质上说，C 实现并不是什么神奇的东西，它只是普通的可执行程序，只不过它能生成别的可执行程序。

和别的可执行程序一样，每个特定的 C 实现都是针对特定的操作系统而开发的，而每一个操作系统都只运行在特定的处理器上。比如说，GCC 不是“一套”软件，它有不同的版本，每个版本只针对特定的处理器—操作系统组合。大体上，你所选择和安装的版本是与你所用的处理器和操作系统相契合的，而这自然也就成了它翻译源文件时的默认选项。如果程序的编写和运行都在同一台计算机上，这种默认的行为没有什么问题；如果生成的可执行文

件将要运行在别的计算机上，而它使用了另一款迥异的处理器，则必须使用-march 选项来用那种处理器的机器指令生成可执行文件。

除此之外，在翻译一个 C 程序时，还要考虑操作系统的问题。操作系统的作用主要有两个，第一个作用是为人们的日常操作提供便利。拿大家比较熟悉的 WINDOWS 来说，一开机，就出现了桌面和开始菜单，桌面上有程序和文档的图标，菜单里有程序列表。双击桌面上的图标，程序就开始运行，或者文档被打开以供浏览、编辑、通过网络发送或者打印；在程序列表中选择一个程序，那个程序就能开始运行。如果没有操作系统，你将无法完成各种操作，也没办法使用计算机。

操作系统的第二个作用是管理硬件和软件（程序），你能够在 WINDOWS、LINUX 等操作系统里随意地执行任何一个程序，是因为它们都服从操作系统的组织和管理。当你安装一个程序时，由操作系统负责提供磁盘空间——磁盘空间是由操作系统来统一管理的，如果每个程序都不服从统一管理而任意读写磁盘空间，就会乱套，甚至覆盖其他程序的内容；当你运行一个程序时，由操作系统负责将其载入内存中的空闲区域并将处理器的控制权交付于它。物理内存是有限的，且由操作系统管理，操作系统知道哪些位置是空闲的，可以使用。操作系统大都是多任务的，可以同时打开和运行多个程序。如果内存紧张，没有空闲位置，它可以将别的程序移到磁盘上以腾出空间来运行新程序。如果同时运行的程序很多，操作系统还要对它们进行周期性的轮转和调度，好让它们都有机会在处理器上执行，看上去就像所有程序都在同时运行一样。

对于程序员来说，使用操作系统的好处是既方便又省力。在没有操作系统的日子里，程序员非常自由，但这种自由的代价是任何事都要亲力亲为，任务很繁重。首先，电脑由很多硬件组成，他需要自己编写代码来管理和使用那些硬件。比如，他需要亲自将文档的内容转换成适合打印机的格式，并编写代码来驱动打印机工作；他需要亲自安排文件在硬盘上的存储形式和存储位置，并编写代码来驱动和访问硬盘。如果他希望电脑在同一时间能运行自己的好几个程序，还必须亲自编写代码来控制哪一个程序暂停，哪一个程序再次投入运行。总之，他要做的事情实在是太多了，这还没有考虑到一个前提条件：如何将编写的程序装入内存，然后让处理器执行？这事情必须得自己想办法来做！

操作系统为程序员们做了大部分的底层工作，它提供了各种设备的驱动和管理功能，这样一来，程序员就不用再编写直接和设备打交道的程序了。如果程序员编写的程序想保存些东西到硬盘，那他可以简单地将要保存的内容提交给操作系统，请求它来完成硬盘控制和数据保存工作。至于保存到哪里，怎么保存，都不重要，只要下次还能读出来就行了。同时，如果有多个程序都在同一时间访问同一个设备，操作系统也能对这些请求进行仲裁和调度，提供排队功能。

当然，也有很多程序不依赖操作系统这类软件就能自主地运行，最典型的就是操作系统本身，以及种类繁多的嵌入式计算机软件，比如智能家电、仪器仪表和工业控制设备内部的控制软件。在这种非通用的电脑上，你要编写的程序通常对硬件有全部的控制权，通常也不借助于其他程序的帮助就能开始运行。

对于 C 程序员来说，你开发的程序需要操作系统或者其他系统软件的支持吗？还是不需要？这是个大问题，有没有操作系统或者其他系统软件的支撑，这是个运行环境，称为执行环境。执行环境是需要在编写程序前就提前规划的。如果你在写程序之前就决定让它运行在操作系统或者其他系统软件之上，那么，该程序的执行环境属于宿主环境；相反，如果你决定让程序能够独立运行，不借助于操作系统或者其他系统软件的帮助，那么，该程序的执行环境属于独立环境。

注意，宿主环境和独立环境与电脑有没有安装操作系统或者其它系统软件无关。即使你的台式 PC 安装了操作系统或者其他系统软件，但是，你的程序在运行时不需要预先启动操

作系统，也不需要操作系统或者其他系统软件的任何支持，那这个程序的执行环境也必须不折不扣地被视为独立环境。

所以，从本质上讲，执行环境并不是指你拥有什么样的电脑，也不是指程序将要运行的电脑有没有操作系统或者其他系统软件，而是指，你决定让程序运行的环境是什么（需不需要操作系统或者其他系统软件的支撑）。

一个我们熟悉的、运行于独立环境下的程序实例是操作系统内核。操作系统不需要借助于任何其他操作系统就能自主运行，但它的确可以用 C 语言来编写。要知道，C 语言的其中一个标签就是“系统开发语言”。

在用 C 语言书写程序时，如果它是针对宿主环境的，那么，源文件中必须有一个名字叫作 main 的函数。但，这是为什么呢？

首先，为了生成一个运行在宿主式环境下的可执行程序，你需要在翻译的时候给出一个选项来告诉 C 实现，生成的可执行程序需要借助于操作系统这样的系统软件才能运行。例如，对于 GCC，这个选项是-fhosted：

```
gcc -fhosted c0201.c
```

当然，这个选项通常是不需要的，因为 GCC 的默认动作是生成宿主式环境的可执行程序。

我们知道，操作系统的功能之一是管理应用程序。当我们在 WINDOWS 下双击一个程序的图标时，操作系统要读取并分析这个程序，为它分配内存空间，加载它，做一些初始化的工作，然后把处理器的控制权交给它。

所以，在这种情况下，C 实现不单单要依据你的源文件来生成相应的机器代码，还要根据操作系统的要求附加一些特定的代码和数据，这样，操作系统就可以根据这些数据知道如何加载这个应用程序，而这些附加的代码则用于执行一个初始化过程，创建一个可以和操作系统通信的、特定的工作环境。一旦初始化完成，这段初始化代码就可以调用函数 main，从而正式开始运行应用程序。所以，对于运行于宿主环境的程序来说，函数 main 其实是指定了一个入口点。“main”是一个约定的名字，C 实现翻译一个程序时，它需要根据这个名字来找到充当入口点的那个函数。

相反地，如果要将 C 的源文件翻译成在独立环境下执行的程序，那么，他可以在运行翻译程序时提供翻译选项，告诉翻译程序，生成的可执行程序必须脱离像操作系统这样的系统软件而独立运行。此时，生成的机器代码比较“纯粹”，与你在 C 源文件中表达的意图一致。除此之外，基本上不包含更多额外的东西。

比如，如果使用 GCC，则-ffreestanding 和–nostartfiles 选项用于指定生成独立式环境的可执行文件：

```
gcc -ffreestanding -nostartfiles c0201.c
```

那么，是不是独立环境下的 C 程序就不能有 main 函数？非也。只不过，如果你希望 C 实现将源文件翻译成在独立环境下运行的程序，那么，翻译程序将不再特殊看待这个 main 函数，而将它视为一个普通的函数。

在前面，我们已经编写了一个能够从 1 加到 100 的程序，这个程序没有任何问题，翻译后可以得到能够在宿主式环境下运行的程序，但是程序运行后不会显示任何结果，计算结果只能在调试器里观察到。

对于 C 语言的初学者来说，不能在屏幕上显示程序的运行结果，会让他们觉得心里空落落的，像少了点什么，不踏实。但是我们已经说了，就我们目前所掌握的知识，尚不足以展开这方面的讨论，还需要再拖一拖，留到后面的章节里揭晓。

实际上，不能在屏幕上显示结果是一件好事，这可以促使大家熟悉调试器，在调试器中

观察程序的行为和执行结果，我们的学生通常缺乏这种训练。程序调试是一门很重要的技术，很多时候，程序中的问题不是靠在屏幕上输出结果来发现的，而必须依靠调试器来观察和分析。

2.5 从 1 加到 N

在上一章里我们讲到了函数，函数是一个可重复使用的代码块，对函数的使用要通过所谓的函数调用来进行。每个函数只做固定的工作，但每次做可能不完全一样。比如计算圆周率，计算方法是固定的，但每次给定的半径却不同，得出的结果自然也不同。

相似地，如果我们不满足于从 1 加到 100，而是想加到 1 000 000 000 以内的任何数，该怎么办呢？这就是函数大显身手的时候了。在下面的程序中，我们将编写一个独立的函数来做到以不变应万变。

```
/*******************c0202.c******************/
unsigned long long int cusum (unsigned long long int r)
{
    unsigned long long int n, sum;

    n = 1;
    sum = 0;

    while (n <= r)
    {
        sum = sum + n;
        n = n + 1;
    }

    return sum;
}

/*从现在开始，为节省篇幅、节约纸张，main 函数一律不再包含末尾的
  return 0;语句，但请确保你的 C 实现支持 C99（噢，很少有不支持的了）。
*/
int main (void)
{
    unsigned long long int x, y, z;

    x = cusum (10);
    y = cusum (100);
    z = cusum (1000);
}   //此函数没有 return 语句，程序执行到此花括号时，如同执行了 return 0;
```

2.5.1 注释

对于初学者来说，初次接触稍微大一点的程序，有些眼花缭乱，这是可以理解的。凡事就怕解释，一解释，就都清楚了。

在这个程序中包含了一些说明性的文字，这部分内容称为注释。虽然说用高级语言编程

有点像说话，但毕竟和说话还差很远，不是那么容易理解。程序写完之后，别人在读的时候不知道你为啥要这样写，都实现了什么功能；即使是你自己，时间一长，也不知道当初为啥要这样写。为了帮助别人理解你的程序，同时也为了有助于自己恢复记忆，就需要在程序中夹带一些说明性的文字，这就是注释。

注释是给人看的，对于实现程序的功能来说并无作用。但是，它夹在程序中，难免会让翻译器误会，以为它是正常的声明和语句。为此，可以将注释的内容夹在“/*”和“*/”之间。当翻译器遇到“/*”的时候，它就把后面的内容当成注释予以忽略；当它遇到“*/”时就知道注释在这里结束。

因为具有开始标记“/*”和结束标记“*/”，这种注释的内容可以超过一行，所以又称多行注释。相反地，还有一种以“//”开始的注释，它只能延续到当前行的行尾，所以又称为单行注释。

多行注释是 C 语言诞生时就支持的形式，而单行注释是从 C99 才开始引入的。在翻译一个程序时，在正式的翻译工作开始前，每个注释都被替换为一个空格字符。

2.5.2 函数调用和函数调用运算符

在这个程序里，除了 main 函数，还有一个叫作 cusum 的函数。我们说过，函数可以从它的调用者那里接收数据以供内部使用，但是，要想接收这些数据，而且能够在函数内部使用，必须依赖于变量。为此，就要在函数名后面的圆括号内声明这些变量以接受调用者传来的数据。作为函数声明的一部分，这些在圆括号内声明的变量称为参数。在这里，所谓的参数就是 r，其类型为 unsigned long long int，简称参数 r。

习惯上，我们把参数 r 叫作形式参数，简称形参。本质上，参数是从外部传递到函数内部的数值，变量 r 只是承载这个数值的中转容器，是形式上的参数，而不是实际的参数，传递的内容（值）才是实际的参数，这就是“形参”这个名称的由来。

函数是可以反复执行的，每调用一次，它就执行一次。每当函数开始它的一次执行时，就会创建圆括号内声明的参数变量；而当函数返回后，这些变量被销毁。所以，每当 cusum 函数开始执行时，就会创建一个变量以接受调用者传递的值，这个变量由标识符 r 指示和代表着，可称之为变量 r，或者参数变量 r。

所谓的“销毁”只是一种形象的说法，变量是一个存储区，没有谁能够毁掉它，它一直就在那里，在存储器或者处理器中。因此，所谓的“销毁”仅仅是指你不能再合法地使用它了，这个存储区域恢复了自由之身，或者又分配给别的用途了。

函数是一个可反复使用的代码块，对它的使用是通过所谓的函数调用进行的，如果函数的返回类型不是 void，则每次调用后还将返回一个数值。

来看 main 函数，我们先是声明了三个 unsigned long long int 类型的变量 x、y 和 z。然后，语句

```
x = cusum (10);
```

的意思是调用函数 cusum 并将数字 10 传递给函数的参数 r，然后，等函数返回后，将返回值保存到变量 x。

当然，这只是大体的说法，对于这条语句我们还必须做更细致的分析。首先，该语句是一个表达式语句，由表达式 x = cusum (10)和一个分号“;”组成。在这个表达式里有一个赋值运算符和一对圆括号，赋值运算符我们认识，但也别把土豆不当干粮，这一对圆括号也是运算符，称为函数调用运算符。

函数调用运算符的优先级高于赋值运算符，所以，这里的 cusum 是函数调用运算符的操作数，而赋值运算符的操作数是左值 x 和表达式 cusum (10)的值。

在函数的声明里，标识符 cusum 的身份是函数的名字，而当它出现在一个表达式应该

出现的位置时，它的身份是表达式，用于指示或者说代表一个函数。在 C 语言里，指示或者代表函数的表达式叫作函数指示符。

显然，如果一个函数指示符的后面是一对圆括号，那它们就合在一起形成一个更大的表达式，称为函数调用表达式。函数指示符是函数调用运算符的左操作数，其他操作数位于圆括号内部，它们是传递给函数的实际参数。

在 C 语言里，每个表达式都有值。表达式 2 + 3 的值是 5，而对于函数调用表达式来说，它的值是函数调用的返回值，这也是函数调用运算符的结果。如果函数的返回类型是 void，则函数返回空值，且函数调用表达式的值也是空值。

对于初学者来说，他们关注的重点往往在函数调用本身，而忽略了函数调用表达式还会有值。对于表达式 x = cusum (10)来说，你应该忘记子表达式 cusum (10)会发起函数调用的这个事实，而把它看成一个黑盒子，这个黑盒子最终会化为一个值，并被保存到变量 x 中。

在 main 函数内的第一条语句里，函数调用表达式 cusum (10)引发函数的第一次执行。当函数 cusum 开始执行时，创建参数变量 r 并将它的存储值修改为调用者传递的参数值 10，然后，变量 r 就可以在函数内部使用了。

我们说过，参数是从外部传递到函数内部的数值，这个数值是由函数的调用者通过函数调用表达式提供，这才是实际上的参数。习惯上，我们把由调用者给出的实际上的参数称为实际参数，简称实参。

后面两条语句

```
y = cusum (100);
z = cusum (1000);
```

执行动作与第一条语句没有本质区别，唯一的区别是传递的实参不同，函数的返回值也不同，读者可以通过调试器 GDB 观察变量 x、y 和 z 的值。

2.5.3 函数原型

现在回到程序开头，将注意力放在函数 cusum 的函数体，它的大部分内容都是我们在上一章里讲过的，变化不大。和从前一样，我们首先声明了两个变量 n 和 sum，然后分别用表达式语句来修改它们的值。和变量 r 一样，变量 n 和 sum 也是在函数 cusum 开始执行时被创建和开辟出来，在函数返回时销毁。当下一次再调用此函数时，又将创建和开辟新的变量 r、n 和 sum。

这就是说，变量 r、n 和 sum 只存在于程序运行过程中的某一段时间。事实就是这样，有些变量的存在时间很长，甚至和整个程序的运行时间一样长，而有些变量的存在时间很短。不管多长多短，变量在程序运行时的存在时间称为变量的生存期。

和上一章相比，while 语句有一点变化，那就是它的控制表达式变成了 n <= r，而不是从前的 n <= 100。所以，现在是用变量 n 的值和变量 r 的值做比较，这要先分别对子表达式 n 和 r 进行左值转换，再对转换后的值进行比较。

在函数体的末尾，return 语句结束当前函数的执行，将控制返回到它的调用者。左值转换是普遍存在的，该语句在执行时，先计算表达式 sum 的值，也就是先进行左值转换，然后向调用者返回转换后的值。

就这样，函数 cusum 结束了它的一次执行。现在回到 main 函数，有三条语句都调用了 cusum 函数，除了传递的实际参数不同外，它们的执行过程其实都差不多。

程序的执行和函数声明的先后次序没有任何关系，在程序中，虽然说 cusum 函数声明在前而 main 函数的声明在后，但是，程序启动后将首先调用 main 函数。

然而，虽然程序的执行顺序和函数的相对位置无关，但却会影响函数调用本身。我们已

经说过，变量必须先声明后使用，函数也是这样，函数的声明必须位于函数调用之前。函数的声明有两种形式，一种是带有函数体的函数声明，一种是不带函数体的函数声明，带有函数体的函数声明称为函数定义。

在源文件 c0202.c 中，函数 cusum 的定义位于 main 函数之前，之后才在 main 函数内调用了它。但是，如下面的程序所示，如果我们将 cusum 函数的定义放在 main 函数之后，则必须在调用之前做一个不带函数体的声明。

```
/*****************c0203.c*****************/
unsigned long long int cusum (unsigned long long int);

int main (void)
{
    unsigned long long int x, y, z;

    x = cusum (10);
    y = cusum (100);
    z = cusum (1000);
}

unsigned long long int cusum (unsigned long long int r)
{
    unsigned long long int n, sum;

    n = 1;
    sum = 0;

    while (n <= r)
    {
        sum = sum + n;
        n = n + 1;
    }

    return sum;
}
```

在不带函数体的函数声明中，参数列表中的标识符（形参的名字）可以省略，而且该声明必须以分号“;”结束。因此，以下两种声明方式都是合法的：

```
unsigned long long int cusum (unsigned long long int r);
unsigned long long int cusum (unsigned long long int);
```

在程序翻译期间，翻译器对函数调用作语法检查，看参数的数量和类型是否与函数的声明一致。如果不一致，将输出错误信息并中止翻译过程。

练习 2.2：

是否可以用表达式 cusum ()和 cusum (100, 20)来调用函数 cusum？为什么？请上机验证。

在 C 语言诞生之初，函数的声明并不是这个样子的，用今天的眼光来看，着实十分古

怪。以源文件 c0203.c 为例，要是用早期的方式来写，会是什么样子呢？

经过改写的源文件如下所示，但并不是完全“仿真”的，因为在那个时代，C 语言还没有引入 unsigned long long int 类型。

```
/****************c0204.c****************/
unsigned long long int cusum ();      //D1

main ()                               //D2
{
    unsigned long long int x, y, z;

    x = cusum (10);
    y = cusum (100);
    z = cusum (1000, 1200);           //S1

    return 0;
}

unsigned long long int cusum (r)      //D3
unsigned long long int r;             //D4
{
    unsigned long long int n, sum;

    n = 1;
    sum = 0;

    while (n <= r)
    {
        sum = sum + n;
        n = n + 1;
    }

    return sum;
}
```

先来看 main 函数的声明，它从 D2 处开始。可以看出，函数 main 的声明中没有返回类型，这在当时是允许的，如果函数的返回类型是 int 的话，则它可以省略。另一个显著的特点是函数名右边只是一对圆括号，内容为空。在那个时代，C 语言里还没有引入关键字“void”，如果函数没有参数，它只能这样写。事实上，即使函数有参数，不带函数体的函数声明也必须这样写。

从 D3 处开始的部分是函数 cusum 的声明，因为带有函数体，这也是它的定义。但是这种定义方式很奇怪，函数名右边的圆括号内只允许是参数的名字（标识符），对参数的声明（D4）夹在 D3 和函数体的左花括号“{”之间。

和现在一样，函数在调用前必须声明。如果函数的定义位于 main 函数之后，则它必须有一个不带函数体的声明。在程序中，函数 cusum 是在函数 main 之后定义的，为了在函数 main 内调用它，D1 处是它的前置声明。按照以往的惯例，函数名右边的圆括号内应当

为空。

由于 D1 处的声明并未指定参数信息，对此函数的调用就只能依靠程序员的自律了。如果他胡来，也没有办法阻止。如语句 S1 所示，即使我们知道这个函数只接受一个参数，我们也可以为它传递两个甚至多个参数。在翻译期间，翻译器连吭都不吭一声，因为它无法从 D1 处获得参数的数量和类型信息。

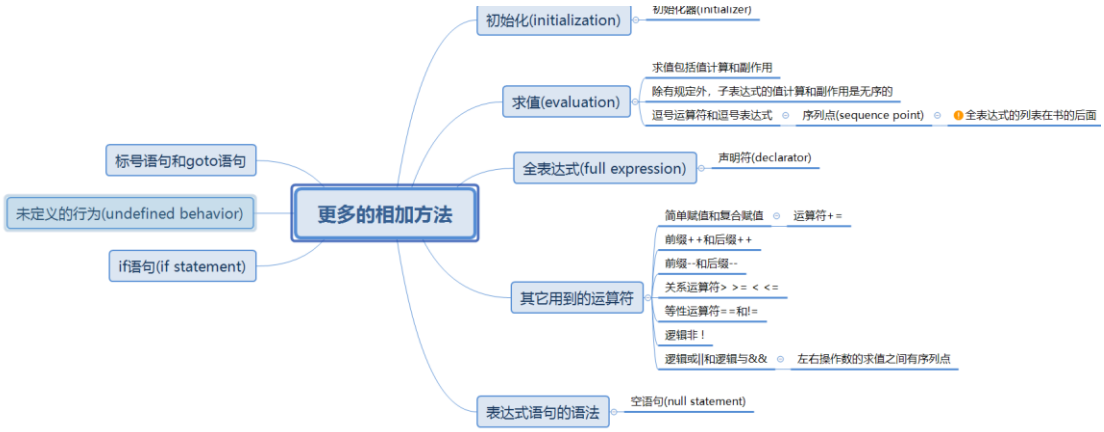
标准化之后的 C 语言借鉴了其他编程语言的经验，对函数的声明做了改进，其中最主要的一点是将参数的声明放在函数名后面的圆括号内，而且必须指定参数的类型；如果函数的声明不带函数体，可省略参数的名字，但必须指定其类型。现在，我们把**带有参数类型声明的函数声明称为函数原型**。

引入函数原型的原因是便于 C 实现在程序翻译期间实施类型检查。在调用函数时，如果参数的数量和类型与函数的声明不匹配，则它很容易被检查出来。

练习 2.3：

在变量的声明中，标识符是变量的名字，在表达式里，它是一个代表变量的（ ）；在函数的声明中，标识符是函数的名字，在表达式里，它是一个代表函数的（ ）；带函数体的函数声明被称为（ ）。

备选答案：(A) 函数定义　(B) 函数指示符　(C) 左值


更多的相加方法
初始化(initialization)
初始化器(initializer)
求值(evaluation)
求值包括值计算和副作用
除有规定外，子表达式的值计算和副作用是无序的
逗号运算符和逗号表达式
序列点(sequence point)
全表达式的列表在书的后面
全表达式(full expression)
声明符(declarator)
其它用到的运算符
简单赋值和复合赋值
运算符+=
前缀++和后缀++
前缀--和后缀--
关系运算符> >= < <=
等性运算符==和!=
逻辑非！
逻辑或||和逻辑与&&
左右操作数的求值之间有序列点
表达式语句的语法
空语句(null statement)
标号语句和goto语句
未定义的行为(undefined behavior)
if语句(if statement)

第 3 章　更多的相加方法

手里的工具种类多了，就能做更多的事情，做起事来也更加灵活。C 是强大、灵活、简洁的语言，这些特性源自它提供了众多的运算符和语句类型。在这一章里，我们将通过改写 cusum 函数来认识其中的一部分。在这个过程中，我们将进一步了解这门编程语言，学习它的更多知识和特性。

3.1 变量的初始化

在原先的程序中，我们先是声明了变量 n 和 sum，然后用两条语句将它们的存储值分别修改为 1 和 0。

然而，C 语言允许我们在声明 n 和 sum 的时候分别指定一个初始值。如此一来，我们就不再需要用两条语句来显式地修改变量的存储值。以下是程序的最新版本。

```
/*****************c0301.c*****************/
unsigned long long int cusum (unsigned long long int r)
{
    unsigned long long int n = 1, sum = 0;

    while (n <= r)
    {
        sum = sum + n;
        n = n + 1;
    }

    return sum;
}

int main (void)
{
    unsigned long long int res = cusum (1000);
}
```

请注意，相较于以前的版本，函数 cusum 内的声明部分已经改变，声明之后的两条语句也已经被移除。在变量 n 和 sum 的声明中，标识符 n 和 sum 分别用“=”连接了一个指定初始值的成分，这个成分叫作初始化器。

在 main 函数内，变量 res 的声明里也有一个初始化器 cusum (1000)，这显然是一个函数调用表达式。

注意，我们是在讨论声明而不是表达式，符号“=”并不是数学里的等号或者等于，也不是赋值表达式里的赋值运算符，这是一种新的用法，其含义可理解为“来自”。

初始化器的作用是在变量创建时，自动地为它赋予一个初始的值。初始化器可以是表达式，比如这里的 0、1 和 cusum (1000)。注意，并不是所有的变量在声明时都可以有初始化器，函数参数的声明里不得含有初始化器。这就是说，参数 r 的声明不允许用“=”连接一个初始化器，这是 C 语言的规定。

练习 3.1：

实验一下，如果为函数 cusum 的参数 r 添加一个初始化器，会怎么样。

3.2 认识复合赋值

接下来，我们继续修改 cusum 函数。这次主要针对 while 语句的循环体，目标是让它变得简洁。简单的语句和表达式不但看起来清爽，也能少打字，何乐而不为。下面就是修改之后的 cusum 函数。

```
unsigned long long int cusum (unsigned long long int r)
{
    unsigned long long int n = 1, sum = 0;

    while (n <= r)
    {
        sum += n;
        n += 1;
    }

    return sum;
}
```

观察一下，你会发现组成循环体的两条语句和以前不一样，变短了，没有了运算符+和运算符=，代之以新的运算符+=。尽管如此，这两条新语句依然是表达式语句，依然是由表达式和分号组成。

运算符“=”仅仅是将其右操作数的值赋给左操作数，所以称为“简单赋值”。新的运算符+=用来取代原先的“=”和“+”，但综合了它们的语义，所以它既执行相加操作，也执行赋值操作，故称为复合赋值运算符。注意，复合赋值运算符+=的“+”和“=”必须连写，不能分开。

复合赋值运算符有很多种，“+=”只是其中之一，其他复合赋值运算符将在本书的后面接触到。和简单赋值运算符一样，复合赋值运算符也需要一左一右两个操作数，而且它的左操作数必须是左值，但并不执行左值转换。

表达式 sum += n 等价于表达式 sum = sum + n，但是在语义上并不一样。前者是将变量 n 的值加到变量 sum，对变量 sum 只操作一次（只需要一条将变量 n 的值加到变量 sum 的机器指令或者汇编语言指令）；后者是读取变量 sum 和 n 的值相加后再保存到变量 sum，对变量 sum 操作两次（需要先读变量 sum 的值，再保存新值到变量 sum，这需要两条机器指令或者汇编语言指令）。

C 语言规定，复合赋值运算符的左操作数必须是一个左值，但不执行左值转换，而是用于接受赋值。所有表达式都有值，复合赋值表达式也不例外，它的值是复合赋值运算符的左操作数被赋值之后的新值。这就是说，表达式 sum += n 的值是左值 sum 被赋值之后的新值。

在第一章里我们已经讲过副作用，有句俗话叫“搂草打兔子”，套用在这里一点都不为过。你看，我们本来是要计算表达式的值，还顺带把变量的值也给改了，这就是副作用。简单赋值表达式有副作用，复合赋值表达式也有副作用，它们都会修改变量的存储值。例如在表达式 sum += n 里，变量 sum 的值就被修改为它原来的值和变量 n 的值相加的结果。

同理，表达式 n += 1 等价于表达式 n = n + 1。不同之处在于，n += 1 中的 n 不执行左值转换，而仅仅是将 1 加到变量 n 并覆盖它原来的值。因此，n += 1 也是有副作用的表达式。

3.3 认识递增运算符

我们知道，表达式 n = n + 1 是赋值表达式，可以用复合赋值表达式 n += 1 来完成相同的操作。

如果仅仅是将变量 n 的存储值在原来的基础上加 1，则表达式 n = n + 1 还有更简单的写法，那就是使用“++”运算符。注意，这两个“+”必须连写，不能分开。

该运算符只有一个操作数，这个操作数可以在左（前）边，也可以在右（后）边，也就是分为前缀形式和后缀形式。如果使用前缀形式，则语句

```
n += 1;
```

可以改写成这样：

```
++ n;
```

相反地，如果使用后缀形式，可以改写成这样：

```
n ++;
```

这里实际上是两个运算符，前缀形式的++称为前缀递增运算符，后缀形式的++称为后缀递增运算符。注意，不管是前缀递增运算符，还是后缀递增运算符，它们的操作数都必须是左值，且不执行左值转换，++ 5 和 6 ++都是非法的。

表达式++ n 和 n ++都是具有副作用的表达式，它们的副作用一样，都是导致变量 n 的存储值被修改为加 1 之后的值。因此，不管是采用哪一种写法，如果变量 n 原先的值是 1，则不管执行这两条语句中的哪一个，执行之后，变量 n 的值都会变成 2。

这样看来，表达式++ n 和 n ++似乎没有什么区别。然而，C 语言的发明者显然不会如此无聊。

在前缀形式中，表达式的值（前缀递增运算符的结果）是操作数加 1 之后的值。这就是说，如果变量 n 原来的值是 1，则表达式++ n 的值是 2；在后缀形式中，表达式的值（后缀递增运算符的结果）是操作数加 1 之前的原值。这就是说，如果变量 n 原来的值是 1，则表达式 n ++的值是 1。

需要再次强调的是，和赋值运算符的左操作数一样，前缀递增和后缀递增运算符的操作数不执行左值转换。

练习 3.2：

你可曾想过怎样才能验证前缀递增和后缀递增运算符的结果是不同的？通过上机翻译并调试以下程序就可以做到（前缀递增和后缀递增运算符的优先级都高于赋值运算符）。

```
int main (void)
{
    int x = 0, y, z;

    y = ++ x;
    z = x ++;
}
```

请编辑和保存上述程序，用-g 参数翻译为可执行文件，然后调试该程序。在第一条语句那里设置断点，然后运行到断点处，单步执行，观察变量 x、y 和 z 的值如何变化。

3.4 初识复杂的表达式

显然，表达式++ n 和 n ++的值是不一样的。也正是利用了这一点，前面的 while 语句可以进一步改写为：

```
while (n <= r)
{
    sum = sum + n ++;
```

```
}
```

在这里，表达式 sum = sum + n ++涉及三种运算符，运算符“++”在这里的优先级最高，“+”次之，“=”的优先级最低。因此，运算符++的操作数是 n；运算符+的操作数是子表达式 n ++的值和中间那个 sum 的值；运算符=的操作数则是 sum + n ++的值和左值 sum。说到底，这还是一个赋值表达式，等价于 sum = (sum + (n ++))，而且这个表达式还有更新变量 sum 和 n 的存储值的副作用。

在第一章里我们已经强调过，操作数的值计算要先于运算符的值计算，也就是先要计算操作数的值，再得到运算符的结果。在这里，要得到表达式 sum = sum + n ++的值（也就是运算符=的结果）就必须先计算子表达式 sum + n ++的值；要计算表达式 sum + n ++的值（也就是运算符+的结果）就必须先计算子表达式 sum 和 n ++的值。子表达式 sum 的值是其左值转换后的值；子表达式 n ++的值是变量 n 递增前的原值。

所以，表达式 sum = sum + n ++的功能是将变量 n 递增前的原值和变量 sum 的值相加，结果依然保存到变量 sum 中。该表达式具有两个副作用，一是修改变量 sum 的存储值；二是递增变量 n 的存储值。

在 while 语句的累加过程中，每次都是先用变量 n 递增前的原值和变量 sum 的原值相加，结果再存回变量 sum，表达式 sum = sum + n ++正是利用了后缀递增运算符的一个特点：该运算符的值是其操作数递增前的原值。

在第一章里我们就已经给出了 while 语句的语法，也知道它的循环体不要求非得是复合语句。在这里，复合语句的花括号内只有一条表达式语句，在这种情况下花括号是可有可无的，还不如将花括号去掉，就像这样：

```
while (n <= r)
    sum = sum + n ++;
```

我们知道 C 语言对程序的格式不做特殊要求，因为这个 while 语句很简单，占用两行似乎太浪费空间了，用一行来书写就行：

```
while (n <= r) sum = sum + n ++;
```

所有赋值运算符的优先级相同，包括赋值运算符=和赋值运算符+=，但后者比前者在用法上更简洁。为此，我们甚至可以这样改写上述 while 语句：

```
while (n <= r) sum += n ++;
```

在这里，表达式 sum += n ++是将表达式 n ++的值加到变量 sum，等价于 sum = sum + n ++。

练习 3.3：

可以把表达式 sum = sum + n ++改成 sum = sum + ++ n 吗？为什么？上机验证你的想法（重点是看结果是否正确）。

3.5 认识关系运算符

从本书一开始到现在，我们在 while 语句的控制表达式里用的都是运算符“<=”。实际上，该运算符是关系运算符，C 语言里的关系运算符包括“>”“>=”“<”和“<=”，分别表示“大于”“大于等于”“小于”和“小于等于”，而“<=”只是其中的一个。

所有关系运算都是非常相似且极易理解的，因此也就不需要多费口舌加以解释。对于以上关系运算符，当对应的关系成立时，关系运算符（关系表达式）的结果为 1；否则关系运算符（关系表达式）的结果为 0。如果使用大于等于运算符“>=”，则上述 while 语句可以是这样的：

```
while (r >= n) sum += n ++;
```

实际上，这只是一个障眼法，原来的控制表达式为 n <= r，现在是将 n 和 r 调换了一

下位置，而运算符自然也要由原来的“<=”改为“>=”。

练习 3.4：

1．若变量 n 和 sum 的初值都为 0，请修改函数 cusum 的 while 语句但不改变它的功能，要求：不得使用复合语句；只能使用关系运算符>和复合赋值运算符+=。提示：复合赋值运算符+=是从右向左结合的。

2．若变量 n 和 sum 的初值都为 0，请修改函数 cusum 的 while 语句但不改变它的功能，要求：不得使用复合语句；只能使用关系运算符<、复合赋值运算符+=和前缀递增运算符。提示：前缀递增运算符的优先级高于复合赋值运算符。

3．在不改变程序功能的前提下，我们将 cusum 函数做了如下修改：

```
unsigned long long int cusum (unsigned long long int r)
{
    unsigned long long int n = 0, sum = 0;

    while (n < r)
        sum = sum + n = n + 1;

    return sum;
 }
```

但是，源文件在翻译的过程中出错，请解释出错的原因并改正这个错误。

4．所有关系运算符的优先级都相同，而且都是从左往右结合的；加性运算符+也是从左往右结合的，而且它的优先级高于关系运算符。给定以下函数 f：

```
int f (int a, int b, int c, int d, int e, int f)
{
    return a + b + c > d > e <= f;
}
```

如果调用它的表达式为 f (1, 2, 3, 7, 8, 9)，则：

(a) 表达式 c > d > e <= f 的意思是变量 c 的值大于变量 d 的值；变量 d 的值大于变量 e 的值；变量 e 的值大于等于变量 f 的值，对吗？为什么？

(b) 为表达式 a + b + c > d > e <= f 添加适当的括号，以体现各运算符的操作数都是谁。

(c) 函数 f 的返回值是多少？请添加一个 main 函数，使之成为一个完整的 C 源文件，上机验证这个结果。

3.6 求值

运算符的优先级仅仅是“一个等级，等级高的运算符优先与旁边的操作数结合”，而并不是指优先计算。运算符的结合性也一样，仅仅是在运算符的优先级相同时，指示哪一个才有权先选择操作数，和谁先计算也没有关系。对于这一点，不单单是很多 C 语言的初学者搞错，即使是那些已经学过了 C 语言的人也经常在这里栽跟头。

我想我们需要一个活生生的例子，尽管截至目前我们已经接触过很多表达式，它们都可以作为例子，但都不够典型。那么，我们先从一个最典型的例子入手。假定 a、b 和 c 都是 int 类型的变量，且我们要计算以下表达式的值：

```
++ a + b * c
```

这里出现了一个我们还没见过的运算符“*”，它的作用是计算两个数的乘积，和我们平时做乘法是一样的。运算符*需要一左一右两个操作数，它的优先级比运算符+要高，但比

运算符++要低，这三个运算符的优先级由高到低分别是++、*和+。现在，你能说是先计算++，再计算*，最后计算+吗？或者说，是先计算出++ a 的值，再计算 b * c 的值，最后计算前两者的和吗？

答案是不能。

优先级高的运算符有权先选自己的操作数，所以 b 和 c 是运算符*的操作数；而 a 则是运算符++的操作数；运算符+的操作数只能是++ a 的结果和 b * c 的结果。即，这个表达式等价于 (++ a) + (b * c)。

因此，要得到运算符+的结果，必须先计算其操作数++ a 和 b * c，但可以先计算++ a 再计算 b * c，也可以先计算 b * c 再计算++ a。进一步地，要计算运算符*的结果，必须先计算其操作数 b 和 c，也就是进行左值转换。但是，可以先计算 b 再计算 c，也可以先计算 c 再计算 b。

当然，这里还存在着其他可能的计算顺序。如果将运算符++的值计算记为 V_{++}，将运算符*的值计算记为 V_{*}，将运算符+的值计算记为 V_{+}，将 b 和 c 的值计算分别记为 V_b 和 V_c，则表达式++ a + b * c 的计算过程可以有多种不同的顺序，以下列举了其中的一部分（假定变量 a、b 和 c 的当前值分别为 0、1 和 2，括号中的数值为计算出来的结果）。

$V_b(1) \rightarrow V_c(2) \rightarrow V_{*}(2) \rightarrow V_{++}(1) \rightarrow V_{+}(3)$

$V_c(2) \rightarrow V_b(1) \rightarrow V_{*}(2) \rightarrow V_{++}(1) \rightarrow V_{+}(3)$

$V_{++}(1) \rightarrow V_b(1) \rightarrow V_c(2) \rightarrow V_{*}(2) \rightarrow V_{+}(3)$

$V_{++}(1) \rightarrow V_c(2) \rightarrow V_b(1) \rightarrow V_{*}(2) \rightarrow V_{+}(3)$

$V_b(1) \rightarrow V_{++}(1) \rightarrow V_c(2) \rightarrow V_{*}(2) \rightarrow V_{+}(3)$

$V_c(2) \rightarrow V_{++}(1) \rightarrow V_b(1) \rightarrow V_{*}(2) \rightarrow V_{+}(3)$

$V_b(1) \rightarrow V_c(2) \rightarrow V_{++}(1) \rightarrow V_{*}(2) \rightarrow V_{+}(3)$

$V_c(2) \rightarrow V_b(1) \rightarrow V_{++}(1) \rightarrow V_{*}(2) \rightarrow V_{+}(3)$

显然，运算符的优先级和它是否被优先计算无关，子表达式的计算顺序通常也没有什么规律可言，但并不影响最终的结果。当然，正如我们曾经强调过的，在这看似没有规律和顺序的计算过程中，最基本的原则是先计算运算符的操作数，再计算运算符本身的值。

表达式++ a + b * c 不仅要计算出一个值，它还有副作用，因为它的子表达式++ a 是有副作用的表达式。但是，这个副作用什么时候发起？

一旦将副作用也考虑进来，事情就变得更加复杂了。我们不单要考虑值计算的顺序，还要关心副作用的发起时间，这就需要一个新的术语“求值”来涵盖这两个层面。我们知道，表达式可以指示变量或者函数，也可以计算一个值，还可能发起一个副作用，而**“求值一个表达式”则通常包括值计算和发起一个副作用**。当然，有些表达式没有副作用，那么它的求值仅仅包含值计算。

所以，我们现在可以这样问：表达式++ a + b * c 求值的时候，子表达式的值计算和副作用按什么顺序进行？

对于这个问题，C 语言的规定是很明确的：除非另有指定，在表达式求值的时候，**子表达式的值计算和副作用之间没有明确的顺序，或者说是无序的**。

之所以这样规定，是希望把决定权交给翻译软件，由它们在翻译程序的时候自主决定，这样可以生成更加紧凑和高效的机器指令。

这里的“另有指定”，是针对一小部分特殊的表达式来说的，比如，对于后缀++运算符来说，它的值计算发生在（修改其操作数所代表的变量的存储值的）副作用之前；对于简单赋值和复合赋值运算符来说，（修改其左操作数所代表的变量的存储值的）副作用发生在其左右操作数的值计算之后。

因此，假定把子表达式++ a 的副作用记为 S_{++}，则表达式++ a + b * c 求值时，其

子表达式的值计算和副作用可以有更多的顺序，以下列举了其中的一小部分：

$V_b(1) \rightarrow V_c(2) \rightarrow V_*(2) \rightarrow V_{++}(1) \rightarrow S_{++}(1) \rightarrow V_+(3)$

$V_c(2) \rightarrow V_b(1) \rightarrow V_*(2) \rightarrow V_{++}(1) \rightarrow V_+(3) \rightarrow S_{++}(1)$

$V_{++}(1) \rightarrow V_b(1) \rightarrow V_c(2) \rightarrow V_*(2) \rightarrow S_{++}(1) \rightarrow V_+(3)$

$V_{++}(1) \rightarrow V_c(2) \rightarrow V_b(1) \rightarrow V_*(2) \rightarrow V_+(3) \rightarrow S_{++}(1)$

$V_b(1) \rightarrow V_{++}(1) \rightarrow V_c(2) \rightarrow V_*(2) \rightarrow S_{++}(1) \rightarrow V_+(3)$

$V_c(2) \rightarrow V_{++}(1) \rightarrow V_b(1) \rightarrow S_{++}(1) \rightarrow V_*(2) \rightarrow V_+(3)$

$V_b(1) \rightarrow V_c(2) \rightarrow V_{++}(1) \rightarrow V_*(2) \rightarrow S_{++}(1) \rightarrow V_+(3)$

$V_c(2) \rightarrow V_b(1) \rightarrow V_{++}(1) \rightarrow V_*(2) \rightarrow V_+(3) \rightarrow S_{++}(1)$

显然，修改变量 a 的存储值的副作用可以发生在整个表达式求值期间的任何时候。再以我们前面所讨论的 while 语句为例：

```
while (n <= r) sum = sum + n ++;
```

整个表达式 sum = sum + n ++的值也是运算符=的值，记为 $V_=$；修改变量 sum 存储值的副作用也是运算符=的副作用，记为 $S_=$；表达式 sum + n ++的值也是运算符+的值，记为 V_+；表达式 n ++的值也是后缀递增运算符的值，记为 V_{++}；表达式 n ++的副作用也是后缀递增运算符的副作用，记为 S_{++}；中间那个表达式 sum 的值是 V_{sum}。那么，这整个表达式的求值顺序可以是：

$V_{++} \rightarrow S_{++} \rightarrow V_{sum} \rightarrow V_+ \rightarrow V_= \rightarrow S_=$

也可以是：

$V_{++} \rightarrow V_{sum} \rightarrow V_+ \rightarrow V_= \rightarrow S_{++} \rightarrow S_=$

还可以是：

$V_{sum} \rightarrow V_{++} \rightarrow V_+ \rightarrow V_= \rightarrow S_= \rightarrow S_{++}$

当然，还可以有其他更多的求值顺序，只要它们符合前面的几个约束条件（运算符操作数的值计算要先于运算符的值计算；后缀递增运算符++和赋值运算符特有的求值顺序）。但是无论求值顺序有多少种，都不影响最终结果的正确性。

练习 3.5：

假定变量 n 和 sum 当前的存储值分别为 1 和 0，请在表达式 sum = sum + n ++里代入这两个值以验证上述几种求值顺序不影响最终的结果。

3.7 认识逗号表达式

现在，让我们回到函数 cusum 在本章开头的原始版本：

```
unsigned long long int cusum (unsigned long long int r)
{
    unsigned long long int n = 1, sum = 0;

    while (n <= r)
    {
        sum = sum + n;
        n = n + 1;
    }

    return sum;
}
```

如果 while 语句的循环体包含了一个以上的声明和语句，使用复合语句通常是必须的选择，但如果复合语句仅由多条语句组成而没有声明，而且这些语句都是表达式语句，则依然可以不使用复合语句，而是把这些表达式用逗号“,”首尾相连，就像这样：

```
while (n <= r) sum = sum + n, n = n + 1;
```

或者这样：

```
while (n <= r) sum += n, n ++;
```

注意，这里的逗号“,”并不是语文里的逗号，它是 C 语言的一个运算符，称为“逗号运算符”。逗号运算符有一左一右两个操作数，逗号运算符和它的操作数共同组成逗号表达式，例如 5, 6 就是一个逗号表达式。

在所有运算符里，逗号运算符的优先级最低。所以，表达式 sum += n, n ++是一个逗号表达式。这就是说，运算符+=的操作数是 sum 和 n；运算符++的操作数是 n；逗号运算符的操作数只能是 sum += n 的值和 n ++的值。

即使还不了解逗号运算符和逗号表达式的作用，我们也能通过推理发现表达式 sum += n, n ++的求值过程有问题。

我们知道，求值一个表达式包括值计算和发起一个副作用，然而并不要求它们是一个连续的过程。那么，对于逗号表达式 sum += n, n ++来说，它的求值过程是怎样的呢？如果先求值 sum += n，而且它的值计算和副作用在求值表达式 n ++之前就能完成，这自然没什么问题，加到变量 sum 的值是变量 n 递增前的原值；但如果先求值 n ++，而且它的值计算和副作用在求值 sum += n 之前就已经完成，那么加到变量 sum 的值是变量 n 递增后的新值，这就违背了我们的初衷和原意，将得不到正确的累加结果。

练习 3.6：

表达式 sum = sum + n ++的值与子表达式的求值顺序无关，变量 n 和 sum 在求值完成后的新值也与子表达式的求值顺序无关，为什么？

3.7.1 全表达式和序列点

在解决逗号表达式的求值顺序问题之前，我们需要先来了解另外两个概念：全表达式和序列点。总体上，如果一个表达式在形式上是独立的，不是其他表达式的组成部分，也不是一个声明符的组成部分，那么它就是一个全表达式。

声明符是变量声明或者函数声明的一部分，用来描述被声明的实体。最简单的声明符是一个标识符，例如在以下声明中，标识符 m 和 n 就是声明符：

```
unsigned long int m, n = 0;
```

有些声明符相对复杂，在下面的函数声明中，声明符是 func (int x)。显然，这个声明符不单纯是标识符，还包括标识符后面的参数列表。

```
int func (int x);
```

取决于声明的是什么东西（类型），声明符可能会多种多样，而且有些声明符里会包含表达式，你很快就会接触到这样的声明符。

继续来讨论全表达式，为了增加感性认识，我们以下面的程序代码为例，来看看哪些是全表达式：

```
unsigned long long int cusum (unsigned long long int r)
{
    unsigned long long int n = 1, sum = 0;

    while (n <= r) sum += n ++;
```

```
    return sum;
}
```

首先，表达式语句由表达式和分号“;”组成，表达式语句中的表达式是全表达式。也就是说，将语句

```
sum += n ++;
```

末尾的分号“;”去掉之后，剩下的部分就是全表达式。

其次，while 语句的控制表达式也是全表达式，所以 n <= r 是全表达式。实际上不单单是 while 语句，但凡是需要控制表达式的语句，其控制表达式往往都是全表达式。

第三，如果 return 语句是由关键字“return”和表达式组成，则该表达式也是全表达式。所以 return 语句中的表达式 sum 是全表达式。

最后，很多初始化器都是全表达式。在这里，用于初始化变量 n 和 sum 的表达式 0、1 是全表达式。

全表达式还有很多，但是凭我们现在所掌握的 C 语言知识还无法全部列举，所以要留到本书的后面再一一介绍。

另一个概念“序列点”则与表达式的求值有关。给定任意两个表达式 A 和 B，如果 A 的值计算和副作用发生在 B 的值计算和副作用之前，则我们说在 A 和 B 的求值之间存在一个序列点。显然，序列点是一个求值的界线，前一个表达式的值计算和副作用已经完成，而后一个表达式的值计算和副作用还没有开始。

C 语言规定，在一个全表达式的求值和下一个全表达式的求值之间存在一个序列点。如图 3-1 所示，在 while 语句内有三个全表达式，所以也存在三个序列点。

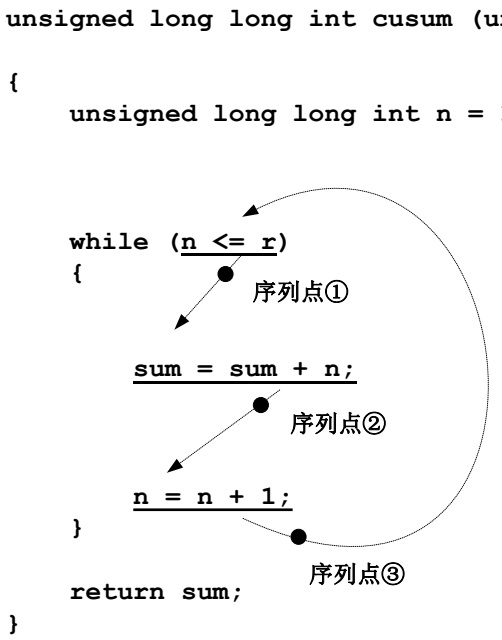


Fig.3-1 序列点示意图

显然，序列点可以保证程序的行为精确可控，执行的结果可以预测。例如，要是 while 语句开始下一轮循环时，全表达式 n = n + 1 的副作用已经发起并完成，则控制表达式 n <= r 的求值可以用到变量 n 的新值；要是没有序列点的存在，你无法保证 n <= r 求值的时候，表达式 n = n + 1 的副作用是否已经发起并完成。在这种情况下，也就无法保证表达式 n <= r 的求值是否能用上变量 n 的新值。

讲完了全表达式和序列点，让我们继续逗号表达式的话题。逗号运算符有一左一右两个操作数，C 语言规定，在其左操作数的求值和右操作数的求值之间有一个序列点。

这就是说，在对表达式 sum += n, n ++求值时，是先求值左操作数 sum += n，当它的值计算和副作用都完成后，才开始求值右操作数 n ++。因此，绝对不会发生我们在前面所担心的事情：先求值 n ++，或者混乱交叉求值。

逗号运算符的值是其右面那个操作数的值，左操作数的值被丢弃。所以，表达式 5, 6 的值是 6；表达式 sum += n, n ++的值是其子表达式 n ++的值。

这么说来，逗号运算符的左操作数似乎没有什么存在的意义和价值。实际上，设计逗号表达式的目的是希望逗号运算符的左操作数有副作用。这样，在实际求值的时候，左操作数的意义体现在它的副作用上，右操作数则提供了整个逗号表达式的值。

从这个意义上来说，表达式 5,6 虽然是合法的逗号表达式，但没有什么用处；表达式 sum += n, n ++的左操作数 sum += n 是有副作用的表达式，我们只关注它的副作用；右操作数是 n ++，它既有副作用，又提供了整个逗号表达式的值。

练习 3.7:

1.在以下代码片段中，逗号表达式为(　　)，它的值为(　　)。如果要把该逗号表达式的值赋给变量 m，应该怎么修改第二行？（注意，我们说过，在所有运算符里，逗号运算符的优先级最低）。

```
long int m, x, y, z = 0;
y = z, x = ++ y;
```

2.对于逗号运算符，如果左右操作数求值之间不存在序列点，那么，请用不同的求值过程推演一下，看一看每当表达式 sum += n, n ++求值完成后，变量 sum、n 和整个表达式的值是否不受求值顺序的影响。

3.8 认识表达式语句

上面我们已经了解到逗号运算符的结果是其右操作数的值，那么，我们可以利用这个特点来将我们的 while 语句改写为更奇特的形式：

```
unsigned long long int cusum (unsigned long long int r)
{
    unsigned long long int n = 1, sum = 0;

    while (sum += n ++, n <= r) ;

    return sum;
}
```

在这里，while 语句的控制表达式是一个逗号表达式，逗号运算符的左操作数是表达式 sum += n ++，用来累加并更新变量 sum 的存储值，它是有副作用的表达式，将更新变量 sum 和 n 的存储值；控制表达式的值是逗号运算符的右操作数 n <= r 的值。

while 语句是反复执行的，在每次执行前，都要对控制表达式求值以决定是否继续循环执行，而我们正是利用了这一点。当然，这里面也有序列点的功劳，因为在两个子表达式的求值之间有一个序列点，所以在求值子表达式 n <= r 的时候，可以保证变量 n 的存储值已经被前一个表达式 sum += n ++求值时的副作用更新过。

很奇怪地，while 语句的循环体仅仅是一个分号“;”，这是什么意思呢？这在 C 语言里称为空语句。空语句不执行任何操作，但有时候还需要它。就像上面的示例一样，如果没有什么需要执行的操作，但是在语法上还不能省掉这条语句，就需要它了。

本质上，空语句是特殊的表达式语句。这是因为——好吧，还是先来看表达式语句的语法组成：

表达式可选 ;

显然，表达式语句由表达式及一个分号“;”组成，但表达式是可选的，如果省略了表达式，则只剩下一个分号，这就成了空语句。

练习 3.8:

在本节中，while 语句的控制表达式是逗号表达式。第一次执行控制表达式后，变量 n 和 sum 的值分别为(　)和(　)；最后一次执行控制表达式后，变量 n 的值比变量 r 的值大(　)。

3.9 认识递减和逻辑求反运算符

所谓条条大路通罗马，有时候换个思路来解决编程问题会找到更简单的方法。你看，为了从 1 加到 N，我们所做的就是 1 加上 2，再加上 3，一直加到 N，为此我们特意声明了一个变量 n 来实施这种数字的递增。

另一方面，从 1 加到 N，和从 N 开始往 1 加没有什么区别，只不过方向相反。既然如此，那我们就用不着多声明一个变量，直接操作参数 r 更方便。以下是函数 cusum 的另一个版本，用的就是这种方法。

```
unsigned long long int cusum (unsigned long long int r)
{
    unsigned long long int sum = 0;

    while (r) sum += r --;

    return sum;
}
```

在这里，while 语句的控制表达式仅仅是 r。每次循环开始前先求值表达式 r（也就是进行左值转换）以判断是否继续循环。除非变量 r 的存储值为 0，否则整个循环过程将持续进行。

在 while 语句的循环体内，我们遇到了一个新的运算符--，和运算符++一样，它只需要一个操作数，而且必须是左值，但不进行左值转换。不同之处在于，运算符++使得操作数的存储值递增，而运算符--则使得操作数的存储值递减。

实际上存在两种递减运算符，即前缀递减运算符和后缀递减运算符。前缀递减运算符需要一个右操作数，例如-- r，前缀递减表达式的值，或者说前缀递减运算符的结果是其操作数递减后的值；前缀递减表达式还有一个副作用，它使得操作数的存储值在原来的基础上减一。

后缀递减运算符需要一个左操作数，例如 r --，后缀递减表达式的值，或者说后缀递减运算符的结果是其操作数递减之前的原值；后缀递减表达式还有一个副作用，它使得操作数的存储值在原来的基础上减一。

前缀递减运算符的优先级和前缀递增运算符的优先级相同；后缀递减运算符和后缀递增运算符的优先级相同。

回到 while 语句的循环体，它是一个表达式语句，由表达式 sum += r --和末尾的分号组成。如果你能理解表达式 sum += n ++，你自然也能理解这个表达式。

表达式 sum += r --是一个复合赋值表达式，因为运算符--的优先级高于+=。为了得到运算符+=的结果，必须先得到运算符的--结果。每次执行循环体时，都会将变量 r 的原值加到变量 sum，变量 r 的存储值递减。除了求得运算符+=和--的结果，这个表达式还有修改变量 sum 和 r 的存储值的副作用。

最后一次执行循环体时，变量 r 的值是 1。此时，表达式 sum += r --是将变量 r 的原值，也就是 1 加到变量 sum，然后将变量 r 的存储值递减。递减之后，变量 r 的值为 0。当 while 语句对控制表达式 r 进行求值以决定是否进行下一轮循环时，因为变量 r 的存储值为 0，所以控制表达式的值也为 0，退出 while 语句。

对于那些非常讲究代码规范的人来说，他们坚持认为这个 while 语句应该写成下面这样才显得直观：

```
while (r != 0) sum += r --;
```

尽管我并不认为大家的抽象思维能力会如此低下，以至于非得如此画蛇添足、狗尾续貂，但我也不得不承认这样做其实也没什么毛病。在这里，“!=”是一个新的运算符，它的意思是“不等于”。注意感叹号“!”和等号“=”一定要连写，不得分开。

运算符!=属于等性运算符，等性运算符共有两个，!=是其中之一，另一个是==，意思是“等于”。等性运算符属于二元运算符，它们需要一左一右两个操作数，并共同组成等性表达式。

如果运算符!=的两个操作数在数值上不相等（“不等于”的关系成立），则等性运算符!=的结果为 1，否则为 0。

如果运算符==的两个操作数在数值上相等（“等于”的关系成立），则等性运算符==的结果为 1，否则为 0。

在这个 while 语句中，是先对左值 r 进行左值转换，然后用转换后的数值和数字 0 比较。如果它们的数值上不相等，则“不等于”的关系成立，控制表达式（等性表达式）的值为 1，可以继续循环；否则控制表达式（等性表达式）的值为 0，退出循环。

以上，控制 while 循环的逻辑是“r 的值不等于 0”。要是我们想使用运算符==，该怎么写呢？此时，控制 while 循环的逻辑是“r 的值等于 0 并非事实”。如果按此逻辑，则上述 while 语句可以改写为

```
while (! r == 0) sum += r --;
```

这里还有一个新的运算符“!”，叫作逻辑求反运算符，它只需要一个右操作数，因此也是一元运算符。逻辑求反运算符的功能很简单：如果操作数的值为 0，则运算符!的结果为 1；如果操作数不为 0，则运算符!的结果为 0。例如，表达式! 0 的值是 1；表达式! 1 的值是 0；表达式! 23 的值是 0；表达式! 500500 的值也是 0。

运算符!的优先级低于等性运算符!=和==，在这里，运算符==的操作数是 r 和 0；运算符!的操作数是表达式 r == 0 的值。因此，表达式! r == 0 等价于! (r == 0)。

以上，是要先比较 0 和 r（经左值转换后）的值，如果变量 r 的值不为 0，则表达式 r == 0 的值为 0（因为关系不成立），而表达式! r == 0 的值为 1，循环可以继续进行。否则，如果变量 r 的值已经递减到 0，则表达式 r == 0 的值为 1（因为关系成立），而表达式! r == 0 的值为 0，退出 while 语句。

历史上曾经发生过因为粗心大意而将 r == 0 写成 r = 0 这样的事，所以有些人强烈建议将 r == 0 写成 0 == r，例如：

```
while (! 0 == r) sum += r --;
```

这样一来，如果将 0 == r 写成 0 = r，这将成为一个非法的表达式而在翻译的时候出错，因为 0 不是左值。然而我认为，将 r == 0 写成 0 == r 说明你已经意识到这里可能会写错，所以具体怎么写已经不重要了。

3.10 参数值的有效性检查

我们编写函数 cusum 的目的是为了获得通用性，能够计算从 1 加到 N 的和，这里的 N 是正整数。问题是这道题只能在有限的范围内求解，函数 cusum 的返回类型和参数类型都

是 unsigned long long int，它可以表示多大的数呢？不知道，这取决于具体的计算机平台，但 C 语言可以保证它不小于 18446744073709551615。

那么，这个限制是什么意思呢？是什么原因造成的呢？我们知道，变量所占用的存储空间大小取决于它在声明时的类型。举个例子来说，在 C 语言里，unsigned char 是整数类型，这种类型的变量都统一规定为 1 个字节大小。

对于绝大多数计算机架构来说，字节的长度是 8 比特，或者说一个字节是由 8 个比特组成，因此，如果深入到变量内部，从比特的层次上来看，当 unsigned char 类型的变量保存的内容为二进制比特序列 00000000 时，具有最小值 0；当它的内容为二进制比特序列 11111111 时，具有最大值 255。

假定我们声明了一个 unsigned char 类型的变量 c，并将其初始化为最大值 255。如图 3-2 所示，图的右侧显示了它在存储器中的位置，以及它的位模式；左侧呢，显示了表达式 c = c + 1 的执行过程。

显然，根据二进制数的加法规则，最左侧也将有一个进位，所以 255 加 1 的结果用二进制表示就是 100000000。但因为变量的大小只有 8 个比特，最后的进位丢失，相加的结果只能是二进制的 00000000。那么，多出来的比特不会拱到邻近的存储区里吗？这怎么可能，隔壁可能是其他变量的地盘，只要计算机还有点用，它就不会允许这种事情发生。

那么，unsigned long long int 类型的变量会如何呢？答案是除了大小之外，其他没有任何区别。如果它可以容纳的最大值是 18446744073709551615，则我们在做从 1 到 N 的加法时，有可能还没有加到 N，结果就已经等于或者超过了这个最大值，再怎么加都没有什么意义了。

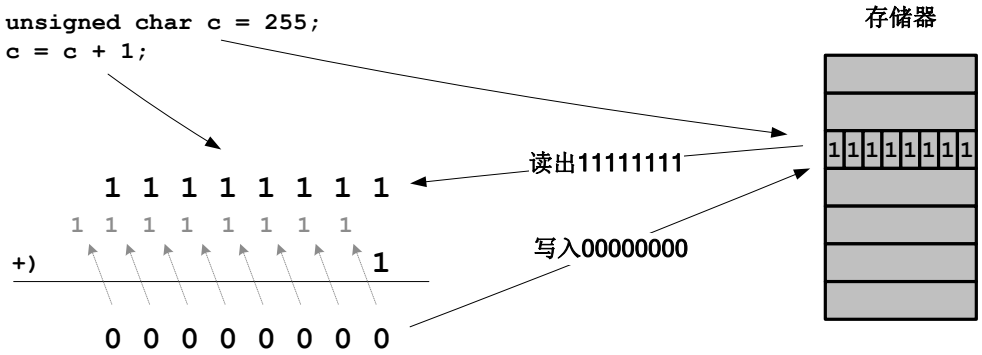


Fig.3-2 算术操作对变量内容的影响

为此，传递给函数 cusum 的参数值不能太大，建议的范围是大于 0 且小于等于 1 000 000 000，这既是经验，也是个人偏好。从经验上来说，从 1 加到 1 000 000 000 的结果可以用 unsigned long long int 类型的变量容纳，这是我做过实验的；从个人偏好上来说，尽管 1 000 000 000 不是上限，参数的值可以稍微再大一些，但这个数字比较好记，不是吗？！

3.10.1 认识 if 语句

尽管函数的调用者传递什么，被调用的函数只能被动接受，但如果真的传入了不恰当的值，在函数内部还是应该做点什么，而不是将错就错，把事情干得一塌糊涂。为了做到这一点，就得对参数的值进行检查和判断，函数 cusum 用来计算 1 到 N 的累加和，原则上返回值不应该小于 1。利用这个特点，如果调用者传入的值超过了我们容许的范围，就直接返回 0，否则就可以进行实际的累加过程。调用者可以检查函数的返回值，如果发现是 0，它就知道本次累加失败了。

为此，我们就要使用 C 语言提供的 if 语句。在 C 语言里，语句用于执行动作，控制程序的执行流程，使用 while 语句可以循环往复地执行同一段代码，而 if 语句则有选择地执行或者不执行某些代码，因此它又称为选择语句。以下是 if 语句的语法。

if (*表达式*) *语句*

或者

if (*表达式*) *语句* **else** *语句*

如上所示，if 语句有两种形式，第一种形式由关键字“if”引导，然后是一对圆括号和由它括住的表达式，这是 if 语句的控制表达式，后面是语句。第二种形式的前半部分和第一种形式相同，但添加了一个由关键字“else”引导的部分，称为 else 子句。

如图 3-3 所示，if 语句的执行过程是这样的：先求值控制表达式，再根据该表达式的值，以及是否有 else 子句进入相应的分支进行处理。

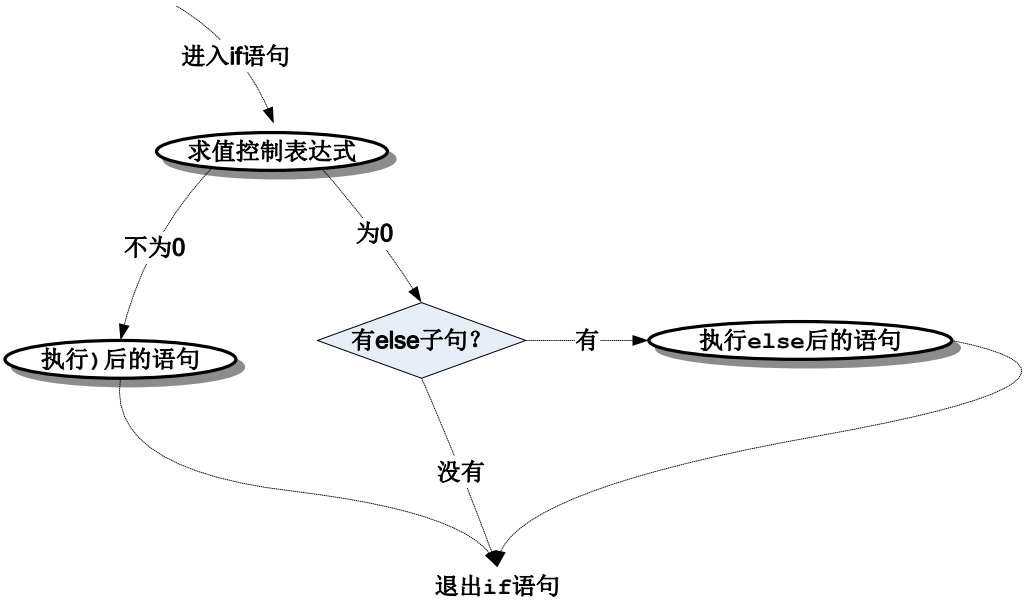


Fig.3-3 if 语句的执行流程

如下面的代码所示，函数 cusum 内使用了 if 语句，if 语句的控制表达式是一个关系表达式 r > 1000000000。在执行期间，表达式 r 执行左值转换，并与 1000000000 进行比较。如果变量 r 的值大于 1000000000，关系成立，控制表达式的值为 1，执行语句

```
return 0;
```

这将导致程序的执行离开 cusum 函数，直接返回到调用者，并将 0 作为返回值。在英语里，“else”是“否则……”的意思，因此，如果上述关系不成立，则执行语句

```
while (r) sum += r --;
```

这是一个 while 语句，它已经是你的老朋友了，我相信用不着再多说什么。

```
unsigned long long int cusum (unsigned long long int r)
{
    unsigned long long int sum = 0;

    if (r > 1000000000)
        return 0;
    else
        while (r) sum += r --;
```

```
    return sum;
}
```

就像 while 语句的循环体一样，如果组成 if 语句的那两个语句由多条组成，则必须使用复合语句。尽管在这里并不需要使用复合语句，但那些讲究程序设计规范的人坚持认为上述 if 语句应该写成这样：

```
    if (r > 1000000000)
    {
        return 0;
    }
    else
    {
        while (r)
        {
            sum += r --;
        }
    }
```

他们的理由听起来也颇有道理：如果程序的功能需要扩充，那么这种格式修改起来十分方便。问题在于，把简单的流程写得令人眼花缭乱可能违背了使程序易读的初衷。

从语法上来看，if 语句的 else 子句是可选的，如果不需要可以省略。在函数 cusum 的以下版本中，if 语句省略了 else 子句，但程序的功能没有变化。

```
unsigned long long int cusum (unsigned long long int r)
{
    unsigned long long int sum = 0;

    if (r > 1000000000) return 0;

    while (r) sum += r --;

    return sum;
}
```

显然，在这里，if 语句的作用仅仅是在变量 r 的值大于 1000000000 时，将控制直接返回到函数的调用者；否则，控制表达式的结果为 0，于是离开 if 语句，直接往后执行 while 语句，但 while 语句并不是 if 语句的组成部分。

如果希望 while 语句成为 if 语句的一部分（子句），则可以修改 if 语句的控制表达式，把它改成 r <= 1000000000，如以下新版的 cusum 函数所示。

```
unsigned long long int cusum (unsigned long long int r)
{
    unsigned long long int sum = 0;

    if (r <= 1000000000) while (r) sum += r --;

    return sum;
}
```

这里，如果变量 r 的值小于等于 1000000000，则控制表达式 r <= 1000000000 的

值不为 0，这将会执行后面的 while 语句。否则，程序的执行离开 if 语句。

3.10.2 认识逻辑或运算符

原则上，函数 cusum 的参数值应该大于 0，毕竟是从 1 开始相加嘛。但如果调用者传递的参数值为 0，那也不会出什么乱子，因为函数 cusum 恰好能够在参数 r 的值为 0 时返回 0。你可以试着阅读前面的代码，或者动手在调试器里观察一下。

当然，如果不怕麻烦，你也可以将变量 r 的值是否为 0 作为判断条件之一。如以下新版的 cusum 函数所示，我们在 if 语句里加入了两个判断条件。

```
unsigned long long int cusum (unsigned long long int r)
{
    unsigned long long int sum = 0;

    if (r == 0 || r > 1000000000)
        return 0;
    else
        while (r) sum += r --;

    return sum;
}
```

非常明显地，if 语句的控制表达式里出现了一个新的运算符“||”，称为逻辑或运算符。注意，这两个竖线必须挨在一起写。逻辑或运算符是二元运算符，有一左一右两个操作数，例如 5 || 6，这样的表达式称为逻辑或表达式。

逻辑或运算符的功能是对两个操作数的值进行逻辑上的相加操作，也就是生活中的“或者”，两者居其一的意思。因此，它的结果是这样决定的：如果左操作数和右操作数的值都是 0，则逻辑或运算符的结果（或者说逻辑或表达式的值）也是 0；否则，如果左操作数和右操作数的值都不为 0，或者至少有一个不为 0，则逻辑或运算符的结果（或者说逻辑或表达式的值）是 1。举例来说，表达式 0 || 0 的值是 0；表达式 0 || 1 的值是 1；表达式 1 || 0 的值是 1；表达式 0 || 9 的值是 1；表达式 5 || 6 的值也是 1。

逻辑或运算符||的优先级低于==和>。因此，在上述代码中，运算符||的操作数分别是 r > 1000000000（的值）和 r == 0（的值），即，表达式 r == 0 || r > 1000000000 等价于(r == 0) || (r > 1000000000)。

表达式 r == 0 || r > 1000000000 的意思很清楚：变量 r 的值要么为 0，要么大于 1000000000，两者可能都不成立，也可能有一个是成立的。如果都不成立，则该表达式等效于 0 || 0；如果前者成立而后者不成立，则该表达式等效于 1 || 0；如果前者不成立而后者成立，则该表达式等效于 0 || 1。在后两种情况下，整个逻辑或表达式的值为 1，将执行 if 语句的第一个子句：

```
return 0;
```

否则，在第一种情况下，也就是该表达式等效于 0 || 0 的情况下，意味着变量 r 的值既不为 0，也不大于 1000000000，整个逻辑或表达式的值为 0，执行 else 子句：

```
while (r) sum += r --;
```

3.10.3 未定义的行为

值得注意的是，逻辑或表达式的求值具有短路效应。这是什么意思呢？逻辑或表达式求值时，是先求值运算符||的左操作数，如果左操作数的值不为 0，则不再求值右操作数，因

为这样做是多余的。只有在左操作数的值为 0 时，才会继续求值右操作数。

非但如此，C 语言还规定，如果运算符||的右操作数会被求值（这意味着左操作数求值的结果为 0），则在其左操作数的求值和右操作数的求值之间存在一个序列点。换句话说，在求值右操作数之前，左操作数的值计算和副作用已经全部完成。

来看一个例子，如果标识符 n 被声明为一个整数类型的变量，则表达式 n ++ || n 求值时，只有在表达式 n ++的值计算和副作用已经完成，且该表达式的值为 0 时，才开始求值表达式 n。

如果在表达式 n ++ || n 求值前，变量 n 的当前值是 0，而且我们把表达式 n 的值计算记为 V_n，把表达式 n ++的值计算记为 V_{++}，副作用记为 S_{++}，整个逻辑或表达式的值（逻辑或运算符的结果）记为 $V_{||}$，则该表达式的求值过程如下：

$V_{++}(0) \rightarrow S_{++}(1) \rightarrow V_n(1) \rightarrow V_{||}(1)$

这里，是先求值左操作数 n ++，且值计算和副作用都已经完成（已经把 1 作为新值写入变量 n），当右操作数求值时，左值转换的结果是刚刚写入的 1。因为这两个操作数求值后的值一个为 0 一个为 1，整个逻辑或表达式的值为 1。

注意，因为序列点的存在，上述求值过程是唯一的，不存在其他可能性。如果没有上述序列点的保证，则这两个表达式的求值将有可能交错进行，其最终结果无法预料。比如说它可能是这样的：

$V_n(0) \rightarrow V_{++}(0) \rightarrow V_{||}(0) \rightarrow S_{++}(1)$

显然，因为逻辑运算符||的左操作数和右操作数在求值后都是 0，所以整个逻辑或表达式的值也为 0。

作为一门编程语言，C 语言的规范描述了程序结构、语法元素、表达式、语句，定义了它们的形式和操作，规定了操作数的类型和范围，同时也描述了可预期的运行结果。但是对于不遵循规范的程序设计，C 语言没有，也无法限定和描述程序的运行结果。在这种情况下，程序的行为是无法预料的，计算结果可能碰巧是正确的，也可能是错的，程序可能会崩溃，等等，不一而足，这些无法预料的行为，称为未定义的行为。

来看另一个例子，在下面的代码片断中，表达式 m = m ++的求值就是未定义的行为，求值完成后，变量 m 的存储值不能确定，取决于不同的翻译器如何安排求值过程。

```
int m = 0;
m = m ++;
```

表达式 m = m ++的求值具有两个副作用，分别是运算符++的副作用和运算符=的副作用，但都是修改变量 m 的存储值，这就很特殊了。这两个副作用哪个在前哪个在后，C 语言并未规定，要由翻译器自主决定。这就是说，该表达式求值的行为是未定义的。因为这个原因，该表达式求值完成后，变量 m 的值可能是 0，也可能是 1。

在任何一个用 C 语言编写的程序中，很多行为是良好定义的，比如 C 语言规定在全表达式的求值之间有序列点，在运算符||的左操作数和右操作数的求值之间存在序列点。正是有了特殊规定，表达式 n ++ || n 的求值不存在未定义的行为。

练习 3.9：

1.我们已经讲过，赋值表达式的值是赋值运算符的左操作数被赋值之后的值，赋值运算符的副作用发生在赋值运算符左右操作数的值计算（而不是求值）之后。依据这一规定，请说明表达式 n = n + 1 和 sum = sum + n 不存在未定义的行为。

2.若 y 是整数类型的变量，判断表达式 y = (y = 0) + 3 的求值是否存在未定义的行为。

3.10.4 摇摆的 else 子句

依据语法，一个 if 语句是由关键字“if”、圆括号、表达式和其他语句组成，这里的“其他语句”可以是任何语句，包括另一个 if 语句。

如果 if 语句的子句也是一个 if 语句，那可就花哨了。为了演示这种情况，我们编写了一个新版的 cusum 函数，它里面的 if 语句就是这种情况。在这个新版的函数里，if 语句首先检查变量 r 的值是否为零，如果为零则直接结束函数的执行并返回到调用者；否则的话，它的 else 子句检查 r 的值是否大于 1000000000。如果条件成立，则也结束函数的执行，将控制返回到调用者；如果不成立，则执行 while 语句，开始累加过程。

```
unsigned long long int cusum (unsigned long long int r)
{
    unsigned long long int sum = 0;

    if (r == 0) return 0;
    else
        if (r > 1000000000) return 0;
        else while (r) sum += r --;

    return sum;
}
```

在这里有两个 if 语句，但第二个 if 语句是第一个 if 语句的组成部分。如图 3-4 所示，第一个被框住的 return 语句是第一个 if 语句的第一个子句；第二个被框住的部分尽管也是 if 语句，但它整体上是第一个 if 语句的 else 子句。

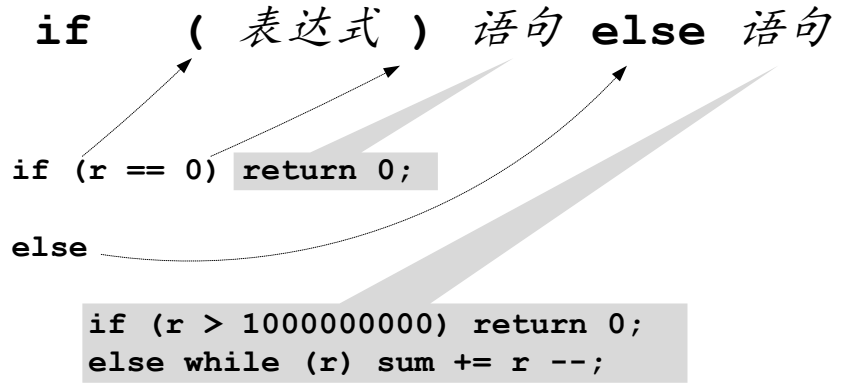


Fig.3-4 规范的 if 语句在语法上是没有歧义的

但是，如果 if 语句嵌套不当，则容易使人迷惑。比如下面的例子，你来说说，里面的 else 子句到底属于第一个 if 语句，还是属于第二个 if 语句？

```
unsigned long long int cusum (unsigned long long int r)
{
    unsigned long long int sum = 0;

    if (r > 0)
        if (r <= 1000000000)
            while (r) sum += r --;
    else
        sum = 0;
```

```
    return sum;
}
```

尽管从排版上来看，这个 else 子句属于第一个 if 语句，但真实的情况与排版无关。如图 3-5 所示，虽然被框住的部分是一个带有 else 子句的 if 语句，但它可以被认为是顶上那个 if 语句的子句（它没有 else 子句）。

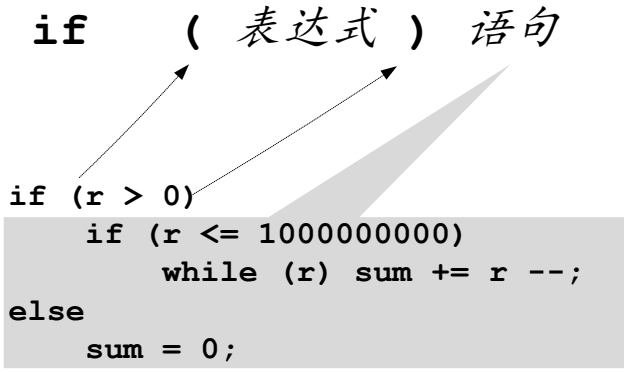


Fig.3-5 有歧义的 if 语句，这是它的第一种解读方式

然而，如图 3-6 所示，我们也可以认为 else 子句属于第一个 if 语句，第二个 if 语句没有 else 子句，且属于第一个 if 语句的第一个子句。

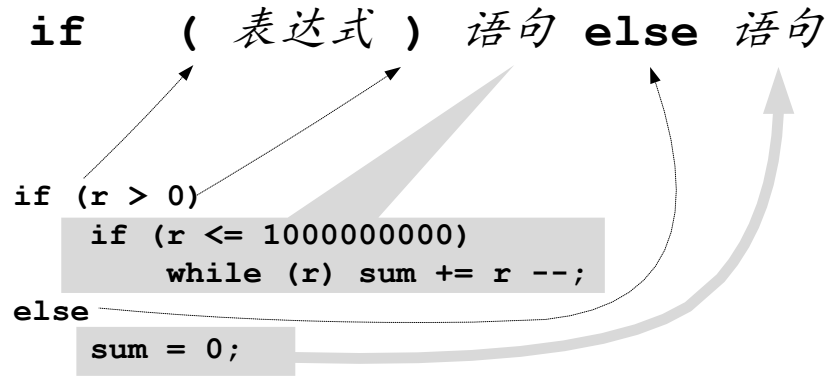


Fig.3-6 有歧义的 if 语句，这是它的第二种解读方式

那么，到底哪种隶属关系才是正确的呢？C 语言规定，一个 else 子句从属于离它最近的那个 if 语句。因此，这个 else 子句实际上是第二个 if 语句的一部分（这就是说图 3-5 才是正确的）。如果这并不是你所期望的，那就应该使用复合语句来明确隶属关系：

```
if (r > 0)
{
    if (r <= 1000000000)
        while (r) sum += r --;
}
else
    sum = 0;
```

3.10.5 认识逻辑与运算符

和逻辑或运算符||相对应的是“逻辑与”运算符&&，它表达逻辑上的相乘关系，对应于生活中的“并且”。和逻辑或运算符一样，逻辑与运算符&&需要一左一右两个操作数，例

如 1 && 0 和 5 && 6。

逻辑与运算符&&的结果是这样决定的：如果左操作数和右操作数的值都不为 0，则该运算符的结果是 1；否则，在任何其他情况下，该运算符的结果都为 0。因此，表达式 0 && 0 的值是 0；表达式 0 && 9 的值是 0；表达式 5 && 6 的值是 1。以下版本的 cusum 函数就使用了逻辑与运算符。

```
unsigned long long int cusum (unsigned long long int r)
{
    unsigned long long int sum = 0;

    if (r != 0 && r <= 1000000000)
    {
        while (r) sum += r --;
        return sum;
    }

    return 0;
}
```

以上，在 if 语句的控制表达式里，运算符<=的优先级最高，!=次之，&&最低。因此运算符&&的操作数分别是 r != 0 和 r <= 1000000000，即，表达式 r != 0 && r <= 1000000000 等价于(r != 0) && (r <= 1000000000)。顺便说一句，运算符&&的优先级高于运算符||。

显然，表达式 r != 0 && r <= 1000000000 所描述的意思是“变量 r 的值不为 0 而且小于等于 1000000000”。如果变量 r 的值符合这个条件，比如为 2，则子表达式 r != 0 描述的关系成立，运算符!=的值是 1；子表达式 r <= 1000000000 所描述的关系也成立，运算符<=的值也是 1，于是运算符&&的值是 1，执行 if 语句的第一个子句（复合语句）。这个子句首先完成累加过程，然后直接用 return 语句结束当前函数的执行，将控制返回到调用者，返回值是累加后的结果。

如果 if 语句的控制表达式计算出一个零值，则程序的执行直接离开 if 语句，执行后面的

```
return 0;
```

这就与前面不一样了。先前版本的 cusum 函数在最后都返回表达式 sum 的值，但这次呢，只有在传入的参数值不符合条件时才会执行最后的这个 return 语句，所以它就应该返回一个零值。

逻辑与表达式的求值具有短路效应，它总是先求值运算符&&的左操作数，如果左操作数的值为 0，则不再求值右操作数，因为这样做是多余的。只有在左操作数的值为 1 时，才会继续求值右操作数。

C 语言规定，如果运算符&&的右操作数会被求值（这意味着左操作数求值的结果为 1），则在其左操作数的求值和右操作数的求值之间存在一个序列点。换句话说，在求值右操作数之前，左操作数的值计算和副作用已经全部完成。

这就是说，如果标识符 n 被声明为指示一个整数类型的变量，则表达式 n ++ && n 的求值不是未定义的行为。因为，在求值表达式 n 时，表达式 n ++的值计算和副作用已经完成，表达式 n 的值是递增之后的新值；在求值表达式 n ++时，表达式 n 的求值还没有开始。

3.11 认识标号语句和 goto 语句

在 C 语言里，有很多方法可以改变语句的执行顺序和流程，比如 while 语句、if 语句和 return 语句。while 语句可以反复执行一段代码，if 语句可以选择不同的分支，return 语句可以立即返回到函数的调用者。

在函数 cusum 的以下版本中，我们没有使用 while 语句，而是用新的语句类型来替代它的功能，它们分别是标号语句和跳转语句。

```
unsigned long long int cusum (unsigned long long int r)
{
    unsigned long long int sum = 0;

    if (r == 0 || r > 1000000000) return 0;

again:
    sum += r;
    r --;
    if (r) goto again;

    return sum;
}
```

说起来，跳转语句是你的老朋友了，因为跳转语句有好几种，而 return 语句就是其中之一。我们现在所用的，是另一个跳转语句，也就是 goto 语句，它可以在包含它的函数内跳来跳去，其语法为

goto *标识符* ;

goto 语句由关键字“goto”、标识符和分号“;”组成，既然是跳转，那肯定得有一个目标，这里的标识符用于指示跳转的目的地。在以上函数中，语句

```
goto again;
```

就是跳转语句，标识符 again 用于指定跳转的目标位置。反过来，为了指定跳转的目标，需要在目标语句前做一个记号或者标记，我们称之为标号。这个标号和它后面的语句一起，共同组成一种新的语句类型：标号语句。因此，标号语句的语法为

标识符 : *语句*

这就是说，由标识符、冒号“:”和语句组成的新语句，称为标号语句。在以上函数中，语句

```
again:
    sum += r;
```

就是标号语句。

现在我们把这两种语句连缀起来，看看它们在以上函数里是如何运作的。标号本身不影响程序的正常执行，就像它不存在一样。我们首先判断变量 r 的值，如果为 0，或者大于 1000000000，则直接返回一个 0 给它的调用者；如若不然，则将变量 r 的当前值加到变量 sum，然后将 r 的存储值递减。显然，标号的存在不影响程序的正常执行。

在接下来的 if 语句中，判断变量 r 的存储值是否已经递减到 0，如果不为 0，则跳转到标号 again 处执行。标号语句执行完后，将继续往下执行其他语句，直至又一次来到 if 语句，再次判断变量 r 的值。如果变量 r 的值依然不为 0，则再次跳转；如果为 0，则不执行 goto 语句，且离开 if 语句往下面执行。

相比之下，为了使代码简洁，我更愿意使用逗号表达式。在函数 cusum 的以下版本中，我们把 if 语句的控制表达式改了一下，但程序的功能不变。我们在前面已经把逗号表达式

讲得很清楚了，这里不再重复，请读者自行分析。

```
unsigned long long int cusum (unsigned long long int r)
{
    unsigned long long int sum = 0;

    if (r == 0 || r > 1000000000) return 0;

again:
    if (sum += r --, r) goto again;

    return sum;
}
```

显然，我们这里的标号语句是：

```
again:
    if (sum += r --, r) goto again;
```

但这个标号语句包含了一个 goto 语句。

根据语法可知，标号和冒号“:”之后只能是语句（单条语句或者复合语句），而不能是声明。所以，在以下版本的 cusum 函数中包含了一个错误，即，标号语句的冒号“:”之后不是语句，而是一个声明。

除此之外，函数的其他部分没有什么问题，而且程序的工作流程是清晰的：先是判断变量 r 的值，如果大于 0 且小于 1000000000 则跳转到标号 next 处执行，否则就直接返回 0 到函数的调用者。

在标号 next 处，声明了变量 sum 并将它初始化为 0，然后使用 while 语句完成累加过程，最后返回累加和（也就是变量 sum 的存储值）到调用者。当然，我们已经说过，冒号之后不应该是声明，所以在程序的翻译阶段，翻译器将报告错误并停止翻译。

```
unsigned long long int cusum (unsigned long long int r)
{
    if (r > 0 && r <= 1000000000) goto next;

    return 0;

next:
    unsigned long long int sum = 0;
    while (r) sum += r --;

    return sum;
}
```

要使这个函数能够正常通过翻译，就得把冒号后面的声明改为语句。说起来简单得难以置信，只需要在冒号后面加一个分号“;”，也就是添加一个空语句，就可解决。以下是修改后的 cusum 函数。

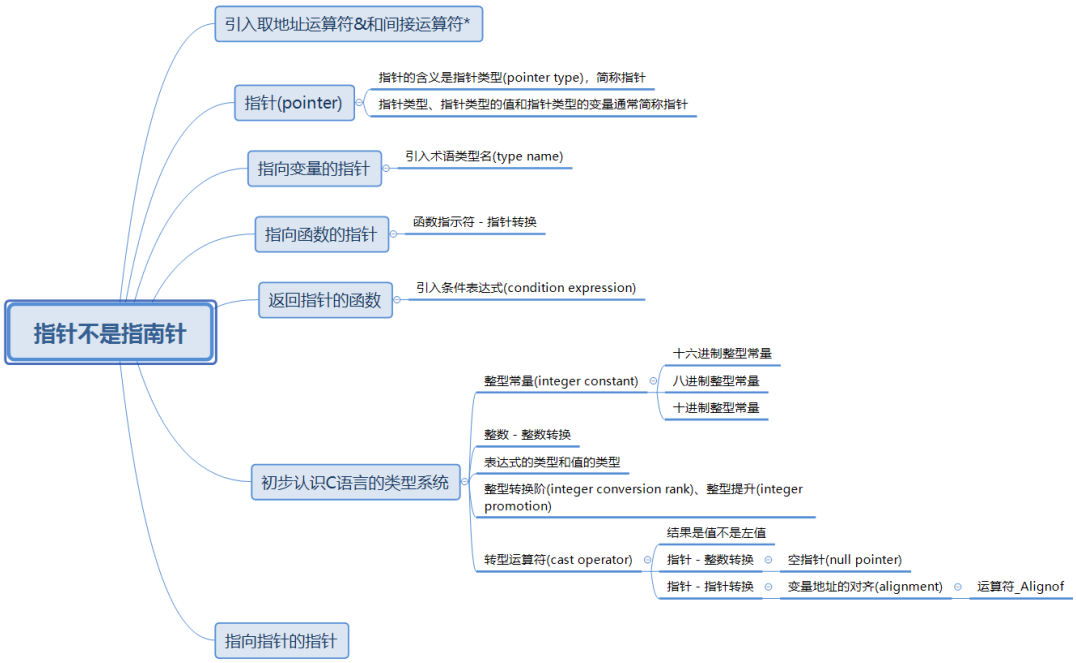
```
unsigned long long int cusum (unsigned long long int r)
{
    if (r > 0 && r <= 1000000000) goto next;
```

```
    return 0;

    next:
        ;
        unsigned long long int sum = 0;
        while (r) sum += r --;

    return sum;
}
```

另一个稍微麻烦一点的办法是使用复合语句，也就是用一对花括号将声明围起来，或者干脆连 while 语句也围在一起。

练习 3.10：

1. 上面已经说了，另一个办法是使用复合语句，请自己上机尝试一下。
2. 使用这几章已经学过的知识，编写一个计算阶乘的程序，使其可以计算 10 以内的正整数的阶乘，至少给出 10 种写法。


指针不是指南针
引入取地址运算符&和间接运算符*
指针(pointer)
指针的含义是指针类型(pointer type)，简称指针
指针类型、指针类型的值和指针类型的变量通常简称指针
指向变量的指针
引入术语类型名(type name)
指向函数的指针
函数指示符 - 指针转换
返回指针的函数
引入条件表达式(condition expression)
初步认识C语言的类型系统
整型常量(integer constant)
十六进制整型常量
八进制整型常量
十进制整型常量
整数 - 整数转换
表达式的类型和值的类型
整型转换阶(integer conversion rank)、整型提升(integer promotion)
转型运算符(cast operator)
结果是值不是左值
指针 - 整数转换
空指针(null pointer)
指针 - 指针转换
变量地址的对齐(alignment)
运算符_Alignof
指向指针的指针

第 4 章　指针不是指南针

我们知道，一个变量就是存储器里的一个存储区。这里所谓的存储器，包括我们平时所说的内存和处理器内部的寄存器。在学习计算机原理这门课的时候我们知道，除非变量位于寄存器中，否则就必定有一个地址，处理器正是通过地址来访问它们。当然，包括 C 在内的各种高级语言屏蔽了这些细节。

然而，如果需要，C 语言也允许你通过地址来定位一个变量，并对它进行各种操作。这种方式虽然看起来有些迂回，但却是非常强悍的功能，你很快就能有所体会。

4.1 认识一元&和一元*运算符

虽然一个变量就是存储器里的一个存储区，但是，只有当程序真正运行时，这个存储区才实打实地确定下来。在我们写程序的时候，还只是假装它已经存在，还要用语句来模拟那些运行时才会执行的操作。

因为写程序的时候变量还不存在，自然需要用标识符来代表它，因此，标识符就是变量的化身，代表着那个变量，就好比我们每个人都有一个名字。在下面的程序里，我们声明了一个变量，标识符 m 是那个变量的名字，代表那个变量；语句 S1 中的表达式 m = 1 是直接操作这个变量，为它塞一个数值。

```
/*************c0401.c************/
int main (void)
{
    int m, w;

    m = 1;                  //S1
    * & m = 2;              //S2，等价于 m = 2
    w = ++ * & m;           //S3，等价于++ m
}
```

这种操作是很直接的，并不涉及变量的地址，因为你“正在操作变量本身”，就像你正在面对面地塞给朋友一个苹果，并不需要知道他住在哪里。标识符 m 可以看成那个朋友的名字，代表那位朋友本身，数值 1 可以看成苹果。那么，如果我想通过变量的地址来操作它，怎么办呢？

在 C 语言里有哼哈二将，一个是一元&运算符，另一个是一元*运算符，它们被称为一元运算符，是因为它们只需要一个右操作数。一元&运算符用于取得变量或者函数的地址，而一元*运算符则根据一个地址来得到那个变量或者函数本身。作为例子，上面那个程序就演示了如何得到一个变量的地址，以及如何通过地址得到一个变量本身。

以上，在 main 函数内声明了变量 m 和 w。语句 S1 我们并不陌生，是将数值 1 保存到变量 m。如图 4-1 所示，这是针对变量 m 本身，是非常直接的操作，就像面对面给别人一个苹果：“来，缪晓红同学，给你一个苹果”。

一元*运算符和一元&运算符的优先级相同，且都高于赋值运算符，但它们是从右往左结合的，所以在语句 S2 中，表达式* & m = 2 等价于(* (& m)) = 2。表达式& m 得到变量 m 的地址，然后，一元*运算符则通过地址得到该地址上的那个变量——实际上就是变量 m。

“变量”不是用于描述表达式的术语，“左值”才是。指示或者说代表变量的表达式称为左值，既然表达式* & m 代表一个变量，那么它就是一个左值；表达式* & m = 2 是将数值 2 赋给这个左值。又因为这个左值实际上是代表变量 m 的，所以语句 S2 等价于 m = 2。

如图 4-1 所示，通过地址来找到变量，或者换句话说，通过一元运算符*来得到一个左值的过程相对迂回了些。就像通过快递把苹果发给缪晓红，这需要通过地址进行。

既然表达式* & m 是一个左值，代表一个变量，那么，它自然也可以作为前缀递增运算符的操作数，因为该运算符要求它的操作数必须是左值。

在语句 S3 中，表达式++ * & m 将递增左值* & m 所代表的那个变量的存储值，因为该左值实际上代表变量 m，故变量 m 的存储值现在是 3，该表达式等价于++ m。

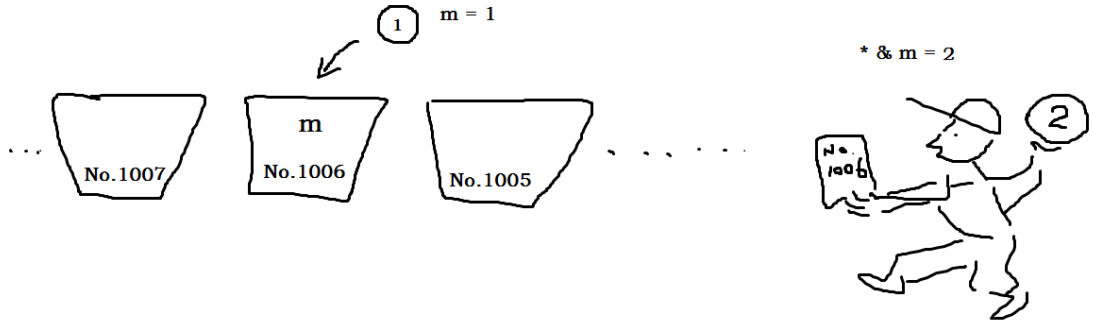


Fig.4-1 变量操作和指针操作示意图

为了加深理解，最好的办法是在调试器内观察语句的执行和变量值的变化。现在，我们将上述程序保存为源文件 c0401.c 并翻译为可执行文件，启动 GDB，将断点设置在源程序的第 6 行。接着，用 r 命令运行程序到断点位置（以 WINDOWS 平台为例）：

```
D:\examples>gcc c0401.c -o c0401.exe -g

D:\examples>gdb c0401.exe -silent
Reading symbols from c0401.exe...done.
(gdb) l
1       int main (void)
2       {
3           int m, w;
4
5           m = 1;
6           * & m = 2;
7           w = ++ * & m;
8       }
(gdb) b 6
Breakpoint 1 at 0x401564: file c0401.c, line 6.
(gdb) r
Starting program: D:\examples\c0401.exe
[New Thread 12564.0x2d10]
[New Thread 12564.0xb34]

Thread 1 hit Breakpoint 1, main () at c0401.c:6
6           * & m = 2;
```

现在，程序中的第 5 行已经执行，变量 m 被赋值为 1，GDB 显示下一次将要执行第 6 行。既然表达式&m 用于取得变量 m 的地址，那为何不看看它到底是啥呢：

```
(gdb) p &m
```

```
$1 = (int *) 0x61fe48
(gdb) p *&m
$2 = 1
```

以上，我们用命令 p &m 来显示这个地址，显示的内容是 0x61fe48，这个十六进制的数字就是变量 m 的内存地址。既然表达式*&m 是一个左值，代表一个变量（在这里实际上是变量 m），我们也用命令 p *&m 显示了该变量的值，其值为 1，是我们刚才赋的值。

接下来，我们用 n 命令执行第 6 行，其功能是把数值 2 保存到左值*&m 所代表的变量里。然后，用命令 p {m, *&m}来显示变量 m 的值，以及左值*&m 所代表的变量的值：

```
(gdb) n
7           w = ++ * & m;
(gdb) p {m, *&m}
$3 = {2, 2}
```

如上所示，我们知道，左值*&m 所代表的变量实际上就是变量 m，而变量 m 的当前值是 2，所以显示的结果应该是两个 2，GDB 显示的结果证实了我们的猜测。

下面继续用 n 命令往下执行第 7 行，这将求值表达式++ * & m 并把结果赋值变量 w。执行后 GDB 停留在第 8 行的右花括号处。此时，我们再用命令 p {m, *&m, w}来显示变量 m、表达式*&m 所代表的变量，以及变量 w 的值：

```
(gdb) n
8       }
(gdb) p {m, *&m, w}
$4 = {3, 3, 3}
```

因为表达式++ * & m 递增左值*&m 所代表的那个变量的存储值，故如上所示，变量 m 的值和左值*&m 所代表的那个变量的值都是 3。运算符++的结果是其操作数递增之后的值，所以表达式++ * & m 的值是 3。这个结果赋给了变量 w，所以变量 w 的值也是 3。

最后，我们再用 c 命令执行程序直至它正常退出，然后用 q 命令退出 GDB，本次调试过程结束：

```
(gdb) c
Continuing.
[Inferior 1 (process 12564) exited normally]
(gdb) q

D:\examples>
```

在第一章里引入“左值”这个概念的时候，很多人可能还不理解它的必要性。现在应该很清楚了，表达式*&m 代表一个变量，但我们不能说“变量*&m”，这通常是不恰当的。

练习 4.1：

判断题：

(1)通过变量的地址可以得到该地址上的变量（ ）。

(2)若 var 是变量，则表达式* & var 等价于左值 var。（ ）

4.2 什么是指针

简单地将一元&运算符的结果视为地址，是有问题的。地址相当于门牌号码，在有些人的眼里，它也许类似于一个整数。然而仅凭一个地址，我们是无法访问数据的。想想看，地址不能告诉你应该以什么类型来读取和解释那个变量的内容，是 signed char，还是 unsigned int？进一步地，因为没有类型信息，翻译软件在翻译程序时，也不知道那个

地址上的变量有多大，应该读取或写入一个字节呢，还是四个字节？

所以，在 C 语言里，一元&运算符的结果（值）并不单纯是一个地址，而是一个包含了类型信息的地址。如果变量 m 的类型是 int，则表达式& m 的值不单包含了地址信息，还将包含这样的信息：这个地址用于访问一个 int 类型的变量。否则的话，表达式* & m = 2 和++ * & m 不能正确执行，因为单纯的地址缺乏类型信息。

那么，如何描述这样的值呢？如图 4-2 所示，从表达式& m 计算出一个值，这个值像指针一样，指向那个地址上的变量。鉴于这种类比特别形象，C 语言引入了指针类型，简称指针，并规定一元&运算符的结果（值）是指针类型。

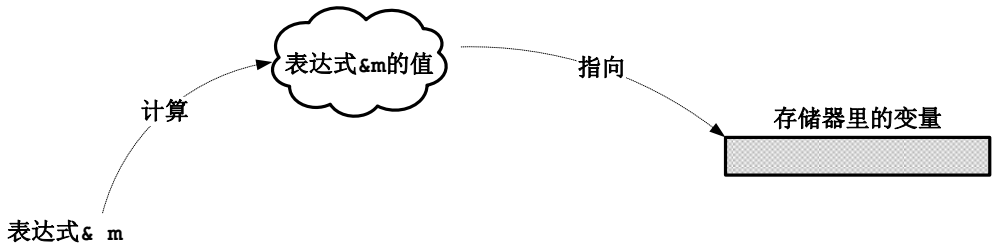


Fig.4-2 指针的含义

在 C 语言里，整数类型是个统称，实际上包含了 char、short 和 long int 等具体的类型。同样地，指针类型也是个统称，根据它所指向的类型，可细分为指向 char 的指针、指向 short 的指针、指向 long int 的指针，等等。

一元&运算符称为取地址运算符，它需要一个右操作数，而且必须是左值或者函数指示符。这是很自然的，只有内存中的变量，或者函数才有地址，你不能取一个常量的地址，比如& 250，这很荒唐。另外需要注意，如果一个左值是一元&运算符的操作数，则不执行左值转换。

为方便起见，人们经常把指针类型的值也叫作“指针”。不过这不会引起混淆，通过具体的上下文大家都能明白所指。比如说，C 语言规定，**如果操作数的类型为 *T*，则一元&运算符的结果是指向 *T* 的指针**。在这里，“指针”就是“指针类型的值”。

在源文件 c0401.c 中，因为变量 m 的类型是 int，所以一元&运算符的结果（也就是表达式& m 的值）是指向 int 的指针。当然了，如果 m 的类型是 char，则一元&运算符的结果是指向 char 的指针。

反过来，为了还原指针所指向的变量和函数，需要使用一元*运算符。C 语言规定，一元*运算符的操作数必须是一个指针（类型的值）。**如果操作数是指向某变量的指针，则一元*运算符的结果是个左值，代表那个变量；如果操作数是指向函数的指针，则一元*运算符的结果是函数指示符，代表那个函数；如果操作数的类型是指向 *T* 的指针，则一元*运算符的结果类型为 *T*。**

因此，在上面的程序中，因为表达式& m 的值是指向 int 的指针，故表达式* & m 的结果是得到一个 int 类型的左值，代表指针所指向的变量（实际上是变量 m）。

4.3 指针类型的变量

既然指针是一种数据类型，那么，我们应该可以声明这种类型的变量，这样就可以保存指针类型的值以方便传递和使用。没错，这当然是可以的。不过，在进入这一话题前，还是让我们先来回忆一下整数类型的变量声明。对于声明

```
int m;
```

我们是这样来解读的：先从标识符 m 开始，读作“标识符 m 的类型是 int”，或者更经常地，读作“m 是 int 类型的变量”，这个变量只用来保存和读出 int 类型的值。

相应地，要把一个指针类型的值保存在一个变量里，这个变量的类型也应该是指针类型才可以。而且，变量的类型必须和指针的类型一致，要保存一个指向 char 类型的指针，变量的类型也必须是指向 char 的指针。再比如，如果变量的类型是指向 int 的指针，则它应该这样声明：

```
int * p;
```

如图 4-3 所示，这照例要从标识符开始认起。因为标识符 p 的右边是分号“;”，所以要向左读。左边是一个星号“*”，所以读作“p 的类型是指针”或者“p 是一个指针”。

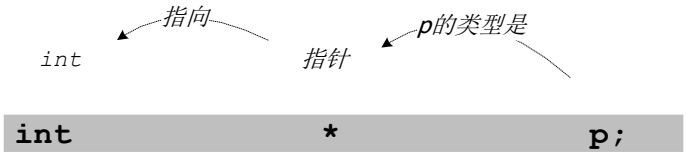


Fig.4-3 声明一个变量，其类型为指向变量指针

对于指针类型来说，指向的类型最为关键，如果指向的类型不同，则它们是不同的指针类型。因此，如图中所示，既然 p 的类型是指针，那么下一步就是将指针所指向的类型读出来。因为标识符 p 右边只是分号“;”，那我们继续向左读，左边是类型指定符“int”，所以读作“指向 int”。

现在，我们把这个过程连起来，就可以说“p 是一个指针，该指针指向 int”，或者说“p 是一个指向 int 的指针”，又或者说“变量 p 的类型是指向 int 的指针”；或者干脆简单地说“p 是一个指针类型的变量”。

然而为了方便，人们更经常地把指针类型的变量叫作指针，相对于“p 是一个指针类型的变量”，人们更喜欢说“p 是一个指针”。在具体的上下文中，这通常不会引起混淆，多数教材和图书都是这么用的，但本书尽量避免这种叫法。

在程序中，我们会声明很多变量和函数，但如果要问起它们是什么类型，类型的名字是什么，该怎么描述呢？为了方便描述复杂的类型，C 语言引入了类型名。所谓类型名，顾名思义，就是类型的名字。要想得到一个类型的类型名，需要先在纸上或者脑海里构造一个变量或者函数的声明，然后去掉标识符和末尾的分号，剩下的部分就是类型名。给定声明：

```
unsigned int var;
signed char * pc;
_Bool do_sth (signed char, signed char);
```

要想指出标识符 var、pc 和 do_sth 的具体类型是什么（或者说它们是什么类型），可以用文字描述，但这样做很啰唆，用类型名比较简洁直观。

为了得到类型名，只需要把这三个标识符从它们的声明中去掉，末尾的分号也去掉，剩下的部分就是类型名。因此我们说变量 var 的类型是 unsigned int；变量 pc 的类型是 signed char *；函数 do_sth 的类型是_Bool (signed char, signed char)。

为了说明如何使用指针类型的变量，下面给出一个示例程序。在程序中，我们声明了三个变量，一个是变量 x，其类型为 int；另外两个分别是变量 p 和 q，它们的类型都是指向 int 的指针。

注意星号“*”的位置，它是独立的，意思是“指针”，不和类型指定符 int 结合，也不和标识符 p、q 结合，但它是声明符的一部分。我们曾经提到过声明符，声明符用于描述被声明的实体，在这里，x 是声明符，* p 和* q 也是声明符。带有星号的 p 和 q 意味着实体是指针，不带星号则意味着实体不是指针。

```
/**********c0402.c********/
int main (void)
```

```
{
    int x, * p = & x, * q;

    q = & x;
    * p = 1;
    x = * p + * q;
}
```

在这里，变量 p 的声明中带有一个初始化器，即表达式& x。因为 x 是一个 int 类型的左值，所以表达式& x 的值是指向 int 的指针，这个指针由变量 x 的地址转化而来。而 p 的类型呢，也是指向 int 的指针，类型匹配。

现在，变量 p 的值是一个指针，实际上是指向变量 x 的，但变量 q 没有初始化，它的内容是随机的，不被认为是一个有效的指针。不过没关系，语句

```
q = & x;
```

就是用来给变量 q 赋值的。变量 q 的类型是指向 int 的指针，而表达式& x 的类型也是指向 int 的指针，两边类型一致，可以赋值。赋值之后，变量 q 的值指向变量 x。换句话说，如图 4-4 所示，变量 p 和 q 的值都是指向变量 x 的指针。

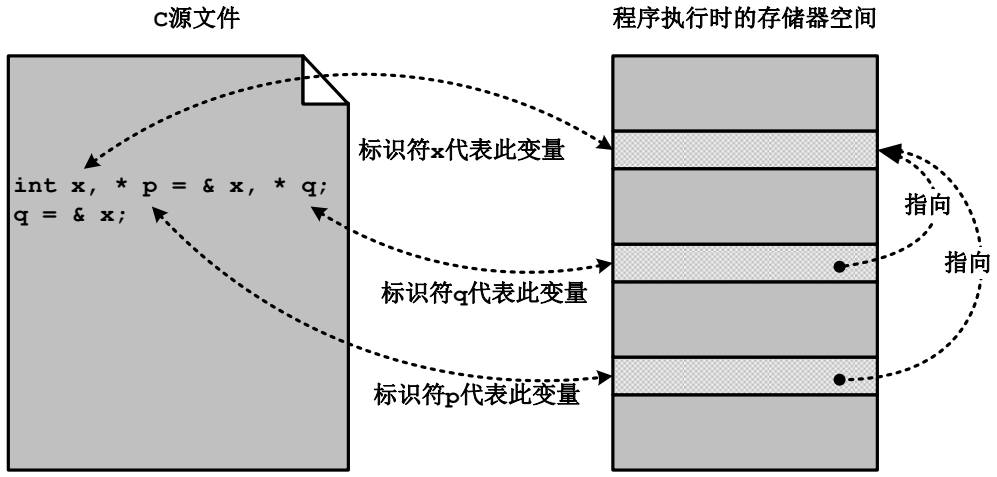


Fig.4-4 指针类型的变量及其值所指向的变量

再来看语句

```
* p = 1;
```

表达式* p 中的 p 是一个左值，要执行左值转换，转换为变量 p 的存储值，这是一个指针（类型的值）。因为一元*运算符的操作数是指针，故表达式* p 的结果是一个左值[2]，代表一个变量，但因为它是赋值运算符=的左操作数，故不再进行左值转换。所以表达式* p = 1 是把 1 赋给左值* p，也就是把 1 保存到左值* p 所代表的那个变量里。

C 语言的一个特点是变量和函数的声明与它们在程序中的用法在形式上一致。在变量 p 的声明中，星号“*”是一个符号，意思是“指针”，“* p”的意思是 p 是一个指针；而在表达式中，星号“*”是一个运算符，“* p”是一个表达式，用于间接地通过变量 p 的值来得到它所指向的变量。

再往下看另一条语句

```
x = * p + * q;
```

在表达式* p 和* q 中，p 和 q 都是左值，需要先进行左值转换，转换为变量 p 和 q 的存储值。左值转换后的值都是指针，所以表达式* p 和* q 又都是左值，都要继续进行

[2] 为方便起见，可直接称之为“左值* p”。本书后面的行文中，类似的情况也照此办理 。

左值转换，转换为它们所代表的变量的存储值。然后，将这两个值相加，并赋给左值 x。上述语句执行后，变量 x 的值为 2。

为了加深理解，我们再换一种说法：左值 p 和 q 经左值转换后的值都是指向变量 x 的指针，所以表达式* p 和* q 的结果都是左值，都代表变量 x。这两个左值经左值转换，转换为变量 x 的存储值，相加后赋给左值 x。因为这个原因，上述语句实际上等价于：

```
x = x + x;
```

练习 4.2：

1. 给定声明：

```
int m, * p, f (void);
```

请指出其中的声明符。

2.拓展训练：在调试本节的程序时，可使用“p p”命令打印变量 p 的值（也就是变量 x 的地址），也可以使用“p &p”打印变量 p 自身的地址；还可以使用“p *p”命令打印 p 的值所指向的那个变量的值（也就是变量 x 的值）。

3. 给定声明：

```
int * x, p = & x, q;
```

请解释该声明合法或者不合法的原因。

4.4 指向函数的指针

按照 C 语言的规定，一元&运算符的操作数必须是左值或者函数指示符。如果是一个左值，则一元&运算符的结果是指向变量的指针；如果是一个函数指示符，则一元&运算符的结果是指向函数的指针。

然而，函数类型是多种多样的，返回类型和参数类型不同的函数是不同的函数类型。相应地，如果两个指针所指向的函数类型不同，则它们是不同的指针类型。下面的程序演示了如何在程序中使用指向函数的指针。

```
/*****************c0403.c***************/
void swap_ab (int * a, int * b)
{
    int temp = * b;

    * b = * a;
    * a = temp;
}

int main (void)
{
    int m = 10086, n = 10010;

    swap_ab (& m, & n);

    void (* pf) (int *, int *) = swap_ab;
    pf (& m, & n);
}
```

这个程序的意图是用函数 swap_ab 交换两个变量的值，但是用了两种不同的方法。在 main 函数里，我们首先声明了变量 m 和 n 并分别初始化为 10086 和 10010，这没有什么

好说的。接下来，我们调用函数 swap_ab 来交换这两个变量的存储值。

现在我们来看函数 swap_ab 的声明，该函数具有两个参数 a、b 且它们的类型都是指向 int 的指针，函数的功能是交换这两个指针所指向的变量的值。由于功能简单，该函数没有什么可返回的，所以不返回任何值。在第一章里我们讲过，如果一个函数不返回值，则它的返回类型必须声明为 void。

4.4.1 函数指示符－指针转换

再回头来看 main 函数，通过语句

```
swap_ab (& m, & n);
```

可以看出，要交换值的变量是 m 和 n。尽管我们已经非常熟悉函数调用，但实际上你可能并不是真的懂它，因为函数调用运算符 () 的左操作数实际上必须是一个指针，而且是指向函数的指针。

问题是，在上述语句中，swap_ab 是一个函数指示符。不过没关系，C 语言规定，除非作为一元&运算符的操作数，否则，函数指示符将自动转换为指向函数的指针，这称为函数指示符－指针转换。

调用函数 swap_ab 时，传递的实际参数是表达式& m 和& n 的值。这是两个指针，分别指向变量 m 和变量 n。

再回到函数 swap_ab，参数 a、b 在该函数开始执行时被创建为两个变量以接受传递给它们的指针。如图 4-5 所示，一旦实际参数被传递给变量 a 和 b，则它们的值现在各自指向 main 函数内的变量 m 和 n。

要交换两个变量的值，需要使用第三个变量，这是很容易理解的。为此，我们在函数 swap_ab 里声明了一个变量 temp，并初始化为表达式* b 的值：

```
int temp = * b;
```

在这里，左值 b 要先执行左值转换，转换为变量 b 的存储值，这是一个指针。然后，一元*运算符作用于这个指针，得到一个（代表变量 n 的）左值。该左值继续执行左值转换，得到变量 n 的存储值，然后用这个值初始化变量 temp。

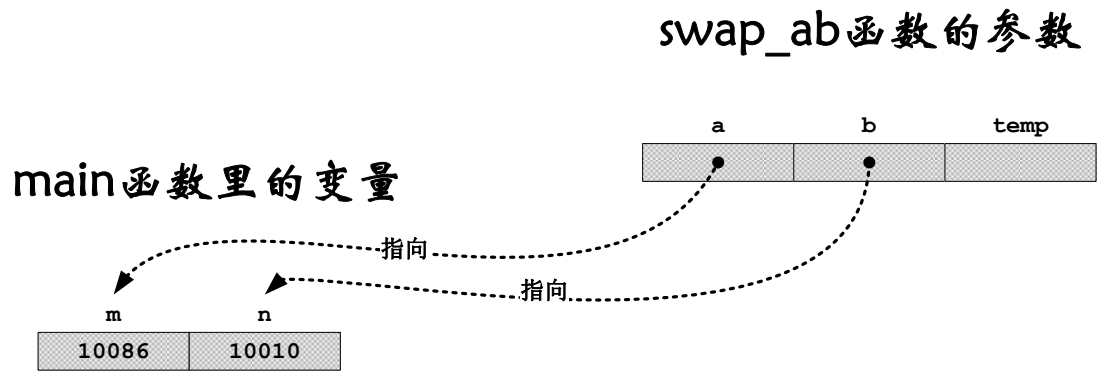


Fig.4-5 函数参数的值指向 main 函数内的变量

你可能觉得我很啰唆，因为一眼就能看出表达式* b 的结果就是变量 n 的存储值。但我这样做的目的是让你学会并习惯如何解析涉及指针的表达式。很多人在分析简单表达式的时候觉得自己很明白，但表达式一复杂就晕头转向、一筹莫展，就是因为只凭感觉而缺乏正确的、科学的分析方法。同样的道理，在第二条语句

```
* b = * a;
```

中，表达式* b = * a 的两个子表达式* a 和* b 的结果都是左值，实际上分别代表

变量 m 和 n。但是，因为左值* b 位于赋值运算符的左侧，不执行左值转换，左值* a 位于赋值运算符的右侧，执行左值转换，然后赋给左值* b。即，读取变量 m 的值并把它保存到变量 n。

在函数 swap_ab 的最后一条语句

```
* a = temp;
```

里，表达式* a = temp 用于将变量 temp 的值保存到左值* a 所代表的变量（实际上是变量 m）。至此，我们完成了两个变量的值的互换。

函数 swap_ab 内没有 return 语句，但这不影响它的返回。对于没有返回值（返回类型为 void）的函数来说，不通过 return 语句返回没有任何问题，当函数的执行到达组成函数体的右花括号“}”时，相当于执行了一条不带表达式的 return 语句。

但是，如果函数的返回类型不是 void，而且函数的返回是因为执行到组成函数体的右花括号“}”，则调用者不得使用函数的返回值，否则程序的行为是未定义的。唯一的例外是从 C99 开始，宿主式环境下的 main 函数通过右花括号“}”返回时，则默认返回 0。不过，要是 main 函数的返回类型不是或者不和 int 等价，则返回值不确定。

再回到 main 函数，调用函数 swap_ab 之后，我们又声明了一个变量 pf，其类型为指向函数的指针：

```
void (* pf) (int *, int *) = swap_ab;
```

如图 4-6 所示，要解读这个声明，依然是从标识符开始。我们说过，C 语言的一个特点是变量和函数的声明与它们在程序中的使用在形式上一致。因此，声明中的非字母符号虽然不是运算符，但却继承了它们的优先级规则。如果不是用圆括号将“* pf”括起来，那么，标识符 pf 将优先与它右边的(int *, int *)进行语法关联。

但是，因为圆括号的存在，标识符 pf 被认为是与它左边的星号“*”进行关联，因此，是需要先向左读，即，“pf 的类型是指针（*）”或者“pf 是一个指针”。

既然是一个指针，那么它必须指向另一个类型。到底指向谁呢？如果(* pf)的右边没有东西，则它可以继续往左读，但是它右边是(int *, int *)，那就意味着该指针指向一个函数。因此，我们进一步往右读做“指向一个函数”。

对函数来说，重要的是它的参数类型和返回类型。因此，必须在声明里提供参数类型和返回类型。于是，我们可以继续读做“第 1 个参数的类型是 int *，第 2 个参数的类型是 int *”。和不带函数体的声明一样，在这里，形参的名字不是必须的。

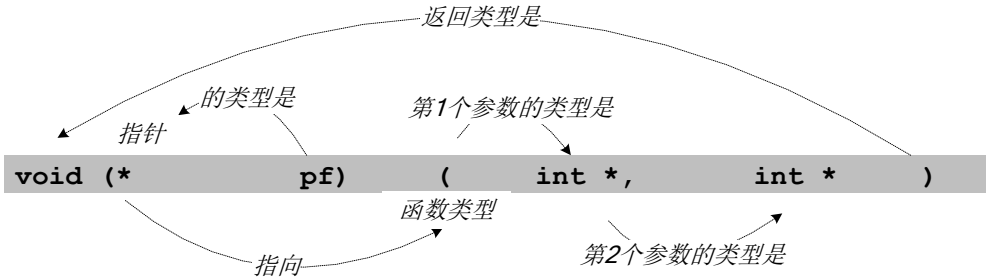


Fig.4-6 声明一个变量，其类型为指向函数的指针

函数的返回类型一定是在左边，于是我们回过头来往左看，那里是一个关键字“void”，于是我们读作“返回类型为 void”。如果一个函数的返回类型是 void，则意味着它不返回任何值，或者说它返回空值。

到此，整个声明的左边和右边再没有其他东西，不再继续往下读，标识符 pf 的类型已经完全确定。笼统地说，pf 是一个变量，其类型为指向函数的指针；再具体一点，pf 是一个变量，其类型为指向 void (int *, int *)类型的指针；如果要用类型名来描述的话

则 pf 是一个变量，其类型是 void (*) (int *, int *)；如果还要更具体的话，就是“pf 是一个变量，其类型为指向函数的指针，被指向的函数有两个参数，其类型都是指向 int 的指针，函数的返回类型为 void”。

在变量 pf 的声明里带有一个初始化器 swap_ab，在这里它是一个函数指示符，必须执行函数指示符—指针转换。因为 swap_ab 的类型是 void (int *, int *)，自动转换为指向这种函数类型的指针，即 void (*) (int *, int *)，和变量 pf 的类型一致，可用于初始化操作。

接下来，语句

```
pf (& m, & n);
```

又一次发起函数调用，不过这一次属于本色调用，因为函数调用运算符的左操作数本来就是指针。表达式 pf 是一个指针类型的左值，故先进行左值转换，转换为该变量的存储值，这是一个指向函数的指针，实际上指向函数 swap_ab。因为函数调用需要一个指向函数的指针，相比之下，这是 C 语言比较喜欢的写法，毕竟不需要做函数指示符—指针转换。当然，如果你非要这么写也是可以的：

```
(* pf) (& m, & n);
```

函数调用运算符 () 的优先级比一元*运算符高，故这里必须用括号来形成一个基本表达式以阻止不恰当的结合。因为 pf 是一个指针类型的左值，左值转换后得到一个指针，而一元*运算符作用于这个指针，得到一个函数指示符。然后，函数指示符又反过来继续转换为一个指针函数的指针。显然，这是在转圈圈，既然是这样，下面的写法也没问题：

```
(& * pf)  (& a, & b);
(* & * pf)  (& a, & b);
(& * & * pf)  (& a, & b);
```

练习 4.3：

1. 为什么上面三种函数调用的写法都没问题？请分析它们的工作原理。
2. 若变量 pf 的类型是指向函数的指针，被指向的函数有两个 char 类型的参数，且返回类型是 int，请写出 pf 的声明，以及它的类型名。

4.5 返回指针的函数

函数不但可以具有指针类型的参数，它的返回类型也可以是指针。在下面的程序中，函数 swaprp 交换两个参数所指向的变量的值，并返回一个指针，该指针指向值较大的那个变量。

```
/***********c0404.c***********/
char * swaprp (char * a, char * b)
{
    char temp = * a;
    * a = * b;
    * b = temp;

    return * a > * b ? a : b;
}

int main (void)
{
    char m = 102, n = 103, * pc;
```

```
    pc = swaprp (& m, & n);
}
```

如图 4-7 所示，因为标识符 swaprp 看上去既可以与星号“*”进行语法关联，也可以与括号“(”进行语法关联，但括号的优先级比星号高，所以，对函数 swaprp 的声明应当这样解读：它的类型是函数，第一个参数是 char *类型的变量 a，第二个参数是 char *类型的变量 b，该函数返回一个指针，指向 char。当然，也可以直接说是返回一个指向 char 的指针。

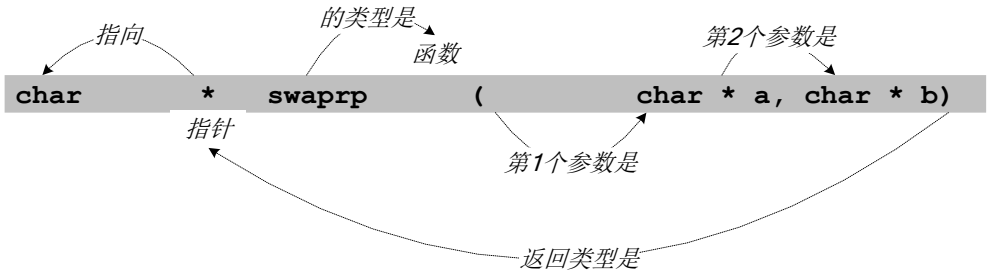


Fig.4-7 声明一个函数，该函数的返回类型是指针

在函数 swaprp 里，变量值的交换和前面相比没有什么不同，唯一的变化是多了一个奇怪的 return 语句：

```
return * a > * b ? a : b;
```

因为函数 swaprp 的返回类型不是 void，所以这里是返回表达式* a > * b ? a : b 的值。这是一个我们没见过的表达式，称为条件表达式。条件表达式由三个表达式 E1、E2、E3，以及条件运算符?和:按下述方式组合而成：

```
E1 ? E2 : E3
```

再来看表达式* a > * b ? a : b，这里涉及三个运算符，其优先级从高到低依次为一元*运算符、关系运算符>和条件运算符?:，所以这是一个条件表达式，等价于((* a) > (* b)) ? a : b。

条件表达式的求值过程是这样的：先求值 E1，如果 E1 的值不为 0，则求值 E2，且整个条件表达式的值来自 E2；如果 E1 的值为 0，则求值 E3，且整个条件表达式的值来自 E3。例如，条件表达式 5 ? 6 : 7 的值是 6，而 0 ? 8 : 9 的值是 9。

这样一来，return 语句的功能就很清楚了：表达式* a 和* b 都是左值，先进行左值转换，关系运算符对左值转换后的值进行比较，若结果为 1，则整个条件表达式的值是左值 a 经左值转换后得到的指针；若结果为 0，则整个条件表达式的值是左值 b 经左值转换后得到的指针。最后，return 语句返回条件表达式的值（指针）给它的调用者。取决于比较的结果，返回的指针指向 main 函数内的变量，要么是变量 m，要么是变量 n。

再来看 main 函数，我们声明了一个指针类型的变量 pc，并用它来接受函数 swaprp 的返回值。因为 pc 是指向 char 的指针，而函数 swaprp 的返回值也是指向 char 的指针，类型一致，可以赋值。

练习 4.4：

1. 在程序中添加一个 char 类型的变量 c，并用函数 swaprp 的返回值所指向的变量的值初始化它。

2. 以下，函数 cusum 用于计算从 1 加到 N 的和，N 是非负整数。参数 sum 用于接受一个指针，累加的结果保存在该指针所指向的变量里；参数 r 也用于接受一个指针，该指针所指向的变量里保存了所要累加的最大值。如果这个值是 0 或者大于 1000000000，则

函数返回 0，否则返回 1。现在请按上述要求将函数体补充完整，并上机验证程序的编写是否正确。

```
_Bool cusum (unsigned long long int * sum, unsigned long long int
* r)
{

}
```

3. 以下类型中，属于变量类型的是_____；属于函数类型的是_____。

(a) char (b) main 函数 (c) _Bool (d) signed int (e) 指针

4. 若* E 是合法的表达式，则该表达式的结果不可能是_____。

(a)值 (b)左值 (c)代表一个变量 (d)代表 E（的值）所指向的那个变量

5.在表达式& E 中,E 必须是_____,如果 E 的类型是 int,则&E 的结果类型是_____。

(a)值 (b)左值或者函数指示符 (c)指向 int 的指针 (d)int

4.6 掌握 C 语言需要建立类型的观念

在初学指针的时候，很多人会有这样的想法：我认为 2000 是一个地址，所以我可以将它赋给一个指针类型的变量，就像这样：

```
int * p;
p = 2000;
```

或者，我认为 2002 是一个变量的地址，一元*运算符用来间接访问那个变量，所以我可以这样写：

```
* 2002 = 10086;
```

初学者有这样的认识应该是正常的,但这样做并不合法。首先,我们在前面讲得很清楚，地址和指针不是一回事；其次，一元*运算符需要一个指针操作数。C 语言有自己的类型系统，指针类型和整数类型是被区别对待的，你认为 2002 是一个地址，但 C 语言只知道它是一个整数——事实上，2002 是一个整型常量。

再比如说，加性运算符+的两个操作数不能是指针类型。你想想看，将两个指针相加是什么意思呢？没有任何实际意义。水稻不能长在石头上，不同的运算符需要不同类型的操作数。在本章的剩余部分里，我们介绍整数类型的转换规则；在本书后面的章节里也将介绍类型和类型转换的知识。在本书的第十二章里，我们将具体介绍 C 语言里的所有运算符和表达式，届时将分类详细介绍每种运算符的操作数类型及转换方法。

练习 4.5：

下面的程序片段合法吗？将它补充为一个完整的程序，然后上机试一试，看翻译器怎么说。

```
int m = 0, * p = & m, * q = & m;
p += q;
```

4.6.1 整型常量

在 C 语言里，变量有类型，值也有类型，给一个变量赋值，或者初始化它，值的类型必须和变量的类型相符。来看一个例子：

```
/****c0405.c****/
```

```
int main (void)
{
    int m = 3700;

    return 0;
}
```

在这个程序中，变量 m 的类型是 int，需要初始化或者保存一个同类型的数值。那么，3700 的类型也是 int 吗？还有 return 语句，它返回 0 值，这个 0 的类型和 main 函数的返回类型 int 一致吗？

在本书第一章里我们已经提到了常量和常量表达式，所以大家知道这里的 3700 和 0 就是常量。本质上，3700 和 0 都是在程序编写时由程序员敲入的文字符号，与一般的文字符号相比，它们的不同之处是：在程序翻译期间，C 实现会将它们识别并转换为整数，且不再可能发生改变，所以这些代表整数的符号称为整型常量。

如表 4-1 所示，整型常量有三种形式，包括十进制常量、八进制常量或者十六进制常量，区分它们的方法也很简单——观察其前缀。

十进制形式的整型常量以非 0 数字字符开头，后面可以是从 0 到 9 的任意数字字符的组合；八进制形式的整型常量以数字 0 开头，后面可以是从 0 到 7 的任意数字字符的组合；十六进制形式的整型常量以“0x”或者“0X”开头，后面可以是以下字符的任意组合：从 0 到 9 的数字字符、从 a 到 f 的字母，以及从 A 到 Z 的字母。

C 语言里的数字（数值）都是有类型的，如果说整型常量的前缀决定了它的基数（采用的数制），那么后缀则指定了它的类型。后缀 u 或者 U 是“unsigned”的意思；后缀 l 或者 L 是“long”的意思；后缀 ll 或者 LL 是“long long”的意思。这几种后缀可以单独使用，也可以组合使用，但只有以下组合是允许的、有意义的：ul、uL、ull、uLL、Ul、UL、Ull、ULL、lu、lU、Lu、LU、llu、llU、LLu、LLU。

因此，2u、3U、0x3fu、0x6dLLu、073ull 也都是整型常量，其类型分别为 unsigned int、unsigned int、unsigned int、unsigned long long int 和 unsigned long long int。其他一些整型常量的例子见表 4-1。

表 4-1 不同形式的整型常量

常量的形式	不带后缀的例子	带后缀的例子
十进制常量	1、2、56、89、5050	3U、7LL、257ULL
八进制常量	0、025、079、0980	02L、037LL、02065ULL
十六进制常量	0x3、0x59、0X87、0XFA	0x7AL、0X556U、0xFACE

十进制常量不会以“0”打头，而八进制常量必须以“0”打头。经验不足的人可能以为数字前面的 0 不会改变数字的大小，一不留神将十进制常量写成八进制常量。

确定整型常量的类型实际上并没有那么简单，这要取决于值的大小、使用的基数、后缀和各种整数类型所能表示的值的范围（这取决于 C 实现）。从传统的 K&R C 开始，到 C89，再到 C99 和 C11，所使用的规则多少有些差别。表 4-2 给出了最新的类型确定规则。

根据整型常量所采用的后缀和基数，结合具体的 C 实现，从所对应的方框内挑选出第一个能够表示常量值的类型，可作为整型常量的类型。

例如，如果 C 实现将 int 和 long int 类型的最大值定为 2147483647，而将 long long int 类型的最大值定为 9223372036854775807，那么整型常量 5000000000 的类型是什么呢？因为它没有后缀，采用的基数是 10（十进制），所以定位到对应的表格，第一个备选类型是 int，但它表示不了这个数；第二个备选类型 long int 也表示不了这个数，只有第三个备选类型 long long int 可以表示，所以整型常量 5000000000 的类型

是 long long int。

表 4-2　整型常量的类型对照表

后缀	常量采用的数制	
	十进制	八进制或十六进制
无后缀	int long int long long int	int unsigned int long int unsigned long int long long int unsigned long long int
u 或 U	unsigned int unsigned long int unsigned long long int	unsigned int unsigned long int unsigned long long int
l 或 L	long int long long int	long int unsigned long int long long int unsigned long long int
ll 或 LL	long long int	long long int unsigned long long int
ul、uL、Ul、UL、lu、lU、Lu 或者 LU	unsigned long int unsigned long long int	unsigned long int unsigned long long int
ull、uLL、Ull、ULL、llu、llU、LLu 或者 LLU	unsigned long long int	unsigned long long int

回到本节开头的程序，现在我们知道，3700 的 0 是 int 类型的整型常量，可以用来初始化 int 类型的变量或者给 int 类型的变量赋值。

顺便说一下，调试器 GDB 提供了一个 ptype 命令，可以返回表达式的类型信息，下面的调试过程演示了它的使用方法。

```
D:\>gdb -silent
(gdb) ptype 0
type = int
(gdb) ptype 0L
type = long
(gdb) ptype 5000000000
type = long long
(gdb)
```

练习 4.6:

整型常量 37、0x98L 和 056ULL 是什么类型？请在 GDB 中验证你的结论。

4.6.2 整数—整数转换

在 C 语言里，不同的类型决定了可以表示的数值范围，以及数值在存储器中如何表示，以什么样的位模式存在。变量是有类型的，从变量中读出的值也是有类型的，这个值的类型和变量的类型一致。

相应地，往变量写入一个值，值的类型应该和变量的类型一致。如果值的类型和变量的

类型不一致，则必须先转换为变量的类型[3]，请看下面的例子。

```
/******c0406.c******/
int main (void)
{
    int m = 3700U;
    unsigned int u = m;
    signed char sc;
    unsigned char uc;
    uc = u;
    sc = u;
    sc = 107LL
}
```

在这里，整型常量 3700U 的确切类型是 unsigned int（对照表 4-2），但变量 m 的类型是 int，这就要将 3700U 从 unsigned int 转换为 int 之后才能初始化。

在所有整数类型中，_Bool 类型最特殊，先来看它。C 语言规定，将任何整数类型的值转换为_Bool 类型时，零值转换为 0，非零值转换为 1；将一种整数类型的值转换为_Bool 之外的整数类型时，如果这个值可以用新类型表示，则转换后的值同原值相比不变。

在这里，3700U 的类型是 unsigned int，但是 int 类型的取值范围里也有 3700 这个数，所以 3700U 是可以用 int 类型来表示的，于是它自动转换为 int 类型的 3700 并用于初始化变量 m。

接下来，变量 u 的类型是 unsigned int，但其初始化器为左值 m，经左值转换后得到一个 int 类型的值 3700。这个 3700 可以用 unsigned int 类型来表示，故又从 int 类型转换为 unsigned int 类型并用于初始化变量 u。

再往下，我们声明了变量 uc 和 sc，并将变量 u 的值赋给它们。凭直觉，这个赋值是有问题的，因为变量 u 的值是 unsigned int 类型的 3700，而变量 uc 和 sc 的类型分别是 unsigned char 和 signed char，应该是无法容纳的。

C 语言规定，将任何一种整数类型的值转换为非_Bool 的另一种**无符号整数类型**时，如果无法用新类型来表示，则转换的方法是将这个值重复地加上或者减去比新类型所能表示的最大值大 1 的数，直到结果可以用新类型来表示；将任何一种整数类型的值转换为另一种**有符号整数类型**时，如果这个值无法用新类型来表示，则转换的结果取决于 C 实现。

在我的机器上，unsigned char 类型可以表示的最大值是 255，所以，变量 uc 不能表示变量 u 的值 3700。在这种情况下，我们不停地用变量 u 的值减去 256，直到结果小于或者等于 255。在做 14 次相减后，结果是 3700-14×256=116。所以，变量 uc 在赋值之后的结果是 116。

但是，在将变量 u 的值赋给变量 sc 后，变量 sc 的结果不确定，这要取决于 C 实现如何将 unsigned int 类型的 3700 转换为 signed char 类型。

最后，107LL 的类型是 signed long long int，变量 sc 的类型是 signed char，但 signed char 类型的取值范围里也有 107，也就是可以表示，故，将 107LL 转换为 signed char 类型并赋给变量 sc 后，变量 sc 的值是 107。

练习 4.7：

若某无符号整数类型可表示的最大值为 M，那么，把-1 赋给这种类型的变量后，该变量的值是什么？

[3] 但有些类型之间是不允许转换的。

4.6.3 表达式的类型

在 C 语言里，表达式可以计算出一个值，例如表达式 5 + 3 的值是 8。值是有类型的，所以表达式也有类型，表达式的类型与其值的类型一致。

不同的运算符需要不同类型的操作数，有些表达式具有固定的类型，而有的表达式的类型则需要根据运算符和操作数的类型来共同确定，不一而足。对运算符、表达式及其类型的完整介绍位于后面的章节，现在，我们通过一个小小的示例程序来大体了解一下。

```
/********c0407.c********/
int max (int a, int b)
{
    if (a >= b) return a;
    else return b;
}

int main (void)
{
    int x = 1, y;
    unsigned char c;

    x += c = 56;
    c += max (x, y);
    y = ++ x;
}
```

我们已经讲过整型常量，这是一个语法上的概念，指的是一种表示整数的语法成分。当这种语法成分实际出现在程序中时，它就摇身一变成了常量表达式。

在程序中，变量 x 的初始化器是常量表达式 1，其类型为 int，所以也可以说它是 int 类型的常量表达式。这个常量表达式在程序翻译期间由代表数字的符号转换为数字 1 并用于初始化同类型的变量 x。

再来看表达式 x += c = 56，它等价于 x += (c = 56)，所以我们先分析它的子表达式 c = 56。左值 c 的类型是其所代表的变量 c 在声明时指定的类型，你可能觉得这样说有些啰唆，因为左值 c 和变量 c 是同一种东西。但是，它们的身份不同，在声明里，c 是变量，在表达式里，c 是左值。

左值 c 的类型是 unsigned char，但是常量表达式 56 的类型是 int，必须要执行整数类型—整数类型转换，将 56 从 int 类型转换为 unsigned char 类型后再赋给左值 c 所代表的变量。

在 C 语言里，赋值表达式的类型是赋值运算符左操作数的类型。因此，表达式 c = 56 的类型是 unsigned char；这个子表达式要计算一个值，这个值是赋值运算符的左操作数被赋值之后的存储值 56，值的类型是赋值表达式的类型，即 unsigned char。

接着来看，子表达式 c = 56 的值被加到左值 x。在运算符+=的右侧，操作数的类型是 unsigned char；在左侧，操作数 x 的类型是 int，这必然要将右侧表达式的结果 56 从它原先的 unsigned char 转换为 int，然后再进行赋值操作。最后，还是那句话，赋值表达式的类型是赋值运算符左操作数的类型，故表达式 x += c = 56 的类型是 int。

继续来看表达式 c += max (x, y)，函数调用表达式的类型是被调用函数的返回类型，它的值是函数的返回值。所以，子表达式 max (x, y) 的类型是 int，这也是该表达式的值的类型，这个值来自函数 max 的返回值。

函数调用的返回值被加到左值 c，左值 c 的类型是 unsigned char 而函数调用表达式的类型是 int，必须将后者从 int 类型转换为 unsigned char 之后才能赋值。由于赋值运算符的左操作数是 unsigned char 类型，故表达式 c += max (x, y) 的类型也是 unsigned char。

如果函数的返回类型是 void，则函数调用表达式的类型也是 void，这样的函数将返回空值，或者说返回不存在的值。这样的表达式可以添加一个分号“;”使其成为表达式语句，但它不能作为所有运算符的操作数。

现在转到 max 函数内部，if 语句的控制表达式是关系表达式 a >= b。在 C 语言里，不管运算符>、>=、<和<=的操作数是什么类型，关系表达式的类型始终为 int。关系成立，关系表达式的结果是 int 类型的值 1；否则，关系表达式的结果是 int 类型的值 0。

最后再回到 main 函数，来看表达式 y = ++ x。子表达式++ x 是前缀递增表达式，它的类型和前缀递增运算符++的操作数相同。因为左值 x 的类型是 int，故表达式++ x 的类型也是 int，与另一个左值 y 的类型一致，可以赋值。

为了更好地理解“表达式的类型”这一主题，我们将上述程序保存为源文件 c0407.c 并翻译为可执行文件，然后在调试器里用 ptype 命令逐一验证如下。

```
(gdb) l
1       int max (int a, int b)
2       {
3           if (a >= b) return a;
4           else return b;
5       }
6
7       int main (void)
8       {
9           int x = 1, y;
10          unsigned char c;
(gdb) l
11          x += c = 56;
12          c += max (x, y);
13          y = ++ x;
14      }
(gdb) b 3
Breakpoint 1 at 0x40155a: file c0407.c, line 3.
(gdb) r
Starting program: D:\examples\a.exe
[New Thread 10100.0x2b00]
[New Thread 10100.0x294]

Thread 1 hit Breakpoint 1, max (a=57, b=0) at c0407.c:3
3           if (a >= b) return a;
(gdb) ptype a >= b
type = int
(gdb) n
5       }
```

```
(gdb) n
main () at c0407.c:12
12          c += max (x, y);
(gdb) ptype c = 56
type = unsigned char
(gdb) ptype x += c = 56
type = int
(gdb) ptype max (x, y)
type = int
(gdb) ptype c += max (x, y)
type = unsigned char
(gdb) ptype ++ x
type = int
(gdb) ptype y = ++ x
type = int
(gdb) c
Continuing.
[Thread 10100.0x294 exited with code 0]
[Inferior 1 (process 10100) exited normally]
(gdb) q
```

因为函数 max 的参数只在函数内有效，所以我们先将断点设置在该函数内。当程序的执行到达函数内的断点时，再用 ptype 命令打印表达式 a >= b 的类型。

然后，连续用 n 命令使程序的执行回到 main 函数，再用 ptype 命令打印各个表达式的类型。

4.6.4 认识整型转换阶和整型提升

有些运算符只在操作数原有的类型上操作，例如前缀递增运算符和后缀递增运算符，它们不要求改变操作数的类型。再比如赋值运算符，它不要求改变左操作数的类型，而是要求右操作数必须转换为左操作数的类型。

相比之下，有些运算符的操作数并不是在它原来的类型上操作，尤其是那些需要两个操作数的运算符。不过这也可以理解，如果操作数的类型不同，值的表示方法也不同，这势必要求它们先转换为一致的类型才能运算。下面以一个程序为例来说明这种转换如何进行。

```
/**********c0408.c*********/
int main (void)
{
    signed char cx = 1, cy = 2;
    signed long int sl = 0;
    unsigned long int ul = 0;

    sl += cx + cy;
    sl += cx * 3L;
    ul += sl <= ul;

    unsigned char uc = -1;
```

```
    cy = - uc ++;
}
```

在这里，表达式 sl += cx + cy 用于将子表达式 cx + cy 的值加到左值 sl。在子表达式中，左值 cx 和 cy 的类型都是 signed char，是不是意味着子表达式 cx + cy 的类型也是 signed char？

非常遗憾的是，非也。我们知道，int 和 unsigned int 类型的长度并不是固定的，随不同的计算机系统而异。在 C 语言的设计者看来，int 和 unsigned int 类型通常应该等于你所用的计算机的自然字长——比如，在 16 位处理器的计算机上，int 类型的长度通常是 16 个比特，在 32 位处理器的计算机上，int 类型的长度通常是 32 个比特。

计算机的字长通常等于处理器内部的寄存器宽度，这样就很清楚了：以自然字长来加工和操作数据效率最高。C 是追求效率的计算机编程语言，它希望包括二元+在内的很多运算符能够以计算机的自然字长来操作。所以它会想：先看一下这两个操作数的类型，看看有没有比 int 或者 unsigned int 短的。如果有，那就先把它加长到 int 或者 unsigned int。加长之后，如果这两个操作数的类型一致，那太好了；如果不一致，再继续以那个较长的为标准来转换那个较短的，使它们最终一致。无论如何，第一步，也是最保守的做法是先将短的类型加宽为 int 或者 unsigned int。

但是，整数类型那么多，到底谁是长的，谁是短的？为此，C 语言引入了整型转换阶的概念。整型转换阶用于确定加宽的方向，即，加宽为何种类型。每种整数类型都有自己的整型转换阶，阶的大小主要取决于类型的宽度，图 4-8 给出了所有标准整数类型的整型转换阶。

由图中可知，每个有符号整数类型的阶等于与其相对应的无符号整数类型；_Bool 类型的阶最低；long long int 和 unsigned long long int 类型的阶最高。

对于包括二元+在内的很多运算符来说，C 语言规定，如果一个操作数相对于 int 类型来说较窄，但它的值能用 int 类型来表示，则将其转换为 int 类型；如果无法表示，则转换为 unsigned int 类型，这个过程叫作整型提升。

那么，怎样才算是比 int 类型窄呢？标准之一就是它的整型转换阶小于 int 和 unsigned int 类型。显然，_Bool、char、signed char、unsigned char、short int 和 unsigned short int 类型的操作数都必须先做整型提升。

这里有两点需要说明，第一，整型提升是一种特殊的整数类型转换，特指从阶较低的整数类型转换（提升）为 int 或者 unsigned int 类型，从 int 类型转换到 long int 类型并不是整型提升；第二，并不是所有运算符的操作数都需要做整型提升，例如递增和递减运算符的操作数就不需要，即使它们是整数类型。

让我们继续来看表达式 cx + cy，左值 cx 和 cy 的类型都是 signed char，所以它们都必须进行整型提升，提升为 int 类型，故表达式 cx + cy 的类型是 int。

原则上，运算符的操作数在整型提升后应具有一致的类型，如果不一致的，还必须做进一步的转换。总的原则是，阶较低的整数类型转换为阶较高的整数类型。

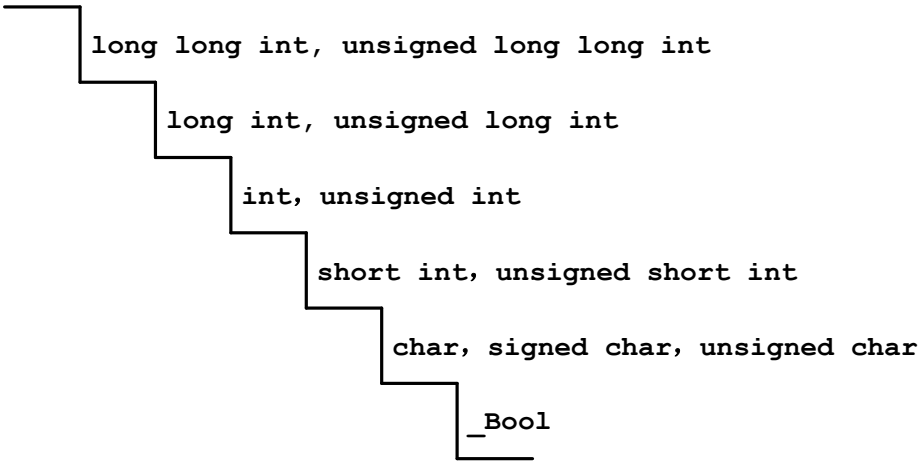


Fig.4-8 标准整数类型的转换阶

来看表达式 sl += cx * 3L 的子表达式 cx * 3L，常量表达式 3L 的类型是 signed long int，而左值 cx 的类型是 signed char。首先将 cx 的值从 signed char 类型提升为 int 类型。提升后类型仍不一致，故必须将 cx 的值再次从 int 类型转换为 signed long int 类型。换句话说，子表达式 cx * 3L 的类型是 signed long int。

再来看表达式 ul += sl <= ul 的子表达式 sl <= ul，左值 sl 的类型是 signed long int，而左值 ul 的类型是 unsigned long int，这两种整数类型的阶都高于 int 和 unsigned int，所以这里不存在整型提升，但它们仍不是同一种类型。如果提升之后两个操作数的类型不同，一个是有符号整数类型，另一个是无符号整数类型，且无符号整数类型的阶高于或者等于那个有符号整数类型，则将有符号整数类型的操作数转换为那个无符号整数类型。因此，必须将 sl 的值从 signed long int 转换为 unsigned long int 类型。

然而，表达式 sl <= ul 的类型会是 unsigned long int 吗？不会的，我们说过，关系表达式的类型始终为 int。这里的奥妙在于，操作数 sl 和 ul 的值统一在 unsigned long int 层面上进行比较操作，然后根据比较的结果生成 int 类型的 0 或者 1。

练习 4.8:

在上面的程序中，表达式 sl += cx * 3L 和 ul += sl <= ul 的类型各是什么，为什么？

4.6.4.1 负号运算符

细心的同学可能已经发现了，整型常量里没有负数。我们在编程时不可避免地会用到负数，例如-67，但是在 C 语言里，67 是整型常量表达式，前面的负号是运算符，称为负号运算符。换句话说，在 C 语言里，负数是通过“运算”得到的，它将一个整型常量表达式的值转换为一个负的内部表示。

负号运算符需要一个右操作数，如果这个操作数是整数类型，必须先整型提升，负号运算符的结果是提升后的负值，结果的类型是提升后的类型；如果操作数不是整数类型，则负号运算符的结果是其操作数的负值，结果的类型与其操作数的类型相同。

在上面的程序里，末尾部分有一个声明：

```
unsigned char uc = -1;
```

在这里，负号运算符的操作数 1 是整型常量表达式，按要求必须先做整型提升，但其类型已经是 int，名义上要做但是没做。最终，表达式-1 的类型也是 int。

表达式-1 的值用于初始化变量 uc，变量 uc 的类型是 unsigned char，这就要把表达式-1 的值从原先的 int 类型转换为 unsigned char 类型。在我的机器上，signed char 类型的最大值是 255，所以转换的方法是 256 + (-1) = 255（参见前面的整数—整数转换）。这就是说，将表达式-1 的值赋给 unsigned char 类型的变量，将使该变量的值为 unsigned char 类型所能表示的最大值。

推而广之，将表达式-1 的值转换为任何一种无符号整数类型，其结果是得到这种无符号整数类型的最大值。

最后来看表达式 cy = - uc ++，在这里，后缀递增运算符的优先级最高，负号运算符的优先级次之，赋值运算符的优先级最低，故它等价于 cy = - (uc ++)。递增运算符不改变其操作数的类型，而且，递增表达式的类型也是其操作数的类型。因为左值 uc 的类型是 unsigned char，故表达式 uc ++的类型也是 unsigned char。

作为负号运算符的操作数，表达式 uc ++的值要从原先的 unsigned char 类型提升为 int 类型，而且这也是表达式- uc ++的类型。提升的过程也是类型转换的过程，由于 unsigned char 类型的值总能用 int 类型来表示，故转换（提升）后的值不变。

因为赋值运算符左操作数 cy 的类型是 signed char，所以表达式- uc ++的值还要从 int 类型转换为 signed char 类型。

我们已经讲过整数—整数转换，如果表达式- uc ++的值能够被 signed char 类型表示，则转换后的值不变；如果不能，因目标类型是有符号整数类型，故转换后的结果不能确定。在我的机器上，转换前，表达式- uc ++的值是 255，但不能被 signed char 类型表示，故转换后的值无法预知。

不同的运算符需要不同的操作数，需要做不同的转换。每种运算符需要什么类型的操作数，如果操作数的类型不同该如何转换，都将在本书的后面逐一介绍。

练习 4.9：

1. 如何知道 unsigned char、unsigned int、unsigned long int 和 unsigned long long int 类型所能表示的最大值（在你的计算机上）？编写一个程序，然后在 gdb 中观察一下到底是多少。
2. 表达式-3u 的结果是多少？结果的类型是什么？

4.6.4.2 转型运算符

一般来说，整数之间的转换是自动进行的。当然，如果你不嫌麻烦的话，也可以手工进行转换，手工转换的方法是使用转型表达式。转型表达式由转型运算符组成，其形式为

(类型名) 表达式

在 C 语言里，有三种运算符非常相似，都使用了一对圆括号，它们是函数调用运算符、转型运算符和基本表达式。不过，它们之间的区别也相当明显：函数调用运算符的操作数在左边和圆括号内；转型运算符的操作数在右边；基本表达式的操作数在圆括号内。

转型表达式的作用是将“表达式”的值从它原先的类型转换为“类型名”指定的那种类型。举个例子来说，在以下声明中，是将整型常量 0x33 从它原先的 int 类型转换为 long long int 类型后，再用于初始化变量 ll：

```
long long ll = (long long) 0x33;
```

注意，被转型的表达式里可能含有别的运算符，如果它们的优先级低于转型运算符，则被转型的表达式应当用圆括号括起来以形成基本表达式，例如：

```
signed char cx, cy;
cx = (signed char) 0x33;
cy = (signed char) (cx + 0x30);
```

以上，转型运算符的优先级高于赋值运算符，表达式 cx = (signed char) 0x33 是将整型常量 0x33 从它原来的 int 类型转换为 signed char 类型，然后赋给左值 cx。

在表达式 cy = (signed char) (cx + 0x30)里，是将表达式 cx 的值从它原先的 signed char 类型提升为 int 类型，再与 int 类型的 0x30 相加，得到一个 int 类型的结果。这个结果再转型为 signed char 类型，赋给变量 cy。

转型运算符的优先级高于加性运算符，所以表达式 cx + 0x30 必须用圆括号括住。如果没有这个圆括号，意思就完全不同了：

```
cy = (signed char) cx + 0x30;
```

这是将表达式 cx 的值转换为 signed char 类型后，再与 int 类型的 0x30 相加。顺便说一句，在相加之前，表达式(signed char) cx 的值还要作整型提升。

4.6.5 指针—整数转换

再回到指针的话题。很多初学者喜欢把指针变量的值看成整数，但这有点像把面粉和馒头相提并论，毕竟它们至少在类型上并不相同。在下面的代码片段中，变量 p 的类型是指向 int 的指针，但 2000 的类型是 int，不能用于初始化一个指针类型的变量，所以这个声明是非法的；在第二行，一元*运算符要求它的操作数是指针类型，但 2008 的类型是 int，所以非法。我建议你创建一个带有 main 函数的源文件，将这两行添加到 main 函数里，然后尝试是否能通过翻译，并观察诊断信息都说了些什么。

```
int * p = 2000;
* 2008 = 10086;
```

然而，对于 C 这样宽容的语言来说，如果你非要将整数转换为指针，也不是不可以，因为我们有转型运算符可以使用：

```
int * p = (int *) 2000;
* (int *) 2008 = 10086;
```

在第一行里，转型表达式(int *) 2000 将 int 类型的 2000 转换为指向 int 的指针类型，然后用于初始化变量 p；在第二行里，转型表达式(int *) 2008 将 int 类型的 2008 转换为指向 int 的指针，然后，一元*运算符作用于这个指针（类型的值），得到一个左值，然后将 10086 赋给左值所代表的变量。

必须要郑重指出的是，上面的做法虽然合法，但运行的时候很危险，程序崩溃的概率几乎是百分之百。原因是，2000 和 2008 通常不会是一个合法有效的变量地址，你不知道这个地址是用来干什么的，如果它能够执行，那你就是破坏了人家原有的代码或者数据；如果它不能正常执行，那就意味着它是一片受处理器和操作系统保护的内存区域，也可能并不对应着任何实际存在的物理内存。

相比之下，下面这个程序可能会稍好一些，但也不能保证会在所有计算机上得到正确的结果，这是因为，在不同的计算机系统上，指针类型的长度可能并不相同。

```
/**********c0409.c**********/
int main (void)
{
    int m = 0;
    unsigned long long int ull;

    ull = (unsigned long long) & m;
    * (int *) ull = 10086;
}
```

以上，我们先是声明了一个 int 类型的变量 m 和一个 unsigned long long 类型的变量 ull；然后，子表达式& m 的类型是指向 int 的指针，这也是其值的类型。这个指针类型的值被转换为 unsigned long long 类型后赋给变量 ull。因为不知道指针的长度，我们使用一个最长的整数类型 unsigned long long 来保存转换后的结果。

接着，在表达式* (int *) ull = 10086 里，左值 ull 执行左值转换，转换为一个 unsigned long long 类型的值。这个值被转型运算符转换为指向 int 的指针，然后一元*运算符作用于这个指针，得到一个 int 类型的左值，这个左值接受赋值。至此，因为表达式(int *) ull 的值实际上指向变量 m，故变量 m 的存储值是 10086。

练习 4.10：

1.表达式(unsigned long long) & m 和(unsigned long long) m 所执行的转换过程有什么区别？

2.编写一个程序完成以下工作：将一个指向函数的指针转换为整数，再将这个整数转换为指向函数的指针，最后，用这个指针调用那个函数。

4.6.5.2 空指针

除非使用转型表达式，用一个整数来初始化指针类型的变量或者给指针类型的变量赋值通常是不可行的，但有一个例外，那就是整型常量 0，它不需要任何显式的转换：

```
signed char * ps1 = 0, * ps2;
ps2 = 0;
```

在 C 语言里，值为 0 的整型常量表达式可以自动转换为任何类型的指针，转换后的结果称为那种类型的空指针。空指针意味着它不指向任何有效的变量或者函数。

4.6.6 指针—指针转换

在 C 语言里，允许将一个指向变量类型的指针转换为指向另一种变量类型的指针，比如将一个指向 int 类型的指针转换为指向 char 类型的指针。

每当我们声明或者定义了某个函数时，就相当于创建了一种函数类型。参数类型不同、返回类型不同的函数属于不同的函数类型，进一步地，指向不同函数类型的指针属于不同的指针类型。

相应地，也可以将一个指向某种函数类型的指针转换为指向另一种函数类型的指针，当它再次转换回原来的类型后，和原先的指针相等。在下面的程序中，我们将一个指向某函数类型的指针转换为指向另一种函数类型的指针，然后再转换回来加以比较，比较之后再用转换回来的指针调用它所指向的那个函数。

```
/*******************c0410.c******************/
int max (int a, int b)
{
    return a >= b ? a : b;
}

int main (void)
{
    int res, (* pf) (int, int) = max;
    void (* px) (void) = (void (*) (void)) pf;

    res = (int (*) (int, int)) px == pf ? 1 : 0;
```

```
    res = ((int (*) (int, int)) px) (1, 2);
}
```

在 main 函数里，第一行声明了变量 res 和 pf，变量 res 的类型是 int，而变量 pf 的类型是指向函数的指针。变量 pf 的确切类型是指向“有两个 int 类型的参数，且返回类型是 void 的函数”的指针。变量 pf 的初始化器是函数指示符 max，将自动转换为指针，且转换后的类型与 pf 的类型一致（参见函数指示符—指针转换）。

在第二行，我们又声明了另一个指针类型的变量 px，它的确切类型是指向“参数类型和返回类型都是 void 的函数”的指针。来看它的初始化器，左值 pf 经左值转换，值的类型是 int (*) (int, int)，与变量 px 的类型不一致，不能直接用于初始化，还需要用转型运算符把它转换为变量 px 的类型，即 void (*) (void)类型。

接下来的一行看起来很复杂，但这只不过是因为它里面包含了一个转型运算符(int (*) (int, int))的缘故。在表达式

```
res = (int (*) (int, int)) px == pf ? 1 : 0
```

里，转型运算符的优先级最高，等性运算符==次之；条件运算符?:又次之；赋值运算符=的优先级最低，所以这个表达式等价于

```
res = ((((int (*) (int, int)) px) == pf) ? 1 : 0)
```

说到底，这是把条件表达式的值赋给左值 res，而条件表达式的值又取决于(int (*) (int, int)) px 和 pf 的比较结果。

等性运算符不但适用于整数，还适用于指针类型的操作数，可用于比较两个指针是否相同，即，是否指向同一个变量或者函数，或者是否都是空指针。变量 px 和 pf 都存储了函数 max 的地址，但它们的类型不同，按规定，只有指向同一种函数类型的指针才能放在一起比较。但是，如果类型不一致，它不会像对待整型操作数那样能够自动转换，指针类型的操作数必须手工转换为一致。不然的话，在翻译程序时，翻译器将愤愤不平地咕哝几句以示抗议。

为此，该表达式是把左值 px 经左值转换后得到的值强制转换为 int (*) (int, int)类型，再与左值 pf 经左值转换后的值作等性比较。对于等性运算符!=和==来说，不管操作数的类型是什么，比较的结果都是 int 类型的 0 或者 1。

最后，表达式 res = ((int (*) (int, int)) px) (1, 2)是将变量 px 的值转换为 int (*) (int, int)类型，并以这种类型调用它所指向的函数。由于函数调用运算符的优先级高于转型运算符，故还必须将转型表达式(int (*) (int, int)) px 用括号括起来，使之成为基本表达式。函数调用的结果（返回值）被赋给 res。

实际上，表达式(int (*) (int, int)) px 的值与变量 pf 的值一样，都是指向函数 max 的指针，所以上述函数调用实际上等效于 res = pf (1, 2)。

练习 4.11：

1. 在上述程序里，第一次赋值和第二次赋值后，变量 res 的值各为多少？请上机验证。
2. 若 p 和 q 都是指针类型的左值，则表达式 p != q 和 p == q 的（结果）类型是什么？若 p 的类型是指向 char 的指针而 q 的类型是指向 int 的指针，则程序翻译时会有警告信息吗？请上机实际验证。

4.6.6.1 变量地址的对齐

在前面的程序中，变量 px 的类型是 void (*) (void)。尽管它的值实际上指向函数 max，但你不能这样调用：

```
px (1, 2)
```

而只能这样调用：

px ()

原因极其简单：在程序翻译期间，C 实现要做类型检查，左值 px 的类型是指向“参数类型和返回类型都是 void 的函数”的指针，但你却传递了两个参数，这不合法。

然而，变量 px 的值实际上指向函数 max，函数调用表达式 px () 虽然合法，但却与函数 max 的声明不一致。按照规定，如果一个指向函数的指针同它实际指向的函数类型不一致，则用这个指针做函数调用时，程序的行为是未定义的。

相似地，如果将一个指向某种变量类型的指针转换为指向另一种变量类型的指针，用转换后的指针访问变量时，也会出各种问题。来看下面的程序片段：

```
char x = 0;
++ * (int *) & x;
```

在这里，变量 x 的类型是 char，只占用 1 个字节的存储空间。紧接着在第二行，一个指向变量 x 的指针被转换为指向 int 的指针，然后递增它所指向的变量。

表达式& x 的类型是指向 char 的指针；表达式(int *) & x 的类型是指向 int 的指针；表达式* (int *) & x 的结果是一个 int 类型的左值；运算符++递增这个左值所代表的变量（的存储值）。

这里的重点在于被递增的变量是 int 类型的。在我的机器上，一个 int 类型的变量占用 4 个字节的存储空间。但是实际上，有 3 个字节并不属于它。

这就尴尬了，你侵犯了别人的领土，那个地方可能属于另一个变量，这样的话你就破坏了另一个变量的值，如果那里恰巧保存的是银行账目，这就更让人头大了；那个地方也许并不对应任何变量，变量必须先分配再使用，访问一个没有分配的存储空间等于拿着空头支票去市场上买东西，当然会被拒绝，拒绝的结果就是程序可能崩溃。

变量的大小只是一个方面的问题，另一个问题是内存地址的对齐。学过计算机原理的同学都知道，处理器读写内存储器时，要先通过地址总线发送一个地址到内存储器。然而，内存储器是按字节组织的，字节是最小的可寻址单元，但它也可以每次读写 2 个字节或者 4 个字节甚至 16 个字节的数据。

这就是说，一个地址可用于访问 1 个字节单元，也可用于访问 2 个连续的字节单元，或者 4 个、8 个连续的字节单元。这种灵活性是有代价的，受硬件布线的限制，这将要求特定类型（长度）的变量只能位于特定的地址，这称为对齐。

对齐用一个整数值来描述，它必须是 2^N，且 N 是非负数。比如在我的机器上，int 类型的变量原则上只能位于 0x00000004、0x00000008、0x0000000C 等地址上，都是一些能够被 4（2^2）整除的地址，故它的对齐是 4。

在任何机器上，char 类型的变量可位于任何地址上，因为它只有一个字节，而字节是内存储器支持的最小可寻址单元。能够将所有地址整除的只有数字 1（2^0），故对于 char 类型的变量来说，其对齐始终为 1。

来看上面的例子，变量 x 的类型是 char，可位于任何内存地址上；但是，左值* (int *) & x 的类型是 int，代表一个 int 类型的变量。在我的机器上，它要求这个变量对齐于能够被 4 整除的地址上。

先不说将 char 类型的变量当成 int 类型的变量来访问是否合法，就说地址，如果变量 x 的地址是 0x00000003，那么，它并不符合 int 类型所要求的对齐，这个地址不能被 4 整除。

有些处理器是强制要求对齐的，比如 Motorola 68K 处理器，在这种计算机上，非对齐的访问将产生一个总线错误。INTEL x86 处理器也建议使用对齐的访问，但同时它并不限制你非得这么做。如果你使用非对齐的访问，它也能工作，只不过要迂回一些。

原则上，我们并不需要考虑变量的对齐问题。你编程时的任务是声明变量，不需要关心

它在哪里。在程序运行时，自然会按照它的类型把它安排在符合要求的地址上。然而，如果是通过指针访问变量，而且指针所指向的类型与它所指向的变量不符，这就是必须要考虑的问题了。

4.6.6.2 认识_Alignof运算符

一旦了解到变量在内存中的位置需要对齐到特定的地址上，你难免想知道特定类型的对齐值是多少，或者它应当位于哪些地址上。

这个问题不难解决，从C11（ISO/IEC 9899:2011）开始，C语言引入了一个新的运算符_Alignof，它用于返回指定类型的对齐值，其语法形式为：

_Alignof (*类型名*)

注意，圆括号内只能是类型名，而不能是表达式（常量、左值或者变量的名字）。我们所认识的运算符都是一些非字母的符号，比如+、=、++、>，等等，但这个运算符却完全是由字母组成，很像一个函数。看起来很荒谬，但这就是C语言。

注意，_Alignof不单单是C语言里的运算符，也是关键字。运算符_Alignof的结果类型是一种无符号整数类型，可能是unsigned int，也可能是unsigned long long int，也可能是别的，但具体是哪种整数类型取决于具体的C实现。

在下面的例子中，我们分别获取char、指向char的指针，以及指向函数的指针这三种类型的对齐值。

```
/************c0411.c**********/
int main (void)
{
    unsigned long long x, y, z;
    x = _Alignof (char);
    y = _Alignof (char *);
    z = _Alignof (int (*) (void));
}
```

不管在哪种计算机系统上，char类型的对齐值始终为1。然而，char *和int (*)(void)类型的对齐值可以随计算机系统而异。还有，尽管指针可以指向函数，但它本身并不是函数，任何指针类型都是变量类型。也就是说，任何指针类型都可用于声明变量，任何指针类型的值都可以保存在变量中。

练习4.12：

如果某类型的对齐值是8，它应当位于哪些地址上（　　）。

A.0x00000000　B.0x00000008　C.0x0000000F　D.0x00000010

4.7 指向指针（类型）的指针

我们知道，指针类型是以它所指向的类型为特征的，可以指向任何类型。那位说了，既然指针也是一种数据类型，且指针可以指向任何类型，那有没有指向指针类型的指针呢？

可以明确地告诉你，有的。在下面的程序中就有一个变量ppi，它的类型就是指向指针的指针。

```
/***************c0412.c**************/
int main (void)
{
    int i, * pi = & i, * * ppi = & pi;
    * * ppi = 10086;                //S1
```

```
    * * ppi = * * ppi + 1;          //S2

    int j;
    pi = & j;                       //S3
    * * ppi = 10010;                //S4

    int * pj = & j;
    ppi = & pj;                     //S5
    ++ * * ppi;                     //S6
}
```

在这个程序里，我们声明了变量 i、pi 和 ppi，其类型分别是 int、指向 int 的指针和指向“指向 int 的指针”的指针。对变量 ppi 的声明使用了两个星号，这个声明的解读方法如图 4-9 所示。首先从标识符 ppi 开始，然后向左读：ppi 的类型是指针，指向的类型是“指向 int 的指针”，或者说，ppi 是指向“指向 int 的指针”的指针。如果使用类型名，则变量 ppi 的类型是 int * *。

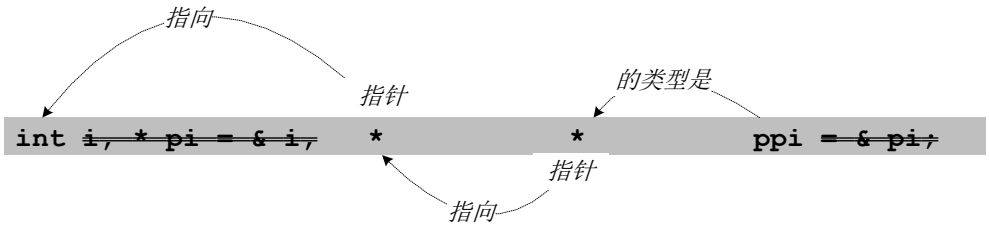


Fig.4-9 解读“指向指针的指针”的声明

在声明中，变量 pi 的类型是 int *，而表达式& i 的类型也是 int *，类型一致，可以用后者的值初始化前者。

进一步地，因 pi 是一个 int *类型的左值，故表达式& pi 的类型是 int * *，这与变量 ppi 的类型一致，可用前者的值初始化后者。

如图 4-10 所示，现在，变量 ppi 的值指向变量 pi，而变量 pi 的值又指向变量 i。通过变量 pi 可以访问变量 i，这是我们已经熟悉的。但事实上，通过变量 ppi 也可以访问到变量 i，语句 S1 就做到了这一点。

在语句 S1 中，表达式* * ppi 等价于* (* ppi)，这是因为一元*运算符是从右往左结合的。左值 ppi 经左值转换后得到一个 int * *类型的值（这个值实际上指向变量 pi），一元*运算符作用于它，得到一个 int *类型的左值（代表变量 pi）。这个左值继续执行左值转换，得到一个 int *类型的值（这个值实际上指向变量 i），最左边的一元*运算符作用于它，得到一个 int 类型的左值（代表变量 i）。最终得到的这个左值是赋值运算符的左操作数，不执行左值转换，且被赋值为 10086。赋值后，变量 i 的值是 10086。

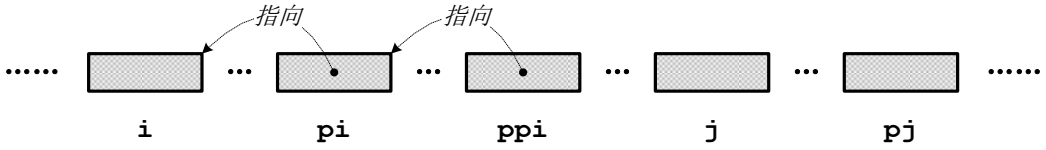


Fig.4-10 程序中的变量及指针的指向关系

同理，在语句 S2 中，表达式* * ppi 是左值，代表变量 i。但赋值运算符左边的* * ppi 不执行左值转换，而用于接受赋值；赋值运算符右边的* * ppi 执行左值转换，转换

后的结果与整数 1 相加。这条语句执行后，变量 i 的值是 10087。

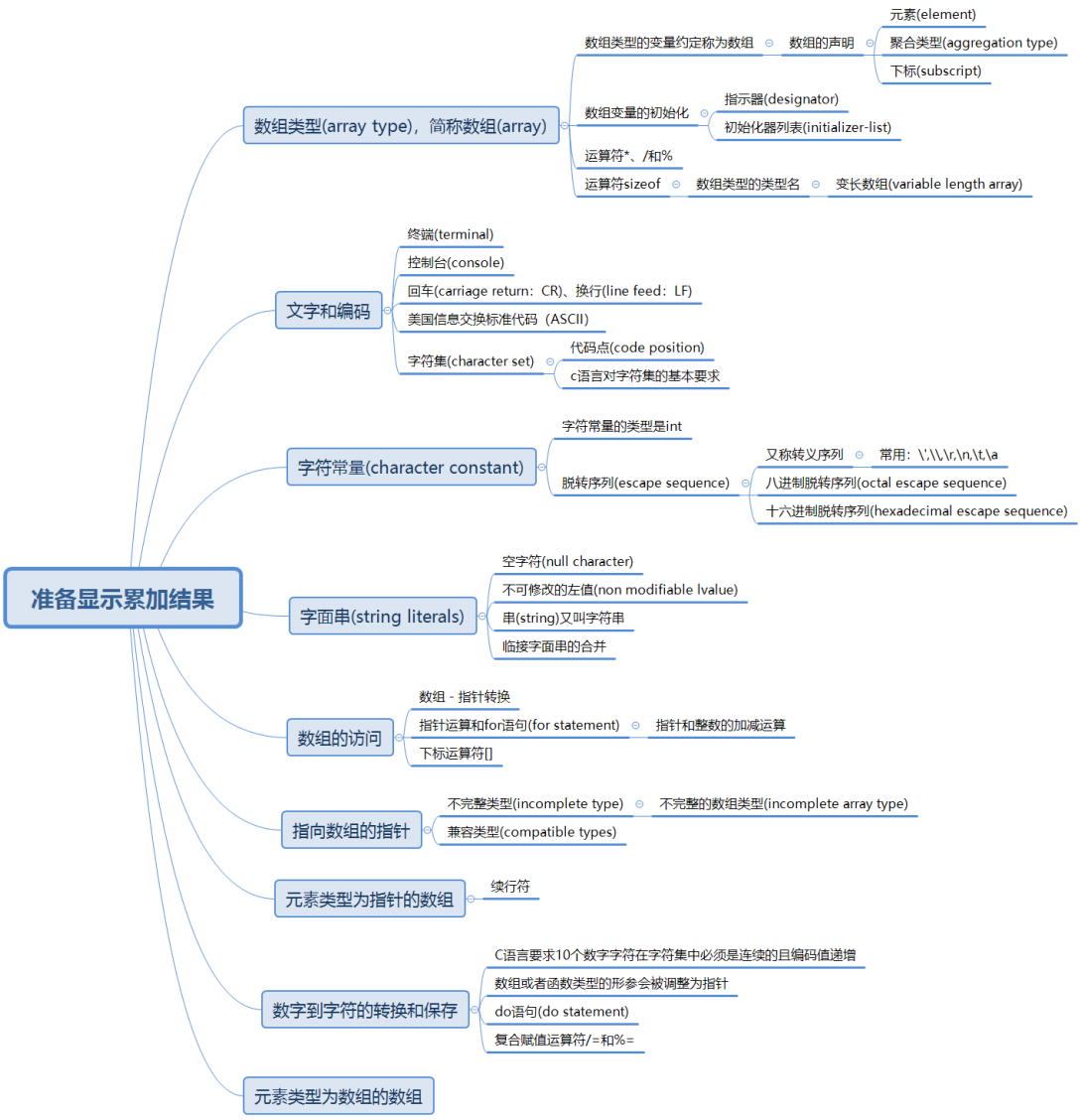
指针可以灵活地指向不同的变量，但这也意味着通过该指针所访问的变量也不一样。在语句 S3 中，变量 pi 的值被改为指向另一个不同的变量 j。于是在语句 S4 中，尽管子表达式* ppi 的结果是左值，依然代表变量 pi，但表达式* * ppi 的结果却是代表变量 j 的左值。所以这条语句是改变了变量 j 的存储值，赋值后，变量 j 的值为 10010。

语句 S5 用于修改变量 ppi 的值，令其指向另一个指针类型的变量 pj。而在此之前，变量 pj 被声明为指向 int 的指针，且被初始化为指向变量 j。

在语句 S6 中，子表达式* ppi 的结果是一个左值，代表变量 pj，而变量 pj 的值现在是指向变量 j 的。所以表达式* * ppi 的结果是代表变量 j 的左值。前缀递增运算符作用于这个左值，将递增它所代表的那个变量（变量 j）的存储值。

练习 4.13：

指针讲到这里，你应该对指针变量的声明技巧有所领悟，知道圆括号具有结合和分隔的作用。如果可能的话，请尝试声明一个变量 ppf，其类型为指向“指向参数类型为 void，返回类型为 int 的函数的指针”的指针。


准备显示累加结果
数组类型(array type)，简称数组(array)
数组类型的变量约定称为数组
数组的声明
元素(element)
聚合类型(aggregation type)
下标(subscript)
数组变量的初始化
指示器(designator)
初始化器列表(initializer-list)
运算符*、/和%
运算符sizeof
数组类型的类型名
变长数组(variable length array)
文字和编码
终端(terminal)
控制台(console)
回车(carriage return：CR)、换行(line feed：LF)
美国信息交换标准代码（ASCII)
字符集(character set)
代码点(code position)
c语言对字符集的基本要求
字符常量(character constant)
字符常量的类型是int
脱转序列(escape sequence)
又称转义序列
常用：\',\\,\r,\n,\t,\a
八进制脱转序列(octal escape sequence)
十六进制脱转序列(hexadecimal escape sequence)
字面串(string literals)
空字符(null character)
不可修改的左值(non modifiable lvalue)
串(string)又叫字符串
临接字面串的合并
数组的访问
数组 - 指针转换
指针运算和for语句(for statement)
指针和整数的加减运算
下标运算符[]
指向数组的指针
不完整类型(incomplete type)
不完整的数组类型(incomplete array type)
兼容类型(compatible types)
元素类型为指针的数组
续行符
数字到字符的转换和保存
C语言要求10个数字字符在字符集中必须是连续的且编码值递增
数组或者函数类型的形参会被调整为指针
do语句(do statement)
复合赋值运算符/=和%=
元素类型为数组的数组

第 5 章　准备显示累加结果

截至目前，我们所处理的数据在规模上非常小。然而有时候我们需要处理大量的、成组的数据。比如一个年级有 300 个学生，为了统计他们的最高成绩、最低成绩、总成绩和平均成绩，或者对成绩进行排序，就必须先将每个学生的成绩存储起来。

怎么存储呢？就我们已经学过的知识而言，唯一的方法就是声明 300 个变量。这当然是非常笨拙的方法，我们也不会这么做，因为 C 语言为我们提供了数组，它可以更有效地组织数据，哪怕是比 300 个更多也不怕。

5.1 什么是数组

从语文上来讲，所谓数组，顾名思义，就是“一组数据”　“成组的数据”，或者说是把多个数据合为一体而形成的数据单位。在 C 语言里，要声明单个 int 类型的变量，我们需要这样做：

```
int var;
```

那么，要声明一大堆，比如 5 个 int 类型的变量（好吧，我知道 5 个不算一大堆，就是举个例子），该怎么办呢？我们不需要 5 个标识符（变量名），只需要指明这个变量由 5 个子变量组成就行：

```
int vars [5];
```

以上，我们就声明了一个特殊的变量，标识符 vars 指示或者说代表一个在程序运行时创建的变量，可以更简单地称之为“变量 vars”；用方括号“[”和“]”括住的数字是子变量的数量，在这里是 5，表明变量 vars 由 5 个子变量组成；最左边的 int 是类型指定符，用于指定所有子变量的类型。

为了描述像 vars 这种类型的变量，C 语言引入了数组类型，简称数组。如图 5-1 所示，对于每个数组类型的变量来说，它是由一系列在存储器中连续分配的子变量组成的，“连续”的意思是强调它们在存储器里必须按顺序一个挨着一个。在上例中，变量 vars 是数组类型的变量，它由 5 个子变量组成。

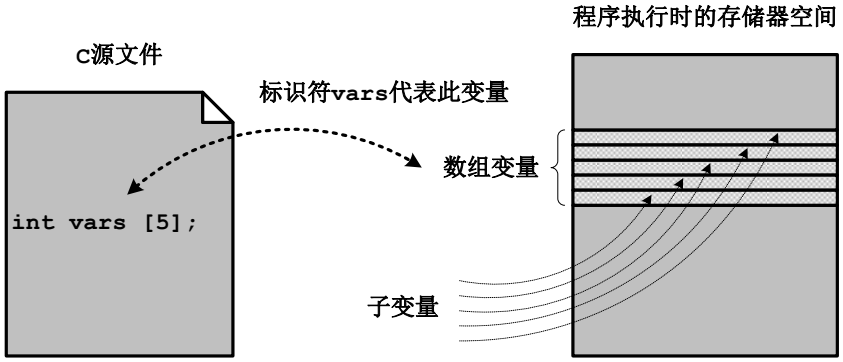


Fig.5-1 数组类型的变量示意图

然而，人们也经常把数组类型的变量叫作数组。例如，如果我们声明了一个数组类型的变量 a，则称之为“数组 a”；再比如上例中，数组类型的变量 vars 简称为数组 vars。这是一种习惯性的叫法，约定俗成。当然，为了更清楚地表达出类型和实体两方面的信息，在本书中也称为“数组变量”。

显然，“数组”一词可能是数组类型，也可能指数组类型的变量，但通常可以借助上下文清楚地分辨所指。当然，本书也尽可能地使用“数组类型”和“数组变量”以明确地加以区分。

对于每个数组类型的变量来说，它的子变量称为数组的元素。数组的元素不但要连续分配，一个挨着一个，还必须具有相同的类型，称为数组的元素类型。在上例中，变量 vars 的元素类型是 int。

5.1.1 数组变量的声明

在对数组有了基本的认知后，我们再来看数组声明的要素和特点。我们知道，解析一个声明要从标识符开始，向左或者向右读。特别地，如果标识符的右边是“[”或者“(”，则必须先向右读。

如果标识符右边是“[”，则它代表一个数组。如图 5-2 所示，在上述声明中，标识符 vars 的右边是“[”，则我们先要向右读，读作“vars 的类型是数组”，或者“vars 是一个数组”。

既然是数组，那么，要是方括号里有数字，那就是数组的元素数量，就继续向右读它的元素数量，即“该数组有 5 个元素”。读完元素数量后，继续向右直至遇到配对的“]”，看“]”的右边有没有东西。如果没有东西，就转而向左读，在这里会遇到“int”，这是数组的元素类型，读作“元素的类型是 int”。

最后，整个过程合在一起，读为“vars 是一个数组，该数组有 5 个元素，元素的类型是 int”。

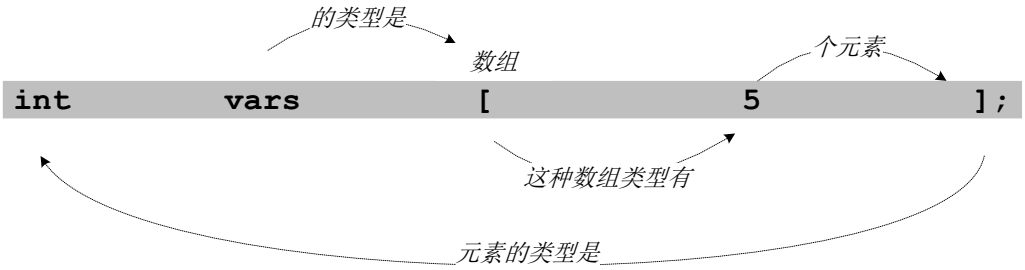


Fig.5-2 数组变量的声明示意图

数组类型是个统称，不同的数组类型是以它们的元素类型和元素数量为特征的。也就是说，元素数量不同，元素类型不同的数组，属于不同的数组类型。

在下例中声明了 4 个数组，数组 a 和数组 b 的类型相同，都属于同一种数组类型，因为它们的元素数量相同，元素的类型也相同；除此之外，任何其他两个数组的类型彼此不同，要么是因为元素数量不同，要么是因为元素类型不同。

```
int a [5], b [5], c [3];
signed char d [256];
```

数组由元素（子变量）聚在一起合成，所以被称为聚合类型。数组变量的元素没有名字，所以只能按序号访问，每个元素的序号称为下标。下标是一个整数，第 1 个元素的下标是 0，第 2 个元素的下标是 1，第 3 个元素的下标是 2，后面的元素依次类推。如果数组有 N 个元素，则最后一个元素的下标是 N-1。

练习 5.1：

1.某数组有 7 个元素，第 6 个元素的下标是几？

5.1.2 数组变量的初始化

我们已经讲过，可以在变量的声明中用符号“=”连接一个初始化器，当变量创建时，就用初始化器的值来初始化变量，例如：

```
int x = 6, * p = & x;
```

如果一个变量以其类型而言是由子变量组成，则它的初始化器必须用一对花括号“{”和“}”围起来。在数学上，我们也是用花括号来形成一个集合。

既然是子变量，那么，每个子变量也都需要一个初始化器。所以，初始化器是迭代组成的，一个数组变量的初始化器由子变量的初始化器组成，子变量的初始化器用逗号“,”分隔开，用来为对应的元素（子变量）提供初始值，例如：

```
/*************c0501.c*************/
int main (void)
{
    int studs [5] = {1, 2, 3, 4, 5};
}
```

在这里，{1, 2, 3, 4, 5}是数组变量 studs 的初始化器，表达式 1、2、3、4、5 分别是其每个元素（子变量）的初始化器。表达式 1 为数组 studs 的第一个元素提供初始值；表达式 2 为数组 studs 的第二个元素提供初始值，后面依次类推。

之所以这里的 1、2、3、4、5 都没有用花括号围起来，是因为数组 studs 的每个元素都不再包含子变量。

现在，我们将上面的程序保存为源文件 c0501.c，然后用-g 选项翻译为可执行程序。在 gdb 中，将断点设置在右花括号“}”所在的那一行，运行程序，用调试命令“p studs”观察数组的内容。如下面的交互过程所示，gdb 以集合的形式显示数组的每个元素：

```
(gdb) p studs
$1 = {1, 2, 3, 4, 5}
(gdb)
```

数组元素的初始化器还可以包含一个指示器，用于指定初始化哪个元素。指示器是一个用方括号围起来的元素下标，例如：

```
/**********c0502.c********/
int main (void)
{
    int a [50] = {[22] = 8};
}
```

这里，[22] = 8 是子变量的初始化器，它由指示器“[22]”、符号“=”，以及一个用于提供实际数值的表达式 8 组成，该初始化器用于初始化那个下标为 22 元素，因为这是指示器指定的元素。

现在，我们将上面的程序保存为源文件 c0502.c，然后用-g 选项翻译为可执行程序。在 gdb 中，将断点设置在右花括号“}”所在的那一行，运行程序，用调试命令“p a”观察数组的内容。如下面的交互过程所示，gdb 以集合的形式显示数组的每个元素：

```
(gdb) p a
$1 = {0 <repeats 22 times>, 8, 0 <repeats 27 times>}
(gdb)
```

和上一个例子不同，这一次 gdb 的 p 命令似乎并未显示所有数组元素的值。实际上它已经全部予以显示，只不过比较简略。首先，“0 <repeats 22 times>”的意思是有 22 个重复出现的 0（包括已经显示的这个 0）。换句话说，数组的前 22 个元素都是 0。然后，中间的 8 表示第 23 个元素的值是 8；最后，“0 <repeats 27 times>”的意思是有 27 个重复出现的 0（包括已经显示的这个 0）。换句话说，从第 24 个元素开始，一直到数组的最后一个元素，它们的值都是 0。

如果数组的初始化器里没有指示器，那么，初始化的顺序是按元素的下标顺序进行，先

初始化下标为 0 的元素，然后是下标为 1 的元素，其他依次类推；如果初始化器的数量少于元素的数量，则剩余的元素被初始化为 0；如果数组的初始化器里有指示器，则后续的初始化将延续指示器的下标进行，直至遇到另一个指示器。

在下面的示例中，数组 a 中下标为 0 的元素被初始化为 1，下标为 1 的元素被初始化为 2，其余的元素被初始化为 0；数组 b 中下标为 0 的元素被初始化为 3，下标为 11 的元素被初始化为 33，下标为 9 的元素被初始化为 5。紧接着，表达式 1 用于初始化下标为 10 的元素，表达式 2 用于初始化下标为 11 的元素，剩余的元素统统被初始化为 0。

```
/************c0503.c**********/
int main (void)
{
    int a [22] = {1, 2}, b [22] = {3, [11] = 33, [9] = 5, 1, 2};
}
```

注意，数组 b 中下标为 11 的元素被初始化两次，但它将保留最后一次被初始化的数值 2，而不是第一次的数值 33。

5.1.3 认识 sizeof 和乘性运算符

如果一个数组的声明中仅有初始化器而未指定数组的元素数量，则数组的元素数量由初始化器决定。具体来说，在所有被初始化的元素中，总有一个下标最大，数组的元素数量由这个元素的下标来确定。

如下面的声明所示，对于数组 a，初始化器 1 和 2 分别用于初始化下标为 0 和 1 的元素，初始化器[199] = 8 和[99] = 7 分别用于初始化下标为 199 和 99 的元素。其中最大的下标是 199，所以数组 a 有 200 个元素。

```
/************c0504.c************/
int main (void)
{
    int a [] = {1, 2, [199] = 8, [99] = 7}, b [] = {1, 2, 3};

    unsigned int siza = 0, sizb = 0, numa = 0, numb = 0;
    siza = sizeof a;                    //S1
    sizb = sizeof b;                    //S2
    numa = sizeof a / sizeof (int);     //S3
    numb = sizeof b / sizeof (int);     //S4
}
```

对于数组 b，初始化器 1、2 和 3 分别用于初始化下标为 0、1 和 2 的元素，最大的下标是 2，所以该数组有 3 个元素。

现在的问题是，数组 a 和数组 b 到底有多大，占用多大的内存空间？它们的元素数量真的如上所说吗？为了加以验证，接下来，我们声明了 4 个变量。siza 和 sizb 分别用于保存数组 a 和数组 b 的大小，以字节计；numa 和 numb 分别用于保存数组 a 和数组 b 的元素个数。

为了获得一个变量或者类型的大小，C 语言从发明之初就引入了 sizeof 运算符。和我们曾经学过的_Alignof 一样，它看起来像是函数，但并不是函数。sizeof 不单单是 C 语言里的运算符，也是关键字，Sizeof、SizeOf 等都是错误的拼写。

sizeof 运算符只需要一个右操作数，可以是表达式，也可以是用圆括号“()”括住的类型名：

sizeof *表达式*

sizeof (*类型名*)

由运算符 sizeof 和它的操作数组成的表达式称为尺寸表达式，或者叫 sizeof 表达式。显然，sizeof 3、sizeof 3ULL、sizeof (char)、sizeof (int)、sizeof (int *)和 sizeof (int (*) (void))都是合法的尺寸表达式。

运算符 sizeof 的结果是其操作数的大小，以字节计，结果的类型是一个无符号整数类型，具体是哪种无符号整数类型，由 C 实现自行决定，但必须足够大，以保证能够容纳所有类型的大小。

如果 sizeof 运算符的操作数是一个类型名，则它返回类型的大小——类型可用于声明变量，它决定了变量的大小，这个大小可视为类型的大小。因此，表达式 sizeof (int *)返回“指向 int 的指针”类型的大小；表达式 sizeof (int (*) (void))返回“指向函数的指针”类型的大小；表达式 sizeof (char)和 sizeof (int)分别返回 char 类型和 int 类型的大小。

如果 sizeof 运算符的操作数是一个表达式，则它仅抽取表达式的类型。这里，3 是整型常量表达式，其类型为 int，故表达式 sizeof 3 的结果是 int 类型的大小；3ULL 也是整型常量表达式，其类型为 unsigned long long int，故表达式 sizeof 3ULL 的结果是 unsigned long long int 类型的大小。

如果运算符 sizeof 的操作数是一个代表变量的表达式，即，左值，则不执行左值转换而直接抽取左值的类型，并返回该类型的大小。在语句 S1 中，表达式 sizeof a 的结果是变量 a 的大小，以字节计。尽管 a 是个左值，但并不执行左值转换。那么，sizeof a 的结果是多少呢？变量 a 是一个数组，而数组的大小取决于元素的数量和每个元素的大小，或者说是元素的数量乘以每个元素的大小。然而，sizeof 并不是通过访问数组 a 来得到这个总大小的，相反，它仅仅是依靠 a 的类型来计算的。

类似地，语句 S2 则用于取得变量 b 的大小；语句 S3 的作用是计算数组 a 的元素个数，它的做法是用数组的大小除以每个元素的大小。在这里，运算符 sizeof 的优先级最高，运算符/次之，运算符=的优先级最低。

运算符/属于乘性运算符。乘性运算符都是二元运算符，也就是需要一左一右两个操作数，它们包括*、/和%。二元*运算符的结果是两个操作数的乘积；运算符/的结果是其左操作数除以右操作数的商，如果两个操作数都是整数，则运算符/的结果是一个舍弃了小数部分的整数；运算符%的两个操作数只能是整数类型，其结果也是一个整数，而且是其左操作数除以右操作数之后所得到的余数。

也就是说，表达式 15 * 6 的结果是 90；表达式 15 / 6 的结果是 2；表达式 15 % 6 的结果是 3。

再来看语句 S3，表达式 sizeof a 得到数组 a 的大小，以字节计，它是所有元素大小的总和。因为元素的类型是 int，故每个元素的大小就是 int 类型的大小，所以我们是用表达式 sizeof (int)来得到每个元素的大小。最后，这两者相除，就是元素的数量，我们将它赋给变量 numa。同样地，语句 S4 也用这种方法来计算数组 b 的元素数量。

要想知道数组 a、b 的大小和元素的数量，最可靠的办法就是在调试器里观察变量 siza、sizb、numa 和 numb 的值。将断点设置在组成函数体的右花括号“}”所在的那一行，然后用 p 命令打印变量的值：

```
(gdb) p {siza, sizb, numa, numb}
$1 = {800, 12, 3, 200}
(gdb) p sizeof (int)
$2 = 4
```

```
(gdb)
```

因为变量 siza、sizb、numa 和 numb 的类型都相同，可以用集合的形式打印。p 命令不限于打印变量的值，而是可以打印任何表达式的值，所以最后一个调试命令打印了 int 类型的大小，在我的机器上，每个 int 类型的对象占据 4 个字节的内存空间。

在带有花括号的初始化器中，被花括号包围的部分称为初始化器列表。在初始化器列表的末尾可以多添加一个逗号“,”。例如：

```
int a [5] = {1, 2, 3, 4, 5,};
```

这个多余的逗号是无害的，引入它的动机和源代码的版本控制有关。在大公司和大项目中，团队开发是常见的事。通常情况下，一个大的软件开发项目会进行分解，并交由不同的人负责完成。然而在实际的工作中难免会有一些交叉的部分，比如两个人都要使用同一个源文件。

团队协作，软件的版本控制非常重要。所有软件的源代码都存放在数据库里，当程序员要求修改某个源文件时，他要先执行一个检出操作，从数据库里调出该文件的最新版本。之后，他可能会修改这个文件，删除一些东西，或者添加一些代码。如果他认为很满意，就需要执行一个检入操作，这将在数据库里生成一个该文件的最新版本，但是以前的版本不受影响。这样，即使第二天他发现前一天的修改非常愚蠢，也可以很容易回退到历史版本。

如果多个程序员的工作都涉及同一个源文件，则他们将产生冲突。安全起见，如果一个程序员在检出时，文件被锁定且不允许其他程序员修改这个文件，直到该程序员检入这个文件并解除锁定。

当然，版本控制系统也可以被设定为允许多个程序员同时修改同一个源文件。在这种情况下，如何协调冲突是非常重要的。假定某个源文件当前的修订版本是 Rev5，其中有这样一个数组声明（注意，程序员都是爱美的人，很注意代码的排版和缩进效果）：

```
int b [] = {
              1, 2, 3,
              4, 5, 6
            };
```

真是凑巧，程序员 A 和程序员 B 都检出该文件并进行了表 5-1 所示的修改。程序员 A 先做了检入操作，检入后，版本控制系统将生成最新版本 Rev6。

表 5-1 程序员 A 和程序员 B 对同一文件的不同修改

程序员 A 的修改	程序员 B 的修改

int b [] = {
1, 2, 3,
40, 50, 60
};
	int b [] = {
	1, 2, 3,
	4, 5, 6,
	7, 8, 9,
	};

于是，当程序员 B 检入的时候，版本控制系统将报告当前文件已经过期，而且他的修改与新版本的某些内容冲突。此时，程序员 B 可以查看程序员 A 都做了哪些修改，并打电话或者发电子邮件与他进行沟通。

经过沟通，程序员 B 意识到程序员 A 的修改很有道理，而他自己添加的那 3 个数也很有必要。怎么办呢？他将选择用程序员 A 的修改更新第 2 行的内容，并使自己新加的那一行也有效。此时，在他即将检入的文件中，数组的最终声明是：

```
int b [] = {
              1, 2, 3,
              40, 50, 60
```

```
    7, 8, 9,
};
```

注意，第 2 行的末尾没有逗号，因为程序员 B 认可程序员 A 对第 2 行的修改，而程序员 A 的这一行本来就没有逗号。在这种情况下，程序员 B 不得不手动为第 2 行添加一个逗号并使此修改有效。最终，程序员 B 检入他最终的修改，也更新到 Rev6。

显然，如果在原始版本 Rev5 中的第 2 行本来就有一个逗号，像添加逗号这种额外的操作就可以省略。

5.1.4 认识变长数组

在 C99 之前，sizeof 运算符并不求值它的操作数，而仅仅是提取它的类型。换句话说，该运算符的结果在程序翻译期间就已经得到了，而且是一个整型常量。

在下面的程序中，在 D1 处声明了变量 siz，初始化为表达式 sizeof 0 的值。因为 0 是个整型常量，其类型为 int，故该表达式等效于 sizeof (int)，也就是得到 int 类型的大小。

在语句 S2 中，sizeof 运算符的操作数是一个赋值表达式，但实际上并不求值。也就是说，它并不会真的把 1 赋给左值 c。

因为 sizeof 运算符的优先级高于赋值运算符，所以要将表达式 c = 1 用括号变成基本表达式（括住的表达式），否则它将等价于 (sizeof c) = 1，这在语法上将导致一个错误，因为 sizeof c 的结果并不是一个左值。

另一方面，每个表达式都有类型，表达式的类型也是该表达式的值的类型。之所以可以提取赋值表达式 c = 1 的类型，是因为赋值表达式的类型就是赋值运算符的左操作数的类型。在这里，左值 c 的类型为 signed char，故 sizeof (c = 1) 等效于 sizeof (signed char)。

C 语言规定，当 sizeof 运算符作用于 char、signed char 和 unsigned char 类型的操作数时，其结果为 1。因此，sizeof (c = 1) 的结果是 1。

```
/***************c0505.c***************/
unsigned int var_arr (int n)
{
    signed char va [n];

    return sizeof va;                       //S1
}

int main (void)
{
    unsigned int n = 30, siz = sizeof 0; //D1

    signed char c = 0;
    siz = sizeof (c = 1);                   //S2

    siz = sizeof (int [30]);                //S3
    siz = var_arr (n);                      //S4
    siz = sizeof (int [++ n]);              //S5
```

```
    siz = sizeof sizeof c;                    //S6
}
```

为了方便描述数组类型，我们也可以使用类型名。回忆一下什么是类型名，类型名可以从声明中抽取，数组类型是以元素类型和元素数量为特征的，利用这一特点不需要借助于声明就可以写出类型名。比如说，数组 a 有 20 个元素，且元素的类型是 int，则我们称 a 的类型是 int [20]。

C 语言规定，如果 sizeof 的操作数是数组类型，则该运算符的结果是数组的大小，以字节计。在这里，操作数既可以是数组类型的变量，也可以是数组类型名。因此，在接下来的语句 S3 中，因为 sizeof 运算符的操作数是一个类型名，代表着“具有 30 个元素，且元素类型是 int”的数组类型，故该运算符的结果是 sizeof (int)×30。

在 C99 之前，声明一个数组时，数组的大小必须是一个整型常量。然而从 C99 开始这一规则被打破了，可以在数组的声明中使用变量来指定它的大小，以这种方式声明的数组称为变长数组。

在上面的程序中，我们在函数 var_arr 里声明了一个变长数组 va，它的长度来自一个变量（函数的参数）n 而不是常量。显然，因为数组的大小直接依赖于函数调用时传入的参数值，你无法预测传入的值会是什么。这就是说，变长数组的大小不能在程序翻译期间得到，而只有在程序实际运行时才能确定。

对于一个完整的数组类型来说，元素的数量和元素的类型缺一不可。要想从一个变长数组类型的操作数中提取类型信息（元素的数量和元素的类型），则 sizeof 运算符必须在程序实际运行时才能开始工作。因此，在语句 S1 中，表达式 sizeof va 的结果不是在程序翻译期间得到，而是要推迟到当前语句实际执行的时候。此时，变量 va 已经声明，n 的值和数组的大小都已经确定。

无论如何，因为数组 va 的元素类型是 signed char，所以在语句 S4 中，函数调用 var_arr 的返回值和变量 n 的值相同，都是 30。然后，这个 30 被赋给变量 siz。

接下来，在语句 S5 中，运算符 sizeof 的操作数是一个类型名，但它描述的是变长数组，其大小需要在程序运行期间求值表达式++ n 才能确定。C 语言规定，如果运算符 sizeof 的操作数是变长数组类型，则求值该操作数。

前缀++表达式的值是其操作数递增后的值，所以，表达式++ n 的值是 31，求值的副作用是变量 n 的值递增为 31。既然元素的数量是 31 个，而元素的类型是 int，所以运算符 sizeof 的结果是 sizeof (int)×31。

运算符 sizeof 是从右往左结合的，因此，在最后一条语句 S6 中，表达式 sizeof sizeof c 等价于表达式 sizeof (sizeof c)。但是，这里并不求值表达式 c 也不求值表达式 sizeof c。我们说过，运算符 sizeof 有自己的结果类型（是一个无符号整数类型），所以左边那个 sizeof 将返回这个类型的大小。在我的机器上，这个类型等价于 unsigned long long int，所以上述语句等效于

```
siz = sizeof (unsigned long long int);
```

数组声明中的初始化器可以少于元素的数量，也可以等于元素的数量，但就是不能多于元素的数量。

5.2 文字和编码

在计算机中，文字信息的存储、传输和处理是相当重要的工作，而 C 语言里的数组又是容纳文字信息的最佳容器。

不管是学生成绩，还是代码、声音、图像和文字，在计算机内部看起来都一样，都表现为无差别的数字，要看你怎么去解释和处理它们。不过话又说回来了，本质上不同的东西，

即便都是数字，它们也具有不同的规律和内在逻辑。比如说，你在键盘上敲出的文章是一大堆数字，数码相机生成的也是一大堆数字，虽然用二进制编辑器打开之后看起来没有什么区别，但你无法用图片浏览器来打开你的文章，因为这根本就不是合法的图像文件；你也无法用文本编辑器打开你的照片，因为它本来就不是文本。

这就是说，如果你用数组来保存一段文字，那么，尽管你保存的实际上是一长串数字，但它们实际上是字符在计算机内部的编码（代码）。

在不同的设备之间传输文本信息，传输的实际上是每个字符的代码。这就涉及一个非常关键的工作：必须制定一个所有设备都认可的字符编码标准，文本的发送方和接收方都知道每个数字代表的是什么字符。否则的话，当你的朋友用手机给你发送一条短信时，你的手机将无法识别，也不能正确显示这些字符。

要做到这一点，首先必须建立一个字符集，或者说字符表。世界上的语言文字那么多，所以这并不是一件轻而易举的工作。然而在计算机发展的早期，没有人会考虑那么长远，因为没有人会想到计算机会流行到每个人都有好几个，像桌面计算机、智能手表、手机、平板电脑这些都是计算机。

在那个时代，包括 C 语言诞生的时候，显示器还不是标准配置，对计算机的控制往往通过电传打字机进行。电传打字机在当时是比较先进的设备了，它是打字机、打印机、卡片阅读机和纸带穿孔机的集合体。电传打字机的打印机可以在纸上打印字符，相当于现在的显示器；打字机呢，相当于现在的键盘。电传打字机可以把输入，也就是人类通过打字机进行的操作传送到计算机，而打印机则可以把计算机的响应打印在纸上。

当时还没有个人计算机，有的只是非常昂贵的大家伙，称为主机。当时也已经有了多用户的操作系统，也就是允许很多人通过电传打字机来共享同一台电脑主机的计算能力。在这种情况下，每一台电传打字机就是一个终端。有些终端离主机很近，有些则很远，需要通过电话线和调制解调器来与主机连接。

在一部叫《Apartment》的外国黑白电影中，我们的主人公 C.C.巴克斯特就职于联合保险公司的普通保单清算部，差不多有百十多号人，每人面前都有一个电传打字机，它们都连接到据说是 IBM 公司的大型主机上。

Fig.5-3 电传打字机

普通的终端对主机的操作能力有限，所以每个主机通常还有一个身份特殊的终端，它是主机的一部分，用于直接对主机进行控制，叫作控制台。比如说，重启主机就只能在控制台上进行，对主机的调整也只通过控制台进行。

这样的计算机系统在现在看来是相当粗笨的，但那个时候却很先进。在这种计算机系统上，诞生了我们现在学习的 C 语言，也产生了 UNIX 操作系统。所以直到现在，C 语言和

UNIX 系统都还保留着那个时代的很多印记。

由于这种在现在看来极不直观的操作环境，所有的设计都只能注重实用，能少打字就少打字，以简单为美，所以 C 语言和 UNIX 系统的语法和命令都非常简洁。例如，输入一个命令之后如果在执行的过程中没有错误，则系统不会显示任何消息，即所谓的“没有消息就是最好的消息”，这很可能是为了节约纸张。

再比如，尽管现在我们用的是显示器和键盘，但它们依然保留了那个时代的很多做法和叫法。我们现在把基于命令行的显示器称为控制台；在屏幕上输出字符称为“打印”；光标移到下一行称为“换行”，移到行首称为“回车”，等等，这都是那个时代才有的叫法。

有些设备可以和主机交换任何数据，例如一个外部的存储设备。这样的设备并不关心主机传来的是什么，也不试图去理解或者解释数据的含义，它只负责将数据写在磁盘或者磁带上，或者把它们读出来传送给主机。对于这样的设备和数据传输，我们称之为“纯二进制模式”的通信。

但是，像电传打字机这样的设备就不同了，发明它们的目的就是为了处理文字符号，它们必须解释主机发来的内容（文字编码）并在纸上打印出形状来。而且，用户在键盘上敲击一下，那是一个字符，要编码之后发送到主机。对于这样的设备和数据传输，我们称之为“文本模式”的通信。

终端和主机之间要想正常通信，必须有一个双方都能识别的字符编码方案，这并不是什么困难的事情。计算机诞生在美国，美国人心想，我们只有 26 个大写字母和 26 个小写字母，以及 10 个阿拉伯数字，外加一些标点符号和用来控制设备通信的代码，这很容易。所以一些大的计算机厂商各自设计了一些字符集和编码方案，比如 EBCDIC 字符集和 ASCII 字符集，等等，让来自不同厂家的设备能够互联。其中，1967 年由美国国家标准协会牵头设计出的美国信息交换标准代码（简称 ASCII）最为流行，并一直延续到现在。

创建了字符集，事情只能算是完成了一半，因为还没有为每个字符分配代码。每个被选入字符集的字符，在整个字符集中的位置是固定的，类似于每个字在字典中都占有一个固定的位置。从第一个字符开始，每个字符都有一个序号，这叫作代码点或者代码位置。

然而，代码点并不是字符编码，它仅仅是一个数学意义上的数字，指示字符在表中的位置。而字符编码（代码）呢，通常是由代码点转换而来，但是考虑到现实的需求和软硬件的限制，可能会有不同的编码方案。

对于像 ASCII 这样很小的字符集来说，字符的编码工作十分简单。美国人的做法是直接将代码点当字符代码来用。比如说，字符“A”是 ASCII 中的第 65 个字符，所以该字符的编码是 65。由于 ASCII 只有 128 个字符，所以只需要 7 个比特就能编码所有字符。现代的计算机每个字节至少有 8 个比特，在存储 ASCII 字符时，第 8 个比特置 0，这个特点深远地影响了其他国家和地区的字符编码方式。

5.2.1 字符数组

你可以想到，汉字有好几万个，无论如何也不可能用 8 比特的长度来表示它们的编码。这怎么办呢？办法还是有的，但要在后面的章节里讨论，现在先了解一下英语系国家和地区的单字节编码。

为了在数组里保存一串英文字符，比如“Chinese”，该怎么办呢？这很简单，直接将每个字符的编码写进数组就可以了。在 C 语言里有关键字和各种各样的运算符，它们都是字符或者由字符组成，比如赋值运算符“=”和关键字“sizeof”。如果一台计算机所使用的字符集里没有这些字符，那你就无法在这台计算机上用 C 语言编程。

所以，尽管 C 语言对使用何种字符集不做限定，对字符如何编码也不关心，但最基本的要求还是有的。具体地说，只有包含以下字符的字符集才能够被 C 语言所接受：

26 个大写英文字母：

```
A   B   C   D   E   F   G   H   I   J   K   L   M
N   O   P   Q   R   S   T   U   V   W   X   Y   Z
```

26 个小写英文字母：

```
a   b   c   d   e   f   g   h   i   j   k   l   m
n   o   p   q   r   s   t   u   v   w   x   y   z
```

10 个十进制数字字符：

```
0   1   2   3   4   5   6   7   8   9
```

29 个图形字符：

```
!   "   #   %   &   '   (   )   *   +   ,   -   .   /   :
;   <   =   >   ?   [   \   ]   ^   _   {   |   }   ~
```

以及空格、水平制表符、垂直制表符、换页符（传统上，这些字符用于控制显示设备或者电传打字机的字符定位）。

绝大多数计算机系统所使用的字符集虽然不是 ASCII，但与它兼容，上述字符的编码也一样。考虑到这一点，如下面的程序所示，我们可以在声明一个数组的时候，直接用字符的 ASCII 编码来初始化它。

```
/********************c0506.c********************/
int main (void)
{
    char a [7] = {67, 104, 105, 110, 101, 115, 101};
    char b [7] = {0x43, 0x68, 0x69, 0x6e, 0x65, 0x73, 0x65};
}
```

在程序里，我们声明了一个数组 a，它有 7 个元素，分别初始化为字符的编码。67 是字符“C”的 ASCII 编码；104 是字符“h”的 ASCII 编码，其他依次类推。显然，如果不告诉你数组 a 里存放的是字符，你一定会认为存放的是整数，这当然也是非常正确的，毕竟数字的含义要看你怎么使用和解释。

值得注意的是，数组 a 的元素类型是 char，这是有历史原因的。在 C 语言发明的时候，字符集都很小，用 char 类型的变量来保存字符编码是恰当的，所以很多人把 char 类型称为字符类型。

不过，数组 a 的初始化器是由整型常量组成，这些常量的类型是 int，但数组 a 的元素类型是 char，类型并不匹配。

char 类型可能等价于 signed char，也可能等价于 unsigned char，这要由具体的 C 实现来决定。但是无论如何，上述 int 类型的常量值（字符编码）都能够用 char 类型的变量容纳，因为 ASCII 的编码值不会大于 127。我们已经学过整数类型－整数类型转换，在这里，可以从 int 类型转换为 char 类型，但转换后的值不变。

再来看数组 b 的声明，在该声明的初始化器里使用了十六进制整型常量。虽然形式不同，但只是数制的问题，本质上没有任何区别。

现在，我们将上述程序保存为源文件 c0506.c 并翻译为可执行文件，然后在 gdb 中调试一下，观察数组的内容。

```
(gdb) b 6
Breakpoint 1 at 0x4016bc: file c0506.c, line 6.
(gdb) r
Starting program: D:\examples\c0506.exe
[New Thread 3872.0x16c0]
```

```
Breakpoint 1, main () at c0506.c:6
6           }
(gdb) p /d a
$1 = {67, 104, 105, 110, 101, 115, 101}
(gdb) p /x a
$2 = {0x43, 0x68, 0x69, 0x6e, 0x65, 0x73, 0x65}
(gdb) p /c a
$3 = {67 'C', 104 'h', 105 'i', 110 'n', 101 'e', 115 's', 101 'e'}
(gdb) p /c b
$4 = {67 'C', 104 'h', 105 'i', 110 'n', 101 'e', 115 's', 101 'e'}
(gdb)
```

如上所示，我们将断点设置在第 6 行，也就是组成函数体的右花括号“}”所在的那一行，然后执行这个程序。当程序的执行停留在断点时，数组 a 和 b 都已经创建并完成初始化。

接着，我们使用带有参数的 p 命令打印数组的元素。虽然我们已经用过 p 命令，但还只是简单地使用。实际上，p 命令是可以带参数的，这些参数用于控制输出的格式。例如，“p/d”以有符号整数的形式打印输出；“p/x”以十六进制的形式打印输出；“p/c”以字符的形式打印输出。显然，前两个输出与数组声明时的初始化器一致；后两个输出不但给出了每个字符的编码值，同时也打印了字符本身。同时，我们还能看到数组 a 和 b 的内容完全相同。

5.2.2 字符常量

在数组声明的初始化器里使用整型常量有两个问题，一是不那么直观，二是可移植性不好。所谓可移植性，是指同一个 C 程序是否能在不做修改或者仅做少量修改就能在不同类型的计算机系统上执行并得到预期的结果。

我们说过，C 语言对所使用的字符集和编码方式不做任何限定，这就带来一个问题：同一个字符，比如“A”，不同的字符集会使用不同的编码。所以，一个整型常量对应的字符是什么，取决于 C 实现所采用的字符集和编码方式。尽管绝大多数计算机系统上的 C 实现都使用兼容于 ASCII 的字符集和编码方式，但为了程序的可移植性，最好不要直接使用字符编码，而是使用以下程序所示的字符常量。

```
/*********************c0507.c******************/
int main (void)
{
    char a [7] = {'C', 'h', 'i', 'n', 'e', 's', 'e'};
    char b [ ] = {'C', 'h', 'i', 'n', 'e', 's', 'e'};
    char c [9] = {'C', 'h', 'i', 'n', 'e', 's', 'e'};
}
```

在程序中，数组 a、b 和 c 的声明里都有初始化器，且这些初始化器都是由字符常量组成的。在 C 语言里，最简单的字符常量由一对单引号''，以及被它围起来的字符序列共同组成。在程序翻译期间，字符常量被转换为字符的编码值，但编码值取决于当前所使用的字符集和编码方式。

字符常量的类型并不是 char，而是 int，所以又称为“整型字符常量”，这可能出乎大多数人的意料。取决于当前所使用的字符集，字符常量'C'、'h'等都被映射为 int 类型

的字符编码值，并转换为 char 类型以初始化数组成员。

数组 a 有 7 个元素，和初始化器内的字符常量一一对应；数组 b 声明时未指定大小，但它的大小可以由初始化器里的字符常量的个数决定，故它的元素数量为 7；数组 c 有 9 个元素，但初始化器内的字符常量只有 7 个，前 7 个元素被初始化为对应的字符常量值，最后两个元素的值为 0。

5.2.3 脱转序列

现在问题来了，单引号“'”用于组成字符常量，但如果字符常量的字符就是单引号本身，怎么办呢？难道是'''吗？

可以明确地说，这样不行。在 C 中，解决这个问题的办法是使用脱转序列。脱转序列更经常地被称为转义序列，它用于使某些字符脱离原先的序列，改变它的含义和解释方法，并被转换为其他字符。

脱转序列以反斜杠“\”引导，后面跟着被转义的字符。所以，要想得到单引号的字符常量，你得用'\''。

然而，一旦引入了反斜杠，则反斜杠本身也难以自保了。因为这个原因，如果一个字符常量是要得到反斜杠本身，则必须使用两个反斜杠，即'\\'。

但这还不是引入脱转序列的主要目的，真正的原因是因为有些字符属于显示不出来的非图形字符，比如换行符、回车符、警示符（遇到这个字符时，电传设备会鸣叫一声以引起注意）、制表符（使字车或者显示器的光标移动若干个字符的位置，以达到文本对齐的效果），等等。这些字符既看不见也没办法通过键盘输入，要想构造它们的字符常量，只能通过脱转序列用别的字符代替。例如，'\a'表示响铃符；'\r'表示回车符；'\n'表示换行符；'\t'表示水平制表符。

你可能会说，既然是非图形字符，直接用字符编码得了，为什么非要构造字符常量。是的，你当然可以直接使用字符的代码，但这样做会失去可移植性；另一方面，既然引入了字符常量这个东西，那么，从语法功能上来讲，它也必须实现非图形字符的常量形式。

然而，不是所有非图形字符都那么幸运，可以拥有斜杠加字母的替代形式。为此，可以使用八进制脱转序列或者十六进制脱转序列。如果字符常量的单引号里是反斜杠和数字，则它是一个八进制脱转序列（所以在这里必须使用八进制数字），例如'\107'；如果是一个由反斜杠、字符 x 和数字（必须使用十六进制）组成的序列，则它是一个十六进制脱转序列，例如'\x89'。

在下面的程序中，数组 a 的初始化器里使用了由各种脱转序列组成的字符常量。其中'\''表示一个单引号；'\n'表示一个换行符；假定我们当前使用的字符集是 ASCII，则字符常量'\101'用的是八进制脱转序列，表示字符“A”；'\x41'用的是十六进制脱转序列，也表示字符“A”；65 是整型常量而不是字符常量，但它是字符“A”的编码；'\\'表示单个的反斜杠字符“\”。

```
/********************c0508.c*******************/
int main (void)
{
    char a [] = {'\'', '\n', '\101', '\x41', 65, '\\'};
    char b [] = {0, '\0', '0'};
}
```

注意，在八进制脱转序列中最多允许 3 个八进制数字，例如'\9'、'\09'、'\009'都被解释为只包含了单个字符的字符常量。

在数组 b 的声明里，初始化器 0 和'\0'是相同的。在 C 语言里，编码值为 0 的字符叫

作空字符。在指定空字符时，可以直接使用整型常量 0，或者用字符常量'\0'来表示。这里的\0 是八进制脱转序列，代表那个编码值为 0 的字符。

注意，字符常量'\0'和'0'是不同的，前者是空字符常量，代表空字符，其字符编码为 0；后者是数字字符 0，其 ASCII 编码为 48。

练习 5.2：

上机观察字符常量'\''、'\n'和'\\'的编码值是多少。字符常量'\101'和'\x41'是同一个字符吗？

5.2.4 字面串和字符串

使用整型常量初始化一个字符数组既缺乏可移植性，又比较麻烦；使用字符常量初始化字符数组呢，可移植性较好，但同样有些麻烦，毕竟还得一个一个地罗列。如果想既省事，可移植性又好，那就只有一个办法：使用字面串。

所谓“串”，是一个连续的字符序列，以遇到的第一个空字符终止，且这个空字符也是串的组成部分，是串的最后一个字符。因为是字符的序列，所以又叫字符串。

字符串以空字符结束是有原因的。字符串在日常的编程工作中被大量使用，比如在屏幕上显示字符串、在字符串里查找指定的字符、将两个字符串合并，比较两个字符串是否相同，等等。如果没有空字符，那么，在操作字符串的时候，我们需要记住它的长度以免越界。有了末尾的空字符，就知道是否到了字符串的末尾，空字符就是一个最好的标志和约定。

只要稍微发挥一下想象，就会发现数组很适合用来容纳字符串。例如，对于以下两个数组的声明：

```
char a = {'s', 'p', 'e', 'e', 'd', '\0'};
char b = {'H', 'e', 'l', 'l', 'o', '\0', 'B', 'o', 'b', '\0'};
```

则如图 5-4 所示，假定这是程序运行时计算机内存储器中的部分内容，数组 a 中包含了一个字符串“speed”，而数组 b 则包含了两个字符串，分别是“Hello”和“Bob”。

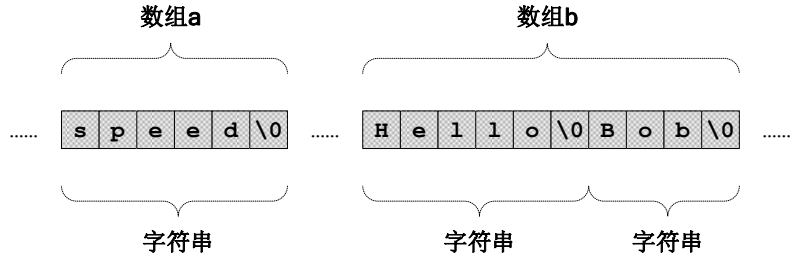


Fig.5-4 包含字符串的数组

除此之外，在 C 语言里，有一种特殊的表达式，它们的类型是数组，而且是字符类型的数组。典型地，这种表达式由一对双引号""，以及位于双引号之间的一串字符组成，例如"hello,world"，这称为字面串。

在程序翻译期间，这种表达式被用于创建一个不可见的、元素类型为 char 的数组，数组的内容除了双引号内的字符外，还会自动在末尾添加一个空字符。在本书的第二章里我们讲了变量的生存期，不管字面串出现在程序中的什么地方，哪怕是在函数内，由它创建的数组具有很长的生存期，从程序启动时开始，到程序退出时终止。但是，因为它没有名字，所以无法直接使用，只能通过该字面串本身来引用。

字面串在源文件里被用作数组类型的表达式，而且是一个左值，指示或者说代表着那个在程序翻译期间所创建的、不可见的字符数组。当然，这个字符数组的内容通常是一个字符串。从它的字面形式上来看，也确实很像一个字符串，所以干脆称它为字面串。当然，另一个原因是它还只是“能够”得到一个字符串，毕竟在程序设计时，这个字符串还不存在，只

具有字面上的形式。

我们知道，如果运算符 sizeof 的操作数是数组类型的，则它返回数组的大小。如下面的程序所示，变量 siz 的声明中带有初始化器 sizeof "speed"。因为字面串"speed"是一个数组类型的左值，具体类型为 char [6]（注意，数组的大小包括末尾的空字符），故这个 sizeof 表达式的结果是 6。

```
/***************c0509.c*************/
int main (void)
{
    int siz = sizeof "speed";

    char a [] = "speed";
    char b [3] = "speed";
    char c [7] = "speed";
    char d [] = "speed\0is\0delphi\0";
    char e [] = "";
    char f [] = "hello" " " "ma'am.";
}
```

字面串是无须声明的数组，或者说，它在程序中的出现就相当于声明。相较于普通的数组，它的特别之处是可直接用于初始化另一个数组变量，这样我们就可以避免使用烦人的整型常量和字符常量。比如在当前程序中，数组 a 就是用字面串"speed"初始化的，该声明等同于

```
char a [] = {'s', 'p', 'e', 'e', 'd', '\0'};
```

如果在声明一个数组时，指定的大小恰好可以容纳字面串的全部内容（包括末尾的空字符），那就再好不过了。但是，如果数组的大小不足以容纳字面串的全部内容，则只能用字面串的前面一部分来初始化；如果数组的大小超过了字面串的长度，则超过的部分自动被初始化为 0。这就是说，数组 b 和 c 的声明等同于

```
char b [3] = {'s', 'p', 'e'};
char c [7] = {'s', 'p', 'e', 'e', 'd', '\0', 0};
```

和字符常量一样，字面串里也可以使用脱转序列，包括八进制脱转序列和十六进制脱转序列。数组 d 的初始化器是字面串"speed\0is\0delphi\0"，里面夹杂了 3 个手动给出的空字符，但实际上还有一个隐藏在末尾的空字符（因为字面串本身隐藏了一个空字符）。

如图 5-5 所示，初始化之后的数组 d 里有 4 个字符串，前 3 个字符串分别是“speed”“is”和“delphi”，最后一个字符串里没有图形字符，只有一个空字符。为方便起见，我们把仅包含空字符的字符串称为空字符串，简称空串。

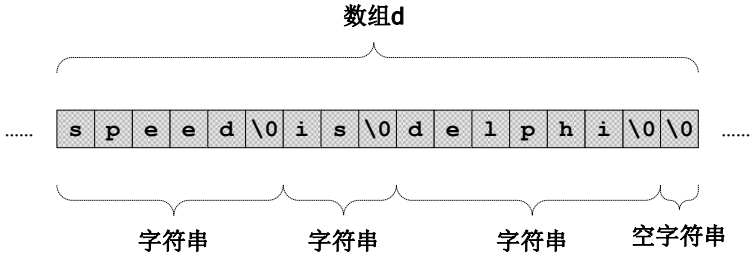


Fig.5-5 含有多个字符串的数组

不包含任何字符的字面串""用于得到一个空串。在程序翻译期间，这将创建一个仅包含空字符的数组。因此，数组 e 的声明实际上等同于

```
char e [] = {'\0'};
```

在 C 语言里，互相邻接的多个字面串将会合并为一个完整的字面串。在数组 f 的声明中，初始化器"hello" " " "ma'am."是由字面串"hello"、" "和"ma'am."按顺序邻接而成，C 实现在翻译期间将它们合并为一个独立的字面串"hello ma'am."。此后，这个合并的字面串用于创建一个不可见的数组，并用它的内容来初始化数组 f。

字面串到底是什么东西？对此问题的解释也从一个侧面关乎 C 语言的设计哲学。C 是讲求实用的语言，为了方便，它可以不讲章法，信手拈来，运算符 sizeof 就是一个例子，而字面串则是又一个典型。给定以下声明：

```
int m = 65535, * p = & m;
char a [] = "hello";
```

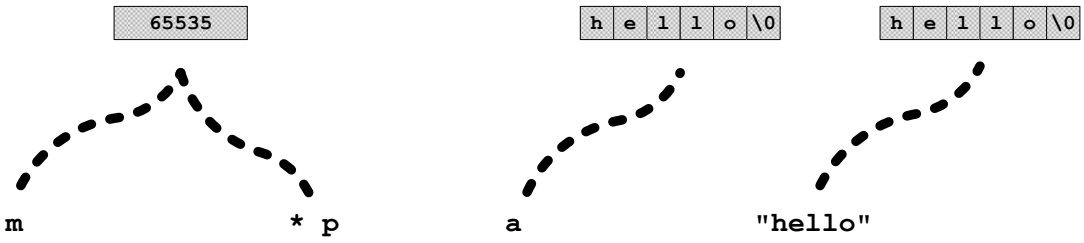


Fig.5-6 字面串是一个指示变量的左值

显然，在声明中，标识符 m 代表那个存储值为 65535 的变量；在表达式里，m 是个左值，代表那个存储值为 65535 的变量；表达式* p 求值的结果是一个左值，它指示那个存储值为 65535 的变量，甚至我们通常说表达式* p 本身在整体上是一个左值，这样可以省略求值过程。同样地，表达式 a 也是一个左值，代表它背后的那个数组。

现在重点来了，和 m、* p、a 一样，字面串"hello"也是一个左值，代表那个用它创建的隐藏数组，只是我们还不习惯。因为字面串是一个数组类型的左值，所以，**但凡是数组类型的表达式可以出现的地方，字面串也能出现**，但字面串还多了一个功能：可以在声明中初始化一个数组变量。

5.3 访问数组元素

在接触数组之前，我们一直与整数类型的变量打交道。这种类型的变量有一个特点，就是可以将一个变量的值保存到另一个变量。在下面的程序中，表达式 a = b 会把变量 b 的值保存到变量 a，然后这两个变量就具有相同的值了。

```
/*************c0510.c*************/
int main (void)
{
    int a, b = 8;
    a = b;

    int c [2], d [2] = {2200, 2300};
    c = d;

    char s [20];
    s = "speed";
}
```

一个数组类型的变量，其内容被解释为那种数组类型的值。按照常理，我们也可以用赋

值表达式把一个数组变量的值保存到另一个数组变量。

然而这是不可以的。原因很简单，C 语言的发明者不是这样设计的，他不想这样做。他所做的，是把**数组类型的左值规定为不允许修改，也不发生左值转换**。

我们知道左值转换，也知道，当一个左值是赋值运算符的左操作数，或者是++、--、一元&及 sizeof 等运算符的操作数时，不发生左值转换。现在，又多了一种限制，那就是该左值不能是数组类型。

所以，在这个示例程序中，表达式 c = d 和 s = "speed"是非法的，c 和 s 都是数组类型的左值，按规定不允许修改，所以不能放在赋值运算符的左侧；表达式 d 和字面串"speed"是数组类型的左值，但不能转换为值。不能用一个数组给另一个数组赋值，也不能用一个数组初始化另一个数组，唯一的例外是可以用字面串初始化一个字符数组。当然，他这样设计还有一个更重要的原因，那就是要在数组和指针之间建立起一种非同寻常的关系。

练习 5.3：

将上述程序保存为源文件 c0510.c，然后尝试翻译它，看看翻译软件会给出什么错误提示信息。

5.3.1 数组—指针转换

那么，如何将一个数组的值赋给另一个数组呢？通常的做法是在元素之间赋值，比如将数组 A 的元素的值赋给数组 B 的对应元素。问题在于，如何访问数组元素呢？指针可能是绕不开的途径——在 C 语言里，数组和指针是截然不同的两种东西，但是，在大多数情况下，数组类型的表达式会转换为指针。

C 语言规定，除非作为 sizeof 和一元&运算符的操作数，或者是一个字面串且被用于初始化数组变量，否则，一个数组类型的表达式会被转换为指向该数组首元素的指针，而且不再是一个左值。如果数组的（元素）类型为 *T*，则转换后的类型为指向 *T* 的指针。

这是可以理解的，比如，要是运算符 sizeof 的操作数是数组类型，而它又转换成了指针，返回的就是指针的大小了。在下面的程序里，变量 m 的类型是 int，它的初始化器是表达式* a，它用于初始化变量 m 的过程为：

首先，因为表达式 a 的类型是数组，数组元素的类型为 int，所以执行数组—指针转换，转换为指向 int 的指针，而且指向数组 a 的第 1 个元素（下标为 0 的元素）；

进一步地，因为一元运算符*的操作数是指针，故该运算符的结果是一个左值，代表一个变量。实际上，这个变量就是数组 a 的首元素；

最后，既然表达式* a 是左值，就要执行左值转换。既然它代表数组 a 的首元素，那么自然要转换为数组 a 首元素的值，也就是 5，并用来初始化变量 m。

```
/***********c0511.c**********/
int main (void)
{
    int a [20] = {5}, m = * a;
    * a = 6;

    char c = * "hello world.";
    * "good" = 'G';
}
```

同理，在表达式* a = 6 中，子表达式* a 是一个左值，代表数组 a 的首元素。作为运算符=的左操作数，它不执行左值转换，而是接受赋值。赋值后，数组 a 的首元素不再是 5，而是 6 了。

变量 c 的声明中带有初始化器* "hello world."，在这个表达式中，字面串"hello world."是数组类型的左值，被转换为指针，指向那个隐藏数组的首元素。

因为一元运算符*的操作数是指向 char 的指针，所以表达式* "hello world."是一个 char 类型的左值，代表那个隐藏数组的首元素，该元素的值是字符 h 的编码，并用于初始化变量 c。

尽管字面串会创建一个数组，且该数组的生存期贯穿整个程序的运行过程，但在程序运行时，这个隐藏的数组可能位于一个受处理器和操作系统保护的内存区域，只能读取而不能写入。在这种情况下，你可以读取这个数组的内容，但不能修改它。一旦你试图去修改数组的元素，则程序的行为是未定义的，处理器和操作系统将会加以阻止，并可能使程序崩溃。这就是说，语句

```
* "good" = 'G';
```

本身是合法的，字面串"good"被转换为指针，一元运算符*得到一个左值。这句的本意是将数组的首字符“g”修改为大写的“G”，但这种修改行为是未定义的，其后果不可预料。

练习 5.4：

1. 将上述程序保存为源文件 c0511.c，然后尝试翻译它，看看翻译软件会给出什么错误提示信息。
2. 在表达式* "good" = 'G'中，字符常量'G'的类型是什么？子表达式"good"的类型是什么？子表达式* "good"的类型是什么？在该表达式求值的过程中，都发生了哪些类型转换？

5.3.2 指针运算和 for 语句

既然数组类型的表达式可以转换为指向数组首元素的指针，那么我们很容易想到，只要移动这个指针，令它依次指向数组的其他元素，不就可以访问数组的所有元素了吗？

正是这样。C 语言规定，假定一个指针 P 指向数组的第 M 个元素，且数组足够大，N 是一个整数，那么 P + N 的结果是一个新的指针，指向该数组的第 M + N 个元素；P - N 的结果也是一个指针，指向该数组的第 M - N 个元素。

在下面的程序示例中，我们演示了指针的加法运算。首先，我们声明了一个数组 a，从初始化器来看，它有 12 个元素，其中，下标为 0 的那个元素的值为 5；下标为 7 的那个元素的值是 7；下标为 11 的那个元素的值是 8，其他元素一律初始化为 0。

数组 b 的大小和数组 a 相同，因为它的大小是用表达式 sizeof a / sizeof (int) 指定的，意思是用数组 a 的大小（以字节计）除以数组元素的大小（以字节计），从而得到数组 a 的元素数量。表达式 sizeof a 和表达式 sizeof (int)的结果在程序翻译的时候就能得到，所以表达式 sizeof a / sizeof (int)的结果是一个程序翻译期间就确定的整型常量，而该表达式是常量表达式。

```
/*************c0512.c**************/
int main (void)
{
    int a [] = {5, [7] = 7, [11] = 8}, b [sizeof a / sizeof (int)];

    for (int x = 0; x < sizeof a / sizeof (int); x ++)
        * (b + x) = * (a + x);

    char c = * ("Are you sure?" + 2);
```

}

接下来的代码是将数组 a 的内容逐元素地复制给数组 b，复制之后，这两个数组所对应的元素具有相同的值。复制操作由一个我们没学过的语句完成，这就是 for 语句了。

for 语句由关键字“for”引导，后面是一对圆括号。在圆括号里，是由分号“;”隔开的三个部分，而且这三个部分都可以省略。和 while 语句一样，for 语句也是循环语句，所以圆括号后面的“语句”也是循环体。

for (*声明可选* ; *表达式可选* ; *表达式可选*) *语句*

for (*表达式可选* ; *表达式可选* ; *表达式可选*) *语句*

for 语句有两种形式，这两种形式的差别仅在于圆括号内的第一部分，一个是声明，一个是表达式。为了便于说明，我们将 for 语句简化为如下更直观的形式：

for (decl 或者 e1; e2; e3) 语句

其中，decl 代表第一种形式里的声明；e1、e2 和 e3 对应于那三个表达式。当然，它们都可以省略。

for 语句的执行过程是这样的，如图 5-7 所示，先处理第一部分，也就是 decl 或者 e1，而且在 for 语句的整个执行过程中只处理一次。如果这一部分是 decl，则处理这个声明；如果是 e1，则求值该表达式。如果没有这一部分，则直接求值 e2。

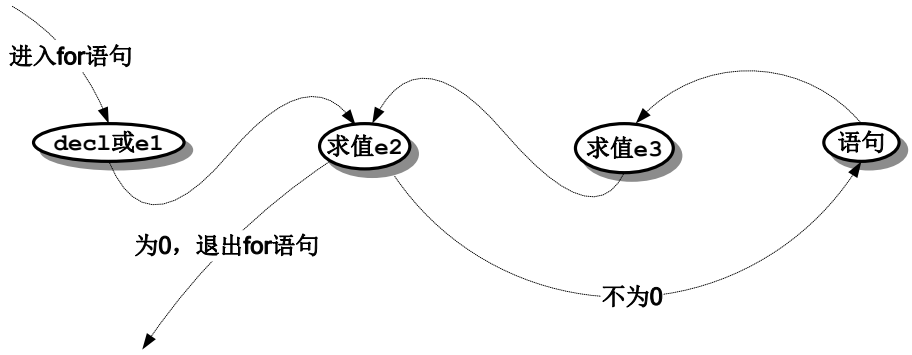


Fig.5-7 for 语句的执行流程示意

e2 是 for 语句的控制表达式，如果 e2 求值的结果不为 0，则执行圆括号后面的语句，也就是循环体；如果为 0，则退出 for 语句。如果 e2 不存在，则在翻译程序时，翻译器会自动插入一个不为 0 的常量。在这种情况下，e2 求值的结果将始终不会为 0，这将产生一个无限循环，也就是所谓的死循环。

执行完循环体后，紧接着求值 e3，然后再次求值 e2。如果没有 e3，则执行完循环体之后直接再次求值 e2。无论如何，都将再次根据 e2 求值的结果决定是执行循环体呢，还是退出 for 语句。

现在回到程序中，来看 for 语句。在圆括号内的第一部分声明了变量 x 并初始化为 0，这一部分仅在进入 for 语句时执行 1 次。

第二部分是一个关系表达式，运算符 sizeof 的优先级最高，/次之，<最低，所以该表达式等价于 x < (sizeof a / sizeof (int))。这是 for 语句的控制表达式，sizeof a / sizeof (int)用于得到数组 a 的元素数量，显然，整个控制表达式的意思是如果变量 x 的值小于数组 a 的元素个数则持续执行 for 语句。

第三部分很简单，表达式 x ++用于在每次执行完循环体后递增变量 x 的值。这样就清楚了：这个 for 语句先声明了变量 x 并令它的初值为 0，接着，判断它的值是否小于数组 a 的元素数量，条件成立就执行循环体做事。每次执行了循环体之后，递增变量 x 的值并继续判断条件，直至条件不成立，离开 for 语句。

再来看循环体，它只有一条语句

```
* (b + x) = * (a + x);
```

在这里，数组类型的表达式 a 和 b 都符合转换为指针的条件。表达式 b 转换为指向数组 b 首元素的指针，其类型为 int *。左值 x 执行左值转换，其结果为一个整数。整数与指针相加，其结果是一个新的指针。因为原指针是指向数组 b 的第 1 个元素，故新指针指向数组 b 的第 x+1 个元素，或者说下标为 x 的元素。

然后，一元*运算符作用于这个新的指针，得到一个左值，指示或者说代表那个下标为 x 的元素。因为它是赋值运算符=的左操作数，故不执行左值转换，而是接受赋值。

同理，表达式* (a + x)是一个左值，代表数组 a 的第 x+1 个元素。然后，经左值转换后，得到那个元素的值，并赋给数组 b 的对应元素。

for 语句的循环体可以是任何语句，包括另一个 for 语句。如果 for 语句的循环体包含多条语句，则必须用花括号构造一个复合语句。

在程序的最后，我们声明了一个 char 类型的变量 c，并将其初始化为表达式* ("Are you sure?" + 2)的值。在这里，字面串"Are you sure?"是数组类型的表达式，不但用于创建一个隐藏的数组，而且自动转换为指向 char 的指针，指向编码值为“A”的那个元素。将这个指针加 2，将得到一个新的指针，指向编码值为“e”的元素。

无论如何，表达式* ("Are you sure?" + 2)是一个左值，执行左值转换后得到的值是字符“e”的编码，并赋给变量 c。

显然，将一个指针和一个整数相加减，不是整数之间相加减那么简单。为了更清楚地显示这种加减法的原理，我们将上述程序保存为源文件并翻译为可执行程序，然后在 gdb 中加以调试，其过程如下：

```
(gdb) b 10
Breakpoint 1 at 0x4016ac: file c0512.c, line 10.
(gdb) r
Starting program: D:\examples\c0512.exe
[New Thread 3804.0x8e8]

Breakpoint 1, main () at c0512.c:10
10          }
(gdb) p a
$1 = {5, 0, 0, 0, 0, 0, 0, 7, 0, 0, 0, 8}
(gdb) p b
$2 = {5, 0, 0, 0, 0, 0, 0, 7, 0, 0, 0, 8}
(gdb) p c
$3 = 101 'e'
(gdb) p a + 0
$4 = (int *) 0x22fe88
(gdb) p a + 1
$5 = (int *) 0x22fe8c
(gdb)
```

以上我们先是打印了数组 a 和 b 的内容，显然，经过一对一复制之后，两个数组的内容完全一致。然后我们打印了变量 c 的值，结果是十进制数 101。在 C 语言和 gdb 里，字符类型是被特殊对待的，gdb 了解到变量 c 的类型是 char，所以很贴心地给出了该编码所

对应的字符“e”。

现在，我们来看看指针和一个整数相加之后的值有什么特点。尽管数组 a 可以自动转换为指向其首元素的指针，但你不能在 gdb 命令行使用“p a”来打印这个指针值，这个命令会打印出数组 a 的内容，毕竟这不是在写程序。

不过，要是我们使用命令“p a + 0”，效果就不同了。gdb 知道这是要将数组 a 转换为指针，然后同整数 0 相加，而且结果同样是指针。实际上，将一个指针和 0 相加，并不会移动指针，这只是一个策略和技巧。

结合上述调试过程和图 5-8 可知，表达式 a + 0 的结果是一个指针，它指向地址为 0x22FE88 的内存位置，所以调试器的输出为：

```
$4 = (int *) 0x22fe88
```

显然，这很像一个转型表达式，但实际上就应该这么理解。即，所打印的结果是一个指针，由整数 0x22fe88 转换而来。

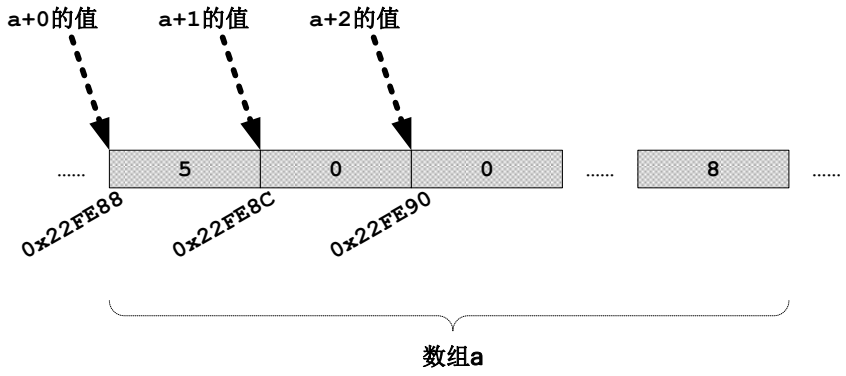


Fig.5-8 指针和整数相加的本质

对比一下命令“p a + 0”和“p a + 1”的打印结果你会发现，将一个指针类型的值加 1，其结果在数值上差 4，而不是差 1，这就与整数加减法不同了。这并不难理解，对于指向类型 T 的指针（值），为它加上整数 1，将得到一个新的指针，新指针在数值上比原指针大 sizeof (T)。

练习 5.5：

将上述程序中的 for 语句改为 while 语句，以实现相同的功能。

5.3.3 下标运算符

本质上，指针是访问数组元素的唯一方法。但就语法形式而言，用一个指针和一个整数相加来指向数组元素，这似乎不够优雅和简洁。再说，让手指和键盘受累，C 语言的发明者也会良心不安。

C 语言的特点之一是变量的声明和对该变量的操作具有形式上的一致性。例如，我们用星号“*”声明一个指针，然后，这个星号又可以出现在表达式里，作为运算符来作用于一个指针，得到那个被指向的实体。

相似地，既然我们用一对中括号“[]”来声明数组，那么，这对中括号也将以运算符的身份出现在表达式里，用于通过下标的形式来指定数组的某个元素。

下标运算符[]用于组成一个代表数组元素的表达式，形如 e1 [e2]。e1 和 e2 是运算符[]的操作数。

在下面的程序中，首先声明了数组 a 和变量 b。然后，表达式 a [0] = 1 是将数值 1 赋给数组下标为 0 的元素；表达式 a [1] = b 是将变量 b 的值赋给数组 a 下标为 1 的元素。在这里，b 要先进行左值转换，以得到它的存储值。

就直觉而言，运算符[]的操作数是数组和数组的下标。这种认识是合理的。不过，由于数组会被转换为指向其首元素的指针，所以真实的情况是，**运算符[]的两个操作数一个是指针类型，另一个是整数类型，而且表达式**e1 [e2]**等同于*** (e1 + e2)。

这就意味着，表达式a [0] = 1等同于* (a + 0) = 1而表达式a [1] = b则等同于* (a + 1) = b。

对于运算符[]的两个操作数，C语言并未规定哪一个是指针，哪一个是整数，因此，才有了程序中那个奇怪的表达式 2 [a] = 1 [a]，这实际上就是 a [2] = a [1]。显然这是将数组a的第2个（下标为1的）元素的值赋给第3个（下标为2的）元素。

```
/*******c0513.c*******/
int main (void)
{
    int a [3], b = 2;

    a [0] = 1;
    a [1] = b;
    2 [a] = 1 [a];

    char c = "Tom." [2];
}
```

在变量c的声明中，初始化器是表达式"Tom." [2]，它等同于* ("Tom." + 2)，是一个左值，代表着那个隐藏数组的第3个元素。表达式* ("Tom." + 2)是个左值，要进行左值转换，得到字符“m”的编码并用于初始化变量c。

练习5.6：

给定声明：

```
int a [5] = {37, 1, 62, 58, 33}, b [7];
```

编写程序将数组a的内容复制到数组b，要求使用下标运算符。

5.3.4 指针的递增和递减

尽管用指针访问数组和用下标运算符访问数组等效，但在程序设计的复杂性方面却各有千秋。有时候，用指针较为方便，而有的时候，用下标运算符更清晰更简洁。在本节，我们通过示例来做一下对比。来看下面的程序，它的任务是将两个字符串连接到一起形成一个新的字符串。

```
/*************c0514.c************/
char * s_joint (char * d, char * s)
{
    char * r = d;

    while (* d != '\0') d ++;
    while ((* d ++ = * s ++) != '\0') ;

    return r;
}

int main (void)
```

```
{
    char a [10] = "Hi,", * ps = 0;

    ps = s_joint (a, "Tom.");
}
```

为了获得通用性，我们编写了一个函数 s_joint 来完成字符串的连接工作。该函数接受两个参数，参数 d 和 s 都是指针类型的变量，它们的值各自指向一个字符串，我们要把变量 s 的值所指向的字符串附加到变量 d 的值所指向的字符串尾部。

函数 s_joint 的返回类型是指向 char 的指针，连接工作完成后，这个返回值指向连接后的新串。实际上，这个返回值就是传递给参数 d 的值，故函数 s_joint 所完成的第一个工作就是先保存参数 d 的值以便返回它。

再来看 main 函数，我们先是声明了一个字符类型的数组 a，它有 10 个元素，是用字面串"Hi,"初始化的；同时，我们还声明了一个变量 ps，其类型为指向 char 的指针，用于接受函数 s_joint 的返回值，它被初始化为空指针。

接下来，我们调用了函数 s_joint 并把返回值赋给左值 ps。调用时，实际参数是表达式 a 和字面串"Tom."，它们都是数组类型的左值，将转换为指向数组首元素的指针（一个指向数组 a 的首元素，一个指向隐藏数组的首元素）。执行函数调用时，这两个指针分别传递给参数变量 d 和 s。

如图 5-9 所示，在程序运行时将至少创建 2 个数组，一个是数组 a，另一个是由字符串"Tom."创建的隐藏数组。你可能会问，字面串“Hi,”不也要创建一个隐藏的数组吗？是的，按照 C 语言的要求，应当是这样的，从本章的开头到现在，我们一直是这样告诉大家的。

问题在于 C 实现，为了提高程序的运行效率，它可能会做一些投机取巧的事。如果它发现一个字面串很短小，而且除了用于初始化一个数组之外，再没有别的用处，那么它可能不会创建一个隐藏的数组。在这里，字面串“Hi,”很短，而且只用来初始化数组 a，那么它将用字符“H”“i”“,”和空字符的编码值直接初始化数组 a。

然而，如果字面串很长，或者在该字面串需要转换为指针的场合，C 实现没有别的选择，只能老老实实地创建一个隐藏的数组。这是因为，如果字面串很长，即使它只用于初始化一个数组，笨办法的开销也会变大，C 实现唯一的选择就是使用处理器的批量传送指令，在两个数组之间进行复制以完成初始化操作。无论字面串有多长，使用批量传送只需要 3 到 5 条机器指令，而一个字符一个字符地赋值则非常麻烦和臃肿。

另一方面，像"Tom." [2]和 s_joint (a, "Tom.")这样的表达式，字面串需要转换为指针，并通过指针访问它背后隐藏的数组，C 实现也只能无条件地用"Tom."创建隐藏的数组。

不过，我们的原则是不考虑 C 实现的因素。所以，图 5-9 依然画出了字面串"Hi,"所创建的那个隐藏数组。

让我们继续回到函数 s_joint 中，字符串连接的基本思路是这样的：因为“Hi,”和“Tom.”是分别位于两个数组中的字符串，而变量 d 和 s 的值虽然指向数组首元素，但实际上指向的是这两个字符串的首字符位置。所以，可先使变量 d 指向第一个字符串的末尾，也就是零终止符所在的位置，然后从这个位置开始，通过指针操作将另一个数组里的字符串复制过来。

现在来看第一个 while 语句，它的任务是移动指针所指向的位置。等性运算符!=的优先级低于间接运算符*，故这个 while 语句的控制表达式等同于(* d) != '\0'。

这个 while 语句的执行过程是这样的：左值 d 的类型是指针，左值转换后得到一个指

针类型的值（指向数组 a 的首元素）。因此，表达式* d 是一个左值（指示或者说代表数组 a 的首元素），左值转换为 char 类型的值并与字符常量'\0'进行比较。如果相等，则表达式* d != '\0'的值为 1，执行循环体 d ++，使对象 d 的值指向下一个数组元素；如果不等，则控制表达式的值为 0，退出循环语句。

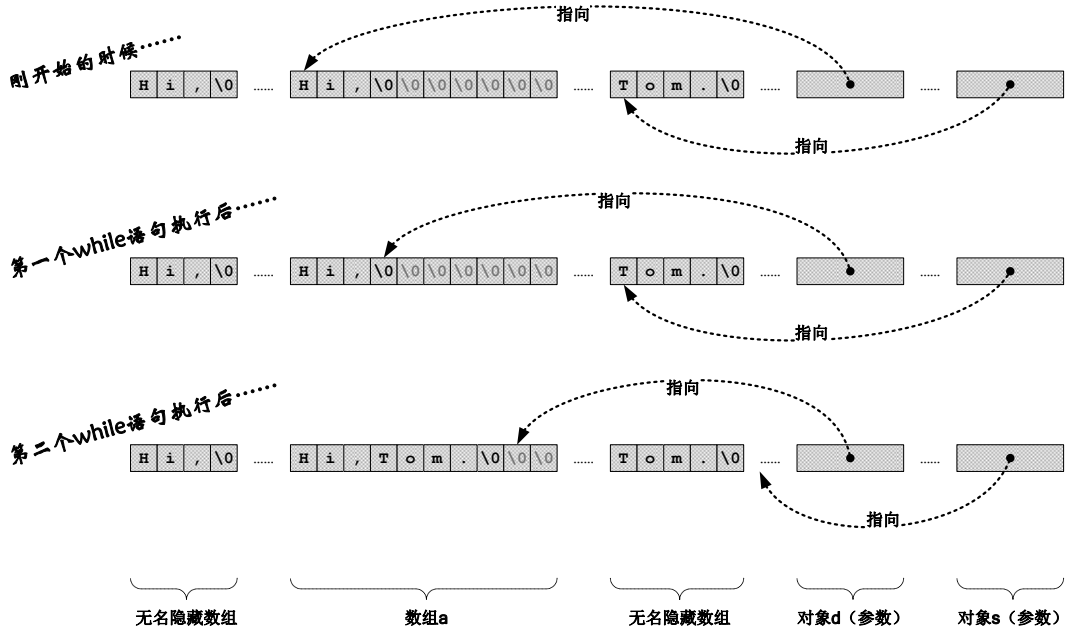


图 5-9 字符串连接过程示意

这是我们第一次看到运算符++也适用于指针类型的左值。我们知道，如果一个指针所指向的类型是 T，将它和一个整数相加减，实际上是将这个指针在数值上加（减）类型 T 的大小，即，在数值上加上或者减去 sizeof (T)。

同样地，运算符++和--也允许指针类型的操作数，但必须是左值，代表一个指针类型的变量。不管是前缀形式还是后缀形式，如果操作数的类型是指针，则这些运算符的结果依然是指针，但不再是左值。

如果 P 是一个左值，其类型为指向 T 的指针，则表达式 P ++的值是 P 递增之前的原始（指针）值；表达式 P --的值是 P 递减前的原始（指针）值；表达式++ P 的值是 P 递增后的（指针）值；表达式-- P 的值是 P 递减后的（指针）值。它们都有一个副作用，即递增或者递减 P 的存储值，使之在数值上比原来增加或者减小 sizeof (T)。如果 P 指向一个数组元素，则递增或者递减之后，P 的值指向下一个或者前一个数组元素（假定递增或者递减后的值依然指向数组内的合法元素）。

无论如何，如图中所示，当第一个 while 语句执行完毕后，变量 d 的值指向数组 a 内的字符串的末尾，即，第一个值为 0 的元素。

第二个 while 语句很复杂，但要想使它保持简洁，这差不多是唯一的写法。我们先来看一下控制表达式(* d ++ = * s ++) != '\0'的组成和作用，总体上来说，这个表达式既完成赋值工作，又完成等性判断工作，在这个过程中还有递增变量 s 和 d 的存储值的副作用。

这里涉及以下几种运算符：间接运算符*、后缀递增运算符++、赋值运算符=和等性运算符!=，它们的优先级从低到高依次是=、!=、*和++。圆括号的作用是形成一个基本表达式（括住的表达式）以打破运算符与操作数的自然结合，因此，这是一个等性表达式，等性运算符!=的操作数是子表达式(* d ++ = * s ++)的值和字符常量'\0'。

要得到整个控制表达式的值，也就是等性运算符!=的结果，需要先计算其操作数'\0'和子表达式(* d ++ = * s ++)的值。

在圆括号内部是一个赋值表达式，即，它等价于* (d ++) = * (s ++)。表达式 s ++的值是变量 s 递增前的原值，这是一个指针，指向一个数组元素；一元*运算符作用于这个指针，得到一个左值，再经左值转换，得到那个元素的值。同理，表达式* d ++是一个左值，用来接受赋值。

注意，s ++和 d ++是有副作用的表达式，会递增变量 s 和 d 的存储值，使它们各自指向数组的下一个元素。但是，在整个控制表达式求值完成之前，这两个副作用发生的时间并不确定，但这没有什么关系。

赋值表达式(* d ++ = * s ++)也要计算出一个值（我们知道，赋值表达式的值是赋值运算符的左操作数被赋值之后的值），它就是刚才赋给左值* d ++的字符编码。这个编码值和字符常量'\0'进行比较，两者相等则运算符!=的结果为 0，退出 while 语句；两者不等则运算符!=的结果为 1，执行循环体。

在这里，while 语句的循环体是空语句，因为所有的工作都让控制表达式做了，它确实没什么可做的。

总体来说，第二个 while 语句的作用是将变量 s 的当前值所指向的字符取出并保存到变量 d 的当前值所指向的位置，如果取出并保存的字符是空字符则退出 while 循环。与此同时还将递增变量 s 和 d 的存储值。

现在，让我们在 gdb 中调试上述程序，并观察程序的运行情况。将断点设置在第 16 行，也就是执行函数调用的那一行，然后运行程序到断点。

```
(gdb) b 16
Breakpoint 1 at 0x40169d: file c0514.c, line 16.
(gdb) r
Starting program: D:\examples\c0514.exe
[New Thread 2628.0xa48]

Breakpoint 1, main () at c0514.c:16
16          ps = s_joint (a, "Tom.");
(gdb) p a
$1 = "Hi,\000\000\000\000\000\000"
(gdb) p ps
$2 = 0x0
(gdb) n
17          }
(gdb) p a
$3 = "Hi,Tom.\000\000"
(gdb) p ps
$4 = 0x22feb2 "Hi,Tom."
(gdb)
```

如以上调试过程所示，我们先是打印数组 a 的内容。和从前不同，这次我们用的是无参数的“p a”命令。如果你数一数，打印的内容

```
$1 = "Hi,\000\000\000\000\000\000"
```

里只有 9 个字符，并不等于数组的元素总数。原因是，数组 a 里包含了字符串（含串尾的空字符）和剩余的 6 个元素。gdb 的做法是先打印字符串，再打印其他元素。打印字

符串时省略串尾的空字符，所以看上去少了一个元素。

命令“p ps”显示了变量 ps 的值。在程序里它被初始化为空指针常量，所以这里就显示为 0。

当我们用“n”命令执行函数调用之后，再来显示数组 a 的内容和变量 ps 的值时，结果变了。显然，数组 a 的内容表明两个字符串连接成功，而变量 ps 的值是 0x22feb2（在你的电脑上可能是别的地址）。因为这个值的类型是指向 char 的指针，gdb 很热情地为你提供了额外的信息和服务，显示了该指针所指向的字符串。

调用 s_joint 函数时需要注意，参数变量 d 的值所指向的字符串，其所在的数组必须有足够的空间以容纳被附加的字符串，而且这个数组不能位于一个受处理器和操作系统写保护的内存区域；也不能和变量 s 的值所指向的字符串在存储空间上重叠或者部分重叠。违反上述任何一条，程序的行为是未定义的，后果不可预料。

练习 5.7:

1. 关于表达式(* d ++ = * s ++) != '\0'，判断以下说法是否正确，并说明原因。

a.先得到运算符++的结果，再计算一元*运算符的值。()

b.先得到子表达式* d ++和* s ++的值，再计算运算符=的值。()

c.先计算表达式(* d ++ = * s ++)和'\0'，再计算运算符!=的值。()

d.运算符++的副作用和它的值计算同时发生。()

2.如果将源文件中的声明

```
char a [10] = "Hi,", * ps = 0;
```

改为

```
char * a = "Hi,", * ps = 0;
```

则程序运行时会怎样？为什么？

3.编写一个函数以比较两个字符串是否相同（包括末尾的空字符）。

使用指针操作来连接两个字符串当然是极简洁、极方便的，但如果使用数组下标操作会怎样呢？让我们来看一看。不过我们仅仅修改函数 s_joint，程序的其他部分不动。

```
char * s_joint (char * d, char * s)
{
    char * r = d;

    int x = 0, y = 0;

    while (d [x] != '\0') x ++;
    while ((d [x ++] = s [y ++]) != '\0') ;

    return r;
}
```

使用数组下标运算符[]需要一个指针操作数和一个整数，如以上新版的 s_joint 函数代码所示，为了移动数组下标，我们声明两个变量 x 和 y 并分别初始化为 0。

第一个 while 语句不断递增变量 x 的值，并最终使左值 d [x]代表那个字符编码为'\0'的元素，从另一个数组里复制过来的字符将从这里开始存放。字符串的连接工作由第二个 while 语句完成，具体执行过程请自行分析。注意，后缀++的优先级和[]相同，但由于表达式 x ++的值是运算符[]的操作数，所以必须先计算++的结果，从而也就不必在意优

先级和结合性的问题了。

当然，额外增加了两个变量 x、y 会令一些人不快。不过，在以下新版本的 s_joint 函数里，我们就解决了这个问题。

```
char * s_joint (char * d, char * s)
{
    char * r = d;

    while (d [0] != '\0') d ++;
    while ((d ++ [0] = s ++ [0]) != '\0') ;

    return r;
}
```

我们知道，表达式 e1[e2]是代表一个数组元素的左值。要得到下一个数组元素，既可以增加 e1 的值，也可以选择增加 e2 的值，反正最终都是左值* (e1 + e2)。

既然如此，我们可以递增变量 s 和 d 的值，毕竟它们都是指针变量；至于另一个操作数，就让它一直为 0 好了。在表达式 d ++ [0] = s ++ [0]中有三种运算符，后缀运算符++和[]的优先级相同，但它们是从左往右结合的，且它们都比=的优先级高，故此表达式等同于((d ++) [0]) = ((s ++) [0])。

练习 5.8:

假设数组 a 的内容是字符串“hello,world.”，编写一个程序将它反转为“.dlrow,olleh”，且不得借助于另一个数组。

5.4 指向数组的指针

我们已经学习了数组—指针转换，在下面的程序中，变量 a 是一个数组，变量 p 是一个指向 int 的指针。在语句 S1 中，表达式 p = a 是将 a 自动转换为指向其首元素的指针并写入变量 p。因为数组的元素类型是 int，所以转换后的结果是指向 int 的指针，赋值运算符两侧的类型一致，可以赋值。

这种自动转换非常方便。当然，如果我们不嫌麻烦的话，则可以自己生成这个指针，方法是取数组首元素的地址。在语句 S2 中，表达式 q = & a [0]是生成指向数组首元素的指针，并将它赋值变量 q。由于下标运算符的优先级高于一元&运算符，所以这个表达式等价于 q = & (a [0])。下标表达式 a [0]是一个左值，代表数组的首元素，其类型为 int，一元&运算符作用于这个左值，得到一个指向 int 的指针。

```
/***************c0515.c**************/
int main (void)
{
    int a [3], * p, * q, (* r) [3];
    p = a;                          //S1
    q = & a [0];                    //S2
    r = & a;                        //S3

    (* r) [0] = 1;                  //S4
    (* r) [1] = 2;                  //S5
    (* r) [2] = (* r) [1] + 1;      //S6
}
```

一元&运算符的操作数必须是左值，如果左值的类型是 T，则一元&运算符的结果类型是指向 T 的指针。那么，如果左值的类型是数组，则我们不就可以得到指向数组的指针了吗？那是自然。在语句 S3 中，a 是数组类型的左值，所以表达式& a 的结果自然就是指向数组的指针。具体地说，左值 a 的类型为 int [3]，所以表达式& a 的类型是指向 int [3]的指针。这个指针是赋给变量 r 的，变量 r 的类型也必须是指向 int [3]的指针。

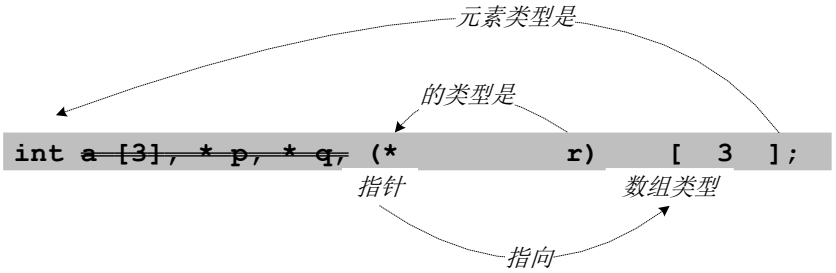


Fig.5-10 声明一个指向数组的指针变量

变量 r 的声明位于 main 函数内的第一行。如图 5-10 所示，因为圆括号阻断了标识符 r 与外界的结合，所以要先向左读，左边是星号“*”，要读作“r 的类型是指针”；然后向右读，遇到一个中特号“[”，读作“指向数组”；数组有 3 个元素；然后再向左读，“元素的类型是 int”。

所以，r 是一个指向数组类型的指针，被指向的数组类型以“有 3 个 int 类型的元素”为特征。简言之，r 是一个指向 int [3]的指针。从声明中去掉标识符 r 就得到了它的类型名 int (*) [3]。

在语句 S4 中，表达式(* r) [0] = 1 是给数组下标为 0 的元素赋值。因为下标运算符的优先级高于一元*运算符，所以这里使用了圆括号。

一元*是个神奇的运算符，如果它的操作数是指向 T 的指针，则该运算符的结果是一个类型为 T 的左值或者函数指示符，代表指针所指向的变量或者函数。在这里，左值 r 的类型是 int (*) [3]，先执行左值转换，得到一个 int (*) [3]类型的值。一元*运算符作用于这个指向数组类型的指针，得到一个左值，其类型为 int [3]，也即数组。既然表达式* r 的类型是数组，那么表达式(* r) [0]的结果也是左值，代表下标为 0 的元素。

接下来，语句 S5 执行类似的操作。稍微复杂一点的是语句 S6，子表达式(* r) [2]是个左值，代表下标为 2 的数组元素，但不执行左值转换；子表达式(* r) [1]也是个左值，代表下标为 1 的数组元素，但要执行左值转换。转换后的值与常量表达式 1 的值相加，再赋给运算符=的左操作数。

为了演示如何使用指向数组的指针，下面是另一个例子，不过这回它是作为函数 fsum 的形参。

```
/**************c0516.c*************/
int fsum (int (* pints) [5])
{
    int sum = 0;

    for (unsigned x = 0; x < sizeof * pints / sizeof (int); x ++)
        sum += (* pints) [x];

    return sum;
}
```

```
int main (void)
{
    int a [] = {3, 10, -5, 6, 22}, r;
    r = fsum (& a);
}
```

以上，函数 fsum 的参数 pints 是一个指向数组的指针，它的值指向一个数组，而该函数的功能是返回这个数组所有元素的累加和。

累加过程由 for 语句完成，变量 x 充当数组的下标，从 0 开始递增。for 语句的终止的条件是变量 x 的值小于数组的元素数量，即 x < sizeof * pints / sizeof (int)。在这里，由于一元*运算符优先级最高，sizeof 次之，乘性运算符/又次之，关系运算符<优先级最低，故这个表达式等价于 x < (sizeof (* pints) / sizeof (int))。

在这个表达式里，左值 pints 的类型是指向数组的指针，执行左值转换，得到一个指针类型的值；一元*运算符作用于它，得到一个数组类型的左值；sizeof 运算符得到数组的大小。表达式 sizeof (int)是数组的元素大小，数组的大小除以数组的元素大小，就得到了元素的数量。

在 for 语句的循环体，表达式(* pints)得到一个数组类型的左值，进一步地，表达式(* pints) [x]得到该数组的下标为 x 的元素，这是个左值，执行左值转换。最后，将转换后的值加到左值 sum。

所有数组元素的累加结果在变量 sum 里，函数 fsum 返回这个结果。但是，将函数的参数声明为指向 int [5]的指针不太明智，因为这是要求通过指针传入的数组必须有 5 个元素，不能多，也不能少。在 main 函数里，数组 a 的声明里虽然没指定大小，但这个大小是通过初始化器来隐式指定的，恰好是 5。在函数调用表达式里，子表达式& a 的结果类型是 int (*) [5]，恰好与形参 pints 的类型一致。

但是，如果数组 a 的大小不是 5，则此程序在翻译时必定招致一个警告，提示我们：函数的实参和形参在类型上不兼容。在 C 语言里，如果两个类型之间高度相似，或者干脆就是同一种类型，则称这两个类型是兼容的，或者说是兼容类型。无论是赋值，还是在函数调用中传递参数，都要求参与的操作数在类型上必须兼容。

为了避免这个警告，同时也为了让这个程序能够接受大小不同的数组，我们可以把函数 fsum 的声明改成这样：

```
int fsum (int (* pints) [], unsigned int siz)
{
    //代码从略。
}
```

对比上面的声明，新的声明有两点变化：第一，被指向的数组类型未指定大小。在 C 语言里，如果某个类型缺少必要的信息，从而无法知道它的大小，则这样的类型称为不完整类型。未指定元素数量的数组类型是不完整的数组类型，所以，形参 pints 所指向的类型是不完整的数组类型。

在 C 语言里，允许存在指向不完整数组类型的指针，原因很简单：所有指向数组的指针都具有相同的大小，与被指向的数组有几个元素无关。尽管指针指向的数组不知道大小，但指针本身的大小是确定的，这种指针类型本身是完整类型。实际上，在 C 语言里，所有指针类型都是完整类型。

第二，为了能够在被调用函数内知道数组的大小，新的声明添加了一个参数 siz，调用者负责将数组的大小通过它传递进来。

问题是，在这个新修改的函数里，形参 pints 的类型是 int (*) []，但假定我们传递的实参类型是 int (*) [5]，它们是兼容的（指针类型）吗？是的，C 语言保证它们是兼容的。

C 语言规定，如果两个数组中的一个具有常量大小，而另一个不具有常量大小（是变长数组）或者未指定大小（是不完整的数组），只要它们的元素类型是兼容的，则它们属于兼容的数组类型。这就是说，数组类型 int [5]和 int []是兼容的。进一步地，既然指向的类型是兼容的，则 int (*) []和 int (*) [5]自然也是兼容的指针类型了。

练习 5.9:

1.若函数 faas 的参数是指向数组的指针，该函数的功能是将下标为 0 和 1 的元素相加，赋给下标为 3 的元素。请写出这个函数的定义。

2.将上述带有参数 siz 的函数 fsum 补充完整，用它代替原来的 fsum 函数，并在你的机器上翻译和调试以验证自己的修改是否正确。

3.C 语言不允许声明不完整类型的变量,除非是一个数组且能够根据初始化器确定它的大小。以下，变量 a 和 pa 的声明是否合法？

```
char a [], (* pa) [];
```

如果数组里保存的是字符串，则它的大小就更不重要了，因为字符串是以空字符结尾的，对数组的访问可以在遇到空字符时结束。在下面的例子中，函数 fstrcmp 用于比较两个数组中保存的字符串是否完全相同，但这两个数组是通过指针传递进来的。

```
/*****************c0517.c************/
_Bool fstrcmp (char (* para) [], char (* parb) [])
{
    char * p = * para, * q = * parb;

    while (* p != '\0' && * q != '\0')
        if (* p ++ != * q ++) return 0;

    return * p == * q;              //S1
}

int main (void)
{
    char a [] = "The Wizard of Oz";
    _Bool b = fstrcmp (& a, & "Goodbye Mr Hollywood");

    char (* pc) [] = & a;
    b = fstrcmp (pc, & a);
}
```

先来看函数 fstrcmp，它的返回类型是_Bool，返回 0 则意味着两个字符串不同，返回 1 则表示两个字符串完全相同。形参 para 和 parb 的类型是指向数组的指针，要比较的字符串在这两个被指向的数组里，而要比较字符串，一般的方法是用两个指针分别指向这两个字符串，然后同时移动指针并比较它们指向的字符是否一样。

为此，我们在函数内部声明了变量 p 和 q，变量 p 的初始化器是* para，左值 para 的类型是指向数组的指针，先执行左值转换，转换为变量 para 的值，值的类型也是指向数

组的指针。一元*运算符作用于这个指针，得到一个数组类型的左值，代表那个被指向的数组；这个左值执行数组－指针转换，又得到一个值，指向数组首元素，其类型为指向 char 的指针，与左值 p 的类型兼容，可以初始化。

同理，变量 q 也被初始化为指向数组首元素的指针。现在，你可以认为变量 p 和 q 各自指向待比较的字符串。

字符串的比较操作是由 while 语句完成的，循环持续进行的条件是变量 p 和 q 的值所指向的字符都不是空字符，也就是都还没有指向字符串的尾部。为此，while 语句的控制表达式为* p != '\0' && * q != '\0'。在这里有三种运算符，其优先级从高到低分别是*、!=和&&，故这个表达式等价于((* p) != '\0') && ((* q) != '\0')。

while 语句的循环体是一个 if 语句，其控制表达式为* p ++ != * q ++，它等价于(* (p ++)) != (* (q ++))。显然，这个表达式比较变量 p 和 q 的值所指向的字符是否相同，同时递增这两个变量的值以指向下一个字符。在两个字符不相同的情况下，整个表达式的值（运算符!=的结果）为 1，直接返回到函数的调用者，返回值为 0；否则继续下一轮的循环。

在 while 语句中，循环能够持续进行的条件是未到达两个字符串的末尾，且 if 语句未发现两个字符不同的情况。那么，当程序的执行到达语句 S1 时，意味着已经到达两个字符串的末尾，或者至少已经到达其中一个字符串的末尾，且前面的比较都是成功的。

然而，就算是已经到达其中一个字符串的末尾，且前面的比较都是成功的，也不见得这两个字符串就是完全相同的，例如“abcd”和“abc”。在这种情况下，变量 p 和 q 的当前值所指向的字符肯定不同，其中一个指向空字符，另一个指向的字符不是空字符。

既然如此，我们可以对变量 p 和 q 的当前值所指向的字符进行比较，并返回比较的结果即可。话说，只有在两个指针所指向的字符都是空字符的情况下，运算符==的结果才有可能为 1。实际上，这个 return 语句也可以这样写：

```
return * p == '\0' && * q == '\0';
```

下面再来看 main 函数，数组 a 是用字面串初始化的，它的声明里未指定大小。按 C 语言的规定，在紧接着方括号“[]”之后的地方，数组 a 尚为不完整类型，但在这一行的分号“;”之后，它便成为完整类型，其大小已经由初始化器确定了。

变量 b 的初始化器是函数调用表达式，被初始化为函数调用的返回值。第一个实参由表达式& a 提供，其类型与形参 para 的类型兼容；第二个实参由表达式& "Goodbye Mr Hollywood"提供，字面串是数组类型的左值，所以一元&运算符的结果是指向数组的指针，且与形参 parb 的类型兼容。

初始化之后，变量 b 的值为 0，这是可以想象到的。接下来，我们声明了一个指向数组的指针变量 pc 并令它指向数组 a。然后，我们再用表达式 fstrcmp (pc, & a)来给左值 b 赋值。这里，左值 pc 经左值转换后的值和表达式& a 的值相同。显然，这次比较的是同一个数组里的字符串，因为这两个实参都指向同一个数组变量。

练习 5.10：

1.用类型名写出表达式& "Goodbye Mr Hollywood"的类型。

2.为什么 fstrcmp 可以比较同一个数组里的字符串而不会引起混乱？

3.如下所示，我们想改写 fstrcmp 函数，使用数组下标来比较两个字符串，而不是先前的指针操作。变量 m 和 n 的值在程序中用做数组下标，请将这个函数补充完整。

```
_Bool fstrcmp (char (* para) [], char (* parb) [])
{
    int m = 0, n = 0;
```

```
    while (_____________________________________________)
        if (_____________________________________________) return 0;

    return _____________________________;
}
```

5.5 元素类型为指针的数组

数组元素的类型也可以是指针，这样它就保存了一大堆指向其他变量或者函数的“地址”。在下面的程序中，变量 arrpi 和 arrps 就是这样的数组。

```
/*************c0518.c************/
int main (void)
{
    int a, b, * arrpi [2];

    arrpi [0] = & a;            //S1
    arrpi [1] = & b;            //S2
    * arrpi [0] = 10010;        //S3
    * arrpi [1] = 10086;        //S4

    char c, d, * arrps [2];

    arrps [0] = "Search";       //S5
    arrps [1] = "Project";      //S6
    c = * arrps [0];            //S7
    d = * arrps [1];            //S8
}
```

先来看变量 arrpi 的声明，如图 5-11 所示，标识符的左边是星号“*”，右边是方括号“[”。尽管声明中的标点符号不是运算符，但却具有作为运算符时的优先级属性，所以标识符 arrpi 先与方括号结合。

依据此图我们就可知道，变量 arrpi 的类型是数组，它有 2 个元素，元素的类型是指向 int 的指针。同理，程序后面的 arrps 也是数组，元素类型为指向 char 的指针。

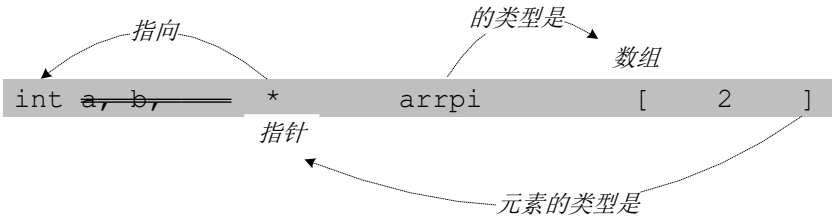


Fig.5-11 元素类型为指针的数组声明

接下来，语句 S1 为数组的第 1 个元素赋值。表达式 arrpi [0]代表数组下标为 0 的元素，是类型为“指向 int 的指针”的左值；表达式& a 的类型也是指向 int 的指针，运算符=的两个操作数类型兼容，可以赋值。同理，语句 S2 为数组下标为 1 的元素赋值。

语句 S3 是把 10010 保存到下标为 0 的元素所指向的变量。在这里，表达式* arrpi [0] = 10010 等价于(* (arrpi [0])) = 10010。子表达式 arrpi [0] 代表数组下标为 0 的元素，是 int *类型的左值，执行左值转换，得到一个指向 int 的指针（实际上是指

向变量 a 的)。一元*运算符作用于这个指针，得到一个 int 类型的左值（实际上是代表变量 a 的)。同理，语句 S4 是把 10086 保存到下标为 1 的元素所指向的变量。

一旦理解了语句 S1~S4，语句 S5~S8 也就非常简单了。这几条语句用的都是我们已经学习过的知识，我的本意是让大家复习一下学过的内容。

练习 5.11:

仿照以上对语句 S1 和 S3 的解析过程，说明语句 S5~S8 的执行原理，重点是字面串的类型，以及它到指针的转换。

既然数组可以保存指向其他变量或者函数的“地址”，那么就有必要来实际观察一下数组的内容。为此，我们将上述程序翻译为可执行文件，然后在 GDB 中调试。当程序执行到组成 main 函数体的右括号“}”时，用 p 命令打印这两个数组的值。

```
(gdb) p arrpi
$1 = {0x61fe48, 0x61fe44}
(gdb) p arrps
$2 = {0x404000 "Search", 0x404007 "Project"}
```

在显示数组 arrps 的内容时，结果有点奇怪。但不要误会，这并不是说数组 arrps 存储了字符串。在处理 char 类型时，GDB 总是显得过分热情。数组 arrps 仅存储了两个指向 char 的指针，这种指针通常用于指向字符串，所以热情的 GDB 索性把指针所指向的字符串一并显示出来。

为了帮助大家了解元素类型为指针的数组，下面是另一个例子。这个程序的特点是用数组来保存一堆指向函数的指针，然后再通过循环语句来自动一一调用这些函数，这听起来似乎有点意思。

```
/**********************c0519.c**********************/
int f_add (int x, int y)
{
    return x + y;
}

int f_sub (int x, int y)
{
    return x - y;
}

int f_mul (int x, int y)
{
    return x * y;
}

int f_div (int x, int y)
{
    return x / y;
}

int f_mod (int x, int y)
```

```
{
    return x % y;
}

int f_max (int x, int y)
{
    return x > y ? x : y;
}

int f_min (int x, int y)
{
    return x < y ? x : y;
}

int f_avg (int x, int y)
{
    return (x + y) / 2;
}

int main (void)
{
    int (* af []) (int, int) = {f_add, f_sub, f_mul, f_div,\
                                f_mod, f_max, f_min, f_avg},\
                r [sizeof af / sizeof (int (*) (int, int))];

    for (unsigned n = 0; n < sizeof r / sizeof (int); n ++)
        r [n] = af [n] (8, 6);
}
```

在这个程序中，我们定义了一堆函数：f_add 用于返回两个参数的和；f_sub 返回两个参数的差；f_mul 返回两个参数的积；f_div 返回两个参数的商；f_mod 返回两个参数相除后的余数（通常称之为模）；f_max 返回两个参数中的大者；f_main 返回两个参数中的小者；f_ave 返回两个参数的平均值。

这些函数除了名字不同外，类型完全相同，都是具有两个 int 类型的参数的、返回类型为 int 的函数，即：int (int, int)。在 main 函数内，将用数组 af 保存指向这些函数的指针，然后再用一个循环语句按顺序访问数组 af 的元素，通过它们做函数调用，数组 r 用于保存函数调用的返回值。

来看 main 函数，首先映入眼帘的是一个声明，它声明了变量 af 和变量 r。因为行太长，我们使用了续行字符“\”，它放在行的末尾。在程序翻译的时候，翻译器会将下一行和当前行连接起来形成一个完整的行。屏幕和文本编辑器的宽度有限，当行太长的时候，文本会自动折到下一行。如果使用续行符，则应当键入一个“\”，然后手动换行。

如图 5-12 所示，变量 af 的类型是数组。因为圆括号的关系，又得往左读，所以数组的元素类型是指针。圆括号之外的右侧是(int, int)，所以，这是指向函数的指针。所以总起来说，变量 af 是一个元素类型为“指向函数的指针”的数组。这是简明的说法，要完整描述就啰唆了：变量 af 是一个元素类型为“指向‘有两个 int 类型的参数的、返回类型

为 int 的函数’的指针”的数组，其类型名为 int (* []) (int, int)，数组的元素类型是 int (*) (int, int)。

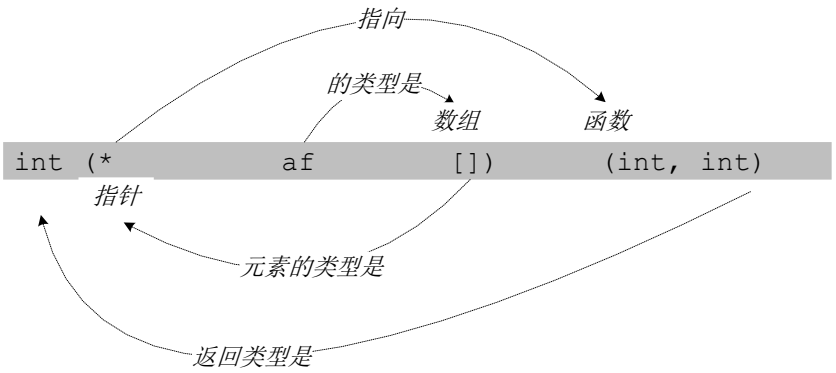


Fig.5-12 元素类型为指向函数的指针的数组声明

数组 af 的初始化器是{f_add, f_sub, f_mul, f_div, f_mod, f_max, f_min, f_avg}，这些都是函数的名字，在这里的身份是函数指示符，将自动执行函数指示符到指针的转换，并按顺序初始化数组的每个元素。这些函数的类型是 int (int, int)，转换为指针后的类型是 int (*) (int, int)，与数组 af 的元素类型兼容，可以初始化。

数组 r 用来保存函数调用的返回值，元素的数量自然应该与数组 af 相同。问题是，数组 af 的大小由初始化器决定，可灵活更改。那么，数组 r 应该怎么声明才能在元素的数量上与数组 af 灵活地保持一致呢？

很简单，程序中用表达式 sizeof af / sizeof (int (*) (int, int))来给数组 r 指定大小。子表达式 sizeof af 取得数组 af 的总大小，以字节计；子表达式 sizeof (int (*) (int, int))取得数组 af 的元素类型的大小。两者相除，就得到了数组 af 的元素数量，并作为数组 r 的元素数量。

接下来，for 语句通过数组 af 的元素调用它们所指向的函数，并把结果保存到数组 r 中。变量 n 的值用作数组 af 和 r 的下标，循环持续的条件是变量 n 的值小于数组 r 的元素数量。

在循环体中，表达式 r [n] = af [n] (8, 6)是一个赋值表达式，即，该表达式等价于(r [n]) = (af [n] (8, 6))。这是因为下标运算符和函数调用运算符的优先级相同，且都高于赋值运算符。

下标运算符和函数调用运算符的优先级相同，但它们是从左往右结合的，所以子表达式 af [n] (8, 6)等价于(af [n]) (8, 6)。也就是说，这是一个函数调用表达式，函数调用运算符的左操作数是表达式 af [n]的值。

在这里，左值 n 先执行左值转换；子表达式 af [n]是一个左值，代表数组 af 的某个元素，经左值转换后得到元素的值，这是一个指向函数的指针，它也是函数调用运算符的左操作数，于是将发起一次函数调用。函数调用的返回值被赋给左值 r [n]。

5.8 将数字转换为字符串

在任何一个字符集中，字母和数字字符都是最基本的成员。对于初学者来说，他们很清楚每个字符或者符号都对应着一个字符编码，但是对于一个阿拉伯数字，例如 9，他们可能分不清数学意义上的 9 和用来表示数字 9 的图形字符在计算机里有什么区别。

在任何字符集里，都有 10 个用来表示阿拉伯数字的图形字符，对应于字符常量'0'、'1'、'2'、'3'、'4'、'5'、'6'、'7'、'8'和'9'，它们的编码取决于字符集及编码方式。对于 ASCII 字符集来说，它们的编码分别是 0x30、0x31、0x32、0x33、0x34、

0x35、0x36、0x37、0x38 和 0x39。

简言之，数字 9 和字符 9 是不同的。为了在屏幕上打印一个“9”，你不能把 9 的数值传给它，而必须传送 9 的字符编码。如果一个数很大，例如 579，这就必须把它的每一个数位都分离出来，也就是分解为数字 5、7 和 9，然后分别转换为字符编码。

对于任何一个数 R，要分解它的每个数位，通常的方法是不停地将它除以 10 然后取余数，直至商为 0。以 579 为例，这个过程是：

579÷10＝57……9

57÷10＝5……7

5÷10＝0……5

分解出每个数位后，下一步的工作是将它们分别转换为图形字符。C 语言对使用何种字符集不加限制，但依然必须符合某些要求。除了本章前面已经提到的那些要求，还要求数字字符的编码必须是连续的、依次递增的。

有了这个保证，我们就可以将一个数字和字符常量'0'的值相加，来得到该数字所对应的图形字符的编码。这个方法好就好在具有可移植性，因为字符常量'0'的值取决于当前所使用的字符集和编码方式。

以 ASCII 字符编码方案为例，字符“0”的编码值是 0x30，如果我们分解出来的数位是 0，则字符“0”的编码是 0+0x30=0x30；如果我们分解出来的数位是 9，则字符“9”的编码是 9+0x30=0x39。

从上面的叙述可见，如果我们想把从 1 加到 N 的结果在屏幕上打印出来[4]，光是把结果送出去是没有用的，必须得用上面的方法把这个结果转换为一系列字符，下面的示例程序就演示了这种转换，是上述转换过程的具体实现。

```
/**********************c0520.c**********************/
char * ull_to_string (unsigned long long int n, char * s)
{
    int x = 0;
    char buf [22], * p = s;

    do
        buf [x ++] = n % 10 + '0';
    while (n /= 10);

    do
        * p ++ = buf [-- x];
    while (x);

    * p = '\0';

    return s;
}

unsigned long long int cusum (unsigned long long r)
{
    unsigned long long int sum = 0;
```

[4] 显示和打印设备通常只接受文本，所以要显示和打印数字，你得先把它转换为文本。

```
    for (unsigned long long int x = 1; x <= r; x ++)
        sum += x;

    return sum;
}

int main (void)
{
    char a [22], * ps = ull_to_string (cusum (1000), a);
}
```

为了通用性和灵活性，从数字到字符序列的转换工作可以组织为函数以便重复使用，为此我们定义了函数 ull_to_string。可想而知，该函数至少需要两个参数：第一个参数是待转换的数字；第二个参数是调用者给出的“容器”和“口袋”，用于存放转换后的字符序列。

在 C 语言里，最长的标准整数类型是 long long int，我们决定这个函数能够转换 unsigned long long int 类型的整数；至于第二个参数，我猜你会说，定义成一个数组吧，调用者可以传入一个数组来接收转换后的字符序列，就像这样：

```
char * ull_to_string (unsigned long long int n, char s [22])
{
    //代码从略
}
```

在这里，参数 n 用于接受待转换的数，s 是存放结果的数组。标准规定 unsigned long long int 类型的最大值起码得是 18446744073709551615，是 20 位的数字，我们将数组 s 定义为 22 个元素基本上是够了。

然而在 C 语言里，函数或者数组类型的参数声明是被特殊对待的。如果函数的参数被声明为“T 的数组”，则它被调整为“指向 T 的指针”；如果函数的参数是“返回 T 的函数”，则它被调整为“指向‘返回 T 的函数’的指针”。所以，刚才那个函数声明等同于：

```
char * ull_to_string (unsigned long long int n, char * s)
{
    //代码从略
}
```

换句话说，参数 s 的类型实际上是指向 char 的指针。如果你非要用数组的形式来声明一个参数，那么，因为上述原因，数组的大小也已经不重要了，不但会被忽略，而且也没有什么作用。在这种情况下，参数 s 可以声明为不完整的数组类型，就像这样：

```
char * ull_to_string (unsigned long long int n, char s [])
{
    //代码从略
}
```

在这里，参数 s 从数组调整为指针与数组的大小无关，并不需要知道它的大小，因此这样的声明是被 C 语言所接受的。

再来一个以函数作为参数的例子，以下，函数 fdemo 只有一个参数 param，该参数被声明为“具有两个 int 类型的参数且返回类型是 int”的函数：

```
void fdemo (int param (int, int))
```

```
{
    //代码从略
}
```

但是，前面我们讲了，该参数实际上被调整为指向函数的指针，所以参数 param 的类型实际上是指向上述函数类型的指针：

```
void fdemo (int (* param) (int, int))
{
    //代码从略
}
```

练习 5.12：

1. 设计一个函数 f，将它的参数 r 声明为数组形式，如 char r [22]。在函数内部用 sizeof 运算符获取参数变量 r 的大小以验证它到底是数组还是指针。

如图 5-13 所示，函数 ull_to_string 是要把变量 n 里的数值分解，得到它的每一个数位，再将各个数位变成数字字符。假定 n 的值是 5050，那么你会发现，用除以 10 取余的方法分解数位时，将依次得到 0、5、0 和 5，而不是我们期望的 5、0、5 和 0。

为此，我们在函数 ull_to_string 里声明了一个数组 buf 以临时存放这些倒序产生的数字字符，然后再把它们正过来。

声明变量 x 的目的是充当数组元素的下标，这样就可以逐一定位数组 buf 的每个元素以写入字符；声明变量 p 的目的是为了保存参数变量 s 的值，这样我们就可以在当前函数里使用变量 p 而保持变量 s 的值不变，并在函数尾部将它返回给调用者。

你可能觉得奇怪，这个指针是调用者传入的，为什么还要返回给调用者？答案是为了方便，这样就可以在调用者那里将函数调用表达式的值作为初始化器（参考 main 函数内变量 ps 的初始化器），或者作为运算符=的右操作数赋给左操作数。

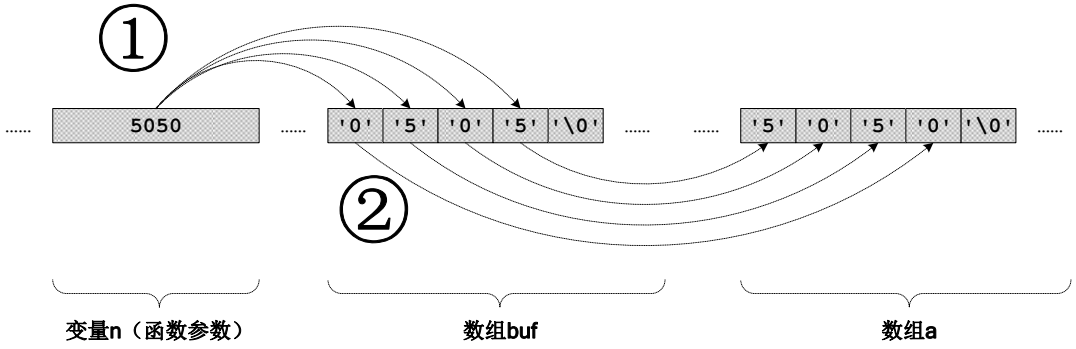


Fig.5-13 数字字符的反向重组原理

为了分解一个整数的各个数位，需要多次执行除以 10 的操作，这就要用到循环语句，比如 while 语句。然而 while 语句是先求值控制表达式再决定是否执行循环体，但是这个分解数位的工作更适合先执行循环体再求值控制表达式，为此我们使用了 do 语句，这是一种新的循环语句，其语法为

do *语句* **while** **(** *表达式* **)** **;**

do 语句由关键字“do”、语句、关键字“while”、一对圆括号和括号内的控制表达式，以及末尾的分号“;”组成。

如图 5-14 所示，do 语句的执行过程是这样的：先执行循环体，即“语句”，然后求值控制表达式，如果表达式的值为 0 则退出 do 语句，不为 0 则继续下一次循环。

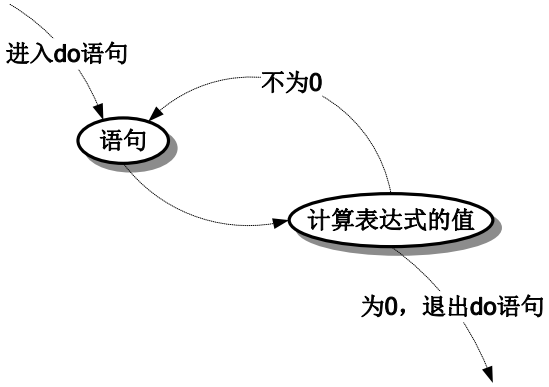


Fig.5-14 do 语句的工作流程

来看第一个 do 语句的循环体：

```
buf [x ++] = n % 10 + '0';
```

这里面有好几种运算符，后缀++和[]的优先级相同，且高于%，%的优先级高于+，=的优先级最低，所以这个语句等同于

```
(buf [x ++]) = ((n % 10) + '0');
```

要计算一个整数除以另一个整数之后的余数，需要使用二元%运算符，我们知道它也是乘性运算符。二元%运算符的操作数必须是整数类型，结果也是整数类型，例如表达式 5%2 的值是 1；表达式 10%2 的值是 0；表达式 777%10 的值是 7。在这里，表达式 n % 10 要先对 n 进行左值转换，将转换后的值除以 10 得到余数。

表达式 n % 10 的值是整数类型，字符常量'0'也是整数类型，将它们相加，其结果也是一个整数，而且是一个数字字符的编码。

顺便说一句，乘性运算符*、/和%，以及加性运算符+、-的操作数是整数类型的，必须先做整型提升，提升后的类型仍不一致的，还要再转换为一致的类型。

练习 5.13：

从类型和类型转换的角度分析表达式 buf [x ++] = n % 10 + '0'的求值过程。

再来看表达式 buf [x ++]，它是一个左值，用来接受那个数字字符的编码值。因为表达式 x ++的值是变量 x 递增之前的原值，所以它既用于定位数组元素以接受赋值，又执行递增操作以便定位下一个元素（在下次循环时）。

当前这个 while 语句的控制表达式为 n /= 10，其中/=是复合赋值运算符，它执行一个复合的操作：先用变量 n 的当前值除以 10，再把商重新保存回变量 n。也就是说，这个表达式等价于 n = n / 10。必须注意的是，符号“/”和“=”必须连写，中间不得有空白。

我们知道，赋值表达式的值是赋值运算符的左操作数被赋值之后的值，在这里，如果表达式 n /= 10 的值（也就是除法的结果）为 0 则退出 do 语句，如果不为 0 则继续执行下一次循环以分解剩余的数位。

上面我们用到了取余运算符%，那就顺便说一句，它也有对应的复合赋值运算符%=，它可用于组成复合赋值表达式 *e1* %= *e2*，在求值时，是用 e1 的值除以 e2，得到的余数再赋给左值 e1。也就是说，表达式 e1 %= e2 等价于 e1 = e1 % e2。

另一个需要注意的问题是，很多人可能觉得循环体位于 do 和 while 之间，所以语句再多也不用加花括号。这是不对的，如果语句超过一条，也必须添加花括号使之成为复合语句，就像这样：

```
do
{
    buf [x] = n % 10;
    buf [x ++] += '0';
}
while (n /= 10);
```

在这里，所有赋值运算符的优先级都低于下标运算符[]，所以表达式buf [x ++] += '0'等价于(buf [x ++]) += '0'，它是将左操作数的值和右操作数的值相加，然后再赋给左操作数。

如图 5-12 所示，拆解出来的各个数位保存在数组 buf 里，但顺序是反着的，这需要按相反的顺序一一取出，并保存到指定的数组里，这个数组的第 1 个元素由参数变量 s 的值所指向。为此，我们又用了一个 do 语句。第二个 do 语句的循环体是语句

```
* p ++ = buf [-- x];
```

因为运算符优先级的关系，表达式* p ++ = buf [-- x]等价于(* (p ++)) = (buf [-- x])。由于在函数的开头我们已经将参数变量 s 的值赋给了变量 p，所以 p 的值指向的位置也是 s 的值所指向的位置。

表达式 buf [-- x]用于取得数组 buf 中的数字字符，在上一个 do 语句里，每分解并保存一个数位后，变量 x 的值就递增，退出 do 语句后，它的值比数组 buf 中最后一个数字字符的下标大 1。因为这个原因，是要先递减变量 x 的值，再用递减之后的值作为下标从数组 buf 取得数字字符。为此，我们只能使用-- x 而不是 x --。总之，表达式 buf [-- x]是一个左值，代表数组 buf 的某个元素，但它是运算符=的右操作数，还要经左值转换后得到该元素的值。

表达式 p ++的值是变量 p 递增前的原值，这是一个指针，指向调用者那里的一个数组元素，所以表达式* p ++的结果是一个左值，代表那个元素，接受赋值，并递增变量 p 的值以指向下一个元素。

在反向传送结束后，还应当在目的字符串的尾部添加一个空字符。恰好，传送结束后变量 p 的值指向最后一个字符的下一个位置，故可直接在此位置写入字符常量'\0'。

为了取得灵活性，我们的目标是可以将任何 unsigned long long int 类型的数转换为可打印的字符串，或者说字符的序列。为此，我们定义了函数 cusum，用来计算从 1 加到 N 的结果。这个函数比较简单，可自行分析。

再来看 main 函数，在它的内部声明了一个字符类型的数组 a，用来存放转换后的字符序列。另一个变量 ps 的类型是指向 char 的指针，它被初始化为表达式 ull_to_string (cusum (1000), a)的值。可见，要完成初始化操作，需要调用 ull_to_string 函数。传入的第一个参数是要转换为字符串的整数值，它来自函数调用表达式 cusum (1000)；第二个参数是指向 char 的指针，它是由数组 a 经数组－指针转换得到的，转换后的值指向数组 a 的首元素，转换后的字符串从这里开始存放。

我们我们知道，函数参数的计算要先于实际的函数调用，故要调用 ull_to_string 函数，必须要先得到 cusum (1000)的结果，也就是发起函数调用，同时还要将左值 a 转换为指针，但这两件事的顺序不确定。

练习 5.14：

将函数 ull_to_string 改为用指针进行操作，也就是不使用下标运算符。

5.9 元素类型为数组的数组

在 C 语言里，构建数组的方法可以递归地使用。因此，可以声明一个数组，该数组的元

素类型也是数组，即数组的数组。例如：

```
int iarr [2][3];
```

这就声明了一个数组 iarr，它有 2 个元素，元素的类型也是数组。因此，iarr 是数组的数组。

如图 5-15 所示，这个声明应当这样来读：先从标识符 iarr 开始，因其右边是一对方括号，所以变量 iarr 的类型是数组，这种数组类型有两个元素，元素的类型呢？右边又是一对方括号，所以元素的类型还是数组。

综上，变量 iarr 是具有 2 个元素的数组，元素的类型是“具有 3 个 int 类型的元素的数组”。或者说，说变量 iarr 是具有 2 个元素的数组，元素的类型是 int [3]。如果用类型名来描述，则变量 iarr 的类型是 int [2][3]。

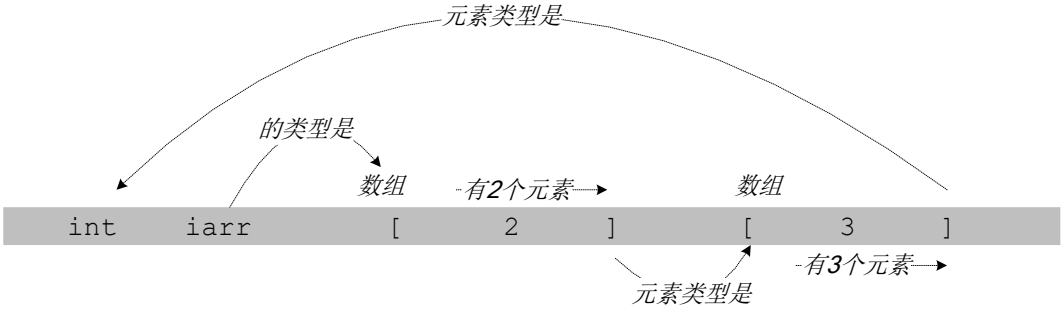


Fig.5-15 多维数组的声明

数组的数组被称为多维数组，声明一个更多维的数组是可能的。例如：

```
char carr [2][3][5];
```

这将 carr 声明为具有 2 个元素的数组；它的每一个元素又是具有 3 个元素的数组；而这 3 个元素中的每一个元素又是具有 5 个 char 类型的元素的数组。

多维数组的本质是“元素的类型也是数组”，或者“每个元素也是数组”，图 5-16 以数组 iarr 和 carr 为例揭示了这种蕴含关系。

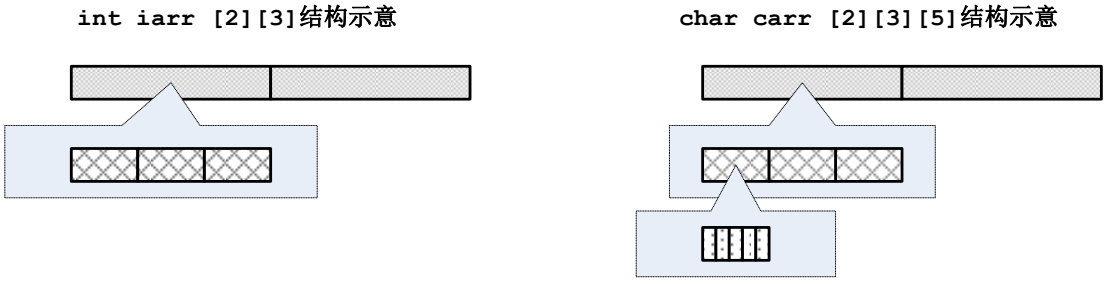


Fig.5-16 多维数组的元素也是数组

为了说明多维数组的声明、初始化和访问，下面给出一个程序。在程序中声明了一个数组 iarr，它具有 2 个元素，元素的类型是数组。如果从声明中去掉标识符，则这个数组的类型是 int [2][3]。

```
/***************c0521.c**************/
int main (void)
{
    int iarr [2][3];
```

```
    iarr [0][0] = 1;                    //S1
    iarr [0][1] = 2;
    iarr [0][2] = 3;
    iarr [1][0] = 4;
    iarr [1][1] = 5;
    iarr [1][2] = iarr [1][1] + 1;      //S2
}
```

就 C 语言的特点而论，符号“[”和“]”在数组的声明中用于指定元素的数量。但是在表达式里，它们的身份是下标运算符，用于指定数组的元素。

来看语句 S1，下标运算符是从左往右结合的，所以子表达式 iarr [0][0]等价于(iarr [0]) [0]。子表达式 iarr [0]得到数组 iarr 下标为 0 的元素，如图 5-17 所示。但这个元素也是一个数组，于是表达式 iarr [0][0]得到数组 iarr 下标为 0 的元素（也是数组）的下标为 0 的元素。这也是一个左值，是赋值运算符的左操作数，而右操作数 1 用来给它赋值。

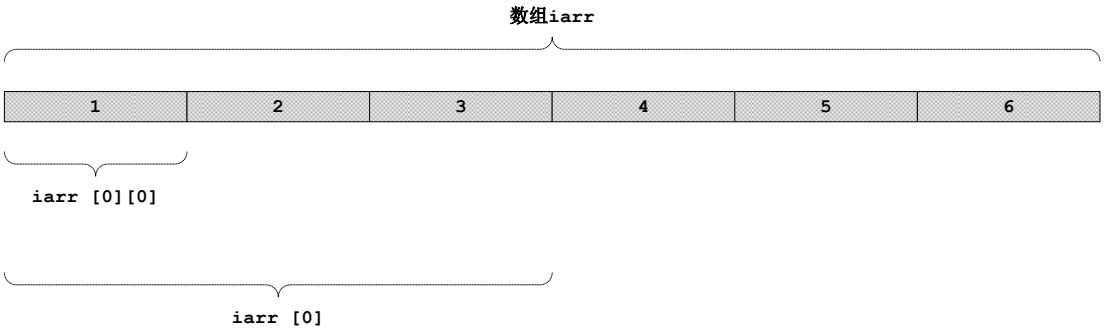


Fig.5-17 多维数组的下标及其所代表的元素

在 C 语言里，类型的匹配很重要。表达式 iarr [0]是 int [3]类型的左值；表达式 iarr [0][0]是 int 类型的左值；赋值运算符右侧的 1 是 int 类型的值。

后面的语句形式雷同，都是代表某个具体的数组元素，不再有什么好说的。唯一需要说明的是在语句 S2 中，表达式 iarr [1][2]是个左值，代表数组 iarr 下标为 1 的元素（这也是个数组）的下标为 2 的元素；表达式 iarr [1][1] 也是个左值，代表数组 iarr 下标为 1 的元素（这也是个数组）的下标为 1 的元素，但它不是赋值运算符的左操作数，要执行左值转换，转换后的值和常量表达式 1 的值相加。

练习 5.15：

我们知道，若 a 是数组类型的左值，则 a [1]等价于* (a + 1)。用这种方法修改本节的程序，将所有语句的表达式修改为等价的指针形式，并从数组—指针转换、指针的加法和一元*运算符的结果是左值等角度解析这些表达式的求值过程。

在本节的最后，我们再来看一个多维数组的例子。这个例子里声明了两个数组 iarr 和 adst，程序的功能是将数组 adst 的每个元素的值都加到数组 iarr 的对应元素中。

```
/************************c0522.c*********************/
int main (void)
{
    int iarr [2][3] = {{1, 3, 5}, {2, 4, 6}};
    int adst [2][3] = {[0] = {10, 11, 12}, [1] = {[0] = 15},\
```

```
                                [1][1] = 16, [1][2] = 17};

    for (int x = 0; x < 2; x ++)
        for (int y = 0; y < 3; y ++)
            iarr [x] [y] += adst [x] [y];
}
```

因为数组是带有子变量的变量，所以 iarr 和 adst 的初始化器需要一个最外层的花括号。然而，由于这两个数组的元素依然是带有子变量的数组，所以还需要内层的花括号。如图 5-18 所示，初始化器{1, 3, 5}用于初始化数组 iarr 的第一个元素；由于这个元素依然是含有 3 个元素的数组，故花括号内的 1、3 和 5 分别用于初始化这些元素。同样的道理，初始化器{2, 4, 6}用于初始化数组 iarr 的第 2 个元素及其子元素。

再来看数组 adst 的初始化器，它使用了指示器。顶层花括号内的指示器[0]对应于数组 adst 下标为 0 的元素（数组），所以初始化器[0] = {10, 11, 12}用于初始化这个元素及子元素。

同理，位于顶层花括号内的指示器[1]对应于数组 adst 下标为 1 的元素，这个元素依然是含有 3 个元素的数组，初始化器{[0] = 15}仅初始化下标为 0 的元素，因初始化器数量不足，剩余两个元素被初始化为 0。

接着，指示器[1][1]直接隶属于顶层的花括号，它指示数组 adst 下标为 1 的元素（数组）的下标为 1 的元素。同理，指示器[1][2]直接隶属于顶层的花括号，它指示数组 adst 下标为 1 的元素（数组）的下标为 2 的元素。

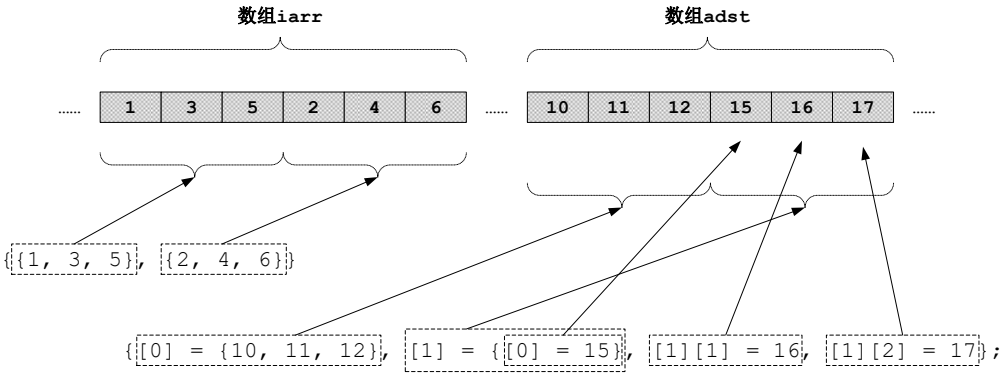


Fig.5-18 初始化器与数组元素的对应关系示意图

程序的主体部分是嵌套的 for 语句，第一个 for 语句的循环体也是一个 for 语句。显然，对于外层 for 语句的每一次循环，内层 for 语句要循环 3 次。也就是说，当变量 x 的值为 0 时，变量 y 的值要经历从 0 到 2 的变化过程；当变量 x 的值为 1 时，变量 y 的值依然要经历从 0 到 2 的变化过程。

在第二个 for 语句的循环体，表达式 iarr [x] [y] += adst [x] [y]利用变量 x 和 y 的上述变化来访问数组的每一个元素，数组 adst 的元素被加到数组 iarr 的对应元素。

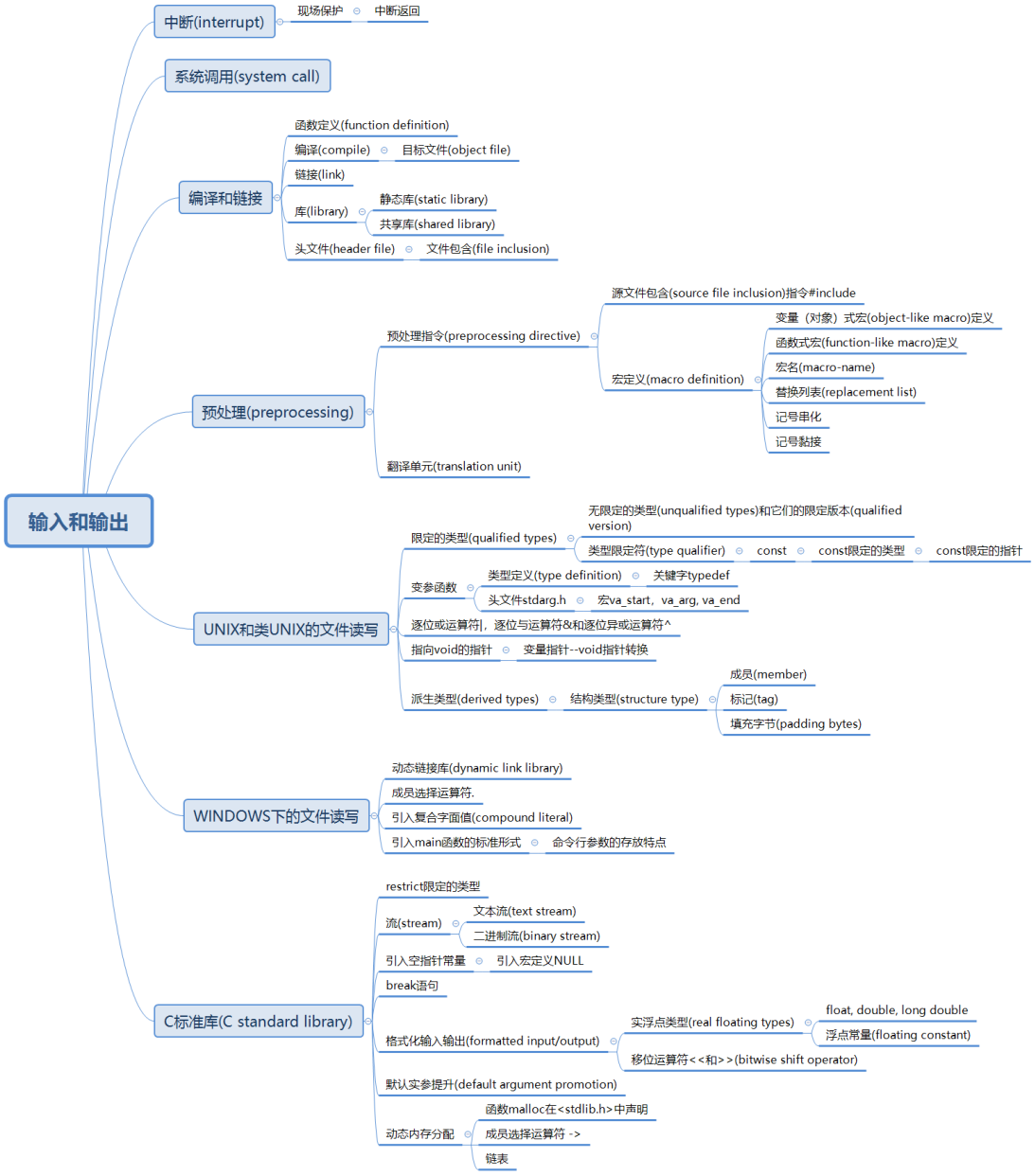
练习 5.16：

1.若数组 ac 有 2 个元素，元素的类型是“具有 20 个 char 类型的元素的数组”，如果要求用字面串来初始化 ac，它该如何声明？

2.在下面的程序中，实参* * a 和形参 p 兼容吗？为什么？对于程序中的数组 a，表达式(int) a 是把数组（的内容或者值）转换为 int 类型吗？为什么？

```
void f (int * p)
```

```
{
    /*... ...*/
}

int main (void)
{
    int a [2][3][7] = {0};
    f (* * a);
}
```


输入和输出
中断(interrupt)
现场保护
中断返回
系统调用(system call)
编译和链接
函数定义(function definition)
编译(compile)
目标文件(object file)
链接(link)
库(library)
静态库(static library)
共享库(shared library)
头文件(header file)
文件包含(file inclusion)
预处理(preprocessing)
预处理指令(preprocessing directive)
源文件包含(source file inclusion)指令#include
宏定义(macro definition)
变量（对象）式宏(object-like macro)定义
函数式宏(function-like macro)定义
宏名(macro-name)
替换列表(replacement list)
记号串化
记号黏接
翻译单元(translation unit)
UNIX和类UNIX的文件读写
限定的类型(qualified types)
无限定的类型(unqualified types)和它们的限定版本(qualified version)
类型限定符(type qualifier)
const
const限定的类型
const限定的指针
变参函数
类型定义(type definition)
关键字typedef
头文件stdarg.h
宏va_start，va_arg, va_end
逐位或运算符|，逐位与运算符&和逐位异或运算符^
指向void的指针
变量指针--void指针转换
派生类型(derived types)
结构类型(structure type)
成员(member)
标记(tag)
填充字节(padding bytes)
WINDOWS下的文件读写
动态链接库(dynamic link library)
成员选择运算符.
引入复合字面值(compound literal)
引入main函数的标准形式
命令行参数的存放特点
C标准库(C standard library)
restrict限定的类型
流(stream)
文本流(text stream)
二进制流(binary stream)
引入空指针常量
引入宏定义NULL
break语句
格式化输入输出(formatted input/output)
实浮点类型(real floating types)
float, double, long double
浮点常量(floating constant)
移位运算符<<和>>(bitwise shift operator)
默认实参提升(default argument promotion)
动态内存分配
函数malloc在<stdlib.h>中声明
成员选择运算符 ->
链表

第 6 章　输入和输出

在对 C 语言有了基本的认识之后，我们现在就可以讨论输入输出的问题了。对于绝大多数初学者来说，这也正是他们比较感兴趣的部分。然而，输入输出并不是 C 语言的一部分，和 C 语言没有关系，这也是为什么我们现在才讲的原因之一。

对于一台计算机来说，处理器位于核心地位，但处理器只关心程序和指令如何执行，数据如何操作，至于这些数据的来源和去向既没有感知，也并不关心。

打比方来说，处理器只是一个来料加工的车间，车间的任务是加工和生产，但并不关心原材料的来源，以及最终的产品送到哪里。至于产品如何包装，如何联系运输企业、港口、码头等这些事情，不是车间的事情。

同样地，C 语言是对处理器内部操作的高级抽象和包装，它仅仅是用语法要素来描述数据的组织、加工和操作，也同样不关心数据的输入和输出如何进行。所以，你不能指望它提供屏幕打印或者文件操作这样的语法要素。

6.1 输入输出那点事

输入输出不是一件容易的事。首先，设备千千万万，各有特色，功能不同，各有各的工作方式。比如说，显示器和打印机是用“点”的组合来呈现文字和图像；喇叭是音频电流转换为声波；磁盘是将计算机内的数字以不同的形式记录下来。

如图 6-1 所示，既然是要连接到计算机，那么，每种设备都需要在计算机一侧安装一个代理，称为 I/O 接口，这是一个信号的中转和加工模块，类似于港口、码头和海关。如果外部设备是模拟的，I/O 接口还要在模拟和数字信号之间来回转换。很多 I/O 接口是用可插拔的板卡来承载的，如声卡、显卡、网卡，等等。

这是可以理解的。很多设备，比如老式显示器和喇叭，都是模拟设备。喇叭需要一个连续变化的、代表着声音强弱和音色变化的电流来驱动。它流经一个线圈，产生变化的磁场，与喇叭内部的固有磁场相互吸引或者排斥，从而推动纸盆运动，产生声波。

但是，声音在计算机内部是以数字的形式存在，是某些人在某个时候通过采样而数字化的声音信号。它把连续变化的音频电流以固定的间隔进行测量（称为采样），然后以数字的大小代表测量的结果（声音的强弱）。

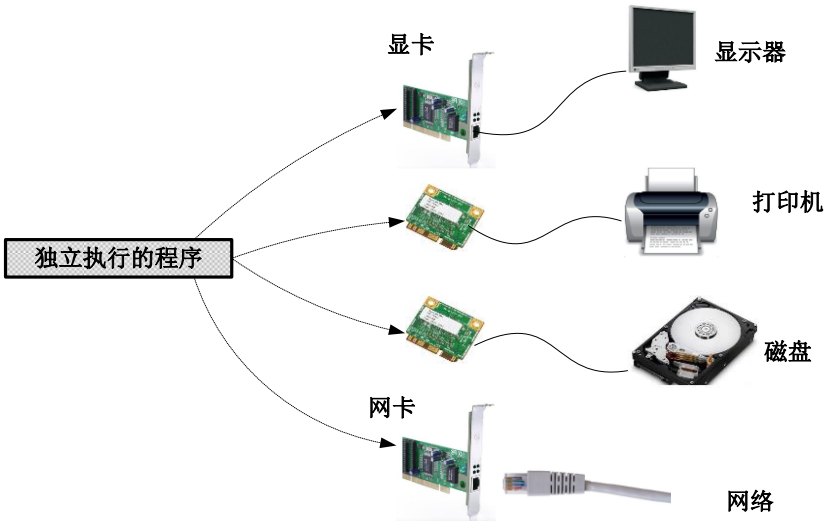


Fig.6-1 输入输出的接口和设备

所以，以声卡为例，这个 I/O 接口的作用是采样和回放——对话筒来的模拟信号（音频电流）进行采样和数字化；对计算机内部来的数字信号加以转换，使之重新变为可以驱动

喇叭发声的音频电流。换句话说，声卡的作用之一就是在数字和模拟信号之间转换。

再比如显示器，它以描点的方式呈现文字和图像，屏幕上的每个点都是一个像素。为了显示文字或者图像，你需要告诉它屏幕上每个点的明暗或者颜色。原则上，显示过程是逐行逐点进行的，从左上角到右下角，这称为一帧。为了图像稳定和变化的需要，每秒钟要重复进行多次，这称为刷新。

在计算机一侧，为了配合显示器的刷新，需要将显示的内容转换为显示器可以接受的形式。除此之外，还需要维持它们之间的同步关系以获得稳定的图像。

如果没有操作系统，一个独立执行的程序需要应付一切繁复琐碎的细节。它需要检查设备是否已经连接到计算机，还需要对它进行初始化，按照 I/O 接口的编程规范做各种准备工作，准备数据，启动设备，在恰当的时机接收或者送出数据，按照 I/O 接口的要求对数据进行必要的加工和转换。如果 I/O 接口复杂的话，这是非常麻烦非常可怕的工作。

然而，如果有了操作系统，这一切就会变得简单。对于普通用户来说，操作系统只是提供了一个方便操作的界面，你可以通过它整理文件、启动或者关闭程序。而对于程序员来说，操作系统是一个方便软件开发的平台。

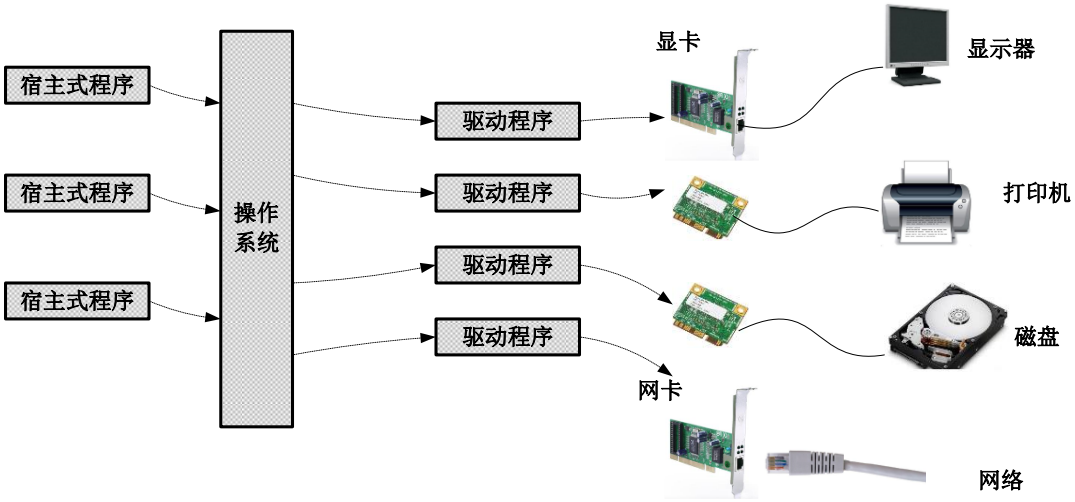


Fig.6-2 操作系统对输入输出的管理

如图 6-2 所示，依赖操作系统才能运行的程序叫宿主式程序，或者叫应用程序。应用程序和操作系统都直接由处理器执行，但是，应用程序要获得执行，必须先由用户发出一个命令（通过命令行输入文件名或者双击一个图标），然后，操作系统负责将应用程序加载到内存，最后让处理器执行应用程序。在应用程序执行的过程中，还会进行一些控制和调度工作，比如任务切换。

多数操作系统允许同时执行多个程序，比如一边听歌一边打游戏，同时还开着聊天软件。典型地，在只有一个处理器的情况下，多个程序需要在处理器上轮流执行。但由于处理器的速度极快，轮流执行通常不会被察觉，从人的视角来看，程序都是同时工作的。

问题在于，操作系统如何知道何时该将一个程序停下来，轮到另一个程序在处理器上执行呢？如果没有一个外部来的信号，这很难办到。就像你告诉一个人去睡觉，并且希望他在没有闹钟和别人提醒的情况下，自己能够在 5 点钟醒过来一样。好在现代的计算机中都有中断机制。中断可以看成是拍一下处理器的肩膀，告诉它有一些特殊的事情发生了，最好是放下手头的工作来处理一下。

中断的来源很多，比如不间断电源的掉电信号。不间断电源设备有电池作为备用能量，当市电停掉时，内部的电池就负责供电，但考虑到电池的容量，它只能维持不太长的时间。如果市电停掉，不间断电源设备就会发出一个中断信号。

再比如，在计算机内部有一个时钟，它会每隔一小段时间（比如几个毫秒）发出一个中断信号。

各种来源的中断信号都汇总到处理器的一个引脚，通过这个引脚来引起处理器的注意。如果处理器被设置为允许响应中断，那它就应该在当前指令执行完成后，放下手头的工作来响应这个中断。每个中断都有编号，称为中断号；每个中断号都对应着一段程序代码，称为中断例程，这都需要提前设置好。当中断发生时，处理器根据中断号找到对应的中断例程，然后执行这个例程。中断例程的末尾必须有一条中断返回指令，处理器执行这条指令，从中断例程返回到原来的地方继续执行。

处理器内部有很多寄存器，指令指针寄存器保存着下一条需要执行的指令，数据寄存器保存着数据和地址；状态寄存器保存着当前程序的各种状态；还有一些寄存器正保存着当前指令的中间结果。在这种情况下，在转入中断处理程序时，必须予以保存，这称为保护现场；当处理器从中断过程返回时，必须予以恢复，就像什么都没有发生过一样才行。

为了进行多任务调度，操作系统需要处理器响应时钟中断。每当时钟中断发生时，不管处理器在执行哪个任务，都必须保护好现场，然后来执行中断处理程序。很好，这个中断处理程序是特意编写的、用于进行任务调度的代码，用于从其他正等待执行的程序中选出一个来执行。如果这个等待执行的程序以前执行过，则恢复现场，让它在处理器上继续执行。

多任务调度和保护现场可以看成你在一个公共的大堂里吃饭，而这个大堂就是计算机的处理器。你正坐在大堂里吃饭，桌子上摆满了杯盘，你也熟悉大堂里每件物品的位置，也知道如何取用。当你正在吃的时候，突然有另一个人也要在这里吃饭。那怎么办呢，要等你吃完嘴里这一口之后，瞬间催眠，然后把你坐的位置、杯盘的位置，以及大堂的格局和摆设都记下来，清场，这就是保护现场。

然后，按另一个人的要求重新布置。当另一个人正吃呢，轮到你吃了，再把那个人的现场记下来。把你的位置、杯盘的位置和大堂的格局和摆设按原样一一恢复。当你醒来时，可以继续接着上一次的进度继续吃饭，你上一次正准备夹哪个菜，现在就可以开始夹了，完全不记得中间还有这一档子事。

有了操作系统，应用程序不必亲自访问输入输出设备，它只需要简单地向操作系统发出请求就行了。如图 6-2 所示，操作系统会把你的请求传递给设备驱动程序。每个设备都有对应的驱动程序，它负责通过 I/O 接口和设备进行通信。设备驱动程序通常是由设备的生产厂家提供，因为只有他们才了解如何与设备打交道，这样也可以把一些涉及内部机密的东西隐藏起来。这样，他们对外宣布如何调用设备驱动程序，至于它怎么与设备通信，就无关紧要了。

顺便说一句，在多任务状态下，多个应用程序可能会同时访问同一个输入输出设备，比如多个程序都要在打印机上输出。在这种情况下，操作系统还必须提供排队功能。

6.2 系统调用

事实上，不单单是输入和输出，操作系统也会提供其他一些功能，总的目的是方便应用程序的编写，让程序员们少写代码，轻松工作。操作系统提供的功能称为系统服务，而为了使用操作系统提供的功能，应用程序需要向操作系统请求这些服务。

对于应用程序来说，这些服务是操作系统的入口点，可以使应用程序从这些入口进入操作系统内部，完成所需要的功能后再返回，这些入口点称为系统调用。所有操作系统都会提供系统调用，但数量或多或少。

早先，操作系统的系统调用通过软中断实现，但新近的处理器通常内置速度更快、效率更高的机器指令，所以操作系统也可能采用这些机器指令快速进入系统服务例程。但是出于兼容性的原因，软中断方式依然可用。软中断，顾名思义，肯定和由硬件发出的中断信号不

同。是的，中断除了可以由硬件产生之外，也可以由软中断指令触发。这等于是让程序员通过一条指令手工产生一个中断，从而让处理器放下手头的工作，转去执行预先安排好的另一段代码。

在基于 INTEL x86 处理器的 LINUX 操作系统上，系统调用的传统入口是 0x80 号软中断。这是系统调用的总入口，包含了大量的子功能，程序员需要在进入前通过寄存器来指定功能号，也就是具体做哪件事，还要通过寄存器来传递做这件事所需要的参数。

为了通过 LINUX 的系统调用来显示文本，需要通过软中断 0x80 进入 LINUX 操作系统内部。但是，这已经超出了 C 语言的能力范围，因为 C 语言是对处理器的包装和抽象，它只用来描述“做什么事”而不是“到底在处理器上如何用机器指令来做”，自然也不可能提供一个能让你发出 0x80 号软中断的表达式和语句。说到底，这件事必须使用机器语言或者汇编语言才能办到。问题在于，C 语言说我又不是汇编语言和机器语言，这种事情不要来找我。那么，这该咋办呢？在编程实践中，这件事最终还是落在了 C 实现身上。

在整个程序设计过程里，C 实现处于微妙的境地。首先，它并不是 C 语言的一部分，也不是操作系统的一部分，但它却和这两者有密切的联系。在这一边，它是 C 语言的具体实现，理解 C 语言的语法；在那一边，它知道操作系统对可执行文件的格式和要求，也知道如何生成操作系统需要的东西，对运行当前程序的处理器也很了解。所以，它也就能够将 C 语言源程序翻译成符合操作系统要求并且能够在那种处理器上运行的可执行文件。

既然 C 实现在 C 语言和操作系统之间左右逢源，对双方都很了解，那么，它就可以拥有一些 C 语言和操作系统都不方便实现的功能。比如，我们可以在 C 源程序里嵌入一段汇编代码，来做一些用 C 语言实现不了的功能。对于程序员来说，虽然不是 C 语言里的东西，但既然有人为此负责，那还有什么可说的。

光说不练假把式，而且越说越糊涂。现在，让我们用一个实例来说明如何通过系统调用在 LINUX 控制台屏幕上打印一行文字“hello world.”。当然，我们说了，这需要在 C 源程序里嵌入汇编代码。

```
/*************c0601.c**************/
int main (void)
{
    int r;

    __asm__ (
               "mov eax, 0x4 \n\t"
               "mov ebx, 0x1 \n\t"
               "mov edx, 0xd \n\t"
               "int 0x80 \n\t"
               : "=a" (r)
               : "c" ("hello world.\n")
             );

    return r;
}
```

汇编指令是提供给 C 实现的，为了与普通的 C 语言代码分开，需要一个标志，这个标志就是“__asm__”，后面圆括号里的内容就是用汇编语言写的代码。如果没有这个标志，C 实现就会把这些汇编语言指令当成 C 语言来翻译。当然，很多人没学过汇编语言，所以看不懂，不过没关系，你不需要看懂，听我说个大概就行了，毕竟我们是来学习 C 语言而不

是汇编语言的。

个人计算机的处理器多是 INTEL x86 系列，它有好几个通用寄存器，分别取名为 EAX、EBX、ECX 和 EDX，等等，通用的意思是它可以由汇编程序员自由地用于不同的目的。软中断 0x80 是一个总入口，为了通过 0x80 号中断提供系统调用服务，LINUX 操作系统要求使用 EAX 寄存器来指定子功能号。因此，"mov eax, 0x4 \n\t"是将十六进制数字 4 传送到 EAX 寄存器，意思是指定 4 号功能，4 号功能是写文件和设备。GCC 允许使用的格式之一是每一行汇编代码用脱转序列\n\t 结束，并且用双引号围起来。

文件和设备很多，需要进一步指明写哪个文件和设备，这需要通过 EBX 寄存器来指定，因此，"mov ebx, 0x1 \n\t"是将十六进制数字 1 传送到 EBX 寄存器，意思是写入代号为 1 的设备。1 代表的是控制台屏幕。

同时，写入的字符数量也要通过 EDX 寄存器指定。"mov edx, 0xd \n\t"的意思是写入 13（0xd）个字符。

写入的内容从哪里来呢？你不必把要写入的内容都一一传进去，你只需要传入一个字符数组的地址到 ECX 寄存器就行了。嵌入式汇编的价值在于它允许寄存器和 C 语言里面的变量交换数据，但这样的指令要用冒号“:”打头以示区别。

第一个冒号用于指定输出功能。当系统调用返回后，实际写到屏幕的字符数由 EAX 寄存器返回。然而我们已经声明了一个 int 类型的变量 r，并希望把这个字符数保存到变量 r 中。在这里，

```
: "=a" (r)
```

的意思类似于 C 语言里的赋值表达式

```
r=a
```

是把 EAX 寄存器的内容传送到变量 r。等号“=”表示具有破坏原内容的写操作，对 32 位 INTEL x86 系列处理器而言，“a”表示 EAX 寄存器。

对不同的处理器而言，输出的写法也不一样。如果你使用的处理器不是 INTEL x86 系列，可参阅 GCC 手册，或者写信告诉我，我们一起探讨。

第二个冒号用于指定输入功能，比如将变量的值传送到寄存器。因此，: "c" ("hello world.\n")的作用是把字符串的首地址传送到 ECX 寄存器，对 32 位 INTEL x86 系列处理器而言，“c”表示 ECX 寄存器。字面串"hello world.\n"首先被用于创建一个不可见的数组，然后执行数组—指针转换，并把这个指针类型的值（地址）传送到 ECX 寄存器。

最后，汇编指令"int 0x80 \n\t"用于触发一个 0x80 号软中断。如图 6-3 所示，当处理器执行这条指令后，将转入中断处理过程，进入操作系统内部执行。在操作系统内部，将依据约定，根据各个寄存器的内容来做不同的处理（在这里是访问显示系统）。

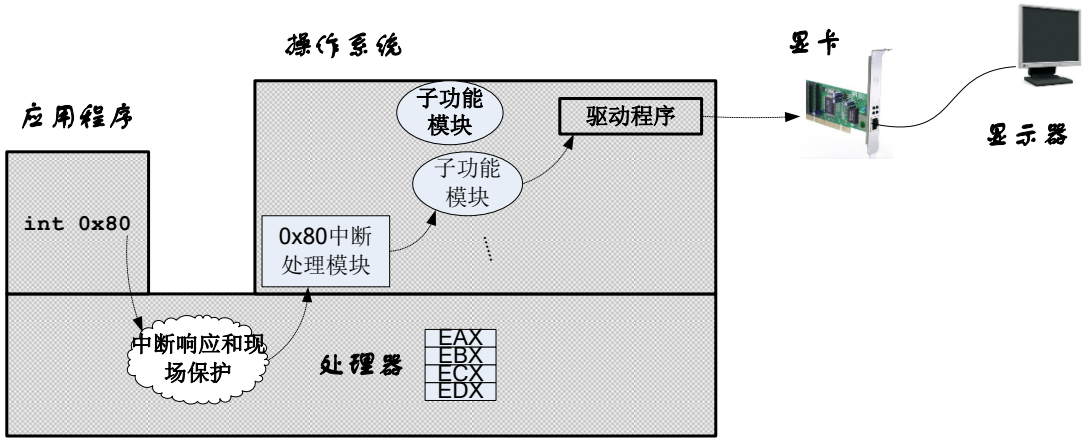


Fig.6-3 应用程序通过系统调用进入操作系统服务的过程

现在，让我们启动 LINUX 操作系统，使用文本编辑软件将上述 C 程序保存为源文件 c0601.c。然后，用下面的命令将它翻译成可执行文件 c0601.out 并加以执行：

```
% gcc c0601.c -o c0601.out -masm=intel
% ./c0601.out
hello,world.
%
```

显然，运行翻译后的可执行文件将打印“hello,world.”。历史上，针对不同的处理器有不同的汇编格式，即使是针对同一种处理器，不同公司的产品也使用不同的汇编指令格式。我们这里并没有使用 GCC 默认的汇编格式，而是使用 INTEL 公司的汇编格式，故必须用-masm 选项来指定为“intel”。

在屏幕上打印固定的内容显然没什么意思，我们需要更加灵活的打印方案，从而可以自由决定打印什么东西，比如从 1 加到 100 的结果。在下面的示例程序中，通过系统调用打印字符串的功能已经被封装为函数 write_string，它可以打印任何字符串。

```
/******************c0602.c*******************/
int write_string (char * s)
{
    int l = 0, r;

    for (char * p = s; * p ++ != '\0'; l ++) ;

    __asm__ (
              "mov eax, 0x4 \n\t"
              "mov ebx, 0x1 \n\t"
              "int 0x80 \n\t"
              : "=a" (r)
              : "c" (s), "d" (l)
             );

    return r;
}

char * ull_to_string (unsigned long long int n, char * s)
{
    int x = 0;
    char buf [20], * p = s;

    do
        buf [x ++] = n % 10 + '0';
    while (n /= 10);

    do
        * p ++ = buf [-- x];
    while (x);
```

```
    * p = '\0';

    return s;
}

unsigned long long int cusum (unsigned long long r)
{
    unsigned long long int sum = 0;

    while (r) sum += r --;

    return sum;
}

int main (void)
{
    write_string ("1+2+3+...+1000=");

    char a [20];

    write_string (ull_to_string (cusum (1000), a));
    write_string ("\n");
}
```

在函数 write_string 内部，需要统计字符串的长度，因为 LINUX 的系统调用需要传入这个长度。为此我们使用了 for 语句，它的工作原理已经在上一章里讲过了，这里就不再重复。

在嵌入式汇编代码里，："c" (s), "d" (l)的作用是将变量 s 的内容（字符串的首地址）传送到 ECX 寄存器，将变量 l 的内容（字符串的长度）传送到 EDX 寄存器，这当然包含了一个左值转换的过程，将左值 s 和 l 转换为它们的存储值。

现在来看 main 函数，它先是打印一串文本“1+2+3+...+1000=”，但是在字面串的末尾没有使用换行符“\n”，我们是希望将计算结果打印在同一行，并位于它后面。

接下来是计算从 1 加到 1000 的结果并转换为字符串，然后打印这个字符串，这三个任务由表达式 write_string (ull_to_string (cusum (1000), a))一次性完成，而且还是一个嵌套的函数调用表达式。

C 语言规定，对于函数调用，在进入函数体执行前，要先完成所有实际参数的求值。这就是说，在函数 write_string 开始执行前，必须先调用函数 ull_to_string 以得到它的返回值；而要想开始执行函数 ull_to_string，则必须先调用函数 cusum 以得到它的返回值。

最后，将上述程序保存为源文件 c0602.c，并在 LINUX 上按下面的方法翻译成可执行文件 c0602.out，并执行：

```
% gcc c0602.c -o c0602.out -std=c11 -masm=intel
% ./c0602.out
1+2+3+...+1000=500500
%
```

翻译选项-std=c11 是必要的，但也可以改为-std=c99。之所以要加入这么一个选项，是因为 C99 之前，for 语句的第一部分只能是表达式而不能是声明，有些 C 实现只能在你打开这一选项时才接受这种写法。

在打印了从 1 加到 1000 的结果之后，再打印个换行符“\n”是必要的。如果不这样的话，结果打印之后，LINUX 的系统提示符将出现在同一行上：

```
% ./c0602.out
1+2+3+...+1000=500500%
```

6.3 编译和链接

有些同学发现，函数 write_string 和 ull_to_string 不单单是当前这个程序需要使用，它似乎还具有通用性，别的程序没准儿也能用上。因此，他们希望将这些函数分离出来，单独保存为一个文件。这个文件既可以自己用，也可以给别的同学用。如果将这个文件起名为 iotool.c，则它的内容是这样的：

```
/******************iotool.c*****************/
int write_string (char * s)
{
    int l = 0, r;

    for (char * p = s; * p ++ != '\0'; l ++) ;

    __asm__ (
              "mov eax, 0x4 \n\t"
              "mov ebx, 0x1 \n\t"
              "int 0x80 \n\t"
              : "=a" (r)
              : "c" (s), "d" (l)
            );

    return r;
}

char * ull_to_string (unsigned long long int n, char * s)
{
    int x = 0;
    char buf [20], * p = s;

    do
        buf [x ++] = n % 10 + '0';
    while (n /= 10);

    do
        * p ++ = buf [-- x];
    while (x);
```

```
    * p = '\0';

    return s;
}

unsigned long long int cusum (unsigned long long r)
{
    unsigned long long int sum = 0;

    while (r) sum += r --;

    return sum;
}
```

以后，如果哪个程序里要用到这三个函数，哪怕是用到其中的一个，就不用再重新编写，直接调用即可。还是以打印从 1 加到 1000 的结果为例，我们应该怎么做呢？首先，要创建一个主程序，它包含了 main 函数，假定该源文件的名字叫 c0603.c，其内容如下：

```
/******************c0603.c******************/
int write_string (char *);
char * ull_to_string (unsigned long long int, char *);
unsigned long long int cusum (unsigned long long);

int main (void)
{
    write_string ("1+2+3+...+1000=");

    char a [20];

    write_string (ull_to_string (cusum (1000), a));
    write_string ("\n");
}
```

在源文件 c0603.c 中，我们用到了 write_string、ull_to_string 和 cusum 这三个函数，它们是在另一个源文件中定义的，我们不需要在当前源文件里重新写一遍。但即使如此，也必须在当前源文件里有所声明。

不过，它们在当前源文件里的声明只是简单地描述了函数名、参数和返回类型，并没有函数体。在 C 语言里，带有函数体的函数声明称为函数定义。但是，如果函数的声明里没有函数体，则指示参数名字的标识符可以省略。因此，如上所示，这三个函数在源文件 c0603.c 中声明都省略了参数的名字，只有类型。注意，与函数定义不同，不带函数体的函数声明末尾是分号“;”。

之所以要在当前源文件里使用不带函数体的函数声明，是方便 C 实现在翻译阶段对参数的数量和类型进行检查。当我们调用一个函数时，C 实现要检查传入的实参是否与函数声明里的形参在数量和类型上一致。

在 C 语言里，一个 C 程序可以由多个源文件组成。在翻译时，C 实现将分别翻译这些源文件，并把它们链接到一起生成可执行程序。但是，这里的“分别翻译”是有讲究的，并不是你想象中的那么简单和直接。例如，你要是直接翻译源文件 c0603.c，将得到一大堆错

误提示：

```
% gcc c0603.c
/tmp/ccHf0HhT.o：在函数‘main’中：
c0603.c:(.text+0x1d)：对‘write_string’未定义的引用
c0603.c:(.text+0x31)：对‘cusum’未定义的引用
c0603.c:(.text+0x45)：对‘ull_to_string’未定义的引用
c0603.c:(.text+0x4d)：对‘write_string’未定义的引用
c0603.c:(.text+0x59)：对‘write_string’未定义的引用
collect2: error: ld returned 1 exit status
```

出现错误的原因很简单，我们并没有在源文件 c0603.c 中定义函数 write_string、ull_to_string 和 cusum，而只是简单地说明了它们的参数类型和返回类型。在这种情况下，对该源文件的翻译不可能也无法生成实际的代码。

实际上，C 实现对源文件的翻译分为两个大的阶段。第一个阶段是编译，对 C 语言源文件的内容进行语法分析，进而生成对应的机器指令。但是，对有些实体（比如函数）的解析是无法完成的，因为它们可能并不是在当前源文件内定义的。在这种情况下，C 实现将记下这些名字，以及它们的属性（比如函数的参数类型和返回类型），等到以后再说。因此，这个阶段的成果并不是最终的可执行文件，而是目标文件。

第二个阶段是链接，是将前面生成的目标文件链接起来，得到最终的可执行文件。在这个阶段，将要处理那些未决的符号，找到它们的定义。除此之外，还要链接一些与操作系统有关的代码，这些代码对于能否让生成的可执行程序在操作系统上运行至关重要。

说了这么多，现在，让我们演示一下如何编译源文件 c0603.c 和 iotool.c，并将生成的目标文件链接在一起得到可执行程序：

```
% gcc -c c0603.c
% gcc -c iotool.c -std=c11
% gcc c0603.o iotool.o -o c0603.out
% ./c0603.out
1+2+3+...+1000=500500
%
```

如上所示，为了将源文件编译成目标文件，需要使用-c 选项，而且，目标文件的默认后缀是.o。一旦为 C 实现指定了目标文件，则它将会把这些目标文件链接在一起，生成最终的可执行文件。

实际上，我们也可以直接把编译和链接合在一起，但前提是必须给出所有的源文件，就像这样：

```
% gcc c0603.c iotool.c -o c0603.out -std=c11
% ./c0603.out
1+2+3+...+1000=500500
%
```

6.4 库

如果一些函数特别有价值，别人都用得上，或者它包含了特殊的算法和秘密，则可以将它们打包成可复用的库。库是包含了机器代码的文件，但它不能单独执行，因为它只是一个功能包，并不具备成为可执行文件所必须的其他组成部分和相关信息。

要生成一个库，可以使用随 GCC 发行的 ar 程序。例如，要从目标文件 iotool.o 生成一个库，可以这样做：

```
% ar r libiotool.a iotool.o
%
```

这将从目标文件 iotool.o 中提取函数，并将它们加入库文件 libiotool.a。参数“r”的意思是在库中加入新的函数，如果已经有同名函数则覆盖它。库的名字可以随意指定，但建议用“lib”引导，并以“.a”为后缀（扩展名）。

用 ar 程序生成的、以“.a”为后缀的文件是静态库。在翻译过程的链接阶段，静态库的内容会被提取出来，成为可执行文件的一部分。此后，可执行文件就和库无关了，当程序执行时，也不再需要静态库。以源文件 c0603.c 为例，将静态库 libiotool.a 链接到可执行文件的过程是这样的：

```
% gcc c0603.c libiotool.a -o c0603.out
% ./c0603.out
1+2+3+...+1000=500500
%
```

和静态库不同，LINUX 里的另一种库是共享库。共享库在链接时，仅在可执行文件中放一个存根，并不将库中的代码链接到可执行文件中。在这种情况下，生成的可执行文件和动态库有依赖关系，程序执行时，根据需要，可能会加载和执行共享库中的代码。使用共享库不但可以减小可执行文件的体积，也有利于代码的升级。当共享库发行更新的版本时，可以直接替换旧的共享库，而不影响可执行文件；当可执行文件调用共享库的代码时，执行的是升级后的代码。以下，我们演示了如何生成共享库，并将它链接到用源文件 c0703.c 所生成的可执行文件中：

```
% gcc -shared -o libiotool.so iotool.o
% gcc c0603.c libiotool.so -o c0603.out
% export LD_LIBRARY_PATH=./
% ./c0603.out
1+2+3+...+1000=500500
%
```

以上，第一行命令用于生成共享库，这是靠选项“-shared”来指示的。因为是要生成共享库，故“-o”选项应当给出共享库的名字，建议用“lib”作为前缀，并以“.so”作为后缀（扩展名）。共享库是用 iotool.o 的内容创建的，所以要在命令中给出这个文件的名字。

第二行命令是用刚刚生成的共享库和源文件 c0603.c 一起生成可执行文件。这个步骤和前面一样，没有什么好说的。

由于可执行程序运行时需要共享库的支持，所以必须将共享库的位置加入库的搜索路径。不然的话，将无法执行程序并提示找不到共享库 libiotool.so。由于我们的这个共享库位于当前目录中，所以我们用 export 命令将当前目录“./”加入库搜索路径的环境变量 LD_LIBRARY_PATH 中。

环境变量是由操作系统维护的一些名字，比如这里的 LD_LIBRARY_PATH，通过名字可以获得与它关联的文本串。这些“名字－文本”的组合为所有程序的运行提供操作系统范围内的环境信息，比如操作系统的安装路径，等等。如果一个应用程序想知道操作系统安装在哪个目录下，它就可以通过相应的环境变量获得。

6.5 头文件、预处理和翻译单元

以上，我们演示了库的创建，以及如何在我们的程序中链接库文件。在这个过程中非常重要的是，如果想要使用库里面的函数，就必须先在自己的程序里做不带函数体的声明。在

链接阶段，C 实现自然会从库中找到这些函数的实现（函数代码）。正是因为源文件 c0603.c 中已经有库中那三个函数的声明，所以我们可以顺利地编译和链接。

然而，库的内容不是工人可读的，当别人拿到这个库后，可能并不知道怎么使用。在程序设计阶段，他们可能并不知道函数的名字，以及参数的类型和函数的返回类型；就算他们知道，也必须在使用之前手工声明一下。如果函数很多，这就是一种负担。

为此，库在对外发行的时候，应该同时发行一个头文件。类似于 C 源文件，头文件也是一个文本文件，其最核心的内容是一些声明，比如对库文件里的函数进行声明。按照约定，头文件的后缀（扩展名）是“.h”。因此，如果要对外发布库文件 libiotool.a，则我们还应当同时发布一个头文件。头文件的名字随意，不必非得和库文件保持一致。对于我们前面生成的库文件 libiotool.a 或者 libiotool.so，头文件可以是 iotool.h，其内容如下：

```
/****************iotool.h****************/
int write_string (char *);
char * ull_to_string (unsigned long long int, char *);
unsigned long long int cusum (unsigned long long);
```

一旦拿到了上述库文件和头文件，则程序员的编程工作会变得相对简单。一方面，他可以直接调用库里面的函数而不用自己编写；另一方面，在调用库里面的函数之前，他不需要自己声明这些函数，只需要将头文件的内容包含进来即可，这称为“文件包含”。作为一个示例，下面的程序通过包含头文件 iotool.h 来使用库中的函数。

```
/*********************c0604.c*******************/
# include "iotool.h"

# define CHAR_BUF_LEN 20
# define MAX_SUM_NUM 1000
# define FIXED_SUFFIX(x) "1+2+3+...+" #x "="
# define PRN_FIXED_SUFFIX(x) FIXED_SUFFIX(x)

int main (void)
{
    write_string (PRN_FIXED_SUFFIX(MAX_SUM_NUM));

    char a [CHAR_BUF_LEN];
    write_string (ull_to_string (cusum (MAX_SUM_NUM), a));
    write_string ("\n");
}
```

在上述程序中出现了很多新的东西，它们都以“#”打头，称为预处理指令，这是我们马上就要在下一节里讲到的内容，现在，我们先将该程序保存为源文件 c0604.c，然后编译和链接到静态库文件 libiotool.a 以生成可执行程序，看看它能不能工作再说：

```
% gcc c0604.c libiotool.a -oc0604.out
% ./c0604.out
1+2+3+...+1000=500500
%
```

从最终的程序运行结果来看，一切都很正常，什么问题都没有。现在，我们来看看这些新加入的都是什么东西。

前面说了，C 语言程序的翻译过程分为两个大的阶段：编译和链接。实际上，在正式开始编译工作之前，还需要对源文件预先处理一下，称为预处理。预处理的主要工作是文件包含和宏的展开。

在源文件 c0604.c 里使用了好几个函数但并没有声明，不过头文件 iotool.h 中已经存在这些声明，所以，直接将该头文件的内容包含进源文件 c0604.c 中就可以了。要做到这一点，只需要在使用这些函数之前加入一条以“# include”开始的源文件包含指令。

预处理指令都以“#”打头，这样就可以将它们与不需要或者不参与预处理的其他文本区分开来。以“# include”打头的预处理指令是源文件包含指令，后面跟一个用尖括号“<>”或者双引号“""”围起来的文件名。在预处理期间，C 实现将用该文件的内容来取代这条预处理指令。也就是说，文件的内容将会被插入到该预处理指令所在的位置。

注意，以“# include”打头的预处理指令是源文件包含指令，什么文件都可以包含进来，并不仅仅限于头文件。值得注意的是，如果“#”用于引导一个预处理指令，则它必须是所在行的第一个字符。换句话说，每个预处理器指令都必须独立成行。

回过头去看源文件 c0604.c，因为是要计算从 1 加到 1000 和结果，所以“1000”出现了两次，第一次是在字面串"1+2+3+...+1000="里，第二次是作为函数调用的参数，也就是 cusum (1000)。下次，如果你想计算从 1 加到 3000 的结果，还得分别去改这两个地方。

这当然不太麻烦，但对于规模较大的软件项目来说，可能需要修改很多地方，甚至经常会跨很多源文件来查找和修改，如果不小心漏掉其中一个，这可能会造成大麻烦。为了避免类似的情况发生，就需要另外一种预处理指令：宏定义。

以“# define”打头的预处理指令是宏定义。宏定义分为两种，第一种是变量式宏定义，取这个名字是因为它有点像在定义一个 C 语言里的变量。“define”后面的标识符被称为宏名，宏名后面的内容（不包括前后的空格和末尾的换行符）称为替换列表。在上述程序中，我们首先定义了这样一个宏：

```
# define CHAR_BUF_LEN 20
```

这里，标识符“CHAR_BUF_LEN”是宏名，它是替换列表“20”的另一个“有意义好识别”的写法。不要求宏名必须使用大写，但在业内有这样的习惯。当这个标识符出现在后面的程序文本中时，C 实现将用对应的替换列表来把它替换掉。例如在程序的后面，这个标识符被用在数组的声明中：

```
char a [CHAR_BUF_LEN];
```

在程序翻译期间，C 实现将首先把这个数组声明里的“CHAR_BUF_LEN”替换为宏定义里的数字，也就是变成这样：

```
char a [20];
```

使用宏的好处是，如果我们想把数组变大，就可以直接修改那个宏定义。因为它位于程序的开头，所以比较方便。

在我们的程序中，另一个变量式宏定义是：

```
# define MAX_SUM_NUM 1000
```

这个宏定义了最大要累加的数，在我们的程序中，它被用在以下程序文本中以执行宏替换：

```
write_string (ull_to_string (cusum (MAX_SUM_NUM), a));
```

在程序翻译期间，这一行里的宏将被替换，因此该语句会被扩展为：

```
write_string (ull_to_string (cusum (1000), a));
```

除此之外，在程序里还有一个地方需要替换，那就是字面串"1+2+3+...+1000="里的 1000。这就需要把宏名 MAX_SUM_NUM 嵌进这个字面串里，得到一个可以随宏定义变化

的字面串。该怎么做呢？难道是这样：

```
write_string ("1+2+3+...+MAX_SUM_NUM=");
```

这绝对是行不通的，字符常量和字面串里的内容都会被原样保留，除非是脱转序列，如果这样能行，那就乱套了。幸运的是，这种问题总会有办法解决的，但在此之前，先要了解另一种宏定义——函数式宏定义。

顾名思义，函数式宏定义在外观上看起来像是一个函数，因为它可以带有参数，而它的使用也很像函数调用。下面是一个函数式宏定义的例子：

```
# define ADD(a,b) (a) + (b)
```

在这里，ADD 是宏名，a 和 b 是宏的参数，参数之间用逗号“,”分隔。此宏一旦定义，则宏调用 ADD(10,20)被扩展为(10) + (20)，而宏调用 ADD(x,50)则被扩展为(x) + (50)。和函数的定义和调用一样，为明确所指，宏定义里的参数称为形参，宏调用里的参数称为实参。

注意，宏定义里的宏名和组成参数列表的括号之间不能有空白，否则它被视为一个变量式宏定义，宏名之后的部分都被视为替换列表。这就是说，如果宏定义是这样的：

```
# define ADD (a,b) (a) + (b)
```

则宏调用 ADD(30,60)将被扩展为(a,b) (a) + (b)(30,60)。显然，被扩展的仅仅是宏名 ADD。

在函数式宏定义中，如果替换列表中有“#”，且它后面紧跟着当前宏的形参，则它们在预处理期间被转换为一个字面串，且该字面串是由与该形参对应的实参转化而来，这称为记号串化。这就是说，给定以下宏定义：

```
# define STR(x) #x
```

则宏调用 STR(hello,world.)会被扩展为字面串"hello,world."，这个字面串在后续的编译和链接期间作为普通的字面串处理。

在函数式宏定义中，如果替换列表中有“##”，且它位于两个记号的中间，则这两个记号会被合并为一个记号；如果这两个记号是宏的形参，则将会把它们对应（传入）的实参合并为一个记号，这称为记号黏接。这就是说，给定以下宏定义：

```
# define MAKEIDEN(x, y, z) x##y##z
```

则宏调用：

```
char MAKEIDEN(a,b,c);
```

会被扩展为：

```
char abc;
```

在后续的编译和链接期间，这个扩展的结果会被当成普通的声明一样处理，即，声明一个 char 类型的变量 abc。

回到我们的程序中来，由于出现在函数式宏定义中的参数才能被转化为字面串，所以我们使用了如下宏指令：

```
# define MAX_SUM_NUM 1000
# define FIXED_SUFFIX(x) "1+2+3+...+" #x "="
```

在预处理阶段，对宏调用的处理是先对实参进行宏扩展，然后再对宏调用本身进行扩展。因此，在我们的预期中，以下宏调用：

```
write_string (FIXED_SUFFIX(MAX_SUM_NUM));
```

将先把 MAX_SUM_NUM 扩展为 1000，接着再扩展宏调用 FIXED_SUFFIX(1000)，最终得到一个粘接的字面串：

```
"1+2+3+...+" "1000" "="
```

这样，根据我们在上一章里所了解到的，这三个字面串将在正式编译期间合并为一个完

整的字面串"1+2+3+...+1000="，然后再用于初始化一个不可见的数组。

然而，在函数式宏定义的替换列表（宏体）中，如果形参的前面是“#”或者“##”，又或者后面是“##”，则调用此宏时，传入的实参并不扩展。因此，上述宏调用实际上会被扩展为：

```
"1+2+3+...+" "MAX_SUM_NUM" "="
```

对此，比较标准的解决方法是实施迂回战术，先定义一个替换列表中没有“#”或者“##”的宏，这样就可以将扩展后的实参传进来，然后再将扩展后的实参传递给前面所定义的那个宏 FIXED_SUFFIX：

```
# define PRN_FIXED_SUFFIX(x) FIXED_SUFFIX(x)
```

我们说过，在正式编译一个 C 程序之前，要先进行预处理。预处理之后，所有的预处理指令都被删除，源文件经过预处理之后就得到了翻译单元。换句话说，对源文件进行预处理的成果是生成了翻译单元。

在 C 实现将一个源文件翻译成可执行文件的整个过程里，如果没有特别的指示，翻译单元只是翻译过程早期的一个临时文件，用于后续的编译过程。而且用完就删，你也没有机会看到它的内容。

为了查看预处理之后的结果，也就是翻译单元的内容，可以要求 C 实现在预处理之后立即停止翻译过程并显示预处理的结果：

```
% gcc -E c0604.c
# 1 "c0604.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "c0604.c"
# 1 "iotool.h" 1
int write_string (char *);
char * ull_to_string (unsigned long long int, char *);
unsigned long long int cusum (unsigned long long int);
# 2 "c0604.c" 2


int main (void)
{
    write_string ("1+2+3+...+" "1000" "=");

    char a [20];

    write_string (ull_to_string (cusum (1000), a));
    write_string ("\n");
}
%
```

这里，选项“-E”的意思是在预处理之后立即停止翻译过程，不进行后续的编译过程。在输出的内容里有一些行标记信息，我们用斜体和浅色显示，以免影响我们阅读真正感兴趣的内容。如果要禁止生成行标记，可以使用“-P”选项。如果要想将预处理的结果保存为

文件，可以用“-o”选项指定一个文件名。

无论如何，经过预处理之后，在生成的翻译单元里将不再包含预处理指令，头文件的内容也已经包含进来，而所有的宏也已经被展开，整个翻译单元里仅剩下声明和函数定义。

练习 6.1：

给定宏定义

```
# define pst (x, y, z) x##y##z
```

则宏调用 pst(_,int,_)扩展之后的结果是什么？如果想要得到_int_，则应当如何修改宏定义？

6.6 UNIX 和类 UNIX 函数库

20 世纪 70 年代初，Brian Kernighan 和 Dennis Ritchie 发明了 C 语言。作为与操作系统的接口，最初的函数库也得以创建。最初，发明 C 语言是为了重写 UNIX 操作系统，而在之后的一段时间里，UNIX 也是 C 程序设计的主要平台。由于这种事实上的近亲关系，UNIX 系统调用当然要用 C 库进行封装以方便使用。

随着 UNIX 逐渐变得流行，它也发展出两个大的分支或者说阵营，一个是 SYSTEM V，另一个是 BSD，而且它们下面还有各种细小的分支。版本之间的差异严重影响了设备之间的互操作性和程序的可移植性，假如 SYSTEM V 里有一个系统调用而 BSD 里没有，程序员该怎么办呢？如果用的话，他的程序将不能在 BSD 系统上工作。在这种情况下，在业界有影响力的大公司和大客户要求对 UNIX 各版本之间的差异加以规范，并对函数库的开发和使用做出标准化的规定。

当然，这是比较正式的说法，要是用有些网友的话来说那就是“UNIX 系统早期发展得太快，以 SYS V 为首的建制派和 BSD 为首的学院派各自搞了很多新玩意儿，相互之间竞争激烈，不兼容之处越来越多，各个商业厂家也首鼠两端，无所适从。于是就有好事者出来一统江湖，把各个山头叫来坐下谈谈”。

协商的最大成果是，由国际电气和电子工程师协会 IEEE 牵头搞了一个标准，名字叫 POSIX，用中国话来说就是“可移植性操作系统接口”。POSIX 在各个方面对 UNIX 操作系统和各个分支加以规范和认证，其中就包括 C 语言的函数库。

统一后的 UNIX 编程接口按照功能进行归类，并以头文件的形式发布。库文件和库代码当然重要，但更重要的是先决定应该有哪些库函数，并把函数的名字和参数统一起来。

至于库函数的具体实现，有些库函数是 UNIX 系统调用的封装，而有些库函数则不依赖于系统调用而独立存在，例如计算字符串长度，这种功能不需要系统调用就可以用 C 语言或者汇编语言实现。

POSIX 标准的库函数按照它们的功能归类，被划分为几十个头文件，要完整地进行介绍不是这本书的厚度所能够做到的，所以我们不妨仅以磁盘文件的读写和屏幕打印为例进行介绍，而这些也正是我们比较关心的。

要读写一个磁盘文件，首先必须创建或者打开它，而创建或者打开一个磁盘文件需要使用 POSIX 库函数 open，它的原型是这样的：

```
int open (const char * pathname, int oflag, ...);
```

6.6.1 限定的类型

函数 open 的第一个参数用于指定要创建或者打开的文件（名），故其类型是指向 char 的指针，用于指向一个字符串。但事实上，该参数的类型并不是指向 char 的指针，而是指向 const char 的指针，这是啥意思呢？

C 语言里的类型，从所描述的实体来分，可分为变量类型与函数类型，而变量类型又拥

有它们的限定版本。也就是说，它们可以用一些关键字如“const”等加以限定，从而形成各自的限定版本，而这些关键字则称为类型限定符。例如，const int 是 int 类型的限定版本；const char 是 char 类型的限定版本。

用类型限定符“const”限定的那些类型是“const 限定的类型”。有些英语基础的人知道这个词代表着“常量”，但在这里并非如此，“常量”一词在 C 语言里有特定的含义，但和“const”无关，可惜很多人和很多书都把它弄错了。“const”的含义和“只读”很接近，即只用于读出而禁止写入，所以，它应该叫“readonly”而不是“const”。用这种类型声明的变量，它的值仅用于读出，而不用于写入和更新操作，下面是一个例子：

```
const unsigned int cx = 0;
```

这就声明一个 const unsigned int 类型的变量 cx，并初始化为 0。如果一个变量的类型是 const 限定的，这种限定不影响它的初始化，但不允许写入操作。因此，下面的语句是非法的，将导致它所在的程序无法通过编译：

```
cx = 65533;
```

然而，如果仅仅是为了限制写入一个变量，则发明关键字“const”是没有任何实际用处的，因为我们可以直接使用常量，比如整型常量和字符常量；即使担心同一个常量被多处使用而难以维护，也可以通过宏定义来解决。实际上，它真正的用处和目的是对程序的翻译进行优化。

我们知道，处理器内部的寄存器速度最快，外部存储器的速度则慢得多。如果一个变量仅用于读出（而不是写入），则它的值只读一次即可，不必在每次用到变量的值时，都真的执行一次存储器读出操作。换句话说，它只需要在开始的时候读一次，然后在寄存器里缓存这个值即可。

但是，C 实现需要程序员给出一个字面上的保证，以表明自己不会写入某个变量，这就是关键字“const”。如果一个变量具有 const 限定的类型，则意味着程序员不准备写入这个变量，而只是读它的值，因此，C 实现可以尽可能地多做一些优化。比如，它可以在第一次访问变量的同时将它的值缓存起来，以后只需要使用这个缓存值而不需要在读取操作上浪费时间。

指针所指向的类型和数组类型也可以是限定的类型。因为数组的类型其实就是其元素的类型，所以，声明一个限定类型的数组也就意味着它的元素也具有相同的限定。在下面的示例中，数组 ca 是“具有 2 个 const int 类型的元素”的数组，而指针 pc 则是“指向 const int 类型的变量”的指针。

```
const int ca [2];
const int * pc = ca;
```

如图 6-4 所示，和往常一样，对 pc 的声明应该从标识符开始，依次往左读为“变量 pc 的类型是指针，该指针指向 const int”，或者说“变量 pc 的类型是指向 const int 的指针”。

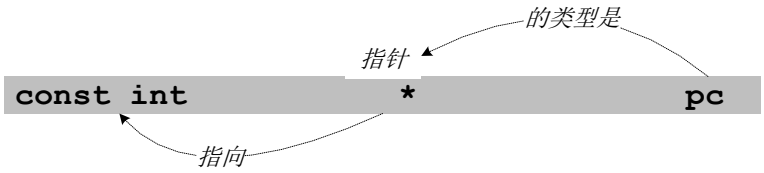


Fig.6-4 声明一个指向限定类型的指针（变量）

在变量 pc 的声明中，作为初始化器，数组 ca 转换为指向其首元素的指针。我们已经学过数组－指针转换，元素类型为 *T* 的数组将自动转换为指向 *T* 的指针并指向数组的首元素，在这里，因数组 ca 的元素类型为 const int，故转换后的类型为“指向 const char

的指针”，也即 const char *，可以用于初始化 const int *类型的变量 pc。

因为数组 ca 的元素类型是限定的，而变量 pc 的值所指向的类型也是限定的，如果用以下两条语句来写入数组的第一个元素（* pc）和第二个元素（ca [1]）则是非法的，会在程序的翻译期间产生诊断信息：

```
* pc = 77;
ca [1] = 103;
```

这里，左值 pc 的类型是 const char *，经左值转换得到同类型的指针（值），故表达式* pc 也是一个左值，类型为 const char，指示一个 const char 类型的变量（实际上是数组 ca 的元素），该变量不可写；表达式 ca [1]访问下标为 1 的元素，但那个元素的类型是 const int，不允许写入。

使用 const 限定的类型来声明变量能够增强程序的可读性，并在一定程度上保护变量的值不被破坏。然而只要耍点花招，这种保护实际上并非牢不可破，同时非常危险。给定以下声明：

```
const int c = 0;
```

因为变量 c 的类型是 const int，故下面的语句是非法的：

```
c = 1;
```

但是，我们可以用一元&运算符得到一个指向变量 c 的指针，也就是指向 const int 类型的指针，然后，通过转型表达式来得到一个指向 int 的指针，并通过这个指针来间接修改变量 c 的值：

```
* (int *) & c = 10086;
```

这里，表达式& c 的结果是指向 const int 的指针（const int *）；转型表达式(int *) & c 把这个指针强制转换为指向 int 的指针（int *），然后，运算符*作用于这个指针类型的值，得到一个左值，实际上代表变量 c。

这样做在语法层面上没有问题，它所在的程序可以顺利通过翻译。但是在现实里，一旦某个变量被声明为 const 限定的类型，则它在运行时有可能被加载到一个特殊的存储区域，虽然这个区域在本质上是可读可写的（就存储器的物理性质而言），但处理器和操作系统会对这个区域的写入操作进行检查并加以阻止。在这种情况下，任何企图写入这个区域的操作都是未定义的行为，都将引发不可预料的后果，严重的将导致程序崩溃。

练习 6.2：

对于上述变量 c，既然通过类型转换就可以突破 const 的限制写入变量，那为什么

```
(int) c = 10086;
```

不可以呢？

引入 const 限定符并不是要建立一种攻防关系，也不是向程序员的智商宣战。所以，请不要做对工作、对程序来说没有意义的事。

如果一个变量的类型是指针，那么，不但它所指向的类型可以是限定的，而且这个指针本身也可以是限定的。在图 6-5 中，声明了一个变量 cpc，虽然第二个 const 离标识符 cpc 很近，但却不应该读成“const 限定的 cpc”。

```
int c = 0, d = 0;
const int * const cpc = & c;
```

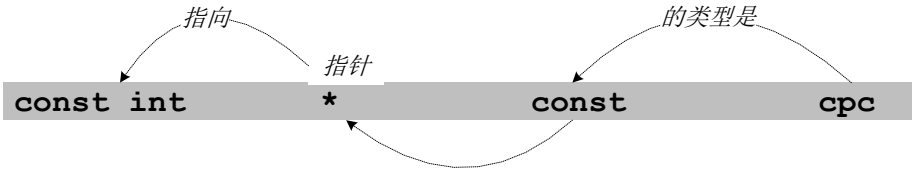


Fig.6-5 声明一个 const 限定的指针

实际上，如图 6-5 所示，这个声明应该从标识符 cpc 开始，依次向左读为“cpc 的类型是 const 指针，（该指针）指向 const int”，或者说“变量 cpc 是一个指向 const int 类型的 const 指针”。我们已经学过了类型名，所以，变量 cpc 的类型是 const int * const，即，指向 const int 的 const 指针变量。

像 cpc 这样的变量，不但不允许通过它写入它的值所指向的那个变量，而且也不允许修改它本身的值。换句话说，变量 cpc 只能在声明的时候初始化为指向某个变量，此后便不允许再指向别的变量，也不允许修改它的值所指向的变量。因此，下面的两条语句都是非法的：

```
cpc = & d;
* cpc = 12345;
```

函数的形参也可以被声明为限定的类型。在下面的程序中，函数 preld 有两个参数 cpc 和 ci。形参 cpc 是 const 限定的指针变量，形参 ci 是 const 限定的整型变量。有人可能会说，它们都是只读的，不可以赋值，而参数传递类似于赋值，应该也不允许。

```
/*************c0605.c************/
int preld (int * const cpc, const int ci)
{
    return * cpc + ci;
}

int main (void)
{
    int c = 1, d = preld (& c, c);
}
```

事实上，它就是可以。C 语言规定，如果函数的形参被声明为限定的类型，则当函数被调用时，它们被创建为具有相同限定类型的形参变量。但是，该变量在接受调用者传递的（实参）值时，被视为具有无限定的类型。

但是，因为 cpc 和 ci 都是限定类型的变量，在函数内部不能修改它们的值，这可以为优化带来好处。

练习 6.3：

1.若变量 ac 是具有 5 个元素的数组，且元素类型为指向 const char 的指针，请写出它的声明。

2.若变量 ca 是具有 5 个元素的数组，且元素类型为指向 const char 的 const 指针，请写出它的声明。

3.若变量 p 是一个 const 指针，指向一个 5 元素的数组，数组的元素类型为指向 const char 的指针，请写出 p 的声明。进一步地，如果元素类型为指向 const char 的 const 指针呢？

6.6.2 变参函数

回到原来的话题，继续讨论 open 函数，它的第二个参数用于指定文件打开的方式，比如是创建文件呢，还是纯粹打开文件；如果是创建文件，则文件已经存在怎么办；打开文件的目的是只用来读呢，还是又读又写，等等，这个参数的类型是 int。

仔细观察，函数 open 的声明和我们学习过函数不同，它的末尾是省略号“...”，意味着后面还可以有其他更多的参数。这样的函数，它们的参数在类型和数量上都不确定，称为可变参数的函数，或者变参函数。看样子，在继续讨论 open 函数之前，我们应该先来认识一下变参函数。变参函数必须至少有一个类型确定的参数，而且必须是最开始的参数，后面的参数无法确定。因为这个原因，它的参数类型列表必须以“, ...”结尾。

下面是一个使用变参函数的例子，但它有一定的特殊性，那就是只能在某些特定的计算机上正确执行，比如基于 32 位 INTEL x86 处理器的计算机系统。至于原因，我们后面将会讲到，而且还将提供一个安全、通用的解决方案。

```
/*****************c0606.c*****************/
long long int sum_ints (unsigned int count, ...)
{
    int * var_start = (int *) & count + 1;
    long long int sum = 0;

    while (count --) sum += * var_start ++;

    return sum;
}

int main (void)
{
    long long int x, y, z;

    x = sum_ints (2, 100, 200);
    y = sum_ints (0);
    z = sum_ints (5, 10, -10, 30, 600, -300);
}
```

在这个程序中，sum_ints 是变参函数，用于累加传入的整数，但传入的整数到底会有多少个，这是不确定的，所以该函数的参数类型列表以“, ...”结束。话虽这么说，但 C 语言规定每个函数的参数至少保证能有 127 个。

对于函数的调用者来说，要传入什么参数，传入多少个，这当然是非常清楚明确的。例如在 main 函数中，语句

```
z = sum_ints (2, 100, 200);
```

就清楚地指明了参数的数量（第一个参数的后面还有 2 个参数）和类型（都是 int 类型）。但是，在被调用函数的内部，该如何获知参数的数量和各自的类型呢？

这就需要一个约定。我们说过，变参函数的第一个参数或者前几个参数必须是类型确定的参数，之所以这么规定，就是因为可以通过这些参数来暗示或者明确指定后续的参数有几个、什么类型。具体到 sum_ints 函数，它的第一个参数是固定的，用于指定后面的参数有几个。当然，它没有指明参数的类型，实际上并不需要，因为这原本就是一个例子，只用来处理 int 类型的数据。

如何传递参数，参数存放在什么地方，如何在函数内部获得这些传入的参数，和 C 语言无关，这是由 C 实现自主决定的部分。不同的 C 实现工作在各自的软硬件平台上，有不同的处理器架构，运行着不同的操作系统，C 实现自会根据处理器和操作系统所提供的现实条件来采取不同的参数传递方案，反正只要达到目的，实现 C 语言函数调用的语义就行。

在基于 32 位 INTEL x86 处理器的计算机系统上，如图 6-6 所示，C 实现会确保变参函数的参数是按顺序传递并集中保存在一个特定的存储区里，函数的调用者把需要传递的参数存放在这里（创建形参所指示的变量并写入实参的值），而函数也从这里取得参数的值（访问参数变量）。因此，只要获得了第一个参数 count 的地址，就可以顺藤摸瓜，一个一个地找到其他参数。

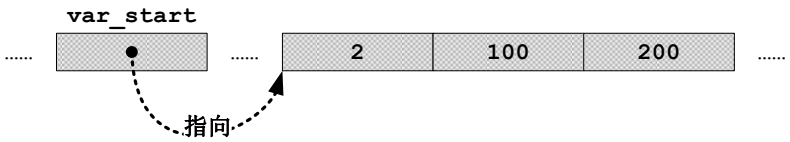


Fig.6-6 参数变量的存放位置

所以，我们做的第一件事，就是生成一个指向变量 count 的指针，并将这个指针移动到变量 count 之后，使之指向第一个可变参数（值为 100 的参数变量），然后用来初始化变量 var_start：

```
int * var_start = (int *) & count + 1;
```

先来看初始化器(int *) & count + 1，一元&运算符的优先级最高，转型运算符次之，加性运算符+的优先级最低。表达式& count 是生成一个指针，指向值为 2 的参数变量。按照计划，将这个指针加 1，就得到一个新指针，指向第一个参数变量（值为 100 的参数变量）。

指针的类型决定了你能从它所指向的变量里读取什么样的值，表达式& count 的类型是指向 unsigned int 的指针，但我们需要一个指向 int 的指针，因为后面的变参都是 int 类型。为此，需要用转型表达式将表达式& count 的值从指向 unsigned int 类型的指针转换为指向 int 的指针。

我们可以把所有参数变量所在的存储区域看成一个数组，那么，表达式(int *) & count + 1 的结果是一个新的指针，这个指针指向下一个数组元素。即，指向值为 100 的参数变量。C 语言规定，任何一种整数类型的有符号和无符号版本占用相同的存储空间。所以，即使表达式& count 的类型是 unsigned int *，但它转换为 int *类型后加 1 同样会使得相加的结果指向第一个可变参数。

由于是用第一个参数 count 来指定后面还有几个参数，我们就可以使用 while 语句来逐一取得这些参数，同时递减变量 count 的值，等它为 0 时结束循环：

```
while (count --) sum += * var_start ++;
```

首先，变量 count 的类型是 unsigned int，它的值不可能为负；如果传递给它的值是 0，则表达式 count --的值是变量 count 递减前的原值（0），从而使循环一次都不进行。

另一方面，依据各运算符的优先级别，表达式 sum += * var_start ++等价于 sum += (* (var_start ++))，这就取得变量 var_start 的值所指向的参数变量的值，并加到变量 sum。同时，变量 var_start 的值递增，指向下一个参数变量。

在 main 函数里，我们声明了三个变量 x、y 和 z，然后用三个函数调用表达式为它们赋值。赋值后，变量 x、y 和 z 的值分别为 300、0 和-330。为了验证，我们将上述程序保存为源文件 c0606.c，然后用-g 选项翻译为可执行文件并上机调试——且慢！这并不是

一件容易的事！

在没有认识变参函数之前，我们从来就是直接访问形参，而不需要关心如何寻找和定位它们，C 实现可以根据程序翻译和运行的平台做内部处理，并隐藏所有细节。相反，对于可变参数，我们需要自己手工编码来寻找和访问它们。

问题在于，翻译一个程序、运行一个程序都是在特定的操作系统上完成的，而且还基于不同的处理器，并由此形成了不同的计算机系统。在不同的计算机系统上，参数传递的方法不同，定位这些参数变量的方法也不相同。

换句话说，源文件 c0606.c 并不能在所有计算机上正确执行（得到正确的结果），但它至少可以在基于 32 位 INTEL x86 处理器的 LINUX 和 WINDOWS 上正确执行。假定我们已经翻译并生成了可执行文件，则其调试步骤如下（以 WINDOWS 平台为例）。

```
(gdb) b 19
Breakpoint 1 at 0x4016f8: file c0606.c, line 19.
(gdb) r
Starting program: D:\exampls\c0606.exe
[New Thread 2548.0xb1c]

Breakpoint 1, main () at c0606.c:19
19      }
(gdb) p {x, y, z}
$1 = {300, 0, 330}
(gdb)
```

如果条件允许的话，你可以自己动手实验一下，要是结果不如预期也没有关系，很可能不是你的问题，是你的计算机系统不符合要求。首先，不是所有的计算机都通过存储器传递参数。比如说，如果你的计算机使用了 ARM 处理器或者 64 位的 INTEL x86 处理器，按照约定，参数将尽可能地通过寄存器传递，当参数过多时，余下的部分才通过存储器传递。在这种情况下，上面的方法就行不通了。

即使能够保证所有参数都通过存储器传送，那也会遇到一个同样严重的问题：两个相邻的参数变量未必是紧挨在一起的。因为对齐的原因，不同类型的参数变量必须位于特定的内存地址，比如能够被 4 整除的地址上。如此一来，在两个相邻的参数变量之间就必然存在“空隙”。在这种情况之下，将指针从前一个变量移到后一个变量，不单单要考虑变量的类型和大小，还必须考虑变量在特定计算机系统上的对齐要求。

就我们的这个程序来说，它之所以能够在多数 32 位的计算机上运行，无非就是参数都通过存储器传送，而且我们传递的参数是 int 和 unsigned int 类型，这两种类型在 32 位的计算机上都是按 4 字节对齐的，相邻的变量之间没有间隙。

所以，处理变参函数没有统一的方法，必须针对不同的计算机系统做具体分析，并编写不同的代码，甚至还要借助于汇编语言。

那么，有没有人能够出面考察所有不同的计算机系统，并写出一个函数库，让我们不用考虑计算机系统之间的差异就能方便地处理可变参数呢？有的，符合 POSIX 标准的函数库都包含了一个头文件 stdarg.h，里面提供了几样宝贝，能够让我们的变参函数适应不同的计算机系统（它们之间的本质差异主要体现在处理器架构和操作系统的不同上）。以下是上面那个程序的改良版本，就是用这个头文件来改善了可移植性。

```
/*****************c0607.c***************/
# include <stdarg.h>
```

```
typedef long long int VARF (unsigned int, ...);

int main (void)
{
    long long int x, y, z, m, n;

    VARF sum_ints;
    x = sum_ints (2, 100, 200);
    y = sum_ints (0);
    z = sum_ints (5, 10, -10, 30, 600, -300);
}

long long int sum_ints (unsigned int count, ...)
{
    long long int sum = 0;

    va_list ap;
    va_start(ap, count);
    while (count --) sum += va_arg(ap, int);
    va_end(ap);

    return sum;
}
```

注意，在这个程序里，源文件包含指令# include 的用法和前面不一样。在本章一开始，为了在源文件里包含头文件 iotool.h，我们是这样写的：

```
# include "iotool.h"
```

注意，这里用的是双引号，而当前程序中用的是尖括号，这有什么区别吗？有的。预处理指令# include 指明了要包含的文件，预处理器需要知道文件在哪里。安装 C 实现是为了编程，编程又需要输入输出，输入输出又需要函数库，函数库是与具体的计算机系统密切相关的。为此，每个 C 实现都会提供一些流行的函数库，有针对 UNIX 和类 UNIX 的 POSIX 函数库，有针对 WINDOWS 的函数库，等等。这些库都配套了头文件，为此，每个 C 实现在安装后都会划出特定的位置（目录或者文件夹）来保存这些配套的头文件和源文件，这是一个由 C 实现定义的位置。

如果预处理指令# include 中的名字是用尖括号围起来的，C 实现将到这个实现定义的位置去寻找它；如果是用双引号围起来的，则 C 实现会用别的办法来寻找它，典型的就是先在当前目录下寻找。如果找不到，就把它当成由尖括号围起来的文件名处理。

在往后的行文中，每当我们提到某个头文件时，如果它是随 C 实现发行的头文件，则用尖括号围起来；否则，就用双引号围起来。

我们知道 C 语言具有非凡的可移植性，用它编写的程序，能够在不同的计算机系统上翻译和执行。但是，说白了，所谓的可移植性，就是在那种计算机系统上有可用的 C 实现，能够将程序翻译成符合那个计算机系统的可执行程序。所谓 C 语言具有很强的可移植性，无非就是有很多好事者为各种不同的计算机系统编写了大量的 C 实现。

C 实现也是程序，为一种计算机系统提供的 C 实现不能在另一种不同的计算机系统上工作。就像你不能把 WINDOWS 上的 Word 文字处理软件拿到 LINUX 上运行一样，为 WINDOWS

提供的 C 实现不能在 LINUX 上运行；GCC 的 LINUX 版本也不能直接拿到 WINDOWS 操作系统上运行；GCC 的 32 位 LINUX 版本也不能拿到 64 位的 LINUX 操作系统上开工。

所以，C 实现具有“定制性”的特征，你要安装 C 实现，第一步就是选择适合你当前计算机系统的那个版本。由于这个原因，不同的 C 实现就可以针对它所运行的计算机系统来定制一些内容。

6.6.2.1 类型定义

现在将注意力集中到 sum_ints 函数，它先是声明了一个 va_list 类型的变量 ap，用来保存指向参数的指针，相当于我们前面的 var_start。不同的计算机系统上有不同的 C 实现，而 va_list 的定义则随 C 实现的不同而有所变化。在基于 32 位 INTEL x86 处理器的计算机系统上，C 实现有可能将它定义为：

```
typedef char * va_list;
```

在这里，“typedef”是 C 语言里的关键字，它唯一的作用是类型定义，也就是将一个标识符定义为它被声明的那种类型。就当前这个例子来说，如果你将关键字 typedef 去掉，那么它将变成

```
char * va_list;
```

显然，这是在声明一个标识符 va_list，其类型为 char *。很好，如果加上关键字 typedef，则这是定义了一个新的类型，类型名为 va_list，它是 char *的别名。从此以后，我们就可以声明这个新类型的变量，或者用它作为函数的参数类型和返回类型：

```
va_list vla;
va_list func (va_list);
```

它们分别等价于

```
char * vla;
char * func (char *);
```

可以定义任何类型的别名。下面是另外一些类型定义的例子：

```
typedef int DWORD;
typedef DWORD dWord;
typedef char * PCHAR;
typedef int ARRAY [255];
typedef int FUNC (int, int);
```

以上，第一行是将标识符 DWORD 定义为 int 类型的别名；第二行是将标识符 dWord 定义为 DWORD 类型的别名，这实际上也是 int 类型的别名；第三行是将标识符 PCHAR 定义为 char *类型的别名；第四行是将标识符 ARRAY 定义为数组类型 int [255]的别名；第五行是将标识符 FUNC 定义为函数类型 int (int, int)的别名。下面，我们将用这些新的类型来声明变量和函数：

```
DWORD x;
ARRAY a, b [20];
FUNC f;
```

实际上，这些声明分别等价于：

```
int x;
int a [255], b [20] [255];
int f (int, int);
```

你可能觉得奇怪，为什么 ARRAY b [20]不是等价于 int b [255][20]，而是等价于 int b [20] [255]呢？要知道，类型定义并不是宏替换，b 的类型是“具有 20 个元

素的数组”，元素的类型是 ARRAY。

类型 va_list 是在头文件<stdarg.h>中定义的。在不同的计算机系统上，函数调用时的参数传递可能采取不同的方式，所以对 va_list 的定义可能并不相同。但是，你只需要在自己的程序里包含这个头文件，并声明 va_list 类型的变量，就可以在任何计算机系统上翻译并正常执行。

接下来，va_start 是一个宏，也是在头文件<stdarg.h>中定义的，用来使 ap 指向变参函数中最后一个已知参数（从右往左数的第一个已知参数），从而为访问后面的变参做准备。该宏具有两个参数，第一个参数是那个被声明为 va_list 类型的变量；第二个参数则是标识符，它必须是变参函数中最后那个已知参数的名字。

再往下看，while 语句的控制表达式和前面一样，都是用 count 来指示变参的数量，并通过递减来控制循环是否结束。

为了获得变参的值，这里使用了另一个宏 va_arg，它同样是在头文件<stdarg.h>中定义的。该宏具有两个参数，第一个参数是那个被声明为 va_list 类型的变量；第二个参数则是类型名。要想获得每一个变参，最重要的是知道它的类型。在程序翻译时，这个宏被扩展为一个表达式，所以它可以直接作为运算符+=的右操作数：

```
sum += va_arg(ap, int)
```

一开始，我们是令 va_list 类型的变量 ap 指向最后一个已知参数。此后，每调用一次 va_arg，都会使 ap 指向下一个参数（变参）并取得（计算出）它的值。

最后，宏 va_end 将修改变量 ap 使它不再可用，它同样是在头文件<stdarg.h>中定义的。如果在一个函数里曾经用 va_start 引用过变参，则它必须调用 va_end 才能保证返回时一切正常。如果一个函数在调用 va_end 之前没有调用过 va_start，或者在调用 va_start 之后没有调用过 va_end，则程序的行为是未定义的，后果不可预料。

现在回到程序的开头，那里定义了一个类型 VARF：

```
typedef long long int VARF (unsigned int, ...);
```

如果没有关键字 typedef，这是声明了一个函数 VARF；因为有关键字 typedef，这是将标识符 VARF 定义为它被声明的类型，即，将标识符 VARF 定义为函数类型 long long int (unsigned int, ...)。也就是说，VARF 现在是一种函数类型的名字（别名）。

我们知道，函数在调用之前必须声明。我们在 main 函数里调用了 sum_ints，但它的定义却位于 main 函数之后。为此，我们需要在调用 sum_ints 之前做一次不带函数体的声明。这个声明可以放在 main 函数之前，也可以放在 main 函数内部，但总的原则是必须放在它的调用点之前（我们选择在 main 函数内部声明）：

```
VARF sum_ints;
```

它实际上也就等价于：

```
long long int sum_ints (unsigned int, ...);
```

练习 6.4：

1.如果 F 是“指向‘有两个 int 类型的参数且返回类型是 int 的函数’的指针”的别名，请问 F 是如何定义的？编写一个程序，用类型 F 声明一个指向函数的指针，并用这个指针调用它所指向的函数。

2.在下面的程序里，函数 var_sum 是个变参函数。参数 fmt 用于接受一个字符串，字符串的内容用于指示对应的变参的类型。比如，“l”表示对应的变参是 long int 类型；“i”表示对应的变参是 int 类型；“I”表示对应的变参是指向 int 的指针；“L”表示对应的变参是指向 long int 的指针。

函数 var_sum 的任务是依据参数 fmt 来取得各个变参的值，或者，变参是指针的，取

得它所指向的变量的值，然后返回累加的结果。

在 main 函数里，我们调用了 var_sum 函数，传入的字符串是“lIiiL”，后面的 5 个参数对应于这个字符串的指示。

现在，请将函数 var_sum 补充完整，并在你的机器上翻译和调试，以检验自己写的是否正确。

```
# include <stdarg.h>

int var_sum (char * fmt, ...)
{
    //请在这里将函数补充完整
}

int main (void)
{
    int x = 50, r;
    long int y = 3;
    r = vs_sum ("lIiiL", 5L, & x, 6, 7, & y);
}
```

6.6.3 认识逐位或、逐位与和逐位异或运算符

在前面部分里，我们围绕函数 open 的声明讲述了限定的类型和变参函数，以及无处不在的类型定义（往后你会发现，到处都有 typedef 的身影）。介绍 open 函数是为了演示如何在 POSIX 框架下读写文件，但这个介绍工作至今没有完成。

现在，我想结合一个完整的示例程序来继续介绍 open 函数，在这个过程中，还将介绍其他 POSIX 库函数，以及文件读写的方法。名义上，下面的示例程序是为 UNIX 系统所写，但是 GCC 已经将相关的头文件和库移植到 LINUX 和 WINDOWS 上了，所以你也可以在这两种平台上编辑、翻译和运行它。

```
/**********************c0608.c*********************/
# include <unistd.h>
# include <fcntl.h>
# include <sys/stat.h>

int main (void)
{
    int fd;
    fd = open ("myfile1.dat", O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);

    if (fd == -1) return -1;

    char name [] = "LiuChangjiang", gender = 'M', age = 36;
    unsigned int score = 2200;

    write (fd, name, strlen (name));
    write (fd, & gender, sizeof gender);
```

```
    write (fd, & age, sizeof age);
    write (fd, & score, sizeof score);

    close (fd);
}
```

先说一下这个程序的功能，它的目的是创建一个文件，然后写入一个人的基本信息，包括姓名、年龄、性别和他在超市购物的积分。

在程序的开头用源文件包含指令引入了几个头文件，这都是 POSIX 标准的头文件。可以在头文件包含指令中使用路径，比如这里的<sys/stat.h>，这是一个相对路径，它表明，在 C 实现预定义的位置（目录或文件夹）下还有一个子目录 sys，头文件 stat.h 位于这个子目录下。

要写入文件，就必须先创建或者打开一个文件，在这里我们是用 open 函数，它是在头文件<fcntl.h>中声明的，这里再次给出它的声明：

```
int open (const char * pathname, int oflag, ...);
```

参数 pathname 的类型是指向 const char 的指针，你应该生成一个包含了文件名的字符串，并将指向这个字符串的指针传递给该参数，这样一来，在 open 函数内部就可以通过这个指针来获得这个文件名。

在 main 函数里，调用此函数时我们直接使用了字面串"myfile1.dat"，这将创建一个隐藏的数组，进而转换为指针。

注意，该指针所指向的类型是 const 限定的，这意味着 open 函数不会，也不能改变那个字符串的内容。有了这个保证，我们才敢用字面串"myfile1.dat"作为参数传递，毕竟字面串所创建的数组通常不允许修改，否则极有可能导致程序崩溃。

函数 open 的第二个参数是文件打开的方式，其类型为 int，然而我们在这里指定的却是莫名其妙的 O_CREAT | O_RDWR，这是什么意思呢？嗯……这是两个对象式宏定义，位于头文件<fcntl.h>中，通常是这样的：

```
#define O_RDWR      00000002
#define O_CREAT     00000100
```

显然，它们分别被定义为两个八进制整型常量，所对应的二进制数分别是（这两个整型常量的类型是 int，且我们假定 int 类型的长度是 32 个比特）：

00000002_8 = 0000 0000 0000 0000 0000 0000 0000 **0**01**0**$_2$

00000100_8 = 0000 0000 0000 0000 0000 0000 **0**1**0**0 0000$_2$

看到没有，这些数字都有一个特点：它们的二进制形式里，只有一个比特是 1，其他都是 0。这是编程界惯用的伎俩，他们喜欢将整数的每一个比特都用做一个标志，通过检查特定的比特是否为 1，来判断发生了什么或者应该做些什么。

每个宏都只负责一个标志位，O_RDWR 仅对应于文件打开方式的第 2 个比特，它的意思是创建或者打开文件的目的是既读又写；O_CREAT 仅对应于文件打开方式的第 7 个比特，它的意思是若文件不存在则创建它。

在这两个宏中间的“|”是 C 语言里的一个运算符，称为逐位或运算符，它需要一左一右两个操作数，且必须都是整数类型，其结果也是整数类型。

逐位或运算符对两个操作数逐位做逻辑或（加）操作，这种操作是“微观”的，是在操作数的二进制比特层面上进行的。以二进制的视角来看，逐位或操作只有在两个操作数相对应的比特都为 0 时，结果中对应的比特才为 0；在其他任何情况下，结果中对应的比特是 1。因此，表达式 O_CREAT | O_RDWR 的结果的二进制形式为：

0000 0000 0000 0000 0000 0000 **0**1**0**0 00**1**0$_2$

该二进制数对应的八进制数是 102。换句话说，八进制数 2 和八进制数 100 逐位或的结果是八进制数 102，用惯常的十进制来说，2 | 64 的结果是 66。一旦将这个结果作为实际参数传递给 open 函数，它将检测所有的标志位，而且会发现这两个比特都是 1，于是就实施相应的处理。

除了逐位或运算符，C 语言里还有逐位与运算符&和逐位异或运算符^。和逐位或运算符一样，它们也都要求一左一右两个操作数，而且都必须是整数类型，逐位与和逐位异或运算符的结果也是整数类型。

和逐位或运算一样，逐位与和逐位异或也在操作数的二进制比特层面上进行操作。以二进制的视角来看，逐位与操作只有在两个操作数相对应的比特都为 1 时，结果中对应的比特才为 1；在其他任何情况下，结果中对应的比特是 0，例如 0 & 0 的结果是 0；0 & 1 的结果是 0；1 & 1 的结果是 1；2 & 3 的结果是 2；56 & 78 的结果是 8。

逐位异或的操作过程是这样的：如果两个操作数相对应的比特是相反的，一个为 0，一个为 1，则结果中对应的比特是 1；在其他情况下，结果中对应的比特是 0。举例来说，0 ^ 0 的结果是 0；1 ^ 1 的结果是 0；0 ^ 1 的结果是 1；2 ^ 3 的结果是 1；56 & 78 的结果是 118。

我们已经知道，函数 open 是一个变参函数。通常情况下，以上两个参数就足够了，但是，如果第二个参数中指定了 O_CREAT，则必须有第三个参数。第三个参数是许可权限标志，用于指定哪些人能够打开这个文件，该参数的类型是 int。

所谓的“人”和“权限”，用过 LINUX 和 WINDOWS 的同学都应该知道，每个使用计算机的人要在操作系统中创建一个账号，并被管理员赋予一定的权限，指定对哪些计算机资源的访问和操作是允许的。在这里，我们指定的是 S_IRUSR | S_IWUSR，这也是两个对象式宏定义，位于头文件<sys/stat.h>。前一个标志 S_IRUSR 意味着只有文件的创建者才能读这个文件；S_IWUSR 意味着只有文件的创建者才能够写入这个文件。

函数 open 返回一个整数来唯一地代表这个文件，叫作文件描述符或者文件句柄。在程序的运行期间，你可以用这个描述符来读写那个文件。对于每一个打开的文件，操作系统会在内部记录它的各种属性和状态，操作系统需要依据这些信息来对文件进行操作，而文件描述符正是操作系统用来定位每个文件的属性和状态信息的线索与“把柄”。

如果函数 open 成功地打开或者创建了文件，则它返回该文件的描述符，否则返回-1。在当前程序里，我们声明了一个 int 类型的变量 fd，并用它接受这个返回值。然后，我们判断这个值是否为-1，如果是-1 则退出当前程序。

练习 6.5：

填空：37 | 21 的结果是（ ）；119 & 1 的结果是（ ）；59 ^ 95 的结果是（ ）。

6.6.4 指向 void 的指针

接着往下看，我们紧接着声明了变量 name、gender、age 和 score。变量 name 的类型是字符数组，用于存放姓名；gender 和 age 分别是性别和年龄。由于积分都是较大的非负数，我们采用 unsigned int 类型。

因为文件已经成功创建，而且要写入的内容都准备停当，现在我们就可以往里面写东西了，这要使用函数 write，该函数的原型是这样的：

```
ssize_t write (int filedes, const void * buff, size_t nbytes);
```

在这里，第一个参数 filedes 的类型是 int，用于指定一个文件描述符，这样它就知道要写入哪个文件；第二个参数 buff 用于指向一个缓冲区，这个缓冲区里有我们要写入的内容。

“缓冲区”是一个经常被用到的术语，是指一个连续的内存空间，比如一个数组。它最

典型的应用是在数据的发送方和接收方之间起到一个调节和缓冲的作用，就像一个蓄水池，如果数据的发送速度比接收速度快，缓冲区可以收纳来不及取走的数据。然而在这里它只是起到一个打包的作用，如果有很多东西要传递，比起分多次传递，每次传递一点点，将它们集中打包一次传递要方便些。

函数 write 并不关心写入的内容是什么，它只需要一个指针，这个指针必须指向这些内容的起始处。问题在于参数 buff 的类型，它应该不偏不倚。想想看，如果它被声明为指向 int 的指针，那就意味着我们只能写入一些 int 类型的数据。然而，要是用户想写的数据是别的类型，该怎么办呢？

事实上，函数 write 允许写入任何类型的数据。早先，它被声明为指向 char 的指针，因为 char 类型的变量最小，而且可以对齐于任何内存地址，这勉强说得过去，因为即使你要写入的数据都是对齐于 8 的，将它拆分成 char 类型来访问也不会有问题。因此，如果我们声明了一个 int 类型的变量，要把它的值写入文件，就生成指向这个变量的指针，并将它强制转换为指向 char 的指针，然后传递给 write 函数。

后来，C 语言引入了关键字 void。如果函数的参数是 void，表示没有参数，或者说参数为空；如果函数的返回类型是 void，表示不返回任何值，或者说返回空值。“空”就是不存在，所以 C 语言不允许声明 void 类型的变量：

```
signed int s;
void v;
```

以上，第一行没问题，第二行是非法的，不能通过编译，因为 v 是“不存在”的，C 实现无法为它分配存储空间。

虽然不能声明 void 类型的变量，但 C 语言却引入了指向 void 类型的指针，我们通常称之为“通用指针”，它可以指向任何变量。虽然瞎话（void）是不存在的事实，但说瞎话的人（指向 void 的指针）却是存在的。

为此，函数 write 的第二个参数 buff 被声明为指向 const void 类型的指针以适应传入的各种指针类型。

由于指向的具体类型不能确定，故指向 void 类型的指针不能用于访问它所指向的变量，因为无法知道变量的大小和数据存储格式；也不能做指针的加减操作，因为无法知道地址增加的长度。这就是说，给定以下声明（这两个声明是正确的）：

```
int x;
void * pv = & x;
```

则以下语句是非法的：

```
* pv = 330;
pv ++;
```

由于这种限制，可以想象到的是在函数 write 内部无法直接处理指向 void 类型的指针。唯一的办法是将用户指定的缓冲区看成一个字节的序列。即，将参数变量 buff 的值转换为指向 char 的指针，转型表达式(char *) buff 可以完成这一工作。

这样做是必然的选择，char 类型的变量可对齐于任何内存地址，用指向 char 的指针来访问用户传入的数据意味着我们是按字节来访问那些数据，所以不会有对齐问题，不管那些数据原来是什么类型。

C 语言规定，指向任何变量类型的指针都可以转换为指向 void 的指针；指向 void 类型的指针也可以转换为指向任何变量类型的指针；一个变量类型的指针转换为指向 void 类型的指针后，再转换回原来的指针类型，其结果同原来相比不变。

函数 write 的第三个参数 nbytes 是要写入的字节数。函数 write 是 UNIX 系统调用的封装，它并不关心你写入的是什么，仅将其视为一个字节的序列。要写入的数据是从变量

buff 的值所指向的位置开始，但是长度需要变量 nbytes 来指定。

如果成功写入，则函数 write 返回实际写入的字节数；如果在写文件的过程中发生了错误，则它返回-1。参数变量 nbytes 的类型是 size_t，这是一种无符号整数类型的别名，经常用于描述 sizeof 运算符的结果；函数 write 的返回类型 ssize_t，是一种有符号整数类型的别名。这些都不是新的类型，而是用 typedef 定义的别名。典型地，它们的定义是这样的：

```
typedef int ssize_t
typedef unsigned long size_t
```

回到程序中，让我们来看一看函数 write 是如何被调用的：

```
write (fd, name, strlen (name));
```

这里，变量 fd 保存了文件描述符，经左值转换后变成 int 类型的数值并赋给相应的形参变量，这没什么好说的；变量 name 是字符类型的数组，经数组—指针转换后，变成指向数组首元素的指针，其类型为 char *。但是它将自动从 char *转换为 void *再赋给形参变量，C 语言支持这种自动转换。

函数 write 的第三个参数是要写入的字节数。在很多情况下，手动统计要写入的长度会比较麻烦，恰好头文件<unistd.h>引入了一个计算字符串长度的函数 strlen，故我们直接用该函数的返回值作为实际参数。函数 strlen 要求传入一个指向字符类型的指针作为参数，它返回字符串的字符个数，不包含末尾的“\0”，其原型为：

```
size_t strlen (const char * str);
```

后面的三个函数调用分别将性别、年龄和积分写入文件中。和数组类型的实参不同，其他类型的变量需要用一元&运算符来得到指向它们的指针并传递给函数 write。因为写入的长度是以字节为单位的，所以一律用运算符 sizeof 得到。

如果多次调用函数 write，则每次写入的内容都位于上一次写入的内容之后；如果多次读取文件，则每次都从上一次读取的内容之后开始。之所以能够做到这一点，是因为在操作系统内部记录着用于操作该文件所需要的各种数据，其中包括一个文件偏移量（俗称文件指针），它是距离文件开头的字节数。每当读取或者写入文件时，将自动调整文件位置，使其位于下一次读写的开始处。

出于某些特殊的目的，你可能希望手工移动文件偏移量，这是允许的。要了解如何移动文件偏移量，以及 UNIX 编程的更多细节，可以参考《Advanced Programming in the UNIX Environment》这本书，作者是 W.Richard Stevens，这本书的中译本名为《UNIX 环境高级编程》。

当完成一个文件的读写后，应当关闭它。关闭一个文件将使得操作系统释放与之相关的各种资源（内部数据）。关闭文件需要使用函数 close，其原型为：

```
int close (int filedes);
```

调用函数 close 来关闭一个文件不是必须的，在当前程序退出时，所有在该程序内打开的文件都将被自动关闭。

现在，将上述程序保存为源文件 c0608.c，并用 gcc 翻译成可执行文件（WINDOWS 和 LINUX 下均可）。执行程序，观察当前目录中是否生成文件 myfile1.dat。打开该文件并观察其内容，所有文本编辑软件都试图将文件的内容解释为图形字符，但前面写入的是整数，只有当它偶尔与某个图形字符的编码是同一个数字时，才会显示为奇怪的字符。

你可能觉得奇怪，为什么在翻译上述程序时没有在命令行指定库文件，毕竟我们在程序中用到的那些函数都在库中。这是不必要的，GCC 会自动添加大多数标准库文件，然后寻找并链接到函数的实现。

既然有 write 函数，当然也会有 read 函数，读和写总是一对互为相反的操作。函数

read 用于从文件中读，其原型如下：

```
ssize_t read (int filedes, void * buff, size_t nbytes);
```

其中，参数变量 filedes 用于接受一个文件描述符；参数变量 buff 用于接受一个指向缓冲区的指针，从文件里读取的内容从它所指向的位置开始存放；参数变量 nbytes 用于接受本次要读取的字节数。每次读取后，系统将用实际读取的字节数修改文件位置，下一次读取的位置就从这里开始。

如果读取成功，该函数返回实际读取的字节数；若已经读到文件尾则返回 0；如果在读的过程中出错则返回-1。

练习 6.6：

编写一个程序读取文件 myfile1.txt 的内容。要求声明四个变量，分别用于保存读取到的姓名、性别、年龄和积分，并在调试器里观察读取的内容是否与前面写入的一致。

提示：函数 open 的第二个参数可以是 O_RDONLY 或者 O_RDWR，但不能同时指定，并且不需要第三个参数。

6.6.5 结构类型

如果你刚才认真做了上面的练习，就一定会觉得这个读出过程有些别扭，因为读出是写入的反过程，你得按顺序读出姓名、性别、年龄和积分。如果要写入和读出很多人的姓名、性别、年龄和积分，就显得笨拙而冗长。因为这个原因，我们现在要介绍一种新的数据类型，用它可以解决上述问题，这就是结构类型。

结构也是 C 语言里的一种类型，很抱歉现在才向大家隆重介绍。还记得吗，如果需要一大堆相同类型的变量，为了避免一个一个地重复声明它们，C 语言引入了数组。但是数组有一个缺点，那就是所有元素的类型必须相同。

在生产和生活中，我们可能需要将一些相互关联的数据组织起来。比如，如果要记录一个人的信息，那么这些信息就包括姓名、性别、年龄、身高、身份证号、职业、家庭关系、教育经历，等等。当然，这些信息可以独立地声明和存储，但这样做有一个缺点，那就是过于零散，不方便引用，如果要记录成千上万人的信息，很快就会陷入混乱。

发明计算机语言的目的是为了解决生产和生活中的实际问题，在 C 语言里引入结构类型可以解决上面的问题。不过，在正式介绍结构类型之前，我们需要先从类型指定符的角度来回顾一下 C 语言里的声明有什么特点。

在第一章里我们曾经介绍过类型指定符，但不那么具体。类型指定符包括 void、signed、unsigned、char、short、int、long，等等，用于在声明中指定类型，比如变量的类型、函数的参数类型和返回类型，等等。在以下对变量 m 的声明里，包含了三个类型指定符：int、signed 和 long，用于组合成一个有符号长整型，即 signed long int。这个声明是合法有效的，只不过类型指定符的组合顺序不符合我们已经形成的认知习惯，有些别扭。

```
int long signed m = 0;
```

在 C 语言里，有些类型是内置的，内置的类型由类型指定符组合而成，不需要声明就可使用，例如 int、signed char 和 unsigned long long int 等。内置的类型是 C 语言里的基本类型，它们是 C 语言的亲生子。

基本类型是构建其他类型的基础，从基本类型构建其他类型称为类型的派生，我们已经学过的指针、数组和函数都是派生类型。指针类型派生自它所指向的类型；数组类型派生自它的元素类型；函数类型派生自它的参数类型和返回类型。

按理说，类型应该先派生，再使用（用来声明变量和函数）。但是，指针、数组和函数类型的派生过程是与变量或者函数的声明合而为一的。换句话说，在 C 语言里不存在数组、

指针和函数类型的指定符，只有在声明之后，才会产生具体的数组、指针和函数类型，无法事先定义这些类型。

举个例子来说，如果我想声明一个数组类型的变量 a，这种数组类型的特征是具有 5 个 int 类型的元素；再声明一个函数 f，这种函数类型的特征是具有 2 个 int 类型的参数且返回类型也是 int，该怎么做呢？按道理，这两个声明应该是这样的：

```
int [5] a;
int (int, int) f;
```

这里，貌似把 int [5]作为类型指定符，指定了一种“具有 5 个 int 类型的元素”的数组类型；貌似把 int (int, int)作为类型指定符，指定了一种“具有两个 int 类型的参数且返回类型也是 int 的函数类型。但是很遗憾，这两个声明是非法的，而且 C 语言里没有指针、数组和函数类型的指定符，所以我们只能这样做：

```
int a [5];
int f (int, int);
```

这里，我们不但声明了变量 a 和函数 f，同时也声明（派生）了一种数组类型和一种函数类型。注意，类型名和类型指定符是不同的，在这里派生的数组类型和函数类型可以用类型名 int [5]和 int (int, int)来描述，但它们不是 C 语言里的类型指定符。

那么，这是不是意味着所有派生类型都没有对应的类型指定符呢？那倒未必。结构类型也是派生类型，派生自它的成员类型，而且结构类型有自己的类型指定符，称为结构指定符。结构指定符的语法形式为

struct *标识符*可选 **{** *成员声明列表* **}**

struct *标识符*

结构指定符要由关键字“struct”引导，后面是一个标识符，以及由一对花括号“{}”围起来的成员声明列表。标识符可以省略，花括号及其围住的成员声明列表也可以省略，但是不能同时省略。下面是一个结构指定符的例子：

```
struct {
    int x;
    int y;
}
```

这个结构的类型指定符省略了标识符，但保留了成员声明列表。结构成员声明列表必须至少声明一个成员，比如在这里就声明了两个 int 类型的成员 x 和 y，分别表示横坐标和纵坐标。注意，最后一个结构成员之后的分号不可省略。

关键字“struct”是固定不变的部分，然而结构的成员却可以根据需要灵活设置，包括它们的类型、数量（最多 1023 个成员）和名字（在结构内部不得重名），各个成员的声明之间用分号隔开。C 语言在形式上比较自由，所以这个结构指定符也可以写成：

```
struct {int x; int y;}
```

和我们以前的声明一样，如果两个相邻的成员具有相同的类型，则它们可以用逗号进行合并，因此上述结构指定符亦可以改写为：

```
struct {int x, y;}
```

结构指定符可用于声明结构类型的变量和函数，例如：

```
struct {int x, y;} crd;
```

现在，如果要问变量 crd 是什么类型？回答是 struct {int x; int y;}类型；该（结构）类型的成员 x 是什么类型的？回答是 int 类型。

原则上，相同的类型指定符代表着同一种类型。在下面的例子中，变量 m 和 n 的声明里都使用了类型指定符 int，我们可认定 m 与 n 的类型相同：

```
int m;
int n;
```

然而 C 语言里存在着很多例外。请看下面的例子：

```
struct {char name [20], gender; int chk;} person1;
struct {char name [20], gender; int chk;} person2;
```

以上，我们声明了两个变量 person1 和 person2，它们的类型指定符完全相同，都是 struct {char name [20], gender; int chk;}。按道理，这两个变量的类型相同，或者说具有相同的结构类型。

然而很遗憾，变量 person1 和 person2 的类型并不相同。原因在于，结构类型指定符虽然也是类型指定符，但它毕竟不是内置类型，而是派生类型，所以，每个出现在程序中的结构类型指定符具有自我声明的性质。换句话说，它虽然是个类型指定符，但它同时也是一种结构类型的声明。非但如此，**带有成员声明列表的结构指定符每出现一次，都将声明出一个新的结构类型。**

因为这个原因，结构类型指定符中的标识符通常不应该省略，这个标识符称为结构类型的标记。这样一来，标记就代表了当前这种结构类型，而我们也就能够在任何时候使用这种类型了。

如下例所示，一旦加上了标记，则第一行不但声明了一种结构类型 struct persn，同时还声明了该结构类型的变量 person1；然后，在第二行里，我们还可以继续声明这种结构类型的变量 person2，并且变量 person1 和 person2 的类型完全相同。

```
struct persn {char name [20], gender; int chk;} person1;
struct persn person2;
```

事实上，我们完全可以先声明标记，然后再用该标记来声明变量，例如（注意第一行，每个结构类型的声明都必须用一个分号结束）：

```
struct persn {char name [20], gender; int chk;};
struct persn person1, person2;
struct persn person3;
```

之所以这会行得通，是因为在声明 person1、person2 和 person3 的时候，我们已经声明了一种结构类型，同时也将标识符 persn 声明为该结构类型的标记，然后再引用这个标记来声明变量时，就是在“指定”那种结构类型。

当然，不加标记也并不意味着就没有办法把某种结构类型“固定”住。我们已经学过类型定义，在结构类型的声明中使用关键字“typedef”就可以做到这一点，例如：

```
typedef struct {char name [20], gender; int chk;} sPersn;
sPersn person1, person2;
sPersn person3;
```

以上，我们声明了一种结构类型，因为它有成员声明列表，然后，又为这种结构类型定义了一个别名 sPersn。在此之后，我们就可以用 sPersn 类型来声明结构变量。

和数组类型一样，结构类型的变量也是由子变量组成，或者说是成员变量。结构类型的变量在声明时也可以带有初始化器以初始化它的每个成员。因为结构变量是带有子变量的变量，故它的初始化器应当用花括号围起来；如果它的成员也是由子变量组成（例如一个数组类型的成员），也需要用花括号围起来；在初始化器里，表达式的顺序要与结构成员声明的顺序一致。下面是一个例子：

```
struct persn person1 = {{"Lizhong"}, 'M', 33};
```

在这里，初始化器{"Lizhong"}用于初始化变量 person1 的第一个成员 name，围住字面串的花括号是可选的；表达式'M'用于初始化第二个成员 gender；表达式 33 用于初

始化第三个成员 chk。

如果初始化器的数量少于结构成员的数量，则初始化的顺序依然是按结构成员声明的顺序进行，但多余的成员都被初始化为一个默认值。至于这个默认值是什么，要取决于那个成员的类型，通常是把整型常量 0 转换为那个成员的类型所得到的结果（值）。比如，如果成员的类型是整型，则自动初始化为 0；如果是一个指针，则初始化为空指针。

从 C99 开始，允许在初始化器里使用指示器。我们知道，对于数组变量的初始化器，可以使用“[]”来指示一个要初始化的数组元素，而对于结构类型的成员，则可以使用“.”和成员的名字来指示。使用这种方式可以不用考虑成员的顺序，例如：

```
struct persn person2 = {.chk = 33, .gender = 'F', .name = "Alice"};
```

我们知道内存是线性的——也就是说，内存空间可以视为字节的序列。尽管结构包含很多成员，但它在内存中是按成员声明的顺序线性存储的，如图 6-7 所示。

需要特别注意的是，因为对齐的缘故，结构内部可能会有无用的填充字节。是否会有填充字节，这取决于结构成员的类型和数量。但可以肯定的是，用结构类型 struct persn 声明的变量都会有填充字节。填充字节的内容是不确定的，通常是一些随机值。

如图 6-7 所示，在我的机器上，变量 person1 的内存地址是 0x22FE88，这也是其第一个成员 name 的起始地址，这是因为结构变量的开头绝不会有填充字节。不单是结构变量的成员有对齐要求，结构变量本身也有自己的对齐要求，它的对齐一定会兼顾其第一个成员的对齐，所以填充字节只可能出现在结构的中间和尾部。

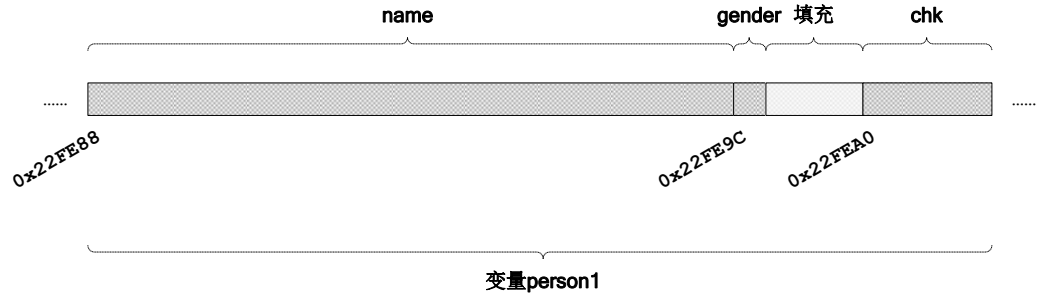


Fig.6-7 结构内部的填充

结构成员 gender 的类型是 char，而 char 类型可以对齐于任何内存地址，所以它将紧接着第一个成员，位于地址 0x22FE9C 处。第三个成员 chk 的类型是 int，在我的机器上是 4 个字节的长度，要求按 4 字节对齐，也就是只能位于可被 4 整除的地址上。为此，只能在 gender 和它之间填充 3 个无用的字节，使得成员 chk 位于地址 0x22FEA0。

练习 6.7：

如此看来，结构的大小并非是其所有成员大小的总和。为了验证这一点，请你编写一个程序，用 sizeof 运算符得到 struct persn 类型的大小并在调试器里观察。

提示：要得到结构 struct persn 的大小，可使用表达式 sizeof (struct persn) 或者 sizeof person1，前者是得到类型的大小，后者是得到变量的大小，效果相同。

为了演示结构类型如何为处理复杂的数据带来方便，下面是一个例子。在这个程序中，我们先是声明了一个结构类型 struct employee。注意这个结构类型是在函数定义之外声明的，位于源文件的开头，这样做的好处是它在源文件的剩余部分始终“可见”，并因此可以随时使用这种类型。

```
/*********************c0609.c*********************/
# include <unistd.h>
```

```
# include <fcntl.h>
# include <sys/stat.h>

struct employee
{
    char name [20];
    char gender;
    char age;
    unsigned int score;
};

int main (void)
{
    int fd;
    fd = open ("myfile2.dat", O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);
    if (fd == -1) return -1;

    struct employee emp = {"WangXiaobo", 'F', 26, 5000};
    write (fd, & emp, sizeof emp);

    close (fd);
}
```

结构类型 struct employee 包含了 4 个成员，分别是姓名（name）、性别（gender）、年龄（age）和活动积分（score）。在函数 main 里，我们先是创建一个文件并返回它的文件描述符，然后声明了一个那种结构类型的变量 emp 并做了初始化。

函数 write 的第二个参数是指向 const void 类型的指针，我们在这里传递的是表达式& emp 的值。表达式 emp 是一个 struct employee 类型的左值，一元&运算符作用于它，得到一个指向结构类型的指针，或者说指向 struct employee 类型的指针，然后这个指针被转换为指向 void 的指针。尽管我们还没有讲过指向结构类型的指针，但它其实十分简单，没什么好说的。如果非要说些什么的话，那么，当前这条 write 调用，以及它前面的那个声明，还可以写成这样：

```
struct employee emp = {"WangXiaobo", 'F', 26, 5000};
struct employee * pemp = & emp;
write (fd, pemp, sizeof (struct employee));
```

看，这里就声明了一个指向结构类型的指针变量 pemp 并初始化为表达式& emp 的值。因为变量 pemp 的类型是 struct employee *，而表达式& emp 的（值的）类型也同样是 struct employee *，故可以初始化。

注意，在这个新的 write 调用里有两处变化，一是我们直接传递了变量 pemp 的值，二是我们传递了结构类型的大小，而不是以前的 sizeof emp，这样更自然些。如果你愿意，把它换成表达式 sizeof * pemp 也是可以的。

在函数 write 内部并不知道数据的原始类型，它也不需要知道，它仅仅是将其视为一个字节的序列并写入磁盘文件。即使原先的类型是一个结构，且内部有填充字节，它也会把这些填充字节原样写入磁盘文件。

现在，让我们看看一个结构类型的变量在调试器里是如何呈现的。以下是调试过程，我

们先将断点设置在第 21 行，此时变量 emp 已经创建并初始化完成：

```
(gdb) b 21
Breakpoint 1 at 0x4016a6: file c0609.c, line 21.
(gdb) r
Starting program: D:\exampls\c0609.exe
[New Thread 2580.0xc6c]

Breakpoint 1, main () at c0609.c:21
22          write (fd, & emp, sizeof emp);
(gdb) p emp
$1 = {name = "WangXiaobo\000\000\000\000\000\000\000\000\000",
  gender = 70 'F', age = 26 '\032', score = 5000}
(gdb)
```

显然，要显示一个结构类型的变量，同样可以使用“p”命令。此时，gdb 将显示各个成员的内容，而且从格式上来看很像一个初始化器。

练习 6.8：

1.编写程序,将我们写入到文件myfile2.dat 中的内容读到一个结构类型的变量中，并在调试器里观察读取的内容是否与前面写入的一致。

2.试从类型和类型转换的角度描述表达式 sizeof * pemp 的求值原理（或者求值过程）。

现代操作系统倾向于使用同一种逻辑和接口来处理物理上不同的设备和文件，比如，它通常会将显示设备映射为一个特殊的文件并赋予一个（可能是固定的）文件描述符。作为一个示例，下面的程序用于向显示器输出一个字符串的内容。

```
/*********************c0610.c*********************/
# include <unistd.h>

int main (void)
{
    char * pbook = "The Laws of Boole's Thought.\n";
    write (STDOUT_FILENO, pbook, strlen (pbook));
}
```

在调用 write 函数之前，应当首先创建或者打开一个文件，但是我们没有这样做，而是直接传递了一个奇怪的参数“STDOUT_FILENO”。实际上这没有什么奇怪的，这是一个宏，在头文件<unistd.h>里被定义为整数型常量 1 的别名：

```
# define STDOUT_FILENO 1
```

我们知道，每个运行在操作系统上的程序，都会有一段初始化的代码。这个初始化的过程包括打开三个标准设备：标准输入、标准输出和标准错误。标准输入通常指键盘，供每个程序获取外部的输入；标准输出通常指显示器，供每个程序输出信息；标准错误通常指显示器，供每个程序输出它的错误信息。因为这三个设备都被当成文件对待，所以它们的文件描述符分别是 0、1 和 2，不需要显式地打开或者创建，它们预定义的，在程序启动后就已经自动打开了。

如果需要，标准输入、标准输出和标准错误可以重定向到其他设备和文件。比如，要是你想把错误信息或者程序的输出打印到一个文本文件里，而不是显示在屏幕上，就可以将它

们重定向为磁盘文件。为演示具体的做法，现以本章前面用源文件 c0604.c 生成的可执行文件 c0604.out 为例：

```
$ ./c0604.out > outfile.txt
$ cat outfile.txt
1+2+3+...+1000=500500
$
```

在这里，“>”的意思是将标准输出重定向，在这里是重定向到文件 outfile.txt。于是，程序的运行结果并不在屏幕上显示，而是创建并写入文件 outfile.txt。当我们显示该文件的内容时，就看到了程序运行的结果。

现在，你可以将上述程序保存为源文件 c0610.c 并翻译成可执行文件。典型地，这个程序应该在 UNIX 系统上翻译和执行，不过好在 GCC 已经在各个平台上都实现了 POSIX 标准的库，包括 LINUX 和 WINDOWS。所以，你当然可以在 LINUX 和 WINDOWS 上编辑、翻译和运行这个程序。在 LINUX 上，该程序的翻译和执行步骤如下：

```
$ gcc c0610.c -o c0610.out
$ ./c0610.out
The Laws of Boole's Thought.
$
```

进一步地，如果想从标准输入接收数据并发送到标准输出，也是很有意思的，下面的程序就用于从键盘接收输入，然后把它们显示在屏幕上。

```
/*******************c0611.c*******************/
# include <unistd.h>

# define PROMPT "->"

int main (void)
{
    char buf [1];

    write (STDOUT_FILENO, PROMPT, strlen (PROMPT));

    while (read (STDIN_FILENO, buf, sizeof buf) > 0)
        write (STDOUT_FILENO, buf, sizeof buf);
}
```

在这个程序中，我们首先在标准输出上打印一个提示符“->”，表示“请在后面输入一些东西”。在程序翻译时，宏 PROMPT 被展开为它的原始形态——字面串，并用于创建一个隐藏的无名数组，然后进一步转换为指向其首元素的指针并传递给 write 函数和 strlen 函数。

在对函数 read 的调用中，STDIN_FILENO 是一个宏，被定义为一个整数 0，是标准输入（通常是指键盘）的文件描述符。

函数 read 在读到文件末尾时返回 0，在发生错误时返回-1。在 while 语句中，函数 read 从标准输入读取，如果返回值大于 0，则执行循环体，将读来的数据写到标准输出，然后继续下一次循环（继续读取）。

通常情况下，在系统内部会给标准输入和标准输出分配一个缓冲区，而且这两个缓冲区是由换行字符驱动的。也就是说，如果标准输入是指键盘，则按下的按键代码会先进入这个

缓冲区，直到缓冲区满，或者按下了回车键，函数 read 才能读到按键；如果标准输出是指显示器，则函数 write 写入的字符先进入缓冲区，直至缓冲区满，或者遇到了换行字符，才真正执行更底层的写入操作。

如下面的命令行操作过程所示，我们只是输入“hello world.”没有用处，只有在按下回车键之后，函数 read 才开始工作，从缓冲区里读取字符，每次 1 个，然后写到标准输出，就这样循环读写，直至将缓冲区读空。

然而，这里有一个问题：输入设备（通常是键盘）被当作文件对待，但毕竟不同于磁盘文件，在什么情况下才意味着“读到了文件尾部”呢？在函数 read 读取之前和之后，标准输入的缓冲区是空的，函数 read 也读不到字符，这是不是意味着已经读到了文件尾部？如果是的话，为什么函数 read 不返回 0 以至于 while 循环依然继续？答案是，这是一个系统的设计问题。想想看，我们通常是等待用户按下键盘，而不是让用户抢在程序开始读键盘之前就按下按键，这不是正常的做法，所以不能以缓冲区是否为空作为依据。

那么，我们该如何才能让函数 read 因遇到文件末尾而返回 0 呢？否则，while 语句将永远停不下来，除非我们强行关闭程序。答案是，在 LINUX 上，可以按下 CTRL+D 来发送一个文件结束信号；在 WINDOWS 上，可以按下 CTRL+Z，这两种方法都将导致系统发送一个文件结束的信号，并使得函数 read 返回 0。

如下面的命令行操作过程所示，当我们在 WINDOWS 上按下 CTRL+Z（同时按下 CTRL 键和 Z 键，或者先按下 CTRL 键不松开，再按下 Z 键）后，再按下回车，函数 read 返回 0，退出循环，程序也就退出了。

和标准输出一样，标准输入也可以被重定向。于是，很有意思的事情出现了：我们可以用上述程序来实现文件复制！如下面的命令行操作过程所示，我们先准备一个文件，例如 myfile.txt，然后用 type 命令（在 LINUX 上是 cat）显示它的内容。

接着，我们在准备运行程序的同时，用“>”将标准输出重定向到文件 out.txt，再用“<”将标准输入重定向到文件 myfile.txt，然后按下回车。因为我们的程序是从标准输入读取，然后写到标准输出，但因为标准输入和标准输出都已被重定向，所以它是从文件 myfile.txt 读取内容，然后写入文件 out.txt（如果文件不存在则创建）。当程序运行结束后，再打开文件 out.txt，你会发现，它的内容和 myfile.txt 完全相同。

```
C:\examples>gcc c0611.c -o c0611.exe
D:\examples>c0611
->hello world.
hello world.
^Z

D:\examples>type myfile.txt
good
morning
!

D:\examples>c0611 > out.txt < myfile.txt

D:\examples>type out.txt
->good
morning
!
```

```
D:\examples>
```

练习 6.9：

利用重定向功能，用上面的程序完成以下操作：一，将键盘输入的内容写入文件 out1.txt；二，将文件 myfile.txt 的内容打印到屏幕。

6.7 WINDOWS 动态链接库

与 UNIX 和 LINUX 相似，WINDOWS 也使用底层的系统调用机制工作。问题在于，这些系统调用并不是公开的，也不固定，并总是随着 WINDOWS 的版本而变化。因为这个原因，我们可能无法直接使用系统调用在屏幕上打印东西。

要用 C 语言编写基于 WINDOWS 的程序，微软公司建议的方法是使用动态链接库。动态链接库类似于前面讲过的共享库，当你在程序中使用共享库的函数时，仅在可执行文件中加入调用库函数的代码，并不将库中的代码链接到可执行文件中。在这种情况下，生成的可执行文件和动态库有依赖关系，仅在需要时才加载库中的代码。

动态链接库都是以“.dll”为后缀（扩展名）的文件，在 WINDOWS 里有着大量的动态链接库，比如著名的 kernel32.dll 和 user32.dll 等，各自提供不同的功能。动态链接库的功能由库内的函数实现，并可以链接到用户的程序。从用户的角度来看，这些库函数是用户程序访问、使用操作系统功能的编程接口。

同样是将数据写入磁盘文件，在 WINDOWS 上同样需要创建或者打开文件、写入并最终关闭文件。但是，要完成这些工作，需要使用动态链接库里的函数，下面就是一个例子：

```
/**********************c0612.c**********************/
# include <windows.h>
# include <strsafe.h>

typedef struct employee
{
    char name [20];
    char gender;
    char age;
    unsigned int score;
} stgEMP;

int main (void)
{
    HANDLE hf = CreateFile ("myfile3.dat", \
                        GENERIC_READ | GENERIC_WRITE, \
                        0, NULL, CREATE_ALWAYS, \
                        FILE_ATTRIBUTE_NORMAL, NULL);

    if (hf == INVALID_HANDLE_VALUE) return -1;

    stgEMP emp;
    StringCchCopy (emp.name, sizeof emp.name, "HuoFengrong");
    emp.gender = 'F';
```

```
    emp.age = 79;
    emp.score = 56666;

    WriteFile (hf, & emp, sizeof emp, & (DWORD) {0}, 0);

    CloseHandle (hf);
}
```

在源文件的第一行，我们包含了头文件<windows.h>，这是个非常重要的头文件，面向 WINDOWS 编程所用到的函数，它们的声明都通过这个头文件引入。当然，这些声明并不都位于该头文件里，事实上，它们分门别类地划分为很多头文件，只不过<windows.h>会将它们一并包含进来。换句话说，每个头文件也可以用#include 预处理指令包含其他头文件。在预处理阶段，嵌套的文件包含将一层一层地被展开到当前源文件里。

和前面一样，我们首先在程序里声明了一个结构类型，但这个声明既像类型定义（有关键字 typedef）又像一个结构声明。实际上，这个声明做了好几样工作：首先，它声明了一种结构类型，因为它有成员声明列表；其次，它声明了一个标记 employee，并使它代表当前的结构类型；最后，它为当前所声明的结构类型定义了一个别名 stgEMP。如此一来，这种结构类型就有了两个名字：struct employee 和 stgEMP，而且它们是同一种结构类型，只不过使用 stgEMP 较为方便。

先来看函数 main 的代码，如第一行所示，在 WINDOWS 里创建或者打开一个文件可以使用函数 CreateFile，它位于动态链接库 kernel32.dll 中，其原型为：

```
HANDLE CreateFile (
        LPCTSTR                 lpFileName,
        DWORD                   dwDesiredAccess,
        DWORD                   dwShareMode,
        LPSECURITY_ATTRIBUTES   lpSecurityAttributes,
        DWORD                   dwCreationDisposition,
        DWORD                   dwFlagsAndAttributes,
        HANDLE                  hTemplateFile
);
```

一旦进入 WINDOWS 编程的世界，就相当于进入了一大片黑暗的丛林（好在你很快就能适应并享受其中），这不仅是因为 WINDOWS 的体系结构非常复杂，而且很多函数都需要一大堆参数。参数多也就罢了，他们还将类型定义发挥到极致。比如在我的计算机上，HANDLE 实际上是 void *类型的别名；DWORD 就是 unsigned long int 类型的别名。

WINDOWS 之所以这么干，第一个原因是方便记忆、理解和识别。例如，“HANDLE”的意思是句柄，就像可以通过把手和手柄来开门一样，通过句柄可以访问到它所关联的、位于操作系统内部的实体，例如文件、设备、窗口、字体、图标等；“DWORD”的意思是“双字”，在主流的认知里，字节是 8 个比特，一个字包含两个字节，一个双字则由两个字组成。

另一方面，定义现有类型的别名是可移植性的需要。同一个类型名，它背后的真实类型可以随 C 实现的不同而变化。假设你用 DWORD 声明了一个变量，那么，如果在某平台上 unsigned int 类型的长度是 16 位而 unsigned long int 类型的长度是 32 位，则该平台上的 C 实现将会把 DWORD 定义为 unsigned long int 的别名；相反，如果在某平台上 unsigned int 和 unsigned long int 类型的长度都是 32 位的（在我的机器上就属于这种情况），则该平台上的 C 实现就可以把 DWORD 定义为 unsigned int 或者 unsigned long int 的别名。但是无论如何，你的程序不用修改就可以在这两种平台上

翻译和执行。

无论如何，你都不用担心是否会出现超出 C 语言类型系统的新类型，所有稀奇古怪的类型都是 C 语言所认可的类型的别名。下面将要介绍我们用到的 WINDOWS 函数，以及它们的参数。你可能有很多疑问，但不要纠结于细节，我在这里也只是一般性地介绍，因为我们的任务是学习 C 语言，领略它在实际编程工作中的价值，而不是全面地介绍 WINDOWS 程序设计。如果你确实有不明白的地方而又查不到资料，可以通过我的个人网站联系我。

函数 CreateFile 的第一个参数是 lpFileName，从该参数的名字上看是指向文件名的指针。该参数的类型是 LPCTSTR，在我的机器上，C 实现将其定义为 const char *类型的别名。所以如程序中所示，我们传入的是字面串"myfile3.dat"，我们已经知道，它是一个数组类型的左值，将用于创建一个隐藏的数组并转换为指向其首元素的指针，然后传递给 WriteFile 函数。

第二个参数是 dwDesiredAccess，意思是你所期望的访问方式，比如读、写或者既读又写，等等。该参数的类型为 DWORD，这意味着你应当传入一个整数，而我们在程序中传入的是逻辑或表达式 GENERIC_READ | GENERIC_WRITE 的值。GENERIC_READ 和 GENERIC_WRITE 是两个宏，由头文件<windows.h>引入，代表着两个有特定位模式的整数，它们做逐位或运算，并将结果传递给 WriteFile 函数。

第三个参数是 dwShareMode，意思是共享模式。该参数的类型是 DWORD，也即一个整数。如果你传入的是 0（就像本程序所做的那样），则意味着本次成功打开某个文件后，将不允许其他程序再次打开、读取、写入或者删除它，即处于独占模式。再比如，要是你传入的是 FILE_SHARE_WRITE（0x00000002），则意味着允许其他程序以写入模式重新打开它。

第四个参数是 lpSecurityAttributes，意思是安全属性，比如哪个用户或者用户组拥有该文件，以及不同的用户账户或者组对该文件有何种访问权限，等等。该参数的类型是 LPSECURITY_ATTRIBUTES，以“LP”打头意味着它是一个指针。没错，它是一个指向结构类型的指针，该结构类型是 WINDOWS 预定义的，通过头文件<windows.h>引入。如果你需要对所创建的文件施加特定的权限，则应当声明这种结构类型的变量，并传入它的地址；如果你传入的是空指针，则操作系统将使用默认的安全属性。

注意，我们在程序中传入的是 NULL，它是一个宏，而且是一个颇具知名度的宏，因为各种 C 库都会定义它，但定义的方式都一样：

```
# define NULL 0
```

或者

```
# define NULL (void *) 0
```

C 语言规定，一个值为 0 的整型常量表达式是一个空指针常量；将这样的表达式转换为 void *类型的，也是空指针常量。显然，宏 NULL 就代表空指针常量。我们当前所使用的这个 NULL 是通过头文件<windows.h>引入的宏。

再来看，第五个参数是 dwCreationDisposition，是指创建文件时的倾向性：当创建一个文件时，文件已经存在，或者打开一个文件时，文件不存在，该怎么办。这个参数的类型是 DWORD，我们在程序中指定的是 CREATE_ALWAYS，这也是通过<windows.h>引入的宏定义，意为“总是创建一个新文件”，不管它是否已经存在。如果要打开一个已经存在的文件，应当使用 OPEN_EXISTING。

第六个参数是 dwFlagsAndAttributes，意思是标志和属性。可以为文件指定的属性包括但不限于隐藏、只读、归档、加密、关闭句柄时删除文件，等等，一大堆。该参数的类型是 DWORD，在本程序里，我们指定的是 FILE_ATTRIBUTE_NORMAL（0x80），意为正常的属性，对于普通的文件操作来说，这个宏就足够了。

第七个参数是 hTemplateFile，意思是模板文件。当创建一个文件时，它的属性可以模仿一个现成的文件，称为模板文件。要想使用这一功能，你需要传入一个模板文件的句柄，但在我们的程序中，这个功能并不需要，所以直接置为 NULL，毕竟 HANDLE 类型本质上是一个指针类型。

最后，如果函数 CreateFile 执行成功，则文件被打开或者创建并返回一个有效的句柄；如果失败，则返回 INVALID_HANDLE_VALUE（无效句柄值），当然，这也是通过头文件<windows.h>引入的宏定义：

```
# define INVALID_HANDLE_VALUE (HANDLE) -1
```

显然，无效句柄值是 HANDLE 类型的-1。变量 hf 的类型是 HANDLE，函数 CreateFile 的返回值类型也是 HANDLE，类型一致，可以做等性比较。当然，我们知道 HANDLE 类型是指针类型 void *的别名。void 类型是不完整的类型，指向 void 的指针是指向不完整类型的指针，很多运算符不允许操作数的类型是 void *，但等性运算符==和!=是为数不多的例外，因为它们不需要通过这个指针访问被指向的实体，也不需要知道被指向的确切类型。

6.7.1 认识成员选择运算符“.”

接下来，我们声明了一个结构类型的变量 emp，其类型为 stgEMP，但我们知道该类型是 struct employee 的别名而已。

变量 emp 并没有初始化，我们需要为它提供值。然而，一旦错过了初始化，能够为结构变量提供一个“整体值”的方法不多。好在“不多”意味着还有希望。和数组不同，在 C 语言里，允许将一个结构变量的值保存到另一个结构变量，就像这样：

```
struct t {char a [20]; int m;} ta = {"hello", 21}, tb;
tb = ta;
```

以上，我们声明了结构类型的变量 ta 并做了初始化；同时还声明了另一个结构类型的变量 tb。接着，我们用一个表达式语句将变量 ta 的值整体上保存（复制）到变量 tb。除此之外，函数的参数类型和返回类型也可以是结构，例如下面的函数声明：

```
struct t frets (struct t tm)
{
    return tm;
}
```

在这里，函数 frets 接受一个 struct t 类型的参数，并返回一个 struct t 类型的值。这个函数只是简单地把形参 tm 的值直接返回给调用者，为此编写一个函数确实十分荒唐，但作为一个例子来说足够简明。

练习 6.10：

给定结构类型 struct t {int x, y;}，若函数 f 接受一个 struct t 类型的参数 m，返回类型也是 struct t，该函数的功能是递增 m 的成员 x 和 y，然后返回 m 的值。在 main 函数内调用函数 f 并把返回值赋值变量 g，请编写这个程序。

除了上面的方法之外，最常用的莫过于为结构变量的成员分别提供值，这也是我们在程序中使用的办法。为了给结构的成员赋值，需要先选择和得到那个成员，这就要用到成员选择运算符，它实在是太不起眼了，仅仅是一个点：.。

如果是要得到结构的成员，或者得到结构成员的值，则运算符.的左操作数必须是结构类型，右边是指示那个成员的标识符（成员的名字），例如 emp.name。如果左操作数是左值，则运算符.的结果也是左值，代表那个成员，或者说得到那个成员；如果左操作数是一个值，则运算符.的结果是得到那个成员的值（而不是左值）。无论如何，结果的类型和那

个成员的类型相同。在表达式 emp.name 中，左操作数 name 是左值，故表达式 emp.name 的结果是左值，代表结构变量 emp 的 name 成员本身。

有同学问了，运算符.的左操作数还可以是值？是的，最典型的例子就是函数，函数返回的是值，而从来不会是左值。给定以下声明：

```
struct t {char name [20]; unsigned int age;};
struct t f (void);
```

在这里，函数 f 的返回类型是 struct t。也就是说，它返回一个结构类型的值。下面这行代码用于从函数 f 的返回值里“抠取”其 name 成员的值：

```
char * p = f ().name;       //D1
char c = p [0];             //D2，未定义的行为
char d = f ().name [0];     //D3
```

因为函数 f 的返回值是 struct t 类型，可作为运算符.的左操作数。但因为它是一个值而不是左值，故表达式 f ().name 的结果是得到成员 name 的值。成员 name 是数组类型的，故我们得到数组的值，也就是数组的内容。说实话，在 C 语言里，能够得到数组的值（内容），这种机会实在不多，因为它像泥鳅一样，瞬间就变成了指针。不信你看——

在函数 f 返回时，返回的是结构类型的值，数组成员 name 也以值的形式随之返回。来看第一个声明 D1，表达式 f ().name 的结果是成员 name 的值，其类型为数组，类型名为 char [20]。但是很快，它又执行数组－指针转换，转换为指向其首元素的指针，并用于初始化指针变量 p。

然而，返回值是临时的，在整个声明 D1 处理完成后即会消失[5]，所以在声明 D2 中，变量 p 的值指向一个无效的内存位置，这是很危险的，表达式 p [0]用来取得该数组下标为 0 的元素，但它访问的是一个无效的数组，这种行为是未定义的，后果难以预料。

在声明 D3 中，函数 f 的返回值会在表达式 f ().name [0]求值完成后消失，函数调用运算符、成员选择运算符和下标运算符的优先级相同，都属于后缀运算符，但它们是从左往右结合的，所以这个表达式等价于(f ().name) [0]。成员选择表达式 f ().name 得到成员 name 的值，这是一个数组类型的值，转换为指向数组首元素的指针，然后，下标运算符得到下标为 0 的元素，是一个左值，经左值转换后，得到那个元素的值。

练习 6.11：

把上面这个小例子补充为一个完整的程序，运行程序并体会成员选择运算符.的功能和用法。

言归正传，在程序中，成员 name 是一个数组，我们想把一个字符串保存进去，据我们已经掌握的知识，这样做是行不通的：

```
emp.name = "HuoFengrong";
```

成员选择运算符.的优先级高于赋值运算符，所以是把字面串赋给结构的成员。表达式 emp.name 是一个左值，代表着结构 emp 的成员 name，这个成员的类型是数组。按规定，数组类型的左值是不可以修改的，而且 emp.name 和"HuoFengrong"都会转换为指针。

基于上述原因，在实际的编程工作中我们经常使用字符串复制（拷贝）的办法来解决这一问题。我们可以自己编写一个函数来执行复制操作，也可以使用现成的库函数，而且有很多现成的库函数可用，我们在程序中用的是 StringCchCopy 函数，其原型为：

```
STRSAFEAPI StringCchCopy ( STRSAFE_LPSTR pszDest, size_t cchDest, STRSAFE_LPCSTR pszSrc
```

[5] 函数调用的返回值通常用于初始化变量，或者用于给赋值运算符的左操作数赋值，这相当于复制了这个值，制作了它的副本。

);

在这个函数出现之前，我们在 WINDOWS 编程中用的是 StrCpy、lstrcpy 或者 strcpy 函数，这些函数的特点是只需要提供目标位置和源字符串。问题在于目标位置只是由一个指针指定，它的长度在函数内部无法检测。历史上，曾经出现过利用这一漏洞使缓冲区溢出而入侵计算机系统的事件，所以它们被标注为不安全的函数。

函数 StringCchCopy 用于替代上述不安全的函数，它是在头文件<strsafe.h>里声明的，没有被头文件<windows.h>连带包含进来，所以必须单独包含。该函数的第一个参数 pszDest 用于指向目标缓冲区，其类型为 STRSAFE_LPSTR，不要被迷惑了，它实际上是 char *类型的别名。在程序中，我们传递的是 emp.name，该表达式的结果是一个数组类型的左值，代表的是结构的数组成员，被转换为指向其首元素的指针。

第二个参数 cchDest 用于指定目标缓冲区的大小，其类型为 size_t，这是一种无符号整数类型的别名，我们前面曾经讲过的。该函数之所以安全，是因为可以指定目标缓冲区的大小以防溢出（写操作越界）。在程序中，我们传递的是结构成员 name 的大小。因为数组的类型是单字节的 char，所以不需要再除以元素类型的大小。

第三个参数 pszSrc 用于指定源字符串，其类型为 STRSAFE_LPCSTR，实际上，它是 const char *的别名。在程序中，我们传入的是字面串，这个字面串是数组类型的左值，被转换为指针，指向那个隐藏数组的首元素。

结构的非数组成员可直接赋值，比如表达式 emp.gender = 'F'为结构的 gender 成员赋值。表达式 emp.gender 的结果是一个左值，代表的是结构的 gender 成员。字符常量'F'的类型是 int，转换为左值的类型（char）后赋值。其他结构成员的赋值大同小异，不再赘述。

6.7.2 复合字面值

在创建或者以写模式打开一个文件后，就可以写入内容了。要写入一个文件，可以使用函数 WriteFile，其原型为：

```
BOOL WriteFile (
  HANDLE          hFile,
  LPCVOID         lpBuffer,
  DWORD           nNumberOfBytesToWrite,
  LPDWORD         lpNumberOfBytesWritten,
  LPOVERLAPPED    lpOverlapped
);
```

第一个参数 hFile 是文件的句柄，应当是函数 CreateFile 的返回值，代表一个已经创建或者打开的文件。在程序里，我们传入的是变量 hf 的值。

第二个参数 lpBuffer 是指向缓冲区的指针，其类型为 LPCVOID，它实际上是 const void *类型的别名。在这里，我们传入的是表达式& emp 的值，其类型为 stgEMP *，并自动转换为 void *类型。

函数 WriteFile 的第三个参数是 nNumberOfBytesToWrite，意思是要写入的字节数，而我们传入的是表达式 sizeof emp 的值。前面已经说过，运算符 sizeof 可用来计算结构类型或者结构变量的大小，以字节计，包括内部的填充字节。

函数 WriteFile 的第四个参数是 lpNumberOfBytesWritten，意思是实际上已经写入的字节数。该参数的类型为 LPDWORD，实际上就是一个指针类型的别名，它的定义可能是这样的：

```
typedef unsigned long int * LPDWORD;
```

显然，函数 WriteFile 希望我们传入一个指针，然后它就可以把实际写入的字节数写入这个指针所指向的变量。当 WriteFile 函数返回后，通过检查这个变量的值，我们就知道实际写入了多少内容。按照常规，上述示例程序应该这样做：

```
DWORD bwritten;
WriteFile (hf, & emp, sizeof emp, & bwritten, 0);
```

或者这样做：

```
DWORD bwritten, * pbw = & bwritten;
WriteFile (hf, & emp, sizeof emp, pbw, 0);
```

但是，考虑到我们这个程序比较简单，也不想知道到底写入了多少个字节，这个变量声明之后也没什么其他用处，故我们在程序中是这样做的：

```
WriteFile (hf, & emp, sizeof emp, & (DWORD) {0}, 0);
```

在这里，(DWORD) {0}是一个特殊的表达式，称为复合字面值，它由一个用圆括号括住的类型名，以及一个初始化器组成，其语法形式为以下之一：

(*类型名* **) { ***初始化器列表* **}**

(*类型名* **) { ***初始化器列表* **,}**

复合字面值用于创建一个没有名字的变量，“类型名”指定了该变量的类型，“初始化器列表”用于指定变量的初始值，例如(int) {0}，这就创建了一个没有名字的变量，变量的类型是 int，且被初始化为 0。以下是另外几个复合字面值的例子：

```
(char) {'\0'}
(char []) {'h', 'e', 'l', 'l', 'o'}
(char []) {"hello, world.\n"}
(struct t {int d [3]; char a [10];}) {.d = {5, 6, 7}, .a = "Hi,Tom"}
(int *) {& (int) {0}}
```

以上，第一个复合字面值将创建一个 char 类型的变量并初始化为字符常量'\0'；第二个复合字面值将创建一个 char 类型的数组，并初始化为一系列字符常量；第三个复合字面值将创建一个 char 类型的数组，并用字面串的内容初始化。字面串本身也将创建一个不可见的数组。

第四个复合字面值将创建一个结构类型的变量，该结构类型有两个成员，一个是 int 类型的数组，另一个是 char 类型的数组，这两个数组的初始化器里都带有指示器。带有成员声明列表的结构声明一旦出现在翻译单元里，都将声明出一种新的结构类型来，所以这个复合字面值不但要创建一个无名的结构变量，还将声明出一种结构类型和一个指示这种结构类型的标记 t。

第五个复合字面值创建一个指针类型的变量，不过它的初始化器很有意思，由一元&运算符和另一个复合字面值组成。所有复合字面值都是一个左值，指示（代表）它所创建的无名变量，而且该左值的类型就是那个类型名所指示的类型。既然复合字面值(int) {0}是一个 int 类型的左值，代表那个由它创建的变量，则表达式& (int) {0}的结果就是一个指针，这个指针旋即被用来生成外层的复合字面值变量。所以，这里是两个嵌套的复合字面值，还会创建两个无名的变量。

既然复合字面值是左值，那么它就能够出现在任何需要左值的地方，例如：

```
(int) {0} = 10086;
char * pc = & (char) {0};
unsigned long int ul = sizeof (int []) {0, 1, 2, 3, 4, 5};
```

在第一条语句中，我们将 10086 赋给左值(int) {0}。但是这个左值所指示的变量是没有名字的，所以赋值之后再也没法使用。

在第二条语句中，既然表达式(char) {0}是一个左值，那我们就可以用一元&运算符得到一个指向 char 的指针，并用于初始化变量 pc。注意变量 pc 的类型也是指向 char 的指针，类型一致，没有问题。在这里，虽然左值(char) {0}所代表的变量没有名字，但变量 pc 的值指向它，那我们就可以通过变量 pc 来访问它。

在第三条语句中，左值(int []) {0, 1, 2, 3, 4, 5}是运算符 sizeof 的操作数，故这将返回由该复合字面值所创建的那个无名变量的大小，以字节计。

接着来看我们的程序，因为函数 WriteFile 的第四个参数要求是一个指向 DWORD 类型的指针，所以我们传递的是表达式& (DWORD) {0}的结果。表达式(DWORD) {0}既是一个复合字面值，也是一个左值，将创建一个无名的对象。一元&运算符作用于它，得到一个指向 DWORD 的指针。

函数 WriteFile 的最后一个参数是 lpOverlapped，意思是重叠（执行）。虽然我们用函数 WriteFile 去写入文件，但它也可以写入设备。当写入一个慢速设备时，你可以让该函数等待完成写入操作，设备确认之后再返回，但也可以不用等待设备的确认立即返回到调用者，这后一种情况称为是异步的或者重叠的，就是说主程序和写入及确认操作互相重叠进行互不干涉。

如果设备很慢，异步操作是占优势的。想象一下，如果你自己写了一个带有打印功能的文稿编辑软件，当用户要打印文稿时，同步打印操作将等待打印机完成打印任务后才开始响应用户的编辑操作，在打印完成之前，他将发现文稿编辑界面按什么键都没有反应。如果采用异步打印操作，则软件会把打印任务安排停当之后立即着手处理用户的操作，打印机自己在后台忙碌。

如果你需要一个异步的写入操作，则你必须传入一个 LPOVERLAPPED 类型的值，这种类型实际上是指向结构体的指针。好在我们用不着异步操作，也不准备过多地涉及这些无关的细节，所以在程序中直接传入一个空指针。

如果文件写入成功，函数 WriteFile 返回非零值；如果写入失败，或者正在完成异步操作，则返回 0。我们说过，异步操作时，该函数将不等待写入完成就将返回到调用者。注意，该函数的返回类型是 BOOL，在我的机器上，它是 int 类型的别名：

```
typedef int BOOL;
```

在大多数平台和编程语言中，BOOL（布尔）类型用来描述逻辑上的“真”“假”。原先在 C 语言里没有布尔类型，它是以非零值表示“真”，零值表示“假”，然而最新的 C 标准引入了它自己的布尔类型_Bool。

顺便说一下，要读取一个文件，可以使用函数 ReadFile，其原型为：

```
BOOL ReadFile (
      HANDLE          hFile,
      LPVOID          lpBuffer,
      DWORD           nNumberOfBytesToRead,
      LPDWORD         lpNumberOfBytesRead,
      LPOVERLAPPED    lpOverlapped
);
```

老实说，这些参数都是极容易理解的，毕竟它们和 WriteFile 函数的参数类似。要读取的文件应该先用 CreateFile 函数打开，并将返回的句柄传递给 hFile 参数；读取的数据放在缓冲区里，指向缓冲区的指针传递给 lpBuffer 参数；要读取的长度（字节数）传递给 nNumberOfBytesToRead 参数；实际写入的字节数由 ReadFile 函数负责填写，但填写的位置由你自己通过参数 lpNumberOfBytesRead 指定。参数 lpOverlapped 的功能和 WriteFile 一样，可以置为空指针。

和 WriteFile 函数一样，该函数的返回类型是 BOOL。如果文件读取成功，则它返回非零值；如果读取失败，或者正在完成异步操作，则返回 0。

练习 6.12：

1.编写程序,将我们写入到文件myfile3.dat 中的内容读到一个结构类型的变量中，并在调试器里观察读取的内容是否与前面写入的一致。

6.7.3 控制台 I/O 和音频播放

和 UNIX/LINUX 系统一样，WINDOWS 也使用同一种逻辑和接口来处理物理上不同的设备和文件。WINDOWS 也有自己的标准输入、标准输出和标准错误设备，通常情况下，标准输出对应着控制台屏幕。如果你想输出到控制台屏幕，只需要把标准输出的句柄当成文件句柄传递给 WriteFile 即可，该函数自会在内部进行相应的处理。然而在此之前，你必须先获得标准输出的句柄才行。

WINDOWS 提供了三个特殊的文件名，分别是“CONIN$”“CONOUT$”和“CONERR$”，通过用函数 CreateFile 打开这三个特殊的文件，你就可以得到标准输入、标准输出和标准错误的句柄，作为示例，下面的程序就演示了如何得到标准输出的句柄并用它输出一些文本信息。

```
/**********************c0613.c**********************/
# include <windows.h>

int main (void)
{
    HANDLE hstdo = CreateFile ("CONOUT$", GENERIC_WRITE, 0, \
                               NULL, OPEN_ALWAYS,\
                               FILE_ATTRIBUTE_NORMAL, NULL);
    char buf [] = "Hello world.\n";
    WriteFile (hstdo, buf, lstrlen (buf), (DWORD []) {0}, NULL);
}
```

在调用函数 WriteFile 时，所传递的第三个参数是另一个函数 lstrlen 的返回值，该函数位于动态链接库 kernel32.dll 中，用于返回一个字符串的长度，不包括末尾的空字符，其原型为：

```
int lstrlen (LPCTSTR lpString);
```

在这里还有一个重要的细节，那就是我们用复合字面值(DWORD []) {0}取代了原先的表达式& (DWORD) {0}。复合字面值(DWORD []) {0}创建了一个无名的数组变量，它只有一个 DWORD 类型的元素。因为这个复合字面值是一个数组类型的左值，所以会自动转换为指向其首元素的指针。

不得不说 WINDOWS 是一个成熟灵活的操作系统，做同一件事，它会提供多种不同的手段和方法。比如，为了获得标准输出的句柄，我们还可以调用 GetStdHandle 函数，下面的程序就演示了这种方法：

```
/**********************c0614.c**********************/
# include <windows.h>

# define PSTR "Jace,Jack,Jackie,Jackson,Jacob,Jacque.\n"

int main (void)
```

```
{
    WriteFile (GetStdHandle (STD_OUTPUT_HANDLE), PSTR, \
               lstrlen (PSTR), (DWORD []) {0}, NULL);
}
```

函数 GetStdHandle 用于获取以下三种标准设备的句柄：标准输入、标准输出和标准错误，其原型为：

```
HANDLE GetStdHandle (DWORD nStdHandle);
```

在这里，参数 nStdHandle 需要一个用整数指示的标准设备。为方便起见，WINDOWS 将它们定义为三个宏。其中，STD_OUTPUT_HANDLE（-11）是指标准输出设备，默认情况下对应于控制台屏幕缓冲区，也就是 CONOUT$。函数 GetStdHandle 返回一个标准设备的句柄，可在调用 WriteFile 函数时使用。

和前面一样，下面我们来尝试从标准输入读取字符并写入标准输出。如下面的代码所示，我们一开始首先获取标准输入和标准输出的句柄，并分别保存到变量 stdi 和 stdo。

```
/************************c0615.c**********************/
# include <windows.h>

# define PROMPT "->"

int main (void)
{
    HANDLE stdi = GetStdHandle (STD_INPUT_HANDLE);
    HANDLE stdo = GetStdHandle (STD_OUTPUT_HANDLE);
    char buf [1];
    DWORD nbread, nbwrite;

    WriteFile (stdo, PROMPT, lstrlen (PROMPT), & nbwrite, NULL);

    while (ReadFile (stdi, buf, sizeof buf, & nbread, NULL) && \
                    (nbread != 0))
        WriteFile (stdo, buf, nbread, & nbwrite, NULL);
}
```

接下来，我们先在标准输出上打印一个提示符，并调用 ReadFile 函数等待输入。然而，用户按下的字符将在计算机系统内部的缓冲区内排队，直到按下回车之后，才能允许 ReadFile 开始实际的读取操作。在所有的字符都读完之后，继续调用 ReadFile 函数时将返回非零值（为真），并且实际读取的字节数为 0。因为这个原因，程序中的 while 语句才会将循环条件设置为该函数的返回值为真，并且变量 nbread 的值不为零。

值得注意的是，变量 nbread 在同一个表达式里既被 WriteFile 函数写入，又紧接着被读取。这没有任何关系，因为我们知道，在运算符&&的左操作数和右操作数的求值之间有一个序列点。

while 语句的循环体是将刚才读取的字符写到标准输出。注意，我们在前面将数组变量 buf 的元素数量声明为 1，意思是每次读写 1 个字节。这是可以修改的，而且这种修改不影响程序的翻译和执行效果，因为我们在调用 ReadFile 时，每次读取的字节数被设置为表达式 sizeof buf 的值，而在调用 WriteFile 时，又将每次写入的字节数设置为刚才读取的字节数。

我们说过，在翻译一个程序时，GCC 会自动加入并链接到默认的导入库。但是，有些用得少的库却并不在“默认”之列，比如 WINDOWS 多媒体函数库。在这种情况下，你必须在翻译时手动指定要链接到的库。

下面的例子程序用于播放一段音乐，所以用到了函数 PlaySound，但是这个函数所在的库并不是自动添加的，需要你手工指定。先来看程序的代码：

```
/************************c0616.c**********************/
# include <windows.h>

# define MSG0 "Playing......\n"
# define MSG1 "Finished.\n"

int main (void)
{
    WriteFile (GetStdHandle (STD_OUTPUT_HANDLE), MSG0, \
               lstrlen (MSG0), (DWORD []) {0}, NULL);

    PlaySound ("zmtx.wav", NULL, SND_FILENAME | SND_SYNC);

    WriteFile (GetStdHandle (STD_OUTPUT_HANDLE), MSG1, \
               lstrlen (MSG1), (DWORD []) {0}, NULL);
}
```

该程序首先在控制台屏幕上打印一行字符“Playing......”，然后开始播放一个音乐文件，当播放结束后，打印一行字符“Finished.”，然后结束程序。播放音乐文件用的是函数 PlaySound，它位于动态链接库 winmm.dll 中，其原型为：

```
BOOL PlaySound (LPCTSTR pszSound, HMODULE hmod, DWORD fdwSound);
```

这个函数只能播放 WAV 格式的音频，这种格式是由微软公司开发的，曾经很流行，现在也很容易找到。虽然函数的声明看起来很简单，但它却能提供很多不同的播放选项。比如，它可以播放独立的波形文件（这些文件通常以.wav 为扩展名），也可以播放 WINDOWS 系统事件的声音（比如开机时欢迎界面的音乐、注销或者关机时的声音，等等），还可以播放位于内存或者位于某个可执行文件内的声音资源（WINDOWS 应用程序可以包含图标、图片和音乐等资源数据）。再有，它可以选择是否循环播放、同步播放还是异步播放。同步播放时，该函数在播放完成后才返回到调用者；异步播放时，该函数启动播放并立即返回到调用者。如果想在播放音乐的同时还能继续执行 PlaySound 后面的代码，可以选择异步播放。

播放的方式和选项是由第三个参数 fdwSound 指定的，它也决定了如何解释第一个参数 pszSound 的内容，以及是否需要第二个参数 hmod。在程序中，我们为这个参数传递了表达式 SND_FILENAME | SND_SYNC 的值。

SND_FILENAME 是一个宏，被定义为一个有特定位模式的整数，意思是第一个参数指定的是文件名，也就是说，要播放的是一个独立的文件；SND_SYNC 也是一个宏，意思是同步播放。同步播放时，函数 PlaySound 不会立即返回，而是等待播放结束才返回。相反，要想启动播放后立即返回并执行下一条语句，可以用另一个宏 SND_ASYNC 来代替它。

第一个参数 pszSound 的类型是指向字符串的指针。因为我们为第三个参数指定了 SND_FILENAME，所以这里的字面串"zmtx.wav"表示一个文件名。函数 PlaySound 只能播放 WAV 格式的音乐文件，但不支持 MP3 等其他格式。这里指定的“zmtx.wav”是我自己机器上的文件，你应该使用你机器上的音乐文件，并用它的名字来取代这个字面串的内

容。

只有在要播放的声音是某个可执行文件内的资源时，才真正需要第二个参数 hmod。这个参数的类型是 HMODULE，和 HANDLE 一样，它也是 void *类型的别名。可执行文件不但可以是指令，还可以包含图标和音频数据，这些都是可执行文件内的资源，PlaySound 函数可以播放可执行文件内的音频资源。但是现在，因为我们是要播放一个独立的文件，按要求，这个传入的参数应当是空指针 NULL。

默认情况下，导入库 libwinmm.a 是不自动提供给链接器的，所以我们必须手工指定它，下面是翻译和执行的过程：

```
D:\exampls>gcc c0616.c -o c0616.exe -lwinmm

D:\exampls>c0616
Playing......
Finished.

D:\exampls>
```

当然，在翻译这个程序时，也可以直接链接到 WINDOWS 动态链接库本身，就像这样（假定你的 WINDOWS 安装在 C 盘根目录下）：

```
D:\exampls>gcc c0616.c -o c0616.exe c:\WINDOWS\system32\winmm.dll
```

没有声音？看看你的音箱是否已经打开并且音量适中；再不行看看声卡驱动程序是否正确安装。有些 WAV 格式的文件可能不太符合标准，也有可能出现播放问题。如果不能播放，请写信告诉我，我将提供替代的解决方案。

我猜这个音乐播放器已经激发了部分同学的兴趣，他们继而会想，这个程序只能播放固定的文件 zmtx.wav，我能不能自由选择播放哪个文件呢？比如，当我在命令行运行这个音乐播放程序时，还能指定一个波形文件，就像这样：

```
D:\exampls>c0618 mixdown.wav
```

这当然是可以的。在这里，“c0618”是可执行文件名，而“mixdown.wav”是命令行参数。不管是 UNIX、LINUX、WINDOWS，还是其他平台，都支持在运行程序时提供一些命令行参数。

命令行界面是操作系统之上的一个用户接口，当你输入一个程序的名字，并附加了命令行参数之后，操作系统将加载并运行这个程序；与此同时，还将把命令行参数保存起来。但是，这些参数是否能够被当前程序访问，取决于函数 main 的声明形式。

6.7.4 函数 main 的定义

6.8 C 标准库

随着计算机技术的发展，C 语言也被移植到其他各种不同的平台。那边厢，美国国家标准协会 ANSI 搞了一个自己的 C 语言标准 ANSI C。后来，它被国标标准化组织 ISO 采纳，其中 ISO C 就是 ANSI C 的国际化版本。

ISO C 是 POSIX C 的一个子集，但 ISO C 在最大程度上致力于各个不同平台上的可移植性，而本书的写作也是基于 ISO C 标准。ISO C 标准不但定义了 C 语言本身，同时也定义了它自己的标准库。该标准库包含了一整套输入输出、（字符）串操纵、存储管理、数学工具，以及其他各种服务的函数。

这些精心设计的库函数可以随 C 实现一起被移植到绝大多数计算机系统中，这里面的秘密在于“同一个接口，不同的实现”。例如，标准库的函数 printf 用于打印文本到标准输

出，名字是统一的，但它的定义却因计算机系统的不同而异，在 UNIX 系统上主要是封装了系统调用，而在 WINDOWS 系统上则会使用动态链接库。然而不管怎样，使用这些库函数的程序基本不需要改动就可以在不同的计算机系统中翻译和运行。

C 标准库所提供的函数是在头文件中声明的。当然，为了易于使用，最好将被声明的函数归类。这样，头文件<stdio.h>将用于声明那些和输入输出有关的函数和类型；头文件<math.h>用于声明那些和数学有关的函数和类型；头文件<string.h>用于声明那些和字符串操纵有关的函数和类型，如此等等。另一方面，仅仅有头文件还不行，如果用到了某个库，它还必须在程序的链接阶段引入。但这不需要操心，C 实现通常会自动做这件事。

6.8.1 流
6.8.2 restrict 限定的类型
6.8.3 C 标准库的实现
6.8.4 标准输入和标准输出
6.8.5 标准 I/O 的缓冲区
6.8.6 直接的输入输出
6.8.7 格式化输出
6.8.7.1 定点数和浮点数
6.8.7.2 浮点类型和浮点常量
6.8.7.3 默认实参提升
6.8.7.4 函数 fprintf 的转换模板
6.8.7.5 认识移位运算符<<和>>
6.8.8 格式化输入
6.8.8.1 函数 fscanf 的转换模板
6.8.9 格式化输入输出的实例

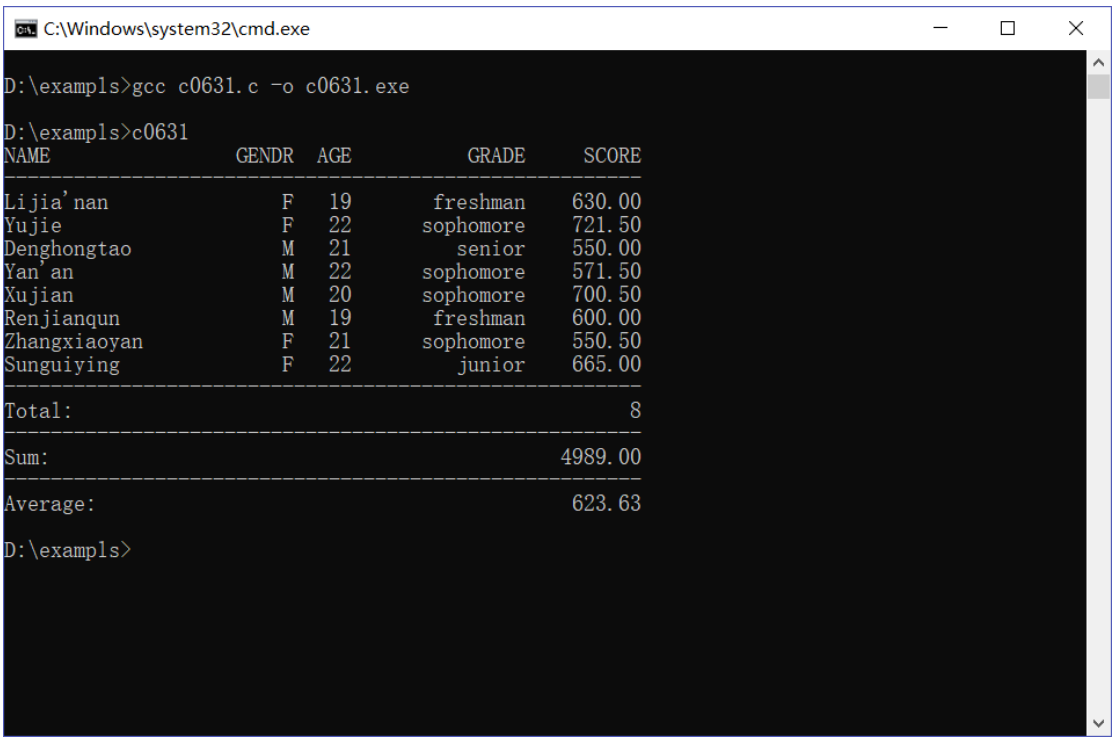


Fig.6-16 学生基本信息的打印样式

6.8.9.1 动态内存分配

6.8.9.2 认识成员选择运算符“->”