第10章

结构和联合

数据经常以成组的形式存在。例如，雇主必须明了每位雇员的姓名、年龄和工资。如果这些值能够存储在一起，访问起来会简单一些。但是，如果这些值的类型不同（就像现在这种情况），它们无法存储于同一个数组中。在 C 中，使用结构可以把不同类型的值存储在一起。

10.1 结构基础知识

聚合数据类型（aggregate data type）能够同时存储超过一个的单独数据。C 提供了两种类型的聚合数据类型，数组和结构。数组是相同类型的元素的集合，它的每个元素是通过下标引用或指针间接访问来选择的。

结构也是一些值的集合，这些值称为它的**成员**(member)，但一个结构的各个成员可能具有不同的类型。结构和 Pascal 或 Modula 中的记录(record)非常相似。

数组元素可以通过下标访问，这只是因为数组的元素长度相同。但是，在结构中情况并非如此。由于一个结构的成员可能长度不同，所以不能使用下标来访问它们。相反，每个结构成员都有自己的名字，它们是通过名字访问的。

这个区别非常重要。结构并不是一个它自身成员的数组。和数组名不同，当一个结构变量在表达式中使用时，它并不被替换成一个指针。结构变量也无法使用下标来选择特定的成员。

结构变量属于标量类型，所以你可以像对待其他标量类型那样执行相同类型的操作。结构也可以作为传递给函数的参数，它们也可以作为返回值从函数返回，相同类型的结构变量相互之间可以赋值。你可以声明指向结构的指针，取一个结构变量的地址，也可以声明结构数组。但是，在讨论这些话题之前，我们必须知道一些更为基础的东西。

10.1.1 结构声明

在声明结构时，必须列出它包含的所有成员。这个列表包括每个成员的类型和名字。

```
struct tag { member-list } variable-list ;
```

结构声明的语法需要作一些解释。所有可选部分不能全部省略——它们至少要出现两个[1]。

这里有几个例子。

```
struct  {
        int     a;
        char    b;
        float   c;
} x;
```

这个声明创建了一个名叫 x 的变量，它包含三个成员：一个整数、一个字符和一个浮点数。

```
struct  {
        int     a;
        char    b;
        float   c;
} y[20], *z;
```

这个声明创建了 y 和 z。y 是一个数组，它包含了 20 个结构。z 是一个指针，它指向这个类型的结构。

警告：

这两个声明被编译器当作两种截然不同的类型，即使它们的成员列表完全相同。因此，变量 y 和 z 的类型和 x 的类型不同，所以下面这条语句

```
z = &x;
```

是非法的。

但是，这是不是意味着某种特定类型的所有结构都必须使用一个单独的声明来创建呢？

幸运的是，事实并非如此。标签(tag)字段允许为成员列表提供一个名字，这样它就可以在后续的声明中使用。标签允许多个声明使用同一个成员列表，并且创建同一种类型的结构。这里有个例子。

```
struct  SIMPLE  {
        int     a;
        char    b;
        float   c;
};
```

这个声明把标签 SIMPLE 和这个成员列表联系在一起。该声明并没有提供变量列表，所以它并未创建任何变量。

这个声明类似于制造一个甜饼切割器。甜饼切割器决定制造出来的甜饼的形状，但甜饼切割器本身却不是甜饼。标签标识了一种模式，用于声明未来的变量，但无论是标签还是模式本身都不是变量。

```
struct  SIMPLE  x;
struct  SIMPLE  y[20], *z;
```

这些声明使用标签来创建变量。它们创建和最初两个例子一样的变量，但存在一个重要的区别——现在 x、y 和 z 都是同一种类型的结构变量。

声明结构时可以使用的另一种良好技巧是用 typedef 创建一种新的类型，如下面的例子所示。

```
typedef struct  {
        int     a;
        char    b;
        float   c;
} Simple;
```

这个技巧和声明一个结构标签的效果几乎相同。区别在于 Simple 现在是个类型名而不是个结构

[1] 这个规则的一个例外是结构标签的不完整声明，在本章后面部分描述。

标签，所以后续的声明可能像下面这个样子：

```
Simple  x;
Simple  y[20], *z;
```

提示：

如果你想在多个源文件中使用同一种类型的结构，你应该把标签声明或 typedef 形式的声明放在一个头文件中。当源文件需要这个声明时可以使用#include 指令把那个头文件包含进来。

10.1.2　结构成员

到目前为止的例子里，我只使用了简单类型的结构成员。但可以在一个结构外部声明的任何变量都可以作为结构的成员。尤其是，结构成员可以是标量、数组、指针甚至是其他结构。

这里有一个更为复杂的例子：

```
struct  COMPLEX {
        float   f;
        int     a[20];
        long    *lp;
        struct  SIMPLE  s;
        struct  SIMPLE  sa[10];
        struct  SIMPLE  *sp;
};
```

一个结构的成员的名字可以和其他结构的成员的名字相同，所以这个结构的成员 a 并不会与 struct SIMPLE s 的成员 a 冲突。正如你接下去看到的那样，成员的访问方式允许你指定任何一个成员而不至于产生歧义。

10.1.3　结构成员的直接访问

结构变量的成员是通过点操作符(.)访问的。点操作符接受两个操作数，左操作数就是结构变量的名字，右操作数就是需要访问的成员的名字。这个表达式的结果就是指定的成员。例如，考虑下面这个声明

```
struct COMPLEX comp;
```

名字为 a 的成员是一个数组，所以表达式 comp.a 就选择了这个成员。这个表达式的结果是个数组名，所以你可以把它用在任何可以使用数组名的地方。类似地，成员 s 是个结构，所以表达式 comp.s 的结果是个结构名，它可以用于任何可以使用普通结构变量的地方。尤其是，我们可以把这个表达式用作另一个点操作符的左操作符，如(comp.s).a，选择结构 comp 的成员 s（也是一个结构）的成员 a。点操作符的结合性是从左向右，所以我们可以省略括号，表达式 comp.s.a 表示同样的意思。

这里有一个更为复杂的例子。成员 sa 是一个结构数组，所以 comp.sa 是一个数组名，它的值是一个指针常量。对这个表达式使用下标引用操作，如(comp.sa)[4]将选择一个数组元素。但这个元素本身是一个结构，所以我们可以使用另一个点操作符取得它的成员之一。下面就是一个这样的表达式：

```
( (comp.sa)[4] ).c
```

下标引用和点操作符具有相同的优先级，它们的结合性都是从左向右，所以我们可以省略所有的括号。下面的表达式

```
comp.sa[4].c
```

和前面那个表达式是等效的。

10.1.4 结构成员的间接访问

如果你拥有一个指向结构的指针，你该如何访问这个结构的成员呢？首先就是对指针执行间接访问操作，这使你获得这个结构。然后你使用点操作符来访问它的成员。但是，点操作符的优先级高于间接访问操作符，所以你必须在表达式中使用括号，确保间接访问首先执行。举个例子，假定一个函数的参数是个指向结构的指针，如下面的原型所示：

```
void    func( struct COMPLEX *cp );
```

函数可以使用下面这个表达式来访问这个变量所指向的结构的成员 f：

```
(*cp).f
```

对指针执行间接访问将访问结构，然后点操作符访问一个成员。

由于这个概念有点惹人厌，所以 C 语言提供了一个更为方便的操作符来完成这项工作——->操作符（也称箭头操作符）。和点操作符一样，箭头操作符接受两个操作数，但左操作数必须是一个指向结构的**指针**。箭头操作符对左操作数执行间接访问取得指针所指向的结构，然后和点操作符一样，根据右操作数选择一个指定的结构成员。但是，间接访问操作内建于箭头操作符中，所以我们不需要显式地执行间接访问或使用括号。这里有一些例子，像前面一样使用同一个指针。

```
cp->f
cp->a
cp->s
```

第 1 个表达式访问结构的浮点数成员，第 2 个表达式访问一个数组名，第 3 个表达式则访问一个结构。你很快还将看到为数众多的例子，可以帮助你弄清如何访问结构成员。

10.1.5 结构的自引用

在一个结构内部包含一个类型为该结构本身的成员是否合法呢？这里有一个例子，可以说明这个想法。

```
struct  SELF_REF1 {
        int     a;
        struct  SELF_REF1 b;
        int     c;
};
```

这种类型的自引用是非法的，因为成员 b 是另一个完整的结构，其内部还将包含它自己的成员 b。这第 2 个成员又是另一个完整的结构，它还将包括它自己的成员 b。这样重复下去永无止境。这有点像永远不会终止的递归程序。但下面这个声明却是合法的，你能看出其中的区别吗？

```
struct  SELF_REF2 {
        int     a;
        struct  SELF_REF2 *b;
        int     c;
};
```

这个声明和前面那个声明的区别在于 b 现在是一个指针而不是结构。编译器在结构的长度确定之前就已经知道指针的长度，所以这种类型的自引用是合法的。

如果你觉得一个结构内部包含一个指向该结构本身的指针有些奇怪，请记住它事实上所指向的是同一种类型的**不同**结构。更加高级的数据结构，如链表和树，都是用这种技巧实现的。每个结构指向链表的下一个元素或树的下一个分枝。

警告：

警惕下面这个陷阱：

```
typedef struct  {
        int     a;
        SELF_REF3 *b;
        int     c;
} SELF_REF3;
```

这个声明的目的是为这个结构创建类型名 SELF_REF3。但是，它失败了。类型名直到声明的末尾才定义，所以在结构声明的内部它尚未定义。

解决方案是定义一个结构标签来声明 b，如下所示：

```
typedef struct  SELF_REF3_TAG {
        int     a;
        struct  SELF_REF3_TAG *b;
        int     c;
} SELF_REF3;
```

10.1.6　不完整的声明

偶尔，你必须声明一些相互之间存在依赖的结构。也就是说，其中一个结构包含了另一个结构的一个或多个成员。和自引用结构一样，至少有一个结构必须在另一个结构内部以指针的形式存在。问题在于声明部分：如果每个结构都引用了其他结构的标签，哪个结构应该首先声明呢？

这个问题的解决方案是使用不完整声明(incomplete declaration)，它声明一个作为结构标签的标识符。然后，我们可以把这个标签用在不需要知道这个结构的长度的声明中，如声明指向这个结构的指针。接下来的声明把这个标签与成员列表联系在一起。

考虑下面这个例子，两个不同类型的结构内部都有一个指向另一个结构的指针。

```
struct  B;

struct  A       {
        struct  B       *partner;
        /* other declarations */
};

struct  B       {
        struct  A       *partner;
        /* other declarations */
};
```

在 A 的成员列表中需要标签 B 的不完整的声明。一旦 A 被声明之后，B 的成员列表也可以被声明。

10.1.7　结构的初始化

结构的初始化方式和数组的初始化很相似。一个位于一对花括号内部、由逗号分隔的初始值列表可用于结构各个成员的初始化。这些值根据结构成员列表的顺序写出。如果初始列表的值不够，剩余的结构成员将使用缺省值进行初始化。

结构中如果包含数组或结构成员，其初始化方式类似于多维数组的初始化。一个完整的聚合类型成员的初始值列表可以嵌套于结构的初始值列表内部。这里有一个例子：

```
struct  INIT_EX {
        int     a;
        short   b[10];
        Simple  c;
```

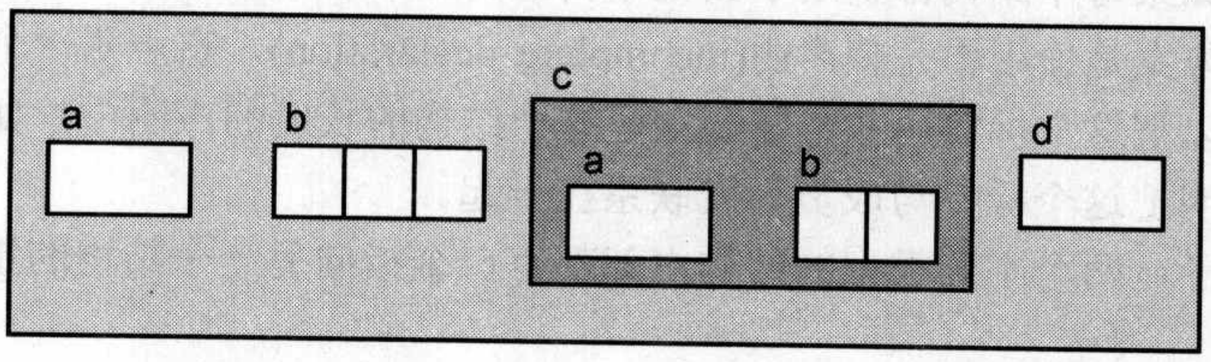
```
} x = {
        10,
        { 1, 2, 3, 4, 5 },
        { 25, 'x', 1.9 }
};
```

10.2 结构、指针和成员

直接或通过指针访问结构和它们的成员的操作符是相当简单的，但是当它们应用于复杂的情形时就有可能引起混淆。这里有几个例子，能帮助你更好地理解这两个操作符的工作过程。这些例子使用了下面的声明。

```
typedef struct  {
        int     a;
        short   b[2];
} Ex2;
typedef struct  EX {
        int     a;
        char    b[3];
        Ex2     c;
        struct  EX      *d;
} Ex;
```

类型为 EX 的结构可以用下面的图表示：

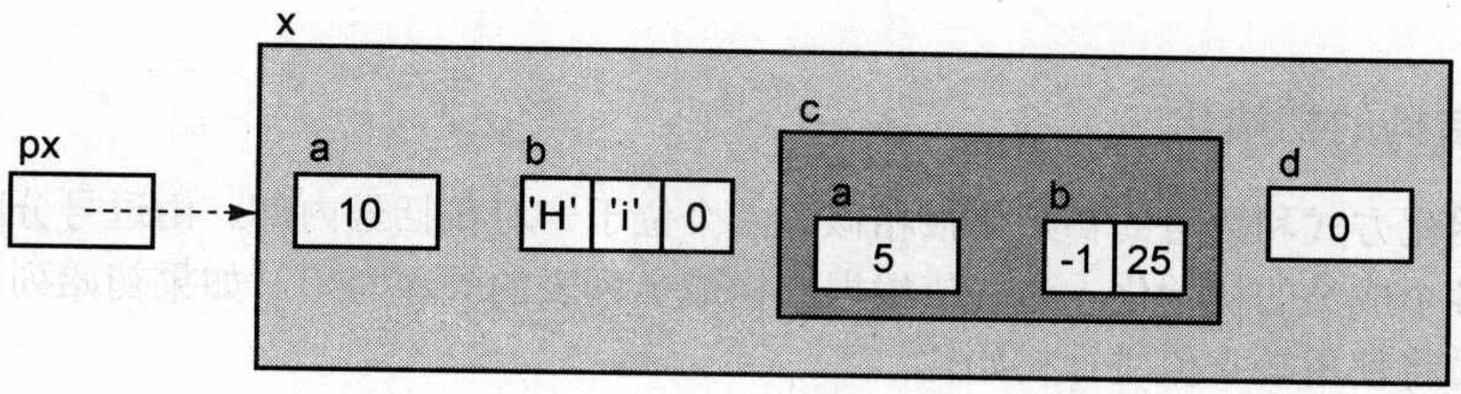


我用图的形式来表示结构，使这些例子看上去更清楚一些。事实上，这张图并不完全准确，因为编译器只要有可能就会设法避免成员之间的浪费空间。

第 1 个例子将使用这些声明：

```
Ex          x = { 10, "Hi", { 5, { -1, 25 } }, 0 };
Ex          *px = &x;
```

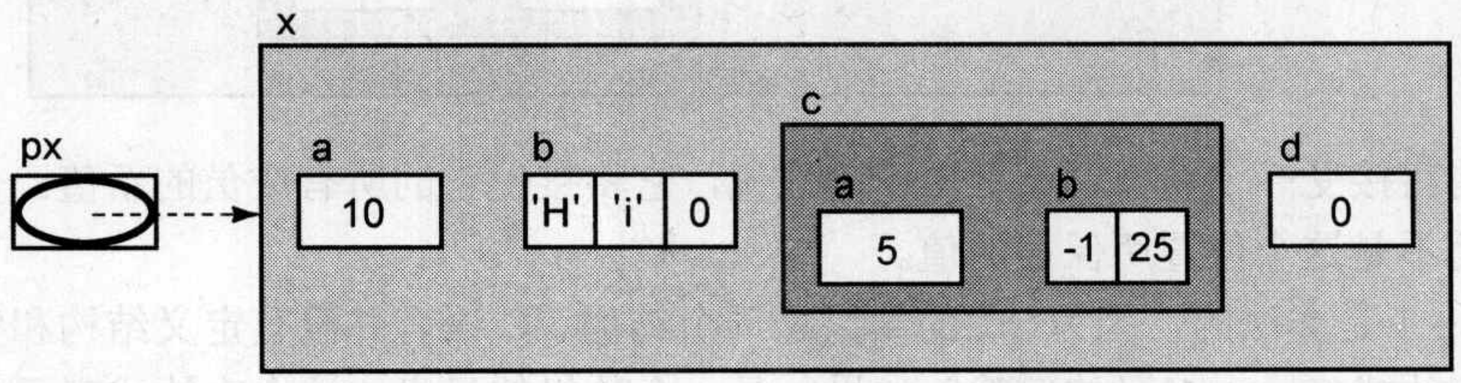
它将产生下面这些变量：



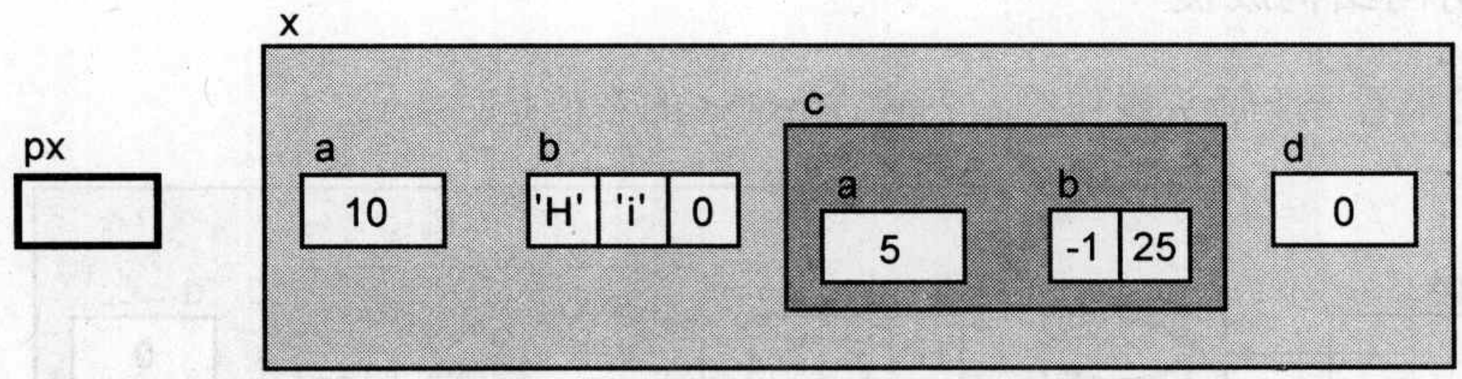
我们现在将使用第 6 章的记法研究和图解各个不同的表达式。

10.2.1　访问指针

让我们从指针变量开始。表达式 px 的右值是：



px 是一个指针变量，但此处并不存在任何间接访问操作符，所以这个表达式的值就是 px 的内容。这个表达式的左值是：

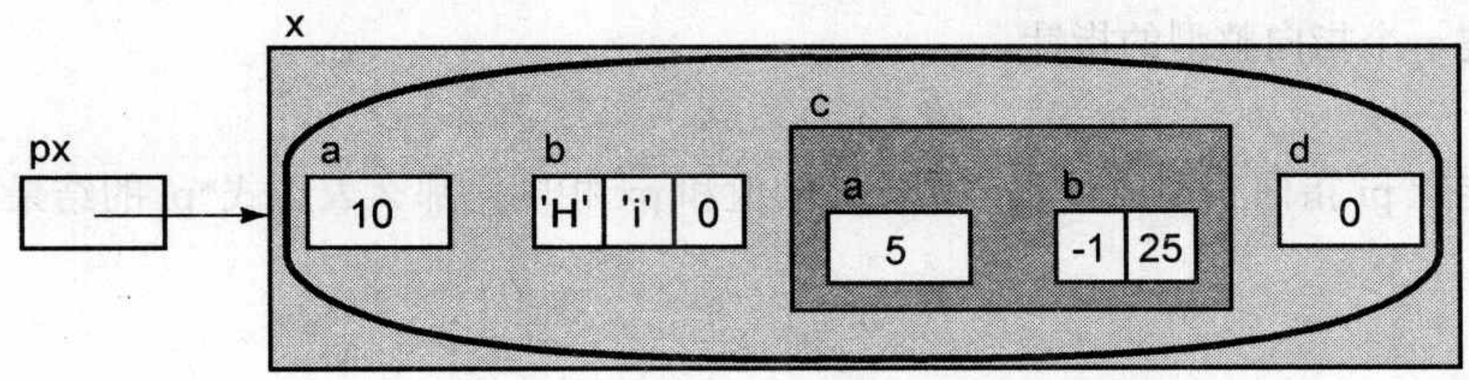


它显示了 px 的旧值将被一个新值所取代。

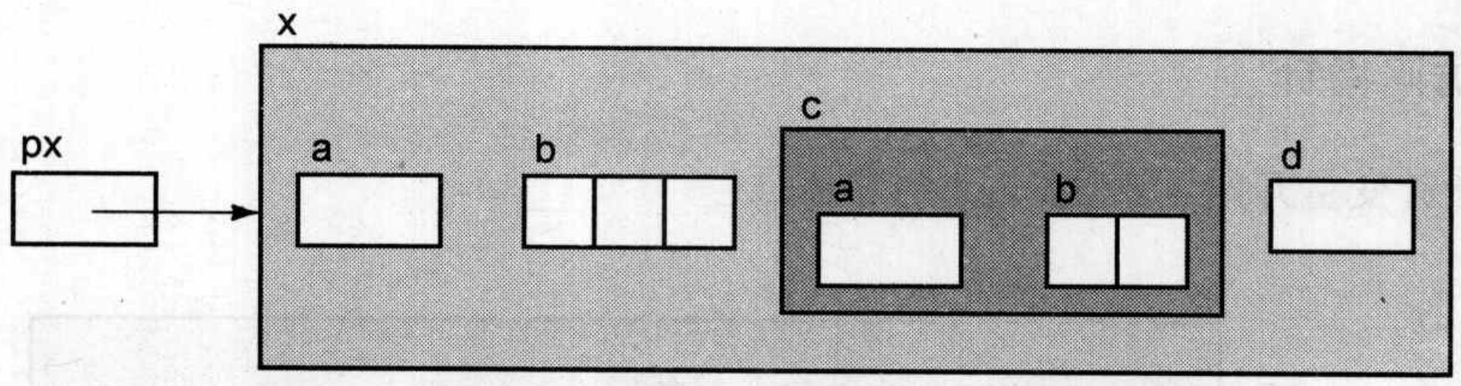
现在考虑表达式 px + 1。这个表达式并不是一个合法的左值，因为它的值并不存储于任何可标识的内存位置。这个表达式的右值更为有趣。如果 px 指向一个结构数组的元素，这个表达式将指向该数组的下一个结构。但就算如此，这个表达式仍然是非法的，因为我们没办法分辨内存下一个位置所存储的是这些结构元素之一还是其他东西。编译器无法检测到这类错误，所以你必须自己判断指针运算是否有意义。

10.2.2　访问结构

我们可以使用*操作符对指针执行间接访问。表达式*px 的右值是 px 所指向的整个结构。



间接访问操作随箭头访问结构，所以使用实线显示，其结果就是整个结构。你可以把这个表达式赋值给另一个类型相同的结构，你也可以把它作为点操作符的左操作数，访问一个指定的成员。你也可以把它作为参数传递给函数，也可以把它作为函数的返回值返回（不过，关于最后两个操作，需要考虑效率问题，对此以后将会详述）。表达式*px 的左值是：

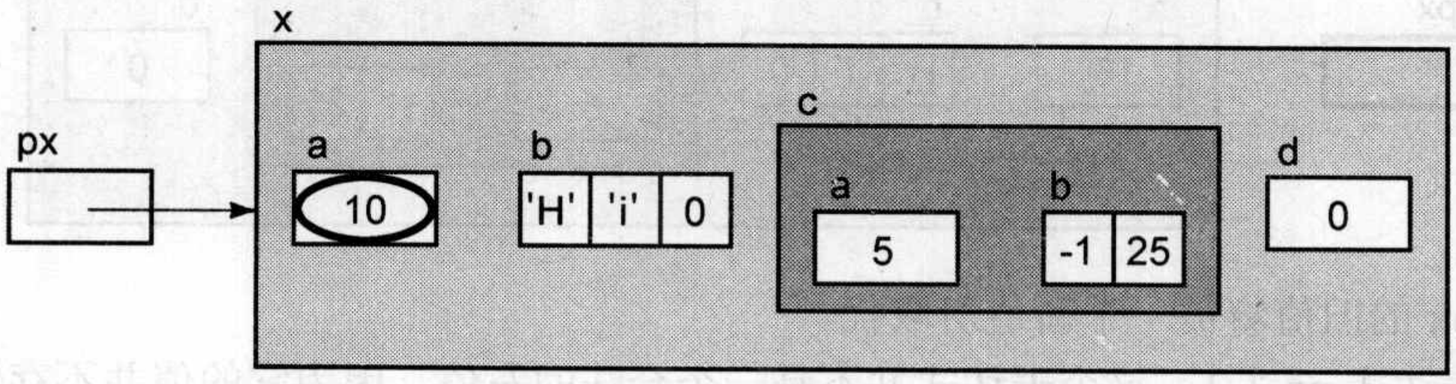


这里，结构将接受一个新值，或者更精确地说，它将接受它的所有成员的新值。作为左值，重要的是位置，而不是这个位置所保存的值。

表达式*px + 1 是非法的，因为*px 的结果是一个结构。C 语言并没有定义结构和整型值之间的加法运算。但表达式*(px + 1)又如何呢？如果 x 是一个数组的元素，这个表达式表示它后面的那个结构。但是，x 是一个标量，所以这个表达式实际上是非法的。

10.2.3 访问结构成员

现在让我们来看一下箭头操作符。表达式 px->a 的右值是：



->操作符对 px 执行间接访问操作（由实线箭头提示），它首先得到它所指向的结构，然后访问成员 a。当你拥有一个指向结构的指针但又不知道结构的名字时，便可以使用表达式 px->a。如果你知道这个结构的名字，你也可以使用功能相同的表达式 x.a。

在此，我们稍作停顿，相互比较一下表达式*px 和 px->a。在这两个表达式中，px 所保存的地址都用于寻找这个结构。但结构的第 1 个成员是 a，所以 a 的地址和结构的地址是一样的。这样 px 看上去是指向整个结构，同时指向结构的第 1 个成员：毕竟，它们具有相同的地址。但是，这个分析只有一半是正确的。尽管两个地址的值是相等的，但它们的类型不同。变量 px 被声明为一个指向结构的指针，所以表达式*px 的结果是整个结构，而不是它的第 1 个成员。

让我们创建一个指向整型的指针。

```
int    *pi;
```

我们能不能让 pi 指向整型成员 a？如果 pi 的值和 px 相同，那么表达式*pi 的结果将是成员 a。但是，表达式

```
pi = px;
```

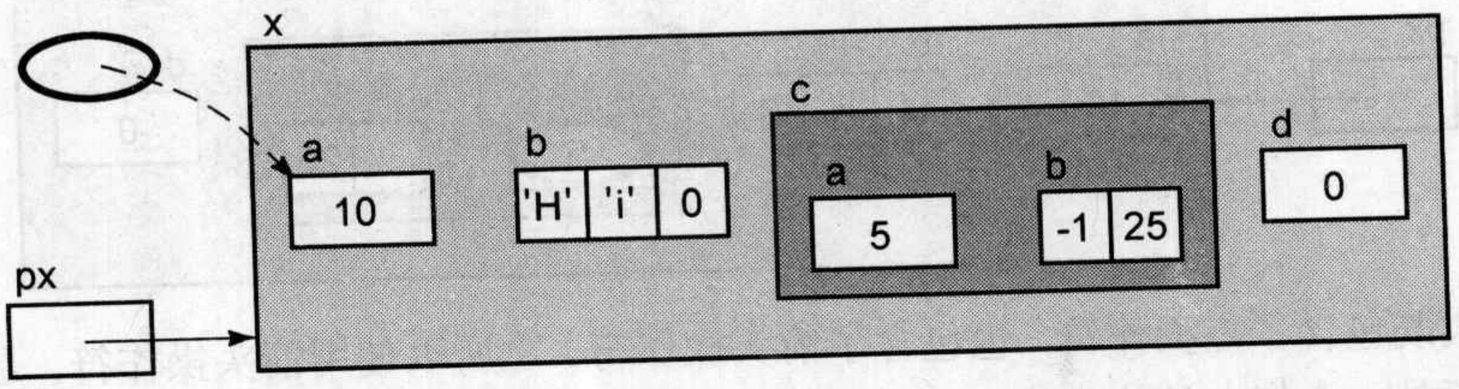
是非法的，因为它们的类型不匹配。使用强制类型转换就能奏效：

```
pi = (int *)px;
```

但这种方法是很危险的，因为它避开了编译器的类型检查。正确的表达式更为简单——使用&操作符取得一个指向 px->a 的指针：

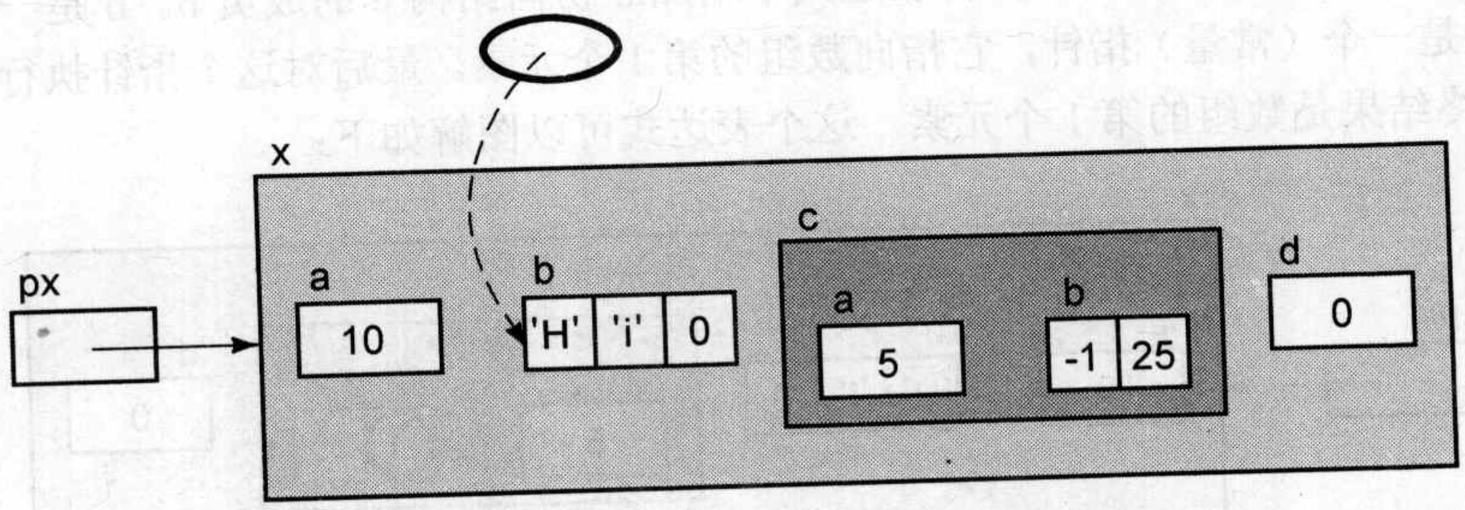
```
pi = &px->a;
```

->操作符的优先级高于&操作符的优先级，所以这个表达式无需使用括号。让我们检查一下&px->a 的图：

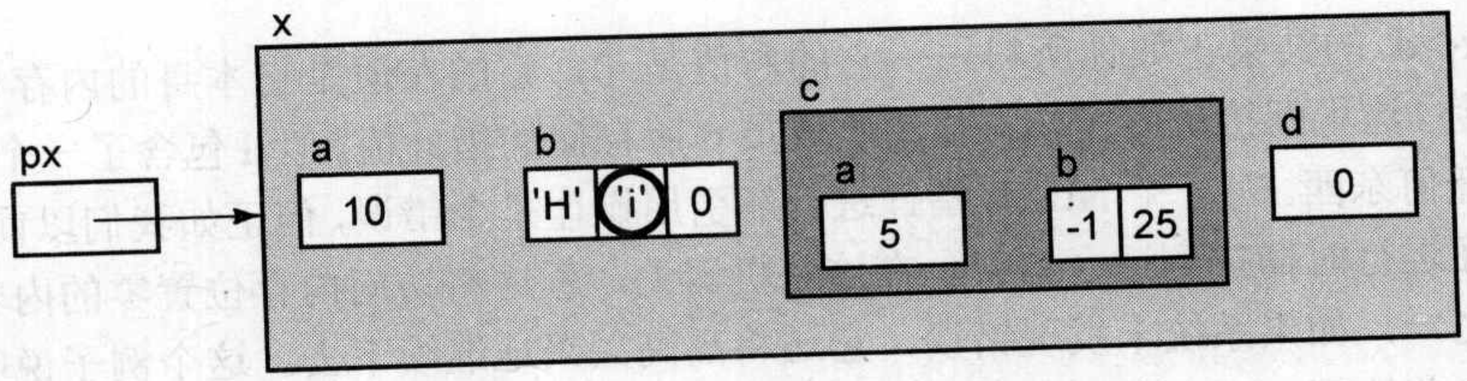


注意椭圆里的值是如何直接指向结构的成员 a 的，这与 px 相反，后者指向整个结构。在上面的赋值操作之后，pi 和 px 具有相同的值。但它们的类型是不同的，所以对它们使用间接访问操作所得的结果也不一样：*px 的结果是整个结构，*pi 的结果是一个单一的整型值。

这里还有一个使用箭头操作符的例子。表达式 px->b 的值是一个指针常量，因为 b 是一个数组。这个表达式不是一个合法的左值。下面是它的右值：

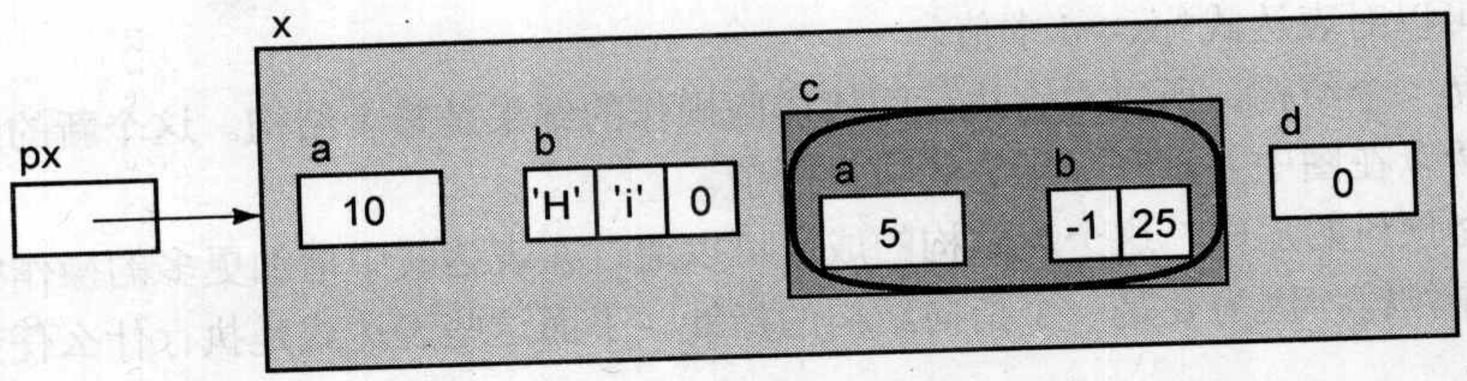


如果我们对这个表达式执行间接访问操作，它将访问数组的第 1 个元素。使用下标引用或指针运算，我们还可以访问数组的其他元素。表达式 px->b[1]访问数组的第 2 个元素，如下所示：

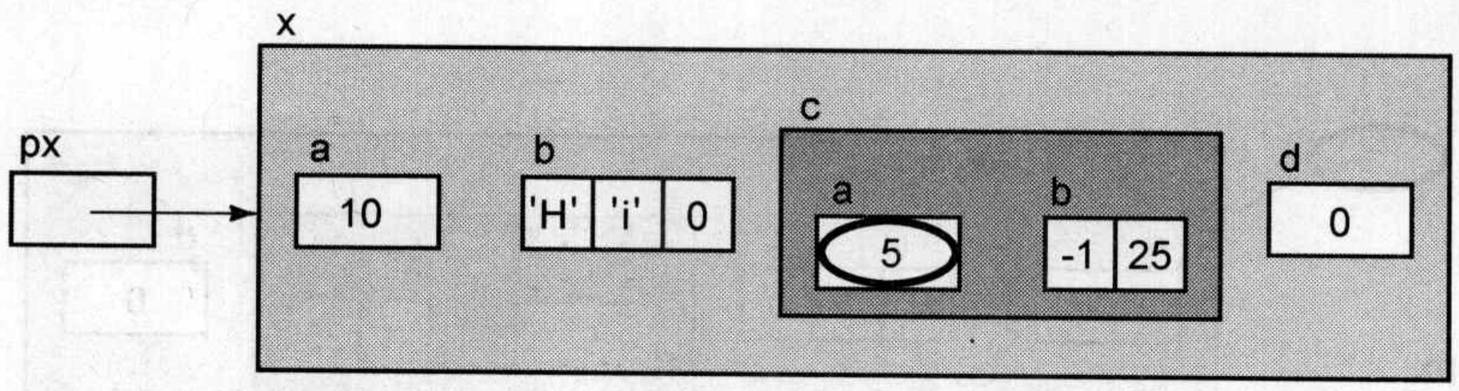


10.2.4　访问嵌套的结构

为了访问本身也是结构的成员 c，我们可以使用表达式 px->c。它的左值是整个结构。



这个表达式可以使用点操作符访问 c 的特定成员。例如，表达式 px->c.a 具有下面的右值：

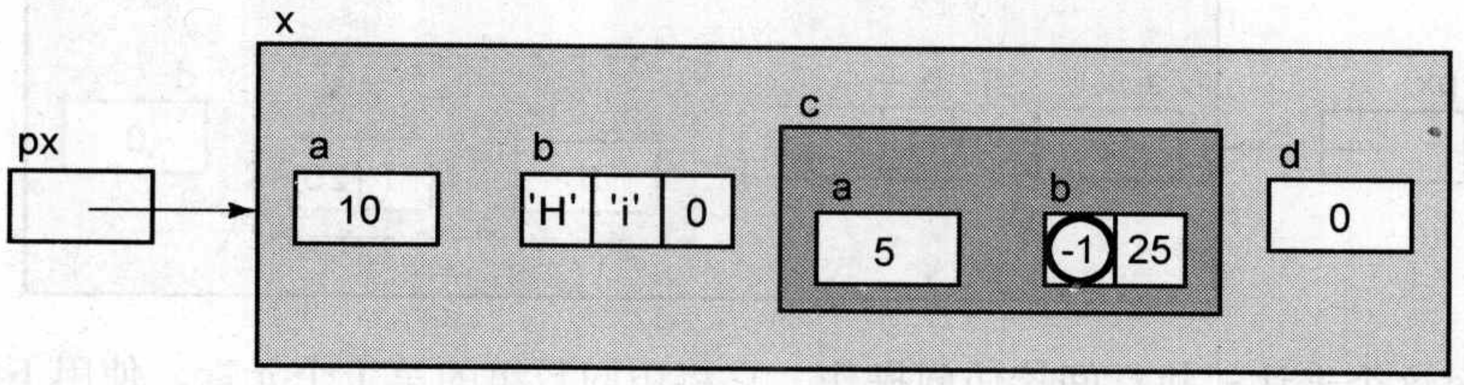


这个表达式既包含了点操作符，也包含了箭头操作符。之所以使用箭头操作符，是因为 px 并不是一个结构，而是一个指向结构的指针。接下来之所以要使用点操作符是因为 px->c 的结果并不是一个指针，而是一个结构。

这里有一个更为复杂的表达式：

```
*px->c.b
```

如果你逐步对它进行分析，这个表达式还是比较容易弄懂的。它有三个操作符，首先执行的是箭头操作符。px->c 的结果是结构 c。在表达式中增加.b 访问结构 c 的成员 b。b 是一个数组，所以 px->b.c 的结果是一个（常量）指针，它指向数组的第 1 个元素。最后对这个指针执行间接访问，所以表达式的最终结果是数组的第 1 个元素。这个表达式可以图解如下：



10.2.5 访问指针成员

表达式 px->d 的结果正如你所料——它的右值是 0，它的左值是它本身的内存位置。表达式 *px->d 更为有趣。这里间接访问操作符作用于成员 d 所存储的指针值。但 d 包含了一个 NULL 指针，所以它不指向任何东西。对一个 NULL 指针进行解引用操作是个错误，但正如我们以前讨论的那样，有些环境不会在运行时捕捉到这个错误。在这些机器上，程序将访问内存位置零的内容，把它也当作是结构成员之一，如果系统未发现错误，它还将高高兴兴地继续下去。这个例子说明了对指针进行解引用操作之前检查一下它是否有效是非常重要的。

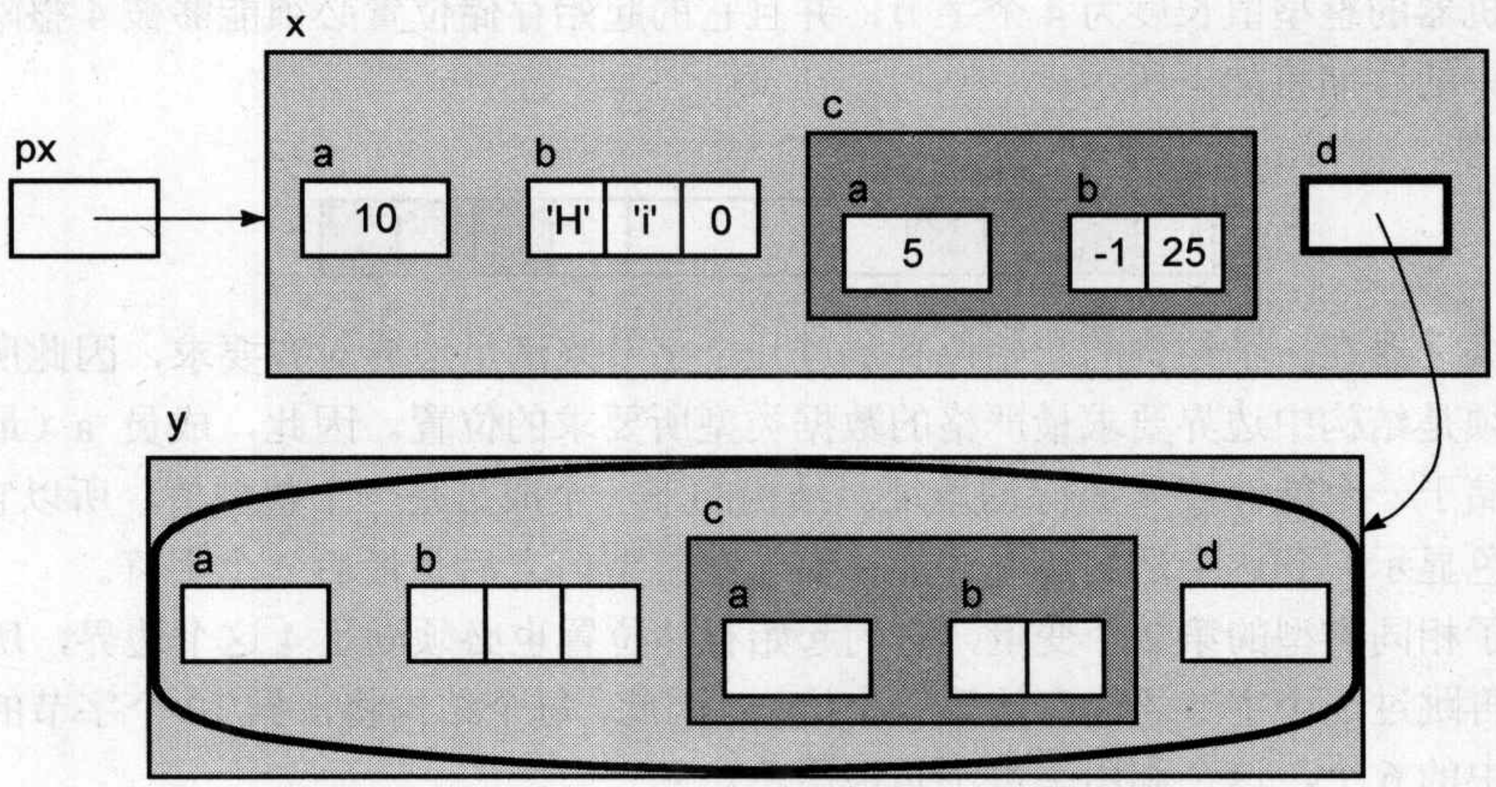
让我们创建另一个结构，并把 x.d 设置为指向它。

```
Ex      y;
x.d = &y;
```

现在我们可以对表达式*px->d 求值。

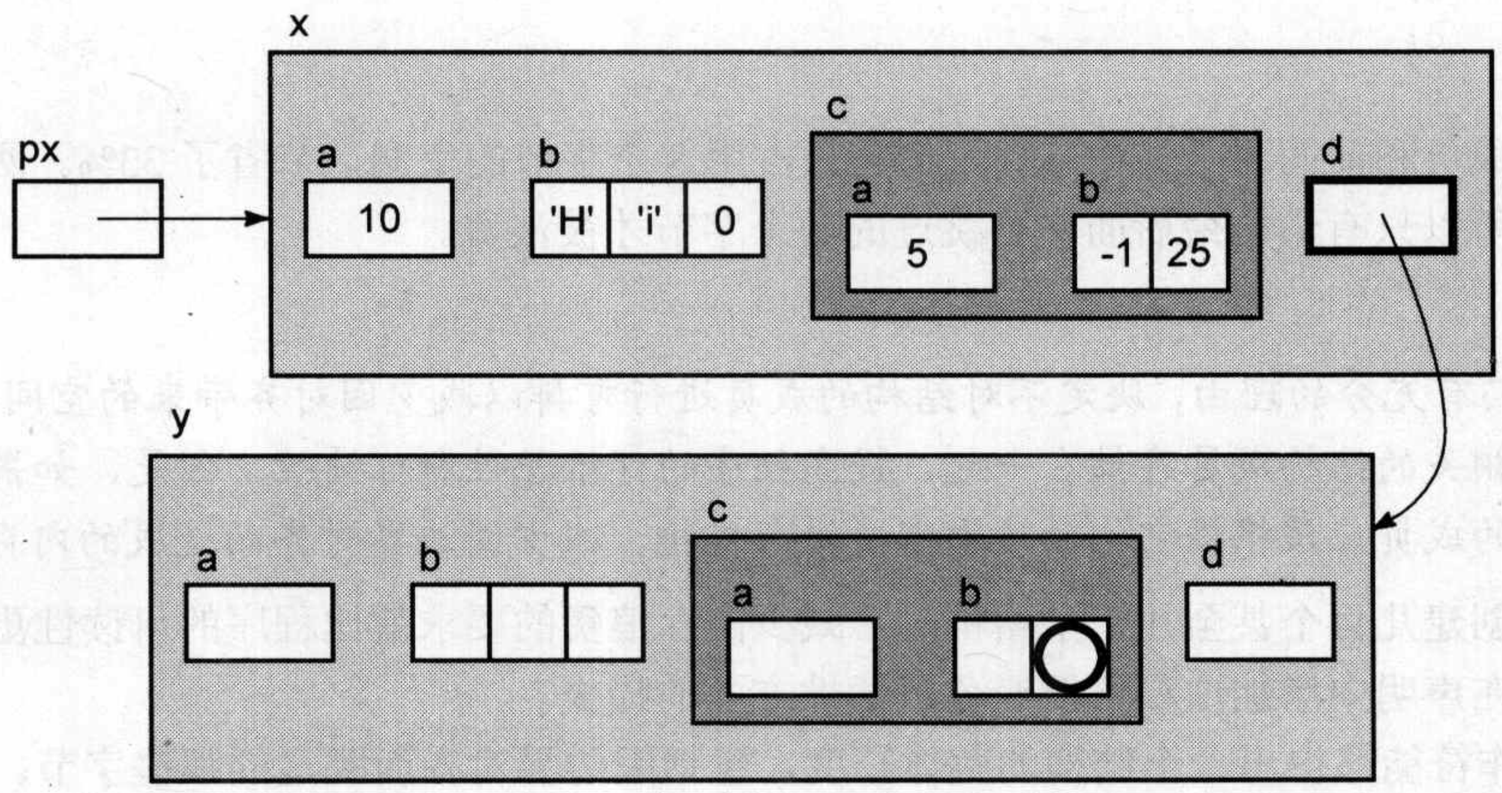
成员 d 指向一个结构，所以对它执行间接访问操作的结果是整个结构。这个新的结构并没有显式地初始化，所以在图中并没有显示它的成员的值。

正如你可能预料的那样，这个新结构的成员可以通过在表达式中增加更多的操作符进行访问。我们使用箭头操作符，因为 d 是一个指向结构的指针。下面这些表达式是执行什么任务的呢？




```
px->d->a
px->d->b
px->d->c
px->d->c.a
px->d->c.b[1]
```

最后一个表达式的右值可以图解如下：



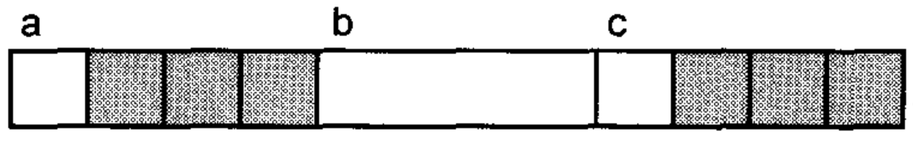
10.3　结构的存储分配

结构在内存中是如何实际存储的呢？前面例子的这张图似乎提示了结构内部包含了大量的未用空间。但这张图并不完全准确，编译器按照成员列表的顺序一个接一个地给每个成员分配内存。只有当存储成员时需要满足正确的边界对齐要求时，成员之间才可能出现用于填充的额外内存空间。

为了说明这一点，考虑下面这个结构：

```
struct  ALIGN   {
        char    a;
        int     b;
        char    c;
};
```

如果某个机器的整型值长度为 4 个字节，并且它的起始存储位置必须能够被 4 整除，那么这一个结构在内存中的存储将如下所示：



系统禁止编译器在一个结构的起始位置跳过几个字节来满足边界对齐要求，因此所有结构的起始存储位置必须是结构中边界要求最严格的数据类型所要求的位置。因此，成员 a（最左边的那个方框）必须存储于一个能够被 4 整除的地址。结构的下一个成员是一个整型值，所以它必须跳过 3 个字节（用灰色显示）到达合适的边界才能存储。在整型值之后是最后一个字符。

如果声明了相同类型的第 2 个变量，它的起始存储位置也必须满足 4 这个边界，所以第 1 个结构的后面还要再跳过 3 个字节才能存储第 2 个结构。因此，每个结构将占据 12 个字节的内存空间但实际只使用其中的 6 个，这个利用率可不是很出色。

你可以在声明中对结构的成员列表重新排列，让那些对边界要求最严格的成员首先出现，对边界要求最弱的成员最后出现。这种做法可以最大限度地减少因边界对齐而带来的空间损失。例如，下面这个结构

```
struct ALIGN2 {
int         b;
char        a;
char        c;
};
```

所包含的成员和前面那个结构一样，但它只占用 8 个字节的空间，节省了 33%。两个字符可以紧挨着存储，所以只有结构最后面需要跳过的两个字节才被浪费。

提示：

有时，我们有充分的理由，决定不对结构的成员进行重排以减少因对齐带来的空间损失。例如，我们可能想把相关的结构成员存储在一起，提高程序的可维护性和可读性。但是，如果不存在这样的理由，结构的成员应该根据它们的边界需要进行重排，减少因边界对齐而造成的内存损失。

当程序将创建几百个甚至几千个结构时，减少内存浪费的要求就比程序的可读性更为急迫。在这种情况下，在声明中增加注释可能避免可读性方面的损失。

sizeof 操作符能够得出一个结构的整体长度，包括因边界对齐而跳过的那些字节。如果你必须确定结构某个成员的实际位置，应该考虑边界对齐因素，可以使用 offsetof 宏（定义于 stddef.h）。

```
offsetof( type, member )
```

type 就是结构的类型，member 就是你需要的那个成员名。表达式的结果是一个 size_t 值，表示这个指定成员开始存储的位置距离结构开始存储的位置偏移几个字节。例如，对前面那个声明而言，

```
offsetof( struct ALIGN, b )
```

的返回值是 4。

10.4 作为函数参数的结构

结构变量是一个标量，它可以用于其他标量可以使用的任何场合。因此，把结构作为参数传递

给一个函数是合法的，但这种做法往往并不适宜。

下面的代码段取自一个程序，该程序用于操作电子现金收入记录机。下面是一个结构的声明，它包含单笔交易的信息。

```
typedef struct  {
        char    product[PRODUCT_SIZE];
        int     quantity;
        float   unit_price;
        float   total_amount;
} Transaction;
```

当交易发生时，需要涉及很多步骤，其中之一就是打印收据。让我们看看怎样用几种不同的方法来完成这项任务。

```
void
print_receipt( Transaction trans )
{
        printf( "%s\n", trans.product );
        printf( "%d @ %.2f total %.2f\n", trans.quantity,
            trans.unit_price, trans.total_amount );
}
```

如果 current_trans 是一个 Transaction 结构，我们可以像下面这样调用函数：

```
print_receipt( current_trans );
```

警告：

这个方法能够产生正确的结果，但它的效率很低，因为 C 语言的参数传值调用方式要求把参数的一份拷贝传递给函数。如果 PRODUCT_SIZE 为 20，而且在我们使用的机器上整型和浮点型都占 4 个字节，那么这个结构将占据 32 个字节的空间。要想把它作为参数进行传递，我们必须把 32 个字节复制到堆栈中，以后再丢弃。

把前面那个函数和下面这个进行比较：

```
void
print_receipt( Transaction *trans )
{
        printf( "%s\n", trans->product );
        printf( "%d @ %.2f total %.2f\n", trans->quantity,
            trans->unit_price, trans->total_amount );
}
```

这个函数可以像下面这样进行调用：

```
print_receipt( &current_trans );
```

这次传递给函数的是一个指向结构的指针。指针比整个结构要小得多，所以把它压到堆栈上效率能提高很多。传递指针另外需要付出的代价是我们必须在函数中使用间接访问来访问结构的成员。结构越大，把指向它的指针传递给函数的效率就越高。

在许多机器中，你可以把参数声明为寄存器变量，从而进一步提高指针传递方案的效率。在有些机器上，这种声明在函数的起始部分还需要一条额外的指令，用于把堆栈中的参数（参数先传递给堆栈）复制到寄存器，供函数使用。但是，如果函数对这个指针的间接访问次数超过两三次，那么使用这种方法所节省的时间将远远高于一条额外指令所花费的时间。

向函数传递指针的缺陷在于函数现在可以对调用程序的结构变量进行修改。如果我们不希望如此，可以在函数中使用 const 关键字来防止这类修改。经过这两个修改之后，现在函数的原型将如

下所示：

```
void print_receipt( register Transaction const *trans );
```

让我们前进一个步骤，对交易进行处理：计算应该支付的总额。你希望函数 comput_total_amount 能够修改结构的 total_amount 成员。要完成这项任务有三种方法，首先让我们来看一下效率最低的那种。下面这个函数

```
Transaction
compute_total_amount( Transaction trans )
{
        trans.total_amount =
            trans.quantity * trans.unit_price;
        return trans;
}
```

可以用下面这种形式进行调用：

```
current_trans = compute_total_amount( current_trans );
```

结构的一份拷贝作为参数传递给函数并被修改。然后一份修改后的结构拷贝从函数返回，所以这个结构被复制了两次。

一个稍微好点的方法是只返回修改后的值，而不是整个结构。第 2 个函数使用的就是这种方法。

```
float
compute_total_amount( Transaction trans )
{
        return trans.quantity * trans.unit_price;
}
```

但是，这个函数必须以下面这种方式进行调用：

```
current_trans.total_amount =
   compute_total_amount( current_trans );
```

这个方案比返回整个结构的那个方案强，但这个技巧只适用于计算单个值的情况。如果我们要求函数修改结构的两个或更多成员，这种方法就无能为力了。另外，它仍然存在把整个结构作为参数进行传递这个开销。更糟的是，它要求调用程序知道结构的内容，尤其是总金额字段的名字。

第 3 种方法是传递一个指针，这个方案显然要好得多：

```
void
compute_total_amount( register Transaction *trans )
{
        trans->total_amount =
            trans->quantity * trans->unit_price;
}
```

这个函数按照下面的方式进行调用：

```
compute_total_amount( &current_trans );
```

现在，调用程序的结构的字段 total_amount 被直接修改，它并不需要把整个结构作为参数传递给函数，也不需要把整个修改过的结构作为返回值返回。这个版本比前两个版本效率高得多。另外，调用程序无需知道结构的内容，所以也提高了程序的模块化程度。

什么时候你应该向函数传递一个结构而不是一个指向结构的指针呢？很少有这种情况。只有当一个结构特别的小（长度和指针相同或更小）时，结构传递方案的效率才不会输给指针传递方案。但对于绝大多数结构，传递指针显然效率更高。如果你希望函数修改结构的任何成员，也应该使用指针传递方案。

K&R C：

在非常早期的 K&R C 编译器中，你无法把结构作为参数传递给函数——编译器就是不允许这样做。后期的 K&R C 编译器允许传递结构参数。但是，这些编译器都不支持 const，所以防止程序修改结构参数的唯一办法就是向函数传递一份结构的拷贝。

10.5　位段

关于结构，我们最后还必须提到它们实现位段(bit field)的能力。位段的声明和结构类似，但它的成员是一个或多个位的字段。这些不同长度的字段实际上存储于一个或多个整型变量中。

位段的声明和任何普通的结构成员声明相同，但有两个例外。首先，位段成员必须声明为 int、signed int 或 unsigned int 类型。其次，在成员名的后面是一个冒号和一个整数，这个整数指定该位段所占用的位的数目。

提示：

用 signed 或 unsigned 整数显式地声明位段是个好主意。如果把位段声明为 int 类型，它究竟被解释为有符号数还是无符号数是由编译器决定的。

提示：

注重可移植性的程序应该避免使用位段。由于下面这些与实现有关的依赖性，位段在不同的系统中可能有不同的结果。

1. int 位段被当作有符号数还是无符号数。

2. 位段中位的最大数目。许多编译器把位段成员的长度限制在一个整型值的长度之内，所以一个能够运行于 32 位整数的机器上的位段声明可能在 16 位整数的机器上无法运行。

3. 位段中的成员在内存中是从左向右分配的还是从右向左分配的。

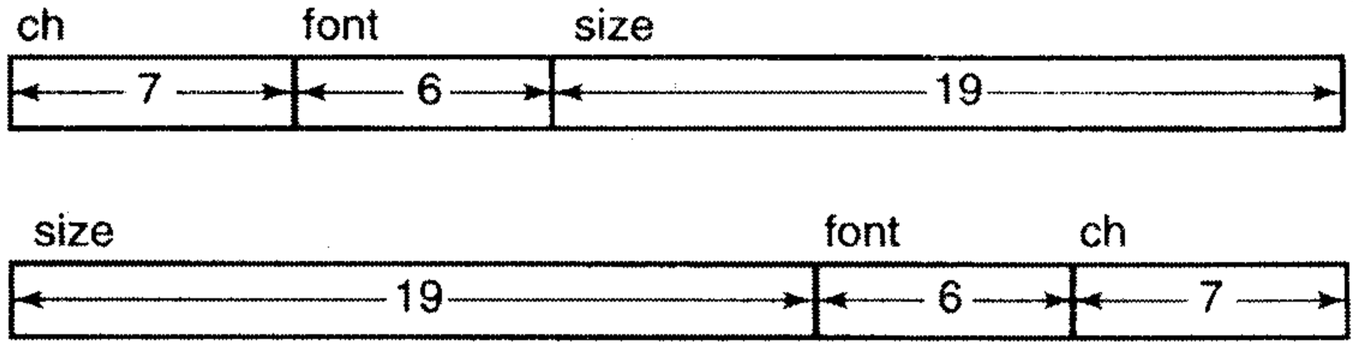
4. 当一个声明指定了两个位段，第 2 个位段比较大，无法容纳于第 1 个位段剩余的位时，编译器有可能把第 2 个位段放在内存的下一个字，也可能直接放在第 1 个位段后面，从而在两个内存位置的边界上形成重叠。

下面是一个位段声明的例子：

```
struct  CHAR    {
        unsigned ch     : 7;
        unsigned font   : 6;
        unsigned size   : 19;
};
struct  CHAR    ch1;
```

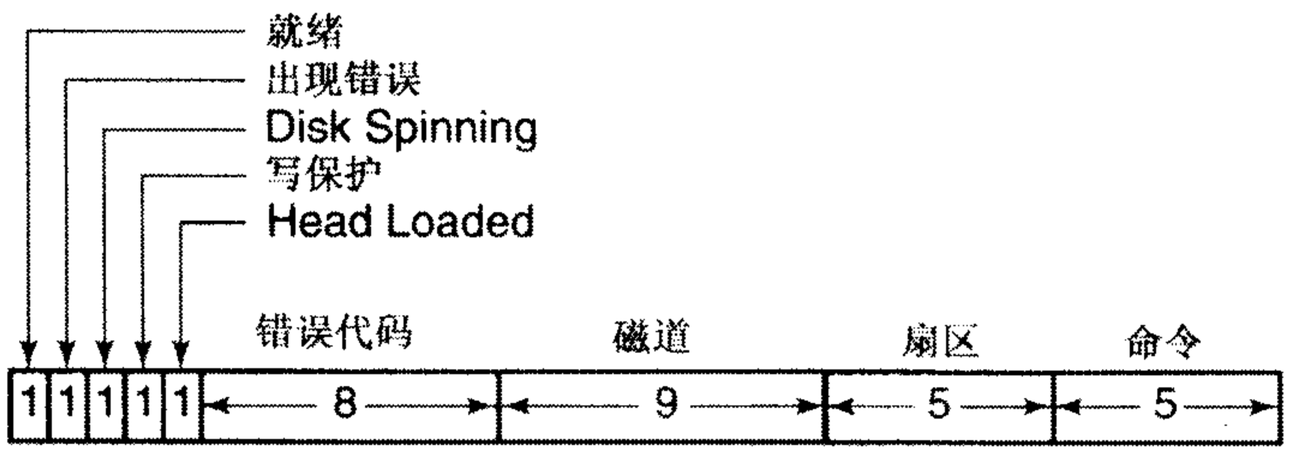
这个声明取自一个文本格式化程序，它可以处理多达 128 个不同的字符值（需要 7 个位）、64 种不同的字体（需要 6 个位）以及 0 到 524 287 个单位的长度。这个 size 位段过于庞大，无法容纳于一个短整型，但其余的位段都比一个字符还短。位段使程序员能够利用存储 ch 和 font 所剩余的位来增加 size 的位数，这样就避免了声明一个 32 位的整数来存储 size 位段。

许多 16 位整数机器的编译器会把这个声明标志为非法，因为最后一个位段的长度超过了整型的长度。但在 32 位的机器上，这个声明将根据下面两种可能的方法创建 ch1。



这个例子说明了一个使用位段的好理由：它能够把长度为奇数的数据包装在一起，节省存储空间。当程序需要使用成千上万的这类结构时，这种节省方法就会变得相当重要。

另一个使用位段的理由是由于它们可以很方便地访问一个整型值的部分内容。让我们研究一个例子，它可能出现于操作系统中。用于操作软盘的代码必须与磁盘控制器通信。这些设备控制器常常包含了几个寄存器，每个寄存器又包含了许多包装在一个整型值内的不同的值。位段就是一种方便的访问这些单一值的方法。假定磁盘控制器其中一个寄存器是如下定义的：



前 5 个位段每个都占 1 位，其余几个位段则更长一些。在一个从右向左分配位段的机器上，下面这个声明允许程序方便地对这个寄存器的不同位段进行访问。

```
struct  DISK_REGISTER_FORMAT    {
        unsigned        command         : 5;
        unsigned        sector          : 5;
        unsigned        track           : 9;
        unsigned        error_code      : 8;
        unsigned        head_loaded     : 1;
        unsigned        write_protect   : 1;
        unsigned        disk_spinning   : 1;
        unsigned        error_occurred  : 1;
        unsigned        ready           : 1;
};
```

假如磁盘寄存器是在内存地址 0xc0200142 进行访问的，我们可以声明下面的指针常量：

```
#define DISK_REGISTER   \
      ((struct DISK_REGISTER_FORMAT *)0xc0200142)
```

做了这个准备工作后，实际需要访问磁盘寄存器的代码就变得简单多了，如下面的代码段所示。

```
/*
** 告诉控制器从哪个扇区哪个磁道开始读取。
*/
DISK_REGISTER->sector = new_sector;
DISK_REGISTER->track = new_track;
DISK_REGISTER->command = READ;

/*
** 等待，直到操作完成（ready 变量变成真）。
```

```
*/
while(  ! DISK_REGISTER->ready )
      ;

/*
** 检查错误。
*/
if( DISK_REGISTER->error_occurred ) {
switch( DISK_REGISTER->error_code ) {
...
```

使用位段只是基于方便的目的。任何可以用位段实现的任务都可以使用移位和屏蔽来实现。例如，下面代码段的功能和前一个例子中第 1 个赋值的功能完全一样。

```
#define DISK_REGISTER   (unsigned int *)0xc0200142

*DISK_REGISTER &= 0xfffffc1f;
*DISK_REGISTER |= ( new_sector & 0x1f ) << 5;
```

第 1 条赋值语句使用位 AND 操作把 sector 字段清零，但不影响其他的位段。第 2 条赋值语句用于接受 new_sector 的值，AND 操作可以确保这个值不会超过这个位段的宽度。接着，把它左移到合适的位置，然后使用位 OR 操作把这个字段设置为需要的值。

提示：

在源代码中，用位段表示这个处理过程更为简单一些，但在目标代码中，这两种方法并不存在任何区别。无论是否使用位段，相同的移位和屏蔽操作都是必需的。位段提供的唯一优点是简化了源代码。这个优点必须与位段的移植性较弱这个缺点进行权衡。

10.6　联合

和结构相比，联合（union）可以说是另一种动物了。联合的声明和结构类似，但它的行为方式却和结构不同。联合的所有成员引用的是**内存中的相同位置**。当你想在不同的时刻把不同的东西存储于同一个位置时，就可以使用联合。

首先，让我们看一个简单的例子。

```
union   {
        float   f;
        int     i;
} fi;
```

在一个浮点型和整型都是 32 位的机器上，变量 fi 只占据内存中一个 32 位的字。如果成员 f 被使用，这个字就作为浮点值访问；如果成员 i 被使用，这个字就作为整型值访问。所以，下面这段代码

```
fi.f = 3.14159;
printf("%d\n", fi.i );
```

首先把 π 的浮点表示形式存储于 fi，然后把这些**相同的位**当作一个整型值打印输出。注意这两个成员所引用的位相同，仅有的区别在于每个成员的类型决定了这些位被如何解释。

为什么人们有时想使用类似此例的形式呢？如果你想看看浮点数是如何存储在一种特定的机器中但又对其他东西不感兴趣，联合就可能有所帮助。这里有一个更为现实的例子。BASIC 解释器的

任务之一就是记住程序所使用的变量的值。BASIC 提供了几种不同类型的变量，所以每个变量的类型必须和它的值一起存储。这里有一个结构，用于保存这个信息，但它的效率不高。

```
struct  VARIABLE        {
        enum    { INT, FLOAT, STRING }  type;
        int     int_value;
        float   float_value;
        char    *string_value;
};
```

当 BASIC 程序中的一个变量被创建时，解释器就创建一个这样的结构并记录变量的类型。然后，根据变量的类型，把变量的值存储在这三个值字段的其中一个。

这个结构的低效之处在于它所占用的内存——每个 VARIABLE 结构存在两个未使用的值字段。联合就可以减少这种浪费，它把这三个值字段的每一个都存储于同一个内存位置。这三个字段并不会冲突，因为每个变量只可能具有一种类型，这样在某一时刻，联合的这几个字段只有一个被使用。

```
struct  VARIABLE        {
        enum    { INT, FLOAT, STRING }  type;
        union   {
                int     i;
                float   f;
                char    *s;
        } value;
};
```

现在，对于整型变量，你将在 type 字段设置为 INT，并把整型值存储于 value.i 字段。对于浮点值，你将使用 value.f 字段。当以后得到这个变量的值时，对 type 字段进行检查决定使用哪个值字段。这个选择决定内存位置如何被访问，所以同一个位置可以用于存储这三种不同类型的值。注意编译器并不对 type 字段进行检查证实程序使用的是正确的联合成员。维护并检查 type 字段是程序员的责任。

如果联合的各个成员具有不同的长度，联合的长度就是它最长成员的长度。下一节将讨论这种情况。

10.6.1 变体记录

让我们讨论一个例子，实现一种在 Pascal 和 Modula 中被称为变体记录(variant record)的东西。从概念上说，这就是我们刚刚讨论过的那个情况——内存中某个特定的区域将在不同的时刻存储不同类型的值。但是，在现在这个情况下，这些值比简单的整型或浮点型更为复杂。它们的每一个都是一个完整的结构。

下面这个例子取自一个存货系统，它记录了两种不同的实体：零件(part)和装配件(subassembly)。零件就是一种小配件，从其他生产厂家购得。它具有各种不同的属性如购买来源、购买价格等。装配件是我们制造的东西，它由一些零件及其他装配件组成。

前两个结构指定每个零件和装配件必须存储的内容。

```
struct  PARTINFO        {
        int     cost;
        int     supplier;
        ...
};

struct  SUBASSYINFO     {
        int     n_parts;
        struct  {
```

```
                char    partno[10];
                short   quan;
        } parts[MAXPARTS];
};
```

接下来的存货（inventory）记录包含了每个项目的一般信息，并包括了一个联合，或者用于存储零件信息，或者用于存储装配件信息。

```
struct  INVREC  {
        char    partno[10];
        int     quan;
        enum    { PART, SUBASSY }       type;
        union   {
                struct  PARTINFO        part;
                struct  SUBASSYINFO     subassy;
        } info;
};
```

这里有一些语句，用于操作名叫 rec 的 INVREC 结构变量。

```
if( rec.type == PART ){
        y = rec.info.part.cost;
        z = rec.info.part.supplier;
}
else {
        y = rec.info.subassy.nparts;
        z = rec.info.subassy.parts[0].quan;
}
```

尽管并非十分真实，但这段代码说明了如何访问联合的每个成员。语句的第 1 部分获得成本(cost)值和零件的供应商(supplier)，语句的第 2 部分获得一个装配件中不同零件的编号以及第 1 个零件的数量。

在一个成员长度不同的联合里，分配给联合的内存数量取决于它的最长成员的长度。这样，联合的长度总是足以容纳它最大的成员。如果这些成员的长度相差悬殊，当存储长度较短的成员时，浪费的空间是相当可观的。在这种情况下，更好的方法是在联合中存储指向不同成员的指针而不是直接存储成员本身。所有指针的长度都是相同的，这样就解决了内存浪费的问题。当它决定需要使用哪个成员时，就分配正确数量的内存来存储它。第 11 章将讲述动态内存分配，它包含了一个例子用于说明这种技巧。

10.6.2　联合的初始化

联合变量可以被初始化，但这个初始值必须是联合第 1 个成员的类型，而且它必须位于一对花括号里面。例如，

```
union {
        int     a;
        float   b;
        char    c[4];
} x = { 5 };
```

把 x.a 初始化为 5。

我们不能把这个类量初始化为一个浮点值或字符值。如果给出的初始值是任何其他类型，它就会转换（如果可能的话）为一个整数并赋值给 x.a。

10.7 总结

在结构中，不同类型的值可以存储在一起。结构中的值称为成员，它们是通过名字访问的。结构变量是一个标量，可以出现在普通标量变量可以出现的任何场合。

结构的声明列出了结构包含的成员列表。不同的结构声明即使它们的成员列表相同也被认为是不同的类型。结构标签是一个名字，它与一个成员列表相关联。你可以使用结构标签在不同的声明中创建相同类型的结构变量，这样就不用每次在声明中重复成员列表。typedef 也可以用于实现这个目标。

结构的成员可以是标量、数组或指针。结构也可以包含本身也是结构的成员。在不同的结构中出现同样的成员名是不会引起冲突的。你使用点操作符访问结构变量的成员。如果你拥有一个指向结构的指针，你可以使用箭头操作符访问这个结构的成员。

结构不能包含类型也是这个结构的成员，但它的成员可以是一个指向这个结构的指针。这个技巧常常用于链式数据结构中。为了声明两个结构，每个结构都包含一个指向对方的指针的成员，我们需要使用不完整的声明来定义一个结构标签名。结构变量可以用一个由花括号包围的值列表进行初始化。这些值的类型必须适合它所初始化的那些成员。

编译器为一个结构变量的成员分配内存时要满足它们的边界对齐要求。在实现结构存储的边界对齐时，可能会浪费一部分内存空间。根据边界对齐要求降序排列结构成员可以最大限度地减少结构存储中浪费的内存空间。sizeof 返回的值包含了结构中浪费的内存空间。

结构可以作为参数传递给函数，也可以作为返回值从函数返回。但是，向函数传递一个指向结构的指针往往效率更高。在结构指针参数的声明中可以加上 const 关键字防止函数修改指针所指向的结构。

位段是结构的一种，但它的成员长度以位为单位指定。位段声明在本质上是不可移植的，因为它涉及许多与实现有关的因素。但是，位段允许你把长度为奇数的值包装在一起以节省存储空间。源代码如果需要访问一个值内部任意的一些位，使用位段比较简便。

一个联合的所有成员都存储于同一个内存位置。通过访问不同类型的联合成员，内存中相同的位组合可以被解释为不同的东西。联合在实现变体记录时很有用，但程序员必须负责确认实际存储的是哪个变体并选择正确的联合成员以便访问数据。联合变量也可以进行初始化，但初始值必须与联合第 1 个成员的类型匹配。

10.8 警告的总结

1. 具有相同成员列表的结构声明产生不同类型的变量。
2. 使用 typedef 为一个自引用的结构定义名字时应该小心。
3. 向函数传递结构参数是低效的。

10.9 编程提示的总结

1. 把结构标签声明和结构的 typedef 声明放在头文件中，当源文件需要这些声明时可以通过

#include 指令把它们包含进来。

2. 结构成员的最佳排列形式并不一定就是考虑边界对齐而浪费内存空间最少的那种排列形式。
3. 把位段成员显式地声明为 signed int 或 unsigned int 类型。
4. 位段是不可移植的。
5. 位段使源代码中位的操作表达得更为清楚。

10.10　问题

1. 成员和数组元素有什么区别？

2. 结构名和数组名有什么不同？

3. 结构声明的语法有几个可选部分。请列出所有合法的结构声明形式，并解释每一个是如何实现的。

4. 下面的程序段有没有错误？如果有，错误有哪里？

```
struct abc {
        int     a;
        int     b;
        int     c;
};
...
abc.a = 25;
abc.b = 15;
abc.c = -1
```

5. 下面的程序段有没有错误？如果有，错误有哪里？

```
typedef struct {
        int     a;
        int     b;
        int     c;
} abc;
...
abc.a = 25;
abc.b = 15;
abc.c = -1
```

6. 完成下面声明中对 x 的初始化，使成员 a 为 3，b 为字符串“hello”，c 为 0。你可以假设 x 存储于静态内存中。

```
struct  {
        int     a;
        char    b[10];
        float   c;
} x =
```

7. 考虑下面这些声明和数据。

```
struct NODE {
        int a;
        struct NODE *b;
        struct NODE *c;
};

struct NODE     nodes[5] = {
        { 5,    nodes + 3,      NULL },
        { 15,   nodes + 4,      nodes + 3 },
        { 22,   NULL,           nodes + 4 },
        { 12,   nodes + 1,      nodes },
        { 18,   nodes + 2,      nodes + 1 }
```

```
};
(Other declarations...)
struct NODE     *np     = nodes + 2;
struct NODE     **npp   = &nodes[1].b;
```

对下面每个表达式求值，并写出它的值。同时，写明任何表达式求值过程中可能出现的副作用。你应该用最初显示的值对每个表达式求值（也就是说，不要使用某个表达式的结果来对下一个表达式求值）。假定 nodes 数组在内存中的起始位置为 200，并且在这台机器上整数和指针的长度都是 4 个字节。

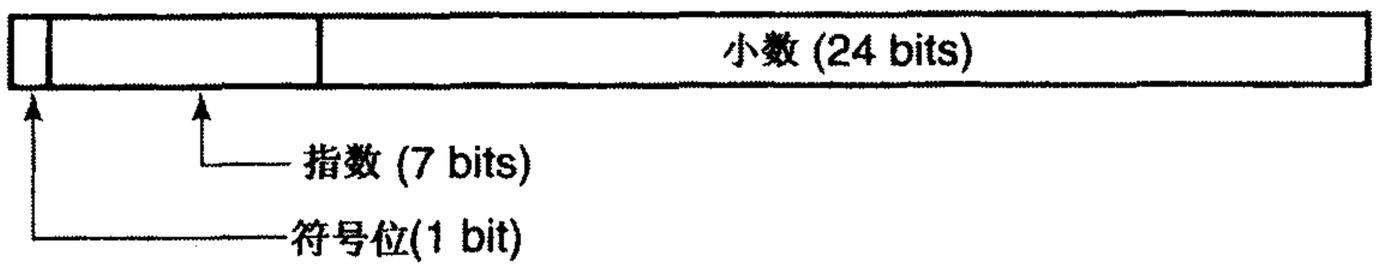
表达式	值	表达式	值
nodes	_____	&nodes[3].c->a	_____
nodes.a	_____	&nodes->a	_____
nodes[3].a	_____	np	_____
nodes[3].c	_____	np->a	_____
nodes[3].c->a	_____	np->c->c->a	_____
*nodes	_____	npp	_____
*nodes.a	_____	npp->a	_____
(*nodes).a	_____	*npp	_____
nodes->a	_____	**npp	_____
nodes[3].b->b	_____	*npp->a	_____
*nodes[3].b->b	_____	(*npp)->a	_____
&nodes	_____	&np	_____
&nodes[3].a	_____	&np->a	_____
&nodes[3].c	_____	&np->c->c->a	_____

8. 在一个 16 位的机器上，下面这个结构由于边界对齐浪费了多少空间？在一个 32 位的机器上又是如何？

```
struct  {
        char    a;
        int     b;
        char    c;
};
```

9. 至少说出两个位段为什么不可移植的理由。

10. 编写一个声明，允许根据下面的格式方便地访问一个浮点值的单独部分。



11. 如果不使用位段，你怎样实现下面这段代码的功能？假定你使用的是一台 16 位的机器，它从左向右为位段分配内存。

```
struct  {
        int     a:4;
        int     b:8;
        int     c:3;
        int     d:1;
} x;
...
x.a = aaa;
x.b = bbb;
x.c = ccc;
x.d = ddd;
```

12. 下面这个代码段将打印出什么？

```
struct  {
        int     a:2;
} x;
...
x.a = 1;
x.a += 1;
printf( "%d\n", x.a );
```

13. 下面的代码段有没有错误？如果有，错误有哪里？

```
union   {
        int     a;
        float   b;
        char    c;
} x;
...
x.a = 25;
x.b = 3.14;
x.c = 'x';
printf( "%d %g %c\n", x.a, x.b, x.c );
```

14. 假定有一些信息已经赋值给一个联合变量，我们该如何正确地提取这个信息呢？

15. 下面的结构可以被一个 BASIC 解释器使用，用于记住变量的类型和值。

```
struct  VARIABLE {

        enum    { INT, FLOAT, STRING }  type;
        union   {
                int     i;
                float   f;
                char    *s;
        } value;
};
```

如果结构改写成下面这种形式，会有什么不同呢？

```
struct  VARIABLE        {
        enum    { INT, FLOAT, STRING }  type;
        union   {
                int     i;
                float   f;
                char    s[MAX_STRING_LENGTH];
        } value;
};
```

10.11　编程练习

★★ 1. 当你拨打长途电话时，电话公司所保存的信息包括你拨打电话的日期和时间。它还包括三个电话号码：你使用的那个电话、你呼叫的那个电话以及你付账的那个电话。

这些电话号码的每一个都由三个部分组成：区号、交换台和站号码。请为这些记账信息编写一个结构声明。

★★ 2. 为一个信息系统编写一个声明，它用于记录每个汽车零售商的销售情况。每份销售记录必须包括下列数据。字符串值的最大长度不包括其结尾的 NUL 字节。

顾客名字(customer's name)	string(20)
顾客地址(customer's address)	string(40)
模型(model)	string(20)

销售时可能出现三种不同类型的交易：全额现金销售、贷款销售和租赁。对于全额现金销售，你还必须保存下面这些附加信息：

生产厂家建议零售价(manufacturer's suggested retail price)	float
实际售出价格(actual selling price)	float
营业税(sales tax)	float
许可费用(licensing fee)	float

对于租赁，你必须保存下面这些附加信息：

生产厂家建议零售价(manufacturer's suggested retail price)	float
实际售出价格(actual selling price)	float
预付定金(down payment)	float
安全抵押(security deposit)	float
月付金额(monthly payment)	float
租赁期限(lease term)	int

对于贷款销售，你必须保存下面这些附加信息：

生产厂家建议零售价(manufacturer's suggested retail price)	float
实际售出价格(actual selling price)	float
营业税(sales tax)	float
许可费用(licensing fee)	float
预付定金(doun payment)	float
贷款期限(loan duration)	int
贷款利率(interest rate)	float
月付金额(monthly payment)	float
银行名称(name of bank)	string(20)

3. 计算机的任务之一就是对程序的指令进行解码，确定采取何种操作。在许多机器中，由于不同的指令具有不同的格式，解码过程被复杂化了。在某个特定的机器上，每个指令的长度都是 16 位，并实现了下列各种不同的指令格式。位是从右向左进行标记的。

单操作数指令		双操作数指令		转移指令	
位	字段名	位	字段名	位	字段名
0–2	dst_reg	0–2	dst_reg	0–7	offset
3–5	dst_mode	3–5	dst_mode	8–15	opcode
6–15	opcode	6–8	src_reg		
		9–11	src_mode		
		12–15	opcode		

源寄存器指令		其余指令	
位	字段名	位	字段名
0-2	dst_reg	0-15	opcode
3-5	dst_mode		
6-8	src_reg		
9-15	opcode		

你的任务是编写一个声明，允许程序用这些格式中的任何一种形式对指令进行解释。你的声明同时必须有一个名叫 addr 的 unsigned short 类型字段，可以访问所有的 16 位值。在你的声明中使用 typedef 来创建一个新的类型，称为 machine_inst。给定下面的声明：

```
machine_inst  x;
```

下面的表达式应该访问它所指定的位。

表达式	位
x.addr	0–15
x.misc.opcode	0–15
x.branch.opcode	8–15
x.sgl_op.dst_mode	3–5
x.reg_src.src_reg	6–8
x.dbl_op.opcode	12–15