标准C语言指南

基于ISO/IEC 9899:2011

李忠 王晓波◎编著

- 基于C语言的最新标准
- 深入解析C语言的方方面面
- 以名词术语为线索的内容组织，方便查阅
- 初学者的向导，进阶提高者的知音

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

标准C语言指南

李 忠 王晓波 编著

電子工業出版社
Publishing House of Electronics Industry
北京 • BEIJING

内 容 简 介

本书基于C语言的最新标准ISO/IEC 9899:2011，力求全面介绍这门计算机语言的各个方面：词法元素、类型、声明、表达式、语句等，全书内容按概念和术语分类组织，示例丰富，查阅方便，适合具有一定C语言基础的各类专业人员和爱好者学习参考。

图书在版编目（CIP）数据

标准C语言指南 / 李忠，王晓波编著. —北京：电子工业出版社，2016.3
ISBN 978-7-121-27430-5

Ⅰ. ①标… Ⅱ. ①李… ②王… Ⅲ. ①C语言—程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字（2015）第249418号

策划编辑：董亚峰
责任编辑：郝黎明
印　　刷：北京京科印刷有限公司
装　　订：三河市华成印务有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路173信箱　邮编 100036
开　　本：787×1 092　1/16　印张：26.5　字数：634千字
版　　次：2016年3月第1版
印　　次：2016年3月第1次印刷
定　　价：58.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

服务热线：（010）88258888。

前 言

市面上和 C 语言有关的图书可以用两个成语来形容：多如牛毛、汗牛充栋。那么，在这种情况下，我为什么还要给牛增加负担呢？

首先，这本书的目标并不是教 C 语言的初学者如何入门，因为市面上并不缺乏这样的书籍，而且占了牛毛中的多数。初学者可以选择的书这么多，如果我再来一本，牛一定会很不高兴的。

其次，这本书也不是 C 语言程序设计的所谓进阶和提高教程，市面上这类书籍也有不少，但我觉得它们的关注点只集中在一些技巧方面，不够系统。

最后，我发现，不管是刚刚入门的 C 语言初学者，还是有一定编程经验的爱好者，都需要一份权威的 C 语言手册以备不时之需，有问题随时可查。以权威论，哪本书也比不了 C 语言的标准文档，但它可能过于严肃，且言简意赅。

本书的定位是一本 C 语言手册，或者说是一本 C 语言的辞典，这样就省了不少和语言无关的废话，仅集中描述语言本身。我的记性不太好，但还好为人师，喜欢在论坛里和别人讨论问题。为了说服别人，难免要引经据典，这个经典就是 C 标准。问题在于标准是英文，而且相对晦涩，如果我写了这本书，引用起来更加自如，网友们自然也很容易接受，而我自己也将十分快活。如果我自己在写程序的时候忘了有些东西该怎么用，查起来同样很方便。因此，这本书其实也是给我自己看、给我自己用的。

下面来说一说如何阅读本书。

在读这本书之前，我亲爱的读者们或多或少地读过其他一些有关 C 语言的书。但不得不说的是，很多书为了通俗易懂，在术语的使用上非常随意。因此，初次拿起这本书的时候，你可能会有些不太适应。这是很自然的，毕竟，本书选择了向标准文档靠拢，追求的目标是用严谨的文字来描述这门计算机语言。

第一个要注意的问题是“对象”这个概念。在其他有关编程的书籍中，通常使用“变量”一词，但这个概念缺乏正式的定义，而且在标准的正文里也没有出现过（仅在几个注脚里使用了几次）。本书向标准文档看齐，只使用“对象”一词。在阅读本书的其他内容之前，读者也应当先掌握这个术语。

第二个要注意的问题是，本书的内容互相交叉，每个词条（术语）都引用了更多的其他术语，这是不可避免的。因为标准文档和本书的术语可能与你看过的其他书籍略有不同，也可能使用了其他书籍中没有用过的术语，所以，在你拿到本书后，科学的做法是先浏览一下

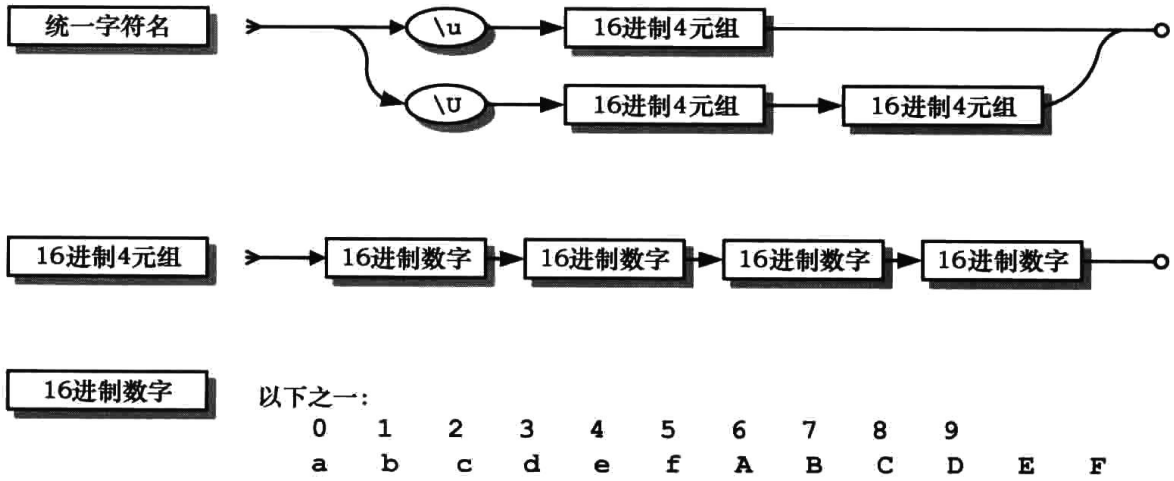
目录，看看都有哪些词条（术语），做到心中有数。当你看到本书某个部分的内容时，遇到自己不太了解的概念时，可以想起它或许在书中的其他地方有详细解释。这不是一本系统化的C语言教程，因此也不建议从头至尾按顺序阅读本书，这样做只能使你受挫。其实，最科学的方法是拿到本书后，先看你感兴趣的部分；或者，只有在你看别的教材遇到问题时再从这本书中的相应部分寻找答案。

第三个要强调的是，要养成用类型系统来解释C语言问题的习惯。

一幅图胜过千言万语，最后一个要说明的问题是如何读语法图。语法图很像铁路图，我认为它很直观，也很容易理解。如下图所示，每个语法图都有一个起始点和一个终点。读它的时候，应当从左边或者左上角开始，沿着线条（或者称为轨道）到达右边，沿途你可能遇到方框（矩形框）、圆或者椭圆。

在最左边的方框内注明了语法图的标题或者名字，即，当前的语法图用于说明谁的语法；其他方框指示一个非终结符，例如图中的“16进制4元组”和“16进制数字”。非终结符的意思是它可以继续扩展，直到能够用某个明确的内容来代替；圆形或椭圆指示一个不能够再继续扩展的、所指明确的终结符；任何沿着轨道可以走通的序列都是合法的；相反，任何沿着轨道走不通的序列都是非法和无效的。

就这个示例而言，通过语法图我们就知道，像“\u0123”和“\Uface0bed”都是合法的统一字符名。



最后，要选择一个与时俱进的编译器，但不要迷信它。编译器也可能出错，就像C标准和本书也会有需要改进的地方一样。事实上，在阅读本书的时候，经验不足的朋友所能遇到的最大障碍就是因为选用的编译器不合适而无法编译书中的示例代码。但是考虑到本书的容量和主题，这恰恰又是本书无法详细指导的内容。如果确实遇到了这方面的问题，可以写信联系我，我的电子邮件地址是 leechung@126.com。当然，最好的方法是通过我的博客与其他读者一同探讨，其链接为 http://blog.163.com/leechung@126，博客里也提供了其他一些实时交流方式。

祝各位读者阅读愉快，学习进步！

李 忠

2015年11月于长春

目 录

第1章 预备知识

尽管本书的内容适合有一定基础的编程人员阅读，但部分初学者也可用它作为参考。在这种情况下，做一些简单的铺垫，并适当地引入一些 C 语言之外的内容是大有裨益的。特别是考虑到本书的很多示例涉及输入和输出，而输入和输出并不是 C 语言的一部分，因此提前介绍这方面的内容是非常必要的。

和其他章节不同，本章的内容基本上可以按顺序阅读而不需要交叉参考其他部分，这完全贴合本章的宗旨。

1.1 源文件

编写和执行 C 程序的第一步是创建文本文件，书写程序。简单的程序可能只有一个文件；复杂一点的程序，出于模块化和便于组织管理等原因，有可能会按照功能和用途等划分为多个文件。无论如何，所有这些文件都包含了一个 C 程序的文本内容，被称为该程序的源文件（source files）。

传统上，C 程序的源文件以.c 作为扩展名。因此，作为实例，假设以下程序被保存为源文件 accu.c。

```
int main (void)
{
    int i = 1, sum = 0;
    /* 计算1~100的整数的和 */
    while (i <= 100) sum += i ++;
    return 0;
}
```

1.2 程序转换（过程）

源文件的内容是只有人类才能理解和识别的文本，不能被处理器这样的硬件识别和执行，它不可能认识这一个个字符，也不可能将它们连缀成单词和句子，更不可能分析它们的语义并采取相应的动作。处理器只能接受专门为它设计的机器指令。所以，需要将源文件转换（translate）为包含相应机器指令的程序。程序转换又称翻译，或者程序翻译。

本节的内容仅供在阅读本书的其他部分时参考，第一次阅读本章时可以略过，因为把程序转换的步骤放在这里仅仅是出于内容和术语上的关联性。

按照标准文档的规定，程序转换的阶段和步骤主要如下。

第 1 个阶段：C 实现根据自己的需要，重新映射源文件中的多字节字符。例如，在 Windows 下编辑的源文件中，换行符为\r\n，要被重新映射为\n；处理三联序列，将它们替换为当前字符集中相应的单个字符。

第 2 个阶段：源文件或者经第一阶段转换后的内容里将会有换行符（也称为新行符），它们将源文件的内容划分成行，称为物理源行。如果换行符是紧跟在反斜杠（续行符）“\”的后面，或者说，物理源行是以反斜杠和换行符结束的，则删除这两个字符，并将这一行和下一行拼接到一起，形成一个逻辑源行。这个过程是递归的：如果下一行还是以反斜杠和换行符结束，则继续拼接。但是，逻辑源行的长度不得超过 4095 个字符。

第 3 个阶段：开始为预处理做准备，主要是分解源文件，得到预处理记号、连续的空白字符和注释；每个注释都被替换为一个空格符。

第 4 个阶段：执行预处理指令。比如，预处理宏会被扩展，包括#include 在内的宏指令都被执行。对#include 指令的执行使得该指令指示的头文件或源文件也要被加工处理，而且也要经历到目前为止的 4 个处理阶段。预处理指令在执行后不再有用，会被删除。

第 5 个阶段：源文件中的字符常量和字面串也将出现在转换后的程序中（字符常量和字面串中可能含有脱转序列）。因此，这一步的工作是将它们从源字符集的成员转换为执行字符集中相应的成员。同它们在源字符集中的编码相比，转换之后可能相同，也可能不同。另一个问题是，它们在执行字符集中可能并不存在对应的成员。在这种情况下，转换后的字符代码取决于 C 的实现，但不会被转换为空字符或者空宽字符。

第 6 个阶段：将互相邻接的字面串黏接到一起，形成一个单一的字面串。有些类型的字面串不允许黏接，并将引发转换错误，请参见“字面串”。

第 7 个阶段：经上述处理后，将得到一系列由空白字符分隔的预处理记号（从现在开始，这些空白字符不再有用）。进一步地，预处理记号被转化为记号并进行语法和语义分析。这一步的成果是得到转换单元。

第 8 个阶段：现在开始进行我们通常所说的编译和链接，每个被引用的对象和函数在解析时都必须能找到它们的定义，不管是在当前转换单元，还是在另一个转换单元，甚至是在库中。最终，所有必需的部分被集中起来，形成一个程序映像，它包含了在执行环境中运行该程序所需要的所有东西和信息。

1.3 环境

C 程序的设计、转换和执行原则上是在两种数据处理系统环境下进行的，分别是转换环境和执行环境（见词条“转换环境”和“执行环境”）。

很多人觉得“数据处理系统”、“环境”这样的措辞过于生硬，不如“计算机”这样的术语通俗。事实的确如此，这是一种在正式文件中惯用的“外交辞令”。“计算机”是指 个人计算机（personal computer：PC）、工作站、服务器、手机、平板电脑、智能设备、设备控制器之类的东西；而“数据处理系统”无非也是指这些东西，但这个术语更能体现出这些设备的

内部本质：数据处理。现在，有很多人在手机和平板电脑上转换他们的 C 程序。

转换环境和执行环境可能是同一个数据处理系统环境，特别是对于正在学习 C 语言的你来说。因为你肯定是在同一台计算机上完成转换和执行工作；不过，更多的时候它们并不相同。商业软件开发者需要在他们自己的计算机系统中完成转换工作，并通过商业渠道分发到不同的用户。在这种情况下，程序的转换是在软件开发者那里，而程序的执行则是在每个用户自己的计算机上。

之所以将这两种数据处理系统环境分得这样明确，是因为不同的系统具有不同的软硬件配置。例如，不同的数据处理系统具有不同的处理器（这意味着它们的机器指令并不相同），运行着不同的操作系统（甚至没有操作系统）。一个典型的例子是，手机内部的处理器可能采用的是 ARM 架构，

而 PC 则可能用的是 Intel X86 处理器，这两种处理器的指令系统完全不同。用手机编写一个在手机上运行的软件是很麻烦的：屏幕很小，它的键盘对于编程来说也不方便。但是，可以在 PC 上建立一个仿真环境，用来编写和转换将要在手机上执行的程序。不过，转换后的结果却不能在 PC 上执行，这的确是具有讽刺意味的事情。

1.3.1 转换环境

转换需要在一个数据处理系统（你所用的个人计算机就是典型的这种数据处理系统）环境中进行，称为转换环境（translation environment）。特定的处理器架构和字符集等要素，是转换环境的组成部分。

1.3.2 执行环境

转换后的可执行程序，它赖以运行的数据处理系统（你所用的个人计算机就是典型的这种数据处理系统）环境称为执行环境（execution environment）。特定的处理器架构和字符集等要素，是执行环境的组成部分。

在编写和转换一个 C 程序之前，需要考虑它的执行环境，因为这关系到源文件的内容（程序应当如何编写），也关系到转换后的程序能否正常执行。通常有两种不同的执行环境，分别是独立式环境（freestanding environment）和宿主式环境（hosted environment）。

这两种环境的划分基于 C 在不同领域里的广泛应用。在很多情况下，可能没有操作系统，或者从操作系统那里得到的支持有限，又或者正在编写一个操作系统。此时，要求程序能独立自主地执行。在这方面，典型的例子包括仪器仪表的固件、控制器等嵌入式领域里的设备。

在宿主式环境下，程序的加载、执行和终止通常要受操作系统的控制和调度，并允许使用操作系统提供的各种功能和组件，比如文件系统。

举例来说，如果制作了一个带有处理器的数字电路，并想用 C 编写一个程序来直接驱动它工作（前提是存在一个针对该处理器的C实现，能够生成被该处理器识别和执行的机器代码），那么，这个简单的电路就是独立式环境；如果想写一个能在 Windows 下运行的窗口程序，那么，Windows 连同运行它的硬件就构成了宿主式环境。

1.4 C实现

转换过程通常需要多个阶段或者说步骤（我们经常说的编译，其实只是转换过程中的一个阶段），这就需要一套软件来分别做每个阶段的转换工作。这套特殊的软件运行在前面所说的转换环境中，将 C 的源文件转换为能在特定执行环境中运行的程序。这样的一套软件，称为 C 的一个实现（implementation），简称“C 的实现”或者“C 实现”。通俗地说，这样的软件产品“实现”或者遵循了标准对 C 语言的语法、语义规则等各个方面的描述和定义。

C 语言可以在大多数计算机架构上实现，但少数计算机只为特定的目的而存在，且它们的硬件组成和工作方式可能无法满足 C 所要求的功能和特性，在它们上面转换和执行 C 程序比较困难。在这些计算机上，可能不存在相应的 C 实现。

相对于“C 实现”，“编译器”一词用得更为广泛。考虑到编译只是转换过程的一个阶段，在某些时候容易引起误会，所以，在本书中只用“实现”这个术语。C 实现是一套软件，它自己是由其他实现转换而来的，却能“生成”软件。

C 的一个实现通常只针对某一个或者某几个特定的处理器架构和操作系统平台。无论如何，源文件只需要编写一次，然后，如果它要在某个执行环境中运行的话，只需要用为那个执行环境提供的 C 实现转换一下即可。这就是说，要想转换为能在某个设备上运行的软件，首先要问的是，针对该设备的处理器架构和操作系统平台（如果有的话），市面上有合适的 C 实现吗？C 是可移植的，但也要有（针对目标架构和平台的）相应的实现。

由于历史原因（尽管非常缓慢，但 C 的标准一直没有停止修改的步伐；同时，很多 C 实现在很久以前就存在了），并不是每一个 C 的实现都跟得上标准的步伐。现在的情况是，绝大多数 C 的实现支持 C89；很多支持 C99；C11 较新，支持它的 C 实现相对较少。

不同的实现对标准的支持程度不同，本书的绝大多数示例代码都可以用 GCC 和 CLANG 的最新版本来转换（除了几个有关多线程的示例）。

在 C 语言标准化之前，执行的事实标准是一本书，由这门语言的创建者所写，名字叫《The C Programming Language》。然后，从 20 世纪 80 年代开始，C 语言开始标准化，也有了它的国际标准，并建议所有 C 实现都依从于这个国际标准。

依从标准的 C 实现分为两种：依从标准的宿主式实现和依从标准的独立式实现，前者接受任何严格依从标准的 C 程序；而后者虽然也接受任何严格依从标准的 C 程序，但这些程序不能使用绝大多数标准库，只能使用头文件<float.h>、<iso646.h>、<limits.h>、 <stdalign.h>、<stdarg.h>、<stdbool.h>、<stddef.h>、<stdint.h>和<stdnoreturn.h>中的内容。这是可以理解的，有很多库函数，比如 printf 和 scanf 等等，都依赖于底层操作系统的支持。

1.5 诊断消息

在转换一个程序的过程中，C 实现会分析源文件的内容，如果它发现了问题，将输出一些消息以引起用户的注意，这称为诊断消息（diagnostic message）。诊断消息就是人们平时所

说的警告和错误信息。

根据问题的性质和严重程度，C 的实现会输出警告消息或者错误消息。比如，有些初学者会将 accu.c 中的 main 写成 mian，或者将“;”写成了“:”，都是不正确的。在这些情况下，转换过程将终止，不生成可执行文件，并产生诊断消息。

如何显示和输出诊断消息并不是很重要的事，标准把这个自由权交给 C 的实现。例如，多数 C 实现会显示错误出现在源文件哪一行的哪个位置。再如，CLANG 会用特殊的颜色显示“warning”以突出这是一个警告消息；默认情况下，它还用绿色显示错误处的代码，并用“^”来指示出错的位置。

1.6 转换单元

程序转换的前期工作主要是预处理。在这个过程中，要执行预处理指令，如#include、#define、#if 指令，等等。有关预处理的细节，请参见“预处理指令”。

源文件经过预处理之后就得到了转换单元。当预处理器执行完源文件中的预处理指令后，会将其删除，所以，转换单元内不再包含任何预处理指令。

举个例子，假定某源文件的全部内容如下：

```
# define PI 3.14
# define S(r) PI * (r) * (r)

int main (void)
{
    float area = S(5.0);
    /* ..... */
}
```

上面的源文件很小、很简单，而且它的预处理指令只有宏定义。所以，经过预处理之后，得到的转换单元仅仅包含以下内容（预处理指令在执行后都会被删除）。

```
int main (void)
{
    float area = 3.14 * (5.0) * (5.0);
    /* ..... */
}
```

转换单元由那些具有文件作用域的声明组成，称为外部声明。具体来讲，外部声明就是函数定义（带函数体的函数声明），以及位于任何函数定义外部的声明。这就是说，如果一个转换单元的全部内容如下，则只有那些用注释标出的部分才是外部声明。

```
int x = 0, y = 1;              //外部声明
float f (float, float);        //外部声明

int main (void)                //外部声明（函数定义）
```

```
{
    /* ..... */
}

float f (float m, float n) //外部声明（函数定义）
{
    return m * n;
}
```

相比之下，如果一个转换单元的全部内容如下，则它不是一个合法的转换单元，不可能继续执行进一步的程序转换工作。

```
int x = 0, y = 0;                    //声明，没问题
typedef int Integer;                 //声明，没问题
printf ("hello, world.\n");          //语句，非法
x = y + 2;                           //语句，非法
void f (int, int);                   //声明，允许
int g (void) {/* ..... */}           //函数定义，允许

int main (void)                      //main 函数的定义，合法
{
    /* ..... */
}
```

原因很简单，上述源文件经过预处理后，得到的转换单元中有多种成分，有声明，有语句，也有函数定义。但是，按规定，除了声明和函数定义外，一个转换单元的直接组成成分不能是其他任意东西（包括语句）。

1.7 输入和输出

本章开头的源文件 accu.c 在转换后可以得到能够在宿主式环境下运行的程序，但是程序运行后似乎并没有看到任何结果。

事实上，程序运行得很正常，而且计算出了 5050 这个结果，只是没有将它显示在屏幕上，或者输出到打印机上。那么，怎样让它做到这一点呢？

C 有完善的数据和控制流处理机制，但并不提供任何输入/输出手段。因此，为了实现这样的目的，往往需要借助于机器语言、汇编语言，或者调用为特定设备而编写的库函数。如果程序是在宿主式环境下运行的，那么，调用操作系统提供的例程（函数）往往是最方便的选择，有时也是唯一的选择。

为了更好地演示如何调用操作系统的功能，我们特地将工作的环境选取为大家都很熟悉的 Windows，并假定下面的程序代码已经被保存为源文件 wincon.c：

```
__stdcall void * GetStdHandle (unsigned long);
```

```
__stdcall int WriteConsoleA (void *,
                    const void *,
                    unsigned long,
                    unsigned long *,
                    void *);

int wsprintfA (char *, const char *, ...);

# define BUFF_SIZE 256
# define STD_OUTPUT -11

int main (void)
{
    int i = 1, sum = 0;
    while (i <= 100) sum += i ++;

    char msg [BUFF_SIZE];

    WriteConsoleA (GetStdHandle (STD_OUTPUT),
                   msg,
                   wsprintfA (msg, "1+2+3+...+100=%d", sum),
                   & (unsigned long) {0},
                   0);
    return 0;
}
```

这个程序要在 32 位 Windows 控制台下转换和运行，用的是 GCC 在 Windows 操作系统上的移植版本 MinGW。

通常来说，为了使用 Windows 提供的功能，应用程序需要借助于 Windows 应用程序编程接口（Application Programming Interface，API），这是一整套例程（函数）、协议和工具的集合。Windows API 是通过一套称为动态链接库（Dynamic-Link Library，DLL）的模块来实现的，DLL 包含了大量的函数和数据，可以被应用程序或者其他 DLL 使用。只要在计算机上安装了 Windows 操作系统，就会存在这些动态链接库。

为了向 Windows 控制台屏幕输出文本，可以使用函数 WriteConsoleA，它的功能是在控制台屏幕缓冲区的当前光标处写一串文本，动态链接库 kernel32.dll 中包含了该函数的代码以及其他相关信息。函数 WriteConsoleA 需要 5 个参数，分别是：

（1）一个指针，指向同控制台标准输出（一般是指屏幕）相关联的对象。该指针是通过调用另一个函数 GetStdHandle 得到的，该函数位于动态链接库 kernel32.dll 中。GetStdHandle 需要接收一个 unsigned long 类型的参数，这是一个与设备相关联的整数。在当前程序中，该

参数的值是-11（unsigned long 是无符号整数类型，但为它指定了一个负值。此时，将用“比 unsigned long 类型的最大值大 1 的数”加上-11 来得到最终结果。这种类型转换的原理参见“类型转换”），指示标准输出（STD_OUTPUT）。有关标准输出的内容将在下一节讲述。

（2）一个指针，指向要输出的字符串。

（3）字符串的长度。字符串的长度通过调用另一个函数 wsprintfA 得到。 wsprintfA 的功能是格式化一个字符串（在这里是"1+2+3+...+100=%d"），原字符串中的指定部分（在这里是“%d”）会被新的内容（在这里是从 sum 的值转换而来的字符串）替换，然后得到一个新的字符串。新的字符串会被写入指定的缓冲区（在这里是指针 msg 所指向的位置），然后 wsprintfA 函数返回新字符串的长度，该函数位于动态链接库 user32.dll 中；

（4）一个指向 unsigned long 类型的指针，用于保存实际写入到控制台标准输出的字符数量，由 WriteConsoleA 函数在返回之前填写。

（5）保留的参数，必须是空指针。

在以上程序中，两个“#define”引导的行是预处理指令，用于定义预处理宏。BUFF_SIZE 和 STD_OUTPUT 是宏的名字（宏名），跟在宏名后面的部分是替换列表。宏 BUFF_SIZE 用于指定存放字符串的缓冲区（数组）的大小；STD_OUTPUT 指定标准输出对应的整数值。要了解宏的作用，可以参见 “预处理指令”。

在程序转换阶段，C 实现先对 wincon.c 进行预处理。接着，翻译程序（compiler，或者称为编译器）将前一阶段得到的转换单元转换成包含机器指令的目标模块。在这一阶段中，以当前程序为例，尽管函数 GetStdHandle、WriteConsoleA 和 wsprintfA 仅仅做了声明，并未定义，即没有函数体，也不影响目标文件的生成。编译器将把它们记下来，留到下一步解决，因为它们可能在其他目标文件或库中定义。目标模块可以独立存放备用，也可以打包到某个库中共享。

最后，由链接程序（linker，或者称为链接器）将目标模块转换为可执行的程序映像，该映像同时包含着在执行环境中运行该程序所必需的信息。典型地，在一个转换单元内可能会调用其他转换单元或者库里的函数，也可以使用在其他转换单元或者库里定义的数据（以当前程序为例，GetStdHandle 等 3 个函数是在动态链接库里定义的）。在这种情况下，所有相互关联的目标模块（由它们对应的转换单元生成）必须链接到一起。如果用到了库，还必须同时链接这些库。所有这些工作完成之后，就得到最终的程序映像（这个文件像影子一样，决定了它将来执行时在内存中的布局）。

回到前面的程序中来。在 main 函数的开始部分，while 语句用于计算 1～100 的和，并调用函数 WriteConsoleA 来输出这个结果。在进入该函数之前，将首先对各个参数求值，求值的过程包括调用 GetStdHandle 和 wsprintfA 函数。

为了使用这 3 个函数，按照 C 标准的要求，首先应当声明它们，这就是源文件开头所做的工作。但是，因为仅仅声明了函数的参数类型和返回类型，而没有函数体，所以，在程序转换的链接阶段，对这些函数的调用将无法链接到它们的目标代码，从而产生诊断消息而无法完成转换工作。

程序转换的第一步是编译生成目标文件，可以使用下面的命令：

```
gcc -c wincon.c
```

这就生成一个目标文件 wincon.o。尽管上述三个函数都没有定义（缺少函数体），但这不

影响目标文件的生成，寻找并链接到函数的定义是链接阶段才做的工作，链接命令如下：

```
ld c:\mingw\lib\crt2.o wincon.o -Lc:\mingw\lib\gcc\mingw32\4.7.1 -
lmingw32 -lmsvcrt --enable-stdcall-fixup c:\windows\system32\kernel32.dll
c:\windows\system32\user32.dll
```

这就完成了链接工作并生成可执行文件 a.exe（可以用-o 选项明确指定可执行文件的名字）。具体地说，链接程序将 GetStdHandle 和 WriteConsoleA 函数链接到动态链接库 kernel32.dll；将 wsprintfA 函数链接到动态链接库 user32.dll。

图 1-1 显示了编译和链接的过程，同时也给出了程序运行的效果。问题在于，我们给链接程序 ld 传递了很多参数，它们都有什么作用呢？

相对于独立式环境下运行的程序，在宿主式环境下运行的程序要复杂得多，因为它不仅仅包含了从源文件的内容所生成的代码，也包括一些和宿主（比如操作系统）交互的代码，这些代码用于做一些初始化（运行前准备）和清理工作。为此，必须将 C 的运行时目标文件 crt2.o 和库文件 libmingw32.a、libmsvcrt.a（通过-lmingw32 和-lmsvcrt 选项）一并链接到最终的可执行程序中。

如果你认为仅仅是 GetStdHandle、WriteConsoleA 和 wsprintfA 函数需要链接到 Windows 动态链接库，那就错了。为了支持在宿主环境下运行而链接进可执行程序的其他运行时文件也需要调用 Windows 动态链接库内的函数来完成必要的初始化和清理工作，而且很凑巧的是，这些函数的定义也位于动态链接库 kernel32.dll 和 user32.dll。

尽管表面上看不出来，但事实上，在将这些函数链接到动态链接库时，还有一个函数调用约定（calling conventions）的问题。函数调用约定决定了调用一个函数时的参数传递顺序和参数传递方式（是使用栈还是寄存器来传递，或者兼而有之）、返回值放在哪里，以及由谁来维护栈的平衡，等等。

一旦要通过栈来传递函数参数，那么，由谁来清除这些参数占用的空间（在函数返回之前或者之后使栈指针恢复到函数调用前的位置），就与调用约定有关了。大多数 Windows API 函数采用的调用约定是 stdcall，其特点是由被调用的函数来做这项工作；相反，cdecl 是 C 自己默认的调用约定，由函数的调用方来负责清理工作（这是必要的，因为 C 语言允许定义一些参数数量不定的函数，比如 printf。在这种情况下，被调用函数无法知道参数的确切数量，从而也不知道如何恢复栈平衡）。错误地指定调用约定可能会带来隐患，所以，如果一个函数的调用约定和默认的不一样，则通常应在函数声明时指定。

在当前程序中，使用了 3 个 Windows API 函数，其中，函数 GetStdHandle 和 WriteConsoleA 的调用约定是 stdcall，而函数 wsprintfA 的调用约定则是默认的 cdecl，这都是微软公司在它们的开发文档中规定的。

问题在于，对于流行的编译器来说，你在程序中起的名字和它在（转换后的）目标文件中的名字不同。也就是说，编译器暗中修改了你原先给出的名称，这种做法由来已久，在很多计算机语言中，需要使用这种机制来支持重载。对于 GCC 来说，如果一个函数的声明中指定了 stdcall，则它在目标文件中的名字会有一个后缀“@nn”，其中，nn 是将所有参数的大小加在一起得到的累加和。对于 GetStdHandle 函数来说，它只有一个 unsigned long 参数，假设该类型的长度是 4 个字节，那么，它在目标文件中的名称将是_GetStdHandle@4。在重载机制下，函数可以同名，但参数不同，所以这个后缀用于将每个函数调用正确地链接到它们各自

的定义。

如图 1-1 所示，我们用 nm 程序显示了目标文件 wincon.o 内的符号，凡是用 stdcall 声明的函数名都带有上述后缀。不单单如此，前面说过，在最终的可执行程序里还有其他一些需要链接到 Windows 动态链接库的函数调用，它们也同样带有后缀。

问题在于，这些函数在 Windows 动态链接库里的名字是不带后缀的。因为这个原因，我们为链接器 ld 提供了--enable-stdcall-fixup 选项，用于告诉链接器，可以将“×××”链接到“×××@nn”而不产生任何警告信息。

如图 1-1 所示，我们在链接器 ld 的参数中指定了动态链接库 kernel32.dll 和 user32.dll。如果从上述链接命令里去掉 c:\windows\system32\kernel32.dll 和 c:\windows\system32\user32.dll，则转换过程无法继续，而且出现一大堆错误信息，说是到某某函数的引用没有定义。

图 1-1　调用操作系统功能的示例

考虑到很多人很喜欢 Windows 的图形用户界面，这里再给出一个例子程序，该程序的功能是播放一段音乐，然后显示一个对话框。程序的转换和运行环境与 wincon.c 相同，都是 Windows。假定源文件为 wingui.c，要播放的波形（音乐）文件是 music.wav，该波形文件应当位于转换后的程序所在的目录。

```
typedef struct {int i;} stgHWND;

__stdcall int MessageBoxA (stgHWND *, const char *, \
                           const char *, unsigned);

__stdcall int PlaySound (const char *, stgHWND *, unsigned long);

# define SND_FILENAME 0x20000
# define MB_OK 0
# define MB_ICONINFORMATION 64
```

```
int main (void)
{
    PlaySound ("music.wav", 0, SND_FILENAME);
    MessageBoxA (0,
                 "音乐播放完毕，请单击“确定”按钮退出当前程序。",
                 "WIN32 编程示例",
                 MB_OK | MB_ICONINFORMATION);
    return 0;
}
```

函数 PlaySound 位于动态链接库 winmm.dll 中，它的功能是播放声音。该函数需要如下 3 个参数：

（1）一个指向字符串的指针，用于指定声音来源；

（2）一个指针，如果要播放的声音来自于一个程序模块内的资源数据（典型的 WIN32 程序除了指令和数据外，还可以包含图标、多媒体、菜单、对话框等资源），则它指向与该程序模块相关联的对象。在其他情况下，它应当是一个空指针；

（3）一个标志，用于控制声音的播放和指定声音的类型。在当前程序中使用的值是 0x20000（SND_FILENAME），它的意思是第一个参数指向文件名。

函数 MessageBoxA 位于动态链接库 kernel32.dll，它的功能是弹出一个可能拥有按钮、系统图标，以及一条简短消息的模态对话框。该函数需要如下 4 个参数：

（1）一个指针，如果要创建的消息框有父窗口，则指向与父窗口相关联的对象的指针。如果要创建的消息框没有父窗口，则它应为空指针；

（2）一个指针，指向要显示的消息文本（字符串）；

（3）一个指针。可以自定义消息框标题栏的内容，用这个指针指向标题栏文本（字符串）；

（4）一个标志，用于控制消息框的外观和行为，它可以是几个不同标志的合并。MB_OK 表示消息框拥有一个“确定”按钮；MB_ICONINFORMATION 用于显示一个“i”形的图标。

这个程序依然要分两步进行转换，首先是生成目标文件：

```
gcc -c -mwindows -fexec-charset=gbk wingui.c
```

这里使用了-mwindows 选项，这样更符合图形用户界面程序的标准样式，当在 Windows 资源管理器窗口里双击运行该程序时，它就不会只显示一个控制台窗口。因为程序中使用了扩展字符（汉字），所以必须使用-fexec-charset=gbk 选项明确指定执行字符集（编码），而不是使用默认的 UTF-8 编码，否则将无法正确显示中文。如果源文件 wingui.c 使用的字符集（源字符集）不是 UTF-8，则还需要使用-finput-charset 选项来指定源字符集。字符集的确是一个大问题，要是处理不得当，将很容易显示乱码。

接下来，我们使用链接器将所有必需的模块链接到一起，生成最终的可执行文件：

```
ld c:\mingw\lib\crt2.o wingui.o -Lc:\mingw\lib\gcc\mingw32\4.7.1 -lmingw32 -lmsvcrt --enable-stdcall-fixup c:\windows\system32\kernel32.dll c:\windows\system32\user32.dll c:\windows\system32\winmm.dll
```

注意，该程序不能播放 MP3 格式的文件，所以最好使用 WAV 格式的文件，如果找不到合适的文件，Windows 目录下的系统声音文件也可以。

程序的转换过程及其（在 Windows 控制台中）运行的效果如图 1-2 所示。注意，如果要在 Windows 中使用宽字符（UNICODE 字符），则程序和它的转换过程都要有所改变。程序的改变如下，显然使用了宽字符版本的 MessageBoxW 函数：

```
# include <stddef.h>

typedef struct {int i;} stgHWND;

__stdcall int MessageBoxW (stgHWND *, const wchar_t *, \
                           const wchar_t *, unsigned);

__stdcall int PlaySound (const char *, stgHWND *, unsigned long);

# define SND_FILENAME 0x20000
# define MB_OK 0
# define MB_ICONINFORMATION 64

int main (void)
{
    PlaySound ("music.wav", 0, SND_FILENAME);
    MessageBoxW (0,
                 L"音乐播放完毕，请单击“确定”按钮退出当前程序。",
                 L"WIN32 编程示例",
                 MB_OK | MB_ICONINFORMATION);
    return 0;
}
```

在这里，MessageBoxW 函数的第二个和第三个参数是指向宽字符串的指针，wchar_t 类型是在头文件<stddef.h>中定义的。一旦在字面串的前面加了“L”，则是一个宽字面串，它意味着不是通常意义上的源字符集到执行字符集的转换，而是从源字符集到宽执行字符集的转换，使用宽字符的编码方案。在这里我们应当指定 UTF-16LE 以适应 Windows 的宽字符编码要求，所以你可以在编译阶段使用以下命令（链接命令不变）：

```
gcc -c -mwindows -fwide-exec-charset=UTF-16LE wingui.c
```

有关字符集和字符编码的更多信息，参见“字符集和字符编码”、“多字节字符”、“宽字符”、“字符串”、“宽字符串”，以及“字面串”和“宽字面串”；如果要了解更多 Windows 程序设计的内容，有一本非常经典的书《Programming Windows》，作者是 Charles Petzold。

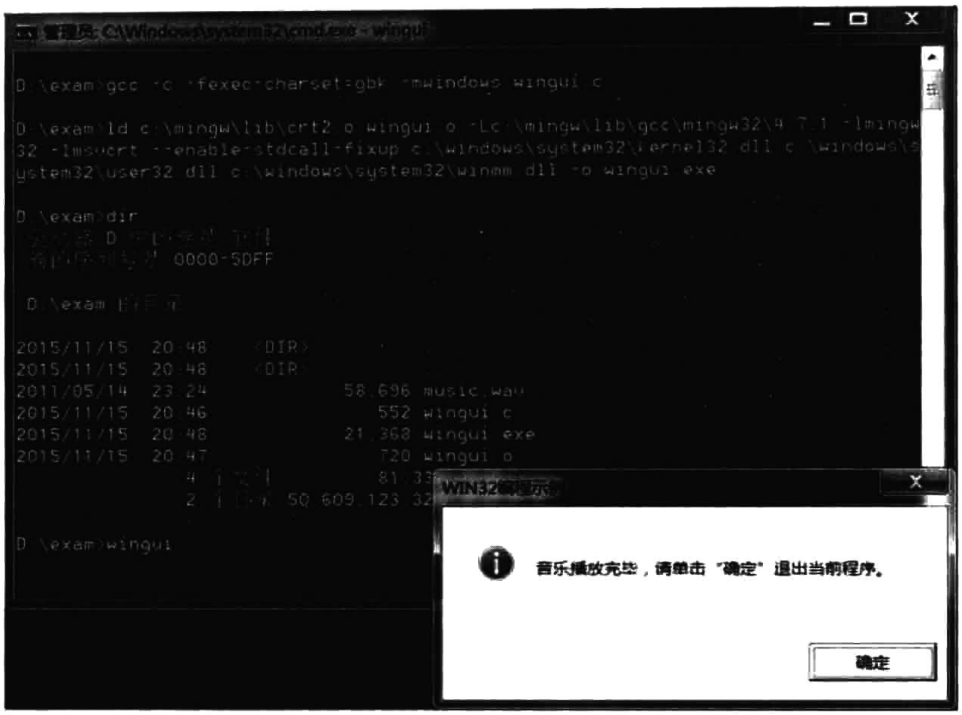


图 1-2 图形用户界面编程示例

1.8 库和头文件

在上一节里，已经接触了动态链接库。库出现和应用的历史，可能比很多人学计算机技术的时间还要长得多。一个大的程序由很多源文件共同组成，这些源文件可以分别编译成独立的目标模块，然后链接在一起形成最终的可执行程序。但是，有些目标模块具有通用性，可以用来共享，即可以在未来的某个时间根据需要链接到不同的程序中。一个公司或者个人在长期的开发活动中必定会积累大量这样的模块。问题在于，如果一个程序要使用很多这样的模块，那么，将多个模块链接到一起的过程是烦琐的，因为需要在链接它们的时候指定模块的名称（包括位置）。但是，将经过编译的模块打包进库，可以由链接器自动从库中寻找它需要的东西并实施链接，使整个过程变得简单。

例如，在上一节里，只需要在转换程序的时候指定库文件 kernel32.dll，则不管在程序中使用了哪些 API 函数，只要它们位于这个动态链接库，链接器都会自动找到它们。但是，在这个例子中，库是由 Microsoft 公司创建和维护的，而不是我们自己。这也说明了，库实际上还可以作为应用程序和操作系统之间的接口工具。

以库在程序转换期间的链接方式而言，库实际上分为两种：静态库和动态库。当链接到一个静态库时，链接器将把库中的相关部分取出，使之成为可执行程序的一部分。当程序执行时，库不再是必需的。

当链接到一个动态库时，库中的相关部分并不会成为可执行程序的一部分。相反，链接器只在转换后的程序中保留一些必要的信息，当程序执行时，才根据这些信息找到这个库，并使用它需要的部分。

如果是在 Windows 下工作，则可能不必直接链接到动态库，而使用一个和动态库相配套的静态库，称为导入库（import library）。当你在 Windows 下生成一个 DLL 文件时，可以同时生成一个 LIB 文件，这就是导入库。使用导入库的目的仅仅是帮助可执行程序在运行期间自动找到和使用动态库，所以导入库通常很小。

使用库来工作很简单。首先要声明库中的函数或者对象，然后使用它们即可。在程序转换的链接阶段，链接器会自动将它们链接到库，我们在前面已经用例子演示了具体的过程和做法。

那么，这是不是意味着人们要亲自手工声明这些函数和数据呢？不一定。如果用户非要这么做，也并没有什么不妥。但是，很多事情已经由 C 实现帮用户做完了。

以前面的例子来说，事实上，Windows API 函数已经在一大堆头（header）文件中声明过了，人们只需要用预处理指令#include 将这些文件的内容包含进源文件即可。以下是源文件 wincon.c 的改进版，假设它的文件名是 hdrcon.c。

```
# include <windows.h>

# define BUFF_SIZE 256

int main (void)
{
    int i = 1, sum = 0;
    while (i <= 100) sum += i ++;

    char msg [BUFF_SIZE];

    WriteConsoleA (GetStdHandle (STD_OUTPUT_HANDLE),
                   msg,
                   wsprintfA (msg, "1+2+3+...+100=%d", sum),
                   & (DWORD) {0},
                   NULL);
    return 0;
}
```

预处理指令#include 的功能是源文件包含，“include”后面要求指定一个头文件或者源文件的名称。在程序转换的预处理阶段，预处理器执行该预处理指令，用那个文件的实际内容来替代#include 指令。

实际上，在头文件<windows.h>中也同样用#include 指令包含了其他更多的头文件（将所有的函数都声明在同一个头文件中并不明智，最好根据函数的功能和分类，将它们声明在不同的头文件中）。这样，经预处理之后，几乎所有 Windows API 函数的声明都会出现在转换单元中。

除了函数声明外，头文件中通常还有相关的宏和类型声明。上例中的 DWORD 类型实际上是 unsigned long 的别名；NULL 是一个宏，实际上它是(void *)0。在本章中不需要过多纠缠

于类型方面的内容，类型方面的知识将在第 3 章详细阐述。

一旦包含了头文件<windows.h>，则不再需要声明它已经声明过的函数，直接在程序中调用即可。程序的转换过程和上一节相同，先是编译生成目标文件：

```
gcc -c hdrcon.c
```

接着，使用链接程序将刚才生成的目标文件和其他必需的运行时模块链接到一起，生成最终的可执行文件：

```
ld  c:\mingw\lib\crt2.o  hdrcon.o  -Lc:\mingw\lib\gcc\mingw32\4.7.1  -lmingw32  -lmsvcrt  --enable-stdcall-fixup  c:\windows\system32\kernel32.dll c:\windows\system32\user32.dll
```

实际上，如果不是希望直接链接到 Windows 的 DLL 文件，则不必这么麻烦，因为 MinGW 已经做了很多准备工作，我们可以直接一步到位：

```
gcc hdrcon.c
```

就这么简单。当然，这里有几个需要说明的问题，第一，除非指定了某些特殊的选项，如-c，则 gcc 命令将暗中调用预处理器、编译器、汇编器和链接器，最终直接生成可执行文件；第二，在链接阶段，当前程序并未直接链接到 Windows 的 DLL 文件，而是直接链接到 MinGW 提供的导入库 libkernel32.a、libuser32.a，等等。而且，在程序转换过程中，GCC 将自动引入并链接到这些导入库。

这并不是 GCC 替我们考虑得很周全。事实上，它只是在为自己考虑。我们说过，宿主式可执行程序包含了初始化和清理的内容（代码），它们也需要使用 Windows 动态链接库。所以，它会暗中自动引入它所需要的部分，比如 libkernel32.a 和 libuser32.a，等等。只不过比较凑巧的是，我们在程序中所使用的那 3 个 Windows API 函数也在这两个库中。

如下面的例子所示，如果你用到了别的库，则有可能需要自己手工引入。假定这个例子的代码被保存为源文件 hdrgui.c（上一节里有个 wingui.c，这是它的改进版）：

```
# include <windows.h>

int main (void)
{
    PlaySound ("music.wav", 0, SND_FILENAME);
    MessageBoxA (0,
                 "音乐播放完毕，请单击“确定”按钮退出当前程序。",
                 "WIN32 编程示例",
                  MB_OK | MB_ICONINFORMATION);
    return 0;
}
```

如果依赖 GCC 的默认自主行为，则按照前述方法转换时将导致失败。所以，在这种情况下，需要使用-l 选项来明确指定 PlaySound 函数所在的导入库。在机器上，这个库的全名是 libwinmm.a，所以程序的转换方法是这样的：

```
gcc hdrgui.c -o hdrgui.exe -lwinmm -fexec-charset=GBK
```

1.9 C标准库

上一节讲了 Windows 环境下的输出，但这只是为了满足读者的好奇心。Microsoft Windows 自成体系，在它上面写的图形用户界面程序不具备异种平台上的可移植性。

输入和输出并非 C 自身的一部分。为了方便地和程序的执行环境沟通，很多计算机程序设计语言都会配有某种形式的库，在这方面 C 语言也不例外。C 的维护者们很早就将一些通用性的函数分离出来，形成了独立的库。所有依从标准的 C 实现都必须实现这些库。

具体来说，C 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 实现一起被移植到任何系统中，而使用这些库函数的程序基本不需要改动就可以转移到那些系统中继续转换和运行。

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；头文件<math.h>用于声明那些和数学有关的函数；头文件<string.h>用于声明那些和字符串操作有关的函数，如此等等。事实上，除了函数声明外，头文件通常还会包括这些函数需要的类型和宏。

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。

输入和输出并不像读者想象中的那么简单。输入输出的对象可以是一些物理设备，如磁带、终端（显示器或者老式电传打字机等）；也可以是由结构化存储设备（USB 闪存盘、硬盘、光盘等）支持的文件。无论是哪种形式的输入输出，都容易使读者联想到水流。因此，不管它们的形式如何迥异，都可以被映射为逻辑上的数据流（streams）。

C 支持两种形式的数据流映射，分别是文本流（text streams）和二进制流（binary streams）。

文本流是以行为单位的、有序的字符序列，每行由零个或多个字符，换行符是行的结束标志，它是每行的最后一个字符，也是行的组成部分。最后一行的末尾是否必须有一个换行符，和 C 实现有关，有的要求，有的不要求。

为了遵从不同的习俗和惯例，在宿主式环境中表示文本时，可以向文本流中增加、删除字符，或者对字符做一些改动。举一个比较典型的例子，UNIX 系统用“\n”表示换行，而在 Windows 里则使用两个字符“\n\r”，即回车和换行。在 Windows 系统中输出一个 C 文本流时，将“\n”转换为“\n\r”，而在输入一个文本流时，将“\n\r”转换为“\n”。

二进制流也是有序的字符序列，但用于透明地记录内部数据。“透明”意味着流中的数据将保持原样而不会被改变，无关乎具体的 C 实现。写入二进制流的数据，当它们再读出来的时候，应当是相同的，没有变化。

为了访问文件，需要将一个流与文件关联。为统一起见，也要将物理设备当做文件以方便输入和输出。将流与文件关联的方式是打开文件或者创建新文件。

当一个程序启动时，将预定义 3 个文本流，并且不需要显式地打开即可使用。它们分别是标准输入（用于读取一般性的输入，通常对应着键盘，或者其他文件和设备）、标准输出（用于写入一般性的输出，通常对应着监视器，或者其他文件和设备）和标准错误（用于写入

一般性的诊断输出，通常对应着打印机、监视器，或者其他文件和设备）。

要使用流访问文件，首先需要使用文件访问函数打开或者创建文件，同时，需要向这些函数传入包含文件名的字符串。打开或者创建成功后，将返回一个指向文件对象的指针，以后即可用这个指针来读取、写入文件，或者进行其他任何操作。

与访问一般的文件和设备不同，标准输入、标准输出和标准错误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。但是，为了用文件访问函数操纵和使用这 3 个流，也需要用指向文件对象的指针来指示它们。这样，stdin 是指向同标准输入相关联的文件对象的指针；stdout 是指向同标准输出相关联的文件对象的指针；stderr 是指向同标准错误相关联的文件对象的指针。

举一个例子，函数 fgetc 从流中取得一个字符，并将其转变为 int 类型，该函数是在头文件<stdio.h>中声明的，它需要一个指向文件对象的指针作为参数。因此，为了使用该函数，需要先用其他函数来创建或者打开文件。如，可以用 fopen 函数来打开一个文件，并且返回一个同流相关联的、文件对象的指针，即：

```
int ci = fgetc (fopen("e:\\exam\\f.txt", "r"));
```

这将打开文件 f.txt，从流中读取一个字符，转化为 int 类型，最后存储到标识符 ci 所指示的对象中。函数 fopen 的第 2 个参数是字符串，用于指定打开的方式，“r”意味着以只读的方式打开。注意，这里使用双反斜杠来分隔文件和路径。一个反斜杠指示脱转序列的开始，两个反斜杠被转换为单个反斜杠（参见“脱转序列”）。

但是，因为标准输入是预定义的，并且在程序启动后就已经自动打开，所以，如果是从标准输入取得一个字符，则可以直接这样做：

```
int ci = fgetc (stdin);
```

为了向流中写入一个字符，可以使用 fputc 函数，该函数向流中写入一个字符，它是在头文件<stdio.h>中声明的。

fputc 函数接受两个参数：第一个参数是要写入流中的字符，这个参数的类型是 int，但会被转换为 unsigned char；第二个参数是指向流的指针，如果这个流和一个位置指示器相关联，则字符被插入到流中的相应位置，否则，字符被添加在流的尾部。

在下面的例子中，函数 fopen 创建或者打开一个文件（视文件存在与否），返回一个指向流的指针。然后，函数 fputc 向流中写入一个字符“x”。

```
fputc ('x', fopen ("c:\\f.txt", "w"));
```

为了向标准输出写入一个字符，也可以这样做：

```
fputc ('x', stdout);
```

为了向流中写入一个字符串，可以使用函数 fprintf，它是在头文件<stdio.h>中声明的。该函数的参数个数不定，但第一个参数必须是同流相关联的、指向文件对象的指针，第二个参数必须是一个字符串，称为格式串，用于指定后面的其他参数在输出时如何转化。

如果是写标准输出，则 fprintf 函数可以省略一些打开或者创建文件的步骤，直接写成类似于这样的形式：

```
fprintf(stdout, "%d months one year.\n", 12);
```

其中，格式串中的“%d”意味着后面对应的参数类型是 int，而且应当将其转化为有符号的十进制形式。

下面的例子将从标准输入（通常是键盘）读取字符，然后将读取的字符代码转化为十进制的形式写到标准输出（通常是监视器），直至遇到文件结尾标志。

```
# include <stdio.h>

int main (void)
{
    int ci;
    while ((ci = fgetc (stdin)) != EOF)
        fprintf(stdout, "----------%d\n", ci);

    return 0;
}
```

其中，文件结尾标志 EOF 是一个宏（在头文件<stdio.h>中定义），将在预处理阶段被替换为一个负值，意味着已经到了文件的尾部，流中已不再有其他输入。在编者的机器上，标准输入被定向到键盘，同时按 Ctrl+Z 组合键即可得到 EOF。

如果只针对标准输入和标准输出操作，那么，这些函数还可以简化。C 语言的实用主义者提供了 getchar 和 printf 函数，前者等效于 fgetc，但只针对标准输入，其函数的声明为

```
int getchar (void);
```

后者等效于 fprintf，但只针对标准输出，其原型为

```
int printf (const char * restrict format, ...);
```

这样，上述程序可以改良为

```
# include <stdio.h>

int main (void)
{
    int ci;
    while ((ci = getchar()) != EOF)
        printf("----------%d\n", ci);

    return 0;
}
```

本书中的很多地方都用到了函数 printf。它的第一个参数是固定的：是一个指向字符串的指针。在这个字符串里，可以包含一个或多个由百分号（%）引导的转换模板部分，它们的数量和样式决定了后面还有几个参数，以及它们的类型。

例如，在下面的例子中，第一个参数里包含了两个转换模板%c 和%d，这意味着，在第一个参数后面还有两个参数，且它们的类型都要求是 int。

```
printf ("The code of '%c' is %d.\n", 'x', 'x');
```

百分号（%）被用做一个转换模板的起始标志，这将带来一个问题：我们如何输出百分号本身呢？答案是将百分号连用两次，即%%。下面是一个示例：

```
printf ("The code of '%%' is %d.\n", '%');
```

转换模板是%zu 的意思是，后面所对应的参数，其类型为 size_t（这是一种无符号整数类型，在头文件<stddef.h>中定义，sizeof 运算符的结果就是这种类型）。下面是一个例子：

```
printf ("The size of type int is %zu.\n", sizeof (int));
```

转换模板"%p "意味着对应的实际参数必须是指向 void 的指针（void *）。在这种情况下，printf 函数将以文本的方式输出指针的内容（值），但输出的格式取决于 C 实现。在下例中，printf 函数将输出对象 x 的地址，以及指针 p 的内容。

```
int x = 0, * p = & x;
printf ("%p, %p.\n", (void *) & x, (void *) p);
```

指向 void 的指针被称为通用的对象指针，如上例所示的那样，为安全起见，在传递给 printf 函数时，非指向 void 的指针要手工转换为指向 void 的指针，因为可变参数不会自动执行类型转换（详情可参见“默认参数提升”）。

转换模板"%x "或者"%X "意味着对应的实际参数必须是 unsigned int 类型，且该参数的值会被转换为无符号的十六进制形式输出。输出中的十六进制数字大于 9 时，对于前者，使用字母 a、b、c、d、e 和 f；对于后者，使用字母 A、B、C、D、E 和 F。例如：

```
printf ("%x, %X.\n", 10086u, 10010U);
```

最后，如果在 printf 函数的第一个参数里没有任何转换模板，则不必为它提供更多的参数。例如：

```
printf ("C Language Guide.\n");
```

这将仅打印字符串的内容。

C 标准库无疑是非常强大的，它足以满足用户日常工作的多数需求。当然，如果用户需要的功能标准库无法满足，也允许用户生成自己独有的库，但这已经不属于标准库的范畴。此外，要了解 C 标准库的细节，以及每个库函数的使用方法，可以阅读 P.J.Plauger 所著的《The Standard C Library》，即《C 标准库》。

第2章 基本概念

2.1 字符集和字符编码

有点基础的读者应该知道，文本在计算机内部的存储不是输入一些文字，然后起个名字保存起来那么简单。输入一个“汉”字，计算机不可能将用户所看到的那个字的形状保存在文本文件中。在计算机内部存储的任何信息以数字的形式存在，包括文字。

为什么学习 C 语言要关心字符编码的问题？这里有两个最基本的原因：第一，程序需要将它的运行结果以人类可以理解的方式输出，其主要内容是由字符组成的文本。但由于历史原因，事实上存在着很多不同种类的字符编码方案；第二，C 实现可以选择它能够识别哪些编码方案（能够在哪些编码方案下工作），编程人员应当使用 C 实现所支持的字符编码方案，而且应当了解 C 语言对字符集和字符编码的基本要求。很多人没有考虑过字符集的问题，也没有意识到它的重要性。原因很简单，这和他们工作的环境有关，很多人只在一个相对封闭的计算机系统上工作，几乎不会在不同的系统之间交换文件资料。就像一个没有走出国门的人，从来没有想过语言交流会遇到障碍。但是，对于长年和不同计算机打交道的用户来说，这些必须了解。

在电子计算机中，字符和其他任何信息一样，都是无差别的数字，这些数字只对创建或者能够识别它们的程序来说才有意义，才能正确处理和还原。正因为这样，用汉字写的文档在一个没有汉字处理能力的计算机上显示时，结果一定是令人沮丧的。但是，由于计算机领域里的先驱们没有打好底子，因此，很难给“字符集”一个准确的定义。

为了使计算机能够处理字符信息，首先要决定选取哪些字符。这样就形成了一个集合，一个表，称为字符表（character repertoire）。当然，可以认为这就是一个字符集（character set），但并非所有人都认同这个观点。

然后，对于任何一个字符表来说，它的每个字符都占有一席之地，并分配到一个唯一的数字，这称为代码点（code point 或者 code position）。注意，它仅仅是数学意义上的数字，通常和字符在字符表中的位置有关，与它在计算机内的数字表示形式、存储方法和传输无关。

最早在计算机工业中应用的，同时也是应用最广泛的字符集是美国信息交换标准代码（The American Standard Code for Information Interchange，ASCII）。ASCII 由美国国家标准协会（ANSI）设计，用来简化不同厂家的计算机和字符设备的连接任务。这些设备包括打印机、电传打字机、监视器，等等。如果存在很多计算机生产厂家，就必须制定一个大家都遵循的标准，这是非常自然的事情，否则来自各个厂家的设备无法互相协同工作。

ASCII 字符集完成于 1967 年，即使是这样一个简单的字符集，也曾在到底是使用 6 个比特、7 个比特或是 8 个比特的问题上产生过分歧。那个时候存储器的单位成本很高，最后折衷的方案就是 7 比特编码。原始的 ASCII 字符集包含了 128 个字符，包括标点符号、大写和小写的拉丁字母、数字 0～9，以及一些控制字符。也就是说，ASCII 字符集有 128 个代码点。

根据不同的需求，以及现实计算机的软硬件限制，可以用不同的方法将代码点变成字符代码，这个过程称为字符编码（character encoding）。还是以 ASCII 为例，它比较简单，字符集的规模很小，所以，直接将代码点变成等数值的 7 比特的数字即可，编码值的范围是二进制的 000 0000～011 1111，用十六进制表示就是 0x00～0x7F。

然而，一个字符集可能对应着多种不同的字符编码方法。

传统的字符编码通常是按 8 比特为一组进行，以方便存储和传输，并且不会涉及字节顺序的问题（参见本节后面和 UTF-16 和 UTF-32 编码方式有关的部分）。这是一个不可继续拆分的基本元组，所以简称 8 位元。在现实世界里，这是所有网络传输设备都支持的最小信息分组。注意，我们没有说“字符编码通常是按字节进行”，因为一个字节未必是 8 个比特，尽管在多数计算机系统上一个字节的确是 8 个比特。而且，字符的编码不依赖于现实的计算机架构。但是，当字符编码在计算机中存储时，就不得不考虑了。但也没什么大不了的，直接将 8 位元扩展到一个字节的长度即可，编码值并不会改变。ASCII 字符特殊一些，它们使用 7 比特编码，因此它的每个字符需要从 7 比特编码扩展为一个字节的长度。

对于像 ASCII 这样简单的字符集来说，单个 8 位元是足够编码它的所有字符了。但是，对于其他一些字符集来说，单个的 8 位元是不够的。汉字比较多，为了给所有汉字编码，每个汉字的字符代码可能需要 2 个连续的 8 位元，这基本上足够了。

现在，假定我们要创建一个包含所有汉字的字符集 HCS，而且这些汉字的出现顺序是按它们在日常生活中的使用频度来排列的。这里还有一个硬性要求，那就是这个字符集必须能够容纳和编码所有汉字字符，而不是像历史上的 GB2312、GBK 那样，只能容纳所有汉字的一个子集。

如果这个字符集也包含原 ASCII 中的字符，那么，每个中英文字符都有自己的代码点，再假定每个字符都使用连续的 2 个 8 位元来编码，则可以直接将代码点变成 16 位长的比特序列。例如，若字符“A”的代码点是 65，则它的字符代码是二进制的 00000000 01000001，用十六进制表示就是 0x00，0x41；若汉字“王”的代码点是 600，则其字符代码是二进制的 00000010 01011000，用十六进制表示就是 0x02，0x58。这就是字符集 HCS 的第一种编码方案，姑且称之为 HCS-1 编码。

问题在于，中英文混排在文档编辑中是很常见的事，为了能够处理英文字符，还必须包含原 ASCII 中的字符，编码也要一致，这样才会使 HCS 也能处理来自英语国家的文档、网页和其他信息。使用 HCS-1 编码只能识别用这种编码方法创建的文档，而不能识别一封来自英语系国家的邮件。原因很简单，和 ASCII 兼容的字符集使用单个 8 位编码来处理英文字符，而 HCS-1 中的英文字符都具有 2 个 8 位的编码。这样就陷入了混乱状态。

解决的办法是创建另一种字符编码方案 HCS-2，使它能够兼容 ASCII。HCS-2 使用变长编码方式，对于原 ASCII 中的字符，只使用单一的 8 位元，且字符代码相同；而对于每个汉字，则使用 3 个 8 位元。但是，如何在一长串的编码中将 ASCII 字符和汉字区分开呢？

ASCII 原本是 7 位编码，当它占据一个 8 位元时，还有一位没有使用。这样，若一个 8 位元的最高位是 0，则表示这是一个 ASCII 字符且只占据 1 个 8 位元；否则，它是某个汉字代码的一部分。为了同原 ASCII 字符区分开，也为了尽可能地将各个汉字的代码区分开，如图 2-1 所示，汉字代码的第一个 8 位元的高两位固定为“11”；其第二个 8 位元的高两位是“10”；其第三个 8 位元的高两位同样是“10”。之所以采用这样的设计，是可以很方便地找到每个汉字编码的起始位置。

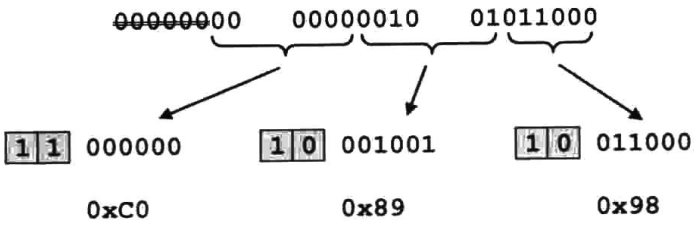


图 2-1 字符编码过程的示例

然后，每个 8 位元的空余部分用来填充汉字编码，留给汉字编码使用的比特总共是 18 个，可以编码的汉字数量也是巨大的，甚至用不完。如此，原 ASCII 中的字符“A”依然拥有 0x41 的编码，而汉字“王”的字符代码则因为这样的编码规则而变成了 0xC0，0x89，0x98。

但有人认为最常用的汉字不过几千个，这几千个汉字可以使用两个 8 位元，而不是 3 个。于是，他们希望 HCS 字符集中原 ASCII 部分使用 1 个 8 位元，常用汉字使用 2 个 8 位元，其他汉字使用 3 个 8 位元，这样可以节省存储空间。如果这种呼声足够高，那么负责 HCS 字符集工作的委员会将采纳这个建议，从而发明出第 3 种 HCS 字符集编码方案，于是该字符集将拥有 3 种完全不同的字符编码方案。

上面以汉字编码为例讲解了字符集和字符编码之间的关系。看起来挺简单的，但它们在现实世界里的历史和发展过程其实既复杂又混乱。比较早的、同时也是最成功的字符集是 ASCII（我们不讨论像莫尔期电码这样的“编码”，只将注意力局限在计算机工业里的字符编码），它在计算机业界中的地位非常牢固，是其他很多种字符集的鼻祖。

随着技术的发展，新材料的应用，以及计算机的使用逐渐向全世界扩散，各个国家和地区的人都希望能在计算机中表示自己熟悉的文字。如果想要制定一个包括全世界所有文字的字符集，现在应该是一个好时机，但这项工作需要有人来组织，还需要在沟通、协调以及字符的收集工作上花大量时间。标准的制定没有好多年才是不可思议的事情，但每个国家和地区的政府以及厂商是等不及的。

于是，每个国家和地区都各自制定了自己的字符集和编码方案，字符集和字符编码的战国时代就这样来临了。很多国家和地区的文字并不复杂，比如希腊文字和希伯来文字，他们只需要在 ASCII 的基础上简单地扩展一下就可以了。ASCII 采用 7 位编码，扩展的 ASCII 采用 8 位编码，以兼容 ASCII。因此，0x00~0x7F 之间的编码值对应 ASCII 字符集中的字符；0x80~0xFF 之间的编码值对应那些国家的本地字符。无论如何，对于这些字符集，每个字符的编码在计算机的存储器里只用一个字节就可以保存，所以历史上称为单字节字符集（single byte character set：SBCS）。

相比之下，很多国家和地区（尤其是东亚地区）的问题是字符数量非常庞大，比如汉字。在这种情况下，使用 8 位编码是远远不够的，每个字符需要更长的编码。同时，为了保持同 ASCII 的兼容性，原 ASCII 中的字符还必须只用一个 8 位，这就形成了今天所说的多字节字符。在这些国家所创建的字符集里，每个本地字符的编码在计算机的存储器里都只用两个字节来保存，所以历史上又称为双字节字符集（double byte character set：DBCS），比如中国国内的 GBK 字符集。当然，“双字节字符集”这个名字具有误导性，毕竟这样的字符集实际上是一个混合体，至少原 ASCII 字符还保持单字节不变。

单字节字符集和多字节字符集（双字节字符集）是各自为政的产物。在那个时代，如果一款软件（比如操作系统）需要面向全球发行，那它必须支持很多国家和地区的字符集和编

码方式。为了方便，每个字符集都有一个数字编号，称为代码页或者内码表。然后，根据软件发行的国家和地区，来选择当前活动的代码页。

但是，代码页是互相冲突的，除了同 ASCII 兼容的部分外，同一个编码，在不同的字符集里，对应的字符通常是不一样的。

建立一个全球性的字符集是大家的共同愿望。1991 年 1 月，在美国加利福尼亚成立了一个叫 Unicode 的协会，通过这个名字就可以知道他们的雄心壮志：制定一个全球性的字符集和编码标准。Unicode 协会有众多政府、学术机构和知名的大公司参与，这是其影响力的来源，同时也不可避免地会有大量的争吵、斗争和妥协。

在之后的岁月里，Unicode 协会与国际标准化组织 ISO 10646 工作组（SC2/WG2）通力协作，一起致力于标准的推广和各自版本的同步工作。相对于 Unicode，ISO 自己的字符集标准是 ISO10646，即统一字符集（universal character set：UCS）。当然，尽管字符的编码方式是同步进行的，但是为了确保字符在跨平台和跨应用程序时的一致性，Unicode 对标准的实现强加了更多的约束，它所做的延伸和扩展工作是 ISO/IEC 10646 所不具备的。

Unicode 编码空间是一个整数范围，从十六进制的 0 到十六进制的 10FFFF，在该编码空间内的任何一个值，都是一个代码点（代码位置）。为了描述一个代码点，可以采用“U”加十六进制整数的方法，如 U+004D。但我们说过，这并不是字符编码。

Unicode 标准的编码方式是 UTF-8、UTF-16 和 UTF-32，这三种编码方式都用于编码 U+0000~U+D7FF 以及 U+E000~U+10FFFF 的 Unicode 代码点。UTF-8 使用变长编码，对于原 ASCII 中的字符使用 8 位编码（这样它就可以兼容 ASCII 字符集）；对于其他字符（比如汉字），根据情况，使用 2 到 4 个 8 位元；UTF-16 对每个代码点（字符）使用固定的 16 位元（16 个比特）；UTF-32 则对每个代码点（字符）使用固定的 32 位元（32 个比特）。因此，这后两种编码方式无法兼容 ASCII。

表 2-1 很好地展示了 UTF-8、UTF-16 和 UTF-32 这三种编码方式的特点及它们的不同之处，这个表来自 http://www.unicode.org/versions/Unicode4.0.0/ch03.pdf#G7404。至于这三种编码方式将代码点转换为具体编码值的过程，不是本书的重点，你需要参阅相关的资料。

表 2-1　Unicode 编码方式的例子

代码点	编码方式	编码
U+004D	UTF-32	0000004D
	UTF-16	004D
	UTF-8	4D
U+0430	UTF-32	00000430
	UTF-16	0430
	UTF-8	D0　B0
U+4E8C	UTF-32	00004E8C
	UTF-16	4E8C
	UTF-8	E4　BA　8C
U+10302	UTF-32	00010302
	UTF-16	D800　DF02
	UTF-8	F0　90　8C　82

在多数计算机架构上，存储器的最小可寻址单元是字节，字的长度可以是2个字节或者4个字节，等等。访问存储器的基本粒度可以是字节，也可以是字，如果按字访问，会遇到字的哪一部分在前（低地址部分）哪一部分在后（高地址部分）的问题。举个例子来说，如果一个文件中的内容如下（这是一个字节的序列，无关紧要的部分用省略号代替）：

```
…… …… D0 B0 …… ……
```

文件的内容可以看成是它们在存储器里的映像。也就是说，如果文件的内容被加载到计算机内部的存储器，则各个字节的相对顺序保持不变。因此，我们就当上面这一行显示的是计算机存储器里的内容，而且字节序列 D0 B0 是一个 UTF-16 编码值。如果我们需要按字来访问这个编码值，那么，将这个字读取到处理器内部后，寄存器的内容在有些计算机上是D0B0，而在另一些计算机上则是 B0D0。反过来，在将一个编码值（或者任何字数据）写入存储器的时候，同样会遇到这个问题。

因为这个原因，就产生了所谓的 UTF-16、UTF-16LE、UTF-16BE、UTF-32、UTF-32LE 和 UTF-32BE 编码方案。编码方案指的是如何将编码方式串化（分解）为字节的序列，编码方式决定了字符的编码，但编码方案是编码方式的细化，它决定了编码的传输和存储格式。

注意，UTF-16 可能对应于 UTF-16LE，也可能对应于 UTF-16BE；UTF-32 可能对应于 UTF-32LE，也可能对应于 UTF-32BE。问题在于，如果编码方案是 UTF-16，那么如何知道它到底是 UTF-16LE 呢，还是 UTF-16BE？而对于 UTF-32，也同样存在这个问题。

答案是，可以使用一个字节顺序标记（byte order mark：BOM）。如果 UTF-16 对应着 UTF-16LE，则需要在整个字节序列的起始处添加字节序列 FF FE（U+FEFF）；如果 UTF-16 对应着 UTF-16BE，则需要在整个字节序列的开头添加一个初始的字节序列 FE FF（U+FEFF）。表 2-2 是几个从 UTF-16 编码方式到各个编码方案的示例，同样来自于上述网络链接。

表 2-2　UTF-16 编码在不同编码方案下的字节序列

UTF-16 编码	编码方案	字节序列
004D	UTF-16BE	00　4D
	UTF-16LE	4D　00
	UTF-16	FE　FF　00　4D　（注：实际对应 UTF-16BE） FF　FE　4D　00　（注：实际对应 UTF-16LE） 00　4D　（注：实际对应 UTF-16BE）
0430	UTF-16BE	04　30
	UTF-16LE	30　04
	UTF-16	FE　FF　04　30　（注：实际对应 UTF-16BE） FF　FE　30　04　（注：实际对应 UTF-16LE） 04　30　（注：实际对应 UTF-16BE）
4E8C	UTF-16BE	4E　8C
	UTF-16LE	8C　4E
	UTF-16	FE　FF　4E　8C　（注：实际对应 UTF-16BE） FF　FE　8C　4E　（注：实际对应 UTF-16LE） 4E　8C　（注：实际对应 UTF-16BE）

续表

UTF-16 编码	编码方案	字节序列
D800　DF02	UTF-16BE	D8　00　DF　02
	UTF-16LE	00　D8　02　DF
	UTF-16	FE　FF　D8　00　DF　02　（注：实际对应 UTF-16BE）
		FF　FE　00　D8　02　DF　（注：实际对应 UTF-16LE）
		D8　00　DF　02　（注：实际对应 UTF-16BE）

同样，如果 UTF-32 对应着 UTF-32LE，则需要在整个字节序列的起始处添加字节序列 FF FE 00 00（U+FEFF）；如果 UTF-32 对应着 UTF-32BE，则需要在整个字节序列的开头添加一个初始的字节序列 00 00 FE FF（U+FEFF）。表 2-3 是几个从 UTF-32 编码方式到各个编码方案的示例，同样来自于上述网络链接。

表 2-3　UTF-32 编码在不同编码方案下的字节序列

UTF-32 编码	编码方案	字节序列
0000004D	UTF-32BE	00　00　00　4D
	UTF-32LE	4D　00　00　00
	UTF-32	00　00　FE　FF　00　00　00　4D　（注：实际对应 UTF-32BE）
		FF　FE　00　00　4D　00　00　00　（注：实际对应 UTF-32LE）
		00　00　00　4D　（注：实际对应 UTF-32BE，无 BOM）
00000430	UTF-32BE	00　00　04　30
	UTF-32LE	30　04　00　00
	UTF-32	00　00　FE　FF　00　00　04　30　（注：实际对应 UTF-32BE）
		FF　FE　00　00　30　04　00　00　（注：实际对应 UTF-32LE）
		00　00　04　30　（注：实际对应 UTF-32BE，无 BOM）
00004E8C	UTF-32BE	00　00　4E　8C
	UTF-32LE	8C　4E　00　00
	UTF-32	00　00　FE　FF　00　00　4E　8C　（注：实际对应 UTF-32BE）
		FF　FE　00　00　8C　4E　00　00　（注：实际对应 UTF-32LE）
		00　00　4E　8C　（注：实际对应 UTF-32BE，无 BOM）
00010302	UTF-32BE	00　01　03　02
	UTF-32LE	02　03　01　00
	UTF-32	00　00　FE　FF　00　01　03　02　（注：实际对应 UTF-32BE）
		FF　FE　00　00　02　03　01　00　（注：实际对应 UTF-32LE）
		00　01　03　02　（注：实际对应 UTF-32BE，无 BOM）

BOM 不是必需的。如果仅用于内部处理，并且只使用某个固定的 Unicode 编码方案，则软件可以不使用 BOM。比如 Windows 内部使用的宽字符编码方案是 UTF-16LE，而 Linux 通常是 UTF-32LE，只需公布出来众所周知即可，可以不用 BOM。但是，如果涉及到不同软件和机器之间的文本交换与传输，这种多对多的关系无法事先统一口径，BOM 的存在就显得很有必要了。

通过以上叙述可知，UTF-8 属于典型的多字节字符编码方式（参见“多字节字符”）；用 UTF-16 和 UTF-32 编码方式得到的编码都具有固定的宽度，故属于宽字符编码方式。但是，宽字符并不是专指采用 UTF-16 或者 UTF-32 编码的字符，“宽字符”这个概念和采用哪种编

码方式无关，参见“宽字符”。

现在的问题是，为什么我们需要 UTF-16 和 UTF-32？或者更宽泛地说，为什么我们需要宽字符呢？这里面的根本原因在于其简单性。想想看，对于一个多字节字符串，要想统计它的长度（含有几个字符），首先要知道它对应的字符集（编码方式），然后，必须从头到尾一点一点地判断哪几个字节对应着一个字符。不单单是统计字符串的长度，其他各种处理工作也同样低效。相反地，如果使用宽字符，则每个字符的编码具有统一的长度，处理起来就要容易得多。而且，考虑到大家都使用了 Unicode 编码且它已经能够表示全球的字符，也就不再需要切换字符集和代码页，更省却了不少麻烦。

看起来 Unicode 终结了所有传统的字符集，我们也顺理成章地进入了字符集大一统的时代，SBCS 和 DBCS 都应当被扫入历史的垃圾堆里。但事实上，这是过度乐观了，毕竟我们还有很多历史包袱。在 Unicode 之前就已经存在的计算机主机、打印机等各种设备、数不清的软件、曾经很流行，但现在依然有人仍在使用的操作系统、硬盘里的文件和文档等等，不可能一夜之间全部消失。即使是在我写这一段文字的时候，世界上依然还有大量的、采用 GBK 字符集的文档正在被创建和使用，一个天真的、仅使用 UTF-8 或者 UTF-16，而不支持其他任何字符集和编码方式的操作系统也不可能打开这样的文档。但是我们也看到了，现实世界里并没有这样的操作系统，这很能说明问题。

另一方面，在计算机系统的里里外外全盘采用宽字符编码方案也不现实，东京和汉城的白领当然会乐见其成，但华盛顿、伦敦和巴黎的人一定会激烈反对：我为什么要多花两倍或者四倍的空间来保存我的文档？就拿本书的主角 C 语言来说，它就要求源字符集和执行字符集都必须是多字节字符集。因此，虽然很多 DBCS 会慢慢变得无足轻重，但 UTF-8 只会越来越流行。

所以，现在的情况是，拿操作系统来说，在内部统一采用 Unicode 的宽字符编码，在对外的接口部分仍支持代码页。举例来说，如果我们要处理加工某个文件，则必须先将它读入存储器。如果它采用的是多字节字符编码方案而不是宽字符编码方案，则需要在读取后转换为宽字符以方便处理。在所有的加工处理工作完成后，如果需要使用多字节字符编码方案保存或者输出，则再将宽字符转换为对应的多字节字符。

以上讲述了字符集和字符编码的关系，并介绍了全球性字符集创建的进展情况。当然，英语系的国家没有这种紧迫性，他们不用汉字也能写出一个合格的 C 程序，C 所使用的字符集也不必非得包括汉字。接下来，我们将注意力和话题转移到与 C 有关的方面。C 语言对使用何种字符集不做限定，对字符如何编码也没有强制性的要求。不过，既然是一门计算机语言，出于语言表达方面的考虑，最基本的要求还是有的。具体地说，它要求基本源字符集（参见“源字符集”）和基本执行字符集（参见“执行字符集”）都包含以下字符：

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:

```
!  "  #  %  &  '  (  )  *  +  ,  -  .  /  :
;  <  =  >  ?  [  \  ]  ^  _  {  |  }  ~
```

空格、水平制表符、垂直制表符、换页符。

本书和C标准中所说的“字母”，仅指以上所列的26个大写拉丁字母和26个小写拉丁字母。

因为C语言可以使用任何符合以上要求的字符集，所以，不要脱离C实现和具体的字符集来假定字符的编码值。例如，有很多人在没有任何前提的情况下说字符“A”的编码值是十进制的65，这是不确切的，尽管在ASCII中，字符“A”的编码值的确是65。就算是ASCII应用非常广泛，但毕竟它不是唯一的字符集，典型的就是EBCDIC，它和ASCII并不兼容，而且早在C语言标准化之前，使用EBCDIC字符集的实现就已经存在了。总之，直接用65来处理字符“A”的程序缺乏可移植性。这就是说，不建议这样写：

```
if (c == 0x65) {/* …… */}
```

而建议这样写：

```
if (c == 'A') {/* …… */}
```

基本源字符集和基本执行字符集的所有成员都要求用一个字节（byte）来表示。字节是数据存储器内的最小可寻址单元。

字节的概念是明确的，但它的长度缺乏标准定义，具体的说明参见“字节”。尽管在很多流行的系统中，一个字节的长度被视为8个比特，但是，这并不是说一个字节就是8个比特。

不管是在基本源字符集，还是在基本执行字符集里，从字符“0”开始，一直到“9”，它们的编码值是依次递增的。

这就是说，下面的代码（它的功能是打印A～Z的26个大写英文字母）是不可移植的，因为C并未规定从字符A开始，一直到Z，它们的编码值必须连续递增。当然，通常可以得到正确的结果，因为多数C实现都使用兼容ASCII的字符集。

```
# include <stdio.h>

void f (void)
{
    for (char c = 'A'; c <= 'Z'; c ++) printf ("%c\t", c);
}
```

相反，下面的代码（它的功能是打印从0到9这10个数字字符）是可移植的，因为标准已经对数字字符的连续性做了规定。

```
# include <stdio.h>

void f (void)
{
    for (char c = '0'; c <= '9'; c ++) printf ("%c\t", c);
}
```

正如已经讲过的，转换环境和执行环境并不总是相同的。因此，源字符集和执行字符集也可能不会恰好是同一个。

显而易见的是，C 实现必须同时支持源字符集和执行字符集。它支持源字符集的原因是，在转换为可执行程序之前，C 的源文件只是一堆字符。只有在“认得”这些字符的前提下，C 实现才能做分析和转换工作。

那么，C 实现为什么要支持执行字符集呢？考虑这样一个场景：很多 C 源程序会包含一些信息文本，用于运行时的加工和输出。源文件的多数内容在转换之后就消失了，特别是那些描述程序功能的部分，它们在转换之后变成了机器指令，但程序中的信息文本，诸如字符常量和字面串，都被保留，因为它们用于存储、显示、打印和传输。在这种情况下，C 实现必须将这些源字符集中的字符转换为执行字符集中的对应字符，即，将这些字符的编码换成执行环境所使用的编码。

不同的执行字符集（编码方式）将影响到可执行程序映像中的字符编码。先来看第一个示例，我们采用 UTF-8 作为执行字符集（编码），这其实是 GCC 默认使用的字符编码方案：

```
# include <stdio.h>

int main (void)
{
    char a [] = "abc 中文";
    printf ("%zu\n", sizeof a);

    return 0;
}
```

如果按下面的方式转换（假定该示例的源文件名是 exam.c 且保存时使用的字符编码方案是 UTF-8）：

```
gcc exam.c -finput-charset=utf-8 -fexec-charset=utf-8
```

在程序转换时，将把字面串"abc 中文"按照指定的编码方案转换为多字节字符的序列，并用于初始化一个不可见的静态数组（参见“字面串”）。

本例的输出是 10，因为“a”、“b”和“c”在 UTF-8 编码方案中都是 8 位，它们在当前程序中共占用 3 个字节；“中”和“文”的 UTF-8 编码都是 3 个 8 位，它们在当前程序中共占用 6 个字节；最后还有一个不可见的空字符，在当前程序中占 1 个字节。

相反地，如果采用下面的方式转换程序：

```
gcc exam.c -finput-charset=utf-8 -fexec-charset=gbk
```

那么，上述示例程序的输出是 8。原因是，“a”、“b”和“c”在 GBK 编码方案中都是 8 位，它们在当前程序中共占用 3 个字节；“中”和“文”的 GBK 编码都是 2 个 8 位，它们在当前程序中共占用 4 个字节；最后还有一个不可见的空字符，在当前程序中占 1 个字节。

在传统的 C 语言里，“字符”实际上就是单字节字符——每个字符的长度是一个字节（参见“字符”）。那个时候，C 语言还没有开始国际化。所谓国际化，指的是能够支持多国的语言、文字、习俗和惯例，但并不是说要用各个国家自己的文字（比如汉字）来书写 C 程序，而是指 C 语言能够胜任不同语言文本的输入、加工和输出任务。“国际化”在英语里是一个很

长的单词“internationalization”，全球有很多东西需要国际化，而所有参与国际化的人都觉得它实在是太长了，于是给它一个缩写“i18n”。这里，i 和 n 分别是这个单词的第一个和最后一个字母，在这两个字母中间还有 18 个字母。

非但如此，传统 C 语言的标准库也是专为单字节字符而设计。比如 strlen 函数，与其说它返回的是一个字符串中的字符个数，毋宁说它返回的是字节数（不包括最后的终止字节）。因此，C 语言国际化将面临两个基本问题：

1．所谓的“字符”，不再单纯是单字节字符，可能是用多个字节来表示的字符，比如汉字。传统的一个字节已经不能适应新的变化的要求；

2．字符串的长度需要两种统计方法：按字节数统计和按字符数统计。前者适用于传统的单字节字符，而后者适用于多字节字符。

C 语言的国际化是从 C89 开始的。首先，它支持在源字符集和执行字符集中使用多字节字符，多字节字符的编码方案也是多种多样的；其次，注释、字面串、字符常量和头文件名中也可以使用多字节字符；第三，支持宽字符常量和宽（字符）串；最后，开发了传统单字节字符操纵函数的并行版本（这些函数是在头文件<wchar.h>和<ctype.h>中声明的），用于处理多字节字符和宽字符，这就是多字节支持扩展（multibyte support extension：MSE）。

让我们先来看第一个例子，这应该是我们所熟悉的，只使用了传统 C 语言的标准库，不需要任何宽字符支持。在这里，函数 isupper 测试一个字符是否为大写字符；函数 strlen 返回一个字符串的长度（包含的字符个数）。

```
# include <stdio.h>
# include <string.h>
# include <ctype.h>

void f (void)
{
   /* Demonstration of single byte string processing */
   char a [] = "Internet Explorer";
   printf ("'%c' is a %s letter.\n", a [0], \
           isupper (a [0]) ? "uppercase" : "lowercase");
   printf ("\"%s\" contains %d characters.\n", a, strlen (a));
}
```

接下来是以上程序的宽字符版本。为了处理宽字符，需要使用传统字符处理函数的并行版本。比如函数 isupper 的宽字符版本是 iswupper；函数 printf 的宽字符版本是 wprintf；函数 strlen 的宽字符版本是 wcslen，如此等等。

```
# include <locale.h>
# include <wchar.h>
# include <wctype.h>

# if defined (PWINDOWS)
#    define LOCSTR ".936"
```

```
# elif defined (PLINUX)
#    define LOCSTR "zh_CN.UTF-8"
# else
#    define LOCSTR "C"
# endif // defined

void f (void)
{
    /* 演示宽字符串的处理 */
    setlocale (LC_ALL, LOCSTR);
    wchar_t a [] = L"IE 浏览器";
    wprintf (L" '%lc' 是一个%ls 字母。\n", \
             a [0], iswupper (a [0]) ? L"大写" : L"小写");
    wprintf (L"“%ls”包含%d 个字符。\n", a, wcslen (a));
}
```

如果要了解与宽字符有关的库函数，可以参见标准文档中与库有关的部分；如果要了解宽字符串的内容，可以参见“字面串”和“字符串”。

C 语言国际化的成果之一是引入了头文件<locale.h>和几个库函数，其中就包括 setlocale 函数，它的功能是设置或者查询与当前所在区域有关的参数，包括字符类型、时间和日期的格式，如此等等。顺便说一下，C89 的制定参照了美国标准 ANSI C，而且花了 5 年时间。问题在于欧洲人觉得这个标准过于美国化，美国人习惯于单一的语言环境和一个很小的字母表，但 C 标准应该考虑不同的地域，尤其是不同的语言文化。最终，区域设置被加入到 C 中，并导致标准的制订工作又多花了两年时间。

现在让我们来看一看这个程序在转换和执行的时候都发生了什么。首先，这个源文件在保存的时候应当使用多字节字符集，比如 GBK 或者 UTF-8。文本编辑器不认得宽字符串，这个“L”只在编译阶段起作用，所以你眼中的宽字符串 L"IE 浏览器"被保存为多字节字符的序列。在程序转换阶段，C 实现（编译器）知道 L"IE 浏览器"里的“L”是宽字符的意思，将其转换为宽字符的序列。至于转换时所用的宽字符编码方案，你可以使用一个编译选项来手工指定，如果未指定，则取决于 C 实现，C 实现会根据它所在的平台（尤其是操作系统）选取一个默认方案。

如果使用 Unicode 字符集和 Unicode 编码，宽字符的编码方案和具体使用的是哪个国家的语言文字无关，但是程序的执行和宽字符的具体操作就不能不考虑这些因素。特别是，如果需要一个从宽字符到多字节字符的转换（在本程序中，wprintf 函数就执行这种转换），则必须指定多字节字符集（代码页）。为此，程序中就需要调用 setlocale 函数来设置和区域有关的参数，很多宽字符处理函数的执行都会受这些参数的影响。以下是该函数的原型：

```
# include <locale.h>
char * setlocale (int category, const char * locale);
```

该函数的第一个参数用于指定和区域有关的项，LC_ALL 表示整个区域的所有项；第二个参数是字符串，其内容取决于 C 实现，用于选取同第一个参数相对应的值。对于 Windows

简体中文版的控制台来说，它的活动代码页是 936，实际上对应于 GBK 字符集，所以这个字符串可以是“.936”，但也可以使用“chs”或者“chinese”；对于 Linux 控制台来说，它使用的是 Unicode 字符集，采用 UTF-8 编码方案，因此这个字符串可以是“zh_CN.UTF-8”。

为了使当前程序能够在 Windows 和 Linux 上编译，同时又不用修改源程序，这里使用了预处理指令（条件编译指令）。如果已经定义了宏 PWINDOWS，则将宏 LOCSTR 定义为字符串“.936”，如果已经定义了宏 PLINUX，则将宏 LOCSTR 定义为字符串“zh_CN.UTF-8”，如果 PWINDOWS 和 PLINUX 都未定义，则将宏 LOCSTR 定义为字符串“C”，它对应着每个程序启动之后的最小环境。GCC 允许使用编译选项-D 来预定义一个可以在编译期间使用的宏，因此，假定上述程序的源文件是 exam.c，则可以在 Linux 上用下面的方法转换它：

```
gcc exam.c -fwide-exec-charset=UTF-32LE -DPLINUX
```

相对应地，如果是在 Windows 上，可以使用下面的方法：

```
gcc exam.c -fwide-exec-charset=UTF-16LE -DPWINDOWS
```

2.1.1　源字符集

本质上，组成源文件的最小成分是单个的字符，如 i、p、q，以及不可见的换行符，等等。和在显示器和打印机上看到的不同，在计算机内部，每个字符被编码为不同的比特序列。保存源文件时所使用的字符集，称为源字符集（source character set）。

在历史上，字符集和编码方式经常被混为一谈。但是，一旦指示了编码方式，也就暗示了对应的字符集。

源字符集的内容包括一个基本部分，在这部分的字符中，有些是 C 语言明确要求必须具备的字符（具体的要求可参见“字符集和字符编码”）。为明确起见，标准将这一部分内容称为基本字符集（basic character set）。源字符集中的基本字符集也可以称为基本源字符集。

除此之外，源字符集中还可以含有基本字符集没有或无法提供的扩展字符（extended characters），通常是各个国家或地区的本地字符（如汉字等）。

在程序转换阶段，如果 C 实现能够自动判断出源字符集，这当然很好，我们也乐见其成。但是，它也许不能判断出源字符集是哪一种，或者根本就不支持源文件创建时所使用的字符集。

很多 C 实现允许以明确的方式指定所使用的源字符集，但前提是你所指定的字符集是它所能够支持的。

例如，可以在 GCC 中使用一个选项-finput-charset 来指定源字符集（实际上是指定一个字符编码方案，但通常情况下该编码方案所对应的字符集也是明确的）：

```
gcc exam.c -finput-charset=gbk
```

2.1.2　执行字符集

用 C 实现转换一个程序时，源文件中的字符常量和字面串的内容都被转换为执行环境所使用的字符集的成员。毕竟，转换后的程序可能要在另一个环境中运行，那个环境有自己的字符集，并用它的字符集来处理程序中的文本。

执行环境所使用的字符集称为执行字符集（execution character set）。显然，如果程序在运行时给出的字符代码并不对应执行字符集中的任何成员，或者对应的成员不正确，我们将得

不到期望的结果（比如，不能正确显示和还原文本信息）。

执行字符集的内容包括一个基本部分，在这部分的字符中，有些是 C 语言明确要求必须具备的字符（具体的要求可参见“字符集和字符编码”）。为明确起见，标准将这一部分内容称为基本字符集。执行字符集中的基本字符集也可以称为基本执行字符集。

除此之外，执行字符集中还可以有基本字符集没有或无法提供的扩展字符，通常是各个国家或地区的本地字符（如汉字等）。

标准要求基本执行字符集中要有表示警报（alert）、退格（backspace）、回车（carriage return）和新（换）行（new line）的字符。

很多 C 实现允许以明确的方式指定所使用的执行字符集。例如，可以在 GCC 中使用一个选项-fexec-charset 来指定执行字符集（实际上是指定一个字符编码方案，但通常情况下该编码方案所对应的字符集也是明确的）。以下面的程序为例，假定该程序的源文件被保存为 exam.c：

```
# include <stdio.h>
# include <stddef.h>

void prn_code (const unsigned char * p, size_t n)
{
    for (size_t x = 0; x < n; x ++) printf ("%X\t", p [x]);
    printf ("\n");
}

int main (void)
{
    char a [] = "X轴";
    prn_code ((unsigned char *) a, sizeof a);

    return 0;
}
```

如果源文件创建时的字符集（编码）是 GBK，而我们要求的执行字符集是 UTF-8，则使用下面的方法来转换程序：

```
gcc -finput-charset=GBK -fexec-charset=UTF-8
```

相反，如果源文件创建时的字符集是 UTF-8，而我们要求的执行字符集是 GBK，则使用下面的方法来转换程序：

```
gcc -finput-charset=UTF-8 -fexec-charset=GBK
```

分别运行两个转换后的程序，你可以看到，不管是 UTF-8 还是 GBK，英文字母的编码总是一个字节；汉字“轴”的 GBK 编码是两个字节，UTF-8 编码是三个字节。

2.2 字符

在传统的 C 语言里，字符（character）是指任意一个字节里的内容，是一个有特定含义的比特序列。由于这个比特序列至少需要用一个原子粒度的存储单元（也就是字节）来承载，所以，传统 C 语言里的“字符”是指单字节字符。

但是，随着 C 语言的国际化，一个大的问题显现出来：汉语、日语、韩语，甚至拉丁文化圈的文字一个字节不够，需要用多个字节来表示，这就引入了“多字节字符”和“宽字符”的概念。因此，如果从广义上来说，字符（character）是信息的一个单位，文本数据由它们组成，通过它们，文本的内容得以展现和表示，同时它们又是对文本数据进行控制和处理的基本单位。

在本书中，“字符”的含义取决于具体的上下文语境，有时指单字节字符，有时指多字节字符或者宽字符。

2.2.1 多字节字符

源字符集和执行字符集（参见“源字符集”和“执行字符集”）实际上都包括一个基本部分，或者说一个子集，称为基本字符集。除此之外，它们还包括一些扩展字符。

扩展字符基本上都是各个国家和地区所使用的本地字符，因其数量庞大，一个字节无法表示，必须使用多个字节。

一旦决定对不同的字符使用不同数量的字节，那么，源字符集和执行字符集中的成员都将需要一个或多个字节来表示。像这样，使用一个以上的字节来表示的字符被称为多字节字符（multibyte character）。

典型的多字节字符是采用 UTF-8 编码方式表示的字符，它是 Unicode 和 ISO/IEC 10646 字符集的一种编码方式。

相对于传统的单字节字符和宽字符，多字节字符更难处理，需要额外的算法和过程。想想看，如何在一个字节流中找出哪几个字节对应于某个字符的编码？但是，因为所有设备都是和“流”打交道的，而“流”的基本成分是字节，所以这也是多字节字符的优势所在。

有关多字节字符的示例，请参见“执行字符集”。

2.2.2 宽字符

我们知道，有些国家和地区的文字比较简单，而有些国家和地区的文字很复杂，字符的数量很多。但是，与多字节字符（参见“多字节字符”）不同，另一种表示字符的方式是采用统一的长度。

问题在于多长是合适的。不管多长，总的原则是能够表示当前所在国家和地区的所有本地字符。在 C89 中定义了一个新的类型 wchar_t，它的宽度可以满足这个要求。wchar_t 是一种整数类型，在头文件<stddef.h>中定义，可能是这样的：

```
typedef unsigned short wchar_t;
```

因此，可以用 wchar_t 类型的宽度作为参照，用这个宽度来表示的字符就是宽字符（wide

character）。

宽字符比多字节字符容易处理，通常用于程序和宿主环境内部的字符表示。问题在于wchar_t类型的宽度取决于C实现。比如，在有的平台上它是16位的，而在另一些平台上它是32位的。当程序要在不同的平台之间移植时，这会造成一些困扰。

为此，C11引入两种更加明确的类型char16_t和char32_t，它们是在头文件<uchar.h>中定义的，它们都是无符号整数类型，前者的宽度保证是16个比特，后者的宽度保证是32个比特。因此，宽字符既可以用wchar_t类型来表示，也可以用char16_t或者char32_t类型来表示。

注意，如果宽字符是用wchar_t类型来表示的，则它的编码方式取决于C实现；如果宽字符是用char16_t类型来表示的，且C实现预定义了值为1的宏__STDC_UTF_16__，则宽字符的编码方式是UTF-16，如果未定义这个宏，则宽字符的实际编码方式取决于C实现；如果宽字符是用char32_t类型来表示的，且C实现预定义了值为1的宏__STDC_UTF_32__，则宽字符的编码方式是UTF-32，如果未定义这个宏，则宽字符的实际编码方式取决于C实现。

2.2.3 空字符

空字符是基本字符集中的一个（成员）字符，长度被定义为一个字节，它的所有位（比特）都是0。空字符用脱转序列\0表示。

空字符放在字符串的尾部，作为这个串的终止标记。作为一个示例，下面的代码用于统计字符串的长度，但不包括尾部的空字符：

```
int s_len (const char * s)
{
    int n = 0;
    while (* s ++ != '\0') n ++;
    return n;
}
```

2.2.4 空白（字符）

“空白字符（white-space character）”是以下字符的统称：

（1）空格，对应的字符常量为' '。

（2）水平制表符，对应的字符常量为'\t'。

（3）垂直制表符，对应的字符常量为'\v'。

（4）换（新）行符，对应的字符常量为'\n'。

（5）换页符，对应的字符常量为'\f'。

“空白（white space）”用来分隔源文件中的预处理记号，它可以是上述的任何空白字符，也可以是注释，因为每个注释在预处理阶段都被替换为一个空白字符。

有关预处理的详细描述，可参见“转换步骤”和“预处理指令”。

2.2.5 空宽字符

空宽字符（null wide character）是编码值为0的宽字符。如果宽字符是用wchar_t类型来

表示的，则空宽字符的长度取决于 C 实现，其值为 L'\0'；如果宽字符是用 char16_t 类型来表示的，则空宽字符是宽度为 16 位的整数 0，即 u'\0'；如果宽字符是用 char32_t 类型来表示的，则空宽字符是宽度为 32 位的整数 0，即 U'\0'。

下面的示例程序定义了三种不同类型的空宽字符，通过程序的输出，可以知道不同类型的空宽字符是如何编码的（建议：在转换该程序时仅根据实际情况指定源字符集，不指定执行字符集和宽字符集而使用缺省设置）：

```
# include <stdio.h>
# include <stddef.h>
# include <uchar.h>

void prn_code (const unsigned char * p, size_t n)
{
    for (size_t x = 0; x < n; x ++) printf ("%X\t", p [x]);
    printf ("\n");
}

int main (void)
{
    wchar_t a [] = {L'\0', L'x', L'\0'};
    prn_code ((unsigned char *) a, sizeof a);

    char16_t b [] = {u'\0', u'x', u'\0'};
    prn_code ((unsigned char *) b, sizeof b);

    char32_t c [] = {U'\0', U'x', U'\0' };
    prn_code ((unsigned char *) c, sizeof c);

    return 0;
}
```

2.3　统一字符名

1991 年，Unicode 和 ISO 开始合作，它们决定共同制定一套适用于多语言文本的通用编码标准。有关 Unicode 的信息，可参见“字符集和字符编码”。

为了方便引用 ISO/IEC 10646 中的字符，从 C99 开始，C 中引入了统一字符名（Universal Character Name，UCN）。如图 2-2 所示，统一字符名由“\u”引导，后跟 4 个十六进制数字，如\u00A8；或者以“\U”引导，后跟 8 个十六进制数字，如\U000CFFFD。

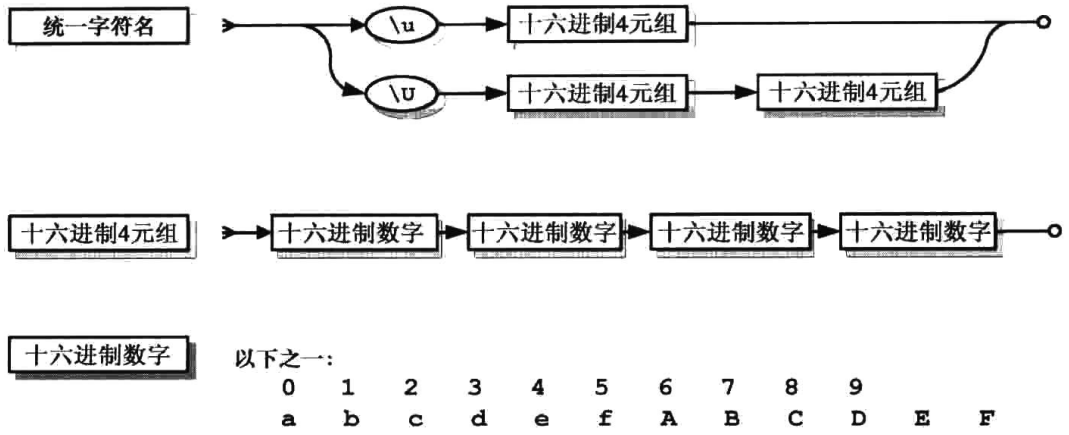


图 2-2 统一字符名语法图

"\u" 后面跟 4 个十六进制数字；"\U" 后面跟 8 个十六进制数字。不管是 4 个十六进制数字，还是 8 个十六进制数字，它们都代表 Unicode 字符集中的代码点。

某些 C 实现允许在标识符中使用（非基本字符集成员的）多字节字符。但是，具体哪些字符可以使用（这同时也决定了哪些统一字符名是可以使用的），是由实现定义的。

不同国家或者地区有不同的文化，比如，中国的程序员在写程序时会涉及一些中国独有的菜名、五行八卦、成语、武侠武功之类的名称，想把它们翻译成奇奇怪怪的英文都很费劲。在这种情况下，很多人会有直接用中文的冲动。但是，并非所有 C 实现都已经支持在标识符中使用统一字符名和自定义字符。因此，不是所有 C 实现都接受下面的声明：

```
int \u00a8\U000000aa = 1, 夫妻肺片 = 3, 囧 = 250;
```

统一字符名可以用于标识符、字符常量、字面串中，以引入非基本字符集中的字符成员。但是，并非所有的 Unicode 代码点都可以用于创建标识符。C 标准文档的附录中给出了有关这方面的详细信息。

2.4 脱转序列（转义序列）

脱转序列（escape sequences）又称转义序列，它使得字符常量或者字面串中的某些字符脱离原先的序列和解释方法，并被转换为其他字符。

无论是源字符集，还是执行字符集，总有些非图形字符难以通过键盘这类设备输入。例如，在 ASCII 字符集中，编码为 7 的字符表示警报，当终端接收到这个字符时，应该以声音或者视觉可见的形式引起操作人员的注意。但是，这个 7 不是'7'，后者在 ASCII 字符集中的编码是 55，或者十六进制 0x37。

程序在运行时，使用的是执行字符集。但是，在程序转换之前，如何用源字符集来表示这些非图形字符呢？答案是由反斜杠 "\" 引导，后面加一个或多个图形字符，用于表示执行字符集中的某些非图形字符。例如，在 C 语言中，用脱转序列\a 来表示前面所说的警报字符。

第一个要介绍的脱转序列是\'。我们知道，可以将一个字符常量赋给 char 类型的对象，方法是将它用单引号括起来，即：

```
char c = 'x';
```

如果一个字符常量就是单引号本身，这种方法不能使用了，因为它会让 C 实现感到迷惑。这时就需要使用脱转序列\'，例如：

```
char c = ''';        //非法
char c = '\'';       //正确
```

同样地，对于字面串而言，也是如此。如果需要在字面串中使用双引号本身，则需要使用脱转序列\"，这是要介绍的第二个脱转序列。例如：

```
char * p = "She said:"Yes, I do."";          //非法
char * p = "She said:\"Yes, I do.\"";        //正确
```

既然反斜杠“\”被用于引导一个脱转序列，那么如果想得到反斜杠字符本身，该怎么办呢？这就涉及要介绍的第三个脱转序列\\，这里的两个反斜杠将只能得到一个反斜杠。例如：

```
# include <stdio.h>

void f (void)
{
   printf ("The back-slash is \\");
}
```

那么，从脱转序列\\得到的 “\”不会与后面的字符一起，重新组成新的脱转序列吗？答案是不会，针对任何一个脱转序列的处理过程不是迭代的，仅仅处理一次。因此，举个例子来说，脱转序列“\\\\”不会先得到“\\\”，再得到“\\”，最后得到一个“\”；也不会先两两组合得到“\\”，再得到“\”，而是得到“\\”。

因为 C 中存在三联序列（参见“三联序列”），所以如果文本中出现了“??/”，那么，你会得到字符“\”，但如果你想得到“??/”本身，则需要将中间的问号替换为一个脱转序列\?，使得 C 实现不会将它误认为是一个三联序列。例如：

```
# include <stdio.h>

void f (void)
{
   printf ("?\?/ is ??/\");
}
```

首先，脱转序列\?表示单个问号“?”，这是我们要讲的第四个脱转序列；其次，在程序转换期间，是先处理三联序列，然后再处理脱转序列。而且，和脱转序列一样，对三联序列的处理也不是迭代的。

因为“?\?/”不是三联序列，所以，C 实现将其变为“??/”后便不再理会；后面的“??/”是三联序列，会被变为“\”。处理完三联序列后，接着处理脱转序列，于是脱转序列“\\”变成单个斜杠“\”。

如果没有最后那个“\”，会造成很多麻烦。这个反斜杠意味着它与后面的双引号一起组成一个脱转序列\"，C 的实现紧接着处理脱转序列\"，这将“删除”最后那个双引号。C 实现继续向前处理，但它已经找不到那个用于终止整个字面串的双引号。所以，如果没有最后的

那个“\”，这段代码将不能通过转换。

下面将要讲到的脱转序列和图形设备的控制有关。为了解释这些脱转序列的含义和功能，标准文档使用术语“活动位置（active position）”，它用于指示显示设备上的一个位置，下一个要显示的单字节字符将出现在这个位置上。

脱转序列\b 使得当前活动位置前边的位置成为新的活动位置。这里的“前”、“后”相对于当前的区域设置，有些国家从左向右书写文字，而有些国家从右向左书写。新的活动位置与当前活动位置相隔一个单字节字符的宽度。如果当前活动位置在当前行的行首，则显示设备采取何种动作是不知道的。

脱转序列\f 使得下一个逻辑页的首行行首成为新的活动位置。

脱转序列\n 使得下一行的行首成为新的活动位置。

脱转序列\r 使得当前行的行首成为新的活动位置。

脱转序列\t 使得活动位置水平移动一个制表位，并使其成为新的活动位置。移动方向取决于当前的区域设置。如果移动操作超出了显示设备的边界，则它会采取什么动作是不知道的。

脱转序列\v 使得活动位置垂直移动一个制表位。如果移动操作超出了显示设备的边界，则它会采取什么动作是不知道的。

脱转序列\0 表示空字符。详情可参见“空字符”。

以上所说的都是简单脱转序列，其他脱转序列还有八进制脱转序列（参见“字符常量”）、十六进制脱转序列（参见“字符常量”）和统一字符名（参见“统一字符名”）。

2.5 三联序列

C 标准允许在源文件中使用如下三个字符的序列来代替它后面的那个字符，这就是所谓的三联序列（trigraph sequences）。

```
??=    #          ??(    [          ??/    \
??)    ]          ??'    ^          ??<    {
??!    |          ??>    }          ??-    ~
```

三联序列是从 C89 开始引入的，主要是考虑到有些编程环境所使用的字符集可能无法提供全部的 29 个图形字符，且标准要求所有 C 实现支持这种替代的拼写方式。比如，有些非英语国家会使用类似 ASCII 的字符集（ISO 859），在这些字符集中，原 ASCII 中的图形字符被替换成了变音符或者其他字符。对于 C 这样大量依赖图形字符的语言来说，这就是一个很大的问题。

C 实现在将源文件转换为可执行程序时，首先要做的工作就是用对应的单个字符替换三联序列，然后才做进一步的处理和转换工作。这意味着，我们可以这样写程序：

```
??= include <stdio.h>

int main (void)
??<
```

```
        printf ("Trigraph sequences.??/n");
        return 0;
??>
```

三联序列可以用在源文件中的任何地方，在程序转换的第一个阶段，源文件中的所有三联序列都会被替换，包括字面串中的三联序列。例如：

```
# include <stdio.h>

void f (void)
{
   printf ("Are you sure ??!");
}
```

输出结果如下：

```
Are you sure |
```

但有时三联序列会和用户希望得到的内容冲突。如果用户希望输出的是??!而不是|，则必须使用某种方法来阻止它们形成三联序列，保证不改变输出的内容，此时最好的方法是使用脱转序列。例如：

```
printf ("Are you sure ?\?!");
printf ("Are you sure ??\!");
```

以上两行的输出结果都是

```
Are you sure ??!
```

2.6 （字符）串

在 C 语言里，由于历史的原因，串（string）是指单字节的、连续的字符序列，以空字符终止。因为这个空字符是串终止的标记，所以，它必须是整个串中唯一的空字符。请参见“空字符”。因为串是字符的序列，所以又称字符串（character string）。

在下例中，数组 a 内部有 5 个空字符，所以实际上可以分解为 5 个字符串，分别位于数组内部偏移为 0、5 个、10 个、17 个和 23 个字节的地方。当前程序先输出数组 a 的大小，以及第一个字符串的长度。然后，依次输出各个字符串：

```
# include <stdio.h>
# include <string.h>

void f (void)
{
    char a [] = "One.\0Two.\0Three.\0Four.\0Five.", * p = a;

    printf ("%zu, %zu\n", sizeof a, strlen (a));
```

```
    printf (p);
    printf (p + 5);
    printf (p + 10);
    printf (p + 17);
    printf (p + 23);
}
```

空字符是字符串的一部分。但是，当提到字符串的长度时，并不包括末尾的空字符，这个长度是字符串所包含的字节数。

字符串是数组类型的对象，数组的元素类型是 char。为了创建这个数组，必须在源文件中使用字面串。请参见“字面串”。

所谓指向字符串的指针，是指这样一个指针，它指向某个字符串的首字符（的地址）。

下面是字符串应用的一个示例，函数 is_keyword 接受一个指向字符串的指针，然后判断该指针所指向的字符串是否为 C 语言关键字。数组 keywords 的元素类型为指向字符的指针（char *），这些指针都是用指向字符串的指针初始化的。注意，只有在两个字符串完全相同时，指向它们的指针*p 和 q 在比较之后才会都指向字符串末尾的空字符“\0”，因此本程序将这一点做为判断两个字符串是否相同的条件和标志。

```
const char * keywords [] = {"auto", "break", "case", "char",
                            "const", "continue", "default",
                            "do", "double", "else", "enum",
                            "extern", "float", "for", "goto",
                            "if", "inline", "int", "long",
                            "register", "restrict", "return",
                             "short", "signed", "sizeof",
                            "static", "struct", "switch",
                            "typedef", "union", "unsigned",
                            "void", "volatile", "while",
                            "_Alignas", "_Alignof", "_Atomic",
                            "_Bool", "_Complex", "_Generic",
                            "_Imaginary", "_Noreturn",
                            "_Static_assert", "_Thread_local"};

_Bool is_keyword (const char * str)
{
    const char * * p = keywords, * q;

    while (q=str, p<keywords+sizeof keywords/sizeof keywords [0])
    {
        while (* * p != '\0' && * q != '\0')
            if (* * p != * q) break; else (* p) ++, q ++;
```

```
            if (* * p == '\0' && * q == '\0') return 1;
            else p ++;
        }

        return 0;
    }
```

当然，如果数组 keywords 的最后一个元素是空指针的话，则程序还可以改写如下（就不用在 while 语句中的控制表达式里计算数组的大小了）：

```
const char * keywords [] = {"auto", "break", "case", "char",
                            "const", "continue", "default",
                            "do", "double", "else", "enum",
                            "extern", "float", "for", "goto",
                            "if", "inline", "int", "long",
                            "register", "restrict", "return",
                            "short", "signed", "sizeof",
                            "static", "struct", "switch",
                            "typedef", "union", "unsigned",
                            "void", "volatile", "while",
                            "_Alignas", "_Alignof", "_Atomic",
                            "_Bool", "_Complex", "_Generic",
                            "_Imaginary", "_Noreturn",
                            "_Static_assert", "_Thread_local",
                            ""};

_Bool is_keyword (const char * str)
{
    const char * * p = keywords, * q;

    while (q = str, * p != '\0')
    {
        while (* * p != '\0' && * q != '\0')
            if (* * p != * q) break; else (* p) ++, q ++;

        if (* * p == '\0' && * q == '\0') return 1;
        else p ++;
    }

    return 0;
}
```

从广义上来说，字符串包括单字节字符串、多字节字符串和宽字符串，具体含义取决于

该术语所在的上下文。

2.6.1 多字节（字符）串

如果一个字符串实际上包含了多字节字符，是一个多字节字符的序列，则这样的字符串称为多字节（字符）串（multibyte string 或者 multibyte character string）。请参见“多字节字符”。

和字符串一样，空字符是多字节字符串的一部分，是串终止的标记，而且必须是整个串中唯一的空字符。多字节字符串的长度是该字符串所包含的字符数（而不是字节数）。

和字符串一样，多字节字符串也是数组类型的对象，数组的元素类型也是 char。为了创建这个数组，必须在源文件中使用字面串。请参见“字面串”。

除非是用带有前缀 u8 的字面串（参见“字面串”）创建，多字节字符串的编码方案由 C 实现决定。为了识别、统计和处理多字节字符串中的每个字符，需要增加额外的算法的处理过程。采用 UTF-8 编码的字符串是典型的多字节字符串。

下面我们用一个实例来加深对多字节字符串的理解。在这个例子中，函数 mstrlen 用于统计指针 p 所指向的多字节字符串中有几个字符。

```
# include <stdio.h>
# include <stddef.h>
# include <string.h>
# include <wchar.h>
# include <locale.h>

size_t mstrlen (const char * s)
{
    size_t ret = 0, cnt = 0;
    mbstate_t mbt = {0};

    while ((ret = mbrlen (s ++, 1, & mbt)) != 0 && ret != (size_t) -1)
        if (ret != (size_t) -2)  cnt ++;

    return cnt;
}

void f (void)
{
    setlocale (LC_ALL, "zh_CN.UTF-8");
    printf ("%zu\n", mstrlen (u8"嗨！Mrs Leta"));    //S1

    setlocale (LC_ALL, "zh_CN.GBK");
    printf ("%zu\n", mstrlen ("嗨！Mrs Leta"));      //S2
```

```
}
```

为了测试 mstrlen 函数，我们提供了两个多字节字符串，它们的内容完全一样，但存储时的编码可能不同。因为使用了 u8 前缀，S1 中的多字节字符串固定采用 UTF-8 编码方式。因为没有任何明确的前缀，S2 中的多字节字符串，其编码方式取决于 C 实现，但可以通过编译选项手工指定。

我们知道，多字节字符的编码取决于使用的字符集（代码页），除非知道当初编码时使用的字符集和编码方式，才能正确统计多字节字符串中的字符个数。为此，必须使用 setlocale 函数进行区域设定（有关区域设定和 setlocale 函数的信息，参见“字符集和字符编码”）。S1 中的多字节字符串固定使用 UTF-8 编码方式，所以也必须将区域设定为 zh_CN.UTF-8。统计 S2 中的多字节字符串时，我们使用的区域设定是 zh_CN.GBK，这意味着，如果使用的是 GCC 编译器，则编译时必须使用-fexec-charset=GBK 选项。这个选项不影响 S1 中的多字节字符串，因为它有 u8 前缀，但会影响到 S2 中的多字节字符串。

这个例子应当在 Linux 下转换和执行。一个可能的问题是你所使用的 Linux 尚未在 locale 中安装 GBK。函数 f 调用的结果是输出两个数字 10，如果第二个数字不是 10，则意味着你必须先安装 GBK，方法如下：

```
sudo localedef  -f GBK -i zh_CN zh_CN.GBK
```

使用 locale 命令可以检查当前所有可用的区域设定：

```
locale -a
```

没有直接用于统计多字节字符串中有几个字符的库函数，但头文件<wchar.h>中提供了库函数 mbrlen 的声明，可以调用它间接完成这一工作，其原型为：

```
# include <wchar.h>
size_t mbrlen (const char * restrict s, size_t n, \
               mbstate_t * restrict ps);
```

函数 mbrlen 从指针 s 所指向的字符串中检视最多 n 个字节，以判断形成下一个多字节字符需要几个字节。很容易想到的是，该函数依赖于当前的区域设定，尤其是当前多字节字符串所使用的字符集（代码页）。

如果该函数的返回值是 0，表示后面的 n 个或更少的字节对应着一个空字符'\0'，这通常意味着检视到了字符串的尾部；如果返回值是介于 1 到 n 之间的数值 x，表示已经用 x 个字节组成了一个多字节字符；如果返回值是(size_t)-2，意味着后面的 n 个字节不对应有效的多字节字符（有可能是 s 正指向组成一个多字节字符的那些中间字节）；如果返回值是(size_t)-1，意味着一个编码错误。

函数 mbrlen 用到了 mbstate_t 类型，它是完整的对象类型，在头文件<wchar.h>和<uchar.h>中定义，用于在字符转换过程中保存转换状态。调用 mbrlen 函数时，需要传递一个指向 mbstate_t 类型的指针。如果提供的是空指针，则 mbrlen 函数将在内部使用一个 mbstate_t 类型的静态对象，但这个内部静态对象不是线程安全的。

2.6.2 宽（字符）串

有别于单字节字符串和多字节字符串，宽（字符）串（wide string 或者 wide character string）是宽字符的序列，且以宽空字符终止。请参见“宽空字符”。

宽空字符是宽字符串的一部分。宽字符串的长度是该字符串所包含的（宽）字符数（而不是字节数）。

宽字符串也是数组类型的对象，数组的元素类型可能是 wchar_t，也可能是 char16_t 或者 char32_t。为了创建这个数组，必须在源文件中使用字面串。请参见“字面串”。

所谓指向宽字符串的指针，是指这样一个指针，它指向某个宽字符串首字符的低地址部分。

如果宽字符串的类型是 wchar_t，则它的编码方式取决于 C 实现；如果宽字符串的类型是 char16_t 且 C 实现预定义了值为 1 的宏__STDC_UTF_16__，则它的编码方式是 UTF-16，如果未定义这个宏，则它的实际编码方式取决于 C 实现；如果宽字符串的类型 char32_t 且 C 实现预定义了值为 1 的宏__STDC_UTF_32__，则它的编码方式是 UTF-32，如果未定义这个宏，则它的实际编码方式取决于 C 实现。

2.7 对象

程序总是要处理数据的。在执行环境中，可以用存储器中的某个区域来保存（写入）或者读出数据，每一个这样的数据存储区域称为一个对象（object）。

C 中的“对象”和其他面向对象的计算机语言中的“对象”没有任何关系。C 中的对象包括（但不限于）数组、数组元素、结构、结构成员、联合、联合成员、枚举、指针，以及那些用内存分配函数（比如 malloc）创建的、具有指派存储期的对象。

对象本身没有类型，但对象和一个类型或者说有效类型相关联，这个类型决定了对象在创建时要分配多少存储空间，也决定了对象的值有什么样的对象表示（参见“类型”、“有效类型”和“对象表示”）。出于这个原因，本书和 C 标准文档中所说的“对象的类型”，其完整的表达是“访问对象时使用的有效类型”。

对象有自己的值，参见“值”。为了访问对象，需要声明一个指示该对象的标识符（参见“标识符”）。如此，这个标识符就和对象建立了关联。例如：

```
int i = 3;
```

这是一个声明，声明了标识符 i，标识符 i 指示一个对象（或者换句话说，它指示存储器里的一个区域）。为了方便起见，通常将 i 所指示的对象称为“对象 i”。因为标识符 i 被声明为 int 类型，所以，对象 i 是一个 int 类型的对象，当写入对象 i 时，写入的内容应当是 int 类型的值；当读对象 i 的内容时，读出的内容应当被解释为 int 类型的值。

例如，给定以下声明，则标识符 i、sum 和它们所指示的对象之间存在如图 2-3 所示的关系。

```
int i = 1, sum = 0;
```

就像刚才所说的，在源文件中，我们用一个标识符“i”来代表暂时不存在的对象，并把它用在各种“操作”中。一旦程序开始运行，那么，对象就会真正被创建，那些操作也才真正开始进行。

注意，“指示”和“指向”是不同的。如图 2-4 所示，指示（或者说指派），说的是标识符和对象之间有一种关联和联系，是一种对应关系；指向则需要一个指针，该指针的值

被解释为对象的地址。

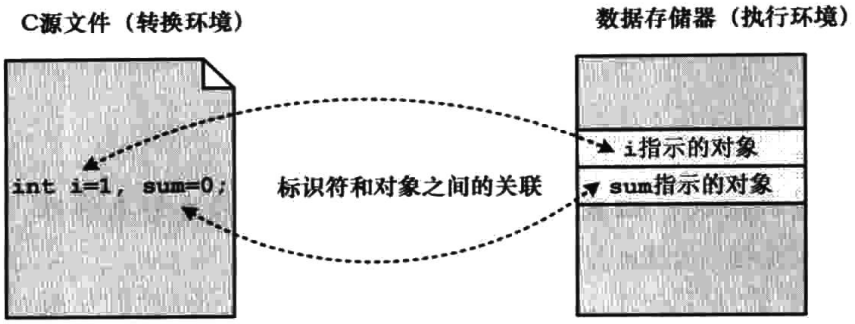


图 2-3 标识符和对象之间的关联（映射）示意图

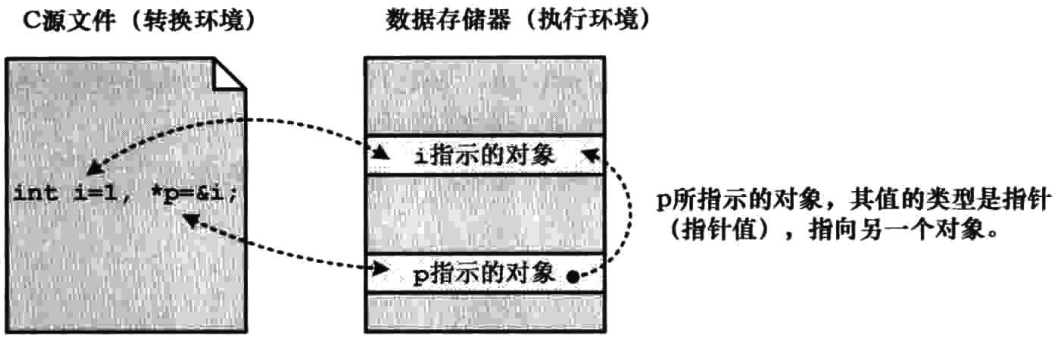


图 2-4 “指示”和“指向”的区别

对象有自己的地址和大小，有些对象因其声明时的类型而必须位于特定的地址，这称为对齐。参见“字节”、“尺寸”和“对齐”。

对象有自己的生存期（参见“生存期”），对象的生存期由它的存储期决定（参见“存储期”），对象的存储期取决于它的声明（参见“存储类指定符”和“作用域”）。

为了方便起见，我们把“标识符 x 所指示的对象”称为对象 x。在本书中可能根据实际情况交替使用这两种说法，但它们表达的是同一个意思。

标准规定，在宿主式环境下，对象的大小不能超过 65535 个字节。这意味着，例如，当我们创建一个数组时，数组的大小应限制在这个范围内；当我们使用内存分配函数 malloc 创建具有指派存储期的对象时，传递给该函数的值也应限制在这个范围内。其他类型的对象也应遵循这个限制。如果你可以用库函数 malloc 申请 3GB 的存储空间也不要高兴，这样的做法缺乏可移植性。标准的这个限制是要兼顾大多数的平台和厂商，且不会被某个 C 实现绑架。

但是，独立式环境下的 C 程序可以不遵循这个限制，如果硬件允许，它可以完全控制整个存储空间，也就是说可以创建更大的对象。在下例中，S1 是合法的，因为 sizeof 运算符的操作数并没有真正引用一个实际存在的对象；S2 是危险的，数组的大小已经超出标准的限制。

```
# include <stdio.h>

void f (void)
{
    printf ("%zu", sizeof (char [99999]));    //S1
```

```
        char a [99999] = {0};                                    //S2
        /* ..... */
    }
```

2.7.1 值

对象的内容自有它的含义，但这种含义需要用某种特定的类型来解释，并具有这种类型所要求的精确度，这就是值（value）。

举例来说，假定在某个平台上，C 实现将一个字节的长度定义为 8 个比特，unsigned short int 和 signed short int 的长度都是 2 个字节，且采用对 2 的补码表示负数。在这种情况下，如果有一个长度为 2 字节的对象，它的内容可以表示为二进制序列

```
11111111  11111111
```

那么，如果该对象的类型是 unsigned short int，则它的值是一个精确的整数 65535；如果该对象的类型是 signed short int，则它的值也是一个精确的整数-1。

对象内容的上述表示方法实际上就是值的对象表示，参见“对象表示”。

2.7.2 访问

在程序执行的时候，可能会读取对象的内容（值），也可能会修改对象的内容（值），这两种行为统称访问（access）。

特别地，如果 C 实现不求值（参见“求值”）一个表达式，那么，即使它表面上看起来会访问相关的对象，而实际的访问也不会发生。例如表达式 sizeof * p，sizeof 运算符的结果是其操作数*p 的尺寸，这个尺寸仅取决于 p 的类型而不需要访问对象 p 和 p 所指向的对象（*p）。

以上所谓的“修改（modifiy）”，既包括要保存的值和原值不同的情形，也包括要保存的值和原值相同的情形。如果要保存的值是 8，而此时对象的值已经是 8，用新 8 代替旧 8，尽管对象的值在前后没有变化，但这也算是修改。

2.7.3 存储期

对象的生存期（参见“生存期”）由它的存储期来确定。在 C 中，有 4 种存储期，分别是静态、线程、自动和指派，请参见各自的词条。

如果一个对象是由子对象组成的，如结构、联合和数组这样的对象，那么，该对象的存储期也是其所有子对象的存储期，它们是统一的。在下例中，对象 t 具有线程存储期，它的所有成员（t.c 和 t.f）也因此具有线程存储期。

```
struct t {char c; float f;};

int thrd_func (void * p)
{
    static _Thread_local struct t t;
    /* ..... */
}
```

2.7.3.1 静态存储期

如果声明了一个指示对象的标识符但没有使用存储类指定符_Thread_local，那么在下述任何一种情况下，该标识符所指示的对象具有静态存储期:

(1) 该标识符是外部链接的。

(2) 该标识符是内部链接的。

(3) 该标识符是无链接的，但在声明时使用了存储类指定符 static。

按照上述说法，在下例中，标识符 x、y 和 z 所指示的对象具有静态存储期，而 w 所指示的对象不具有静态存储期。

```
int x;                    //外部链接
static int y;             //内部链接

int f (void)
{
    int w;                //无链接
    static int z;         //使用了存储类指定符 static
    /* ..... */
}
```

特别地，以下两种类型的对象也同样具有静态存储期:

(1) 在程序转换期间，用字面串初始化的静态数组(参见“字面串”)。

(2) 由复合字面值创建的对象，且该复合字面值不在任何函数的内部。

下例中，所有复合字面值都已经用粗体标出，并分别标记为 C1、C2 和 C3，其中，"eMachine"是字面串。

```
char * pa = (char []) {"eMachine"};     //C1
int f (char *, int *, int *);

int demo (void)
{
    int * pb = & (int) {0};             //C2
    return f (pa, pb, & (int) {3});     //C3
}
```

显然，按照上面的说法，用字面串"eMachine"初始化的静态数组(在程序中看不见，可参见“字面串”)和 C1 中的复合字面值具有静态存储期； C2 和 C3 中的复合字面值只具有自动存储期。

值得注意的是，要区分作用域和存储期的不同。标识符的作用域决定了它在哪个范围内是可见的，在程序中的哪些地方可用；对象的存储期则决定了它能够存在多久。如下例所示，标识符 f 的作用域从 A 处开始，在 B 处结束，它在这个范围内可见。也就是说，这个标识符只能在这个范围内使用。但是，它所指示的对象早在整个程序启动时就已经创建并存在了。

```
int fdemo (float m)
{
```

```
        static float f /* A */ = 0.05f;
        f += m;
        /* ..... */
    /* B */}
```

具有静态存储期的对象只初始化一次，而且是在程序启动之前初始化。如果在它的声明中没有初始化器，则 C 实现将其初始化为默认值（不同类型的对象其默认值略有不同，参见“初始化”）。

下例中，标识符 c 和 d 都具有文件作用域；标识符 f 尽管只有块作用域，但它的声明包含存储类指定符 static，所以，它们所指示的对象都具有静态存储期，在整个程序启动时创建，并且只初始化一次。

```
char c;
static int d;

int fdemo (float m)
{
    static float f = 0.05f;
    f += m;
    /* ..... */
}
```

当然，在 C 中，有些东西颇具迷惑性。看起来每次调用函数 fdemo 时，都会创建对象 f，并且将它的值初始化为 0.05f，但如上所述，事实并非如此。对象 f 早在程序启动时已经创建，并且只用 0.05f 初始化一次，不管函数 fdemo 被调用多少次，都不会再用 0.05 来初始化它。

2.7.3.2 线程存储期

在多线程环境下，为了某种便利性，人们可能会让多个线程共用一个公共的对象。但这也是危险的，如果不引入一种解决竞争的机制，对象值的读取和更新将处于无序状态。举例来说，线程在进行系统调用时，系统会使用一个对象来返回本次调用的状态，而线程可以在进行下一次系统调用前随时查看这个状态。问题是，如果多个线程都使用同一个对象来保存状态信息，那么，在线程 A 做了一次系统调用并影响了对象的值，但还没有及时获取这个状态值之前，线程 B 可能也做了一次系统调用，并影响（修改）了原先的状态值。这样，线程 A 将得到一个错误的状态信息。

使用线程本地存储（thread local storage，TLS）可以解决这个问题，每个线程都使用相同的标识符来访问对象，看起来它们访问的是同一个对象。但是实际上，C 实现会为每个线程创建它们各自的副本，它们访问的是不同的对象。

在声明一个对象类型的标识符时，如果使用了存储类指定符_Thread_local，那么，该标识符所指示的对象具有线程存储期。但是，在声明这样的对象时，语法上有某些约束，具体的阐述可参见“_Thread_local”。

在下面的例子中，用同一个函数 thrd_func 创建了两个线程，它们完成同样的工作，即各自分两次计算 1～100 的整数和。第一遍累加时，使用 sum 所指示的对象保存结果，但该标识符未用存储类指定符_Thread_local 声明，而是使用限定符 volatile，以避免不必要的优化。第

二遍累加时，使用 tlssum 所指示的对象保存结果，该标识符是用存储类指定符_Thread_local 声明的。

```
# include <stdio.h>
# include <threads.h>

volatile int sum;
_Thread_local int tlssum;

int thrd_func (void * p)
{
    int x = 1;

    while (x <= 100) sum += x ++;
    printf ("no TLS:%d\n", sum);

    x = 1;

    while (x <= 100) tlssum += x ++;
    printf ("with TLS:%d\n", tlssum);

    return 0;
}

int main (void)
{
    thrd_t t1, t2;

    thrd_create (& t1, thrd_func, 0);
    thrd_create (& t2, thrd_func, 0);

    thrd_join (t1, 0);
    thrd_join (t2, 0);

    return 0;
}
```

其中，头文件<threads.h>定义了一些用于支持多线程执行的宏、类型、枚举常量和函数。函数 thrd_create 用于创建一个线程；函数 thrd_join 用于加入一个线程。如果依从 C11 的实现不支持多线程，则它必须定义一个宏__STDC_NO_THREADS__，以表明它不提供该头文件及相关的工具。

一个用存储类指定符_Thread_local 声明的标识符所指示的对象具有线程存储期，因此，

每个使用该标识符的线程都会在启动时创建和初始化一个独立的对象，且它的生存期贯穿创建它的那个线程的执行过程。

尽管每个线程内的对象都创建于同一个声明，但它们是各自独立的；使用同一个标识符在每个线程内访问的也是各自的对象。有时候，你可能希望在一个线程内访问位于另一个线程内的、具有线程存储期的对象，但这取决于C实现是否支持。

但是，一个很容易想到的问题是，既然具有线程存储期的对象在每个线程内部都有一个，而且通常为线程私有，为什么还要将它定义为公共的？这似乎很荒谬。

答案是这样可以解决一些历史遗留问题。在多线程流行之前，C运行库就存在了。C运行库包含大量的公共对象（比较著名的是errno），所以不是线程安全的。重新设计C的运行库代价较高，通过引入线程本地存储，可以将C运行库的公共对象创建为每个线程独立的副本，以免它们互相干扰。

此外，在每个线程的各个部分（函数）之间通过传递参数来共享对象效率不是太高，使用线程本地存储机制也是一个不错的办法。

2.7.3.3 自动存储期

一个标识符，若它同时满足以下三个条件，则它所指示的对象具有自动存储期：

（1）该标识符指示一个对象；

（2）该标识符是无链接的；

（3）该标识符在声明的时候没有使用存储类指定符static。

例如，以下作为函数参数的指针对象p及结构类型的对象t，都具有自动存储期。

```
void f (const char * p)
{
    struct t {/* …… */} t;
    /* …… */
}
```

具有自动存储期的对象只存在于程序执行期间的特定阶段。具体来说，因为这样的对象总是在块内声明的，所以，只在每次进入那个块时才会创建和初始化（如果在它的声明中包含了初始化器），每当离开它们所在的块后，这些对象就不存在了。例如：

```
void blkdemo (int x)
{
    int y = x;
    for (int i = 0; i < x; i ++) y ++;
     /* …… */
}
```

上例中，每当程序的执行进入函数blkdemo时，创建对象x，离开函数时，它就不复存在；同理，每当进入函数体时，创建对象y，程序执行到其声明点时，执行初始化（如果它有初始化器），当程序的执行离开函数体时，对象y不复存在。

块可以是复合语句，也可以是选择语句和迭代语句（参见“块”），所以，上面的for语句本身就是一个块。当for语句被执行时创建对象i，离开for语句后，对象即会消失。

特别地，用存储类指定符 register 和 auto 声明的标识符，它们所指示的对象毫无疑问地属于自动存储期。

有时候，你可能希望在一个线程内访问位于另一个线程内的、具有自动存储期的对象，但这取决于 C 实现是否支持。

2.7.3.4 指派存储期

用 aligned_alloc、calloc、malloc 和 realloc 等内存管理函数分配和得到的存储空间（或者说对象）具有指派存储期。如果分配成功，则返回一个适当对齐的指针（值）。这个指针（值）可以赋给任何类型的指针对象，并使用这个指针完成后续的访问。

具有指派存储期的对象，其生存期延续至为它分配的存储空间被销毁（解除分配）。典型地，可以使用 free 和 realloc 等内存管理函数来销毁已分配的存储空间。下面是一个用于解释指派存储期的例子。

```
# include <stdlib.h>

struct t {char c; int i; float f;};
# define N 20

int main (void)
{
   struct t * pt = malloc (N * sizeof (struct t));
   pt -> f = 0.001f;
   /* ..... */
   free (pt);

   return 0;
}
```

2.7.4 生存期

任何一个对象都会在程序的执行过程中存在一段时间，或短或长，最长可以和程序的执行时间一致。所谓对象的生存期（lifetime），就是指程序执行过程中的一个时间段，对象在这段时间里一定会存在（存储空间里保证有这个对象），而且是可以访问的。

具有静态存储期的对象，不管它是在哪里声明的，它的生存期贯穿整个程序的执行过程，和整个程序的执行时间一致。

下例中的 x 和 z 具有静态存储期，但 y 则不然。所以，对象 x 和对象 z 的生存期和整个程序的执行时间一致，对象 y 的生存期仅局限于函数 f 的执行期间：

```
int x = 0;

void f (int y)
{
```

```
    static int z;
    /* …… */
}
```

如果一个对象具有线程存储期，则其生存期从线程创建的时候开始（此时也创建了该对象），一直贯穿此线程的执行过程，和整个线程的执行时间一致。

如果一个对象具有自动存储期，那么我们就知道它必然是在块内声明的。在这种情况下，如果这个对象不是变长数组，则它的生存期可以描述如下：

（1）该对象的生存期仅限于程序在那个块内的执行期间，在程序的执行进入块时创建，这是其生存期开始的时候；当程序的执行离开它所在的块时，对象被销毁，这是其生存期终止的时候；

（2）当程序在块内执行时，可以调用任何函数，也可以进入内部块执行，所有这些过程都算做该对象的生存期。

创建这个对象时，它的初始值是不确定的。如果在它的声明里带有初始化器，或者它是用复合字面值创建的，那么，也只有在程序的执行到达声明或者复合字面值的位置时，才会执行初始化操作，对它初始化；否则，该对象的存储值依然是不确定的。

块可以多次重复进入。但是，根据上述第（1）点，上次进入这个块时所创建的对象已经过了生存期，但再次进入这个块并不意味着会“激活”那个过了生存期的对象。事实上，每次进入该块时都将创建一个新的对象，或者说创建该对象的一个新实例。

先来看下面的第一个示例：

```
# include <stdio.h>

void f (void)
{                                          //A
   printf ("House of cards.");             //B
   if (/* …… */) {/* …… */}                //C
   int x, y = 0;                           //D
   if (/* …… */) {/* …… */}                //E
}
```

上例中，函数体是一个块。尽管对象 x 和 y 是在 D 处声明的，但是它们在块的起始处，（即 A 处）就已经被创建了，只不过此时它们的初始值是不确定的。尽管如此，因为标识符作用域的关系，它们在 D 之前是不可能访问的。

当程序的执行到达 D 处时，因为 x 没有初始化器，所以它的值依然是不确定的；y 有初始化器，于是将它的值初始化为 0。

在块内执行的时候，调用了 B 处的 printf 函数，也可能进入 C、E 处的内部块，但在这上面耗费的时间依然算做 x 和 y 的生存期。最后，每次进入 x 和 y 直接隶属的这个块时，情况都和上面所说的完全一样。

在下面的例子中，每次进入 while 语句的循环体（块）时，将创建 3 个对象：标识符 y 指示的对象、pi 指示的对象，以及 pi 所指向的复合字面值对象，但此时它们的值是不确定的。当程序的执行来到 D 处时，y 依然具有不确定的值，而复合字面值对象被初始化为 0，pi 的值

被初始化为复合字面值对象的地址。

```
int x = 3;
while (x --)
{
    int y, * pi = & (int) {0};        //D
    /* ..... */
}
```

但是，参看下例，因为第一次进入块时跳过了 D 处的初始化部分，所以尽管在进入块时创建了 3 个对象即标识符 y 指示的对象、pi 指示的对象，以及复合字面值对象，但是至少在 S 处之前，它们的值是不确定的。执行 continue 语句后，如果 x 的值不为 0，则进入 while 语句的循环体（这也是一个块），每次进入这个块时，对象 y 具有不确定的值；复合字面值对象初始化为 0；对象 pi 被初始化为复合字面值对象的地址：

```
int x = 3;
goto skip;
while (x --)
{
    int y, * pi = & (int) {0};        //D
  skip:
    ;                                 //S
    /* ..... */
    continue;
}
```

在块内跳转时，尽管不会重新创建对象，但可能造成两种后果：对象未被初始化，或者重复（多次）被初始化。

请看下面的例子，进入该块时，创建 x、y 和 z 所指示的对象，初始的值不确定。当程序在块内的执行到达 D1 时，x 所指示的对象被初始化为 1。块执行到达 S1 时，y 指示的对象被初始化为 1。块内的第一次执行不会到达 D2，所以 z 指示的对象不会被初始化，值不确定。第一次执行到 S2 时，表达式 x+y 的值是 1，跳转到标号 loop 处，但这并不会重新创建那 3 个对象，因为只是块内跳转，并未离开当前块。

程序的执行第二次到达 S1 时，x 所指示的对象保持原值（0），y 所指示的对象被重新初始化为 1。此次程序的执行会到达 D2，故 z 被初始化为 5。当执行到达 S2 时，表达式 x+y 的值是 0，程序的执行离开当前块。

```
{
    int x = 1, y;                     //D1
  loop:
    y = 1;                            //S1
    if (x --) goto skip;
    int z = 5;                        //D2
  skip:
```

```
    if (x + y) goto loop;          //S2
}
```

变长数组类型会特殊一些，毕竟，在程序的执行到达它的声明点之前，C 实现并不知道应该为它分配多少存储空间。所以，如果 D 是一个声明，且声明了一个具有自动存储期的变长数组对象，那么该对象的生存期可以描述如下：

（1）在进入 D 所在的块时，并不创建该对象。相反，只是在程序的执行到达 D 处时才会创建该对象，这也是其生存期开始的时间；

（2）该对象的生存期结束于程序的执行离开 D 的作用域。典型地，包括跳出 D 所直接隶属的块（完成这个块的执行、跳出这个块），或者跳到 D 之前的位置。

因为变长数组的声明不能有初始化器，所以它在创建的时候具有不确定的值。而且，每次从外部进入 D 的作用域时，都将用不确定的值重新创建该变长数组对象的实例。

以下面的代码为例，当程序的执行进入函数体（块）时，在 D 处之前，不会创建对象 vla，它是不存在的。程序的执行到达 D 处时，将创建 vla 所指示的变长数组对象；离开函数体（块）后，也就离开了 vla 的作用域，变长数组对象的生存期结束。如果 S 处的跳转语句会执行的话，则刚才那个变长数组对象的生存期结束，一个新的变长数组对象会被创建出来，并开始它自己的生存期。

```
void f (int m)
{
  /* …… */
 lb_next:
  char vla [m];                    //D
  while (m --) vla [m] = 0;
  /* …… */
  if (/* …… */) goto lb_next;      //S
}
```

禁止越过变长数组的声明而直接进入其作用域。下例中，语句 S 是不合法的，它企图跳过变长数组 vla 的声明 D 而直接进入其作用域。

```
void f (int m)
{
  if (! m) goto next;              //S: 非法
  char vla [m];                    //D
  while (m --) vla [m] = 0;
 next:
  ;
  /* …… */
}
```

如果 D 是变长数组的声明，那么，可以在 D 的作用域内进行任何跳转动作，但跳转行为对变长数组对象没有什么影响，既不会重新创建变长数组对象，也不会改变其存储值。

因此，在下例中，可能会用 goto 语句多次跳转到标号 next 处执行，但每一次跳转都保证

不会重新创建 vla 的实例（新的对象），而且 vla 原来的值也不会受到影响。

```
void f (int m)
{
    char vla [m];
next:
    ;
    /* ..... */
    if (m --) goto next;
}
```

如果一个对象具有指派的存储期，则其生存期在对象被创建后开始，延续并终止于为它分配的存储空间被销毁（解除分配）。总之，可以根据需要灵活控制。

最后一种生存期是在 C11 中提到的“临时生存期”。C 语言里存在函数调用表达式，尽管函数调用看起来特殊，但函数调用表达式和别的表达式在应用上没什么区别。每个表达式都有自己的类型，每个表达式都会计算一个特定的值（除非是 void 类型的表达式），所以，如下例所示，不单单是表达式 x 会计算一个值，表达式 f()也会计算出一个值。

```
int f (void), y, x = 3, z;
y = x;
z = f ();
```

进一步地，如果函数的返回值是结构或者联合类型，也可以引用其成员的值，这和从一个左值中访问成员的值也没多出什么额外的语义。因此，如下例所示，和表达式 t.c 相比，尽管表达式 f().c 是从函数调用的返回值中取出一个成员（的值），但在求值上没有什么额外之外，并不需要考虑决定函数返回值的过程。

```
struct t {int i; char c;} t = {1, 'm'}, f (void);
int x = t.i;
char c = f ().c;
```

以上两个示例都可以用依从 C89（ISO/IEC 9899:1990）的实现来编译而不会有什么问题。如果运算符.的左操作数是一个左值，则该运算符的结果也是一个左值；如果.运算符的左操作数不是左值，则该运算符的结果也不是左值。

同时在 C89 中，只有一个数组类型的左值才能转换为指针。这就是说，依从于 C89 的实现不会接受以下程序中的 S1 和 S3，因为函数调用表达式的结果不是左值，所以这两个成员选择表达式的结果自然也不是左值：

```
# include <stdio.h>

struct stg {char a [64];};

struct stg f (void)
{
    struct stg m = {"Jack et la mécanique du coeur.\n"};
    return m;
```

```
}

int main (void)
{
    const char * p = f ().a;    //S1
    printf (p);                 //S2
    return printf (f ().a);     //S3
}
```

换句话说，表达式 f().a 将得到一个数组类型的值，但不是左值。C99 标准去掉了这一限制，允许从数组类型的值转换为指针。放开这个限制自然有很多好处，但它也带来了一些问题：既然可以从一个数组类型的值转换为指针，那么，自然也可以在程序中使用这个指针，而且从语法上挑不出任何毛病，就像上例中的 S2 和 S3 那样。但是，在使用这个指针的时候，访问它所指向的对象是安全的吗？

这是个尖锐的问题，几乎就要捅破 C 语言那本来就很脆弱的逻辑保护层，因为这势必要涉及到函数返回值的保存和访问机制，而这正是 C 语言的维护者保证不需要我们关心的部分。考虑到指针的使用通常只涉及下一个序列点，所以 C99 规定，如果企图在下一个序列点访问（当然也包括修改）函数调用的结果，行为是未定义的。类似的规定还有 4 个，都与“下一个序列点”和“未定义的行为”有关，分别涉及条件运算符、赋值运算符、逗号运算符和附录 J。

就以上程序而言，因为在两个全表达式之间有一个序列点，而组成表达式语句的表达式就是全表达式，所以在 S1 和 S2 之间有一个序列点；又因为在实际的函数调用和函数参数的求值之间存在一个序列点，所以在 S3 中，参数 f().a 和实际进入 printf 函数内执行之间也有一个序列点。

这就是说，在 C99 中，S2 和 S3 的行为都是未定义的。如果一个结构或者联合类型的表达式不是左值，且它没有数组类型的成员，这都没问题；当它含有数组类型的成员时，势必涉及到地址和指针，这才是麻烦的根源。

C99 的做法有些迂回，并未明确地从生存期方面加以约束。而且考虑到 S3 在 C++里是有效的，C11 标准文档删除了那些与“下一个序列点”和“未定义的行为”有关的 5 处表述，同时增加了一个段落。它规定，如果一个结构或者联合类型的表达式 E 不是左值（典型地，函数返回的是值而不是左值，函数调用表达式不是左值），且它含有一个数组成员（不一定非得是直接包含，可以通过嵌套的结构或者联合成员间接包含），那么，这个数组成员的值将依托（关联）于一个具有自动存储期和临时生存期的对象 X。

进一步地，对象 X 的生存期起始于 E 求值完毕，终止于 E 所在的那个全声明符或全表达式求值完毕。对象 X 的初始存储值是表达式 E 的值，在 X 的生存期内，它的值是不允许改变的，违反这一规定将导致未定义的行为。

对于上例中的 S3，这一规定是适用的。表达式 printf(f().a)是一个全表达式，表达式 f().a 的结果依托于一个具有自动存储期和临时生存期的对象，该对象的生存期从函数返回后开始（即表达式 f().a 完成求值），终止于表达式 printf(g().a)的求值全部完成。这就是说，S3 在 C11 中不会有未定义的行为，但 S2 依然是未定义的行为。

如果不能确定自己的编译器是否支持 C11 的这一特性，则最安全的做法是先固定函数 f

的返回值，将上例中的 S3 分解为以下两行：

```
struct stg m = f ();
return printf (m.a);
```

下述代码的行为是未定义的，因为它企图修改一个具有自动存储期和临时生存期的对象。

```
printf ("%x", ++ (f ().a [0]));
```

尽管复合字面值创建的对象没有名称，但它毕竟也是对象，有自己的生存期，它的生存期可以根据它出现的位置来决定。

如果复合字面值出现在任何函数体的外部，则它创建的对象具有静态存储期。否则，它是一个具有自动存储期的对象。根据前面的描述，如果复合字面值具有静态存储期，它的生存期从程序执行时开始，到程序终止时结束；如果复合字面值具有自动存储期，它的生存期仅局限于程序的执行从进入一直到离开其直接隶属的那个块。

考虑以下程序片断，n、p0、p1、(int){n}和(int){0}都位于所有函数体的外部，所以，n、p0、p1 和(int){0}都具有静态存储期。(int){n}是非法的，因为 n 不是常量。毕竟，具有静态存储期的对象，其初始化器只能包含常量表达式。相反地，复合字面值(int){0}是合法的，它所创建的无名对象具有静态存储期。

复合字面值(char[]){*s, 0}同样是合法的，因为它具有块作用域和自动存储期。用复合字面值(char[]){*s, 0}所创建的无名数组，被转换成指向其第一个元素的指针，并赋给 p。同样具有块作用域和自动存储期的复合字面值是(char *) {"I beg you parden?\n"}，尽管它出现在函数调用表达式中，是以函数调用的实际参数出现，但它在调用函数 f 之前就已经创建了。

```
int n = 5;
int * p0 = & (int) {n};     //非法
int * p1 = & (int) {0};

void f (char * s)
{
    char * p = (char []) {* s, 0};
    /* ..... */
}

int main (void)
{
    f ((char *) {"I beg you parden?\n"});
    return 0;
}
```

在下例中，while 语句也是一个块，它的每一次循环都将创建 pi 所指示的对象，以及一个新的复合字面值对象。每一次循环结束时，它们的生存期结束。

```
int x = 3;
while (x --)
{
```

```
    int pi = & (int){0};
    /* ..... */
}
```

基于以上原因，下面的代码是危险的，因为在离开 while 语句后，数组的每个元素都是指向无效对象的指针。

```
int * pa [3], x = 3;
while (x --)
{
    pa[x] = & (int) {x};
    /* ..... */
}
```

如果一个复合字面值对象具有自动存储期，则它是在一个块内，而且是在程序的执行从外部进入这个块时创建。如果在进入时跳过了它的初始化部分，则其值是不确定的。

```
goto skip;
{
    int * pi = & (int){0};
    skip:
        ;
        /* ..... */
}
```

和变长数组不同，不离开复合字面值所在的块，仅仅在块内重复执行，不影响它的生存期，也不会重新创建复合字面值对象。但是，如果程序在块内的重复执行会经过它所在的位置，则每次经过它时都要再次进行初始化操作。

以下面的代码片断为例，如果调用函数 f 时传递的实际参数是 3，则将输出 4 行完全一样的内容。尽管在 S2 处会改变复合字面值对象的内容，但每当程序的执行到达 S1 处时，都会重新执行初始化操作，将复合字面值对象的内容初始化为 0。

```
# include <stdio.h>

void f (int x)
{
  next:
    ;
    int * pi = & (int){0};          //S1
    printf ("%p, %d\n", pi, * pi);
    ++ * pi;                        //S2
    if (x --) goto next;
}
```

只能在一个对象的生存期内引用它，如果违反这一准则，程序的行为是未定义的，后果不可预料。同时，如果一个指针指向某个对象，那么，当那个对象的生存期结束之后，指向

它的指针也成为无效指针。

在下例中，S4 和 S5 的行为是未定义的。首先，在 S1 处定义了一个数组 a，但这个数组仅具有自动存储期，它的生存期仅限于当前函数。S2 处的 return 语句带有一个表达式 a，这个数组类型的表达式自动转换为指针，当这个指针在 S5 处使用时，它所指向的对象已经过了生存期，不存在了。

在 S3 处的语句里，return 语句同样带有一个表达式。与 S1 不同，在这里，字面串并不是做为数组的初始化器，而是独立存在的表达式，所以它用于初始化一个不可见的静态数组，然后从这个数组转换为指向其首元素的指针。由于这个数组具有静态存储期，在程序运行期间始终存在，所以能在 S6 中访问它的每一个元素。

最后来看函数 h 里的指针对象 pi，在它的声明中没有初始化器，因此，创建的时候具有不确定的值。在 if 语句形成的块中，它指向对象 x，此时它是有效指针；出了 if 语句所形成的块后，它再次变成无效指针。

```
# include <stdio.h>

char * f (void)
{
    char a [] = "A Bite of China."; //S1
    return a;                       //S2
}

char * g (void)
{
    return "Aye aye sir.";          //S3
}

int h (int flg)
{
    int * pi;
    if (flg)
    {
        int x = 0;
        pi = & x;
        /* ..... */
    }
    return * pi;                    //S4: 未定义的行为
}

int main (void)
{
    printf ("%s%d\n", f (), h (1)); //S5: 未定义的行为
```

```
    return printf (g ());          //S6
}
```

如果一个对象在声明的时候使用了存储类指定符 register，则不能获取它的地址。相反，如果对象在声明的时候未使用存储类指定符 register，在它的生存期内，可随时获取它的地址，而且每次得到的地址都相同。

这就是说，对象一旦创建，它在存储器中的位置是不变的，因此被认为拥有常量地址（constant address）。而且，只要还在对象的生存期内，我们随时可以使用这个地址得到最后一次保存的值。

“换句话说，在该对象的生存期内，如果有多个指向该对象的指针，则即使这些指针创建的时间不同，它们的值也是相等的（拥有相同的指针值）。当然，该对象的地址可能会随着程序的每一次执行而不同，但在程序的每一次执行期间，它的地址是不变的。

上面的这两段叙述不需要示例也能很容易理解，但我还是给出了下面的例子。通过程序的输出可以知道，在对象 global 的生存期内，每次获取的地址都一样。

```
# include <stdio.h>

int global;

void * fetcha (void)
{
    printf ("%p\n", (void *) & global);
    /* ..... */
    return & global;
}

int main (void)
{
    printf ("%p\n", (void *) & global);
    return printf ("%p\n", fetcha ());
}
```

2.7.5 对齐

受硬件布线的限制，或者为了提高存储器访问效率，要求特定类型的对象在存储器里的位置只能开始于某些特定的字节地址（有关字节地址，参见“字节”），而这些字节地址都是某个数值 N 的特定倍数（以不超过实际的存储空间为限），这称为对齐（alignment）。进一步地，我们称那个对象是对齐于 N 的。

举一个实际的例子，在编者的计算机上，int 类型的对象，可以位于 0x00000004、0x00000008、0x0000000C 等字节地址上，都是 4 的倍数（但不能超过物理内存芯片可以提供的实际地址范围）；long long int 类型的对象，只能位于 0x00000008、0x00000010、0x00000018、0x00000020 等任何一个字节地址上，都是 8 的倍数（但不能超过物理内存芯片

可以提供的实际地址范围）。

再比如，char 类型的对象可以位于任何字节地址上，如 0x00000001、0x00000002、0x00000003 等，都是 1 的倍数（但不能超过内存的实际地址范围）。

细心的读者可能会发现这里没有提到字节 0x00000000。在 C 中，这是一个特殊的字节地址，任何对象都不能起始于这个位置。

对于完整的对象类型来说，“对齐”限制了它在存储器中可以被分配到的地址。实际上，这是一个由 C 实现定义的整数值。

未对齐的存储器访问对不同的计算机来说会有不同的效果。在有些硬件架构上（比如 Intel x86 系列），未对齐的访问不会引发实质性的问题，也不会影响结果的正确性，但会使处理器对存储器的访问变得笨拙；在另一些硬件架构上，未对齐的访问将导致总线错误。

C11 引入了两个关键字“_Alignas”和“_Alignof”，_Alignas 作用于指定的类型，或者某个类型的对象，只在声明中使用，强制它们按要求对齐。

```
int _Alignas (8) foo;
struct s {int a; int _Alignas (8) bar;};
```

以上将使 int 类型的对象 foo 和结构类型的成员 bar 按 8 字节对齐。

对齐的数值可以通过_Alignof 运算符得到。_Alignof 运算符的操作数要求是用括号括起来的类型名，例如：

```
# include <stdio.h>

void f (void)
{
   printf ("%zu, %zu, %zu, %zu\n",
                _Alignof (char),
                _Alignof (int),
                _Alignof (int [33]),
                _Alignof (struct {char c; float f;})
           );
}
```

运算符_Alignof 的结果类型是 size_t，是在头文件<stddef.h>中定义的一个无符号整数类型。_Alignof 返回的是一个有效的基础对齐（参见“基础对齐”）。但有效的对齐也包括可能存在的扩展对齐（参见“扩展对齐”）。

值得注意的是，对齐并不是两个连续分配的同类型对象的地址所间隔的字节数，尽管有时候它们的确一样。类型的大小和对齐没有必然联系。

比如，在编者的机器上，int 类型的对齐是 4，这种类型的对象只能位于字节地址 0x00000004、0x00000008、0x0000000C，……，两个连续的字节地址间隔为 4，和它们的对齐相等。但是，再看下面的例子：

```
# include <stdio.h>

void f (void)
```

```
{
    typedef struct {char a [13]; int i;} A;
    printf ("%zu, %zu\n", sizeof (A), _Alignof (A));
}
```

在编者的机器上，类型 A 的大小是 20，对齐是 4，等于 int 类型的对齐。类型 A 的对象同样可以位于字节地址 0x00000004、0x00000008、0x0000000C，……，因为 int 类型的对齐更为严格，所以，只要保证结构成员 i 的字节地址能够被 4 整除即可。如果类型 A 的某个对象位于字节地址 0x00000004，则该对象的成员 i 则位于 0x00000014；如果类型 A 的某个对象位于字节地址 0x00000008，则该对象的成员 i 则位于 0x00000018。但这种类型的对象较大，如果要在存储器中连续分配多个这样的对象，则它们的地址只能分别起始于 0x00000004、0x00000018、0x0000002C，……，相互间隔的字节数并不等于类型 A 的大小。

无论如何，所有对象的对齐都要求必须是 2^N，N 为非负整数。

要判断两个对象是否具有相同的对齐，只需要看它们的数值。数值上相等的，对齐也相同；数值上不相等的，对齐也不同；如果数值较大，则表示它的对齐更严格；如果数值较小，则表示它的对齐较弱。

不同的对象，按其类型不同有不同的对齐要求。如果对它们进行比较，则可以分为 3 种：弱对齐、强对齐和严格对齐。它们在数值上由小到大排列，弱对齐的适应性较好，具有弱对齐的对象，也可以位于（对齐于）强对齐和严格对齐的地址上。

进一步地，char、signed char 和 unsigned char 类型的对象具有最弱的对齐要求。

2.7.5.1 基础对齐

所谓基础对齐，是指位于合理区间上的那些对齐要求。那么，怎么才算是位于合理区间呢？

一个 C 实现可以提供一个适用于所有上下文环境的最大对齐，在数值上等于以下表达式求值的结果。

```
_Alignof (max_align_t)
```

任何小于或者等于这个数值的对齐要求都被视为基础对齐。注意：max_align_t 是在头文件<stddef.h>中定义的对象类型。

举例来说，如果表达式_Alignof (max_align_t)的值为 8，则 1、2、4、8 都是基础对齐。

2.7.5.2 扩展对齐

所谓扩展对齐，是指超出合理区间的那些对齐要求。那么，怎么才算是位于合理区间呢？

一个 C 实现可以提供一个适用于所有上下文环境的最大对齐，在数值上等于以下表达式求值的结果。

```
_Alignof (max_align_t)
```

任何大于这个数值的对齐要求都被视为扩展对齐。注意：max_align_t 是在头文件<stddef.h>中定义的对象类型。

举例来说，如果表达式_Alignof (max_align_t)的值为 8，则 16、32、64、128 都是扩展对齐。

2.8　字节

字节（byte）是存储器里的可寻址单元，而且每个这样的单元都是由一系列连续的比特组成。对于任何硬件架构来说，要想让 C 语言能够在它上面实现，条件之一是字节的长度必须足以容纳任何基本执行字符集的成员（参见“执行字符集”）。

字节的具体长度没有标准定义。在有些计算机上，一个字节的长度是 9 个比特。有些机器的字长是 36 位，那么，因为字总是等于若干个连续的字节（的合并），因此，在这些机器上，一个字节可能包含 9 比特，也可能包含 12、18 或者 36 个比特。

在 C 中，一个字节所包含的比特数是实现定义的，但要求不能少于 8 个比特。C 实现会提供一个头文件<limits.h>，其中定义了宏 CHAR_BIT，它的数值可能会随不同的平台而异，但无论如何它都是一个字节所包含的比特数。

字节地址（byte address）在本书中用于描述对象在存储器中的起始位置。字节是存储器中的最小可寻址单元，每个字节都有一个对应的存储器地址，称为字节地址。

例如，如果一个内存芯片的容量是 256 个字节，则有效的字节地址分别是 00、01、02、…、FE，FF。

2.9　行为

我们通过书写 C 程序来指定计算机所要执行的动作，这些动作所产生的结果、效果或者说后果，就是行为（behavior）。所以，行为可能是指结果、外观、现象和采取的动作，等等。从大的方面说，因执行错误的程序而导致火花或者设备损坏，这是行为；从小的方面说，得到一个正确或者不正确的数值，这也是行为。

标准文档涉及的行为包括：未定义行为、实现定义行为、未指定行为和本地指定行为，这几个术语用来给程序的行为分类，可分别参见其各自的词条。

存在这些未指定、未定义或者由实现和本地指定行为的原因有两方面：一方面，人们希望 C 能够在大多数硬件架构上实现；但是另一方面，硬件架构本身的差异性较大。因此，对 C 中的某些方面不进行严格规定是有意的，也很灵活。

未定义行为和未指定行为表面上容易区分，但在面对一个具体的样例时却可能难以辨别。在“初始化器”部分有一个示例，揭示了这两者之间的微妙之处。未定义行为通常意味着错误和风险；实现定义行为影响程序的可移植性；未指定行为在某些条件下是可控的，它可能变成实现定义行为，也可能变成未定义行为。区域指定行为则与选定的国家和地区有关。

依从标准的 C 程序在任何依从标准的 C 实现上都能正常转换，但是严格依从标准的 C 程序只能使用标准中指定的语言特性、最小的实现限制（比如 int 类型的取值范围等）和库文件，而且它的输出不会受未指定行为、未定义行为和实现定义行为的影响。这实际上将 C 程序分为三种：错误的程序、不可移植的程序和可移植的程序。

错误的程序是始终需要警惕和避免的，但程序的可移植并不是什么大问题，你所写的 C 代码

可以是能够移植的，也允许是不可移植的，只要它能很好地完成你的工作。如果需要，未指定行为、实现定义行为和本地指定行为也并不可怕，尽管它们影响到程序的可移植性，但如果可移植性并不是你所需要的，也没什么关系。

2.9.1 未定义行为

生活中充满了不确定性，良好的意愿和行动未必能产生预期的结果，这样的例子在生活中比比皆是。同样地，仅仅是语法上的正确性，并不能保证正确的行为。比如下面的例子，这些代码在语法上没有任何问题，但并不能得到正确的结果，因为指针 p 没有初始化。

```
int * p, x = * p;
```

类似于这样的问题可能会在程序中的任何地方出现，也可能以任何形式出现，比如所写的程序本身含有错误（错误的数据、逻辑错误、错误的指令等），或者不可移植（例如，企图在 Unix 平台上转换和运行一个 Windows 程序），等等，要想一一列举，或者试图捕捉到所有可能的错误情形是非常困难的，也是不现实的，唯一可能的做法是从大的方面指出在哪些情况下程序和代码的行为是可以预期的，在哪些情况下从无从知道会有什么样的结果。因此，所有这些无法规定，从而也无法预测的行为，称为*未定义行为*（undefined behavior: UB）。未定义行为的后果是不可预测的，程序转换可能会失败、程序在执行时可能会崩溃或者结果不正确，或者即使偶尔得到了想要的结果也只是碰巧，等等。

有时候说“程序的行为是未定义的”或者“未定义的”，其实都是指未定义行为。未定义行为通常意味着那种做法是不安全的，后果是不可预料的。

很多人对下面的表达式比较熟悉，因为它们曾经在网络和书籍中被广泛讨论过。

```
i = ++ i + 1
i = ++ i + i ++
```

大家认为讨论这种未定义行为没有必要，因为不写这种代码就可以避免。事实真的是这样吗？考虑下面的代码片断：

```
i = (* p = 0) + * q
```

其看上去没什么问题，但是，如果 i、p 和 q 的声明如下：

```
int i = 10086, * p = & i, * q = & i;
```

那么，上面表达式的行为就是未定义的。因为不管怎样，i、*p 和*q 都指示同一个对象，所以表达式的副作用和值计算是无序的。

2.9.2 未指定行为

C 是讲求代码执行效率的语言，“即使牺牲程序的可移植性，也要让它跑得快”，是 C 的精神和信条。所谓*未指定行为*（unspecified behavior），是 C 实现得到的特权之一，标准指出可以使用哪些值或者采用哪些行为，但不强迫必须使用或者遵循其中的哪一个。根据当前的硬件架构或者其他现实条件，C 实现可以自由选择一种它认为是最有效率的做法。

在下面的例子中，函数 f 需要两个参数。但是在调用这个函数时，这两个参数的求值顺序是未指定的，可能先求值 x，也可能先求值 x+1，或者交叉求值，取决于编译器认为哪一种方法最利于优化代码。

```
int x = 0, f (int, int);
```

```
f (x, x + 1);
```

未指定行为是无害的，所谓“条条大路通罗马”，它只是提供了多种可能性，但任何一种可能性都是正确的，因此，除非违反了标准的约束条件，否则不会导致未定义行为。上例中，x和x+1按什么顺序求值，不影响函数调用f(x, x + 1)的结果；但是，函数调用f(x, ++ x)的行为却是未定义的，参见“求值”。

2.9.3 实现定义行为

实现定义行为（implementation-defined behavior）实质上是未指定行为（参见“未指定行为”），但是，如果 C 实现愿意，它可以做出明确的选择，但是必须在开发文档中加以说明，告知用户它采取的是哪种行为。

例如，如果位字段成员的类型是 int，则它可以被视为 signed int，也可以被视为 unsigned int，到底使用哪一个则由 C 实现来决定，且必须在开发文档中注明。

有时候人们说“取决于 C 实现”或者“由 C 实现决定”，其实都是指实现定义行为。如果对程序的可移植性没有要求，则可以编写依赖于具体 C 实现的代码。

2.9.4 区域指定行为

不同的国家和地区有不同的语言、习俗和惯例，这就要求 C 实现依据各个区域的特点而采取不同的行为（每个 C 实现都需要在它的开发文档中列出这些选项），这就是本地指定行为（locale-specific behavior）。

例如，很多库函数的结果依赖于特定的区域。文字的方向、时间、日期和货币的格式等，都依赖于不同的区域。和区域有关的头文件是<locale.h>，它声明了两个函数、一个类型，并包含了若干宏定义，可以进行与本地化有关的操作。其中，库函数 setlocale 用于设置和当前区域有关的内容，它将影响到程序中与区域有关的功能。

第3章 类 型

从大的方面来说，C 中的类型（types）分为对象类型和函数类型。前者用于描述对象，后者用于描述函数，可分别参见各自的词条。

对象的类型决定了它的存储内容应当怎么解释（参见词条“值”），以及它所占用的存储空间的大小（参见词条“类型表示”）。

可以将一种对象类型的值转换为另一种对象类型，但并不是所有对象类型之间都可以进行相互转换（参见词条“类型转换”）。

3.1 类型图

C 语言的类型划分，以及它们之间的关系如图 3-1 所示。

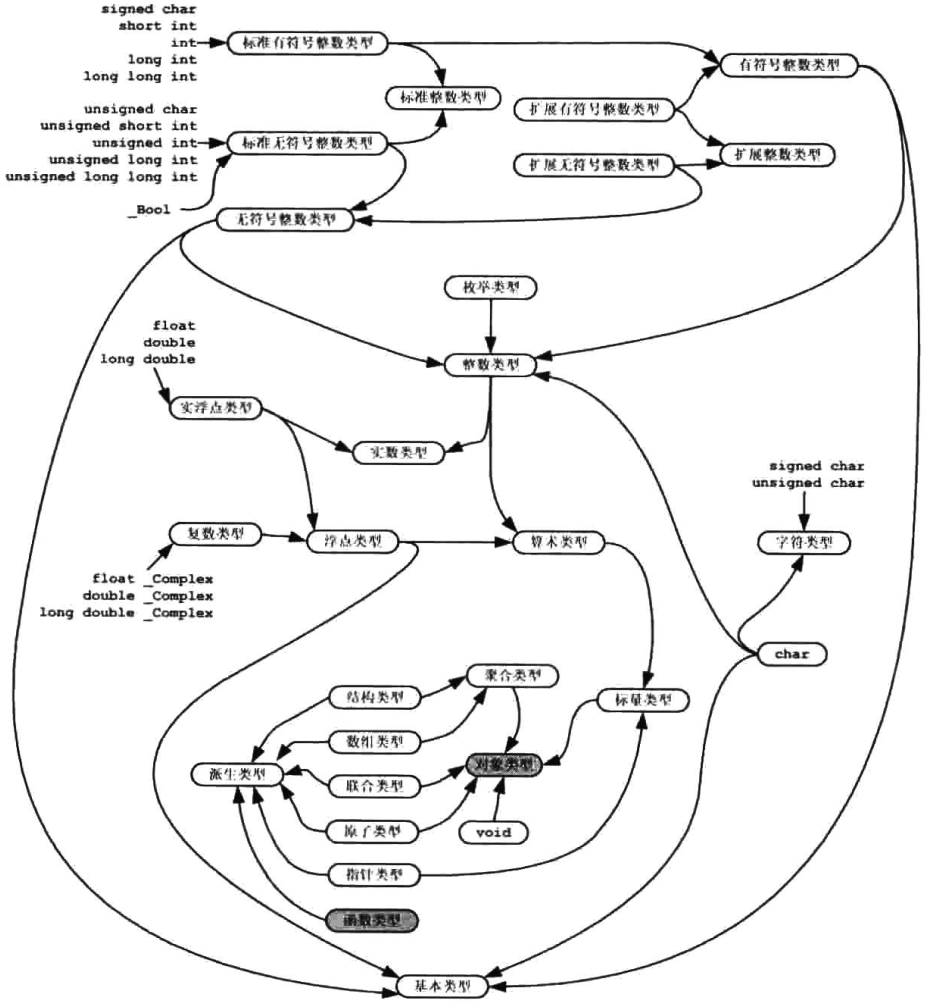


图 3-1 C 语言类型图

3.2 基本类型

基本类型（basic types）包括无符号整数类型、有符号整数类型、浮点类型和 char（类型），具体可参见各自的词条。

基本类型都是完整的对象类型，它们都具有已知的大小（参见“完整类型”）。

3.2.1 无符号整数类型

对于每一个有符号整数类型而言，它们都对应着一个无符号整数类型（unsigned integer types）。例如，signed char 是有符号整数类型，它有一个对应的无符号整数类型 unsigned char。

无符号整数类型包括标准无符号整数类型和扩展无符号整数类型，它们分别与标准有符号整数类型和标准无符号整数类型对应。具体内容可参见其各自的词条。

对于每一种无符号整数类型来说，它所需要的存储空间大小及对齐要求，都和它对应的有符号整数类型相同。

无符号整数类型是用关键字 unsigned 来指示的。例如，对于有符号整数类型 signed int 而言，存在着一个对应的无符号整数类型 unsigned int。

无符号整数类型的最小值总是 0，其最大值是由 C 实现定义的，标准要求在头文件 <limits.h>中通过宏来定义它们各自的最大值。

下例中，函数 num_binary 用于按顺序提取组成对象 vari 的值的每个比特，将它们转换成字符串。对象 vari 属于无符号整数类型，且假定指针 str 指向的对象有足够的空间接收转换后的字符串。

```
# include <limits.h>
# include <stddef.h>

typedef unsigned long UTYPE;

void num_binary (UTYPE vari, char * str)
{
    for (size_t x = sizeof (UTYPE) * CHAR_BIT; x > 0; x --)
        * (str ++) = (vari >> x - 1 & 1) + '0';
    * str = '\0';
}
```

细心的读者可能会问，为什么函数内的循环代码不写成这样：

```
for (size_t x = sizeof (UTYPE) * CHAR_BIT - 1; x >= 0; x --)
    * (str ++) = (vari >> x & 1) + '0';
```

因为这样写是错误的，会形成一个无限循环（死循环），无符号整数类型的值（在这里是 x 的值）总是大于或者等于零。

在参加运算时，无符号整数类型不存在所谓的“溢出”。如果无符号整数值 X 用无符号整

数类型 U 表示不了，则被换算成 X%M（也就是 X 除以 M 所得的余数）。这里，M 是比类型 U 的最大值大 1 的数。

在下例中，假设 unsigned short 类型可以容许的最大值 USHRT_MAX 是 65535，将 int 类型的对象 u 初始化为 60000 是没有问题的，毕竟 unsigned short 类型的对象 u 可以表示这个值。

```
unsigned short u = 60000, us;
us = u + 5555;  //us 的值是 19
```

再来看表达式 us=u+5555。为了计算子表达式 u+5555 的值，需要先将 u 的值提升为 int 类型，再与 int 类型的值 5555 相加，结果是 int 类型的值 65555。但是，将这个结果赋给对象 us 时，遇到的问题是，us 的类型是 unsigned short，无法表示这个值。在这种情况下，赋给 us 的值其实是 65555 与比 USHRT_MAX 大 1 的数的模，即 65555%65536，结果是 19。

尽管这个过程是自动发生的，但难免会有人想亲自验证一下，于是写出以下代码：

```
# include <limits.h>

void f (void)
{
   unsigned short u = 60000, us;
   us = (u + 5555) % (USHRT_MAX + 1);
}
```

表面上看起来没有问题，但这段代码可能达不到预期的结果。如果 unsigned short 类型所能容许的最大值 USHRT_MAX 是 65535，则“比 unsigned short 类型的最大值大 1 的数”和 USHRT_MAX+1 是两回事。原因很简单，在计算表达式(USHRT_MAX + 1)的值时，USHRT_MAX 会先被提升为 int 类型，然后与 int 类型的值 1 相加，结果是 int 类型。如果这个结果不能被 int 类型容纳（在有些平台上，这是可能的），则表达式(USHRT_MAX + 1)的结果将不会是 65536！

理论上 long long int 的宽度比 unsigned short 要大得多，所以这样就没问题了：

```
us = (u + 5555) % (USHRT_MAX + (long long int) 1);
```

3.2.1.1 标准无符号整数类型

如图 3-1 所示，与标准有符号整数类型对应的无符号整数类型，再加上 C99 引入的_Bool 类型，统称为标准无符号整数类型（standard unsigned integer types）。具体来说，标准无符号整数类型包括：unsigned char、unsigned short int、unsigned int、unsigned long int、unsigned long long int 和_Bool。

以上类型的详细描述，可参见其各自的词条。

除_Bool 类型外，标准要求 C 实现在头文件<limits.h>中通过定义相应的宏来给出每一种类型的最大值。

就取值范围而论，unsigned char 是 unsigned short int 的子集；unsigned short int 是 int 的子集；int 是 long int 的子集；long int 是 long long int 的子集，但它们未必都要是真子集。

3.2.1.1.1 _Bool

在 C 语言引入_Bool 类型之前，尽管它的类型系统中没有布尔（或者称为逻辑类型），但也丝毫不影响它应付各种复杂的编程任务。这其中的诀窍就在于用表达式的值和 0 做比较，

值等于0时做一件事，否则做另外一件事。

_Bool类型是从C99开始引入的，一个声明为_Bool类型的对象，只需要能够表示0和1两个值即可。

_Bool属于标准无符号整数类型，其长度（占用的字节数）取决于C实现，至少是1个字节。所以那种认为_Bool类型只需要一个比特的想法是错误的。

如果源文件中包含了头文件<stdbool.h>，则可以使用bool来代替_Bool，并用true和false为布尔类型的对象赋值。这是因为在该头文件中定义了4个宏：

```
# define bool       _Bool
# define true       1
# define false      0
# define __bool_true_false_are_defined 1
```

因此，如果某源文件包含了头文件<stdbool.h>，则它可以使用这些宏，以下是一个源文件的例子。

```
# include <stdbool.h>

int g (bool);

bool f (void)
{
    bool b = true;
    /* ..... */
    return g (b) == 3;
}
```

_Bool对象所包含的比特数至少是CHAR_BIT（在头文件<limits.h>中被定义为字符类型的对象所包含的比特数，也是一个字节所包含的比特数）。尽管如此，_Bool类型的宽度（参见“宽度”）可以只有1个比特。也正是因为这个原因，在将一个非零值赋给一个_Bool类型的对象时，它将变为数值1：

```
_Bool b = 5;          //b所指示的对象，在赋值后为1
```

3.2.1.1.2 unsigned char

unsigned char属于标准无符号整数类型，也是与有符号整数类型signed char对应的无符号整数类型，这种类型的对象所能容纳的最大值UCHAR_MAX要求大于等于255，即不小于28-1。宏UCHAR_MAX是在头文件<limits.h>中定义的。

标准要求，C实现都应当使unsigned char类型的对象所能容纳的最大值为2CHAR_BIT-1。宏CHAR_BIT在头文件<limits.h>中被定义为字符类型的对象所包含的比特数，也是一个字节所包含的比特数。例如，在9比特一个字节的系统（实现）中，unsigned char类型的对象所能容纳的最大值是511。

在下例中，声明了一个unsigned char类型的标识符c，并向其所指示的对象中存储了一个unsigned char类型的最大值。

```
unsigned char c = 255;
```

但是，这不具备可移植性。正确的做法如下。

```
unsigned char c = UCHAR_MAX;
```

unsigned char 类型可能等价于 char 类型，但这取决于用户所使用的 C 实现。但无论如何，char 和 unsigned char 是两种截然不同的类型，且互不兼容。

3.2.1.1.3 unsigned short int

unsigned short int 属于标准无符号整数类型，是与有符号整数类型 short int 对应的无符号整数类型，标准要求这种类型的对象所能容纳的最大值 USHRT_MAX 要大于等于 65535，即不小于 2^{16}-1。宏 USHRT_MAX 是在头文件<limits.h>中定义的。

在下面的例子中声明了一个 unsigned short int 类型的对象。

```
unsigned short int usi = 23;
```

3.2.1.1.4 unsigned int

unsigned int 属于标准无符号整数类型，是与 signed int 对应的无符号整数类型，标准要求这种类型的对象所能容纳的最大值 UINT_MAX 要大于等于 65535，即不小于 2^{16}-1。宏 UINT_MAX 是在头文件<limits.h>中定义的。

在下面的例子中声明了一个 unsigned int 类型的对象。

```
unsigned int ui = 2300;
```

3.2.1.1.5 unsigned long int

unsigned long int 属于标准无符号整数类型，是与 signed long int 对应的无符号整数类型，标准要求该类型的对象所能容纳的最大值 ULONG_MAX 要大于等于 4294967295，即不小于 2^{32}-1。宏 ULONG_MAX 是在头文件<limits.h>中定义的。

在下面的例子中声明了一个 unsigned long int 类型的对象。

```
unsigned long int uli = 2300UL;
```

3.2.1.1.6 unsigned long long int

unsigned long long int 属于标准无符号整数类型，是与 signed long long int 对应的无符号整数类型。标准要求 unsigned long long int 类型的对象所能容纳的最大值 ULLONG_MAX 要大于等于 18446744073709551615，即不小于 2^{64}-1。宏 ULLONG_MAX 是在头文件<limits.h>中定义的。

unsigned long long int 类型是从 C99 开始引入的。

在下面的例子中声明了一个 unsigned long long int 类型的对象。

```
unsigned long long int ull = 2300ULL;
```

3.2.1.2 扩展无符号整数类型

在标准无符号整数类型的基础上，C 实现可以根据实际需求添加一些与扩展有符号整数类型对应的无符号整数类型，称为扩展无符号整数类型（extended unsigned integer types）。比如，C 实现可以定义 128 位的扩展无符号整数类型，以弥补现有标准无符号整数类型的不足。

3.2.2 有符号整数类型

这是一个统称，有符号整数类型包括标准有符号整数类型和扩展有符号整数类型，具体可参见其各自的词条。

3.2.2.1 标准有符号整数类型

标准有符号整数类型（standard signed integer types）是一个统称，由以下5种有符号整数类型组成：signed char、short int、int、long int、long long int，具体描述参见其各自的词条。

标准有符号整数类型的取值范围是由C实现定义的，用最小值和最大值来界定。标准要求C实现在头文件<limits.h>中通过定义相应的宏来给出。

在编者的机器上，int类型和long int类型的取值范围是一样的，都是-2147483647～+2147483647。换句话说，就取值范围而论，signed char是short int的子集；short int是int的子集；int是long int的子集；long int是long long int的子集，但它们未必都是真子集。

3.2.2.1.1 signed char

signed char属于标准有符号整数类型，标准要求这种类型的对象所能容纳的最小值SCHAR_MIN小于等于-127；所能容纳的最大值SCHAR_MAX要大于等于+127。宏SCHAR_MIN和SCHAR_MAX是在头文件<limits.h>中定义的。

signed char类型可能等价于char类型，但这取决于用户所使用的C实现。但无论如何，char和signed char都是截然不同的类型，而且互不兼容。

在下面的例子中声明了一个signed char类型的对象。

```
signed char = 23;
```

3.2.2.1.2 short int

short int属于标准有符号整数类型，标准要求这种类型的对象所能容纳的最小值SHRT_MIN要小于等于-32767，即-（2^{15}-1）；所能容纳的最大值SHRT_MAX要大于等于+32767，即2^{15}-1。宏SHRT_MIN和SHRT_MAX是在头文件<limits.h>中定义的。

在下面的例子中声明了一个short int类型的对象。

```
short int sint = 230;
```

3.2.2.1.3 int

int属于标准有符号整数类型，标准要求这种类型的对象所能容纳的最小值INT_MIN小于等于-32767，即-（2^{15}-1）；所能容纳的最大值INT_MAX大于等于+32767，2^{15}-1。宏INT_MIN和INT_MAX是在头文件<limits.h>中定义的。

由于C实现和执行环境的硬件架构紧密相关，因此，建议C实现将int类型的长度定义为最适合硬件架构的自然尺寸。注意，这不是保证，也不是强制性的要求。对于一个64位架构的处理器来说，64位是使它工作效率最高的自然尺寸，但运行在该处理器上的C编译器可能比较老旧，只能将int定义成具有16位或32位的存储空间。具体的情况要通过查看<limits.h>文件中的宏INT_MAX和INT_MIN而决定。

为了保证可移植性，不要使用in 为了保证可移植性，不要使用int类型的对象来保存-

32767～32767 以外的整数值，如果因特殊原因非这样做不可，应在程序中确保这样做是安全的，或者使用宽度更大类型。事实上，不仅仅是 int 类型，对它类型来说，要保证其可移植性，也必须参考和遵循头文件<limits.h>中的相应定义。

在下面的例子中声明了一个 int 类型的对象。

```
int i = 2300;
```

3.2.2.1.4 long int

long int 属于标准有符号整数类型，标准要求这种类型的对象所能容纳的最小值 LONG_MIN 要小于等于-2147483647，即-（2^{31}-1）；所能容纳的最大值 LONG_MAX 要大于等于+2147483647，即 2^{31}-1。宏 LONG_MIN 和 LONG_MAX 是在头文件<limits.h>中定义的。

在下面的例子中声明了一个 long int 类型的对象。

```
long int lint = 2300L;
```

3.2.2.1.5 long long int

long long int 属于标准有符号整数类型，标准要求这种类型的对象所能容纳的最小值 LLONG_MIN 小于等于 -9223372036854775807，即 -（2^{63}-1）；所能容纳的最大值 LLONG_MAX 大于等于 +9223372036854775807，即 2^{63}-1。宏 LLONG_MIN 和 LLONG_MAX 是在头文件<limits.h>中定义的。

long long int 类型是从 C99 开始引入的。在下面的例子中声明了一个 long long int 类型的对象。

```
long long ago = 2300LL;
```

3.2.2.2 扩展有符号整数类型

所有 C 实现都必须定义标准规定的有符号整数类型（标准有符号整数类型），但它也可以定义自己的有符号整数类型，称为扩展有符号整数类型（extended signed integer types）。

例如，Microsoft Visual C++中的__int64，这是一个 64 位的有符号整数类型。因为那时 C 中还没有 long long int。一般来说，这些由 C 实现自定义的类型，它们的名称通常都以两个下画线开头，或者以一个下画线外加一个大写字母开头。

3.2.3 浮点类型

这是一个统称，浮点类型包括复数类型和实浮点类型，具体可参见其各自的词条。其中，复数类型是从 C99 开始引入的。

每个浮点类型都有一个对应的实数类型，且总是一个实浮点类型。实浮点类型对应的实数类型就是它本身的类型；复数类型对应的实数类型可以通过去掉类型名中的_Complex 关键字得到。例如，double 类型对应的实数类型是 double；double _Complex 类型对应的实数类型实际上是 double。

每个复数类型实际上可以看做两个实数类型的组合，即一个实数的数组，且该数组只有两个元素。第一个元素对应着复数的实部，第二个元素对应着复数的虚部。这不是单纯的类

比，而是在复数和数组之间具有等同性，它们在对齐要求和类型表示上也是完全一样的。

这意味着，复数的实部和虚部具有相同的类型，且是连续存储的。下面用一个数组来说明这种等效性。

```
double _Complex z = 2.0 + 5.0i;
double a [2] = {2.0, 5.0}; //z 具有和 a 一样的表示和对齐要求
```

3.2.3.1 复数类型

复数类型（complex types）又称复数浮点数类型，它是从 C99 开始引入的，而且是以下 3 种类型的统称：float _Complex、double _Complex 和 long double _Complex。

复数 z 可以被转换为 z=x+y×i，然后在笛卡儿坐标系上表示。其中，x 和 y 是实数，i 是虚数单位，$i^2=-1$。这样，x 是 z 的实部，y 是 z 的虚部。复数在实部和虚部分别是两个实浮点数（可能是 float、double 或者 long double），而且类型相同。

C 的实现可以不支持复数类型。

3.2.3.2 实浮点类型

实浮点类型（real floating types）是以下 3 种类型的统称：float、double、long double，分别参见其各自的词条。

传统上，前两种类型分别称为“单精度浮点类型”和“双精度浮点类型”。“float 类型的值的集合”是“double 类型的值的集合”的子集，而后者又是“long double 类型的值的集合”的子集。

任意一个浮点数 x 都可以表示成如下形式。

$$x = sb^{e}\sum_{k=1}^{p} f_k b^{-k},\ e_{min} \ll e \ll e_{max}$$

这里，s 是符号，+1 或者-1；b 是指数部分的进制或者基数，必须大于 1；e 是指数，如上所示，必须大于等于 e_{min} 并且小于等于 e_{max}；p 是精度，也就是 b 进制的有效位数；f_k 是有效数字，是一个小于 b 的非负整数。

进制或者基数 b 的选择取决于 C 实现，是在头文件<float.h>中定义的，必须大于等于 2。

浮点数包括一些特殊的数值，如无穷大和非数字（Not a Number，NaN），使用 NaN 的表达式其结果也为 NaN。C99 之后的标准库允许使用这些特殊值。

3.2.3.2.1 float

float 类型通常称为单精度浮点类型，标准要求 float 类型的最小值 FLT_MIN 不小于 1E-37；其最大值 FLT_MAX 不小于 1E+37；小数数位 FLT_DIG 要求不少于 6。

以上所提到的宏是在头文件<float.h>中定义的。

3.2.3.2.2 double

double 类型通常称为双精度浮点类型，其最小值 DBL_MIN 要求不小于 1E-37；其最大值 DBL_MAX 要求不小于 1E+37；小数数位 DBL_DIG 要求不少于 10。

以上所提到的宏是在头文件<float.h>中定义的。

3.2.3.2.3 long double

long double 类型的最小值 LDBL_MIN 要求不小于 1E-37；其最大值 LDBL_MAX 要求不小于 1E+37；小数数位 LDBL_DIG 要求不少于 10。

以上所提到的宏是在头文件<float.h>中定义的。

3.2.4 char

在计算机发展的早期，仅有少数国家和少数行业才能使用它，而且主要是英语系的国家和地区。因此，在那个年代，字符集的规模很小，如果要求不高，7 位的编码就足够使用。

在 C 中，char 类型的长度是一个字节。尽管在多数流行的硬件系统中是用 8 个比特来组成一个字节的，但是，考虑到实际上有的硬件系统使用 9 个比特（或者更多），所以，C 标准委员会决定将 8 比特作为最低限度的要求。各个 C 的实现可以根据自己的情况在此基础上予以加长，但是都必须在头文件<limits.h>中定义一个宏 CHAR_BIT，指示一个字节所包含的比特数。

char 是整数类型，尽管历史上经常用 char 类型的对象来保存字符的编码值（基本执行字符集的成员）。例如：

```
/* c0 所指示的对象，其值为字母 A 的编码，是一个正的整数值 */
char c0 = 'A', c1 = 97;
```

“char”的名称反映了这样一种事实：在早期，人们用 char 类型来处理他们用到的所有字符，那个时候的字符编码接近或等于存储器的最小可寻址单元——字节，所以 char 类型被定义为一个字节的宽度，从而又被用于作为最小的整数类型。但是，随着计算机的飞速发展，以及它在各国和各行业的广泛应用，字符集和字符编码的国际化问题被提了出来，char 类型不可能表示所有字符，如汉字。

任何基本执行字符集的成员都是单字节的，可以用 char 类型来表示。进一步地，如果一个 char 类型的对象保存的是基本执行字符集的成员，则该对象的值一定不是负数。至于其他字符，则无法保证它们的符号性。一方面，char 类型到底是有符号的整数类型，还是无符号的整数类型，这取决于 C 实现；另一方面，基本执行字符集之外的成员不保证具有小于 128 的编码值。

char、signed char 和 unsigned char 都是一个字节的长度，如果 char 是有符号整数类型，则它在取值范围、类型的表示和行为方面与 signed char 一样；否则和 unsigned char 一样。标准要求每一个 C 实现都必须做出这种选择。

```
/*
**下面的代码可以确定正在使用的 C 的实现是如何定义 char 类型的
**另一个方法是查看 limits.h 文件，找到 CHAR_MIN，看它是 0 还是 SCHAR_MIN。
**如果使用的是 gcc，则可以使用-fsigned-char 或者-funsigned-char 编译
**选项来强制 char 表现得像 signed char 或者 unsigned char 一样
*/
printf ("Type char is equivalent to %ssigned char.", \
                                        (char)-1>0 ? "un" : "");
```

其中，常量表达式-1 的类型是 int，但它的值很小，可以被 signed char 类型的对象容纳

(可以转换为 signed char 类型而保持不变)，也可以转化为 unsigned char 类型的值。若 char 类型等效于 signed char，则表达式(char)-1 的值和-1 相同，且小于 0；否则，表达式(char)-1 的结果是一个正值，且为 unsigned char 类型所能容许的最大值。

最后，不管 char 是被定义为等价于 signed char，还是 unsigned char，它们都是 3 种截然不同的类型，而且互不兼容。

3.3 标准整数类型

这是一个统称，标准整数类型（standard integer types）包括标准无符号整数类型和标准有符号整数类型，具体可参见其各自的词条。

任何一个有符号整数类型的对象，如果它当前的值不是负数，则这个值也必然能被与之相对应的无符号整数类型表示。而且，(这意味着）有符号整数类型在表示一个非负值的时候，与它对应的无符号整数类型相比，在方法上是一模一样的。

这就是说，例如，任何一个 signed int 类型的非负值（从 0 到 INT_MAX)，总是能够用 unsigned int 类型的对象容纳，而且它们具有相同的内部表示。

3.4 扩展整数类型

这是一个统称，扩展整数类型（extended integer types）包括扩展无符号整数类型和扩展有符号整数类型，具体可参见其各自的词条。扩展无符号整数类型和扩展有符号整数类型是两种互相对应的类型。

3.5 算术类型

这是一个统称，算术类型（arithmetic type）包括浮点类型和整数类型，具体可参见其各自的词条。

3.5.1 整数类型

整数类型（integer type）包括无符号整数类型、有符号整数类型、枚举类型和 char（类型)。要了解这些类型的具体描述，可参见其各自的词条。

在 C 中，整数类型的宽度决定了该类型的对象可以拥有哪些值，也就是值的范围。问题在于，对于绝大多数整数类型来说，C 实现可以自由决定它们的实际宽度，只要能够保证取值范围符合标准的最低要求即可。

举例来说，标准要求 int 类型的最小值不得大于-32767，最大值不得小于+32767，但对于很多 C 实现来说，int 类型的取值范围实际上是-2147483647~+2147483647。

这当然不算什么大问题，但对于有些人来说并不是特别方便。所以，从 C99 开始，标准引入了头文件<stdint.h>，该文件定义了一些具有固定宽度的整数类型，现将那些所有 C 实现都必须定义的整数类型列举如下：

（1）具有指定最小宽度的整数类型。它们是 int_least8_t、uint_least8_t、int_least16_t、uint_least16_t、int_least32_t、uint_least32_t、int_least64_t 和 uint_least64_t。

显然，这些整数类型的类型名具有这样的一般形式：

```
int_leastN_t
uint_leastN_t
```

这里，以“int”打头的是有符号整数类型，宽度至少为 N 个比特；以“uint”打头的是无符号整数类型，宽度至少为 N 个比特。

（2）具有最小宽度的最快整数类型。它们是 int_fast8_t、int_fast16_t、int_fast32_t、int_fast64_t、uint_fast8_t、uint_fast16_t、uint_fast32_t 和 uint_fast64_t。

显然，这些整数类型的类型名具有这样的一般形式：

```
int_fastN_t
uint_fastN_t
```

这里，以“int”打头的是有符号整数类型，宽度至少为 N 个比特；以“uint”打头的是无符号整数类型，宽度至少为 N 个比特。另外，它们相对于等宽的其他整数类型来说，操作速度通常是最快的。当然，标准和 C 实现并不保证在所有场合下都能够最快。

（3）最大宽度的整数类型 intmax_t 和 uintmax_t。前者保证可以容纳所有有符号整数类型的值；后者保证可以容纳所有无符号整数类型的值。

在下例中，定义了函数 f，它接受两个 int_least64_t 类型的参数，这两个参数的宽度都至少为 64 个比特。该函数的返回类型是 intmax_t，保证可以容纳所有有符号整数类型的值。

```
# include <stdint.h>

intmax_t f (int_least64_t x, int_least64_t y) {/* ..... */}
```

3.5.1.1 枚举类型

“枚举”听起来像是掰着指头数数，它们之间确实有一些相像之处。在程序中使用整型常量是很普遍的事，但比起直接使用这些数字，将它们按某种性质或者说含义组织起来，似乎更有意义。

枚举指汇集一些有意义的、命名的整型常量以方便在程序中使用。每个特定的这种汇集和集合，都会形成一种特定的枚举类型（enumerated type）。或者说，每个不同的枚举被视为一种不同的枚举类型。

每个枚举都是以关键字“enum”引导的，然后是一个可选的标识符（枚举标记或者简称标记），最后是位于一对花括号之内的枚举成员，称为枚举器列表。有关枚举类型的声明，可参见“枚举指定符”。

例如，下面定义了一个用于表示开关状态的枚举类型 enum Switch。

```
enum Switch {ON = 0, OFF = 1};
```

每个枚举都是由一些被定义为枚举常量的标识符组成的，这些枚举常量的类型是 int，请

参见“枚举常量”。每个枚举常量都有一个值，定义枚举常量值的方法是在枚举常量的后面用“=”连接一个整型常量表达式。在上例中，分别用常量表达式 0 和 1 定义了枚举常量 ON 和 OFF 的值。因此，枚举常量 ON 和 OFF 的类型都是 int，它们的值分别是 0 和 1。

枚举常量的值如果是用一个常量表达式来定义的，则常量表达式的值不能超出 int 类型所能表示的值的范围。据此，下面两个枚举类型定义是错误的。

```
# include <limits.h>
# include <stdio.h>

enum Switch {ON = 0.0, OFF = 1LL};
enum Cordin {X = printf ("x"), Y = INT_MAX + 1, Z = LLONG_MAX};
```

以上，用于定义枚举常量 ON 的表达式 0.0 是浮点常量，而不是整型常量，且不能自动转换为整型常量，所以非法；定义枚举常量 OFF 的表达式 1LL，其类型是 long long int，可自动转换为 int 类型，且转换后的值是一样的，在 int 类型可以表示的值的范围内，所以没有问题；定义枚举常量 X 的表达式是函数调用表达式，虽然该函数返回一个 int 类型的值，但这个表达式并不是常量表达式，所以非法；定义枚举常量 Y 的表达式也是 int 类型，但是这个表达式的值无法用 int 类型来表示，所以非法；定义枚举常量 Z 的表达式 LLONG_MAX，其类型是 long long int，这个值很大，如果 int 类型的取值范围是 long long int 类型取值范围的真子集（这是当前最普遍的情况），则它无法转换为 int 类型的值，自然会是非法的，否则就没有问题。

对于任何枚举类型，如果枚举器列表中的第一个枚举常量没有用“=”连接常量表达式，则它的常量值是 0。

从枚举器列表中的第二个枚举常量开始，如果某个枚举常量没有用“=”连接常量表达式，则它的常量值是前一个枚举常量的值加一。

例如，Left、Right、Up、Down 的常量值分别为 0、1、2、3：

```
enum e1 {Left, Right, Up, Down,};
```

又如，Sun、Moon、Earth 的常量值分别是 3、6 和 7。

```
enum e2 {Sun = 3, Moon = 6, Earth,};
```

一个枚举常量的值可以和另一个枚举常量相同，这是允许的。如果有这种需要，则必须使用“=”来生成一个与其他枚举常量具有相同值的枚举常量。

```
enum Choice {YES = 0, NO, OK = 0, CANCEL};
enum Status {Close = 0, Off = 0, On = 1, Open = On};
```

被定义为枚举常量的标识符有其自己的作用域，不要被组成枚举器列表的花括号所迷惑，被定义为枚举常量的标识符，其作用域并非仅限于花括号之内，实际上，被定义为枚举常量的标识符被划归普通标识符（参见“名字空间”），适用于普通标识符的作用域规则也同样适用于枚举常量。两个以上的枚举常量定义都使用相同标识符的，不得具有相同的作用域。

在下例中，D1 中的枚举常量 X 和 Y 仅具有函数原型作用域，在程序中的其他位置不可见；D2 中的枚举常量 ON 和 OFF 具有文件作用域。

D3 中的枚举常量 ON 和 OFF 位于内部块中，具有块作用域。D3 中定义的枚举类型 enum Switch 隐藏了 D2 中定义的枚举类型 enum Switch，而且 D3 中定义的枚举常量 ON 和 OFF 也隐藏了 D2 中定义的枚举常量 ON 和 OFF。

D4 中定义的枚举类型 enum Cordin，具有块作用域。但它并非是隐藏了 D1 中的枚举类型 enum Cordin，因为后者仅具有函数原型作用域，即使没有 D4，它在此处也不可见。

D5 中的枚举常量 X 来自 D4，而不是 D1，原因同上。

D6 中的枚举常量 ON 来自 D3，因为 D3 隐藏了 D2。

除了 D1 和 D4，如果枚举常量 Y 在当前转换单元内没有其他定义，则 D7 是非法的，因为在这个点上，D1 中的枚举常量 Y 和 D4 中的枚举常量 Y 都不可见，前者仅具有函数原型作用域，后者仅具有块作用域。

D8 中的枚举常量 OFF 来自于 D2 而不是 D3，因为后者仅具有块作用域。

```
void f (enum Cordin {X, Y});              //D1

enum Switch {ON, OFF} g (void)            //D2
{
    enum Switch {ON = 1, OFF = 0};        //D3
    enum Cordin {X, Y};                   //D4
    int x = X;                            //D5
    int on = ON;                          //D6
    /* ..... */
}

int y = Y;                                //D7
int off = OFF;                            //D8
```

枚举常量在名字空间上被划归为普通标识符，而且是无链接的，所以不允许和另一些同名的普通标识符有相同的作用域。下例中，枚举类型 e1 和 e2 都包含了枚举常量 Flower，这是非法的；同样，枚举常量 Grass 和声明为 char 类型的标识符 Grass 重名，这也是不允许的。

```
enum e1 {Tree, Flower};
enum e2 {Grass, Flower};
char Grass = 0;
```

再如下例，D1 中的枚举常量 e 和枚举类型标记 e 相同，但它们分属不同的名字空间，虽然容易混淆，但这是允许的：

```
enum e {a, b, c, d, e,};          //D1
enum e e;                         //D2: 非法
enum e f;                         //D3
struct a {char a; double d;};     //D4
char a;                           //D5: 非法
struct e {char a; double d;};     //D6: 非法
```

D2 的意思是声明一个枚举类型的对象 e，但 e 和 D1 中的枚举常量 e 位于同一个名字空间，都是普通标识符，因此该声明非法；

D3 和 D2 功能相同，但不存在 D2 中的问题，所以是合法的；

D4 复杂点，但同样合法。因为 a 和 d 是结构成员，有独立的名字空间，不会与结构标记

a 和 D1 中的枚举常量 a、d 冲突。同时，结构标记 a 和 D1 中的枚举常量 a 也分属不同的名字空间；

D5 是非法的，因为这里的 a 是普通标识符，而 D1 中的枚举常量 a 也是普通标识符，属于同一个名字空间；

D6 也是非法的，所有标记都属于同一个名字空间，在 D1 中的 e 是枚举标记，它不能再做为结构标记。和 D4 一样，结构成员 a 和 d 都有自己的名字空间，不会与 D4 的结构成员冲突，也不会与 D1 中的枚举常量 a、d 冲突。

每一个枚举能够包含多少个枚举常量，这在数量上是有所限制的，标准的要求是保证不少于 1023 个。当然，这是最低限度的要求，C 实现可能会允许更多，但使用更多的枚举常量会使程序缺乏可移植性。

枚举类型属于整数类型的一种，但到底是哪种整数类型则不是固定的，取决于 C 实现。它可能和 char、int 类型兼容，也可能兼容于任何其他有符号或者无符号整数类型。但是，对于每个特定的枚举类型 E 来说，不管它兼容于哪个整数类型，这种整数类型都必须能够表示 E 的每一个枚举常量的值。也正是因为这个原因，在程序转换期间，C 实现将推迟这种（整数类型的）选择，直至看到最后一个枚举常量。

下面的例子用于确定和打印枚举类型 enum e 与哪种整数类型兼容，同时也演示了如何在程序中使用枚举类型。

```
# include <stdio.h>

enum e {Row, Col};

void fenum (enum e * pe)
{
    printf(_Generic (* pe,
                     char : "char.\n",
                     signed : "signed int.\n",
                     unsigned : "unsigned int.\n",
                     default : "unknown.\n")
           );
    if (* pe == Row) {/* …… */}
}

int main (void)
{
    enum e e = Col;
    fenum (& e);
    return 0;
}
```

下面是另一些例子，在编者的机器上，CHAR_BIT 被定义为整型常量 8，则枚举类型

enum Colours 和 enum Weekday 可能兼容于 char，也可能兼容于 int 或者 unsigned int。然而，枚举类型 enum Mountain 绝不会兼容于 char，因为枚举常量 Everest、Chogori 和 Konka 的值不能用 char 类型来表示。

```
//以下是用于描述赤橙黄绿青蓝紫的枚举类型
enum Colors {red, orange, yellow, green, blue, indigo, purple};
//以下是用于描述5个工作日的枚举类型
enum Weekday {Mon = 1, Tue, Wed, Thu, Fri};
//以下是描述珠穆朗玛峰、乔戈里峰和贡嘎雪山高度的枚举类型
enum Mountain {Everest = 8844, Chogori = 8611, Konka = 7556};
```

枚举类型是整数类型的一种，可以说它兼容于 char、int 或者 unsigned int，但不能说它就是 char、int 或者 unsigned int，毕竟它们不是相同的类型。枚举类型就是枚举类型，就像 char 可能等价于 signed char，也可能等价于 unsigned char 一样，但它们并不是相同的类型，这是很多人容易犯的错误。

因为枚举类型是整数类型的一种，因此，它可以像其他整数类型一样参与各种运算，也可以定义枚举类型的对象，作为函数的返回类型和参数类型。例如：

```
enum Colors {red, orange, yellow, green, blue, indigo, purple};

enum Colors paint (enum Colors cl) //定义参数和返回类型为enum Colors的函数
{
    if (cl == blue) return cl;      //在表达式中使用枚举类型
    /* ..... */
}

int main (void)
{
    enum Colors cl = red;       //定义enum Colors类型的对象
    return paint (cl);
}
```

在声明任何枚举类型时，用于终止枚举器列表的“}”出现之前，该枚举类型尚属于不完整类型；在此之后，则成为完整类型。下例中，第一行中的 B = sizeof (enum e)是非法的，因为此时枚举类型 enum e 还是不完整的；相反，第二行是合法的。

```
enum e {A, B = sizeof (enum e)};    //非法
enum f {C = sizeof (enum e), D};    //允许
```

枚举类型是对象类型，可以将一个枚举类型的值（可能需要经过适当的转换）赋给另一个枚举对象，前提是它们具有相同的（枚举）类型。例如：

```
enum e {Disabled, Enabled,} e1 = {Enabled}, e2;
e2 = e1;
```

3.6 实数类型

实数类型（real type）包括实浮点类型和整数类型，具体可参见其各自的词条。

3.7 字符类型

这是一个统称，字符类型包括 char、signed char 和 unsigned char，具体可参见其各自的词条。

传统的 C 语言只有 char 类型。C89 引入了有符号的字符类型 signed char 和无符号的字符类型 unsigned char。

由于历史的原因，char 类型可能在类型的表示等方面等价于 signed char，也可能等价于 unsigned char，至于到底等价于哪一种则取决于 C 实现。

同样地，char 类型的对象，其所能容纳的最小值 CHAR_MIN 和最大值 CHAR_MAX 可能分别等于 SCHAR_MIN 和 SCHAR_MAX，也可能分别等于 0 和 UCHAR_MAX（这几个宏在头文件<limits.h>中定义）。

这意味着，要想判断某个 C 实现是如何对待 char、signed char 和 unsigned char 这 3 种类型的，只需要判断 CHAR_MIN 是否等于 SCHAR_MIN，以及 CHAR_MAX 是否等于 SCHAR_MAX 即可。

注意：char、signed char 和 unsigned char 是 3 种截然不同的类型，而且是互不兼容的 3 种类型。

3.8 派生类型

在 C 语言中，可以在现有类型的基础上构建新的类型，这个构建新类型的过程称为类型派生（type derivation），简称派生。进一步地，从其他类型派生出来的新类型称为派生类型（derived types）。可以从对象或者函数类型构建出任意数量的派生类型，包括数组类型、结构类型、联合类型、函数类型、指针类型和原子类型。这些构建派生类型的方法可以递归使用。

结构类型、联合类型、数组类型、指针类型、函数类型、原子类型都属于派生类型。结构和联合类型派生自它的成员类型；数组类型派生自它的元素类型；指针类型派生自它所指向的类型；函数类型被认为是派生自它的返回类型，具体可参见其各自的词条。

以下，在 D1 中声明了结构类型 struct t，该类型派生自它的成员类型 char 和 float；在 D2 中，标识符 d 的类型是数组，该数组类型派生自它的元素类型 double；在 D3 中，标识符 pt 的类型是指针，该指针派生自它所指向的类型 struct t；在 D4 中，标识符 f 的类型是函数，该函数类型派生自它的返回类型 struct t。

```
struct t {char c; float f;};    //D1
```

```
double d [5];                    //D2
struct t * pt;                   //D3
struct t f (void);               //D4
```

在多数声明中，对象的类型完全由声明指定符决定。例如：

```
int i;
long long ago;
struct t t;
```

其中，i 是 int 类型；ago 是 long long int 类型；t 是结构类型。

类型的派生可以从声明符（参见“声明符”）一侧进行，这些从声明符一侧派生出来的新类型称为派生的声明符类型（derived declarator types）。派生的声明符类型包括指针、数组和函数。例如：

```
int (* p) (void);
float a [3];
double f (void);
```

其中，在每个声明中加粗的部分都是声明符，但声明的类型都是派生类型：p 是指针；a 是数组；f 是函数。

又比如下面的例子，D1 中的 pi 是指向 int 类型的指针；D2 中的 ago 是 20 个元素的数组，数组的元素类型是 long long；D3 中的 fd 是具有 1 个 double 类型的参数，且返回 double 类型的函数；D4 中的 ta 是 3 个元素的数组，数组的元素类型是 struct t；D5 中的 p 是指向数组的指针，数组的元素类型是 struct t；D6 中的 q 是 5 个元素的数组，该数组的元素类型是指向 struct t 的指针。

```
int * pi;                //D1
long long ago [20];      //D2
double fd (double);      //D3

struct t ta [3];         //D4
struct t (* p) [5];      //D5
struct t * q [5];        //D6
```

3.8.1 数组（类型）

数组（array）是一种派生类型，它用于描述这样的对象：由一系列在存储器中连续分配的子对象（元素）组成，而且这些对象都具有同一类型。这里，“连续”的意思是强调要按顺序一个挨着一个。这些元素都具有相同的类型，称为元素类型。声明数组类型的时候，元素类型要求是完整类型（参见“完整类型”）。

不同的数组类型是以它的元素类型和元素数量为特征的。也就是说，元素数量不同，或者元素类型不同的数组是不同的数组类型。数组类型派生自它的元素类型，如果元素的类型是 T，则数组类型经常被称为“T 的数组”。

数组是从声明符一侧派生的，在数组类型的声明中，声明指定符用于指定数组的元素类型。数组类型从元素类型构建被称为“数组类型派生”。有关数组声明的具体描述，可参见词

条“声明符”。

因为数组类型是从声明符一侧派生的，因此属于派生的声明符类型。下面的示例说明了数组的声明方法：先是一个左中括号“[”，然后是数组元素的个数，最后是一个匹配的右中括号“]”，例如：

```
int ia [100];
```

其声明了一个包含 100 个元素的数组，数组的元素类型是 int，故可以称为“int 类型的数组”。而且，int 类型是完整类型，符合数组类型定义的要求。

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一个数组，即数组的数组，这样的数组通常称为多维数组。例如：

```
int ib [2][3];
```

其声明了一个数组 ib，如图 3-2（a）所示，ib 是具有 2 个 int [3]类型的元素的数组。因此，ib 是数组的数组。

声明一个更多维数组是可能的。例如：

```
char c [2][3][5];
```

声明标识符 c 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 int 类型的元素的数组。该数组内部构成如图 3-2（b）所示。

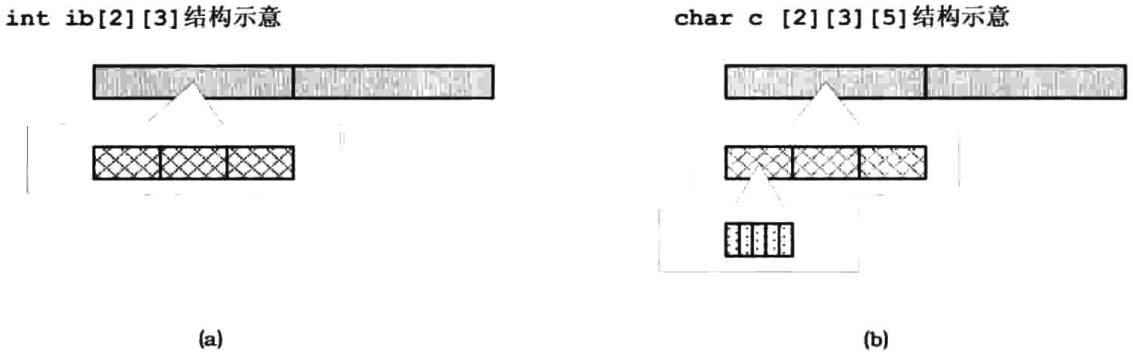


图 3-2 多维数组内部构成

数组类型派生自它的元素类型，所以，通过递归构建的多维数组，其本质是一维数组，只不过它的元素类型依然是数组，或者说，是一个派生自其他数组类型的数组。

下面是数组类型派生自元素类型的更多示例。但是，要理解它们，需要先参考“初始化器”、“声明符”、“结构”、“字面串”等相关内容。

```
# include <stdio.h>

struct t {char c; int i;} at [3] = {{'x', 0}, {'y', 1}, {'z', 2}};
char * ap [3] = {"One", "Two", "Three"};
char aa [2] [100] = {"Tree", "Flower"};
char (* apa [2]) [100] = {& aa [0], & aa [1]};
int (* afp [2]) (const char * restrict, ...) = {printf, printf};
```

首先，数组 at 的元素类型是结构 struct t，它有 3 个元素，并分别进行了初始化（参见“初始化”）；数组 ap 有 3 个元素，元素的类型是指向 char 的指针，即 char *。数组 ap 的初始

化器中又包含了三个初始化器"One"、"Two"和"Three"，C 实现先是用它们创建三个隐藏的静态数组（请参见“字面串”），然后将这三个静态数组转换为指向其首元素的指针，并分别用于初始化数组 ap 的三个元素；

数组 aa 有 2 个元素，元素的类型依然是数组，即 char [100]。表面上看，数组 aa 的初始化器和 ap 类似，但它们工作的方式不同。在这里，字面串"Tree"和"Flower"先是被用于创建两个隐藏的静态数组，然后，因为 aa 的元素类型也是数组，所以，是用这两个静态数组的内容分别初始化 aa 的两个元素；

数组 apa 有 2 个元素，元素的类型是指向数组的指针，即 char (*) [100]。数组 apa 的初始化器很好理解，aa[0]和 aa[1]的类型是数组，即 char [100]，&aa[0]和&aa[1]的类型则是指向数组的指针，即 char (*) [100]，与 apa 的元素类型兼容，故直接用于初始化。

数组 afp 有 2 个元素，元素的类型是指向函数的指针，即

```
int (*) (const char * restrict, ... )
```

显然，被指向的函数与库函数 printf 是兼容的，可直接用 printf 函数的地址来初始化这 2 个元素。顺便地，以下给出了使用这些数组的方法：

```
int n = 0, arr [5];
arr [n ++] = at [0].c;
arr [n ++] = * (ap [0]);
arr [n ++] = aa [0] [0];
arr [n ++] = (* (apa [0])) [0];
arr [n ++] = afp [0] ("ISO/IEC 9899:2011");
```

以上，表达式 at[0]的类型是结构，表达式 at[0].c 得到结构成员 c 的值；表达式 ap[0]的类型是指向 char 的指针，表达式*(ap[0])的结果是得到“指针所指向的对象”的值；表达式 aa[0]的类型是数组，表达式 aa[0][0]得到数组第一个元素的值；表达式 apa[0]的类型是指向数组的指针，表达式* (apa [0])得到那个数组，表达式(* (apa [0])) [0]得到数组第一个元素的值；表达式 afp[0]的类型是指向函数的指针，表达式 afp [0] ("ISO/IEC 9899:2011") 调用那个函数并得到返回值。

数组的元素类型要求是完整类型。因此，下面的两个声明都是不合法的。数组 uc 是具有 3 个元素的数组，元素类型是 unsigned char []，但是这个元素类型的大小未知，即元素的类型是不完整类型；数组 v 元素的类型不能是 void，因为 void 是不完整类型。

```
unsigned char uc[3][];
void v [30];
```

以下的例子演示如何使用数组（使用下标访问数组元素和赋值操作）：S1 是给数组的第一个元素赋值，数组元素的下标总是从 0 开始；S2 是给数组的第二个元素赋值；因为数组 c 只有两个元素，所以 S3 是危险的，下标越界。但是 C 实现不检查数组下标是否越界；S4 中，表达式 z[1][0]的类型是 char，表达式 c[0]的类型也是 char，因此可以直接赋值。

```
void f (void)
{
    char c [2], z [2][2];
```

```
    c [0] = 22;     //S1
    c [1] = 33;     //S2
    c [2] = 55;     //S3

    z [0][0] = 0;
    z [0][1] = 1;
    z [1][0] = c[0];    //S4

    z [1][1] = c[1];
}
```

数组类型是以它的元素类型和元素数量为特征的。因此，元素类型不同，或者大小不同的数组属于不同的数组类型。

例如，对于下列声明，数组 a 和 c 尽管大小相同，但元素类型不同，故 a 和 c 属于不同类型；尽管元素的数量相同，但 fa 和 fb 也属于不同类型，因为前者的元素类型是 float [30]，而后者则是 float [50]：

```
char c [100];
int a [100];
float fa [100][30], fb [100][50];
```

数组类型的表达式不是可修改的左值（参见“表达式”、“左值”、“可修改的左值”）。因此，两个数组类型的对象不允许直接赋值，即使它们具有相同的类型（相同的大小和相同的元素类型）。下面的例子演示了如何给数组的每个元素赋值。

```
#define N 100

int i, a [N], b [N];
   /*一些初始化数组 a 的代码 */
b = a;              //非法
i = 0;
while (i < N)    //常规的做法是在对应的元素之间赋值
{
   b [i] = a [i];
   i = i + 1;
}
```

下面的代码很多人都很熟悉，函数 cccpy 用于将一个'\0'结尾的字符数组，也就是字符串，复制到另一个数组：

```
# include <stdio.h>

char * cccpy (const char * src, char * dst)
{
```

```
    char * tmp = dst;
    while ((* (dst ++) = * (src ++)) != '\0') ;
    return tmp;
}

void g (void)
{
    char a [30];
    printf (cccpy ("LeeChung.\n", a));
}
```

如果想在数组之间赋值，或者从函数返回一个数组（函数的返回类型不能是数组），可以用结构（参见“结构”）对数组进行包装，就像这样：

```
# include <stdio.h>

struct t {char a [33];} f (void)
{
    return (struct t) {"All roads lead to Rome."};    //S1
}

int main (void)
{
    struct t t1, t2;
    t1 = f ();                                        //S2
    t2 = t1;                                          //S3
    return printf (t2.a);
}
```

在上例的 S1 中，我们用复合字面值（参见“复合字面值”）创建了一个没有名字的、临时的结构对象，并返回这个对象给调用者。

然后，在 S2 和 S3 中，我们演示了如何将结构类型的值赋给结构对象，尤其是这种结构类型包含数组成员。当我们把一个结构类型的值赋给一个结构对象时，结构中的数组成员也被拷贝（复制），除非它是弹性数组成员（参见“弹性数组成员”）。

以前，数组在声明时，用于指定其长度的表达式只能是整型常量表达式。从 C99 开始，数组在声明时，不再要求它的大小必须是一个常量，这称为变长数组。有关变长数组的详细描述可参见“变长数组”。

3.8.1.1 变长数组

以前，声明一个数组时必须指定常量大小，但现在这种限制已经不太严格。如果 C 实现允许，那么，一个数组在声明时所指定的大小可以不是一个（整型）常量，或者其元素的类型不具有常量大小。在这种情况下，所声明的是一个变长数组（Variable Length Array，VLA）。

变长数组是从 C99 开始引入的，但标准并不强制 C 实现必须支持变长数组。以下是几个变长数组的例子。

```
void fvla (int x)
{
    float f [x];          //元素数量（长度）不具有常量大小
    double d [30] [x];    //元素类型不具有常量大小
    /* ..... */
}
```

对象 f 的长度不具有常量大小，这很直观，但是对象 d 的元素类型不具有常量大小，这不直观。实际上，如果将那一行分成两行就直观了，即：

```
typedef double tv [x];
tv d [30];
```

现在，对象 d 的元素类型是 tv，而 tv 不具有常量大小。

注意，变长数组不会随着程序的执行而动态改变其长度。相反，它的长度是在声明时确定的，但不是一个常量，而是来自一个非常量表达式的值。在声明或者定义之后，便不再改变其大小。例如：

```
char x = 2, a [x];  //在当前点上，数组 a 的大小是 2 个字节
x = x + 1;          //增加 x 的大小，并不会改变数组 a 的大小
a [2] = 9;          //危险！依然只有 a[0] 和 a[1] 是有效的
```

非常量的表达式只在程序执行的特定时间求值。例如，它不能在程序启动时（main 函数执行之前）求值。既然变长数组的大小不是一个常量（表达式），因此一个标识符不可能既具有文件作用域，又被声明为变长数组类型。

下面给出了合法和不合法的例子。特别地，企图将 vint 声明为静态存储期的对象，这是非法的，变长数组对象和生存期和其他对象不完全相同，参见“存储期”和“生存期”。

```
int n = 5;
float vla [n];          //不允许（文件作用域）
int g (int vla [n]);    //允许（函数原型作用域而非文件作用域）

void f (int n)
{
    int vla [n];        //允许（块作用域）
    static vint [n];//非法：变长数组对象不能具有静态存储期
/* ..... */
}
```

上面这个例子同时也显示了变长数组与其他数组的不同之处：每次进入函数 f 内部时，都会创建一个新的数组，其长度为 n（随着传入的参数而变化），但是在创建之后，数组的大小不再改变。

变长数组在声明时不能有初始化器。因此，下面的声明是非法的（它的声明符没有问题，错在初始化器）。

```
char vla [n ++] = "variably modified.";
```

变长数组对象只在程序的执行到达其声明的位置时创建（参见“存储期”和“生存期”），而不是在程序启动时创建，也不可能在线程启动时创建。换句话说，变长数组类型的对象不会具有静态存储期或者线程存储期，例如：

```
int n = 5;
float vlax [n];                    //非法，企图使 vlax 具有静态存储期

void f (int n)
{
    static int vlay [n];           //非法，企图使 vlay 具有静态存储期
    int vlaz [n];                  //合法
}
```

从前面的叙述来看，如果一个标识符被声明为变长数组类型，那它只能是无链接的。进一步地，不可能用外部或者内部链接来使同一个标识符的多个声明都指示同一个变长数组对象。在下例中，D 处的声明是非法的，因为它企图使变长数组类型的标识符具有外部链接。

```
void f (int n)
{
    int vla [n];
    if (/* …… */)
    {
        extern int vla [n]; //D
        /* …… */
    }
}
```

从表面上来看，函数的参数也可以是变长数组，但是它仍然被调整为指向数组首元素的指针。在下例中，尽管可以在函数 f 的内部访问对象 n，但它并不是来自函数参数，而是来自 D 处的声明，语句 S 的输出印证了这一点。同时，在函数 f 内，a 依然被转换为指针。数组 a 的大小在它在生存期间不会改变，即使在 while 语句中改变了 n 的值。

```
# include <stdio.h>

int n;                             //D
void f (int a [n]);

void g (void)
{
    int a [n = 3];
    a [0] = 1;
    a [1] = 3;
    a [2] = 5;
```

```
    f (a);
    printf ("%d\n", n);     //S
}

void f (int a [n])
{
    while (n --)
        printf ("%d\n", a [n]);
}
```

变长数组属于可变修改类型，具体可请参见“可变修改类型”。

3.8.1.2 可变修改类型

在任何一个声明中，如果全声明符（参见“全声明符”）的某个组成部分包含了变长数组类型，则该全声明符被称为是“可变修改的（variably modified）”，它所声明的类型被认为是可变修改类型（variably modified type）。

递归地，从其他可变修改类型派生的类型同样是可变修改的。

例如，给定声明：

```
int n = 50;
int (* p) [n];
```

则 p 的类型是可变修改的。

在下例中，标识符 a 的类型是可变修改的；arrPtr 的类型也是可变修改的；p 的类型同样是可变修改的。

```
void f (int n)
{
    char a [n];
    /* …… */
    typedef int * arrPtr [n];
    arrPtr p;
    for (int x = 0; x < n; x ++)
        p [x] = & a [x];
    /* …… */
}
```

3.8.2 结构

结构（structure）是一种派生类型，用于描述这样一种对象：它由一些在存储器中按顺序分配的子对象（成员对象）组成。

从 C99 开始，如果需要，还可以在尾部附加一个不完整的数组，称为弹性数组成员，具体可参见“弹性数组成员”。

数组只能聚合单一类型的数据，而很多事物具有复杂的、不同类型的属性。例如，人可

以有名字、性别、身高、年龄等属性。和数组不同，不要求结构的成员属于同一类型，它们的类型可能相同，也可能不同。使用结构类型的好处是可以将不同类型的数据组织到一起以方便处理。

有关结构类型的声明，可参见“结构或联合指定符”，这里只做简略性的描述。例如，以下声明了一个标识符 tom，它指示了一个结构类型 struct t 的对象。

```
struct t
{
    char name [20];          //姓名
    float height;            //身高
    float weight;            //体重
} tom;
```

以上其实声明了结构类型 struct t 和该结构类型的对象 tom。换句话说，它们可以分开声明，像这样：

```
struct t
{
    char name [20];
    float height;
    float weight;
};
struct t tom;
```

其中，“struct”是声明结构类型必不可少的关键字；标识符“t”是标记，它的作用是将一个名字同当前所声明的结构类型相关联；一对花括号及花括号中的内容是结构的成员声明，给出了当前结构是由哪些类型组成的。

在上述声明中，

```
struct {char name [20]; float height; float weight;}
```

是类型指定符，用于指定一种结构类型，所以又称结构或联合指定符；花括号内部是结构声明列表，即成员列表。

根据上述定义，结构的成员列表不能是空的，至少要有一个成员。当然，尽管标准是这样要求的，但是如下例所示，很多 C 实现对此加以扩展，并容许没有成员的结构。如果程序对可移植性有要求，则应避免使用这种依赖实现的特性。

```
struct {} s;                 //非法，成员数量不能为零
```

结构的成员在数量上是有所限制的，具体的数量由 C 实现自己决定，但标准要求的最低限度是保证不少于 1023 个。

结构的成员必须属于完整的对象类型，而且不能是变长数组。但从 C99 开始，结构的最后一个成员可以是不完整的数组类型，称为弹性数组成员。参见“弹性数组成员”。由于此限制，想让结构或联合的成员是函数类型或者 void 类型是不可能的（void 类型也是对象类型，但它是不完整的类型）。下面给出了一个不合法的结构声明。

```
void f (int n)
{
```

```
    struct s {char c; int a [n];};
    /* ..... */
}
```

在本书中，将混用“结构”和“结构类型”。有时，这两个概念用于泛指各种不同的结构类型，即结构类型的总称；而有时，这两个概念指代某个具体的结构类型。具体是什么意思，取决于它们所在的上下文语境。

为了访问结构的成员，可以使用后缀运算符“.”或者“->”。当用于访问结构对象的成员时，运算符“.”要求它的左操作数是结构类型的表达式，右操作数是结构成员的名称；运算符“->”要求它的左操作数是指向结构对象的指针，右操作数是结构成员的名称。详情可参见“后缀表达式”和“成员选择”。下例中，s 是结构对象，而 ps 是指向结构对象的指针，语句 S1 和 S2 分别显示了如何通过它们访问结构的成员。

```
struct s {char c; float f;} s, * ps = & s;
s.c = 'x';          //S1
ps -> f = 0.01;     //S2
```

不要僵化地看待事物，例如，对于上面的 S1 和 S2，它们分别等效于下面的语句。

```
(& s) -> c = 'x';
(* ps).f = 0.01;
```

表达式&s 的结果是指向 s 的指针，而表达式*ps 的结果是结构对象。尤其要说明的是，运算符“.”和“->”不要求它的左操作数必须是一个左值。所以，尽管&s 的结果不是左值，它也能作为运算符“->”的左操作数。

大家知道，每种对象类型都有自己的对齐要求。这种对齐要求不会因为它们是结构的成员类型而发生改变。这就是说，对于任何类型 T，如果它也是某个结构成员的类型（位字段成员除外），则该成员将按照 T 所要求的方式对齐。当然，每种类型应当如何对齐是 C 实现定义的。假定对于某个 C 实现来说，long long 类型的对象对齐于 8 的整数倍的字节地址上，则即使该类型的对象以结构的成员而存在，也必须按同样的方式对齐。

```
struct t {char c; long long l;};        //成员 l 的对齐是 8
```

由于结构的每个成员（对象）都有自己的对齐要求，这将导致它们之间有可能会存在间隙（空洞），这就是内部填充。

结构成员的大小和它们的对齐要求又决定了整个结构类型（对象）的对齐要求，这使得结构对象的尾部也可能会有填充以适应这种要求。

无论如何，结构对象的开头绝不会有任何填充。

下面的示例表明了结构成员的大小和对齐是如何影响整个结构类型的大小和对齐。

```
# include <stdio.h>

struct s {int i;};
struct t {char c; long long l; int i;};

void f (void)
{
```

```
    printf ("%zu, %zu\n", sizeof (struct s), _Alignof (struct s));
    printf ("%zu, %zu\n", sizeof (struct t), _Alignof (struct t));
}
```

sizeof 运算符用来计算对象或者类型的大小，以字节计。注意，它和+、-、*、/、_Alignof等一样，是运算符，而不是函数，也不是宏。

以上，如果采用以 Intel x86 32 位处理器为目标架构的 GCC 编译器（MinGW），且sizeof(int)的结果为 4，sizeof(long long)的结果为 8，则，程序的输出表明，struct t 的对齐是 8，大小是 24，在结构内部和尾部都有如图 3-3 所示的填充。

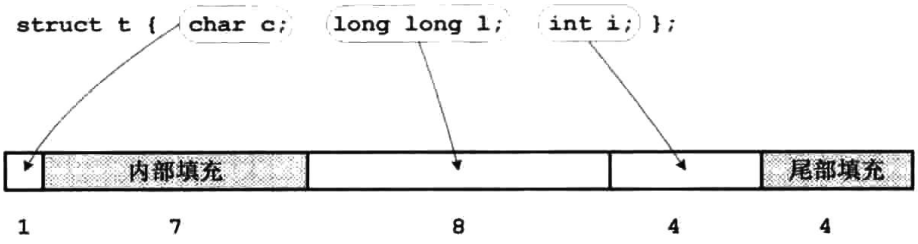


图 3-3 结构的成员对齐与填充

结构的成员可以是位字段。有关位字段成员的详细描述可参见“位字段”。

结构的成员被视为整个结构对象的子对象。在一个结构对象的内部，非位字段成员和承载位字段的存储单元，它们的地址按其声明的顺序增长。也就是说，结构成员的存储顺序和它们声明的顺序是一致的。

如果想得到结构成员在结构内的偏移，可以使用 offsetof。它是一个宏，在头文件<stddef.h>中定义，具有两个参数：

offsetof(*结构, 成员名*)

在程序中使用该宏的话，则宏调用被扩展为一个整型常量表达式，该表达式的值是结构成员距离结构起始处的偏移量，以字节计。下面是该宏的一个用法示例：

```
# include <stdio.h>
# include <stddef.h>

void f (void)
{
    struct t {char c; float f;};
    printf ("%zu\n", offsetof(struct t, f));
}
```

结构的成员可以是另一个结构，或者是一个联合，这种关系可以递归地构建，这并不奇怪。但如果结构的成员是匿名结构或者匿名联合，则它们的成员隶属关系会有所不同，具体的描述可参见“匿名结构”和“匿名联合”。

任何一个结构 S 的声明中，在用于终止成员列表的“}”出现之前，S 属于不完整类型。在此之后，S 成为完整类型。以下的声明是非法的。因为在声明成员 t 的时候，struct t 是不完整的，而成员 t 的声明需要知道 struct t 的完整信息（大小）。

```
struct t {int i; struct t t;};
```

相反，以下声明是合法的。尽管在声明成员 pt 的时候，struct t 是不完整类型，但声明一个指针并不需要有关该类型的具体信息。

```
struct t {int i; struct t * pt;};
```

以下的声明也是合法的。在声明 uv 和 uw 的时候，struct t 已经是完整类型。

```
struct t {int i; struct t * pt;} uv;
struct t uw;
```

下面的声明是非法的，在 struct t 成为完整类型之前，企图用 sizeof 运算符得到它的大小是不现实的，只能导致错误。

```
struct t {char c; int a [sizeof (struct t)];};
```

但是，一旦某个结构类型成为完整类型，则马上可以使用，例如：

```
struct t {char c; int i;} t = {'x', sizeof (struct t)};
```

这也从另一个侧面解释了为什么结构不能包含自己的实例。但是，它可以包含指向它自己实例的指针，因为指针是完整类型。在下面例子中，对 struct s 的声明是无效的。因为在到达“}”之前，struct s 是不完整类型，且结构中不能有函数类型的成员，但结构的成员可以是指向自己某个实例对象的指针，或者指向函数的指针。所以，对 struct t 的声明是有效的。

```
typedef int F (int, int);                    //定义标识符 F 为函数类型
struct s {int i; struct s s; F f;};          //无效
struct t {int i; struct t * t; F * f;};      //有效
```

前面说过，在结构对象的开头不存在填充，所以结构对象在存储器中的起始位置（地址）也是其第一个成员对象的起始位置（地址）。指向结构的指针和指向该结构第一个成员的指针之间可以互相转换，参见“指针-指针转换”。

结构类型是对象类型，可以将一个结构对象的值赋给另一个结构对象，前提是它们具有相同的类型。但是要注意，结构对象的赋值不会复制弹性数组成员的内容，具体可参见“弹性数组成员”。下面的例子展示了结构的使用、结构对象之间的直接赋值，以及如何使用后缀运算符“.”来访问结构的成员。

```
# include <stdio.h>

struct stgBOX {      //用于描述盒子的结构
    float  length,   //长
           width,    //宽
           height;   //高
};

int main (void)
{
    //b1 和 b2 用于描述一个生活中的“盒子”，具体什么盒子并不重要
    struct stgBOX b1, b2;

    //依次输入第一个盒子的长、宽、高
    scanf ("%f%f%f",& b1.length, & b1.width, & b1.height);
```

```
    b2 = b1; //结构赋值

    //输出第二个盒子的体积
    printf ("The volume of box 2 is:%.2f", b2.length *
                                           b2.width *
                                           b2.height);

    return 0;
}
```

在编程实践中，可以用结构定义一个链表，或者定义一个拥有结构元素的数组。下面的例子中，定义数组 a 和数组 t，它们各有 10 个元素，元素的类型是 struct t。

```
struct t {int i; float f;} a [10] ;
struct t t [10];
a [0].i = 9999;
a [0].f = 99.99;
t [0] = a [0];        //表达式t[0]和a[0]的类型相同，都是struct t
```

下面这个示例定义了结构类型的数组 sa，结构的内部有两个成员，分别是指向函数的指针 p 和 char 类型的数组 a:

```
# include <stdio.h>

void f (char * s) {printf (s);}

void g (void)
{
    struct t
    {
        void (* p) (char *);
        char a [100];
    } sa [3] = {
                {f, "There are bushes in view, at 9 o'clock.\n"},
                {f, "Copy that.\n"},
                {f, "Roger.\n"}
               };

    for (int i = 0; i < 3; i ++) sa [i]. p (sa [i]. a);
}
```

以上，数组 sa 有 3 个元素，且元素的类型是结构 struct t。结构 struct t 的第一个成员是指向函数的指针，被指向的函数在类型上与函数 f 相同。正是因为如此，对 sa 的初始化使用了函数 f 的地址。有关初始化的内容，参见“初始化器”。

看起来数组的每个元素都是“函数-参数”对。因此，我们依次访问数组的每一个元素（结构）并做函数调用操作。sa[i].p 依次得到函数指针，sa[i].a 依次得到指向数组首元素的指针。

3.8.2.1 位字段

结构和联合的成员可以仅由指定数量的比特组成（含符号位，如果需要的话），这样的成员称为位字段（bit field）成员。

位字段是从 C99 开始引入的，其最明显的好处是节省存储空间。位字段成员在声明时需要指定宽度，这个宽度位于一个冒号“:”之后。位字段成员的类型用于指定这些比特可以表示什么样的值。例如，在下面的结构内容定义中，包括了一个位字段成员 b，由 1 个比特组成。

```
struct t {int i; _Bool b:1;} t;
t.b = 1;     //b 的存储值为 1
```

在声明一个位字段（成员）时，它的宽度由一个整型常量表达式指定，前面不能有负号(-)；如果位字段成员的类型是 T，则整型常量表达式的值不允许超过 T 的宽度（参见“宽度”）。下面的结构声明是非法的，原因在于，位字段成员 i 的宽度是一个负值（常量本身不会是负数，但它的前面如果有负号运算符，则是一个结果为负的表达式），且位字段成员 b 的宽度超出了_Bool 类型的最大宽度。

```
# include <limits.h>

int f (void)
{
    struct t {int i:-3; _Bool b:CHAR_BIT + 1;};
    /* ..... */
}
```

只能声明整数类型的位字段。换句话说，位字段成员只能是有符号整数类型或者无符号整数类型，而且只具有指定数量的比特。

以下声明了浮点类型和指针类型的位字段，故这个声明是非法的。

```
struct t {float i:3; int * pi:6;};
```

以下这些声明都是合法的，位字段 ub 是无符号整数类型，取值范围是 0～7；位字段 sb 是有符号整数类型，取值范围是-4～+3。

```
struct {unsigned int ub:3; signed int sb:3;} x;
x.ub = 7;
x.sb = 7;
unsigned int ui = x.ub; //ui 的值为 7
signed int si = x.sb;   //si 的值是-1
```

按标准要求，位字段的类型原则上必须是_Bool、signed int（这还不同于 int，马上就要讲到）、unsigned int，以及它们的限定版本。当然，有些 C 的实现提供了超出标准之外的扩展特性，允许用户使用其他类型，甚至是原子类型来声明位字段。唯一的问题是，使用并依赖于这种扩展特性的程序是缺乏可移植性的。在下例中，位字段成员 i 可能是不允许的。但很多 C

实现提供了标准之外的扩展功能，它们可能支持这样的做法。

```
struct t {long int i: 7; char c;};
```

如果位字段成员的类型是 int（或者，它是用 int 的别名声明的），则它可能被视为 signed int，也可能被视为 unsigned int。但到底是哪一个取决于 C 实现。在下例中，位字段成员 bf 的类型是 signed int 还是 unsigned int？如果是前者，而且假定 C 实现采用对 2 的补码来表示负数，则 bf 的值是-1（它的宽度是 3，符号占了一位，实际用于表示值的比特只有 2 个）；如果是后者，则 bf 的值是 7。

```
typedef int Integer;
struct {char c; Integer bf:3;} x;
x.bf = 7;   //bf 的值可能是-1，也可能是 7
```

每个合法的位字段都需要一个可寻址的存储单元来承载。因此，C 实现总是分配一个长度足够的可寻址存储单元来保存位字段的值。如果还有剩余，且足以容纳下一个相邻的位字段，则它们会被紧凑地安排在同一个单元内。

如果 M 和 N 是前后相邻的两个位字段，且承载 M 的存储单元还有剩余，但并不足以容纳它后面的位字段 N，那么，是将 N 的一部分放在该剩余部分以实现跨单元存储，还是为 N 单独分配一个新的存储单元，由 C 实现自行决定。

位字段成员可以没有名称，被称为未命名的位字段。因为没有名称，所以没有办法在程序中使用。那么，它存在的意义何在呢？答案是，它可以用于结构内部的填充，使之具有用户所希望的内部布局。在下面的例子中，未命名的位字段成员改变了结构的大小和对齐（相对于没有第二个成员的情形）。

```
struct t {char c; int: 16;};
```

有时候，出于某些原因，我们不希望一个位字段和它后面的位字段共用一个承载它们的存储单元，或者，不希望与它前面的位字段共用一个承载它们的存储单元。为了达到这个目的，可以在它们中间插入一个宽度为 0 的位字段。需要注意的是，宽度为 0 的位字段不能有名称（即只能是一个未命名的位字段）。在下例中，因为不允许共用同一个存储单元，所以，两个结构类型的大小可能会有一倍的差距。

```
struct a {int i:3; int j:5;};
struct b {int i:3; int :0; int j:5;};
```

在结构类型 struct b 的声明中，宽度为 0 的位字段只是用来告诉 C 实现，它前面的位字段不欢迎其他位字段与它共享同一个存储空间。

3.8.2.2 弹性数组成员

一般来说，结构或者联合的成员都必须是完整的对象类型。但是从 C99 开始，允许结构类型的最后一个成员是不完整的数组，称为弹性数组成员（flexible array member）。这里还有个前提，弹性数组成员的前面必须至少还有一个成员，且它们都是完整的对象类型。

结构是非常灵活的数据类型，在大多数情况下足以满足各种业务需求。比如下面的示例，我们希望创建一个动态链表，来记录一些和工作有关的浮点数据。函数 s_create 用于创建链表的节点并设置每个成员的值。

```
# include <stdlib.h>
```

```
struct s {int id; struct s * next; float f [30];};

struct s * s_create (int id)
{
    struct s * ps = malloc (sizeof (struct s));
    /* ..... */
    return ps;
}
```

其中，结构成员 id 用于识别每一个独立的节点；next 用于指向下一个节点，通过它可将各个节点串连起来；f 是一堆业务数据，可能是成绩，也可能是国民经济的指标。

在每个节点上，f 都有 30 个元素。对于小型应用来说，数组大小不是问题。但对于大型应用来说，这就是问题所在。想想看，如果有些节点只需要 f 具有 3 个元素即可，30 个太多了，而有的节点却需要 30000 个元素，30 个又太少，这怎么办呢？

如果将 f 声明为具有 30000 个元素的数组，这足够了吧？问题在于，如果链表很长，则又过于浪费。反正是动态分配的存储空间，有些人想出了下面的办法。

```
# include <stdlib.h>

struct s {int id; struct s * next; float f [1];};

struct s * s_create (int id, int n)
{
    struct s * ps = malloc (sizeof (struct s) + n * sizeof (float));
    /* ..... */
    return ps;
}
```

其中，结构的成员 f 被声明为只有一个元素，但是可以根据实际需要，动态分配超过结构本身大小的存储空间。这样，分配的空间包括两个部分，前面是结构本身所需要的，后面是额外延伸的部分。因为数组是结构的最后一个成员，所以可以假设后面的额外部分都可以为数组所用。

这个办法很巧妙，而且通常不会出现问题，但是在 C99 之前它并不是一种安全的做法。引入弹性数组成员，只是使这种需求合法化。其实，这样的需求是合理的，因为结构或者联合的成员不能是变长数组（有些 C 实现提供了这种扩展），而将它们声明为普通数组又缺乏弹性。

弹性数组成员只能是结构的最后一个成员，所以下例中，struct a 的声明是非法的，因为弹性数组成员的前面要求至少有一个完整类型的成员；struct b 的声明是合法的；struct c 的声明是非法的，因为弹性数组成员必须是最后一个结构成员；struct d 的声明是合法的，因为尽管 f 的前一个成员是匿名结构，但它的命名成员 i 被看做 struct d 的成员。

```
struct a {double d [];};                    //非法
```

```
struct b {char c; double d [];};          //合法
struct c {char c; double d []; int i;};   //非法
struct d {struct {int i;}; float f [];};  //合法
```

以上是弹性数组成员的简单介绍，下面讲述弹性数组成员的用法。和想象中不同，弹性数组成员并不是作为一个指针来使用的，相反，为了使用弹性数组成员，典型的做法是分配一个足够大的对象，以容纳结构对象和弹性数组成员。这就是说，是在结构对象的后面为数组成员留出空间，空间的大小取决于到底分配了多少空间（根据需要分配）。

下面的例子演示了如何使用弹性数组成员。函数 s_create 的功能是创建一个 struct t 类型的对象，并用给定的长度 len 为它的弹性数组成员分配空间，最后返回一个指针，该指针指向该结构类型的对象。while 语句用于初始化数组的内容，因为表达式(*s)的类型是 struct t，所以特意使用了表达式(*s).f[len]=0.0，而不是 s->f[len]=0.0，尽管后者也是正确的。

```
# include <stdlib.h>

struct t {int n; float f [];};

struct t * s_create (int len)
{
    struct t * s = malloc (sizeof (struct t) + len * sizeof (float));
    s -> n = len;
    while ((len = len - 1) >= 0) (* s) . f [len] = 0.0;

    return s;
}
```

在上面的例子中，malloc 函数在宿主系统中分配内存空间，它返回一个 void 对象的指针，并隐式地转换为 struct t 类型的指针；成员 n 的作用是跟踪数组 f 的长度；while 语句用于给数组 f 的每个元素置初始值 0.0。

数组最后一个元素的下标是 len-1（或者 n-1），while 语句执行时，先对表达式 len=len-1 求值，该表达式的值是 len 每次被更新后的值。如果表达式的值（即 len 更新后的存储值）大于等于 0，则对下标 len 所指示的数组元素赋值。while 语句持续执行，直到表达式的值小于 0。

弹性数组成员只是被看做一个普通的数组，实际上没有任何元素。因此，除非明确地分配了足够的存储空间，否则，即使是访问它的第一个元素，或者生成首元素的后一个元素的指针，都是未定义行为。例如：

```
struct stg {int n; float f[];} s;

s.f [0] = 1.0;                  //未定义行为
float * pf = & s.f [0];         //有效。可以获得弹性数组成员第 1 个元素的地址
* pf = 5.0;                     //未定义行为

struct stg * t = malloc (sizeof (struct stg) + sizeof (float));
```

```
t ->f [0] = 2.0;                    //有效
t ->f [1] = 3.0;                    //未定义行为
```

弹性数组成员在声明时并没有指定大小。因为这种特殊性，它在计算结构类型的大小时被排除在外，就像不存在这个成员一样。但是，尽管弹性数组成员不直接为它所在的结构类型贡献大小，但有可能会间接地影响到结构类型的大小，因为它的存在可能会导致结构对象的尾部填充。

因为弹性数组成员被当做不存在，所以不要企图初始化它。

```
struct s {int i; char c [];} fs = {0, {'x'}};        //错误!
```

在编者的机器上，下例中的 struct s 大小为 1 而 struct t 的大小为 8。尽管成员 li 不占用存储空间，但它的存在导致成员 c 后面出现 7 个字节的填充。

```
struct s {char c;};
struct t {char c; long long li [];};
```

如果 S 是含有弹性数组成员的结构类型，则 S 不能是其他结构或者联合的成员类型（但可以是指向 S 的指针），也不能是数组的元素类型。这个限制是弹性数组成员在应用上的特点造成的。因为这个原因，下例中的两个声明 D1 和 D2 都为非法。

```
struct t {char c; long long li [];};
struct s {struct t t;};      //D1
struct t at [3];             //D2
```

每个结构和数组元素都具有固定的大小，弹性数组成员的使用需要比结构更大的对象空间，这个特点使得它不能成为其他结构的成员或者数组元素。因此，如果给定以下结构类型的声明：

```
struct t {int i; char c [];};
```

那么下面的声明是不允许的。试想一下，对象 s 的成员是 t，t 也是结构对象，含有弹性数组成员 c，你如何为成员 c 分配存储空间？就算是有办法分配，在它的后面还有 s 的成员对象 j，对象 c 会挤占和覆盖对象 j 的存储空间。同样地，对象 at 也有着同样的处境。

```
struct s {struct t t; int j;} s;        //不允许
struct t at [2];                        //不允许
```

但是，数组的元素或者结构的成员可以是指向“包含弹性数组成员的结构”的指针。

```
struct s {struct t * pt; int j;} s;     //可以
struct t * pa [2];                      //可以
```

尽管结构类型的对象之间可以相互赋值（前提是它们属于兼容的结构类型），但是结构对象之间的赋值操作不复制弹性结构成员的内容。下面的例子解释了这一点。

```
# include <stdlib.h>

void f (void)
{
   struct t {int n; float f [];} * s1, * s2;

   s1 = malloc (sizeof (struct t) + 2 * sizeof (float));
```

```
    s2 = malloc (sizeof (struct t) + 2 * sizeof (float));
    /* ..... */
    * s2 = * s1;      //仅复制成员 n 的值
    /* ..... */
}
```

3.8.2.3 匿名结构

如果一个结构或者联合包含了这样的成员：

① 没有名称；

② 它被声明为结构类型，但是只有成员列表而没有标记。

则这个成员就是一个匿名结构（anonymous structure）。

在下例中，struct t 和 union u 的第二个成员都是匿名结构。

```
struct t {int i; struct {char c; float f;};};
union u {int i; struct {char c; float f;};};
```

现在的问题是，如何才能访问匿名结构的成员？若某个匿名结构 S 是结构或者联合 X 的成员，那么 S 的成员被认为是 X 的成员。进一步地，对于多层嵌套的情况，如果符合以上条件，则可以递归地应用这种关系。

在下面的例子中，struct t 包含了一个没有标记、没有名称的结构成员，这个结构成员的成员 c 和 f 被认为属于 struct t。

```
struct t
{
    int i;
    struct s {int j, k:3;};          //有标记的成员
    struct {char c; float f;};       //无标记，且未命名的成员
    struct {double d;} s;            //命名的成员
} t;

t.i = 2006;
t.j = 5;                             //非法
t.k = 6;                             //非法
t.c = 'x';                           //正确
t.f = 2.0;                           //正确
t.s.d = 22.2;
```

因为同样的原因，下面的类型声明将在转换期间得到一个表示错误的诊断信息，因为 struct tag 的第二个成员是匿名结构，而匿名结构的成员中又有一个是匿名结构，所以，匿名结构的成员 i 和 f 被认为是 struct tag 的成员，这意味着 struct tag 有两个成员的名称相同，都是 i。

```
struct tag
{
    struct {int i;};
```

```
    struct {struct {int i; float f;}; double d;};
    char c;
};
```

尽管匿名结构的成员被认为是隶属于包含该结构的上层结构的成员，但它的初始化器依然必须采用被花括号包围的形式，具体参见“初始化器”。

在下例中，尽管匿名结构的成员 x 被认为是属于包含它的那个结构 struct t，但它的初始化器仍然需要使用一对花括号。

```
struct t {char c; struct {int x;};};
struct t t = {'x', 1};      //非法
struct t t = {'x', {1}};    //合法
```

3.8.3 联合

联合（union）是一种派生类型，用于描述这样一种对象：它由一些在存储空间上互相重叠的子对象（成员对象）组成。和结构类型一样，不要求联合的成员都属于同一类型，它们的类型可能相同，也可能不同，但是无论如何，所有成员共享部分或者全部存储空间。联合和结构在声明时，具有相同的语法结构，但联合类型的声明是以关键字 union（而不是 struct）开始的，例如：

```
union {char c; int i;} u;
```

联合对象的存储布局和结构对象全然不同。以上，成员 c 和成员 i 尽管具有不同的类型，但却共享整个联合对象 u 开始的一部分存储空间。

有关联合类型的声明方法可参见“结构或联合指定符”。在上述声明中，

```
union {char c; int i;}
```

是类型指定符的一种，用于指定一种联合类型，所以又称结构或联合指定符；花括号内部是结构声明列表，即成员列表。

根据上述定义，联合的成员列表不能是空的，至少要有一个成员。当然，尽管标准是这样要求的，但是如下例所示，很多 C 实现对此加以扩展，并容许没有成员的联合。如果程序对可移植性有要求，则应避免使用这种依赖实现的特性。

```
union {} u;          //非法，成员数量不能为零
```

联合的成员在数量上是有所限制的，具体的数量由 C 实现自己决定，但标准要求的最低限度是保证不少于 1023 个。

联合的成员必须属于完整的对象类型，而且不能是变长数组。特别地，联合的成员可以是位字段。有关位字段成员的详细描述可参见“位字段”。

下面给出了一个不合法的联合声明，因为它的第二个成员是变长数组。

```
void f (int n)
{
    union u {char c; int a [n];};
    /* ..... */
}
```

在本书中，将混用“联合”和“联合类型”。有时这两个概念用于泛指各种不同的联合类

型，即联合类型的总称；而有时这两个概念指代某个具体的联合类型。具体是什么意思，取决于它们所在的上下文语境。

为了访问联合的成员，可以使用后缀运算符“.”或者“->”。在使用运算符“.”访问联合对象的成员时，该运算符的左操作数是联合类型的表达式，右操作数是联合对象的成员名；在使用运算符“->”访问联合对象的成员时，该运算符的左操作数是指向联合对象的指针，右操作数是被指向的联合对象的成员名称。有关成员访问的详细描述，可参见“后缀表达式”和“成员选择”。例如：

```
union u {char c; float f;} u, * pu = & u;
u.c = 'x';
pu -> f = 0.01;
```

尽管联合可以有多个成员，但实际上，由于它们共用联合对象的起始部分，在任何时候都只能保存一个成员的值。此外，读取一个联合成员的值时，要确保最近一次对联合对象的写入操作也是通过这个成员进行的，否则后果难以预料。

很容易想到的是，联合的大小必须足以容纳最大的那个成员。

因此，就前例中声明的对象 u 来说，经过以下赋值操作后，联合对象最终保存的是成员 f 的值 19.00，而不是字符 x 的编码值。

```
u.c = 'x';
u.f = 19.00;
```

也就是说，引用联合对象成员的前提是联合的上一次赋值是通过该成员进行的。引入联合类型的本意是，当多个对象不需要同时存在且都很大时，用联合来组织它们可以节省存储空间。

在下例中，先将整数类型的值'x'写入了联合对象的成员 c，然后又读取了成员 f 的值，这种行为会造成不可预知的后果。

```
union u {char c; float f;} u;
void f (float);
u.c = 'x';
f (u.f);
```

和结构类型一样，联合的非位字段成员也有自己的对齐要求。但由于联合的特殊性（所有成员对象是部分或者全部重叠的），要由 C 的实现权衡每个成员的类型，由此来决定它们的对齐。另外，和结构一样，整个联合对象的尾部可能存在填充，但是它的开头不会有任何填充。

同样很容易理解的是，联合类型的长度不小于最长的那个成员的长度。另外，因为对齐的需要，联合类型可能会有尾部填充，在这种情况下，联合类型的长度会大于最长的那个成员的长度。例如：

```
# include <stdio.h>

void f (void)
{
```

```
    union u {int i; char c [21];};
    printf ("%zu, %zu\n", sizeof (union u), _Alignof (union u));
}
```

在编者的机器上，联合类型 union u 的大小是 24 个字节，且对齐于 4，如图 3-4 所示，该联合类型将需要 3 个字节的尾部填充。

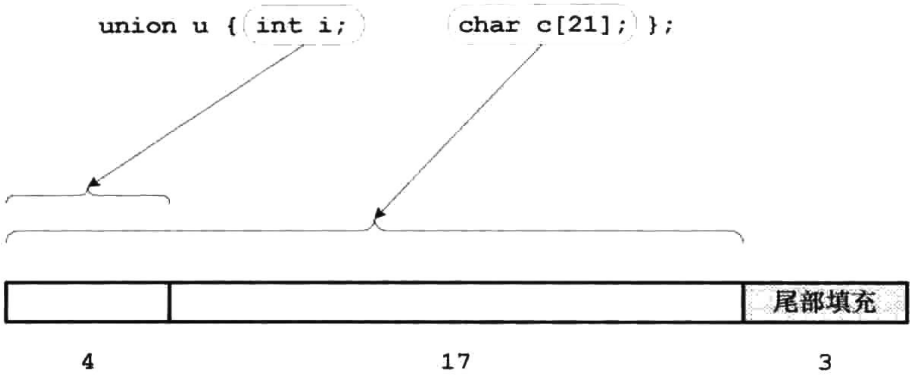


图 3-4 联合对象的成员布局和填充

尾部填充可以确保在连续分配同类型的多个联合对象时，这些对象及它们内部的每个成员都能正确对齐。特别地，对于结构或者联合类型的数组来说，按照定义，数组元素必须是连续分配的；此外，每个数组元素（结构或者联合）的成员必须正确对齐，这就使得结构或者联合类型在必要的时候，必须具有尾部填充。

如果两个以上的结构在成员上具有这样的特点：它们的第一个成员在类型上是兼容的，或者都是类型兼容且宽度相同的位字段。按照这个方法，再依次比较它们的其他成员，直至到达某个结构声明的末尾（遇到“}”），或者遇到成员类型不兼容或者位字段宽度不同的情况。此时，我们说这些结构的成员具有相同的开始部分。

进一步地，如果这些结构都是同一个联合的成员，那么，这些相同的开始部分是公共的初始成员，将共享同一片存储空间，或者说共享一个公共的初始序列。

再进一步，如果最近一次向该联合对象写入时，是通过这几个结构中的任何一个进行的。

那么，可以使用任何一个结构来访问这些公共初始成员，但前提是，在访问这些成员的时候，可以看见该联合的完整声明（因为这可以使 C 实现用一致的方法和策略决定每个结构初始成员间的填充，以及是否合并相邻位字段）。

下面的例子是有效的，联合类型 union u 的成员 s 和成员 t 都是结构类型，且它们对应的初始成员在类型上是兼容的（int 和 signed 是兼容类型）。

```
union u
{
    struct {int x;} s;
    float f;
    struct {signed y; char c;} t;
    struct {char c; int i;} m;
};
```

```
void f (void)
{
    union u u;
    u.t.y = 22;
    u.t.c = 'x';                    //联合u的存储值来自结构对象t
    if (u.s.x > 0) {/* …… */}   //使用另一个结构对象s访问联合的值
}
```

和前面的例子相比，下面的代码是无效的。其原因并不复杂，仅仅是因为联合的完整声明在函数f内并不可见。当C实现处理到这里时，它有可能为结构 struct s 和 struct t 选择不同的位字段合并方案，这就无法保证用不同的结构访问到相同的内容。

```
struct s {int i : 15; int j : 3; int k : 5;};
struct t {int x : 15; int y : 3; int z : 12;};

int f (struct t * pt)
{
    pt->y ++;
    /* …… */
    return pt->y;
}

int g (void)
{
    union {struct s s; struct t t; double d;} u;
    u.s.i = 0;
    u.s.j = 1;
    /* …… */
    return f (& u.t);
}
```

联合的成员可以是另一个联合，或者是另一个结构，这种关系可以递归地构建，这并不奇怪。但如果联合的成员是匿名结构或者匿名联合，则成员的隶属关系会有所不同，具体的描述可参见“匿名结构”和“匿名联合”。

在任何一个联合类型U的声明中，用于终止成员列表的“}”出现之前，U尚属于不完整类型。在此之后，U成为完整类型。以下的声明是非法的。因为在声明成员 u 的时候，union u是不完整的，而成员u的声明需要知道union u的完整信息（大小）。

```
union u {int i; union u u;};
```

相反，以下声明是合法的。尽管在声明0成员 pu 的时候，union u 是不完整类型，但声明一个指针并不需要有关该类型的具体信息。

```
union u {int i; union u * pu;};
```

以下的声明也是合法的。在声明uv和uw的时候，union u已经是完整类型。

```
union u {int i; union u * pt;} uv;
union u uw;
```

下面的声明是非法的，在 union u 成为完整类型之前，企图用 sizeof 运算符得到它的大小是不现实的，只能导致错误。

```
union u {char c; int a [sizeof (union u)];};
```

但是，一旦某个联合类型成为完整类型，则马上可以使用，即

```
union u {char c; int i;} u = {sizeof (union u)};
```

前面说过，在联合对象的开头不存在填充，所以联合对象在存储器中的起始位置（地址）也是其任何一个成员对象的起始位置（地址）。指向联合的指针和指向该联合任何一个成员的指针之间可以互相转换，参见“指针-指针转换”。

联合类型是对象类型，可以将一个联合对象的值赋给另一个联合对象，前提是它们具有相同的类型。

3.8.3.1 匿名联合

如果一个结构或者联合包含了这样的成员：

① 没有名称；

② 它被声明为联合类型，但是只有成员列表而没有标记。

则这个成员就是一个匿名联合（anonymous union）。在下例中，struct t 和 union u 的第二个成员都是匿名联合。

```
struct t {int i; union {char c; float f;};};
union u {int i; union {char c; float f;};};
```

现在的问题是，如何才能访问匿名联合的成员？答案如下：若某个匿名联合 U 是结构或者联合 X 的成员，则 U 的成员被认为是 X 的成员。进一步地，对于多层嵌套的情况，如果符合以上条件，那么可以递归地应用这种关系。

在下面的例子中，struct t 包含了一个没有标记、没有名字的联合成员，这个联合的成员 c 和 f 被认为属于 struct t。

```
struct t
{
    int i;
    struct s {int j, k:3;};          //有标记的成员
    union {char c; float f;};        //无标记，且未命名的成员
    struct {double d;} s;            //命名的成员
} t;

t.i = 2006;
t.j = 5;                             //非法
t.k = 6;                             //非法
t.c = 'x';                           //正确
t.f = 2.0;                           //正确
```

```
t.s.d = 22.2;
```

因为同样的原因，下面的类型声明将在转换期间得到一个表示错误的诊断信息，因为 struct tag 的第二个成员是匿名联合，而匿名联合的成员中又有一个是匿名联合，所以，匿名联合的成员 i 和 f 被认为是 struct tag 的成员，这意味着 struct tag 有两个成员的名称相同，都是 i。

```
struct tag
{
    struct {int i;};
    union {union {int i; float f;}; double d;};
    char c;
};
```

3.8.3.2 标记

在同一个转换单元中，可能需要多次引用同一个结构、联合或者枚举类型。要达到这个目的，需要使用标记来指代特定的结构、联合或者枚举类型。例如，在下面的代码中，A 处声明了完整的联合类型 union u，B 处和 C 处通过一个标记 u 来使用刚才声明过的那种联合类型。

```
union u {char c; double d;};              //A
union u a [sizeof (union u) * 10];        //B
union u u = {'x'};                        //C
```

大家知道，如果没有标记，则两个完全一样的结构或联合指定符代表两个截然不同的结构或者联合类型。例如，在下面的代码中，x 和 y 的类型是不同的，它们也都和函数 f 的返回类型不同，尽管在这三个声明中类型指定符看起来完全一样。

```
union {char c; double d;} x;
union {char c; double d;} f (void)
{
    union {char c; double d;} y;
    /* ..... */
}
```

3.8.4 指针类型

指针在 C 语言里具有重要的地位，它为动态存储器管理、数据操纵和硬件访问提供了强有力的手段。

指针类型（pointer type）是派生类型的一种，它派生自对象类型（在这种情况下它是指向对象的指针）或者函数类型（在这种情况下它是指向函数的指针）。和数组、函数类型一样，指针类型也是在声明符一侧派生的，用于声明一个指针类型的声明符。以其最简单的形式来说，指针的声明符是一个“*”跟着一个标识符，例如：

```
int * p;    //声明一个指向 int 的指针 p，或者说，p 的类型是 int *
```

以上，声明指定符 int 在这里是类型指定符；声明符* p 标明 p 是一个指针。如下例所示，

星号可以单独存在，也可以和类型指定符或者标识符结合，实际上没有什么区别。

```
int* p;
int *p;
```

尽管如此，C 的这种自由性说明不了任何问题。就指针声明的语法而言，星号是声明符的一部分，而不是声明指定符的一部分。因此，要声明两个指针，需要两个声明符，同时很自然地，每个声明符都包含一个星号。

```
double * pm, * pn;  //pm 和 pn 都是指向 double 类型的对象的指针
double * px, y;     //px 是指针，y 不是
```

有关声明一个指针的方法，可参见“声明符”。

指针是以它所指向的类型为特征的，不同的指针类型之所以有区别，是因为它们指向的类型不一样。如下例所示，pi 和 pf 是不同的指针类型，pi 是指向 int 的指针，而 pf 是指向函数的指针。

```
int * pi, (* pf) (void);
```

可以声明指向不完整对象类型的指针，但不完整的枚举类型和元素类型不完整的数组除外。在下面的例子中，pi 是一个指向数组的指针，数组元素的类型是 int。尽管它所指向的数组是不完整的，但元素类型是完整的对象类型 int；pif 也是指向数组的指针，但数组的元素类型不完整（是 float[]），所以它的声明非法（即使是数组的声明，也不允许元素类型是不完整的）；pv 是指向 void 的指针，ppv 是指向指针（void *）的指针；pe 是指向枚举类型 enum e 的指针，但该枚举类型是不完整的，所以 pe 的声明非法（注意，任何时候都不允许提前引用一个不完整的枚举类型）；pt 是指向不完整结构类型的指针；pat 是指向数组的指针，但数组的元素类型是不完整的结构，故 pat 的声明非法；pc 是指向数组的指针，数组的元素类型是 char。顺便说一句，因为 c 的类型是数组（char [100]），故表达式&c 的类型是指向数组的指针（char (*) [100]）。

```
# include <stdlib.h>

void f (void)
{
    /* 假定在此处看不到 enum e 和 struct t 的其他声明 */
    int (* pi) [] = (int (*) []) malloc (100);  //合法
    float (* pif) [3] [];                       //非法
    void * pv, * * ppv;                         //合法
    enum e * pe;                                //非法
    struct t * pt, (* pat) [2];                 //pt 合法，pat 非法
    char c [100];
    char (* pc) [] = & c;                       //合法
}
```

指针是对象类型，任何合法的指针都是完整的对象类型。这是因为，指针本身也是对象，有自己的类型，也有值；声明一个指针的前提是要给出它所指向的类型，而一旦明确了所指向的类型，指针的大小也就确定了，即不存在不完整的指针类型。

指针类型也是对象类型，指针类型的对象，它的值被视为一个线索，可以根据这个线索间接访问该指针所指向的对象，或者调用该指针所指向的函数。因为这个原因，指针类型被称为“引用类型”，而指针所指向的类型（对象或者函数）称为“被引用类型”。指针的声明、指针对象和指针所指向的对象这三者之间的关系如图 3-5 所示。

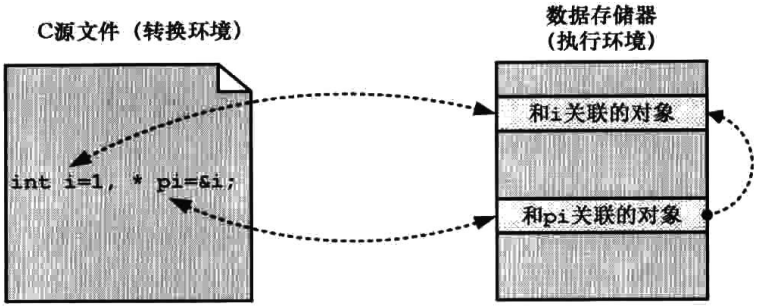


图 3-5　指针示意图

在本书中，“指针”一词具有多种含义。有时候，所谓的“指针”是泛指各种不同的指针类型，即指针类型的总称；而在另一些时候，这个概念特指某个具体的指针类型或者指针对象。具体是什么意思，取决于它所在的上下文语境。

尤其需要注意的是，在有些时候我们提到的“指针”是指某个指针类型的对象，而在另一些时候则是指“指针类型的值”。比如我们说“在某些情况下，数组类型的表达式会被转换为指针”，在这里，数组类型的表达式并非被转换为指针类型的对象，而是被转换为指针类型的值。下面是一个例子：

```
int a [3], p;
a;              //S1
p = a;          //S2
```

在上面的 S1 中，我们会说“数组 a 被转换为指针”，但实际上完整的表述是“数组 a 被转换为指针类型的值”，且这个值被丢弃；与此不同，在 S2 中，这个值被赋给指针类型的对象 p。

指针类型的值经常被很多人理解为“地址”，这当然有助于形象地理解指针到底是什么，它有什么用，但是，对于和它有关的赋值、参数传递和转换操作来说，比较科学的方法就是借助于类型系统，将它理解为一个值，一个具有指针类型的值，而不要使用地址。比如，赋值运算符两边的表达式就需要兼容或者相同的类型（可能需要适当的转换），你可以说将一个指针类型的值赋给一个指针对象，而不能说将一个地址赋给指针对象，这很不合理。

上面说了，“指针”可能是指“指针类型”，也可能是指“指针类型的对象”，还可能是指“指针类型的值”。进一步地，只能把一个指针类型的值赋给一个指针类型的对象，但前提是这个值和那个对象具有相同的（指针）类型。来看一个例子：

```
# include <stdio.h>

void fdemo (void)
{
    int i = 1, * pi, (* pf) (const char * restrict, ...);
    pi = 'x';           //S1
```

```
    pi = & i;           //S2
    pf = & printf;      //S3
}
```

其中，表达式'x'的类型是 int，不是指针，但对象 pi 要求一个指针类型的值，故 S1 非法；如果一个对象或者函数的类型是 T，则&T 的结果类型是“指向 T 的指针”，这个值可以赋给指针对象。对象 i 的类型是 int，&i 的结果类型是指向 int 的指针 int *，与 pi 的类型匹配，可以赋值，故 S2 合法；&printf 是指向函数的指针，与 pf 的类型匹配，故 S3 也合法。

一元运算符*指示间接操作，如果 P 的类型是“指向 T 的指针”，则表达式*P 的类型是 T，可参见“一元表达式”中的“间接（*）”。下面是几个实例。

```
int i, j, k, * p;

p = & i;        //p 和&i 的类型都是“指向 int 的指针”
* p = 2;        //对象 i 的当前值为 2。表达式*p 指示对象 i

p = & j;
* p = 33;       //对象 j 的当前值是 33。表达式*p 指示对象 j

p = & k;
* p = i * j;    //对象 k 的当前值是 66。表达式*p 指示对象 k
```

指针可以派生自结构或联合类型。声明指向结构或者联合类型的指针，和目前为止的其他指针声明一样非常直观。注意，通过结构或者联合类型的指针引用成员，需要使用后缀运算符->，该运算符的左操作数要求是一个指向结构或联合对象的指针，右操作数要求是成员的名称，具体可参见“成员选择”。下面是几个实例。

```
struct t {int i; float f;} * pv, t;
struct t * pw = & t;        //pw 和 pv 的类型相同，都是 struct t *
pw -> i = 0;
pw -> f = 0.0;
pv = pw;                    //此后，pv 和 pw 指向同一结构类型的对象
```

指针可以派生自另一个指针类型，例如：

```
int * * p;
```

那么，p 是一个指向“指向 int 的指针”的指针，或者说是“指向 int *的指针”，又或者说是“指向指针的指针”。下面的例子简单地演示了如何使用这种类型的指针。

```
int i, * pi, * * ppi;
pi = & i;       //有效。pi 的类型是 int *，表达式&i 的类型也是 int *
ppi = & i;      //无效。表达式&i 的类型是 int *，而 ppi 的类型是 int **
ppi = & pi;     //有效。ppi 的类型是 int **，表达式&pi 的类型也是 int **
* * ppi = 1972; //等效于*(*ppi) = 1972
```

特别地，对于上述最后一条语句，因为 ppi 指向对象 pi，而 pi 又指向对象 i，所以，表达式*ppi 指示对象 pi，表达式**ppi 可以看成是*pi。换句话说，表达式**ppi 指示对象 i，这最

后一条语句实际上访问的是对象 i。

再来看另一个声明：

```
int * p [3];
```

p 首先是一个具有 3 个元素的数组，数组元素的类型是指向 int 的指针。换句话说，这是一个“指向 int 的指针”的数组。相反地，要想声明一个指向数组的指针，则必须用()阻止标识符和[]的结合，即：

```
int (* p) [3];
```

因为如果没有括号，标识符应当先和“[3]”结合，然后才是“*”。现在因为有括号的存在，p 首先是一个指针，它指向一个 3 元素数组，数组元素的类型是 int。

使用同样的方法，可以声明指向函数的指针，以及返回指针的函数。在下面的例子中，fp 首先是一个函数，该函数接收两个 int 类型的参数，并返回一个指针，指向 struct t。换句话说，fp 是返回 “指向 struct t 的指针”的函数。

相反，pf 首先是一个指针，它指向一个函数，该函数接收两个 int 类型的参数，并返回一个 struct t 类型的值。

```
struct t * fp (int, int);
struct t (* pf) (int, int);
```

接下来，声明一个指向数组的指针，该数组具有 2 个元素，元素的类型是指向函数的指针。

```
int (* (* ptf) [2]) (int, int);
```

其中，ptf 首先是一个指针，指向一个 2 元素数组；又因为括号的关系，数组的元素不是函数，而是另一个指针，指向一个函数，该函数具有两个 int 类型的参数，并返回 int 类型的值。

以下声明一个数组，该数组具有 2 个元素，元素的类型是指针，指向返回函数指针的函数。

```
void (* (* aftf [2]) (void)) (int, int);
```

首先，aftf 是一个 2 元素数组，并且因为括号的关系，数组元素的类型是指针，指向一个函数，该函数不接收参数，并且返回一个指向函数的指针，这个函数接收两个 int 类型的参数，并且不返回任何值。简单地说，aftf 是元素类型为“指向‘返回函数指针的函数’的指针”的数组。

为了理解这种复杂的声明，下面是一个更好的示例：

```
# include <stdio.h>

void f (int i, int j)
{
   printf ("%d, %d\n", i, j);
}

void (* fretf (void)) (int, int)
{
```

```
    return & f;                                          //S1
}

int main (void)
{
    void (* (* aftf [2]) (void)) (int, int);             //D1

    aftf [0] = & fretf;                                  //S2
    aftf [1] = & fretf;                                  //S3

    int i = 0;
    while (i < 2)
    {
        void (* fp) (int, int) = (* aftf [i]) ();        //D2

        (* fp) (i, i);                                   //S4
        i ++;
    }

    return 0;
}
```

首先，函数 f 接收两个 int 类型的参数，但不返回任何值；从函数 fretf 的定义来看，它是一个没有参数的函数，且返回类型为 void (*) (int, int)。也就是说，返回一个指向函数的指针，那个被指向的函数接收两个 int 类型的参数，但并不返回任何值。这里，语句 S1 中的表达式 &f 其类型是指向函数的指针 void (*) (int, int)。

其次，在 main 函数内部，D1 声明了一个数组 aftf，它具有两个元素，元素的类型是指针，指向一个没有参数，且返回指针的函数，这个返回的指针指向接收两个 int 类型参数且无不返回任何值的函数。

再次，语句 S2 和 S3 给数组 aftf 的每个元素赋值。因为数组 aftf 的元素类型为 void (* (*) (void)) (int, int)，表达式&fretf 的类型也是 void (* (*) (void)) (int, int)，它们类型相同，所以将后者的值赋给前者是允许的。

指向 void 的指针（void *）经常被称为“通用对象指针”，指向 void 的指针可以转换为指向任何对象类型的指针；反之，指向任何对象类型的指针都可以转换为指向 void 的指针。

第四，在 D2 中，将 fp 声明为指向函数的指针，被指向的函数具有两个 int 类型的参数，且返回 void 类型。在它的初始化器中，表达式* aftf[i]等效于*(aftf[i])，因为表达式 aftf[i]的类型是指针 void (* (*) (void)) (int, int)，实际上指向函数 fretf，所以表达式*aftf[i]求值的结果是函数指示符 fretf。接着，表达式(*aftf[i])()是函数调用，求值的结果（本次函数调用的返回值）是指向函数 f 的指针。

最后，因为刚才表达式(*aftf[i])()求值的结果是指向函数 f 的指针，并赋给了指针对象 pf，

所以在语句 S4 中，表达式*fp 求值的结果是函数指示符 f，同时执行函数调用。

下面的例子用于演示指向数组的指针，数组的元素类型是“指向函数的指针”。这个例子先是声明和初始化这个指针，然后以不同的形式使用这个指针：

```
# include <stdio.h>

void f (void) {printf ("Pride and prejudice.\n");}

void g (void)
{
    void (* (( * pfa) [1])) (void) = & (void (* [1]) (void)) {f};
    (* pfa) [0] ();              //S1
    (* (* pfa)) ();              //S2
    (* ((* pfa) [0])) ();        //S3
}
```

首先，pfa 是一个指向数组的指针，被指向的数组有 1 个元素，元素的类型是指向函数的指针，被指向的函数不接受参数，且返回类型是 void。为了初始化这个指针，我们在它的初始化器里使用了复合字面值（参见“复合字面值”）：

```
(void (* [1]) (void)) {f}
```

这将创建一个不可见的、具有 1 个元素的数组，且数组的元素类型是指向函数的指针，并用函数 f 的地址初始化这个数组对象。最后，一元&运算符得到这个数组的地址，并用于初始化对象 pfa。

接下来，在 S1 中，表达式*pfa 的结果是 pfa 所指向的数组；表达式(*pfa)[0]的结果是数组的第一个元素，也就是指向函数的指针。最后，运算符()调用这个函数。

S2 和 S1 尽管形式上不同，但本质上是一样的。表达式*pfa 的结果是 pfa 所指向的数组，且被自动转换为指向其首元素的指针，所以表达式*(*pfa)的结果是数组的第一个元素，也就是指向函数的指针。最后，运算符()调用这个函数。

S3 和 S2 完全一样，只不过多了一次解除引用的操作。传统版本的 C 要求对函数指针显式地解引用，但新标准同样支持。但对标准 C 语言来说，这并不是必需的，详情参见“函数调用”。

3.8.4.1 空指针常量

所谓空指针常量（null pointer constant），是指值为 0 的整型常量表达式，或者是将该表达式的值转换为 void *（指向 void 的指针）类型后得到的新值。

经常地，人们使用 NULL 来代表空指针常量，它是一个宏，在头文件<stddef.h>中定义：

```
# define NULL (void *) 0
```

空指针常量经常用于初始化指针对象，给指针对象赋值，或者参与指针的比较运算，等等。下面是几个例子（粗体的部分是空指针常量）。

```
#include <stddef.h>
int * p = 'x' - 'x', * q = 0, * r = NULL, * v = '\0';
```

值得说明的是，指针 p 的初始化器是'x' - 'x'，因为'x'是整型常量，所以表达式'x' - 'x'的结果也是整型常量 0，即空指针常量；\0 是空字符常量，而'\0'是整型常量表达式，也是空指针常量。

相反的，下面用粗体印刷的表达式都不是空指针常量（但它们属于空指针，可参见“空指针”）：

```
int * p = (int *) 0, * q = (void *) NULL;
```

C 实现不保证在表示整型常量 0 的值时，所有比特都是 0（参见“对象表示”）。因此，空指针常量的值也不保证其所有的比特都是 0。当然，这对程序员来说是透明的，即使他们不了解这一点也不会有问题。

如果一个指针的值是空指针常量，则它不指向任何有效的对象或者函数。换句话说，任何一个指针，如果它指向一个（有效的）对象或者函数，则它的值都不会是 0。如果 p 是空指针常量，指针 q 指向一个有效的对象或者函数，则 p 和 q 在比较时是不相等的。但是，任何两个空指针常量在比较时都是相等的。看下面的例子，因为指针 p 的值是空指针常量，所以对它的解除引用操作将产生未定义行为。

```
float * p = 0;
* p = .01f;      //未定义行为!
```

3.8.4.2 空指针

空指针常量（参见“空指针常量”）可以被转换为任何其他指针类型，转换后得到的结果称为那种类型的空指针（null pointer）。下面是两个空指针的例子，其中 pi 是指向 int 类型的指针，它的值是空指针；pf 是指向返回 int 类型的函数的指针，它的值也是空指针。

```
# include <stddef.h>
int * pi = NULL, (* pf) (void) = NULL;
```

上述的类型转换是自动进行的，也可以显式地将空指针常量转换为空指针，例如：

```
float * fp = (float *) 0LL;
```

这里，0LL 是空指针常量，而表达式(float *) 0LL 的结果是一个空指针。

和空指针常量一样，空指针不指向任何有效的对象或者函数。换句话说，任何一个指针，如果它指向一个（有效的）对象或者函数，则它的值都不会是 0。如果指针 p 的值是空指针，指针 q 指向一个有效的对象或者函数，则 p 和 q 在比较时是不相等的。但是，任何两个空指针在比较时都是相等的。看下面的例子，如果 p 和 q 都是空指针，则代码中被加粗印刷的表达式在求值时将产生未定义行为。如果去掉这个表达式，函数 f 将返回非 0 值，因为两个空指针在比较时是相等的。

```
# include <stdbool.h>

bool f (int * p, float * q)
{
   * p = * q = 0;
   return (void *) p == (void *) q;
}
```

下面是另外一个例子，空指针常量 0 被转换为 int 类型的指针，转换之后是一个空指针。同时，指针 p 指向一个对象（n），所以 p==q 是不成立的。

```
int n = 1, * p = & n, * q = 0;
if (p == q) {/* ····· */}
```

不同类型的空指针可能具有不同的对象表示。空指针的对象表示不要求所有比特都是 0，因为这和执行环境的地址结构有关。但这对程序员来说是透明的，即使他们不了解这一点也不会有问题。

3.8.5 函数

函数是一种组织程序的手段，可以用函数将重复使用的代码组织在一起，形成代码块以方便重复使用（调用）。函数可以接收从调用者那里传入的参数，也可以（向调用者）返回一个值。

用于描述函数的那些类型称为函数类型（function types）。函数类型的特点是它有返回类型，也有参数类型，还包括参数的数量，这些都是一个函数有别于另一个函数的特征，也是一种函数类型区别于另一种函数类型的特征。如果两个函数具有不同的返回类型，它们就是不同的函数类型；如果两个函数具有不同的参数数量或者参数类型，它们也是不同的函数类型。例如，以下是 3 个不同的函数类型。

```
void f (int);
int g (void);
void h (const char *, ...);
```

在调用一个函数之前，应当先声明它。声明一个函数，就是要说明它的名称、返回类型、参数的数量和每个参数的类型，可能还包括一个函数体。函数体包含了声明和语句，函数的功能由函数体来完成。如果函数的声明带有函数体，则这个函数声明也是函数定义。

有关函数声明和函数定义的内容，可参见“函数声明”和“函数定义”、“原型”；有关函数调用的内容，可参见“函数调用”。下面是一个有关函数声明和函数定义的例子。

```
# include <stdio.h>

int fdemo (void);                          //D1

int main (void)                            //D2
{
    return printf ("%d\n", fdemo ());
}

int fdemo (void)                           //D3
{
    int x = 0, sum = 0;
    while (x ++ < 100)
        if (x % 2) sum += x;
```

```
    return sum;
}
```

其中，在 main 函数内调用 fdemo 前要先声明，这是 D1 的作用。函数在调用之前也必须定义，这是 D3 的作用。函数 printf 的声明在头文件<stdio.h>中，当前源文件在经过预处理后，即可得到它的声明，不需要用户声明。D2 是 main 函数的定义。main 函数比较特殊，它由 C 实现调用。因为函数定义是带有函数体的声明，所以如果将 D3 放在 D1 的位置，同时将原来的 D1 去掉，也是可以的。

截止到目前，经过标准化之后的 C 语言依然支持传统（旧）风格的函数声明和函数定义，但提倡使用函数原型，可参见“函数声明”和“函数原型”以了解相关的情况。

声明一个函数时，它的返回类型不能是数组，也不能是函数，但可以是指向数组或者函数的指针类型。例如：

```
int f (int) [];              //错误，返回一个数组
int (* f (int)) [];          //允许，返回指向数组的指针
int g (int) (void);          //错误，返回一个函数
int (* g (int)) (void);      //允许，返回一个指向函数的指针
```

C 语言的函数不接收数组类型和函数类型的参数，也无法向一个函数传递这样的参数。但是，在声明函数的参数时，可以指定数组或者函数类型，但这并不意味着真的可以接收这样的参数。如果函数的参数以数组的形式出现，则被调整为指向数组首元素的指针；如果函数的参数以函数的形式出现，则它同样不意味着可以接收函数类型的参数，而是被调整为指向函数的指针。

注意，上述调整不是递归的，而是只进行一次。

先来看第一个示例：

```
void demo (int a [3], int f (void))
{
    int b [3];
    a ++;           //E1
    b ++;           //E2
    f ();           //E3
    (* f) ();       //E4
}
```

以上，函数 demo 接受两个参数，第一个参数名义上是数组，元素类型是 int，但它被调整为指向数组首元素的指针，该参数等价于

```
int * a
```

第二个参数名义上是函数，但它被调整为指向函数的指针，该参数等价于

```
int (* f) (void)
```

在这种情况下，E1 是正确的，函数的参数被调整为指针，a 的真实“身份”是一个指针类型的对象，它的值是调用者传递的指针值；E2 是错误的，运算符++只能用于左值，b 的类型是数组，在这里被转换为指向其第一个元素的指针，且不再是一个左值；E3 是正确的调用，而 E4 也同样是正确的调用。在标准 C 语言中，E3 和 E4 这两种形式实际上是一样的。要了解这其中的原因，请参见“函数调用”。

下面是另一个示例，函数 f 的参数是以两维数组的形式出现，实际上被调整为指向数组首元素的指针。因该数组的元素类型是 int [3]，故调整后的类型是 int (*) [3]，参数的调整不是递归的，故这就是 a 的实际类型。

来看表达式 sizeof * a，因为 a 的类型是 int (*) [3]，所以，表达式*a 的类型是数组，即 int [3]，表达式 sizeof * a 的结果是数组的大小。

```
# include <stdio.h>

unsigned f (int a [2] [3])
{
    return sizeof * a;
}

void g (void)
{
    printf ("%d\n", f ((int [] [3]) {{1, 2, 3}, {4, 5, 6}}));
}
```

在这个示例中，因为参数的调用不是递归的，所以函数 f 的定义实际上等价于

```
unsigned f (int (* a) [3])
{
    return sizeof * a;
}
```

在 C 中，函数的声明可能很复杂，下面是一个典型的例子（但还不是一个特别复杂的例子）：

```
float (* (* f (float (* (float *)) [5])) (float, float)) [5];
```

实际上，这里声明了一个函数 f，但它的参数类型是什么，返回类型又是什么呢？大家可以自己思考这些问题。提示：函数 f 的声明过程可分解为以下几个步骤：

```
typedef float (* A) [5];
typedef A F1 (float, float);
typedef A F2 (float *);
F1 * f (F2);
```

函数类型属于建立在其他类型之上的派生类型，而且被认为派生自它的返回类型。正是因为这个派生关系，如果返回类型是 T，则函数类型经常被称为“返回 T 的函数”。

在本书中，将混用“函数”和“函数类型”。有时候，这两个术语用于泛指各种不同的函数类型，即函数类型的总称；而有时候，这两个概念指代某个具体的函数类型或者函数。具体是什么意思，取决于它们所在的上下文语境。

3.8.6 原子类型

先考虑一个场景：在多线程环境中，两个线程共享同一个对象，这样会发生什么情况？如果对象的初始值为 0，每个线程的工作是读对象的值，将它加一并写回去。在理想情况下，线程 A 和线程 B 的工作是这样的：

```
A <- 0
A -> 1
B <- 1
B -> 2
A <- 2
A -> 3
...
```

但是，线程的切换不是理想可控的，因此，在实际的执行过程中，线程的工作过程完全可能是这样的：

```
A <- 0
B <- 0
A -> 1
A <- 1
A -> 2
B -> 1
...
```

显然这里出现了竞争，不解决竞争带来的问题，程序将不能计算出预期的结果。通过引入原子类型可以解决这个问题。原子类型描述的是

```
_Atomic (类型名)
```

这种构造所指示的类型，它也是一种派生类型，派生自这里的“类型名”所指定的类型。原子类型是 C11 引入的，标准并不强制 C 实现必须支持这一类型。以下是两个原子对象的示例：

```
_Atimic (int) ai;
const _Atomic (int) * pai;
```

相对于其他类型的对象，只有原子对象可以避免数据访问竞争，它们可以同时被多个线程修改，或者一个线程修改而其他线程读取而不会出现不一致的情况。

如果 C 实现支持原子类型，则它会提供头文件<stdatomic.h>。该头文件定义了几十种不同类型的宏，比如 atomic_bool 和 atomic_int 和 atomic_uintmax_t 等等，以方便定义各种内置类型的原子版本，例如：

```
# include <stdatomic.h>

atomic_int ai;                  //原子的 int 类型
const atomic_int * pai;         //原子的 const int 类型
```

下面是一个完整的示例，我们声明了两种类型的对象 atom_i 和 norm_i，前者是原子类型，后者则不是。同时，我们创建了 10 个线程，让它们分别将 atom_i 和 norm_i 的值累加 10000 次。结果显示，累加之后 atom_i 的值总是 10000，而 norm_i 则不然。

```
# include <stdio.h>
# include <threads.h>
# include <stdatomic.h>
```

```
atomic_int atom_i;
int norm_i;

int thrd_proc (void * p)
{
   for (int x = 0; x < 10000; x ++) norm_i ++, atom_i ++;

   return 0;
}

int main (void)
{
   thrd_t thrds [10] ;

   for (int x = 0; x < 10; x ++)
      thrd_create (& thrds [x], thrd_proc, 0);

   for (int x = 0; x < 10; x ++)
      thrd_join (thrds [x], 0);

   return printf ("%d, %d\n", norm_i, atom_i);
}
```

3.9 标量

这是一个统称，标量类型（scalar type）包括指针类型和算术类型，具体可参见其各自的词条。在本书中，有时不说“标量类型”，而直接说“标量”，这两种说法是一个意思。

3.10 聚合类型

这是一个统称，聚合类型（aggregate type）包括结构类型和数组类型。具体描述可参见其各自的词条。

联合类型不属于聚合类型，原因是一个联合类型的对象每次只能保存一个成员的值。

3.11 对象类型

用于描述对象的那些类型被称为对象类型（object types）。在C中，不属于函数的其他类

型都属于对象类型。对象类型包括字符类型、算术类型、结构类型、联合类型、数组类型、指针及 void 类型，等等。

3.12 void

void 类型是对象类型，这种类型不能表示具体的值，或者说它没有值，又或者说它是由空值（不存在的值）组成的。void 类型是不完整的类型，而且没有办法使之成为完整类型。

因为 void 不是完整的对象类型，所以不能声明 void 类型的对象。当然，也不存在 void 类型的对象。但是，可以在函数的声明中使用 void 作为参数类型和返回值类型，以表示不接收任何参数或者不返回任何值。例如：

```
void x;                    //不允许，不存在 void 类型的对象
void f (void);             //有效，函数不返回值，不接收任何参数
int g (void v);            //不允许，void 类型的参数声明不能有标识符
```

如果一个表达式没有值，或者说具有空值，则这样的表达式称为 void 表达式，具体参见“void 表达式”。void 表达式的值在任何地方都是不允许使用的，也不能隐式或显式地转换为其他类型。例如：

```
void f (int);
f (0) + 1;                 //错误，不允许使用 void 表达式的值
int i = (int) f (0);       //错误，不允许将 void 表达式的值转换为其他类型
```

也可以创建指向 void 类型的指针。指针是完整的对象类型，所以指向 void 的指针也是完整的对象类型。可以将任何对象类型的指针转换为 void 指针，然后从 void 指针转换为原来的类型，这都是允许的。有关这方面的情况，可参见 “类型转换”。

3.13 限定的类型

C 中的大部分类型都可以用称为限定符（qualifier）的关键字 const、volatile、restrict 和 _Atomic 加以限定。根据需要，它们可以单独使用，也可以组合使用，这样就形成了某种类型的一系列限定版本。

一个类型的未限定版本及它的各个限定版本都是截然不同的类型，但属于同一个类型域（参见“类型域”），类型的表示方法相同，并且有相同的对齐要求。

现有的 4 个类型限定符分别于 1989 年（const 和 volatile）、1999 年（restrict）和 2011 年（_Atomic）开始成为 C 语言的一部分。它们的描述分别参见“const”、“volatile”、“restrict”和“_Atomic”。

用限定符限定的类型是新的类型。因此，int 是一种类型，const int 是另一种类型，它们互不兼容。有些读者以为 const 只用于“修饰”一种类型，使之具有只读的属性，这是不确切的。实际上，限定的结果是产生了新的类型。

在下面的例子中，D1 中的 const int 等效于 int const，将 const 放到 int 的前面，可以直观

地体现这一点。但是，这仅仅考虑了视觉效果，实际上，它们的顺序并不重要，有关这方面的解释，参见“声明指定符”。

```
const int i = 0;                              //D1

const struct t {float f;} t = {0.0};          //D2

struct t v;                                   //D3

t.f = 0.0;                                    //非法，成员是只读的
v.f = 0.0;                                    //允许
t = v;                                        //非法，对象 t 是只读的
v = t;                                        //允许
```

D2 声明了一个类型为 const struct t 的标识符 t。作为副产品，同时声明了一个类型 struct t（而不是 const struct t），不要弄错了。也就是说，D2 实际上等效于

```
struct t {float f;};
const struct t t = {0.0};
```

D3 声明了标识符 v，但它和对象 t 的类型并不相同。前者是 struct t，后者是 const struct t。

传统的 C（K&R C）中没有这些限定符，引入这些类型限定符的目的是为程序的优化提供更有效的手段和方法。针对特定的硬件架构和操作系统环境，可以在不牺牲正确性和安全性的情况下使转换后的代码更快速、更有效地执行，这就是优化（optimization）。对多数 C 实现来说，优化是一个可选项，需要在转换程序时用编译选项指定。

很多重要的优化技术基于缓存的原理。举个例子，考虑这样一个理想化的场景：在转换程序时，C 实现可以“记住”最后一次访问对象时的值，并在下次需要读这个对象的值时直接使用它。

但是，鉴别代码是否可以优化并不是简单的事情，不能完全依赖 C 实现，毕竟它只是软件，缺乏同人类相媲美的智力。所以，从 C89 开始，C 先是引入了 const 和 volatile，又引入了 restrict，目的很简单，就是要求程序员分担一部分程序优化的责任。

每个限定符都像是一个信号，程序员用它来向 C 实现保证他会做什么，不会做什么；C 实现在转换程序时，一旦看到这些信号，就知道在优化程序上，哪些是自己可以做的，哪些是不可以做的。

我们知道，很多类型派生自其他类型。如果类型 T 派生自限定的类型 P，那么，P 的限定符不会通过派生过程传递给类型 T。下面结合指针、结构和数组等类型来阐述这一规则。

指针派生自它所指向的类型。如果一个类型是 const int，则可以派生出一个如下所示的指针类型 const int *。但是，该指针的类型不是“const 指针”，而是“指向一个 const int 类型的指针”，指针本身是无限定的。

```
//pi 的类型是“指向 const int 的指针”，但指针本身并非“const 指针”
const int * pi;
```

要想生成一个 const 指针，必须对指针单独进行限定：

```
int * const cp;          //cp 是 const 指针，它派生自无限定的 int 类型
const int * const cpc;   //cpc 是 const 指针，它派生自 const int 类型
```

结构类型派生自它的成员类型。如下例所示，struct t 派生自两个成员类型 const int 和 const float，但 struct t 并不是 const 限定的类型。进一步地，pt 的类型是指向结构类型 struct t 的指针，但该指针并不是 const 限定的（指针），仅仅是它所指向的结构具有两个 const 限定的成员。

```
struct t {const int i; const float f;};
struct t t, * pt;
```

数组类型派生自它的元素类型。如下例所示，a 的类型是数组，尽管它的元素类型是 const int，但这个派生的数组并非是“const 数组”，而是“元素类型为 const int 的数组”。当然，const 数组在 C 中是不存在的，只有数组的元素是只读的。

```
const int a [3];
```

函数类型被认为是派生自它的返回类型。看起来函数本身可以是限定的（如只读函数），但这是没有任何意义的。

以上的讲解的内容适用于 volatile 和 restrict。

类型限定符有一个特点：同一个类型限定符可以多次出现在同一个声明指定符（参见“声明指定符”）中或者指定符限定符列表（参见“结构或联合指定符”的语法图）中。但是，不管出现多少次，它在那里都只相当于出现了一次。

。

如下例所示，两个声明指定符 int 连用是非法的，但两个声明指定符 const 连用却不会有任何问题：

```
int int i;                                  //非法
const volatile const const int ci; //等效于const volatile int ci;
int fn (const const int);                   //等效于int fn(const int);
```

3.13.1 const 限定的类型

用类型限定符“const”限定的那些类型是 const 限定的类型。关键字“const”的本意不是“常量”，“常量”一词在 C 中的特定的含义，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”。

有时会创建这样的对象：它的值仅用于读出，而不用于写入和更新操作。当然，如果足够小心，对象的类型不是问题，要做的仅仅是不要编写出修改对象值的代码。

此外，如果一个对象仅用于读出（而不是写入），则它的值只读一次即可，不必在每次用到对象的值时，都真的执行一次底层的存储器读出操作。换句话说，它只需要在开始的时候读一次，然后缓存这个值即可。

但是，编译器（C 实现）需要一个字面上的保证，这就是关键字“const”。如果一个对象是 const 限定的类型，则意味着程序员不准备写入这个对象，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对象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。

当然，有时候可以绕开这种限制，用一个表达式（具有非 const 限定的类型）来修改该对象的值，但这种行为的后果是未定义的。例如，下面的例子定义了一个指针 pi，令它指向对

象 x。可以看出，我们并不希望通过 pi 来修改它所指向的对象，所以才将 pi 被声明为指向 const int 类型的对象。因为这个原因，S1 是非法的。

但是，通过强制类型转换，可以将一个表达式的类型转换为另一种类型以绕开这种限制，就像 S2 所做的那样：

```
int x = 0;
const int * pi = & x;
(* pi) ++;              //S1
(* (int *) pi) ++;     //S2
```

上面这个例子在运行的时候不会有什么异常，因为 pi 所指向的对象 x 实际上并不是 const 限定的类型，而且对象 pi 并不是 const 指针。但下面的代码就不一定了：

```
const char * const p = "Yes, my lord.";
* (char *) p = 'x';      //将p的类型强制转换为char *
```

首先，字面串"Yes, my lord."最终会用于初始化一个不可见的静态数组，且这个数组很可能位于一个只读的存储区域（这取决于 C 实现）。为了防止错误地使用 p 和 p 所指向的这个只读区域，我们将 p 定义为本身是只读的，且指向只读位置的指针。但是这段代码表明，只要用心，我们就能轻松地突破这个限制，但后果却难以想象。

限定符 const 可以看做程序员向 C 实现传递一个信息：只是读取对象的值，而不会改变它。在程序转换期间，C 实现会检查代码是否违反了 const 的语法规则。下面的例子很好地展示了这一点，语句 S1 中，对象 c 的类型是指向 const char 的指针，所以*c 指示 const char 类型的对象，不可以为该对象赋值，所以非法。

语句 S2 中，标识符 i 指示一个 const int 类型的对象，同样不可以为它赋值，所以也是非法的；

```
void  f (const char * c)
{
    * c = 'x';              //S1

    const int i = 0;
    i = 1;                  //S2

    * (int *) & i = 1;      //S3
}

void g (void)
{
    char a [3];
    f (a);                  //S4
}
```

在语句 S3 中，表达式&i 的类型是 const int *，通过强制类型转换，得到的新类型是 int *。此后，可以用运算符*来修改对象 i 的值，但这是未定义行为。

顺便说一下，在语句 S4 里，表达式 f(a)等效于 f(&a[0])。这是因为，除非作为运算符&和 sizeof 的操作数，否则数组类型的表达式 a 将被转换为指向其第一个元素的指针。

注意，由于“const”这个单词的原因，有些读者会将 const 和“常量”联系起来，这是一种误会，在 C 中，它们之间毫无关联。

使用 const 限定符能够增强程序的可读性，并在一定程度上保护对象的值不被破坏。但是，正如上例中所示的那样，这种保护实际上并非牢不可破，同时非常危险。

const 限定符有自己的作用范围。如果多个程序共享同一个对象，那么它在一些程序中是 const 限定的，而在另一些程序中则不是；如果多个线程共享同一个对象，那么情况也可能同样如此。

在上面的例子中，数组 a 的元素在函数 g 中是可以修改的，但在函数 f 中却是只读的。如果 C 实现认为可以，它会将一个用“const 限定的类型”定义的对象放在当前程序的只读存储区域。比较典型的例子是放在一个被标记为只读的页面，当企图写入这样的页面时，会被处理器阻止并产生异常中断信号。当然，这只是可能，并且是由 C 实现决定的。

要理解 const 限定符的意义，有一点很重要：不要把它看作一种攻防关系。例如，当程序中出现了这样的声明：

```
const int i = 0;
```

用户的本意是自己不准备修改对象 i 的存储值，而且希望 C 实现根据这种意图提供适当的优化。

3.13.2 volatile 限定的类型

用类型限定符“volatile”限定的类型是 volatile 限定的类型。const 的意思是保证对象的值不会被修改，而 volatile 则正好相反，它警告 C 实现，对象的值会改变，而且是以不受控制的方式改变。

如果一个类型是“volatile 限定的类型”，则意味着用该类型所定义的对象，它的值不单单会被当前程序的代码修改，还可能潜在地被其他程序和代码修改。因此，这也意味着，C 实现不能在转换期间对访问该对象的代码做优化处理。

类型为“volatile 限定的类型”的对象，其值对于当前程序来说是“易变的”，不能保证它存储的值就是用户上次保存的值，即使在此期间当前程序并没有修改这个对象的值。因此，对这种对象的访问（读和写）不能依赖于缓存特性。例如，它可能对应着一个硬件端口，或者 BIOS 数据区的某个位置，或者几个程序和线程共用的存储区域，等等。例如在 PC 中，键盘接口有一个端口用于存放按键代码，程序可以读取它的内容，但不能保证它的内容不变，因为硬件会随时用新的按键代码取代先前的代码。

不适当的优化就如同以不正确的方式使用锤子，很可能伤到自己。例如：

```
int * ready = (int *) 0xFFFFFFCAUL;


* ready = 0;
while (! * ready) ;
```

在上面这个例子中，ready 指向一个硬件地址，我们假定其他程序（如一个定时器中断处理过程）也访问这个地址。当前程序的意图很明显，就是把表达式*ready 作为一个标志，当

它的值为 0 时，说明当前程序必须继续等待，因为数据还没有准备好，或者设备还没有就绪，等等；一旦表达式*ready 的值为非零，则退出 while 循环，继续向下执行。

一元运算符!执行逻辑否定，因此，若表达式*ready 的值为 0，则表达式!*ready 的值为 1；若*ready 的值不为零，则表达式!*ready 的值为 0。仅有一个分号“;”的语句是空语句，不执行任何操作。

如果在程序转换时不做任何优化，这段代码会工作得很好，while 语句执行时，会先读对象的存储值，然后做出判断。在这种情况下，while 语句会一直反复执行，直到对象的值由 0 变成 1。

但是，当对程序做优化处理时（可尝试在 gcc 下使用-O3 选项），因为看到表达式*ready 的初始值是 0，表达式!*ready 的值是 1，所以它可以很自然地认为这是一个无限循环，类似于

```
while (1) ;
```

为了不在优化程序的时候伤及无辜，最好的做法是将 ready 声明成 volatile 限定的类型。

```
volatile int * ready = (int *) 0xFFFFFFCAUL;
```

如此，C 实现就知道，即使是对程序进行优化，对于 while 的每一次循环都要读取对象的存储值。

下面是另外一个例子，这里声明了一个指针 pseconds_since_boot，它指向一个数据存储器的物理地址 0xfffffff0。假定底层硬件每隔一秒增加该存储单元的数值，使之能够实时保持自开机以来所经历的秒数。

注意，pseconds_since_boot 的类型是“指向 const volatile unsigned long 类型的指针”，而指针本身并不是只读的、不可修改的。限定符 const 可以防止当前程序的代码无意中改变了这个地址中的值；限定符 volatile 禁止程序在转换时做不适当的优化。

```
const volatile unsigned long * pseconds_since_boot
                              = (unsigned long *) 0xfffffff0;
* pseconds_since_boot = 2000;    //非法
```

3.13.3 restrict 限定的类型

用类型限定符“restrict”限定的类型是 restrict 限定的类型。但是，restrict 限定符仅适用于指针类型。如果一个指针类型是 restrict 限定的，则它所指向的对象和该指针便有了一种特殊的联系：在一个块内（函数体或者复合语句），所有到这个对象的引用必须直接或者间接通过这个指针（的值）进行。

有了这个保证，C 实现便可以在块的开始处安全地缓存“该指针所指向的对象”的值，执行各种操作，读取和更新操作也只针对这个缓存的值进行。最后，在退出块之前，再将缓存的值（可能已经更新过）刷新到那个指针所指向的对象。毕竟，对于某些复杂的操作而言，缓存一个对象的值，并将这个值用于后续计算的开销，比频繁地通过指针访问那个对象的值要小得多。退出块之后，其他指向那个对象的指针即可重新使用。

如果没有这个限定符，则意味着在当前块内，还可能存在着其他指向这个对象的指针。因此，缓存对象的值是不安全的。

下面用具体的实例来解释在 C 中引入 restrict 限定符的原因。假定函数 fdemo 的定义如下：

```
void fdemo (int * p1, int * p2, int n)
{
    for (int i = 0; i < n; i ++)
    {
        * p1 += i;
        * p2 += i;
    }
}
```

为了防止 C 实现对函数 fdemo 的代码过度优化，不用常量来指定 for 语句的循环次数，相反，循环次数由调用者传入的参数 n 指定。函数 fdemo 所做的事情很简单，就是对 p1 和 p2 所指向的对象做累加操作，要累加的数是 0 到 n-1。

再假定，使用以下代码来调用函数 fdemo。

```
int i = 0;
fdemo (&i, &i, 10);
```

这就是说，在函数 fdemo 内部，指针 p1 和 p2 指向同一个 int 类型的对象。当然，这在 C 中是很正常的事情，不会影响到函数 fdemo 的正确执行，在 for 循环中，先读取 p1 所指向的对象的值，进行累加后再写回该对象，再对 p2 做同样的事情，最后从函数 fdemo 返回后，对象 i 的值是 90。

如果修改函数 fdemo 的定义，则情况就大不相同了。

```
void fdemo (int * restrict p1, int * restrict p2, int n)
{
    for (int i = 0; i < n; i ++)
    {
        * p1 += i;
        * p2 += i;
    }
}
```

同前一种定义形式相比，这里只是将 p1 和 p2 声明为 restrict 限定的指针，其他方面没有任何变化。

如果一个指针使用了 restrict 关键字，则意味着该指针是“访问它所指向的对象”的唯一途径。因此，为了优化函数 fdemo 的代码，C 实现将首先分别缓存 p1 和 p2 所指向的对象的值，以免频繁地用指针访问对象而丧失效率。缓存的值随后用于累加操作，累加完成后，再将缓存的值（可能已经改变）刷新回它们各自所指向的对象。

问题在于，这次依然用相同的代码来调用函数 fdemo：

```
int i = 0;
fdemo (&i, &i, 10);
```

函数 fdemo 的参数 p1 和 p2 都被声明为 restrict 限定的，但它们又都指向相同的对象，这就违背了双方在 restrict 上的约定。因此，在函数 fdemo 内部，for 循环里的两个累加过程都假定只有自己在改变对象的值，因此用的是缓存的值，而不是通过实际访问对象得到的值。同

时，每当对象的缓存值改变后，也不会刷新到对象中。于是，离开 for 循环之后，它们各自的缓存值都是 45。在退出函数 fdemo 之前，这两个缓存值又先后被刷新到指针所指向的对象，这等于分两次将 45 写回到那个对象中。最后，从函数 fdemo 返回后，对象 i 的值是 45。

3.14 完整类型

对象的大小取决于它的类型，C 实现要创建一个对象就必须知道它的大小。如果对象的类型可以提供足够的信息让 C 实现知道对象的大小，那么这个类型就是一个完整的对象类型，或者简称完整类型（complete type）。

在程序中，很多操作需要知道并依赖于对象的大小，这就要求它的类型是完整的。比如下面的例子，对象 a 的类型是 char [128]，该类型的大小是确定的；unsigned int 是完整的对象类型，它可用于声明对象 x；反过来，对象 x 的类型是 unsigned int，是完整的对象类型，它可以作为运算符 sizeof 的操作数。

```
char a [128];
unsigned x = sizeof x;
```

基本类型（参见“基本类型”）都是完整的对象类型，它们都具有已知的大小。

3.15 不完整类型

与完整类型（参见“完整类型”）不同，有些类型缺少某些信息，无法知道它的大小，像这样的类型称为不完整类型（incomplete type）。在下例中，标识符 a 的类型是 char []；标识符 b 的类型是 char [][3]；指针 pt 所指向的类型是 struct t，它们都是不完整类型。

```
/* 假定在此处看不到 struct t 的其他声明 */
char a [], b [] [3];
struct t * pt;
```

没有指定大小的数组类型是不完整类型。如果用这样的类型声明了一个标识符，则我们说这个标识符的类型是不完整的数组类型。要使这个标识符的类型成为完整的数组类型，只需在晚些时候用明确的大小重新声明一下（外部或者内部链接）即可。例子：

```
# include <stdio.h>

int a [];       //在当前点上，a 的类型 int [] 是不完整类型，a 是外部链接的

void f (void)
{
    printf ("%zu\n", sizeof a);         //S1
    printf ("%d\n", a [0] = 1972);      //S2
}
```

```
int a [3];        //在当前点上，a 的类型 int [3]是完整类型，a 是外部链接的

void g (void)
{
    printf ("%zu\n", sizeof a);          //S3
    printf ("%d\n", a [0] = 1981);       //S4
}
```

在程序转换期间，第一次遇到声明

```
int a [];
```

时，它是不完整类型，因为没有指定它的大小。问题在于，S1 中的 sizeof 运算符用于求得其操作数的大小，且如果它的操作数是数组，数组也不会转换为指针，这就要求它的操作数必须是完整类型。但它的操作数 a 并不是完整类型，故 S1 是非法的；在 S2 中，赋值表达式

```
a [0] = 1972
```

尽管也使用了数组 a，但在这里，它要被转换为指向其首元素的指针，且这个转换并不需要知道数组的大小，而且 C 也从不对数组做越界访问的检查工作。因此，S2 是合法的。

接下来，我们重新声明了 a，且这一次指定了其大小：

```
int a [3];
```

此时，C 实现才能确定要为 a 分配多少空间。换句话说，数组 a 现在的类型是完整的。所以，S3 和 S4 都是合法的。

这就是说，在一个转换单元内的不同点，一个对象类型可以是完整的，也可以是不完整的。

如果在同一个源文件内同时声明 a 的完整和不完整类型有点奇怪，则在一个多人协作的程序中，组成一个程序的各个源文件之间可以通过这种方式来共享相同的实体（对象或者函数）。

在下面的例子中，源文件 a.c 中声明了一个完整的数组类型（因为存在初始化器，尽管对 cc 的声明看起来是不完整类型，但是在声明点之后，它是完整类型）。

```
const char cc [] = "Yes, I do.";
void f (void) {/* …… */}
```

在组成当前程序的另一个源文件 b.c 中，可以再次声明 cc，但这一次它是不完整类型。因为函数 g 只是以指针的方式操作对象 cc，故 cc 在这里是否为完整类型并不重要。

```
# include <stdio.h>

void g (void)
{
    extern const char cc [];
    printf (cc);
}
```

在转换单元内引用一个结构或者联合时，如果在当前位置看不到该结构或者联合的成员，那么它就是一个不完整的（结构或者联合）类型。要使它成为完整类型，只需以后在相同的作用域内，重新声明那个结构或者联合类型的标记，但这次一定要带有成员。

先来看一个简单的例子，以下，在 D3 处能看见 D1，所以 D3 不是结构类型的声明，而是指针对象 ps 的声明，这里只是指定了 D1 中的结构类型（参见“结构或联合指定符”）。但 D1 是不完整类型，所以在 D3 处也看不到结构类型 s 的成员（但声明一个指向结构类型的指针并不需要知道结构的完整信息）。要使 struct s 成为完整类型，需要在 D1 的作用域内，重新声明，并带有成员即可，这就是 D6 的作用。

结构类型 t 是在 D2 处声明的，而且是完整类型，带有成员信息。D4 是一个结构类型的声明，它和 D2 的作用域相同，且标记也相同，所以它和 D2 是同一结构类型（参见“结构或联合指定符”），它是结构类型 t 的再次声明。尽管这次声明没有成员列表，但这里能看到 t 的成员，所以 D4 处的结构类型 t 是完整类型。同理，D5 中的结构类型 t 也是完整类型，而且指定的是在 D2 和 D4 处声明的结构类型。

```
/* 假定在此处看不到结构类型 struct s 和 struct t 的任何声明 */
struct s;                       //D1

void f (void)
{
   struct t {/* …… */};         //D2
   struct s * ps;               //D3
   struct t;                    //D4
   struct t t, * pt;            //D5
}

struct s {/* …… */};            //D6
```

注意，void 类型也是对象类型，而且是不完整类型。有别于其他不完整类型，void 不能补充声明为完整类型。

3.16 类型域

类型的划分将表达式和运算符的操作数彼此区分开来，而类型域（type category）则以另一种方法将适当的类型重新合并。

任何非派生的类型，都自动构成一个单独的类型域。进一步地，任何一种整数类型都从属于整数类型域；任何一种浮点类型都从属于浮点类型域；任何一个算术类型都从属于算术类型域；任何一个实数类型都从属于实数类型域；任何一个复数类型都从属于复数类型域，等等。

至于派生类型的类型域，如果它不是派生自其他派生类型，则它从属的类型域就是其自己的类型；否则是一个派生类型的最外层派生（就像前面那些派生类型的构造一样）。

3.17 类型的表示

所谓类型的表示（representations of types）是指，对于特定的类型，存储它的值所需要的存储空间数量、数据的编码方式，以及如果该类型的值需要多个字节来存储的话，这些字节的存储顺序。

首先要明确的是，任何一个 char、signed char 或者 unsigned char 类型的对象，都是一个字节的长度，其包含的比特数是 CHAR_BIT（这是一个宏，在头文件<limits.h>中定义），而且保证不含填充比特，具体可参见“填充比特”。

其次，unsigned char 类型的对象使用纯二进制计数法来表示它的值，具体可参见“纯二进制计数法”。另外，可以确保使用纯二进制计数法来表示值的还包括无符号类型的位字段，具体可参见“位字段”。

对象在存储器中的状态是一个或者多个连续的字节。当然，位字段除外，因为它们可能不足一个字节的长度，或者其长度不是字节的整数倍。对象的字节数取决于它的类型，字节的安排顺序及编码方式原则上由 C 实现决定，但是 C 标准也在某些方面做了一些强制性的规定，本节讲述的就是这些规定，但未必准确，也未必全面，最终还要以标准的文档为准。

如果一个对象不是位字段，而且它的长度是 N 个字节，则因为一个字节所包含的比特数是 CHAR_BIT，故它必然是由 N×CHAR_BIT 个比特组成的。至于位字段对象，它是一个比特的序列，其长度（数量）在声明时指定。位字段的值由这些比特形成，这些比特位于承载它们的那个可寻址的存储单元中。

值在对象中的形态可以通过它的对象表示来了解，具体内容可参见“对象表示”。

某些对象可能存在填充比特，这使得一个值可能具有多种对象表示，具体内容可参见“填充比特”。

填充比特和值的对象表示是透明的，即使用户不知道它们的存在也不用担心，因为它们不影响运算的结果。即使一个值有多种对象表示，在使用任何一种运算符对其进行操作时，结果依然是正确的。另外，如果要写入对象的值有多个对象表示，则 C 实现可以自由选择使用哪一个对象表示。

举例来说，如果某个 C 实现将 CHAR_BIT 定义为 9，且它的 unsigned int 类型有一个填充比特。那么，以下代码可以正常执行，且不会影响到结果的正确性。

```
unsigned int i = 82789;
i >>= 1;
```

对象表示是值在对象中的存在形态，但有些对象表示并不能形成该对象的合法存储值，这种对象表示称为自陷表示，具体可参见“自陷表示”。

组成一个对象的所有比特分为 3 类（部分）：值比特、符号比特和填充比特。值比特是真正用于表示值的部分；符号比特用于有符号数，用于表示数的正负，只能有一个符号比特；填充比特可能用于特殊的目的，也可能没有任何用处。值比特总是需要的，但是符号比特和填充比特存在与否，可能和对象的类型有关。

3.17.1 纯二进制计数法

纯二进制计数法（pure binary notation）和十进制相似，都是位置计数法。纯二进制计数法用数字 0 和 1 来表示一个整数，整数的值由每个数位的值决定，而每个数位的值又取决于它的位置。

具体来说，整数的值由每个数位的值相加得到（但可能不包括最高的符号位），每个数位的值等于该数位上的数字乘以 2^n，n 从 0 开始，由最低位向最高位依次加一。

unsigned char 类型的对象用纯二进制计数法来表示它的值。因为一个字节包含的比特数是 CHAR_BIT，所以 unsigned char 类型所能表示的值为 $0 \sim 2^{CHAR\_BIT}-1$。

无符号类型的位字段也用纯二进制计数法来表示它的值。如果一个无符号的位字段由 3 个比特组成，则它的取值为 0～7。

其他类型的对象也可能会用纯二进制计数法来表示它们的值，可参见“对象表示”一节中的有关内容。

3.17.2 对象表示

有时候，用户希望知道值在对象中的形态。要做到这一点，需要借助于 unsigned char 类型，因为它很特殊：长度是 1 个字节，没有填充比特，采用纯二进制记数法表示它的值（参见“类型的表示”），最重要的，它是访问任何对象的有效类型（参见“有效类型”）。

因此，我们可以先把存储在非位字段对象中的值复制到一个 unsigned char 类型的数组对象中，而且保证数组的大小和对象的字节数相同。至于位字段对象，组成它的那些比特由可寻址的存储单元承载，这是一个比特的序列。

字节的数组和比特的序列能够帮助用户观察值的组成。因此，我们把这些字节的集合或者比特的序列称为值的对象表示，简称对象表示（object representation）。

举例来说，如果一个 unsigned char 类型的对象拥有最大值 UCHAR_MAX，那么在 CHAR_BIT == 8 的情况下，它的对象表示为 0xFF。

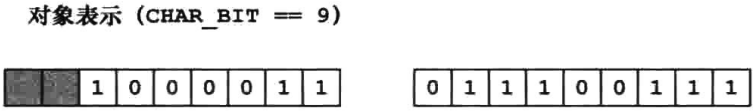
下面是另一个例子，用于演示如何得到+0.0 和-0.0 这两个浮点值的对象表示。

```
# include <stdio.h>

int main (void)
{
    union {double f; unsigned char c [sizeof (double)];} u;

    u.f = +0.0;
    for (unsigned i = 0; i < sizeof u; i ++)
        printf("%.2X\t", u.c [i]);

    printf ("\n");

    u.f = -0.0;
    for (unsigned i = 0; i < sizeof u; i ++)
```

```
        printf("%.2X\t", u.c [i]);

    return 0;
}
```

同类型的两个值，如果它们的对象表示完全相同，那么这两个值在比较时肯定是相等的。但是反之则不然：即使两个值在比较时相等，而且它们的类型也相同，也并不意味着它们具有一模一样的对象表示。不过这里不涉及浮点中的 NaN，它是一个非数字的特殊值。

前者较容易理解，对于后者，在值的对象表示中，可能存在与值无关的填充比特（参见“填充比特”）。如图 3-6 所示，假定它是一个（对象的值的）对象表示，显然，在该对象所在的硬件架构上，一个字节包含 9 个比特。



注意：深色的格子代表填充比特，其他是值比特。

图 3-6 值的对象表示

显然，该对象表示有两个填充比特。但是因为填充比特的值是未指定的，所以不管它是什么，在这里都不影响该对象的值。这就是说，以下各个不同的对象表示：

```
001000011 011100111
011000011 011100111
101000011 011100111
111000011 011100111
```

都表示同一个值（粗体部分是填充比特的内容）。换句话说，填充比特的组合和值比特一起，是同一个值的不同对象表示的各种替代形式。值得一提的是，填充比特可能位于对象表示中的任何位置，而不一定在头部或者尾部。

进一步地，即使两个值相等，也不一定是同一个值。比较典型的例子是浮点数中的正零和负零，以及有符号整数类型里的正常零和负零。浮点数中的+0 和-0 在比较时是相等的，但它们不是同一个值；在对 1 的补码和符号带大小的数值表示方案中，如果 C 实现支持负零，则正常零和负零在比较时是相等的，但它们不是同一个值。

本节余下的部分将集中讨论整数类型值的对象表示。

第一，对于 unsigned char 和 char（如果 char 等价于 unsigned char）类型的值来说，它们的对象表示中仅包含值比特，而没有符号比特和填充比特（可参见“值比特”、“符号比特”和“填充比特”）。

第二，对于其他任何无符号整数类型的值来说，它们的对象表示中包含值比特和可能（只是有可能）存在的填充比特。

第三，表示无符号整数类型的值时，仅使用值比特，而且采用纯二进制计数法（参见“纯二进制计数法”）。如果一个无符号整数类型的对象表示中有 N 个值比特，则该类型可以表示的值为 $0 \sim 2^N-1$。

第四，对于 signed char 和 char（如果 char 等价于 signed char）类型的值来说，它们的对

象表示中仅包含值比特和一个符号比特，没有填充比特。

第五，对于其他任何有符号整数类型的值来说，它们的对象表示中包含一个符号比特、值比特和可能（只是有可能）存在的填充比特。

第六，表示有符号整数类型的值时，标准允许采用 3 种方案：符号带大小、对 1 的补码和对 2 的补码。具体选择哪一种方案，取决于 C 的实现。假定在一个有符号整数类型的对象表示中有 N 个值比特，则

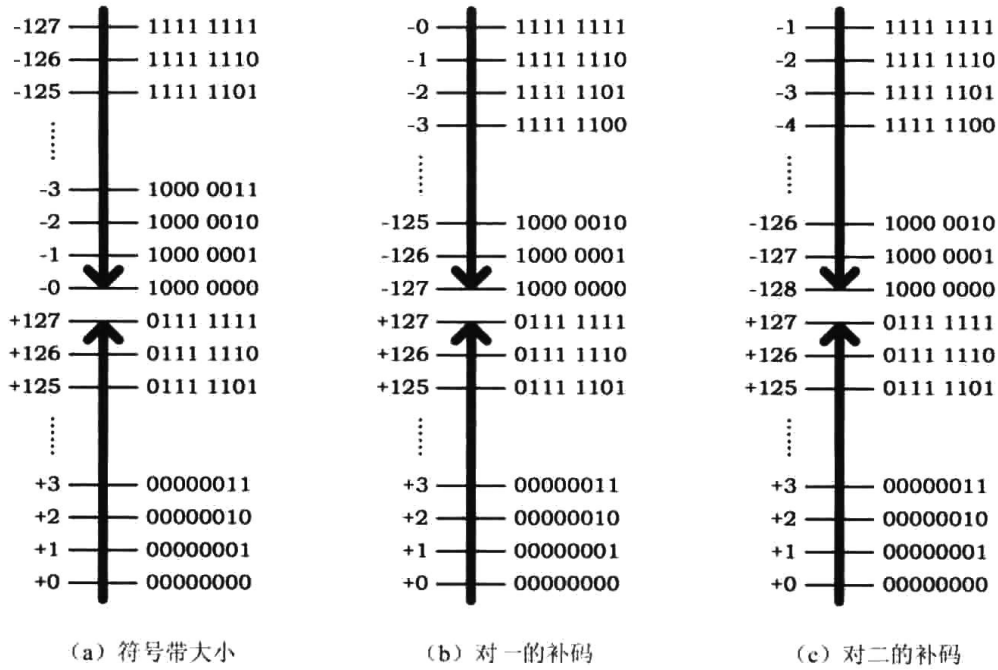
（1）如果采用符号带大小方案，符号比特仅表示值的正负，值的大小由值比特采用纯二进制计数法得到。因此，使用这种方案的有符号整数类型可以表示的值为$-(2^N-1)$～2^N-1。显然，在这种方案中有两个零值：正常零和负零（参见“负零”），如图 3-7 所示。

（2）如果采用对 1 的补码方案，则有符号整数类型的值由符号比特的值和值比特形成的值相加得到。如果符号比特为 0，则它表示的值是 0；如果为 1，则它表示的值是$-(2^N-1)$。值比特形成值的方法也是纯二进制计数法。值比特可以形成的值为 0～2^N-1，因此，使用这种方案的有符号整数类型可以表示的值为$-(2^N-1)$～2^N-1。这种方案同样存在正常零和负零，如图 3-7 所示。

（3）如果采用对 2 的补码方案，则有符号整数类型的值由符号比特的值和值比特形成的值相加得到。如果符号比特为 0，则它表示的值是 0，如果为 1，则它表示的值是-2^N；值比特形成值的方法也是纯二进制计数法。值比特可以形成的值的为 0～2^N-1，因此，使用这种方案的有符号整数类型可以表示的值为-2^N～2^N-1。

为了更直观地展示这 3 种方案的特点和区别，图 3-7 给出了每种方案的示例，箭头左边是值，右边是相对应的对象表示。使用 signed char 类型的好处是它的对象表示中不存在填充比特。

以下 3 个图示都假定为 signed char 类型，且 CHAR_BIT==8。



（a）符号带大小　　（b）对一的补码　　（c）对二的补码

图 3-7　有符号整数类型的对象表示

非常清楚的是，在前两种方案（符号带大小和对一的补码）中，零值有两种不同的对象

表示。在符号带大小和对 1 的补码中，都存在符号位是“1”的零。这种对象表示在某些 C 的实现中被视为自陷表示。对于任何整数类型来说，所有比特都是“0”的对象表示是零值的正常表示形式。

3.17.2.1 负零

一个有符号整数类型的值可以使用三种表示方法：符号带大小、对 1 的补码和对 2 的补码。在前两种方案里，零值有两种不同的对象表示。

如果 C 实现用符号带大小或者对 1 的补码表示负数，而且将符号位是“1”的零视为合法的正常值，则将这种零的对象表示称为负零（negative zero）以示区别。如果 C 实现不支持负零，那么这样的零是具有自陷表示的值。如果某个对象具有自动存储期且未被初始化，则它很可能拥有这样的自陷值。

另外，对于以下代码，经常有人问这样一个问题：对象 i 及指针 pi 的值是否所有比特都是“0”？基于以上的阐述，这个问题很清楚，即使不考虑填充比特，答案也是“不一定”。

```
# include <stddef.h>
int i = 0, * pi = NULL;     //宏NULL是在头文件<stddef.h>中定义的
```

标准并不要求 C 实现必须支持负零，即使它采用符号带大小或者对 1 的补码来表示负数。在这种情况下，只有一种合法的零。进一步地，如果它支持负零，那么负零值只能由&、|、^、~、>>、<<、&=、|=、^=、>>=或者<<=这些二元运算符产生；或者由其中一个操作数是负零的+、-、*、/、+=、-、=、*=或/=运算符产生。

注意，这不是说以上两种方式只能产生负零，产生的也可能是正常零。所以，实际上产生的是正常零还是负零，负零在实际存储到对象时是否会被转换为正常零，都是未指定的，由 C 实现自己决定。

以下，假定 C 实现支持负零，且表达式~y 的值（或者说运算符~的值）是零，那么这个零到底是负零还是正常零，取决于操作数 y 的原始内容。如果它是正常零，那么在保存到对象 x 时保持不变；如果它是负零，那么在保存到对象 x 时会不会转换为正常零，这是不确定的。

```
int x = ~ y;
```

以下，假定 C 的实现支持负零，且表达式 x+=y 的值（或者说运算符=+的值）是零，那么 x 和 y 至少有一个必须是负零。如果运算符的结果是正常零，那么在保存到对象 x 时保持不变；如果它是负零，那么在保存到对象 x 时会不会转换为正常零，这是不确定的。

```
int x, y;
/* …… */
x += y;
```

如果 C 实现用符号带大小或者对 1 的补码表示负数，但它又不支持负零，那么二元运算符&、|、^、~、<<和>>的结果不能是负零，违反此规定的行为是未定义的。

我们知道，对于任何一种有符号整数类型 S 来说，都有一个无符号整数类型 U 与之对应。给定任何一个类型 S 的值，如果它为非负（符号比特是 0）且不是自陷表示，那么这个值的对象表示对于类型 U 来说，也同样不是一个自陷表示，而且表示同一个值。

这就是说，对于以下声明：

```
long long ago = 250;
```

对象 ago 的值是 250，它可能有多个有效（非自陷）表示。这些非自陷表示也被认为是 ago 被声明为 unsigned long long 时的有效对象表示，而且表示同一个值。

3.17.2.2 精度

在整数值的对象表示中，排除符号比特和填充比特，剩余的部分是真正用于表示数值大小的，其数量取决于整数的类型，称为整数类型的精度，简称精度（precision）。参见“符号比特”和“填充比特”。

例如，当一个字节的比特数为 8 时，signed char 类型的精度是 7，unsigned char 类型的精度是 8。

3.17.2.3 宽度

在整数值的对象表示中，用于表示精度的那些比特和符号比特一起被称为整数类型的宽度（width）。参见“符号比特”和“精度”。

例如，当一个字节的比特数为 8 时，char、signed char 和 unsigned char 这三种类型的宽度都是 8。

3.17.3 自陷表示

对于任何对象类型来说，它的每个值都有自己的对象表示，而且可能对应着多个对象表示。但是反过来，有些对象表示并不对应着那种类型的任何有效值，这就是所谓的自陷表示（trap representation）。

例如，如果对象表示中含有校验位，则错误的校验将导致一个无效的对象表示（自陷表示），从而不能表示一个有效的值。

再如，如果一个 C 实现采用对 1 的补码来表示负数，且它不支持负零，那么负零的对象表示就是一个典型的自陷表示。有关对 1 的补码可参见“对象表示”。

字符类型（char、signed char 和 unsigned char）有其特殊性：没有填充比特；采用纯二进制计数法来表示值；不存在自陷表示。所以，用字符类型的左值来读取任何一个对象的存储值是安全的，不管该存储值是否具有自陷表示。反之，使用其他类型的左值来读取一个具有自陷表示的存储值是不安全的未定义行为。下例中，对象 i 具有自动存储期，因为没有初始化，所以它的存储值可能具有自陷表示。但是，这里将它的地址赋给一个指向 char 的指针，并通过字符类型来读取它的存储值，这是没有问题的。

```
void f (void)
{
    int i;
    char * pc = (char *) & i;
    if (* pc == '\x6') {/* …… */}
}
```

再来考虑一个结构类型的赋值问题：

```
void f (void)
```

```
{
    struct {int i; double d;} x, y;
    x.i = 0;
    y = x;
}
```

现在的问题是，并没有为对象 x 的成员 d 赋值，它可能具有自陷表示的值。那么，将 x 的值赋给 y，这里是否有未定义行为？

答案是没有问题，这是合法的。结构和联合类型有其特殊性，对于一个结构或者联合类型的对象来说，它可能没有初始化，也可能有没有被赋值，它的成员也可能具有自陷表示的值，但这一切都绝不会使这个结构或者联合对象具有自陷表示的值。

3.17.3.1 未指定的值

在需要一个值的场合，如果有很多非自陷表示（参见“自陷表示”）的有效值可供选择，但不能确定这个值到底是什么，标准也未指定应该使用哪一个，那么，对于这样一个尚不明确的值，我们称之为未指定的值（unspecified value）。

正如以上所暗示的，未指定的值不是具有自陷表示的值。由于在值的对象表示中存在填充比特和填充字节，而这些填充部分不对应于任何类型，也就不存在所谓的自陷表示。

首先，如果一个整数值的对象表示中存在填充比特，则这些填充比特各自是 0 还是 1，它们组合在一起形成什么值，是未指定的。

其次，结构或者联合对象中的任何填充字节都具有未指定的值，即使是在修改了结构、联合对象的存储值，或者修改了结构、联合对象任何成员的存储值之后也是如此。也正是因为这个，将一个结构类型的对象的值赋给另一个结构类型的对象时，不需要复制填充比特。

联合类型的对象要特殊一些，因为各个成员的对象表示有重叠的部分。当用户把一个值保存到联合对象的某个成员 M 后，如果从整个联合对象的值的对象表示中去掉与成员 M 相关（重叠）的部分，再去掉尾部填充，则剩余的部分具有未指定的值。

这就是说，对于如下示例：

```
union {char c; int i; double d;} u;
u.c = 0;
```

不考虑和成员 c 对应的那部分对象表示，与 i 和 d 相对应的那部分对象表示分别具有未指定的值。

3.17.3.2 不确定的值

在需要一个值的场合，如果不知道这个值到底会是什么，但它可能是一个未指定的有效值，也可能是一个具有自陷表示的无效值，那么，对于这样一个尚不明确的值，我们称之为不确定的值（indeterminate value）。有时也可称之为“不明确的值”。

3.17.4 符号比特

在有符号整数值的对象表示中，有且只有一个用来表示数值符号（正或负）的比特，这个比特称为符号比特（sign bit）。符号比特也可以称为符号位。

3.17.5 值比特

在整数值的对象表示中，除符号比特和填充比特外（如果有的话），那些真正用于表示数值的比特，称为值比特（value bits）。

3.17.6 填充比特

在整数值的对象表示中，不能用于表示数值，也不用于表示数值符号的比特，称为填充比特（padding bits）。填充比特又称填充位，它只在极少数的系统中存在。

在多数硬件系统内的数据存储器中，每个存储单元的每一个比特都有对象的值有关。但是标准也照顾一些特殊的硬件系统，因为它们在表示值的时候，可能会存在填充比特。

为什么会有填充比特？当然原因是多种多样的。例如，有些数字信号处理器只有浮点类型而没有整数类型。建立在这种机器上的 C 实现只能用指数部分为零的浮点数来模拟整数运算。而为了给整数类型的比特数量凑整，可能需要一些用不到的填充比特。

再如，有些机器不支持无符号数的算术运算而只能使用符号比特是 0 的有符号数来模拟无符号数。在这种情况下，符号比特对于无符号数来说被视为填充比特。这也可能是不要求 UINT_MAX 大于 INT_MAX 的一个原因。

填充比特的存在会使同一个值有多种对象表示，但是运算符对值的操作不受填充比特的影响，在这方面对程序员是透明的，即使他并不知道填充比特的存在。

标准要求 char、signed char 和 unsigned char 类型的对象，以及无符号类型的位字段不允许存在任何填充比特。

3.18 兼容类型

在执行函数调用、用左值访问一个对象、执行参数传递时，都要求参与的操作数在类型上与对象或者函数参数相同，至少是高度相似。所谓兼容类型（compatible types），是指多个类型之间高度相似，或者它们都是相同的类型。

C 并非强类型的计算机语言，所以，同一个对象或者函数的多个类型声明不必完全相同或者完全等价。尤其是，很多时候我们需要在不完整类型和完整类型之间建立关联。在这种情况下，C89 引入了兼容类型的概念。之后，C99 引入了变长数组，变长数组和不完整的数组类型以及具有常量大小的数组都兼容。那么，两个类型怎样才算高度相似，或者说它们是兼容类型呢？

第一，如果两个类型相同，则它们是兼容类型。类型相同的情况包括：类型名相同或者等价；一方是用另一方的类型定义的别名；双方都是同一种类型的别名。比如，short 和 short int 是兼容的，因为它们是同一种类型；unsigned 和 unsigned int 是兼容的，它们也是同一种类型；const int 和 int 是不兼容的，它们是两种不同的类型。给定以下声明：

```
typedef int Integer;
int x, y;
Integer z;
```

```
void f (Integer, int);
void g (int, Integer);
```

则 x、y 和 z 的类型是兼容类型；f 和 g 的类型也是兼容类型。

第二，两个兼容的类型，加上相同的限定符（只要限定符相同即可，限定符的先后顺序并不重要）后，得到的两个新类型也是兼容类型。给定以下声明：

```
const char c1;
volatile char const c2;
char const c3;
const int d;
```

所有 char 类型都是兼容的，但它们与 int 类型都不兼容。上例中，如果没有任何限定符，c1、c2 和 c3 都彼此兼容，但实际上它们是有限定符的，因此只有 c1 和 c3 的类型是兼容的，因为这两个 char 类型具有相同的限定，但它们与 c2 互不兼容；d 的类型和其他类型都不兼容。

第三，如果两个指针所指向的类型是兼容的，而且这两个指针本身也具有相同的限定（限定符的顺序并不重要），则这两个指针类型也是兼容的。例如：

```
const char (* const p) [22], (* const q) [22];
```

其中，p 和 q 的类型是兼容类型，因为它们都是 const 限定的指针，且各自指向具有 22 个“const char 类型的元素”的数组。

下例中，pa 是指向 char *的指针，pb 是指向 const char *的指针。它们指向的类型并不兼容，故 pa 和 pb 不是两个兼容类型的指针。

```
char * * pa;
const char * * pb;
```

第四，如果两个数组类型都具有相同的常量大小，且元素类型是兼容的，那么这是两个兼容的数组类型。

这就是说，int[2]和 int[2]是兼容的数组类型；int [2][3]和 int[2][3]也是兼容的数组类型；int[2][3]和 int[3][2]不是兼容的数组类型，因为前者的元素类型是 int[3]，而后者的元素类型是 int[2]。

在下例中，a 的元素类型是 int[m]，指针 p 所指向的数组，其元素类型是 int[n]，这两个数组的元素类型兼容（原因见下面的第六条）。再加上 a 的大小是 8，p 所指向的数组大小也是 8，所以这是两个类型兼容的数组。进一步地，将表达式&a 的值（指向 int[8][m]的指针）赋给 p 是允许的。

```
int m = 2, n = 3, a [8] [m], (* p) [8] [n];
p = & a;     //可以
```

第五，两个数组，一个是常量大小而另一个不具有常量大小或未指定大小，但只要它们的元素类型是兼容的，则它们也属于兼容的数组类型。

这就是说，int[]和 int[3]是兼容的数组类型；int[][2]和 int[3][2]是兼容的数组类型，因为它们的元素类型都是 int[2]；int[][2]和 int[2][3]是不兼容的数组类型，因为前者的元素类型是 int[2]，而后者是 int[3]。

对于以下声明：

```
int a [], b [sizeof (int) * 10], c [40];
float d [], e [40];
```

a 和 b 的类型是兼容类型；a 和 c 的类型也是兼容类型；b 和 c 的类型是兼容类型的条件是 sizeof (int) == 40（注意，这是在转换期间求值，因此结果是一个常量）；d 和 e 的类型是兼容类型，除上述之外，任何两个类型都不兼容。

下例中，因为指针 p1、p2 和 p3 所指向的类型都是兼容类型，所以在指针之间赋值都是合法的。

```
int (* p1) [], (* p2) [], (* p3) [22];
/* ..... */
p2 = p3;
p1 = p2;
```

同样地，int[n][n]和 int[2][3]是兼容的数组类型，因为它们有兼容的元素类型 int[n]和 int[3]，而且只有后者是常量大小（3）。

第六，两个数组，如果都未指定大小，或者都不具有常量大小，但只要它们的元素类型兼容，则它们也是两个兼容的数组类型。

这就是说，int[]和 int[]是兼容的数组类型；int[][2]和 int[][2]也是兼容的数组类型，毕竟它们的元素类型是兼容的，都是 int[2]。再如，对于声明：

```
int n = 3, a [n], b [n + 1], c [33];
```

a 和 b 的类型是兼容类型；a 和 c 的类型也是兼容类型；b 和 c 的类型同样是兼容类型。

同样地，int[n][n]和 int[n][3]是兼容的数组类型，因为它们有兼容的元素类型 int[n]和 int[3]，而且这两个数组都不是常量大小（都是 n）。

【思考题】下面的数组 a、b，它们的类型兼容吗？

```
int m = 2, n = 3;
int a [m][2][n + 1];
int b [6][m][n];
```

在需要两个数组必须兼容的操作环境中，仅仅是类型上的兼容或者说高度相似可能还不够。如果它们的大小不同，则行为是未定义的。下例中有三个赋值操作，运算符=的右操作数是数组的，需先做数组-指针转换。然后，左右两个操作数的类型都是兼容的。但除非数组的大小满足各行注释中所标明的条件，否则，行为是未定义的。

```
void f (int m, int n)
{
    int va [m] [n], (* p) [5];
    p = va;                    //兼容，但除非 n==5，否则行为是未定义的

    m = /* ..... */, n = /* ..... */;
    int vb [m] [n] [3] [m], (* q) [n] [m] [n + 1];
    q = vb;                    //兼容，但除非 m==n+1 && m==3，否则行为是未定义的

    m = /* ..... */, n = /* ..... */;
    int (* r) [m] [3] [n], (* s) [m] [m] [5];
    r = s;                     //兼容，但除非 m==3 && n==5，否则行为是未定义的
```

```
}
```

第七，如果两个函数的返回类型是兼容的，而且它们都是用传统 K&R 风格声明的，则这两个函数的类型是兼容的。这意味着不会比较它们的参数数量和类型。

以下，f 和 g 的类型是兼容类型，但 f 的类型和 r 的类型不兼容，g 的类型和 r 的类型也不兼容，因为函数 r 的返回类型与 f 和 g 不同。

```
int f (), g ();
float r ();
```

第八，如果两个函数的声明都使用了函数原型，且它们的参数类型列表不是以省略号终止的，则只有在同时满足以下条件时，它们的类型是兼容的：

（1）它们的返回类型是兼容的；

（2）参数类型列表中的参数数量相同；

（3）参数类型列表中，相对应的参数在类型上是兼容的。

下例中，函数 f 和函数 g 的类型是兼容类型：

```
float f (float, float), g (float,float);
```

第九，如果两个函数的声明都使用了函数原型，且它们的参数类型列表也都是以省略号终止的，那么，只有在同时满足以下条件时，它们的类型是兼容的：

（1）它们的返回类型是兼容的；

（2）参数类型列表中，省略号之前的参数在数量相同；

（3）在参数类型列表中的省略号之前，对应的参数在类型上是兼容的。

给定以下函数类型声明：

```
int f (int, int);
int g (int, int);
void h (const char *, ...);
void k (const char *, ...);
```

f 和 g 的类型是兼容的，h 和 k 的类型也是兼容的，任何其他两种类型都不兼容。

需要特别提醒的是，当函数的参数是数组或者函数类型时，它们会分别被调整为指向数组首元素的指针，以及指向函数的指针。所以，判断两个参数类型是否为兼容类型或者复合类型（由两个兼容类型构建的类型，参见“复合类型”）的时候，依据的是它们调整之后的类型。如果参数的类型是限定的，也被当作无限定的版本对待。这条规则适用于本节后面的所有内容。

这就是说，对于以下声明：

```
void f (int [], int (const char *));
void g (int * restrict, int (* const) (const char *));
```

对于函数 f 而言，数组类型 int []被调整为指向其首元素的指针 int *，故函数类型 int (const char *)也被调整为指向函数的指针 int (*) (const char *)；对于函数 g 而言，其第一个参数的限定符 restrict 和第二个参数的限定符 const 被忽略。综合以上所述，f 的类型和 g 的类型是兼容类型。下面的代码用于验证这个结论：

```
void f (int [], int (const char *));
void g (int * restrict, int (* const) (const char *));
```

```
int main (void)
{
    void (* pf) ( int [], int (const char *));
    pf = g;
    /* …… */
}

void g (int * restrict a, int (* const pf) (const char *))
{
    /* …… */
}
```

第十，如果两个函数的返回类型是兼容的，一个使用了函数原型，另一个使用了传统K&R形式但没有函数体，如果前者的参数类型列表不是以省略号终止的，且每个参数的类型在应用默认参数提升（参见“默认参数提升”）后同原类型兼容（或者不变），则这两个函数的类型是兼容的。

如果是K&R形式的函数声明且没有函数体，则无法判定它的参数数量和类型，兼容性的判定只能依据原型一方进行推测。给定以下函数类型声明：

```
int f ();
int g (int, ...);
int h (float f) {/* …… */}
int r (double d) {/* …… */}
int m (float f, ...);
```

那么，f和r的类型是兼容类型，因为f的标识符列表为空，r的参数类型是double，在应用默认参数提升后还是double。除此之外，f和其他任何两个函数类型都不兼容。其实验证它们是否兼容很简单，先定义一个指针pf，它指向的类型和f的类型兼容，然后尝试用其他函数类型的指针为它赋值。

```
int (* pf) () = r;  //允许，运算符=两边的类型是兼容类型
pf = m;             //将产生诊断消息，运算符=两边的类型不兼容
```

第十一，如果两个函数的返回类型是兼容的，一个使用了函数原型，另一个使用了传统的K&R形式且带有函数体（这意味着它可能会有标识符列表和声明列表，参见“函数定义”），如果两者的参数数量相同，且后者每个标识符的类型在应用默认参数提升后，同前者相对应的参数在类型上兼容，则这两个函数的类型是兼容的。

如果K&R形式的函数声明有函数体，则可以检查它的声明列表以得到参数的类型，并和函数原型进行比对。给定以下函数类型声明：

```
int r (c, f) char c; float f; {/* …… */}
int s (i, d) int i; double d; {/* …… */}
int t (int, double);
int v (char, float);
```

```
int main (void)
{
    int (* pf1) (int, double) = r;  //允许，运算符=两边的类型是兼容类型
    int (* pf2) (char, float) = s;  //将产生诊断消息，运算符=两边的类型不兼容
    /* ..... */
}
```

那么，t 的类型和 r 的类型是兼容类型，同时 t 的类型和 s 的类型也是兼容类型，但是 v 的类型和其他类型不兼容。进一步地，指向函数的指针是否兼容，则看它们指向的函数类型是否兼容，这就是为什么 r 可以做为 pf1 的初始化器，而 s 不能是 pf2 的初始化器。

第十二，在同一个转换单元内，两个结构、联合或者枚举类型是否兼容，要依据“结构或联合指定符”、“枚举指定符”及“标记”中的相关规定进行判定。

我们知道，对于任何一个结构或者联合类型来说，它可以有多次声明，但带有内容定义（成员列表）的声明只允许出现一次，否则就是非法的，例如：

```
struct s {int a, b};
struct s {int a, b};    //非法：重定义了struct s
```

但是，如果没有标记，则具有相同内容定义的结构或者联合类型都是新的类型，而且互不兼容，如：

```
struct {int a, b} sa;
struct {int a, b} sb = {0, 1};
sa = sb;    //非法：不兼容的类型
```

因为这些原因，和其他类型不同，不同转换单元里的结构、联合和枚举引用很难实现。使用一个在其他转换单元里声明的结构、联合和枚举类型时，必须要知道它的定义，否则将无法完成任何操作。这是一对矛盾，结构、联合和枚举类型的完整声明必须出现在使用它们的转换单元里，这会导致它在每个转换单元里都是新的类型；另一方面，我们还希望它们能够实现跨转换单元的相互引用。标准 C 语言考虑到了这种情况，为了使同一程序的各个独立编译部分能够互相引用，标准将满足以下条件的、分别在不同的转换单元内声明的结构、联合或者枚举类型视为兼容类型。

（1）都是结构类型，都是联合类型，或者都是枚举类型；

（2）在它们的声明中都使用了同一标记；

（3）如果它们在各自的转换单元内都是完整类型，则：

① 成员数量相同，相对应的成员在类型上兼容；

② 相对应的成员要么不指定对齐，要么指定等价的对齐；

③ 相对应的成员要么都没有名称，要么名称相同；

④ 两个都是结构类型或者两个都是联合类型的，如果相对应的成员都是位字段，则宽度必须相同；

⑤ 两个都是枚举类型的，相对应的成员必须具有相同的枚举常量值。

例如，假定某个程序由源文件 main.c 和 pub.c 组成，且它们的内容如下。经过预处理之后，它们的内容不会有太大变化。依据上述描述，两个转换单元内的结构类型 struct stype 是

兼容类型，而两个联合类型 utype 不兼容。

◆ 源文件 main.c 的内容：

```
struct stype {int i; float f; unsigned : 5;};
union utype {int i; unsigned : 6};

struct stype f (void);

int main (void)
{
    struct stype st = f ();
    /* ..... */
}
```

◆ 源文件 pub.c 的内容：

```
struct stype {int i; float f; unsigned : 5;};
union utype {int i; unsigned : 5;};

struct stype f (void)
{
    struct stype x = {.i = 5, .f = 0.0};
    /* ..... */
    return x;
}
```

如果声明了指示同一个对象或者函数的多个标识符但它们的类型不兼容，则程序的行为是未定义的，后果难以预料。

这就是说，假定某个程序由源文件 intf.c 和 fltf.c 组成，且它们的内容如下。

◆ 源文件 intf.c 的内容：

```
int d, f (void);

int main (void)
{
    d = f ();
    /* ..... */
}
```

◆ 源文件 fltf.c 的内容：

```
float d = 0.0f;

float f (void)
{
    return 2.0f;
```

```
}
```

则对标识符 d 的两次声明指示同一个对象，但它们的类型并不兼容；同样，对函数 f 的两次声明也指示同一个函数，但它们的类型也不兼容。在这种情况下，程序的转换会顺利进行，但代码的行为是未定义的。

两个类型兼容并不能保证对它们的操作是有意义的、安全的。如下例所示，pf 的类型和 &f 的类型是兼容类型，因为它们都是指向同一个函数类型的指针。正是因为这样，才可以将后者的值赋给前者。但是，从函数 f 的定义来看，它需要两个 int 类型的参数，而不是一个 double 类型的参数。因为函数的声明是传统形式的，所以 C 实现不会阻止用户传递一个 double 类型的参数（2.0），但这种行为的后果是难以预料的。

```
int f ();

int main (void)
{
    int (* pf) () = & f;
    return (* pf) (2.0);          //未定义行为
}

int f (i, j)
int i, j;
{
    return i - j ? i : j;
}
```

3.19 复合类型

除了需要兼容类型来完成很多操作外，有时也需要在兼容类型的基础上建立它们的公共类型。复合类型的概念是和兼容类型的概念一起，从 C89 开始引入的。对于 C 实现来说，复合类型的第一个用途是在编译程序的时候将那些指示同一个对象或者函数的标识符链接在一起，这个符号（标识符）的类型是来自它各个声明的复合类型。

复合类型的第二个用途和条件运算符有关。条件运算符?:的结果类型可能是其后两个操作数类型的公共类型，也就是它们的复合类型。

可以在两个兼容类型的基础上构建出新的公共类型，这个新的类型也同构建它的两个类型兼容，这个新的类型称为复合类型（composite types）。

需要特别指出的是，构建复合类型的过程可以是递归的，复合类型可以在其他复合类型的基础上构建。

首先，可以构建两个数组类型 A、B 的复合类型，前提是它们是兼容的。其方法是，构建 A、B 元素类型的复合类型，用这个复合类型构建的数组类型就是 A 和 B 的复合类型。

（1）如果两个数组类型 A 和 B 是兼容的，且其中一个具有常量大小，那么，这意味着另

一个数组也具有相同的常量大小，或者未指定大小，又或者指定的大小不是常量。在这种情况下，它们的复合类型是具有该常量大小的数组，其元素类型是 A、B 元素类型的复合类型。

这意味着，int[]和 int[3]的复合类型是 int[3]。给定声明

```
int n = 8;
```

那么 int[22]和 int[n]是兼容类型，它们的复合类型是 int[22]；char[][3]和 char[6][n]也是兼容类型，它们的复合类型是 char[6][3]。也就是说，它们的元素类型分别是 char[3]和 char[n]，是兼容类型且形成复合元素类型 char[3]。

再比如下面这个合法的例子，条件表达式的结果是一个指针，但这个指针所指向的类型是必须是一个复合类型，由其后两个操作数&a 和&b 所指向的类型构建而成。

```
int (* f (int n)) [3]
{
    static int a [3], b [3];
    return n == 0 ? & a : & b;
}
```

（2）如果两个数组类型 A 和 B 是兼容的，但都未指定大小，则它们的复合类型依然是未指定大小的数组，其元素类型是 A、B 元素类型的复合类型；进一步地，有一方是变长数组的，复合类型也是未指定大小的变长数组。

下例中，条件表达式的结果类型是参数 a 和 b 的类型的复合类型，该复合类型也是未指定大小的数组。

```
extern int a [], b [];

int f (int n)
{
    return (n == 0 ? a : b) [0];
}
```

给定两个声明：

```
int f (int a [], int b [*]);
int f (int a [*], int b []);
```

则它们的复合类型可以是

```
int f (int a [*], int b [*]);
```

也可以是

```
int f (int a [], int b []);
```

（3）在其他情况下，数组类型 A、B 中至少有一个是变长数组。如果它们都提供了用于指定大小的表达式，则 C 实现通常只求值其中一个表达式，用那个表达式的值做为复合类型的大小。但这很容易出现问题，因此行为是未定义的。

如果 A、B 中有一个是未指定大小的数组，另一个是变长数组，提供了用于指定大小的表达式且求值这个表达式，则复合类型是变长数组，其大小是那个表达式的值。

下例中，条件表达式?:的后两个操作数是指针，因此结果类型是指向后两个操作数类型的复合类型的指针。在这里，C 实现将陷入两难：一方面，条件运算符只根据第一个操作数的

结果求值后两个操作数中的一个，而不是都求值；另一方面，因为第三个操作数所指向的数组未指定大小，所以构建复合类型时必须参考第二个操作数所指向的数组。但如果不求值（调用）printf 函数（printf 函数返回本次写到标准输出的字符个数，在这里被用于指定变长数组的大小），则无法做到这一点，而不管 b 的值是什么。

```
int f (int b, void * pv, void * qv)
{
        return (b
                ?
                (int (*) [printf ("Eval")]) pv
                :
                (int (*) []) qv
               ) [0] [0];
}
```

在这种情况下，有些 C 实现将无法正常完成程序转换（在编者所用的 Clang 版本上直接导致编译器崩溃），而另一些 C 实现不管 b 的值是什么，都求值（调用）printf 函数。

当然，如果将程序修改一下，则不存在上述问题。

```
int f (int b, void * pv, void * qv)
{
        return (b
                ?
                (int (*) [printf ("Eval")]) pv
                :
                (int (*) [8]) qv
               ) [0] [0];
}
```

这是因为条件表达式的第三个操作数是指向 int[8]的指针，即使 b 的值为 0，它也不需要求值 printf 表达式。

其次，两个互相兼容的函数类型也可以构建它们的复合类型，且这个构建过程也是递归进行的。

（1）如果一个函数使用原型，另一个使用传统 K&R 形式，那么它们的复合类型是一个函数原型；

（2）相反，如果这两个函数类型都使用了原型，则它们的复合类型依然是函数原型，且这个复合类型的每个参数类型都是相对应的两个原始参数类型的复合类型。

例如，对于以下声明：

```
int f (), g (int, double)
```

f 和 g 的类型是兼容类型（应用前面讲过的规则）。再根据指针类型兼容的规则，以下两个指针类型是兼容类型。

```
int (*) ()
int (*) (int, double);
```

再比如，给定以下两个兼容的函数声明。

```
void m (int (*) ());
void n (int (*) (int, double));
```

则以下函数原型是上述两个兼容类型的复合类型。

```
void k (int (*) (int, double));
```

3.20 类型转换

出于各种原因，往往需要将某个操作数的值从一种类型转换为另一种类型。转换可能是自动进行的，这称为隐式转换（implicit conversion）；也可能需要用转型表达式显式地强制进行。

从一种类型转换为另一种类型，可能涉及表示方法的改变，但如果是在兼容类型之间进行转换，则不会改变值和值的对象表示（具体可参见“对象表示”）。

了解类型转换在什么时候发生，以及如何进行，对程序的编写者很重要。比如下面的例子，对象 x 具有 int 类型的最大值，x 加 3 的值不能被 int 类型的对象容纳，但能够被 long long int 类型的对象容纳。

```
# include <limits.h>

void f (void)
{
    int x = INT_MAX;
    long long y = x + 3;
    /* ..... */
}
```

既然如此，有人就可能觉得对象 y 一定能够得到正确的结果。事实上，因为 x 和 3 的类型都是 int，那么表达式 x+3 的结果也是 int。在将这个结果转换为 long long int 类型的值并赋给对象 y 之前，表达式 x+3 的结果已经因无法用 int 类型来表示而被截短为不正确的值了。

不要做愚蠢的事，下面的做法于事无补：

```
long long y = (long long) (x + 3) ;
```

至于原因，表达式 x+3 的结果类型是 int，在将表达式 x+3 的结果从 int 类型强制转换为 long long int 类型的值之前，就已经因无法用 int 类型来表示而被截短为不正确的值了。

要想得到正确的结果，你可以选择以下三种方法中的任何一种：

```
long long y = (long long) x + (long long) 3;
long long y = (long long) x + 3;
long long y = x + (long long) 3;
```

不管采用哪一种方式，运算符+的两个操作数都将通过常规算术转换提升到 long long int 类型，并且这也是运算符+的结果类型。参见“常规算术转换”。

3.20.1 标量-_Bool 转换

可以将任何标量类型的值转换为_Bool 类型。若标量类型的值为 0，则结果为 0；否则，转换的结果为 1。特别地，对于指针类型到_Bool 类型的转换，如果是空指针，则转换的结果是 0，否则是 1。例如：

```
char * f (void), g (_Bool);
_Bool b1 = 0.01, b2 = 33, b3 = f ();
g (f ());
```

其中，b1 的值是 1；b2 的值是 1；b3 的值取决于函数调用的返回值，如果返回的是空指针，则 b3 为 0，否则为 1。

3.20.2 整数-整数转换

_Bool 类型也是整数类型，但它比较特殊，将其他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其他整数类型时，如果这个值可以用新的类型来表示，则转换后的值同原值相比保持不变。例如：

```
short s = -1;
signed char c = s;
```

其中，-1 的类型是 int，short 类型也可以表示这个值，所以-1 从 int 类型转换为 short 类型后不变；同样地，short 类型的值-1 也能被 signed char 类型的对象容纳，所以-1 从 short 类型转换为 signed char 类型后也不变。

再如：

```
void f (_Bool b, enum E e)
{
    long i = b, j = e;
    /* ..... */
}
```

这里是将_Bool 类型和枚举类型的值转换为 long int 类型，它们可以正常进行转换，没有问题。

但是，将一种整数类型的值转换为其他整数类型时，如果无法用新的类型来表示，该怎么办呢？这要看新类型是无符号整数类型还是有符号整数类型。

（1）将任何整数类型的值转换为非_Bool 的其他无符号整数类型时，如果这个值无法用新类型表示，则转换的方法是将这个值重复地加上或者减去比新类型的最大值大 1 的数，直到结果可以用新类型来表示。例如：

```
unsigned char c = -1;
```

这里，一个负值是不可能用无符号整数类型来表示的。假定 sizeof (unsigned char)==8，那么 unsigned char 的最大值是 2^8-1，即 255；比这个最大值大 1 的数是 256，所以，-1 加上 256

等于255，这个值位于unsigned char类型可以表示的值的范围之内，是转换后的值。

（2）将一个整数类型的值转换为任何其他有符号整数类型时，如果这个值无法用新类型表示，则转换的结果将取决于C实现。下例中，unsigned int类型的最大值UINT_MAX肯定不能用int类型表示，在编者的机器上，转换后的结果是一个负值。

```
# include <limits.h>
/* ..... */
int i = UINT_MAX;
```

在有些机器上，当无法将一个整数类型的值转换为另一个有符号整数类型时，可能会引发一个表示异常的信号。

3.20.3 实浮点-整数转换

将一个实浮点类型的有限值转换为_Bool类型时，零值转换为0，而非零值转换为1；将一个实浮点类型的值转换为除_Bool之外的其他整数类型时，小数部分将被丢弃。但是，丢弃小数部分之后，剩余的整数部分可能无法用新类型来表示，在这种情况下，程序的行为是未定义的。实浮点类型的值在截尾后太大，无法用指定的整数类型来表示，或者实浮点类型的值为负，但要用无符号整数类型来表示，等等，都属于这种情况。例如：

```
unsigned i = 2.2f, j = -8.8;
```

其中，对象i的值是将浮点常量2.2f的值截尾后得到的整数2，但对象j的值无法预测。

将一个整数类型的值转换为实浮点类型时，可能会出现3种情况：①此值能够用新类型精确地表示，此时，转换后的值同原值相比保持不变；②此值可以用新类型来表示，但不能精确地表示，此时，转换后的值可能比原值稍大或稍小，具体如何选择取决于C实现；③转换后的值超出了新类型所能够表示的值的范围，此时，行为是未定义的。

3.20.4 实浮点-实浮点转换

和整数类型到实浮点类型的转换类似，将一个实浮点类型的值转换为另一个实浮点类型时，可能会出现3种情况：①此值能够用新类型精确地表示，此时，转换后的值同原值相比保持不变；②此值可以用新类型来表示，但不能精确地表示，此时，转换后的值可能比原值稍大或稍小，具体如何选择，取决于C实现；③转换后的值超出了新类型所能够表示的值的范围，此时，行为是未定义的。

3.20.5 复数-复数转换

其基本原则只有一条：将一个复数类型的值转换为另一个复数类型时，实部和虚部要分别进行转换。也就是说，旧类型的实部转换为新类型的实部，旧类型的虚部转换为新类型的虚部。因为复数类型的实部和虚部都是实数（实浮点类型和整数类型），所以，实部和虚部的转换要采用“实浮点一实浮点转换”和“整数一整数转换”的规则。

3.20.6 实数-复数转换

将一个实数类型的值转换为复数类型时，将实数类型的值转换为复数类型的实部（根据

两者的具体类型选用本节中列出的其他转换方式）；再将复数类型值的虚部设为正零或者无符号零。

反之，将一个复数类型的值转换为实数类型时，复数类型值的虚部被丢弃，然后将复数类型实部的值转换为实数类型（根据两者的具体类型选用本节中列出的其他转换方式）。

3.20.7 左值转换

我们知道，左值是潜在地指示对象的表达式。当左值出现在某些特定的位置时，所表达的意思是需要访问它所指示的对象以获得一个值。例如：

```
int x = 1, y, f (int);
y = x + 3;
f (y);
```

上例中，只关注表达式 y=x+3 和 f(y)。前者，x 是一个左值，它出现在赋值运算符=的右边，需要访问它所指示的对象以获得一个值，这样才能完成相加操作；后者，y 是一个左值，需要访问它所指示的对象以获得一个值，以便传递给函数 f。

左值转换（lvalue conversion）是将一个左值转换（替代）为它所指示的那个对象的值。但是，如果左值的类型是数组，或者是 sizeof、++、--和一元&运算符的操作数，又或者是=和.运算符的左操作数，则不执行左值转换。

因此，在下例中的 E1 中，运算符=右边的表达式 i 将执行左值转换，被转换为它所指示的对象的存储值，并且和 1 相加；而在 E2 中，运算符=右边的 i 同样要进行左值转换，在运算符=的左边，左值*(p+1)本身不会进行左值转换，但它的子表达式 p 要进行左值转换，转换为指针类型的值，然后和整型常量 1（需进行适当的类型转换）相加。

```
int i = 0, a [2] = {1, 0}, * p = a;
i = i + 1;          //E1
* (p + 1) = i;      //E2
```

左值转换后得到的不再是左值。进一步地，若左值的类型是 q 限定的类型 T，或者是原子的类型 T，则左值转换后，值的类型是 T；否则，值的类型和左值的类型相同。例如：

```
const int i = 0;    //i 的类型为 const int
int j = i;          //左值 i 在转换后的类型为 int
```

左值可能并不指示一个有效的对象，或者不再指示一个有效的对象。在这种情况下，左值转换的行为可能是未定义的，程序运行的结果难以预料。如下例所示，为了初始化对象 x，需要对 p 进行左值转换以得到一个指针类型的值，但 p 此时并未指定一个有效的对象。

```
char c = 0, * p = & c;
p ++;
int x = * p;
```

3.20.8 数组-指针转换

如果一个数组类型的表达式是 sizeof、一元&运算符的操作数，或者是以字面串的形式出现在声明中的初始化器部分（请参见“初始化器”），则不发生任何转换；否则，它会被转换为指向其首元素的指针。

另外需要注意的是，转换后得到结果不是左值。下面是第一个示例：

```
char * a [10], * * p;
p = a;
```

这里，a 是数组，其元素类型是指针（char *），p 是指向指针的指针。之所以能将 a 赋给 p，是因为，数组 a 自动转换为指向其首元素的指针，因数组 a 的元素类型是 char *，故转换后的类型是指向 char *的指针 char * *。

下面是第二个示例：

```
int a [2], * p, (* pa) [];

* a = 0;                    //S1
p = ++ a;                  //S2
p = a + 1;                 //S3
pa = & a;                  //S4
* p = sizeof a;            //S5

char x [] = "China.\n";    //D1
char * y = "France.\n";    //D2
char z = "British.\n" [0]; //D3
```

其中，a 被声明为数组类型，在 S1 中，数组 a 被转换为指向数组首元素的指针，类型为 int *。该指针被一元*运算符解除引用后，指示数组的第一个元素。

在 S2 中，数组 a 依然会被转换为指针。但是，数组被转换为指针后不是左值，但++运算符要求它的操作数是一个左值，故 S2 是非法的。

同样，在 S3 中，数组 a 被转换为指针，然后参与指针与整型常量的运算，这是合法的表达式。

在 S4 中，数组 a 是一元&运算符的操作数，因此，数组本身不会被转换为指针，而是直接作为一元&运算符的操作数。当然，表达式&a 的结果是一个指针，类型是“指向数组的指针”，即 int (*) [2]。类似的事情发生在 S5，数组不被转换为指针，而是直接作为 sizeof 运算符的操作数，sizeof 运算符的结果是数组的大小。

就像 D1 那样，当字面串用于初始化一个数组时，它不会被转换为指针。但是，在 D2 中，它用于给一个指针赋值。在这种情况下，字面串会用于初始化一个不可见的静态数组，同时，该静态数组被转换为指针，指向其第一个元素。实际上，其等效于：

```
char * y = & "France.\n" [0];
```

如果想得到“指向数组的指针”，则应该使用如下代码。

```
char (* py) [] = & "France.\n";
```

D3 和 D2 相比只有一点是类似的：字面串被转换为指向数组首元素的指针。但是，D3 使用下标运算符[]得到数组第一个元素的值。既然字面串被转换为指向数组首元素的指针，那么，使用下面的方法也可以达到相同的目的：

```
char z = * "British.\n";
```

有人问对于声明

```
char * p [3] = {"silly", "foolish", "stupid"};
```

为什么

```
p ++;
```

是非法的？

当然，它肯定是非法的，因为 p 是数组，它会被转换为指向数组首元素的指针（因为其元素类型是 char *，所以转换后的类型是 char * *），并且不再是一个左值，而是一个值。但是，后缀运算符++要求它的操作数是左值。

后来此人又问为什么在作为函数参数时，它又合法了（如下列代码所示）？

```
void f (char * p [])
{
    p ++;          //允许
    /* ..... */
}
```

这是因为当 p 作为函数参数时，它指示一个指针类型的对象。这是一个通过声明得到的指针，而不是转换来的。

又如，给定声明：

```
int a [22], * p = a, (* q) [22] = & a;
```

则以下表达式都是正确的：

```
* a = 1;
* p = 1;
* * q = 1;
```

以上第一行，a 的类型是 int [22]，被自动转换为指向其首元素的指针（int *），接着解除引用，得到一个 int 类型的对象，也就是数组 a 的首元素；

以上第二行，p 是指向 int 的指针，并指向数组 a 的首元素。解除引用后得到一个 int 类型的对象，也就是数组 a 的首元素；

以上第三行，q 是指向数组的指针（int (*) [22]），解除引用后得到一个数组（int [22]），并自动转换为指向其首元素的指针（int *），解除引用后得到数组 a 的首元素。

再如：

```
struct t {int i; char a [33];} at [3];
* (at [0].a) = 'x';          //S1
char * p = at -> a;          //S2
* (at -> a + 1) = 'y';       //S3
```

其中，at 是一个结构的数组。在 S1 中，表达式 at[0]的类型是 struct t；表达式 at[0].a 的类型是 char [33]，因为是数组类型，故转换为指向 char 的指针，且指向数组第一个元素。然后，运算符*对该指针解除引用，并得到一个左值。

在 S2 中，表达式 at->a 的类型是数组 char[33]，被转换为指向 char 的指针，且指向数组的第一个元素。这个指针可以赋给对象 p。

在 S3 中，表达式 at->a 的类型是数组 char[33]，被转换为指向 char 的指针，且指向数组的第一个元素。然后，表达式 at->a+1 是指向数组的下一个元素。最后，运算符*将该指针解

除引用，并得到一个左值。

数组转换为指针的确切说法是“数组转换为指针类型的值”，而不是转换为一个指针类型的对象，这一点必须明确。

另一个必须明确的事情是，数组和指针在本质上不是一回事，将这两者放在一起讨论，在很多时候是没有意义的，尽管前者在某些情况下会隐式地转换为后者。数组和指针在本质上就是两种东西，不能将它们等同起来。

上网的时候，经常看到有些初学者提一些看起来简单，但非常有价值的问题，下面就是一例。给定声明

```
int a [3] [2];
```

为什么下面的语句会输出两个相同的地址：

```
printf ("%p, %p", (void *) (a + 1), (void *) * (a + 1));
```

就像本书前言中所说，不用类型系统，有些东西仅凭地址无法解释清楚。首先我们要明确一点：如果一个数组类型的表达式不是 sizeof 和一元&的操作数，且不是用于初始化一个数组，则它会被转换为指向数组首元素的指针。接下来：

第一，a 是具有 3 个元素的数组，元素的类型是 int [2]；

第二，在表达式 a+1 中，a 转换为指向其首元素的指针。元素的类型是 int [2]，所以是转换为指向 int [2]的指针 int (*) [2]（实际上指向的位置是原数组的元素 a [0]或者 a [0][0]）。加 1 后，指针的类型不变，还是 int (*) [2]，但指向的位置变了（指向下一个 int [2]，实际上指向的位置是原数组的元素 a [1]或者 a [1][0]）；

第三，在表达式*(a+1) 中，a 转换为指向其首元素的指针，元素的类型是 int [2]，所以是转换为指向 int [2] 的指针 int (*) [2]（实际上指向的位置是原数组的元素 a [0]或者 a [0][0]）。加 1 后，指针的类型不变，还是 int (*) [2]，但指向的位置变了（指向下一个 int [2]，实际上指向的位置是原数组的元素 a [1]或者 a [1][0] ）。解引用后，我们从指针 int (*) [2]得到数组 int [2]本身（实际上这个数组对应着原数组的元素 a [1]或者 a [1][0]），接着又被转换为指向其首元素的指针 int *（尽管指针的类型变了，但实际上指向的位置是原数组的元素 a [1]或者 a [1][0]）。

如图 3-8 所示，尽管两个表达式的类型不同，是不同类型的指针，但它们实际上指向数组 a 的同一个位置，所以会输出相同的地址。

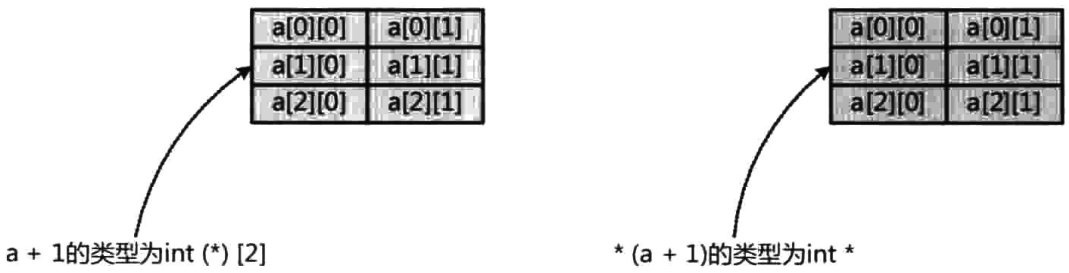


图 3-8　不同类型的指针指向数组内的同一个位置

可以使用存储类指定符 register 声明一个数组。如果 C 实现认为可以做到这一点，那么它可以将数组放在寄存器中。如果一个数组在声明时使用了存储类指定符 register，则不允许将它转换为指针，这是明显且很容易理解的，同时也表明数组和指针实际上并不是等价的。

3.20.9 函数指示符-指针转换

函数指示符是指示函数的表达式，或者说是函数类型的表达式（可参见“函数指示符”），它不能作为 sizeof 运算符的操作数。

如果函数指示符不是 sizeof 的操作数，也不是一元&运算符的操作数，那么它会被转换为指向函数的指针。换句话说，如果函数指示符的类型是“返回 T 的函数”，则转换后的类型是“指向‘返回 T 的函数’的指针”。例如：

```
int f (void), (* pf) (void), (* apf [5]) (int);

f ();                          //S1
(* f) ();                     //S2
(& f) ();                     //S3
(* ( & f)) ();                //S4
pf = f;                        //S5
pf = & f;                      //S6
pf ();                         //S7
(* pf) ();                    //S8
(* apf [0]) (10086);          //S9：(*apf[0])是函数指示符
```

以上所有的语句都是合法的。在 S1 中，f 是函数指示符，被转换为指向“返回类型为 int 的”函数的指针，并调用该函数；在 S2 中，函数指示符 f 同样被转换为指向函数的指针，被一元*运算符解除引用后，又成为函数指示符，然后被转换为指向函数的指针。

在 S3 中，f 是函数指示符。因为它是一元&运算符的操作数，所以它不会转换为指针，它本身依然是函数指示符，但&运算符的结果是指向函数的指针。这就是说，尽管 f 不会被转换为指针，但表达式&f 的结果是指针。

在 S4 中，函数指示符 f 不会被转换为指针，但表达式&f 的结果是指向函数的指针。然后，该指针又被一元*运算符解除引用，还原成函数指示符。最后，因为要调用这个函数，所以函数指示符再被转换为指向函数的指针。

S5、S6、S7、S8 和 S9 的情况也类似，请读者自行分析（注意，指针 pf 解除引用后是函数指示符）。

3.20.10 指针-void 指针转换

指向任何一个对象类型的指针都可以转换为指向 void 的指针（void *）。C 语言可以确保这个指针转换为原来的指针类型后完全不变。

这就是说，对于以下代码：

```
int i = 0, * p = & i, * q;
void * pv = p;          //从 int *转换为 void *
q = pv;                 //从 void *转换回 int *
if (p == q) {/* …… */}
```

可以肯定，p==q 是成立的。

但是，以上黑体部分的表述并不意味着将指向一种对象类型的指针转换为指向 void 的指

针，再将后者转换为指向另一种不同对象类型的指针是没有问题的，例如：

```
char c = 0;
void * pv = & c;
* (long long int *) pv = 1;
```

首先，char 类型的对象对于它自己在存储器中的位置并不挑剔，因为它的对齐是 1。但是，long long int 类型的对齐要求通常比 char 类型更严格。从任何对象类型的指针到 void *类型的转换不会有任何问题；从 void *类型转换到任何对象类型的指针也没有问题。但问题在于，转换后的结果类型是 long long int *，用这个指针访问 long long int 类型的对象没有问题，但实际上它要访问的是 char 类型的对象。如果这个 char 类型的对象并非恰好位于 long long int 类型所要求的字节地址上，对它解除引用时，程序的行为是不可预料的。

值得注意的是，void 类型是对象类型，指向 void 的指针是指向对象类型的指针。所以，不允许将指向函数类型的指针转换为指向 void 的指针；也不允许从指向 void 的指针转换为指向函数的指针。

在下例中，企图将指向函数的指针（由函数指示符 printf 转换而来）转换为指向 void 的指针（p），然后再转换为指向 int 类型的指针（q）。尽管最后又重新转换为原先的函数指针，但这种做法是禁止的。

```
# include <stdio.h>

void f (void)
{
    void * p = printf;
    int * q = p;
    ((int (*) (const char * restrict, ...)) q) ("Possible right ?\n");
}
```

3.20.11 整数-指针转换

从整数类型到指针类型的转换包括以下几个方面：

第一，值为 0 的整型常量表达式可以转换为空指针常量，参见“空指针常量”。

第二，可以将空指针常量转换为任意类型的空指针，参见“空指针”。

第三，可以在任意两种类型的空指针之间进行转换。尽管不同类型的空指针可能具有不同的对象表示，但它们能够被正确识别并完成转换。而且，将任何两个空指针放在一起进行比较时都是相等的。因此，以下两个空指针在比较时是相等的。

```
# define NFPTR (float *) 0
if ((int *) NULL == (int *) NFPTR) {/* …… */}
```

上述两行代码实际上和下面这一行完全等价：

```
if ((int *) NULL == (int *) (float *) 0) {/* …… */}
```

第四，允许将任何一种整数类型的值转换为任何一种指针类型的值。但是除非从值为 0 的整型常量表达式转换（此时将得到空指针常量和空指针），从其他整数值（包括值为 0 的非常量表达式）转换时，转换后的结果取决于 C 的实现且不可移植。转换后得到的指针可能对

齐不正确，也可能并不指示一个有效的对象或者函数，这个指针值还可能具有自陷表示（参见“自陷表示”）。

在下例中的第一行，尽管对象 n 的值是 0，但 n 并不是一个整型常量表达式，所以指针 pn 并不是一个 NULL 指针。

```
int n = 0, * pn = (int *) n;
* (long long *) 777 = 0;
```

在编者的机器上，long long int 类型对齐于 8，这种类型的对象只能位于能够被 8 整除的那些字节地址上，(long long *) 777 这样的指针则不符合要求。此外，它也不指示一个合法的、已正确定义的被引用实体。

第五，任何指针类型的值也可以转换为任何整数类型的值。但是转换的结果也是由 C 实现定义的，不可移植。特别地，即使是空指针，转换后的结果也未必是整数值 0，这和执行环境的地址结构有关。也有可能发生转换后的结果不能被整数类型表示的情况。

例如，在有些机器上，指针类型的长度可能会超过最长的整数类型，所以将指针类型转换为整数类型时，转换的结果可能会超过任何整数类型的值的范围。例如：

```
char * ptr;
/* …… */
unsigned int n = (unsigned int) ptr;
```

上例中，假定指针 ptr 需要的宽度是 64 个比特，而且 unsigned int 的宽度是 32 个比特，则程序的行为是未定义的。

朋友二玉给我发来一段代码，他想知道这里面是不是有问题：

```
# include <stdio.h>

int main (void)
{
    return printf ("%X\n", main);
}
```

这个小程序的意图很明显，表达式 main 在这里将执行从函数指示符到指针的转换（参见“函数指示符-指针转换”），然后再将这个指针传递给 printf 函数。从第一章里我们知道，转换模板“%X”要求一个 unsigned int 类型的参数，但是在这里，指针的宽度未必会和 unsigned int 类型的宽度相同。在调用格式化输入/输出函数时，如果给出的实参不够，或者实参的类型不正确，则程序的行为是未定义的。

那么，也许我们可以这样做：

```
printf ("%X\n", (unsigned int) main);
```

当然，这里不会有什么大问题，但如上所述，从指向 main 的指针到整数类型的转换未必会得到你想要的结果。

在第一章里，我们曾经用过转换模板“%p”，也许我们可以这样做：

```
printf ("%p\n", (void *) main);
```

但这同样有问题：指向 void 的指针被称为通用对象指针，标准允许将指向任何对象类型的指针转换为指向 void 的指针，也允许将指向 void 的指针转换为指向任何对象类型的指针。

但是，将一个指向函数的指针转换为指向 void 的指针是否会丢失信息，这取决于具体的硬件架构和 C 实现。

3.20.12 指针-指针转换

可以将某个指针类型的值转换为另一个不同指针类型的值。因为指针类型是以它指向的类型为特征的，所以指向某个对象类型的指针可以转换为指向另一个不同对象类型的指针。

在不同的指针类型之间转换时，最大的问题是对齐。一个指向对象类型的指针，它的值在转换成指向另一种对象类型的指针后，可能无法按新类型正确对齐。在这种情况下，程序的行为是未定义的，不可预料。反之，如果转换后的对齐没有问题，则转换后的结果可以转换回原来的指针类型，并且和原来的指针在比较时是相等的。

在下例中，char 类型的对象 c 具有最弱的对齐要求，而 int 类型的对象则不然。将指向 char 的指针&c 转换为指向 int 的指针 pi，可能出现对齐问题。例如，如果 int 类型的对象要求它的地址必须能够被 4 整除，而对象 c 当前的地址恰恰不能被 4 整除，则转换后肯定不是正确对齐的。

```
char c = 0;
int * pi = (int *) & c;      //转换可能导致丢失信息
char * pc = (char *) pi;     //转换回原来的类型
if (pc == & c) {/* …… */} //有不匹配的可能
```

比较好的改正方案是先转换值的类型，再转换指针的类型。

```
char c = 0;
int i = c;
int * pi = & i;
if (pi == & i) {/* …… */}
```

还有一种方法是使用_Alignas 关键字，来强制 char 类型的对象具有像 int 类型一样的对齐。

```
_Alignas (int) char c = 0;
int * pi = (int *) & c;
char * pc = (char *) pi;
if (pc == & c) {/* …… */}
```

这里，将 c 声明为和 int 类型具有相同的对齐要求。这样，将指向 c 的指针转换为 int *类型不会有任何问题，再转换回原来的类型也不会有任何问题。

在下面这个例子中，指向结构的指针被转换为指向数组的指针 pa 和指向 unsigned char 类型的指针 pc。实际上，pc 也是指向数组首元素的指针。

```
void f (struct t * pt)
{
    unsigned char (* pa) [] = (unsigned char (*) []) pt,\
             * pc = (unsigned char *) pt;
    /* …… */
}
```

此时，如果想访问数组的第一个元素（实际上也是访问结构对象的第一个字节），pa 和

pc 的用法是不同的：

```
unsigned char c1 = (* pa) [0],\
               c2 = * pc;              //或者 c2=pc[0];
```

因为 pa 是指向数组的指针，所以表达式*pa 的结果类型是数组。

指向字符类型的指针（char *）和指向 void 类型的指针（void *）是同等对待的。在关键字 void 被引入 C 语言之前，char *类型的角色就是现在 void *类型的角色。所以，如果任何指向对象的指针类型都可以被转换为 void *类型，那么它们也可以安全地转换为 char *类型。

如果有一个指向对象 X 的指针 P，则可以将 P 的值转换为指向字符类型的指针，而且可以保证转换后得到的指针指向对象 X 的低字节部分。

进一步地，可以用加一操作或者++运算符递增这个指针。每递增一次，都会得到一个新的指针值，新指针指向对象 X 的剩余字节。当然，递增操作要以对象 X 的大小为限。

也就是说，对于以下代码：

```
void f (char *);

double d = 1.0;
char * pc = (char *) & d;
f (pc);
f (++ pc);
```

将指向对象 d 的指针（&d）转换为指向 char 的指针（pc）是安全的，而且 pc 指向 d 的低字节部分。利用这个特性，可以探查和检验特定类型的值的对象表示，例如：

```
# include <stdio.h>
# include <stddef.h>

void f (void)
{
    double d = 3.14;
    printf ("The double %.2f(%a) is:\n", d, d);
    for (size_t n = 0; n < sizeof d; ++ n)
        printf ("0x%02x ", ((unsigned char *) & d) [n]);
}
```

C 标准禁止在指向对象的指针和指向函数的指针之间进行转换，但允许从指向一种函数类型的指针转换为指向另一种函数类型的指针；也允许从指向一种对象类型的指针转换为指向另一种对象类型的指针。同时，如果将转换后的结果再转换回原来的指针类型，则它和原指针一定相等。例如：

```
int f (void)
{
    int (* pf) (void) = f;
    void (* pfv) (int, int) = (void (*) (int, int)) pf;    //S1
    int (* pf0) (void) = (int (*) (void)) pfv;             //S2
```

```
    return pf0 == pf;
}
```

其中，pf 的类型是指向函数的指针，函数的返回类型是 int，且不接收任何参数。可以令它指向函数 f，因为它们的类型是相同的。

pfv 的类型也是指向函数的指针，函数的返回类型是 void，且接收两个 int 类型的参数。尽管 pf 和 pfv 所指向的函数类型不同，但依然可以将 pf 转换为 pfv 的类型（语句 S1），再转换回原来的类型（语句 S2）后赋给 pf0，并保证 pf0 和 pf 在比较时是相等的。

将一个指向函数的指针转换为指向另一种函数类型的指针后，新指针所指向的函数类型必须同原指针所指向的函数类型兼容（参见“兼容类型”，以了解两个函数类型互相兼容的条件）。否则，用新指针做函数调用操作时，程序的行为是未定义的，结果不可预料。

例如：

```
void f (int i) {/* …… */}

void g (void)
{
    int (* pf) (const char *) = (int (*) (const char *)) f;
    pf ("hello, world.\n");
}
```

其中，函数 f 的返回类型是 void，接收一个 int 类型的参数。但是，指针 pf 所指向的函数类型却不同，返回类型是 void，接收一个 const char *类型的参数。将前一个函数类型的指针转换为后一个函数类型的指针是合法的，但是，用转换后的指针进行函数调用很容易导致程序在执行时出现问题。

如果一个指针指向无限定（不带限定符）的类型 T，那么可以将它转换为指向“限定的类型 T”的指针。同时，可以保证原先那个指针的值和新指针的值进行比较时，一定是相等的。

在下例中，指针 p 指向的类型无任何限定符，即指向 int 类型。在它传递给函数 f 时，被转换为指向“const 和 volatile 限定的 int”的指针。

```
void f (const volatile int *);

int n = 0, *p = & n;
f (p);
const int * q = (const int *) p;    //S1
if (p == q) {/* …… */}               //S2
```

除了转换为指向 const volatile int 类型的指针外，在上面的 S1 中，指针 p 又从指向 int 的指针转换为指向 const int 的指针。在 S2 中，转换前和转换后的指针在比较时是相等的。再比如下面的例子，if 语句的控制表达式将始终求值为 1：

```
char c;
if (& c == (const volatile char *) & c) {/* …… */}
```

如果存在着一个指向结构对象的指针，则可以将该指针加以适当的转换，使之成为一个

新的、指向结构对象第一个成员的指针（如果第一个成员是位字段，则指向的是该位字段驻留的那个存储单元）。

反之，如果存在着一个指向结构对象第一个成员（或者该成员所驻留的存储单元，如果它是位字段）的指针，则可以将该指针加以适当的转换，使之成为一个新的、指向包含该成员的那个结构对象的指针。下面的两个例子很好地解释了这些特点。先来看第一个例子。

```
# include <stdio.h>

void f (void)
{
    struct s {int i:5; double d;} s = {7, 7.5}, * ps = & s;

    int * pi = (int *) ps;                    //D1
    * pi = 8;
    printf ("%d, %f\n", s.i, s.d);

    ps = (struct s *) pi;                     //D2
    ps->d = 8.0;
    printf ("%d, %f\n", s.i, s.d);
}
```

其中，ps 是指向结构类型的指针，在 D1 处转换为指向 int 类型的指针，这样它可以指向该结构对象的第一个成员。在 D2 处，又将这个 int 类型的指针转换成指向结构的指针。

再来看第二个例子，这个例子之所以显得有些复杂，是因为这里使用了一些有关类型转换的知识。

```
# include <stdio.h>

struct t {char a [100]; float d;};

void fmptsp (char (*p) [100])
{
    struct t * u = (struct t *) p;                //S1
    printf ("%c, %f\n", u->a[0], u->d);
}

void sptfmp (struct t * u)
{
    u->d = 6.0;
    char (* p) [100] = (char (*) [100]) u;        //S2

    (* p) [0] = 'x';                              //S3
```

```
    fmptsp (p);
}

int main (void)
{
    struct t u;
    sptfmp (& u);

    return 0;
}
```

在函数 sptfmp 内的语句 S2 中，p 是指向数组（char [100]）的指针，u 的类型是指向结构的指针。我们将 u 的值转换为指向数组的指针，并赋给对象 p。依据上述规定，p 实际上指向结构的第一个成员。

在 S3 中，表达式 p 的类型是 char (*) [100]，所以*p 的类型是 char [100]，而表达式(*p)[0]的类型是 char。这就是说，我们通过指向结构第一个成员的指针来间接访问了结构的这个成员。

在函数 fmptsp 内则相反，S1 将指向结构首个成员的指针转换成指向结构的指针，然后用这个指针来访问结构的每一个成员。

因为所有联合对象的成员都共用全部或者部分存储空间，所以如果存在着一个指向联合对象的指针，则它可以被转换为一个新的、指向联合对象任何一个成员的指针（如果某个成员是位字段，则指向的是该位字段驻留的那个存储单元）。

反之，如果存在着一个指向联合对象任何一个成员（或者该成员驻留的存储单元，如果它是位字段）的指针，则可以将该指针转换为一个新的、指向包含该成员的那个联合对象的指针。

```
union u {unsigned char c; double d;} u;
* ((unsigned char *) & u) = 'x';          //&u 为指向联合的指针

/* ..... */

double * p = (double *) & u;             //&u 为指向联合的指针
((union u *) p)->c = 'y';

/* ..... */

((union u *) p)->d = 3.0;

/* ..... */
```

3.20.13 整型提升

前面各个小节所讲述的转换适用于赋值和参数传递等，例如：

```
char c = 'y', a [3], f (int);
f (c);          //对象 c 的值将被转换为 int 类型
int * p = a;    //数组 a 将被转换为指向其首元素的指针
```

对于其他运算符而言，仅有前面所列举的转换是不够的。有些运算符并不在操作数原来的类型上操作，而是先将短值加宽。下面是一些典型的例子。

```
void f (signed char);

signed char c = - 1;
f (~ c);
f (c + 3);
f (c + 3LL);
```

其中，因为一元运算符～的结果类型是 int，而 c 的类型是 signed char，所以要将 c 的值转换为 int 类型后各位取反。然后这个 int 类型的值转换为 signed char 类型的值，再传递给函数 f。当然，这并不是我们关注的重点，下同。

再来看表达式 c+3，常量表达式 3 的类型是 int，但 c 的类型是 signed char，两个值的类型不同，它们怎么相加呢？

解决之道是，将运算符的操作数都转换为相同的类型，即，将对象 c 的值加宽，转换成 int 类型，然后相加，结果的类型是 int。

最后来看表达式 c+3LL。常量表达式 3LL 的类型是 long long int，所以 c 的类型也要加宽，转换为 long long int 类型，这也是它们相加后结果的类型。

下面的代码通过输出各表达式的结果类型的长度，可以帮助读者清楚地认识类型转换过程，以证明上面的解释。

```
# include <stdio.h>

void f (void)
{
    signed char c = - 1;
    printf ("%zu\n", sizeof ~ c);
    printf ("%zu\n", sizeof (c + 3));
    printf ("%zu\n", sizeof (c + 3LL));
}
```

总体上，如果原类型相对于 int 来说较窄，则会首先转换成 int 类型；如果 int 类型无法表示原类型的值，则转换成 unsigned int 类型。以下类型被视为相对于 int 类型来说较窄。

（1）整型转换阶小于 int 和 unsigned int 的类型。

（2）整型转换阶等于 int 和 unsigned int 的整数类型，但不是 int 和 unsigned int 类型。

（3）_Bool、int、signed int 和 unsigned int 类型的位字段。因为位字段的类型通常只能是 _Bool、int、signed int 或者 unsigned int，而且位字段对象可以容纳的值的范围相对于它的类

型来说，只是一个子集。顺便说一句，在这里，int 和 signed int 并不重复，原因是如果位字段的类型是 int，那么它究竟是 signed int 还是 unsigned int，是由 C 实现定义的。

统一转换为 int 或者 unsigned int 类型的原因可能是这两种类型通常具有硬件架构所提供的自然长度，如寄存器的字长。对于上述 3 点所涉及的类型，如果它们取值范围（位字段的取值范围取决于它声明时的宽度）内的所有值都能够用 int 类型来表示，则转换为 int 类型，否则转换为 unsigned int 类型。这个转换过程称为整型提升（integer promotions）。以上 3 点未涉及的类型则保持不变。

下面用几个例子加以说明。

```
struct {unsigned int u : 3; char c;} t;
unsigned short s;
void f (long long int);
long long int lli;
int i;

/* ..... */

f (t.u + s);
f (lli + i);
f (~ t.c);
```

在编者的机器上，int、unsigned int 及 unsigned short 类型的取值范围分别如下。

```
#define INT_MAX     2147483647
#define INT_MIN     (-INT_MAX-1)
#define UINT_MAX    0xffffffff
#define USHRT_MAX   0xffff
```

首先，对于表达式 t.u+s 来说，位字段 u 的取值是 0～7，所以它整型提升到 int 类型；对象 s 的类型是 unsigned short，它的整型转换阶小于 int 和 unsigned int，且其所有值都能用 int 类型来表示，故整型提升到 int。

其次，对于表达式 lli+i 来说，对象 lli 的类型是 long long int，整型转换阶高于 int 和 unsigned int，因此不需要转换，保持不变；对象 i 的类型是 int，也保持不变。当然，就相加运算而言，还需要将两者转换为同一种类型，但那是常规算术转换的事，对于它们的整型提升是到此为止的。

最后，表达式～t.c 来说，因为表达式 t.c 的类型是 char，整型转换阶低于 int，因此需要提升为 int。

要了解哪些类型的整型转换阶于小 int 和 unsigned int，以及整型转换阶的定义和说明，请参见“（整型）转换阶”。

3.20.13.1 （整型）转换阶

整型转换阶（integer conversion rank）是每个整数类型都有的、用于确定如何进行整型提升和类型转换的等级或级别，阶的大小和类型的宽度有关。

（1）总体来说，整型转换阶具有传递的特点：如果整数类型 T1 的阶大于 T2，而 T2 的阶大于 T3，那么可以肯定 T1 的阶大于 T3。

（2）long long int、long int、int、short int 和 signed char 的阶依次递减。

（3）任何无符号整数类型的阶等于与之相对应的有符号整数类型（如果有）的阶。依据这一规则可以推导出，unsigned long long int、unsigned long int、unsigned int 和 unsigned char 的阶也是依次递减的。

（4）任何有符号整数类型的阶都大于比它精度低的有符号整数类型，任何两个有符号整数类型的阶都不相同。对任意两个同符号、不同整数转换阶的整数类型而言，具有较高整数转换阶的类型，在值的范围上涵盖低阶的类型。

（5）任何标准整数类型的阶都大于与其等宽的扩展整数类型，而且_Bool 类型的阶低于任何其他标准整数类型。

（6）特别地，char、signed char 和 unsigned char 类型的阶是相同的，任何枚举类型的阶都等于它所兼容的那个整数类型。

（7）两个精度相同的扩展有符号整数类型，它们的阶在相比较时孰大孰小，由 C 实现决定，但仍然要服从那些用于确定整型转换阶的规则。

整型提升和结果可能和用户想象的不一样，下面这个例子来自网络。

```
unsigned char temp;
/* ..... */
if(! ~ temp) return 0;
/* ..... */
```

以上代码片断的功能是判断对象 temp 的值，如果为 0xFF，则退出当前函数。当然，假定 C 的实现将 CHAR_BIT 定义为 8。但程序的执行非常不稳定，这是为何呢？其实原因很简单，因为 unsigned char 类型的阶较 int 和 unsigned int 要低，所以要先整型提升，提升后的类型是 int。如果对象 temp 的当前值是 0xFF，则提升后的值是 0x000000FF（假定 int 类型的宽度是 32 个比特），运算符～的结果相应地会变成 0xFFFFFF00，不为 0，所以运算符!的结果是 0，return 语句不会被执行。

3.20.14 常规算术转换

对很多需要算术类型操作数的运算符而言，操作数的类型不同是很平常的事情。对此，一个普遍的、常规的解决方案是，先确定操作数及结果的公共实数类型，然后转换到这个类型并进行操作。

常规算术转换以其名称来说，在涉及算术操作数的表达式中普遍应用。但是，它只针对算术类型，对其他类型（如指针类型）则没有影响。

常规算术转换（usual arithmetic conversions）的主要思想是：将每个操作数（包括实数类型和复数类型）对应的实数类型转换为所有操作数的公共实数类型，但不改变各自的类型域（在转换前为实数类型的，转换后依然是实数类型；转换前是复数类型的，转换后依然是复数类型）。通常来说，这个公共实数类型也是运算结果对应的实数类型。

结果类型所属的类型域取决于两个操作数的类型。如果两个操作数都是实数，则结果的类型也是实数；如果两个操作数都是复数，则结果的类型也是复数；如果一个操作数是实数

类型，另一个是复数类型，则结果是复数类型。

第一，若两个操作数对应的实数类型都是 long double，则不需要转换；若只有一个操作数对应的实数类型是 long double，则需要将另一个操作数的类型转换为相应实数类型为 long double 的新类型，但不改变该操作数的类型域。

在下例中，运算符+的两个操作数类型不同，需要进行转换。因为 f 的类型是 long double，所以也要将 zd 的类型从 float _Complex 转换为 long double _Complex，显然，转换后依然是复数类型（类型域保持不变）。

```
long double f = 1.0;
float _Complex zd, g ();
/* ..... */
g (zd + f);      //表达式 zd+f 的结果类型是 long double _Complex
```

如果一个操作数是实数类型，另一个操作数是复数类型，则运算的结果是复数类型。而且，不要求 C 实现将它们转换为同一类型（不改变各自的类型域），这样做的原因是提高性能。

第二，若两个操作数中没有任何一个对应的实数类型是 long double，则看它们是否都是 double。若两个操作数对应的实数类型都是 double，则不需要转换；若只有一个操作数对应的实数类型是 double，则需要将另一个操作数的类型转换为相应实数类型为 double 的新类型，但不改变该操作数的类型域。

在下例中，运算符+的两个操作数类型不同，需要进行转换，将 zf 的类型从 float _Complex 转换为 double _Complex，转换后依然是复数类型。

```
float _Complex zf;
double _Complex zd;
void f ();
/* ..... */
f (zf + zd);     //表达式 zf+zd 的结果类型是 double _Complex
```

如果两个操作数都是复数，则运算的结果依然是复数。

第三，若两个操作数中的任何一个对应的实数类型既不是 long double，也不是 double，则看它们是否为 float。若两个操作数对应的实数类型都是 float，则不需要转换；若只有一个操作数对应的实数类型是 float，则需要将另一个操作数的类型转换为相应实数类型为 float 的新类型，但不改变该操作数的类型域。

在下例中，运算符+的两个操作数类型不同，需要进行转换，因为一个操作数的类型是 float 而另一个是 long long int，所以要将 long long int 类型转换为 float。

```
float f;
long long int i;
void g ();
/* ..... */
g (f + i);       //表达式 f+i 的结果类型是 float
```

第四，因为常规算术转换只涉及算术类型，如果任何一个操作数对应的实数类型都不是 long double、double 和 float，那么说明一个问题：它们都是整数类型。在这种情况下，要先对两个操作数做整型提升，如果两个操作数提升后的类型相同，则不再进行转换。

下例中，c 的类型是 char，需提升为 int。而因为 1 的类型是 int，故不再进行转换。

```
void f (char c)
{
    if (c > 1) {/* …… */}
}
```

第五，在经过第四步的整型提升后，如果两个操作数的类型不同，但都是无符号整数类型或者都是有符号整数类型，则将阶较低的那个操作数的类型转换为阶较高的那个操作数的类型。

在下例中，对于表达式 s>l，因为 int 类型可以表示 short 类型的所有值，所以对象 s 的值先被提升为 int 类型，操作数 1 的值不需要改变类型。提升之后，运算符>的两个操作数都是有符号整数类型，但 long 类型的阶较之 int 类型高，故将 int 类型的值进一步转换为 long 类型，这样>运算符的两个操作数就具有了相同类型。

```
short s = 0;
long l = 0;
if (s > l) {/* …… */}
```

第六，在经过第四步的整型提升后，如果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，那么：

（1）如果无符号整数类型的阶高于或等于有符号整数类型，则有符号整数类型的那个操作数转换为另一个操作数的类型（无符号整数类型）。

在下例中，对于表达式 s>u，对象 s 的值先提升为 int 类型，u 的值不改变类型。提升之后，运算符>的两个操作数一个是有符号类型，另一个是无符号类型，因 unsigned long 类型的阶高于 int，所以要将 int 类型的值转换为 unsigned long。

```
short s = 0;
unsigned long u = 0;
if (s > u) {/* …… */}
```

（2）如果无符号整数类型的阶低于有符号整数类型，则要看那个有符号整数类型是否能够表示无符号整数类型的所有值。如果可以表示，则无符号整数类型的操作数转换为另一个操作数的类型（有符号整数类型）；如果不能表示，则两个操作数都要转换到那个有符号类型的无符号版本。

在下例中，对于表达式 us>l，对象 us 的类型是 unsigned short，这需要先做整型提升。假定 int 类型不能表示所有 unsigned short 类型的值（在有些 C 实现上，short 和 int 类型的宽度是一样的），则对象 us 的值只能提升为 unsigned int 类型，但无论如何，l 的值不改变类型。

```
unsigned short us = 0;
long l = 0;
if (us > l) {/* …… */}
```

提升之后，运算符>的两个操作数一个是 long int 类型，另一个是 unsigned int 类型。此时，有符号整数类型 long int 的阶高于无符号整数类型 unsigned int。如果 long int 类型不能表示所有 unsigned int 类型的值（这是可能的，对于有些 C 实现来说，long int 和 int 类型的宽度是一样的），则这两个操作数的值都要转换为有符号整数类型的无符号版本，即 unsigned long int 类型。

相反，如果 long int 类型可以表示所有 unsigned int 类型的值（这是可能的，对于有些 C

实现来说，long int 类型的宽度大于 int 类型的宽度），则只需将 unsigned int 类型的操作数的值转换为另一个操作数的类型（long int）。

3.20.15 默认参数提升

在调用一个函数时，传递进去的每个值是否需要转换、如何转换，取决于函数的声明是采用原型还是采用传统形式。

在调用一个函数时，如果该函数是用传统（K&R）形式声明的，那么就要看传递给函数的值是何种类型。是整数类型时，要先做整型提升；是 float 类型时，要先转换为 double 类型，这就是所谓的默认参数提升（default argument promotions），或者称“缺省参数提升”。默认参数提升对其他参数类型没有影响。

如果被调用的函数使用了原型，那么传递的参数如何转换及其他相关信息，可参见“函数调用”。

下面给出了需要和不需要默认参数提升的实例。

```
char c = 0, f ();
void g (const char *, float);

f (1.0f);
f (c);
f (1, c);
g ("hello.\n", 2.0f);
g (0);          //错误：参数的数量不匹配
```

其中，函数 f 是用传统 K&R 形式声明的，因此，所有对 f 的调用都能编译通过，但行为都是未定义的。1.0f 是 float 类型，要转换为 double 类型；c 是 char 类型，也要进行整型提升。对函数 g 的声明采用的是原型形式，所以不需要进行默认参数提升，而且对参数的类型和数量都要严格检查。

3.21 有效类型

对象的类型关乎对象的很多属性，例如对象的大小、值的表示、值的有效性、可以存储的值的范围，以及所允许的运算（操作）。

总体上，存储在对象中的值及函数的返回值，它们的意义要靠访问它的表达式的类型来确定。访问一个对象时，所使用的类型应当和它原本的类型一致，这被视为是有效的，否则程序的行为是未定义的。例如：

```
int i;
* (float *) & i = 2.0;
```

在这个例子中，i 所指示的对象应当存储 int 类型的值，但在随后的访问中，却使用了浮点类型。int 和 float 这两种类型的对象表示不同。

如果访问一个对象时所用的类型 T 和当初声明这个对象时指定的类型一致（在下文中称

“声明时的类型”），则这是安全有效的做法，我们称T是该对象的有效类型（effective type）。

但问题是，有些对象并不存在所谓声明时的类型，比如那些用内存分配函数创建的、具有指派存储期的对象。在这种情况下，使用哪种类型来访问才是正确的呢？或者说，如何确定该对象的有效类型呢？

如果一个对象不存在声明时的类型，那么：

（1）如果用非字符类型的左值向该对象做了一个写操作，修改了对象的存储值，那么，对于本次操作和后续的任何纯读取操作而言，左值的类型就是该对象的有效类型。例如：

```
# include <stdlib.h>

void f (void)
{
    void * p = malloc (10);      //S1
    * (int *) p = 0;             //S2
    int * q = p, l = * q + 1;    //S3
}
```

其中，在S1中实际上存在两个对象，一个是对象p，另一个是p所指向的对象。p的类型是指针，即void *，但p所指向的对象不存在有效类型；在S2中，p所指向的对象现在的有效类型是表达式*(int *)p的类型，即int类型；在S3中，p所指向的对象（也是q指向的对象）的有效类型依然是int。

（2）如果该对象的存储值是采用复制的方式写入的（不管是使用库函数memmove或者memcpy，还是用字符类型逐字节复制），那么，对于本次复制操作和后续的纯读取操作而言，该对象的有效类型和那个被复制的对象一样，如果后者有的话。下例中，出于复制操作的原因，指针p所指向的对象便具备long long int类型：

```
# include <stdlib.h>

void f (void)
{
    long long ago = 1860;
    void * p = malloc (100);

    for (unsigned i = 0; i < sizeof ago; i ++)
        ((char *) p) [i] = ((char *) & ago) [i];
}
```

（3）对于以上两点未涉及到的情形，该对象的有效类型就是访问该对象的左值的类型。

原则上，若对象的有效类型是T，则该对象的存储值只能用以下类型的左值访问：

（1）和T兼容的类型及其限定版本，它们是相同或者高度相似的类型；

（2）和T及其限定版本相对应的有符号或者无符号类型，它们的宽度相同；

（3）聚合、联合类型，前提是它们（或者递归地，它们的聚合、联合成员）拥有前两种类型的成员；

（4）字符类型。字符类型不存在填充位，也不存在自陷值。

第4章 词法元素

在C实现转换一个C程序时，它首先将源文件看作一个字符的序列，读入这些字符，将源文件分解为由空白字符隔开的预处理记号，然后执行预处理指令，得到用于语法和语义分析的记号。

4.1 预处理记号

在程序转换阶段，源文件的文本流会被解析为一系列预处理记号（preprocessing token），它们包括头文件名、标识符、预处理数字、字符常量、字面串、标点符号（含运算符），以及其他非空白字符。

除了头文件名之外，C实现在处理文本流时，以能够组成上述预处理记号的最长字符序列为准则。这意味着，如果源文件中有文本

```
m+++n
```

则它会被解析为

```
m ++ + n
```

而不是m + + + n、m + ++ n、m + ++ n，或者其他形式。原因很简单，依据上面的准则，m是最长的合法标识符而m+和m++等都不是；++是最长的合法运算符（标点符号）而+++和+++n都不是；后面的+也是最长的合法运算符而+n不是。

再比如，文本序列

```
m+++++n
```

会被解析为

```
m ++ ++ + n
```

而不是

```
m ++ + ++ n
```

尽管后者能够通过编译而前者不能，原因可参考上一个示例自行分析。

头文件名只在#include指令和#pragma指令内识别。

4.2 记号

源文件的文本流先转换为预处理记号（参见“预处理记号”），经预处理（执行预处理指令）后，即可得到记号（token）。

记号是 C 语言中最小的词法元素，它包括标识符、关键字、常量、字面串和标点符号（含运算符）。具体可参见“标识符”、“关键字”、“常量”和“字面串”。

4.3 标识符

标识符（identifier）是 C 程序中用于标识对象、函数、typedef 名、标号名等实体的字符序列。图 4-1 给出了标识符的语法图。

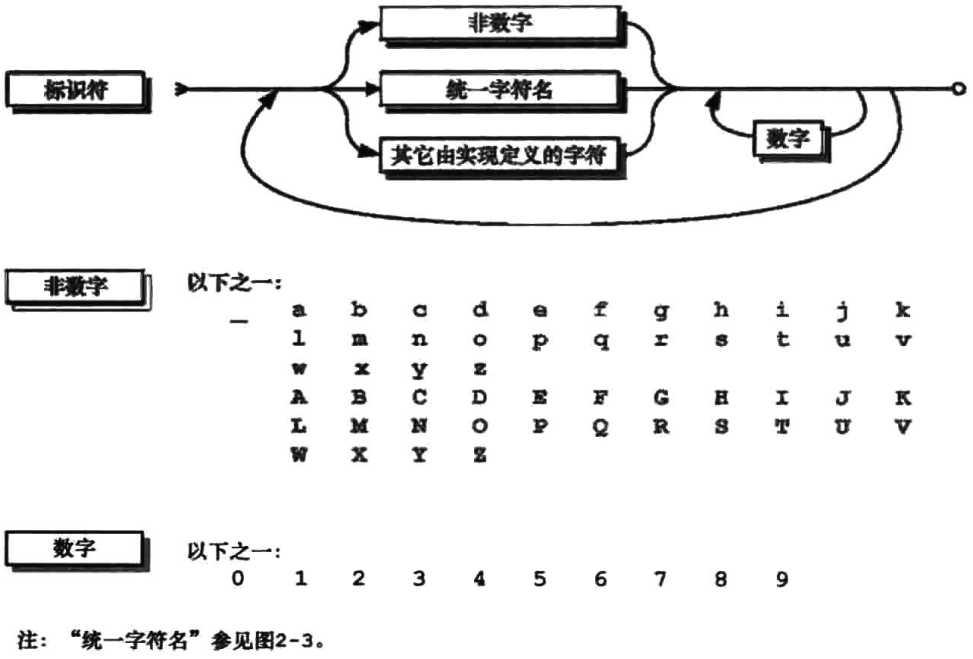


图 4-1 标识符语法图

在标识符中，字母的大小写是明确区分的。如果两个标识符的拼写完全一样，包括大小写，则它们是相同的标识符。例如，标识符“Hello”和“Hello”是相同的标识符，但它们都和“HeLLo”不同。

从 C99 开始，允许在标识符中使用统一字符名和实现定义的字符。统一字符名用于引用 ISO/IEC 10646 字符集中的成员，标准文档的附录中给出了哪些字符编码可用于组成标识符的统一字符名，也给出了当标识符的第一个字符由统一字符名指定时，哪些字符编码是不允许使用的。

所谓实现定义的字符，是指不属于基本字符集的多字节字符，比如各个国家和地区的本地字符。但具体哪些字符是可以使用的，以及它们所对应的统一字符名是什么，取决于 C 实现。

原则上，标识符的长度没有限制，唯一的问题是 C 实现只能识别和处理有限的长度。事实上，如果标识符很长，则只有最开始的一部分是最重要的，称为有效长度。

标识符的有效长度取决于 C 实现，也取决于它的链接属性，标准对此有明确的规定：如果标识符具有内部链接属性（只在它所在的转换单元内使用），则标准规定其最长不得超过 63 个字符，每个统一字符名或者源字符集中的扩展字符被视为一个字符。

如果标识符是外部链接的（要被其他转换单元引用），则对有效长度的限制更加严格，因

为这需要考虑并受限于其他因素，如调试程序或链接程序。总之，标准规定其最长不得超过31个字符。统一字符名对应的扩展字符可能产生很长的外部标识符，因此，如果使用了统一字符名，则小于\U0000FFFF的统一字符名被视为6个字符；大于\U00010000的统一字符名被视为10个字符。在每个转换单元内，允许声明4095个外部链接的标识符，以及511个具有块作用域的标识符；在每个源文件内，允许定义4095个宏名。

【思考题】在下面给出的标识符中，哪些是合法的，哪些是非法的，为什么？

(1)hello　　(2)30years　　(3)_wait　　(4)name-01

4.3.1 预定义标识符

从C99开始，C中预定义了标识符__func__。这是一个非常有用的标识符，在程序转换阶段，C的实现会隐式地在每个函数体的开头创建该标识符的声明：

```
static const char __func__ [] = "函数名";
```

这里的“函数名”是标识符__func__所在函数的名称。这意味着，用户可以这样使用它：

```
# include <stdio.h>

void fdemo (void)
{
    char c = __func__ [0];
    printf (__func__);
}
```

尽管预定义标识符不是关键字，但不要尝试重新定义这个标识符，否则行为是未定义的。预定义标识符__func__在调试程序时有用，它可以打印出错误发生在哪个函数中。例如：

```
# include <stdio.h>
# include <stddef.h>

void my_func (const char * p)
{
    if (p == NULL)
    {
        printf ("In '%s':NULL pointer error.\n", __func__);
        return;
    }
    /* ..... */
}
```

【思考题】一个C程序很复杂，而且将近有几十个地方都调用了函数func。现在，我们需要在函数func内显示是谁调用了它（调用者的函数名），比较笨拙的方法是先修改func，添加一个字符指针，然后找到每一个调用点，将调用者的名字传递给func。但是，有没有更好的办法来减小工作量呢？

4.3.2　名字空间

尽管不建议使用相同的标识符声明不同的对象和函数，但如果真的这么做了，应该怎么处理呢？这是语言本身需要关注的事情。例如，同样使用了相同的标识符，为什么下面的两个声明

```
int * a;            //D1
char a [22];        //D2
```

不能共存（使用了相同的标识符 a），而这两个声明

```
int * t;                            //D3
struct t {int t; _Bool b;} m;       //D4
```

却能共存（使用了 3 个相同的标识符 t）？

答案就隐藏在使用它们的句法上下文。之所以 D1 和 D2 不能共存，是因为如果有一个声明

```
long l = * a;
```

这里将无法区分 a 到底指示上述哪个对象，即无法确定是 D1 中的 a，还是 D2 中的 a。相反，如果在程序中出现了表达式

```
t = & m.t;
```

则可以很明确地区分出，第一个 t 是 D3 中的 t；第二个 t 是 D4 中的结构成员。

如以下代码所示，D3 中的标识符 a 指示在 D1 中声明的对象；D4 中的 a 和关键字 struct 连用，指示的是在 D2 中声明的结构类型；S1 中有两个 a，左边的 a 指示在 D1 中声明的对象，右边的 a 指示结构的成员，因为它是成员选择运算符 . 的右操作数。

```
char * a = 0;                       //D1
struct a {char a; _Bool b;};        //D2
char * pa = a;                      //D3
struct a b = {'x', 0};              //D4
a = & b.a;                          //S1
```

通过以上的例子可以看出，在一个转换单元里，可能有多个声明使用相同的标识符。当用到这个标识符时，如果它的多个声明在当前位置上都可见，那么该标识符对应于它的哪个声明，要靠它所在的句法上下文来辨别。具体地说，每个标识符在声明之后，都属于以下 4 个名字空间（name spaces）之一，从它所在的句法上下文可以判断它所在的名字空间：

（1）标号名。标号名在声明和使用上非常独特，很容易区分。当 goto 后面跟一个标识符时，该标识符不可能指示一个对象；同样，在需要一个表达式的地方也不能出现标号名。

（2）结构、联合和枚举的标记。它们出现时，前面总会有 struct、union 或者 enum，因此很容易区分。

（3）每个结构类型的成员，或者每个联合类型的成员。每个结构或者联合类型都各自为其成员形成一个名字空间。区分它们的方法是看成员选择表达式的左操作数是谁。

（4）上述之外的其他标识符，称为普通标识符，也共同形成一个名字空间。所有被前面 3 个名字空间排除在外的标识符都属于这个名字空间。

基本规则：在转换单元内的某个地点，如果能够同时看到多个同名的标识符，且它们分别指示不同的实体（对象、函数、标号等），则它们必须各自位于不同的名字空间。下面是一

个典型的例子，D1 和 D2 中都声明了标识符 i，但都是作为普通标识符出现的，属于上述第 4 个名字空间，这是不允许的；在 D2 中，标识符 c 作为同一结构的成员出现（声明）了两次，而且都属于上述第 3 个名字空间，这是不允许的；D2 和 D3 都声明了标识符 t，它们同属于上述第 2 个名字空间，这也是不允许的。

```
int i;                          //D1
struct t {char c; int c;} i;    //D2
union t {/* ····· */} j;        //D3
```

如果标识符同名，但它们都分属于不同的名字空间，则是允许的，也是正当的、合法的。与上面的例子相比，下面的例子没有任何问题，任何一个 t 都能从上下文中判断它到底指代哪个对象。

```
struct t {char t; _Bool b;};
struct s {char t; _Bool b;} s = {0, 1};
unsigned int t = s.t;
t:
printf ("%d\n", t);
if (t ++ != sizeof (struct t)) goto t;
```

再比如下面的示例，尽管在 D1 和 D2 中都声明了以粗体显示的标识符 s，但它们能够共存，是因为它们指示同一个实体。进一步地，它们也不和斜体、带下划线以及正常字体显示的标识符 s 冲突，毕竟他们所属的名字空间不同。在函数 f 内，表达式 s.s.s 是合法的，C 实现可根据名字空间来正确地对应它们的实体。

```
struct s {struct {char s;} s;} s;   //D1
extern struct s s;                  //D2

void f (void) {s.s.s = 'x';}
```

【思考题】给定声明：

```
struct t {int t;} t;
```

这 3 个标识符 t 为什么可以共存？

4.3.3 作用域

对一个 C 程序来说，标识符几乎是不可或缺的，它有时用于指示对象或者函数，有时被声明为结构、联合或者枚举的标记，有时被声明为结构或者联合的成员，有时被声明为枚举常量，还可以是 typedef 名、标号名、宏名或者作为宏的参数出现，等等。

通常情况下，在使用一个标识符的时候，要求它在当前位置是可见的。换句话说，它应当在前面的程序文本中有所声明（说明），否则就不知道它是什么，也不可能知道对它的使用是否合法。每一个标识符都只在程序文本（源文件中）的某个部分是可见的、起作用的、能够使用的，这个部分就是该标识符的作用域（scope）。

有 C 语言中有 4 种作用域，分别是文件作用域（file scope）、块作用域（block scope）、函数作用域（function scope）和函数原型作用域（function prototype scope），有关它们的具体描述，可参见其各自的词条。

在转换单元内的不同地点，同一个标识符可以指示不同的实体。其原因有两个：一是它们可能分属于不同的名字空间，但这不是现在讨论的主题，具体内容可参见“名字空间”；二是它们处于不同的作用域。

对于下面的程序片断，要想成功编译是不可能的。原因不仅仅是 F1、F2、F3 和 F4 处的这几个标识符 f 位于同一个名字空间。实际上，这里存在着两对冲突，而不是它们 4 个彼此冲突。因为 F1 和 F2 属于同一个（文件）作用域，所以它们是直接冲突的；F3 和 F4 位于同一个（块）作用域，它们也是直接冲突的。

```
double f /* F1 */ = 0.0;

void f /* F2 */ (int f /* F3 */)
{
    float f /* F4 */ = 0.0;
    /* …… */
}
```

要解决这个问题，只需要让它们在各自的作用域内避免冲突即可。下面给出的是修改之后的程序代码。

```
double d = 0.0;

void f (int f)
{
    float d = f;
    /* …… */
}
```

两个相同的标识符，如果它们各自指示不同的实体，且属于同一个名字空间，要使它们不冲突，可以采用让它们的作用域重叠的办法，即所谓的外部块和内部块。在这种情况下，外部块内的标识符被内部块中的同名标识符隐藏。

如下例所示，在 D1 处声明的标识符 f 是函数名，具有文件作用域，在 D2 处声明的标识符 f 是函数参数，具有块作用域（在这里是函数体）。本来函数名在函数体内是可见的，但它在函数体内被隐藏了，取而代之的是作为函数参数的 f。

在这种情况下，D3 的行为是我们所期望的，因为 pi 是一个指向 int 类型的指针，而表达式&f 的结果类型同 pi 的类型一致。在 D4 中，本意是得到指向当前函数 f 的指针，因为 pf 的类型是指向函数的指针。问题在于，被声明为函数的 f 在块内被隐藏了，所以 D4 是不正确的。

在 D5 中，标识符 p 被声明为一个指向 char 类型的指针。在进入内部块后，它依然是可见的，甚至可以在 D6 中将它的值赋给另一个指针 q。但是在声明 D7 中的同名标识符 p 后，原来的 p 被隐藏了。

```
void f /* D1 */ (int f /* D2 */)
{
    int * pi = & f;                    //D3: 这个 f 是什么?
    void (* pf) (int) = & f;           //D4: 这个 f 是什么?
```

```
    char * p = 0;                          //D5
    /* ..... */
    if (p == NULL)
    {
        char * q = p;                      //D6: 此处的p是D5中声明的p
        char p [] = "hello, world.\n"; //D7
       /* ..... */
    }
    char * q = p;                          //D8
}
void (* pf) (int) = & f;                   //D9
```

但一旦离开内部块，外部块中的同名标识符立即重新可见。正是因为这个原因，D8 中使用的标识符 p 是在 D5 中声明的 p；D9 中使用的标识符 f 是在 D1 中声明的 f。

在下例中，D1 中的标识符 Integer 和 D2 中的标识符 e、Door、Floor 都具有文件作用域，但它们在函数内部被同名的标识符隐藏。在函数体外它们又重新可见；D3、D4、D5 和 D6 中的标识符 Integer、e、a、b 具有块作用域，仅在它们所在的块内可见，故可以在它们所在的块外重新声明 D7、D8 处的标识符 a、b 和 e，而且 D7 中的 Integer 指的是 D1 处声明的 Integer；D8 中的 enum e 指的是 D2 处声明的 enum e。

```
typedef int Integer;                   //D1
enum e {Door = 5, Floor};              //D2

void f (void)
{
    typedef signed long Integer;       //D3
    enum e {Door, Floor};              //D4
    Integer a, b;                      //D5
     enum e e;                         //D6
}

Integer a, b;                          //D7
enum e e;                              //D8
```

标识符的作用域决定了它的可见性，以及是否可用。比如声明：

```
struct stgNODE {int data; struct stgNODE * next;};
```

这里，允许在结构 struct stgNODE 内部声明一个指向它自己的指针成员，因为在声明这个成员的时候，标识符 stgNODE 是可见的。问题是，在这个时候，类型 struct stgNODE 并不是完整类型，所以也只能声明它的指针成员（声明一个结构的指针并不需要它的完整类型信息）。相似地，下面的声明也是允许的：

```
typedef struct stgNODE {int data; struct stgNODE * next;} * PNODE;
```

其合法的原因和前一个例子相同。但是，下面的做法：

```
typedef struct stgNODE {int data; PNODE next;} * PNODE;
```

或者这样的做法：

```
typedef struct {int data; PNODE next;} * PNODE;
```

就不行了。在这两个例子中，我们是要定义类型 PNODE，也就是结构类型的别名。但是，在声明结构成员 next 的时候，PNODE 还不可见，还没有出现，或者说不在它的作用域内，所以是非法的。

【思考题】在下面的代码片断中，D1 处声明的是什么？标识符 s 和 t 分别具有何种作用域？D2 和 D3 这两个声明合法吗？

```
struct s {int i;} * f (struct t {float f;} * t);   //D1

void f (void)
{
    struct s s;      //D2
    struct t t;      //D3
}
```

在本书中将多次出现“相同的作用域”、“作用域是相同的”或者“同一作用域”等类似的措辞。不管两个标识符的作用域是否开始于同一个点（位置），只要它们终止于同一个点（位置），则它们属于同一作用域，或者说具有相同的作用域。

在下面的例子中，标识符 i 和 j 具有相同的作用域；标识符 x 和 y 也属于同一作用域。

```
int i;
void f (int x)
{
    int y = x;
    /* ..... */
}
int j;
/* ..... */
```

显然，即使两个标识符属于同一作用域，它们的开始位置通常并不相同，比如上例中的标识符 i 和 j。

不同类型的标识符，其作用域开始的位置略有差别：如果是一个（结构、联合或者枚举类型的）标记，它的作用域开始于它的声明，起点是它在声明中出现的位置；如果是一个枚举常量，它的作用域开始于它的定义，起点是它在枚举器列表中出现的位置；其他标识符的作用域立即开始于它所在的声明符之后。

【思考题】在下面的代码片断中，哪些标识符和哪些标识符具有相同作用域？每个标识符的作用域都开始于什么位置？

```
enum Sig {VGA, SDI, HDMI};

void f (void)
{
    int x, y;
```

```
    struct s {char c; float f;} t;
}
```

4.3.3.1 文件作用域

文件作用域经常被称为“全局”，但这个称谓并不十分恰当，因为它并不真的是“整个文件”。具有文件作用域的标识符，不管它们的作用域从哪里开始，结束点都是当前所在转换单元的末尾。精确地说，同时满足以下两个条件的标识符都具有文件作用域：

（1）在任何块之外声明的标识符；

（2）在函数参数列表之外声明的标识符。

例如，下面的标识符 i 和 main 都具有文件作用域，i 的作用域从它在 D1 中被放置的地方开始，延续到当前转换单元的末尾；main 的作用域从它在 D2 中被放置的地方开始，延续到整个当前转换单元的末尾。

```
int i = 0;          //D1

int main (void)     //D2
{
    /* ..... */
}
```

4.3.3.2 块作用域

块作用域，顾名思义，仅限于块内。一个具有块作用域的标识符，不管它的作用域开始于块内的什么位置，都结束于它当前所在块的末端。精确地说，在块内声明的标识符，或者在一个函数定义（这意味着是带有函数体的函数声明）的参数列表中声明的标识符，都具有块作用域。

在下面的例子中，标识符 x 和 y 都具有块作用域。其中，x 的作用域从它在 D1 中的位置开始，延续到右花括号之前；y 的作用域从它在 D2 中的位置开始，延续到右花括号之前。

```
void f (int x)              //D1
{
    long long int y = x;    //D2
    /* ..... */
}
```

特别要注意的是，块并非特指花括号和其中的内容。事实上，函数体是一个块；复合语句也是块，选择语句和迭代语句也是块。有关块的具体描述可参见“块”。例如：

```
void blkdemo (int x)
{
    int y = 3;
    for (int i = 0; i < x; i ++) y ++;
    i ++;   //非法
}
```

上例中，标识符 x、y 和 i 都具有块作用域，且假定 i 未在当前函数之外声明过（或者说

在函数体内看不见 i 的另一个声明）。标识符 x 的作用域从它在声明中出现的位置开始，延伸并终止于函数体的右花括号；标识符 y 的作用域从它在声明中出现的位置开始，延伸并终止于函数体的右花括号。因为 for 语句本身就是一个块，所以标识符 i 的作用域从它在声明中出现的位置开始，延伸并终止于它所在的 for 语句末尾。当从 for 语句中退出后，就超出 i 的作用域，所以表达式 i++是非法的。

4.3.3.3 函数作用域

函数作用域就范围而论，它包括整个函数。如果一个标识符具有函数作用域，则意味着它是在一个函数内声明的。但是，不管它是在函数内的什么位置声明的，它在那个函数内的任何位置都可见，但仅局限于那个函数。

标号名是唯一具有函数作用域的标识符，而且可以在函数内的任何地方使用它。标号的声明有独特的句法外观：标识符（标号名）后面跟一个冒号（:）以及语句。

如下例所示，与别的作用域不同，尽管标号 lb 是在 B 处定义的，但它却可以在 A 处使用。当然，它也可以在 C 处使用。

```
void f (void)
{
   goto lb;                    //A
   /* ····· */
 lb:;                          //B
   /* ····· */
   if (/* ····· */) goto lb; //C
}
```

4.3.3.4 函数原型作用域

如果标识符是在一个函数的参数列表中声明的（这意味着该函数是原型形式的），且该参数列表不是函数定义的一部分，那么该标识符具有函数原型作用域。

具有函数原型作用域的标识符，其作用域是从它在声明中的位置开始，延续到整个原型结束。下例中，标识符 c 和 i 都只具有函数原型作用域。

```
int f (char c, int i);
```

4.3.4 链接

如果在程序中有一个标识符的多次声明，在这些声明都合法的前提下，它们可能指示不同的实体，也可能指示同一个对象或者函数。请看下面的示例：

```
int x;                    //D1
int y;                    //D2
{
   int x;                 //D3
   extern int y;          //D4
   {
```

```
        int x;              //D5
        extern int y;       //D6
    }
}
```

以上，标识符 x 被声明了三次，分别是 D1、D3 和 D5，这三次声明位于不同的作用域，并且也创建了三个彼此不同的对象；标识符 y 也被声明了三次，分别是 D2、D4 和 D6，这三次声明也位于不同的作用域。尽管如此，这三个 y 都指示同一个对象，而不是分别创建各自不同的对象。

同一个标识符可能会多次声明。不管声明多少次，也不管是在同一作用域内声明的，还是在不同作用域内声明的，如果想让它们指示（关联）同一个对象或者函数（而不是分别指示各自独立的对象或者函数），那么都可以用链接（linkage）的方法来达到这个目的。

C 中有 3 种链接类型：外部链接、内部链接和无链接，请参见其各自的词条。所有外部链接的同名标识符指示同一个对象或者函数；在同一个转换单元内，所有内部链接的同名标识符都指示同一个对象或者函数；所有无链接的标识符都指示独立的实体。

大体上，外部链接的标识符和其他同名标识符指示同一个对象或者函数，这里的“其他同名标识符”可能位于当前转换单元，也可能位于其他转换单元，或可能位于库中；内部链接的标识符和当前转换单元内部的其他同名标识符指示同一个对象或者函数；无链接的标识符指示独立的实体（不存在指示函数的、无链接的标识符）。

在下例中，共存在 5 个对象，而不是 3 个。D1 中声明的 x 是无链接的，尽管它在当前块及内部块中可见，但它指示一个独立的对象，和 D3、D5 及 D7 中声明的 x 所指示的对象不同。因为 D3、D5 和 D7 中声明的标识符 x 具有外部链接属性，所以这三个 x 指示同一个对象。同理，D2 中声明的 y 是无链接的（这里的关键字 static 并非用于指示内部链接属性），它指示独立的对象，和 D4 及 D7 中声明的 y 所指示的对象不同，D4 中的 y 和 D7 中的 y 都是外部链接的，所以它们指示同一个对象。D6 中声明了标识符 z，它是外部链接的，而 D7 中的 z 也是外部链接的，所以这两个 z 指示同一个对象。

```
int fscope (void)
{
    int x;                  //D1
    static int y;           //D2
    {
        extern int x;       //D3
        extern int y;       //D4
         /* ..... */
    }
    extern int x;           //D5
    extern int z;           //D6
    /* ..... */
}
```

```
int x, y, z;                    //D7
```

注意，在同一个转换单元内部，一个标识符的多次声明在链接属性上不得冲突，不能既是外部链接的，又是内部链接的。假定某个转换单元的全部内容如下：

```
int x;                  //D1
static int y;           //D2
void f (void)           //D3
{
    extern int x;       //D4
    extern int y;       //D5
}
static int x;           //D6
extern int y;           //D7
static void f (void);   //D8
```

那么，D1 中的 x 是外部链接的，因为在 D4 处使用了存储类指定符 extern，且能看到 D1 中的 x，故 D4 处的 x 也是外部链接的。但是，D6 中的 x 分明是内部链接的，与 D1 和 D4 冲突，所以非法。

同理，D2 中的 y 是内部链接的，因为在 D5 处使用了存储类指定符 extern，且能看到 D2 中的 y，故 D5 处的 y 也是内部链接的。同样，在 D7 处使用了存储类指定符 extern，且也能看到 D2 中的 y，所以 D7 处的 y 也是内部链接的，这三个 y 都指示同一个对象，没有问题。

最后，D3 中的 f 是外部链接的，但 D8 中的 f 分明是内部链接的，所以这两个 f 的链接属性冲突，是非法的。

假定一个转换单元的全部内容如下，在 D1 处声明的 f，其作用域从它在 D1 中出现的位置开始，延伸并终止于 main 函数体的右花括号。因为它是一个函数，且没有使用任何存储类指定符，所以视为使用 extern。因为当前示例是一个转换单元的全部内容，所以在 D1 处看不到 f 的其他声明，包括 D2。这就使得 D1 中声明的 f 是外部链接的。但是，在 D2 中声明的 f 是内部链接的（使用了存储类指定符 static），这两个声明冲突。

```
int main (void)
{
    int f (void);           //D1
    /* ..... */
}

static int f (void)         //D2
{
    /* ..... */
}
```

上例中，即使在 D1 中使用存储类指定符 static 也无效。如果一个函数是在块内声明的，则它不得使用除 extern 之外的其他存储类指定符（参见“存储类指定符”）。下例中，块内的

声明只有D1和D2正确且完全等效，其他都为非法。

```
int f (void)
{
   int f (void);            //D1
   extern int f (void);     //D2
   static int f (void);
   register int f (void);
   return 0;
}
```

另一个例子，假如一个转换单元的全部内容如下，每一行的注释给出了它们各自的链接属性。

```
static char * f (void);     //f 是内部链接的
char * f (void)             //f 依然是内部链接的
{
    /* ..... */
}
char * g (void);            //g 是外部链接的
static char * g (void)      //错误：同 g 的前一次声明的链接属性冲突
{
    /* ..... */
}
void h (void);              //h 是外部链接的
inline void h (void);       //h 依然是外部链接的
inline void l (void);       //l 是外部链接的
void l (void);              //l 依然是外部链接的
inline void m (void);       //m 是外部链接的
extern void m (void);       //m 依然是外部链接的
static void n (void);       //n 是内部链接的
inline void n (void);       //n 依然是内部链接的
static int a;               //a 是内部链接的
int a;                      //错误：重复定义
static int b;               //b 是内部链接的
extern int b;               //b 依然是内部链接的
int c;                      //c 是外部链接的
static int c;               //错误：同 c 的前一个链接属性冲突
extern int d;               //外部链接
static int d;               //错误：同 d 的前一个链接类型冲突
```

【思考题】

（1）在下例中，D1中的标识符g、D2中的标识符F、D3中的标识符g和D4中的标识符

g 各具有什么样的链接属性，为什么？

```
# include <stdio.h>

static int g (int, int);          //D1

int main (void)
{
   typedef int F (int, int);      //D2
   F g;                           //D3
   return g (1, 1);
}

int g (int x, int y)              //D4
{
   return printf ("x = %d; y = %d\n", x, y);
}
```

（2）假如在某个转换单元中仅存在以下两个关于 m 的声明：

```
extern int m;
static int m;
```

为什么是非法的？如果将它们的顺序调换，即：

```
static int m;
extern int m;
```

为什么又合法了？

（3）假定在某转换单元里仅存在着以下关于函数 f、g、h 和 k 的声明：

```
extern void f (void);
static void f (void);

void g (void);
static void g (void);

void h (void);
static void h (void) {/* …… */}

extern void k (void);
static void k (void) {/* …… */}
```

为什么这些声明都是非法的？如果将它们的顺序调换，即：

```
static void f (void);
extern void f (void);
```

```
static void g (void);
void g (void);

static void h (void) {/* ····· */}
void h (void);

static void k (void) {/* ····· */}
extern void k (void);
```

为什么又合法了？

4.3.4.1 外部链接

外部链接（external linkage）意味着一个标识符有和同一程序中的某些同名标识符指示相同对象或者函数的潜在可能性，而不管它们是位于同一个转换单元或者同一个库中，还是位于各个不同的转换单元和库中。

换句话说，外部链接是标识符的一个属性，具有外部链接的标识符，它指示的对象可能是由其他外部链接的同名标识符定义；或者，它定义的对象被其他具有外部链接的标识符指示着。

标识符的外部链接属性可以通过以下几个方面予以确定。

首先，一个具有文件作用域的标识符，如果它被声明为对象类型，且其声明中不带任何存储类指定符，那么该标识符具有外部链接属性。因此，在下例中，指示对象的标识符 g 和指示函数的标识符 f 都具有外部链接，而 i 则没有，因为它不具有文件作用域。同时，这两次对 g 的声明都指示了同一个对象。

```
int g;

int f (void)
{
    int i;
    /* ····· */
}

int g = 0;
```

当然，这只是一个代码片段。实际情况是其他地方也可能存在着对 g 的具有外部链接的声明，或者在当前转换单元内的其他地方，或者在另一个转换单元内，也可能是在一个库中。但是，它们都指示同一个对象。

注意，如果同一个标识符的两次或多次声明并不指示同一个对象或者函数，则不能位于同一作用域内（否则属于重复声明或者重复定义）。

其次，若一个标识符是用存储类指定符 extern 声明的，且在它的作用域内能看见该标识符的另一个具有外部链接的声明，那么该标识符也是外部链接的。

假定某个转换单元的全部内容如下：

```
int x = 2;                          //D1
extern int y;                       //D2
void f (void) {/* …… */}            //D3

void demo (void)
{
    extern int x;                   //D4
    x ++;

    if (x > 0)
    {
        extern int x;               //D5
        x ++;

        extern void f (void);       //D6
        f ();

        extern int y;               //D7
         y ++;
    }
}
```

那么，在 D5 处声明的标识符 x，其作用域从它在 D5 中出现的位置开始，延伸并终止于它所在的块的结尾。在它的作用域内，能看到在 D4 中声明的同名标识符 x。同样，在 D4 中声明的标识符 x，其作用域从它在 D4 中出现的位置开始，延伸并终止于它所在的块的结尾。在它的作用域内，能看见 D1 中声明的、具有外部链接的标识符 x。所以，这 3 个标识符 x 都是外部链接的，都指示同一个对象。

同理，在 D6 中声明的标识符 f，其作用域从它在 D6 中出现的位置开始，延伸并终止于它所在的块的结尾。在它的作用域内，能看见在 D3 处声明的、具有外部链接的函数 f，所以，这两个标识符 f 都是外部链接的，都指示同一个函数。

在 D7 中声明的标识符 y，其作用域从它在 D7 中出现的位置开始，延伸并终止于它所在的块的结尾。在它的作用域内，能看见在 D2 处声明的、具有外部链接的标识符 y，所以，这两个标识符 y 都是外部链接的，都指示同一个对象。

进一步地，因为 y 在当前转换单元内没有定义，所以在组成当前程序的其他转换单元或者库内，一定会存在具有外部链接属性的标识符 y。当然，在组成当前程序的其他转换单元或者库内，可能也存在其他具有外部链接属性的标识符 x 和 f，它们和当前转换单元内的 x 和 y 一起，指示同一个对象或者函数。

再次，若一个标识符是用存储类指定符 extern 声明的，且在它的作用域内，看不见该标识符的其他声明；或者即使能看见，也只是被声明为无链接的，那么该标识符是外部链接的。

假定某个转换单元的全部内容如下。

```
void demo (void)
{
    extern int x;                    //D1
    extern void f (int);             //D2
    f (++ x);
}
void f (int m) {/* …… */}            //D3
```

那么，因为在 D1 中声明的标识符 x，其作用域从它在 D1 中出现的位置开始，延伸并结止于它所在的块结束，在它的作用域内看不到其他同名标识符 x 的声明；在 D2 中声明的标识符 f，其作用域从它在 D2 中出现的位置开始，延伸并终止于它所在的块结束，在它的作用域内看不到其他同名标识符 f 的声明，所以，这里的 x 和 f 都是外部链接的，D2 中声明的 f 和 D3 中声明的 f 指示同一个函数。

事实上，因为 x 在当前转换单元内没有定义，所以，在组成当前程序的其他转换单元或者库内，也一定会存在具有外部链接属性的标识符 x。当然，在组成当前程序的其他转换单元或者库内，可能存在其他具有外部链接属性的标识符 f，它们和当前转换单元内的 f，指示同一个函数。

例如，假定组成某个转换单元的全部内容如下。

```
void fscope (void)
{
    int x = 3;                  //D1
    while (x -- > 0)
    {
        extern int x;           //D2
        extern int y;           //D3
        x ++, y ++;
    }
}
```

其中，在 D1 和 D2 中都声明了标识符 x，在 D1 中声明的 x 能够在 D2 的作用域内看见。但是，因为在 D1 中声明的 x 是无链接的（指示独立的对象），所以在 D2 处声明的 x 是外部链接的，它和 D1 处声明的 x 并不指示同一个对象，而是分别指示不同的对象。因为 x 在当前转换单元内没有定义，所以除非在组成当前程序的其他转换单元或者库内具有 x 的外部定义，否则表达式 x++将在程序转换时出错。

同理，在 D3 中声明的标识符 y，其作用域从它在 D3 中出现的位置开始，延伸并终止于它所在的块结束。在它的作用域内，看不见其他有关标识符 y 的声明。加之 y 在当前转换单元内没有定义，所以除非在组成当前程序的其他转换单元或者库内具有 y 的外部定义，否则表达式 y++将在程序转换时出错。

最后，如果一个标识符被声明为函数类型，且它的声明中未使用任何存储类指定符，则视为在声明时使用了存储类指定符 extern。举个例子，假定某个转换单元的全部内容如下：

```
static int f (void) {/* …… */}     //D1
int g (void) {/* …… */}            //D2
```

```
void demo (void)
{
    int f (void);                    //D3
    int g (void);                    //D4
    int h (void);                    //D5
    f (), g (), h ();
}
```

那么，D3、D4 和 D5 都可以认为是用存储类指定符 extern 声明的，即

```
extern int f (void);
extern int g (void);
extern int h (void);
```

在 D3 中声明的标识符 f，其作用域开始于它在 D3 中出现的位置，延伸并终止于它所在的块结束。在它的作用域内，能看到 f 的另一个声明 D1，而且在 D1 中声明的 f 具有内部链接属性，所以 D3 中声明的 f 也具有内部链接属性，它们指示同一个函数。关于内部链接，可参见“内部链接”

在 D4 中声明的标识符 g，其作用域开始于它在 D4 中出现的位置，延伸并终止于它所在的块结束。在它的作用域内，能看到 g 的另一个声明 D2，而且在 D2 中声明的 g 具有外部链接属性，所以 D4 中声明的 g 也具有外部链接属性，它们指示同一个函数。当然，在组成当前程序的其他转换单元或者库内，可能也存在其他具有外部链接属性的标识符 g，它们和当前转换单元内的所有 f 都指示同一个函数。

在 D5 中声明的标识符 h，其作用域开始于它在 D5 中出现的位置，延伸并终止于它所在的块结束。在它的作用域内，看不到其他有关标识符 h 的声明。加之 h 在当前转换单元内没有定义，所以在组成当前程序的其他转换单元或者库内，也一定会存在具有外部链接属性的标识符 h，它们统统指示同一个函数。

在对 C 语言进行标准化之前，对于声明为对象类型的标识符，至少存在 4 种外部链接模型（这里不再一一列举）。标准综合了它们各自的特点，尽可能地满足不同环境的要求，并照顾已经存在的实现。

标准模型要求有且只有一个外部定义（声明中包含了初始化器，或者没有初始化器但未使用存储类指定符），而同一标识符的任何其他外部链接的对象声明都属于纯引用。

假定某程序由 main.c、file1.c 组成，那么，表 4-1 说明了被声明为外部链接的标识符是如何指示同一个对象或者函数的。

这两个文件中都有标识符 g 和 f 的声明（函数 f 在文件 file1.c 中既是声明，又是定义），且都具有外部链接（都具有文件作用域），所以，D1 和 D5 中的 g 指示同一个对象。但是很显然，只有 D5 是定义，其他都是纯引用；D3 和 D6 中的 f 指示同一个函数。D2 和 D4 这两个声明位于同一个文件内，声明的是同一个标识符，且都具有外部链接，所以这两个 m 也指示同一个对象。但是很显然，只有 D4 是定义，D2 被视为一个纯引用。

表 4-1 指示同一个函数或对象的外部链接声明

main.c	file1.c

extern int g; //D1
int m; //D2
int f (int); //D3
 	
int main (void)
{
m = 3;
g = f (m);
/* ..... */
}
int m = 0; //D4
	int g = 0; //D5
	
	int f (int i) //D6
	{
	g = 10086;
	/* ..... */
	return g * i;
	}

下面来看外部链接属性的具体应用。如果一个程序由很多源文件组成，并且希望在这些源文件中共用同一些对象或者一些函数，那么最好的办法是在一个头文件中声明它们，用不用存储类指定符 extern 无所谓。然后，在另一个源文件中集中定义。如表 4-2 所示，假定这两个文件的名称是 public.h 和 public.c。

表 4-2 在头文件中应用外部链接

public.h	public.c
extern int ipub; extern void fpub (int); /* ..... */	int ipub = 0; void fpub (int x) {/* ..... */} /* ..... */

此后，可以在所有需要用到这些对象或者函数的源文件中包含这个头文件，并且在编译程序时，连同 public.c 一起编译即可。假定源文件 file1.c 需要使用上面的对象 ipub，它的内容如下：

```
# include "public.h"

int main (void)
{
    ipub = fpub (0);
    /* ..... */
}
```

源文件 file1.c 在预处理之后成为转换单元，头文件 public.h 中的内容将出现在该转换单元中，且在当前转换单元内看不到它们的其他声明，故它们都是外部链接的。

4.3.4.2 内部链接

和外部链接不同，内部链接属性只在转换单元内部起作用。内部链接（internal linkage）意味着一个标识符有和同一转换单元中的某些同名标识符指示相同对象或者函数的潜在可能性。但是，它不和其他转换单元或者库内的同名标识符有任何关系。

换句话说，内部链接是标识符的一个属性，具有内部链接的标识符，它指示的对象可能是由其他内部链接的同名标识符定义；或者，它定义的对象被其他具有内部链接的标识符指示着。

标识符的内部链接属性可以通过以下几个方面予以确定。

首先，一个具有文件作用域的标识符，如果它被声明为对象类型或者函数类型，且在它的声明中包含了存储类指定符 static，则该标识符是内部链接的。注意，只有文件作用域的函数声明可以包含存储类指定符 static。

在下面的例子中，D1 和 D5 这两个声明中的标识符 x 指示同一个对象。D4 中声明的标识符 x 是无链接的，指示一个独立的对象，因为它不具有文件作用域，和内部链接属性无关（关键字 static 不再指示链接属性，而是具有其他意义，可参见“无链接”、“static”和“存储期”）。另外，D2 和 D3 中的标识符 f 指示同一个函数。

```
static int x;                  //D1
static int f (int);            //D2

int demo (void)
{
    /* ..... */
    return f (++ x);
}

static int f (int m)           //D3
{
    static int x;              //D4
    /* ..... */
}

static int x = 0;              //D5
```

其次，若一个标识符是用存储类指定符 extern 声明的，且在它的作用域内能看见该标识符的另一个具有内部链接的声明，则该标识符也是内部链接的。

假定某个转换单元的全部内容如下：

```
static int x;                          //D1
static void f (int);                   //D2

void demo (void)
{
    extern int x;                      //D3
    void f (int);                      //D4
    f (++ x);
}
```

```
static void f (int x) {/* …… */}   //D5
```

那么，在 D3 中声明的标识符 x，其作用域开始于它在 D3 中出现的位置，延伸并终止于它所在的块结束。在它的作用域内，能够看见 x 的另一个声明 D1，而且它在 D1 中被声明为内部链接。所以，在 D3 中声明的标识符 x 是内部链接的。同时，这两个 x 指示同一个对象。

尽管 D4 没有使用任何存储类指定符，但它是一个函数声明，所以它被看成是用存储类指定符 extern 声明的（参见“外部链接”）。在这种情况下，在 D4 中声明的标识符 f，其作用域从它在 D4 中出现的位置开始，延伸并终止于它所在的块结束。在它的作用域内，能看见 f 的另一个声明 D2，而且它在 D2 中被声明为内部链接。所以，在 D4 中声明的标识符 f 是内部链接的。同时，这两个 f 指示同一个函数。

在 D5 中声明的标识符 f，其作用域开始于它在 D5 中出现的位置，延伸并终止于当前转换单元的末尾。在它的作用域内，只能看见 f 的另一个声明 D2（但看不见 D3），而且它在 D2 中被声明为内部链接。因为它们都是内部链接的，所以指示同一个函数。

注意，内部链接不像外部链接，它不是跨文件（转换单元）的。因此，假如文件 file1.c 和 file2.c 中都有以下文件作用域的声明：

```
static char c;
```

则它们分别指示独立的对象，而不会指示同一个对象。

【思考题】在表 4-3 中给出了两个转换单元 A、B 的内容，为什么 A 是正确的，而 B 不正确？

表 4-3　链接指定

A	B

static int x;
static int f (int);
 	
int demo (void)
{
  /* …… */
  return f (++ x);
}
 	
extern int x;
int f (int m) {/* …… */}
	extern int x;
	
	int f (int m) {/* …… */}
	
	int demo (void)
	{
	  /* …… */
	  return f (++ x);
	}
	
	static int x;
	static int f (int);

4.3.4.3　无链接

无链接（no linkage）意味着一个标识符指示独立的实体，和其他标识符没有任何联系。如果一个标识符的声明符合以下条件之一，则它是无链接的：

（1）不是被声明为对象或者函数，而是声明为其他东西，如 typedef 名、结构标记、联合标记和枚举标记，等等；

（2）被声明为函数参数；

（3）被声明为（指示）一个对象，但只有块作用域且在声明时不包含存储类指定符 extern。

所以下例中，对标识符 x 的声明尽管包含了关键字 static，但这个 static 不指示链接属性，而指示下面将要讲到的对象生存期。因此，标识符 x 是无链接的。

```
void f (void)
{
   static int x;
   /* ..... */
}
```

再如，下例中的标识符 F、t、g、e、Male、Female 和 a 都是无链接的。

```
typedef void F (int, int);
struct t {/* ..... */};
enum e {Male, Female,};

int f (void g (const char *))
{
   char a [] = "Pride and Prejudice.\n";
   g (a);
   /* ..... */
}
```

4.4 常量

从语法上，常量分为整型常量、浮点常量、枚举常量和字符常量，可参见其各自的词条。

每个常量都有它自己的类型，这个类型也是常量值的类型；每个常量都有自己的值，常量的值要求能用这个类型来表示。常量的类型取决于常量的值，以及该常量是以什么形式定义的。

例如，下面的常量分别是 int 和 long long int 类型：

```
5
5LL
```

又如，下面的常量分别是 float 和 double 类型。

```
0.001f
5.6
```

而在下例中，常量 Big 的值无法用它的类型来表示，因为枚举常量的类型是 int，而这里给它的值是 INT_MAX + 1（比 int 类型的最大值大 1）。

```
# include <limits.h>
```

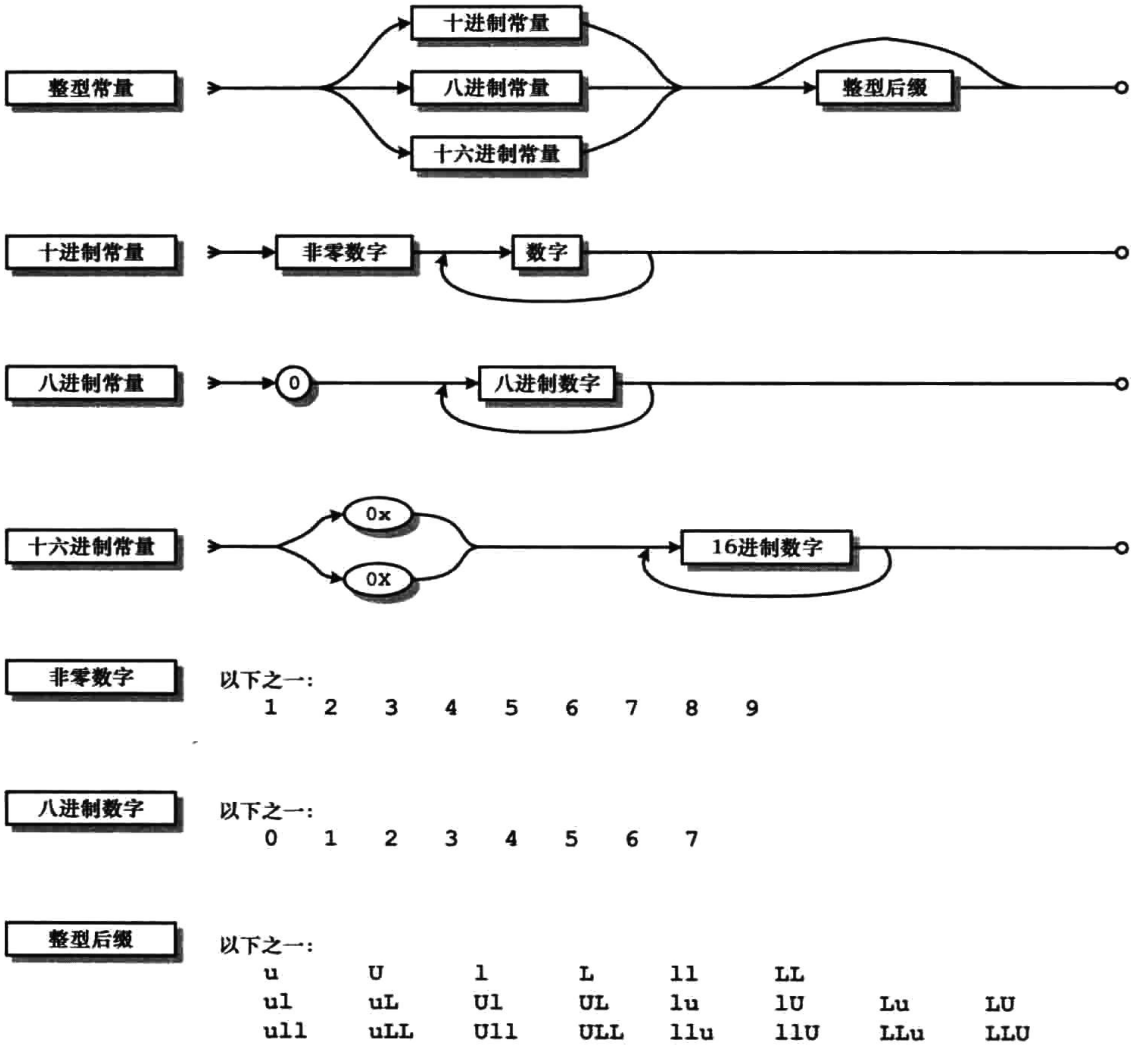
```
enum e {Small = 0, Big = INT_MAX + 1};
```

4.4.1 整型常量

整型常量的语法形式如图 4-2 所示，它可以使用十进制、八进制或者十六进制的形式，区分它们的方法也很简单，即观察其前缀。十进制形式的整型常量以非 0 数字开头；八进制形式的整型常量以数字 0 开头；十六进制形式的整型常量以“0x”或者“0X”开头。

依图 4-2 可知以下都是整型常量。

```
2
2u
3U
0x3fu
0xLLu
073ull
```



注意：“数字”参见图 4-1；“十六进制数字”参见图 2-3。

图 4-2 整型常量的语法形式

如果说整型常量的前缀决定了它的基数（值的计算方法），那么它的后缀决定了其类型。

例如，u 和 U 表示无符号；l 和 L 表示“long”；ll 和 LL 表示“long long”。因此，如果一个整型常量为 5LL，则意味着它的类型是 long long int；如果一个整型常量为 5ULL，则意味着它的类型是 unsigned long long int。

十进制常量不会以“0”打头，而八进制常量必须以“0”打头。有时候，聪明人也会因为一时糊涂而将十进制常量写成八进制常量。比如下面的示例，为了片面追求对齐的效果而将十进制 99 误写成八进制的 099：

```
switch (x)
{
    case 099 : /* …… */
    case 100 : /* …… */
    default : /* …… */
}
```

确定整型常量类型的方法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 4-4 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 类型的最大值定为 32767，而将 long int 类型的最大值定为 2147483647，那么整型常量 5000 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 5000 不能用 int 类型来表示，但可以用第二个备选类型 long int 来表示，所以整型常量 5000 的类型是 long int。

表 4-4　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

如果整型常量的值超出了表 4-4 中任何一种类型所能表示的值的范围，C 实现将尝试用一个扩展的整数类型作为该值的类型。如果连一个可以表示这个值的扩展整数类型都不存在，那么这个值没有类型。

在下例中，整型常量 366666666666666666666667 太大，可能无法为它选取一个合适的类型。在这种情况下，它没有类型。最终这个值依然要转换为 int 类型，但这已经涉及类型转换。

```
int x = 366666666666666666666667;
```

整型常量都是非负数。因此，如以下代码所示，-22 不是整型常量，而是一元表达式，整型常量 22 是运算符-的操作数。

```
long long ll = -22;
```

对于二进制、八进制、十进制和十六进制之间的换算，读者应该已经很熟悉，这些内容在此不再赘述。

4.4.2 浮点常量

浮点常量包括十进制浮点常量和十六进制浮点常量。

使用十进制来表示一个浮点数是很自然的，如-1.25×10^2，其前面是有效数字，后面是以 10 为底的指数，这是我们熟悉的做法。

以前，浮点数在计算机内部的表示并不统一，所以才有了 1985 年的 IEEE 754 二进制浮点标准，它选择 2 作为浮点数指数部分的底，而不是生活中常用的 10。

浮点常量的语法如图 4-3 所示。由图可知，十进制浮点常量 1.025E+3f 就是 1.025×10^3 或者 1025.0；十六进制浮点常量的有效数字使用十六进制的形式，指数部分的底为 2，因此，十六进制浮点常量 0x3.FP+2F 在形式上等价于 $0x3.f\times2^2$，将有效数字部分换算成十进制后是 3.9375，再乘以 2 的 2 次方，等于十进制数 15.750。浮点常量的十六进制形式有其方便之处，它可以用一种非常规整和直观的形式指定一个取值范围，比如 0.00000P0～0.FFFFFP0。

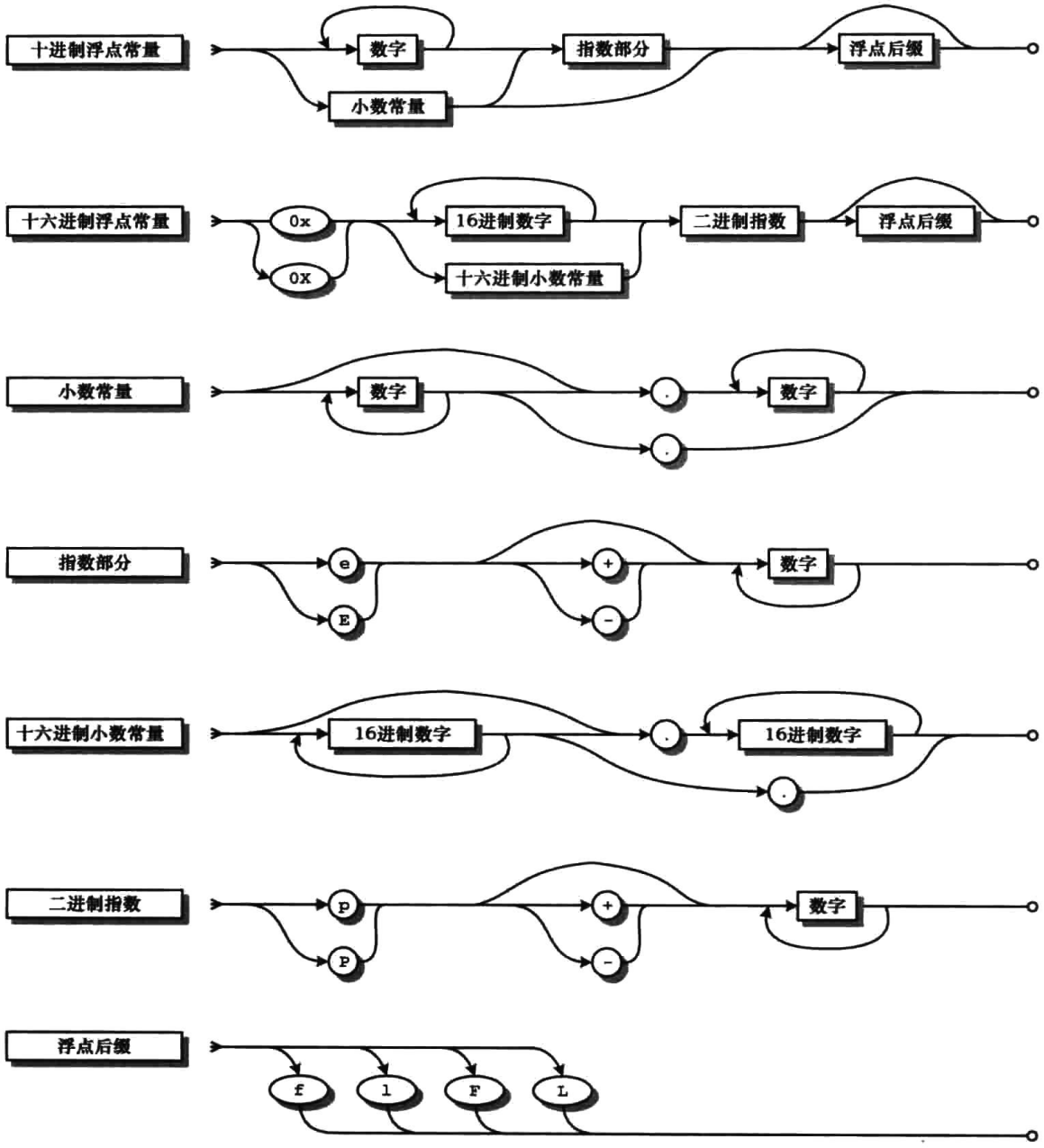
浮点常量的十六进制形式是从 C99 开始引入的，为了同十进制形式有所区别，使用 p 或者 P 将小数和指数部分隔开。下面是其他一些有效的浮点常量。

```
.25
2.5
2.
3.0f
5.F
0.
.0
0.00036L
220e-3
1.5E2F
0xa.p2
0X5b.P-1f
0x.6p+2F
```

问题是，在有些计算机上，浮点数的指数部分并非以 2 为底，典型的是 IBM 公司生产的 S/370。以 2 为底当然好，但如果底大于 2，也是允许的。无论如何，C 实现都会在头文件<float.h>中将底的数值定义成一个名为 FLT_RADIX 的宏（以下宏定义中的数字 2 只是一个示例，具体的数值可能大于 2）。

```
# define FLT_RADIX  2
```

综上所述，浮点常量的有效数字部分是有理数，使用十进制或者十六进制表示；指数部分是十进制整数。十进制浮点常量的指数部分以 10 为底，十六进制常量的指数部分以 2 为底。指数部分对于十进制形式的浮点常量来说是可选的，但对于十六进制形式的浮点常量来说是必需的。原因在于，有效数字部分可能以“f”或者“F”结尾，如果省略指数部分，则会同浮点后缀混淆。



注意：“数字”参见图 4-1；“十六进制数字”参见图 2-3。

图 4-3 浮点常量的语法

如果浮点常量没有后缀，则它的类型是 double；如果后缀是 f 或者 F，它的类型是 float；如果后缀是 l 或者 L，它的类型是 long double。在下例中，1.0f 是 float 类型的浮点常量；2.0 是 double 类型的浮点常量；3.0l 是 long double 类型的浮点常量。

```
long double m = 1.0f + 2.0 + 3.0l;
```

和整型常量一样，浮点常量没有负数。因此，如以下代码所示，-0x.625p+2F 不是浮点常量，而是一元表达式，浮点常量 0x.625p+2F 是一元运算符-的操作数。

```
long double ld = -0x.625p+2F;
```

4.4.3 枚举常量

每一个枚举都是一些标识符的集合，而其中的每一个标识符都被视为一个常量，称为枚举常量（enumeration constant）。

从另一方面来说，枚举类型的成员称为枚举常量。在下例中，Red、Green 和 Blue 称为枚举常量。

```
enum {Red = 0, Green, Blue};
```

每个枚举常量都有自己的值，值的类型是 int。实际上，也可以说枚举常量的类型是 int。枚举常量的值如何确定，以及每个枚举类型可以定义多少个枚举常量，可参见“枚举类型”。

4.4.4 字符常量

字符常量的语法如图 4-4 所示，这实际上将字符常量分成两类：整型字符常量和宽字符常量。

整型字符常量的类型是 int，在形式上是由一对单引号括起来的、一个以上的多字节字符组成的序列。下面是一些整型字符常量的例子，分别采用的是源字符集成员、简单脱转序列、统一字符名、八进制脱转序列和十六进制脱转序列。

```
'x'
'abc'
'\?'
'\u3021'
'喜'。
'\033'
'\x3f'
```

一个比较容易犯的错误是将整型字符常量，如'y'的类型看作 char，这当然是不对的。示例：

```
char c = 'y';
```

其中，'y'的类型是 int，它被转换为 char 类型的值，用于初始化对象 c。因为这个原因，给定声明：

```
char c = 'x' , f (char);
```

则函数调用

```
f ('x')
```

是将 int 类型的'x' 转换为 char 类型之后才传递给函数 f；而函数调用

```
f (c)
```

是直接将 c 的值传递给函数 f。进一步地，

```
sizeof 'x' == sizeof (int)
sizeof c == sizeof (char)
```

都是成立的，但是，

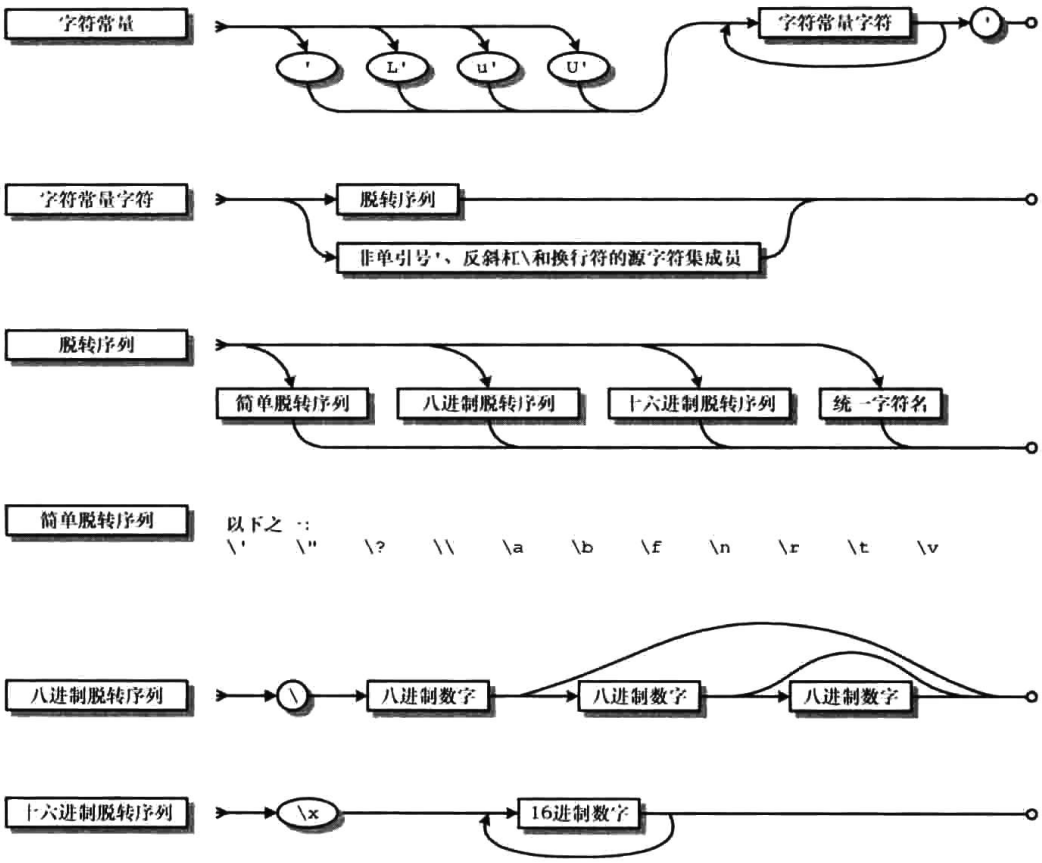
```
sizeof 'x' == sizeof (char)
sizeof c == sizeof (int)
sizeof 'x' == sizeof c
```

都不成立。

宽字符常量在形式上和整型字符常量相似，但带有一个前缀，该前缀可以是“L”、“u”或者“U”。如果前缀是“L”，则宽字符常量的类型是 wchar_t；如果前缀是“u”，则类型是 char16_t；如果前缀是“U”，则类型是 char32_t，如 L'x'、u'福'，等等。

字符常量的组成在图 4-4 中描述得足够清楚。其中，\'表示单（个）引号；\"表示双引号；\?表示问号（?）；\\表示反斜杠（\），\a、\b、\f、\n、\r、\t 和\v 可参见“脱转序列”。

如果字符常量使用了八进制脱转序列，则最多允许使用 3 个八进制数字，具体长度以遇到的第一个非数字字符为准（以最多 3 个为限）；如果采用了十六进制脱转序列，则构建脱转序列时的长度没有限制，具体长度以遇到的第一个非十六进制数字为准，即它们都应以最长的合法序列为原则。



注意：“统一字符名”和“十六进制数字”参见图 2-3；“八进制数字”参见图 4-2。

图 4-4　字符常量的语法

在以下的例子中，八进制脱转序列最多只允许使用 3 个八进制数字，而十六进制脱转序列没有限制。

```
int a = '\12a';              //脱转序列为\12
int b = '\03371';            //脱转序列为\033
int c = '\xdefault';         //脱转序列为\xdefa
int d = '\xfghijk';          //脱转序列为\xf
int e = '\xabcd00375g';      //脱转序列为\xabcd00375
```

在形式上没有问题，并不意味着一个八进制和十六进制脱转序列的值没有限制。事实上，如果字符常量没有前缀，则八进制和十六进制脱转序列的值要求能用 unsigned char 类型表示；如果有“L”前缀，则八进制和十六进制脱转序列的值要求能用 wchar_t 来表示；如果有“u”前缀，则要求能用 char16_t 类型表示；如果有“U”前缀，则要求能用 char32_t 类型来表示。

下面介绍字符常量的值。在最简单的情况下，整型字符常量仅包含单个字符，如'a'，或者一个脱转序列，如'\21'。如果它们能够被映射为单字节的执行字符，则整型字符常量的值就是映射后的数值，就像映射后的值位于一个 char 类型的对象中，然后被转换为 int 类型一样。

但是，如果整型字符常量中包含的字符多于一个（如'ab'），或者包含的字符和脱转序列不能映射为单字节的执行字符（如'人'、'刍狗'），此时，整型常量的值是什么由 C 实现来决定（和系统架构以及执行字符集有关）。

再来看宽字符常量的值，宽字符常量在物理源文件中的原始形态是一个多字节字符，或者多字节字符的序列（这是由源字符集决定了的，毕竟源字符集的成员只能是单字节字符、多字节字符而非宽字符）。比如宽字符常量 L'人'、u'刍狗'或者 U'阿 Q'，它们在源文件中被编码为一个多字节字符或者多字节字符的序列。

在程序转换阶段，如果宽字符常量的原始形态只是一个多字节字符，则它的值是与该多字节字符对应的宽字符，前提是这个对应的宽字符在执行字符集中是存在的。若宽字符常量的前缀是“L”，转换的方法如同调用库函数 mbtowc；若前缀是“u”，如同调用库函数 mbrtoc16；若前缀是“U”，则如同调用库函数 mbrtoc32。举个例子，若指定的源字符集是 UTF-8，指定的执行宽字符编码是 UTF-16LE，则源文件中的字符常量 L'人'将从 3 个字节的多字节字符转换为 16 位的宽字符。

相反地，如果对应的宽字符在执行字符集中不存在，或者对应多个成员，又或者宽字符常量的原始形态是多个多字节字符，则宽字符常量的值取决于 C 实现。

就像我们已经知道的那样，字符常量，或者说脱转序列'\0'通常用于表示一个空字符。

4.5 字面串

广义上的（字符）串（string）是一个单字节字节的序列，以第一个遇到的空字符终止，或者是一个多字节字符的序列，以第一个遇到的空字符终止，又或者是一个宽字符的序列，以第一个遇到的空宽字符终止，参见“字符串”和“宽字符串”。

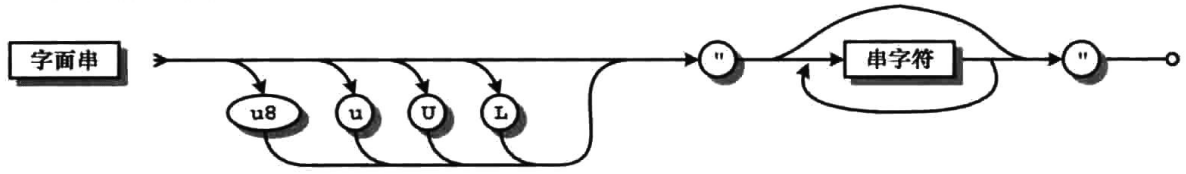
串是数组类型的对象，为了创建这个数组，必须在源文件中有所体现，即以字面的形式出现，这就是所谓的字面串（string literal）。当然，它也只是一个词法记号。

换句话说，从字面串可以得到一个串。字面串的语法形式如图 4-5 所示，下面是 3 个字面串的例子。

```
""
"hello"
"楚国 800 年"
```

字面串分为三种，如图 4-5 所示，不带任何前缀的字面串称为字符字面串；如果字面串带有“u8”前缀，则它是一个使用 UTF-8 编码方案的多字节字面串；如果字面串的前缀是“L”、“u”或者“U”，则它是一个宽字面串。以下是这几种字面串的例子。

```
u8""
u8"hello"
L"嘻，技止此耳！"
u"xyz"
U"标准 C 语言指南"
```



串字符：① 双引号（“”）、反斜杠（\）以及换行符之外的源字符集成员；

② 脱转序列（参见“脱转序列”及图 4-4）。

图 4-5　字面串的语法

如果程序中使用了字面串，那么它将用于在程序转换期间初始化一个不可见的静态数组，而且这个数组的大小恰好足以容纳字面串经下述转换后的内容。

因为字面串的原始形态是多字节字符的序列（这是由源字符集决定了的，毕竟源字符集的成员只能是多字节字符而非宽字符），在程序转换阶段，C 实现在这个序列的尾部追加一个空字符。然后：

（1）如果字面串没有前缀，则用这个序列以字节为单位初始化一个不可见的静态数组，数组的元素类型是 char;

（2）如果字面串的前缀是“u8”，则将这个序列中的多字节字符转换为对应的 UTF-8 编码，并用于初始化一个不可见的静态数组，数组的元素类型是 char;

（3）如果字面串的前缀是“L”，则将这个序列中的多字节字符转换为对应的宽字符（转换的方法类似于使用库函数 mbstowcs 来进行），并用于初始化一个不可见的静态数组，数组的元素类型是 wchar_t;

（4）如果字面串的前缀是“u”，则将这个序列中的多字节字符转换为对应的宽字符（转换的方法类似于使用库函数 mbrtoc16 来进行），并用于初始化一个不可见的静态数组，数组的元素类型是 char16_t;

（5）如果字面串的前缀是“U”，则将这个序列中的多字节字符转换为对应的宽字符（转换的方法类似于使用库函数 mbrtoc32 来进行），并用于初始化一个不可见的静态数组，数组的元素类型是 char32_t。

下面这个例子看起来简单，但从字面串的角度来看并不简单，因为字面串"hello, world.\n"不是原样传递给函数 printf 的，它首先在程序转换阶段用于初始化一个看不见的静态数组，在调用 printf 函数时，这个数组转换为指向其首元素的指针，并传递给 printf 函数。也就是说，后者仅仅得到了一个指针。

```
# include <stdio.h>

void f (void)
{
   printf ("hello, world.\n");
}
```

字符串和字面串毕竟不是一个概念，字符串以空（宽）字符终止，在这个空（宽）字符的前面不能有其他空（宽）字符，但字面串内部可以包含空（宽）字符。例如，在下列中，"To be\0 or not to be."是字面串，但是中间的'\0'将得到一个很短的串，其内容为'T'、'o'、' '、'b'、'e'和'\0'。

```
# include <stdio.h>

void f (void)
{
   char a [] = "To be\0 or not to be.";
   printf ("%zu, %s", sizeof a, a);
}
```

下面是字面串应用的一个示例，函数 is_keyword 接受一个指向字符串的指针，然后判断该指针所指向的字符串的内容是否为 C 语言关键字。

数组 keywords 的元素类型为 char，而且这个数组是用字面串来初始化的，所以数组的内部有很多空字符。

注意，两个字符串在比较时，只有在它们的内容完全相同时，指向它们的指针在比较之后才会都指向字符串末尾的空字符“\0”，这个可以做为判断两个字符串是否相同的条件和标志。

```
const char keywords [] = "auto\0break\0case\0char\0const\0"
                         "continue\0default\0do\0double\0"
                         "else\0enum\0extern\0float\0for\0"
                         "goto\0if\0inline\0int\0long\0"
                         "register\0restrict\0return\0"
                         "short\0signed\0sizeof\0static\0"
                         "struct\0switch\0typedef\0union\0"
                         "unsigned\0void\0volatile\0while\0"
                         "_Alignas\0_Alignof\0_Atomic\0"
                         "_Bool\0_Complex\0_Generic\0"
                         "_Imaginary\0_Noreturn\0"
```

```
                    "_Static_assert\0_Thread_local";

_Bool is_keyword (const char * str)
{
    const char * p = keywords, * q;

    while (q = str, p < keywords + sizeof keywords)
    {
        while (* p != '\0' && * q != '\0')
            if (* p != * q) break; else p ++, q ++;

        if (* p == '\0' && * q == '\0') return 1;
        else while (* p ++!= '\0') ;
    }

    return 0;
}
```

为了方便，有时直接将字面串当做数组。在下面的例子中，函数 uitoh 用于将一个无符号整数转换为十六进制的形式输出。转换的过程大家都很熟悉，即不停地除以 16，得到的余数是各个数位，而商则用于递归地得到其他数位。因为字面串的类型是数组，故它可以直接用于下标运算。

```
# include <stdio.h>

void uitoh (unsigned int u)
{
    if (u / 16) uitoh (u / 16);
    printf ("%c", "0123456789ABCDEF" [u % 16]);
}
```

字面串可以作为初始化器，用于初始化一个数组。这个数组有可能被存放在存储器中的一个只读区域，因此，企图修改这个数组内容（元素）的行为是未定义的。

如下例所示，因为 C 实现已经用字面串"oops!"创建了一个静态数组（如前面所述），所以它的工作是按声明的要求创建一个数组 a，并用静态数组的内容初始化 a 的各个元素。

```
char a [] = "oops!";
```

这实际上等效于：

```
char a [] = {'o', 'o', 'p', 's', '!', '\0'};
```

当然，只是等效（意思是从数组 a 被初始化后的内容来看，效果一样），不是等价，因为这里面还有一个不可见的静态数组未体现出来。因此，它实际上等价于（假定 C 实现会为所有字面串"oops!"创建一个共同的实例）

```
char a [sizeof "oops!"];
```

```
for (size_t x = 0; x < sizeof "oops!"; x ++) a [x] = "oops!" [x];
```

或者（假定C实现会为所有字面串"oops!"创建一个共同的实例）

```
char a [6];
a [0] = "oops!" [0];
a [1] = "oops!" [1];
a [2] = "oops!" [2];
a [3] = "oops!" [3];
a [4] = "oops!" [4];
a [5] = "oops!" [5];
```

进一步地，如果有如下代码：

```
char b [5] = "oops!";
```

则它实际上等效于：

```
char b [5] = {'o', 'o', 'p', 's', '!'};
```

这是因为数组 b 只有 5 个元素，所以最后的空字符无法存放。有关数组初始化的内容，可参见“初始化器”。

再来看另一个例子，尽管C实现依然会用字面串"oops!"创建一个静态数组（如前面所述），但是由于 p 的类型是指针，所以这个静态数组要用于初始化指针。在这种情况下，这个静态数组将被转换为指向其首元素的指针，然后用这个指针来初始化对象 p。

```
char * p = "oops!";
```

下面是另一个相似的例子。

```
char * f (void) {return "Be there or be square.";}

int g (char * s) {/* …… */}

int main (void)
{
   return g (f ());
}
```

上面的程序可以正确工作。字面串"Be there or be square."是具有静态存储期的数组，其类型为 char [23]，它的生存期开始于程序启动之前，随着程序的执行结束而终止。

因为函数的返回类型是 char *，所以字面串将被转换为指向其第一个元素的指针，该指针作为函数的返回值是安全的，毕竟它所指向的对象是数组的首元素，而此数组具有静态存储期，在整个程序的生存期内都保证存在。

如果两个以上的字面串具有完全相同的内容，则它们有可能被创建为同一个静态数组对象，而不是被创建为独立的静态数组对象。

如果在程序中的不同位置都出现了完全相同的字面串，则它们可能对应着不同的静态数组，也可能对应着同一个静态数组。再者，它们可能会被安置在一个只读的存储区域中。因此，修改一个字面串所对应的静态数组的行为是未定义的。

在下例中，6 个"Casablanca,1942"可能对应独立的静态数组，也可能是部分独立的，还

可能对应着同一个静态数组。也正是因为这个原因，在 S1 中，尽管两个指针的比较是允许的，但比较的结果却是未指定的，可能相等，也可能不相等。而在 S2 中，企图修改字面串所对应的静态数组是非法的，其行为是未定义的；在 S3 中，printf 函数输出的结果也是未指定的，可能是三个相同的地址，也可能是完全不同的地址，也可能是部分相同的地址。

```
# include <stdio.h>

void f (void)
{
    if ("Casablanca,1942" == "Casablanca,1942") {/* …… */}     //S1
    "Casablanca,1942" [0] = 'x';                                //S2
    printf ("%p\t%p\t%p\n", (void *) "Casablanca,1942",
                            (void *) & "Casablanca,1942",
                            (void *) & "Casablanca,1942" [0]);  //S3
}
```

原则上，互相邻接的字面串可以黏接在一起，这将得到一个完整的、单一的字面串。但是，不允许宽字面串和 UTF-8 字面串的黏接。

关于相邻字面串的黏接，下面是一个很好的示例。

```
char * s = "Long long ago,"
           "there lived in Greece "
           "a" " great thinker "
           "named Aristotle." "\n";
```

在程序转换阶段的黏接后，上面各行等效于：

```
char * s = "Long long ago,there lived in Greece a great thinker named Aristotle.\n";
```

有关不同前缀的字面串黏接，下例中的 D1 是合法的，而 D2 则不合法。

```
char a [] = "Hello," "Kitty.";          //D1
char b [] = L"hello" u8"world.\n";      //D2
```

不管黏不黏接，最终形成的单个字面串，其长度不能超过 4095 个字符，超出这个限制可能不受支持。

4.6　注释

C 源文件可以包含注释，注释分为多行注释和单行注释。多行注释可以跨越多行书写，由/*开始，并由与之配对的*/结束，中间是注释的具体内容。单行注释被限制在一行之内，由//开始引入注释的内容。

注释对程序的执行没有任何作用，它仅仅是一些供人阅读的文本，通常用来解释程序的功能和意图，这对于程序的维护是有好处的。

多行注释的例子如下。

```
/*
** 今天的天气很好，阳光明媚，真是适合编写程序啊
** 这是我编写的第一个C程序，具有纪念意义
*/
```

单行注释的例子

```
int x = 0;  //这是一个单行注释
```

在程序转换期间，注释将被删掉，并用一个空格字符代替。

在程序转换期间，对注释的处理早于源文件的句法和语义分析，以及可执行代码的生成。这意味着，即使将

```
return 0;
```

写成

```
return/**/0;
```

也没有任何问题。尽管 return、/**/和 0 之间没有加入空格符，但注释被删掉之后，依然会在 return 和 0 之间插入空格符。

在传统方式中，字符序列“/*”引入一个注释。后面是注释的具体内容，直至遇到另一个字符序列“*/”为止，这是注释的结束标志。但是，当这两个序列出现在字符常量、字面串的内部，或者位于另一个注释的内部时，它们并不被认为是注释。

下面是一个正确的注释，也是另一个多行注释的例子，只是看起来有些复杂。

```
/*****************************************
**程序名称：main.c
**编写日期：2015-1-1
**作者：你可管我
*****************************************/
```

因为字符序列“*/”是注释结束的标志，所以这种形式的注释不能嵌套。下面的注释将在程序转换期间产生一个意味着错误的诊断信息。

```
/* 注释以/*开始，以*/结束*/
```

在下面的例子中，因为字符序列“/*”和“*/”出现在字面串中，所以 C 实现不将它们视为注释的开始和结束标志。

```
printf ("A comment include /* and */.\n");
```

在程序转换时，注释将被删除，并用空格字符代替。因此，下面的代码

```
f = g ()/**/* h ();
```

等效于：

```
f = g () * h ();
```

如果一个人的注意力只集中在运算符和指针运算上面，再加上组成程序的标点符号缺少空白和间隔，就会出一些莫名其妙的问题，例如：

```
int x, * pi;
/* ..... */
x = 250/*pi;
```

最后一行是用 250 除以*pi，即用 250 除以 pi 所指向的对象的值。但 C 实现将“/*”看做

注释的开始。所以，良好的写法应当如下。

```
x = 250 / * pi;
```

从C99开始，字符序列“//”引入一个单行注释，后面是注释的内容，一直延伸到（但不包括）新行符（通常是不可见的行终结符）。但是，当这个序列位于字符常量、字面串的内部，或者位于另一个注释的内部时，它并不被认为是注释的开始。

也就是说，“//”用于注释单行文本。以下是一个（有效的）使用//引入注释的例子。

```
f ();   //领导说将函数命名为f太不规范，是缺乏编程素养的表现
```

注意，C 实现在转换一个程序时，对续行符的处理早于对注释的处理。也就是说，要先将物理源行合并成逻辑源行。如果合并之后的内容包括注释，则将注释替换成空白字符。在下面的例子中，注释中包括续行符：

```
i++;    //The value computation of the result is sequenced \
before the side effect ofupdating the stored value of the \
operand.
j++;
```

以上程序片断在转换时，将首先把物理源行合并成逻辑源行，并删除注释（用空白字符替换），只保留以下两条语句。

```
i++;
j++;
```

带有换行符的注释可能会引起问题，特别是当人们只专注于反斜杠本身的含义时。下面的片断摘自一个在 Windows 环境下转换和执行的程序，程序员使用了一个单行注释，但总是无法正确设置默认的路径。

```
//为了安全起见，必须明确设置当前的默认路径到C:\
SetDefaultPath("C:\\");
```

这是因为注释行的末尾是“\”，于是下面一行代码也变成了注释的一部分。

类似地，下面的片断最终得到了两行注释，你可以自己分析一下为什么。

```
//\
i++;
/\
/ f();
```

如果注释以“//”开始，则 C 的实现期望看到一个结束注释的新行符，而不是“*/”。因此，下面的注释没有问题：

```
//*/
```

而如下两行代码

```
x=9//**/3
+2;
```

等效于：

```
x=9
+2;
```

或者

```
x=9+2;
```

相反地，如果注释以“/*”开始，则 C 的实现将不理会“//”，而是寻找一个结束注释的“*/”。例如：

```
/*//*/f();
```

等效于

```
f();
```

下面的“//”出现在字面串内部，因此不会被当作注释处理。

```
"a//b"
```

4.7 关键字

关键字（keyword）是指那些具有固定拼写且大小写敏感的记号，它们只能用于特定的目的而不能作为普通标识符使用。C 中的关键字如下。

```
auto            break           case            char
const           continue        default         do
double          else            enum            extern
float           for             goto            if
inline          int             long            register
restrict        return          short           signed
sizeof          static          struct          switch
typedef         union           unsigned        void
volatile        while           _Alignas        _Alignof
_Atomic         _Bool           _Complex        _Generic
_Imaginary      _Noreturn       _Static_assert  _Thread_local
```

带下画线的关键字是在较新的标准中引入的，因为出现得较晚，而在此之前人们已经编写了大量的 C 程序，下画线和首字母大写可以降低和已有的 C 程序中的标识符发生冲突的几率。例如，bool 有可能已经被某些程序当做标识符，用_Bool 就会好一些。

下面的关键字是从 C99 开始引入的：_Bool、_Complex、_Imaginary、inline 和 restrict；

下面的关键字是从 C11 开始引入的：_Alignas、_Alignof、_Atomic、_Generic、_Noreturn、_Static_assert 和_Thread_local。

关键字和标识符都是大小写敏感的。这意味着声明

```
int Int, INt, INT, iNt, iNT;
```

是合法的，这里声明了 5 个不同的标识符。

关键字是 C 语言保留的。不过，尽管不能定义和关键字相同的标识符，却可以定义和关键字相同的宏名。这是因为在程序转换阶段，预处理在前，语义和语法分析在后。

这里不准备详细解释每一个关键字的功能和使用方法，它们在本书中可以找到。如果读者已掌握了书中的其他部分，自然也就知道了这些关键字的功能和用法。

第 5 章　声明和定义

5.1　声明

声明（declaration）的语法如图 5-1 所示。通常，声明只是一个说明，是对一个标识符的解释，为标识符指定某些属性，诸如类型、存储期和链接等。

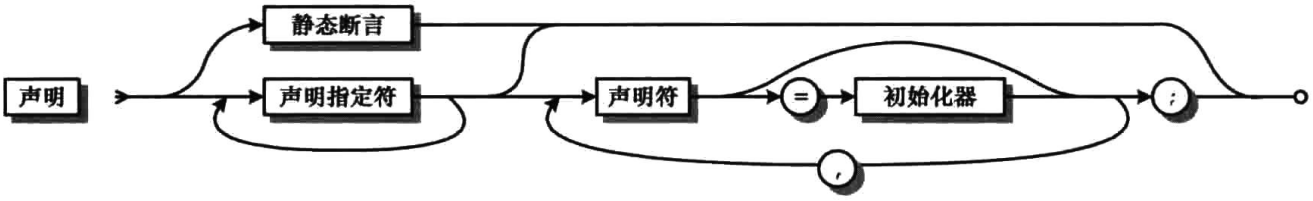


图 5-1　声明的语法

图中所有的非终止符，如“静态断言”、“声明指定符”、“声明符”和“初始化器”，均可参见其各自的词条。为了对这张图有一个感性认识，这里举几个声明的例子。

```
# include <stdio.h>

int x = 0, f (int), (* g) (const char * restrict, ...) = printf;
struct t {char c; float f;};
char a [] = {0, 1, 2, 3,};
enum {Male, Female,};
```

以上代码声明了指示对象的标识符 x（简称对象 x）、指示函数的标识符 f（简称函数 f）、指示对象的标识符 g（它是指向函数的指针，而指针也是完整对象类型）、结构标记 struct t、指示数组对象的标识符 a，以及枚举常量 Male 和 Female。

以上声明中，声明指定符如下：

```
int
struct t {char c; float f;}
char
enum {Male, Female,}
```

声明符如下：

```
x
f (int)
(* g) (const char * restrict, ...)
a []
```

初始化器如下（其中{0, 1, 2, 3,}还可以继续分解为 0、1、2 和 3）：

```
0
```

```
printf
{0, 1, 2, 3,}
```

首先，如图中所示，声明可以是一个静态断言。静态断言是从 C11 开始引入的，具体描述请参见“静态断言”；

其次，被声明的实体可以是 typedef 名，也就是一个现有类型的别名。参见“类型定义”和“typedef”。下例中，typedef 名 F 是一个函数类型的别名；typedef 名 Integer 是 int 类型的别名；typedef 名 PChar 是 char *类型的别名。

```
# include <stdio.h>

typedef int F (int, int, char *, char *);
typedef int Integer;
typedef char * PChar;

F * demo (void)
{
    Integer i = 1, j = 2;
    PChar s1 = "SPG-422", s2 = "ECO-422D";
    F my_f;
    my_f (i, j, s1, s2);
    return my_f;
}

int my_f (int x, int y, char * p, char * q)
{
    return printf ("%d, %d, %s, %s\n", x, y, p, q);
}

int main (void)
{
    demo () (5, 6, "7", "8");
    return 0;
}
```

再次，被声明的实体也可以是对象。此时，声明的是一个指示对象的标识符。如下例所示，除了联合类型 u 的声明外（它仅仅被认为是一个联合类型及标记 u 的声明），其他都是对象的声明。其中，a 是数组对象；t 是结构对象。这里有两个关于对象 i 的声明，但它们其实指示同一个对象，因为它们都是外部链接的，指示同一个对象（参见“链接”）。

```
int i, a [22];
struct t {char c; float f;} t;
union u {struct t; long double l;};
extern int i;
```

第四，被声明的实体还可以是函数。此时，声明的是一个指示函数的标识符。如果函数的声明也是一个定义，则还包括参数的声明。在 C 中，要求函数必须先声明再使用。传统的 C 语言和标准的 C 语言使用不同的函数声明方式，下面将分别予以介绍。如果在函数的声明中包括了函数体，则函数声明就是函数定义。

以下都是函数声明，有的声明没有函数体。带有函数体的声明称为函数定义（参见“定义”和“函数定义”）。这里有两个关于函数 f 的声明，一个有函数体而另一个没有，但它们其实指示同一个函数。至于其中的原因，可参见“链接”。

```
int f (void);
int g (void) {return f ();}
int f (void) {/* …… */}
```

函数在调用之前应当声明。在传统的 C 语言（指标准化之前的 C，又称 K&R C）中，函数的声明很简单，即

```
int f ();
```

这样就声明了一个叫做 f 的函数，它的返回类型是 int，但缺乏与参数有关的任何信息。事实上，这个函数可能是有参数的，但无法知道参数的数量和类型。正是因为如此，当调用这个函数时，C 实现不检查传入的参数数量和类型。

```
int f ();

int main (void)
{
    f ();                          //允许
    f (1);                         //允许
    return f (2, 'x', & main);     //允许
}

int f () {/* …… */}
```

如上例所示，因为函数 f 的声明采用了传统的形式，所以使用的 3 种调用方式都能通过编译，但实际上我们可以看到，函数 f 被定义为根本不接收任何参数。如果传入的参数在数量和类型上与函数的定义不符，其行为是未定义的。

传统形式的函数定义也很特别，参数声明夹在声明符和构成函数体的复合语句之间，除了上面的示例外，下面还有一个例子。

```
int f ()
{
    /* …… */
}

int g (x, y)
char x; long y;
{
    /* …… */
```

```
}
```

以上给出了两个函数 f 和 g 的定义。函数 f 不接收任何参数；函数 g 接收两个参数，第一个参数的类型是 char，第二个参数的类型是 long int。需要注意的是，在传统形式的函数定义中，参数的声明位于声明符和函数体的左花括号之间。

标准 C 语言使用函数原型，这是一种更好的声明形式，且有助于 C 实现对函数调用过程进行优化。下面是一个函数原型的例子，由上面的例子改写而成。

```
int f (void);
int g (char, long);

int main (void)
{
    f ();                          //允许
    f (1);                         //非法
    f (2, 'x', & main);            //依然非法
    g ();                          //非法
    return g ('x', 10086);         //允许
}

int f (void) {/* …… */}
int g (char x, long y) {/* …… */}
```

很显然，如果采用原型，则必须指明参数的数量和类型，即使用所谓的参数类型列表。如果函数不接收任何参数，则其参数类型列表要求使用关键字“void”。在程序转换阶段，C 实现也将更严格地对传入的参数进行检查。有关函数原型的其他描述，可参见“函数原型”。

最后，被声明的实体可以是结构、联合或者枚举类型的标记，如果声明中带有成员列表的，还包括成员的声明。枚举类型的成员称为枚举常量。如下例所示，第一行声明了结构类型的标记 struct t，同时也声明了与该标记相关联的结构类型；第二行声明了联合类型的标记 union u，同时也声明了与该标记相关联的联合类型；第三行声明了枚举类型的标记 enum Component，同时也声明了与该标记相关联的枚举类型。

```
struct t {char c; float f;} t;
union u {char c; float f;};
enum Component {Diode, Triode, Tube,};
```

从图 5-1 可以看出，声明的语法中包括一些可选的部分。结构、联合和枚举类型的声明可以没有声明符，其他类型的声明都要有声明符；带有声明符的声明可以指定初始化器。

依据图 5-1 所构造的声明未必全部都是合法的，因为它还需要一些附加的规则和限制，详见本章的其他部分。下面是两个非法声明的例子：

```
int;
struct {int i; char a [22];};
```

这里，第一个声明之所以有问题，是因为它的声明指定符是 int，换句话说，它声明的不是标记，也不是枚举常量，所以必须有声明符，而它没有；第二个声明之所以有问题，是因为声明指定符指定了一个结构类型。按要求，该声明要么有标记，要么有声明符，或者兼而

有之，例如：

```
struct s {int i; char a [22];};     //具有标记 struct s
struct {int i; char a [22];} s;     //有声明符 s
struct t {int i; char a [22];} t;   //既有标记 struct t 也有声明符 t
```

声明符包含了被声明的标识符。但是，从前面的示例可以看出，声明符可能不仅仅只是一个标识符；或者，不仅仅只有标识符。

多次声明同一个标识符，且使它们具有相同的作用域和名字空间，这是可能的。根据它们的链接属性，可以分为如下两种情形。

（1）它们用于通过内部链接或者外部链接来指示同一个对象或者函数。在这种情况下，它们的类型必须兼容。例如：

```
extern int m;
int f (int, int);
/* …… */
int f (double d) {/* …… */}
int m;
```

其中，对标识符 m 的两次声明都使用了 int 类型，作用域相同，类型也兼容，都是外部链接的，因此指示同一个对象；函数 f 的两次声明都具有文件作用域，但它们在类型上是冲突的，因为参数类型不同。

（2）如果它们都是无链接的，则只能是 typedef 名和结构、联合与枚举的标记。如果是 typedef 名，则它的每次声明都必须具有相同的类型，同时不能是可变修改类型；如果是标记，则同样有一些约束条件，具体请参见“结构或联合指定符”及“枚举指定符”。

为了帮助读者理解这一规则，下面给出了一个非常有用的实例：

```
void f (int n)
{
    typedef int INT;                //D1: 允许
    typedef int INT;                //D2: 允许
    typedef signed INT;             //D3: 允许
    typedef unsigned INT;           //D4: 非法
    typedef float VLA [n];          //D5: 允许
    n ++;
    typedef float VLA [n];          //D6: 非法
    struct t {char c; float f;};    //D7: 允许
    struct t {char c; float f;};    //D8: 非法
    struct t;                       //D9: 允许
    struct t;                       //D10: 允许
    double d;                       //D11: 允许
    double d;                       //D12: 非法
}
```

同一个标识符的多次声明能够以相同的名字空间和作用域共存吗？typedef 名可以，但需要它的每次声明都指定相同的类型。因此，D1、D2 和 D3 都没有问题，毕竟 int、signed、signed int 都是同一种类型。但是 D4 有问题，它不能出现在这里，因为它的类型是 unsigned，与前面的定义冲突。

对 typedef 名的另一个限制是，要想使同一个 typedef 名的多次声明都具有相同作用域，则其不能是可变修改类型。所以，D5 没有问题，但 D6 有问题。

D7 和 D8 不能共存，因为这里使用了标记 t，这意味着它们的类型相同。用户可以多次声明同一种结构类型，但只能有一个带有成员列表，这里出现了两次，D7 中出现了一次，D8 中又出现了一次，这是不允许的。

D9 和 D10 没有问题，多次声明相同的结构标记是允许的。

D11 和 D12 不能共存，因为它们是无链接的，而且不符合以上条件（不属于 typedef 名和标记），因此冲突。

块内的声明，它们起作用的方式类似于语句。每当程序的执行到达声明的位置时，按声明符在声明中出现的顺序执行以下操作：

（1）对于自动存储期的对象，若它的声明中有初始化器，则求值初始化器并写入这些初始值；

（2）对于变长数组，求值声明符。

基于以上原因，再加上每个全声明符的末尾都有一个序列点（参见“序列点”），故下面的声明（字体加粗的部分）可以保证没有问题，不存在未定义的行为：

```
# include <stdio.h>

void f (void)
{
    char a = 0, b = a ++, c = b;
    int x = 0, y [++ x], z [++ x];
    printf ("%zu, %zu, %zu\n", sizeof x, sizeof y, sizeof z);
}
```

这里，在初始化对象 b 时，对象 a 的初始化已经完成；在初始化对象 c 时，对象 b 的初始化也已经完成，且 a++的副作用也已经完成。类似地，在求值声明符 y[++x]时，对象 x 的初始化已经完成；在求值声明符 z[++x]时，++x 的副作用也已经完成。

要了解更多的信息，请参见“存储期”和“生存期”。

为了防止你在遇到编译问题时不知所措，这里顺便说一下，上述程序代码没有问题，但截止到本书出版的时候，GCC 的最新版本号是 5.2.0，如果加上一个编译选项“-Wall”，则会产生虚假的警告信息：warning: operation on 'x' may be undefined [-Wsequence-point]，提示对 x 的操作可能是未定义的行为。“-Wall”选项允许所有的警告，如果你不确定程序中的某个地方是否有问题，可以用它来试试。

针对这个问题编者已经向 GCC 提交了报告，通过下面这个链接可以了解报告的内容以及 GCC 内部对这个问题的讨论细节：https://gcc.gnu.org/bugzilla/show_bug.cgi?id=67661。

5.1.1 （函数）原型

传统 C 语言和标准 C 语言在函数声明上的不同之处在于后者使用函数原型（或简称原型）。

函数原型（function prototype）是函数类型的声明，相对于传统形式的函数声明，它的典型特征是包含了参数类型的声明。函数原型说明了每个参数的类型（因此也说明了参数的数量），以及函数的返回类型。

这样，在转换一个 C 程序时，C 实现就可以对参数的数量和类型加以匹配和检查，并进行必要的数据类型转换（如果能够自动转换的话）。在下面这个例子中，在声明函数 f 的同时也声明了其参数类型（显然，函数 f 接收两个 int 类型的参数）。

```
int f (int, int);
```

没有函数体的函数原型要包含参数类型列表，参数类型列表中可以包含标识符（但是在不包括函数体的函数原型中似乎没有用处），例如：

```
int f (int i, int j);
```

如果是包括了函数体的函数原型，则参数列表中不但要有参数类型，而且要有标识符。以下是采用原型形式的函数定义。

```
int f (int i, int j) {return i * j;}
```

如果使用函数原型，且参数列表为 void，则表示该函数不接收任何参数；如果返回类型为 void，则表示该函数不返回任何值。例如：

```
int func (void);          //函数原型。不接收任何参数，且不返回任何值

int main (void)           //函数定义（原型形式）
{
    return func ();       //函数调用
}

int func (void)           //函数定义（原型形式）
{
    /* …… */
}
```

当使用原型形式创建一个参数类型和数量都不确定（可变）的函数时，在它的参数类型列表中至少要有一个固定参数，这些固定参数有确定的类型且位于参数类型列表的开头，然后是一个逗号和一个省略号。例如：

```
int f (char *, ...);                          //参数可变的函数声明
int g (int i, int j, ...) {/* …… */}          //参数可变的函数定义
```

经常用到的库函数 printf 就是一个非常典型的例子，根据需要，它可以接收不同数量和不同类型的参数。

5.1.2 外部声明

我们知道，C 源文件在经过预处理后就得到转换单元，转换单元由声明组成，而这些声明分

为两类：函数定义和位于函数外部的声明。函数定义则又囊括了函数名、参数和位于函数体内部的其他声明和语句，但参数和函数体都不是转换单元的直接组成部分。

以任何函数的定义为参照，位于函数外面的声明被认为是外部声明。但是话又说回来了，任何函数定义本身也是一个外部声明。

从作用域的角度来说，在一个转换单元中，任何具有文件作用域的声明都是外部声明（external declaration）。

在下例中，D1~D4 都是外部声明；D5 中，对 h 的声明是外部声明，对 m 的声明不是外部声明； D6 不是外部声明。

```
typedef struct stNode {int data; struct stNode * next; \
                       } NODE, * PNODE;      //D1
int x = 0, y = 1, * px = & x;                //D2
float f (float, float);                      //D3
int g (int, int);                            //D4
void h (int m)                               //D5
{
    const char * p = "Pelles C";             //D6
    /* ..... */
}
```

5.2 定义

定义（definition）也是声明，但声明并不都是定义。“定义”这个概念仅适用于对象声明、函数声明、枚举常量的声明和 typedef 名的声明，而且需要通过判断是否符合以下条件来考量。

首先，如果是声明一个对象，而且该声明会使 C 实现为对象分配存储空间，那么这个声明也是一个定义。因此，下例中，D1、D3 和 D4 都是对象的定义，但 D2 和 D5 仅仅是声明，不是定义。

```
static int x;                        //D1
struct t {char c; float f;};         //D2

int demo (void)
{
    int x = 0;                       //D3
    struct t t;                      //D4
    /* ..... */
}

extern int x;                        //D5
```

其次，如果是声明一个函数，而且该声明带有函数体，则这个声明也是一个定义，称之为

“函数定义”、“函数的定义”、“定义了一个函数”。下例中，D1 和 D3 仅仅是声明了一个函数，它们不是函数的定义，只有 D2 是函数的定义，因为它有函数体。

```
float f (float, float);      //D1

int demo (void)              //D2
{
   void g (int);             //D3
   /* ..... */
}
```

再次，任何枚举常量的声明，只要它是合法的，都是一个定义。如下例所示，枚举常量 One、Two、Three、Four 和 Five 的声明也是它们的定义，而且 Three 被声明了两次（当然，它们具有不同的作用域，否则就会冲突），这两次声明也都是它的定义。

```
enum A {One, Two, Three,};

int demo (void)
{
   enum B {Three, Four, Five,};
   /* ..... */
}
```

最后，如果同一个 typedef 名的多次声明都具有相同的作用域，则该 typedef 名的定义是它的第一次声明；相反，只声明了一次的，该声明也是它的定义。毕竟，typedef 名可以重复声明，即使它们位于同一个作用域内。如下例所示，D1 是 typedef 名 INT 的定义，其他则不是。注意，D3 是非法的，尽管允许多次声明一个 typedef 名（即使它们的作用域相同），但不能使它们的类型不同（冲突）。

```
typedef int INT;             //D1
typedef int INT;             //D2
typedef long long INT;       //D3: 非法
```

5.2.1 内联定义

术语“内联定义”仅适用于内联函数。如果一个函数在某个转换单元中的定义使用了函数指定符 inline 而没有使用存储类指定符 extern 和 static，那么：

（1）如果在同一个转换单元中该函数不存在其他具有文件作用域的声明，则上述定义就是该函数的内联定义。

（2）如果在同一个转换单元中该函数存在其他具有文件作用域的声明，而且它们都使用了函数指定符 inline 而没有使用存储类指定符 extern 和 static，则上述定义也是该函数的内联定义。

（3）在其他情况下，该函数的上述定义不是它的内联定义。

因为上述的原因，如果一个转换单元的全部内容如下，则 D1 和 D2 都是内联定义，但 D3 不是。

```
inline void f (int);
```

```
inline float g (float, float);
inline static double h (double, double);

int inlinedemo (void)
{
   h (0.01, 0.02);
}

inline void f (int m)                                    //D1
{
   /* …… */
}

inline float g (float d, float e)                        //D2
{
   /* …… */
}

inline static double h (double f, double g)              //D3
{
   /* …… */
}
```

注意，函数的内联定义不是它的外部定义。

5.2.2 外部定义

外部定义（external definition）也是外部声明，但外部声明并非都是外部定义。外部定义包括外部对象定义和函数定义（但不包括内联定义）。具体可参见“外部声明”、“外部对象定义”、“声明”和“内联定义”。

也就是说，一个指示对象的标识符，如果它的外部声明也是一个定义，则该声明是一个外部定义；一个指示函数的标识符，如果它的声明也是一个函数定义（这意味着有函数体）但不是该函数的内联定义，则该声明也是一个外部定义。例如（假定这是一个转换单元的全部内容）：

```
extern int global;                    //D1
int x = 0, y = 1, * px = & x;         //D2
float f (float, float);               //D3
void g (int m) {/* …… */}             //D4
struct t {char c; float f;};          //D5
struct t t = {'x',0.01f};             //D6
inline int h (void) {/* …… */}        //D7
```

其中，D1 仅仅是一个具有外部链接的声明，指示一个在其他地方定义的对象，所以它不是外部定义；D2 不但是声明，而且造成了存储空间的分配，所以是外部定义；D3 仅仅是一个具有外部链接的函数声明，指示一个在其他地方定义的函数，所以它不是外部定义；D4 是带有函数体的函数声明，所以是外部定义；D5 仅仅是结构类型的声明，更不会导致存储空间的分配，所以它也不是外部定义；D6 声明了结构类型的对象，并且一定会导致存储空间的分配，所以它是外部定义；至于 D7，虽然它是函数定义，但它是一个内联定义（在当前转换单元内，函数 h 只有这一个声明），按规定，内联定义并不是外部定义。

在同一个转换单元内，同一个标识符可以有多个内部链接的声明，但其中只能有一个属于外部定义。

因此，在下例中，两个用粗体显示的函数定义只能留一个，但其他有关 f 的声明可以共存。

```
static int f (void);
static int f (void) {/* …… */}
static int f (void);
static int f (void) {/* …… */}
static int f (void);
```

5.2.2.1 外部对象定义

外部对象定义（external object definitions）首先是一个外部声明（参见“外部声明”）。进一步地，一个指示对象的标识符，如果它的外部声明也是一个定义，则该声明就是一个外部定义，同时，也是一个外部对象定义。但是，如何界定一个外部声明是一个对象定义呢？或者说如何界定一个外部声明是一个外部对象定义呢？

首先，一个外部声明，如果它声明的是一个指示对象的标识符（对象声明），而且在声明中给出了初始化器，不管有没有该标识符的其他试探性定义（参见“试探性定义”），它都是外部定义，即外部对象定义。下例中，D1 和 D2 都是外部对象定义，因为标识符 x 和 t 都具有文件作用域，被声明为对象类型，且分别带有初始化器。至于 D3 和 D4，D3 是函数定义而 D4 是结构类型 struct t 和结构对象 t 的声明。

```
int x = 0;                                          //D1
struct t {char c; float f;} t = {'x', 0.01f};       //D2
int f (void) {/* …… */}                             //D3
struct t {char c; float f;} t;                      //D4
```

其次，在一个转换单元中，如果一个标识符的声明全是试探性定义，那么可以为这个标识符假想一个新的、具有文件作用域的声明。在这个假想的声明中，标识符的类型是其所有试探性定义的复合类型（参见“复合类型”），而且这个声明有一个值为 0 的初始化器。至此，这个假想的声明就是该标识符的外部对象定义。

假定一个转换单元的全部内容如下，那么 D1 和 D4 都是标识符 a 的试探性定义；D2 和 D5 都是标识符 b 的试探性定义；D3 和 D6 都是标识符 pf 的试探性定义。注意，pf 是一个指针，它是一个对象，而不是一个函数，这点要注意，毕竟现在讨论的是外部对象定义，而不是函数定义。

```
char a [];                                                    //D1
int b [] [3];                                                 //D2
float (* pf) (float (*) [3], float (*) (const char *));       //D3

int split (void) {/* …… */}

char a [6];                                                   //D4
int b [2] [3];                                                //D5
float (* pf) (float (*) [], float (*) ());                    //D6
```

那么，根据以上所述，这 3 对试探性定义的效果就像在当前转换单元内有如下 3 个外部（对象）定义。

```
char a [6] = {0};
int b [2] [3] = {{0}};
float (* pf) (float (*) [3], float (*) (const char *)) = 0;
```

下面一个例子是关于数组类型的外部对象定义的，而且比较特殊。如果在一个转换单元中，对一个数组类型的标识符 c 只声明了一次，即

```
char c [];

int f (void)
{
   c [0] = 'x';
   return * c;
}
```

那么，因为在整个转换单元中数组 c 只声明了一次，而且属于不完整类型（缺乏数组大小的信息），所以这个声明也是它的试探性定义。根据前面的规则，它的效果相当于在当前转换单元中有如下外部声明：

```
char c [] = {0};
```

这样的声明是非常普遍的，我们经常会在声明一个数组时忽略它的大小，而直接给出初始化器，让 C 的实现自己计算这个数组的大小。正因为如此，在程序启动时，对象 c 会拥有一个元素，而且被初始化为 0。

但是，尽管对象 c 有一个被初始化为 0 的元素，但是在转换单元中，标识符 c 依然是不完整类型，也就是说，在函数 f 内，对数组 c 的操作依然是不合法的。

5.2.2.2 试探性定义

一个外部声明，如果它声明的是一个指示对象的标识符，而且在声明中给出了初始化器，那它就是一个外部对象定义。

但是，如果声明的是一个指示对象的标识符，而且在声明中没有带初始化器，这意味着什么呢？在程序转换过程中，当 C 实现看到这样的声明时，它还要看这个声明是否使用了存储类指定符。如果没有使用任何存储类指定符，或者仅使用了存储类指定符 static，则这样的声明被视为一

个试探性定义（tentative definition）。

在下例中，D1 和 D2 是两个外部对象声明，其中，D1 是对象 i 的试探性定义，直至 C 的实现看到 D2，这时 D2 是外部对象定义，而 D1 是同一个对象的外部声明。

```
int i;                                          //D1
int f (void) {/* …… */}
int i = 0;                                      //D2
```

再来看另一个例子：

```
int x = 0;                                      //D1
struct t {char c; float f;} t = {'x', 0.01f};   //D2
int f (void) {/* …… */}
int x;                                          //D3
struct t {char c; float f;} t;                  //D4: 非法
```

其中，D1 和 D2 分别是标识符 x 和 t 的外部对象定义（参见“外部对象定义”），而 D3 是标识符 x 的试探性定义。但是，为什么 D4 不是标识符 t 的试探性定义，而被视为非法呢？原因是不能违反标准的其他规定：若多个结构类型的声明使用了相同的标记，且它们的作用域相同，那么，在所有这些声明中，结构声明列表（结构成员）只能在这些声明中出现一次（参见“结构”或“联合指定符”）。思考一下，如果要将 D4 改成标识符 t 的试探性定义，该怎么做呢？

对于一个试探性定义来说，如果它使用了存储类指定符 static（因为试探性定义是外部声明，所以这个 static 指示内部链接），那么它所声明的标识符要求是完整的对象类型，而不能是不完整类型。所以，在下面的例子中，标识符 a 的试探性定义是非法的；标识符 pf 的试探性定义是合法的。尽管 pf 所指向的类型并不是完整类型，但 pf 本身的类型是完整的对象类型（指针是完整的对象类型）。

```
static int a [];
static float (* pf) [];

int split (void) {/* …… */}
```

5.2.2.3 函数定义

函数定义是函数声明的特殊形式，是包括函数体的函数声明。函数体是一个复合语句，也就是一对匹配的花括号，以及可选地，位于花括号内部的其他声明和语句。

传统 C 语言的函数定义形式比较古怪，如刚才声明的函数 func，它的定义可能是这样的：

```
int func (i, j)
int i, j;
{
    return (i > 0) ? i : j;
}
```

可以看到，传统 C 语言的函数定义和函数原型不同，在声明符中，函数名的后面是用括号括住的标识符列表，即

```
    i, j
```

然后是针对这些标识符的声明列表，即

```
    int i, j;
```

最后是复合语句（函数体），即

```
    {
        return (i > 0) ? i : j;
    }
```

甚至，在C99之前，如果函数的定义没有指定返回类型，则它的返回类型默认是int。这就是为什么经常可以在较老的C程序中看到这样的main函数：

```
    main ()
    {
        /* ..... */
    }
```

但是从C99开始，标准要求函数定义必须指定返回类型。

如下例所示，当调用一个函数时，被调用的函数在当前转换单元内可能是有定义的（存在一个有函数体的函数声明），也可能是无定义的——也就是说，函数的代码可能位于另一个转换单元，或者位于另一个已编译的目标文件或库。但是无论如何，只要能在编译或者链接阶段解决“函数f的代码到底在哪里”这个问题，程序转换就能成功。

```
    int f ();          //f的定义可能在当前转换单元、其他转换单元，或者库中

    int main ()
    {
        f ();
        f (1);
        f (2, 3);
        f (2.0, 'm');
        return f (1, 2, 3, 4, 5);
    }

    /* ..... */
```

问题在于，以上对函数f的5个调用中哪一个才是正确的呢？答案是取决于函数f的定义。如果函数f的定义如下：

```
    int f (i) int i; {/* ..... */}
```

则只有第2种调用方式是正确的，其他函数调用都因为传入的参数在类型和数量上同函数的定义不符而导致未定义行为。

通过以上的例子可以发现，如果函数是用传统C语言的形式声明的，因为缺少必要的信息，调用这个函数时，C实现不对传入的实际参数在类型和数量方面做任何检查工作。相反，如果采用函数原型，则要好得多，具体情况可参见“函数原型”。下面给出一个函数原型的实例。

```
int f (int, int);

int main (void)
{
    f ();                       //非法
    f (1);                      //非法
    f (2, 'x', & main);         //非法
    return f (1, 2);            //正确
}

int f (int i, int j)
{
    return (i > 0) ? i : j;
}
```

可见，如果函数调用是受原型控制的，则C实现会对传入的参数进行检查。

不管函数是用传统形式声明的，还是使用了原型，以下几条规则都是必须遵守的。

首先，函数的返回类型不能是数组，而只能是完整的对象类型。如果函数不返回值，则它的返回类型要声明为void。

因此，下面的声明是不对的。

```
typedef int A [8];
A f (int, int);
float g (void) [6];
```

其中，函数f返回一个数组（有8个int类型的元素），函数g也返回一个数组（有6个float类型的元素），但它们都是非法的。

其次，函数的声明可以不使用任何存储类指定符。如果要用，也只能使用extern或者static以指示链接属性。块内的函数声明不允许使用static。

下例中，第一个声明看起来是函数声明，其实只是一个类型定义，并非声明一个函数，所以使用存储类指定符typedef是没问题的；第二个声明使用了存储类指定符register，这是不允许的（具体的原因可参见“register”）；第三个声明也没问题，它表明h是内部链接的；函数bad的声明是非法的：

```
typedef int f (int, int);   //可以
register double g (void);   //非法
static void h (void)        //可以
{
    static void bad (void); //非法
}
```

标准要求C实现支持创建至少接收127个参数的函数；相应地，在调用一个函数时，也至少能够传递127个参数。

5.2.2.3.1 main 函数

在 C 中，main 函数是比较特殊的。在宿主式环境下，main 作为整个程序的主函数出现，程序启动并完成初始化操作后，将调用 main 函数。

在调用一个函数前应先声明，通常是给出一个原型。main 函数的调用是由 C 实现负责进行的，但它自己并不声明 main 函数的原型，而是将这个责任交给程序编写者。从另一方面来说，这样做也是有道理的：main 函数可以有不同的形式，C 实现不提供原型，而由程序编写者直接定义，也能避免同一函数不同原型之间的冲突。

因此，main 函数只在程序中定义即可。如果不从执行环境接收任何参数，它可以定义为返回 int 的、无参数的形式，即

```
int main (void) {/* …… */}
```

如果需要同执行环境交互，从执行环境接收参数，也可以定义为具有两个参数的形式及其等价形式，即

```
int main (int argc, char * argv []) {/* …… */}
```

在下例中，因为 DWORD 被定义为 int 类型的别名，而且 argv 是一个数组（其元素类型为指向 char 的指针），作为函数的参数，它会被调整为指向“指向 char 的指针”的指针，即 char * * ，因此，它是与上述形式等价的。

```
typedef int DWORD;
DWORD main (DWORD argc, char * * argv) {/* …… */}
```

除了以上两种形式之外，如果 C 实现提供了其他手段，则 main 函数可以使用实现定义的参数形式。

例如，下面的 main 函数定义中使用了 4 个参数。

```
int main (int argc, char * * argv, char * * envp, char * * apple)
{
    /* …… */
}
```

但是，象这样的 main 函数声明非严格依从标准的。

宿主式环境下的 main 函数，其返回类型是 int，而不是 void。事实上，的确有很多书会这样写：

```
void main ()
```

但这是错误的。在《C Programming FAQs》一书中，针对有些作家在他们的著作中使用 void main ()，作者 Steve Summit 委婉地嘲讽道：

“Perhaps its author counts himself among the target audience. Many books unaccountably use void main() in examples. They're wrong.”

翻译过来的意思是：可能（那些书的）作者也把自己当成其目标读者中的一员，许多书不负责任地在他们的例子中使用 void main ()，但那是错的。

上述的标识符 argc 和 argv 等，可以改为用户喜欢的其他名称，因为它们仅具有块作用域。以上述具有两个参数的 main 函数为例，它还可以定义为：

```
int main (int a, char * * b) {/* …… */}
```

对C实现来说，若main函数被定义为以上具有两个参数的形式，则：

（1）argc的值要求大于等于零（非负）；

（2）数组成员argv [argc]要求是空指针；

（3）如果argc的值大于0，那么数组成员argv [0]～argv [argc-1]分别包含的是指向字符串的指针。这些字符串的内容是在程序启动前，由宿主环境给出的相关信息，具体内容取决于C实现；

（4）如果argc的值大于0，那么数组成员argv [0]指向一个表示当前程序名的字符串。如果不能从宿主环境中得到程序名，则它指向一个空串，即argv [0][0]的值应为空字符。如果argc的值大于1，那么argv [1]～argv [argc-1]所指向的字符串都表示提供给当前程序的参数，在命令行环境中，这些都是通常所说的命令行参数。

下面是一个示例程序，假定转换后的文件名是a.out，则它可以在执行时显示为其提供的各个命令行参数。

```
# include <stdio.h>

int main (int argc, char * * argv)
{
    while (argc --) printf ("arg.%d is:%s\n", argc, argv [argc]);
    return 0;
}
```

例如，在Linux环境下，为a.out提供命令行参数的方法可以是

```
./a.out p1 p2 p3
```

（5）可以修改对象argc和argv的内容，以及argv的（指针）元素所指向的字符串。argc、argv和字符串的生存期贯穿整个程序的执行过程，从程序启动到程序终止，在此期间可随意访问它们，并在执行读取操作时返回最近一次修改和存储的值。

下面的例子很简单，仅仅是读取和显示传递给当前程序的每个参数。这里通过修改argv的值来使其指向下一个串，虽然到最后无法恢复argv的初始值，但它在这里已不再有用。

```
# include <stdio.h>

int main (int argc, char * argv [])
{
    while (* argv ++ != '\0') printf ("%s\n", * argv);
    return 0;
}
```

从C99开始标准规定，在宿主式环境下，当程序的执行到达main函数体的右花括号“}”时（这意味着没有遇到return语句），默认返回0值，就像执行了

```
return 0;
```

或者

```
exit (0);
```

一样。但是，如果用的是C99之前的编译器，则没有这个保证。

注意：库函数 exit 是在头文件<stdlib.h>中声明的，调用它将导致当前程序的执行正常终止，其原型如下，详细的描述参见标准文档的库函数部分。

```
# include <stdlib.h>
_Noreturn void exit (int status);
```

5.3 静态断言

静态断言（static assertions）用于在转换期间（主要是在预处理的后期）对程序编写者描述的失败条件进行检查和判断，从而产生一个警告或者失败的消息。

静态断言声明的语法形式如图 5-2 所示。

图 5-2 静态断言的语法

静态断言是这样工作的：在程序转换期间，如果常量表达式的值不是 0，则不会有任何外在效果，就像它不存在一样；否则，C 实现应当产生诊断信息，同时可能中止正在进行的转换过程。声明中指定的字面串是自定义的消息文本，它是诊断信息的一部分。

如果能在预处理阶段就及早发现一些影响程序转换的问题当然是好的，比如你可以用预处理指令#if 和#error 来检测这些问题并抛出诊断消息。但是，唯一的问题是预处理指令的功能有限。

C11 引入了静态断言，因它求值的时间较晚，此时表达式的类型已知，这使得它可以捕获那些在预处理阶段就能够检测到的错误。

如果组成字面串的字符不是源基本字符集（参见“源字符集”）的成员，则它有可能无法正确输出。另一个要注意的问题是，这里的常量表达式要求具有整数类型的值，或者要求是一个整型常量表达式。

在下面的例子中，程序只能在 sizeof (int)大于等于 4 的环境下转换和执行，否则不能成功进行转换。为此，可以使用静态断言声明来达到这个目的。

```
_Static_assert (sizeof(int)>=6, "Required sizeof(int)>=4!");
/* ..... */
```

5.4 声明指定符

声明指定符（declaration specifiers）是一个统称，下面给出了它的组成，同时也说明了这些组成部分的功能。

（1）类型指定符：用于指定被声明的实体的类型。取决于声明的形式，它们可能会和声明符共同决定最终声明的是什么类型，毕竟有些类型是从声明符一侧派生的，如指针、数组和函数。

（2）存储类指定符：用于指示被声明的实体的存储期和生存期，或者指示标识符的链接

类型。

（3）类型限定符：用于声明原子的，或者限定的类型（相对于它们的非原子和无限定版本而言）。

（4）函数指定符：用于在声明函数时指定一些特殊的属性。

（5）对齐指定符：仅用于对象类型的声明，指示被声明的对象所要求的对齐。

上述声明指定符详见其各自的词条。

5.4.1 类型指定符

类型指定符用于在声明中指定被声明实体的类型（当然，最终的类型有可能还要结合声明符才能确定）。类型指定符包括 void、char、short、int、long、float、double、signed、unsigned、_Bool、_Complex。

从声明的语法图（参见“声明”）可知，声明中允许多个声明指定符的组合，声明指定符包括类型指定符，所以允许一个以上的类型指定符。这也意味着，声明指定符可以包含多个类型指定符的合理组合。注意，这里要求的是合理的组合，但是像 void char、unsigned _Bool 这些组合就是不合理的，即非法组合。对于合理的组合，有很多是我们比较熟悉的，如：unsigned char、unsigned long long int、float _Complex、long double _Complex。

有些类型指定符的组合是同义的，代表同一种类型，例如：

（1）short、signed short、short int 和 signed short int 是同义的；

（2）unsigned short 和 unsigned short int 是同义的；

（3）int、signed 和 signed int 是同义的（除了位字段之外。在用于位字段时，int 究竟是等同于 signed int 还是 unsigned int，是由 C 实现定义的）；

（4）unsigned 和 unsigned int 是同义的；

（5）long、signed long、long int 和 signed long int 是同义的；

（6）unsigned long 和 unsigned long int 是同义的；

（7）long long、signed long long、long long int 和 signed long long int 是同义的；

（8）unsigned long long 和 unsigned long long int 是同义的。

例如，要声明一个 int 类型的标识符，可以使用

```
int i;
```

也可以使用

```
signed i;
```

还可以使用

```
signed int i;
```

复数类型不是强制要求的，C 实现可以支持，也可以不支持。因此，这取决于用户所使用的 C 实现，如果它不支持复数类型，则不应该在声明中使用类型指定符_Complex。

如果在声明中包含了一个以上的类型指定符，那么它们的组合顺序无关紧要。同时，类型指定符和其他声明指定符的组合顺序也无关紧要，C 语言在这方面不存在特别的限制。也就是说，允许类型指定符以任意次序组合，只要是合理的。例如，下面的声明虽然看起来不

符合用户平时的做法，但同样是正确的。

```
char signed c;                  //即signed char c;
long int unsigned ui;           //即unsigned long int ui;
int signed si = 5;              //即signed int si = 5;
int long signed long ll;        //即signed long long int ll;
long _Complex double lcd;       //即long double _Complex lcd;
char signed const cc;           //即const signed char cc;
_Complex double const long _Alignas (128) comp;
```

上面的最后一个声明是什么意思呢？很简单，标识符 comp 的类型是 const long double _Complex，并且按 128 对齐。

对于一个合法的声明而言，声明中不能没有类型指定符，所以声明指定符中必须至少包含一个类型指定符。如果一个标识符具有文件作用域，而在声明时又省略了类型指定符，那么多数 C 实现将认为它的类型指定符是 int。例如，下例中的 i、j 和 main。特别是 main 函数，在比较老旧的代码中仍能翻到这种省略类型指定符的写法。但是，省略类型指定符，并由 C 实现将其默认为是 int 的做法是过时的，也不符合标准要求，应予摒弃。

```
# include <stdio.h>

i, j = 0;

main ()
{
    printf ("%d, %d\n", i, j);
    return 0;
}
```

结构或联合指定符、原子类型指定符、枚举指定符、typedef 名具有各自的特殊性，具体的情况可参见其各自的词条。

5.4.2 结构或联合指定符

在声明中，结构或联合指定符用于指定一个结构类型，或者一个联合类型，其语法形式如图 5-3 所示。图中未展开的非终结符，可参考其各自的词条。

结构指定符和联合指定符在语法形式上极为相似，唯一的区别是，前者以关键字 struct 开始，而后者以关键字 union 开始。结构声明列表包含了结构或者联合的成员（的声明），这部分也称为成员声明、成员列表，或者内容定义。

结构或联合指定符中的标识符（标记）和成员列表都是可选的，但不能同时省略。以下是几个结构或联合指定符的例子。

```
struct s
struct t {int i; const char c; signed int h : 3; unsigned : 5;}
struct {char c; float f;}
```

```
union {char c; struct t {char a [32];} t;}
```

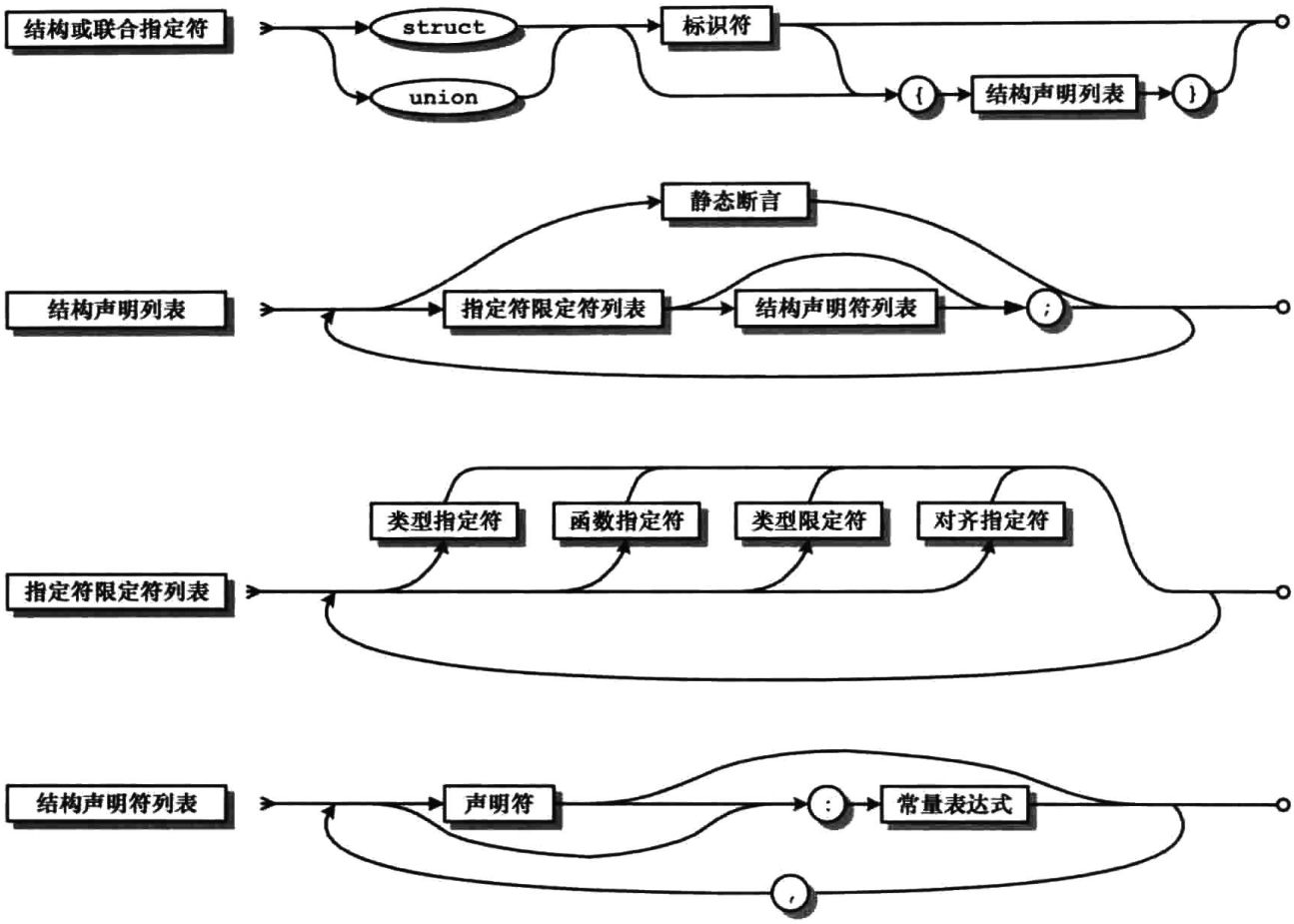


图 5-3　结构或联合指定符的语法

在给出（声明）结构或者联合的成员时，要为它提供类型指定符，这是必不可少的。下例中有两个声明，分别是 D1 和 D2。其中，D1 中的成员声明是非法的，因为其第二个成员 c 没有类型指定符；D2 中的成员声明是合法的，其第一个成员的声明中有类型指定符 char，第二个成员的声明中有类型指定符 struct t。

```
struct t {int i; c;};                    //D1
struct s {char c; struct t * pt;};       //D2
```

关键字 struct 和 union 后面的标识符是结构或联合的标记，它是可选的。如果省略了这个标记，那么必须保留成员列表（内容定义），而且声明的是一个未命名的结构或者联合类型。

下面的声明是合法的，但是它们的类型指定符中没有给出标记。因此，在这两个声明之外，将不再可能声明和 s、u 同类型的其他实例。

```
struct {int i; char c;} s;
union {char c; double d;} u;
```

当然，用户可能不以为然，并写出以下代码来证明 s 和 t 的类型相同，都是同一种结构类型：

```
struct {int i; char c;} s;
struct {int i; char c;} t;
```

事实上，有这样的想法是正常的，但并不正确。原因很简单，如果一个结构或联合指定符具有以下形式：

```
struct {/* …… */}
```

则它是一个结构类型的声明，而且声明的是一个新的、独一无二的结构类型。同理，如果一个结构或联合指定符具有以下形式：

```
union {/* …… */}
```

则它是一个联合类型的声明，而且声明的是一个新的、独一无二的联合类型。

换句话说，在程序中，每一个没有标记的结构或联合声明，声明的都是独立的、截然不同的类型。

正是因为这个原因，上述的两个标识符 s 和 t，尽管它们的类型指定符完全相同，但其实属于两个完全不同的结构类型，你可以用结构类型的赋值将 t 赋给 s 试试，这将引发诊断信息。

用这种方式声明的类型用途有限。在其声明点之外，将无法使用这种类型。因此，无标记的声明最好带有声明符，比如下例中的 stgNOD。

```
typedef struct {int i; char c;} stgNOD;
struct {int i; char c;} t;
stgNOD nod; //nod 和 t 的类型不同，尽管它们是具有相同成员列表的结构类型
```

如果一个结构或联合指定符具有以下形式：

```
struct 标识符 {/* …… */}
```

则它是一个结构类型的声明，而且声明的是一个新的、完整的结构类型。同时，这里的标识符也被声明为当前这种结构类型的标记，用以代表当前这种结构类型。同理，如果一个结构或联合指定符具有以下形式：

```
union 标识符 {/* …… */}
```

则它是一个联合类型的声明，而且声明的是一个新的、完整的联合类型。同时，这里的标识符也被声明为当前这种联合类型的标记，用以代表当前这种联合类型。

作为示例，在下面的代码中，D1 中的类型指定符指定了一个新的、完整的结构类型，标识符 t 是这个结构类型的标记；D2 中的类型指定符指定了一个新的、完整的联合类型，标识符 u 是这个联合类型的标记；D3 和 D4 表明，一旦声明了某个结构或者联合类型且它带有标记，则以后就可以通过标记来引用该类型（如用来声明更多的同类型对象）。

```
struct t {char c; int i;} t1, t2, * pt;     //D1
union u {char c; int i;};                   //D2
struct t t;                                 //D3
union u u, v, w, x, y, z;                   //D4
```

下面是几个综合性的示例。

在以下代码中，从 D1A 开始，一直到 D1B，这些代码共同组成了第一个声明的类型指定符，它包含结构声明列表，所以声明了一个新的结构类型 struct s；在这个类型指定符内部，还嵌套了一个类型指定符，同样含有结构声明列表，按照上面的说法，这同时也声明了一个新的类型 struct t。声明的新类型可以在后续的代码中使用，因此，D2 中声明了一个这种新类型的对象。

```
struct s                    //D1A
```

```
{
    char c;
    struct t
    {
        int i;
        _Bool b;
    } t;
}                                   //D1B
s = {'x', {0, 0}};

struct t t = {1, 1};                //D2
```

又如，在以下代码中，在定义函数 f 时顺便声明了结构类型 struct s，并且，在声明函数的参数时，该类型是可用的；在函数内部，对象 m 的声明中带有初始化器，这个初始化器具有一个“副作用”，即同时声明了结构类型 struct t。在此之后，这两种结构类型都可用（要注意它们的作用域，struct s 具有文件作用域，而 struct t 仅具有块作用域，所以 D3 为非法）。

```
# include <stddef.h>

struct s {char c; int i;} f (struct s ss)
{
    size_t m = sizeof (struct t {int x; float f;});
    struct t t;                     //D1: 合法
    struct s s;                     //D2: 合法
    /* ...... */
}

int main (void)
{
    struct s s = f ();
    struct t t;                     //D3: 非法
    /* ...... */
}
```

第三个例子，我们在声明一个 typedef 名的同时，也声明了一个结构类型 struct t，因为这里出现了结构成员列表。于是，我们可以在后面声明 struct t 的对象，也可以声明 PKG 类型的对象：

```
typedef struct t {char c; float f;} * PKG;
struct t t = {'x', 0.01f};
PKG p = & t;
```

我们知道，可以多次重复定义一个 typedef 名，哪怕是它们具有相同的作用域和名字空间（参见“声明”），例如：

```
typedef int INT;
typedef int INT;
typedef signed INT;
```

但是，为什么下面的 D1 和 D3 是合法的，而 D2 不合法呢？原因很简单，如果没有标记，则即使成员完全相同，两个结构的声明也将得到完全不同的结构类型。因此，在 D1、D2 和 D3 中声明的结构类型都不相同，尽管它们一模一样。同一个 typedef 名的多次声明若具有相同的作用域，则必须指定相同的类型，且不能是可变修改类型。

```
typedef struct {char c; float f;} stgD;     //D1: 允许
typedef struct {char c; float f;} stgD;     //D2: 非法
typedef struct {char c; float f;} stgE;     //D3: 允许
```

然而，即使是将 D1 和 D2 修改成下面这样，也是不可以的，因为，结构类型列表的每一次出现都声明出新的结构类型，但前提是这种结构类型以前没有声明过，也就是说不能重复定义同一种类型：

```
typedef struct t {char c; float f;} stgD;
typedef struct t {char c; float f;} stgD;  //重复定义 struct t
```

因此，要使 D2 合法且 D1 和 D2 也能共存，可以修改成这样：

```
typedef struct t {char c; float f;} stgD;
typedef struct t stgD;
```

如果一个结构类型的声明具有以下形式：

```
struct 标识符 ;
```

那么，如果在当前位置能够看到另一个使用相同标记的结构类型声明 S，且其作用域和当前的这个声明相同，则当前声明的类型被指定为结构类型 S（使用相同标记的、具有相同作用域的结构类型声明，所声明的都是同一种结构类型）；若 S 的作用域和当前的这个声明不同，则当前声明的类型被指定为一个新的、不完整的结构类型，并隐藏前面的结构类型 S。

如果在当前位置看不到另一个使用相同标记的结构类型声明，则当前声明的类型被指定为一个新的、不完整的结构类型。

同样地，如果一个联合类型的声明是这样的：

```
union 标识符 ;
```

那么，如果在当前位置能够看到另一个使用相同标记的联合类型声明 U，且其作用域和当前的这个声明相同，则当前声明的类型被指定为联合类型 U（使用相同标记的、具有相同作用域的联合类型声明，所声明的都是同一种联合类型）；若 U 的作用域和当前的这个声明不同，则当前声明的类型被指定为一个新的、不完整的联合类型，并隐藏前面的联合类型 U。

如果在当前位置看不到另一个使用相同标记的联合类型声明，则当前声明的类型被指定为一个新的、不完整的联合类型。

下例中，在 D2 处能看到使用相同标记的结构声明 D1，但两者作用域不同，故 D2 隐藏了 D1，D2 声明的是一个不同于 D1 的结构类型。D7 处的声明也使用了相同的标记，且 D2 和 D7 的作用域相同，因此，D2 和 D7 所声明的是同一结构类型，D7 是将 D2 中的结构类型补充为完整类型。

在 D4 处，能看到另一个相同标记的结构类型声明 D3，且它们的作用域相同，因此，D4

中声明的结构类型被指定为 D3 中声明的类型。换句话说，D3 和 D4 声明的是同一结构类型。

在 D5 处，看不到使用同一标记 w 的任何结构类型声明。因此，这里实际上声明了一个新的结构类型。使用标记，且作用域相同的结构类型声明，声明的是同一结构类型，因此，D5 处指定和声明的结构类型，和 D6 处声明的结构类型相同，D6 将 D5 处声明的结构类型补充声明为完整类型。

```
/* 在此处看不到结构类型 struct w 的任何声明 */
struct s {/* …… */};          //D1

void f (void)
{
    struct s;                 //D2
    struct t {/* …… */};      //D3
    struct t;                 //D4
    struct w;                 //D5
    /* …… */
    struct w {/* …… */};      //D6
    struct s {/* …… */};      //D7
}
```

上述的声明形式有它的必要性，例如：

```
struct s {int i; float f;};                  //D1

if (/* …… */)
{
    struct s;                                //D2
    struct t {char c; struct s * ps;};       //D3
    struct s {char c; struct t * pt;};       //D4
}
```

其中，在 D1 中将标识符 s 声明为结构类型的标记，其在内部块中也是可见的。因此，如果没有 D2，则 D3 中的 ps 所指向的类型是在 D1 中声明的结构类型。但是，内部块中的 D2 将外部的 struct s 隐藏，并重新声明了一个新的结构类型 struct s。这将导致 D3 中的 ps 所指向的类型是 D2 中声明的新类型，该类型在 D4 中进一步被补充声明为完整类型。而且，在 D4 中看到的 struct t 是完整类型。

如果一个结构或联合指定符具有以下形式：

```
struct 标识符
```

且紧跟在标识符后面的不是成员列表（内容定义）或者分号（；），那么，若在当前位置看不到其他使用相同标记的结构类型声明，则它声明的是一个不完整的结构类型（可以在后面补充声明为完整类型），并将这里的标识符声明为该结构类型的标记；若在当前位置能看到其他使用相同标记的结构类型 S，则它用来指定结构类型 S（而且不会重新声明这个标记）。

同样地，如果一个结构或联合指定符具有以下形式：

union *标识符*

且紧跟在标识符后面的不是成员列表（内容定义）或者分号（;），那么若在当前位置看不到其他使用相同标记的联合类型声明，则它声明的是一个不完整的联合类型（可以在后面补充声明为完整类型），并将这里的标识符声明为该联合类型的标记；若在当前位置能看到其他使用相同标记的联合类型 U，则它用来指定联合类型 U（而且不会重新声明这个标记）。

例如，下例中，假定在 D1 的开头处看不到其他名为“t”的标记。因此，当标记 t 第一次出现在 D1 中时，类型指定符 struct t 声明了一个不完整的结构类型。在 D2 处，因为能看到前面的同名标记 s，故此时类型指定符 struct s 用来指定前面已经声明的结构类型。最后，D3 处的声明将前面的不完整类型 struct t 补充声明为完整类型。

```
struct s {int i; struct t * pt;};  //D1
/* ····· */
struct s s, * ps;                  //D2
struct t {char c; double d;};      //D3
```

若多个结构类型的声明使用了同一的标记，且它们具有相同的作用域，则它们声明的是同一个结构类型。这条规则也同样适用于联合类型的声明。

在下例中，D2~D6 这 5 个声明使用了相同的标记，标记的作用域也相同，但声明的并非同一个结构类型，这似乎与以上规定相悖。事实上，因为在 D2 处能看见使用了相同标记的 D1，所以 D2 中的 struct t 并非是一个结构类型声明，而是用于指定 D1 中声明的类型。除此之外，D3~D5 都是结构类型的声明。因此，当前块内的 D3~D5 声明的是同一个结构类型。如果去掉 D1，则 D2 也是结构类型的声明，声明了一个不完整的结构类型，并将 t 声明为这个不完整结构类型的标记，于是，D2~D5 声明的是同一个结构类型。

```
struct t {/* ····· */};           //D1

void f (void)
{
    struct t * pt;                //D2
    struct t;                     //D3
    struct t {/* ····· */};       //D4
    struct t;                     //D5
    struct t t2, at [3];          //D6
    /* ····· */
}
```

如果两个结构类型的声明使用了不同的标记，则它们声明的是不同的结构类型；如果两个结构类型的声明使用了相同的标记，但作用域不同，则它们声明的也是不同的结构类型。当然，这条规则也同样适用于联合类型。

在下面的例子中，D1、D2、D3 和 D4 分别声明的是 4 个不同的结构类型，D1、D3 和 D4 中声明的结构类型之所以不同，主要是因为它们的作用域不同，而 D2 则除了作用域的因素外，所使用的标记与其他 3 个声明都不相同。

```
struct t {/* ····· */};           //D1
```

```
struct s {/* …… */};              //D2

int f (void)
{
    struct t {/* …… */};          //D3
    if (/* …… */)
    {
        struct t;                 //D4
        /* …… */
    }
    /* …… */
}
```

上面已经说明了，可以多次地重复声明同一种结构或者联合类型，同时也说明了在什么情况下这些声明被视为同一种结构或联合类型。但是，无论同一种结构或者联合类型被声明多少次，如果它们具有相同的作用域，带有内容定义的声明只能出现一次。下面的例子很好地说明了这一点。

```
struct t {char c; double d [];};
struct t;                              //允许
struct t t;                            //允许
struct t {char c; double d [];};       //非法：重复定义
struct t {int i;};                     //非法：重复定义
```

从结构或联合指定符的语法可以看出，允许在其成员列表中使用静态断言。例如：

```
struct t
{
    char c;
    _Static_assert (sizeof (signed int) < 17, "ERROR");
    int i : 17;
};
```

结构或联合的成员也可以是另一个结构或者联合，这种关系可以递归应用。当然，嵌套的层数肯定有所限制，标准要求 C 实现至少支持 63 层。

【思考题】下面的代码有什么问题？

```
struct {char a [32];} x = {"Code::Blocks."};
struct {char a [32];} y;
y = x;
```

5.4.3　枚举指定符

枚举指定符用于在声明中指定一个枚举类型，其语法形式如图 5-4 所示。它以关键字 enum 开始，后面的标识符也称为枚举标记。枚举器列表包含了枚举常量，这部分也称为枚举内容或者枚举内容定义。下面是一个枚举指定符的示例。

```
enum e {Up = 3, Down, Left, Right,}
```

其中，“enum”是关键字；“e”是标识符或者枚举标记；被一对花括号括起来的内容是枚举器列表，其中最后一个逗号是可以省略的；“Up”、“Down”、“Left”和“Right”分别是枚举常量；“3”是常量表达式，指定了枚举常量Up的值。

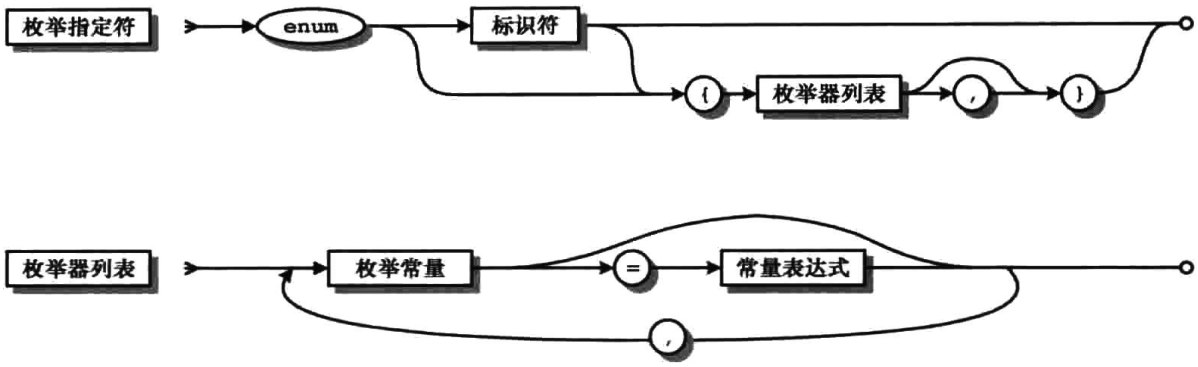


图 5-4　枚举指定符的语法

从枚举指定符的语法来看，显然，枚举器列表的尾部可以有一个逗号。这个特性是从C99开始引入的，而且是为了同声明中的初始化器部分保持一致（可以在初始化器列表的末尾添加一个额外的逗号。具体情况可参见“初始化”）。

添加这个额外的逗号是无害的，从实际效果来说，加不加最后这个逗号并没有什么不同。下例定义了一个enum e类型的对象e1，并用枚举常量Dog的值进行初始化。

```
enum e {Dog, Cat,} e1 = {Dog,};
```

当然，下面这个例子更能说明在末尾添加逗号的优点（请注意格式上的对齐效果，这种对齐有利于以后的扩展）。

```
enum e
{
    Ida,
    Idb,
    Idc,
    Idd,
    Ide,
};
```

如果一个枚举指定符具有以下形式：

```
enum {/* …… */}
```

则它声明的是一个新的、独一无二的枚举类型。换句话说，在程序中，每一个没有标记的枚举类型声明，声明的都是独立的、截然不同的类型。

与结构、联合类型不同，两个枚举类型的声明若具有相同作用域，则它们定义的枚举常量不能同名，这会引起冲突。下面是一个不合法的例子。

```
enum {A, B, C};      //合法
enum {C, D, E};      //非法：这里的枚举常量C与前一个枚举常量C冲突
```

用这种方式声明的枚举类型用途有限。在它的声明点之后，将无法使用这种类型（但可

以使用枚举常量)。因此，无标记的声明最好带有声明符，例如下例中的声明符 Chrono。

```
typedef enum {Morning, Noon, Evening, Night,} Chrono;
Chrono c1, c2 = Morning, * pc;
```

如果一个枚举指定符具有以下形式:

```
enum 标识符 {/* …… */}
```

则它声明的是一个新的、完整的枚举类型。同时，这里的标识符也被声明为当前这种枚举类型的标记。一旦有了标记，则可以在其他位置方便地声明这种枚举类型的实体。

```
enum e {Bowl, chopsticks, plates,};
enum e e1, e2, * pe, e [3];
```

再如下例所示，在声明函数 f 的同时，也声明了一个枚举类型，同时还将标识符 e 声明为当前结构类型的标记。从此，即可使用 enum e 来声明一些该枚举类型的对象。

```
enum e {Yes, No} f (void)
{
    /* …… */
}

void g (void)
{
    enum e e = f ();
    /* …… */
}
```

如果有多个枚举类型的声明使用了同一标记，且具有相同的作用域，则它们声明的是同一枚举类型。

相反，如果多个枚举类型的声明使用了不同的标记，则无论它们的作用域是否相同，都是截然不同的类型；如果多个枚举类型的声明都使用了相同的标记，但它们的作用域各不相同，则也是截然不同的类型。

在下面的例子中，因为标记不同，所以 D1 所声明的枚举类型和 D2、D3、D4 所声明的枚举类型不同，而不管作用域是否相同；尽管标记相同，但因为作用域不同，故 D2 所声明的枚举类型和 D3、D4 所声明的枚举类型不同。最后，因为作用域相同，标记也相同，所以 D3 和 D4 所声明的是同一枚举类型。

```
enum e1 {Font, Color, Brush,};                  //D1
enum e2 {FF, REW, PLAY, STOP, PAUSE, REC,};     //D2

void f (void)
{
   enum e2 {Window, Menu, Button,};             //D3
   enum e2 gui;                                 //D4
    /* …… */
}
```

与结构和联合类型相比，枚举类型有一个特殊之处，即，如果一个枚举指定符是这样的：

```
enum 标识符
```

而且，标识符（标记）后面不是枚举内容（枚举器列表），则它用于指定一个枚举类型，被指定的枚举类型必须是在当前位置可见的完整类型。正是因为这个原因，下面的声明是非法的。正确的做法是将它们颠倒一下。

```
enum e e, * pe;
enum e {Mouse, Keyboard, Speaker, Microphone,};
```

上面已经说明了，可以多次重复声明同一种枚举类型，以及在什么情况下这些声明被视为同一种枚举类型。如果存在着多个使用相同标记的枚举类型声明，且它们的作用域相同，那么，在这些声明中只能有一个带有枚举器列表（有内容定义或者定义了枚举常量）。下面的例子很好地说明了这一点。

```
enum e {Liquid, Solid, Gas,};
enum e;                          //允许
enum e e, * pe;                  //允许
enum e {Tom, Jerry, Merry,};     //非法：重复定义
enum e {I, Me, My, Mine,};       //非法：重复定义
```

5.4.4 原子类型指定符

“原子”的意思是整体进行，不被（线程或信号等）分割或者中断。其主要的出发点是为那些工作在多线程、中断状态下的 C 程序创建一个标准的方法。原子类型使得对象的访问和内存模型、总线宽度、指令集和高速缓存限制等无关。

原子类型是从 C11 开始引入的，但它是一个可选的特性，不要求 C 实现必须支持。任何依从 C 标准的实现都必须预定义宏__STDC_NO_ATOMICS__为整型常量 1，以表明它不支持原子类型，也不提供相应的头文件<stdatomic.h>。这意味着，如果没有定义此宏（对于 C11 之前的实现），或者这个宏被定义为整型常量 0，则表明不支持原子类型。

原子类型指定符的语法形式如图 5-5 所示。需要注意的是，图中的类型名不允许指定数组类型、函数类型和另一个原子类型，也不能是限定的类型。

图 5-5 原子类型指定符的语法

下面的例子展示了合法及非法的原子类型声明。

```
_Atomic (unsigned int) ai; //合法
_Atomic (char [20]) c;     //非法，指定了数组类型
_Atomic (const int) aci;   //非法，指定了限定的类型
```

5.4.5 typedef 名

typedef 名是一个标识符，用于代表某种类型，是那种类型的别名。用存储类指定符

typedef 声明的标识符是 typedef 名，声明的方法参见“类型定义”。

因为 typedef 名指示了某种类型，所以它可以作为类型指定符用在声明中。在下面的例子中，D1~D5 这 5 个声明都使用 typedef 名作为类型指定符。

```
typedef int F (void), INT, * PTR, ARR [6];

void demo (void)
{
    F g, * f = g;                    //D1
    INT x = 10010;                   //D2
    PTR p = & x;                     //D3
    ARR a = {0, 1, 2, 3, 4, 5,};     //D4
    INT y = f (), z = g ();          //D5
}

int g (void) {/* …… */}
```

注意，typedef 名不能和其他类型指定符一起使用。声明中使用的类型指定符是一个 typedef 名的，不得再添加其他类型指定符。例如，typedef 名 Integer 是 int 类型的别名。在 D1 中，原意是将标识符 li 声明为 long int 类型，但这是不允许的。

```
typedef int Integer;
long Integer li;                     //D1: 非法
const volatile Integer li;           //D2: 合法
```

5.5　存储类指定符

存储类指定符包括 typedef、extern、static、_Thread_local、auto 及 register。这些存储类指定符有些用于指示标识符的链接属性，有些用于指示对象的存储期，typedef 则用于定义某个现有类型的别名。

存储类指定符不多，看起来很简单，但却是 C 语言中最难理解和掌握的部分之一。在不同的上下文中，它们的出现代表着不同的含义和作用，要了解它们的用法和用途，可参见其各自的词条，还可参见“链接”、“存储期”。

声明中的存储类指定符很少组合使用。因此，除了 _Thread_local 可以与 extern 组合，以及 _Thread_local 可以与 static 组合外，只允许在声明中使用一种存储类指定符。也就是说，其他任意组合都是非法的。例如：

```
typedef static int i;
extern static int j;
```

这两个声明都是非法的，包含了多个存储类指定符。

存储类指定符 typedef 只用于声明一个现有类型的别名，当它用于声明聚合或者联合类型的别名时，并不作用于这些类型的成员。而且，用存储类指定符 typedef 声明的标识符是

无链接的。

相反地，typedef 之外的其他存储类指定符用于给被声明的对象指定各种属性。如果被声明的是聚合或者联合对象，那么，除了链接属性外，由这些存储类指定符产生的其他属性也将同样作用于该对象的成员，以及递归地，成员的成员。这不难理解，例如，如果一个声明为结构、数组或者联合对象的标识符是外部链接的，那么它的成员并不是外部链接的，并不会和其他标识符指示同一个对象；一个结构、数组或者联合对象的存储期（生存期），和它所有成员（元素）的存储期（生存期）是相同的。

如果是在块内声明一个函数（这样的声明不能有函数体），则要么使用存储类指定符 extern，要么什么存储类指定符都不用。在块内的函数声明中使用 extern 之外的存储类指定符是非法的。假定一个转换单元的全部内容如下。

```
/* 此处有意留空 */                    //D1

int main (void)
{
    static int f (void);          //D2：非法
    return f ();
}

static int f (void)               //D3
{
    /* …… */
}
```

这里的原意是希望标识符 f 具有内部链接。因为在 main 函数内调用了 f，而 f 的定义又在 main 函数的定义之后，所以需要在调用前声明，这就是为什么要使用 D2。

问题是，在块内声明一个函数时，不允许使用存储类指定符 static。那么，这里是否可以使用 extern 呢？也不行，函数的声明用不用 extern 都一样。因为在 D2 处看不到另一个有关 f 的声明，所以 D2 中的 f 是外部链接的，它需要一个位于当前转换单元、其他转换单元或者库中的 f 来完成链接。问题是，C 实现需要一个外部链接的 f，但却在 D3 处找到一个内部链接的 f，而同一个标识符不能既是内部链接的，又是外部链接的，所以这两个声明冲突。

那怎么修改呢？很简单，只要将 D3 处的函数定义放到 D1 处即可。

5.5.1 typedef

将 typedef 归为存储类指定符，这会使某些具有 C 语言基础的人讶异。这种怀疑是有道理的，因为 typedef 和对象的存储期似乎没有关系。事实上，它们确实没有关系，这么做仅仅是因为在描述声明的语法构成时比较方便。

存储类指定符 typedef 用于类型定义（可参见“类型定义”）。类型定义的本质是将一个标识符定义成它被声明的那种类型。如果一个声明中使用了存储类指定符 typedef，那么该声明中的每个声明符都会定义一个称为 typedef 名（参见“typedef 名”）的标识符，并且用这个标识符（typedef 名）来指示它被声明的那种类型。

例如：

```
typedef int * Pint, Func (void), Array [22];
Pint pi;          //pi 是一个指向 int 类型的指针
Func * pf;        //pf 是一个指向函数的指针
Array a;          //a 是一个数组，数组的类型为 int [22]
```

其中，第一行语句将标识符 Pint 定义为指针类型 int *的别名；将 Func 定义为函数类型 int (void)的别名；将 Array 定义为数组类型 int [22]的别名。注意，使用 typedef 的声明并不能引入和创建新的类型，它仅仅用于定义与指定类型同义的别名。

C 语言允许使用存储类指定符 typedef 定义变长数组类型的别名。但是，变长数组比较特殊，因为它的大小不是一个常量，只有在程序的执行到达其声明符所在的位置时才能通过求值声明符中的表达式来确定。

在下例中定义了 3 个变长数组类型 A、B 和 C。

```
# include <stdio.h>

int main (void)
{
    int n = 5;
    typedef char A [n + 1], B [n + 2], C [n + 3];
    n ++;          //不影响上述变长数组的大小
    printf ("%zu, %zu, %zu\n", sizeof (A), sizeof (B), sizeof (C));
    A a;
    B b;
    C c;
    return printf ("%zu, %zu, %zu\n", sizeof a, sizeof b, sizeof c);
}
```

在过了 typedef 名的定义点之后，变长数组的大小即可固定，不会再改变，即使 n 的值被改变（如上例中的 n++）。

可变修改类型的标识符都只能具有块作用域。同样的道理，只能在块内将一个 typedef 名声明为可变修改类型的别名。

所以，在下面的例子中，只有 D2、D3、D4 和 D5 是正确的，D1 是非法的。原因在于 D1 声明的是变长数组且具有文件作用域；D2 声明的不是可变修改类型；D3~D5 都是可变修改类型且具有块作用域。特别地，D3 声明的是变长数组，D4 声明的是指向变长数组的指针，D5 声明的是函数，该函数返回一个指向变长数组的指针。

```
int n = 30;
typedef VLArray [n];                    //D1
typedef Array [20];                     //D2

void f (int n)
{
```

```
    typedef VLArray [n];                    //D3
    typedef int ( * PVLA) [n];              //D4
    typedef char (* (F (void))) [n];        //D5
    /* ..... */
}
```

如果使用得当，类型定义可以增强程序的可读性。下面就是一个典型的例子，signal 是一个很重要的函数，用于处理计算机系统中的中断和异常信号，其原型如下。

```
void (* signal (int, void (*) (int))) (int);
```

此时，比较难理解其意思。但如果将它声明成如下形式，则效果更好。

```
typedef void F (int);
F * signal (int, F *);
```

这样就很清楚了，signal 是一个返回“指向函数的指针”的函数，它需要两个参数，一个是 int 类型，另一个是指向函数的指针。但滥用 typedef 名也可能造成相反的效果，例如（这些例子都是正确的）：

```
typedef unsigned int u;
typedef int i;
typedef struct {u ui; const i:4;} t;
t t0;
t f (t (t));
```

以上示例中的倒数第二行的语句，将标识符 t0 声明为结构类型；最后一行语句，将标识符 f 声明为返回结构类型的函数。函数 f 只有一个参数，参数的类型是“返回结构类型，并接收结构类型的参数”的函数。

以名字空间的观点来看，typedef 名属于普通标识符。所以它不允许和别的普通标识符重名，这会引起冲突，除非它们的作用域不同。具体内容可参见“名字空间”。

例如，在下面的例子中，尽管这里有 3 个同名标识符 t，但它们分属于不同的名字空间，只有第一行里的 t 是普通标识符，所以不会引起冲突。

```
typedef int t;
struct t {char c; int t;};
```

但是，在以下代码中，typedef 名 t 和结构类型的对象 t 都是普通标识符，它们在一起会引起冲突了。

```
typedef int t;
struct t {char c; int t;} t;
```

大家知道，可以在内部块中重新声明同一个标识符，这是合法的，而且不会引起冲突，例如：

```
char c = 'x';
{
    int x = c;
    /* ..... */
    long long int c = 0;
```

```
    /* ..... */
}
```

实际上，这对 typedef 名也是一样的，即：

```
typedef int I;          //这里的“I”是 typedef 名
{
    I i = 0;
    i ++;
    /* ..... */
    char I = 0;         //这里的“I”指示一个对象
    I ++;
    /* ..... */
}
```

5.5.1.1　类型定义

关键字 typedef 的作用是类型定义（type definition），也就是将一个标识符（称为 typedef 名）与一个类型相关联。但是，类型定义并非引入一个新的类型，那个标识符仅仅是指定类型的同义词。

类型定义的语法形式类似于对象或者函数声明，只是以 typedef 开头。例如：

```
typedef int f (int, int);
```

其中，将 f 定义为函数类型，该函数接收两个 int 类型的参数，并返回一个 int 类型的值。很显然，类型定义的本质是将标识符定义为它被声明的那种类型。

又如：

```
typedef struct {int i; float f;} stg;
```

这将标识符 stg 定义为它被声明的类型，也就是被定义为上述结构类型，此后即可将 stg 应用在声明中并作为类型指定符。

```
stg sa, * sb;
sa.i = 0;
sa.f = 0.0;
sb = & sa;
```

再如：

```
typedef int A [3];
A a [5];            //a 的类型是 int [5][3]
```

注意，a 的类型是 int [5][3]，而不是 int [3][5]。原因很简单，从声明来看，a 是具有 5 个元素的数组，而每个元素的类型是 A，即 int [3]。

定义一个类型的别名比较简单，也容易理解。但是，将这个别名用在声明中时，可能会造成意想不到的麻烦。请看下面的示例：

```
typedef int * A;
const A a;
```

初看起来，标识符 a 的类型 const int *，因为以上第二行似乎等效于

```
const int * a;
```

事实上，这是错的。你应当这样来解读：a 的类型是 const A，而 A 是一个指针类型的别名，A 所代表的类型是指向 int 的指针，简而言之就是指针。所以，a 的类型是 const 限定的指针（而不能理解为指向“const 限定的类型”的指针）。既然声明的是一个 const 限定的指针，所以，上例中的第二行其实是等效于

```
int * const a;
```

有关如何声明指向“const 限定的类型”的指针和指向某类型的“const 限定的”指针，请参见“声明符”。

函数的返回类型不能是数组或者函数，但可以是数组或者函数的指针。也就是说，可以定义返回一个数组或者函数指针的函数。因此，下例中的 D1 和 D2 是无效的；D3 和 D4 是有效的。

```
typedef void F (int);   //定义 F 为不返回任何值的函数类型
typedef char A [100];   //定义 A 为具有 100 个字符类型元素的数组类型

F functioa (void);      //D1：无效声明
A functiob (void);      //D2：无效声明
F * functioc (void);    //D3：有效
A * functiod (void);    //D4：有效
```

不考虑程序的意义，下面的例子展示了如何定义和使用返回数组指针或者函数指针的函数。

```
# include <stdio.h>
# include <stdlib.h>

typedef void F (int);
typedef char A [100];

void fa (int x) {printf("%d\n", x);}
F * fb (void) {return fa;}
A * fc (void) {return malloc (sizeof (A));}

int main (void)
{
   F * f = fb ();  //f 的类型是 F *，函数 fb 的返回值类型也是 F *
   f (5);

   A * a = fc ();  //a 的类型是 A *，函数 fc 的返回值类型也是 A *
   printf ("%zu\n", sizeof * a);   //表达式*a 的类型是 A
   free (a);
```

```
    return 0;
}
```

以上，指向函数的指针都是完整的对象类型。比如上面的 f，它本身是一个对象，是一个函数类型的指针，指向某个函数。但是，请看下面的例子：

```
typedef int F (void);
F f;        //D
```

这里的 f 不是对象，而是一个函数（的名字），D 的作用是声明一个名字为 f 的函数，函数 f 应该在别的地方有所定义。 因为在声明中使用了 typedef 名，所以看上去比较古怪。

无论如何，以上两行实际上等效于下面的声明：

```
int f (void);
```

把一个函数的声明弄得如此麻烦有意义吗？有的。不管有没有实际用处，语法上的完备性需要支持这种做法。另一方面，如果在一个程序中多次用到同一个函数类型，那么，为它定义一个别名是最好的选择，请看下面的示例：

```
# include <stdio.h>

typedef int F (const char *);

F * h (F f)
{
    f ("abcde\n");
    F g;
    return g;
}

int main (void)
{
    F f;
    f ("12345\n");
    F g;
    g ("67890\n");
    return h (f) ("@#$%^\n");
}

int f (const char * p)
{
    return printf (p);
}

int g (const char * p)
```

```
{
    return printf (p);
}
```

因为多次用到同一种函数类型，所以这里将它定义一个别名 F。为了更清楚地搞明白函数类型别名的意义和用法，下面给出了不使用别名的等效写法：

```
# include <stdio.h>

int (* h (int f (const char *))) (const char *)
{
    f ("abcde\n");
    int g (const char *);
    return g;
}

int main (void)
{
    int f (const char *);
    f ("12345\n");
    int g (const char *);
    g ("67890\n");
    return h (f) ("@#$%^\n");
}

int f (const char * p)
{
    return printf (p);
}

int g (const char * p)
{
    return printf (p);
}
```

声明有声明的语法（参见图 5-1），所以使用存储类指定符 typedef 将一个标识符定义为函数类型的别名时，不能带有函数体，语法图上未表明可以有函数体。下面给出了一个非法的示例。

```
typedef int F (void) {return 0;}
```

另外，下面的函数定义也是奇怪而不合法的。代码的编写者认为 F 被定义为函数类型，则其可以声明一个函数 f 并为它补充函数体，但实际上不存在这种声明形式。

```
typedef int F (void);
```

```
F f {/* ····· */}
```

typedef 名可以重复声明，即可以用存储类指定符 typedef 多次（重复）声明同一个标识符。但是，如果这些重复声明的 typedef 名具有相同的作用域，那么它们的类型必须相同，而且不是可变修改类型。

下例中，D1、D2 和 D3 中的标识符 INT 具有相同的作用域。但是，只有 D1 和 D2 中指定的类型是相同的（int、signed 及 signed int 是同一种类型，除非是在位字段的声明中），所以这两个定义是合法的，而 D3 则不合法；D4 非法的原因是，可变修改类型的标识符不能具有文件作用域（参见“typedef”）；D5 是合法的，因为它所声明的标识符 VLA 具有块作用域；D6 是非法的，因为可变修改类型的标识符不能在同一个块内重复定义，在 D5 中已有此标识符；D7 是合法的，原因不再赘述。

```
typedef int INT;                //D1: 合法
typedef signed INT;             //D2: 合法
typedef unsigned INT;           //D3: 非法
int n = 5;
typedef int VLA [n];            //D4: 非法

void f (void)
{
    typedef int VLA [n];        //D5: 合法
    typedef int VLA [n];        //D6: 非法
    {
        typedef int VLA [n];    //D7: 合法
    }
}
```

5.5.2 extern

关键字“extern”意味着这不是对象或者函数的定义，它的定义在外部，在其他地方，而且一定是个具有文件作用域的定义。

在声明中，存储类指定符 extern 用于指示标识符的外部链接或者内部链接属性。而到底是外部链接还是内部链接，要根据具体的情况确定，具体的描述可参见“链接”、“外部链接”和“内部链接”。

比较常见的误区是将“extern”视为“外部链接”的代名词，以为只要标识符是用存储类指定符 extern 声明的，就一定是外部链接，实则不然。例如，假定某个转换单元的全部内容如下。

```
int x;                  //D1
static int y;           //D2

int main (void)
{
```

```
    extern int x;          //D3
    extern int y;          //D4
    /* ..... */
}

extern int z;              //D5
/* ..... */
```

那么，D3 中的 x 是用存储类指定符 extern 声明的，其作用域从它在 D3 中出现的位置开始，延伸并终止于它所在的块结束。在它的作用域内，能看见在 D1 处声明的另一个标识符 x，且是外部链接的，因此，在 D3 处声明的 x 也是外部链接的，这两个 x 指示同一个对象。如果在组成当前程序的其他转换单元或者库中也有被声明为对象的标识符 x，则它们和 D1、D3 处的 x 一起，指示同一个对象。

D4 中的 y 是用存储类指定符 extern 声明的，其作用域从它在 D4 中出现的位置开始，延伸并终止于它所在的块结束。在它的作用域内，能看见在 D2 处声明的另一个标识符 y，且是内部链接的，因此，在 D4 处声明的 y 也是内部链接的，它们指示同一个对象。如果在组成当前程序的其他转换单元或者库中也有被声明为对象的标识符 y，则它们所指示的对象是独立的，和 D2、D4 所指示的那个对象并不相同。

D5 中的 z 是用存储类指定符 extern 声明的，其作用域从它在 D5 中出现的位置开始，延伸并终止于它所在的块结束。在它的作用域内，看不见标识符 z 的任何其他声明，所以，标识符 z 是外部链接的，它所指示的对象应该是在其他转换单元或者库中声明（定义）的。

5.5.3 static

在用做存储类指定符时，关键字 static 有不同的含义，具体作用取决于具体的声明，现描述如下。

（1）如果被声明的标识符具有文件作用域，则存储类指定符 static 将这个标识符指定为内部链接。

（2）如果一个标识符是在块内声明的，没有使用存储类指定符_Thread_local 且被声明为对象类型，则存储类指定符 static 的作用是将这个标识符所指示的对象指定为具有静态存储期。但是，这不适用于变长数组（参见“变长数组”），变长数组对象不能具有静态存储期。但是，指向可变修改类型的指针可以具有静态存储期。下例中，S1 是非法的，企图使变长数组 vla 具有静态存储期，S2 是合法的，指向变长数组对象的指针可以具有静态存储期。

```
void f (int n)
{
    static int vla [n];          //S1: 非法
    static int (* pvla) [n];     //S2: 合法
}
```

（3）如果一个标识符是在块内声明的，使用了存储类指定符_Thread_local 且被声明为对象类型，则存储类指定符 static 用于配合_Thread_local，将这个标识符所指示的对象指定为具有线程存储期。但是，这不适用于变长数组（参见“变长数组”），变长数组对

象不能具有线程存储期。但是，指向可变修改类型的指针可以具有线程存储期。

要了解和存储类指定符 static 有关的知识，可参见“_Thread_local”、“链接”和“存储期”。在下例中，分别在 A、B、C、D 这 4 处使用了关键字 static，但它们的意思不太相同。

```
static /* A */ int x = 1;

static /* B */ void demo (int a [static /* C */ 5])
{
    static /* D */ int y = 0;
    extern int x;              //E
    /* ..... */
}

extern int x;                  //F
```

其中，A 处的 static 用于指示标识符 x 是内部链接的，而不是用于指示静态存储期。因为 x 具有文件作用域，本来就具有静态存储期， static 的作用不再是指定存储期。同理，B 处的 static 也用于指示标识符 demo 是内部链接的。

C 处的 static 不用做存储类指定符，它只是保证数组 a 至少有 5 个元素，这样 C 实现就能做相应的优化工作。具体可参见“函数声明符”和“数组声明符”。

D 处的 static 用于指定对象 y 具有静态存储期。因为标识符 y 具有块作用域，故这个 static 不用于指示链接，而用于指示存储期。

【思考题】上例中，E 处的标识符 x 和 F 处的标识符 x 是内部链接的还是外部链接的，为什么？

5.5.4 _Thread_local

在多线程开始流行之前，C 的源文件和库可能会定义一些具有文件作用域的标识符，这些标识符所指示的对象都具有静态存储期。但是，在进入多线程时代后，如果不能正确处理这些源文件和库，则它们也会成为负担，毕竟它们不是线程安全的。

一个比较好的想法是将这些具有文件作用域的标识符定义为线程本地存储，并重新编译相关的部分。这样，当线程执行时，每个线程都会得到一个本地的实例。

另一方面，在每个线程的各个部分（函数）之间通过传递参数来共享对象效率不是太高，使用线程本地存储机制也是一个不错的办法。

有了这些想法之后，C 引入了关键字_Thread_local。

关键字_Thread_local 是从 C11 开始引入的。有关线程存储期的详细描述和示例，可参见“线程存储期”。

存储类指定符_Thread_local 不能用于声明一个函数类型，而只能用于对象类型的声明中，但还不能是变长数组。因此，如下声明是非法的。

```
_Thread_local int f (void);
_Thread_local int vla [f ()];
```

如果多个（内部或者外部链接的）标识符都指示同一个对象，那么在它们的声明中要么

都使用存储类型指定符_Thread_local，要么都不使用，应当保持统一。这种要求是合理的，在多线程环境下，“多个标识符都指示同一个对象”意味着在该线程中有一个线程本地对象，该线程代码中的多个标识符都指示该线程本地对象。如果它们的声明不统一，C 实现将不能正确领会程序的意图。至于其他线程，也应按此理解。

如果一个标识符具有块作用域，那么它不能只有存储类指定符_Thread_local 而没有 extern 或者 static。因此，下面的对象声明是非法的。

```
int thrd_func (void * p)          //线程函数
{
    _Thread_local int x = 1;      //非法
    /* ····· */
}
```

这种限制有其合理性，能够兼顾有关链接、作用域和存储期上的传统语义。下面是一个非常好的例子，不但解释了存储类指定符_Thread_local 的不同用法，而且将它们与自动存储期和静态存储期做了比较。

```
# include <threads.h>
# include <stdatomic.h>

int thrd_func (void * p)
{
    /* ····· */
    if (/* ····· */)
    {
        int auto_obj = 0;                                    //D1
        static _Atomic int p_block = ATOMIC_VAR_INIT(0);     //D2
        static _Thread_local int t_block = 0;                //D3
        extern _Atomic int p_globl;                          //D4
        extern _Thread_local int t_globl;                    //D5
        p_block ++;
        t_block ++;
        p_globl ++;
        t_globl ++;
    }
    /* ····· */
}

int main (void)
{
    thrd_t t1, t2;
    thrd_create (& t1, thrd_func, 0);
```

```
    thrd_create (& t2, thrd_func, 0);
    thrd_join (t1, 0);
    thrd_join (t2, 0);
    return 0;
}

_Thread_local int t_globl = 3;                          //D6
_Atomic int p_globl = ATOMIC_VAR_INIT(33);              //D7
```

首先，在 D7 中声明的标识符 p_globl 具有外部链接。这意味着，它指示的对象会在整个程序（进程）启动时创建和初始化，而且只创建一个。要访问这个对象，只需用存储类指定符 extern 重新声明这个标识符即可，就像在 D4 中所做的那样。也正是因为如此，这个对象在每个线程内都能够访问，而且访问的是同一个对象，这必然会涉及数据访问的同步问题。所以，这里将 p_globl 声明为原子类型。

其次，在 D6 中声明的标识符 t_globl 具有外部链接。但它又同时使用了存储类指定符 _Thread_local，这意味着，如果它在线程中使用，则每个使用它的线程都要创建一个独立的对象，而且互不相干。在每个线程中使用它时，要先用存储类指定符_Thread_local 和 extern 重新声明该标识符，就像在 D5 中所做的那样。因为 t_globl 被声明为具有线程存储期，因此，每当与它有关的线程被创建时，将创建一个仅属于当前线程的对象。

再次，D3 中声明的标识符 t_block 是无链接的，而且被声明为具有线程存储期，它所指示的对象在每个使用它的线程中都要单独创建，随线程一起创建，而且只初始化一次，使用它在 D3 中的初始化器（0）。只是，因为标识符 t_block 仅有块作用域，所以不是任何地方都能通过它来访问对象，而是仅限于它的作用域内。传统上，在块内将一个标识符声明为对象类型时，存储类指定符 static 有“始终只创建一个实例”的意思，因此，标准要求在当前这种情况下，必须连带使用存储类指定符 static。此外，创建两个线程存储期的对象，并使用同一个标识符来访问是可能的，例如：

```
static _Thread_local int x;
{
    static _Thread_local int x = 1;
    /* ..... */
}
```

当线程创建时，将同时创建两个对象。只是，每当进入上面的内部块时，访问的是其中的一个对象，而返回到上层块时，访问的是另一个对象。

最后，在 D2 中声明的标识符 p_block 是无链接的，因为使用了存储类指定符 static，它所指示的对象具有静态存储期，在整个程序（进程）启动时创建，而且只创建一个。这个对象在每个线程内都能够访问，而且访问的是同一个对象，这必然会涉及数据访问的同步问题。所以，这里将 p_block 声明为原子类型。但是，标识符 p_block 仅有块作用域，所以，不是任何地方都能通过它来访问该对象，而仅限于它的作用域内。此外，对象 auto_obj 具有自动存储期，每当程序的执行进入它所在的块时创建，返回到上层块时销毁，和进程、线程都没有关系。

上述代码中的 ATOMIC_VAR_INIT 是一个宏，在头文件< stdatomic.h>中定义，用于初始化一个原子类型的对象。如果一个原子类型的对象不用这个宏来初始化，则在它初始化的时候，其他线程对它的访问操作（即使是原子操作）有可能出现竞争的情况。

5.5.5 register

若一个标识符被声明为对象类型，而且声明时使用了存储类指定符 register，这意味着用户希望访问这个对象时应尽可能地快，越快越好。例如：

```
for (register int i = 0; i < 10; i ++) {/* …… */}
```

懂得汇编语言和计算机原理的读者都知道，速度最快的器件莫过于寄存器（register），这里实际上建议使用寄存器。但是，这个建议有没有效果，C 实现会不会采纳这个建议，都还不一定。在最坏的情况下，它可能会把 register 当成 auto 对待。

注意，c 如果一个标识符的声明中含有存储类指定符 register，则它不能具有文件作用域（或者说不能是一个外部声明）。在下面的例子中，非法使用存储类指定符 register 的地方都已经标出，没标出的地方都是合法的。

```
register int m;                    //非法

void regf (register int x)
{
    register int y = x;
    /* …… */
}

register int regg (void)           //非法
{
    /* …… */
}
```

若指示一个对象的标识符是用存储类指定符 register 声明的，那么即使 C 实现没有采纳这个建议，而为这种对象分配了可寻址的存储空间（而不是寄存器），它也不能作为一元运算符&的操作数。

对于用存储类指定符 register 声明的数组，不能隐式转换为指向其第一个元素的指针，只能作为 sizeof 的操作数。例如：

```
register char c [22];
* c = 'a';                         //非法。不允许c转换为指向其第一个元素的指针
char * pc = c;                     //非法。原因同上
size_t len = sizeof c;             //合法
c [1] = 'x';
```

以上最后一行使用下标表达式访问数组 c 的元素，但它依然会被隐式地转换为指针（参见“数组下标”），所以这也是非法的。

使用存储类指定符 register 声明的对象具有自动存储期，可参见“存储期”和“自动存储期”。

5.5.6 auto

存储类指定符 auto 只能用于块内的声明中，它是默认存在的。正因为如此，它基本上不使用，很少在程序中看到它们。

如果一个标识符的声明中含有存储类指定符 auto，则它不能具有文件作用域（或者说不能是一个外部声明），也不能是函数参数的声明。

在下面的例子中，非法使用存储类指定符 auto 的地方都已经标出，没标出的地方都是合法的。

```
auto int m;                 //非法

void regf (auto int x)  //非法
{
    auto int y = x;         //等同于int y = x;
    /* …… */
}

auto int regg (void)    //非法
{
    /* …… */
}
```

使用存储类指定符 auto 声明的对象具有自动存储期，具体可参见“存储期”和“自动存储期”。

5.6 类型限定符

类型限定符用于（和其他类型指定符共同）声明或者指定某种限定的类型，或者原子类型。类型限定符包括 const、volatile、restrict 和_Atomic，其含义和用法详见其各自的词条。

尽管值的类型可以是限定的，但是这些类型限定符仅仅对那些指示对象的表达式（左值）有意义。

如果多次使用了同一个类型限定符且它们都直接作用于同一个实体（声明符），则效果等同于只使用了一个（次）。例如：

```
typedef const int INT;
const INT my_int;
```

以上，对标识符 my_int 的声明相当于

```
const const int my_int;。
```

尽管关键字 const 出现了两次，但该声明相当于 const 只出现了一次，即

```
const int my_int;
```

针对数组类型的限定，被限定的实际上是数组的元素类型。因此，在下面的示例中，尽管表面上看起来 arr1 是 const 限定的数组，但实际上，它的元素类型是 const 限定的，即“const int”。也正因为如此，D2 是非法的。

类型限定符“restrict”仅用于指针，所以 D3 是非法的，因为 arr2 的元素类型并不是指针；相反，D4 是合法的，arr3 的元素类型是指针（int *）。

```
const int arr1 [33];        //D1
arr1 [0] = 1;               //D2
restrict int arr2 [33];     //D3
int * restrict arr3 [33];   //D4
```

注意，在一个声明中，如果声明指定符中既有类型指定符，又有类型限定符，则类型限定符是作用于声明符的。请看下面的示例。

```
const struct t {char c; float f;} u;   //D
struct t v;
typedef const struct {char c; float f;} stru;
stru x, y;
```

以上，尽管 D 既声明了结构类型 struct t 又声明了标识符 u，但类型限定符 const 仅作用于声明符 u。即，u 的类型是 const 限定的，但 struct t 不是。换句话说，声明 D 等效于

```
struct t {char c; float f;} const u;
```

正因为如此，v 的类型也不是 const 限定的。对于声明符 stru 来说，它被定义为 const 限定的结构类型，因此，用它作为类型指定符声明的对象 x 和 y 都是 const 限定的类型。

5.6.1 const

作为类型限定符中的一员，关键字“const”在声明指定符中表示“const 限定的类型”，参见“const 限定的类型”。

5.6.2 volatile

作为类型限定符中的一员，关键字“volatile”在声明指定符中表示“volatile 限定的类型。如果一个对象的值不单单被当前程序修改，还可能被硬件（包括处理器）和其他程序修改，则可以将它声明为 volatile 限定的。

有关 volatile 的其他描述可参见“volatile 限定的类型”。

5.6.3 restrict

作为类型限定符中的一员，关键字“restrict”在声明指定符中表示“restrict 限定的类型”，参见“restrict 限定的类型”。

只有指向对象的指针才能使用 restrict 限定符。

因此，在下面的例子中，D1 是非法的，因为它企图用“restrict”限定一个非指针类型的对象；D2 也是非法的，原因同 D1；D3 是非法的，尽管它对 py 的限定是合法的，但对 py 所指向的类型加以限定是非法的；D4 是合法的，因为 pz 是指向对象类型的指针；D5 是非法的，

因为 pf 是指向函数的指针。

```
restrict int x;                  //D1
restrict int * px;               //D2
restrict int * restrict py;      //D3
int * restrict pz;               //D4
int (* restrict pf) (void);      //D5
```

5.6.4 _Atomic

_Atomic 指示原子类型，它不能用于数组类型的声明，也不能用于函数类型的声明。例如：

```
_Atomic int at;
```

有关原子类型的更多描述，可参见“原子类型”。

5.7 函数指定符

函数类型指定符包括 inline 和_Noreturn，其含义和用法可分别参见其各自的词条，即“inline”和“_Noreturn”。

只有在一个标识符被声明为函数类型时，才允许在它的声明指定符中给出函数指定符 inline 和_Noreturn，因为它们只适用于函数。

如果程序运行在宿主式环境下，那么它的 main 函数有特殊的意义和作用，不能带有这两个函数指定符。

函数指定符 inline 和_Noreturn 可以多次出现在声明中，但无论出现多少次，都等效于只出现一次。

下面是关于如何使用这两个函数指定符的示例。其中，在 D1 中重复使用了函数指定符 inline，其等效于只使用了一次；在 D2 中，为 main 函数指定了_Noreturn，如果程序准备在宿主式环境下编译和运行，则它是非法的。但是，如果程序准备在独立式环境下编译和运行，则它不会出现问题。

```
inline inline void g (int x)    //D1
{
    /* ..... */
}

_Noreturn int main (void)       //D2
{
    /* ..... */
}
```

5.7.1 inline

关键字 inline 是从 C99 开始引入的，它是一个函数指定符（属于声明指定符的一种），用于声明一个内联函数。有关内联函数的更多信息，可参见 “内联定义”。

所谓内联函数，是指那些用函数指定符（声明指定符）inline 声明的函数。

对内联函数的调用将有可能（只是可能，这取决于 C 实现）使函数的代码被放置在调用点，以降低函数调用的开销，这称为 “内联替换” 或者 “内联扩展”。虽然看起来内联替换类似于宏替换，但是内联替换不是单纯的程序文本替换，也不会创建新的函数。

将一个函数声明为内联函数，意味着程序的编写者希望对该函数的调用尽可能快，越快越好。但是，该建议在多大程度上是有效果的，会不会被采纳，则取决于 C 实现。如果函数很大、很复杂，则 C 实现很可能放弃内联替换。

一个很容易想到的问题是，要想进行内联替换，则必须知道函数的定义，即知道函数的内容（函数体）。可见性是一个大问题，但是这个问题需要分 3 种情况来区别对待，而且需要先了解有关内部链接和外部链接的知识。

首先，如果内联函数具有内部链接属性，则没有问题，原因是内部链接的函数只能在它所在的转换单元内调用，在调用点必然可以看见内联函数的定义。如果一个函数是用函数指定符 inline 和存储类指定符 static 定义的，则它会按用户所期望的方式工作：关键字 inline 使得它成为内联函数，而 static 使它具有内部链接。

在下面的例子中，函数 f 只能在当前转换单元内调用，而不能在组成同一个程序的其他转换单元内调用，这不是问题，因为具有内部链接的函数只能在它所在的转换单元内调用，这符合预期。

```
inline static int f (void)
{
    /* ..... */
}

int ildemo (void)
{
    return f ();
}
```

其次，如果在一个函数的声明中既使用了函数指定符 inline，又使用了存储类指定符 extern，则这个函数既是一个内联函数，又具有外部链接。这样的内联函数当然可以在它所在的当前转换单元内调用，但它也可以在组成同一个程序的其他转换单元内调用，因为它是外部链接的。

下例中，内联函数 f 具有外部链接属性，它既可以在函数 ildemo 中调用，又可以在组成当前程序的其他转换单元中调用。

```
inline extern int f (void)
{
    /* ..... */
```

```
}

int ildemo (void)
{
    return f ();
}
```

最后，一个外部链接的函数，如果它在某个转换单元内的声明都使用了函数指定符 inline 而没有使用存储类指定符 extern，则它仅仅是一个内联定义（参见“内联定义”）而不是外部定义。调用一个函数时，该函数必须具有外部定义，但一个函数的内联定义不被视为它的外部定义。因此，前面的两个例子都可以正常编译和转换，但下面的例子就会出现错误（假设这是一个转换单元的全部内容，而且组成当前程序的其他转换单元或者库中没有函数 f 的定义）。

```
inline int f (void);

int demo (void)
{
    return f ();
}

inline int f (void)
{
    /* ..... */
}
```

原因很简单，任何一个函数都必须有外部定义才能调用，而不管这个定义在哪里（另一个转换单元，或者库）。在本节的第一个例子中，函数 f 是内联函数，具有内部链接，而且其定义也是外部定义；在本节的第二个例子中，函数 f 是内联函数，具有外部链接，而且其定义也是外部定义。但是在当前例子中，函数 f 的定义是内联定义，但内联定义不算是外部定义，所以，在函数 demo 内部调用函数 f 时，会因为没有 f 的外部定义而导致程序转换失败。

内联替换只是一个备选方案，如果不实施内联，则必须使用它的外部定义。所以，每个内联函数都必须有一个外部定义。就当前这个例子而言，应当如何为它提供一个外部定义呢？

第一个方案：在组成当前程序的其他转换单元内给出函数 f 的外部定义。如表 5-1 所示，在转换单元 a.c 中给出了函数 f 的内联定义，而在转换单元 b.c 中给出它的外部定义，当程序转换时，C 实现可以根据需要使用它们中的任何一个。

取决于具体的编译器，用户可能需要使用适当的编译选项，因为如果编译器选择兼顾 C99 之前的标准，则它对程序的要求不会太严格，看不出效果。例如，对于 gcc，可能需要手工指定-std=c99 或者-std=c11（最新的版本不需要这些选项，截止到本书完稿时，gcc 发布的最新版本是 5.3）。

表 5-1　内联定义和外部定义示例

a.c	b.c

inline int f (void);
 	
int main (void)
{
    return f ();
}
 	
inline int f (void)
{
    /* ..... */
}
	int f (void)
	{
	    /* ..... */
	}

第二个方案：破坏，让一个内联定义不再是内联定义。如下例所示，假如这是一个转换单元的全部内容，那么根据“内联定义”的概念，如果没有 D 处的声明，则函数 f 的定义是内联定义（而不是外部定义）；一旦加上 D 处的声明，则破坏了函数 f 的定义是内联定义的条件，它的定义就成为外部定义。

```
inline int f (void);

int main (void)
{
    return f ();
}

inline int f (void)          //它是内联定义还是外部定义取决于D是否存在
{
    /* ..... */
}

extern int f (void);         //D
```

引入内联定义，只是提供了外部定义之外的另一种调用选择，但是并没有指定将调用它们中的哪一个。也就是说，这是未指定的行为，由 C 实现自行决定。但是，如果要在程序中获取函数的地址，则获取的总是外部定义（而非内联定义）的地址。进一步地，如果使用这个地址来进行函数调用，则调用的可能是它的内联定义，也可能是外部定义。表 5-2 给出了一个实例，该程序由源文件 a.c 和 b.c 组成，它采用的是第一种方案，可以实际转换和运行以查看它的运行结果。

表 5-2　实际转换

a.c	b.c

# include <stdio.h> 	
 	
inline int f (void);
 	
int main (void)
{
　　printf ("%d\n", f ());
　　printf ("%d\n", (& f) ());
　　return 0;
}
 	
inline int f (void)
{
　　return 0;
}
	int f (void)
	
	{
	　　return 1;
	}

在实践中，为了方便起见，通常将函数的内联定义放在头文件中。如表 5-3 所示，当前程序由 main.c 和 public.c 组成，在头文件 public.h 中给出了内联函数 f 的内联定义，在 public.c 中，引入这个内联定义，同时用一个外部链接的声明使它变成外部定义。如此，所有需要调用这个函数的源文件都可以用#include 指令来包含这个头文件。在这个程序中，既有函数 f 的内联定义，有它的外部定义。

表 5-3　头文件中的内联定义

public.h	public.c

inline int f (void)
{
　/* …… */
}
/* …… */
	# include "public.h"
	
	extern int f (void);
	/* …… */

例如，假如源文件 main.c 的内容如下。

```
# include <stdio.h>
# include "public.h"

int main (void)
{
    return printf ("%d\n", f ());
}
```

那么，因为转换单元是经过预处理的源文件，所以源文件 main.c 经过预处理之后，得到的转换单元包含了函数 f 的内联定义。同理，源文件 public.c 也用预处理指令#include 包含了

头文件 public.h，该源文件里还存在函数 f 的另一个声明，而且使用了存储类指定符 extern，这就使得函数 f 在源文件 public.c 中的定义成为外部定义，很好地解决了上面提出的问题。

根据前面的叙述可知，一个函数的内联定义和外部定义分别位于同一程序的不同转换单元中。但是，内联定义和外部定义的函数体应当相同，否则程序的行为是未定义的。进一步地，如果在函数的内联定义中创建任何静态存储期或者线程存储期的对象，则 C 实现很难将它们与外部定义中的这些对象链接在一起，这将创建各自独立的对象。所以，标准禁止在一个函数的内联定义中创建任何可修改的、具有静态存储期或者线程存储期的对象，也不允许使用任何具有内部链接的标识符。

因此，在下面的例子中，D1、D2 和 D3 都是禁止的，D1 企图在函数体内创建静态存储期的对象；D2 企图引用内部链接的标识符 x；D3 企图引用内部链接的标识符 g。

```
static int x;
static int g (void) {/* ····· */}

inline int f (void)
{
    static int m = 0;    //D1
    x ++, m ++;          //D2
    return g ();         //D3
}
```

5.7.2 _Noreturn

绝大多数函数在执行完毕后，会将控制返回给它的调用者。但是，如果一个函数是用函数指定符_Noreturn 声明的，则表明程序的编写者不准备将控制返回给它的调用者。这是一个优化提示，如果不小心违反了这个规定，那么程序的行为是未定义的，C 实现将给出一个诊断信息。

函数指定符_Noreturn 是一个优化提示，等于告诉 C 实现当前函数不会返回给它的调用者，不需要为函数的返回做任何准备和清理工作。该函数指定符用于在 C11 中重新声明一些不用返回的库函数，包括 abort、exit、_Exit、quick_exit、longjmp、thrd_exit 等。当然，也可以用它来定义自己的函数，前提是它们从不返回。

关于函数指定符_Noreturn 的用法，以下是一个正面的例子。

```
_Noreturn void raise_error (void)
{
    printf ("A error encounted, abort.\n");
    abort ();    //库函数 abort 是用_Noreturn 声明的，且从不返回
}
```

相反，以下是一个反面的例子。C 实现并没有为当前函数的返回做任何准备工作（比如栈的清理和平衡栈指针），但函数实际上在打印一行内容之后立即执行返回动作，这样做的后果是难以预料的。

```
_Noreturn void raise_error (void)
{
   printf ("Out of memory.\n");
}
```

此外，很多与执行环境交互的库函数都是用函数指定符_Noreturn 声明的。例如：

```
# include <stdlib.h>
_Noreturn void abort(void);
_Noreturn void exit(int status);
_Noreturn void _Exit(int status);
```

5.8　对齐指定符

在声明一个对象时，根据实际需要，可以使它对齐于某些特定的字节地址，这时可以在声明中使用对齐指定符。换句话说，对象指定符只适用于对象类型的声明，它使标识符所指示的对象或者该对象的成员按指定的要求对齐。对齐指定符的语法形式如图 5-6 所示。

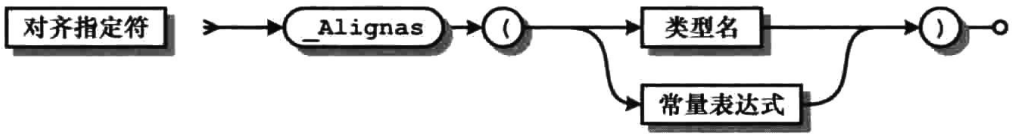


图 5-6　对齐指定符的语法

对齐指定符的第一种形式使用了类型名，这可以看做给出一个样例，使 C 实现参照该样例实施对齐。也就是说，对齐要求和类型名所指定的类型相同。

在下例中，用户的意图是使对象 c 也具有 long long int 类型的对齐要求。

```
_Alignas (long long int) char c;
```

对齐指定符的第二种形式更为直观，因为它直接用一个常量表达式给出了对齐要求。这里的常量表达式要求是整型常量表达式，它求值的结果要求是一个基础对齐（参见“基础对齐”）或者扩展对齐（参见“扩展对齐”）。如果求值的结果是一个扩展对齐，则当前的上下文环境应当能够支持这种对齐。

实际上，第一种形式终究要转化为第二种形式，因此，第一种形式等效于

```
_Alignas (_Alignof (类型名))
```

注意，这里的“类型名”要用实际的类型名来代替。下面是其他一些使用对齐指定符的例子。

```
_Alignas (16) int i;
int _Alignas (long long) j;
```

下例中，对标识符 i 的声明是非法的，因为指定的对齐必须是 2 的幂，而给出的常量表达式 3 并不符合这个要求：

```
int _Alignas (3) i = 0;
```

如上所述，对齐指定符的第二种形式要求是一个整型常量表达式。如果这个表达式求值的结果是0，则不会有任何效果，就像没有这个对齐指定符一样。

下例中，对标识符j的声明虽然指定了对齐要求，但没有效果：

```
int _Alignas (0) j;      //等效于 int j;
```

对齐指定符也不是随便乱用的，它只适用于对象的声明，包括但不限于结构和联合的非位字段成员。因此，用存储类指定符typedef定义一个类型的别名时，不能使用对齐指定符，因为它声明的是一个typedef名而非对象；声明位字段、函数、函数参数的时候，也不能使用对齐指定符，对齐属性对它们而言没有意义。如果在声明中出现了存储类指定符register，则因为寄存器的使用和对齐无关，更不能使用对齐指定符。

这里是两个非法使用对齐指定符的例子。

```
register int _Alignas (16) i = 0;        //有存储类指定符 register
_Alignas (double) void f (double);       //不适用于函数
```

相反地，下面是一个正确使用了对齐指定符的例子。

```
struct t {_Alignas (2) char c; _Alignas (double) int i;} t;
```

允许在一个对象的声明中多次使用对齐指定符，只要它们都是合法的对齐指定。在这种情况下，C实现将选择和应用那个最严格的对齐要求。

在下例中，要求对象i对齐于8比要求它对齐于1更严格。假定int类型的最低对齐要求是4，那么对象i将最终对齐于8。

```
_Alignas (8) int _Alignas (1) i;
```

针对同一个对象多次使用对齐指定符的，或者为该对象及其成员都使用了对齐指定符的，C实现将计算出一个合并的对齐，这个合并的对齐不得小于对象的类型所要求的对齐，也不得小于任何成员对象的类型所要求的对齐。如果int类型的对齐要求是4，long long int类型的对齐要求是8，那么对于以下声明，前两个没有问题，最后一个是非法的：

```
_Alignas (4) int a [6];
_Alignas (8) int b [8];
_Alignas (1) _Alignas (4) long long int c;
```

这是因为，为数组a和b指定的对齐可以和数组元素类型所要求的对齐相同，或者更严格。而对于对象c来说，即使采用了最严格的对齐4，它也小于long long int类型所要求的对齐8。

同样地，下例中，为数组b指定的对齐和double类型一样，该数组的元素类型是unsigned char，可以对齐于任何字节地址，所以它也是合法的。

```
_Alignas (double) unsigned char b [5 * sizeof (double)];
```

与上面的示例相反，下面的声明都是不合法的。原因很简单，为数组对象指定的对齐在数值上小于其元素类型所要求的对齐。

```
_Alignas (1) int a [6];
_Alignas (char) long long int b [5 * sizeof (double)];
```

在编者的机器上，下例中的D1是合法的，结构的成员c是字符类型，可以对齐于任何字节地址（对齐要求是1），这里为它指定的对齐是4；结构类型struct s所要求的对齐是4，原

则上，对象 s 所要求的对齐也是 4，但这里为它指定的对齐是 8。因此，这里有 3 个对齐要求：成员 c、结构类型 struct s、结构对象 s。

类似地，在 D2 中，结构的成员 c 是字符类型，可以对齐于任何字节地址（对齐要求是 1），我们为它指定的对齐是 16。问题在于，结构类型 struct t 所要求的对齐也是 16。原则上，对象 t 所要求的对齐也是 16，但为它指定的对齐是 8，这是不行的，所以 D2 非法。

```
_Alignas (8) struct s {_Alignas (4) char c;} s;     //D1
_Alignas (8) struct t {_Alignas (16) char c;} t;    //D2: 非法
```

如果多次声明了同一个对象，这些声明分散位于组成同一个程序的不同转换单元，且使用了不同的对齐指定符，则行为是未定义的，程序可能出错。

当多次声明了同一个对象时，不管它们位于同一个转换单元还是位于不同的转换单元，如果在它的定义中使用了对齐指定符，那么该对象的其他任何声明要么指定同等的对齐，要么不使用对齐指定符。

在下例中，D4 是对象 i 的定义，使用了对齐指定符。它在 D2 处的声明没有使用对齐指定符，这是允许的；它在 D3 处的声明使用了对齐指定符，而且用的是常量表达式，该表达式的值是 16，等于对象 i 在定义时指定的对齐，所以也是合法的。唯一有疑问是 D1，如果 long long int 所要求的对齐和 int 一样，则 D1 也没有问题，否则 D1 就是非法的。

```
void f (void)
{
    extern _Alignas (long long) int i;              //D1
    extern int i;                                   //D2
    extern _Alignas (sizeof (char) * 16) int i;     //D3
}
_Alignas (16) int i = 0;                            //D4
```

当多次声明了同一个对象时，不管它们位于同一个转换单元还是位于不同的转换单元，如果在它的定义中没有使用对齐指定符，那么该对象的其他任何声明都不可以使用对齐指定符。

正是因为这个原因，下面的程序将转换失败，因为在对象 i 的定义（D2）中没有使用对齐指定符，而在它的声明中却出现了对齐指定符（D1）。

```
int main (void)
{
    extern _Alignas (sizeof (int)) int i;       //D1
    i = 1;
    extern int i;
    /* ..... */
}
int i = 0;                                      //D2
```

再如，以下代码中对象 c 被声明了 3 次，分别位于 D1、D2 和 D4。其中，D4 是它的定义，使用了对齐指定符。其他两次声明 D1 和 D2 可以使用对齐指定符，也可以不使用。如果使用，则要求具有等同的对齐。不管怎样，对象 c 的 3 次声明都没有问题。相反，对象 d 的

两次声明 D3 和 D5 违反了规定，是不合法的。

```
unsigned char c;                                    //D1

void f (int n)
{
   extern _Alignas (int) unsigned char c;           //D2
   extern long _Alignas (64) double d;              //D3
}

_Alignas (int) unsigned char c = 'x';               //D4
long double d = 2.0f;                               //D5
```

5.9 声明符

声明符的语法如图 5-7 所示。在一个声明中，每个声明符声明一个标识符。声明符可以用指针、标识符、数组和函数参数直接构建，例如下例中字体加粗的部分。其中，pf 是指针，指向一个函数；d 是变长数组；pa 是指针，指向数组。

```
int i;
unsigned char x, y = 0;
float * f;
int (* pf) (const char * restrict, ...);
double d [n];
char (* pa) [] = & (char []) {0, 1, 1,};
```

当然，也可以在其他声符的基础上递归构建，例如下例中字体加粗的部分。p 是指针，指向一个数组，数组的元素类型是 int *；f 是函数，返回一个指针，指向数组 float [5]。

```
int * (* p) [3];
float (* f (void)) [5];
```

5.9.1 指针声明符

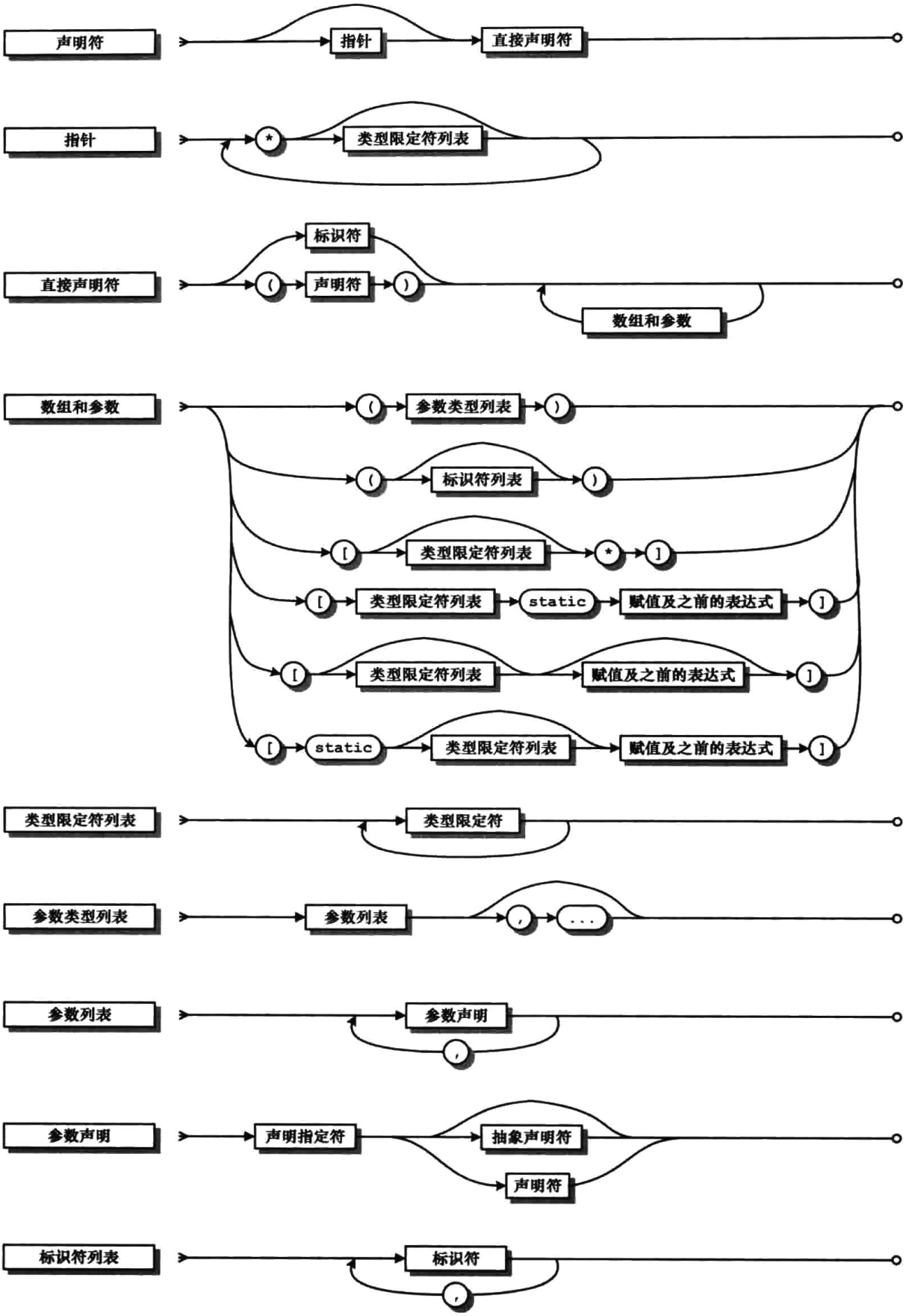
指针声明符声明一个指针，如一个指向 float 类型的指针：

```
float f = 1.0f, * pfloat = & f;
```

这里，指针 pfloat 本身是一个对象，但是它具有指针类型的值，即对象 f 的地址。通过这个指针值（地址），可以间接访问对象 f。

可以声明指向数组的指针，例如：

```
char a [] = {"Moscow",}, (* pa) [] = & a;
```



抽象声明符　见图 5-8（类型名）

图 5-7　声明符的语法

这里，pa 被声明为指向数组的指针，a 的类型是数组，所以表达式&a 的类型是指向数组的指针，可以用于初始化 pa。和 pa 不同，下面的 pb 是指向 char 的指针，数组 b 自动转换为指向其第一个元素的指针，可以用这个指针初始化对象 pb，因为它们类型相同。

```
char b [] = {"Soviet",}, * pb = b;
```

可以声明指向结构或者联合的指针，例如：

```
struct s {char c; float f;} s, * ps = & s;
struct t {char c; float f;};
struct t t, * pt = & t;
```

这里，对象 s 的类型是 struct s；对象 t 的类型是 struct t。因此，&s 和&t 的类型分别是 struct s *和 struct t *，指针 ps 和 pt 的类型也分别是 struct s *和 struct t *。

可以声明指向函数的指针，例如：

```
# include <stdio.h>

void f (int x, int y) {/* ····· */}

int main (void)
{
    int (* pm) (void) = main;
    void (* pf) (int, int) = f;
    int (* ppf) (const char * restrict, ...) = printf;
    return ppf ("ppf is a pointer to function return int.\n");
}
```

这里，pm 的类型是指向“不接收任何参数的、返回 int 的函数”的指针；pf 的类型是指向“接收两个 int 类型参数的、不返回任何值的函数”的指针；ppf 也是一个指针，它所指向的函数在类型上与函数指示符 printf 相同（printf 函数是在头文件<stdio.h>中声明的）。

当然，还可以声明更复杂的指针。这里有几个例子，但是并不算太复杂。

```
char c, * pc = & c, * * ppc = & pc, * * * p3c = & ppc;
void (* (* p0) [5]) (const char *);
int (* (* p1) (const char *)) [6];
char * (* (* const p2) [5]) (int);
char * * const * const p3;
struct t {char c; int a [22];} (* (* pa) [3]);
```

其中，标识符 pc 被声明为一个指向 char 类型的指针；ppc 被声明为一个指向“指向 char 的指针”的指针；p3c 被声明为一个指针，它指向另一个指针，而后者又是指向 char 的指针。

p0 被声明为一个指向数组的指针，被指向的数组有 5 个元素，元素的类型是指向函数的指针，被指向的函数接收一个 const char *类型的参数，但不返回值。

p1 被声明为一个指向函数的指针，该函数接收一个 const char *类型的参数，并且返回一个指向数组的指针，被指向的数组有 6 个 int 类型的元素。

p2 被声明为一个指向数组的、const 限定的指针，被指向的数组具有 5 个元素，元素的类

型是指向函数的指针。被指向的函数接收一个 int 类型的参数，其返回类型是 char *。

p3 被声明为一个指向“const 限定的指针”的指针，而且 p3 自己本身也是一个 const 限定的指针。p3 所指向的那个指针是一个指向“指向 char 的指针”的指针。

pa 被声明为一个指向数组的指针，被指向的数组有 3 个元素，元素的类型是指向结构的指针。

5.9.2　数组声明符

最简单的数组声明符由一个标识符、一个“[”和一个“]”组成，例如：

```
float f [];
```

这将声明一个数组 f，但它的大小未知，因为没有指定数组的大小。未指定大小的数组类型是不完整类型。

相反地，要声明一个完整的数组类型，需要用表达式给出数组的大小，而且数组的元素类型也必须是完整类型，例如：

```
struct t {char c; float f;} at [5];
```

这里，数组的大小（元素的数量）是已知的，数组的元素类型是结构，而且是完整的结构类型。

但是，从 C99 开始，数组的大小不必是一个常量，它的元素类型也不必具有常量大小。若果真如此，则声明的是一个变长数组（参见“变长数组”）。例如：

```
void f (int n)
{
    float f [n];

    typedef double VLA [n];
    VLA u [3];
    VLA v [n];
    VLA w [n] [n];

    struct t;
    struct t * at [3] [n];
}
```

其中，f 是变长数组，其大小并非由常量表达式指定；u 也是变长数组，其元素类型是变长数组（可变修改的）；v 和 w 都是变长数组，不但大小不是用常量表达式指定，而且它的元素类型也是变长数组；at 也是变长数组类型，它具有 3 个元素，但元素的大小未知。注意，尽管 struct t 是不完整类型，但数组 at 却是完整的数组类型，因为它的元素类型是指向结构的指针，指针在任何时候都是完整的对象类型。

在声明一个没有函数体的函数时，或者当用户需要用存储类指定符 typedef 定义一个函数类型的别名时，如果该声明使用的是函数原型，且参数的类型是变长数组，则用于指定这个变长数组大小的表达式可以用“*”来代替。

反过来说，这里的“*”用于指示一个变长数组类型的函数参数。例如：

```
int f (int a [*]);
typedef void F (int x [*]);
```

但是，这种做法不能出现在函数定义的参数类型列表中，也不能出现在其他任何形式的声明中。例如：

```
int a [*];                                  //非法
void f (int [restrict *], int (*) [*]);     //正确
void g (int a [*], int i);                  //正确
void h (int [*], int) {/* …… */}            //非法，不适用于函数定义
```

大家知道，如果一个函数原型没有函数体，且它的参数声明中含有标识符，则这个标识符具有函数原型作用域（可参见“作用域”及“函数原型作用域”）。如果一个变长数组具有函数原型作用域，且它的大小是一个表达式而不是“*”，则使用这个表达式和使用“*”没有区别。

例如，给定声明

```
int n = 5;
```

则函数原型

```
void f (int a [n]);
```

等效于

```
void f (int a [*]);
```

5.9.3 函数声明符

函数声明符总体比较简单，但是考虑到传统的函数声明和函数原型不一样，所以需要分开来介绍。

在传统 K&R 形式的函数声明中，若不带函数体，则函数声明符中的参数部分要求是“(”和“)”，中间不能有任何内容，如

```
float f ();
```

这意味着，传统 K&R 形式的函数声明无法用来判定参数的类型和数量。

相反地，同样采用传统的 K&R 形式来声明一个函数，如果带有函数体（这意味着一个函数定义），则函数声明符中包含一个标识符列表；如果标识符列表为空，则表示这个函数不接收任何参数。例如：

```
void f () {/* …… */}                       //D1
int h (x, y) int x; int y; {/* …… */}      //D2
void g ();                                  //D3
```

其中，D1 是一个函数定义，但这个函数被声明为不接收任何参数，因其标识符列表为空；D2 也是一个函数定义，但它的标识符列表不空，所以它接收两个参数；D3 仅仅是一个函数声明，它的参数类型和数量都不清楚。正是因为这样，当调用这几个函数时，C 实现可能无法对传入的参数实施检查。所以，针对上面的函数声明，下面的调用不会在转换时出现问题，但程序运行的时候可能会出现问题。

```
f ();
f (1);
h ('x');
```

```
h (5, 6);
g ();
g (1);
g (3, 0.1f, 'x');
```

标准 C 引入了函数原型，这有利于转换期间的参数检查工作。函数原型在函数声明符中包含了参数类型列表。如果函数不接收参数，则不管是函数声明还是函数定义，参数类型列表仅仅是一个 void（不得声明 void 类型的标识符）；函数接收参数的，参数类型列表中包含了以逗号分隔的类型名。如果是函数定义，则还必须为每个参数提供用于指示形参对象的标识符，但对无函数体的函数声明来说，标识符是可选的。

下面是几个函数原型的例子。

```
void f0 (void);
float f1 (float, float);
void f2 (const char * p) {/* …… */}
int * f3 (int (* pa) [], int (* pf) (void)) {/* …… */}
```

上例中的声明都采用了函数原型。对函数 f0 的声明不是函数定义，参数类型列表中的 void 表明它没有参数；对函数 f1 的声明也不是函数定义，从参数类型列表中可以知道它接收两个 float 类型的参数，但省略了参数名称。当然，参数可以有名称，因此该声明等效于以下声明：

```
float f1 (float x, float y);
```

对 f2 的声明也是它的定义，从参数类型列表中可知，该函数接收一个 const char *类型的参数，但这里的标识符 p 不能省略。对函数 f3 的声明也是它的定义，从参数类型列表中可知，该函数接收两个参数，第一个参数是指向数组的指针，第二个参数是指向函数的指针。

不管函数是用传统 K&R 形式声明的，还是采用函数原型声明的，在声明它的参数时，唯一可以使用的存储类指定符是 register。

对于函数原型，如果参数类型列表不是函数定义的一部分，那么即使是在声明参数时使用了存储类指定符 register，也不会起作用，而是直接被忽略。

下面是几个示例。

```
void f (register int);                    //忽略 register
void f (register int i) {/* …… */}        //有效，register 起作用
int g (x) register int x; {/* …… */}      //有效，register 起作用
```

函数的返回类型不能是数组或者函数，也不能具有数组或者函数类型的参数。尽管有时候函数的参数以数组或者函数的形式出现，但也不意味着它们接收这样的参数。实际上，如果参数的类型是返回 T 的函数，则被调整为指向“返回 T 的函数”的指针；如果参数的类型是 T 的数组（元素类型为 T 的数组），则被调整为指向 T 的指针。这些知识在本书中的其他部分已经介绍过。

例如：

```
char f (void) [10];
char (* g (void)) [10];
int h (int a [], int f (void));
```

其中，对函数 f 的声明是非法的，因为它企图返回一个数组；对函数 g 的声明是合法的，其参数类型是 void（不接收参数），返回一个指向数组的指针 char (*)[10]；对函数 h 的声明也是合法的，第一个参数名义上是数组，但被调整为指向数组首元素的指针 int *，第二个参数名义上是函数，但被调整为指向函数的指针 int (*)(void)。上面 3 个例子的参数在经过调整之后都是指针，指针是完整对象类型，所以编译不会出现问题。

接下来讨论另一个话题，对于函数声明：

```
int demo (const int a []);
```

数组 a 的元素类型是 const int。在程序转换时，参数 a 将被调整为指针 const int *。那么，如果想使调整后的指针本身是常量指针，该怎么办呢？进一步地，如果想使调整后的指针是一个限定的指针，该怎么办呢？

如果函数的参数是一个数组，元素类型为 T，而且在它的“[”和“]”中间有限定符 q，则该数组类型的参数在调整后，将得到一个指向类型 T 的、q 限定的指针。

下面是几个非常典型的例子。

```
float f (float a [restrict], float b [restrict]);
void g (const int m [const]);
long h (long int n [const])
{
    n ++;                    //非法，指针 n 是常量指针，其值不可修改
    return ++ * n;
}
```

其中，形参 a 调整之后的类型是 float * restrict，即一个 restrict 限定的指针，指向 float。形参 b 在调整后也是同一种类型。

形参 m 在调整之后的类型是 const int * const，即一个 const 限定的指针，指向 const int。

形参 n 在调整之后的类型是 long int * const，即一个 const 限定的指针，指向 long int。

如果函数的参数是一个数组，而且在它的“[”和“]”中间有关键字 static 及一个常量表达式，则这是一个优化提示，用于告诉 C 实现，函数的调用者保证该数组的第一个元素总是存在的、可供访问的，元素的数量由 static 后面的表达式给出，而且至少有这么多，这有助于在进入函数时预取数组的内容。

在下例中，第一行是函数 f 的声明（不带函数体），第二行是它的定义。首先，参数 a 会被调整为指针，其类型为 char * restrict。不仅如此，因为这里使用了关键字 static，所以 C 实现可以假定调用该函数时，传入的指针不会是空指针，因为原始数组的首元素总是可以访问的，而且至少有 5 个元素。据此它可以对该函数的代码做适当的优化。

```
void f (char [restrict static 5]);
void f (char a [restrict static 5]) {/* …… */}
```

传统 K&R 形式的函数可以接受不同类型和不同数量的参数。但即使采用了函数原型，也可以声明一个参数的类型和数量都不确定的函数。其方法是让参数类型列表终止于一个省略号，用于表明跟在最后一个逗号后面的参数在类型和数量上都不确定。例如：

```
void f (int, ...);
```

注意，为了获取这些数量未知的参数，可以使用一些宏，这些宏在头文件<stdarg.h>中定

义。如何使用这些宏不是本节讨论的主题，请参见其他相关资料。

再来看另一个问题，对于以下声明：

```
typedef float xx;
int f (int xx) {/* …… */}
```

其中，作为 typedef 名的标识符 xx 和作为函数参数的 xx 不会冲突，因为标识符 xx 在函数声明符中的声明隐藏了它先前的属性。但是，如果函数声明采用了传统 K&R 风格，则可能引起混乱：

```
typedef float xx;
int f (xx) int xx; {/* …… */}
```

在上面的函数声明中，是先出现标识符 xx，然后对它声明。这就出现了问题：函数声明符中的 xx 是参数名（K&R 风格的函数声明），还是 typedef 名（函数原型）？标准规定，如果一个函数参数的名称也可以被视为 typedef 名，则它被解释为 typedef 名。

正是因为如此，处于函数参数位置的 xx 被视为 typedef 名。也就是说，这是一个类型名，后面没有标识符，这应该是一个不带函数体的函数原型。问题在于，它后面的内容使其看起来是传统风格的函数定义，所以这是一个非法的声明。

在下面的例子中，若注释掉 D1，则 D2 和 D4 是非法的（D2 有时候只是引起编译警告）；若保留 D1，则 D3 是非法的。

```
typedef char * pchar;                          //D1
void f (pchar);                                //D2
void f (pchar) char * pchar; {/* …… */}        //D3
void f (pchar p) {/* …… */}                    //D4
```

5.9.4　全声明符

全声明符（full declarator）首先是一个声明符，但它自身是独立和完整的，不是其他声明符的组成部分。在下例中，有两个字体加粗的地方，它们分别是两个全声明符，这两个全声明符又分别包含了标识符 x 和 pa。x 的类型是 int；pa 的类型是数组，数组的元素类型是“指向函数的指针”，该函数接收两个 int 类型的参数，并返回一个 int 类型的值。

```
int x = 0, (* pa [5]) (int, int);
```

5.10　初始化

使用初始化器提供的值、C 实现提供的默认值来修改对象存储值的行为，称为初始化（Initialization）。请参见“初始化器”。

很多书和教材，包括有些被认为是很经典的那些，将初始化和赋值混为一谈，这是不正确的。初始化是初始化，赋值是赋值，将它们分开，有助于更好的学习和理解这门语言：

```
void f (void)
{
    int a = 0, * b = & a, c;     // “=”的作用是指定初始化器
```

```
    c = * b ++;                    //“=”的作用是赋值
}
```

如果一个具有自动存储期的对象在声明的时候没有提供初始化器，则它在创建的时候不会初始化，因而具有不确定的值。一个较好的习惯是在它使用之前明确地赋值，比如下面的例子，对象 x 具有自动存储期，在声明时没有初始化器。当它在循环语句中使用时，先被赋值为 0，然后才开始使用，这是合法的：

```
int x;          //x 的值此时尚不确定
for (x = 0; x < 30; x ++) {/* …… */}
```

如果一个对象被声明为具有静态或者线程存储期，但是在它的声明中没有提供初始化器，那么，它在创建的时候，C 实现会依据它的类型选择以下方式之一进行初始化：

（1）如果是指针类型，会被自动地初始化为空指针；

（2）如果是算术类型，会被自动地初始化为 0。注意，有些 C 实现会同时使用无符号零和负零，在这种情况下，可能会被初始化为正零，也可能会被初始化为无符号的零；

（3）如果是数组类型，依照上述 1、2 所指定的方法初始化其每个元素，元素类型依然是数组的，递归使用本条；

（4）如果是结构类型，依照前述 1、2 和 3 所指定的方法初始化其每个成员。结构的成员依然是结构的，递归使用本条；结构的成员是联合的，依照第 5 条的方法初始化。最后，结构内部的任何填充比特都被初始化为 0；

（5）如果是联合类型，依照前述 1~4 所指定的方法初始化其第一个命名（有名字的）成员；联合的成员依然是联合的，递归使用本条。最后，联合内部的任何填充比特都被初始化为 0。

下例中，对象 x、p、a、pa、pf、t 和 pt，包括数组 a 的每一个元素、结构 t 的每一个成员，甚至结构成员 a 的每一个元素，都在程序启动的时候初始化，尽管这里没有给出任何一个初始化器：

```
int x, * p, a [3], (* pa) [3], (* pf) (void);
struct t {char c; float f; char a [3]; struct t * pt;} t, * pt;
int f (void) {/* …… */}
```

以上，对象 x 被初始化为零值（0 或者+0）；指针对象 p 被初始化为空指针；对象（数组）a 的每一个元素都被初始化为整数 0；指针对象 pf 被初始化为空指针；结构对象 t 的成员 c 被初始化为整数 0，成员 f 被初始化为浮点数 0.0，成员 a 的每个元素都被初始化为整数 0，成员 pt 被初始化为空指针；对象 pt 被初始化为空指针；结构对象 t 内部各成员之间的空隙和尾部的填充（如果有的话）都被初始化为 0。

标量对象的初始化器比较简单，只是一个表达式。如果标量对象的声明中带有初始化器，则该对象的初始值直接来自该表达式，适用于简单赋值（参见“简单赋值”）的类型约束和类型转换过程也同样适用于标量对象的初始化，除了一点：对象的初始化会忽略其声明中的类型限定符。例如：

```
char * p = {"Initializer.",};
```

这里，p 的类型是指向 char 的指针，指针是完整的对象类型。字面串的类型是数组，它将转换为指向其首元素的指针（char *）。所以，这个声明等效于：

```
char * p = "Initializer.";
```

在下面的例子中，尽管 f 被声明为 const 限定的类型，不可以赋值，但它在初始化的时候，类型限定符被忽略。另外，浮点常量 0.01 的类型是 double，但它是在被转换为 float 之后才用于初始化对象 f 的。

```
const float f = 0.01;   //合法，初始化会忽略类型限定符
f = 0.625;              //非法，f 具有 const 限定的类型
```

结构和联合对象的初始化器复杂一些，可以是一个初始化器列表，请参见“初始化器”。不过，结构和联合对象的初始化器也可以是单一的表达式，前提是两者的类型兼容。例如下面的例子，第一行中，对象 t 的初始化器采用了初始化器列表的形式，但第二行中的两个初始化器是形式上单一的表达式：

```
struct t {char c; float f;} finit (void)
{
    struct t t = {'x', 0.01f}, f (void);    //D1
    struct t ta = t, tb = f ();             //D2
    return tb;
}
```

如果一个结构或者联合对象包含未命名的成员，且该对象的初始化器是单一表达式，则整个对象的值，包括未命名成员的值，都来自于初始化器（表达式）的值。在其他情况下，结构或者联合对象中的未命名成员不参与初始化。

结构对象中的未命名成员具有不确定的值。联合对象特殊一些，因为所有成员共用全部或者部分存储空间，其未命名成员的值将受其他命名成员的影响。

首先，只有那些自动存储期的结构或者联合对象才允许使用单一表达式做为初始化器，因为具有静态存储期的结构或者联合对象需要一个在转换期间求值的常量表达式做为它的初始化器，但不存在结构或者联合类型的常量和常量表达式（参见“常量”和“常量表达式”）：

```
struct t {char c; float f;} a = {'x', 0.01f};
struct t b = a;                             //非法，初始化器只能是常量表达式

void f (void)
{
    struct t a = {'x', 0.01f}, b = a;   //合法
}
```

其次，不管结构对象具有何种存储期，如果它没有初始化器，或者初始化器具有列表的形式，则未命名的成员都不会被初始化。例如：

```
# include <limits.h>

struct t {char c; int : 3; float f;} t = {'x', 0.01f};
union u {signed : CHAR_BIT * sizeof (int); signed s;} u = {9};
```

上例中，在初始化对象 t 时，'x'用于初始化对象 t.c；紧接着，0.01f 用于初始化对象 t.f 而不是 c 后面那个未命名的位字段成员（对象）。初始化完成后，未命名成员对象的值是不确定

的；初始化联合对象 u 时，'x'用于初始化第一个命名的成员对象 u.s。因为成员对象 s 和未命名成员的存储空间完全重叠，所以，可以认为该未命名成员的值也是 9。

每一个用花括号括起来的初始化器都关联着一个对象，在初始化此对象及其子对象的值时，使用此初始化器。以下将分几个层次来详细讨论对象、子对象和初始化器的关联，以及它们是如何运作的。

首先，如果初始化器关联的是一个标量类型的对象，则该对象没有子对象，初始化器直接用于为这个对象提供初始值。例如：

```
double d = {0.01};
```

其次，从 C99 开始，允许在聚合和联合对象的初始化器中使用指示器列表，以指示一个特定的子对象。但是，由“[”、常量表达式和“]”组成的指示器只能和数组类型的对象关联；由“.”和标识符组成的指示器只能和结构、联合类型的对象关联。

在下例中，花括号中的初始化器和对象 fa 关联，该初始化器用于初始化对象 fa 的各个子对象。特别之处在于，这里使用了 3 个指示器列表来指示每一个子对象，但每个指示器列表中都只包含一个指示器：指示器[2]指示下标为 2 的数组元素，它后面的初始化器 0.6 是为它指定的初始值；指示器[0]指示下标为 0 的数组元素，它后面的初始化器 0.1 是为它指定的初始值，其他以此类推。很显然，如果使用指示器，则不必按子对象的原始顺序为它们提供初始值。

```
float fa [3] = {[2] = 0.6, [0] = 0.1, [1] = 0.6};
```

对于多维数组，每个指示器列表由多个指示器组合而成。在下例中有 3 个指示器列表，每个指示器列表由 2 个指示器组合而成。对于每个指示器列表而言，从左往右，第一个指示器所指示的对象，是直接包围它的那对花括号所关联的对象的子对象；第二个指示器所指示的对象，是上一个指示器所指示的对象的子对象；其他指示器以此类推。

这就是说，就下例中的第二个指示器列表[0][1]而言，直接包围它的花括号与对象 ia 关联，所以指示器[0]指示的对象是 ia 的子对象 ia[0]；指示器[1]指示的对象是 ia[0]的子对象 ia[0][1]。

```
int ia [2][3] = {[0][0] = 5, [0][1] = 6, [1][1] = 8};
```

又如

```
int ia [1][1][1] = {[0][0] = {[0] = 1}};
```

其中，最外层的一对花括号与对象 ia 关联；被它直接包围的是指示器列表[0][0]，左边的指示器[0]指示 ia 的子对象 ia[0]，右边的指示器[0]指示 ia[0]的子对象 ia[0][0]；进一步地，对象 ia[0][0]的初始化器被另一对花括号包围，所以，第二对花括号内部的指示器[0]指示 ia[0][0]的子对象 ia[0][0][0]。实际上，上面这个例子与下面的代码等效。

```
int ia [1][1][1] = {[0][0][0] = 1};
```

显然，带有花括号的整个初始化器与对象 ia 关联，在花括号内部，从左往右，第一个指示器[0]指示对象 ia 的子对象 ia[0]，第二个指示器指示对象 ia[0]的子对象 ia[0][0]，第三个指示器[0]指示对象 ia[0][0]的子对象 ia[0][0][0]。

同样地，可以在结构对象的初始化器中使用指示器列表。但与数组不同的是，组成指示器列表的每个指示器由“.”和结构成员的名称组成。例如：

```
struct t {char c; float f;} t = {.f = 0.01f, .c = 'x'};
```

其中，包围初始化器的那对花括号与对象 t 关联；指示器列表.c 指示对象 t 的子对象（成

员）c；指示器列表.f 指示对象 t 的子对象（成员）f。

下面是一个稍微复杂的例子，这是一个数组，数组的成员是结构。

```
struct t
{
    char c;
    float f;
} ta[2] = {[0]={.c = 'x', .f = 0.01f}, [1]={.f = 0.02f, .c = 'b'}};
```

其中，最外层的一对花括号与对象 ta 关联；被它直接包围的指示器列表[0]指示 ta 的子对象 ta[0]；进一步地，对象 ta[0]的初始化器也被一对花括号包围，这对花括号内部的指示器.c 指示对象 ta[0]的子对象（成员）c，指示器.f 指示对象 ta[0]的子对象（成员）f。当然，上述的代码与下面的代码等效。

```
struct t
{
    char c;
    float f;
} ta[2] = {[0].c = 'x', [0].f = 0.01f, [1].f = 0.02f, [1].c = 'b'};
```

其中，初始化器最外层的一对花括号与对象 ta 关联；在初始化器内部，这里要考虑指示器列表[0].c，指示器[0]指示对象 ta 的子对象 ta[0]（这是一个结构类型的数组元素），指示器.c 指示对象 ta[0]的子对象 ta[0].c，这是结构对象的成员。其他指示器列表指示对象的过程与此相同。

联合的情况与结构类似，但联合每次只允许保存一个成员对象的值。不过，使用指示器列表的好处是可以指定初始化哪个成员。下面给出了 3 个例子，请读者结合前面的叙述自行予以分析。

```
union u {char c; double d;} u0 = {.d = 0.5};
union u u1 = {.c = 'x'};
union u ua [2] = {[0] = {.c = 'x'}, [1] = {.d = 0.02f}};
```

在初始化一个数组时，可以在子对象的指示器中使用（#define 定义的）宏。当然，它的本质仅仅是一个简单的宏替换而已。既然可以使用宏替换，使用枚举常量也是很自然的事。例如：

```
# define zero 0
enum {One = 1, Two, Three,};
int a [3] = {[zero] = 1, [One] = 2, [Two] = 3};
```

但是，指示器的“[”和“]”内部必须是常量表达式，不是求值为常量的表达式是不允许的。例如：

```
int x = 0;
int a [3] = {[x] = 1}; //非法
```

再次，如果在初始化器内部没有使用任何指示器列表，那么对于聚合或者联合类型的对象及其子对象来说，它们的初始化是按特定的顺序进行的，将在下面逐一分类叙述。

（1）对于数组类型的对象来说，在使用初始化器的时候，如果没有使用指示器列表，则

初始化的顺序是按子对象的下标顺序进行的。子对象的划分方法取决于初始化器的形式。例如：

```
int a [5] = {1, 2, 3, 4, 5};
```

这是一个简单而熟悉的例子。这里只有一对花括号。这意味着，数组对象 a 的子对象将按下标递增的形式划分为 a[0]、a[1]、a[2]、a[3]和 a[4]。于是，花括号中的 1 分配给子对象 a[0]，2 分配给子对象 a[1]，其他以此类推。

（2）如果初始化器的数量少于元素或者成员的数量，则多余的元素或者成员被初始化为 0，就像处理一个静态存储期的对象一样。例如：

```
int a [5] = {1};
```

其中，数组 a 的第一个元素在初始化之后是 1，即 a[0]的值是 1，其他元素的值在初始化之后是 0。当然，本节的开头已经说过，初始化器的数量不能多于数组元素的数量，这种行为是非法的。

又如：

```
int a [2][3] = {{1, 2, 3}, {4, 5, 6}};
```

其中，a 是一个二维数组，而且初始化器是嵌套的，最外层的左花括号直接嵌套了另一个左花括号。为方便说明，记为$\{_1\{_2$。左花括号$\{_1$和对象 a 关联，左花括号$\{_2$和对象 a 的子对象 a[0]关联。于是，初始化器{1, 2, 3}用于初始化子对象 a[0]的 3 个元素 a[0][0]、a[0][1]和 a[0][2]。

因为数组按下标的顺序初始化，所以 a[0]初始化完成之后开始初始化 a[1]。这时，恰好又有另一对带有花括号的初始化器{4, 5, 6}，于是，初始化器{4, 5, 6}用于初始化子对象 a[1]的 3 个元素 a[1][0]、a[1][1]和 a[1][2]。

再如：

```
int a [2][5] = {{1, 2, 3}};
```

这个例子比较特殊，但是并不复杂。和上一个例子相同，初始化器{1, 2, 3}用于初始化对象 a[0]的 5 个子对象，a[0][0]被初始化为 1，a[0][1]被初始化为 2，a[0][2]被初始化为 3。但是，因为内层花括号内的初始化器在数量上少于被初始化的对象（元素），所以，a[0][3]和 a[0][4]被初始化为 0。

对于整个对象 a 而言，括号内的初始化器在数量上少于剩余的元素，故子对象 a[1]被初始化为 0（换句话说，a[1][0]、a[1][1]、a[1][2]、a[1][3]和 a[1][4]都被初始化为 0）。

（3）字符类型的数组在声明时，可以指定一个字符字面串或者 UTF-8 字面串作为它的初始化器，有或没有花括号括起来均可。例如：

```
char a [16] = "Delphi";
char b [16] = u8"亲爱的";
```

注意：字符字面串和 UTF-8 字面串的类型都是以空字符（参见“空字符”）结尾的数组，所以，用字面串初始化一个数组时，数组应当足够大以容纳暗中附加在字面串尾部的空字符。实际上，这两个例子分别等效于：

```
char a [16] = {"Delphi"};
char b [16] = {u8"亲爱的"};
```

不管采用哪种方式，以上的第一个例子（即数组 a）等效于：

```
char a [16] = {'D', 'e', 'l', 'p', 'h', 'i', '\0'};
```

不管采用哪种形式，数组 a 的元素都是 16 个，而初始化器提供的字符只有 7 个（包括尾部隐含的“\0”），所以，a[7]之后的元素自动初始化为 0。数组 b 的初始化器使用了 UTF-8 字面串，且难以拆分成单个的字符，所以这里不再给出拆分为单个字符的例子。

相反地，如果初始化器提供的字符（包括隐含的空字符）在数量上大于或者等于数组的元素，则它会被截断。例如：

```
char a [5] = {"Delphi"};
```

其中，因为数组只有 5 个元素，所以它等效于：

```
char a [5] = {'D', 'e', 'l', 'p', 'h'};
```

相应地，可以用带有“L”编码前缀的字面串初始化一个 wchar_t 类型的数组；而带有“u”编码前缀的字面串可用于初始化一个 char16_t 类型的数组；char32_t 类型的数组可以用带有“U”编码前缀的字面串来初始化。

注意，字面串前后的花括号是可选的，且这几种编码前缀的字面串都隐含了尾部的空宽字符（参见“空宽字符”），如果数组有足够的空间，则初始化后其尾部会有一个空宽字符；否则字面串会做截断处理。例如：

```
# include <stddef.h>
# include <uchar.h>

void f (void)
{
   wchar_t x [16] = L"有事出班早奏，";
   char16_t y [16] = u"无事卷帘退朝。";
   char32_t z [12] = U"钦此！";
   /* …… */
}
```

（4）对于结构类型的对象来说，如果在初始化器中没有使用指示器列表，则初始化的顺序按各个成员声明的顺序进行。例如：

```
struct t {char c; float f;} t = {'x', 0.01f};
```

其中，初始化器{'x', 0.01f}和对象 t 关联，在花括号内，初始化器'x'和 t 的子对象（成员）c 关联，0.01f 和子对象 f 关联。

又如：

```
struct t {int i; struct {char c; float f;} t;};
struct t t = {8, {'x', 0.01f}};
```

其中，结构 struct t 的内部嵌套了另一个结构，结构对象 t 的子对象分别是 t.i、t.t。因为结构对象的初始化是按成员被声明的顺序进行的，i 是对象 t 的子对象；c 和 f 分别是对象 t.t 的子对象，故 8 用于初始化第一个声明的成员 i；用花括号围住的'x'和 0.01f 分别用于初始化内部结构的成员 c 和 f，按它们声明的先后顺序。

再如：

```
struct t {int i; struct {char c; float f;} t;};
struct t at [2] = {{0, {'a', 0.1f}}, {1, {'b', 0.2f}}};
```

其中，数组 at 的元素类型是结构，数组对象 at 的子对象分别是 at[0]和 at[1]。进一步地，对象 at[0]的子对象是 at[0].i、at[0].t；对象 at[1]的子对象是 at[1].i、at[1].t。

初始化器最外层的一对花括号与对象 at 关联；初始化器{0, {'a', 0.1f}}和对象 at[0]关联；初始化器{1, {'b', 0.2f}}与对象 at[1]关联；初始化器 0 与对象 at[0].i 关联；初始化器{'a', 0.1f}与对象 at[0].t 关联；初始化器 1 与对象 at[1].i 关联；初始化器{'b', 0.2f}与对象 at[1].t 关联。

初始化器'a'与对象 at[0].t.c 关联；初始化器 0.1f 与对象 at[0].t.f 关联；初始化器'b'与对象 at[1].t.c 关联；初始化器 0.2f 与对象 at[1].t.f 关联。

（5）对于联合类型的对象来说，如果在初始化器中没有使用指示器列表，则初始化的是联合对象的第一个命名（有名字的）成员。例如：

```
union u {char c; float f;} u = {22};
union v {char : 6; float f;} v = {22};
```

其中，对象 u 的初始化器 22 仅用于初始化它的子对象 u.c；对象 v 的初始化器仅用于初始化它的子对象 f，但要做浮点到整数类型的转换。

在初始化聚合或者联合类型的对象时，花括号和指示器列表可以组合使用。如果一个指示器中包含了数组下标，则后面的初始化过程将延续此下标继续进行，直至遇到另一个指示器或者指示器列表。例如：

```
int a [7] = {3, 9, [4] = 3, 6};
```

按前面所述，初始化器 3 和 4 分别用于初始化数组 a 的子对象 a[0]和 a[1]。然后，这里出现了一个指示器[4]，将对象 a 的子对象 a[4]初始化为 3，由于后面的初始化器是 6，并非以指示器的形式出现的，故 6 用于初始化对象 a 的子对象 a[5]。除了上面这 4 个子对象外，数组 a 的其他子对象都被初始化为 0。

又如：

```
int a [2][3] = {{1, 2}, [1][1] = 7, 6};
```

其中，初始化器 1 和 2 分别用于初始化对象 a[0]的子对象 a[0][0]和 a[0][1]。然后，其遇到了一个指示器列表，于是初始化器 7 用于初始化数组 a 的子对象 a[1][1]。进一步地，后面的初始化器 6 自然用于初始化数组 a 的子对象 a[1][2]。除了这些子对象外，数组 a 的其他子对象统统被初始化为 0。这个例子实际上等效于：

```
int a [2][3] = {{1, 2}, {[1] = 7, 6}};
```

因为初始化器{[1] = 7, 6}与数组 a 的子对象 a[1]关联，所以，初始化器[1]=7 等效于 a[1][1]=7，初始化器 6 与对象 a[1][2]相关联。

再如：

```
struct t {int i; float f;};
struct t at [5] = {{3, 0.1f}, [2] = {6, 0.8f}, {9, 0.2f}};
```

这里，数组 at 的元素类型是结构，它的子对象分别是 at[0]、at[1]、at[2]、at[3]和 at[4]。于是，根据前面的知识，最外层的花括号与对象 at 关联，内层的花括号分别与各个数组元素（结构）关联。初始化器 3 用于初始化对象 at[0]的子对象 at[0].i；初始化器 0.1f 用于初始化对象 at[0]的子对象 at[0].f。接着，出现了指示器[2]，它指示数组 at 的子对象 at[2]。因此，初始化器 6 用于初始化对象 at[2]的子对象 at[2].i；初始化器 0.8f 用于初始化对象 at[2]的子对象 at[2].f。数组下标继续增加，于是，初始化器{9, 0.2f}用于初始化 at 的下一个子对象 at[3]。进一

步地，初始化器 9 用于初始化对象 at[3]的子对象 at[3].i；初始化器 0.2f 用于初始化对象 at[3]的子对象 at[3].f。除了上述这些子对象外，数组 at 的其他子对象一律被初始化为 0。

如果一个数组的声明仅有初始化器而没有指定数组的大小，则数组的大小由初始化器决定。具体来说，在所有被初始化的元素中，有一个下标最大，数组的大小可以通过这个下标值来确定。在初始化器结束的位置之后，该数组即成为完整类型。例如：

```
int a [] = {0, 5, 8,}, x = sizeof a;
int b [][2] = {{1}, {2}, {3},}
int c [][3] = {[9] = {0}, [5] = {8}, [3] = {7},};
double d [] = {[1999] = 0.0};
```

数组 a、b 和 c 在声明时都没有指定大小。但是，在 a 的初始化器中给出了 3 个常量表达式，数组下标依初始化器的序列增长，最大下标是 2，所以数组 a 有 3 个元素，元素的类型是 int。在声明对象 x 的时候，a 已经成为完整类型，所以可以用 sizeof 取得其长度。但是，下面的例子不正确。

```
int a [] = {0, 5, sizeof a,};
```

数组 b：它是数组的数组，其元素类型是 int[2]。整个数组对象的初始化器又由另外 3 个初始化器组成，数组 b 的下标依初始化器的序列增长，最大下标是 2，所以，数组 b 具有 3 个元素，元素类型是 int [2]。

数组 c：它也是数组的数组，其元素类型是 int[3]。该数组的初始化器由指示器组成，指示的最大下标是 9，所以它具有 10 个元素，其完整类型是 int[10][3]。

数组 d：它的初始化器由指示器组成，指示了一个下标 1999，所以这个数组有 2000 个元素。最后一个元素是明确地初始化的，其余元素都被初始化为 0。

变长数组的声明中不能有初始化器。

下例中，可以为对象 x 指定初始化器；数组 y 的大小由它的初始化器决定，初始化器提供了 3 个值，所以数组的大小是 3 个元素；z 是变长数组，按规定，它不能有初始化器。

```
int x = 2;                          //正确。x 的类型是完整类型
float y [] = {1.0, 3.0, 5.0};       //正确。y 的最终类型是完整类型
double z [x] = {1.0, 2.0};          //错误。z 的类型是 VLA
```

具有静态存储期的对象在程序启动时创建和初始化，具有线程存储期的对象在线程启动时创建和初始化。因此，它们的初始值不能依赖于那些只有当程序运行到特定的阶段和位置时才能确定的值。鉴于此，这两种存储期的对象在声明时，它们的初始化器只能由常量表达式和字面串组成。

下例中，对象 g、p、q、s、ps、t、i、os 和 m 都具有静态存储期，所以要求它们的初始化器必须仅包含常量表达式。对象 g 的初始化器是 0，这是一个常量表达式；对象 p 的初始化器是&g，因为对象 g 具有静态存储期，故&g 是一个常量指针；对象 q 的初始化器是 0，这是一个空指针常量；对象 s 的初始化器是由两个常量表达式 0 和'x'组成的，这是合法的；对象 ps 的初始化器是&s，因对象 s 是具有静态存储期的对象，故&s 是一个常量指针；对象 t 和对象 s 的情况类似，毕竟这里的 sizeof 和_Alignof 表达式都是常量表达式（求值为常量）；对象 i 具有静态存储期，但它的初始化器是 g，表达式 g 的值需要在程序运行时求值才能得到，不是常量表达式，故为非法；对象 os 具有静态存储期，它的初始化器是表达式 g 和'x'，问题在于

g 并不是常量表达式，因此为非法。

在函数 f 的内部，对象 m 是用存储类指定符 static 声明的，故具有静态存储期，但它的初始化器是 n，表达式 n 的值需要在程序运行时求值才能得到，故为非法；对象 at 具有自动存储期，对它的初始化器没有特别的要求。

```
# include <stddef.h>

int g = 0, * p = & g, * q = 0;
struct s {int i; char c;} s = {0, 'x'}, * ps = & s;
struct t {size_t i, j;} t = {sizeof (char), _Alignof (char)};
long long int i = g;          //非法
struct s os = {g, 'x'};       //非法

void f (int n)
{
   static int m = n;          //非法
   struct t at = {n, n};
}
```

同样的原因，对于下面的例子，D1 是非法的，D2 是合法的：

```
void f (void)
{
    int x = 0;
    static int * px = & x;      //D1
    static int i = 5;
    static int * pi = & i;      //D2
}
```

一个在块内声明的、指示对象的标识符，如果它是无链接的，则意味着它指示一个独立存在的实体，因此，在其声明中可以有初始化器（变长数组除外）；如果这个标识符具有外部链接或者内部链接（使用了存储类指定符 extern），则意味着它指示一个在别处定义的实体，所以在其声明中不能有初始化器。下面的示例很清楚地展示这种区别。

```
int i = 0;                      //允许
static int j = 1;               //允许

int f (void)
{
   double d = 0.05;             //允许：d 是无链接的
   extern int i = 0;            //非法：具有外部链接且有初始化器
   extern int j = 0;            //非法：具有内部链接且有初始化器
   /* ..... */
}
```

5.10.1　初始化器

初始化器（Initializer）是对象声明或者复合字面值的一部分，用于为对象提供初始值。当程序的执行到达该声明或者复合字面值的位置时，用初始化器提供的值初始化对象的存储值。

初始化器的语法形式如图 5-8 所示，最简单的初始化器可能是一个常量或者表达式，比如下例中的 0 和&x，已经用粗体标出：

```
int x = 0, * px = & x;
```

对于聚合或者联合类型的对象，或者复合字面值，它们的初始化器在形式上必须是用花括号围起来的列表，称为初始化器列表。如果是用字面串来初始化一个字符类型的数组，则花括号可以省略。下例中的初始化器都已经用粗体标出。

```
char a [] = {"Warcraft II"};
char b [] = "Command & Conquer";
char c [] = {'D', 'i', 'a', 'b', 'l', 'o', '\0'};

struct t {char a [100]; int i;};
struct t t = {"Delta Force", sizeof t.a};
```

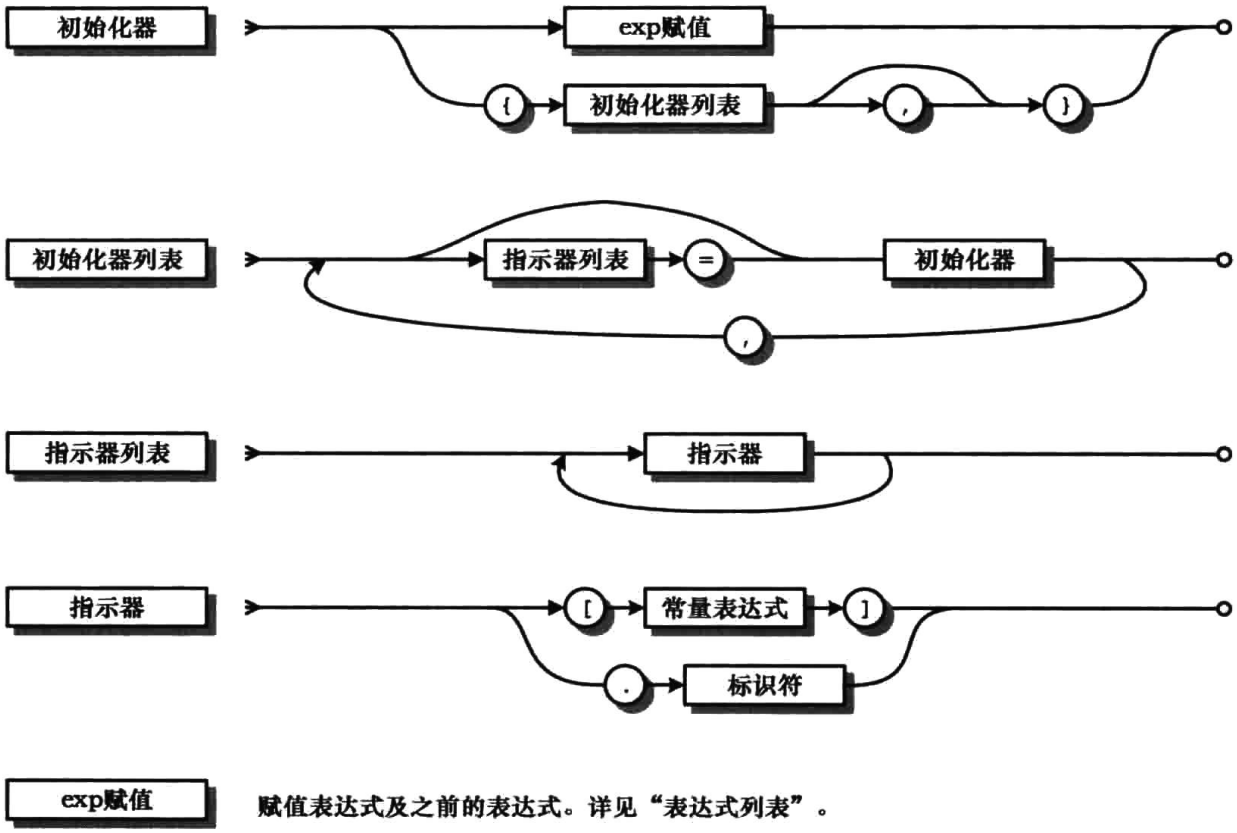


图 5-8　初始化器的语法

结合初始化器的语法来看，显然，初始化器是递归构造的。因此，上例中的有些初始化器是由更小的初始化器组合而成。比如初始化器{'D', 'i', 'a', 'b', 'l', 'o', '\0'}，它是由初始化器'D'、'i'、'a'、'b'、'l'、'o'和'\0'组成；初始化器{"Delta Force", sizeof t.a}由初始化器"Delta Force"和

sizeof t.a 组成。

新的标准允许在初始化器中使用指示器。比如下面的例子，字体加粗的部分都是指示器。[0]用于指定数组 d 的第 1 个元素，[10]用于指定数组 d 的第 11 个元素；.a 用指定结构对象 t 的成员 a，.i 用于指定结构对象 t 的成员 i：

```
double d [] = {[0] = 0.01, [1] = 0.02, [10] = 0.03};

struct t {char a [100]; int i;};
struct t t = {.a = "Delta Force", .i = sizeof t.a};
```

指示器可以组合使用，这样就形成了指示器列表。有些类型的派生非常复杂，指示器列表可以方便地应付（指定）这些复杂类型的元素或者成员，从而使初始化的过程更加灵活。下例中，字体加粗的部分是指示器列表。[0][0]用于指定数组 c 的元素 c[0][0]，[1][0]用于指定数组 c 的元素 c[1][0]；.a[0]用于指定结构成员 a（它是一个数组）的第 1 个元素，即 t.a[0]，.a[1]用于指定结构成员 a 的第 2 个元素，即 t.a[1]；单个指示器是指示器列表的简单（特殊形式），比如.i。

```
char c [2][2] = {[0][0] = 'x', [1][0] = 'y'};

struct t {char a [100]; int i;};
struct t t = {. a [0] = 'x', . a [1] = '\0', .i = sizeof t.a};
```

下面是一个综合性的示例，对象 a 的初始化器不需要非得是用花括号包围的（具体的说明参见“初始化”），但对象 b 的初始化器必须这么做。

```
char a [] = "The sun came out.\n";
int b [] = {1, 2, 3, 4, 5};

struct t
{
    char c;
    struct
    {
        int i;
        struct
        {
            float f;
        };
    };
    double d;
};

struct t x = {'x', {0, {1.0f}}, 2.0};
struct t y = {.c = 'x', .i = 0, .f = 1.0f, .d = 2.0};
```

```
struct t z = {'x', 0, 1.0f, 2.0};
```

同样地，对象 x 的初始化器是正确的，它是一个结构，且内部嵌套了两个匿名结构（参见“匿名结构”），按要求都必须使用带有花括号的初始化器以指示层次关系。但是，对象 y 的初始化器使用了指示器，且匿名结构的成员被看成是包含它的那个结构的成员，所以这里不需要嵌套的花括号。最后，对象 z 的初始化器未采用标准的做法，这是不合法的，C 实现可能在转换期间产生诊断信息。

初始化器所提供的初始值仅针对它所关联的对象及子对象，额外的初始化器既不会影响到其他对象，也是非法的。在下例中，声明符 x 的初始化器没有问题；声明符 y 的初始化器虽然比较奇怪，但也合法；但是数组 z 只有两个元素，它的初始化器却提供了 3 个元素的值（程序员的意思可能是用多余的初始化器初始化数组 z 之后的内存单元），这是不合法的。

```
int x = 0, y = {0,}, z [2] = {1, 2, 3,};
```

标量类型（即算术类型和指针类型）的对象在初始化器中只需要一个单一的值。因此，这种类型的对象在声明时，初始化器外围的花括号可以省略。典型的就是上例的对象 x，它的初始化器是 0，而不是{0}。

不是所有的声明都能带有初始化器，函数类型不可以，有些对象类型也不可以。具体地说，只有在一个对象被声明为以下类型时，才能带有初始化器。

（1）未指定大小的数组；

（2）完整的对象类型，变长数组除外。

第二种情况也包括数组、结构、联合及枚举类型的对象。同时，如果是数组对象的声明，则要么不指定大小，要么必须指定常量大小。

以上的限制也意味着，结构内部的弹性数组成员也是不能初始化的。给定声明

```
struct t {int data; float * pf; char a [];};
```

则以下声明中的初始化器是非法的，原因在于结构的成员 a 是弹性数组成员，不允许被初始化：

```
struct t t1 = {1, 0, {"Invalid. \n"}};
```

相反，以下声明中的初始化器则是合法的。顺便说一句，在这两个例子中，结构的成员 pf 都是指针，而指针都是完整的对象类型。

```
struct t t2 = {2, 0};
```

非复合字面值组成部分的初始化器是全表达式，原则上，每个全表达式和下一个全表达式之间存在一个序列点。这就是说，下面的代码是没有问题的，良好定义的，控制到达 D 处时，对象 x 的值为 0，y 的值为 1，z 的值为 2。

```
int x = 0, y = ++ x, z = ++ x; //D
```

但这里有个例外，对于初始化器列表中的每个初始化器或者独立的子表达式，它们之间的求值是不确定顺序的，但并非无序。因此，副作用的顺序也是未指定的，如果有的话。

下例中，A1、A2 和 A3 处的++x 到底哪一个先求值（值计算和副作用），是不确定的，唯一肯定的是，它们不会交叉进行，所以数组 a 的 3 个元素逃不开这 3 个值：1、2 和 3，至于它们值和元素之间的对应关系则也是不确定的。

B1、B2 和 B3 的情况与此类似。另外可以确定的是，在 A4 处，x 的值是 3，在 B4 处，x

的值是6。

```
int x = 0;

int a [3] = {++ x,          //A1
             ++ x,          //A2
             ++ x,          //A3
            };
                            //A4
int b [3] = {
             [2] = ++ x,    //B1
             [1] = ++ x,    //B2
             [0] = ++ x,    //B3
            };
                            //B4
```

复合字面值（参见“复合字面值”）也是带有初始化器的表达式，所以适用于初始化和初始化器的规则也适用于复合字面值的初始化器。但是，复合字面值的初始化器不是全表达式。尽管如此，如果复合字面值不违反求值的规则，则它也不会有未定义的行为，例如：

```
int x;
int * p = (int []) {x = 0, x = 1, x = 2};
```

尽管我们无法确切地知道这三个赋值表达式的求值顺序，但它们毕竟不会交叉执行。相比之下，这样的代码就成问题了：

```
int x;
int * p = (int [])
                {x = 0, x = 1, [9] = 9}  //A
                +
                x;                       //B
```

以上，A 处和 B 处的求值是无序的，所以这里有未定义的行为。无序的求值如果作用于同一个标量对象必将导致未定义的行为，而明确的顺序（序列点）或者不确定的顺序则不会，这些示例显示了它们之间的微妙之处。

5.11 类型名

类型名（type name）就是“类型的名称”。很多时候需要指定一个类型的名称，如转型表达式、函数原型、复合字面值，等等（见图 5-9）。

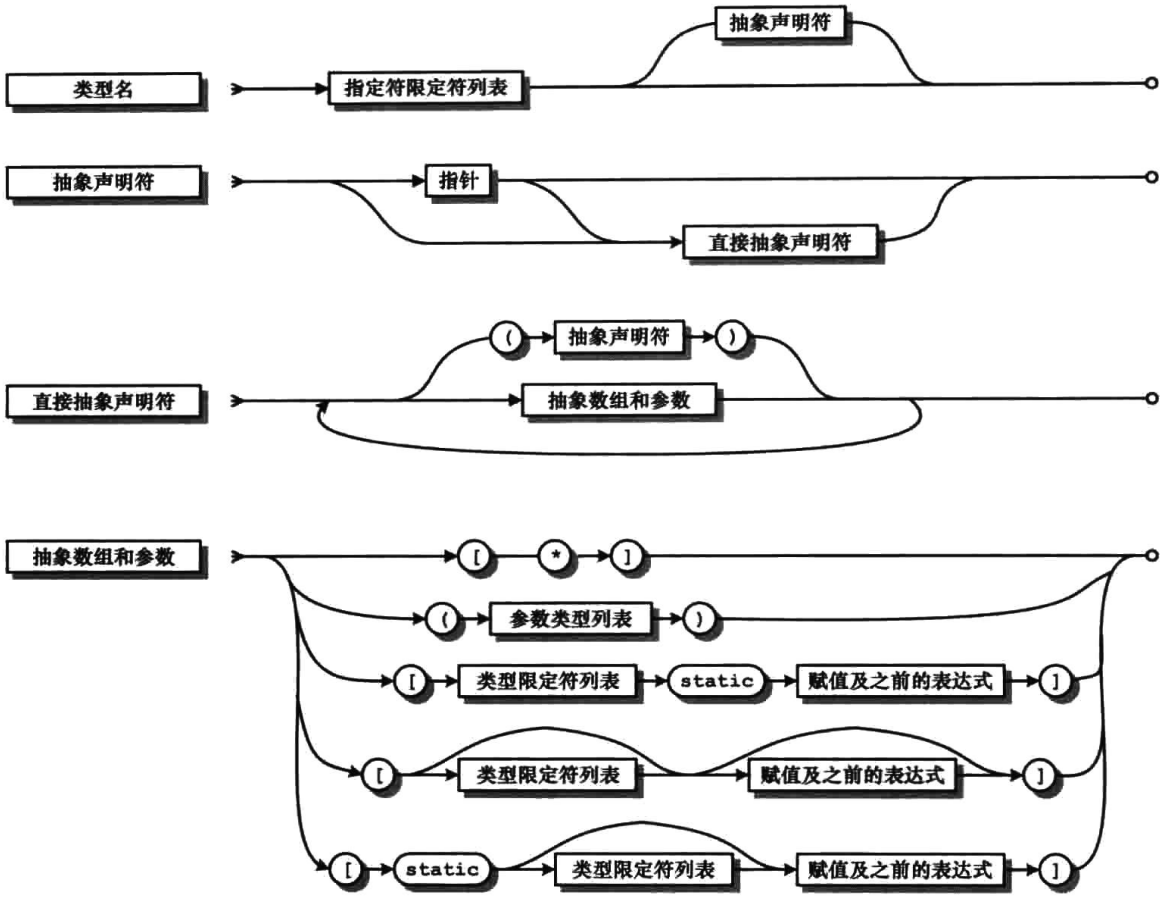


图 5-9 类型名的语法

从语法上来说，一个类型名就是省略了标识符（被声明的实体）的声明（对象声明或者函数声明）。

下例中，x 和 y 的类型是 int，“int”就是类型名；px 的类型是 const int *，py 的类型是 const int * restrict，这也是两个类型名；f 的类型是 float (float,float)，这也是类型名；dx 的类型是 double [3]，dy 的类型是 double (*) [3]，dz 的类型是 double * [3]，它们依然是类型名；g 的类型是 float (* (float, float)) [3]，它依然是类型名。

```
int x, y;
const int * px, * restrict py;
float f (float, float);
double dx [3], (* dy) [3], * dz [3];
float (* g (float, float)) [3];
```

第6章 表 达 式

每个表达式（expression）总是用于“表达”些什么。从形式上看，一个指示对象或者函数的标识符，甚至是一个常量，都能形成一个表达式。更复杂地，一些运算符和它们的操作数也可以连接成一个序列，这也是表达式。表达式可以出现在声明和各种语句里。下面是一些表达式的例子，已经在代码中用粗体标出。

```
int x = 0, f (void);
1972;
f;
x ++;
f ();
x = sizeof (int);
```

C 中的表达式多种多样，取决于它们的形式和出现的位置，有的表达式指示对象，有的表达式指示函数，有的表达式可以计算出一个值，有的表达式还可能具有副作用。甚至，有些表达式可以同时做多种事情，也就是说，上述这些功能可以兼而有之。

比如，上例中的表达式 0 是常量表达式，它计算出一个零值，然后赋给对象 x；表达式 1972 也是常量表达式，计算出一个常量值，然后被丢弃；表达式 f 指示一个函数，但它在这里同样没有用处。

表达式 x++包含了一个子表达式 x，表达式 x 指示一个对象。表达式 x++会计算出一个值（对象 x 的原始内容），同时会发起一个副作用（将加一后的值写回对象 x）。

表达式 f()包含了一个子表达式 f，表达式 f 指示一个函数。表达式 f()会计算出一个值（函数调用的返回值），同时，函数调用可能会有副作用。

表达式 x=sizeof(int)更复杂，它包含了两个子表达式 x 及 sizeof(int)。表达式 x 指示一个对象；表达式 sizeof(int)计算出一个值。最后，表达式 x = sizeof (int)会计算出一个值（对象 x 被赋值后的新值），同时会发起一个副作用（对象 x 的存储值会被修改）。

C 语言表达式的大体分类可参见词条“表达式列表”；由于表达式的类型和运算符密切相关，在本书中，表达式和运算符的归类和命名采用相同的法则。例如，组成后缀表达式语法形式的运算符称为后缀运算符，其他以此类推。

6.1 表达式列表

C 中的表达式可以大体上划分为：

（1）基本表达式

（2）后缀表达式

（3）一元表达式
（4）转型表达式
（5）乘性表达式
（6）加性表达式
（7）移位表达式
（8）关系表达式
（9）等性表达式
（10）逐位与表达式
（11）逐位异或表达式
（12）逐位或表达式
（13）逻辑与表达式
（14）逻辑或表达式
（15）条件表达式
（16）赋值表达式
（17）逗号表达式

各表达式的详细叙述可参见其各自的词条。

注意，这个顺序是有意安排的，在本书的其他部分将频繁参照此表。如果提及“后缀表达式及之前的表达式”，则指后缀表达式和基本表达式；如果提及“一元表达式及之前的表达式”，则指一元表达式、后缀表达式和基本表达式，其他措辞所涉及的表达式以此类推。

6.2 全表达式

总体上，如果一个表达式在形式上是独立的，即，不是其他表达式的组成部分，也不是一个声明符（参见“声明符”）的组成部分，那么它就是一个全表达式（full expression）。下面是全表达式的例子。

```
int a [sizeof (int) * 8] = {0}, * p = & (int) {1};
/* ..... */
if (p) * a = (* p) ++;
a [1] = ++ a [0], a [2] = ++ a [1];
```

第一，如果一个表达式语句不是空语句，那么，去掉该语句后面的分号之后，剩下的部分是一个全表达式。因此，上例中的表达式

```
* a = (* p) ++
a [1] = ++ a [0], a [2] = ++ a [1]
```

都是全表达式，它们都是独立的，不是其他表达式的组成部分。

第二，if、switch、do 或者 while 语句的控制表达式是全表达式。上例中，if (p)中的 p 是一个全表达式。

第三，在 for 语句的语法组成中，括号中的第一部分可以是声明，也可以是表达式，如果是表达式，则它是全表达式；第二部分如果未被省略，它也是一个全表达式；第三部分如果

也未被省略，它同样是一个全表达式。

第四，如果 return 语句是由关键字 return 和一个表达式组成的，则此表达式也是一个全表达式。

第五，用于组成声明符的表达式容易被误认为是全表达式，但千万不要弄错了，它不是全表达式。上例中的声明符 a [sizeof (int) * 8]包含了一个表达式 sizeof (int) * 8，但这个表达式不是全表达式。

第六，一个初始化器是否为全表达式，要看它是不是复合字面值的组成部分。如果不是，则它就是一个全表达式。上例中，{0}是一个全表达式，& (int) {1}也是全表达式，但{1}则不是全表达式，因为它是复合字面值& (int) {1}的组成部分。注意，复合字面值本身也是一个表达式。

每个全表达式和位于它后面的那个表达式之间存在一个序列点，它们之间有确定的前序和后序关系。具体内容可参见“序列点”、“前序”和“后序”。

6.3 函数指示符

在 C 中，每个表达式都有它自己的类型，有些表达式的类型是函数，或者说指示函数。指示函数的表达式称为函数指示符（function designator）。给定以下程序片断：

```
int f (void);
typedef void F (int, int);
struct t {F * f;} t = {/* ..... */};

f ();                   //S1
(* t.f) (0, 1);        //S2
```

其中，S1 中的表达式 f 和 S2 中的表达式*t.f 都是函数指示符。子表达式 f 在这里不再解释，它是典型的函数指示符；结构 t 的成员 f 是指向函数的指针，解除引用后是函数指示符。如果不是为了用于说明函数指示符，则 S2 可以简单地写成：

```
t.f (0, 1);
```

这是因为函数调用表达式需要一个指向函数的指针，如果是函数指示符，则需要转换为指针类型。当然，这种转换是自动进行的。参见“函数指示符-指针转换”。

6.4 左值

有些表达式潜在地指示对象，可以通过该表达式访问它指示的对象。这样的表达式被称之为左值（lvalue）。因为不存在 void 类型的对象，所以不存在 void 类型的左值。

“左值”这个名称既暗示它通常位于赋值运算符=的左边（left），又意味着它是对象的定位器（locator）。注意，不要说“左值表达式”，因为左值本身就是一个表达式。在下面的例子中，表达式 a 和 b 都是左值，但表达式 3、b+3 和 a=b+3 不是左值。

```
a = b + 3;
```

“潜在地”的意思是在形式上或者理论上，左值指示一个对象。但实际上，它所指示的对象可能已经不复存在，或者本身就不是一个有效的对象。在这种情况下，对它求值的行为是未定义的。例如：

```
# include <stdlib.h>

void f (void)
{
    int * px = 0, * py = malloc (sizeof (int));
    * px = 10010;          //S1
    /* …… */
    free (py);
    * py = 10086;          //S2
    /* …… */
}
```

其中，在语句 S1 中，左值*px 并未指示一个有效的对象；在语句 S2 中，左值*py 也不再指示一个有效的对象。因此，*px=10010 和*py=10086 这两个表达式的求值会产生不可预料的后果。

尽管经常提到“对象的类型”，但对象其实没有类型，因为左值指示对象，所以，所谓“对象的类型”，实际上是指示该对象的左值的类型，且要求是一个有效类型（参见“有效类型”）。下面是一些左值的例子。

```
int i, a [3];
struct s {float f;} s;
char c;
c = "%f\n" [0];      //c、"%f\n"（字面串）和"%f\n"[0]都是左值
i = 0;               //表达式 i 是左值
a [0] = i;           //表达式 a、a[0]和 i 都是左值
s.f = 0.0;           //表达式 s 和 s.i 都是左值
```

特别地，在函数定义中，函数参数列表中的每一个标识符都是左值，都指示一个具有自动存储期的对象。具体情况可参见“函数定义”和“形参”。下例中，函数 f 的定义中声明了两个标识符 x 和 ps，在函数被调用时，将创建两个对象，并接受调用者传递的值（对象 x 接受整数值，对象 ps 接受指针值）。此后，就可以在函数内部访问这两个对象。

```
void f (int x, char * ps)
{
    printf ("%d…%s.\n", x, ps);
}
```

6.4.1 可修改的左值

通过左值可以修改对象的存储值，但这并不绝对。如果一个左值的类型是以下所述中的

任何一个，则不能用于修改它指示的对象。

（1）数组类型。

（2）不完整类型。

（3）const 限定的类型。

以上第 3 种情形同样适用于结构或者联合对象，以及它们的成员对象（递归地，包括成员的成员）。

被排除在以上情况之外的左值，可用于修改它们所指示的对象，这样的左值称为可修改的左值（modifiable lvalue）。“可修改的左值”这一术语具有误导性，但这也是没有办法的事。以下面的代码为例，这里存在一个数组对象。要想修改数组的值，必须借助于左值，也就是 S1 和 S2 中的 a，而无法直接修改。因此，从形式上来看，修改的是左值。不过我们都很清楚，虽然修改的是左值，但实际上修改的是左值指示的对象。在这里，修改了 a，也就是修改了 a 所指示的对象。

```
int a [1];
a = 0;      //S1
a [0] = 0;  //S2
```

正是因为如此，上例中，尽管数组是可修改的，但不能通过 S1 中的左值 a 修改。但是，S1 从形式上来看似乎是用 0 来修改左值 a，所以我们说 a 不是可修改的左值；同理，在 S2 中，a 是左值，它与数组下标运算符组成表达式 a[0]，这个表达式也是左值。尽管是修改 a[0]所指示的对象（数组对象的子对象），但 S2 从形式上来看似乎是用 0 来修改左值 a[0]，而且这里的修改是允许的，所以我们说 a[0]]是可修改的左值。

再来看另一个例子，给定下述代码

```
const int x = 0;

struct s
{
   const char c;
   struct t
   {
      const int i;
      float f;
   } t;
   double d;
} s1, s2;

void * pv = & x;
/* ..... */
x ++;              //E1
s1 = s2;           //E2
s1.c = 'x';        //E3
```

```
s1.t.i = 0;          //E4
* pv --;             //E5
```

则 E1 中的 x、E2 中的 s1、E3 中的 s1.c、E4 中的 s1.t.i 和 E5 中的*pv 都是不可修改的左值。x 是左值，但它的类型是 const 限定的；s1 也是左值，但它的某些成员的类型是 const 限定的；s1.c 也是左值，但它指示的结构成员的类型是 const 限定的；s1.t.i 和 s1.c 类似；表达式 *pv 同样是左值，但该表达式属于不完整的对象类型。

6.5 值计算

除非是 void 类型的表达式，每个表达式都有一个特定的值。计算表达式的值，又称值计算（value computation），这涉及多个方面，如确定常量的值、读取对象的存储值、得到函数调用的返回值，等等。因为左值是潜在地指示对象的表达式，而为了计算它的值，还需要确定对象的实体（如解引用一个指针），值计算通常也包括这个过程。

在本书中，一旦提到“计算……的值”，或者“得到……的值”，都是指值计算。给定程序片断：

```
int x, y, f (int);
/* …… */
f (x + y);
```

则是先分别计算表达式 x 和 y 的值（谁先计算都有可能），再计算表达式 x+y 的值，并将这个值作为函数的参数传递。最后，函数的返回值就是整个全表达式 f(x+y)的值。

常量是最简单的表达式，它的值通常在程序转换期间计算。void 表达式计算一个不存在的值，具体可参见词条“void 表达式”。

对于含有运算符的表达式，表达式的值被认为运算符的值或者结果。这里有一个简单的例子，表达式 a + b，假定 a 和 b 已经被声明为指示一个对象（即，a 和 b 是左值），那么可以肯定的是，对 a 和 b 的值计算一定会在它们开始相加之前完成。

这就是说，在计算表达式的值时，总是先计算（运算符的）操作数的值，然后再计算运算符的值，操作数的值计算先序于运算符的值计算。对于表达式 a * b + c / d 来说，要想得到运算符*的结果（记为 V_*），必须先计算操作数 a 和 b 的值（这两个操作数的计算顺序不能确定，但无关紧要）；要想得到运算符/的结果（记为 $V_/$），也必须先计算操作数 c 和 d 的值（这两个操作数的计算顺序同样不能确定，但无关紧要）；要想最终得到运算符+的结果（即整个表达式的值），则必须先计算操作数 a*b 和 c*d 的值，即先计算 V_*和 $V_/$（这两个操作数的计算顺序同样不能确定，但不影响结果的正确性）。

值计算是表达式要完成的主要工作。不要被表达式的形式所迷惑，例如

```
x = 5
```

在初学者眼里，他们只看到该表达式做了一件事：给标识符 x 所指示的对象赋值。实际上，改变对象 x 的存储值，这只是该表达式的副作用（参见“副作用”），该表达式还要计算出一个值（表达式的值），它是对象 x 赋值之后的值。不了解这一点，也就无法理解象

```
x = y = 5
```

这样的表达式是怎么工作的。

6.6 void 表达式

一种比较特殊的表达式是 void 类型的表达式，简称 void 表达式。void 表达式不能计算出一个具体的值，因此被认为计算的是一个不存在的值，且 void 表达式的值不能在任何地方以任何方式使用。void 表达式存在的唯一价值是得到它的副作用，而不是它的值。当然，如果连副作用都没有，则它毫无用处。

一个 void 表达式的典型例子是返回 void 类型的函数。

```
void fretv (int);
fretv (0);
float f = fretv (3) + 2.0; //非法
```

在以上最后一行，表达式 fretv (3) + 2.0 是非法的，原因在于 fretv(3)是 void 表达式，不可能使用一个 void 表达式的值。

使用转型表达式，可以将其他表达式的值强制转换为 void 类型，这意味着明确地丢弃值或者指示符，如下例所示。

```
int i = 0, * pi = & i, f (int);
(void) i ++;     //丢弃表达式的值，但该表达式具有副作用
(void) * pi;     //表达式*pi 是一个对象的指示符，但被丢弃
(void) f;        //表达式 f 是一个函数的指示符，但被丢弃
(void) f (0);    //表达式 f(0)是一个函数调用表达式，返回值被丢弃
```

注意，以上 4 条表达式语句中的“(void)”是不必要的。表达式语句存在的意义是作为一个 void 表达式得到它的副作用，表达式的值总是被丢弃，而不管它是否显式地转换为 void 类型。当然，到目前为止，很多 C 实现在表达式没有副作用时依然会产生诊断信息（警告）。事实上，C 实现可以忽略中间两条表达式语句，因为这两个表达式没有任何副作用，它们的值也没有任何用处。更多的内容可参见“表达式语句”。

除非将一个 void 表达式的值转换为 void 类型，否则针对 void 表达式的其他任何显式或隐式转换都是不允许的。例如：

```
void f (int);
int x = (int) f (0);        //E1: 非法
(void) f (0);               //E2: 允许
(void) (void) 3;            //E3: 允许
(void) void;                //E4: 非法
```

其中，E1 中的表达式(int) f (0)是非法的，表达式 f(0)是 void 表达式，返回一个不存在的值，不允许将 void 类型的值转换为 int 类型；E2 是允许的，符合以上所述的原则；E3 中的表达式也是允许的，表达式(void)3 的类型是 void，而(void)(void)3 是将 void 表达式的值转换为 void 类型；E4 中的表达式中是非法的，右边的 void 是关键字，不是表达式。

6.7 副作用

在程序执行期间，它可能会做一些改变执行环境的事情，这些改变称为副作用（side effects）。典型地，这些改变执行环境的事情包括：修改一个对象的存储值；访问（读或写）一个 volatile 限定的对象；修改一个文件的内容；调用一个函数，且该函数在执行过程中做了上述操作。

访问一个 volatile 限定的对象时，因为该对象是用类型限定符 volatile 声明的，故意味着不但当前程序会修改它的值，其他程序（包括硬件）都有可能改变它的存储值。例如，一个被系统定时器硬件不断刷新的存储器单元。给定以下声明：

```
int a, b = 0;
volatile int * pc = (volatile int *) 0xFFFFFACE;
```

则

```
a = b ++ - * pc
```

是有副作用的表达式，而且有 3 个副作用，分别是读易变对象*pc（指针 pc 所指向的对象是易变的）、修改对象 a 和 b 的存储值。

6.8 序列点

给定两个表达式 A 和 B，如果 A 的副作用和值计算发生在 B 的副作用和值计算之前，则称在 A 和 B 的求值之间存在一个序列点（sequence point）。

很显然，以这个点（序列点）为分水岭和界线，之前的表达式已经完成了值计算和副作用，即已经完成求值，但序列点之后的表达式还没有开始进行。来看下面的例子，这是两条表达式语句，按照定义，在表达式 a=b++和 f(a)之间存在一个序列点：

```
a = b ++;
f (a);
```

下面是另一个序列点的例子。按照规定，在 if 语句的控制表达式 x>=0 和表达式 i++之间存在一个序列点：

```
if (x >= 0) i ++;
```

对序列点的描述散见于本书的各个部分，与本书内容相关的序列点存在于：函数调用时，其实际参数求值和调用行为实际发生之间；运算符&&、||与,的第一个操作数和第二个操作数之间；条件运算符?:的第一个操作数和第二个或者第三个操作数之间（取决于哪一个操作数会求值）；每一个全声明符的末尾；每个全表达式和位于它后面的全表达式之间。

正是因为如此，在下例中，A 处的全表达式 x++和全表达式 y++之间有一个序列点；B 处的全表达式 y 和全表达式++y 之间有一个序列点；D 处的表达式 x==3 和表达式 y（或者++y，取决于表达式 x==3 的结果）之间存在着一个序列点；唯一例外的是 C 处，它和 B 还不一样，尽管 B 处的 y 和++y 是初始化器（全表达式），但它们不是同一个初始化器列表的组成部分。而 C 处的 3 个++x 则不然，在这种情况下，这 3 个初始化器的求值是不确定顺序的（但不是

无序。更详细的讨论，参见“初始化器”），这里没有未定义的行为，这 3 个数组元素的值有一个为 1，有一个为 2，还有一个为 3。

```
int f (int x, int y)
{
    if (x ++) y ++;                          //A
    int m = y, n = ++ y;                     //B
    x = 0;
    int a [3] = {++ x, ++ x, ++ x};          //C
    /* ..... */
    return x == 3 ? y : ++ y;                //D
}
```

其他序列点都和库函数有关，可参阅标准文档以获得最完整和最权威的描述。

6.8.1 前序

给定单线程上的任意两个求值 A 和 B，如果总是能够确定 A 完成时，B 还尚未开始，即，A 在 B 之前进行，则称 A 是前序（sequenced before）于 B 的。

因为表达式的求值包括值计算和副作用，因此前序意味着一个表达式的值计算和副作用已经完成，而另一个表达式的值计算和副作用还没有开始，或者说前者的求值在后者的求值开始之前已经完成。

下例中，表达式 x--前序于表达式 y++，因为在选择语句的控制表达式 x--和下一个全表达式 y++之间存在一个序列点：

```
if (x --) y ++;
```

前序关系有如下特点：

（1）前序关系是成对的，一个相对于另一个；

（2）前序关系是传递的，如果 A 前序于 B，B 前序于 C，则 A 也前序于 C；

（3）它是一种偏序关系，如果 A 前序于 B，A 也前序于 C，则无法保证 B 是否前序于 C 或者 C 是否前序于 B，它们既可能是无序的（参见“无序”），也可能是不确定顺序的（参见“不确定顺序”）；

（4）前序关系是反对称的，不存在 A 前序于 B，而 B 又前序于 A 的情况。

6.8.2 后序

给定单线程上的任意两个求值 A 和 B，如果总是能够确定 A 完成时，B 还尚未开始，即，B 在 A 之后进行，则称 B 是后序（sequenced after）于 A 的。

因为表达式的求值包括值计算和副作用，因此后序意味着一个表达式的值计算和副作用总是发生在另一个表达式的值计算和副作用之后。

下例中，表达式 y--后序于表达式 x--，因为在逗号运算符的左操作数 x--和右操作数 y--之间存在一个序列点：

```
return x --, y --;
```

6.8.3 无序

给定单线程上的任意两个求值 A 和 B，如果不能确定 A 是否会 B 之前完成，也不能确定 A 是否会在 B 之后完成，而对于 B 来说，也同样不能确定是否先于 A 完成或者后于 A 完成，则称 A 和 B 是无序（unsequenced）的。

因为表达式的求值包括值计算和副作用，因此无序意味着两个表达式的求值过程有可能是交叉、交错进行的，比如，给定表达式 X 和 Y，先计算表达式 X 的值，再计算 Y 的值，然后发起 Y 的副作用，最后发起 X 的副作用。

6.8.4 不确定顺序

给定单线程上的任意两个求值 A 和 B，如果它们的顺序只有两种可能：A 在 B 之前完成（B 在 A 之后完成），或者 B 在 A 之前完成（A 在 B 之后完成），则称 A 和 B 是不确定顺序（indeterminately sequenced）的。

因为表达式的求值包括值计算和副作用，因此不确定顺序意味着两个表达式的求值无法确定谁先谁后，但这两个表达式的求值绝对不会交错进行（即，不是无序）。

6.9 优先级

如果一个表达式有多个运算符或者多个操作数，则运算符的优先级（precedence）决定了谁先和它附近的操作数结合在一起。进一步地，这也决定了表达式的类型。举个简单的例子，对于表达式 a*b+c 来说，乘法运算符*的优先级高于运算符+，所以乘法运算符*优先与操作数 b 结合，而不是运算符+优先与 b 结合。于是上述表达式等同于(a*b)+c，这是一个加性表达式。

再如，对于表达式(3 + x) * f ()来说，括号括住的表达式在运算符与操作数的结合上有优先权，因此，运算符+优先与 3 和 x 结合。然后，函数调用运算符()的优先级高于乘法运算符*，故 f 优先与()结合，这是一个乘性表达式。

在表达式列表（参见词条“表达式列表”）中，表达式类型的出现顺序暗示了各种运算符的优先级。

（1）组成同一种表达式的那些运算符具有相同的优先级。例如，组成乘性表达式的运算符有*、/和%，它们的优先级相同。

（2）出现在表达式列表前面的表达式，组成它的运算符，其优先级高于后面那些表达式的运算符。例如，乘性运算符（*、/和%）的优先级高于加性运算符（+和-）。

每种表达式所包括的运算符，可参见各表达式的词条。运算符的优先级只表明一个操作数优先和哪个运算符结合，但不能决定表达式的求值顺序。例如表达式 a*b+c，运算符*（而不是+）优先与 b 结合，但这只能决定一件事：运算符*的结果来自于操作数 a 和 b，运算符+的结果来自操作数 a*b 和 c。但是，运算符*的操作数 a、b 谁先计算，运算符+的操作数 a*b 和 c 谁先计算，都是不确定的，取决于 C 实现。要了解更多关于表达式求值顺序的描述，参见“求值”。

6.10 结合性

在处理运算符的优先级问题时，优先级较高的运算符优先与它旁边的操作数结合。如果所有的运算符优先级相同，则运算符的结合性（associativity）决定了它们应该按照怎样的顺序依次与旁边的操作数结合。结合方式有两种，分别是左结合和右结合，具体内容可参见词条“左结合”及“右结合”。

例如，有表达式

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

则乘法运算符*的优先级高于运算符+，所以，该表达式等同于

```
(a * b) + c + d
```

但问题是，操作数 c 是应该和左边的运算符+结合呢，还是和右边的运算符+结合？由于运算符+是左结合（从左向右）的，所以，上述表达式进一步等效于

```
((a * b) + c) + d
```

又如，表达式为

```
a = b = c = 1
```

赋值运算符=是右结合的，因此，该表达式等同于

```
a = (b = (c = 1))
```

运算符的结合性只表明一个操作数优先和哪个运算符结合，但不能决定表达式的求值顺序。例如表达式 a+b-c，运算符+和-具有相同的优先级，且它们都是左结合的。因此，运算符+（而不是-）优先与 b 结合，但这只能决定一件事：运算符+的结果来自于操作数 a 和 b，运算符-的结果来自于操作数 a+b 和 c。至于运算符+的操作数 a 和 b 谁先计算，运算符-的操作数 a+b 和 c 谁先计算，都是不确定的，取决于 C 实现。要了解更多关于表达式求值顺序的描述，参见“求值”。

6.10.1 左结合

在处理一个表达式时，可能所有的运算符都具有相同的优先级。在这种情况下，取决于具体的表达式，如果要求最左边的运算符优先与它旁边的操作数结合，然后依次是右边的那些运算符，那么我们说这些运算符是左结合的。下面是一个左结合的例子。

```
s. t. f ()
```

其中，运算符.和()的优先级相同，但它们同属于后缀运算符，故该表达式等同于

```
((s. t). f) ()
```

左结合的运算符包括：后缀运算符、乘性运算符、加性运算符、移位运算符、等性运算符、逐位与运算符、逐位异或运算符、逐位或运算符、逻辑与运算符、逻辑或运算符和逗号运算符。

6.10.2 右结合

在处理一个表达式时，可能所有的运算符都具有相同的优先级。在这种情况下，取决于

具体的表达式，如果要求最右边的运算符优先与它旁边的操作数结合，然后依次是左边的那些运算符，那么我们说这些运算符是右结合的。下面是一个右结合的例子。

```
x = y ^= 3
```

其中，运算符=和^=的优先级相同，但它们同属于赋值运算符，且赋值运算符是右结合的，故该表达式等同于

```
x = (y ^= 3)
```

例如，对于表达式

```
x ? y : z ? 0 : 1
```

这里使用了两个条件运算符? :，且此运算符是右结合的，因此，该表达式等同于

```
x ? y : (z ? 0 : 1)
```

同样地，表达式

```
a ? x ? y : z : b
```

等同于

```
a ? (x ? y : z) : b
```

要了解其中的原理，可参见词条“条件表达式”。

相对于左结合的运算符来说，右结合的运算符比较少，它包括一元运算符、转型运算符、条件运算符和赋值运算符。

6.11 求值

求值（Evaluation）是对表达式的处理过程，通常包括两方面的动作：一是计算表达式的值（值计算），二是发起一个可能存在的副作用。当然，也有可能什么都不做。

值得注意的是，不要求这两个动作是连续进行的。换句话说，求值未必是一个连贯的过程。例如，对于表达式++ x + y ++来说，它的求值大体上包括如下几个过程：求值++x、求值y++，以及计算运算符+的值，而且这个过程中有副作用（将修改后的值写回到对象 x 和 y）。

求值包括值计算和副作用，如果将计算子表达式++x 的值记为 V_{++x}，它的副作用记为 S_{++x}；将计算子表达式 y++的值记为 V_{y++}，它的副作用记为 S_{y++}；再将计算运算符+的值（也是整个表达式的值）记为 V_{+}，则按照运算符的优先级和结合性，这些事件发生的顺序可能是 $V_{++x}\rightarrow S_{++x}\rightarrow V_{y++}\rightarrow S_{y++}\rightarrow V_{+}$；也可能是 $V_{y++}\rightarrow V_{++x}\rightarrow V_{+}\rightarrow S_{++x}\rightarrow S_{y++}$。当然，也可能是其他顺序，具体按哪种顺序进行取决于 C 实现，只要保证 V_{+}发生在 V_{++x} 和 V_{y++}之后（操作数的值计算必须先于运算符的值计算），并且 V_{y++}发生在 S_{y++}之前（后缀递增表达式的值计算前序于修改对象存储值的副作用，参见“后缀递增”）即可。

这就是说，在任何表达式的求值过程中，其所有子表达式的值计算和副作用是无序的，但是按照标准的指引，在本书中有特别说明或者明确指定了顺序的情况例外。

再比如，表达式

```
++ a + b - c
```

按照运算符的优先级和结合性，它等同于表达式

```
((++ a) + b) - c
```

如果将计算++a 的值记为 V_{++a}，副作用记为 S_{++a}；将计算 b 和 c 的值分别记为 V_b 和 V_c（它们没有副作用）；将计算运算符+的值记为 V_+；将计算运算符-的值记为 V_-，则运算符的优先级和结合性不能决定表达式的求值顺序，表达式的实际求值顺序可能为

$V_{++a} \rightarrow S_{++a} \rightarrow V_b \rightarrow V_+ \rightarrow V_c \rightarrow V_-$

也可能为

$V_{++a} \rightarrow V_b \rightarrow V_c \rightarrow V_+ \rightarrow V_- \rightarrow S_{++a}$

还可能是

$V_c \rightarrow V_b \rightarrow V_{++a} \rightarrow S_{++a} \rightarrow V_+ \rightarrow V_-$

还有其他更多可能会发生的顺序，这里不再一一列举。至于到底选择哪一种顺序，取决于 C 实现，只要保证 V_+发生在 V_{++a} 和 V_b 之后，V_-发生在 V_+和 V_c 之后即可。

在前面的例子中，子表达式的求值顺序不影响最终结果的正确性。但这不是普遍真理。事实上，如果有多个副作用针对的是同一个标量对象，但这些副作用之间是无序的，则程序的行为是未定义的；针对同一个标量对象，如果既有副作用又有值计算，且它们之间是无序的，程序的行为也是未定义的。举例来说，给定以下代码：

```
int x = 0, f (int, int);
x ++, ++ x;      //S1
x ++ * ++ x;     //S2
f (x, ++ x);     //S3
x ++ + x;        //S4
```

则 S1 是没有问题的，因为 S1 中的全表达式是逗号表达式，尽管逗号运算符的左操作数x++与右操作数++x 在求值时都有副作用且作用于同一个标量对象 x，但是在逗号运算符的左操作数和右操作数的求值之间有一个序列点（参见“逗号表达式”），所以 x++的副作用前序于++x 的副作用，这里不存在未定义的行为，S1 执行完毕后，整个表达式的值是确定的，对象 x 最终的存储值也是确定的。

S2 中的全表达式是一个乘性表达式，等价于(x++)*(++x)。对于这种类型的表达式，标准并未特殊规定求值顺序，所以子表达式的求值是无序的。如果将 x++的值计算记为 V_{x++}，副作用记为 S_{x++}；将++x 的值计算记为 V_{++x}，副作用记为 S_{++x}，计算运算符*的值记为 V_*，则这整个全表达式的求值顺序可能是（假定对象 x 的初始存储值为 0）

V_{x++}（0）→S_{x++}（写 1 到对象 x）→V_{++x}（2）→S_{++x}（写 2 到对象 x）→V_*（0）

也可能为

V_{x++}（0）→V_{++x}（1）→S_{x++}（写 1 到对象 x）→S_{++x}（写 1 到对象 x）→V_*（0）

还可能是

V_{++x}（1）→S_{++x}（写 1 到对象 x）→V_{x++}（1）→S_{x++}（写 2 到对象 x）→V_*（1）

甚至是

V_{++x}（1）→S_{++x}（写 1 到对象 x）→V_{x++}（1）→V_*（1）→S_{x++}（写 2 到对象 x）

还有其他更多可能会发生的顺序，这里不再一一列举。至于到底选择哪一种顺序，取决于 C 实现，只要保证 V_*发生在 V_{++x} 和 V_{x++}之后即可。但是很显然，求值顺序不同，整个表达式的值也不同，对象 x 最终的存储值也不同，说明 S2 的行为是未定义的。

S3 中的全表达式是一个函数调用，标准只是规定在实际参数的求值和函数体的执行之间

有一个序列点，但实际参数的求值之间并未指定顺序。如果将参数 x 的值计算记为 V_x；将参数++x 的值计算记为 V_{++x}，它的副作用记为 S_{++x}；实际的函数调用记为 C_f，则这整个全表达式的求值顺序可能是（假定对象 x 的初始存储值为 0）

V_x（0）→V_{++x}（1）→S_{++x}（写 1 到对象 x）→C_f

也可能为

V_{++x}（1）→S_{++x}（写 1 到对象 x）→V_x（1）→C_f

当然还有其他可能会发生的顺序，这里不再一一列举。至于到底选择哪一种顺序，取决于 C 实现，只要保证 C_f 发生在 V_{++x}、S_{++x} 和 V_x 之后即可。但是很显然，求值顺序不同，传递给函数 f 的实际参数值也不同，这可能会影响到函数的返回值，也就是整个表达式的值，说明 S3 的行为是未定义的。

最后来看 S4 中的全表达式，它等同于(x ++) + x。对于这种类型的表达式，标准并未特殊规定求值顺序，所以子表达式的求值是无序的。如果将计算 x++的值记为 V_{x++}，其副作用记为 S_{x++}；将计算 x 的值记为 V_x；再将得到运算符+的值记为 V_+，则整个表达式的求值顺序可能是（假定对象 x 的初始存储值为 0）

V_{x++}（0）→S_{x++}（写 1 到对象 x）→V_x（1）→V_+（1）

也就是说，整个表达式的值是 1。但是，求值顺序也可能是

V_x（0）→V_{x++}（0）→S_{x++}（写 1 到对象 x）→V_+（0）

还可能是

V_x（0）→V_{x++}（0）→V_+（0）→S_{x++}（写 1 到对象 x）

还有其他更多可能会发生的顺序，这里不再一一列举。至于到底选择哪一种顺序，取决于 C 实现，只要保证 V_+发生在 V_x 和 V_{x++}之后即可。但是很显然，求值顺序不同，整个表达式的值也不同，对象 x 最终的存储值也不同，说明 S4 的行为是未定义的。

和上面的大多数示例相反，再来看另一个表达式

```
i = i + 1
```

其中，运算符=左边的 i 有一个副作用，右边的 i 计算一个值，它们是无序的吗？或者说，这个表达式的行为是未定义的吗？因为这是一个赋值表达式，所以我们要先来了解一下赋值表达式。赋值表达式有以下几个特点（参见词条“赋值表达式”）。

（1）运算符=的左操作数指示一个对象，运算符=的功能是向这个对象中存储一个值。

（2）表达式的值是运算符=的左操作数在赋值之后的值。

（3）计算运算符=左右两个操作数的值，这些操作前序于修改运算符=左操作数的值（的副作用）。

也就是说，只是在计算出 i+1 的值后，才会修改 i 的存储值，所以这个表达式没有问题。

相比之下，表达式

```
i = i ++
```

只是规定了何时修改运算符=左操作数所指示的对象（这是一个副作用），但 i++的副作用在整个表达式求值期间不知道什么时候发生，这两个副作用是无序的。

同样地，表达式

```
(i = 0) + i
```

左边的 i 的副作用相对于右侧 i 的值计算来说也是无序的，所以，该表达式的行为同样是未定义的。

求值包括值计算和副作用，但是对 void 表达式的求值只关心其副作用，可参见词条“void 表达式”。

6.12 基本表达式

基本表达式通常作为其他表达式的基本构件而存在。基本表达式包括标识符、（各种）常量、字面串、泛型选择和括住的表达式。其中，泛型选择是新近引入的。

被声明为对象类型或者函数类型的标识符是表达式的一个基本构件，因此，这两种类型的标识符是基本表达式。进一步地，若一个标识符被声明为对象类型，则它是一个左值；若被声明为函数类型，则它是一个函数指示符。参见“声明”、“左值”、“函数指示符”和“标识符”。以下是左值和函数指示符的示例。

```
int a = 0, f (int, int);
a = a + 3;          //运算符=两边的a都是指示对象的基本表达式
f ();               //这里的f是指示一个函数的基本表达式
```

常量也是表达式的基本构件，因此，它也是一种基本表达式。常量表达式的类型和它的值以及该常量的字面形式有关。例如，2ULL 表示一个 unsigned long long int 类型的常量，其值为 2。有关常量的话题可参见词条“常量”。下例中，字体加粗的部分是使用了常量的表达式。

```
long long ago = 'x' + 2LL;
```

其中，'x'和 2LL 都是典型的常量，前者是字符常量，后者是整型常量。这里要顺便讨论一下类型提升的话题。ago 的类型是 long long int，在赋值之前，需要先计算运算符“=”右边的表达式，并将结果转换为 long long int 类型。

但是，在运算符=的右侧，'x'和 2LL 的类型不同，'x'的类型是 int，2LL 的类型是 long long int，后者的转换阶高于前者，所以需要先将'x'的类型提升为 long long int，然后计算出相加的结果。结果的类型也是 long long int，和对象 ago 的类型相同，不必再做转换即可直接赋值。

表达式的基本构件还包括字面串，所以字面串也是基本表达式，而且是数组类型的左值。有关字面串的话题可参见词条“字面串”。例如下例中，字体加粗的部分是使用了字面串的表达式。

```
char c = "Primary Expression" [0];
```

可以用括号来改变运算符的优先级和结合性，这是大家都知道的。如果一个表达式以“(”开始，以“)”结束，中间是另一个表达式（可以是任意一个表达式），那么这样的一个表达式称为括住的表达式。

例如，要构建一个赋值表达式，则运算符=两边的操作数要求是赋值表达式及之前的表达式，即：

```
x = y ++
```

但是，这并不意味着不能使用那些在表达式列表（参见词条“表达式列表”）中，位于赋值表达式之后的表达式，你只需要为它们加上括号，使之成为基本表达式，如：

```
x = (y ? 0 : 1)
```

如果不使用括号来构建基本表达式，则上面的赋值表达式会变为条件表达式（因为运算符=优先与 x 和 y 结合），如：

```
x = y ? 0 : 1
```

括住的表达式被看作基本表达式，它的类型和值与去掉括号后的表达式是一样的。给定以下声明：

```
int x, y = 0, f (void);
void g ();
```

则下面的语句中使用了括住的表达式：

```
(x) = (y) + 1;  //等价于x = y + 1;
(f) ();         //等价于f ();
((g) ());       //等价于g ();
```

以上，因为 x 和 y 都是左值，所以(x)和(y)也是左值；因为 f 是函数指示符，所以(f)也是函数指示符；因为 g 是函数指示符，故(g)依然是函数指示符，(g)()是 void 表达式，所以((g)())也是 void 表达式。

泛型选择也是基本表达式，具体可参见词条“泛型选择”。

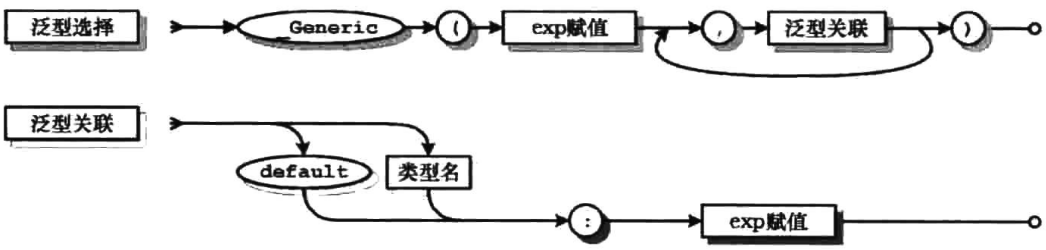
6.12.1 泛型选择

在编程实践中，可能会碰到这样的情况：想实现相同或者类似的功能，但需要根据操作数的类型分别实施不同的处理。为此，不得不定义多个不同名的函数。

一个典型的例子是数学函数，如正弦函数。为了应对 double、float、long double、double _Complex、float _Complex 和 long double _Complex 这些类型的参数，标准库定义了相应的函数 sin、sinf、sinl、csin、csinf 和 csinl。

比较明显的是，这些函数都做同一种事情。但是，能不能只定义一个函数，然后根据参数类型的不同来做相应的处理呢？在 C 中的确做不到这一点。为了方便使用，程序员及某些 C 实现，会选择使用宏定义的办法，在程序转换期间自动选择相应的函数，这就是所谓的泛型宏。因为缺乏标准和（C 的）实现层面的支持，大家的做法各不相同。

从 C11 开始，C 标准引入了泛型选择。泛型选择的语法形式如图 6-1 所示。



注：非终结符“exp赋值”是指赋值表达式及之前的表达式（参见词条“表达式列表”）；其它非终结符请参见同名词条。

图 6-1 泛型选择的语法

泛型选择是基本表达式，它在程序转换期间求值，其主要目的是从多个表达式中挑选一个做为结果。泛型选择表达式的类型就是被挑选出的那个表达式的类型；泛型选择表达式的值取决于被挑选出的那个表达式的值。下面通过一个实例来解释泛型选择表达式的功能。

```
# include <math.h>
# include <complex.h>

# define sin(x) _Generic(x,  \
                    float: sinf, \
                    double: sin, \
                    long double: sinl,   \
                    float _Complex: csinf,   \
                    double _Complex: csin,   \
                    long double _Complex: csinl) (x)
int main (void)
{
    float f = sin(2.0f);
    double _Complex d = sin(3.0+5.0i);
    /* ..... */
}
```

如上例所示，使用泛型选择可以方便和体面地解决正弦函数问题。

在泛型选择表达式中，第一个表达式称为控制表达式（上例中的 x），它并不求值，C 实现只提取它的类型信息。这个表达式要求是“赋值表达式及其之前的表达式”。

这意味着，它可以是除逗号表达式之外的任何表达式。这并不是说逗号表达式被绝对禁止了，事实上，如果要使用逗号表达式，则只需在两端加上括号，使之成为括住的表达式（基本表达式）。

如上例所示，在控制表达式的后面是一个或多个泛型关联，每个泛型关联由类型名、冒号和一个表达式组成。如果某个泛型关联中的类型名和控制表达式的类型兼容（匹配），则泛型选择的结果表达式就是该泛型关联中的表达式。

在上例中，如果 x 的类型是 double，则最终选择的是表达式 sin，当然，该表达式也是函数指示符。

泛型关联中的类型名所指定的类型必须是完整的对象类型。也就是说，象 void 这种不完整类型以及函数类型都是不允许的。

此外，在同一个泛型选择中，不允许两个或多个泛型关联的类型名所指定的类型互相兼容。换句话说，不允许控制表达式匹配多个泛型关联的类型名。

如果需要，可以使用一个 default 泛型关联。它的价值在于，如果控制表达式的类型和任何一个泛型关联的类型名所指定的类型都不兼容（匹配），则自动选择 default 泛型关联中的表达式。但是，一个泛型选择中只允许有一个 default 泛型关联。

下面的例子都是反面典型，每一行的注释标明了它们非法的原因。

```
typedef void F (int);
int m, f (void), g (void);
m = _Generic(m,
             F: f(),        //非法，泛型关联的类型名指定了函数类型
```

```
        F *: g(),      //允许，指向函数的指针是完整对象类型
        int: 1,
        signed: 2,     //非法，与前面的 int 兼容
        default: 3,
        default: 5     //非法，与前面的 default 泛型关联重复
        );
```

一个泛型选择表达式必须得到一个表达式做为结果，且这个表达式必须与控制表达式的类型兼容，这是一个硬性要求。因此，如果泛型选择中没有 default 泛型关联，则控制表达式的类型必须和某个泛型关联的类型名所指定的类型兼容（匹配），否则将招致一个错误。

作为总结，下面的例子展示了如何利用泛型选择来判断两个类型是否兼容。

```
# include <stdio.h>

# define is_compatible(x, T) _Generic(x, T:1, default:0)

void f (void)
{
   printf ("%s\n", is_compatible('x', int) ? "yes" : "no");
   int i = 0;
   printf ("%s\n", is_compatible(i++, int) ? "yes" : "no");
   printf ("%s\n", is_compatible(i, const int) ? "yes" : "no");
}
```

要特别注意的是，泛型选择不能识别数组类型，因为数组类型的表达式会被转换为指向其首元素的指针。具体情况可参见“数组-指针转换”。

在下面的例子中，字面串"ab"会被转换为指针，而尽管 pa 是指向数组的指针，表达式*pa 的类型是数组，但这个数组也会被转换为指针。因此，当程序运行时，写到标准输出的是“char *”和“int *”而不是“char [3]”和“int [5]”。

```
# include <stdio.h>

void f (int (* pa) [5])
{
   printf (_Generic ("ab",
                     char * : "char *",
                     char [3] : "char [3]")
           );

   printf ("\n");

   printf (_Generic (* pa,
```

```
                int * : "int *",
                int [5] : "int [5]")
        );
}
```

6.13 后缀表达式

后缀表达式包括复合字面值、数组下标、函数调用、(结构或联合的)成员选择，后缀递增(++)和后缀递减(--)，详情可参见其各自的词条。

组成后缀表达式语法形式的运算符称为后缀运算符。

以下是几个后缀表达式的例子，以粗体形式出现的都是后缀表达式。

```
int f (int), x = 0;
x ++;
f (x);
struct t {int i;} a [] = {{1}, {2}, {3},};
a [0].i ++;
```

后缀表达式是左结合的，因此，表达式

```
a [0].i ++
```

等同于

```
((a [0]).i) ++
```

6.13.1 复合字面值

复合字面值属于后缀表达式。

有时可能会遇到这种情况：有一个函数 fn，它接收一个结构类型的参数。为了调用这个函数，必须先为传递这样的参数创建一个对象。但是，这个对象仅仅是为了调用函数而创建，在程序的其他地方用不到它：

```
struct t {int i; char c;};
void fn (struct t);

struct t t = {0, 'a'};
fn (t);
/* 后面的代码不再用到 t */
```

这样编写的程序可以工作得很好，但总觉得有些违和感。如果 fn 是用户自己写的函数，可以改进它的参数类型以避免使用结构，但如果是一个位于库中的系统函数呢？复合字面值可以帮用户避开这种问题。

一个用“(”和“)”括住的类型名(参见“类型名”)，后面跟着一个前后两边分别是“{”和“}”的初始化器(参见“初始化器”)，这样的后缀表达式称为复合字面值。

复合字面值会创建一个对象，但这个对象没有名称。同时，复合字面值对象会有一个初

始值，这个值来自于初始化器列表，这个值同时也是复合字面值（表达式）的值。

下例中，复合字面值（以粗体形式给出的部分）创建了一个没有名称的对象。同时，复合字面值作为表达式，也要计算出一个值，这个值来自于它的初始化器列表。所以，该复合字面值所创建的对象初值为0，而表达式(int) {0}的值也是0。

进一步地，因为该复合字面值位于赋值运算符=的右侧，所以，它所指示的对象将执行左值转换（参见“左值转换”），转换后的值是该对象的存储值，也就是表达式(int) {0}的值。最后，这个值赋给对象i，赋值后，对象i的值也为0，即

```
int i = (int) {0};
```

作为表达式，复合字面值也有自己的类型，这个类型是其类型名所指定的类型。复合字面值会创建没有名称的对象，而“对象”一词则暗示类型名不允许指定函数类型，而只能是对象类型。但是，也并非所有对象类型都是允许的，如变长数组类型、不完整的对象类型就不可以。唯一允许的不完整类型是未指定大小的数组，因为可以从初始化器列表中获得它的大小。在这种情况下，复合字面值是完整的数组类型。

若给定声明：

```
int n = 3;
void f (int);
```

则下面是一些复合字面值的例子。其中，D5 是非法的，因为它企图创建变长数组类型的复合字面值。

```
(char) {'x'};                          //D1
(float []) {0.01, 0.02, 0.03};         //D2
 (int *) {0};                          //D3
(int *) {& n};                         //D4
(int [n]) {1, 2, 3};                   //D5
(void (*) (int)) {f};                  //D6
(int *) {& (int) {0}};                 //D7
```

D5 之外的复合字面值都是合法的，D1 的类型是 char；D2 的类型是数组，元素类型为 float；D3 的类型是指向 int 的指针，它是用空指针常量 0 初始化的；D4 的类型和 D3 相同，但它是用对象 n 的地址初始化的；D6 的类型是指向函数的指针（指针是完整的对象类型），它和函数 f 的类型相同，所以它使用指向 f 的指针来初始化（f 是函数指示符，在这里被转为指向函数的指针，参见“函数指示符-指针转换”）。复合字面值可以嵌套，所以，D7 是复合字面值嵌套使用的例子，这将创建两个复合字面值对象。

复合字面值是表达式，而且还是左值。也正是因为如此，可以编写如下代码：

```
char c = (char []) {"2015-06-27"} [0];
```

上例中加粗的部分是复合字面值，它的类型是数组，这是一个左值，因此可以应用下标运算。

再如下面的例子，字体加粗的部分是复合字面值，它的类型是数组，具有3个元素。这里用运算符[]和&得到其首元素的地址，这是一个指向 char 的指针（char *），可以合法地赋给指针对象 p。

```
char * p = & (char []) {'a', 'b', 'c'} [0];
```

因为在当前这种场合下，数组会自动转换为指向其首元素的指针（参见“数组-指针转换”），所以，此行语句等同于下面这行语句。

```
char * p = (char []) {'a', 'b', 'c'};
```

如果想得到指向数组的指针，那么可以使用下面的方法。

```
char (* p) [] = & (char []) {'a', 'b', 'c'};
```

下面是另一个完整的示例，用粗体给出的部分是复合字面值，它的类型是 long long int。为了验证这一点，这里使用了泛型选择表达式和 printf 函数，如果复合字面值的类型的确是 long long int，则把这个类型名以字符串的形式写到标准输出。

```
# include <stdio.h>

void f (void)
{
   printf (_Generic ((long long int) {8},
                     long long int : "long long int.\n",
                     default : "Unknown.\n")
           );
}
```

现在来考虑一下开始的那个例子。使用复合字面值，可以避开这种做法，这对有“代码洁癖”的程序员来说很有帮助（第一个函数调用里的复合字面值比较特殊，因为它的初始化器部分使用了指示器.i 和.c，这种用法的详情可参见“初始化器”）：

```
struct t {int i; char c;};
void fn (struct t);

fn ((struct t) {1, 'x'});
fn ((struct t) {.i = 0, .c = 'a'});
```

下面的例子用于创建指针的数组（元素类型为指针的数组）。这是一个嵌套的复合字面值，外部的复合字面值用于创建一个数组，内部的复合字面值用于创建数组的元素，同时用一元&运算符将其变成指向 int 的指针（int *）。因为这是一个元素类型为指针的数组（int * []），在将它赋给 p 时，隐式转换为指向其首元素的指针，即 int * *。

```
int * * p = (int * []) {& (int) {0}, & (int) {1}, & (int) {2},};
```

位于函数外部的复合字面值与块内的复合字面值，它们所创建的对象具有不同的存储期，前者具有静态存储期，在程序启动时创建和初始化；后者具有自动存储期，在程序的执行进入它所在的块时创建，并在程序的执行到达它所在的位置时初始化。具体的讲解和示例参见“存储期”，这里从略。

复合字面值最好的用途是做为函数调用时的参数（且此参数不再使用）。做为学习复合字面值的示例，尽管下面的代码都是合法的，但没有什么意义（这可以做为思考题，请你说说这些代码的功能）：

```
int m = sizeof (int []) {5, 6, 7};
(int) {0} = 99;
```

```
int n = 0;
* ((int *) {& n}) = 56;
```

6.13.2 数组下标

数组下标表达式属于后缀表达式。

数组下标用于指示一个数组对象的元素，即数组对象的子对象，它是一个左值。数组下标表达式由一个表达式 E，一个“[”、另一个表达式、以及一个“]”组成，其中 E 要求是后缀表达式及之前的表达式（参见词条“表达式列表”。如果要使用其他表达式，则可以用括号将它变为基本表达式），所以 E 本身也可能是另一个数组下标表达式。

下面是一些下标表达式的例子。

```
int a [2] = {0, 0}, b [2] [3];
b [0] [1] = a [0] ++;
```

最后一行代码中的 b[0]、b[0][1]和 a[0]都是下标表达式。注意，下标表达式 b[0][1]是从另一个下标表达式 b[0]递归构建的。

数组下标需要两个不同类型的操作数：一个是指向完整对象类型的指针；另一个是整数。数组下标表达式的结果类型是那个指针所指向的类型。例如：

```
void f (char * p)
{
    char c = p [0]/* E1 */;
    /* …… */
}

void g (void)
{
    int a [2];
    a [0]/* E2 */ = 1;
    (& a [0]) [1]/* E3 */ = 2;
    f (a);
}
```

在 E1 中，p[0]就是典型的下标表达式，p 是指针类型，而 0 是整数类型。在 E2 中，左值 a 将自动转换为指向其第一个元素的指针；在 E3 中，是先生成指向数组 a 首元素的指针（&a[0]），另一个操作数是整数 1，该表达式实际上等同于 a[1]。两者的区别是前者已经转换为指针，而后者还需要隐式转换。

再来看一个例子，因为在上述语法说明中并未明确指明哪个操作数是指针，哪个操作数是整数，所以，给定声明：

```
int a [3];
```

则以下 3 个语句都包含了正确的数组下标表达式。

```
a [0] = 1;
1 [a] = 1;
```

```
2 [a] = ++ 0 [a];
```

其中，左值 a 被转换为指向其第一个元素的指针。如果采用以上后两种写法，对于多维数组的情况就要小心，不要出错。下面给出了正确和不正确的例子。

```
int a [2][3];
a [0][0] = 1;     //可以
0 [a][1] = 2;     //完全正确
1 [0][a] = 3;     //非法
```

最后一个表达式之所以不正确，是因为后缀运算符是左结合的，所以该表达式等同于

```
(1 [0]) [a] = 3
```

显然，这是一个递归构建的下标表达式。而对于下标表达式 1[0]来说，按要求，1 和 0 中必有一个操作数是指针类型，但 1 和 0 都不是指针。

数组下标表达式 E1[E2]等价于(*((E1)+(E2)))。在下面的例子中，a[0]等同于指针操作*(a+0)，而指针操作*(a+1)等价于下标操作 a[1]。

```
int a [2];
a [0] = 10086;       //等同于*(a+0) = 10086;
* (a + 1) = 95533;  //等同于a[1]=95533;
```

又如

```
int a [2][3];
a [0][1] = 10086;
```

根据运算符的结合性，表达式 a[0][1]等同于(a[0])[1]。按照上面的规则，这个表达式等同于* (a [0] + 1)。这其中的 a[0]可以继续转换，因此，它又等同于表达式* (* (a + 0) + 1)。类似地，更多维度的数组也可按此办理。

有些读者认为(*((E1)+(E2)))括号太多，有碍观瞻，有这种认识的读者可考虑以下内容：

```
int n, a [2] = {0, 1};
a [n = 1] ++;        //其等价于*(a+n=1)++还是(*(a+(n=1)))++ ?
```

在使用下标指示数组元素时，将数组等同于指针是比较方便的做法，同时它也带来一个额外的效果，即，如果一个指针指向某个对象，那么即使这个对象不是数组，也可以认为它是一个数组，且这个数组只有一个元素。换句话说，在下标操作中，指向对象的指针依然被认为是指向数组首元素的指针，且这个数组只有一个元素。例如：

```
# include <stdio.h>

void f (void)
{
      int i = 10086, * pi = & i;
      printf  ("%d, %d\n", pi [0], * pi);       //这将会输出两个相同的数
}
```

下面是另一个数组下标表达式的例子，请结合复合字面值，以及下标运算与指针运算的等效性自行分析：

```
# include <stdio.h>
```

```
void f (void)
{
    int * * p = (int * []) {& (int) {0}, & (int) {1}, & (int) {2},};
    printf ("%d, %d, %d\n", p [0] [0], p [1] [0], p [2] [0]);
}
```

6.13.3 函数调用

函数调用表达式属于后缀表达式。

一个用于指代被调用函数的表达式（指向函数的指针，或者函数指示符）E，加上一个“(”，和一个可选的表达式列表，以及一个“)”，就组成了函数调用表达式。

其中，E 是后缀表达式及之前的表达式（参见词条“表达式列表”。如果要使用其他表达式，则可以用括号将它变成基本表达式），所以 E 本身也可能是另一个函数调用表达式；表达式列表（如果有）指定了传递给函数的实际参数，每个参数之间要用逗号分开。

注意，表达式列表中的每个表达式要求是赋值表达式及之前的表达式（参见词条“表达式列表”）。但这并不是说绝对禁止逗号表达式（只有逗号表达式排在赋值表达式的后面），只是要先把逗号表达式用括号变为基本表达式。下例中，字体加粗的部分都是函数调用表达式：

```
double f1 (void), f2 (double, double), x, y, z;
/* …… */
f1 ();
f2 (0.1, 0.5);
f2 (0.2, (x ++, y ++, z ++));  //第二个参数是逗号表达式
```

在下面这个例子中，函数 f 的返回类型是指向函数的指针；表达式 f()(1)是一个后缀表达式，它是在另一个后缀表达式 f()的基础上递归构建而成。

```
void (* f (void)) (int) {/* …… */}

void g (void)
{
    f () (1);
}
```

表达式 E 指代被调用的函数，可以是一个指向函数的指针，也可以是函数指示符。如果是函数指示符，则将自动执行“函数指示符-指针转换”。

下面是一个通过复合字面值及结构成员进行函数调用的例子，在第二个函数调用中，结构的成员 pf 是一个指向函数的指针，可以直接进行函数调用。

```
# include <stdio.h>

void f (int i) {printf ("%d\n", i);}

void g (void)
{
```

```
    (void (*) (int)) {f} (0);

    struct t {void (* pf) (int);} t = {f};
    t.pf (1);
}
```

调用函数时，传递的实际参数在进入函数体执行前完成各自的求值。显而易见的是，它们都必须是完整的对象类型，否则将无法求值和传递。每个实际参数将传递给与之相对应的形式参数（实际上是传递给形参所指示的对象）。参见“形参”和“实参”。

下例中，在调用函数 f 时，为它传递了两个参数 3 和 0.01f。D 处是函数 f 的声明，在这种情况下，3 传递给声明 D 中的第一个参数；0.01f 传递给声明 D 中的第二个参数。

```
int f (int, float);          //D

int g (void)
{
    return f (3, 0.01f);
}
```

表达式 E 是函数指示符的，先求值为指向函数的指针。函数指示符和每个实际参数的求值前序于实际的函数调用。也就是说，它们之间存在一个序列点。但是，各个实际参数的求值是无序的。因此，函数调用表达式

```
fn (x ++, y ++, z)
```

在函数体开始执行前，表达式 x++、y++和 z 必须完成求值，但这 3 个表达式的求值是无序的。

调用一个函数时，如果它是用原型声明的，则要求实参的数量和类型必须与原型一致。当然，在进行这种检查之前，形参或者实参是数组或者函数类型的，将分别自动调整为指向数组首元素的指针和指向函数的指针。实参能够自动转换为形参类型的，也视为一致。

形参可能会被声明为限定的类型，并在函数调用时创建相同限定的形参对象。但是，该对象在接收调用者传递的（实参）值时，被视为是具有无限定的类型。

请看下面的例子。

```
const int x = 0;
void f (const int);
x ++;          //非法
f (x, 1);      //非法：参数太多
f (x);         //合法
```

其中，函数 f 的声明采用了原型的形式，函数调用也是受原型控制的。因为 x 具有 const 限定的类型，故表达式 x++是非法的；函数 f 在声明时具有 1 个参数，所以 f(x,1) 也是非法的。尽管函数的参数在声明时具有 const 限定的类型，但这不影响参数传递，不会因为形参是 const 限定的类型而传递不进去。

再来看另一个示例：

```
void f (float f, char a [], void g (void)) {/* …… */}
void g (int a []) {/* …… */}
```

```
void h (void)
{
    f (2.0, "", 0);     //S1: 合法
    g ("");             //S2: 非法
}
```

以上，S1 和 S2 中的函数调用都是受原型控制的。函数 f 需要三个参数，其类型分别是 float、指向 char 的指针和指向函数的指针。在 S1 中，2.0 的类型是 double，但可以自动转换为 float；""是字符串，自动转换为指向 char 的指针；0 是空指针常量，自动转换为指向函数的空指针，类型是 void (*) (void)。请参见“类型转换” 和“字面串”。

函数 g 需要一个指针类型的参数（int *），但实参的类型却是 char *，无法自动从 char *转换到 int *，所以 S2 是非法的。

若函数调用不是受原型控制的，即被调用的函数不是采用原型的形式声明的，则将对每一个传递给函数的（实际）参数实施默认参数提升（参见“默认参数提升”），前提是它们可以被提升。

下例中，函数 krf 是用传统 K&R 的形式定义的，在调用这个函数时，字符类型的参数 c 被提升为 int；float 类型的参数被提升为 double，但指针类型的参数保持不变：

```
int krf (x, y, z)
char x; float y; float * z;
{
    /* ..... */
}

void g (void)
{
    char c = 'x';
    float f = 3.0;
    krf (c, f, & f);
}
```

表达式 E 所指代的函数和实际被调用的函数必须是兼容的函数类型，否则程序的行为是未定义的。参见“兼容类型”。

先来看一个简单的例子，在下面的示例中，函数 f 被定义为接受一个 int 类型的参数，并返回一个 int 类型的值。但是，在函数 g 内，f 先被隐式地转换为指向函数的指针，接着，这个指向函数的指针被强制转换为指向另一种函数类型（不接受参数，返回类型为 void）的指针，然后按这个新类型进行函数调用，但这种做法的后果是不可预料的：

```
int f (int x) {/* ..... */}

void g (void)
{
```

```
    ((void (*) (void)) f) ();
}
```

相比之下，下面这个示例稍微复杂一点：

```
# include <stdio.h>

void (* f (void)) (void)
{
    static _Bool b = 0;
    printf (b -- ? "Copy that.\n" : "Do you copy?\n");
    return (void (*) (void)) f;
}

void g (void)
{
    f () ();
}
```

以上，函数 f 被定义为不接受任何参数，且返回一个指向函数的指针，被指向的函数也不接受任何参数，且返回类型是 void。

在函数 f 的内部定义了_Bool 类型的对象 b，它的初值为 0 且具有静态存储期。将它的值减 1 后，非零值被转换为 1；再减 1，又变成 0；再减 1，又变成 1（请参见“类型转换”中和 _Bool 类型有关的部分）。这意味着如果我们多次调用函数 f，则 printf 函数将一直交替输出“Do you copy?\n”和“Copy that.\n”。当然这不是最重要的，最重要的地方是，函数 f 将返回自己的地址给调用者。但是很明显，返回的类型和函数 f 自己的类型并不相同（所以在返回之前做了转换操作）。

来看函数 g，在 g 的内部调用了函数 f，而且调用了两次。第一次调用，即 f()，是合法的，因为函数指示符 f 在转换为指针后，它所指向的类型就是 f 被定义的类型。但是，第二次调用，即 f()()，是非法的。这是因为，第一次调用时虽然返回了一个指向函数的指针，且这个指针实际上指向函数 f，但这个指针所指向的类型却并不兼容于函数 f。具体地说，这个返回值（指针）所指向的类型是

```
void (void)
```

而函数 f 的类型却是

```
void (* (void)) (void)
```

前者的返回类型是 void，而后者是有返回值的，返回一个指向函数的指针。在这种情况下，以上程序的行为是不可预料的。

要改进上面的例子，可以使用下面的方法，这将在第一次调用函数 f 之后，将它的返回值强制转换为正确的类型，然后再以这个正确的类型再次做函数调用：

```
# include <stdio.h>

void (* f (void)) (void)
```

```
{
    static _Bool b = 0;
    printf (b -- ? "Copy that.\n" : "Do you copy?\n");
    return (void (*) (void)) f;
}

void g (void)
{
    ((void (* (*) (void)) (void)) f ()) ();
}
```

下面是一个更复杂的改进版本，将多次调用函数 f，具体的转换和调用过程，可以做为一道思考题，请你自行分析：

```
# include <stdio.h>

void (* f (void)) (void)
{
    static _Bool b = 0;
    printf (b -- ? "Copy that.\n" : "Do you copy?\n");
    return (void (*) (void)) f;
}

void g (void)
{
    typedef void (* (* F) (void)) (void);
    ((F) ((F) ((F) f ()) ()) ()) ();
}
```

C 语言允许直接或者间接的递归函数调用。下面是一个递归的例子，比较老套，目的是计算 1～100 的整数和。

```
# include <stdio.h>

long long cusum (int n)
{
    if (n == 1) return 1;
    else return n + cusum (n - 1);
}

int main (void)
{
    int n = 100;
```

```
    return printf ("1+2+3+...+%d=%lld\n", n, cusum (n));
}
```

标准要求 C 实现至少支持向函数传递 127 个参数。相应地，在定义函数时，要至少支持接收 127 个参数。

6.13.3.1 形参

形参（parameter）又称形式参数（formal parameter），是指函数声明（或者定义）的一部分，或者是函数式宏定义的一部分，总之是出现在函数声明和函数式宏定义中的参数。具体内容可参见“函数式宏定义”。

如果是函数声明或定义的一部分，则它指示一个具有自动存储期的对象，在函数被调用时创建，并用来接收传入的值，函数返回时撤销；如果是函数式宏定义的一部分，则它是紧跟在宏名之后的、一对圆括号内的标识符。

以下是形参的例子。在这个例子中，字体加粗的部分都是形参（x、y、a 和 b）。

```
# include <stdio.h>

# define F(x, y) x * y - y

void f (char a, char b)
{
    printf ("%c, %c", F(a,b), F(b,a));
}
```

C 语言中的函数不接受数组和函数类型的参数，这样的参数会分别转换为指向数组首元素的指针和指向函数的指针；形参的声明允许采用数组和函数的形式，但这并不意味着真的能从调用者那里“接受”这种参数，事实上，C 实现会将数组和函数类型的形参分别调整为指向数组首元素的指针和指向函数的指针。

来看下面这个例子：

```
void f (int a [3], int g (void)) {/* …… */}

void demo (void)
{
    int a [6], g (void);
    f (a, g);
}
```

尽管函数 f 的两个参数看起来是数组和函数，但事实上，它们会被调整为指向 int 的指针和指向函数的指针。因此，函数 f 的上述定义与下面的定义等价：

```
void f (int * a, int (* g) (void)) {/* …… */}
```

在调用函数 f 时，尽管传递的是数组 a 和函数 g，但它们会被转换为指向 int 的指针和指向函数 void(void)的指针。

6.13.3.2 实参

实参（argument）又称实际参数（actual argument），是指在调用函数时，或者调用一个宏时，指定或者说给出的参数。在函数调用表达式中，实参是实际传递给被调用函数的参数（值）；在调用一个宏时，实参是传递给宏体的预处理记号。具体内容可参见“函数式宏定义”。

如果是用于调用一个函数，那么实参位于函数调用表达式中，它们用一对括号括住，彼此用逗号分隔；如果是用于调用一个函数式宏，同样是用对一对括号括住，彼此也是用逗号分隔的。

以下是实参的例子。在这个例子中，字体加粗的部分都是实参。特别地，调用 printf 函数时的实参 F(a,b)、F(b,a)分别是宏调用。于是，对于宏调用 F(a,b)来说，a 和 b 是它的两个实参；对于宏调用 F(b,a)来说，b 和 a 也是它的两个实参。

```
# include <stdio.h>

# define F(x, y) (x) * (y) - (y)

void f (char a, char b)
{
    printf ("%c, %c", F(a,b), F(b,a));
}
```

C 语言中的函数不接受数组和函数类型的参数。可以在函数调用表达式中使用数组，但这并不意味着真的会将整个数组传递给被调用函数。事实上，数组会被转换为指向其首元素的指针，函数会被转换为指向函数的指针，这符合“数组-指针转换”和“函数指示符-指针转换”的原则，请参见“数组-指针转换”和“函数指示符-指针转换”。

在下例中，传递给函数 f 的两个实参分别被转换为指向数组首元素的指针和指向函数的指针，即分别是 char *和 void (*) (char *)。这种转换与 C 语言对函数形参的约束是一致的（参见“形参”），毕竟形参中的数组和函数也会做这种转换。

```
# include <stdio.h>

void g (char * s)
{
    printf (s);
}

void f (char a [] , void g (char *))
{
    g (a);
}

int main (void)
{
```

```
    f ("E: \\projects\\findx\\bin\\Debug\\findx.exe\n", g);
    return 0;
}
```

6.13.4 成员选择

成员选择表达式属于后缀表达式。

表达式 E，加上一个运算符“.”或者“->”，再加上一个标识符，即可构成成员选择表达式。其中，E 是后缀表达式及之前的表达式（参见词条“表达式列表”。如果要使用其他表达式，则可以用括号将它变为基本表达式），所以 E 本身也可能是另一个成员选择表达式。标识符指示结构或者联合的某个成员。

如果使用运算符“.”，则表达式 E 是一个结构或者联合类型的值或者左值；如果使用了运算符“->”，则表达式 E 为指向结构或者联合的指针。注：上述结构和联合类型可以是原子的，也可以是限定或无限定的。

成员选择表达式的值是成员的值，成员选择表达式的类型和那个被选择的成员相同。如果 E 是左值，或者 E 的类型是指针，则成员选择表达式的结果是一个左值；否则，成员选择表达式不是一个左值。如果 E 是限定的类型，则成员选择表达式的类型也是限定的。

例如，给定声明：

```
struct t {int i; struct {float f;} v;} t, * pt = & t, sa [3];
```

则以下的语句都包含成员选择表达式（字体加粗的部分都是）。

```
t.i = 0;
t.v.f = 0.0;
pt -> i ++;
sa [0].i = 0;
sa [0].v.f = 0.0;
```

又如，在下面的代码片段中，t 是左值，t.i 也是左值，故 E1 是合法的；函数返回一个结构，可以取得其成员的值，故 E2 是合法的；但是，函数返回的不是左值，故 E3 是非法的；&t 是一个指针，(&t)->i 是一个左值，故 E4 是合法的；赋值表达式 s=t 的结果不是左值，(s=t).i 的结果也不是左值，故 E5 是非法的；同理，在 E6 中，(s=t).i 不是左值，但它在这里的角色是初始化器，不要求必须是一个左值，故它是合法的；m 是一个左值，但它是 const 限定的结构，尽管 m.i 的结果也是左值，但不是可修改的左值，故 E7 也是非法的。

```
struct t {int i;} f (void), s, t = {0};
t.i ++;                 //E1：合法
int x = f ().i;         //E2：合法
f () ++;                //E3：非法
(& t)->i ++;            //E4：合法
(s = t).i ++;           //E5：非法
int y = (s = t).i;      //E6：合法
const struct t m;       //m 是具有 const 限定的类型
m.i = 3;                //E7：非法
```

6.13.5 后缀递增

后缀递增表达式属于后缀表达式。

表达式 E 加上一个运算符++，即可构成后缀递增表达式。其中，E 是后缀表达式及之前的表达式（参见词条“表达式列表”。如果要使用其他表达式，则可以用括号将它变为基本表达式）。

下面是一些后缀递增表达式的例子。

```
int i = 0, j = i ++, * p = & j;
(* p) ++;
for (char * p = "Grace Of Monaco 2014"; * p != '\0'; p ++) {/* …… */}
```

后缀递增表达式的结果不是左值，它的值是 E 所指示的对象在递增操作前的原值，而不是递增后的新值；后缀递增是具有副作用的表达式，这个副作用会修改 E 所指示的对象，将它的存储值变为表达式 E+1 的值。

下面的示例演示了后缀递增表达式的值和副作用。

```
# include <stdio.h>

void f (void)
{
    int i = 0, * pi = & i;

    printf ("%d\n", i ++);              //表达式 i++的值是 i 的原值
    printf ("%d\n", i);                 //查看表达式 i++的副作用

    printf ("%p\n", (void *) pi ++);    //表达式 pi++的值是对象 i 的地址
    printf ("%p\n", (void *) pi);       //查看表达式 pi++的副作用
}
```

通过以上例子可以看出，指针类型的递增很特殊。详情参见词条“加性表达式”，其中有指针类型的运算。但有些人可能认为以上最后两个 printf 函数调用可以合并为

```
printf ("%d,%d\n", i ++, i);
```

但这肯定是不可以的。原因很简单，i++和 i 的求值顺序是未指定的。

E 要求是实数类型或者是指针类型。不管是哪种类型，都可以是原子的、限定或者无限定的，而且必须是可修改的左值；后缀递增表达式的结果类型和 E 的类型相同。

所以，类似 i++ ++、250++或者(p++) = 3 这样的表达式都是非法的。下面是其他例子。

```
void f (const int * const p)
{
    p ++;                   //E1: 非法
    (* p) ++;               //E2: 非法
    const int * q = p;
    q ++;                   //E3: 允许
    (* q) ++;               //E4: 非法
```

```
    /* ..... */
}
```

其中，E1 中的 p 是左值（一个 const 限定的指针），但不是可修改的左值；E2 中的*p 是左值，但它指示的对象具有 const 限定的类型；E3 中的 q 是左值，而且是可修改的左值；E4 的情况和 E2 类似。

特别要指出的是，是先计算和得到后缀运算符++的值（这也是整个后缀递增表达式的值），再修改其操作数的存储值。这就是说，后缀++运算符的值计算和它的副作用有确定的前序和后序关系。

6.13.6 后缀递减

后缀递减表达式属于后缀表达式。

表达式 E 加上一个运算符--，即可构成后缀递减表达式。其中，E 是后缀表达式及之前的表达式（参见词条“表达式列表”。如果要使用其他表达式，则可以用括号将它变为基本表达式）。

下面是一些后缀递减表达式的例子。

```
int i = 3, j = i --, * p = & j;
(* p) --;
```

后缀递减表达式的结果不是左值，它的值是 E 所指示的对象在递减操作前的原值，而不是递减后的新值；后缀递减是具有副作用的表达式，这个副作用会修改 E 所指示的对象，将它的存储值变为表达式 E-1 的值。

下面的示例用于演示后缀递减表达式的值和副作用。

```
# include <stdio.h>

void f (void)
{
    int i = 3, * pi = & i;

    printf ("%d\n", i --);              //表达式 i--的值是 i 的原值
    printf ("%d\n", i);                 //查看表达式 i--的副作用

    printf ("%p\n", (void *) pi --);    //表达式 pi--的值是对象 i 的地址
    printf ("%p\n", (void *) pi);       //查看表达式 pi--的副作用
}
```

通过以上例子可以看出，指针类型的递减很特殊。详情可参见词条“加性表达式”，其中有指针类型的运算。有时有人可能认为以上两个 printf 函数调用可以合并为

```
printf ("%d,%d\n", i --, i);
```

但这肯定不可以。原因很简单，i—和 i 的求值顺序是未指定的。

E 要求是实数类型或者是指针类型。不管是哪种类型，都可以是原子的、限定或无限定的，而且必须是可修改的左值；后缀递减表达式的结果类型也是 E 的类型。

所以，以下表达式都是非法的。

```
i -- --
250 --
(p --) = 3
```

下面是另外一些例子。

```
void f (const int * const p)
{
    p --;                    //E1：非法
    (* p) --;                //E2：非法
    const int * q = p;
    q --;                    //E3：允许
    (* q) --;                //E4：非法
    /* ..... */
}
```

其中，E1 中的 p 是左值（一个 const 限定的指针），但不是可修改的左值；E2 中的*p 是左值，但它指示一个对象具有 const 限定的类型；E3 中的 q 是左值，而且是可修改的左值；E4 的情况和 E2 类似。

特别要指出的是，是先计算和得到后缀运算符--的值（也就是整个后缀递减表达式的值），再修改其操作数的存储值。这就是说，后缀--运算符的值计算和它的副作用有确定的前序和后序关系。

6.14 一元表达式

一元表达式包括前缀递增（++）表达式、前缀递减（--）表达式、尺寸（sizeof）表达式、对齐（_Alignof）表达式、地址（&）表达式、间接（*）表达式、正号（+）表达式、负号（-）表达式、按位反（~）表达式和逻辑非（!）表达式，详情可参见其各自的词条。

下面是几个一元表达式的例子。

```
int x = 0, y = - ! * & x, a [2] = {0, 1}, * p = a, z = ~ * ++ p;
```

在上面的例子中，& x、* & x、! * & x、- ! * & x、++ p、* ++ p、~ * ++ p 都是一元表达式。

组成一元表达式语法形式的运算符称为一元运算符；一元运算符是右结合的。因此，表达式

```
! * & p
```

等价于表达式

```
! (* (& p))
```

一元运算符+、-、~和!统称为一元算术运算符。

6.14.1 前缀递增

前缀递增表达式属于一元表达式。

运算符++加上一个表达式 E，即可构成前缀递增表达式。其中，E 是一元表达式及之前的表达式（参见词条“表达式列表”。如果要使用其他表达式，则可以用括号将它变为基本表达式）。

若给定声明：

```
int a [2] = {0, 1}, * p = a;
```

那么，++ a [0]、++ p 都是合法的前缀递增表达式。

前缀递增表达式的结果不是左值，它的值是 E 所指示的对象在递增操作后的新值，即它在数值上等于表达式 E+1 的值；前缀递增是具有副作用的表达式，这个副作用会修改 E 所指示的对象，将它的存储值变为表达式 E+1 的值。

例如：

```
# include <stdio.h>

void f (int i)
{
    printf ("%d\n", ++ i);  //表达式++i 的值是 i 递增后的新值
    printf ("%d\n", i);     //查看表达式++i 的副作用
}
```

有人可能认为以上两个 printf 函数调用可以合并为

```
printf ("%d,%d\n", ++ i, i);
```

但这肯定是不可以的。原因很简单，++i 和 i 的求值顺序是未指定的。

E 要求是实数类型的左值，或者是指针类型的左值。不管是哪种类型，都可以是原子的、限定或无限定的类型，而且必须是可修改的左值；前缀递增表达式的结果类型和表达式 E 的类型相同。

所以，类似-- ++ i 或者(++ p) = 3 这样的表达式都是非法的。下面是其他例子。

```
void f (const int * const p)
{
    ++ p;                   //E1：非法
    ++ * p;                 //E2：非法
    const int * q = p;
    ++ q;                   //E3：允许
    ++ * q;                 //E4：非法
    ++ & q;                 //E5：非法
    /* …… */
}
```

其中，E1 中的 p 是左值（一个 const 限定的指针），但不是可修改的左值；E2 中的*p 是左值，但它指示一个对象具有 const 限定的类型；E3 中的 q 是左值，而且是可修改的左值；E4 的情况和 E2 类似；E5 中的&q 虽然是指针，但不是左值。

在语义上，前缀递增表达式++E 等价于表达式 E+=1（可参见词条“复合赋值”）。

6.14.2 前缀递减

前缀递减表达式属于一元表达式。

运算符--加上一个表达式 E，即可构成前缀递减表达式。其中，E 是一元表达式及之前的表达式（参见词条“表达式列表”。如果要使用其他表达式，则可以用括号将它变为基本表达式）。

给定声明：

```
int a [2] = {0, 1}, * p = & a [1];
```

那么，-- a [1]、-- p 都是合法的前缀递减表达式。

前缀递减表达式的结果不是左值，它的值是操作数 E 所指示的对象在递减操作后的新值，即它在数值上等于表达式 E-1 的值；前缀递减是具有副作用的表达式，这个副作用修改 E 所指示的对象，将它的存储值变为表达式 E-1 的值。

下面的示例演示了前缀递减表达式的值和副作用。

```
# include <stdio.h>

void f (int i)
{
    printf ("%d\n", -- i);  //表达式--i 的值是 i 递减后的新值
    printf ("%d\n", i);     //查看表达式--i 的副作用
}
```

有时有人可能认为以上两个 printf 函数调用可以合并为

```
printf ("%d,%d\n", -- i, i);
```

但这肯定是不可以的。原因很简单，—i 和 i 的求值顺序是未指定的。

E 要求是实数类型的左值，或者是指针类型的左值。不管是哪种类型，都可以是原子的、限定或无限定的，而且必须是可修改的左值；前缀递减表达式的结果类型和表达式 E 的类型相同。

所以，类似-- ++ i 或者(++ p) = 3 这样的表达式都是非法的，++i 和++p 的结果都不是左值。下面是其他一些前缀递减表达式的例子。

```
void f (const int * const p)
{
    -- p;                    //E1：非法
    -- * p;                  //E2：非法
    const int * q = p;
    -- q;                    //E3：允许
    -- * q;                  //E4：非法
    -- & q;                  //E5：非法
    /* …… */
}
```

其中，E1 中的 p 是左值（一个 const 限定的指针），但不是可修改的左值；E2 中的*p 是左值，但它指示一个对象具有 const 限定的类型；E3 中的 q 是左值，而且是可修改的左值；E4 的情况和 E2 类似；E5 中的&q 虽然是指针，但不是左值。

在语义上，前缀递减表达式--E 等价于表达式 E-=1（可参见词条“复合赋值”）。

6.14.3 地址

地址表达式属于一元表达式。

一个“&”加上一个表达式 E，即可构成地址表达式。其中，E 是一元表达式及之前的表达式（参见词条“表达式列表”。如果要使用其他表达式，则可以用括号将它变为基本表达式），所以 E 本身也可能是另一个一元表达式。

运算符&的操作数 E 应为左值，但不能是位字段，也不能是用存储类指定符 register 声明的对象类型。下面是两个无效地址表达式的例子。

```
register int x;
struct t {int x : 3;} t;
int * px = & x, * pt = & t.x;
```

其中，表达式&x 和&t.x 都是非法的。

一元运算符&的结果是操作数 E 的地址，但不是左值。如果 E 的类型是 T，则地址表达式的结果类型是“指向 T 的指针”。具体来说，如果 E 是函数指示符，则&E 的结果是指向函数的指针；若 E 是对象类型，则&E 的结果是指向对象的指针。例如：

```
# include <stdio.h>

void f (void)
{
    char c, * p = & c;                                    //E1
    int (* pf) (const char * restrict, ...) = & printf;   //E2
    (* pf) ("0x%p\n", (void *) & "hello,world!" [0]);
    char a [5], (* pa) [5] = & a;                         //E3
}
```

在上例的 E1 中，c 的类型是 char，故&c 的类型是指向 char 的指针（char *），&c 的值可以赋给 p，因为它们类型相同；E2 中，printf 是标准库函数，&printf 是指向 printf 函数的指针，其类型和 pf 相同，可以赋值；E3 中，a 的类型是数组（char [5]），故&a 的类型是指向数组的指针（char (*) [5]），与 pa 的类型相同，可以赋值。

pf 解引用之后是函数指示符，%p 表示要输出一个地址，在这里是字面串被创建为静态数组后第一个元素的地址。此代码实际上可以简单地写成

```
pf ("0x%p\n", (void *) "hello,world!");
```

原因是字面串在程序转换阶段用于初始化一个静态数组，而这个数组在这里被转换为指向其首元素的指针，可参见“字面串”和“数组-指针转换”。

如果 E 是一个间接表达式（参见“间接”），或者换句话说，如果一个地址表达式的形式为&*X，那么*和&运算符都不被求值，就像它们被忽略一样（即，&*X 相当于 X），但这两

个运算符的操作数依然要符合各自的规定（也就是说，表达式&*22 依然是不合法的，因为*的操作数应为指针类型，但22是整型常量），且表达式的结果依然不是左值。

请看下面的示例：

```
int f (int *);
f (& * (int *) 0xF000E000));    //E1

int i = 0;
(& * i) ++;                     //E2

int x = 0, y = 0, * px = & x;
& * px = & y;                   //E3
```

上例中，E1 没有问题，传递给函数 f 的是指针值(int *) 0xF000E000，唯一的问题在于这个指针是否指向一个有效的对象；E2 是无效的，尽管&*i 相当于 i，但 i 不是指针类型的操作数，违反了要求；E3 将一个指针值（y 的地址）赋给另一个指针&*px，但这是无效的，因为，尽管名义上&*px 相当于 px，但&*px 的结果不是一个左值，赋值运算符=的左操作数必须是左值。

如果一元运算符&的操作数是运算符[]的结果，如&E1[E2]，由于表达式 E1[E2]相当于*((E1)+(E2))，所以表达式&(E1[E2])又相当于&(*((E1)+(E2)))。于是根据前面所讲述的内容，表达式&(*((E1)+(E2)))相当于*和&运算符被忽略，即相当于((E1)+(E2))，但不是左值。请看下面的示例：

```
char * c = & "chinese." [0];    //& "chinese." [0]相当于"chinese." + 0
```

下例中，可以将 E1 中的表达式&ar[2]替换为 ar+2 或者&*(ar+2)，尽管&ar[2]不是左值，但*(&ar[2])指示一个对象，是左值；可以将 E2 中的&ar[3]替换为&*(ar+3)或者 ar+3。

```
int ar [5] = {1,2,3,4,5,}, * par;
*(& ar [2]) = 7;        //E1
par = & ar [3];         //E2
```

6.14.4 间接

间接表达式表达式属于一元表达式。

一个“*”加上一个表达式 E，即可构成间接表达式。其中，E 是一元表达式及之前的表达式（参见词条“表达式列表”。如果要使用其他表达式，则可以用括号将它变为基本表达式），所以 E 本身也可能是另一个一元表达式。

一元运算符*指示间接引用，其操作数 E 要求是一个指针。下面这个例子是非法的，因为22是整型常量，不是指针类型。

```
int i = * 22;
```

相反，下面的例子是有效的，在 E1 里，p 是一个指针，*p 解除引用；在 E2 中，a 自动转换为指向数组第一个元素的指针，运算符*将其解除引用。

```
int i, * p = & i, a [3];
* p = 0;    //E1
```

```
* a = 1;     //E2
```

一元运算符*的结果取决于其操作数E的类型。如果E是指向函数的指针，则结果是函数指示符；如果E是指向对象的指针，则结果是左值，指示该对象；如果E的类型是“指向类型T的指针”，则结果的类型是T。

在下面的示例中，pi是指向int类型对象的指针，所以在语句S中，表达式*pi的结果是int类型的左值。

```
int i = 1, * pi = & i;
(* pi) ++;       //S
```

下例中，pj的类型是const int *，按上述规则，在语句S中，*pj的类型是const int，是一个左值，但不是可修改的左值。故此语句非法。

```
const int j = 0, * pj = & j;
* pj = 1;       //S
```

下例中，对于表达式*a=3，a自动转换为指向其第一个元素的指针。因其第一个元素也是数组（int [5]），故*a指示一个数组，对一个数组赋值是非法的。相反，对于表达式*a[0] = 3，[]运算符的优先级比一元运算符*高，所以该表达式等效于*(a[0])=3。表达式 a[0]的结果是数组类型（int [5]），继续转换为指向其第一个元素的指针，所以表达式*a[0]指示数组元素a[0][0]，该表达式是合法的。

```
int a [3] [5];
* a = 3;        //非法
* a [0] = 3;    //合法，即*(a[0]) = 3
```

下例中，pa的类型是指向数组的指针。因此，表达式*pa的类型是数组（int [5]）。进一步地，函数f的返回值是数组的大小，这个大小等于sizeof (int)×5。

```
# include <stddef.h>

size_t f (int (* pa) [5])
{
     return sizeof (* pa);
}
```

E不能是无效指针。比如说，E可能是空指针，E所指向的对象可能过了生存期；E的值是一个对象的地址，但这个地址相对于该对象的类型来说，不是正确对齐的（参见“对齐”），凡此种种都被视为无效指针，对无效指针解引用的行为是未定义的。

在下面这个例子中，因为pi是一个空指针，所以向printf函数传递参数*pi的行为是未定义的。

```
# include <stdio.h>

void f (void)
{
    int * pi = 0;
    printf ("%d\n", * pi); //行为未定义
```

```
}
```

在下面这个例子中，联合对象 u 必然是按 int 类型的要求对齐的，因为 char 类型的对象可以对齐于任何地址。在这种情况下，&u.a[1]所得到的地址肯定不符合 int 类型的对齐要求。因此，表达式*(int *)&u.a[1]的行为就是未定义的，有些计算机架构要求严格对齐的访问，在这些机器上会导致总线错误。

```
union {char a [sizeof (int) + 1]; int i;} u;
* (int *) & u . a [1] = 0;
```

在下例中的块内，pc 指向一个有效的对象（c 所指示的对象，或者对象 c）。离开 c 所在的块后，pc 具有无效值，表达式*pc 的行为是未定义的。

```
char * pc;
{
    char c = 0;
    pc = & c;
    /* ..... */
}
* pc = 68;
```

如果 X 是运算符&的有效操作数，则*&X 相当于 X。有关运算符&的内容，参见词条“地址”。

下例中，E1 是合法的，*&y 相当于 y，这是一个左值；E2 也是合法的，表达式&*px 相当于 px 且是一个对象的指针。

```
int x = 0, y = 0, * px = & x;
* & y = 3;          //E1
* (& * px) = 1;     //E2
```

6.14.5 正号

正号表达式属于一元表达式。

一个“+”加上一个表达式 E，即可构成正号表达式。其中，E 是一元表达式及之前的表达式（参见词条“表达式列表”。如果要使用其他表达式，则可以用括号将它变为基本表达式），所以 E 本身也可能是另一个一元表达式。

下面是正号表达式的简单示例。

```
int x = + 5, y = + x;
```

操作数 E 要求是算术类型；正号表达式（或者一元运算符+）的结果不是左值；若 E 是整数类型，则表达式的结果是 E 整型提升后的值，结果的类型为提升后的类型；否则，结果的值是 E 的值，结果的类型和 E 的类型相同。

下面的例子可以证实，c 的类型是 char，但表达式+c 的类型是 char 提升后的类型。这个例子使用了泛型选择表达式，用于得到表达式 c 和+c 的类型；printf 函数的第一个参数是格式串，但由于太长，无法写在一行内，这里将它分成两个字面串。在程序转换阶段，C 实现将它们黏接成一个字面串，这方面的信息可参见“字面串”。

```
# include <stdio.h>
```

```
void f (char c)
{
   printf ("type of c:%s\ntype of +c:%s\n" \
           "size of c:%zu\nsize of +c:%zu\n", \
           _Generic (c, \
                     char: "char", \
                     int: "int", \
                     default: "Unknown"), \
           _Generic (+ c, \
                     char: "char", \
                     int: "int", \
                     default: "Unknown"), \
           sizeof c, \
           sizeof + c
          );
}
```

正号表达式等价于0+E，因此，结果的符号和操作数E的符号一样。例如：

```
# include <stdio.h>

void f (void)
{
    signed char c = - 1;    //S1
    printf ("%d\n", + c);   //S2
}
```

尽管表达式+c在求值时，对c做了整型提升（提升为int类型），但仅仅是提升后的宽度发生了变化而已，符号没有改变。最后，表达式+c在求值后得到的结果依然同c原来的符号性保持一致。语句S1和S2可以合并为一行，即

```
printf ("%d\n", + (signed char) - 1);
```

6.14.6 负号

负号表达式属于一元表达式。

一个“-”加上一个表达式E，即可构成负号表达式。其中，E是一元表达式及之前的表达式（参见词条“表达式列表”。如果要使用其他表达式，则可以用括号将它变为基本表达式），所以E本身也可能是另一个一元表达式。

下面是负号表达式的简单示例。

```
int x = - 5, y = - x;
```

操作数E要求是算术类型；负号表达式（或者一元运算符-）的结果不是左值；若E是整数类型，则表达式的结果是E整型提升后的负值，结果的类型为提升后的类型；否则，结果

的值是E的负值，结果的类型和E的类型相同。

下面的例子可以证实，c 的类型是 char，但表达式-c 的类型是 char 提升后的类型。这个例子使用了泛型选择表达式，用于得到表达式 c 和-c 的类型；printf 函数的第一个参数是格式串，但由于太长，无法写在一行内，这里将它分成两个字面串。在程序转换阶段，C 实现将它们黏接成一个字面串，这方面的信息可参见“字面串”。

```
# include <stdio.h>

void f (char c)
{
   printf ("type of c:%s\ntype of -c:%s\n" \
           "size of c:%zu\nsize of -c:%zu\n", \
           _Generic (c, \
                     char: "char", \
                     int: "int", \
                     default: "Unknown"), \
           _Generic (- c, \
                     char: "char", \
                     int: "int", \
                     default: "Unknown"), \
           sizeof c, \
           sizeof - c
          );
}
```

整型提升对于整数之外的类型（比如浮点和指针类型）不起作用，正号和负号不会改变一个浮点数的类型。例如：

```
# include <stdio.h>

void f (void)
{
   printf ("type of 1.0f:%s\ntype of -1.0f:%s\n" \
           "size of 1.0f:%zu\nsize of -1.0f:%zu\n", \
           _Generic (1.0f, \
                     float: "float", \
                     double: "double", \
                     default: "Unknown"), \
           _Generic (+ 1.0f, \
                     float: "float", \
                     double: "double", \
                     default: "Unknown"), \
```

```
                sizeof 1.0f, \
                sizeof - 1.0f
               );
}
```

负号表达式等价于0-E，因此，结果的符号和操作数E的符号相反。例如：

```
# include <stdio.h>

void f (void)
{
    signed char c = - 1;
    printf ("%d\n", - c);
    printf ("%lld\n", - (long long) 3);
}
```

尽管表达式-c在求值时，对c做了整型提升（提升为int类型），但仅仅是提升后的宽度发生了变化而已，符号没有改变。但是，表达式-c在求值后得到的结果不同于c原来的符号性。同样，long long int类型的值3作为一元运算符-的操作数求值后，也变为负数。

6.14.7 按位反

按位反表达式属于一元表达式。

一个“~”加上一个表达式E，即可构成按位反表达式。其中，E是一元表达式及之前的表达式（参见词条“表达式列表”。如果要使用其他表达式，则可以用括号将它变为基本表达式），所以E本身也可能是另一个一元表达式。

下面是按位反表达式的简单示例。

```
int x = ~ 5, y = ~ x;
```

操作数E要求是整数类型；按位反表达式（或者一元运算符~）的结果不是左值；表达式的结果是E整型提升后，逐位取反（原来的比特是0的，变为1；原来的比特是1的，变为0）后得到的值，结果的类型为提升后的类型。

下面的例子可以证实，c的类型是char，但表达式~c的类型是char提升后的类型。这个例子使用了泛型选择表达式，用于得到表达式c和~c的类型；printf函数的第一个参数是格式串，但由于太长，无法写在一行内，这里将它分成两个字面串。在程序转换阶段，C实现将它们黏接成一个字面串，这方面的信息可参见“字面串”。

```
# include <stdio.h>

void f (char c)
{
    printf ("type of c:%s\ntype of ~c:%s\n" \
            "size of c:%zu\nsize of ~c:%zu\n", \
            _Generic (c, \
                      char: "char", \
```

```
                int: "int", \
                default: "Unknown"), \
        _Generic (~ c, \
                char: "char", \
                int: "int", \
                default: "Unknown"), \
        sizeof c, \
        sizeof ~ c
        );
}
```

如果 E 经整型提升后是无符号整数类型，则表达式~E 等价于用该无符号类型的最大值减去 E。例如

```
# include <stdio.h>
# include <limits.h>

int f (void)
{
    unsigned x = 1;
    printf ("%x, %x\n", ~ x, UINT_MAX - x);
    return ~ 1UL == ULONG_MAX - 1UL;
}
```

上例中，x 提升后的类型（依然）是 unsigned int，该类型的最大值是 UINT_MAX（它是一个宏，在头文件<limits.h>中定义）。也就是说，表达式~x 等价于 UINT_MAX-x；在上例最后一行中，1UL 的类型是 unsigned long int，这种类型的最大值是 ULONG_MAX。最终，运算符==的结果是 1，所以函数 f 将返回 1。

注意，有符号整数的表示方法因 C 实现不同而异，故~运算符的操作数若为有符号整数，则结果可能无法移植。

6.14.8 逻辑非

逻辑非表达式属于一元表达式。

一个“!”加上一个表达式 E，即可构成逻辑非表达式。其中，E 是一元表达式及之前的表达式（参见词条“表达式列表”。如果要使用其他表达式，则可以用括号将它变为基本表达式），所以 E 本身也可能是另一个一元表达式。

下面是逻辑非表达式的简单示例。

```
int x = ! 5, y = ! x;
```

操作数 E 要求是标量；逻辑非表达式（或者一元运算符!）的结果不是左值；逻辑非表达式求值时，要把操作数 E 的值和 0 相比较。如果等于 0，则表达式的值为 1；如果不等于 0，表达式的值是 0。也就是说，!E 等效于表达式 E==0（参见词条“等性表达式”）。

在下面的例子中，将检查 p 的值，若其为 0，则表示无效指针，所以先输出警示信息，再

直接返回调用者。

```
# include <stdio.h>

void f (int * p)
{
   if (! p) {printf("Invalid pointer.\n"); return;}
   /* ..... */
}
```

不管操作数的类型如何，逻辑非表达式（或者一元运算符!）的结果类型是 int。

下面的例子可以证实，c 的类型是 char，但表达式!c 的类型是 int；尽管整型常量 5LL 的类型是 long long int，但表达式!5LL 的类型依然是 int。这个例子使用了泛型选择表达式，用于得到表达式 c、!c、5LL 和!5LL 的类型；printf 函数的第一个参数是格式串，但由于太长，无法写在一行内，这里将它分成两个字面串。在程序转换阶段，C 实现将它们黏接成一个字面串，这方面的信息可参见“字面串”。

```
# include <stdio.h>

void f (char c)
{
   printf ("type of c:%s\ntype of !c:%s\n" \
           "size of c:%zu\nsize of !c:%zu\n", \
           _Generic (c, \
                     char: "char", \
                     int: "int", \
                     default: "Unknown"), \
           _Generic (! c, \
                     char: "char", \
                     int: "int", \
                     default: "Unknown"), \
           sizeof c, \
           sizeof ! c
          );

   printf ("\n");

   printf ("type of 5LL:%s\ntype of !5LL:%s\n" \
           "size of 5LL:%zu\nsize of !5LL:%zu\n", \
           _Generic (5LL, \
                     int: "int", \
                     long long int: "long long int", \
```

```
                    default: "Unknown"), \
            _Generic (! 5LL, \
                    int: "int", \
                    long long int: "long long int", \
                    default: "Unknown"), \
            sizeof 5LL, \
            sizeof ! 5LL
           );
}
```

6.14.9 尺寸

尺寸表达式属于一元表达式。

尺寸表达式有两种形式：运算符 sizeof 加上一个表达式 E，或者加上一个用括号括住的类型名 T，即

```
sizeof E
sizeof (T)
```

其中，E 是一元表达式及之前的表达式（参见词条“表达式列表”。如果要使用其他表达式，则可以用括号将它变为基本表达式），所以 E 本身也可能是另一个一元表达式。

下面是尺寸表达式的简单示例。

```
sizeof 0.5
sizeof (long long int)
```

有些人习惯在任何时候都为 sizeof 的操作数加上括号，例如：

```
int a, b = sizeof (a);
```

在这种情况下，因为 a 并不是类型名，所以，(a)是基本表达式，即它是一个括住的表达式。如果要使用那些位于一元表达式之后的表达式（参见词条“表达式列表”），则必须用括号使之成为基本表达式。如果不注意，很容易出现问题，例如：

```
sizeof 5 * 3
sizeof x ++
```

此时，因为运算符*的优先级比 sizeof 低，所以，第一个表达式等价于

```
(sizeof 5) * 3
```

相反，因为后缀运算符++的优先级比 sizeof 高，所以，第二个表达式等价于

```
sizeof (x ++)
```

多数运算符以标点符号出现，虽然这种对比和反差使得 sizeof 看起来更像是函数或者宏，但它们是运算符。

sizeof 运算符的结果不是左值，它的值是操作数的尺寸，以字节计。而且，这个大小和操作数的类型有直接关系。

考虑到可移植性，sizeof 的结果类型被定义为 size_t，这是一种无符号整数类型，在<stddef.h>及其他头文件中定义。

size_t 可能对应于 unsigned int，也可能对应于 unsigned long long int，或者对应于其他无

符号整数类型，到底是哪一种，取决于 C 实现，但要求能够表示所有对象的大小，而不管这些对象是何种类型。

下面的例子用于计算数组的大小，其中就用到了 sizeof 运算符。

```
# include <stdio.h>

void f (void)
{
    typedef int array [5];
    array a, b [5];
    printf ("sizeof (array) = %zu\nsizeof a = %zu\n \
                sizeof b = %zu\nsizeof (int) = %zu.\n", \
                sizeof (array), sizeof a, \
                sizeof b, sizeof (int));
}
```

值得再次强调的是，sizeof 运算符的结果类型是一种无符号整数类型，可以被转换为其他类型，只要新的类型能够表示 sizeof 的结果。例如：

```
unsigned long long int m = sizeof (int);
```

但是，如果处理不当，程序中就会出现隐患，代码的执行过程将和你预期的不同，而你也得不到正确的结果。下面是一个例子，假定 a 是一个任意类型的数组：

```
int i = -1;
while (i ++ < sizeof a / sizeof a [0]) a [i] = i;
```

在这里，表达式

```
sizeof a / sizeof a [0]
```

的结果是无符号整数类型，因为运算符/的两个操作数都是无符号整数类型 size_t。但是，运算符<的左操作数 i ++的结果类型却是 int。因为类型不同，所以它们必须进行常规算术转换（参见“常规算术转换”），转换后的类型也是无符号整数类型。

while 语句第一次执行时，表达式 i++的结果是-1，类型是 int，经常规算术转换，-1 被转换为无符号整数类型，且得到该无符号整数类型的最大值，这导致

```
i ++ < sizeof a / sizeof a [0]
```

始终不成立，循环体不会执行。

不管 sizeof 运算符的操作数是表达式 E 还是类型名 T，它们都不能是函数类型，也不能是不完整类型。特别地，E 不能是指示位字段的表达式。

因此，下面的做法都是不合法的，main 是函数类型；void 是不完整类型，而 t.i 指示一个位字段。

```
# include <stdio.h>

void f (void)
{
    struct t {int i : 3;} t;
```

```
    printf ("%zu,%zu",%zu, sizeof main, sizeof (void), sizeof t.i);
}
```

有人认为 sizeof 的结果在程序转换期间就已经得到，但事实并非总是如此。如果 sizeof 运算符的操作数是变长数组，则只能在程序运行期间先求值那个用于指定数组大小的操作数，然后才能得到 sizeof 的结果，且结果不是常量；相反地，如果 sizeof 运算符的操作数不是变长数组，则不求值该操作数且 sizeof 运算符的结果是一个在程序转换期间就能得到的整型常量。

在下面的例子中，sizeof运算符的操作数是否求值，已在它们旁边的注释中做了说明：

```
# include <stdio.h>

void siz_demo (void)
{
    int n = 1, f (int);

    /* 以下语句不求值n++，因为在编译阶段可分析出n++的结果是int类型 */
    printf ("%zu\n", sizeof (n ++));

    /* 以下语句不求值（调用）函数f(n)，因为sizeof的操作数是一个指针 */
    printf ("%zu\n", sizeof (int (*) [f (n)]));

    /* 以下语句不调用函数f，因为只需要查看它的返回类型 */
    printf ("%zu\n", sizeof f (n));

    printf ("%d\n", n);      //n的当前值为1

    /* 以下语句将求值++n，因为只有这样才知道数组的大小 */
    printf ("%zu\n", sizeof (int [++ n]));

    printf ("%d\n", n);      //n的当前值为2
}
```

下面是另外一个例子，说明 sizeof 运算符并非总在程序转换期间求值，很可能会在程序执行期间求值，前提是涉及变长数组。

```
# include <stddef.h>

size_t fs (int n)
{
    char buf [n];
    return sizeof buf;      //在执行时才求值
}
```

值得注意的是，char、signed char 和 unsigned char 及它们的限定版本在任何 C 实现上都是一个字节的长度。因此，当 E 的类型或者 T 所指定的类型是 char、signed char、unsigned char 或者它们的限定版本时，sizeof 运算符的结果为整型常量 1。

也就是说，给定声明

```
char c;
signed char sc;
unsigned char uc;
```

则表达式 sizeof c、sizeof sc、sizeof uc、sizeof (char)、sizeof (signed char)和 sizeof (unsigned char)的结果都是 1。

如果 E 的类型或者 T 所指定的类型是数组，则 sizeof 运算符的结果是该数组的总字节数（元素类型的长度乘以元素的数量）。下例中，表达式 sizeof b 的结果是数组 b 的大小，在数值上等于 50×sizeof (size_t)。这个示例的真正价值在于，虽然函数的参数 a 是以数组的形式出现，但它会被调整为指向元素类型的指针。因此，a 的实际类型是指向 char 的指针（char *），而表达式 sizeof a 的结果不是数组的大小，而是指针的大小 sizeof (char *)：

```
# include <stddef.h>

size_t f (char a [5])
{
    size_t b [50], c = sizeof b;
    /* ····· */
    return sizeof a;
}
```

可以利用 sizeof 表达式来计算数组的元素数量。下例中，因为 arr 是数组，故 sizeof arr 的结果是整个数组的尺寸，以字节计；而 arr[0]指示数组 arr 的第 1 个元素，sizeof arr[0]的结果是数组元素类型的尺寸。

```
char arr [256];
size_t n = sizeof arr / sizeof arr [0];
```

如果 E 是函数的形参，且 E 的类型是数组或者函数，则会被调整为指向数组第一个元素的指针或者指向函数的指针。在这种情况下，sizeofE 的结果是指针类型的大小。

在下面的例子中，函数 f 的第一个参数实际上是指向 const int 的指针（const int *）；第二个参数实际上是指向函数的指针（void (*) (int)）。自然地，当它们作为 sizeof 的操作数时，得到的是指针的大小。

```
# include <stdio.h>

void f (const int a [], void g (int))
{
    printf ("%zu,%zu", sizeof a, sizeof g);
}
```

如果 E 的类型或者 T 所指定的类型是结构或者联合，则结果是结构或联合类型所需要占

用的总字节数，包括内部和尾部的填充。

在下例中，为对象 p 申请的存储空间可能大于结构成员 data 和 next 的大小之和，对象 node 的大小也是如此。

```
# include <stdio.h>
# include <stdlib.h>

typedef struct stgNODE {char data; \
                        struct stgNODE * next;} NODE, * PNODE;

void f (void)
{
   PNODE p = (PNODE) malloc (sizeof (struct stgNODE));
   NODE node;
   printf ("%zu, %zu, %zu\n", sizeof * p, sizeof node, \
                              sizeof node.data + sizeof node.next);
}
```

6.14.10 对齐

对齐表达式属于一元表达式。

运算符（关键字）_Alignof 加上一个用括号括住的类型名，即可构成一个对齐表达式。下面是对齐表达式的简单示例。

```
_Alignof (int)
```

需要特别注意的是，_Alignof 运算符的操作数不能是表达式而只能是类型名。因此，下面的做法是错误的。

```
_Alignof (32)
_Alignof (x ++)
```

_Alignof 运算符的结果不是左值，值的大小取决于操作数（类型名）所指定的类型，是这个类型的对齐要求。

考虑到可移植性，_Alignof 的结果类型被定义为 size_t，这是一种无符号整数类型，在<stddef.h>及其他头文件中定义。有关 size_t 类型的完整说明，参见“尺寸”。例子：

```
# include <stddef.h>

struct t {char c; float f;};

size_t f (void)
{
   return _Alignof (struct t);
}
```

_Alignof 的操作数（类型名）所指定的类型不能是函数，也不能是不完整类型。C 实现

不求值该操作数，因为它只需要（提取）操作数的类型信息即可。

如果_Alignof 的操作数（类型名）指定的类型是数组，则结果并不是数组本身的对齐要求，而是其元素类型的对齐要求，这一点务必注意。_Alignof 运算符（对齐表达式）的结果是一个整型常量。

下面是一个示例，该例子表明，由于_Alignof 的操作数只能是类型名，所以它不被求值。

```
# include <stddef.h>
# include <stdio.h>

void f (int n)
{
    /* 以下语句输出打印的是数组元素类型的对齐要求，而且++n 不被求值 */
    printf ("%zu\n", _Alignof (int [++ n]));
    printf ("%d\n", n);     //n 的值没有变化
}
```

关键字_Alignof 是从 C11 开始引入的。

6.15 转型表达式

有时，类型转换可以隐式地进行。例如，将一个 signed int 类型的值赋给一个 unsigned int 类型的左值。但是，在另一些场合，类型转换不会安静地发生，而是需要以明确的方式指示。

每个值有其自己的类型，同时也有其独特的对象表示，而且值的对象表示受制于值的类型。本质上，转型表达式的作用是将一个值的对象表示解释为另一种类型，将其视为另一种类型的值所具有的对象表示（参见“对象表示”）。

一个用括号括住的类型名，加上一个表达式 E，即可构成转型（cast）表达式。其中，E 是转型表达式及之前的表达式（参见词条“表达式列表”。如果要使用其他表达式，则可以用括号将它变为基本表达式），所以 E 本身也可能是另一个转型表达式。

转型表达式的作用是将 E 的值转换为另一种类型，即类型名所指定的类型。值得注意的是，指针类型和浮点类型不能互转。

下面是两个转型表达式的例子，第一个例子将浮点常量 2.5f 转换为 int 类型的值；第二个例子将 pv 指向的对象转型为结构类型的对象。

```
# include <stdlib.h>

void f (void)
{
    int i = (int) 2.5f;     //将浮点数 2.5 强制转换为 int 类型
    struct t {/* …… */};
    void * pv = (void *) malloc (sizeof (struct t));
    struct t * tp = (struct t *) pv;
```

```
}
```

因为不允许在指针和浮点类型之间互转，所以，下例中，将 double 类型的值 2.0 转换为指向 void 的指针是不对的。

```
void f (void * pv);
f ((void *) 2.0);
```

因为类型名后面的表达式也可以是另一个转型表达式，所以下面的转型表达式（字体加粗的部分）也是合法的。

```
int i = (int) (char) 'x';
```

转型表达式是右结合的。所以，上述表达式等价于以下字体加粗的部分：

```
int i = (int) ((char) 'x');
```

转型表达式中的类型名所指定的类型只能是 void 或者标量。如果是标量，则它还可以是原子的、限定或无限定的。操作数（表达式）E 的类型要求是标量。

下面的例子给出了一些转型表达式，有些是合法的，有些是不合法的，代码中已经用注释做了说明。

```
struct t {int a; char c;} t, * pt;
char * p = (char *) t;        //非法，t 的类型不是标量
char * q = (char *) pt;       //合法，从一种指针类型转换为另一种指针类型
```

下面的例子很有说明问题的价值。对象 a 的类型是 const int，它的值不允许修改。但是，可以用一元运算符&获取它的地址，这将得到一个 const int *类型的指针。通过转型操作将这个指针的类型转换为 int *。这样 C 实现就不会阻止用户通过这个新指针来修改 a 的存储值，但这种行为是未定义的。

```
const int a = 10;
* (int *) & a = 9;            //这是能够办到的，但是很可怕，很要命
```

再来看一个相反的例子，pv 的类型是指向 void 的指针（void *），它可以在 E1 中转换为指向 int 类型的指针（int *），并访问这个指针所指向的对象。但是，在 E2 中，pv 的值被转换为指向 const int 的指针（const int *），这时不能通过这个新指针来修改对象的存储值。

```
void * pv = & (int) {0};
* (int *) pv = 1;              //E1: 合法
* (const int *) pv = 2;        //E2: 非法
```

下面这段代码演示了转型表达式在底层硬件访问中的应用。

```
# define BIOS ((const volatile unsigned long * const) 0xfffffff0)
* BIOS = 33;                   //非法，企图修改一个只读对象
int x = * BIOS;
```

本章的前面已经介绍过下标（[]）、间接（*）和地址（&）运算符，现在又介绍了转型表达式，下面是一个综合性的示例，可以从另一个角度来重新认识这些运算符：

```
# include <stdio.h>

void f (void)
{
```

```
    int a [5] [6] = {[3][2] = 6};
    int (* p) [4] = (int (*) [4]) & a;
    printf ("%d\n", (* (p + 5)) [0]);
}
```

以上，我们先是声明了一个数组 a，它带有一个初始化器，并将它的元素 a[3][2]初始化为 6，于是，其他元素都初始化为 0（参见“初始化”和“初始化器”）。

a 是一个两维数组，我们令指针 p 指向这个数组。但是，p 是指向数组 int[4]的指针，而不是指向 int[5]的指针。换句话说，尽管 p 的确是指向 a 的，但它指向的是另一种数组类型。p 的类型是 int (*) [4]，&a 的类型是 int (*) [5][6]，p 的类型和&a 的类型不同，后者不能直接用来初始化前者，所在我们使用了转型表达式，也就是上例中字体加粗的部分。

那么，如何用 p 来访问 a 的元素 a[3][2]呢？

如图 6-2 所示，因为 p 是指向 int[4]的指针，所以 p+5 的结果是一个新的指针，这个新的指针与 p 相比，中间间隔了 5 个 int[4]这样的数组（参见“加性表达式”）。

进一步地，因为 p+5 的结果类型是指向数组 int[4]的指针，所以，*(p+5)的结果是数组 int[4]。从图中可以看出，(*(p+5))[0]的结果是数组元素，这个元素其实就是数组 a 的元素 a[3][2]。

最后，printf 函数输出的结果是 6。

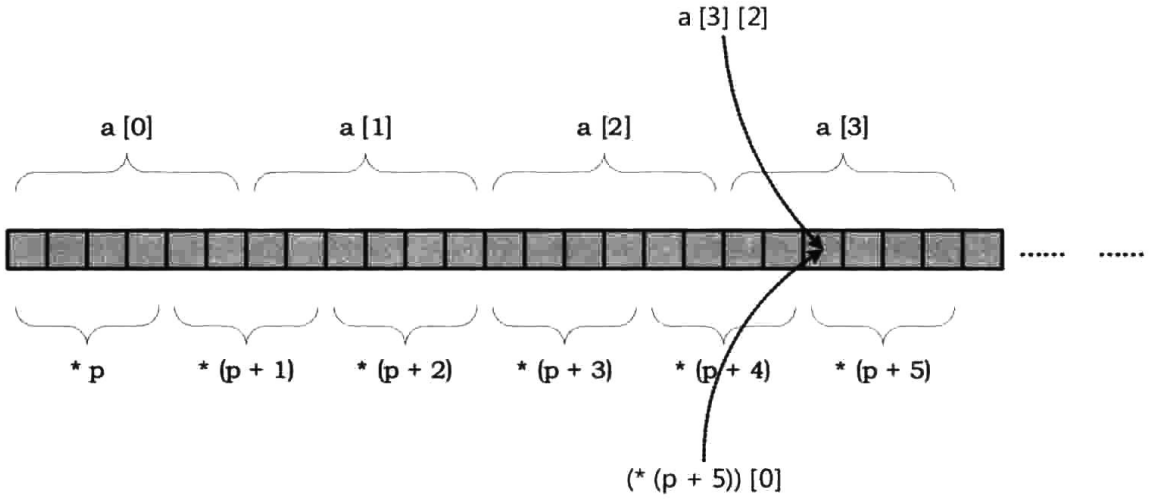


图 6-2　以不同类型来观察同 一个数组时的情况

正如用户可能已经知道的，将一个表达式的值从它原来的类型强制转换为 void 类型表示显式地丢弃值：

```
(void) printf ("hello,world.\n");  //显式地丢弃函数调用的返回值
```

转型表达式的结果不是左值。

如果对一个表达式转型的结果和转型前相同，包括值和类型，则实际的转换过程可能并不会发生。考虑下面的例子。

```
float f = 1.5678, g;
g = (float) f;                    //E1

struct s {int i; float f;};
```

```
struct s s0 = {1, 1.0f};

struct s s1 = (struct s) s0;    //E2

struct t {int j; float g;};
struct t t0 = (struct t) s0;    //E3: 非法
```

其中，E1 中的转型操作是合法的，但实际的转换可能并不会发生，因为 f 的类型和它在转型后相比没有变化；E3 是非法的，原因在于转型表达式中的类型名不能指定结构类型；但这里指定了结构类型。而且最重要的是，s0 和类型和 t0 的类型并不相同；E2 同样非法，但很多 C 实现却不会报告错误，在它们看来，因为 s0 和 s1 的类型都是 struct s，所以不会进行转型操作，直接将 s0 的值赋给 s1。

转型操作和隐式转换的规则相同。在下面的例子中，char 类型可能等价于 unsigned char，也可能等价于 signed char。如果情况属于前者，则根据相关规则，char 类型的值总是可以用 unsigned 类型来表示，所以转换后的值不变，c 和 u 所指示的对象具有相同的值。

```
char c = /* ····· */;
unsigned u = (unsigned) c;
```

相反，如果 char 类型等价于 signed char，则 c 和 u 所指示的对象是否具有相同的值，要视 c 的值而定。如果 c 所指示的对象拥有正值或者 0，则转换后的值也是不变的；如果 c 所指示的对象拥有负值，则它无法用 unsigned 类型来表示，所以根据相关规则，需要用这个负值重复加上或者减去比 unsigned 类型的最大值大 1 的数，直到结果能用 unsigned 类型来表示。假设 c 的值是-1，unsigned 的最大值是 4294967295，那么转型后的结果就是-1+4294967296，即 4294967295。

当然，如果要将 char 类型的值转换成 unsigned char 类型的值，然后存储在 u 所指示的对象中，则可以这样做：

```
unsigned u = (unsigned char) c;
```

或者

```
unsigned u = (unsigned)(unsigned char) c;
```

6.16 乘性表达式

乘性表达式包括乘法（*）表达式、除法（/）表达式和取余（%）表达式，详情可参见其各自的词条。

组成乘性表达式语法形式的那些运算符（见上）称为乘性运算符，乘性运算符是左结合的。因此，表达式

```
a * b / c % d
```

等价于

```
((a * b) / c) % d
```

类似于做除法的验算，商乘以乘数加余数等于被除数，如果 a/b 的商是可以表示的，则表

达式(a/b)*b + a%b 应等于 a（运算符%的作用是取余数）；否则，a/b 和 a%b 的行为是未定义的。

例如，对于很多 C 实现来说，int 类型的最小值 INT_MIN 在头文件<limits.h>中定义为

```
#define INT_MIN (-INT_MAX-1)
```

这就是说，在这些 C 实现上，INT_MAX 的绝对值比 INT_MIN 的绝对值小 1。进一步地，INT_MIN/−1 的商（INT_MAX+1）会溢出。因此，表达式(INT_MIN/-1) + INT_MIN%-1 的行为是未定义的。

6.16.1 乘法

乘法表达式属于乘性表达式。

一个表达式 E1，加上一个“*”和另一个表达式 E2，即可构成乘法表达式。其中，E1 是乘性表达式及之前的表达式，E2 是转型表达式及之前的表达式。显然，E1 本身也可能是另一个乘性表达式。参见词条“表达式列表”，如果想让 E1 和 E2 也能是排在后面的那些表达式，则只需用括号将它们变为基本表达式。

下面是几个乘法表达式的例子。

```
int a = 20 * 3, b = a * 10;
```

E1 和 E2 是二元运算符*的两个操作数，乘法表达式的结果是 E1 和 E2 的乘积。E1 和 E2 都要求是算术类型，在相乘之前要先进行常规算术转换。乘法表达式的结果也是算术类型，但不是左值。

在下例中，操作数(char)5 要先转换为 long double，再进行乘法计算，结果也是 long double 类型。最后，再将 long double 类型的结果转换为 unsigned int 类型后，赋值给对象 m。

```
unsigned m = sizeof ((char) 5 * (long double) 3.0);
```

6.16.2 除法

除法表达式属于乘性表达式。

一个表达式 E1，加上一个“/”和另一个表达式 E2，即可构成除法表达式。其中，E1 是乘性表达式及之前的表达式，E2 是转型表达式及之前的表达式。显然，E1 本身也可能是另一个乘性表达式。参见词条“表达式列表”，如果想让 E1 和 E2 也能是排在后面的那些表达式，则只需用括号将它们变为基本表达式。

下面是几个除法表达式的例子。

```
int a = 100 / 5, b = a / 10;
```

E1 和 E2 是二元运算符/的两个操作数，除法表达式的结果是 E1 除以 E2 的商。E1 和 E2 都要求是算术类型，在相除之前要先进行常规算术转换。如果 E2 的值为 0，则程序的行为是未定义的。除法表达式的结果也是算术类型，但不是左值。

当 E1 和 E2 都是整数类型时，除法表达式的结果是丢弃小数部分的代数商（通常称为趋零截尾）。这意味着，表达式 100/30 的值是 3。

6.16.3 取余

取余表达式属于乘性表达式。

一个表达式 E1，加上一个“%”和另一个表达式 E2，即可构成取余表达式。其中，E1 是乘性表达式及之前的表达式，E2 是转型表达式及之前的表达式。显然，E1 本身也可能是另一个乘性表达式。参见词条“表达式列表”，如果想让 E1 和 E2 也能是排在后面的那些表达式，则只需用括号将它们变为基本表达式。

下面是几个取余表达式的例子。

```
int a = 100 % 5, b = a % 10;
```

E1 和 E2 是二元运算符%的两个操作数，取余表达式的结果是 E1 除以 E2 后的余数。E1 和 E2 都要求是算术类型，在运算之前要先进行常规算术转换。如果 E2 的值为 0，则程序的行为是未定义的。取余表达式的结果是整数类型，但不是左值。举例来说，表达式 100%30 的值是 10。

6.17 加性表达式

加性表达式包括加法（+）表达式和减法（-）表达式，详情可参见其各自的词条。组成加性表达式语法形式的运算符称为加性运算符，加性运算符是左结合的。

这就是说，表达式

```
a + b - c - d
```

等价于

```
((a + b) - c) - d
```

6.17.1 加法

加法表达式属于加性表达式。

一个表达式 E1，加上一个“+”和另一个表达式 E2，即可构成加法表达式。其中，E1 是加性表达式及之前的表达式，E2 是乘性表达式及之前的表达式。显然，E1 本身也可能是另一个加性表达式。参见词条“表达式列表”，如果想让 E1 和 E2 也能是排在后面的那些表达式，则只需用括号将它们变为基本表达式。

下面是几个加法表达式的例子：

```
int a = 100 + 5, b = a + 10, c = a + b;
```

E1 和 E2 是二元运算符+的两个操作数，加法表达式的结果是操作数 E1 加上 E2 所得到的和，但不是左值。E1 和 E2 的类型要求是以下几种情形之一。

（1）同为算术类型。

（2）一个是指向完整对象类型的指针，另一个是整数类型。

如果 E1 和 E2 都是算术类型，则相加之前，会先各自进行常规算术转换，表达式的结果也是算术类型（E1 和 E2 的公共类型）。

例如，如果对象 a 和 b 的类型都是 char，则表达式 a+b 的结果类型是 int。再如，下例中浮点常量 2.5f 的类型是 float，需要先转换为 double 类型，表达式 2.5f+d 的结果是 double 类型（还需要进一步转换为 long double 才能赋给 ld）。

9200LL 是 long long int 类型的常量，c 需要先转换为 long long 类型，表达式 9200LL+c 的结果类型是 long long， printf 函数将输出两个相同的长度数值。

```
# include <stdio.h>

void f (char c, double d)
{
    long double ld = 2.5f + d;
    printf("%zu, %zu\n", sizeof (long long), sizeof (9200LL + c));
}
```

如果 E1 和 E2 中的一个是指针，另一个是整数类型，则表达式 E1+E2 的结果也是指针，并且和那个指针操作数的类型相同。这种指针运算的意义在于，如果指针操作数指向数组的第 e 个元素，那么在数组足够大的情况下，将它与另一个整数类型的操作数 n 相加，得到的新指针将指向同一数组的第 e+n 个元素。

先来看一个简单的例子，给定声明

```
char a [] = "Android", * p = a;
```

则表达式 p + 3 的结果也是一个指针，指向数组 a 的第 4 个元素“r”。

如果 E1 和 E2 中的一个是指针，但不是指向数组的元素，而是指向其他完整对象类型 T 的指针，则它的行为也如同指向数组首元素的指针，该数组只有一个元素，且元素的类型是 T。

下面的例子有助于理解这一点。

```
int i = 10086;
char * p = (char *) & i;
printf ("%x, %x, %p, %p\n", * (p + 0), * (p + 1), \
                            (void *) (p + 0), (void *) (p + 1));
```

其中，对象 i 并不是一个数组，对象 p 是一个指向 char 的指针，它指向的其实是对象 i 的起始位置。在这里，p 并非指向数组元素，但如同指向一个数组的首元素。因此，表达式 p+0 和 p+1 可以指向这个“数组”的第 1 个和第 2 个元素。当然，被指向的这些“元素”实际上是组成对象 i 的各个字节单元。

下面是另一个例子：

```
char * a [10] , * * p;
p = a;          //S1
p = a + 1;      //S2
p = p + 1;      //S3
```

以上，a 是数组，其元素类型为指针（char *），p 是指向指针的指针（char * *）。在 S1 中，数组 a 自动转换为指向其首元素的指针。因其元素类型是 char *，故转换后的指针是 char * * 类型，可以将这个指针类型的值赋给 p，p 指向数组 a 的第一个元素 a[0]。

在 S2 中，数组 a 自动转换为指向其首元素（a[0]）的指针，被指向的元素不是数组，但可视为数组，且表达式 a+1 的结果是指向其下一个元素的指针，这个元素实际上对应于数组 a 的元素 a[1]；

同理，在S3中，指针p已经在S2中被修改为指向数组元素a[1]，虽然p指向的是数组a的元素，并不是指向数组，但视为指向数组，且表达式 p+1 的结果是指向其下一个元素的指针，这个元素实际上对应于数组a的元素a[2]。

如果 P 是指向数组最后一个元素的指针，则 P+1 的结果仍然是指针，指向数组最后一个元素的下一个位置。但是，该指针不能是一元运算符*的操作数，否则行为是未定义的。

下例中，q指向数组最后一个元素的下一个位置，这是允许的，而且可以用于和另一个指针进行比较，但不能对它解引用，这会导致未定义行为。

```
int i, * p = & i, * q = p + 1;
if (p < q)              //合法
    * q = 1033;         //未定义行为
```

尽管“指向数组最后一个元素的下一个位置”的指针在解引用时是无效的，但 C 支持的这种机制在其他情况下是安全和有用的，比如它使遍历数组变得更简单：

```
# define N 20
int ar [N], * p;
for (p = & ar [0]; p < & ar [N]; p ++) {/* …… */}
```

6.17.2 减法

减法表达式属于加性表达式。

一个表达式 E1 加上一个“-”和另一个表达式 E2，即可构成减法表达式。其中，E1 是加性表达式及之前的表达式，E2 是乘性表达式及之前的表达式。显然，E1 本身也可能是另一个加性表达式。参见词条“表达式列表”，如果想让 E1 和 E2 也能是排在后面的那些表达式，则只需用括号将它们变为基本表达式。

下面是几个减法表达式的例子。

```
int a [3] = {1, 2, 3,}, * p;
p = & a [1] - 1;
a [0] = a [2] - a [1];
```

减法表达式的结果是操作数 E1 减去 E2 所得到的差值，但不是左值。E1 和 E2 的类型要求是以下几种情形之一。

（1）同为算术类型。

（2）一个是指向完整对象类型的指针，另一个是整数类型。

（3）同为指向完整对象类型的指针。在这种情况下，操作数 E1 和 E2 的类型被视为是无限定的（限定符被忽略，如果有的话）。

如果 E1 和 E2 都是算术类型，则相减之前，会先各自进行常规算术转换，表达式的结果也是算术类型。

例如，如果对象 a 和 b 的类型都是 char，则表达式 a-b 的结果类型是 int。再如，在下面的例子中，浮点常量 2.5f 的类型是 float，需要先转换为 double 类型，表达式 2.5f+d 的结果是 double 类型（还需要进一步转换为 long double 才能赋给 ld）。

9200LL 是 long long int 类型的常量，c 需要先转换为 long long 类型，表达式 9200LL-c 的结果类型是 long long，printf 函数将输出两个相同的长度数值。

```
# include <stdio.h>

void f (char c, double d)
{
    long double ld = 12.5f - d;
    printf ("%zu, %zu\n", sizeof (long long), sizeof (9200LL - c));
}
```

如果 E1 是指针，E2 是整数类型，则 E1-E2 的结果也是指针，结果的类型和 E1 的类型相同。这种指针运算的意义在于，如果指针操作数指向数组的第 e 个元素，那么在数组足够大的情况下，将它与另一个整数类型的操作数 n 相减，得到的新指针将指向同一数组的第 e-n 个元素。

即使减性运算符-的指针操作数不是指向数组的元素，而是指向其他完整对象类型 T 的指针，它的行为也如同指向数组首元素的指针，该数组只有一个元素，且元素的类型是 T。

下面的例子有助于理解这一点：

```
int i, * p = & i;
p [0] = 3;
```

下面是另一个例子，因为 p 指向数组 a 的第 3 个元素，将指针 p 减去 1，意味着使 p 指向数组的第 2 个元素。

```
int a [3] = {1, 2, 3,}, * p = & a [2];
* (p - 1) = 0;
```

如果 E1 和 E2 都是指针，则它们应指向同一个数组的元素。特别地，允许它们指向数组最后一个元素的下一个位置。

两个指针相减的意义在于，可以得到（由两个指针所指向的）数组元素之间的下标差值。这种运算的结果大小取决于 C 实现，其类型（一种有符号整数类型）为 ptrdiff_t，是在头文件<stddef.h>中定义的。

指针 E1 和指针 E2 相减的结果必须能够用 ptrdiff_t 类型表示，否则程序的行为是未定义的。

如下例如示，假定在某个 C 实现中，stddef.h 文件有如下定义。在这种情况下，两个指针相减的结果不能大于 LONG_MAX（一般来说这种情况是不可能出现的）。

```
# ifndef __PTRDIFF_TYPE__
# define __PTRDIFF_TYPE__ long int
# endif
typedef __PTRDIFF_TYPE__ ptrdiff_t;
```

再如，给定声明：

```
int a [10], * p = a, * q = & a [5];
```

则表达式 q-p 的值是 5。注意，下标的差值就是间隔的元素数量，和它们间隔的字节数是不能混淆的。这里，下标的差值是 5，但是它们之间间隔了 sizeof(int)×5 个字节。

6.18 移位表达式

移位操作使得 C 语言看起来更像低级语言。学习过汇编语言的人应该懂得移位操作的价值，这种低层次的处理能力显然拓展了 C 的应用领域，在很长一段时间里，C 一直是操作系统、设备驱动程序和嵌入式程序开发的首选语言。

移位表达式包括左移（<<）表达式和右移（>>）表达式，详情可参见其各自的词条。下面是几个移位表达式的例子。

```
int x = 0 << 5, y = x >> 3 << 2;
```

组成移位表达式语法形式的运算符称为移位运算符；移位运算符是左结合的。因此，表达式

```
a >> b >> c << d
```

等价于

```
((a >> b) >> c) << d
```

假定 int 类型的宽度是 32 位，则以下代码可以取出 0x7abcf3e5 中间的 8 位作为 u 的值。因为<<和>>运算符是左结合的，故先计算(unsigned)0x7abcf3e5<<12（在此之前，先将表达式 12 的类型提升为 unsigned），结果是 unsigned 类型的值 0xcf3e5000。然后将它右移 24 位，得到 unsigned 类型的值 0x000000cf。

```
unsigned u = (unsigned) 0x7abcf3e5 << 12 >> 24;
```

6.18.1 左移

左移表达式属于移位表达式。

一个表达式 E1，加上一个“<<”和另一个表达式 E2，即可构成左移表达式。其中，E1 是移位表达式及之前的表达式，E2 是加性表达式及之前的表达式。显然，E1 本身也可能是另一个移位表达式。参见词条“表达式列表”，如果想让 E1 和 E2 也能是排在后面的那些表达式，则只需用括号将它们变为基本表达式。

下面是一个左移表达式的例子，它先将 1 左移 10 次，然后将这个左移的结果再继续左移 6 次。

```
1 << 10 << 6
```

就功能和作用而言，二元运算符<<将其左操作数 E1 向左逐位移动，移动的次数由其右操作数 E2 指定。这意味着，如果 E2 的值为 0，则不移位。例如：

```
unsigned x = 5 << 3, y = x << 0;          //x 和 y 的值都是 40
```

E1 和 E2 都要求是整数类型，在逐位左移前要分别执行整型提升。以左操作数 E1 提升后的宽度为准，移出左边界之外的位被丢弃，右边空出来的位用 0 填充。

若 E1 是无符号整数类型，则 E1<<E2 的结果是(E1) × 2^{E2} 和 M 的模，M 是比“E1 提升后的类型所容许的最大值”大 1 的数。

左移表达式（运算符<<）的结果不是左值，结果的类型是 E1 提升后的类型。

例如，给定表达式：

```
1 << 2
```

因为 1 和 2 的类型都是 int，结果类型也是 int。假定某一个 C 实现将 INT_MAX 定义为 32767，则表达式 1<<2 的结果等于（1× 2^2）% 32768，即 4。

在下面的例子中，printf 函数用于输出表达式 c 和 c<<1 的类型（名），以及类型的大小。对于表达式 c<<1，1 是 int 类型的常量，因此，c 先被提升为 int 类型。运算符<<的结果也是 int 类型。这可以，通过由 printf 函数输出的类型的大小看出来。

```
# include <stdio.h>

void f (void)
{
    unsigned char c = 128;
    printf ("type of c:%s\ntype of c << 1:%s\n" \
            "size of c:%zu\nsize of (c << 1):%zu\n", \
            _Generic (c, \
                      unsigned char: "unsigned char", \
                      int: "int", \
                      default: "Unknown"), \
            _Generic (c << 1, \
                      unsigned char: "unsigned char", \
                      int: "int", \
                      default: "Unknown"), \
            sizeof c, \
            sizeof (c << 1)
           );
}
```

注意，如果右操作数 E2 为负值，或者该值所指示的移动次数大于等于左操作数 E1 提升后的宽度，行为是未定义的。

换句话说，移位操作的次数要求控制在合理的范围内。另外，有些读者认为右操作数为负能够改变位移的方向，这是错误的。

给定以下声明（宏 CHAR_BIT 是在头文件<limits.h>中定义的）：

```
char c = -1;
int n = sizeof (int) * CHAR_BIT;
```

那么，表达式 c << n 的行为是未定义的。

这里，因为 c 要从 char 类型提升为 int 类型，n 的值是 int 类型的宽度，它等于<<运算符左操作数提升后的宽度，所以行为是未定义的。

如果 char 等价于 signed char，那么 c 的值为负，即使 n 的值小于 int 类型的宽度，行为也是未定义的。

若 E1 是有符号整数类型且具有非负值，同时(E1) × 2^{E2} 的值能用结果类型来表示，则它

就是结果值，否则，行为是未定义的。

这意味着，如果 int 类型的最大值 INT_MAX 是 32767，那么表达式 1 << 15 的结果是 2^{15}，即 32768，不能用该表达式的结果类型（int）来表示，该表达式的行为是未定义的。

6.18.2 右移

右移表达式属于移位表达式。

一个表达式 E1 加上一个“>>”和另一个表达式 E2，即可构成右移表达式。其中，E1 是移位表达式及之前的表达式，E2 是加性表达式及之前的表达式。显然，E1 本身也可能是另一个移位表达式。参见词条“表达式列表”，如果想让 E1 和 E2 也能是排在后面的那些表达式，则只需用括号将它们变为基本表达式。

下面是一个右移表达式的例子，它先将 1 右移 3 次，在此基础上再右移 6 次（实际上等于移动 9 次）。

```
1 >> 3 >> 6
```

就功能和作用而言，二元运算符>>将其左操作数 E1 向右逐位移动，移动的次数由其右操作数 E2 指定。这意味着，如果 E2 的值为 0，则不移位。例如：

```
unsigned x = 8 >> 3, y = x >> 0;          //x 和 y 的值都是 1
```

E1 和 E2 都要求是整数类型，在逐位右移前要分别执行整型提升。以左操作数 E1 提升后的宽度为准，移出右边界之外的位被丢弃。至于左边空出来的位该如何填充，则视 E1 的值和类型而定。

若 E1 属于无符号类型，或者是有符号类型且具有非负值，则 E1>>E2 的值是 E1 除以 2^{E2} 所得商的整数部分。

例如，8>>2 的结果是 8 除以 2^2 的商，即 2；5>>2 的结果是 5 除以 2^2 的商，即 1，余数不要（从二进制的形式来看，组成数值 5 的对象表示的一些比特在移位的过程中被删除了）。

若 E1 是有符号类型且其值为负，则右移表达式的结果值取决于 C 实现。具体来说，若 E1 的值为负，那么，对于左边空出来的位，有些 C 实现用 0 填充，而另一些 C 实现可能用符号位来填充。

右移表达式（运算符>>）的结果不是左值，结果的类型和 E1 提升后的类型相同。

在下面的例子中，printf 函数用于输出表达式 c 和 c>>1 的类型（名），以及类型的大小。对于表达式 c>>1，1 是 int 类型的常量，因此，c 先被提升为 int 类型。运算符>>的结果也是 int 类型。printf 函数将输出两个相同的长度数值。

```
# include <stdio.h>

void f (void)
{
   unsigned char c = 128;
   printf ("type of c:%s\ntype of c >> 1:%s\n" \
           "size of c:%zu\nsize of (c >> 1):%zu\n", \
           _Generic (c, \
                     unsigned char: "unsigned char", \
```

```
                    int: "int", \
                    default: "Unknown"), \
            _Generic (c >> 1, \
                    unsigned char: "unsigned char", \
                    int: "int", \
                    default: "Unknown"), \
            sizeof c, \
            sizeof (c >> 1)
           );
}
```

注意，如果右操作数 E2 为负值，或者该值所指示的移动次数大于等于左操作数 E1 提升后的宽度，则行为是未定义的。

换句话说，移位操作的次数要求控制在合理的范围内。另外，有些读者认为右操作数为负能够改变位移的方向，这是错误的。

给定以下声明（宏 CHAR_BIT 是在头文件<limits.h>中定义的）：

```
char c = -1;
int n = sizeof (int) * CHAR_BIT;
```

那么，表达式 c >> n 的行为是未定义的。

这里，因为 c 要从 char 类型提升为 int 类型，n 的值是 int 类型的宽度，它等于<<运算符左操作数提升后的宽度，所以行为是未定义的。

如果 char 等价于 signed char，那么 c 的值为负，即使 n 的值小于 int 类型的宽度，行为也是未定义的。

6.19 关系表达式

一个表达式 E1，加上一个关系运算符和另一个表达式 E2，即可构成关系表达式。关系运算符可以是>、>=、<、<=中的任何一个，分别表示比较时的大于、大于等于、小于和小于等于关系。因此，关系表达式有以下 4 种形式：

```
E1 > E2
E1 >= E2
E1 < E2
E1 <= E2
```

其中，E1 是关系表达式及之前的表达式，E2 是移位表达式及之前的表达式。显然，E1 本身也可能是另一个关系表达式。参见词条“表达式列表”，如果想让 E1 和 E2 也能是排在后面的那些表达式，则只需用括号将它们变为基本表达式。

关系表达式（关系运算符）的结果不是左值。如果相应的关系成立，则结果是 1；如果不成立，则结果是 0。

关系表达式的结果类型是 int。

下面的示例输出了两个关系表达式 3ULL > 5ULL 和 5 * 8 <= 20 * 2 的结果类型，以及结果类型的大小。

```
# include <stdio.h>

void f (void)
{
   printf ("type of 3ULL>5ULL:%s\ntype of 5*8<=20*2:%s\n" \
           "size of 3ULL>5ULL:%zu\nsize of 5*8<=20*2:%zu\n", \
           _Generic (3ULL > 5ULL, \
                     int: "int", \
                     default: "Unknown"), \
           _Generic (5*8<=20*2, \
                     int: "int", \
                     default: "Unknown"), \
           sizeof (3ULL>5ULL), \
           sizeof (5*8<=20*2)
          );
}
```

操作数 E1 和 E2 的类型应为以下两种情况之一。

（1）都是实数类型；

（2）都是指针，但要求它们所指向的类型在去掉限定符后是兼容的。

不管 E1 和 E2 的类型如何，它们的类型或者它们所指向的类型，如果是有限定符的，则限定符被忽略，因为限定符对比较操作来说没有意义。

在下例中，p 是 const 限定的指针，q 不是限定的指针，但它们依然可以比较；x 是 const 限定的类型，y 的类型则没有限定符，但它们也可以比较。

```
void fptrcmp (void * const p, const void * q)
{
   if (p < q) {/* ····· */}
   const int x = 0;
   int y;
   /* ····· */
   if (x >= y) {/* ····· */}
}
```

当操作数 E1 和 E2 都是算术类型时，先要执行常规算术转换，使它们具有一致的类型。但是，这影响不到结果的类型，结果的类型依然是 int。

请看下面的例子，因为表达式(unsigned)0 的类型是 unsigned int，所以-1 要从 int 提升为 unsigned int，提升后的值是 unsigned int 类型的最大值，其值远大于 0，所以表达式-1>=(unsigned)0 的结果为 1，但结果的类型和这一切没有关系，总是 int，这从该语句的输出上可以看出来。

```
# include <stdio.h>
```

```
void f (void)
{
   printf ("type of (-1 >= (unsigned) 0):%s\n" \
           "size of (-1 >= (unsigned) 0):%zu\n" \
           "result of (-1 >= (unsigned) 0):%d\n", \
           _Generic ((-1 >= (unsigned) 0), \
                     int: "int", \
                     unsigned int: "unsigned int", \
                     default: "Unknown"), \
           sizeof (-1 >= (unsigned) 0), \
           -1 >= (unsigned) 0 \
          );
}
```

如果 E1 和 E2 都是指针，关系表达式的结果依赖于它们所指向的对象在地址空间中的相对位置，现分述如下。

（1）若 E1 和 E2 都指向同一个对象，则比较时，E1 和 E2 相等。指向同一个对象的情形包括指向同一个数组对象、指向同一个结构对象、指向同一个联合对象，以及指向同一个数组对象的同一个元素、指向同一个结构对象的同一个成员、指向同一个联合对象的同一成员，等等。

下例中，px 和 qx 指向同一个对象；pt 和 qt 指向同一个对象；pa 和 qa 指向同一个对象，pu 和 qu 也指向同一个对象。最终，if 语句的控制表达式求值的结果是 1：

```
int x, * px = & x, * qx = & x;
struct t {int m;} t, * pt = & t, * qt = & t;
int a [3], (* pa) [3] = & a, (* qa) [3] = & a;
union u {char c; float f;} u, * pu = & u, * qu = & u;
if (px >= qx && pt <= qt && pa >= qa && pu <= qu) {/* …… */}
```

同样地，在下例中，pt 和 qt 指向同一个对象；pa 和 qa 指向同一个对象，pu 和 qu 也指向同一个对象。

```
struct t {int m;} t;
int * pt = & t.m, * qt = & t.m;

int a [3], * pa = & a [1], * qa = & a [1];
union u {char c; float f;} u;
char * pu = & u.c, * qu = & u.c;
```

（2）若 E1 和 E2 都指向数组元素，且这两个元素属于同一数组对象，则指向较大下标值的那个在比较时大于指向较小下标值的那个。

下例中，指针 p 指向的元素比指针 q 指向的元素的下标值大，故表达式 p>=q 所表示的关系是成立的。

```
int a [3], * p = & a [2], * q = & a [0];
```

```
if (p >= q) {/* …… */}
```

下面这个示例用于将字符串中的内容反转。例如，若原始字符串为"abcde"，则反转之后的内容是"edcba"。这个示例之所以能够工作，就是因为可以比较两个指向数组元素的指针。函数 rev_str 接受一个指向源字符串的指针 str，将其反转后，返回一个指向结果串的指针：

```
char * rev_str (char * str)
{
    char c, * p = str, * q = str;

    while (* p != ' \0') p ++;
    if (p -- == str) return str;
    do
        c = * str, * str = * p, * p = c;
    while (-- p > ++ str);

    return q;
}
```

（3）若 E1 和 E2 都指向同一数组的最后一个元素的下一个位置，则比较时，E1 和 E2 相等。

下例中，关系运算符>=的两个操作数++pa 和++qa 都指向数组最后一个元素的下一个位置，它们在比较时是相等的。

```
int a [3], * p = & a [2], * q = & a [2];
if (++ p >= ++ q) {/* …… */}
```

（4）若 E1 和 E2 中的一个指向数组的某个元素，另一个指向同一数组的最后一个元素的下一个位置，则比较时，后者大于前者。

下例中，指针 p 指向数组第一个元素（数组 a 自动转换为指向其首元素的指针），加上 3 后，得到的新指针指向数组最后一个元素的下一个位置，因此新指针大于指针 p。

```
int a [3], * p = a;
if (p + 3 >= p) {/* …… */}
```

（5）若 E1 和 E2 都指向结构成员，且这两个成员都属于同一结构对象，一个指向较晚声明的成员而另一个指向较早声明的成员，则比较时前者大于后者。

在下例中，成员 f 的声明较 m 晚，因此，指向前者的指针大于指向后者的指针。

```
struct t {int m; float f;} t;
if ((void *) & t.f >= (void *) & t.m) {/* …… */}
```

（6）如果 E1 和 E2 都指向联合成员，且这两个成员都属于同一个联合对象，那么在比较时，E1 和 E2 相等。

联合总是有其特殊性。在下例中，不管在声明时谁早谁晚，指向成员 c 的指针必须和指向成员 f 的指针相等。

```
union u {char c; float f;} u;
if ((void *) & u.f >= (void *) & u.c) {/* …… */}
```

除以上所述的几点外，其他情况下的指针比较行为都是未定义的。

两个指针做关系比较，其目的是确定它们的相对位置，没有这个前提，有些代码也许可以工作，但很难说有什么实际意义。例如：

```
int x = 0, y = 0;
if (& x > & y) {/* ..... */}
```

上述代码的结果是难以预料的，每次执行这段代码时，对象 x 和 y 在存储器中的位置谁前谁后都是随机的，C 实现可能不会根据这两个标识符在声明时的出现顺序来安排。

为了使指针的比较更为方便，即使 E1 和 E2 不是指向数组元素，而是指向其他完整对象类型 T，它的行为也如同指向一个数组的首元素，该数组长度为 1（只有一个元素）的数组的第一个元素，且该数组的，且元素类型为 T。

因为此原因，再结合上一段的描述，下面的大于等于表达式是合法的，而且大于等于关系是成立的。

```
int * p = & (int) {0};
if (p >= p + 1) {/* ..... */}
```

组成关系表达式语法形式的运算符称为关系运算符；关系运算符是左结合的。因此，表达式

```
a > b >= c < d
```

等价于

```
((a > b) >= c) < d
```

注意，表达式 a<b<=c 并不意味着 b 大于 a，并且 b 小于等于 c，而是等同于(a<b)<=c，即先比较 a 和 b，将比较的结果（0 或者 1）再和 c 进行比较，才能得到整个表达式的值。

因此，如果要确保 a 的值小于 b，b 的值小于等于 c，则应该使用表达式 a<b && b<=c。有关运算符&&，可参见词条“逻辑与表达式”。

6.20 等性表达式

一个表达式 E1，加上一个等性运算符和另一个表达式 E2，即可构成等性表达式。等性运算符可以是==或者!=，分别表示比较时的等于和不等于关系。因此，等性表达式有以下两种形式：

```
E1 == E2
E1 != E2
```

其中，E1 是等性表达式及之前的表达式，E2 是关系表达式及之前的表达式。显然，E1 本身也可能是另一个等性表达式。参见词条“表达式列表”，如果想让 E1 和 E2 也能是排在后面的那些表达式，则只需用括号将它们变为基本表达式。

等性表达式（等性运算符）的结果不是左值。如果相应的关系成立，则结果是 1；如果不成立，则结果是 0；等性表达式的结果类型是 int。

下面的示例演示了等性表达式的结果。

```
# include <stdio.h>

void f (void)
{
    printf ("%d, %d\n", 3 != 5, 5 * 8 == 20 * 2);
}
```

操作数E1和E2的类型应为以下几种情形之一。

（1）都是算术类型。

（2）都是指针，它们指向的类型在去掉限定符后是兼容的。

（3）都是指针，一个是指向对象类型，另一个指向限定或无限定的void。

（4）都是指针，一个指向任意类型，另一个是空指针常量。

对于上述任何一对操作数，==和!=这两种关系总有一个是成立的。

若E1和E2都是算术类型，则在比较前，要先进行常规算术转换，使它们具有一致的类型。具体内容可参见“常规算术转换”。但是，这不影响结果的类型，结果的类型总是int。

下面的例子用于输出表达式-1 == UINT_MAX的类型及类型的大小。这里，int类型的常量-1需要先转换为unsigned int，转换后的结果是unsigned int类型的最大值UINT_MAX，所以这个比较的结果是1，结果的类型是int。

```
# include <stdio.h>
# include <limits.h>

void f (void)
{
   printf ("type of (-1 == UINT_MAX):%s\n" \
           "size of (-1 == UINT_MAX):%zu\n" \
           "result of (-1 == UINT_MAX):%d\n", \
           _Generic ((-1 == UINT_MAX), \
                     int: "int", \
                     unsigned int: "unsigned int", \
                     default: "Unknown"), \
           sizeof (-1 == UINT_MAX), \
           -1 == UINT_MAX \
          );
}
```

若E1和E2都是复数，则它们的实部和虚部要分别进行比较。E1等于E2的前提是它们的实部和虚部分别是相等的。

在下例中，两个复数的实部和虚部分别相等，因此这两个复数也是相等的。

```
if (2.0 + 3.0i == 2.0 + 3.0i) {/* …… */}
```

若E1和E2一个是实数，另一个是复数，那么在比较时，要先将实数转换为复数，再按前面的规则进行比较。

在下例中，将f和c1、c2进行相等性比较，需要先将f的值转换为double _Complex类型，最后的结果是前者为0，后者为1。

```
# include <stdio.h>

int main (void)
{
    return printf("%d, %d\n", 2.0f == 2.0 + 3.0i, 2.0f == 2.0 + 0.0i);
}
```

若E1和E2都是指针且其中有一个是空指针常量（参见“空指针常量”），则先要将空指针常量转换为另一个指针操作数的类型，即转换为空指针，再进行比较。比较的结果参见后面的叙述。

下例中，表达式p==0等价于p==(void*)0或者p==NULL（宏NULL在头文件<stddef.h>和其他头文件中被定义为空指针常量），它需要先转换为p的类型，即转换为int*，再做比较。如果p是空指针，则下面的比较是相等的。

```
int * p = /* …… */;
if (p == 0) {/* …… */}
```

若E1和E2有一个是指向对象类型的指针，而另一个是指向限定或未限定版本的void的指针，则要将前者的类型转换为后者的类型，再进行比较。

因此，如下例所示，pi是指向int类型的指针；pv是指向void的指针，它们相比较前，要将pi的类型从int *转换为void *。

```
_Bool f (int * pi, void * pv) {return pi == pv;}
```

若E1和E2都是指针类型，且符合以下情形之一，则它们在比较时是相等的。

（1）都指向同一个对象。

（2）都指向同一个函数。

（3）都是空指针或者空指针常量。

（4）都指向同一数组最后一个元素的后一个位置。

（5）E1和E2中的一个指向数组A最后一个元素的下一个位置，另一个指向数组B的首元素，且这两个数组以A在前，B在后的方式直接相邻。直接相邻意味着它们之间没有任何间隔，比如下标值之差为1的两个数组元素是直接相邻的。

可以灵活理解“指向同一个对象”的意思。例如，如果两个指针都指向同一个结构对象的同一个成员，它们被视为指向同一个对象；再如，一个指针指向某个数组对象，另一个指针指向同一数组的首元素，那么，这两个指针的类型经适当转换后可以被视为指向同一个对象。实际上，指向结构对象的指针和指向同一结构对象第一个成员的指针也可以按此办理。下面是一个例子。

```
struct t {int i; float f;} t;
if ((void *) & t == & t.i) {/* …… */}
```

因为任何指针都可以转换为指向void的指针且不会丢失任何信息，而且，等性运算符允许任何指针和指向void的指针进行比较，所以以上代码可以正常工作。实际上，因为&t和&t.i都是指向对象类型的指针，将哪一个转换为void *类型是无所谓的。因此此表达式也可以

写成：

```
& t == (void *) & t.i
```

下例中，main 和&main 是相同的，因为函数指示符作为运算符==的操作数时，被转换为指向函数的指针，因此表达式 main==&main 的值为 1。

```
# include <stdio.h>

void f (void)
{
   struct s {int i; char c;} sa [8];
   printf("%d, %d, %d\n", main == & main,\
          & sa [1] + 1 == & sa [2],\
          (void *)(& sa [0].i + 1) == & sa [0].c);
}
```

再来看表达式&sa[1]+1==&sa[2]，&sa[1]和&sa[2]的类型都是 struct s *，即指向 struct s 类型的指针。

很重要的一点是，为了方便等性比较，对于表达式 E1 == E2 和 E1 != E2，如果 E1 和 E2 是指向类型为 T 的、非数组对象的指针，则这两个表达式的行为就如同 E1 和 E2 是指向数组首元素的指针，且该数组只有 1 个元素，元素的类型为 T。。

因此，上例中，&sa[1]也可以看做指向数组首元素的指针，该数组只有一个元素，且元素的类型是 struct s，而&sa[1]+1 可以被认为是指向该数组最后一个元素的下一个元素。考虑到&sa[2]也可以被看做指向数组首元素的指针，且这两个数组直接紧邻，因此，&sa[1]+1==&sa[2]的值也是 1。

表达式&sa[0].i 的类型是指向 int 的指针，即 int *，它可以看做指向数组首元素的指针，该数组只有一个元素，且元素的类型是 int，而&sa[0].i+1 则可以被认为是指向该数组最后一个元素的后一个元素。考虑到&sa[0].c 也可以被看成是指向数组首元素的指针，且这两个数组直接紧邻（char 类型的对象可以对齐于任何字节地址，因此，结构类型 struct s 的成员 i 和成员 c 之间不存在任何填充），所以，它们之间的相等性比较是成立的。注意，在本例中，表达式&sa[0].i + 1 和表达式&sa[0].c 具有不同的类型，应当将它们中的一个转换为指向 void 的指针，在这里是(void *)(&sa[0].i + 1)。

组成等性表达式语法形式的运算符==和!=称为等性运算符；等性运算符==和!=是左结合的。因此，表达式 a==b==3 并不意味着 a、b 都等于 3，相反，它意味着(a==b)==3。因为 a==b 的结果可能是 0，也可能是 1，因此，表达式 a==b==3 的结果是 0。如果希望判断 a 和 b 是否都等于 3，则应该使用 a==3&&b==3。

6.21 按位与表达式

一个表达式 E1，加上一个“&”运算符和另一个表达式 E2，即可构成按位与表达式。其中，E1 是按位与表达式及之前的表达式，E2 是等性表达式及之前的表达式。显然，E1 本身

也可能是另一个按位与表达式。参见词条“表达式列表”，如果想让 E1 和 E2 也能是排在后面的那些表达式，则只需用括号将它们变为基本表达式。

操作数 E1 和 E2 都要求是整数类型，而且要先进行常规算术转换，以取得一致的宽度。然后，将前一个步骤得到的结果逐位做与（AND）操作。也就是说，两个对应的比特，当且仅当它们都是“1”时，结果中的对应比特才是“1”。

按位与表达式（运算符&）的结果不是左值，结果的类型是 E1 和 E2 经常规算术转换后得到的公共类型。因此，表达式 3 & (long long) 5)的结果是 1；结果类型是 long long int。

下面的例子假定用于检测硬件端口的最低位是否被置 1。*pport 的类型是 long long int，故 int 类型的常量 1 必须先提升为 long long int，二元运算符&的结果也是 long long int。

```
const volatile long long * pport = (volatile long long *) 0xFFFFFF00;
if (* pport & 1) {/* …… */}
```

组成按位与表达式语法形式的运算符&称为按位与运算符；按位与运算符&是左结合的。因此，表达式

```
a & b & c & 5
```

等价于

```
((a & b) & c) & 5
```

因为这个原因，表达式 a&0&b 等价于(a&0)&b，结果是 0。

6.22 按位异或表达式

一个表达式 E1，加上一个“^”运算符和另一个表达式 E2，即可构成按位异或表达式。其中，E1 是按位异或表达式及之前的表达式，E2 是按位与表达式及之前的表达式。显然，E1 本身也可能是另一个按位异或表达式。参见词条“表达式列表”，如果想让 E1 和 E2 也能是排在后面的那些表达式，则只需用括号将它们变为基本表达式。

操作数 E1 和 E2 都要求是整数类型，而且要先进行常规算术转换，以取得一致的宽度。然后，将前一个步骤得到的结果逐位做异或（XOR）操作。

也就是说，如果对应的两个比特是相反的，一个为“0”而另一个为“1”，则结果值中对应的比特是“1”，否则为“0”。

例如，表达式 0^0 的结果是 0；0^1 的结果是 1；0^5 的结果是 5；3^2 的结果是 1。

按位异或表达式（运算符^）的结果不是左值，结果的类型是 E1 和 E2 经常规算术转换后得到的公共类型。因此，表达式

```
3 ^ (long long) 5)
```

的结果是 6；结果类型是 long long int。

组成按位异或表达式语法形式的运算符^称为按位异或运算符；按位异或运算符^是左结合的。因此，表达式

```
a ^ b ^ c ^ 5
```

等价于

```
((a ^ b) ^ c) ^ 5
```

6.23 按位或表达式

一个表达式 E1，加上一个“|”运算符和另一个表达式 E2，即可构成按位或表达式。其中，E1 是按位或表达式及之前的表达式，E2 是按位异或表达式及之前的表达式。显然，E1 本身也可能是另一个按位或表达式。参见词条“表达式列表”，如果想让 E1 和 E2 也能是排在后面的那些表达式，则只需用括号将它们变为基本表达式。

操作数 E1 和 E2 都要求是整数类型，而且要先进行常规算术转换，以取得一致的宽度。然后，将前一个步骤得到的结果逐位做或（OR）操作。

也就是说，如果对应的两个比特都是“0”，则结果中对应的比特就是“0”，在任何其他情况下，结果中对应的比特都是“1”。

例如，表达式 0|0 的结果是 0；0|1 的结果是 1；5|6 的结果是 7；5|2 的结果是 7。

按位或表达式（运算符|）的结果不是左值，结果的类型是 E1 和 E2 经常规算术转换后得到的公共类型。

因此，表达式

```
3 | (long long) 5)
```

的结果是 7；结果类型是 long long int。

组成按位或表达式语法形式的运算符|称为按位或运算符；按位或运算符|是左结合的。因此，表达式

```
a | b | c | 5
```

等价于

```
((a | b) | c) | 5
```

6.24 逻辑与表达式

一个表达式 E1，加上一个“&&”运算符和另一个表达式 E2，即可构成逻辑与表达式。其中，E1 是逻辑与表达式及之前的表达式，E2 是按位或表达式及之前的表达式。显然，E1 本身也可能是另一个逻辑与表达式。参见词条“表达式列表”，如果想让 E1 和 E2 也能是排在后面的那些表达式，则只需用括号将它们变为基本表达式。

逻辑与表达式执行的是逻辑意义上的操作。操作数 E1 和 E2 都要求是标量，当且仅当 E1 和 E2 求值的结果都不为 0 时，逻辑与表达式的结果是 1；在其他任何情况下，结果是 0。逻辑与表达式的结果类型是 int，不是左值。

因此，表达式 0&&0 的结果是 0；3&&5 的结果是 1；8&&0 的结果也是 0。

逻辑与表达式（运算符&&）保证是从左到右求值的。这意味着，如果 E1 和 E2 都会求值的话，则 E1 的值计算和副作用将保证前序于 E2 的值计算和副作用，E1 和 E2 的求值之间有一个序列点。

事实上，逻辑与表达式求值时，总是先求值 E1，如果 E1 的值为 0，则对 E2 求值已无必要，因此不求值 E2，而且整个逻辑与表达式（运算符&&）结果为 0。

请看下面的例子，在数学上，1/0 是不合法的。但是，因为&&运算符的左操作数为 0，

所以右操作数不被求值，整个表达式是合法的，值为0。

```
if (0 && 1/0) {/* …… */}
```

因为序列点的存在，下例中，表达式x++&&x--不存在未定义行为：

```
int x = 1, y = x ++ && x --;
```

组成逻辑与表达式语法形式的运算符&&称为逻辑与运算符；逻辑与运算符&&是左结合的。这意味着，表达式

```
3 && a && 1 && b
```

等价于

```
((3 && a) && 1) && c
```

基于以上所述，显然，表达式0&&b&&c的值是0，且不求值b和c。

6.25 逻辑或表达式

一个表达式E1，加上一个“||”运算符和另一个表达式E2，即可构成逻辑或表达式。

其中，E1是逻辑或表达式及之前的表达式，E2是逻辑与表达式及之前的表达式。显然，E1本身也可能是另一个逻辑或表达式。参见词条“表达式列表”，如果想让E1和E2也能是排在后面的那些表达式，则只需用括号将它们变为基本表达式。

操作数E1和E2都要求是标量，若它们当中有一个求值的结果不为0，则逻辑或表达式的结果就是1；若它们求值的结果都是0，则E1||E2的结果是0。

逻辑或表达式的结果类型是int，但结果不是左值。

因此，表达式0||0的结果是0；3||5的结果是1。

逻辑或表达式（运算符||）保证是从左到右求值的。这意味着，如果E1和E2都会求值的话，则E1的值计算和副作用将保证前序于E2的值计算和副作用，E1和E2的求值之间有一个序列点。

事实上，逻辑或表达式求值时，总是先求值E1，，如果E1的值不为0，则对E2求值已无必要，因此不求值E2，而且整个逻辑或表达式（运算符||）结果为1。

请看下面的例子，在数学上，1/0是不合法的。但是，因为||运算符的左操作数为0，所以右操作数不被求值，所以整个表达式是合法的，值为0。

```
if (1 || 1/0) {/* …… */}
```

因为序列点的存在，下例中，表达式x++||x--不存在未定义行为：

```
int x = 0, y = x ++ || x --;
```

组成逻辑或表达式语法形式的运算符||称为逻辑或运算符；逻辑或运算符||是左结合的。这意味着，表达式

```
3 || a || 1 || b
```

等价于

```
((3 || a) || 1) || c
```

基于以上所述，显然，表达式1||b||c的值是1，且不求值b和c。

6.26 条件表达式

由表达式 E1、“?”、表达式 E2、“:”，以及表达式 E3 按顺序组成的序列（E1?E2:E3）是一个条件表达式。

其中，E1 是逻辑或表达式及之前的表达式；E2 是任意表达式，E3 是条件表达式及之前的表达式。显然，E3 本身也可能是另一个条件表达式。参见词条“表达式列表”，如果想让 E1、E2 和 E3 也能是排在后面的那些表达式，则只需用括号将它们变为基本表达式。

因为 E2 可以是任何表达式，而 E3 可以是另一个条件表达式，或者条件表达式之前的表达式，因此，以下条件表达式中与 E1 对应的部分是一个赋值表达式，因为赋值表达式在表达式列表中排在逻辑表达式之后，所以要用括号使之成为基本表达式；与上述 E2 对应的部分是一个逗号表达式；与上述 E3 对应的部分是一个移位表达式。

```
int x, y = (x = 5) ? 5, 6, 7 : 8 >> 2;
```

条件表达式（条件运算符）的结果取决于 E1 的值。若 E1 的值为 0，则条件表达式的结果取自于 E3，否则，取自于 E2。条件表达式（条件运算符）的结果不是左值。

注意，“取自”的意思是可能需要经过适当的转换。下例中，因为第一个操作数不为 0，故条件表达式的值来自操作数(char)2，并经常规算术转换后，提升为 int 类型（因为第三个操作数的类型是 int）。所以条件表达式的值是 2，表达式的结果类型是 int。操作数的类型不同，转换的规则也不一样，具体的规则下面还会进一步介绍。

```
5 ? (char) 2 : 3
```

因为条件表达式的结果不是左值，所以下面的代码不能工作：

```
void f (int a, int b)
{
    a == b ? a : b = 3;
   /* ..... */
}
```

但是，一元*运算符的结果是左值，所以上述代码可以修改如下：

```
void f (int a, int b)
{
    * (a == b ? & a : & b) = 3;
   /* ..... */
}
```

E1 的类型应为标量，而且一定会被求值。若 E1 的值为 0，则求值 E3，且它们之间存在一个序列点；否则，求值 E2，且 E1 和 E2 的求值之间存在一个序列点。

由于序列点的存在，下例中，y 的初始化表达式不存在未定义行为：

```
int x, y;
/* ..... */
y = x ? x ++ : x --;
```

条件表达式的结果类型取决于 E2 和 E3 的类型，现分述如下。

（1）E2 和 E3 可以同为算术类型。在这种情况下，它们需要进行常规算术转换，条件表

达式（条件运算符）的结果类型是转换后的公共类型。

下例中，条件表达式的（结果）类型是 long long int。

```
1 ? 2 : 3LL
```

（2）E2 和 E3 可以是同一种结构或联合类型。在这种情况下，结果的类型就是该结构或联合类型。

下例中，条件表达式 x != y ? s1 : s2 的类型是 struct s，从这个结构类型的值中提取一个成员的值（运算符.的左操作数可以不是左值）并赋给对象 y。

```
struct s {int i; float f;};

void frstg (struct s s1, struct s s2)
{
    int x, y;
    /* ..... */
    y = (x != y ? s1 : s2).i;
    /* ..... */
}
```

如果 E2 和 E3 不是同一种结构类型，则程序转换将会出错，比如下面这三个条件表达式，只有最后一个是合法的，前两个都不正确。当然，这三个表达式都没有什么用处，仅仅为了说明问题。

```
struct t {char c;} t = {'x'};
struct s {char c;} s = {'y'};

0 ? s : t;
1 ? (struct {char c;}) {'x'} : (struct {char c;}) {'y'};
1 ? (struct g {char c;}) {'x'} : (struct g) {'y'};
```

（3）E2 和 E3 可以同为 void 类型。在这种情况下，结果的类型也是 void。典型地，这种条件表达式通常只关注其副作用。

下例中，条件表达式具有（不存在的）void 类型的值。

```
void f (int), g (int);
int h (void);

h () ? f (1) : g (2);
```

（4）E2 和 E3 可以同为指针，但除非有一个是空指针常量，或者是指向 void 的指针，否则，它们所指向的类型在去掉限定符后必须是兼容的。总体上，结果的类型是一个指针，所指向的类型是 E2 和 E3 所指向的类型的复合类型（显然，兼具 E2 和 E3 所指向类型的所有限定符）。

举例来说，给定以下声明：

```
int * pi = /* ..... */;
const int * cpi = /* ..... */;
```

则，因为pi所指向的类型是int *，cpi所指向的类型是const int *，因此，表达式

```
pi != cpi ? cpi : pi
```

的（结果）类型是 pi 所指向的类型的限定符（无）加上 cpi 所指向的类型的限定符（const），即const int *。

进一步地，尽管上述表达式的结果不是左值，但表达式

```
* (pi != cpi ? cpi : pi)
```

的结果却是一个左值，不过并非一个可修改的左值，所以表达式

```
* (pi != cpi ? cpi : pi) = 1
```

是非法的。

注意，这里所讨论的限定符，是“指针所指向的类型”的限定符，而不是指针的限定符。为了更清楚地加以区别，下面用泛型选择来加以说明。

```
const int * restrict pci = /* …… */;
volatile int * pvi = /* …… */;

printf("Result type of expression 'pci ? pci : pvi' is %s.\n",
       _Generic ( pci ? pci : pvi,
                  const volatile int * : "const volatile int *",
                  default : "???"
                )
       );
```

其中，pci 是一个指针，是 restrict 限定的，它所指向的类型是 const 限定的；pvi 所指向的类型是 volatile 限定的。结果表明，表达式 pci ? pci : pvi 的结果类型是 const volatile int *，而不是 const volatile int * restrict（条件运算符的结果不是左值，指针本身的限定符被丢弃）。

（5）本规则是第 4 个规则的扩展，主要是考虑到 E2 和 E3 中有一个是任意指针，另一个是空指针常量的情况。在这种情况下，条件表达式的结果是一个指针，所指向的类型就是那个任意指针的类型。

给定以下声明：

```
const int * restrict cpi = /* …… */;
```

则，表达式 cpi ? cpi : 0 的（结果）类型是 const int *，指针本身的限定符被丢弃。进一步地，表达式 cpi ? (void *) 0 : (volatile void *) 0 或者 cpi ? 0 : (volatile void *) 0 的（结果）类型是 volatile void *，因为 0 和(void *)0 是空指针常量，但(volatile void *)0 不是，它是指向 volatile void 类型的空指针。

（6）本规则是第 4 个规则的扩展，主要是考虑到 E2 和 E3 中有一个是除空指针常量之外的任意指针，另一个指针指向“限定或无限定的 void”这种情况。在这种情况下，条件表达式的结果是一个指针，指向限定的 void 类型，且限定符取自 E2 和 E3 所指向的类型。

给定以下声明：

```
const int * cpi = /* …… */;
volatile void * pv = /* …… */;
```

那么，表达式 cpi ? cpi : pv 的（结果）类型是 const volatile void *。这里使用了泛型选择

来演示如何验证这个结果，你可以将以下泛型选择表达式放在printf函数中加以输出。

```
_Generic (cpi ? cpi : pv, const volatile void * : 1, default : 0)
```

也就是说，条件表达式 cpi ? cpi : pv 的（结果）类型是 const volatile void *。再如，条件表达式 cpi ? (volatile void *)0 : cpi 的结果类型是 const volatile void *。

组成条件表达式语法形式的运算符称为条件运算符，条件运算符是从右向左结合（右结合）的。这意味着表达式

```
a ? b : c ? d : e
a ? b ? c : d : e
```

分别等同于

```
a ? b : (c ? d : e)
a ? (b ? c : d) : e
```

要理解这样做的原理，只需要掌握以下步骤即可。

（1）从右边开始向左找，直至遇到第一个未被处理的“?”。

（2）再从“?”开始向右找第一个遇到的“:”。

（3）在“?”左边的操作数前加“(”，在“:”右边的操作数后加“)”。

（4）返回1，继续处理其他条件运算符?:。

下面的例子用于演示条件运算符的结合性，程序的功能如下：如果 x 的值是 0，则函数 f 返回 0；如果 x 的值大于 0，则返回 1；如果 x 的值小于 0，则返回-1。

```
int f (int x)
{
    return x ? x > 0 ? 1 : -1 : 0;
}
```

下面这个例子源于一个网友的作业题，题目的要求编是写一个函数，函数原型如下。

```
int find (int a [], int n, int x, int * p, int * q);
```

其目的是在 n 个元素的数组 a 中查找 x ，返回 x 在 a 中的出现次数，x 在 a 中首次出现的下标和最后一次出现的下标分别由参数 p、q 传回。

下面的代码不考虑无效指针的情况。

```
int find (int a [] ,int n, int x , int * p, int * q)
{
    int cnt = 0;
    while (n --)
        if (a [n] == x)
            * (cnt ++ ? p : q) = n;
    if (cnt == 1) * p = * q;
    return cnt;
}
```

6.27 赋值表达式

赋值表达式包括简单赋值和复合赋值，详情可参见各自的词条。例如：

```
int x, y;
x = 0;
y += 3;
```

在上面的例子中，x=0 及 y+=3 都是赋值表达式，前者是简单赋值，后者是复合赋值的一种形式。

对赋值表达式的求值将更新赋值运算符左操作数的存储值，这是赋值表达式的一个副作用，且该副作用后序于其左右操作数的值计算，但赋值运算符左右操作数的求值是无序的。

这意味着，给定声明

```
int i = 0;
```

则赋值表达式

```
i += 1;
```

是先计算 i 和 1 的值，再将计算结果（2）更新到 i 所指示的对象中。再如：

```
void f (int * pa, int i, int j)
{
    pa [i ++] = ++ j;
}
```

这里可以大体上分为 3 个表达式：pa[i++]、++j 和 pa[i++]=++j，而且这 3 个表达式的求值都有副作用，pa[i++]的副作用是更新 i 所指示的对象有存储值；++j 的副作用是更新 j 所指示的对象的存储值；pa[i++]=++j 的副作用是更新 pa[i++]所指示的对象的存储值。

运算符=的两个操作数 pa[i++]和++j 都要分别求值，而且是无序的。也就是说，可能先求值 pa[i++]，也可能先求值++j，也可能是交叉进行的。但是这无关紧要，毕竟更新表达式 pa[i++]所指示的对象的存储值是发生在 pa[i++]和++j 的值计算之后。尽管它们的副作用不保证顺序，但它们针对不同的对象，这里既没有未定义的行为，也不影响整个表达式结果的正确性。

相反地，因为针对同一个对象 j 的多个求值是无序的，表达式 pa[j++] = j 的行为是未定义的。

很多读者只关注于赋值表达式的“赋值”，实际上赋值只是一个副作用，每个赋值表达式都有自己的值，也有自己的类型。那么，赋值表达式的（值的）类型是什么呢？有人可能认为是赋值运算符左操作数 E1 的类型。实际上，不完全是这样的。

要得到赋值表达式的（值的）类型，可以先假设对 E1 进行了左值转换（可参见“左值转换”），赋值表达式的（值的）类型就是 E1 左值转换后的类型。

换句话说，假设对赋值运算符的左操作数进行了左值转换，那么赋值表达式的类型就是左值转换后的类型。具体来说，若左值的类型是限定的，则赋值表达式的类型是左值类型的非限定版本；若左值是原子类型，则赋值表达式的类型是左值类型的非原子版本。例如，给定声明：

```
volatile int * restrict pci;
```

则以下两个表达式的（结果）类型是不同的。

```
pci = 0
* pci = 0
```

其中，第一个表达式中的 pci 是 restrict 限定的指针，左值转换后的类型将不再是 restrict 限定的指针。所以，表达式 pci=0 的结果类型是 volatile int *。

同理，在表达式* pci = 1 中，左值*pci 的类型是指针 pci 所指向的类型，即 volatile int，假定对*pci 进行了左值转换，那么，转换后的类型是 int。也就是说，表达式* pci = 1 的类型是 int。做为思考题，请用泛型选择表达式来验证以上结论。

【思考题】若对象 c 的类型是 char，请编写程序验证赋值表达式 c = 0LL 的类型也是 char。

组成赋值表达式语法形式的运算符称为赋值运算符；赋值运算符是右结合的。因此，表达式

```
x = y = z += 3
```

等价于表达式

```
x = (y = (z += 3))
```

6.27.1 简单赋值

简单赋值表达式属于赋值表达式。

一个可修改的左值 E1，加上一个运算符 “=” 和另一个表达式 E2，即可构成简单赋值表达式。

其中，E1 是一元表达式及之前的表达式；E2 是赋值表达式及之前的表达式。显然，E2 本身也可能是另一个赋值表达式。参见词条 “表达式列表”，如果想让 E1 和 E2 也能是排在后面的那些表达式，则只需用括号将它们变为基本表达式。

例如：

```
int x, y;
x = 0;
y = x += 3;
```

其中，简单赋值表达式 x=0 是由基本表达式 x、运算符=，以及整型常量 0 组成的；简单赋值表达式 y=x+=3 是由基本表达式 y、运算符=，以及另一个赋值表达式 x+=3 组成的。就像刚才所说的，E1 要求是一元表达式及之前的表达式，E2 要求是赋值表达式及之前的表达式，如果使用其他表达式，则要用括号使之成为基本表达式。例如

```
int x, y, z, a [3];
/* ..... */
* (a + 1) = (x ++, y ++, z ++);
```

简单赋值使用运算符 “=”，用于将 E2 的值转换为赋值表达式的类型，并替换 E1（所指示的对象的）存储值。

考虑以下代码：

```
char c;
int i = 0;
long l;
```

```
l = (c = i);
```

其中，i 的值先被转换为表达式 c=i 的类型，即 char；表达式 c=i 的值又被转换为外层表达式的类型，即 long int。

又如：

```
void f (int n)
{
    unsigned char c;
    /* …… */
    if ((c = n) == -1) {/* …… */}
}
```

n 的值是通过函数调用传入的，被转换为表达式 c=n 的类型（unsigned char），截短后保存到 c 所指示的对象中。为了同 int 类型的常量-1 进行比较，表达式 c=n 的值重新转换为 int 类型，但不可能为负。因此，(c = n) == -1 永远不会成立。

简单赋值表达式（运算符=）的结果来自于 E1，是 E1 所指示的对象在赋值操作完成后的值。赋值表达式的结果不是左值。

这意味着，为了得到赋值表达式的值，应该在赋值之后，重新读赋值运算符=左操作数所指示的对象。但是，允许 C 实现直接将赋给左操作数的值作为赋值表达式的值，即使左操作数所指示的对象具有 volatile 限定的类型。

因此，在下面的例子中，指针 pvi 所指向的对象是 volatile 限定的，这意味着它的值随时有可能变化（即使是在赋值之后和计算整个赋值表达式的值之前）。尽管如此，C 实现也可能直接将赋给该对象的值作为整个表达式* pvi = 0 的值传递给函数 f。

```
volatile int * pvi = /* …… */;
void f (int);
f (* pvi = 0);
```

因为简单赋值表达式的结果不是左值，所以，表达式(a = b) = c 是非法的。

简单赋值表达式中的 E1 和 E2 具有类型上的限制，现分述如下。

（1）都是算术类型，其中，E1 可以是原子的，也可以是限定（const 限定符除外）或无限定的。

在下例中，表达式*pi 和 j 都是算术类型。

```
volatile int * pi = /* …… */;
const int j = 0;
* pi = j;
```

（2）都是结构类型，或者都是联合类型，但要求它们的类型在去掉限定符之后是兼容的，且 E1 的类型不能是 const 限定的，它的成员也不能是 const 限定的。

例如，在下面的例子中，s1、s2 和 t1 都是结构类型，s1 和 s2 是同一种结构类型，但 s1 是 const 限定的结构类型，只能将它的值赋给 s2 而不是相反；t1 和 s1 的类型不同，不能赋值。

```
struct s {int i;};
struct t {int i;};

const struct s s1 = {0};
```

```
struct s s2;
struct t t1;
s2 = s1;
t1 = s1;                    //错误：类型不兼容
```

（3）都是指针，要求它们所指向的类型在去掉限定符之后是兼容的，且 E1 所指向的类型具有 E2 所指向的类型的全部限定符。E1 本身的类型可以是原子的、限定（const 限定符除外）或无限定的指针。

例如：

```
char * p1 = 0;
const char * p2;
p2 = p1;     //合法
```

首先，p1 和 p2 都是指针；其次，它们指向的类型分别是 char 和 const char，去掉限定符后，类型是兼容的，都是 char；最后，p2 指向的类型(const char)有一个限定符 const，p1 指向的类型（char）无限定符，这可以看做 p2 所指向的类型拥有 p1 指向的类型的所有限定符（无），所以 p2=p1 是合法的赋值。

又如：

```
char * const * volatile p1 = 0;
const char * restrict * p2;
p2 = p1;                    //非法
```

首先，p1 是 volatile 限定的指针；p2 也是指针类型，但无限定符。其次，p1 指向另一个 const 限定的指针（char * const），去掉这个限定符后是 char *；p2 也指向另一个 restrict 限定的指针（const char * restrict），去掉这个限定符后是 const char *。显然 char *和 const char *并不兼容。所以，p2=p1 不是合法的赋值。

同样是刚才的例子，简化后语句如下，可能看得更清楚。

```
char * * p1 = 0;
const char * * p2;
p2 = p1;     //非法
```

其中，p1 指向另一个无限定的指针（char *）；p2 也指向另一个无限定的指针（const char *）。既然 p1 和 p2 指向的类型都不是限定的（都不是限定的指针），则直接看这两个被指向的类型是否兼容即可，比较的结果是 char *和 const char *并不兼容，故 p2=p1 不是合法的赋值。

在下例中，左操作数 pci 是 restrict 限定的指针；右操作数 pvi 是 const 限定的指针；pci 和 pvi 所指向的类型分别是 const volatile int 和 volatile int，这两个类型去掉限定符后是兼容的；右操作数所指向的类型的限定符是 volatile，在左操作数 pci 所指向的类型中，也包含该限定符。因此，pci=pvi 是合法的。

```
const volatile int * restrict pci;
volatile int * const pvi = 0;
pci = pvi;  //合法
```

（4）都是指针，一个指向对象，另一个指向限定或无限定的 void 类型，且 E1 所指向的类型具有 E2 所指向的类型的全部限定符。E1 本身可以是原子的、限定（const 限定符除外）

或无限定的指针。

例如（解释同上，但稍有不同，读者可自行分析）：

```
const volatile void * restrict p1;
const volatile char * restrict p2;
volatile void * const pv = 0;
p1 = pv;
p2 = pv;
```

（5）都是指针，E1 是原子的、限定（const 限定符除外）或无限定的任意指针类型；E2 是空指针常量。

给定声明：

```
volatile unsigned long int * vp;
void (* pf) (int, int);
```

则，将空指针常量赋给指针的例子是

```
vp = (void *) 0;
pf = 0;
```

（6）E1 的类型是原子的、限定（const 限定符除外）或无限定的_Bool；E2 是任意指针类型。

这种赋值初看起来有些奇怪，但很容易理解。因为，任何标量类型的值，若其为 0，则转换为_Bool 类型后依然为 0，否则转换后为 1。下面是一个例子，malloc 是内存分配函数，分配成功时返回指向已分配空间的指针，否则返回空指针。使用该函数需要包含头文件<stdlib.h>。

```
char * p;
_Bool b = (p = malloc (100));
```

必须考虑到这样一种情况：如果为对象 A 赋的值是从对象 B 中读出来的，且 A 和 B 的存储空间重叠，则它们应当完全重叠，而且这两个对象的类型在去掉限定符后要求是兼容的；否则，程序的行为是未定义的。

下面给出了几个例子。

```
# include <limits.h>

void f (void)
{
    int i = 0;
    i = i;                          //E1: 合法

    volatile int * pi = & i;
    * pi = i;                       //E2: 合法

    union {char c; int i; } u = {0};
    u.1 = u.i;                      //E3: 未定义行为
    i = u.i;
```

```
    u.c = i;                    //E4: 允许
}
```

其中，前两个赋值（E1 和 E2）都是合法的，因为它们只是将自己的值读出后再写入，运算符=左右两个操作数所指示的对象都是自己；E3 的行为都是未定义的，联合的两个成员 i 和 c 是部分重叠的，此外，它们的类型也不兼容；但是，通过引入一个中间对象，E4 即是合法的。

6.27.2 复合赋值

一个表达式 E1，加上一个复合赋值运算符和另一个表达式 E2，即可构成复合赋值表达式。

复合赋值运算符包括*=、/=、%=、+=、-=、<<=、>>=、&=、^=和|=。因此，复合赋值表达式有以下形式。

```
E1 *= E2（在形式上等价于 E1 = E1 * E2）
E1 /= E2（在形式上等价于 E1 = E1 / E2）
E1 %= E2（在形式上等价于 E1 = E1 % E2）
E1 += E2（在形式上等价于 E1 = E1 + E2）
E1 -= E2（在形式上等价于 E1 = E1 -E2）
E1 <<= E2（在形式上等价于 E1 = E1 << E2）
E1 >>= E2（在形式上等价于 E1 = E1 >> E2）
E1 &= E2（在形式上等价于 E1 = E1 & E2）
E1 ^= E2（在形式上等价于 E1 = E1 ^ E2）
E1 |= E2（在形式上等价于 E1 = E1 | E2）
```

其中，E1 是一元表达式及之前的表达式，E2 是赋值表达式及之前的表达式。显然，E2 本身也可能是另一个赋值表达式。参见词条“表达式列表”，如果想让 E1 和 E2 也能是排在后面的那些表达式，则只需用括号将它们变为基本表达式。

上述形式上的等同性并不意味着 E1 会求值两次，实际上，它只求值一次。下面是一个有关复合赋值表达式的例子。

```
int x = 0, y;
x += y = 3;
```

其中，复合赋值表达式 x+=y=3 是由基本表达式 x、运算符+=，以及另一个赋值表达式 y=3 组成的。就像刚才所说的，E1 要求是一元表达式及之前的表达式，E2 要求是赋值表达式及之前的表达式，如果使用其他表达式，则应用括号使之成为基本表达式：

```
int x, y, z, a [3];
/* ..... */
* (a + 1) += (x ++, y ++, z ++);
```

复合赋值表达式（复合赋值运算符）的结果来自于 E1，是 E1 所指示的对象在赋值操作完成后的值。赋值表达式的结果不是左值。

这意味着，为了得到赋值表达式的值，应该在赋值之后，重新读左操作数所指示的对象。但是，允许 C 实现直接将赋给左操作数的值作为赋值表达式的值，即使左操作数所指示的对象具有 volatile 限定的类型。因此，在下面的例子中，指针 pvi 所指向的对象是 volatile 限定

的，这意味着它的值随时有可能变化（即使是在赋值之后和计算整个赋值表达式的值之前）。尽管如此，C 的实现也可能直接将赋给该对象的值作为整个复合赋值表达式* pvi += 1 的值传递给函数 f。

```
volatile int * pvi = /* …… */;
void f (int);
f (* pvi += 1);
```

因为复合赋值表达式的结果不是左值，所以，表达式

```
(a += b) &= c
```

是非法的。

在复合赋值表达式中，E1 的类型可以是原子的、限定或无限定的。如果使用的运算符是+=和-=，则要么 E1 是指向完整对象类型的指针，E2 是整数类型；要么，E1 和 E2 都是算术类型。

如果使用的是其他复合赋值运算符，则对 E1 和 E2 的类型要求取决于复合赋值运算符中的+、-、*、/、%、>>、<<、&、^和|。这些二元运算符对其操作数的要求详见“加性表达式”、“乘性表达式”、“移位表达式”、“按位与表达式”、“按位异或表达式”和“按位或表达式”。

例如，%运算符的两个操作数应为整数类型，所以%=运算符的左、右操作数都必须是整数类型。

下例中，pc 是 restrict 限定的指针，指向类型 char *。复合字面值的类型是 char[3]，自动转换为指向其第一个元素的指针，即类型也为 char *，因此可以用来初始化 pc。

```
char * restrict pc = (char []) {1, 2, 3};
pc -= 1;
* pc += 1;
```

但重点是，对于表达式 pc-=1，运算符-=的左操作数是指针，右操作数是整数类型，因此是合法的。这样，pc 将指向数组的下一个元素。

对于表达式*pc+=1，运算符-=的左操作数（*pc）是算术类型（char），右操作数 1 也是算术类型（int），是合法的。

6.28 逗号表达式

一个表达式 E1，加上一个“,”和另一个表达式 E2，即可构成逗号表达式。其中，E1 可以是任何一个表达式，E2 是赋值表达式及之前的表达式。参见词条“表达式列表”，如果想让 E1 和 E2 也能是排在后面的那些表达式，则只需用括号将它们变为基本表达式。

因为逗号表达式排在表达式列表的末尾，所以，既然 E1 可以是任何表达式，则它可能是另一个逗号表达式。

下例中，字体加粗的部分是两个逗号表达式的例子，另一部分则不是，因为它仅仅用于分隔两个声明符。

```
int x = (1, 2), y = 0, f (void);
x ++, f ();
```

逗号运算符的左操作数 E1 作为 void 表达式求值。这意味着，左操作数的值被丢弃，通常只关注其副作用。

逗号表达式（逗号运算符）的结果不是左值，这个值是 E2 的值，值的类型和 E2 的类型相同。

以上例来说，声明中的初始化器要求一个赋值表达式及之前的表达式（参见“表达式列表”），所以，必须用括号将逗号表达式 1, 2 括住，使之成为基本表达式。否则，整个声明就是无效的，像这样：

```
int x = 1, 2, y = 0, f (void);
```

这意味着对象 x 的初始值为 1，然后声明了另一个标识符“2”，但这样的标识符是无效的（参见“标识符”）。

回到逗号表达式(1, 2)中，逗号运算符的左操作数 1 是无用的，它的值被丢弃，该逗号表达式的值是逗号运算符右操作数 2 的值。因此，该表达式除了用于演示外，并无其他用处，可以直接将整个声明改为

```
int x = 2, y = 0, f (void);
```

再来看第二个逗号表达式，子表达式 x++的值被丢弃，但保留了它的副作用。整个（逗号）表达式的值是函数 f 的返回值，同样被丢弃。下面是另一个逗号表达式的例子，逗号表达式的值（即函数 f 的返回值）被用于和整数 6 相加，然后赋给对象 y。

```
y = (x ++, f ()) + 6;
```

逗号表达式求值时，先求值 E1，再求值 E2，在它们之间存在一个序列点。

下例中，因为逗号在声明中通常用于分隔标识符，故在作为初始化器出现时，需要加上括号使之成为基本表达式。逗号运算符的左操作数 x++作为 void 表达式求值，仅保留副作用（更改了对象 x 的存储值），该表达式的值被丢弃；逗号运算符的类型是其右操作数--x 的类型，逗号运算符的值也是右操作数--x 的值。最后，因为在逗号表达式左右操作数的求值之间存在一个序列点，故表达式(x++, --x)不存在未定义行为。

```
int x = 1, y = (x ++ , -- x);
```

当逗号表达式用做函数的参数时，为它加上括号，使其成为基本表达式，这是相当重要的。在下例中，逗号运算符的左右操作数类型不同，但也没关系。

```
void fcomma (const char *, char, int);
float f = 0.0f;
char c;
fcomma ("hello", (f += 0.1, c = 'x'), 5);
```

组成逗号表达式语法形式的运算符“,”称为逗号运算符；逗号运算符是左结合的。因此，表达式

```
f (), g (), x ++
```

等价于表达式

```
(f (), g ()), x ++
```

逗号表达式应该以自然的方式使用。也就是说，应该用在它需要的地方，从而使得程序看起来更紧凑。例如：

```
# include <stdlib.h>
```

```
/*
 * 功能：在字符串中找出连续最长的数字串，并把这个数字串的长度返回
 * 如果存在两个以上同长度的最长串，则返回第一个最长串
 * 函数原型：unsigned maxlenstr (char * * pouts, char * pins)
 * 输入参数：char * pins 输入字符串
 * 输出参数：char * * pout 连续最长的数字串，若不存在，则返回空串。
 * pout 需要的空间在函数内用 malloc 函数申请，由调用处负责释放。
 * 返回值：连续最长的数字串的长度
 */
unsigned maxlenstr (char * * pouts, char * pins)
{
    unsigned cnt = 0, c = 0;
    char * pos;

    while (* pins ++ != '\0')
        if (* (pins - 1) >= '0' && * (pins - 1) <= '9')
            c ++;
        else
        {
            if (c > cnt) cnt = c, pos = pins - c - 1;
            c = 0;
        }

    if ((c = cnt) != 0)
    {
        * pouts = malloc (c + 1);
        (* pouts) [c] = '\0';
        while (c --) (* pouts) [c] = pos [c];
    }
    else * pouts = 0;

    return cnt;
}
```

下面是另一个逗号表达式的例子，用于计算1～100的整数和。

```
int i = 1, sum = 0;
while (sum += i, i ++ < 100) ;
```

6.29 常量表达式

有些表达式在程序转换期间求值，而且求值的结果是一个常量。例如：

```
'x' + sizeof (int)
```

这里，'x'和 sizeof (int)都是常量表达式，而这整个加法表达式也是常量表达式。求值为常量的表达式很多，可将它们归为一类，称为常量表达式。常量表达式包括，但不限于算术常量表达式、整型常量表达式和地址常量表达式（指针常量表达式），允许 C 实现使用其他形式的常量表达式。

常量表达式求值为一个常量。因此，但凡是需要使用常量的地方，都可以使用常量表达式。

下例中，对象 x、y 和 z 的初始化器都是常量表达式

```
int x = 0, y = 8 >> 2, z = sizeof (int) * 16;
```

一个表达式是不是常量表达式，可以从它的形式上看出来。因为是在程序转换期间求值，所以包含赋值、递增、递减、函数调用或者逗号运算符的表达式通常不是常量表达式。但这也不是绝对的。如果它们不被求值，或者包含它们的表达式或子表达式不会被求值，则该表达式也是常量表达式。

例如，sizeof运算符的操作数通常不会被求值，因此，给定声明

```
int x, f (void);
```

则，表达式

```
sizeof (x ++)
sizeof f ()
sizeof (x = f ())
sizeof (f (), x)
```

都是常量表达式。第一个常量表达式不求值 x++，是因为不求值也会知道该表达式的类型是 int；第二个常量表达式不求值 f()，因为只要知道其返回类型即可；第三个常量表达式不求值 x=f()，因为只要知道赋值表达式的结果类型即可；第四个常量表达式不求值逗号表达式 f(),x，因为只要知道逗号表达式的结果类型即可，即 x 的类型（int）。

所有表达式都有自己的类型，常量表达式也不例外。常量表达式的值不能超出其类型所容许的范围。

例如，常量表达式 INT_MAX+100 的类型是 int，但得到的常量值超出了 int 类型可以表示的值的范围。同理，

```
# include <limits.h>
enum {a = INT_MAX, b};
```

也违反了上面的约束条件，因为枚举常量 b 的值超出了 int 类型可以表示的值的范围。

在 C 中，有多种场合需要常量表达式，如位字段的大小、枚举常量的值、数组的大小、case 标号的值、预处理指令中的整型常量，以及具有静态存储期的对象的初始化器。

以上讲到的最后一点很重要，它意味着，要验证一个表达式是否为常量表达式，只需创

建一个具有静态存储期的对象，然后将此表达式作为其初始化器即可。例如：

```
int i = 0;      //合法
int j = i;      //非法
void f (void) {/* ..... */}
```

其中，对象 i 和 j 都具有静态存储期，所以它们的初始化器必须是常量表达式。0 是常量表达式，而 i 则不是。

【思考题】给定声明

```
int x = 0;
```

则以下表达式中，属于常量表达式的是（　　）。

A. `0.5d, 3LL`

B. `sizeof (x ++)`

C. `sizeof (x ++), 200`

6.29.1 整型常量表达式

整型常量表达式，顾名思义，这样的表达式求值为常量，且它的值是整数类型的。

整型常量表达式通常是运算符和以下常量操作数的组合：

（1）整型常量、枚举常量和字符常量。

（2）求值为整型常量的 sizeof 表达式。

（3）_Alignof 表达式。

（4）转型表达式，但是，除非作为运算符 sizeof 的操作数，转型表达式求值的结果必须是整型常量。

给定声明：

```
int x;
enum e {Red, Blue, Green,};
```

则以下表达式都是合法的整型常量表达式。

```
0
Red
9000LL
3 + 'x'
Blue + 1
(int) 3.14
sizeof (x)
sizeof ((float) 3.0)
_Alignof (long long int)
```

相反，在下面的代码片段中，表达式 sizeof arr 不是整型常量表达式，因为它的操作数是一个变长数组。

```
# include <stddef.h>

void f (int n)
```

```
{
    char arr [n];
    size_t t = sizeof arr;
  /* 或者将以上两行语句合并为 size_t t = sizeof (char [n]); */
}
```

6.29.2 算术常量表达式

算术常量表达式，顾名思义，这样的表达式求值为常量，且它的值是算术类型。算术常量表达式通常是运算符和以下常量操作数的组合：

（1）整型常量、枚举常量、浮点常量和字符常量。

（2）求值为整型常量的 sizeof 表达式。

（3）_Alignof 表达式。

（4）转型表达式，但是，除非作为运算符 sizeof 的操作数，转型表达式求值的结果必须是算术类型的常量。

给定声明：

```
int x;
enum e {Red, Blue, Green,};
```

则以下表达式都是合法的算术常量表达式。

```
0
0.05f
Red
9000LL
3 + 'x'
Blue + 1
(int) 3.14
sizeof (x)
(double) 5
 (int)(5 / 3.14)
6 / 0.30103 + 0.5
sizeof ((float) 3.0)
_Alignof (long long int)
```

相反，在下面的代码片段中，表达式 sizeof arr 不是算术常量表达式，因为它的操作数是一个变长数组。

```
# include <stddef.h>

void f (int n)
{
    char arr [n];
    size_t t = sizeof arr;
```

```
    /* 或者将以上两行语句合并为 size_t t = sizeof (char [n]); */
}
```

6.29.3 地址常量

地址常量是指针类型的常量，它的（常量）值通常被解释为地址。地址常量包括以下几种。

（1）空指针和空指针常量。

（2）指向左值的指针，且该左值指示一个具有静态存储期的对象。

（3）指向函数的指针。

（4）通过转型表达式从整型常量转换来的指针。

有多种创建地址常量的方法，比较典型的就是使用一元&运算符。从以上几点可知，(void *)0、(int *)0、(char *)0、&main（在宿主式环境下）、&x（如果 x 是一个具有文件作用域的对象声明）和转型表达式(int *) 0xFFFFFFF0 的结果都是地址常量。给定以下代码片段：

```
int x;
struct t {char c; float f;} t;
union u {char c; float f;} u;

void f (void)
{
    static int y;
    static char a [22];
    double d;
}
```

那么，& x、& t、& u、& y、& a、& f、& t.f、& u.f、& a [0]、& a [5] + 1、(char *) a 都是地址常量。

但是，& d 不是地址常量，尽管该表达式的值是一个指针，但并非指向具有静态存储期的对象。要验证表达式是否为地址常量，只需创建一个指向“具有静态存储期的对象”的指针，然后将表达式作为该指针的初始化器。例如：

```
int x;
struct t {int i;} t;

void f (void)
{
    int y;
    int a [22];
```

static i 的对象的初始化器必须是一个常量表达式。

第7章 语句和块

7.1 语句

在程序中，由语句（statements）来指定要完成什么样的动作。C 中的语句大体上可以分为标号语句、复合语句、表达式语句、选择语句、迭代语句和跳转语句。

语句通常是顺序执行的，但在标准文档和本书中有特别说明的除外。典型的例子包括程序的执行遇到跳转语句或者选择语句的时候。

7.2 标号语句

标号语句用于标识一个可以执行的程序入口，这个入口是执行跳转和分支选择的目标，但标号本身并不会改变程序的执行流程。

可以将正常的语句执行流程看成是高速公路，标号是入口。我们知道，高速公路入口只能“进高速”，不能“下高速”，所以正常行驶的车辆将无视高速入口，径直行驶。程序的执行也是一样，标号并不影响程序的正常执行，除非遇到“高速出口”，也就是 break 语句、continue 语句、return 语句，等等。

标号语句的语法形式如图 7-1 所示。

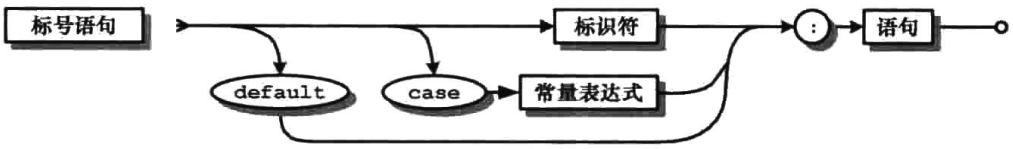


图 7-1 标号语句的语法

case 标号和 default 标号只有在 switch 语句的内部才有意义，所以它们不能位于其他地方。它们的构造、功能和具体用法可参见“选择语句”。

声明为标号的标识符是标号名（label name）。标号名是唯一具有函数作用域的标识符，所以在同一个函数内不允许声明相同的标号名（典型地，标号的声明是一个标识符，后跟一个冒号）。

下面是一个不正确的声明和使用标号的例子。

```
void fun_with_label (int i)
{
    /* ..... */
    goto rep;
    /* ..... */
```

```
rep: i = 0;        //与后面的标号名重复
    /* ····· */
rep: ;             //与前面的标号名重复
    /* ····· */
}
```

标号名可以放在任何语句的前面。给一个语句加上标号名仅仅是标识一个跳转目标，但标号名本身不会影响和改变程序的执行流程，就像它根本不存在一样。

从语法上看，标号名后面必须跟语句（而不是声明，或者其他内容）。进一步地，我们可以得出如下结论：

（1）标号名不能单独出现在块和转换单元的末尾。下面的做法是错误的。

```
{
    label:              //非法。将产生诊断信息
}
```

（2）标号名后面不能是声明。下面的做法是错误的。

```
here : int i = 0;   //非法。后面必须是语句
```

在编程实践中，经常遇到的情况是标号名后面的内容只能以声明开始。在这种情况下，可以在声明之前加一个空语句来解决此问题。例如：

```
extern int f (int *);

void flabel (void)
{
    /* ····· */
    lb: ;
        int i = 0;
        if (f (& i)) goto lb;
    /* ····· */
}
```

标号语句也是语句，因此下面的做法正确。

```
{
la:
lb: {/* ····· */}
lc:
        switch (x) {/* ····· */}
}
```

其中，标号名 la 后面必须是语句，lb 和它后面的语句形成一个标号语句，但这个标号语句又是由标号名 lb 和它后面的语句组成的标号语句。标号名 lb 后面的语句是复合语句（参见“复合语句”）；标号名 lc 后面的语句是 switch 语句（参见“选择语句”）。

7.3 复合语句

复合语句不同于单条语句，如图 7-2 所示，复合语句是由一对花括号，以及可选地，位于花括号中的一些声明和语句组成的。

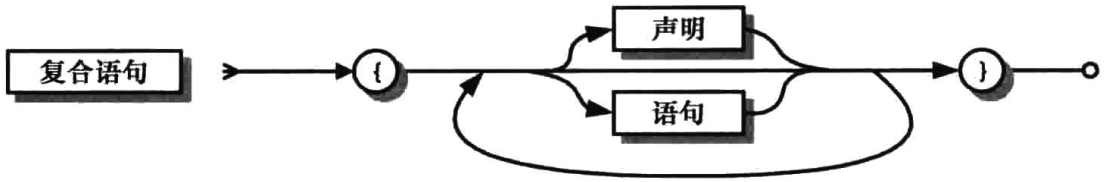


图 7-2 复合语句的语法

显然，以下都是复合语句（在本例中会涉及“空语句”、“表达式语句”和“声明”，可参见相关的词条）。

```
#include <stdio.h>

void f (void)
{                                          //函数体是复合语句
    {}                                     //复合语句的内部可以为空
    {;}                                    //由空语句组成的复合语句
    {{}}                                   //嵌套的复合语句
    {                                      //嵌套的复合语句
        int i;                             //复合语句内的声明
        {
            i = 10086;                     //复合语句内的表达式语句
            {
                int j = 10010;             //复合语句内的声明
                printf ("%d, %d\n", i, j); //复合语句内的表达式语句
            }
        }
    }
}
```

7.4 表达式语句

表达式语句的主要成分是表达式，在表达式后面加一个分号“;”即可构成表达式语句。严格来说，表达式语句的语法形式如图 7-3 所示。

图 7-3　表达式语句的语法

显然，表达式是可选的，如果省略了表达式，则只剩下一个分号，这就形成一个空语句（null statement）。空语句不执行任何操作。

表达式语句的作用主要是得到它（求值时）的副作用，但表达式的值被丢弃。具有副作用的表达式包括赋值、自增、自减和函数调用，等等。表达式语句在执行时，先求值表达式，再丢弃表达式的值（结果）。下例中 S1～S5 都是表达式语句。

```
# include <stdio.h>

void f (int * i, int * x, int * y, int fn (void))
{
    (* i) ++;                          //S1
    * x = 3;                           //S2
    * y = * x;                         //S3
    printf ("hello, c!\n");            //S4
    fn ();                             //S5
}
```

因为历史的原因，很多 C 实现在表达式没有副作用时会产生警告信息，以提示这可能是一个违背程序员本意的疏忽。

如果能够推断表达式的值不会被使用，或者没有产生副作用的必要，则 C 实现可以不对一个全表达式或者它的一部分求值。在下面的例子中，函数 f 内的所有语句，即 S1～S6 都是表达式语句，也都是合法的，但大部分可以被忽略，除了 S1 和 S5。

原因很简单，pvi 是指向 volatile 限定类型的指针，表达式*pvi 被认为是具有副作用的；在 S6 中使用了对象 c 的值，因此，在 S5 中表达式 c++的副作用是有必要的，会被使用的，故必须求值表达式 c++。

```
# include <stdio.h>

void f (void)
{
    volatile int * pvi = (volatile int *) 0xFFFFFFF0;
    int a = 10010, b [33], c = 1, d = 2;
    * pvi;                             //S1
    a;                                 //S2
    b;                                 //S3
    10086;                             //S4
    c ++, d ++;                        //S5
    printf ("%d \n", c);               //S6
}
```

因为每个表达式语句中的表达式也是一个全表达式，所以，表达式语句中的表达式和位于它后面的全表达式之间存在一个序列点。具体内容可参见"序列点"和"全表达式"。

如下例所示，这是两个表达式语句，在表达式 f()和 g()之间存在一个序列点。

```
int f (void), g (void);
f ();
g ();
```

空语句仅仅是一个分号，即

```
;
```

它什么也不做，但在某些场合下又需要它。下面是一个例子，用于计算 1～100 的整数和。因为求和的操作在 for 语句的第三个表达式中进行，而循环体什么也不用做，于是可以使用空语句。

```
int sum = 0;
for (int i = 1; i <= 100; sum += i ++) ;
```

7.5 选择语句

选择语句用于改变程序原有的执行顺序和流程，它包括 if 语句和 switch 语句，具体描述可参见其各自的词条。

上述两种选择语句的语法形式如图 7-4 所示。

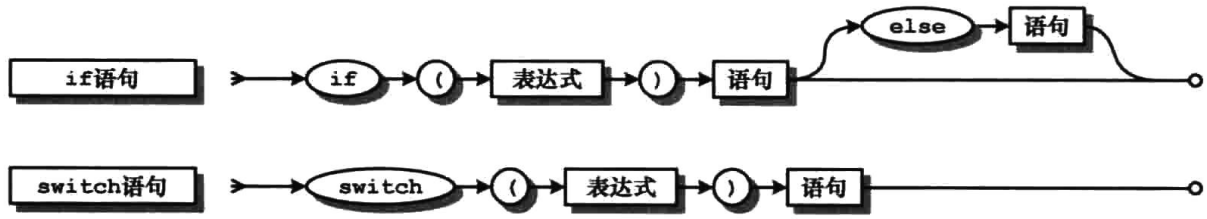


图 7-4 选择语句的语法

选择语句中的表达式又称控制表达式，它的值决定了要执行的语句。从语法图中可知，控制表达式必须放在括号中。

每个选择语句及其子语句都会各自形成一个块，有关这方面的详细描述和示例可参见"块"。

因为每个选择语句中的控制表达式也是一个全表达式，所以，在选择语句中的控制表达式和位于它后面的全表达式之间存在一个序列点。具体内容可参见"序列点"和"全表达式"。

如下例所示，在 if 语句的控制表达式 x==0 及它后面的表达式 x++之间存在一个序列点。

```
if (x == 0)     x ++;
```

7.5.1 if 语句

if 语句中的控制表达式要求是标量（类型），但是一定要谨慎使用，特别是对于浮点类型的表达式来说，精确的相等性是不可移植的。

if 语句的工作方式：如果控制表达式的值不为 0，则执行第一个语句；如果控制表达式的值

为0，且具有else分支，则执行else分支中的语句；如果没有else分支，则离开if语句继续向后执行。

下面是一个if语句的例子，用于判断对象c中的内容是不是数字。

```
# include <stdio.h>

void f (char c)
{
    if ('0' <= c && c <= '9')
        printf ("Digit.\n");
    else
        printf ("Non-digit.\n");
}
```

如果在if语句中包含了标号名，且第一个语句（关键字if之后的语句）的执行是通过标号直接进入的，则第二个语句（关键字else之后的语句，如果有的话）不会执行。

在下面的例子中，如果调用函数f时传递的实参为2，则表达式x==0的结果为0，故先输出“Second.”，然后连续输出两个“First.”。

```
# include <stdio.h>

void f (int x)
{
    if (x == 0)
  lb:
        printf ("First.\n");
    else
        printf ("Second.\n");

    if (x --) goto lb;
}
```

如果在语法上没有问题，那么一个else子句被认为是离它最近的if语句的一部分。

请看下面这个例子，最后一行的else子句到底属于if (x)还是if (y)呢？答案是它属于if (y)，尽管其在排版上有些令人迷惑，但事实上与排版无关。

```
if (x)
    if (y) z = 0;
else z = 1;
```

当然，如果需要使else子句属于if(x)，则可以使用花括号，例如：

```
if (x)
{
    if (y) z = 0;
}
else z = 1;
```

7.5.2 switch 语句

switch 语句中的控制表达式要求是整数类型，不必是常量。但如果在 switch 语句内部有 case 标号，则组成每一个 case 标号的表达式要求是整型常量表达式。

以下是一个 switch 语句的完整例子。

```
# include <stdio.h>

# define prn(s) printf ("Type int is same as %s.\n", s);

void f (void)
{
    switch (sizeof (int))
    {
        case sizeof (short int):
            prn ("short int");
            break;
        case sizeof (long int):
            prn ("long int");
            break;
        default:
            prn ("other type");
    }
}
```

在 switch 语句中，控制表达式的值决定了 switch 语句的主体如何执行。但在此之前，必须使控制表达式和每个 case 标号中的表达式在类型上取得一致，这样才能进行下一步的比较工作。具体来说，要先对控制表达式做整型提升，然后，如果有哪个 case 标号中的常量表达式与控制表达式提升后的类型不一致，则将常量表达式的值转换为控制表达式提升后的类型。

提升类型后，如果控制表达式的值等于某个 case 标号的值，则执行该标号后的语句；如果表达式的值不等于任何一个 case 标号的值，则要看是否存在 default 标号。若有 default 标号，则执行 default 标号后的语句，若没有，则跳过 switch 的主体继续向后执行。

下面是一个 switch 语句的例子，函数 days_of_month 用于返回某个月的天数（不考虑闰年）。

```
enum Months {Jan = 1, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov,
Dec};

int days_of_month (enum Months m)
{
    switch (m) {
        default:
```

```
            return 31;
        case Feb:
            return 28;
        case Apr:
        case Jun:
        case Sep:
        case Nov:
            return 30;
    }
}
```

这个例子用到了 return 语句，它用于将控制返回到当前函数的调用者，具体可参见“return 语句”。

再来看下面的例子：

```
int x = 2;

switch (1)
{
    while (printf ("x:"), x --)
    {
        case 1: printf ("%d\n", x);
    }
}
```

在这段代码中，while 语句嵌套在 switch 语句中。但是，代码执行时，不会先从 while 开始执行，而是直接执行 case 1 后面的语句，因为 switch 语句的动作就是进行匹配和进入分支。因此，这段代码执行时，会先输出 x 的当前值。

通常情况下，程序的执行是顺序的，当控制来到组成 while 语句体的“}”时，必然要回到循环的开始处，重新判断是否需要再次 while 语句体，这会导致输出“x:”。此后，代码的执行完全受 while 语句的控制。因此，这段代码的运行结果如下。

```
2
x:1
x:0
x:
```

在下面这个例子中，函数 is_digit 在 x 为数字字符时返回 1，为其他字符时返回 0，然后根据返回值对数字字符和其他字符做不同的处理。

```
if (is_digit (x))
    do_digit (x);
else
    do_other (x);
```

假如程序在运行一段时间后，收集到的反馈表明，x 为数字字符，特别是为 0 和 1 的时候比较多。那么，为了提高执行效率，降低调用 is_digit 时的开销，可以考虑将上面的程序片断改

写如下。

```
switch (x)
    default:
        if (is_digit (x))
            case '0': case '1': do_digit (x);
        else
            do_other (x);
```

这段代码是这样执行的：如果 x 的值是'0'或者'1'，则直接调用 do_digit 函数；否则，执行标号 default 后面的语句，调用 is_digit 函数，判断返回值，如果是'2'~'9'（'0'和'1'已经判断过了），调用 do_digit 函数；如果是其他字符，则调用 do_other 函数。

思考：在一个 switch 语句中，如果有两个或多个 case 标号的值在经过转换后是相等（同）的，会发生什么？应该进入哪一个语句执行？因此，这种情况是禁止的，在 switch 语句中，不允许存在两个 case 标号的值在提升后相等的情况。

在下例中，表达式 sizeof (char)的结果是 1，这将与前一个 case 标号中的常量表达式重复，导致程序转换失败。

```
switch (x)
{
    case 1: /* …… */
    case sizeof (char): /* …… */
}
```

case 标号和 default 标号只能出现在 switch 语句中，但它们在 switch 语句中的位置和出现的顺序并不重要。特别地，每个 switch 语句只能有一个 default 标号。

因此，下例中的 S1、S2 和 S3 都是非法的。

```
# include <stdio.h>

void f (int x)
{
    case '0':                                   //S1：非法
    default: printf ("Right?\n");               //S2：非法

    switch (x)
    {
        default: printf ("Default 1.\n"); break;
        case 0: printf ("Zero.\n"); break;
        default : printf ("Default 2.\n");      //S3：非法
    }
}
```

尽管通常标号不会使用太多，但实际上在一个 switch 语句中（不包括嵌套的 switch 语句）case 标号的数量还是有限制的，标准的要求是保证不少于 1023 个。

正如这里反复强调的，标号用来标识跳转目标，一旦控制进入选择语句，标号本身不会

影响和改变程序的执行流程。

如下例如示，如果 x 的值为 0，则程序将从 case 0 后的语句开始，一直向下执行（包括 case 1 后的语句），直至遇到如 break、return、goto 这样的语句。

```
switch (x)
{
  case 0:
    g ++;
  case 1:
    g += 2;
    break;
  default:
    g += 3;
}
```

下面来讨论另一个问题，先来看如下一段代码。

```
# include <stdio.h>

void f (int x)
{
    switch (x)
    {
        int i = 0;
        printf ("%d\n", i);
    default:
        i = 1;
    case 0:
        /* ..... */
    }
}
```

这段代码没有任何问题，而且它的意图是比较清楚的：程序员希望在进入 switch 语句后，首先将对象 i 初始化为 0，并打印出来；然后，如果 x 的值为 0，则进行一些处理；如果 x 的值不为 0，则将 i 的值修改为 1，再做相同的处理工作。

但是，正如前面已经介绍过的，由于 switch 语句执行上的特点，它会比较 x 和标号的值，然后直接进入某个标号后的语句，而不可能有机会初始化 i，也不可能有机会调用 printf 函数。

这里的真实情况如下：标识符 i 具有块作用域，在进入 switch 语句（switch 语句本身形成一个块）时，会创建它所指示的对象（对象 i），但只有在控制到达 i 声明的位置时，它才有机会初始化。因此，switch 语句执行时，直接进入 case 0 后面的语句执行，对象 i 虽然已经被创建，但具有不确定的值，而未必是我们期望的 0，也不要指望 printf 函数会被调用。

好在上述代码还能被 C 标准所接受，但下面的代码就不行了：

```
int n = 5;
```

```
switch (x)
{
    int a [n];          //非法
default:
    a [0] = 1;
case 0:
    /* ..... */
}
```

原因很简单，非变长数组类型的、具有块作用域的对象，在进入它所在的块时创建。相反，变长数组类型的对象只在控制到达它声明的位置时才会创建。不能越过变长数组的声明直接从外部进入它的作用域（有关这一规则的详细内容和示例参见“生存期”）。因此，如果在程序中声明了一个可变修改类型的标识符（对象），且在它的作用域内出现了 case 或者 default 标号，那么必须将整个 switch 语句放在该标识符的作用域内。

这意味着，对于上面的示例，如果一定要使用变量修改类型，则必须修改如下。

```
int n = 5, a [n];

switch (x)
{
default:
    a [0] = 1;
case 0:
    /* ..... */
}
```

7.6 迭代语句

迭代语句用于重复执行相同的代码。迭代语句包括 while 语句、do 语句和 for 语句，具体可参见其各自的词条。

以上 3 种迭代语句的语法形式如图 7-5 所示。

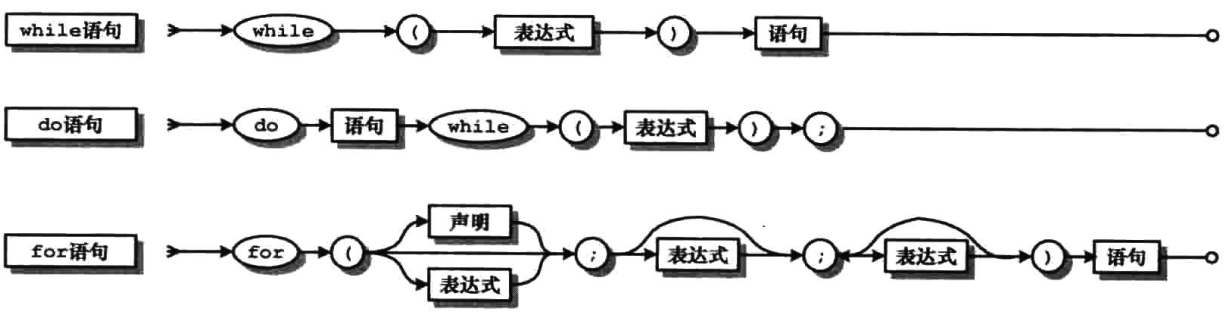


图 7-5 迭代语句的语法

迭代语句语法图中的表达式（对于 for 语句来说，指其第二个表达式，即中间的表达式）称为控制表达式，它们都要求是标量。

迭代语句中的“语句”部分称为循环体，每当控制进入迭代语句时，它将循环执行，直到控制表达式的值为 0。

每次执行完循环体后，都要重新求值控制表达式，并在控制表达式的值不为 0 时重新执行循环体；控制表达式的值为 0 时，离开迭代语句继续往下（后）执行。

以下是几个迭代语句的例子。

```
# include <stdio.h>

void f (void)
{
    int x, y, z;
    x = y = z = 3;
    while (x --) printf ("*");
    do printf ("$"); while (y --);
    for (; z --;) printf ("*");
}
```

因为每个迭代语句中的控制表达式（for 语句的控制表达式是可选的）也是一个全表达式，所以，在迭代语句中的控制表达式和位于它后面的全表达式之间存在一个序列点。具体内容可参见“序列点”和“全表达式”。

如下例所示，在 while 语句的控制表达式 x--和它后面的表达式 printf ("%d\n", x)之间存在一个序列点：

```
#include <stdio.h>

void f (void)
{
    while (x --) printf ("%d\n", x);
}
```

7.6.1　for 语句

为了方便后面的说明，现将 for 语句中的声明和表达式做如下区分：

for （ decl 或者 e1 ； e2 ； e3 ） *语句*

其中，decl 是声明；e1、e2 和 e3 分别是图 7-5 中 for 语句包含的 3 个表达式，e2 是控制表达式，“语句”是循环体。

for 语句先求值控制表达式 e2，结果不为 0 时才执行循环体；结果为 0 时，离开 for 语句向下执行。

for 语句中的 decl、e1、e2 和 e3 都可以省略。如果省略 e2，则 C 实现将用一个不为 0 的常量填补。在这种情况下，e2 求值的结果将始终不会为 0。如下例所示，省略 e2 将产生一个无限循环，即通常所说的死循环。

```
for (;;) {/* …… */}
```

如果在 for 语句中存在 decl，且声明了一个标识符，则该标识符的作用域从它所在的位置开始，延伸到整个 for 语句的结束点。程序执行时，对 decl 部分的处理工作要先于 e2 的第一次求值，而且在整个 for 语句的执行过程中只处理一次。

正是因为这个原因，在下面的例子中，可以在声明对象 i 之后，再用它的值初始化对象 j。同样因为此原因，可以在声明了 i 和 j 之后，在 e2、e3 部分及循环体中使用它们。

```
# include <stdio.h>

void f (void)
{
    for (int i = 1, j = i + 9; i < j; i += 2, j ++)
        printf ("%d, %d\n", i, j);
}
```

注意，在这个语句中，声明了两个标识符 i 和 j，而且 e3 部分是一个逗号表达式。如果觉得逗号表达式看着别扭，可以替换为如下几种等效的形式，因为 for 语句是灵活的。

```
# include <stdio.h>

void f (void)
{
    for (int i=1,j=i+9;i<j;printf("%d, %d\n", i, j),i+=2,j++) ;

    for (int i = 1, j = i + 9; i < j; i += 2)
        printf ("%d, %d\n", i, j ++);

    for (int i = 1, j = i + 9; i < j;)
    {
        printf ("%d, %d\n", i, j);
        i += 2;
        j ++;
    }
}
```

在 for 语句中声明的标识符通常作为计数器使用，以控制循环次数，但在 C99 之前这是不允许的，当时唯一的做法是在整个函数体的开始部分声明，即：

```
# include <stdio.h>

void f (void)
{
    int i, j;
    for (i = 1, j = i + 9; i < j; i += 2, j ++)
```

```
        printf ("%d, %d\n", i, j);
    }
```

但是很显然，它们只在 for 语句内部使用，离开 for 语句之后通常不再使用，而且还要冒着不小心误用它们的风险。

如果在 for 语句中存在 e1，则它会在 e2 第一次求值之前作为一个 void 表达式求值，而且在整个 for 语句的执行过程中只求值一次；e3 在每一次循环后求值，也作为 void 表达式求值。

通常情况下，for 语句的“decl”或者“e1”部分可用于声明或者初始化用做计数器的对象，e3 部分可用于递增控制循环次数的计数器，例如：

```
# include <stdio.h>

void f (void)
{
    int a, b = 55;
    for (a = 1; b <= 75; b = (++ a * 5) + 50)
        printf ("%d\n", b);
}
```

现在的问题是，第一次输出的是 55 还是 60？请自己思考。

如果 for 语句中存在 decl 部分，则标识符的声明只能使用存储类指定符 auto 或者 register。不管有没有指定这两个存储类指定符，C 实现都会根据情况做出适当的选择。如果对象不太大，则将被放在寄存器中以加快执行速度。

为了加深对迭代语句的认识，可使用下面的例子。这个示例的特点是将 for 语句中的 e2 部分改成了逗号表达式（注意逗号表达式的求值特点）。

```
# include <stdio.h>

void f (void)
{
    int sum = 0;
    for (int i = 1; sum += i ++, i <= 100;) ;
    printf ("%d\n", sum);
}
```

上例中的 e2 部分有些复杂，如果按照下面的方法修改，则需要将 i 初始化为 0，而不是原来的 1。

```
# include <stdio.h>

void f (void)
{
    int sum = 0;
    for (int i = 0; ++ i <= 100; sum += i) ;
    printf ("%d\n", sum);
```

```
}
```

最后一个 for 语句的例子来自于我的女儿，今天（2015 年 12 月 2 日星期三）她告诉我，数学老师给他们出了一道题：给定 8 个数 1、3、5、7、9、11、13 和 15，看其中是否有任意 3 个数（可以重复使用）的和是 30。没办法，数学成绩不好的我只能用计算机来解决这个问题，所幸计算机只用了 0.094 秒就给出了答案：没有，不可能。我的程序如下（计算机没有别的本事，就一个字：快。所以我让它尝试所有可能的组合）：

```
# include <stdio.h>
# include <stddef.h>

int main (void)
{
    int a [] = {1, 3, 5, 7, 9, 11, 13, 15};

    for (size_t x = 0; x < sizeof a / sizeof * a; x ++)
        for (size_t y = 0; y < sizeof a / sizeof * a; y ++)
            for (size_t z = 0; z < sizeof a / sizeof * a; z ++)
                if (a [x] + a [y] + a [z] == 30)
                    printf ("%d+%d+%d=30\n", a [x], a [y], a [z]);

    return 0;
}
```

7.6.2 while 语句

while 语句的工作方式很简单，在执行该语句时，先求值控制表达式，值不为 0 时才执行循环体，否则离开 while 语句向下执行。

在下例中，要求输入一个整数，再按输入的长度创建变长数组（这里使用了逗号表达式，具体可参见“逗号表达式”）。之后，用 while 语句为它的每一个元素赋值。

```
# include <stdio.h>

void f (void)
{
    printf ("->");
    int n, a [(scanf ("%d", & n), n)];
    while (n --) a [n] = n;
}
```

需要说明的是库函数 scanf，它是在头文件<stdio.h>中声明的，用于从标准输入读取内容，它的原型如下。

```
int scanf (const char * restrict format, ...);
```

其中，format 所指向的串指定了哪些输入是可接收的，以及如何转换。转换后的结果保

存在后面相应的参数中。在上例中，%d 意味着一个十进制有符号整数，输入的字符代码将被转换为整数。

如果出错，则 scanf 的返回值是 EOF，否则返回有多少个参数已接受了输入。

在下例中，while 语句的作用是移动指针 s，直至遇到字符串末端的空字符'\0'，然后将 s 定位在这个空字符之后。注意，while 语句的循环体是空语句。

```
void f (char * s)
{
    while (* s ++ != '\0') ;
    /* …… */
}
```

下面的代码是上面例子的另一种写法，只是 while 语句的控制表达式被固定为整型常量 1，由于这个原因，跳出循环的任务只能在循环体内完成（这里使用了 break 语句，具体可参见“break 语句”）。

```
void f (char * s)
{
    while (1)
        if (* s ++ == '\0') break;
    /* …… */
}
```

7.6.3　do 语句

对于 do 语句来说，对控制表达式的求值发生在循环体的每一次执行之后，也就是先执行循环体，并且只有在控制表达式的值为 1 时才进行下一次循环。因此，do 语句的循环体至少会执行一次。

下面的例子用于向标准输出写 10 个数字，对象 cnt 的初值为 0。因为程序的目的是向标准输出写 0～9，所以，使用 do 语句比使用 while 语句方便。

```
# include <stdio.h>

void f (void)
{
    int cnt = 0;

    do
        printf ("%d\n", cnt);
    while (++ cnt <= 9);
}
```

7.7 跳转语句

跳转语句的功能是令控制（程序的执行）无条件地转到当前程序中的指定（特定）位置。跳转语句的特点是跳出当前语句，就像日常生活中所谓的捷径。

跳转语句的语法形式如图 7-6 所示，它包括 goto 语句、continue 语句、break 语句和 return 语句，具体内容可参见其各自的词条。

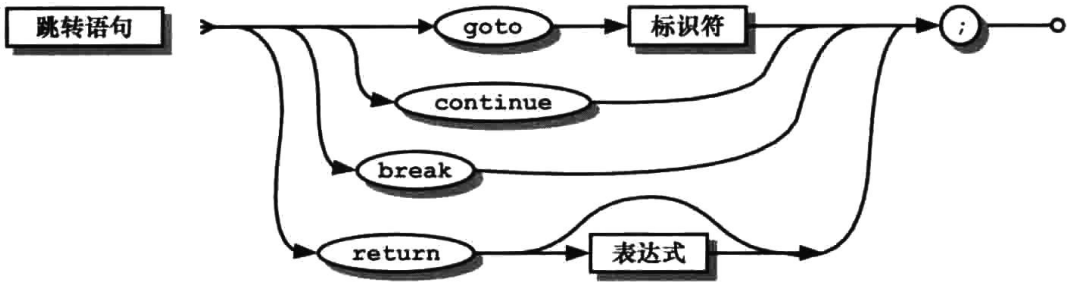


图 7-6 跳转语句的语法

7.7.1 goto 语句

关键字 goto 的后面跟着一个标识符，而且必须是一个标号的名称，这就是一个 goto 语句。这个标识符，以及使用这个标识符的 goto 语句必须位于同一个函数的内部，因为标号名是唯一具有函数作用域的标识符。

下面是一个使用 goto 语句的例子。函数 fn 接收一个参数 r，其功能是寻找任意 3 个小于等于 1000 的整数，要求它们的乘积为 r。如果没有找到符合条件的 3 个整数，则返回“假”值；如果找到了任意一个这样的组合，则对这 3 个整数进行某些处理，然后函数返回“真”值。

```
# include <stdbool.h>

bool fn (long long r)
{
    int x, y, z;

    for (x = 0; x <= 1000; x ++)
        for (y = 0; y <= 1000; y ++)
            for (z = 0; z <= 1000; z ++)
                if (x * y * z == r) goto lb_next;

    return false;
```

```
lb_next:
    /* ..... */
    return true;
}
```

这是 3 个嵌套的 for 语句，对 x、y 和 z 的检查工作在最内层的循环中进行。因为不需要找到符合条件的全部 x、y 和 z，因此，当找到第一个符合条件的组合时，离开这 3 个 for 语句，执行其后面的内容。此时，如果不使用 goto 语句，就要另想办法一层一层地退出这 3 个 for 语句。

不恰当地使用 goto 语句会导致一些问题。例如，它可能会跳过一些对象的初始化代码。在下例中，对象 i 和 j 在进入函数时创建，但只有在控制到达它们声明的位置时才会被初始化。但是很显然，goto 语句跳过了对 i 赋值的语句，也跳过了对 j 进行初始化的声明，所以在调用 printf 函数时，它们的存储值是不确定的。

```
# include <stdio.h>

void f (void)
{
    int i;

    goto lb_next;
    i = 7;
    /* ..... */
    int j = 8;
    /* ..... */

lb_next:
    printf("%d, %d\n", i, j);
}
```

如果声明了一个可变修改类型的标识符，而且企图从该标识符的作用域之外跳进它的作用域，那么这是 C 标准所禁止的，C 实现也不可能允许这样的做法。原因很简单，变长数组只在程序的执行到达其声明的位置时才会创建。在下面的例子中，请读者先自己判断哪些 goto 是允许的，哪些是非法的。

```
int n = 3;
goto lba;                    //S1
goto lbb;                    //S2
{
    goto lbb;                //S3
lba:;
    unsigned u [n];
lbb:;
```

```
        goto lba;           //S4
    lbc:;
        goto lbb;           //S5
    }
    goto lba;               //S6
    goto lbb;               //S7
```

标识符 u 的作用域从它被声明的地方开始，一直延伸到包围它的块结束。S1 是允许的，因为当前语句和标号 lba 都在 u 的作用域之外；S2 是非法的，因为尽管当前语句在 u 的作用域之外，但标号 lbb 在 u 的作用域之内；S3 是非法的，原因和 S2 一样；S4 和 S5 都是允许的，因为这两条语句都在 u 的作用域之内；S6 是允许的，原因和 S1 相同；S7 是非法的，原因和 S2 相同。

7.7.2 continue 语句

continue 语句只能位于迭代语句的循环体中，它导致的后果是跳转到当前正在执行的循环体末尾（从而准备开始下一轮循环）。

在下面这个例子中（假定对象 x 是无符号整数类型且拥有正值），那么，因为循环体中只有 continue 自己，所以它什么也不做，直接将控制转移到循环体末尾，准备开始下一轮循环。

```
while (x --) continue;                    //等同于while(x--);
```

下面这个实例用于根据数据分布的区间来做相应的处理工作。如果数值小于 60 则不做任何处理，直接处理下一个数值，使用 continue 语句可以方便地做到这一点。

```
# include <stddef.h>

int a[/* …… */];
/* …… */
for (size_t i = 0; i < sizeof a / sizeof * a; i ++)
    if (a [i] < 60) continue;             //先排除掉一大部分
    else if (a [i] <= 70) {/* …… */}
    else if (a [i] <= 80) {/* …… */}
    else if (a [i] <= 90) {/* …… */}
    else  {/* …… */}
```

经验表明，通过合理组织程序，可以在不使用 continue 语句的情况下完成同样的工作。但是，上例表明，continue 自有它的优势，可以使用户以执行起来更有效率的方式来组织程序。

7.7.3 break 语句

break 语句只能位于 switch 语句或者迭代语句中；它导致的后果是终止执行它直接隶属的 switch 或者迭代语句。

下面是两个 break 语句的用法示例，没有什么实际用处，只是为了说明问题。

```
switch (8) break;   //什么也不做
for (;;) break;     //同样什么也不做
```

又如下面的例子，函数 getchar 从标准输入接收一个字符，如果输入的是字符 q，则直接跳出 while 循环。

```
int c;
while ((c = getchar ()) != EOF)
    if (c == 'q') break;
    else {/* …… */}
}
```

7.7.4　return 语句

return 语句只能出现在函数体内，用于终止当前函数的执行，并将控制返回到它的调用者那里。

一个函数可以有多个 return 语句。如果该函数的返回类型是 void，则它内部的 return 语句只能是关键字 return 后跟一个分号；否则，关键字 return 后面要带一个表达式。

下面是两个 return 语句的例子。因为函数 f 的返回类型是 void，所以函数体内的 return 语句不包含表达式；相反，函数 g 的返回类型是 float，所以它必须返回一个 float 类型的值。相应地，它的 return 语句必须包含一个 float 类型的表达式。

```
void f (void)
{
    /* …… */
    return;
}

float g (void)
{
    /* …… */
    return 2.0f;
}
```

毫无疑问，函数调用表达式的类型和该函数的返回类型是一致的。此外，如果 return 语句是带有表达式的，要求表达式的类型和 return 语句所在的函数（的返回类型）一致；不一致的，必须转换该表达式的值，使之兼容于函数的返回类型。

在下面的例子中，函数 ret_demo 的返回类型是 signed long long int，而表达式 x ? f () : g () 的类型是 char，所以将隐式地从 char 类型转换为 signed long long，即：

```
extern char f (void), g (void);

signed long long ret_demo (int x)
{
    /* …… */
    return x ? f () : g ();
```

```
}
```

在下面的代码片断中，函数 double_bits 接收一个无符号整数类型的参数，然后判断组成这个值的那些比特是否具有以下特征：从左往右，或者从右往左，如果出现了值为 1 的比特，则在它之后应当连续出现奇数个这样的比特，被填充比特分隔的情况不予考虑（参见“填充比特”）；如果传入的参数符合该要求，则函数返回 1，否则返回 0。

举例来说，假定传入的是一个 unsigned char 且 CHAR_BIT==8，则，如果传入的值具有以下对象表示，则该值是符合要求的：

```
00000011
11001111
11111111
```

相反，下面的对象表示不符合要求：

```
00000001
10000001
00000111
```

在下面的代码片断中，函数体内存在两个 return 语句。

```
# include <stddef.h>
# include <limits.h>

typedef unsigned long long int UINT;

_Bool double_bits (UINT u)
{
   _Bool flg = 0;

   for (size_t t = 0; t < sizeof (UINT) * CHAR_BIT; t ++)
      if (u >> t & 1) flg = ! flg;
      else if (flg) return 0;

   return 1;
}
```

如果读者觉得上面的例子过于复杂，则可以分析下面的例子。如果 x 的值为 0，则返回 0；如果 x 的值大于 0，则返回 1；否则返回-1。

```
int f (int x)
{
   if (x > 0) return 1;
   else if (x) return -1;
   else return 0;
}
```

7.8 块

块是 C 的一个语法单位，用于将一些声明和语句组织在一起。从另一个层面来看，它是某些语句类型的统称。使用块的概念，会更有利于描述标识符的作用域、对象的存储期和生存期，等等。有关这方面的描述，可参见“选择语句”和“迭代语句”。

最显而易见的是，一个复合语句也是一个块。但是块不一定是复合语句。例如，选择语句和迭代语句也是块，但不是复合语句。

每个选择语句都在整体上形成一个块（属于包围它的块的子块）；每个选择语句的子语句也同样自成一个块（属于整个选择语句的子块）。

因为选择语句和迭代语句中的控制表达式可以包含具有副作用的复合字面值和类型声明，如果不对它们的作用域做出清晰的界定，则会造成一些混乱。从 C99 开始，明确地将整个选择语句及其子语句定义为块。

在下面的例子中，if 语句的控制表达式具有“声明了一个类型 struct t”的效果，但它的作用域在 C99 之前并不明确。

```
struct t {float f;};                                    //D1

void which_t (void)
{
    if (sizeof (struct t {int i;}) == 4)                //D2
    {
        struct t t0;            //本句中的 struct t 指的是 D2
        /* …… */
    }

    struct t t1;                //分析本句中的 struct t 指的是 D1 还是 D2?
    /* …… */
}
```

在最后一个声明中，t1 的类型到底指的是 D1 还是 D2？通常情况下，人们更多地倾向于 D2，但反对的人也有其道理。毕竟，这缺乏一个明确的说法。但从 C99 开始，标准明确地将选择语句作为一个块。因此，离开了 if 语句之后，D2 不再可见。在 C99 中，上面的代码等效于：

```
struct t {float f;};                                    //D1

void which_t (void)
{
    {
        if (sizeof (struct t {int i;}) == 4)            //D2
```

```
        {
            struct t t0;
            /* ..... */
        }
    }

    struct t t1;          //从C99开始，这里的struct t指的是D1
    /* ..... */
}
```

再来看一个复合字面值的例子，代码如下。

```
# include <stdio.h>

void f (void)
{
    char * p, * q;

    if ((p = & (char) {'?'}))
        q = & (char) {'y'};
    else
        q = & (char) {'n'};

    printf ("%c, %c\n", * p, * q);
}
```

首先，复合字面值的生存期仅限于它所在的块。在调用 printf 函数时，指针 p 和 q 原先所指向的对象还存在吗？通常，人们倾向于给出肯定的回答。但是，按照 C99 委员会的新说法，if 语句和它的每一个子语句都自成一个块。因此，上面的代码和下面的代码是等效的。

```
# include <stdio.h>

void f (void)
{
    char * p, * q;

    {
        if (* (p = & (char) {'?'}))
        {
            q = & (char) {'y'};
        }
        else
        {
            q = & (char) {'n'};
        }
```

```
    }

    printf("%c, %c\n", * p, * q);          //未定义行为
}
```

也就是说，从 C99 开始，一旦离开了 if 语句，将不再保证指针 p 和 q 依然指向一个有效的对象。

每个迭代语句都在整体上形成一个块（属于包围它的那个块的子块）；每个迭代语句的循环体也同样自成一个块（属于整个迭代语句的子块）。

对此的解释和前面所说的选择语句一样。因为这个原因，迭代语句内声明的标识符在迭代语句外部是不可见的。在下例中，因为标识符 sum 在 for 语句内声明，所以离开 for 语句之后，它不再可见。

```
# include <stdio.h>

void f (void)
{
    /* 假定在此处看不到标识符 sum 的其他声明 */
    for (int i = 1, sum = 0; i <= 100; sum += i ++) ;
    printf ("%d\n", sum);    //非法
}
```

要解决这个问题，正确的代码如下。

```
# include <stdio.h>

void f (void)
{
    int sum = 0;
    for (int i = 1; i <= 100; sum += i ++) ;
    printf ("%d\n", sum);
}
```

块可以嵌套，但层数有限制。标准的要求是保证不少于 127 层，是否会多于这个数量，取决于 C 实现。

第 8 章　预处理指令

在程序转换阶段，C 程序的原始文本称为源文件，源文件经预处理后得到转换单元，具体可参见“程序转换”。

预处理工作由预处理程序或者说预处理器来完成，它可以是一个独立的程序，也可以不是。预处理的主要任务是执行预处理指令。

预处理指令包括条件包含、源文件包含、宏替换、行控制、抛错、杂注和空指令，具体可参见其各自的词条。

每个预处理指令以预处理记号“#”开头，后面跟着一系列其他预处理记号，最后结束于“#”后面的第一个换行符。

但是，前面带“#”的并非都是预处理指令，只有在以下两种情况下，以“#”引导的记号序列才是预处理指令：

（1）预处理记号“#”的前面没有其他字符，或只有空白（参见“空白字符”）。在这种情况下，说明“#”出现在源文件的开头。

（2）预处理记号“#”的前面是空白，且这些空白中至少有一个换行符。在这种情况下，“#”位于一个新行上，但不一定是该行的第一个字符。

假定下面的例子是一个源文件的开始部分。

```
# include <stdio.h>
  # define T(x) #x "??"
```

一般来说，空白在文本编辑器中是看不到的，所以这里用两个特殊的符号来指示它们的存在：“□”代表任何空白字符，“↵”代表换行符。假定以上代码在用这两个符号显示出空白字符之后是这个样子的：

```
#□include□<stdio.h>□□□□□↵
□□#□define□T(x)□#x□"??"↵
```

显然，第一行的尾部是 5 个空白字符和一个换行符；第二行的开头是两个空白字符，这可能是程序员在无意中留下的。

在预处理阶段，预处理器看到的不是我们眼中的“行”，它只看到记号和空白字符的序列。于是，它会看到这个序列的第一个字符是“#”，所以它将此“#”引导的记号序列视为第一个预处理指令，且该预处理指令结束于它后面的第一个换行符。

预处理器继续工作，用于结束第一个预处理指令的换行符，连同后面的两个空白，都位于第二个“#”的前面，依据上面的描述，这个“#”也引导一个预处理指令。

显然，第三个“#”引导的记号序列并不是一个预处理指令。

8.1 源文件包含

源文件包含（source file inclusion）指令的功能是搜索指定的文件，并将它的内容包含进来，放在它当前所在的位置。

源文件包含指令的格式如图 8-1 所示。

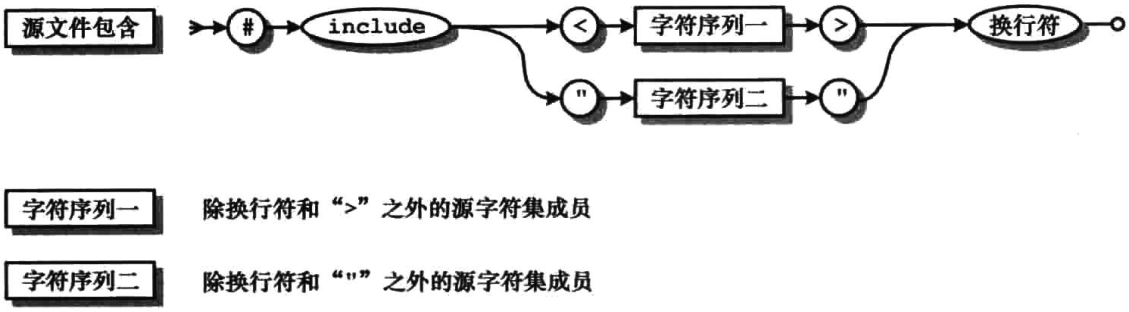


图 8-1　源文件包含指令的语法

以上，“字符序列一”和“字符序列二”用于组成被包含的文件名，例如：

```
# include <stdio.h>
# include "myfile.h"
```

组成文件名的字符必须是源字符集的成员，是否区分大小写由 C 实现自行决定。通常情况下，它是数字字符或非数字字符组成的序列，外加一个“.”，以及任何一个单一的非数字字符，如 head0.h。

如果采用第一种格式，组成文件名的字符不能是换行符和“>”；如果采用第二种格式，则不能是换行符和“"”。无论在哪种情况下，第一个字符都不能是数字。

这两种格式的不同之处在于如何搜索指定的文件，但这都取决于 C 实现。通常情况下，如果给出的是标准库文件或者由 C 实现提供的其他文件，则应使用第一种格式。在这种情况下，C 实现通常会在特定的系统路径内搜索。但是，系统路径具体是指文件系统内的哪些位置，会随 C 实现不同而异。

如果给出的是自定义的文件，如自己编写的文件，则应使用第二种格式。在这种情况下，C 实现通常会在当前目录中搜索。如果找不到，则它将按第一种格式处理。

C 实现可能允许在上述的“文件”中包含路径，也可能允许通过环境参数或者编译选项来直接提供搜索路径。在这种情况下，C 实现将按这些指定的位置搜索。具体的做法需要参考编译器手册。

文件包含的过程可以是递归的。也就是说，被包含进来的文件内容中也有文件包含指令，但标准只保证 15 层（含）以内的嵌套是允许的。

文件可能会出现彼此包含的情况。在这种情况下，同一个文件可能会在某个源文件内被包含多次。显然，重复出现的内容（如标识符）会在程序转换期间被视为非法。

避免发生这种情况的做法是使用条件包含。如果源文件 main.c 包含了头文件 fish.h 和 shrimp.h，而这两个文件也包含了 aquatic.h，那么，头文件 aquatic.h 的内容可以是这样的：

```
# ifndef AQUATIC_H_
# define AQUATIC_H_

/* 在此书写aquatic.h的其他内容 */

# endif // AQUATIC_H_
```

在预处理阶段，C 实现先处理 main.c，然后，不管是通过头文件 fish.h 还是 shrimp.h，头文件 aquatic.h 的内容会有两次被包含进来的机会。第一次被包含时，要执行它的预处理指令，此时宏 AQUATIC_H_尚未定义。因此，C 实现将插入#ifndef 和#endif 之间的内容（包括这个宏定义），然后执行其中的预处理指令，此时将定义宏 AQUATIC_H_。第二次包含头文件 aquatic.h 时，宏 AQUATIC_H_已经定义，此时将不会包含#ifndef 和#endif 之间的内容。

8.2 宏替换

宏的作用是把一个标识符（预处理记号）指定为其他一些称为替换列表的预处理记号，当这个标识符出现在后面的文本中时，将用对应的替换列表把它替换掉。可以定义两种形式的宏，分别是对象式宏、函数式宏，具体可参见“对象式宏定义”和“函数式定义”。

替换列表前后的空白和替换列表没有关系，它们不是替换列表的组成部分，也不会随替换列表一起出现在被替换的位置。

因此，这意味着

```
# define STR      "10086 "
char a [] = STR;
```

在宏替换之后为

```
char a [] = "10086 ";
```

而不是

```
char a [] =      "10086 ";
```

宏定义中的替换列表可以为空；宏名可以是关键字。但是，除非确实需要，一般不建议这样做。例如，如果用户拿到的程序中有 C 最近才引入的关键字_Noreturn，但用户的编译器并不支持，则为了以最小的代价使代码顺利编译，可以使用如下代码。

```
# define _Noreturn
```

宏不允许重复定义，除非它们的替换列表完全等价，对于函数式宏定义来说，它们的参数在数量上相同，对应的标识符拼写一致。

两个替换列表完全等价的前提是它们包含相同数量的记号序列，顺序和拼写一致，均以空白分隔。

在下面的例子中，字体加粗的宏定义都是非法的，有些是因为替换列表不等价，有些是因为参数不一致。

```
# define A
# define A
```

```
# define A 3

# define F(x) x ++
# define F(x) x++
# define F(x, y) x + y
# define F(x) x ++
```

宏定义的可见性是从它定义之后开始的，一直到用#undef 指令取消定义。如果未取消定义，则可见性延伸到当前源文件末尾。

可以用#undef 指令取消先前的宏定义。该指令的格式为“#”及“undef”，后面跟一个标识符。如果该标识符以前被定义为宏，则它不再是一个宏的名称；否则没有任何效果。

下面是一个示例。

```
# define N 120
int a [N];
# undef N
# define N 30
```

一旦替换列表中的形参被替换，且替换列表中的#和##也处理完成，则预处理器将重新扫描结果序列及后续的源文件内容，以期找到其他可以替换的宏名。

例如：

```
# define A B
# define B C
# define C int x

A;
```

以上代码的最后一行在扩展之后结果如下：

```
int x;
```

又如：

```
# define F(x) G(x)*2
# define G(y) F(y)*5

F(a);
```

则上述代码的最后一行在扩展之后结果如下：

```
F(a)*5*2;
```

宏扩展不是递归的。如果在一个宏名的替换列表中出现了它自己，则不管是直接的，还是通过另一个宏间接引入的，都不再扩展当前这个宏名。

例如：

```
# define A A + B + 5
# define B A*2

A;
```

以上代码的最后一行在扩展之后结果如下：

```
A + A*2 + 5;
```

同时，扩展到此为止。

同样地，即使宏在扩展后的结果是一个有效的预处理指令，它也不会被执行，除了预处理运算符_Pragma（参见“杂注”）。

8.2.1 对象式宏定义

如果一个预处理指令具有如图 8-2 所示的形式，则它定义了一个宏。这种形式的宏定义称为对象式宏定义。

图 8-2 对象式宏定义的语法

“标识符”是宏的名称，简称“宏名”；“替换列表”是宏的内容或者宏体。当宏名出现在宏定义之后的文本序列中时，会被替换为宏体。但是，出现在字符常量和字面串中的宏名不会被替换，因为字符常量和字面串也是预处理记号，而且有特殊的语义。

标识符和替换列表之间只能是空白（参见“空白字符”）。

下面是一个对象式宏定义的例子，PI 是宏名，3.14159 是替换列表。

```
# define PI 3.14159

float squar (float r)
{
    return PI * r * r;
}
```

下面是另一个例子，它表明替换列表可以有多个（用空白分隔的）记号；同时，它也表明字面串中的宏名不会被替换。

```
# include <stdio.h>

# define ID x, y, z

void f (void)
{
    int ID;
    x = y = z = 10086;
    printf ("%s, %d, %d, %d\n", "ID", x, y, z);
}
```

8.2.2 函数式宏定义

如果一个预处理指令具有如图 8-3 所示的形式之一，则它定义的是一个具有参数的宏，称之为函数式宏定义。

函数式宏定义可以有参数，下文中称为“形参”。参数位于宏名之后的一对括号中，即图中的标识符列表和“...”，参数之间用逗号分隔。

如果具有标识符列表，则每个标识符（参数）的作用域从它出现的位置开始，延伸到当前预处理指令的结尾（换行符）。

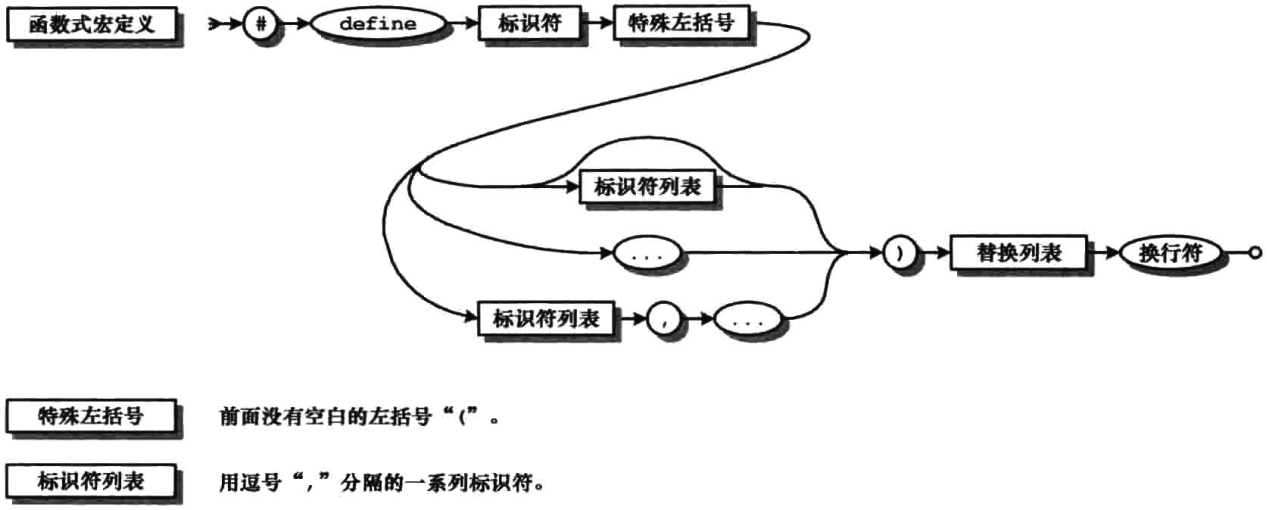


图 8-3 函数式宏定义的语法

正如图中所示，宏名和位于它右边的左括号之间不能有空白，否则左括号及它后面的记号都被视为替换列表。换句话说，这不是一个函数式宏定义，而是对象式宏定义。

函数式的宏一经定义，一旦该定义之后的文本序列中出现此宏名及一个左括号，则被视为一个宏调用。如果宏的形参出现在宏的替换列表中，且它的前面不是“#”和“##”，后面也没有“##”，则替换列表中的这个参数在内部的宏都扩展之后被替换为传入的实参，即参数替换（Argument substitution）。

下面是一个函数式宏及其使用的例子。

```
# define MUL(x, y) (x) * (y)

int f (int m, int n)
{
    return MUL(m + 1, n + 1);
}
```

在这个例子中，宏 MUL 在定义时有明确的参数数量，而在宏调用时也给出了足够的参数，因此，

```
return MUL(m + 1, n + 1);
```

被扩展（替换）为

```
return (m + 1) * (n + 1);
```

显然，在替换列表中正确而恰当地使用括号是很重要的，如果宏 MUL 被定义为

```
# define MUL(x, y) x * y
```

那么，

```
return MUL(m + 1, n + 1);
```

会被扩展（替换）为

```
return m + 1 * n + 1;
```

这肯定是违背了我们的原意。

函数式宏定义中的宏名和它后面的左括号“(”之间不能有空白，否则宏名后面的内容都被视为替换列表。

举例来说，如果宏 ADD 被定义为

```
# define ADD (x, y) (x) + (y)
```

那么，宏名“ADD”后面的(x, y) (x) + (y)被认为是替换列表，且 ADD 是一个对象式宏。在这种情况下，以下含有宏名的文本

```
ADD(m + 1, n + 1);
```

被扩展（替换）为

```
(x, y) (x) + (y)(m + 1, n + 1);
```

字体加粗的部分来自宏 ADD 的替换列表。

在数量上，宏调用时提供的实际参数必须和宏定义中的形式参数一致，除非宏定义时的参数列表是“...”，或者以“,...”结尾。

如果宏在定义时的参数列表是以“,...”结尾的，则宏调用时必须给出足够的参数，且必须等于或多于标识符列表中的标识符个数。

就上例而言，宏 MUL 需要两个参数，所以下面的调用是非法的：

```
MUL(a, b, c);
```

宏调用时，位于宏名之后、且位于最外层的那一对括号用于包围实际参数。换句话说，这对括号中的内容是宏调用的实际参数。

因此，如果传递的参数是一个逗号表达式，或者要传递的参数是逗号本身，则你可以将它们括起来；位于内部括号中的逗号不用于分隔参数。例如，

```
MUL((a, b), c);
```

如果要将逗号本身作为参数传递，则要用括号括起来，即

```
MUL(a, (,));
```

从 C99 开始，C 语言支持没有参数的函数式宏定义，以模拟不带参数的函数。从 C99 开始，C 语言支持没有参数的函数式宏定义，以模拟不带参数的函数。这允许用户在定义一个函数的同时提供一个同名的宏。例如：

```
# include <stdio.h>

void prdigit (void)
{
   printf ("0123456789\n ");
}

# define prdigit() printf ("0123456789\n ")

void f (void)
{
   prdigit();
```

```
    (prdigit)();
}
```

在这个例子中，函数名和宏名都称为 prdigit，必须把宏放在函数定义的后面，否则函数定义中的函数名会被替换。

如果要执行实际的函数调用，则必须将函数名用括号括起来。这样，在预处理阶段，因为包围实参的左括号“(”之前有一个右括号“)”，于是，标识符之后并不是执行宏调用所期望的左括号，这样的标识符不会被识别为宏名。

因此，上例中，函数 f 会被扩展（替换）如下（用括号括住的函数名或者说函数指示符是一个基本表达式，参见“基本表达式”）：

```
void f (void)
{
    printf ("0123456789\n");
    (prdigit)();
}
```

如果宏在定义时没有参数（标识符列表为空，且没有“,...”），而在调用时给出了参数，同样是非法的。就上例而言，如下的宏调用是不允许的。

```
prdigit(a);
```

如果函数式宏在定义时有参数，但仅仅是一个“如果函数式宏在定义时有参数，但仅仅是一个“...”，则定义的是一个具有可变参数的宏。当调用这样的宏时，C 实现会将所有实参，以及分隔它们的逗号，打包成一个参数传递给宏。为此，函数式宏定义中的替换列表中应当包含“__VA_ARGS__”，它被视为一个形参，宏调用时传入的可变参数将打包传递给它。如果替换列表中没有__VA_ARGS__，则无法接收任何参数。

为了接收打包后的参数，宏在定义时，其替换列表中要有__VA_ARGS__，例如

```
# define MPARA(...) __VA_ARGS__
```

那么，宏调用

```
MPARA(int x = 0, y = 1);
```

会被扩展（替换）为

```
int x = 0, y = 1;
```

当然，除了__VA_ARGS__之外，宏定义的替换列表中还会有其他内容，例如：

```
# define MPARA(...) char c; __VA_ARGS__; float f;
```

在这种情况下，如果有一个宏调用

```
MPARA(int x = 0, y = 1)
```

则它被扩展（替换）为

```
char c; int x = 0, y = 1; float f;
```

如果函数式宏定义的形参仅仅是“...”，且替换列表中没有__VA_ARGS__，则无法接收任何参数。所以，对于宏定义

```
# define MPARA(...) char c;
```

宏调用

```
MPARA(int x);
```

会被扩展为

```
char c;
```

如果函数式宏在定义时有参数，且为一个标识符列表和“,...”，则定义的是一个具有可变参数的宏，且这个宏有固定参数。

当调用这样的宏时，C 实现先替换固定参数，然后将剩余的参数，以及分隔它们的逗号，打包成一个参数传递给替换列表中的形参__VA_ARGS__。

为了接收打包后的参数，宏在定义时应当使用__VA_ARGS__，如

```
# define MPARA(x, y, ...) int x, y; __VA_ARGS__
```

如果按下面的方式调用宏：

```
MPARA(eggs, rocks, char a [] = "London");
```

则宏将被扩展（替换）为

```
int eggs, rocks; char a [] = "London";。
```

当然，如果没有__VA_ARGS__，则无法接收剩余的参数。如果有多个__VA_ARGS__，则剩余的参数会打包在一起，传递给每一个__VA_ARGS__。例如，对于宏定义

```
# define MPARA(x, y, ...) int x, y; __VA_ARGS__; __VA_ARGS__
```

则宏调用

```
MPARA(eggs, rocks, char a [] = "London");
```

会被扩展（替换）为

```
int eggs, rocks; char a [] = "London"; char a [] = "London";
```

当然了，因为标识符 a 被定义了两次，因此在编译阶段被视为非法。

注意，如果在定义一个函数式宏的时候，其参数中没有使用“...”，则它的替换列表中也不允许出现__VA_ARGS__。正因如此，下面的示例是非法的。

```
# define M(x, y) x, y, __VA_ARGS__
```

在用于指定宏参数的标识符列表中，不允许存在相同的标识符。下面分别是合法与非法的例子。

```
# define PF(x, x) x * x          //非法
# define PF(x, y) x * x + y * y  //合法
```

8.2.2.1 记号串化（#）

记号串化可以将函数式宏定义中的实参转换为字符串。为了做到这一点，在替换列表中的形参必须紧跟在预处理记号“#”之后。

在函数式宏定义中，如果替换列表中有“#”，则其后面的预处理记号必须是当前宏的形参。在预处理期间，“#”连同它后面的形参一起被用实参形成的字符串取代。

正因如此，下面的示例中，字体加粗的宏定义是非法的。

```
# define F(x) #x
# define F(x) #y
```

对于上面的合法宏定义，宏调用

```
char a [] = F(powerful processor);
```

被扩展为

```
char a [] = "powerful processor";
```

如果传入的实参中间有空白，则不管有多少，都被转换为一个空格；参数开头和末尾的空白都被删除。

例如，对于上面那个合法的宏定义，宏调用

```
char a [] = F(powerful    processor);
```

被扩展为

```
char a [] = "powerful processor";
```

再如，宏定义为

```
# define G(x, y, z) #x#y#z
```

则宏调用

```
char a [] = G( Long , long , ago );
```

被扩展为

```
char a [] = "Long""long""ago";
```

也正是因为这样，空的参数会被扩展为空字符串。例如，对于上述宏定义，宏调用

```
G( , ,)
```

将被扩展为

```
""""""
```

如果传递的实参是字符常量或者字面串，且内部有“"”或者“\”，则宏扩展时会在它们的前面放置一个“\”；如果传递的实参为字面串，则会在字面串两端的“"”之前放置一个“\”。

唯一的例外是统一字符名，如果“\”用于引导一个统一字符名，则不能肯定是否会在它前面放置一个“\”，标准把决定权留给 C 实现。

正是因为如此，对于上述宏定义，宏调用

```
char a [] = G(%\%, '\"', "\n\"");
```

会被扩展为

```
char a [] = "%\%""'\\\"'""\"\\n\\\"\"";
```

如果宏在定义时有多个参数，那么各参数之间的记号串化顺序是不确定的，不要做任何依赖其串化顺序的事情。

记号串化也适用于可变参数，例如：

```
# define FV(x, ...) #x #__VA_ARGS__
char a [] = FV(  **  ,  %%  ,  ^^  ,  @@  );
```

上述第二行在扩展之后变为

```
char a [] = "**" "%% , ^^ , @@";
```

8.2.2.2 记号黏接（##）

记号黏接的作用是将几个预处理记号合并为一个。在一个函数式宏定义中，如果一个预处理记号的前面或者后面有“##”，则该记号将与它前面或者后面的记号合并；如果该预处理记号是宏的形参，则用实参执行合并。

例如：

```
# define F(x, y, z) x##y##r
char F(a, b, c);
```

上述第二行是宏调用，其扩展之后如下。

```
char abr;
```

如果在宏调用时传递的实参不对应任何预处理记号，则 C 实现会用一个占位符来做记号黏接操作，在整个黏接过程结束之后再将其删除。

也就是说，宏调用

```
char F(,,);
char F(a,,b);
```

会被扩展为

```
char ;
char ab;
```

需要注意的是，在函数式宏定义中，“##”不能位于替换列表的开头和结尾。如果宏在定义时有多个参数，那么各参数之间的记号黏接顺序是不确定的，不要做任何依赖其黏接顺序的事情。

有时，记号黏接之后会碰巧得到一个统一字符名。在这种情况下，程序的行为是未定义的。

例如，统一字符名\udefa 恰好是字符序列“C:\\udefaults”的一部分。在下例中，用两个宏初始化数组，使用宏的本意是只有驱动器名不同。但是，因为上面提到的原因，程序的行为是不能预料的，包括有可能无法顺利转换。

```
# define QUOTE(x) #x
# define PATHNAME(d) QUOTE(d##udefaults)

void f (void)
{
    char af1 [] = PATHNAME(C:\\);
    char af2 [] = PATHNAME(D:\\);
    /* ..... */
}
```

8.2.3 预定义宏

下面的宏是强制性的，每个依从C标准的实现都必须定义。

（1）__DATE__ 对源文件进行预处理的日期。这是一个字面串，其内容依次为月、日和年份。

（2）__TIME__ 对源文件进行预处理的时间。这是一个字面串，其内容依次为时、分和秒。

（3）__FILE__ 当前源文件的名称，是一个字面串。

（4）__LINE__ 当前行的行号。

（5）__STDC__　C 实现将它定义为整型常量 1，以表明自己是依从 C 标准的。

（6）__STDC_HOSTED__　如果当前实现是基于操作系统的，则将它定义成整型常量 1；否则定义为 0。基于操作系统的实现有完整的标准库。

（7）__STDC_VERSION__　当前 C 实现所支持的标准版本，被定义为整型常量。

相反，下面的预定义宏是可选的，不要求 C 实现必须定义（下面的列表也许并不完整，更完整的列表可参见标准文档）。

（1）__STDC_ISO_10646__　被定义为一个整型常量，指示一个年份和月。如果存在这个定义，则表示类型为 wchar_t 的宽字符，其编码遵循 ISO 10646 标准及其指定日期的修订和增补方案。

（2）__STDC_UTF_16__　被定义为整型常量 1，表示 char16_t 类型的值采用 UTF-16 编码方案。

（3）__STDC_UTF_32__　被定义为整型常量 1，表示 char32_t 类型的值采用 UTF-32 编码方案。

（4）__STDC_IEC_559__　被定义为整型常量 1，表示当前 C 实现依从 IEC 60559 浮点运算标准。

（5）__STDC_IEC_559_COMPLEX__　被定义为整型常量 1，表示当前 C 实现依从 IEC 60559 复数运算标准。

（6）__STDC_NO_ATOMICS__　被定义为整型常量 1，表示当前 C 实现不支持原子类型，且不提供头文件<stdatomic.h>。

（7）__STDC_NO_COMPLEX__　被定义为整型常量 1，表示当前 C 实现不支持复数类型，且不提供头文件<complex.h>。若定义了该宏，则不能定义__STDC_IEC_559_COMPLEX__。

（8）__STDC_NO_THREADS__　被定义为整型常量 1，表示当前 C 实现未提供头文件<threads.h>。

（9）__STDC_NO_VLA__　被定义为整型常量 1，表示当前 C 实现不支持变长数组和可变修改类型。

不允许在#define、#undef 指令中使用上述预定义宏。换句话说，这些宏既不能被重新定义，也不能取消定义。

8.3　条件包含

条件包含指令的功能是根据条件有选择性地保留或者放弃源文件中的某些内容，如图 8-4 所示，它们以#if、#elif、#ifdef 或者#ifndef 指令开始，结束于#endif，中间可以有#else 和#elif 指令，但每个完整的条件包含指令内部只能有一个#else 指令。

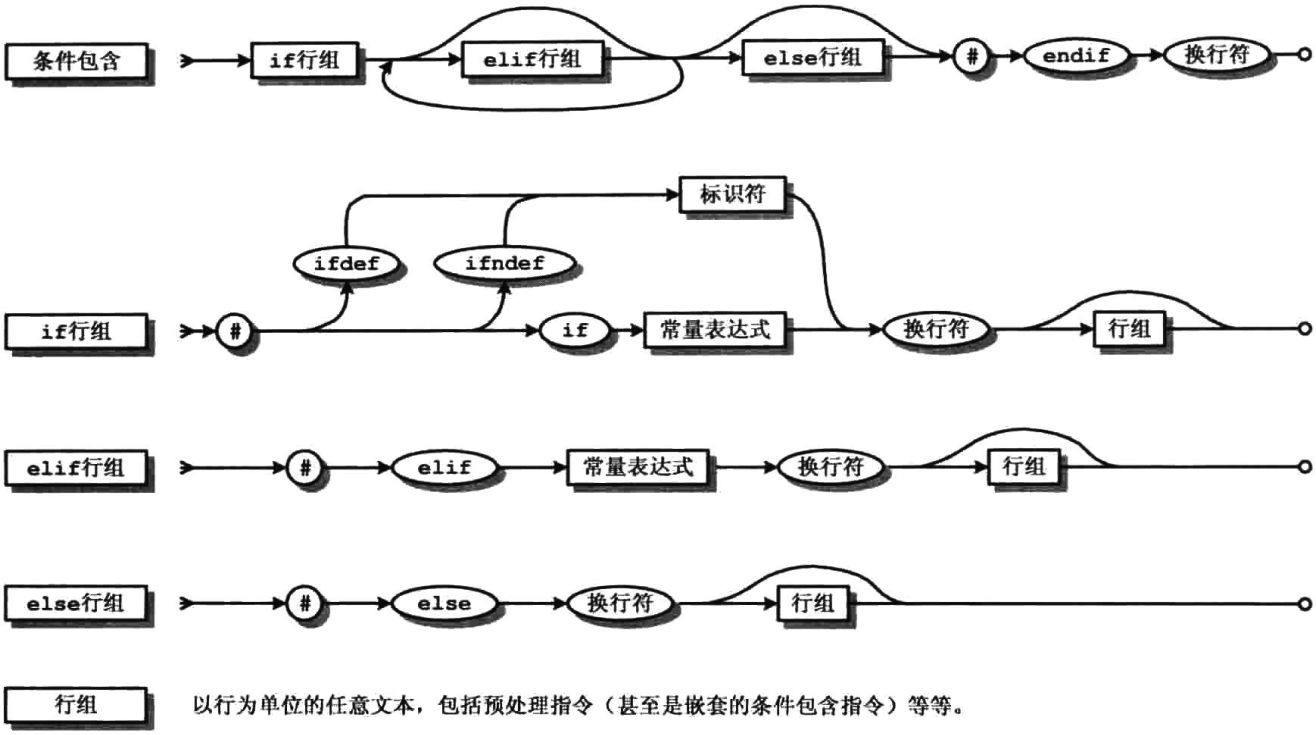


图 8-4　条件包含的语法

下面是一个条件包含的完整示例，从#indef 开始，中间有#elif 和#else，最后以#endif 指令结束。

```
# ifdef _INTEL_
# define _INTEL_
# include "intel.h"
char * getband (void) {return "intel chip.";}

# elif defined (_AMD_)
# define _AMD_
# include "amd.h"
char * getband (void) {return "amd chip.";}

# elif defined (_ARM_)
# define _ARM_
# include "arm.h"
char * getband (void) {return "arm chip.";}

# else
# define _OTHER_
# include "other.h"
char * getband (void) {return "unknow chip.";}
# endif
```

#if 和#elif 后面的预处理记号在宏扩展之后应当是一个常量表达式，也称控制表达式。但是，如果此记号是预处理运算符 defined 的操作数，则不执行宏扩展。在宏扩展时，凡是没有定义过的标识符都被替换为“0”。

预处理运算符 defined 只能出现在#if 和#elif 指令中，其格式如下（斜体部分需要用自己的内容来取代）。

```
defined 标识符
defined (标识符)
```

如果指定的标识符是一个已经用#define 指令定义过的宏名，且未被取消定义，则该运算符的结果是 1，否则为 0。

例如：

```
# if defined (INTEL) && defined (ARM)
/* …… */
# endif
```

#ifdef 和#ifndef 后面要求有一个标识符，这两个指令的功能是检查标识符是否为已经定义的宏名，即它们分别等价于（斜体部分需要用自己的内容来取代）：

```
#if defined 标识符
#if !defined 标识符
```

#if、#elif、#ifdef 和#ifndef 指令的处理过程如下：求值控制表达式或者预处理运算符 defined，若其值为 1，则保留同它直接关联的行组，其余到#endif 为止的行组都放弃；若控制表达式的值为 0，且存在#elif 指令，则依次求值每个#elif 指令，直到其中有一个控制表达式的求值为 1，且保留和该指令直接隶属的行组，放弃其他#elif 指令的行组；若所有控制表达式的值都为 0 且存在#else 指令，则保留该#else 指令直接关联的行组，放弃剩余行组；如果没有#else 指令，则放弃该#if 或者#elif 指令的所有行组。

8.4　行控制

在程序转换阶段，如果源代码有错误，则将终止转换，并给出诊断信息，通常还会给出错误所在行的行号。

但是，如果源文件在提交给预处理器之前被其他工具加工过，则这些工具可能会修改源文件，在其中加入自己生成的内容。在这种情况下，当程序转换失败时，所提示的行号相对于看到的源文件来说是不正确的。

为了避免这种情况，在将源文件提交给整个编译过程之前，可以在那些要被其他软件处理的文本后面使用行控制指令。

行控制指令以“#”和“line”引导，后面是行号和可选的字面串。行控制指令将下一行的行号指定为用户给出的值。

行号必须是十进制整数，不可以是 0，也不可以大于 2147483647，它用于改变预定义宏__LINE__的值；如果字面串存在，则用于改变预定义宏__FILE__的值。

如果在“line”之后的预处理记号中存在宏，则先扩展这些宏，再执行行控制指令。

下面是行控制指令的示例，在这个例子中，假定文本行

```
<%PCL: 3, "ab"%>
```

是被一个第三方工具识别和处理的控制命令，它将被扩展为若干行C源代码。

```
# include <stdio.h>

# define N 8

int main (void)
{
   <%PCL: 3, "ab"%>
   # line N "file00.c"
   return printf ("%d, %s", __LINE__, __FILE__);
}
```

8.5 抛错

抛错指令以“#”和“error”引导，后面是其他预处理记号，通常是错误信息。抛错指令用于在预处理期间发出一个诊断信息，再停止转换。抛错是人为的动作，如果你的源程序需要一些特殊的条件才能转换，比如，要求int类型的长度至少是32位，或者使用了一些新特性，但不确定编译器是否支持，都可以用预处理指令加以判断，如果不符合要求，则停止编译，并通过抛错指令告诉操作员具体的原因。

下面是一个抛错指令的示例。

```
# ifndef __STDC__
# error Not suitable platform.
# endif
```

如果不希望消息文本中的内容被宏错误地替换，则可以使用字面串。例如：

```
# error "Not suitable platform. "
```

任何时候，只要执行了抛错指令，程序的转换过程必须停止，而且程序的转换必须是以失败告终。

8.6 杂注

杂注指令用于向C实现传递额外的信息（编译选项），对程序转换的某些方面进行控制。一般地，如果这些选项是由C实现提供的，则它可以识别。

杂注指令以“#”开始，跟着“pragma”，后面是其他预处理记号，即所谓的选项。

例如，下面的杂注指令用于指示C实现将结构成员对齐到字节边界。

```
# pragma pack(1)
```

如果“pragma”后面紧跟着一个“STDC”，即“pragma”后面的记号是以“STDC”开头的，则意味着它是一个标准杂注，否则是一个实现定义的杂注。标准杂注不执行宏替换。

下面是一个标准杂注的例子。

```
# pragma STDC FP_CONTRACT ON
```

无法使用宏扩展生成#pragma 指令。如果有这种需求的话，则可以使用 C99 新增的一元预处理指令_Pragma，它用来建立一个杂注表达式，其语法形式如图 8-5 所示。

首先，在预处理阶段，C 的实现将字面串分解，转换为预处理记号的序列。其次，将编码前缀和包围字面串的双引号删除，用“""”代替“\"”，用“\”代替“\\”。最后，处理得到的字符序列，就像它们是#pragma 指令中出现的预处理记号一样。

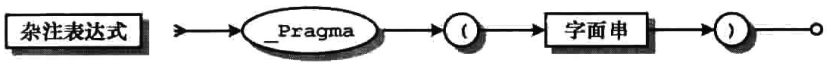


图 8-5　杂注表达式的语法

例如，表达式

```
_Pragma ("STDC FP_CONTRACT ON")
```

执行的结果就好比当前位置是下面的杂注指令：

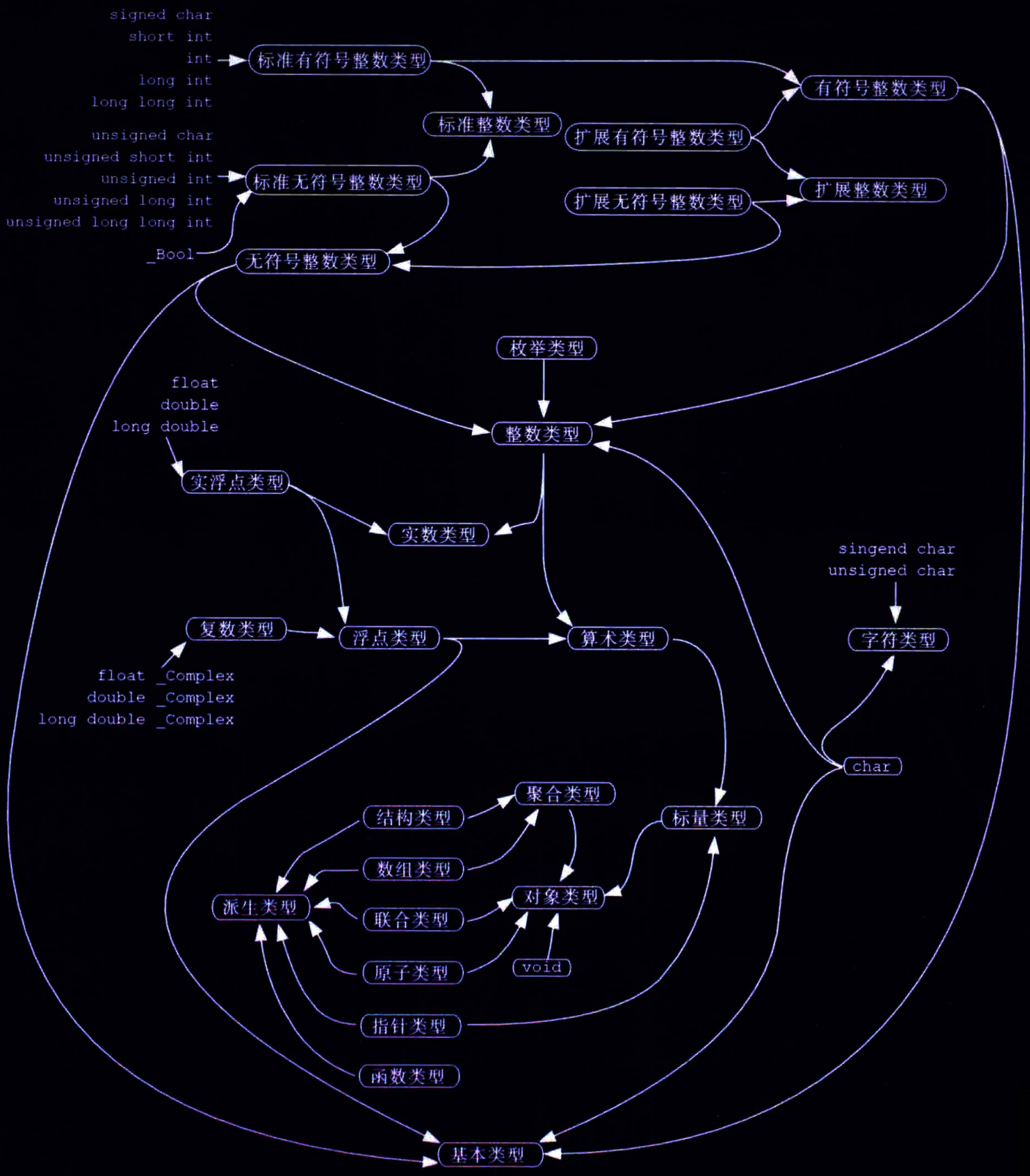
```
# pragma STDC FP_CONTRACT ON
```

8.7　空指令

空指令只有一个“#”，自成一行，即

```
#
```

也就是说，在结束该指令的换行符之前，除了空白和“#”之外，不存在其他任何记号。空指令的使用没有任何效果。



上架建议：C 语言
ISBN 978-7-121-27430-5
9 787121 274305 >
定价：58.00 元

策划编辑：董亚峰（dyf@phei.com.cn）
责任编辑：郝黎明
封面设计：朝天世纪