第3章 数据

程序对数据进行操作，本章将对数据进行描述。描述它的各种类型，描述它的特点以及如何声明它。本章还将描述变量的三个属性——作用域、链接属性和存储类型。这三个属性决定了一个变量的“可视性”（也就是它可以在什么地方使用）和“生命期”（它的值将保持多久）。

3.1 基本数据类型

在C语言中，仅有4种基本数据类型——整型、浮点型、指针和聚合类型（如数组和结构等）。所有其他的类型都是从这4种基本类型的某种组合派生而来。首先让我们来介绍整型和浮点型。

3.1.1 整型家族

整型家族包括字符、短整型、整型和长整型，它们都分为**有符号(singed)**和**无符号(unsigned)**两种版本。

听上去“长整型”所能表示的值应该比“短整型”所能表示的值要大，但这个假设并不一定正确。规定整型值相互之间大小的规则很简单：

长整型至少应该和整型一样长，而整型至少应该和短整型一样长。

K&R C：

注意，标准并没有规定长整型必须比短整型长，只是规定它不得比短整型短。ANSI 标准加入了一个规范，说明了各种整型值的最小允许范围，如表 3.1 所示。当各个环境间的可移植性问题非常重要时，这个规范较之 K&R C 就是一个巨大的进步，尤其是在那些机器的系统结构差别极大的环境里。

表 3.1　　变量的最小范围

类　　型	最小范围
char	0 到 127
signed char	–127 到 127

bbs.theithome.com

续表

类　　型	最小范围
unsigned char	0 至 255
short int	–32767 到 32767
unsigned short int	0 到 65535
int	–32767 到 32767
unsigned int	0 到 65535
long int	–2147483647 到 2147483647
unsigned long int	0 到 4294967295

short int 至少 16 位，long int 至少 32 位。至于缺省的 int 究竟是 16 位还是 32 位，或者是其他值，则由编译器设计者决定。通常这个选择的缺省值是这种机器最为自然（高效）的位数。同时你还应该注意到标准也没有规定这 3 个值必须不一样。如果某种机器的环境的字长是 32 位，而且没有什么指令能够更有效地处理更短的整型值，它可能把这 3 个整型值都设定为 32 位。

头文件 limits.h 说明了各种不同的整数类型的特点。它定义了表 3.2 所示的各个名字。limits.h 同时定义了下列名字：CHAR_BIT 是字符型的位数（至少 8 位）；CHAR_MIN 和 CHAR_MAX 定义了缺省字符类型的范围，它们或者应该与 SCHAR_MIN 和 SCHAR_MAX 相同，或者应该与 0 和 UCHAR_MAX 相同；最后，MB_LEN_MAX 规定了一个多字节字符最多允许的字符数量。

表 3.2　　变量范围的限制

	signed		unsigned
类　　型	最　小　值	最　大　值	最　大　值
字符	SCHAR_MIN	SCHAR_MAX	UCHAR_MAX
短整型	SHRT_MIN	SHRT_MAX	USHRT_MAX
整型	INT_MIN	INT_MAX	UINT_MAX
长整型	LONG_MIN	LONG_MAX	ULONG_MAX

尽管设计 char 类型变量的目的是为了让它们容纳字符型值，但字符在本质上是小整型值。缺省的 char 要么是 signed char，要么是 unsigned char，这取决于编译器。这个事实意味着不同机器上的 char 可能拥有不同范围的值。所以，只有当程序所使用的 char 型变量的值位于 signed char 和 unsigned char 的交集中，这个程序才是可移植的。例如，ASCII 字符集中的字符都是位于这个范围之内的。

在一个把字符当作小整型值的程序中，如果显式地把这类变量声明为 signed 或 unsigned，可以提高这类程序的可移植性。这类做法可以确保不同的机器中在字符是否为有符号值方面保持一致。另一方面，有些机器在处理 signed char 时得心应手，如果硬把它改成 unsigned char，效率可能受损，所以把所有的 char 变量统一声明为 signed 或 unsigned 未必是上上之策。同样，许多处理字符的库函数把它们的参数声明为 char，如果你把参数显式声明为 unsigned char 或 signed char，可能会带来兼容性问题。

提示：

当可移植问题比较重要时，字符是否为有符号数就会带来两难的境地。最佳妥协方案就是把存储于 char 型变量的值限制在 signed char 和 unsigned char 的交集内，这可以获得最大程度的可移植性，同时又不牺牲效率。并且，只有当 char 型变量显式声明为 signed 或 unsigned 时，才对它执行算术运算。

bbs.theithome.com

一、整型字面值

字面值(literal)[1]这个术语是字面值常量的缩写——这是一种实体，指定了自身的值，并且不允许发生改变。这个特点非常重要，因为 ANSI C 允许**命名常量**（named constant，声明为 const 的变量）的创建，它与普通变量极为类似。区别在于，当它被初始化以后，它的值便不能改变。

当一个程序内出现整型字面值时，它是属于整型家族 9 种不同类型中的哪一种呢？答案取决于字面值是如何书写的，但是你可以在有些字面值的后面添加一个后缀来改变缺省的规则。在整数字面值后面添加字符 L 或 l（这是字母 l，不是数字 1），可以使这个整数被解释为 long 整型值，字符 U 或 u 则用于把数值指定为 unsigned 整型值。如果在一个字面值后面添加这两组字符中的各一个，那么它就被解释为 unsigned long 整型值。

在源代码中，用于表示整型字面值的方法有很多。其中最自然的方式是十进制整型值，诸如：

```
123        65535        -275[2]
```

十进制整型字面值可能是 int、long 或 unsigned long。在缺省情况下，它是最短类型但能完整容纳这个值。

整数也可以用八进制来表示，只要在数值前面以 0 开头。整数也可以用十六进制来表示，它以 0x 开头。例如：

```
0173     0177777          000060
0x7b     0xFFFF           0xabcdef00
```

在八进制字面值中，数字 8 和 9 是非法的。在十六进制字面值中，可以使用字母 ABCDEF 或 abcdef。八进制和十六进制字面值可能的类型是 int、unsigned int、long 或 unsigned long。在缺省情况下，字面值的类型就是上述类型中最短但足以容纳整个值的类型。

另外还有字符常量。它们的类型总是 int。你不能在它们后面添加 unsigned 或 long 后缀。字符常量就是一个用单引号包围起来的单个字符（或字符转义序列或三字母词），诸如：

```
'M'          '\n'       '??('          '\377'
```

标准也允许诸如'abc'这类的多字节字符常量，但它们的实现在不同的环境中可能不一样，所以不鼓励使用。

最后，如果一个多字节字符常量的前面有一个 L，那么它就是**宽字符常量**(wide character literal)。如：

```
L'X'          L'e^'
```

当运行时环境支持一种宽字符集时，就有可能使用它们。

提示：

尽管对于读者而言，整型字面值的书写形式看上去可能相差甚远。但当你在程序中使用它们时，编译器并不介意你的书写形式。你将采用何种书写方式，应该取决于这个字面值使用时的上下文环境。绝大多数字面值写成十进制的形式，因为这是人们阅读起来最为自然的形式。但这也不尽然，这里就有几个例子，此时采用其他类型的整型字面值更为合适。

---

[1] 译注：在本书中，literal 这个词有时译为字面值，有时译为常量，它们的含义相同，只是表达的习惯不一。其中，string literal 和 char literal 分别译为字符串常量和字符常量，其他的 literal 一般译为字面值。

[2] 从技术上说，-275 并非字面值常量，而是常量表达式。负号被解释为单目操作符而不是数值的一部分。但是在实践中，这个歧义性基本没什么意义。这个表达式总是被编译器按照你所预想的方法计算。

bbs.theithome.com

当一个字面值用于确定一个字中某些特定位的位置时，将它写成十六进制或八进制值更为合适，因为这种写法更清晰地显示了这个值的特殊本质。例如，983040 这个值在第 16～19 位都是 1，如果它采用十进制写法，你绝对看不出这一点。但是，如果将它写成十六进制的形式，它的值就是 0xF000，清晰地显示出那几位都是 1 而剩余的位都是 0。如果在某种上下文环境中，这些特定的位非常重要时，那么把字面值写成十六进制形式可以使操作的含义对于读者而言更为清晰。

如果一个值被当作字符使用，那么把这个值表示为字符常量可以使这个值的意思更为清晰。例如，下面两条语句

```
value = value - 48;
value = value - \60;
```

和下面这条语句

```
value = value - '0';
```

的含义完全一样，但最后一条语句的含义更为清晰，它用于表示把一个字符转换为二进制值。更为重要的是，不管你所采用的是何种字符集，使用字符常量所产生的总是正确的值，所以它能提高程序的可移植性。

二、枚举类型

枚举(enumerated)类型就是指它的值为符号常量而不是字面值的类型，它们以下面这种形式声明：

```
enum Jar_Type { CUP, PINT, QUART, HALF_GALLON, GALLON };
```

这条语句声明了一个类型，称为 **Jar_Type**。这种类型的变量按下列方式声明：

```
enum Jar_Type milk_jug, gas_can, medicine_bottle;
```

如果某种特别的枚举类型的变量只使用一个声明，你可以把上面两条语句组合成下面的样子：

```
enum { CUP, PINT, QUART, HALF_GALLON, GALLON }
     milk_jug, gas_can, medicine_bottle;
```

这种类型的变量实际上以整型的方式存储，这些符号名的实际值都是整型值。这里 CUP 是 0，PINT 是 1，以此类推。适当的时候，你可以为这些符号名指定特定的整型值，如下所示：

```
enum Jar_Type { CUP = 8, PINT = 16, QUART = 32,
        HALF_GALLON = 64, GALLON = 128 };
```

只对部分符号名用这种方式进行赋值也是合法的。如果某个符号名未显式指定一个值，那么它的值就比前面一个符号名的值大 1。

提示：

符号名被当作整型常量处理，声明为枚举类型的变量实际上是整数类型。这个事实意味着你可以给 Jar_Type 类型的变量赋诸如–623 这样的字面值，你也可以把 HALF_GALLON 这个值赋给任何整型变量。但是，你要避免以这种方式使用枚举，因为把枚举变量同整数无差别地混合在一起使用，会削弱它们值的含义。

3.1.2 浮点类型

诸如 3.14159 和 6.023×10^{23} 这样的数值无法按照整数存储。第一个数并非整数，而第二个数远远超出了计算机整数所能表达的范围。但是，它们可以用浮点数的形式存储。它们通常以一个小数

bbs.theithome.com

以及一个以某个假定数为基数的指数组成，例如：

```
.3243F×16^1          .11001001000011111×2^2
```

它们所表示的值都是 3.14159。用于表示浮点值的方法有很多，标准并未规定必须使用某种特定的格式。

浮点数家族包括 float、double 和 long double 类型。通常，这些类型分别提供单精度、双精度以及在某些支持扩展精度的机器上提供扩展精度。ANSI 标准仅仅规定 long double 至少和 double 一样长，而 double 至少和 float 一样长。标准同时规定了一个最小范围：所有浮点类型至少能够容纳从 10^{-37} 到 10^{37} 之间的任何值。

头文件 float.h 定义了名字 FLT_MAX、DBL_MAX 和 LDBL_MAX，分别表示 float、double 和 long double 所能存储的最大值。而 FLT_MIN、DBL_MIN 和 LDBL_MIN 则分别表示 float、double 和 long double 能够存储的最小值。这个文件另外还定义一些和浮点值的实现有关的某些特性的名字，例如浮点数所使用的基数、不同长度的浮点数的有效数字的位数等。

浮点数字面值总是写成十进制的形式，它必须有一个小数点或一个指数，也可以两者都有。这里有一些例子：

```
3.14159        1E10        25.        .5        6.023e23
```

浮点数字面值在缺省情况下都是 double 类型的，除非它的后面跟一个 L 或 l 表示它是一个 long double 类型的值，或者跟一个 F 或 f 表示它是一个 float 类型的值。

3.1.3　指针

指针是 C 语言为什么如此流行的一个重要原因。指针可以有效地实现诸如 tree 和 list 这类高级数据结构。其他有些语言，如 Pascal 和 Modula-2，也实现了指针，但它们不允许在指针上执行算术或比较操作，也不允许以任何方式创建指向已经存在的数据对象的指针。正是由于不存在这方面的限制，所以，用 C 语言可以比使用其他语言编写出更为紧凑和有效的程序。同时，C 对指针使用的不加限制正是许多令人欲哭无泪和咬牙切齿的错误的根源。不论是初学者还是经验老道的程序员，都曾深受其害。

变量的值存储于计算机的内存中，每个变量都占据一个特定的位置。每个内存位置都由**地址**唯一确定并引用，就像一条街道上的房子由它们的门牌号码标识一样。指针只是地址的另一个名字罢了。指针变量就是一个其值为另外一个（一些）内存地址的变量。C 语言拥有一些操作符，你可以获得一个变量的地址，也可以通过一个指针变量取得它所指向的值或数据结构。不过，我们将在第 5 章才讨论这方面的内容。

通过地址而不是名字来访问数据的想法常常会引起混淆。事实上你不该被搞混，因为在日常生活中，有很多东西都是这样的。比如用门牌号码来标识一条街道上的房子就是如此，没有人会把房子的门牌号码和房子里面的东西搞混，也不会有人错误地给居住在“罗伯特·史密斯”的“埃尔姆赫斯特大街 428 号的先生”写信。

指针也完全一样。你可以把计算机的内存想象成一条长街上的一间间房子，每间房子都用一个唯一的号码进行标识。每个位置包含一个值，这和它的地址是独立且显著不同的，即使它们都是数字。

bbs.theithome.com

一、指针常量(pointer constant)

指针常量与非指针常量在本质上是不同的，因为编译器负责把变量赋值给计算机内存中的位置，程序员事先无法知道某个特定的变量将存储到内存中的哪个位置。因此，你通过操作符获得一个变量的地址而不是直接把它的地址写成字面值常量的形式。例如，如果我们希望知道变量 xyz 的地址，我们无法书写一个类似 oxff2044ec 这样的字面值，因为我们不知道这是不是编译器实际存放这个变量的内存位置。事实上，当一个函数每次被调用时，它的自动变量（局部变量）可能每次分配的内存位置都不相同。因此，把指针常量表达为数值字面值的形式几乎没有用处，所以 C 语言内部并没有特地定义这个概念[1]。

二、字符串常量(string literal)

许多人对 C 语言不存在字符串类型感到奇怪，不过 C 语言提供了字符串常量。事实上，C 语言存在字符串的概念：它就是一串以 NUL 字节结尾的零个或多个字符。字符串通常存储在字符数组中，这也是 C 语言没有显式的字符串类型的原因。由于 NUL 字节是用于终结字符串的，所以在字符串内部不能有 NUL 字节。不过，在一般情况下，这个限制并不会造成问题。之所以选择 NUL 作为字符串的终止符，是因为它不是一个可打印的字符。

字符串常量的书写方式是用一对双引号包围一串字符，如下所示：

```
"Hello"      "\aWarning!\a"      "Line 1\nLine2"          ""
```

最后一个例子说明字符串常量（不像字符常量）可以是空的。尽管如此，即使是空字符串，依然存在作为终止符的 NUL 字节。

K&R C：

在字符串常量的存储形式中，所有的字符和 NUL 终止符都存储于内存的某个位置。K&R C 并没有提及一个字符串常量中的字符是否可以被程序修改，但它清楚地表明具有相同的值的不同字符串常量在内存中是分开存储的。因此，许多编译器都允许程序修改字符串常量。

ANSI C 则声明如果对一个字符串常量进行修改，其效果是未定义的。它也允许编译器把一个字符串常量存储于一个地方，即使它在程序中多次出现。这就使得修改字符串常量变得极为危险，因为对一个常量进行修改可能殃及程序中其他字符串常量。因此，许多 ANSI 编译器不允许修改字符串常量，或者提供编译时选项，让你自行选择是否允许修改字符串常量。在实践中，请尽量避免这样做。如果你需要修改字符串，请把它存储于数组中。

我之所以把字符串常量和指针放在一起讨论，是因为在程序中使用字符串常量会生成一个“指向字符的常量指针”。当一个字符串常量出现于一个表达式中时，表达式所使用的值就是这些字符所存储的地址，而不是这些字符本身。因此，你可以把字符串常量赋值给一个“指向字符的指针”，后者指向这些字符所存储的地址。但是，你不能把字符串常量赋值给一个字符数组，因为字符串常量的直接值是一个指针，而不是这些字符本身。

如果你觉得不能赋值或复制字符串显得不方便，你应该知道标准 C 函数库包含了一组函数，它们就用于操纵字符串，包括对字符串进行复制、连接、比较以及计算字符串长度和在字符串中查找特定字符的函数。

---

[1] 有一个例外：NULL 指针，它可以用零值来表示。更多的信息请参见第 16 章。

bbs.theithome.com

3.2　基本声明

只知道基本的数据类型还远远不够，你还应该知道怎样声明变量。变量声明的基本形式是：

说明符（一个或多个）　声明表达式列表

对于简单的类型，声明表达式列表就是被声明的标识符的列表。对于更为复杂的类型，声明表达式列表中的每个条目实际上是一个表达式，显示被声明的名字的可能用途。如果你觉得这个概念过于模糊，不必担忧，我很快将对此进行详细讲解。

说明符(specifier)包含了一些关键字，用于描述被声明的标识符的基本类型。说明符也可以用于改变标识符的缺省存储类型和作用域。我们马上就将讨论这些话题。

在第 1 章的例子程序里，你已经见到了一些基本的变量声明，这里还有几个：

```
int  i;
char  j, k, l;
```

第 1 个声明提示变量 i 是一个整数。第 2 个声明表示 j、k 和 l 是字符型变量。

说明符也可能是一些用于修改变量的长度或是否为有符号数的关键字。这些关键字是：

```
short     long     singed     unsigned
```

同时，在声明整型变量时，如果声明中已经至少有了一个其他的说明符，关键字 int 可以省略。因此，下面两个声明的效果是相等的：

```
unsigned short int      a;
unsigned short          a;
```

表 3.3 显示了所有这些变量声明的变型。同一个框内的所有声明都是等同的。signed 关键字一般只用于 char 类型，因为其他整型类型在缺省情况下都是有符号数。至于 char 是否是 signed，则因编译器而异。所以，char 可能等同于 signed char，也可能等同于 unsigned char，表 3.3 中并未列出这方面的相等性。

浮点类型在这方面要简单一些，因为除了 long double 之外，其余几个说明符(short, signed, unsigned)都是不可用的。

表 3.3　　相等的整型声明

short short int	signed short signed short int	unsigned short unsigned short int
int	signed int signed	unsigned int unsigned
long long int	signed long signed long int	unsigned long unsigned long int

3.2.1　初始化

在一个声明中，你可以给一个标量变量指定一个初始值，方法是在变量名后面跟一个等号（赋值号），后面是你想要赋给变量的值。例如：

```
int  j = 15;
```


bbs.theithome.com

这条语句声明 j 为一个整型变量，其初始值为 15。在本章的后面，我们还将探讨初始化的问题。

3.2.2 声明简单数组

为了声明一个一维数组，在数组名后面要跟一对方括号，方括号里面是一个整数，指定数组中元素的个数。这是早先提到的声明表达式的第 1 个例子。例如，考虑下面这个声明：

```
int       values[20];
```

对于这个声明，显而易见的解释是：我们声明了一个整型数组，数组包含 20 个整型元素。这种解释是正确的，但我们有一种更好的方法来阅读这个声明。名字 values 加一个下标，产生一个类型为 int 的值（共有 20 个整型值）。这个“声明表达式”显示了一个表达式中的标识符产生了一个基本类型的值，在本例中为 int。

数组的下标总是从 0 开始，最后一个元素的下标是元素的数目减 1。我们没有办法修改这个属性，但如果你一定要让某个数组的下标从 10 开始，那也并不困难，只要在实际引用时把下标值减去 10 即可。

C 数组另一个值得关注的地方是，编译器并不检查程序对数组下标的引用是否在数组的合法范围之内[1]。这种不加检查的行为有好处也有坏处。好处是不需要浪费时间对有些已知是正确的数组下标进行检查。坏处是这样做将使无效的下标引用无法被检测出来。一个良好的经验法则是：

如果下标值是从那些已知是正确的值计算得来，那么就无需检查它的值。如果一个用作下标的值是根据某种方法从用户输入的数据产生而来的，那么在使用它之前必须进行检测，确保它们位于有效的范围之内。

我将在第 8 章讨论数组的初始化。

3.2.3 声明指针

声明表达式也可用于声明指针。在 Pascal 和 Modula 的声明中，先给出各个标识符，随后才是它们的类型。在 C 语言的声明中，先给出一个基本类型，紧随其后的是一个标识符列表，这些标识符组成表达式，用于产生基本类型的变量。例如：

```
int   *a;
```

这条语句表示表达式*a 产生的结果类型是 int。知道了*操作符执行的是间接访问操作[2]以后，我们可以推断 a 肯定是一个指向 int 的指针[3]。

警告：

C 在本质上是一种自由形式的语言，这很容易诱使你把星号写在靠近类型的一侧，如下所示：

```
int*  a;
```

这个声明与前面一个声明具有相同的意思，而且看上去更为清楚，a 被声明为类型为 int*的指针。但

---

1 从技术上说，让编译器准确地检查下标值是否有效是做得到的，但这样做将带来极大的额外负担。有些后期的编译器，如 Borland C++5.0，把下标检查作为一种调试工具，你可以选择是否启用它。

2 译注：indirection，也有译作间接寻址的，本书译为间接访问。

3 间接访问操作只对指针变量才是合法的。指针指向结果值。对指针进行间接访问操作可以获得这个结果值。更多的细节请参见第 6 章。

bbs.theithome.com

是，这并不是一个好技巧，原因如下：

```
int*  b, c, d;
```

人们很自然地以为这条语句把所有三个变量声明为指向整型的指针，但事实上并非如此。我们被它的形式愚弄了。星号实际上是表达式*b 的一部分，只对这个标识符有用。b 是一个指针，但其余两个变量只是普通的整型。要声明三个指针，正确的语句如下：

```
int   *b, *c, *d;
```

在声明指针变量时，你也可以为它指定初始值。这里有一个例子，它声明了一个指针，并用一个字符串常量对其进行初始化：

```
char  *message = "Hello world!";
```

这条语句把 message 声明为一个指向字符的指针，并用字符串常量中第 1 个字符的地址对该指针进行初始化。

警告：

这种类型的声明所面临的一个危险是你容易误解它的意思。在前面一个声明中，看上去初始值似乎是赋给表达式*message，事实上它是赋给 message 本身的。换句话说，前面一个声明相当于：

```
char   *message;
message = "Hello world! ";
```

3.2.4　隐式声明

C 语言中有几种声明，它的类型名可以省略。例如，函数如果不显式地声明返回值的类型，它就默认返回整型。当你使用旧风格声明函数的形式参数时，如果省略了参数的类型，编译器就会默认它们为整型。最后，如果编译器可以得到充足的信息，推断出一条语句实际上是一个声明时，如果它缺少类型名，编译器会假定它为整型。

考虑下面这个程序：

```
int   a[10];
int   c;
b[10];
d;

f( x )
{
      return x + 1;
}
```

这个程序的前面两行都很寻常，但第 3 和第 4 行在 ANSI C 中却是非法的。第 3 行缺少类型名，但对于 K&R 编译器而言，它已经拥有足够的信息判断出这条语句是一个声明。但令人惊奇的是，有些 K&R 编译器还能正确地把第 4 行也按照声明进行处理。函数 f 缺少返回类型，于是编译器就默认它返回整型。参数 x 也没有类型名，同样被默认为整型。

提示：

依赖隐式声明可不是一个好主意。隐式声明总会在读者的头脑中留下疑问：是有意遗漏类型名呢？还是不小心忘记写了？显式声明就能够清楚地表达你的意图。

bbs.theithome.com

3.3 typedef

C 语言支持一种叫作 typedef 的机制，它允许你为各种数据类型定义新名字。typedef 声明的写法和普通的声明基本相同，只是把 typedef 这个关键字出现在声明的前面。例如，下面这个声明：

```
char        *ptr_to_char;
```

把变量 ptr_to_char 声明为一个指向字符的指针。但是，在你添加关键字 typedef 后，声明变为：

```
typedef    char    *ptr_to_char;
```

这个声明把标识符 ptr_to_char 作为指向字符的指针类型的新名字。你可以像使用任何预定义名字一样在下面的声明中使用这个新名字。例如：

```
ptr_to_char       a;
```

声明 a 是一个指向字符的指针。

使用 typedef 声明类型可以减少使声明变得又臭又长的危险，尤其是那些复杂的声明[1]。而且，如果你以后觉得应该修改程序所使用的一些数据的类型时，修改一个 typedef 声明比修改程序中与这种类型有关的所有变量（和函数）的所有声明要容易得多。

提示：

你应该使用 typedef 而不是#define 来创建新的类型名，因为后者无法正确地处理指针类型。例如：

```
#define  d_ptr_to_char   char *
d_ptr_to_char   a, b;
```

正确地声明了 a，但是 b 却被声明为一个字符。在定义更为复杂的类型名字时，如函数指针或指向数组的指针，使用 typedef 更为合适。

3.4 常量

ANSI C 允许你声明常量，常量的样子和变量完全一样，只是它们的值不能修改。你可以使用 const 关键字来声明常量，如下面例子所示：

```
int        const     a;
const      int       a;
```

这两条语句都把 a 声明为一个整数，它的值不能被修改。你可以选择自己觉得容易理解的一种，并一直坚持使用同一种形式。

当然，由于 a 的值无法被修改，所以你无法把任何东西赋值给它。如此一来，你怎样才能让它在一开始拥有一个值呢？有两种方法：首先，你可以在声明时对它进行初始化，如下所示：

```
int        const  a = 15;
```

其次，在函数中声明为 const 的形参在函数被调用时会得到实参的值。

当涉及指针变量时，情况就变得更加有趣，因为有两样东西都有可能成为常量——指针变量和它所指向的实体。下面是几个声明的例子：

---

[1] typedef 在结构中特别有用，第 10 章有这方面的一些例子。

bbs.theithome.com

```
int        *pi;
```

pi 是一个普通的指向整型的指针。而变量

```
int    const    *pci;
```

则是一个指向整型常量的指针。你可以修改指针的值，但你不能修改它所指向的值。相比之下：

```
int    * const  cpi;
```

则声明 pci 为一个指向整型的常量指针。此时指针是常量，它的值无法修改，但你可以修改它所指向的整型的值。

```
int    const    * const  cpci;
```

最后，在 cpci 这个例子里，无论是指针本身还是它所指向的值都是常量，不允许修改。

提示：

当你声明变量时，如果变量的值不会被修改，你应当在声明中使用 const 关键字。这种做法不仅使你的意图在其他阅读你的程序的人面前得到更清晰的展现，而且当这个值被意外修改时，编译器能够发现这个问题。

#define 指令是另一种创建名字常量的机制[1]。例如，下面这两个声明都为 50 这个值创建了名字常量。

```
#define    MAX_ELEMENTS   50
int        const   max_eleemnts = 50;
```

在这种情况下，使用#define 比使用 cosnt 变量更好。因为只要允许使用字面值常量的地方都可以使用前者，比如声明数组的长度。const 变量只能用于允许使用变量的地方。

提示：

名字常量非常有用，因为它们可以给数值起符号名，否则它们就只能写成字面值的形式。用名字常量定义数组的长度或限制循环的计数器能够提高程序的可维护性——如果一个值必须修改，只需要修改声明就可以了。修改一个声明比搜索整个程序修改字面值常量的所有实例要容易得多，特别是当相同的字面值用于两个或更多不同目的的时候。

3.5　作用域

当变量在程序的某个部分被声明时，它只有在程序的一定区域才能被访问。这个区域由标识符的作用域（scope）决定。标识符的作用域就是程序中该标识符可以被使用的区域。例如，函数的局部变量的作用域局限于该函数的函数体。这个规则意味着两点。首先，其他函数都无法通过这些变量的名字访问它们，因为这些变量在它们的作用域之外便不再有效。其次，只要分属不同的作用域，你可以给不同的变量起同一个名字。

编译器可以确认 4 种不同类型的作用域——文件作用域、函数作用域、代码块作用域和原型作用域。标识符声明的位置决定它的作用域。图 3.1 的程序骨架说明了所有可能的位置。

[1] 第 14 章有完整的描述。

bbs.theithome.com

3.5.1 代码块作用域

位于一对花括号之间的所有语句称为一个代码块。任何在代码块的开始位置声明的标识符都具有**代码块作用域**(block scope)，表示它们可以被这个代码块中的所有语句访问。图 3.1 中标识为 6、7、9、10 的变量都具有代码块作用域。函数定义的形式参数（声明 5）在函数体内部也具有代码块作用域。

当代码块处于嵌套状态时，声明于内层代码块的标识符的作用域到达该代码块的尾部便告终止。然而，如果内层代码块有一个标识符的名字与外层代码块的一个标识符同名，内层的那个标识符就将隐藏外层的标识符——外层的那个标识符无法在内层代码块中通过名字访问。声明 9 的 f 和声明 6 的 f 是不同的变量，后者无法在内层代码块中通过名字来访问。

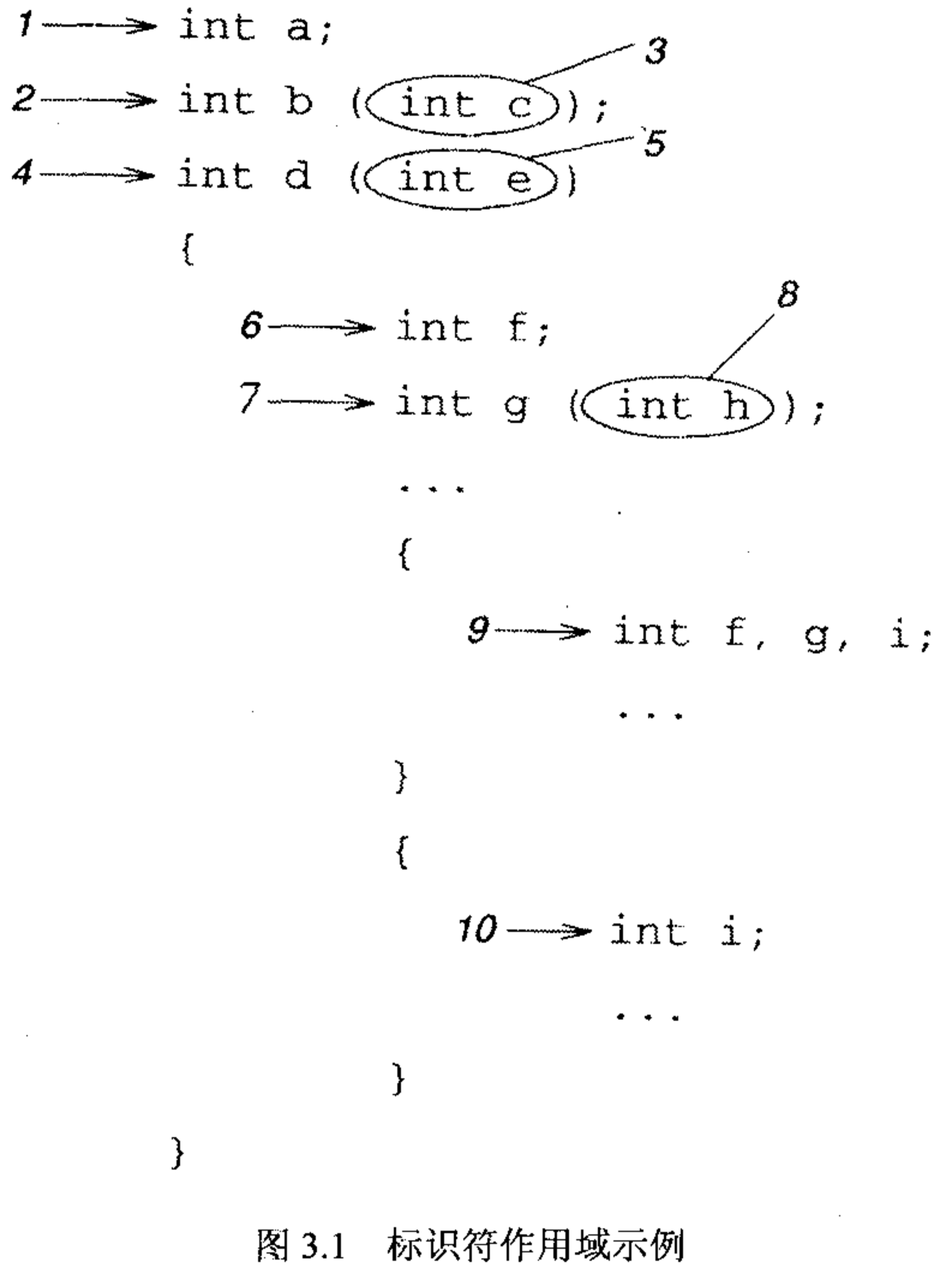

图 3.1　标识符作用域示例

提示：

你应该避免在嵌套的代码块中出现相同的变量名。我们并没有很好的理由使用这种技巧，它们只会在程序的调试或维护期间引起混淆。

不是嵌套的代码块则稍有不同。声明于每个代码块的变量无法被另一个代码块访问，因为它们的作用域并无重叠之处。由于两个代码块的变量不可能同时存在，所以编译器可以把它们存储于同一个内存地址。例如，声明 10 的 i 可以和声明 9 的任何一个变量共享同一个内存地址。这种共享并不会带来任何危害，因为在任何时刻，两个非嵌套的代码块最多只有一个处于活动状态。

bbs.theithome.com

K&R C：

在 K&R C 中，函数形参的作用域开始于形参的声明处，位于函数体之外。如果在函数体内部声明了名字与形参相同的局部变量，它们就将隐藏形参。这样一来，形参便无法被函数的任何部分访问。换句话说，如果在声明 6 的地方声明了一个局部变量 e，那么函数体只能访问这个局部变量，形参 e 就无法被函数体所访问。当然，没人会有意隐藏形参。因为如果你不想让被调用函数使用参数的值，那么向函数传递这个参数就毫无道理。ANSI C 扼止了这种错误的可能性，它把形参的作用域设定为函数最外层的那个作用域（也就是整个函数体）。这样，声明于函数最外层作用域的局部变量无法和形参同名，因为它们的作用域相同。

3.5.2　文件作用域

任何在所有代码块之外声明的标识符都具有**文件作用域**(file scope)，它表示这些标识符从它们的声明之处直到它所在的源文件结尾处都是可以访问的。图 3.1 中的声明 1 和 2 都属于这一类。在文件中定义的函数名也具有文件作用域，因为函数名本身并不属于任何代码块（如声明 4）。我应该指出，在头文件中编写并通过#include 指令包含到其他文件中的声明就好像它们是直接写在那些文件中一样。它们的作用域并不局限于头文件的文件尾。

3.5.3　原型作用域

原型作用域(prototype scope)只适用于在函数原型中声明的参数名，如图 3.1 中的声明 3 和声明 8。在原型中（与函数的定义不同），参数的名字并非必需。但是，如果出现参数名，你可以随你所愿给它们取任何名字，它们不必与函数定义中的形参名匹配，也不必与函数实际调用时所传递的实参匹配。原型作用域防止这些参数名与程序其他部分的名字冲突。事实上，唯一可能出现的冲突就是在同一个原型中不止一次地使用同一个名字。

3.5.4　函数作用域

最后一种作用域的类型是**函数作用域**(function scope)。它只适用于语句标签，语句标签用于 goto 语句。基本上，函数作用域可以简化为一条规则——一个函数中的所有语句标签必须唯一。我希望你永远不要用到这个知识。

3.6　链接属性

当组成一个程序的各个源文件分别被编译之后，所有的目标文件以及那些从一个或多个函数库中引用的函数链接在一起，形成可执行程序。然而，如果相同的标识符出现在几个不同的源文件中时，它们是像 Pascal 那样表示同一个实体？还是表示不同的实体？标识符的**链接属性**(linkage)决定如何处理在不同文件中出现的标识符。标识符的作用域与它的链接属性有关，但这两个属性并不相同。

链接属性一共有 3 种——external（外部）、internal（内部）和 none（无）。没有链接属性的标识符(none)总是被当作单独的个体，也就是说该标识符的多个声明被当作独立不同的实体。属于 internal 链接属性的标识符在同一个源文件内的所有声明中都指同一个实体，但位于不同源文件的多个声明则分属不同的实体。最后，属于 external 链接属性的标识符不论声明多少次、位于几个源文件都表示同一个实体。

bbs.theithome.com

图 3.2 的程序骨架通过展示名字声明的所有不同方式，描述了链接属性。在缺省情况下，标识符 b、c 和 f 的链接属性为 external，其余标识符的链接属性则为 none。因此，如果另一个源文件也包含了标识符 b 的类似声明并调用函数 c，它们实际上访问的是这个源文件所定义的实体。f 的链接属性之所以是 external 是因为它是个函数名。在这个源文件中调用函数 f，它实际上将链接到其他源文件所定义的函数，甚至这个函数的定义可能出现在某个函数库。

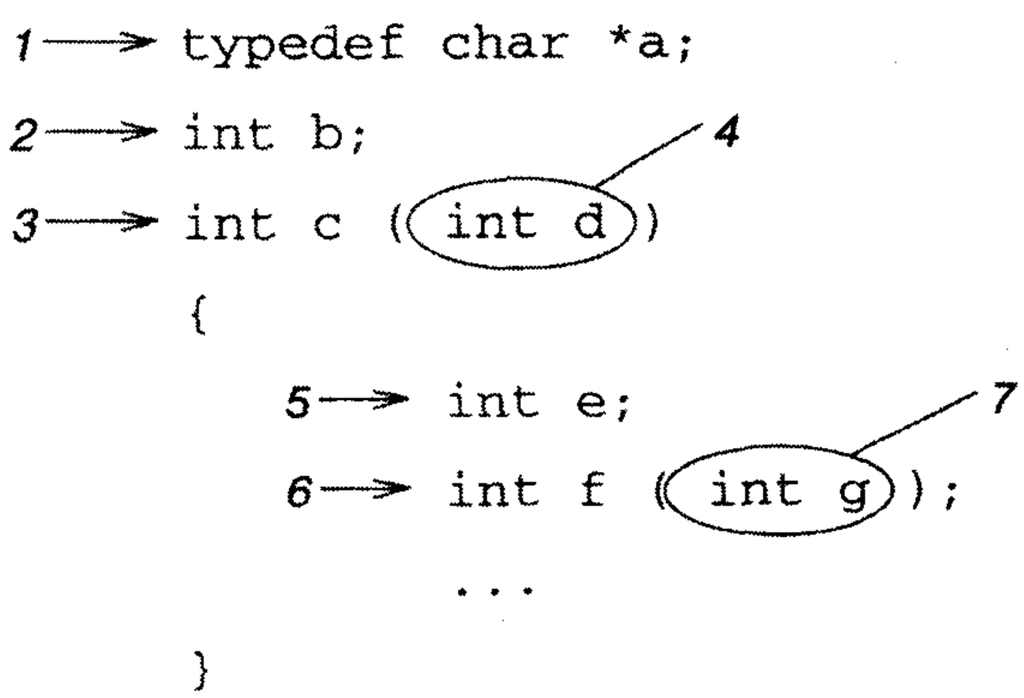

图 3.2　链接属性示例

关键字 extern 和 static 用于在声明中修改标识符的链接属性。如果某个声明在正常情况下具有 external 链接属性，在它前面加上 static 关键字可以使它的链接属性变为 internal。例如，如果第 2 个声明像下面这样书写：

```
static    int       b;
```

那么变量 b 就将为这个源文件所私有。在其他源文件中，如果也链接到一个叫做 b 的变量，那么它所引用的是另一个不同的变量。类似，你也可以把函数声明为 static，如下：

```
static   int  c( int d )
```

这可以防止它被其他源文件调用。

static 只对缺省链接属性为 external 的声明才有改变链接属性的效果。例如，尽管你可以在声明 5 前面加上 static 关键字，但它的效果完全不一样，因为 e 的缺省链接属性并不是 external。

extern 关键字的规则更为复杂。一般而言，它为一个标识符指定 external 链接属性，这样就可以访问在其他任何位置定义的这个实体。请考虑图 3.3 的例子。声明 3 为 k 指定 external 链接属性。这样一来，函数就可以访问在其他源文件声明的外部变量了。

提示：

从技术上说，这两个关键字只有在声明中才是必需的，如图 3.3 中的声明 3（它的缺省链接属性并不是 external）。当用于具有文件作用域的声明时，这个关键字是可选的。然而，如果你在一个地方定义变量，并在使用这个变量的其他源文件的声明中添加 external 关键字，可以使读者更容易理解你的意图。

当 extern 关键字用于源文件中一个标识符的第 1 次声明时，它指定该标识符具有 external 链接属性。但是，如果它用于该标识符的第 2 次或以后的声明时，它并不会更改由第 1 次声明所指定的

bbs.theithome.com

链接属性。例如，图 3.3 中的声明 4 并不修改由声明 1 所指定的变量 i 的链接属性。

```
1 ──→ static int i;
      int func()
      {
         2 ──→ int j;
         3 ──→ extern int k;
         4 ──→ extern int i;
                  ...
      }
```

图 3.3　使用 extern

3.7　存储类型

变量的存储类型(storage class)是指存储变量值的内存类型。变量的存储类型决定变量何时创建、何时销毁以及它的值将保持多久。有三个地方可以用于存储变量：普通内存、运行时堆栈、硬件寄存器。在这三个地方存储的变量具有不同的特性。

变量的缺省存储类型取决于它的声明位置。凡是在任何代码块之外声明的变量总是存储于静态内存中，也就是不属于堆栈的内存，这类变量称为静态(static)变量。对于这类变量，你无法为它们指定其他存储类型。静态变量在程序运行之前创建，在程序的整个执行期间始终存在。它始终保持原先的值，除非给它赋一个不同的值或者程序结束。

在代码块内部声明的变量的缺省存储类型是自动的(automatic)，也就是说它存储于堆栈中，称为自动(auto)变量。有一个关键字 auto 就是用于修饰这种存储类型的，但它极少使用，因为代码块中的变量在缺省情况下就是自动变量。在程序执行到声明自动变量的代码块时，自动变量才被创建，当程序的执行流离开该代码块时，这些自动变量便自行销毁。如果该代码块被数次执行，例如一个函数被反复调用，这些自动变量每次都将重新创建。在代码块再次执行时，这些自动变量在堆栈中所占据的内存位置有可能和原先的位置相同，也可能不同。即使它们所占据的位置相同，你也不能保证这块内存同时不会有其他的用途。因此，我们可以说自动变量在代码块执行完毕后就消失。当代码块再次执行时，它们的值一般并不是上次执行时的值。

对于在代码块内部声明的变量，如果给它加上关键字 static，可以使它的存储类型从自动变为静态。具有静态存储类型的变量在整个程序执行过程中一直存在，而不仅仅在声明它的代码块的执行时存在。注意，修改变量的存储类型并不表示修改该变量的作用域，它仍然只能在该代码块内部按名字访问。函数的形式参数不能声明为静态，因为实参总是在堆栈中传递给函数，用于支持递归。

最后，关键字 register 可以用于自动变量的声明，提示它们应该存储于机器的硬件寄存器而不是内存中，这类变量称为寄存器变量。通常，寄存器变量比存储于内存的变量访问起来效率更高。但是，编译器并不一定要理睬 register 关键字，如果有太多的变量被声明为 register，它只选取前几个实际存储于寄存器中，其余的就按普通自动变量处理。如果一个编译器自己具有一套寄存器优化方

bbs.theithome.com

法，它也可能忽略 register 关键字，其依据是由编译器决定哪些变量存储于寄存器中比人脑的决定更为合理一些。

在典型情况下，你希望把使用频率最高的那些变量声明为寄存器变量。在有些计算机中，如果把指针声明为寄存器变量，程序的效率将能得到提高，尤其是那些频繁执行间接访问操作的指针。你可以把函数的形式参数声明为寄存器变量，编译器会在函数的起始位置生成指令，把这些值从堆栈复制到寄存器中。但是，完全有可能，这个优化措施所节省的时间和空间的开销还抵不上复制这几个值所用的开销。

寄存器变量的创建和销毁时间和自动变量相同，但它需要一些额外的工作。在一个使用寄存器变量的函数返回之前，这些寄存器先前存储的值必须恢复，确保调用者的寄存器变量未被破坏。许多机器使用运行时堆栈来完成这个任务。当函数开始执行时，它把需要使用的所有寄存器的内容都保存到堆栈中，当函数返回时，这些值再复制回寄存器中。

在许多机器的硬件实现中，并不为寄存器指定地址。同样，由于寄存器值的保存和恢复，某个特定的寄存器在不同的时刻所保存的值不一定相同。基于这些理由，机器并不向你提供寄存器变量的地址。

初始化

现在我们把话题返回到变量声明中变量的初始化问题。自动变量和静态变量的初始化存在一个重要的差别。在静态变量的初始化中，我们可以把可执行程序文件想要初始化的值放在当程序执行时变量将会使用的位置。当可执行文件载入到内存时，这个已经保存了正确初始值的位置将赋值给那个变量。完成这个任务并不需要额外的时间，也不需要额外的指令，变量将会得到正确的值。如果不显式地指定其初始值，静态变量将初始化为 0。

自动变量的初始化需要更多的开销，因为当程序链接时还无法判断自动变量的存储位置。事实上，函数的局部变量在函数的每次调用中可能占据不同的位置。基于这个理由，自动变量没有缺省的初始值，而显式的初始化将在代码块的起始处插入一条隐式的赋值语句。

这个技巧造成 4 种后果。首先，自动变量的初始化较之赋值语句效率并无提高。除了声明为 const 的变量之外，在声明变量的同时进行初始化和先声明后赋值只有风格之差，并无效率之别。其次，这条隐式的赋值语句使自动变量在程序执行到它们所声明的函数（或代码块）时，每次都将重新初始化。这个行为与静态变量大不相同，后者只是在程序开始执行前初始化一次。第 3 个后果则是个优点，由于初始化在运行时执行，你可以用任何表达式作为初始化值，例如：

```
int
func( int a )
{
        int     b = a + 3;
```

最后一个后果是，除非你对自动变量进行显式的初始化，否则当自动变量创建时，它们的值总是垃圾。

3.8 static 关键字

当用于不同的上下文环境时，static 关键字具有不同的意思。确实很不幸，因为这总是给 C 程序员新手带来混淆。本节对 static 关键字作了总结，再加上后续的例子程序，应该能够帮助你搞清

bbs.theithome.com

这个问题。

当它用于函数定义时，或用于代码块之外的变量声明时，static 关键字用于修改标识符的链接属性，从 external 改为 internal，但标识符的存储类型和作用域不受影响。用这种方式声明的函数或变量只能在声明它们的源文件中访问。

当它用于代码块内部的变量声明时，static 关键字用于修改变量的存储类型，从自动变量修改为静态变量，但变量的链接属性和作用域不受影响。用这种方式声明的变量在程序执行之前创建，并在程序的整个执行期间一直存在，而不是每次在代码块开始执行时创建，在代码块执行完毕后销毁。

3.9　作用域、存储类型示例

图 3.4 包含了一个例子程序，阐明了作用域和存储类型。属于文件作用域的声明在缺省情况下为 external 链接属性，所以第 1 行的 a 的链接属性为 external。如果 b 的定义在其他地方，第 2 行的 extern 关键字在技术上并非必需，但在风格上却是加上这个关键字为好。第 3 行的 static 关键字修改了 c 的缺省链接属性，把它改为 internal。声明了变量 a 和 b（具有 external 链接属性）的其他源文件在使用这两个变量时实际所访问的是声明于此处的这两个变量。但是，变量 c 只能由这个源文件访问，因为它具有 internal 链接属性。

```
1       int                 a = 5;
2       extern    int       b;
3       static    int       c;

4       int d( int e )
5       {
6                 int                 f = 15;
7                 register int        b;
8                 static    int       g = 20;
9                 extern    int       a;
10                ...
11                {
12                          int                 e;
13                          int                 a;
14                          extern    int       h;
15                          ...
16                }
17                ...
18                {
19                          int       x;
20                          int       e;
21                          ...
22                }
23      ...
24      }

25      static    int  i()
26        {
27                  ...
28        }

29      ...
```

图 3.4　作用域、链接属性和存储类型示例

变量 a、b、c 的存储类型为静态，表示它们并不是存储于堆栈中。因此，这些变量在程序执行

bbs.theithome.com

之前创建，并一直保持它们的值，直到程序结束。当程序开始执行时，变量 a 将初始化为 5。

这些变量的作用域一直延伸到这个源文件结束为止，但第 7 行和第 13 行声明的局部变量 a 和 b 在那部分程序中将隐藏同名的静态变量。因此，这 3 个变量的作用域为：

a 第 1 至 12 行，第 17 至 29 行
b 第 2 至 6 行， 第 25 至 29 行
c 第 3 至 29 行

第 4 行声明了 2 个标识符。d 的作用域从第 4 行直到文件结束。函数 d 的定义对于这个源文件中任何以后想要调用它的函数而言起到了函数原型的作用。作为函数名，d 在缺省情况下具有 external 链接属性，所以其他源文件只要在文件上存在 d 的原型[1]，就可以调用 d。如果我们将函数声明为 static，就可以把它的链接属性从 external 改为 internal，但这样做将使其他源文件不能访问这个函数。对于函数而言，存储类型并不是问题，因为代码总是存储于静态内存中。

参数 e 不具有链接属性，所以我们只能从函数内部通过名字访问它。它具有自动存储类型，所以它在函数被调用时被创建，当函数返回时消失。由于与局部变量冲突，它的作用域限于第 6 至 11 行，第 17 至 19 行以及第 23 至 24 行。

第 6 至 8 行声明局部变量，所以它们的作用域到函数结束为止。它们不具有链接属性，所以它们不能在函数的外部通过名字访问（这是它们称为局部变量的原因）。f 的存储类型是自动，当函数每次被调用时，它通过隐式赋值被初始化为 15。b 的存储类型是寄存器类型，所以它的初始值是垃圾。g 的存储类型是静态，所以它在程序的整个执行过程中一直存在。当程序开始执行时，它被初始化为 20。当函数每次被调用时，它并不会被重新初始化。

第 9 行的声明并不需要。这个代码块位于第 1 行声明的作用域之内。

第 12 和 13 行为代码块声明局部变量。它们都具有自动存储类型，不具有链接属性，它们的作用域延伸至第 16 行。这些变量和先前声明的 a 和 e 不同，而且由于名字冲突，在这个代码块中，以前声明的同名变量是不能被访问的。

第 14 行使全局变量 h 在这个代码块内可以被访问。它具有 external 链接属性，存储于静态内存中。这是唯一一个必须使用 extern 关键字的声明，如果没有它，h 将变成另一个局部变量。

第 19 和 20 行用于创建局部变量（自动、无链接属性、作用域限于本代码块）。这个 e 和参数 e 是不同的变量，它和第 12 行声明的 e 也不相同。在这个代码块中，从第 11 行到第 18 行并无嵌套，所以编译器可以使用相同的内存来存储两个代码块中不同的变量 e。如果你想让这两个代码块中的 e 表示同一个变量，那么你就不应该把它声明为局部变量。

最后，第 25 行声明了函数 i，它具有静态链接属性。静态链接属性可以防止它被这个源文件之外的任何函数调用。事实上，其他的源文件也可能声明它自己的函数 i，它与这个源文件的 i 是不同的函数。i 的作用域从它声明的位置直到这个源文件结束。函数 d 不可以调用函数 i，因为在 d 之前不存在 i 的原型。

3.10 总结

具有 external 链接属性的实体在其他语言的术语里称为全局(global)实体，所有源文件中的所有

[1] 实际上，只有当 d 的返回值不是整型时才需要原型。推荐为你调用的所有函数添加原型，因为它减少了发生难以检测的错误的机会。

bbs.theithome.com

函数均可以访问它。只要变量并非声明于代码块或函数定义内部，它在缺省情况下的链接属性即为 external。如果一个变量声明于代码块内部，在它前面添加 extern 关键字将使它所引用的是全局变量而非局部变量。

具有 external 链接属性的实体总是具有静态存储类型。全局变量在程序开始执行前创建，并在程序整个执行过程中始终存在。从属于函数的局部变量在函数开始执行时创建，在函数执行完毕后销毁，但用于执行函数的机器指令在程序的生命期内一直存在。

局部变量由函数内部使用，不能被其他函数通过名字引用。它在缺省情况下的存储类型为自动，这是基于两个原因：其一，当这些变量需要时才为它们分配存储，这样可以减少内存的总需求量。其二，在堆栈上为它们分配存储可以有效地实现递归。如果你觉得让变量的值在函数的多次调用中始终保持原先的值非常重要的话，你可以修改它的存储类型，把它从自动变量改为静态变量。

这些信息在表 3.4 中进行总结。

表 3.4　作用域、链接属性和存储类型总结

变量类型	声明的位置	是否存于堆栈	作用域	如果声明为 static
全局	所有代码块之外	否[1]	从声明处到文件尾	不允许从其他源文件访问
局部	代码块起始处	是[2]	整个代码块[3]	变量不存储于堆栈中，它的值在程序整个执行期一直保持
形式参数	函数头部	是[2]	整个函数[3]	不允许

3.11　警告的总结

1. 在声明指针变量时采用容易误导的写法。
2. 误解指针声明中初始化的含义。

3.12　编程提示的总结

1. 为了保持最佳的可移植性，把字符的值限制在有符号和无符号字符范围的交集之内，或者不要在字符上执行算术运算。
2. 用它们在使用时最自然的形式来表示字面值。
3. 不要把整型值和枚举值混在一起使用。
4. 不要依赖隐式声明。
5. 在定义类型的新名字时，使用 typedef 而不是#define。
6. 用 const 声明其值不会修改的变量。
7. 使用名字常量而不是字面值常量。
8. 不要在嵌套的代码块之间使用相同的变量名。

---

[1] 存储于堆栈的变量只有当该代码块处于活动期间，它们才能保持自己的值。当程序的执行流离开该代码块时，这些变量的值将丢失。

[2] 并非存储于堆栈的变量在程序开始执行时创建，并在整个程序执行期间一直保持它们的值，不管它们是全局变量还是局部变量。

[3] 有一个例外，就是在嵌套的代码块中分别声明了相同名字的变量。

bbs.theithome.com

9. 除了实体的具体定义位置之外，在它的其他声明位置都使用 extern 关键字。

3.13 问题

1. 在你的机器上，字符的范围有多大？有哪些不同的整数类型？它们的范围又是如何？
2. 在你的机器上，各种不同类型的浮点数的范围是怎样的？
3. 假定你正编写一个程序，它必须运行于两台机器之上。这两台机器的缺省整型长度并不相同，一个是 16 位，另一个是 32 位。而这两台机器的长整型长度分别是 32 位和 64 位。程序所使用的有些变量的值并不太大，足以保存于任何一台机器的缺省整型变量中，但有些变量的值却较大，必须是 32 位的整型变量才能容纳它。一种可行的解决方案是用长整型表示所有的值，但在 16 位机器上，对于那些用 16 位足以容纳的值而言，时间和空间的浪费不可小视。在 32 位机器上，也存在时间和空间的浪费问题。

   如果想让这些变量在任何一台机器上的长度都合适的话，你该如何声明它们呢？正确的方法是不应该在任何一台机器中编译程序前对程序进行修改。**提示**：试试包含一个头文件，里面包含每台机器特定的声明。
4. 假定你有一个程序，它把一个 long 整型变量赋值给一个 short 整型变量。当你编译程序时会发生什么情况？当你运行程序时会发生什么情况？你认为其他编译器的结果是否也是如此？
5. 假定你有一个程序，它把一个 double 变量赋值给一个 float 变量。当你编译程序时会发生什么情况？当你运行程序时会发生什么情况？
6. 编写一个枚举声明，用于定义硬币的值。请使用符号 PENNY、NICKEL 等。
7. 下列代码段会打印出什么东西？

```
enum Liquid { OUNCE = 1, CUP = 8, PINT = 16,
     QUART = 32, GALLON = 128 };

enum    Liquid  jar;
...
jar = QUART;
printf( "%s\n", jar );
jar = jar + PINT;
printf( "%s\n", jar );
```

8. 你所使用的 C 编译器是否允许程序修改字符串常量？是否存在编译器选项，允许或禁止你修改字符串常量？
9. 如果整数类型在正常情况下是有符号类型，那么 signed 关键字的目的何在呢？
10. 一个无符号变量可不可以比相同长度的有符号变量容纳更大的值？
11. 假如 int 和 float 类型都是 32 位长，你觉得哪种类型所能容纳的值精度更大一些？
12. 下面是两个代码片段，取自一个函数的起始部分。

```
int    a  =  25;
```

```
int       a;
a = 25;
```

它们完成任务的方式有何不同？

13. 如果问题 12 中代码片段的声明中包含有 const 关键字，它们完成任务的方式又有

bbs.theithome.com

何不同？

14. 在一个代码块内部声明的变量可以从该代码块的任何位置根据名字来访问，对还是错？

15. 假定函数 a 声明了一个自动整型变量 x，你可以在其他函数内访问变量 x，只要你使用了下面这样的声明：

```
extern  int  x;
```

对还是错？

16. 假定问题 15 中的变量 x 被声明为 static。你的答案会不会有所变化？

17. 假定文件 a.c 的开始部分有下面这样的声明：

```
int          x;
```

如果你希望从同一个源文件后面出现的函数中访问这个变量，需不需要添加额外的声明，如果需要的话，应该添加什么样的声明？

18. 假定问题 17 中的声明包含了关键字 static。你的答案会不会有所变化？

19. 假定文件 a.c 的开始部分有下面这样的声明：

```
int          x;
```

如果你希望从不同的源文件的函数中访问这个变量，需不需要添加额外的声明，如果需要的话，应该添加什么样的声明？

20. 假定问题 19 中的声明包含了关键字 static。你的答案会不会有所变化？

21. 假定一个函数包含了一个自动变量，这个函数在同一行中被调用了两次。试问，在函数第 2 次调用开始时该变量的值和函数第 1 次调用即将结束时的值有无可能相同？

22. 当下面的声明出现于某个代码块内部和出现于任何代码块外部时，它们在行为上有何不同？

```
int          a = 5;
```

23. 假定你想在同一个源文件中编写两个函数 x 和 y，需要使用下面的变量：

名字	类型	存储类型	链接属性	作 用 域	初 始 化 为
a	int	static	external	x 可以访问，y 不能访问	1
b	char	static	none	x 和 y 都可以访问	2
c	int	automatic	none	x 的局部变量	3
d	float	static	none	x 的局部变量	4

你应该怎样编写这些变量？应该在什么地方编写？注意：所有初始化必须在声明中完成，而不是通过函数中的任何可执行语句来完成。

24. 确认下面程序中存在的任何错误（你可能想动手编译一下，这样能够踏实一些）。在去除所有错误之后，确定所有标识符的存储类型、作用域和链接属性。每个变量的初始值会是什么？程序中存在许多同名的标识符，它们所代表的是相同的变量还是不同的变量？程序中的每个函数从哪个位置起可以被调用？

```
1    static    int       w = 5;
2    extern    int       x;

3    static    float
4    func1( int a, int b, int c )
5    {
```


bbs.theithome.com

```
6               int       c, d, e = 1;
7               ...
8               {
9                         int       d, e, w;
10                        ...
11                        {
12                                  int       b, c, d;
13                                  static  int  y = 2;
14                                  ...
15                        }
16              }
17              ...
18              {
19                        register  int  a, d, x;
20                        extern    int  y;
21                        ...
22              }
23  }

24  static    int  y;

25  float
26  func2( int a )
27  {
28            extern    int  y;
29            static    int  z;
30            ...
31  }
```


bbs.theithome.com