# 第2章 表 达 式

本章讨论C（以及C++）语言的最基本的元素：表达式。读者将会看到，C/C++中的表达式本质上比其他的计算机语言更通用、更强大。表达式是由这些原子元素形成的：数据和运算符。数据可以用变量或常量表示。像大多数其他计算机语言一样，C/C++支持几种不同类型的数据，它也提供了多种运算符。

## 2.1 五种基本数据类型

在C子集中，有五种基本数据类型：字符、整型、浮点、双精度和无值类型（分别是char, int, float, double 和 void）。读者将会看到，C语言中所有其他数据类型都是以这些类型之一为基础的。这些数据类型的长度和范围因处理器和编译器的变化而变化，然而，在任何情况下一个字符都占一个字节。一个整数的长度通常与程序执行环境中字的长度一样，对于大多数16位的环境，例如DOS或Window 3.1，一个整数是16位。对于大多数32位的环境，例如Windows 2000, 一个整数是32位。然而，如果想要程序能够移植到最大范围的环境中，不能对整数的长度做出限定。重要的是要理解C和C++都仅规定了每种数据类型的最小范围，而不是它的字节长度。

**注意：** C语言定义了五种基本的数据类型，C++又增加了两种：bool和wchar_t。这些将在本书的第二部分介绍。

浮点值的准确格式取决于它们是如何实现的。整型通常对应于宿主机上一个字的长度。char类型的值通常被用于容纳由ASCII字符集定义的值，对在那个范围以外的值，不同编译器的处理也不相同。

float和double类型的取值范围取决于用来表示浮点数的方法。然而，无论是用什么方法，范围都相当大。标准C规定一个浮点值的最小范围是从1E-37到1E+37。每种浮点值精度的最小位数示于表2.1中。

**注意：** 标准C++没有规定基本类型的最小长度或范围，相反，它只说明它们必须满足一定的要求。例如，标准C++规定一个int将“有由执行环境的体系结构建议的长度”。在所有的情况下，这都将满足或超过由标准C规定的最小范围。每个C++编译器都在头文件<climits>中规定了基本类型的长度和范围。

类型void或者明确地声明一个函数不返回值，或者创建通用的指针。我们将在随后的章节中讨论这两种用途。

## 2.2 修饰基本类型

除了void类型外，基本数据类型的前面可以有各种修饰符。可以使用修饰符来改变基本类型的意义，以准确地适应各种情况的需要。修饰符有下面所列的几种：

signed（有符号）
unsigned（无符号）
long（长整型）
short（短整型）

表2.1 ANSI/ISO C标准定义的所有数据类型

| 类型 | 典型的长度（以位计） | 最小范围 |
|---|---|---|
| char | 8 | -127～127 |
| unsigned char | 8 | 0～255 |
| signed char | 8 | -127～127 |
| int | 16或32 | -32767～32767 |
| unsigned int | 16或32 | 0～65535 |
| signed int | 16或32 | 与int相同 |
| short int | 16 | -32767～32767 |
| unsigned short int | 16 | 0～65535 |
| signed short int | 16 | 与short int相同 |
| long int | 32 | -2147483647～2147483647 |
| signed long int | 32 | 与long int相同 |
| unsigned long int | 32 | 0～4294967295 |
| float | 32 | 6位精度 |
| double | 64 | 10位精度 |
| long double | 80 | 10位精度 |

可以对整型应用修饰符signed, short, long和unsigned。可以对字符型应用unsigned和signed。也可以对double类型应用long修饰符。表2.1显示了所有有效的数据类型组合，以及它们的最小范围和近似的位宽度（这些值也适用于典型的C++实现）。记住，这个表显示了这些类型会有的最小范围，就像标准C/C++规定的那样，而不是它们通常的范围。例如，在使用2的补码运算的计算机上，所有的整数都会有至少32767到-32768的范围。

可以对整型使用signed，但这是冗余的，因为默认的整数定义就假定了一个有符号数。signed的最重要的用处是在char被默认为无符号数的应用中修饰char。

有符号整数和无符号整数的区别是对整数高阶位的解释。如果指定一个有符号整数，编译器生成代码时将整数的高阶位用做标志位。如果标志是0，则数值为正；如果为1，则数值为负。

一般来讲，负数采用2的补码的形式表示，即对数中所有位取反（除了标志位），给这个数加1，并设置标志为1。

有符号整数对许多算法都很重要，但是它们的绝对值只有其无符号值的一半。例如，下面是32767：

0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

如果高阶位设置为1，这个数将被翻译为-1。然而，如果把这个数声明为无符号整数，当高阶位设置为1时，这个数为65535。

当只使用一个类型修饰符时（即，当它不被放在基本类型前时），我们就假定为int。因此，下面的类型修饰符是等价的：

| 修饰符 | 等价形式 |
| --- | --- |
| signed | signed int |
| unsigned | unsigned int |
| long | long int |
| short | short int |

尽管int是隐含表示的，许多程序员都会指定int。

## 2.3 标识符名称

在C/C++中，变量、函数、标签和各种其他用户定义对象的名称都被称为标识符。这些标识符的长度可以是一个字符，也可以是几个字符。第一个字符必须是一个字母或下划线，其后的字符必须是字母、数字或下划线。下面是一些正确和不正确的标识符名称：

| 正确的 | 不正确的 |
| --- | --- |
| Count | 1count |
| test23 | hi!there |
| high_balance | high...balance |

在C语言中，标识符可以是任意长度的。然而，不是所有的字符都必须是有效的。如果在外部链接过程中使用标识符，那么至少前6个字符是有效的。这些标识符，称为外部名，包括函数名和在文件之间共享的全局变量。如果标识符不用在外部链接过程中，那么至少前31个字符是有效的。这种类型的标识符称为内部名，它包括局部变量的名称。在C++中，对标识符的长度没有限制，并且至少前1024个字符是有效的。如果把C程序转换到C++程序，这种区别可能很重要。

在一个标识符中，对大写和小写字母的处理是不同的。因此，count, Count和COUNT是三个不同的标识符。

标识符不能和C或C++关键字相同，也不能和C或C++库中的函数同名。

## 2.4 变量

读者可能知道，变量是内存中一个命名的位置，用于存储一个可被程序修改的值。所有的变量必须在使用前声明。声明的一般形式如下：

```
type variable_list;
```

其中，type必须是带有任意修饰符的有效数据类型，variable_list可由一个或更多的用逗号分开的标识符名组成。下面是一些声明的例子：

```
int i,j,l;
short int si;
unsigned int ui;
double balance, profit, loss;
```

记住，在C/C++中，变量名和它的类型无关。

### 2.4.1 在哪里声明变量

变量可以在三个地方声明：在函数内部、在函数参数的定义中、在所有函数的外部。这些变量分别是局部变量、形式参数和全局变量。

### 2.4.2 局部变量

在函数内部声明的变量称为局部变量。在一些C/C++文献中，这些变量也称为自动变量。本书使用更常用的术语“局部变量”。局部变量仅可被声明它的块的内部语句所引用，换句话说，局部变量在其块外是不可知的。记住，代码块以左花括号开始，以右花括号结束。

局部变量仅存在于它们声明时所在的代码块执行时，即，局部变量在进入代码块时生成，在退出代码块时销毁。

在局部变量声明时所在的代码块中，最常见的当属函数。例如，考虑下面的两个函数：

```
void func1(void)
{
  int x;

  x = 10;
}

void func2(void)
{
  int x;

  x = -199;
}
```

整型变量x被声明了两次，一次在func1()中，一次在func2()中。在func1()中的x和func2()中的x没有任何关系。这是因为每个x仅为声明它的块中的代码所知。

C语言包含关键字auto，可以使用它来声明局部变量。然而，因为默认时所有的非全局变量都被假定是auto，这个关键字实际上从未使用过。因此，本书中的例子将不使用它（据说auto包含于C中是为了提供与它的前任B语言的源代码级兼容性。还有，在C++中也支持auto，目的是提供和C语言的兼容性）。

出于方便和习惯，大多数程序员在函数的花括号后、其他语句前立即声明函数所用的所有变量。然而，可以在任何语句块内声明局部变量。由函数定义的块仅是一个特例，例如，

```
void f(void)
{
  int t;

  scanf("%d%*c", &t);

  if(t==1) {
    char s[80]; /* this is created only upon
                   entry into this block */
    printf("Enter name:");
    gets(s);
    /* do something ... */
  }
}
```

其中，局部变量s就是在if代码块的入口处创建的，并在其出口处销毁。此外，s仅在if块内可知并且不能在别的地方引用——甚至在包含它的函数的其他部分。

在使用变量的代码块内声明它们可以帮助防止不想要的副作用。因为变量在其声明的块外是不存在的，所以不能意外地改变它。

关于在哪里声明局部变量，在C(由C89定义的)和C++语言之间有一个很重要的区别。在C语言中，必须在块的开始处、先于任何“动作”语句前声明所有的局部变量。例如，在C89中，下面的函数是错误的。

```
/* For C89, this function is in error,
   but it is perfectly acceptable for C++.
*/
void f(void)
{
  int i;

  i = 10;

  int j; /* this line will cause an error */
  j = 20;
}
```

然而，在C++中，这个函数是有效的，因为可以在一个块内的任何地方声明一个局部变量，只要是在使用它们之前(关于C++变量声明的问题将在本书第二部分详细讨论)。有趣的是，C99允许你在一个块内的任何地方定义变量。

因为局部变量是在声明它们的块中入口处创建、出口处销毁的，一旦出了块，它们的内容就丢失了。当调用函数时，要特别记住这一点。当函数被调用时，它的局部变量被创建，一旦函数返回，它们就被销毁。这意味着局部变量不能在两次调用间保留它们的值(然而，通过使用static修饰符，可以指导编译器保留它们的值)。

除非被指明，否则局部变量存储在堆栈上。堆栈是一个动态可变存储区的事实解释了局部变量通常不能在函数调用间保留它们的值的原因。

可以把一个局部变量初始化为某些已知的值，这个值在每次进入声明变量的代码块时赋给该变量。例如，下面的程序十次打印了数值10。

```
#include <stdio.h>

void f(void);

int main(void)
{
  int i;

  for(i=0; i<10; i++) f();

  return 0;
}

void f(void)
{
  int j = 10;

  printf("%d ", j);
```

```
    j++; /* this line has no lasting effect */
}
```

### 2.4.3 形式参数

如果一个函数要使用变元，它必须声明将接受变元值的变量。这些变量称为函数的形式参数（formal parameters），它们的行为与函数内任何其他局部变量一样。正如下面的程序片段所示的，它们的声明出现在函数名后、括号内：

```
/* Return 1 if c is part of string s; 0 otherwise */
int is_in(char *s, char c)
{
  while(*s)
    if(*s==c) return 1;
    else s++;

  return 0;
}
```

函数is_in( )有两个参数：s和c。如果在c中指定的字符包含于字符串s中，这个函数返回1；否则，返回0。

必须通过声明它们来指明形式参数的类型，就像刚才所示的那样。然后，可以在函数内像常规的局部变量那样使用它们。记住，像局部变量一样，它们也是动态的，在函数的出口处被销毁。

类似于局部变量，可以对一个函数的形式参数赋值或在任何合法的表达式中使用它们。即使这些变量接受传给函数的变元的值，也可以像任何其他局部变量那样使用它们。

### 2.4.4 全局变量

不像局部变量，全局变量（global variables）在整个程序内都是可知的，可以被任何代码段所使用。此外，它们在程序执行时保持它们的值。可以通过在任何函数的外面声明全局变量来创建它们。任何表达式都可以访问它们，不管表达式是位于什么代码块中。

在下面的程序中，变量count是在所有函数的外面声明的。尽管它的声明出现在main( )函数之前，在第一次使用前，可以把它放在任何地方，只要不是在函数中。然面，通常最好在程序的顶部声明全局变量。

```
#include <stdio.h>
int count; /* count is global */

void func1(void);
void func2(void);

int main(void)
{
  count = 100;
  func1();

  return 0;
}

void func1(void)
```

```
{
  int temp;

  temp = count;
  func2();
  printf("count is %d", count); /* will print 100 */
}

void func2(void)
{
  int count;

  for(count=1; count<10; count++)
    putchar('.');
}
```

仔细研究这个程序后，会发现尽管main( )和func1( )都没有声明变量count，两者都可以使用它。然而，func2( )声明了一个称为count的局部变量。当func2( )引用count时，它指的是局部变量，而不是全局变量。如果全局变量和局部变量同名，在局部变量声明所在的代码块内对该变量名的引用都是指那个局部变量，对全局变量没有任何影响。这可能是很方便的，但是如果忘记了它，程序即使看起来是正确的，也可能导致运行时的奇怪行为。

全局变量的存储是在一个固定的存储区中，这个固定的存储区是由编译器为此而设定的。当程序中的许多函数使用同一数据时，全局变量会很有帮助。然而，应该避免使用不必要的全局变量。它们在程序运行的整个时间占据存储空间，而不是在需要它们时才占据。此外，在应使用局部变量的地方使用一个全局变量会降低函数的通用性，因为它依赖某些必须在其本身外定义的东西。最后，使用大量的全局变量会使程序因为不可知或不期望的副作用而出错。在开发大型程序时一个主要的问题就是，因为在程序中别的地方使用了它而偶然地改变了变量的值。在C/C++语言中，如果在程序中使用了太多的全局变量，这种情况就可能发生。

## 2.5 const和volatile限定符

有两种限定符控制了对变量的访问或修改方法：const和volatile。它们必须出现在它们限定的类型修饰符和类型名的前面。这些限定符一般称为cv限定符。

### 2.5.1 const

const类型的变量不可以被程序改变（然而，可以给const变量赋予一个初值）。编译器可以把这种类型的变量放到只读存储器(ROM)中。例如，

```
const int a=10;
```

创建了一个整型变量a，其初值为10，程序不可以修改它。然而，可以在其他类型的表达式中使用这个变量a。const可以通过一个明确的初始化或通过一些依赖于硬件的方法接受它的值。

可以使用const限定符来保护由一个函数的变元所指向的对象，使其不能被那个函数所修改，即，当一个指针传给一个函数时，那个函数可以修改由指针指向的实际变量。然而，如果在参数声明中这个指针被规定为const，那个函数代码就不能修改它指向的东西。例如，下面程序中的sp_to_dash( )函数对它的字符串变元中的每个空格都打印了连字符，即，字符串“this is a test”将被打印为“this-is-a-test”。在参数声明中使用const保证了在函数内的代码不能修改参

数指向的对象。

```
#include <stdio.h>

void sp_to_dash(const char *str);

int main(void)
{
  sp_to_dash("this is a test");

  return 0;
}

void sp_to_dash(const char *str)
{
  while(*str) {
    if(*str== ' ') printf("%c", '-');
    else printf("%c", *str);
    str++;
  }
}
```

如果编写 sp_to_dash( )，使得字符串可以被修改，它将不能编译。例如，如果像下面这样编码 sp_to_dash( )，会接收到一个编译时错误：

```
/* This is wrong. */
void sp_to_dash(const char *str)
{
  while(*str) {
    if(*str==' ') *str = '-'; /* can't do this; str is const */
    printf("%c", *str);
    str++;
  }
}
```

在标准库中的许多函数在其参数声明中都使用const。例如，strlen( ) 函数具有这样的原型：

size_t strlen(const char *str);

它规定 str 为 const 将保证 strlen( )不会修改由 str 指向的字符串。一般来讲，当一个标准库函数不需要修改由一个调用参数指向的对象时，它被声明为 const。

也可以使用 const 来验证程序没有修改一个变量。记住，const 类型的变量可以被程序外的某些东西修改。例如，一个硬件设备可以设置它的值。然而，通过把一个变量声明为 const，可以证实因为外部事件，对那个变量的任何改变都会出现。

### 2.5.2 volatile

修饰符volatile告诉编译器变量值可以以任何不被程序明确指明的方式改变。例如，一个全局变量的地址可以传给操作系统的时钟例程并用来保存系统的真实时间。这种情况下，变量的内容改变而无需程序中的任何明确的赋值语句。这是很重要的，因为大多数C/C++编译器通过假定变量的内容不变（如果它不出现在赋值语句的左边）而自动地优化某些表达式。因此，每次引用时，可能不需要重新检验它。还有，在编译过程中，某些编译器会改变表达式的求值顺序。volatile 修饰符可以防止这些改变。

可以同时使用const和volatile。例如，如果假定0 × 30是仅由外部条件改变的端口值，那么下面的声明将会防止出现任何偶发的副作用：

```
const volatile char *port = (const volatile char *) 0x30;
```

## 2.6 存储类限定符

C语言支持四种存储类限定符：

extern
static
register
auto

这些限定符通知编译器如何存储随后的变量。使用这些限定符的声明的一般形式如下所示：

*storage_specifier type var_name;*

注意存储限定符位于变量声明的其余部分之前。

**注意：** C++中增加了另一个存储类限定符mutable，我们将在第二部分描述它。

### 2.6.1 extern

在讨论extern以前，简单描述一下C/C++链接。C和C++定义了三种链接：外部的、内部的和无链接。通常，函数和全局变量有外部链接，这意味着它们对组成程序的所有文件都是可用的。被声明为静态的全局对象（在下一节描述）有内部链接，这些仅在它们声明时所在的文件内是可知的。局部变量没有链接，因此仅在它们自己的块内是可知的。

extern的主要用途是指定一个对象被声明为在程序中别的地方有外部链接。要理解为什么这很重要，必须理解一个声明和一个定义之间的区别。一个声明说明了对象的名称和类型。一个定义会导致给对象分配存储空间。同一个对象可以有许多声明，但仅有一个定义。

大多数情况下，变量声明也是定义。然而，通过在变量名前加上extern限定符，可以声明一个变量而没有定义它。因此，当需要引用一个变量，而这个变量是在程序的其他部分定义的时，可以使用extern声明那个变量。

下面是一个使用extern的例子。注意全局变量first和last是在main( )之后声明的。

```
#include <stdio.h>

int main(void)
{
  extern int first, last; /* use global vars */

  printf("%d %d", first, last);

  return 0;
}

/* global definition of first and last */
int first = 10, last = 20;
```

这个程序输出 10 20，因为 printf( )语句使用的全局变量 first 和 last 被初始化为这些值。因为在 main( )中的 extern 声明告诉编译器 first 和 last 是在别处声明的（在这个例子中，是在后面的同一文件中），程序可以编译而不出现错误，即使 first 和 last 是在它们的定义之前就使用了。

重要的是要理解：前面程序中所示的 extern 变量声明是必需的，仅仅因为 first 和 last 并没有在它们在 main( )中使用之前声明。一旦它们的声明出现在 main( )之前，那么就不需要 extern 语句了。记住，如果编译器发现了还没有在当前块内声明的变量，编译器就会检查它是否匹配在封闭的块内声明的任何变量。如果不匹配，编译器检查前面声明的全局变量。如果找到了匹配，编译器假定那是正被引用的变量。在需要使用一个变量、而这个变量是在后面的文件中声明的时候，就需要 extern 限定符。

如前所述，extern 允许声明一个变量而不定义它。然而，如果初始化那个变量，那么 extern 声明就变成了一个定义。这是非常重要的，因为对象可以有多个声明，但是只能有一个定义。

extern 有一个重要的用途，是与多文件程序有关的。在 C/C++ 中，一个程序可以散布在两个或更多的文件中，分别进行编译，然后链接在一起。如果是这种情况，一定有某种方法告诉所有的文件此程序要求全局变量。要这样做，最好（并且也最可移植）的方法是在一个文件中声明所有的全局变量并在其他文件中使用 extern 声明，如图 2.1 所示。

在文件 2 中，全局变量列表是从文件 1 中复制来的，并且 extern 限定符添加到了声明中。extern 限定符告诉编译器在它之后的变量类型和名称是在别的地方定义的。换句话说，extern 让编译器知道这些全局变量的类型和名称是什么，而不必再一次为它们分配存储空间。当链接程序把两个模块链接起来时，将解析所有到外部变量的引用。

在现实世界的多文件程序中，extern 声明一般包含在一个头文件中，这个头文件简单地包括了每个源代码文件。比起手工地在每个文件中复制 extern 声明，这既容易，又少有错误。

在 C++ 中，extern 限定符有另外一个用途，我们将在本书第二部分中描述。

**注意**：extern 也可以应用到一个函数声明中，但这样做是冗余的。

**文件 1**

```
int x, y;
char ch;
int main(void)
{
/* ... */
}

void func1(void)
{
x = 123;
}
```

**文件 2**

```
extern int x, y;
extern char ch;
void func22(void)
{
x = y / 10;
}

void func23(void)
{
y = 10;
}
```

图 2.1 在不同的编译模块中使用全局变量

## 2.6.2 静态变量

无论是在它们自己的函数中还是在文件中，静态变量都是永久变量。不像全局变量，它们在函数或文件外是不可知的，但是在两次调用之间它们的值是不变的。当编写通用函数和其他

程序员可能使用的函数库时，这个特征很有用。静态（static）对局部变量和全局变量有不同的影响。

**静态局部变量**

在把static修饰符应用于局部变量时，编译器为它创建永久的存储区，就像它为全局变量创建存储区一样。在静态局部变量和静态全局变量之间的主要区别是：静态局部变量仅对它在其中声明的块是可知的。简言之，静态局部变量是一种局部变量，它的值在两次函数调用之间保持不变。

静态局部变量对于独立函数的创建是非常重要的，因为有些例程必须在调用之间保留其值。如果不允许静态变量，将必须使用全局变量，就可能导致产生副作用。有一个函数从静态局部变量中受益了，它就是数序列生成器。这个生成器基于前面的值生成新值，可以使用全局变量保存这个值。然而，每次在程序中使用这个函数时，必须声明它为全局变量并且确信它不和已经存在的任何其他全局变量相冲突。最好的解决方案是声明一个变量，这个变量保持所生成的数为静态的，就像在下面这个程序中那样：

```
int series(void)
{
  static int series_num;

  series_num = series_num+23;
  return series_num;
}
```

在这个例子中，变量series_num在函数调用之间总是存在的，而不是像一般的局部变量那样生成和销毁。这意味着每次对series( )的调用都会在序列中生成一个基于前一个数的新值，而无须把那个变量声明为全局的。

可以给静态局部变量赋一个初值。这个值仅在程序开头赋予一次，而不是像一般的局部变量那样，在每次进入代码块时都要赋值。例如，下面的series( )初始化series_num为100：

```
int series(void)
{
  static int series_num = 100;

  series_num = series_num+23;
  return series_num;
}
```

如程序所示，这个序列总是以同一个值开始，在这个例子中为123。虽然这可以为某些应用所接受，大多数序列生成器需要让用户指定开始点。要给series_num一个用户指定的值，一种方法是让它成为全局变量，然后让用户设置它的值。然而，不把series_num定义为全局的是使它成为静态的关键的一点。这导致了static的第二种用途。

**静态全局变量**

把限定符static应用于全局变量，就是在告诉编译器创建一个仅在声明它的文件中可知的全局变量。这意味着：即使变量是全局的，在其他文件中的例程可能不知道它或直接改变它的内容，这就避免了副作用。对于局部静态变量不能做这项工作的情况，可以创建一个小文件（这个小文件仅包含需要此全局静态变量的函数），单独编译这个小文件并使用它而不必担心出现

副作用。

为了演示一个全局静态变量，改写前一节中的序列生成器的例子，以便一个种子值通过调用第二个函数 series_start( )初始化这个序列。包含 series( ), series_start( )和 series_num 的完整程序如下所示：

```
/* This must all be in one file - preferably by itself. */

static int series_num;
void series_start(int seed);
int series(void);

int series(void)
{
  series_num = series_num+23;
  return series_num;
}

/* initialize series_num */
void series_start(int seed)
{
  series_num = seed;
}
```

调用 series_start( )，用某个已知整数值初始化序列生成器。在那之后，调用 series( )来生成序列中的下一个元素。

回想一下：局部静态变量的名称仅为它们在其中声明的代码块所知，全局静态变量的名称仅为它们驻留其中的文件所知。如果把 series( )和 series_start( )函数放到库中，可以使用这些函数，但是不能引用变量 series_num，因为对程序中的其他代码来说，变量 series_num 是隐藏的。事实上，甚至可以声明和使用程序中另一个称为 series_num 的变量（当然，在另一个文件中）。本质上，static 修饰符允许变量仅为需要它们的函数所知，而没有不想要的副作用。

静态变量使你可以隐藏程序的一部分而不为其他部分所知。当管理一个非常大并且复杂的程序时，这是一个很大的优点。

**注意**：在 C++ 中，仍然支持前面所述的 static 的用法，但是不推荐使用它。这意味着新代码中不推荐使用它，相反，应该使用名字空间，我们将在第二部分讨论名字空间。

### 2.6.3 寄存器变量

register 存储限定符最初只应用于 int, char 或指针类型的变量。然而，现在拓宽了 register 的定义，以便它适用于任何类型的变量。

最初，register 限定符要求编译器把一个变量的值保存在 CPU 的寄存器而不是内存中（一般的变量存储在内存中）。这意味着对寄存器变量的操作可能比对正常的变量快得多，因为寄存器变量实际上保存在 CPU 中并且不要求内存访问来确定或修改它的值。

今天，我们已大大扩展了 register 的定义，现在它可以应用于任何类型的变量。标准 C 简单地声明“对对象的访问是尽可能快的”（标准 C++ 说明 register 是“一个实现暗示，即如此这般声明的对象将被广泛使用”）。在实际中，字符和整数仍然存储在 CPU 的寄存器中。较大的对象，例如数组很明显不能存储在寄存器中，但是它们仍然可以被编译器优先处理。取决于 C/C++

编译器的不同实现和它的操作系统环境，寄存器变量可以按编译器所适合的方式处理。事实上，在技术上也允许编译器完全忽略register限定符并把用它修饰的变量当做普通变量来处理，但在实际中很少这样做。

只能把register限定符应用于局部变量和函数中的形式参数，不允许全局寄存器变量。下面是一个使用寄存器变量的例子，这个函数计算 $M^e$ 的值。

```
int int_pwr(register int m, register int e)
{
  register int temp;

  temp = 1;

  for(; e; e--) temp = temp * m;
  return temp;
}
```

在这个例子中，e, m 和temp被声明为寄存器变量，因为它们都在循环内使用。寄存器变量在速度方面被优化的事实使得它们很适宜用在循环中。一般来讲，寄存器变量用在它们能最好发挥作用的地方，经常是对同一变量进行多次引用的地方。这是非常重要的，因为虽然可以把任意数目的变量声明为register类型的，但是并非所有的都会得到同样的访问速度方面的优化。

在任何一个代码块内用于速度优化的寄存器变量的数目是由环境和C/C++的具体实现决定的。不用担心声明了太多的寄存器变量，因为当达到限度时，编译器自动把寄存器变量转换为非寄存器变量（这保证了代码在各种处理器间的可移植性）。

通常，至少有两个char或int类型的寄存器变量实际保存在CPU的寄存器中。因为环境差异很大，所以应查看编译器的用户手册来决定是否能应用其他的优化选项。

在C语言中，不能使用&运算符（在本章后面讨论）找到寄存器变量的地址。这很有意义，因为寄存器变量可能存储在CPU的寄存器中，通常是不可寻址的，但是这种限制不适用于C++。然而，在C++中取得寄存器变量的地址可能阻止它被完全优化。

尽管寄存器的描述已在其传统的意义上拓宽了，在实践中，一般来讲，它仍然只对整数和字符类型有重要的影响。因此，对于其他的变量类型，不应该指望在速度上有显著的提高。

## 2.7 变量初始化

在声明变量时，通过在变量名后放一个等号和一个值，我们可以为变量赋值。初始化的一般形式为：

*type variable_name = value;*

下面是一些例子

```
char ch = 'a';
int first = 0;
float balance = 123.23;
```

全局和静态局部变量只能在程序的开始处被初始化，局部变量（除了静态局部变量外）在每次进入声明它们的块时被初始化。没有被初始化的局部变量在对它们进行第一次赋值之前是未知的，未初始化的全局和静态局部变量被自动设置为0。

## 2.8 常量

常量指程序不能改变的固定值，常量可以是基本数据类型中的任意一种。表示每个常量的方式取决于它的类型，常量也称为文字（literals）。

字符常量被封闭在单引号中。例如'a'和'%'都是字符常量。C和C++都定义了宽字符（最常用在非英语语言环境中），它是16位长。要指定一个宽字符常量，在字符前面加上L，例如，

```
wchar_t wc;
wc = L'A';
```

其中，wc被赋予A对应的宽字符常量，宽字符的类型是wchar_t。在C语言中，这种类型在一个头文件中定义并且不是内嵌的类型。在C++中，wchar_t是内嵌的。

整型常量被指定为一个不带分数部分的数字，例如，10和-100是整数常量。浮点常量要求小数点后跟数的小数部分，例如，11.123是一个浮点常量。C/C++也允许使用特定的符号表示浮点数。

浮点类型有两种：float和double。也存在几个基本类型的变种，可以使用类型修饰符来生成。默认时，编译器把一个数值常量放到将保存它的最小可兼容数据类型中。因此，假定16位整数，数值10默认时是int型，但是103 000是long型。即使数值10也可以当做一个字符类型，编译器将不会这样做。最小类型原则的惟一例外是浮点常量，它被假定为double型。

对于要编写的大多数程序，编译器的默认值就足够用了。然而，可以通过使用一个后缀来精确地指定数值常量的类型。对于浮点类型，如果数字后面跟有F，则这个数作为float处理。如果数字后面跟有L，则这个数变为一个long double。对于整数类型，后缀U代表unsigned，而L代表long。下面是一些例子：

| 数据类型 | 常量范例 |
|---|---|
| int | 1 123 21000 -234 |
| long int | 35000L -34L |
| unsigned int | 10000U 987U 40000U |
| float | 123.23F 4.34e-3F |
| double | 123.23 1.0 -0.9876324 |
| long double | 1001.2L |

### 2.8.1 十六进制和八进制常量

有时，使用一个基于8或16而不是10（我们的标准十进制系统）的数字系统会更容易。基于8的数字系统称为八进制系统，使用数字0到7表示。在八进制中，数字10与十进制中的8是一样的。基于16的数字系统称为十六进制系统，使用数字0到9加上字母A到F表示，字母A到F分别代表10, 11, 12, 13, 14和15。例如，十六进制数10是十进制中的16。因为经常使用这两个数字系统，C/C++允许用十六进制或八进制而不是十进制指定整数常量。一个十六进制常量必须由0x后跟十六进制形式的常量组成。下面是一些例子：

```
int hex = 0x80;    /* 128 in decimal */
int oct = 012;     /* 10 in decimal */
```

### 2.8.2 字符串常量

C/C++支持另一种类型的常量：字符串。字符串是括在双引号中的一串字符，例如"this is a test"是一个字符串。在一些范例程序的printf( )语句中，我们已经见过字符串的例子。尽管C允许定义字符串常量，它实际上没有字符串数据类型（然而，C++定义了字符串类）。

不应该把字符串和字符相混淆。一个字符常量括在单引号中，比如'a'。然而，"a"是一个仅包含一个字母的字符串。

### 2.8.3 反斜杠字符常量

单引号中的字符常量大部分都用做打印字符。然而，有一些则不可能从键盘输入，例如回车。因此，C/C++包含了特别的反斜杠字符常量（示于表2.2中），以便可以很容易地把这些特殊字符作为常量输入，这些也称为转义序列。可以使用反斜杠代码而不是ASCII码来确保程序的可移植性。

例如，下面的程序输出一个新行和一个制表符，然后打印字符串This is a test。

```
#include <stdio.h>
int main(void)
{
  printf("\n\tThis is a test.");

  return 0;
}
```

表2.2 反斜杠代码

| 代码 | 意义 |
|---|---|
| \b | 退格 |
| \f | 换页 |
| \n | 换行 |
| \r | 回车 |
| \t | 水平制表位 |
| \" | 双引号 |
| \' | 单引号 |
| \0 | 空值 |
| \\ | 反斜杠 |
| \v | 垂直制表位 |
| \a | 报警 |
| \? | 问号 |
| \N | 八进制常量（其中N是一个八进制常量） |
| \xN | 十六进制常量（其中N是一个十六进制常量） |

## 2.9 运算符

C/C++具有很多内嵌的运算符。事实上，它的运算符比大多数其他计算机语言具有更重要的意义。主要有四类运算符：算术、关系、逻辑和位运算符。此外，还有一些特殊的运算符，用于完成特殊的任务。

### 2.9.1 赋值运算符

可以在任何有效的表达式中使用赋值运算符。大多数计算机语言（包括 Pascal, BASIC 和 FORTRAN）并不能这样做，它们把赋值运算符作为一个特殊语句处理。赋值运算符的一般形式如下：

```
variable_name = expression;
```

其中，表达式可以像一个常量一样简单，也可以像你要求的那样复杂。C/C++ 使用一个等号来表示赋值（不像 Pascal 或 Modula-2，它们使用:= 结构表示赋值）。赋值的目标，或左边必须是一个变量或一个指针，不能是一个函数或常量。

在 C/C++ 文献中，以及在编译器的出错信息中，经常会看到这两个术语：lvalue 和 rvalue。简单地讲，lvalue 是可以出现在赋值语句左边的任何对象，实际上，“lvalue”意味着“变量”。术语 rvalue 指赋值右边的表达式并简单地意味着表达式的值。

### 2.9.2 赋值中的类型转换

当一种类型的变量和另一种类型的变量相混合时，就会出现类型转换。在赋值语句中，类型转换原则是很简单的：赋值右边（表达式一侧）的值被转换为左边（目标变量）的类型，如下所示：

```
int x;
char ch;
float f;

void func(void)
{
  ch = x;     /* line 1 */
  x = f;      /* line 2 */
  f = ch;     /* line 3 */
  f = x;      /* line 4 */
}
```

在第 1 行中，整数变量 x 左边的高阶位被删除，将低 8 位填充到 ch 中。如果 x 是在 255 和 0 之间，ch 和 x 将有相同的值。否则，ch 的值将仅反映 x 的低阶位。在第 2 行中，x 将接受 f 的非小数部分。在第 3 行中，f 把存储在 ch 中的 8 位整数值转换为浮点型的同一值。第 4 行也是这样，除了 f 将把一整数值转换为浮点值外。

在把整数转换为字符、长整数转换为整数时，将去掉适宜数目的高阶位。在大多数 16 位环境中，这意味着当从整数转换为字符时，将丢失 8 位；当从长整数转换为整数时，将丢失 16 位。对于 32 位环境，当从整数转换为字符时，将丢失 24 位；当从整数转换为短整数时，将丢失 16 位。

表 2.3 总结了赋值类型转换。记住，int 到 float，或 float 到 double 等的转换并不增加精度或准确性，这种转换仅仅改变了值表示的形式。此外，当把 char 转换为 int 或 float 时，某些编译器总是把 char 变量作为正数处理，不管它的值是什么。还有一些编译器在进行转换时把比 127 大的 char 变量值作为负数处理。一般来讲，对字符，应该使用 char。当需要避免可能的移植性问题时，使用 int, short int 或 signed char。

为了使用表2.3进行还没有给出的转换，简单地一次转换一个类型，直到完成为止。例如，要从double转换为int，首先从double转换为float，然后再从float转换为int。

表2.3 常见类型转换的结果

| 目标类型 | 表达式类型 | 可能丢失的信息 |
|---|---|---|
| signed char | char | 如果值大于127，目标是负数 |
| char | short int | 高8位 |
| char | int (16位) | 高8位 |
| char | int (32位) | 高24位 |
| char | long int | 高24位 |
| short int | int (16位) | 无 |
| short int | int (32位) | 高16位 |
| int (16位) | long int | 高16位 |
| int (32位) | long int | 无 |
| int | float | 小数部分，可能更多 |
| float | double | 按精度圆整结果 |
| double | long double | 按精度圆整结果 |

### 2.9.3 多重赋值

C/C++允许在一条语句中使用多重赋值给多个变量赋予同一个值。例如，下面的程序给x，y和z赋值0：

```
x = y = z = 0;
```

在专业级程序中，经常使用这种方法给变量赋予相同的值。

### 2.9.4 算术运算符

表2.4列出了C/C++的算术运算符。运算符+,-, *和/的用法与它们在大多数其他计算机语言中的用法相同，可以把它们应用到几乎所有的内嵌数据类型中。当把/应用于整数或字符时，结果取整。例如，5/2在整除时等于2。

模运算符%在C/C++中的用法也与它们在其他语言中的用法相同，生成整数除法的余数。然而，不能把它应用于浮点类型。下面的程序片段演示了%的用法：

```
int x, y;

x = 5;
y = 2;

printf("%d ", x/y); /* will display 2 */
printf("%d ", x%y); /* will display 1, the remainder of
                       the integer division */

x = 1;
y = 2;

printf("%d %d", x/y, x%y); /* will display 0 1 */
```

最后一行打印一个0和一个1，因为在整数除法中1/2为0，且余数为1。

一元减法的结果等于用-1乘它的操作数，即，任何数在其前面放一个减号将改变其符号。

表2.4 算术运算符

| 运算符 | 作用 |
| --- | --- |
| - | 减法，也称一元减 |
| + | 加法 |
| * | 乘法 |
| / | 除法 |
| % | 取模 |
| -- | 减量 |
| ++ | 增量 |

### 2.9.5 增量和减量

C/C++包含了两个其他计算机语言中没有的有用运算符，即增量和减量运算符++和--。++运算符给它的操作数加1，而--则减1。换句话说：

```
x = x+1;
```

与下面的代码相同

```
++x;
```

而

```
x = x-1;
```

与下面的代码相同

```
x--;
```

增量和减量运算符可以放在操作数之前（前缀），也可以放在操作数之后（后缀）。例如，

```
x = x+1;
```

可以写成

```
++x;
```

或

```
x++;
```

然而，当在表达式中使用这些运算符时，在其前缀和后缀形式之间有一个区别。当增量或减量运算符放在它的操作数之前时，增量或减量操作在获得表达式中使用的操作数的值之前执行。如果这个运算符位于它的操作数后，则操作数的值在进行增量或减量操作之前获得。例如，

```
x = 10;
y = ++x;
```

设置y的值为11。然而，如果把这个代码写成

```
x = 10;
y = x++;
```

y被设置为10。两种方法中，x都被设为11，区别在于设置的时间。

大多数C/C++编译器在增量和减量操作时生成快而有效的目标码——比通过使用对应的赋

值语句生成的代码更好。因此，可能时，应该尽量使用增量和减量运算符。

下面是算术运算符的优先级：

| | |
|---|---|
| **最高级** | ++ -- |
| | -(一元减) |
| | * / % |
| **最低级** | + - |

对同一优先级的运算符，编译器是按从左到右的顺序求值的。当然，可以使用括号改变求值的顺序。C/C++对待括号的方式实际上与所有其他计算机语言相同，括号强制使一个操作或一组操作具有更高的优先级。

### 2.9.6 关系和逻辑运算符

依据关系运算符的术语，关系指的是一个值和另一个值之间具有的关系。依据逻辑运算符的术语，逻辑指的是这些关系连接的方式。因为关系和逻辑运算符经常一起使用，所以这里我们把它们放在一起讨论。

真和假值的思想强调了关系和逻辑运算符的概念。在C语言中，真值是除了0以外的任何值，假则是0。如果为假，使用关系或逻辑运算符的表达式返回0；如果为真，则返回1。

C++完全支持真值和假值的0/非0概念，然而，它也定义了bool数据类型和Boolean常量true和false。在C++中，一个0值自动转换为false，而一个非0值则自动转换为true。反之亦然：true转换为1，false转换为0。在C++中，关系或逻辑运算的结果是true或false。但是因为这被自动转换为1或0，所以关于这个问题在C和C++之间的区别大部分来讲是学术上的。表2.5显示了关系和逻辑运算符。这里显示了使用1和0的逻辑运算符的真值表。

| p | q | p && q | p \|\| q | !p |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 |

**表2.5 关系和逻辑运算符**

| 关系运算符 | |
|---|---|
| **运算符** | **作用** |
| > | 大于 |
| >= | 大于等于 |
| < | 小于 |
| <= | 小于等于 |
| == | 等于 |
| != | 不等于 |
| **逻辑运算符** | |
| **运算符** | **作用** |
| && | 与（AND） |
| \|\| | 或（OR） |
| ! | 非（NOT） |

关系和逻辑运算符的优先级都比算术运算符低，即，像10 > 1+12这样的表达式在求值时可把它写成10 > (1+12)，当然，结果为假。

可以把几个操作组合进一个表达式中，如下所示：

```
10>5 && !(10<9) || 3<=4
```

在这个例子中，结果为真。

尽管C和C++都不包含异或(XOR)逻辑运算符，可以很容易地创建一个函数，这个函数使用其他逻辑运算符执行这项任务。当且仅当一个操作数（不是两个）为真时，XOR操作的结果为真。下面的程序包含函数xor()，它返回对它的两个变元执行异或操作的结果。

```
#include <stdio.h>

int xor(int a, int b);

int main(void)
{
  printf("%d", xor(1, 0));
  printf("%d", xor(1, 1));
  printf("%d", xor(0, 1));
  printf("%d", xor(0, 0));

  return 0;
}
/* Perform a logical XOR operation using the
   two arguments. */
int xor(int a, int b)
{
  return (a || b) && !(a && b);
}
```

下表显示了关系和逻辑运算符的相对优先级：

| | |
|---|---|
| **最高级** | ! |
| | > >= < <= |
| | == != |
| | && |
| **最低级** | \|\| |

像算术表达式一样，可以使用括号来改变在关系和/或逻辑表达式中的求值顺序。例如，

```
!0 && 0 || 0
```

为假。然而，当给这同一个表达式加上括号时（如下所示），结果就为真：

```
!(0 && 0) || 0
```

记住，所有的关系和逻辑表达式都生成一个真或假的结果。因此，下面的程序片段不仅是正确的，而且将打印出数1。

```
int x;

x = 100;
printf("%d", x>10);
```

### 2.9.7 位运算符

不像其他的语言，C/C++支持全部位运算符。因为C的设计目的是取代汇编语言进行大多数编程任务，所以需要能够支持可用汇编程序做的大部分操作，包括位操作。位操作涉及测试、设置或移动一个字节或字中的实际位，其中的字节或字对应于char和int数据类型及其变种。不能对 float, double, long double, void, bool 或其他更复杂的类型进行位操作。表 2.6 列出了适用于位操作的运算符，这些操作适用于操作数的各个位。同一真值表控制着按位AND, OR和NOT (1 的补码)操作，就像它们的逻辑对应物一样，除了它们是逐位操作的。异或有下面的真值表：

| p | q | p ^ q |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |
| 0 | 1 | 1 |

如该表所示，仅当一个操作数为真时，XOR 的结果才为真；否则为假。

位操作经常出现在设备驱动程序的应用中，例如调制解调器程序、磁盘文件例程和打印机例程，因为位操作可被用于屏蔽掉某些位，例如奇偶校验位（奇偶校验位确认字节中其余的位不变，它通常是每个字节中的高阶位）。

表 2.6 位运算符

| 运算符 | 作用 |
|---|---|
| & | 与（AND） |
| \| | 或（OR） |
| ^ | 异或(XOR) |
| ~ | 1 的补码(NOT) |
| >> | 右移 |
| << | 左移 |

考虑把位运算 AND 作为去掉一位的一种方法，即在操作数中为 0 的任何位都会使得结果中对应的位被置为0。例如，下面的函数从调制解调器端口中读取一个字符并重置奇偶位为0：

```
char get_char_from_modem(void)
{
  char ch;

  ch = read_modem(); /* get a character from the
                        modem port */
  return(ch & 127);
}
```

奇偶位经常由第8位表示，它通过用一个其第1位到第7位被设置为1、第8位设置为0的字节和它进行“与”操作而被设置为0。表达式ch & 127正是将ch中的位和组成数127的位相“与”而成的，其结果是ch的第8位被设置为0。在下面的例子中，假定ch接收了字符“A”且奇偶位被设置为：

```
奇偶位
↓
11000001        ch包含一个“A”且奇偶位被设置为1
01111111        二进制中的127
&_____          按位“与”(AND)
01000001        没有奇偶位的“A”
```

与按位“与”相反，按位“或”(OR)操作可被用来设置一位。在操作数中设置为1的任何位可以使结果中对应的位被设置为1。例如，下面是128|3的情形：

```
10000000        二进制的128
00000011        二进制的3
|_____          按位“或”(OR)操作
10000011        结果
```

“异或”通常简写为XOR，当且仅当相比较的位不同时，才将对应的位设置为1。例如，127^120的情形如下：

```
01111111        二进制的127
01111000        二进制的120
^_____          按位“异或”(XOR)
00000111        结果
```

记住，关系和逻辑运算符总是生成一个结果，不是真就是假，而类似的位操作可以生成与具体操作一致的任意值。换言之，位操作可以生成0或1以外的值，而逻辑运算符的运算结果总是0或1。

移位运算符>>和<<，按指定的那样将一个值中的所有位右移或左移。右移语句的一般形式是：

value >>*右移位数*

左移语句的一般形式是：

value <<*左移位数*

当位从一端移出时，另一端补0位(对有符号负数而言，右移将引入1，目的是保留符号位)。记住，移位不是循环，即，从一端移出的位并不回送到另一端。移出的位丢失了。

在解码来自一个外部设备、如D/A转换器的输入并读取状态信息时，移位操作是非常有用的。移位运算符也可以快速地乘和除整数。右移一位有效地把一个数除2，而左移一位等价于乘以2，如表2.7中所示。下面的程序演示了移位运算符：

```
/* A bit shift example. */
#include <stdio.h>

int main(void)
{
  unsigned int i;
  int j;

  i = 1;
```

```
  /* left shifts */
  for(j=0; j<4; j++) {
    i = i << 1; /* left shift i by 1, which
                   is same as a multiply by 2 */
    printf("Left shift %d: %d\n", j, i);
  }

  /* right shifts */
  for(j=0; j<4; j++) {
    i = i >> 1; /* right shift i by 1, which
                   is same as a division by 2 */
    printf("Right shift %d: %d\n", j, i);
  }

  return 0;
}
```

1的补码运算符~，颠倒操作数中每一位的状态，即，所有的1被置为0，所有的0被置为1。

位操作符经常用于加密例程中。如果想要使一个磁盘文件不可读，可以对它执行某些按位操作。一个最简单的方法是使用1的补码来颠倒字节中的每一位，以补码每个字节，如下所示：

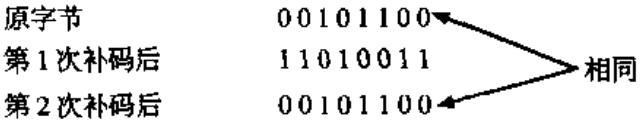


**表2.7 移位运算符的乘和除**

| 无符号 char x; | 每个语句执行后的 x | x 的值 |
|---|---|---|
| x = 7; | 00000111 | 7 |
| x = x<<1; | 00001110 | 14 |
| x = x<<3; | 01110000 | 112 |
| x = x<<2; | 11000000 | 192 |
| x = x>>1; | 01100000 | 96 |
| x = x>>2; | 00011000 | 24 |

* 每左移一位相当于用2乘。注意在x<<2后信息丢失了，因为一位已移出一端。

** 每右移一位相当于除以2。注意其后的除法不会返回任何丢失的位。

注意对同一行进行两次的求补，总是得到原来的数。因此，第一次求补表示对字节编码，第二次求补把字节解码回原来的值。

可以使用下面所示的encode( )函数来编码一个字符。

```
/* A simple cipher function. */
char encode(char ch)
{
return(~ch); /* complement it */
}
```

当然，使用encode( )编码的文件更容易受到破坏。

### 2.9.8 ?运算符

C/C++包含一个非常重要且方便的?运算符，可以取代某些if-then-else形式的语句。?运算符

是三元的，其一般形式为：

```
Exp1 ? Exp2 : Exp3;
```

其中，Exp1, Exp2 和 Exp3 是表达式。注意冒号的用法和位置。

?运算符的作用是：在计算 Exp1 后，如果结果为 true, 则计算 Exp2 并将结果作为整个表达式的值。如果 Exp1 为 false，则计算 Exp3 的值并将结果作为整个表达式的值。例如，在下例中：

```
x = 10;
y = x>9 ? 100 : 200;
```

y 被赋值 100。如果 x 比 9 小，y 的值将变为 200。若用 if-else 语句改写，可得到下面的程序：

```
x = 10;
if(x>9) y = 100;
else y = 200;
```

我们将在第 3 章详细讨论?运算符与其他条件语句间的关系。

### 2.9.9 指针运算符 & 和 *

指针是对象的内存地址。指针变量是专门用来声明保存某一特定类型对象的指针的变量。在某些程序中，知道变量的地址是非常有用的，然而，在 C/C++ 中，指针有三个主要的功能：它们可以提供快速引用数组元素的方法；它们允许函数修改它们的调用参数；最后，它们支持链表和其他动态数据结构。第 5 章将专门讨论指针，然而，本章将简短地讨论两个用来操作指针的运算符。

第一个指针运算符是 &，这是一个返回其操作数的内存地址的一元运算符（记住，一元运算符仅要求一个操作数）。例如，

```
m = &count;
```

把变量 count 的内存地址放到 m 中。这个地址是这个变量在计算机中的内部位置，它与 count 的值没有关系。可以把 & 理解为“什么内容的地址”，因此，前面的赋值语句意味着“m 接受了 count 的地址”。

要更好地理解这条赋值语句，假定变量 count 是在内存 2000 的位置，同时假定 count 的值为 100，那么，在经过了前面的赋值后，m 将变成 2000。

第二个指针运算符是 *，它是 & 的逆操作。* 是一个一元运算符，返回占用操作数所指的内存地址的变量的值。例如，如果 m 包含变量 count 的内存地址，那么

```
q = *m;
```

把 count 的值放到 q 中。现在 q 的值为 100，因为 100 存储在位置 2000 处，而 2000 是存储在 m 中的内存地址。我们可以把 * 理解为“从地址中取值”。在这个例子中，可以把语句读作“q 接受了在地址 m 处的值”。

不幸的是，乘法符号 * 和“从地址中取值”的符号是同一个，并且位 AND 和“什么内容的地址”的符号也是同一个。其实，这些运算符互相之间没有关系。& 和 * 比所有其他算术运算符（除了一元减，它们具有相同的优先级）具有更高的优先级。

包含内存地址（即指针）的变量声明时必须在变量名前加上 *。这指示编译器它包含一个

指针。例如，要把ch声明为一个指向字符的指针，写为，

```
char *ch;
```

其中，ch不是一个字符，而是一个指向字符的指针，这有很大的区别。在这个例子中，指针指向的数据类型char称为指针的基类型。然而，指针变量本身是一个包含基类型对象地址的变量。因此，字符指针（或任何指针）足以容纳由计算机结构所定义的任何地址。然而，原则上，一个指针仅应该指向其基类型的数据。

可以在同一条声明语句中把指针和非指针变量相混合。例如，

```
int x, *y, count;
```

声明x和count为整型类型，y为指向整型类型的指针。下面的程序使用* 和&运算符把值100放到称为target的变量中。正如所期望的，这个程序在屏幕上显示值10。

```
#include <stdio.h>

int main(void)
{
  int target, source;
  int *m;

  source = 10;
  m = &source;
  target = *m;

  printf("%d", target);

  return 0;
}
```

### 2.9.10 编译时运算符 sizeof

sizeof是一个编译时使用的一元运算符，它返回变量或sizeof后面括号中类型的字节长度。例如，假定int是4字节，double是8字节，

```
double f;
printf("%d ", sizeof f);
printf("%d", sizeof(int));
```

则上面的程序执行后将显示8 4。

记住，要计算一个类型的长度，必须把类型名封在括号里。对于变量名，这不是必需的，尽管如果这样做并没有害处。

C/C++使用typedef定义了一个特殊的类型size_t，它与无符号整型基本对应。从技术上讲，由sizeof返回的值是size_t类型的，然而，可以把它想做（并用做）一个无符号整型值。

sizeof主要用于帮助生成依赖于内嵌数据类型长度的可移植代码。例如，想像一个数据库程序，该程序需要为每个记录存储6个整型值。如果想要把此数据库程序迁移到很多种计算机上，不必假定整数的长度，但必须通过使用sizeof确定它的实际长度。按照这一原则，可以使用下面的程序把一条记录写到磁盘文件中。

```
/* Write 6 integers to a disk file. */
void put_rec(int rec[6], FILE *fp)
```

```
{
  int len;

  len = fwrite(rec, sizeof(int)*6, 1, fp);
  if(len != 1) printf("Write Error");
}
```

如所示的那样编码后，put_rec( )在任何环境下都可以正确的编译和运行，包括使用16和32位整数的那些。

最后强调一点，sizeof是在编译时求值的，在程序中，它生成的值被当作常量处理。

### 2.9.11 逗号运算符

逗号运算符把几个表达式串在一起。逗号运算符的左边总是被求值为void，这意味着右边的表达式变成了以逗号分开的整个表达式的值。例如，

```
x = (y=3, y+1);
```

首先将值3赋给y，然后将值4赋给x。因为逗号运算符的优先级比赋值运算符的优先级低，所以必须使用括号。

本质上，逗号会引发一系列操作。当在一个赋值语句的右边使用它时，所赋的值是用逗号分开的最后那个表达式的值。

逗号运算符与一般英语中词组“do this and this and this.”中字“and”的意义相同。

### 2.9.12 点号(.)和箭头(->)运算符

在C中，点号（.）和箭头（->）运算符用于访问结构和联合的元素。结构和联合是复合（也称为聚合）数据类型，可以在一个名字下引用（参见第7章）。在C++中，点号和箭头运算符也用于访问类的成员。

当直接处理结构或联合时，使用点号运算符。当使用结构或联合的指针时，就用到箭头运算符。例如，给定下面的程序片段：

```
struct employee
{
  char name[80];
  int age;
  float wage;
} emp;

struct employee *p = &emp; /* address of emp into p */
```

你会写出下面的语句，把值123.23赋给结构变量emp的wage成员：

```
emp.wage = 123.23;
```

然面，使用emp指针的同样的赋值则是：

```
p->wage = 123.23;
```

### 2.9.13 方括号[ ]和圆括号( )运算符

括号是增加括号中操作的优先级的运算符。方括号执行数组下标化操作（我们将在第4章

详细讨论数组）。如果给定了一个数组，在方括号内的表达式就给那个数组提供一个下标。例如，

```
#include <stdio.h>
char s[ 80];

int main(void)
{
  s[ 3] = 'X';
  printf("%c", s[ 3] );

  return 0;
}
```

首先把值'X'赋给数组s中的第4个元素（记住，所有的数组都以0开始），然后打印那个元素。

### 2.9.14 优先级小结

表2.8列出了C语言定义的所有运算符的优先级。注意所有的运算符，除了一元运算符和?，都是从左到右结合的。一元运算符 (*, &, –)和?是从右到左结合的。

表2.8 C语言运算符的优先级

| | |
|---|---|
| 最高级 | ( ) [ ] -> . |
| | ! ~ ++ -- (type) * & sizeof |
| | * / % |
| | + - |
| | << >> |
| | < <= > >= |
| | == != |
| | & |
| | ^ |
| | \| |
| | && |
| | \|\| |
| | ?: |
| | = += -=*= /= 等 |
| 最低级 | , |

注意：C++定义了另外一些运算符，我们将在本书第二部分讨论。

## 2.10 表达式

运算符、常量和变量是表达式的组成部分。在C/C++中的表达式是这些元素的任意有效的组合。因为大多数表达式都倾向于遵循一般的代数法则，它们经常被认为是理所当然的。然而，表达式的某些方面却是C和C++所特有的。

### 2.10.1 求值顺序

C和C++中都没有指定表达式中子表达式的求值顺序，这就让编译器可以自由地重新安排表达式以生成最优化的代码。然而，它也意味着代码永远不会依赖子表达式求值的顺序。例如，下面的表达式

```
x = f1() + f2();
```

不能保证f1( )在f2( )之前调用。

### 2.10.2 表达式中的类型转换

当在一个表达式中混合有不同类型的常量和变量时，它们都被转换为同一类型。编译器把所有的操作数转换为最大操作数的类型，这称为类型升级。首先，所有的char和short int值被自动转换为int（这个过程也称为整型化升级）。一旦完成了这一步，就逐步进行其他所有的类型转换，像在下面的类型转换算法中描述的那样：

```
如果有一个操作数为 long double，那么另一个转换为 long double;
否则，如果有一个操作数为 double，那么另一个转换为 double;
    如果有一个操作数为 float，那么另一个转换为 float;
    如果有一个操作数为 unsigned long，那么另一个转换为 unsigned long;
    如果有一个操作数为 long，那么另一个转换为 long;
    如果有一个操作数为 unsigned int，那么另一个转换为 unsigned int
```

有一个例外：如果一个操作数为long，另一个为unsigned int，并且如果unsigned int的值不能用long来表示，则两个操作数都转换为unsigned long型。

一旦应用了这些转换原则，每对操作数都变成同类型的，每次操作的结果与两个操作数的类型也是一样的。

例如，考虑在图2.2中出现的类型转换。首先，字符ch转换为整型，然后ch/i的结果转换为double型，因为f*d是double型。f+i的结果是float型，因为f是float型的。最终结果是double型。

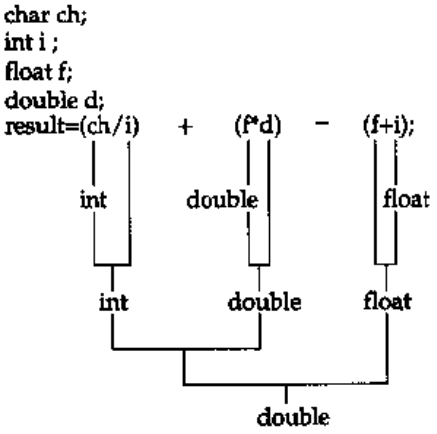


图2.2 类型转换范例

### 2.10.3 强制转换

可以使用强制转换（cast）把一个表达式强制转换为一特定类型，强制转换的一般形式是：

*(type) expression*

其中，type是一个有效的数据类型。例如，为了确保表达式x/2的结果为float型，应该写为：

```
(float) x/2
```

从技术上讲，强制转换是运算符。作为运算符，强制转换是一元的，与其他一元运算符具有同样的优先级。

尽管在编程中强制转换通常用的并不是很多，但是当需要时，它们确实是非常有帮助的。例如，假定你希望使用一个整数控制循环，然而对它执行计算时又要求小数部分，就像在下面的程序中这样：

```
#include <stdio.h>

int main(void) /* print i and i/2 with fractions */
{
  int i;

  for(i=1; i<=100; ++i)
    printf("%d / 2 is: %f\n", i, (float) i /2);

  return 0;
}
```

如果没有强制转换(float)，就会仅执行一次整数除。强制转换保证了也显示答案的小数部分。

**注意**：C++中又增加了四个强制转换运算符，例如const_cast和static_cast。这些运算符将在本书第二部分讨论。

### 2.10.4 空格和括号

为了更易于阅读，可以在表达式中加入空格。例如，下面两个表达式是一样的：

```
x=10/y~(127/x);

x = 10 / y ~(127/x);
```

冗余或附加的括号不会引起错误或降低表达式的执行速度。可以使用括号来使表达式求值的顺序更清楚。例如，下面两个表达式中哪一个更容易读呢?.

```
x = y/3-34*temp+127;
x = (y/3) - (34*temp) + 127;
```

### 2.10.5 复合赋值

赋值语句有一个变种，称为复合赋值，即简化某些赋值操作的编码。例如，

```
x = x+10;
```

可以写成

```
x += 10;
```

运算符+=告诉编译器把x的值加上10赋给x。所有的二元运算符（要求两个操作数的运算符）都有复合赋值运算符。一般来讲，像下面这样的语句：

var = var operator expression

可以改写为

var operator = expression

另一个例子，

```
x = x-100;
```

与下面的语句相同

```
x -= 100;
```

在专业级C/C++程序中，广泛使用了复合赋值，所以应该熟悉它。复合赋值也常常称为简化赋值，因为它更简洁。