第二章 数据类型和运算

总体要求

- 掌握 C 的各种数据类型
- 掌握 C 的表达式
- 掌握常用的 C 运算符

核心技能点

- 掌握一些常用数据类型
- 掌握变量的定义和使用
- 掌握赋值、算术、逻辑、关系运算符

扩展技能点

- 理解条件、逗号、sizeof 运算符
- 了解混合运算规则
- 初步掌握显式类型转换

相关知识点

- C 编译器的使用

学习重点

- 熟练掌握变量的定义和使用
- 熟练掌握赋值、算术、逻辑、关系表达式

C 语言提供了丰富的数据类型和大量的运算符，这大大提高了 C 语言的表达能力，有助于编写复杂的应用程序。这一章里，我们将学习数据类型的概念，以及 C 语言提供了哪些数据类型，同时还将学习到 C 的运算符以及表达式的概念。最后提到了一些编码规范。

2.1 问题的引入

速算比赛的组织者提出的关于程序的要求较多，而读者现在可能缺乏相应的知识。所以，现在将整个问题分解成最基本的小问题，而这些小问题会涉及到 C 语言最基础的知识点。

首先要考虑信息的表示问题。在组织者提出的要求中，有一个是要求信息在内存中的存储。那么先来分析一下哪些信息需要存储。根据要求，以下这些信息时需要存储的：

- 参赛者编号
- 参赛者姓名
- 参赛者性别
- 参赛者年级
- 得分
- 等级

经过分析可以发现，这些信息的性质是不一样的。例如：参赛者年级可能是 3（表示 3 年级），得分可能是 92，它们都是数学意义上的整数；而经过处理后的等级是 A-E 中的一个，是一个字母符号；而姓名 Peter 既不是整数，也不是单个的字母符号，而是由字母符号组成的有序序列（称之为“串”）。显然，这些信息不能用同样的方式表示和处理。那么，这些信息该如何用 C 语言的方式表达呢？又该着呢样处理呢？

再者，以上信息是需要存储的。现在读者都已知道，信息是存储在计算机的内存中的，而内存是由一个个字节单元组成的。那么，每一类信息在存储时，应该占据多少个字节单元呢？如果存储方式定下来后，其中的数据又如何进行修改呢？

以上的这些问题，都涉及到了 C 语言最基本的知识单元：数据类型和变量。因此，在下面的章节中，

将就此问题进行深入的探讨。

2.2 数据类型

日常生活中用到的数据，除了它们本身的值外，同时还隐含了它们的类型（type）信息。例如，某学生是 3 年级学生，那么“3”这个数不仅描述了它在计数系统中所处的位置，也就是术语所说的“值（value）”，同时，在常识认知系统中，由于它没有小数点，因此它是一个整数（integer）。同样的道理，价格标签上的数“99.99”是数学意义上的实数（real）。英文字母“A”告诉了我们，它既不是一个整数，也不是一个实数，而是一个字母符号。姓名“Peter”是一个字符串（string）。那么，C 语言又是怎么表达类型信息的呢？

C 语言提供了丰富的数据类型（data type）来表达信息（及其存储方式）。C 语言的数据类型有整数类型、浮点类型、枚举类型、指针类型、数组类型、结构体类型以及联合体类型等。前三者一般统称为**简单**（simple）类型，指针类型是一种**地址**（address）类型，而后三者被称为**结构化**（structured）类型。

图 2-1 是 C 语言类型的分布图。

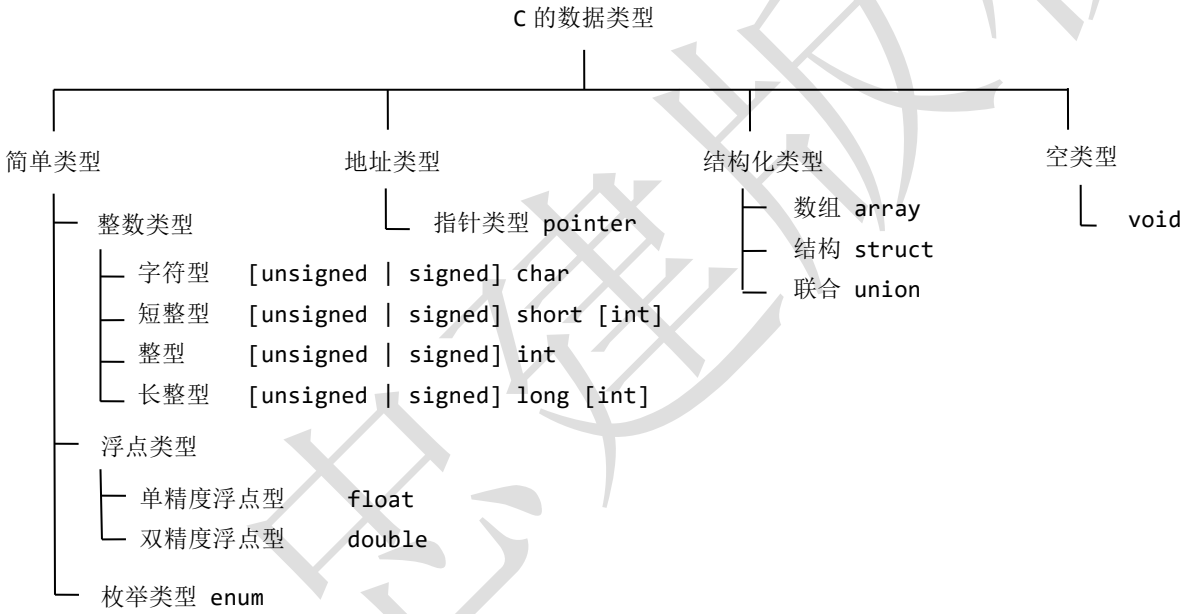

图 2-1 C 的类型

{图 2-1 中没有列出 C99 增加的数据类型，例如_Bool、_Complex 等等。有兴趣的读者可以去查阅 C99 标准的相关资料。

这里解释一下图 2-1 中的用于描述类型的[unsigned | signed]。unsigned 意为无符号，表示它修饰的类型的数只能是 0 或者正数；signed 意为有符号。“[]”表示它括起的内容可以省略；“|”表示它前后的修饰符只能二选一。

如果一个整数类型没有前面的符号修饰符，那么根据 C 语言的规定，这个类型默认为是 signed。

此外，void 类型是一种非常特殊的类型。这是一种不能定义该类型变量和敞亮的未完成（incomplete）类型，一般用在函数的参数和返回值描述中，表示函数没有参数或者没有返回值（详见第 6 章：函数）。或者与指针类型复合在一起形成一种称为“万能指针”类型 void *（详见第 8 章：指针）。}

简单类型中的整数和浮点类型又合称为**数值类型**（numeric types）。这里先就数值类型进行介绍，其它类型在后续章节中讨论。

2.2.1 整数类型

简单地讲，属于整数类型的数就是没有小数点的数，它是一种**内建**（built-in）类型。也就是说，

不用附加的说明，C 编译器也能明白其含义。下面不同进制的数都属于整数类型：

15　-123　10110101_2　177777_8　$715E84A32C90DF6B_{16}$

{下标 2、8、16 分别表示它们属于二进制、八进制、十六进制，无下标表示它们属于十进制。但请读者注意：C 语言不支持二进制数。}

读者已经知道，所有数据在计算机内部都是用二进制的形式存放的。整数也不例外。例如，16 进制整数 715E84A32C90DF6B 将用长度为 64 的二进制串来表示和存放，共需要 8 个字节。

可以看到，整数的大小（长度）会是变化的。针对于这个问题，C 语言提供了完整的解决方案。对于不同的应用要求，C 语言提供了四种常用的标准整数类型：char，short int，int，long int。这四种类型既可以表示正数，也可以表示负数。然而，一些应用程序中的某些部分会只用到正数。C 语言提供的描述符 signed 或 unsigned 会加在类型的前面以表示这种区别。

不同的计算机，或者不同的操作环境，除了 char 类型的长度固定为 1 字节外，其它的整数类型的长度（系指一个整数占据的二进制位数）有可能是不同的。一般的规则是（仅长度）：

char ＜ short ≤ int ≤ long

{一般来说，我们用一个整型的长度来表示计算机处理数据的基本长度单位，称为字长（word length）。日常使用的微型计算机的字长一般都是 32 位。

不同的编译器对整数的长度也有不同的约定。现在流行的 C 语言编译器的约定是这样的：char 的长度为 1bytes（8bits），short 为 2bytes（16bits），int 为 4bytes（32bits），long 为 4bytes（32bits）。如果作为 unsigned 整数，那么它们能表示的最大数分别是：2^8-1、2^{16}-1、2^{32}-1 和 2^{32}-1。如果数值超过了这个范围，那么将会得到错误的结果。

提醒读者注意：一般的程序不会用到（绝对值）特别大的整数，因此使用 int 类型就可以满足需求，可以不必太关心整数的长度问题。}

2.2.2　字符类型

字符型 char 也是一种整数类型，因为它采用了其它整数相同表示模式。但 char 也比较特殊，因为在大多数情况下，并不用它来表示整数，而是使用它的本意：表示字符（character）。一个字符可以是一个英文字母、一个数字，或者是一个其它的特殊符号。

下面的例子都是字符：

‘A’　‘z’　‘5’　‘@’　‘+’

字符在计算机中的存储采用其 ASCII 码。ASCII 码是一种国际标准，用一个字节来代表一个字符的代码。所以，有符号字符的表示范围是从-128 到 127，无符号字符的表示范围是 0 到 255。后者是扩展 ASCII 码的取值范围。例如：’A’的 ASCII 码是 65，’a’的 ASCII 码是 97，’0’的 ASCII 码是 48，空格的 ASCII 是 32。（详情请参阅附录中的 ASCII 表）。

{不是所有 ASCII 码表中的字符都能被打印机打印或显示器显示的。一般地，ASCII 码在 32-126 之间的字符属于可打印字符的范畴。}

一个有趣的事实是，由于字符的存储值 ASCII 码是个整数，因此在某些情况下会把字符数据当作整型数据来参与运算（它本来就是一种整数类型），或者用字符型数据来表示比 short 更短的整数。

因为 ASCII 表是一张有序表，所以其中的字符可以参与比较运算。例如，’A’的 ASCII 码是 65，’B’的是 66，因此可以说，’A’＜’B’。至于其它的很多运算，比如’A’+’B’，其实是没有意义的。

不幸的是，有时初学者还是要犯类似的错误，特别是用到数字字符参与运算的时候，它可能会给初学者错觉，而出现使用上的错误。

【案例 1】一些初学者会认为数 0 和字符’0’是一回事，但这是错误的：数 0 的值就是 0；而字符’0’的值是它的 ASCII 码，也就是 48。

【案例 2】一些初学者会以为字符’9’减去字符’5’会得到字符’4’，但是答案却是整数 4。实际上，两个字符数据在做减法时，不是字符的字面值在进行运算，而是字符数据的 ASCII 码在参与运算。字符’9’的 ASCII 码是 57，字符’5’的 ASCII 码是 53，所以’9’-’5’等价于 57-53，结果就是 4，而不是字符’4’的 ASCII 码 52。

除了用一对单引号括起来的可打印符号，字符数据还可以用转义序列来表示，其形式为'\ddd'，其中 ddd 是使用 8 进制表示的字符的 ASCII 码；或使用'\xdd'，其中 dd 使用 16 进制表示的 ASCII 码。其中，符号'\'称为**"转义字符"**，其含义是其后的符号（序列）不做常规字面解释，而是有其它含义。

我们常用转义序列来表示一些不可打印字符。表 2-1 给出了常用的转义字符。

表 2-1 常用转义字符

转义字符	含义	ASCII 码
\n	新行，光标从当前位置移到下一行的开头	10
\r	回车，光标从当前位置回到本行开头	13
\t	横向制表符。一般地，一个制表符占据 8 个空白字符宽度	9
\\	反斜杠\本身	92
\'	单引号'本身	39
\"	双引号"本身	34
\ddd	八进制表示的字符，例如'\101'表示字符'A'	ddd
\xdd	十六机制表示的字符，例如'\41'表示字符'A'	dd

虽然转义字符在书写时用了多个字符来表示，但一定要清楚，无论''中的字串有多长，它都只是一个字符。

在程序中使用单个字符的情况并不多见，而常用的方式是使用多个字符构成的串，称之为**字符串**（string）。以下是字符串的例子：

"This is a C-style string"　　"A"　　""

可以看出，C 字符串是用双引号""括起来的字符序列。请读者注意，字符串"A"与字符'A'是不一样的，不仅仅是因为用到的引号不同，最重要的区别在于，字符'A'仅仅包含了一个字符，而字符串"A"除了包含一个字符'A'外，还包含了一个用于结束字符串的特殊符号—一个 ASCII 码等于 0 的特殊字符'\0'（这是一个用八进制表示的转义字符）。这是 C 语言对于字符串的特别设计。基于此，所有 C 字符串的存储长度都比它的长度也就是实际包含的字符数目要多 1。最有趣的字符串是""，称之为空串，虽然没有包含任何的有效字符，其长度为 0，但是它的存储长度却是 1，因为它包含了一个结尾'\0'。

{使用字符串时常犯的错误是在其中包含有转义字符时。例如：在 Windows 环境下表示目录的字符串"C:\newbook"在打印时不是预期的那样，而是打印成：

C:

ewbook

其中的目录分隔符'\'和其后的字符'n'被联合解释成了转义字符'\n',也就是换新行。所以，正确的书写应该是这样的："C:\\newbook"。也就是说，如果要取转义字符'\'的字面本意，那么就应该使用\\。}

对于字符类型还有一点说明。几乎所有的程序设计语言都依托于英语，所以它们的字符集也是基于英语的，这对处理其它语言，比如汉语、日语等，带来了困难。一个最简单的困难是：ASCII 字符集基于 8 位二进制编码，因此只能容纳英语字母、数字以及其它共 256 个符号；而最常用到的汉字就有 6700 多个，8 位编码无法适应这样的要求。因此，在颁布的国际标准 Unicode 中，采纳了中国国家标准 GB2312（简体中文字符集），使用 16 位编码来表示汉字。那么，一个汉字字符的代码就要占据两个 ASCII 字符的宽度。为了适应标准，C 语言引入了宽字符集（wide character set）的概念，并用标准类型 wchar_t 来存储宽字符，试图用语言的内部机制解决问题。不过，wchar_t 类型的使用颇为困难，因此建议读者慎用。

{一个替代的解决方案是将一个汉字当作一个 C 字符串使用。这样做就没有使用 C 语言的内部解决方案，而是将问题交给编译器所寄宿的操作系统来解决。然而，如果操作系统没有多语言支持，那么输出的汉字或汉字字符串就是一堆看不懂的奇怪符号。

以下是使用替代方案的汉字字符和字符串的例子：

"汉"　　//长度为 2，存储长度为 3

"C 中的汉字"　　//长度为 9，存储长度为 10

关于字符串的进一步使用我们会在"数组"一章中详细讲解。

这里进一步说明字符集（character set）的情况。除了上面提到的 ASCII、GB2312 之外，常用到的字符集还有 GBK（同 GB2312）、BIG5（繁体中文）、UTF-8（一种 Unicode 变体）等。它们在需要多国语言支持的环境中发挥着重要作用。但一个令人烦恼的问题是，如果一个应用的两个部分（例如 Web 应用中的前台网页和后台数据库系统）使用了不同的字符集，那么系统间的数据交换必须要进行字符集转换。一个好的建议是：尽量使应用中各部分使用的字符集都是统一的。具体的做法请读者查阅相关资料。}

2.2.3　浮点类型

浮点数指的是带有小数部分的数，用来表示数学意义上的实数。实数的表示是：

整数部分.小数部分

其中小数部分看起来也是个整数。除了小数点，整数部分和小数部分在特定情况下可以缺省，但不能两者都缺。下面是一些浮点数的例子：

```
3.14159    -123.450.957      6.      .2
```

C 语言常用的标准浮点类型有两种：float、double。两种浮点数占据的内存大小依赖于机器或者是浮点表示格式(IEEE 或者其它格式)。不过二者的表示范围都非常大：以 IEEE 格式的浮点数为例，最短的 float 类型（4 字节）的表示范围有 6 到 7 个有效数字，范围大约是-10^{38}到10^{38}；double 类型（8 字节）的表示范围有 15-16 个有效数字，范围大约是-10^{308}到10^{308}。

{一般情况下，浮点数的表示范围都大于所有整数的表示范围。因此，如果计算的结果超过了整数的范围（例如一个整数的高阶阶乘或幂次），那么就应该将结果用浮点数来存储。}

对于那些很大或者很小的浮点数，一般我们都会用到科学计数法：

```
3.0E-10    0.5E8      -7.6E20
```

它们分别表示了3.0×10^{-10}、0.5×10^{8}和-7.6×10^{20}。可以看到，数中字母 E 前面的数表示底数，而其后的数表示 10 的幂次。

需要注意的是，由于所有的计算机都是用二进制的方式来存储数据的，因此这影响了浮点数的表示精度，浮点数的存储值和实际值是有很微小的差别的，比如 1.0，它的存储值也许会是 0.9999998。在大多数情况，这个差别不会对应用造成影响，但是在某些情况下我们不能忽略累积的误差。

2.3　标识符与变量

在第一章中曾经提到过，一种程序设计语言拥有自己的词汇，并且程序员常常利用词法规则创造出新的词汇。这些词汇在 C 语言中称之为**“标识符**（identifier）”，而变量名就是标识符的一种。

2.3.1　标识符

为了使 C 程序更加易读易写，程序员会用一些有意义的名字来标识程序元素，比如数据、函数名等等。那些用来标识程序元素的名字就是标识符。使用用户自定义标识符也是所有高级语言的特征。

标识符是由字母 A-Z，a-z、数字 0-9 和下划线’_’混合而成的。不过，所有的标识符必须以字母或者是下划线’_’开头。

下面的例子都是合法的 C 标识符：

```
print      anObject      sema4      _send2Fax
```

而下面一些都不是合法的标识符及错误原因，如表 2-2 所示：

表 2-2　非法标识符及错误原因

非法标识符	错误原因
32bytes	以数字开头
an Object	包含空格
million$	包含无效符号$
nine-five	包含 C 运算符-
double	double 是一个 C 关键字，又称保留字

C 语言的词汇表里保留了一些词汇，称为**关键字**(key words)或**保留字**(reserved words)，它们不能作为用户自定义的标识符使用。详见附录 A：C99 关键字。

在命名一个标识符的时候，最好遵循一些常用的约定。例如：

- 取一个有意义的名字。例如，在命名一个用于计数的标识符时，名字 counter 显然好于只把它简单地命名为 c。
- 如果名字由多个单词组成，那么除了第一个单词外，以后每一个单词的第一个字母用大写，例如 sendToFax。注意，C 语言是大小写敏感的，例如，object 和 Object 是不一样的。

{上述约定不是 C 语言标准的一部分，而属于程序员的编程风格。风格是没有标准的，但必须以程序容易阅读为准则。}

习题 2-1　请辨别下列哪些标示符是合法的，如果不合法，请说明原因。

```
auto     Binary       cost4x       Continue
you#me       _illegel gotoLine123  MAINFUNCTION     “King”
Send_Documents_To_Printer_In_Managers_Office
```

2.3.3 变量

读者已经知道，程序中用到的数据会被存储在内存单元中，而这些内存单元都用它们唯一的地址来标识。可以想象，通过地址来访问内存单元是多么的麻烦。所以，在高级语言程序中，为了能够方便地访问内存单元，都会用一个标识符来命名内存单元，说简单一点，就是为内存单元取一个容易记住的名字。这样在访问那个内存单元时，可以使用它的名字而不是它的唯一地址。

在为内存单元命名的同时，还需要指定数据的类型，以便编译器能确定该数据将占据多大的空间以及数据可以参与的运算。由于内存单元里的数据在程序运行时是被频繁读取和修改的，是可变的，因此这个内存单元被称为**“变量**(variable)**”**，而在程序中命名一个变量及指定类型的过程就是**“变量声明**(variable declaration)**”**。例如对于学生编号、姓名和年级，就可以使用如下变量声明：

```
int studentID;
char gender;
int  grade;
```

一般地，变量声明的语法是：

类型名 变量名[= 值][, …];

其中的“=值”表示的功能称为**“初始化”**，其含义是将符号=左边的变量的值指定为给定值；“…”表示在一个类型名下可以同时声明多个同类型变量，每个变量名间用“,”隔开，例如：

```
int    studentID, grade; //声明了两个整型变量
```

而以下的声明是错误的：

```
int a, b          //漏掉了结尾的“;”
int a, b,     //结尾的“;”误写成了“,”
int a; b;     //a 和 b 之间的分隔符“,”误写成“;”
int a, int b;   //b 前面的 int 是多余的
```

除了特殊的场合，未初始化变量的值是不定的，不能直接使用该变量参与运算，否则会造成不可预知的结果。解决问题的方式是在声明的同时给出初始值。例如：

```
char gender = ‘M’;
int  studentID = 1;
int  grade = 3;
```

另一种解决方式是在程序的可执行语句中显式地给变量赋值。例如：

```
grade = 3;
```

这称为**“赋值语句”**，其中的运算符“=”称为“赋值运算符”。该语句的含义是将=右面的值复制到左面的变量单元中。此例中，变量 grade 将会获得确定的值 3。

{变量的初始化和使用赋值语句虽然在形式和功能上都非常相似，但二者的执行路线和时间是完全不同的：初始化工

作是在程序编译时完成的，赋值是在程序运行时完成的。因此，不能将初始化视为赋值。}

一个变量在初始化时可以使用另一个已经被初始化过的变量的值，例如：

```
int i, j = 1, k = j;  //i 的值未定，j 的值是 1，k 的值和 j 一样等于 1
```

一旦声明了一个变量，这个变量在运行时就会在内存中占据一定大小的空间。例如变量 grade 在内存中的映像如下图 2-2 所示：

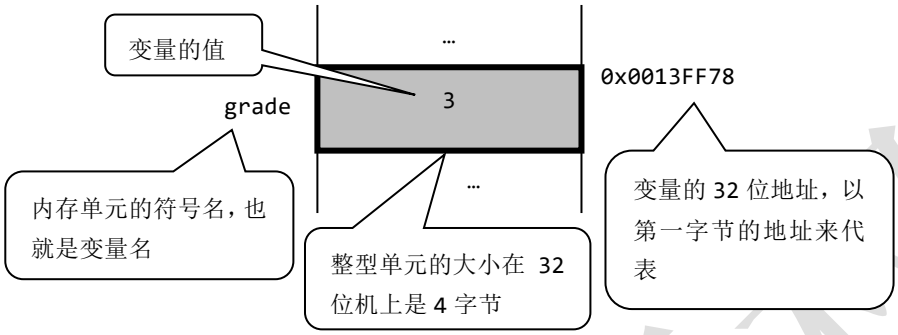

图 2-2 变量的内存映像

如果一个变量一经声明，那么就应该只在该变量中存入指定类型的数据。例如，往一个整型单元中存入一个浮点数是不应该的。但反过来，往一个浮点单元中存入一个整数却是可行的。

{一般情况下，要将一个类型为 T1 数存入到一个类型为 T2 的变量中（或者初始化），如果 T2 的表示范围比 T1 大，那么这项操作是可行的。这种现象符合“类型相容原则”，即“大”类型可以容纳“小”类型。而反过来则有问题，是否产生错误要视编译器的行为。但请读者注意，从大往小赋值或初始化有潜在的数据丢失危险。}

特别提醒读者注意：根据 C 语言的规定，变量在使用之前，必须提前声明它们，并且把它们放在所有可执行语句之前。

习题 2-2　请说明下列变量声明及其初始化中，哪些是正确的，哪些是错误的。

```
char c = 65;
int i = ‘A’;
float f = i;
double d = f;
long l = d;
int a = b = 0;
```

2.4　常量和枚举类型

在程序中我们常常会直接使用一些诸如 3.14、’A’这样的字面值直接参与运算。这些字面值就是所谓的**字面常量**(literal constant)，其含义为不能修改的量。字面常量使用起来可能很方便，但也有可能给程序的维护带来困难。因此，C 语言提供了更多的使用常量的方法。

2.4.1　常量

1. 字面常量

字面常量是有类型的，其类型遵循编译器的内部约定。一般地，没有小数点以及不是用科学技术法表示的数被当作 int 常量，否则就被当作 double 常量；用单引号括起来的是 char 常量；用双引号括起来的是字符串常量。

一些前缀符号可以用于常量，如表 2-4 所示。

表 2-4 常量前缀

前缀	含义	实例	可省略
+	正号	+101	√
-	负号	-65.98	×

0	八进制标志	0377	×
0x/0X	十六进制标志	0xABCD	×
L	宽字符或宽字符串	L'A' L"C-style string"	√

此外，一些后缀符号可以用于常量，如表 2-5 所示。

表 2-5 常量后缀

后缀 [1]	含义	实例	可省略
U	无符号整数	127u	√
L	长整数	97L	√
UL	无符号长整数	34ul	√
F	float 类型浮点数	1001.05f	√
LF	double 类型浮点数	537602.7lF	√

注 1： 不分大小写。

2. 命名常量

字面常量的使用方便，但却有明显的缺点：

- 当在程序中多处使用了相同的字面常量，而后来又要对这个常量进行修改时就会显得非常麻烦，这无疑增加了源代码的维护开销；
- 字面常量常常没有明确的类型信息，它们的类型采用编译器的约定。

为了解决上述的问题，C 语言提供一种更好的**命名常量**（named constant）方式来描述常数。以下是一个命名常量声明：

```
const float PI = 3.14;
```

const 关键字的作用就是"冻结"一个量，它的值不能被修改，是一个**只读**（read only）量。

在此后进行的运算中，可以使用已定义的命名常量，例如：

```
area = r * r * PI;
```

使用命名常量有这样一些优点：

- 对于程序作者之外的其他人员，字面常量往往是一堆"神仙数字"，很难理解其含义。而命名常量的名字就可以非常清楚地说明该常量的用途，便于代码的阅读理解；
- 一处修改，全程有效。

所以，笔者强烈建议在程序中使用命名常量。

请读者注意，命名常量必须被初始化，否则，将没有任何机会给这个常量一个确定值了。

3. 用#define 定义符号常量

使用常量的另一种方式是使用编译预处理指令#define 来定义一个符号常量，例如：

```
#definePI  3.14
```

其中，"#define"是一条编译预处理指令（请读者之一前导符号#的存在）。这条预处理指令的意思是：将符号 PI 定义为字符串"3.14"（而不是数），在此后的程序中可以使用 PI 来代替字面常量 3.14。例如：

```
area = r * r * PI;
```

上述语句会被编译器在编译之前替换为：

```
area = r * r * 3.14;
```

{请读者注意这些事实：

- #define 指令后没有分号；
- 定义的字符串不需要用双引号括起来；

关于#define 具体的使用情况请参阅第三章：编译预处理。}

习题 2-3　请读者思考：为什么命名常量必须被初始化？

习题 2-4　请说明下列声明或其初始化中，哪些是正确的，哪些是错误的。

```
const char flag;
```

```
const long MAX = 1024L;
const double cd = d;
```

2.4.2　枚举类型

程序中经常会遇到使用某些值来表述系统的状态，比如交通控制的红绿灯状态。可以使用一个字符或者整数来表示它们，例如使用'r'、'g'、'a'来表示。但是它们很不直观。最好的方法是使用枚举常量，这涉及到枚举类型的使用。

枚举类型是一种简单类型，其定义语法如下：

enum 枚举标志名 { 枚举常量[, 枚举常量] };

下面就是关于交通灯的一个枚举定义：

```
enum Trafficlight { RED, GREEN, AMBER };
```

上例中定义了三个枚举常量：RED、GREEN 和 AMBER。这三个常量不是字面常量，也不是字符串，请读者在使用时注意。

枚举类型是一种有序类型，也就是说，其定义列表中的常量是有序的。例如上例，有：

```
RED < GREEN < AMBER
```

实际上，C 语言会为每一个枚举常量对应了一个整数。在没有显式给出其对应关系时，编译器将自动为每一常量分配一个值：第一个的值为 0，此后依次递增。例如上例，常量 RED、GREEN 和 AMBER 的对应的值分别是 0、1 和 2。

C 语言允许显式地为每一个枚举常量赋予值，其语法如下：

```
enum Trafficlight { RED, GREEN = 3, AMBER };
```

在这种情况下，RED 的值为 0，而 GREEN 的值被指定为 3，而其后的每一常量都从这个值开始递增。也就是说，AMBER 的值为 4。

不像其它数值类型，枚举值不能被直接输入和输出。不过，我们可以通过其它方式完成这项操作。

虽然枚举常量可以被当作一个整数使用，但这里笔者强烈不建议读者这么做，仅将枚举值当作状态的表示，而不使用它们参与运算。

特别提醒，枚举标识名，以及以后要学习的结构标识名都不是类型名，因此必须在声明变量时在标识名前面加上 enum 等关键字。例如，定义枚举变量的语法为：

```
enum Trafficlight tl;
```

习题 2-5　请读者设计一个枚举类型来表示一周的 7 天。

习题 2-6　请读者设计一个枚举类型来表示一年的 12 个月。

2.5　运算符和表达式

C 语言提供了非常丰富的**运算符**（operator）。在程序中，使用运算符来连接运算对象，从而构成了完成一定运算功能的**表达式**（expression）。

2.5.1　运算符和表达式概述

1）运算符

除了常规的加减乘除等算术运算，C 语言还提供了其它一些完成运算的符号。这些符号称为“**运算符**”。每一种 C 运算符都会连接不同数目的运算对象：至少 1 个，最多三个。运算对象一般称为“**操作数**（operand）”。根据所连接的操作数数目的不同，C 运算符分为三类：

- **单目**（unary）运算符。C 语言的单目运算符有：+（正号）、-（符号）、++、--、!、~、^、sizeof、*、&、（类型名）等 11 个。它们只连接一个操作数，一般出现在操作数的前面。但有些单目运算符，如++、--，既可以出现在操作数的前面，也可以出现在后面，它们有不同的运算结果；
- **双目**（binary）运算符。除了单目运算符和?:之外，其它的都是双目运算符。它们连接两个操作数，这两个操作数分布在运算符的两侧；

- **三目**（tertiary）运算符。C 语言的三目运算符只有一个，就是?:。这是一个非常有特色的运算符，它连接三个操作数。

2) 表达式

C 表达式是一种运算单元，能够完成指定的运算并得到结果。以双目运算符为例，其格式类似于：

左操作数 运算符 右操作数

这与数学上常用的运算式是一致的。

最简单的 C 表达式是一个变量名，或者是一个常量名，或者是一个字面常量，它们都可以不包含任何的运算符。例如：

```
2       //整数常量表达式
grade   //变量表达式
“abc”   //字符串常量表达式
```

很多的表达式除了有不同的运算符外，可能还会加入()，使其变得复杂。例如：

```
(a + b) / 2
```

任何一个合法的 C 表达式都会计算出一个值，值的类型依赖于参与运算的类型。例如：单变量表达式的结果值就是变量的值,而表达式结果的类型就是变量的类型。而其它较复杂的表达式会有一些不同。

{在任何一条合法的 C 表达式后跟上一个分号;就构成了 C 语句。与表达式不同，语句完成一定的功能，但不产生值结果。关于语句在后续的章节中讲解。}

2.5.2 赋值运算符和赋值表达式

赋值表达式形如：

数据对象 = 表达式

其中，一个**数据对象**（data object）要占据一定大小的内存单元，其内容能够被修改。绝大多数情况下，一个数据对象是一个变量。这样的数据对象又称为“**左值**（lvalue）”，顾名思义，就是能出现在赋值号左边的值。常量因其不能被修改而不能作为左值，除了常量声明外，它只可能出现在赋值号的右边。这种值称为“**右值**（rvalue）”。

赋值表达式的直观含义是将表达式的值复制到数据对象中。

以下是赋值表达式的实例：

```
i = 0          //变量被常量赋值
f = 1.5E8
a = b          //变量被变量赋值
```

{请读者注意：1. 赋值号不是数学上的等号；2. 表达式后面没有分号！否则，将成为一条语句。}

赋值表达式要完成两项操作：

(1). 将右边表达式的值装入左边数据对象的内存单元；

(2). 计算出整个表达式的值，它等于右边表达式的值。例如 a = b，不仅变量 a 被装入了变量 b 的值（也就是 a 和 b 的值相等），并且整个赋值表达式的值也等于 b，而且 a 和赋值表达式 a=b 结果的类型与 b 相同。

要完成一个赋值操作，要求左右值拥有相同或者相容的数据类型。

{类型相容的意思是：左右值都是数字类型，并且左值的表达范围比右值的表达范围更大。例如：浮点类型的表达范围就比所有的整型都大。如果有 int a = 0; double b = 1.0; 那么：

```
b = a //ok
```

a = b //error。由于 b 的范围比 a 大，因此可能存在潜在的数据丢失危险。}

因为赋值表达式不是数学意义上的等式，因此，以下在数学上不成立的表达在高级程序设计语言里却是合法的

```
i = i + 2
```

如果 i 的初始值是 0，那么这个表达式首先完成的 i + 2，得到结果 2，然后再将结果存回到变量 i 中。此后，i 的值就从 0 被修改为 2，而整个赋值表达式的值也等于 2，其类型是整型。

2.5.3　算术运算符和算术表达式

算术表达式使用算术运算符和/或括号来连接两个操作数。有这样一些算术运算符：

+　-　*　/　%　++　--

以下是算数表达式的实例：

```
a + b
price * discount
17 % 3
++i
```

其中，运算符%被称为取模（或取余）运算符。简单地说，取模运算符的作用是求一个数除以另一个数的余数。以下是取模运算的实例：

```
17 % 3  //结果为 2
-17 % 3 //结果为 1
17 % -3 //结果为 2
```

这里要特别提到除法运算。C 语言规定，如果参与除法的两个操作数都是整数，那么除法结果也是一个整数，即两个数的商。例如有 `int a = 2, b = 5`，那么 `a / b` 的结果是 `0`，而不是 `0.5`；而 `b / a` 的结果是 `2`，而不是 `2.5`。请初学者一定要注意这个问题。

所有的双目算术运算符，包括以后要学习的所有非赋值类的双目运算符，都要按指定运算产生一个结果，而这个结果被存储在一个匿名临时单元中。匿名的意思是不知道它的名字，也不知道它的地址，所以程序员无法访问到它。可以认为，那个匿名单元是一个**常量**对象，因此它不能作为左值。也就是说，上述的双目表达式不能出现在赋值号的左边，例如赋值表达式 `a + b = 2` 是不正确的，因为 `a + b` 不是左值。

算数运算符中，有两个非常有特色的运算符：++和--（注意：在书写时两个符号间不能有空格），不仅仅因为它们是单目运算符，还在于它们完成的奇妙的运算。下面就来仔细讨论一下它们的用法。

1） 自加运算符++。

++运算符连接一个操作数，既可以出现在其前面（称为**前缀**），也可以出现在其后面（称为**后缀**），它们会产生不同的效果。

- 前缀自加。例如，有声明 `int i=1`，那么前缀自加表达式

  ```
  ++i
  ```

  完成的运算等价于

  ```
  i = i + 1
  ```

  也就是说，最终结果是 `i=2`，并且整个表达式的值也等于 2。

  再来看一个例子。如果有声明 `int a = 0, b = 0`，那么赋值表达式

  ```
  a = ++b
  ```

  完成的运算是：

  (1)b 自加，此后 b=1；而++b 这个表达式的值也等于 1；

  (2)a 被赋予++b 这个表达式的值，也就是 1。

- 后缀自加。例如，有声明 `int i=1`，那么后缀自加表达式

  ```
  i++
  ```

  完成的运算分两步完成：

  ```
  t = i   //t 是一个临时单元，此时 t=1
  i = i + 1
  ```

  也就是说，最终结果是 `i=2`，并且整个表达式的值等于 t，也就是 1。

  根据上述规则，如果有声明 `int a = 0, b = 0`，那么赋值表达式

  ```
  a = b++
  ```

  完成的运算是：

  (1)b 自加，此后 b=1；而 b++这个表达式的值等于 0；

(2)a 被赋予 b++这个表达式的值，也就是 0。

2） 自减运算符--。

自减运算符--的行为非常类似于自加运算符++，只不过完成的是减 1 运算，因此这里就不再赘述。

从上面的讲述可以看到，++和--都要完成隐含的赋值操作，因此它们连接的操作数必须是个左值，绝大多数情况下是个变量，变量的类型也必须是数字类型（布尔类型除外）。

{正是由于++和--作为前缀和后缀是有区别的，所以这里建议大家尽量使用前缀方式，以免造成不必要的麻烦。}

【例 2-1】 ++和--运算符的使用。

```
//ex2-2.c
#include <stdio.h>

int main()
{
    int a = 17, b = 3;

    //a = 17, b = 3
    b = ++a;
    printf("After b = ++a, a = %d, b = %d\n", a, b); //a = 18, b = 18

    //a = 18, b = 18
    b = a--;
    printf("After b = a--, a = %d, b = %d\n", a, b); //a = 17, b = 18

    return 0;
}
```

例 2-1 的结果是：

2.5.4 关系运算符和关系表达式

关系表达式使用关系运算符和/或括号连接操作数。有这样一些关系运算符：

> >= ==（等于） < <= !=（不等于）

所有关系运算符都是双目运算符，其运算结果是一个逻辑值，称为**布尔值。**布尔值的取值范围只有两个：真和假。在 C 语言中，用 0 值代表逻辑假，用 1 表示真。实际上，C 把所有非 0 值都当做逻辑真，包括非 0 负数。不过，关系和逻辑运算如果得真结果，那么它的值只是 1。

{在 C99 标准中，逻辑值的类型是_Bool。_Bool 类型的取值范围只有两个值：0 和 1。但遗憾的是，不是所有的 C 编译器都支持_Bool 类型。}

关系表达式完成比较运算，只能产生真假结果，也就是只能产生 0 或者 1 的结果。下面是一些例子。

假设有声明

```
char ch1 = 'A', ch2 = 'B';
int a = 0, b = 1, c = 2;
double x = 11.1, y = 22.2, z = 33.3;
```

那么：

```
ch1 < ch2      //表达式的值真=1，因为'A'的 ASCII 码为 65，'B'的为 66，65<66
a + b != c     //表达式的值真=1。关系运算符的优先级比算术运算符低，所以先做加法
x >= y * z     //表达式的值假=0
```

关于关系运算有几个值得注意的问题：

1） 赋值号=和等号==的使用。初学者往往会在需要==的地方误写成=，因为那是数学意义上的等号。但这可能会带来不正确的结果，而这种错误属于逻辑错误，编译器和链接器都无法检测出来，所以非常难于排错。举个例子来看：设 ch1='a'，ch2='b'，如果将 ch1==ch2 写成了 ch1=ch2，那么将会产生不是预期的结果：前者是关系表达式，会得到 0 值；而后者是赋值表达式，结果是 ch2 的值，也就是'b'。由于'b'的 ASCII 码是 98，因此被当做是真。显然，这是两个南辕北辙的结果。

2） 浮点数的比较。由于浮点数在存储上的精度误差，因此可能会使==和!=不能进行精确比较。解决的方式是求两个浮点数差值的绝对值是否小于一个给定的很小的数，例如：x == y 最好写成 fabs(x-y) < 1E-6。

3） 不能使用类似于数学上的 a<b<c 这样的形式来直接表示三个数的大小关系，只能将其分解为 a<b && b<c 这样的方式。符号&&表示并且，详见下节。

{函数 fabs()是 C 语言数学库中的一个函数，其功能是求实数的绝对值。它的声明被包含在头文件 math.h 中。值得注意的是：求整数绝对值的函数是 abs()。

a<b<c 这个关系运算符级联表达式在语法上居然是正确的，但其含义并非是它字面表达的那样。这个表达式被解读为(a<b)<c。如果 a=1，b=2，c=3，那么 a<b<c 在数学上成立。但在 C 中，a<b 得结果 1，那么再做 1<c，得结果 0，这显然不是想要的结果。}

2.5.5　逻辑运算符和逻辑表达式

逻辑表示式使用逻辑运算符和/或括号连接操作数。C 语言提供三个逻辑运算符：

&&（逻辑与）　　||（逻辑或）　　!（逻辑非）

其中，&&、||是双目运算符，!是前缀单目运算符，它们的运算规则如表 2-6：

表 2-6 逻辑运算规则

a	b	!a	a && b	a || b
0	0	1	0	0
0	1	1	0	1
1	0	0	0	1
1	1	0	1	1

{逻辑或运算符||由两个紧邻的竖线符号组成。竖线符号键的位置在回车键上方，并且是个上位键（即在按此键的同时需按住 Shift 键）。}

逻辑表达式中的操作数往往是关系表达式：

```
a < b && b < c        //1 && 1 = 1
z > y || y < x        //1 || 0 = 1
!(ch1==ch2)           //!0 = 1
```

由于 C 语言对逻辑真假值的灵活处理，可以发现，在一个值与 0 进行比较时，下面两种方式是等价的：

- !a 和 a==0（如果 a 是 0 值，!a=1，0==0 得到结果 1，二者结果相同。反之亦然）
- a 和 a!=0（如果 a 是 0 值，0!=0 得到结果 0，二者结果相同。反之亦然）

据此，可以写出这样的逻辑表达式，其中包括变量和算术表达式：

```
!c && (b + 2)              //!2 && 3 => 0 && 1 = 0
```

然而，这并不是好的表达，最好老老实实地把它写成：

```
(c == 0) && (b + 2 != 0)
```

2.5.6　条件运算符和条件表达式

条件表达式形如：

关系表达式 ? 表达式 1 : 表示式 2

其中条件运算符**?:**是 C 语言仅有的一个三目运算符。

条件表达式完成的运算是：

- 如果关系表达式的值为真（1），那么整个条件表达式的值为表达式 1 的值；
- 否则，为表达式 2 的值。

这是一个实例：

```
int a = 1, b = 2, c;
c = a > b ? a : b
```

因为 a>b 的结果为 0，因此右边的条件表达式的结果为 b，而 c 被赋予 b，整个赋值表达式的值也是 b。

使用条件表达式可以有效地简化源代码，同时不降低可读性。

【例 2-2】 条件运算

```
//ex2-2.c
#include <stdio.h>

int main()
{
    int a = 1, b = 2, c;

    c = a > b ? a : b;
    printf("a=%d, b=%d, c=%d\n", a, b, c);

    return 0;
}
```

例 2-2 的结果是：

```
Xxxxx
```

2.5.7 逗号运算符和逗号表达式

除了作为分隔符出现外，逗号还可以作为运算符出现。使用逗号运算符的逗号表达式形如：

表达式 1 ，表达式 2

在执行时，两个表达式按先后顺序依次计算，而整个逗号表达式的值等于最右边表达式 2 的值。例如有 b = 2, c = 4，那么在表达式 b * 3, c – 2 中，首先计算表达式 b * 3 的值，得到结果 6，但在本例中，这个结果被忽略了；接下来计算表达式 c – 2 的值，得到结果 2，最后整个逗号表达式的结果为 2。

在很多情况下，逗号只是作为分隔符出现的，比如在变量声明中，在函数的参数列表中。它偶尔会被当作运算符，一般出现在那种只能用到一个表达式，但又需要完成若干不同运算的场合，例如在 for 语句的循环控制部分。

2.5.8 复合赋值运算符和复合赋值表达式

在众多的赋值运算符中，除了=外，其它的都是复合赋值运算符，它们的形式如：

数据对象 运算符= 表达式

这等价于

数据对象 = 数据对象 运算符 表达式

其中的运算符必须是双目运算符。

由于有赋值操作，因此式中的数据对象必须满足在赋值表达式中讲解到得一切规定。以下是一些实例：

```
a += 2          //a = a + 2 = 2
b *= c – 5      //b = b * (c – 5) = b * -3 = -3
```

```
c %= 2          //c = c % 2 = 1
```

2.5.9　sizeof 运算符

基于可移植的考虑，C 语言提供了一种用于计算类型或其变量在内存中所占字节数的运算符，这就是 sizeof 运算符，其语法如下：

sizeof(类型名) 或 sizeof(变量)

下列代码示意了 sizeof 运算符的使用方法。

【例 2-3】sizeof 运算符的使用

```
//ex2-3.c
#include <stdio.h>

int main()
{
      double d;

      printf("size of char:%d\n", sizeof(char));
      printf("size of int:%d\n", sizeof(int));
      printf("size of double:%d\n", sizeof(double));
      printf("size of d:%d\n", sizeof(d));

      return 0;
}
```

程序的结果如下：

```
xxxxx
```

注意：这个程序在不同的编译环境或不同的硬件平台上可能会有不同的运行结果。

习题 2-7　如有声明

```
int a = 2, b = 4;
```

请分析下列表达式执行后 a、b 的值以及整个表达式的结果。

```
(1) a = b = a
(2) b -= a++
(3) a -= b *= b /= 2
(4) a = b >= a
(5) b = a && a > 0
(6) a = b / 10 ? 1 : 0
(7) b = a / b, a * b
```

习题 2-8　给定声明

```
float x, y, z;
```

请将下列数学公式用合法的 C 表达式来描述：

(1) $\frac{x^2}{4}+\frac{y^2}{9}+\frac{z^2}{25}$

(2) $\frac{4x^2-3xy+y^2+5x-2y-1}{2x-y}$

2.6　混合运算

在前面所举表达式的实例中，大多数只包含一个运算符。而在实际应用当中，用到的表示式是比较

复杂的，包含多个运算符，从而完成功能复杂的运算。不过，有一些问题需要注意。下面的内容将就此展开讨论。

2.6.1 运算符的优先级规则

假设有声明 `int a = 1, b = 2, c = 3;`，表达式

```
a + b – c
```

是按`(a + b) – c` 的顺序进行的，而不是 `a + (b – c)`。这种性质称为“**运算符的结合性**”。一般说来，运算符和操作数的结合是从左至右进行的。不过，这个规律会被破坏。考虑表达式

```
a + b * c
```

根据经验，知道运算是这样发生的：

```
a + (b * c) = 1 + 6 = 7
```

而不是按出现的先后顺序计算：

```
(a + b) * c = 3 * 3 = 9
```

造成以上结果的原因是，运算符*比+有更高的**优先级**（precedence），它指明了在混合运算中哪些运算先于其它运算发生。（详见附录 C：运算优先级）。

在例 2-6 中，如果试图用表达式：

```
a = b * 3, c – 2
```

来接收逗号表达式的值，那么请问最终 a 的结果是多少？

相信大多数的读者会回答是 2。如果没有赋值的存在，右边逗号表达式的值的确是 2。但请读者查阅附录 C，可以发现，逗号运算符拥有最低的优先级，因此上述表达式的正确解读应该为：

```
(a = b * 3), c – 2
```

因此，a 的结果应该是 6。

现在通过一些实例来更深入地了解结合性。

设有复合赋值表达式 `a = b = c`，很多读者会认为 b 被 c 赋值，a 被 b 赋值，其实不然。因为所有赋值运算符的结合性是从右至左，所以这个表达式正确的解读为：

```
a = (b = c)
```

也就是说，首先执行表达式 `b = c`，使 b 拥有与 c 一样的值，同时表达式 `b = c` 也拥有与 c 一样的值；此后这个值被赋给了 a。换句话说，变量 a 是被赋予了另一个赋值表达式的值，而不是被 b 赋值。最后整个复合表达式的值等于 c。

再来看一个复杂一点的例子。

```
int i = 0, j;
j = ++++i
```

请问 j 的值等于多少？

这不仅是让读者头疼的问题，也是让编译器不知所措的问题，因为表达式`++++i` 有多种解释：

```
+(+(+(+i)))
+(+(++i))
++(++i)
+(++(+i))
++(+(+i))
```

到底哪个是最合理的呢？不同的编译器对待类似的问题有不同的处理方式，得到的结果也就不一样。所以，这个例子的教训是：不要试图去编写看似节约代码但却更加复杂的表达式。如果一定要在一个表达式里完成，可以在运算符中加入空格或者圆括号来明确地告诉编译器读者的用意，例如：`j = ++ ++i`。不过，这仍不是最佳选择。

读者可能已经注意到，大多数运算符的结合性都是从左到右，而所有的单目操作符都是从右到左结合的。这是有意义的。考虑表达式`-+a`，如果从左到右结合，就是`(-+)a`，这显然没有意义；而从右到左的结合就是`-(+a)`，这就显得很自然。

为了让优先级较低的运算提前进行，我们可以使用圆括号()来提升其优先级，例如表达式(a + b) * c 中，+运算先于*运算发生。在这里提醒各位读者，C 语言中没有用数学上常用的中括号[]和大括号{}来表达括号的层次，所有出现括号的地方都使用()。[]和{}被赋予了其它含义：[]是数组的标志，()是函数的标志。关于二者的知识将在后续的章节中学习到。

2.6.2　类型转换

表达式中可能包含不同类型的操作数。当出现类型不一致时，表示范围小的那些操作数的类型（以下称短类型）会转换成为表示范围更大的那些操作数的类型（以下称长类型）。这就是类型转换的概念。常见的类型转化有隐式转换和强制转换两种。

1．隐式转换（implicit conversion）

类型转换常发生在数字类型之间，发生场合有以下几处：

- 赋值

假设有这样的声明和赋值：

```
int a = 9;
double d;
d = a;
```

赋值语句中，整型变量 a 首先被转换成为一个临时的浮点变量，然后这个临时变量再将值赋给浮点变量 d，此后 d 拥有值 9.0，而 a 保持不变，仍然是 9。

上述从短类型向长类型的转换是由编译器自动完成，无需额外的编码。因此，这种转换称为隐式转换。

而反过来，当长类型向短类型赋值时，由于长类型变量保存的值可能大于短类型变量能容纳的最大值，因此会造成数据丢失。所以，这类操作应该避免。

- 混合运算

混合运算中也会出现隐式类型转换。例如：

```
int a = 10, b = 20;
double d = 30.0, r, x;
```

那么在表达式 r = a * d + b / 2.0 中，a 首先被转换成 double 类型的 10.0，然后再与 d 相乘，得到临时结果 300.0;第二步是 b 转换成 double 类型的 20.0,然后再与 2.0 相除,得到临时结果 10.0;最后两个临时结果相加的第三个临时结果 310.0，然后再赋给变量 r。

类似于 x = a / b – d / 20 的表达式很容易让人产生错觉。原本期望得到结果 0.5 – 1.5 = -1.0，但结果却是-1.5。产生这个差异的原因在于，第一个计算的表达式 a / b 由于其两个操作数都是整型，因此除法操作是整除操作，应该得到结果 0。最终整个表达式得到结果-1.5 就顺理成章了。所以，为了得到期望的结果，必须用到强制类型转换。

2．强制类型转换（type casting）

强制类型转换的意思是程序员用代码明确地告诉编译器这里有类型转换发生，其形式是：

（类型名）表达式

这里，（类型名）是 C 的一种运算符，称为**类型强制转换运算符**，其功能是将其后表达式的值强制转换成为类型名指定的类型。这里的类型名必须是一个简单类型，包括指针类型，不能是结构化类型，比如数组或是结构。

承接上面的例子，如果要得到期望的结果，可以用表达式

```
x = (double)a / b – d / 20
```

来得到正确结果-1.0。

因为强制类型转换运算符的优先级高于其它很多运算符的，所以要把表达式 r * x / -100.0 的结果强制转换成为 int 类型，表达式

```
(int)(r * x / -100.0)
```

是正确的，得到结果 3；而表达式

```
(int)r * x / -100.0
```

是错误的，它得到的结果是 310 * -1.0 / -100.0 = 3.1，还是一个 double 类型的数据。因此，最好是给被转换的表达式加上一对()，一劳永逸地解决问题。

与隐式转换规则一样，强制类型转换只影响原数据的一个临时拷贝，而对原数据没有任何影响。

强制类型转换一般发生在长类型向短类型转换的场合，或者是在程序员需要重新解释一个数据的含义的时候。在这种场合，程序员一定要非常清楚自己在干什么，因为数据的重新解释存在潜在的危险，特别是针对指针的时候。

【例 2-4】 复合运算和类型转换

```
//ex2-4.c
#include <stdio.h>

int main()
{
    int anInt = 9;
    double aDbl;
   int a = 10, b = 20;
   double d = 30.0, r1, r2, x1, x2;

    aDbl = anInt;
    anInt = 12.345;
    printf("anInt = %d, aDbl = %f\n", anInt,aDbl);

    r1 = a * d + b / 2.0;
    x1 = a / b – d / 20;
    x2 = (double)a / b – d / 30;
    printf("r1=%f, x1=%f, x2=%f\n", r1, x1, x2);

    r2 = r1;
   r1 = (int)(r1 * x2 / -100.0);
   r2 = (int)r2 * x2 / -100.0;
   printf("r1=%f, r2=%f\n", r1, r2);

    return 0;
}
```

例 2-4 的结果是：

```
xxxxxx
```

{这里要简单解释一下为什么要用浮点类型来接收强制整型转换结果。格式化输出函数 printf()是一个不那么智能的函数，它不能自动感知数值的类型，而必须在输出一个值之前明确地告诉它这个值是什么类型的。因此，如果最终的 r1、r2 都用整数格式输出，那么都会得到结果 3。而用浮点格式，我们就可以清楚地看到，r1 的小数部分被截掉了。}

2.7 编程实例

现在读者已经学习过了最基础的程序设计方法，所以在这一节里，将对学过的知识作一些总结，同时学习如何进行简单的算法设计。

【例 2-5】 编写一个程序，从键盘输入三个整数，输出最大的那个。

【解题思路】这里严格按照程序设计的三个过程来进行分析。

1. 问题分析

数的两两比较可以说是最简单不过的一件事。但因为没有能同时比较三个数的大小的操作，所以只能将这三个数进行两两比较。经过思考，可以制定如下的解题步骤，用术语讲，就是算法：

设有三个整数 x,y,z；

1）任取其中两个，这里不妨定为 x,y；

2）取一个中间数 t1 = max{x, y}；

3）再取一个中间数 t2 = max{z, t1}；

4）输出 t2；

5）结束。

可以证明，这个算法是正确的。下面就来看看如何实现 max{x,y}。两个数取最大的那个可以用简单的比较完成：

如果有 x > y，那么 max{x,y} = x；否则 max{x,y} = y

好了，现在已经得到了算法，同时其正确性也得到了证明，那么就可以进行下一步了。

2. 程序实现

一旦算法确立后，程序实现其实比较简单。只要将算法中的每一步翻译成对应的 C 语句。在此之前，不妨多做一步，将用数学形式描述的算法先转换成用普通语言描述的步骤：

1）开始；

2）用 scanf 输入三个数 x,y,z；

3）比较 x,y，用条件运算符可以完成，其最大结果存在中间变量 t1 中；

4）比较 z,t1，同样用条件运算符，其最大结果存在中间变量 t2 中；

5）用 printf 输出 t2；

6）结束。

根据上面的描述，可以写出类似与下面的代码：

```
//ex2-5.c
#include <stdio.h>

int main()
{
    int x, y, z;
    int t1, t2;

   printf(“Please input three integers:”);
   scanf(“%d%d%d”, &x, &y, &z);

    t1 = x > y ? x : y; //t1 是 x、y 中较大的那个
   t2 = z > t1 ? z : t1; //t2 是 t1、z 中较大的那个，也就是三个书中最大的那个

    printf(“The max number is %d.\n”, t2);

    return 0;
}
```

经过编译连接，可以顺利得到可执行代码。现在进入下一个阶段。

3. 测试

为了测试程序是否有错，需要精心的设计测试用例，特别是要准备一些边缘或极端条件下的数据。为此，准备了一下几组测试数据，每次运行使用一组：

1) 345 987 162

2) -345 -987 -162

3) 345 0 -162
4) 0 0 0
5) -100 -100 -100

可以看到，上几组数据都能得到正确结果：

Xxxxxx

但这并不意味这我们的程序完全正确。请读者试试这两组数据：

a b c
20897207l632092368437 30297348610968623519 7936281916386238612

将会产生如下的两种错误结果：

Xxxxxx

程序的输出显然不正确。出错的原因很简单：程序要求输入 3 个整数，但在这里却输入了三个字符，或者输入的整数太大，超过了 int 类型的表达范围。很明显，scanf 函数没有能够正确地处理输入数据不是要求的类型的情况（更确切地说，程序员没有重视 scanf 函数是否正确执行）。一般地，把这种情况成为“**异常**(exception)”。异常处理是一个高级话题，现在学到的知识还不足以覆盖这个内容。实际上，C 语言的异常处理能力很弱。这就对 C 程序员提出了很高的要求。

【例 2-6】有函数 $y=4x^5-6x^4+3x^3-x^2-5x+7$，自变量 x 的取值范围是实数。现在要求输入 x 的值，打印对应的 y 值。

【解题思路】

1. 问题分析

一般情况下，数学公式本身就是一种算法。不过，在将算法转换成程序代码的过程中可能存在一些困难。本例的难点在于如何写出正确的混合运算表达式。

已知函数是一个关于 x 的多项式，多项式中的每一项都是关于 x 的幂次。如果有一个乘方运算符，那么一切问题都会很简单。但遗憾的是，C 语言没有这样的运算符，所以，在幂次不高的情况下，都用变量自身的多次乘积来计算乘方。

根据上面的分析，可以用这样的 C 表达式来计算乘方：

x^5 = x * x * x * x * x

那么，整个函数表达式可以写成：

```
y =  4.0 * x * x * x * x * x
   - 6.0 * x * x * x * x
   + 3.0 * x * x * x
   - 1.0 * x * x
   - 5.0 * x
   + 7.0
```

表达式虽然正确，但是不是太过于冗长和低效？

通过仔细观察上面的表达式，可以发现：除了常量项，其它的各项都有公因子 x 可以提取。因此，通过层层提取，可以将表达式写成：

```
y = x * (x * (x * (x * (4.0 * x – 6.0) + 3.0) – 1.0) – 5.0) + 7.0
```

这就比原始的表达式要好的多。

2. 编程实现

根据前面的分析，编写代码如下：

```
//ex2-6.c
#include <stdio.h>

int main()
{
    double x, y;
```

```
    printf("Please input x:");
    scanf("%lf", &x);

    y = x * (x * (x * (x * (4.0 * x - 6.0) + 3.0) - 1.0) - 5.0) + 7.0;
    printf("y=%lf\n", y);

    return 0;
}
```

3．测试

运行程序，结果如下：

```
Xxxxxxx
```

然后与手工运算的结果对比，可以验证程序的正确性。同样地，应该准备一些边缘数据。

通过上面的两个例子，可以感觉到，虽然程序完成了任务，但总觉得有些地方不是特别合理。比如，当例 2-5 输入字符时，应该有办法进行判断；例 2-6 中，如果 x 的次数变得更高，或者阶乘的阶数更大，那么解决方案中的表达式会变得非常的复杂。想象一下一条表达式语句写了十几行的情形…这对每一个程序员来说都是一场噩梦。

其实解决的方法并不神秘。在后续的章节中，读者会学习分支和循环结构，通过它们，可以优化本节中的程序，增加它们的健壮性，并使它们能更加通用。

习题 2-9　请编写一个程序完成如下功能：从键盘输入摄氏温度值，输出对应的华氏温度值。摄氏温度到华氏温度的转换公式为：$Fahrenheit=\frac{9}{5}Celsius+32$

习题 2-10　请编写一些程序来实现习题 2-8 中给定的运算。要求 x,y,z 的值都从键盘输入。

2.8　C 程序的书写风格

学习至此，相信读者已经能够编写一些简单的 C 语言程序了。不过，笔者要问一下各位读者：在编写程序的时候，是否自觉地使用了某种风格来规范代码的书写呢？

一个 C 源程序就像是一篇文章，而文章的一个主要任务就是使阅读者能够容易看懂其内容。所以，规范化的书写是非常必要的。这里对初学者给出一些建议，希望能主动运用。

1．给标识符一个有意义的名字；
2．一条语句占一行；
3．左右花括号都独自占据一行；
4．用一个空行分开逻辑段落；
5．加入适当的注释；
6．采用缩进的书写格式，就像教材中的例子那样。这可能要多使用 tab 键；
7．在运算符的前后加上空格；
8．语法上遵循最新的 C 语言标准的规则或建议，例如 main()的返回类型是 int。

限于篇幅，其它的建议就不多讲了。最重要的是能在实践中运用。

需要提到的是，C 程序的书写风格实际上不属于语言标准，而更多的是约定俗成的行业规则。所以，笔者再给初学者提出这样的建议：在实践中养成属于自己的、但一定是良好的编程风格，并始终坚持这种风格。

2.9　解决方案

回到 2.1 节的问题上。现在读者已经有了关于变量的知识，因此可以为学生信息数据做出定义。

经过简单的分析，可以得出：

- 编号号是一个类似于 1001 的整数，因此可以用一个整型变量表示；

- 姓名形如“Peter”，因此可以用一个字符串变量表示；
- 性别有两种：Male 和 Female，可以简单地用字母’M’和’F’加以区分，因此可以用一个字符变量表示；
- 年级是一个整数，因此用一个整型变量表示；
- 得分是也是一个整数，因此可以用一个整型变量表示；
- 等级指明了用一个字母表示，因此可以用一个字符变量表示。

根据上述分析，可以写出如下的代码：

```
//solution1.c
#include <stdio.h>

//枚举类型及其常量声明，用于变量 gender 的取值范围
enum GENDER { MALE = ‘M’, FEMALE = ‘F’ };

//枚举类型及其常量声明，用于变量 level 的取值范围
enum LEVEL { levelA = ‘A’, levelB, levelC, levelD, levelE };

int main()
{
  //学生信息声明
  int   studentID;         //学生编号
  char name[30];           //姓名，用字符数组存放。姓名的最大长度不超过 30 个字符
  enum GENDER gender;      //性别，取值 MALE 和 FEMALE 之一
  int grade;               //年龄
  int score;               //得分
  enum LEVEL level;        //等级

  return 0;
}
```

也许读者看到这个程序可能感到有些疑问：声明 char name[30]又是什么意思？

关于这个问题，在现阶段读者们可以不必关心，而只需要关注如何用一组变量来表示学生的完整信息就可以了。在第 6 章和第 7 章中，将会解答读者的疑问。

本章小结

在本章中，详细地学习了 C 的最基本的语法成分：

- 简单类型：在程序中用不同的类型来表达不同大小和性质的数据；
- 常量变量：用于表达预定义的常数，或者存储程序的中间结果，或者在程序间传递信息。注意：原则上不同类型的数据不能互相赋值，但相容原则例外；
- 表达式和运算符：完成指定的运算。需要注意的是，C 的每一个表达式本身也是有值的，另外还需要注意运算符的优先级和结合性。

紧接着通过一个实例来加深印象，最后给出了书写 C 程序的一些建议。在后续的学习中，读者将看到这些基本知识的运用情况。

这里给出一个建议：在编写程序的时候，请注意语法细节，尽量避免出现难于察觉的逻辑错误，同时养成良好的个人风格。这会对读者参与复杂系统工程的开发带来好处。