第7章　结构、联合、枚举和用户定义的类型

C语言提供了创建用户自定义数据类型的五种不同方式。

1. 结构：这是在一个名字下的一组变量，有时也称为聚集（aggregate）数据类型。
2. 位域：这是结构的一种变形，允许对字中的位进行访问。
3. 联合：这可以将同一块内存定义为两种或多种不同的变量类型。
4. 枚举：这是一个命名的整数常量列表。
5. typedef关键字：这为一个已存在的类型定义一个新的名字。

C++支持上面的所有方式并添加了类，类将在第二部分描述。这里介绍创建自定义数据类型的其他方法。

注意： 在C++中，结构和联合都具有面向对象和非面向对象的属性。本章仅讨论它们的类C的、非面向对象的特点，面向对象的特点将在本书后面描述。

7.1　结构

结构是用同一名字引用的变量的集合，它提供了将相关信息组织在一起的手段。结构声明形成一个用于创建结构对象（即结构的一个实例）的模板。组成结构的变量称为成员（结构成员一般也称为元素或域）。

一般情况下，结构的所有成员在逻辑上都是相关的。例如，在邮件列表中，姓名和住址通常就是用结构表示的。下面的代码段显示了如何声明一个定义了姓名和地址域的结构。关键字struct告诉编译器已声明了一个结构。

```
struct addr
{
  char name[ 30] ;
  char street[ 40] ;
  char city[ 20] ;
  char state[ 3] ;
  unsigned long int zip;
} ;
```

注意，这个声明用分号结束。这是因为一个结构声明就是一条语句。结构的类型名是addr，addr标识了这个特殊的数据结构，并成为其类型限定符。

此时，实际上没有创建变量，仅仅定义了数据的形式。在定义一个结构时，你正在定义一个复合变量类型，而不是一个变量，除非声明了一个实际上不存在的变量。在C中，要声明一个addr类型的变量（即一个物理对象），必须写成如下形式：

```
struct addr addr_info;
```

这条语句声明了一个类型为addr的变量addr_info。在C++中，可以使用下面这种简便的形式：

```
addr addr_info;
```

可以看到，不需要关键字struct。在C++中，一旦声明了一个结构，就可以使用它的类型名声明这种类型的变量，而不需要在前面加上关键字struct。会有这种区别的原因在于：C中的结构名不是一个完整的类型名。事实上，标准C把结构名称为标记。在C中，当声明变量时，必须在标记前加上关键字struct。而在C++中，结构名是一个完整的类型名，可以被其自身使用来定义变量。然而，要记住，在C++的程序中使用C的类型定义也是合法的。因为本书第一部分中的程序对C和C++都是合法的，它们将使用C的声明方法。记住C++支持简便形式。

声明了结构变量（例如addr_info）后，编译器自动为所有变量分配足够的内存空间。图7.1展示了addr_info在内存中是如何存放的。这里假定字符占1个字节，长整型占4个字节。

可以在声明结构的同时，声明一个或多个变量，例如：

```
struct addr {
  char name[ 30] ;
  char street[ 40] ;
  char city[ 20] ;
  char state[ 3] ;
  unsigned long int zip;
} addr_info, binfo, cinfo;
```

定义了一个结构类型addr，并声明了这个类型的变量addr_info，binfo，cinfo。重要的是要理解每个结构对象包含它自己的结构成员的副本。例如，binfo的zip域与cinfo的zip域是不同的，因此在binfo中改变zip不会影响cinfo中的zip。

如果只需要一个结构变量，则不必使用结构类型名。这意味着下面的结构：

```
struct {
  char name[ 30] ;
  char street[ 40] ;
  char city[ 20] ;
  char state[ 3] ;
  unsigned long int zip;
} addr_info;
```

声明了一个结构变量addr_info，就像前面的结构一样。

结构声明的一般形式为：

```
struct struct-type-name {
  type member-name;
  type member-name;
  type member-name;
  .
  .
  .
} structure-variables;
```

其中，struct-type-name或structure variables可以省略，但不能同时省略两个。

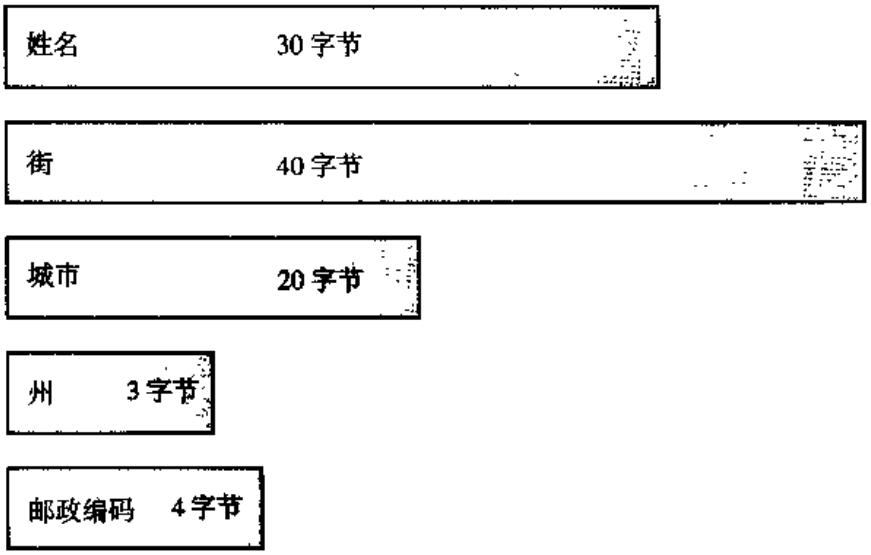

图 7.1 内存中的 addr_info 结构

7.1.1 访问结构成员

可以使用点(.)运算符来访问结构的各个成员。例如，下面的代码把邮政编码 12345 赋给前面声明过的结构变量 addr_info 中的 zip 域。

```
addr_info.zip = 12345;
```

在结构变量名后面跟有一个点和引用各个成员的成员名。访问结构成员的一般形式为：

```
structure-name.member-name
```

因此，要把邮政编码打印到屏幕上，可以这样写：

```
printf("%lu", addr_info.zip);
```

该语句打印结构变量 addr_info 的 zip 变量中包含的邮政编码。

同样，可以使用字符数组 addr_info.name 来调用 gets( )，如下所示：

```
gets(addr_info.name);
```

它传递一个指向成员 name 起始位置的字符指针。

因为 name 是一个字符数组，因此，如果要访问 addr_info.name 的各个字符，可对 name 使用下标。例如，可以用下面的代码一次一个字符地打印 addr_info.mame 的内容：

```
register int t;

for(t=0; addr_info.name[t]; ++t)
  putchar(addr_info.name[t]);
```

7.1.2 结构赋值

可以把一个结构包含的信息赋给同一类型的另一个结构，也就是说，并非一定要将所有成员分别赋值。下面的程序演示了结构赋值：

```
#include <stdio.h>

int main(void)
```

```
{
  struct {
    int a;
    int b;
  } x, y;

  x.a = 10;

  y = x; /* assign one structure to another */

  printf("%d", y.a);

  return 0;
}
```

赋值后，y.a 的值是 10。

7.2 结构数组

结构最常见的用途之一是用在数组中。要声明结构数组，必须先定义一个结构，然后声明这个类型的数组变量。例如，要声明一个类型为 addr、由 100 个成员组成的结构数组，可以这样写：

```
struct addr addr_info[100];
```

这建立了结构 addr 的 100 个变量。

要访问某个特定的结构，应给结构名加上下标。例如，要打印结构 3 的 zip，可以写为：

```
printf("%lu", addr_info[2].zip);
```

和其他数组变量一样，结构数组的下标也从 0 开始。

7.3 向函数传递结构

本节专门讨论将结构及其成员传递给函数这一问题。

7.3.1 向函数传递结构成员

向函数传递结构变量的成员，实际上是将该成员的值传给了那个函数。因此，传递的是简单变量（当然，除非那个元素是复杂变量，比如字符数组）。例如，考虑下面的结构：

```
struct fred
{
  char x;
  int y;
  float z;
  char s[10];
} mike;
```

下面的范例将每个成员传给函数：

```
func(mike.x);      /* passes character value of x */
func2(mike.y);     /* passes integer value of y */
func3(mike.z);     /* passes float value of z */
```

```
func4(mike.s);     /* passes address of string s */
func(mike.s[ 2] ); /* passes character value of s[ 2] */
```

然而，如果希望传递结构中个别成员的地址，可以在结构名前加运算符&。例如，要把结构mike的成员地址传给函数，可以这样写：

```
func(&mike.x);     /* passes address of character x */
func2(&mike.y);    /* passes address of integer y */
func3(&mike.z);    /* passes address of float z */
func4(mike.s);     /* passes address of string s */
func(&mike.s[ 2] ); /* passes address of character s[ 2] */
```

注意，运算符&应放在结构名之前，而不是成员名之前。同时，还应注意字符串s已经指明了一个地址，所以不再需要&了。

7.3.2 向函数传递完整的结构

当使用一个结构作为函数的变元时，整个结构就使用标准的按值调用方式传递。当然，这意味着对于传给函数的结构，函数内部的任何改变都不会影响作为变元的那个结构。

当将结构作为参数使用时，最需注意的是，变元的类型必须与参数类型相匹配。例如，在下面的程序中，变元arg和参数parm都被声明为同一结构类型：

```
#include <stdio.h>

/* Define a structure type. */
struct struct_type {
  int a, b;
  char ch;
} ;

void f1(struct struct_type parm);

int main(void)
{
  struct struct_type arg;

  arg.a = 1000;

  f1(arg);

  return 0;
}

void f1(struct struct_type parm)
{
  printf("%d", parm.a);
}
```

正如这个程序所示的，在声明结构参数时，必须将结构类型声明为全局的，以便在程序的任何地方都可使用。例如，在main( )中声明的struct_type，在f1( )中就不能用。

如上所述，在传递结构时，变元与参数的类型必须匹配。物理上相似是不够的，它们的类型名必须匹配。例如，将上面的程序改写如下就不正确，而且不能编译，因为调用f1( )的变元和参数的类型名各不相同。

```
/* This program is incorrect and will not compile. */
#include <stdio.h>

/* Define a structure type. */
struct struct_type {
  int a, b;
  char ch;
} ;

/* Define a structure similar to struct_type,
   but with a different name. */
struct struct_type2 {
  int a, b;
  char ch;
} ;

void f1(struct struct_type2 parm);

int main(void)
{
  struct struct_type arg;

  arg.a = 1000;

  f1(arg); /* type mismatch */

  return 0;
}

void f1(struct struct_type2 parm)
{
  printf("%d", parm.a);
}
```

7.4 结构指针

与允许指向其他变量类型的指针一样，C/C++也允许指向结构的指针。下面介绍结构指针的一些特点。

7.4.1 声明结构指针

与声明其他指针一样，声明结构指针时要将*置于结构变量名之前。例如，假定有一个前面定义过的结构addr，下面将addr-pointer声明为该结构的指针：

```
struct addr *addr_pointer;
```

记住，在C++中，不必在上面的声明前加关键字struct。

7.4.2 使用结构指针

结构指针有两种主要用途：一是使用按引用调用把结构传递给函数，二是创建依赖于动态分配的链表及其他动态数据结构。本章只讨论第一种。

将完整的结构（最简单的除外）传递给函数有一个主要缺点，就是当执行函数调用时，所有结构成员进出堆栈的开销太大（记住，变量通过堆栈传递给函数）。对于带有几个成员的简

单结构，开销并不很重要，但如果结构包含很多成员，或者某些成员是数组，则运行时的性能就会大大下降。解决这一问题的方法就是只传递结构指针。

当把一个结构指针传递给函数时，只有结构地址进栈。这意味着调用函数的速度很快。另一个好处是，在某种情况下，函数引用被用做变元的实际的结构而不是其副本，通过传递指针，函数可以修改调用中结构的内容。

要获得结构变量的地址，应在结构名前面加上 &，例如，给定下列程序段：

```
struct bal {
  float balance;
  char name[ 80] ;
} person;

struct bal *p; /* declare a structure pointer */
```

则

```
p = &person;
```

将结构 person 的地址放到指针 p 中。

为了用指向结构的指针访问该结构的成员，必须使用运算符 ->。例如，下面的语句引用域 balance：

```
p->balance
```

->被称为箭头运算符，它是由一个减号后面跟一个大于号构成的。给定一个结构指针，要访问结构成员时，用箭头代替圆点。

阅读下面这个简单程序，看看如何使用结构指针。它使用软件计时程序在屏幕上打印时、分、秒：

```
/* Display a software timer. */
#include <stdio.h>

#define DELAY 128000

struct my_time {
  int hours;
  int minutes;
  int seconds;
} ;

void display(struct my_time *t);
void update(struct my_time *t);
void delay(void);

int main(void)
{
  struct my_time systime;

  systime.hours = 0;
  systime.minutes = 0;
  systime.seconds = 0;

  for(;;) {
    update(&systime);
```

```
    display(&systime);
  }

  return 0;
}

void update(struct my_time *t)
{
  t->seconds++;
  if(t->seconds==60) {
    t->seconds = 0;
    t->minutes++;
  }

  if(t->minutes--60) {
    t->minutes = 0;
    t->hours++;
  }

  if(t->hours==24) t->hours = 0;
  delay();
}

void display(struct my_time *t)
{
  printf("%02d:", t->hours);
  printf("%02d:", t->minutes);
  printf("%02d\n", t->seconds);
}

void delay(void)
{
  long int t;

  /* change this as needed */
  for(t=1; t<DELAY; ++t) ;
}
```

通过改变DELAY的定义可调用计时的长短。

可以看出，定义了一个全局结构my_time，但是没有声明变量。在main( )内部，声明了一个结构变量systime并将之初始化为00:00:00。这意味着systime只为函数main( )所知。

变量systime的地址被传给了修改时间的函数update( )及显示时间的函数display( )，两个函数中的变元都声明为指向结构my_time的指针。

在update( )和display( )中，systime的每个成员都通过指针访问。因为update( )接受指向systime结构的指针，它可以修改其值。例如，到24:00:00时，要将小时设置为0，update( )包括下面一行代码

```
if(t->hours==24) t->hours = 0;
```

这行代码告诉编译器使用t的地址（main( )函数中变量systime的地址），并将hours复位为0。

注意，当对结构本身操作时，用点运算符访问结构成员；当用一个结构指针时，用箭头运算符。

7.5 结构中的数组和结构

结构成员可以是简单类型，也可以是聚合类型。一个简单成员可以是任何内部数据类型，如整型或字符型。我们前面遇到过的一个聚合类型成员是addr中的字符数组。其他聚合数据类型包括其他数据类型和结构的一维或多维数组。

研究了前面的例子后，我们可以按期望处理结构成员——数组。例如，考虑下面的结构：

```
struct x {
  int a[10][10]; /* 10 x 10 array of ints */
  float b;
} y;
```

要引用结构 y 中 a 的下标为 3, 7 的整数，可写为

```
y.a[3][7]
```

当一个结构是另一结构的成员时，称为嵌套结构。例如，在这个例子中，结构address嵌套在 emp 中：

```
struct emp {
  struct addr address; /* nested structure */
  float wage;
} worker;
```

其中，结构 emp 被定义为包含两个成员。第一个成员是类型 addr 的结构，含有雇员的地址。另一个是 wage，存放雇员的工资。下面的代码把 93456 赋给 address 的成员 zip：

```
worker.address.zip = 93456;
```

如你所见，每个结构的成员按从最外到最内的顺序引用。C语言标准规定结构至少可以嵌套 15 层。标准 C++ 建议，结构至少可以嵌套 256 层。

7.6 位域

与大部分其他计算机语言不同，C/C++提供了一个内嵌的特征来访问字节中的位，即位域。位域很有用，因为：

- 如果存储空间受限，可以在一个字节中存储几个布尔变量（真/假）；
- 某些设备传输被编码为一个字节中的位的状态信息；
- 某些加密程序需要访问字节中的位。

虽然这些功能都可用位运算符来实现，位域增加了代码的清晰度（并且可能提高效率）。

C/C++ 使用基于结构的方法来访问位。事实上，位域是结构成员的特殊类型，它以位为单位定义域的长度。位域定义的一般形式如下：

```
struct struct-type-name {
 type name1 : length;
 type name2 : length;
```

```
.
.
type nameN : length;
} variable_list;
```

其中，type是位域的类型，length是域中位的数目。位域必须声明为整型或枚举型。长度为1的位域应该声明为unsigned，因为单个位不能有符号。

位域通常用于分析来自硬件设备的输入，例如，串行通信适配器的状态端口返回如下的状态字节：

位	置位时的含义
0	清除-发送信号改变
1	“数据设置就绪”改变
2	“结束位”被检测到
3	接收信号改变
4	清除信号改变
5	数据设置就绪
6	响铃
7	接收信号

可以用下面的位域表示一个状态字节中的信息：

```
struct status_type {
  unsigned delta_cts: 1;
  unsigned delta_dsr: 1;
  unsigned tr_edge:   1;
  unsigned delta_rec: 1;
  unsigned cts:       1;
  unsigned dsr:       1;
  unsigned ring:      1;
  unsigned rec_line:  1;
} status;
```

我们可以用类似于下面显示的程序来激活一个程序，以确定何时发送或接收数据：

```
status = get_port_status();
if(status.cts) printf("clear to send");
if(status.dsr) printf("data ready");
```

给位域赋值与给其他任何类型的结构或员赋值一样，例如，下面的语句清除ring域：

```
status.ring = 0;
```

由此可见，可用点运算符访问每个位域。然而，如果结构是通过指针引用的，则应使用箭头运算符->。

并非必须为每个位域命名。这样就能方便地使用希望的位，跳过无用位。例如，如果我们只关心cts和dsr位，可以像下面这样声明结构status_type：

```
struct status_type {
  unsigned :    4;
  unsigned cts: 1;
```

```
  unsigned dsr: 1;
} status;
```

注意，如果不用，dsr后面不用的各个位就不必声明了。

把位域和其他类型的结构成员混合使用是合法的。例如：

```
struct emp {
  struct addr address;
  float pay;
  unsigned lay_off:    1; /* lay off or active */
  unsigned hourly:     1; /* hourly pay or wage */
  unsigned deductions: 3; /* IRS deductions */
};
```

定义了一个雇员记录，这个雇员记录使用一个字节来存放三条信息：雇员的状态、雇员是否有薪金以及扣除的数量。如果不用位域，这些信息需要3个字节。

位域变量也有某些限制：不能使用位域变量的地址；位域变量不能构成数组；它们不能被声明为静态的。不知道各个机器的域到底是从右到左还是从左到右，因为任何使用位域的代码对机器都有一定的依赖性。各种特定的实现可能施加其他的限制。

7.7 联合

联合是内存位置，可由两个或多个不同类型的变量共享。联合提供了一种把同一位模式翻译成两个或更多模式的方法。声明一个联合与声明一个结构类似。它的一般形式如下：

```
union union-type-name {
 type member-name;
 type member-name;
 type member-name;
  .
  .
  .
} union-variables;
```

例如：

```
union u_type {
  int i;
  char ch;
};
```

这个声明没有创建任何变量。可以通过在声明后面加上变量名来声明变量，也可以用单独的声明语句声明变量。在C中，要声明一个以前定义过的类型为u_type的联合变量cnvt，可写为：

```
union u_type cnvt;
```

在C++中声明联合变量时，只需要使用类型名，不需要在它前面加上关键字union。例如，在C++中是这样声明cnvt的：

```
u_type cnvt;
```

在C++中，可以在这个声明前加上关键字union，但是它是冗余的。在C++中，一个联合的名称定义了一个完整的类型名。在C中，联合名是它的标记，前面必须有关键字union(这类似于前面描述的结构)。然而，因为本章中的程序对C和C++都是合法的，所以将使用C风格的声明形式。

在cnvt中，整数i和字符ch共享同一个内存地址。当然，i占2个字节，而ch只占1个字节。图7.2展示了i与ch共享同一内存地址。在程序中无论何处，都可以作为一个整数或一个字符来引用存储在cnvt中的数据。

当声明一个联合变量时，编译器自动分配足以保存联合中最大变量类型的内存。例如(假定整数占两个字节)，cnvt为2个字节，因此可以包含i，即使ch只要求1个字节。

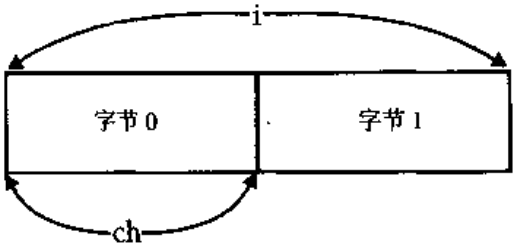

图7.2　i与ch使用联合cnvt(假定整数占2个字节)

要访问联合中的成员，使用的语法与用于结构的语法相同。如果直接对联合操作，用点操作符；如果通过指针访问联合变量，则使用箭头运算符。例如，要将整数10赋给cnvt的成员i，可写为：

```
cnvt.i = 10;
```

在下例中，将指向cnvt的指针传递给函数：

```
void func1(union u_type *un)
{
  un->i = 10; /* assign 10 to cnvt through
                 a pointer */
}
```

联合常用于类型转换，这是因为可以用不同的方法引用包含在联合中的数据。例如，可以用联合处理组成double的字节，用以改变它的精度或执行一些不常见的“舍入”运算。

在需要进行非标准类型转换时，要知道联合的用处，应考虑将短整数写入磁盘文件的问题。C/C++标准库未专门定义将短整数写入磁盘文件的函数。当用fwrite()将任意类型的数据(包括整数)写入一个文件时，使用fwrite()对这样一个简单操作来说开销太大了。使用联合可以很容易地创建一个函数putw()，它把短整数的二进制表示一次一字节地写入磁盘文件(这个例子假定短整数是2字节长)。要明白怎么回事，首先创建一个由一个短整数和一个2字节的字符数组组成的联合：

```
union pw {
  short int i;
  char ch[2];
};
```

现在，可以用pw创建如下的函数putw()了。

```
#include <stdio.h>

union pw {
  short int i;
  char ch[2];
};

int putw(short int num, FILE *fp);

int main(void)
{
  FILE *fp;

  fp = fopen("test.tmp", "wb+");

  putw(1000, fp); /* write the value 1000 as an integer */
  fclose(fp);

  return 0;
}

int putw(short int num, FILE *fp)
{
  union pw word;

  word.i = num;

  putc(word.ch[0], fp); /* write first half */
  return putc(word.ch[1], fp); /* write second half */
}
```

虽然putw( )用了一个短整数来调用，但仍然可以使用标准函数putc( )一次一个字节地将整数中的每个字节写入磁盘文件。

注意：C++支持一个特殊联合类型：匿名联合（anonymous union），将在本书的第二部分中讨论。

7.8 枚举

枚举是一个有名字的整型常量的集合，它指出了这种类型的变量具有的所有合法值。在日常生活中枚举亦是常见的，例如，美国硬币的枚举有：

penny, nickel, dime, quarter, half-dollar, dollar

枚举的定义与结构定义相似，关键字enum表示枚举类型的开始。一般形式为：

enum *enum-type-name* { *enumeration list* } *variable_list;*

其中，类型名和变量列表均是可选的（但至少必须有一个）。下面的程序定义了一个称为coin的枚举：

```
enum coin { penny, nickel, dime, quarter,
            half_dollar, dollar};
```

可以使用枚举类型名来声明枚举类型的变量。在C中，下面的程序声明money是coin类型的一个变量。

```
enum coin money;
```

在C++中，可以使用下面这种简便形式声明变量money：

```
coin money;
```

在C++中，枚举名指定了类型。在C中，枚举名是它的标记，它要求关键字enum来补充它（这类似于前面描述的结构和联合的情况）。

给出这些声明后，下列语句是完全有效的：

```
money = dime;
if(money==quarter) printf("Money is a quarter.\n");
```

枚举的关键在于每个符号都代表一个整数值，它们可以用在任何可使用整数的地方。每个符号都给定了一个比前一个大1的值。第一个枚举符号的值是0，所以

```
printf("%d %d", penny, dime);
```

在屏幕上显示0 2。

通过初始化可以给一个或多个符号指定值。这可用符号名后面带一个等号和一个整数值来实现。一旦使用了一个初始值，出现在它后面的符号被赋予比这个初始值更大的值。例如，下面的代码将100赋给quarter：

```
enum coin { penny, nickel, dime, quarter=100,
            half_dollar, dollar};
```

现在这些符号的数值为：

penny	0
nickel	1
dime	2
quarter	100
half_dollar	101
dollar	102

有关枚举，存在一个普遍但是错误的假设：符号可以直接地输入输出。实际情况并非如此。例如，下面的代码不能按要求执行：

```
/* this will not work */
money = dollar;
printf("%s", money);
```

切记，符号dollar只是一个整数的名字，而不是一个字符串。同理，要用下面的代码获得希望的结果也是不可能的：

```
/* this code is wrong */
strcpy(money, "dime");
```

也就是说，一个包含符号名的字符串并不能自动转化成那个符号。

实际上，创建用于输入、输出枚举符号的代码是十分乏味的（除非愿意设定它们的整数值）。例如，下面的代码用于显示money中包含的硬币种类：

```
switch(money) {
```

```
    case penny: printf("penny");
      break;
    case nickel: printf("nickel");
      break;
    case dime: printf("dime");
      break;
    case quarter: printf("quarter");
      break;
    case half_dollar: printf("half_dollar");
      break;
    case dollar: printf("dollar");
  }
```

有时，可以声明一个字符串数组，用枚举值作为下标将枚举值转换成相应的字符串。例如，下面的代码也可以输出相应的串：

```
char name[ ][ 12] ={
  "penny",
  "nickel",
  "dime",
  "quarter",
  "half_dollar",
  "dollar"
};
printf("%s", name[ money] );
```

当然，这种方法只有在没有符号被初始化时才是可行的，因为串数组的起始值必须为0，按升序每次增1。

因为枚举值必须手工转换才能成为可读的控制台I/O的字符串值，它们多用于不必做这样转换的程序中。例如，我们常常使用枚举定义编译器的符号表。通过提供编译时冗余检查来保证变量赋予有效值，枚举常被用来验证一个程序的有效性。

7.9 用sizeof来保证可移植性

你已经见到，可以使用结构、联合及枚举创建变长变量，这些变量的实际长度因机器而异。运算符sizeof计算任何变量或类型的长度，并有助于从程序中消除依赖机器的代码。这个运算符在涉及结构或联合时非常有用。

下面的讨论假定了一个对许多C/C++编译器来说很常见的实现，其数据类型的长度如下所示：

类型	长度（字节）
char	1
int	4
double	8

下面的代码在屏幕上打印1，4和8。

```
char ch;
int i;
```

```
double f;

printf("%d", sizeof(ch));
printf("%d", sizeof(i));
printf("%d", sizeof(f));
```

结构的长度与它的成员的长度总和一样大，或比它们更大，例如：

```
struct s {
  char ch;
  int i;
  double f;
} s_var;
```

这里，sizeof(s_var)至少为13（8+4+1）。s_var的大小可能比13大，因为编译器允许展宽一个结构，以达到字或段对齐（一段为16字节）。因为结构的大小可能比它所有成员的总和大，所以要知道一个结构的大小时必须用sizeof。

因为sizeof是一个编译时运算符，所以计算任何变量长度所必需的所有信息在编译时是已知的，这对联合有特殊的意义，因为联合的长度总是等于最大成员的长度。例如，考虑下面的代码段：

```
union u {
  char ch;
  int i;
  double f;
} u_var;
```

这里，sizeof(u_var)为8。在运行时，u_var实际取何值并无关系，用户关心的是它包含的最大成员的长度，因为联合必须等于其最大成员的长度。

7.10 typedef

可以使用关键字typedef定义新的数据类型名。你实际上并没有创建新的数据类型，而是为已存在的类型定义一个新的名称。这个过程有助于改善那些依赖于机器的程序的移植性。如果要为程序所用的每个依赖机器的数据类型定义自己的类型名，当改变到新环境时，仅仅需要改变typedef语句。通过给标准数据类型以描述性的名字，typedef也可以帮助自文档化代码。typedef语句的一般形式为：

typedef *type newname;*

其中，type是任何合法的数据类型，newname是该类型的新名称。你定义的新名称只是新添到现存类型的名称中的，并没有替换现存类型名。

例如，可以使用下面的代码为float创建一个新名称：

```
typedef float balance;
```

该语句告诉编译器识别balance为float的别名，随后就可以用balance创建一个float变量：

```
balance over_due;
```

其中，over_due为类型balance的浮点变量，balance是float的另一个名字。

既然balance已经定义了，它就可以用在另一个typedef中。例如，

```
typedef balance overdraft;
```

告诉编译器识别overdraft为balance的另一个名字，亦即float的另一个名字。

使用typedef使程序更加容易阅读，更容易移植到新机器上，但必须牢记：实际上并未建立任何新的数据类型。