第3章　数据类型（Data Types）

什么是数据类型？数据类型是指：

（1）一定的数据在计算机内部的表示方式；

（2）该数据所表示的值的集合；

（3）在该数据上的一系列操作。

在数学上，专家们典型地用代数群论对数据类型进行研究（☞参考文献[6]CH5.4）。在计算机语言中，将数据用一定的数据类型来描述是为了将一系列相同性质的数据归类，统一值域和规范操作，以便这些数据在描述问题、数据抽象（☞CH11.1.2）中得到更好的运用，从而通过数学和计算机的手段解决问题。

对于要解决的具体问题，一般的做法是将问题数量化，描述成一定数据类型下的实体和相关的操作，通过语言的编译器对其进行识别，最后让计算机执行操作，运行获得求解结果。

以数据类型规定数据的描述和行为的编程手段，有利于数据的逻辑描述和正确性检查，有利于数据操作的高质和高效。

在 C++中，数据类型不仅规定了某一数据是整数、浮点数或者自定义的类型名称，而且规定了数据的组织形式以及操作方法。数据类型是程序设计中描述数据的工具，对数据类型的选取或规定形式，直接决定了编程中解决问题的具体方法。

C++中的数据类型，有语言既定的内部数据类型（inner types），也有程序员自定义的外部数据类型。其中，内部数据类型有：

- **整数类型(int)；**
- **字符类型(char)；**
- **布尔类型(bool)；**
- **单精度浮点(float)；**
- **双精度浮点(double)。**

还可以通过数组、指针、引用等定义基于上面这些数据类型以及其他外部数据类型的变异类型。例如：

- 整型数组(int[]);
- 浮点引用(double&);
- 字符指针(char*)。

内部数据类型及其变异构成了C++的基本数据类型。注意，有些书对于内部数据类型（inner types）和基本数据类型（base types）是不加区分的。内部数据类型是指语言本身具有的数据类型，主要指整数类型和其相关的衍生类型，以及浮点类型和其相关的衍生类型。基本数据类型主要是指编程中可自由运用的，对内部数据类型做适当“变形”所构成的数据类型。

程序员自定义的数据类型主要是指用class关键字（☞CH8）构造的数据类型。除此之外，用enum、union、struct关键字也能定义单纯空间意义上的数据类型。

要解决具体问题，必须首先学会用数据类型描述问题中的具体事物。世界上的问题形形色色，仅用语言内部的数据类型描述事物是远远不够的，还必须借助于语言所提供的数据类型描述机制自定义数据类型。要自定义数据类型，就必须先了解语言的内部和基本的数据类型。因为自定义数据类型是以内部数据类型为基础的。

3.1 整型（int Types）

使用整数是人们描述自然现象的最基本的数学方式，因此，以解决人类计算问题为目标的计算机语言提供了整数数据类型，即整型。它规定了整数的表示形式、整数的运算（操作），以及整数在计算机中的表示范围。

❑ 3.1.1 二进制补码（Binary Complement）

通常的计算机语言在计算机内部都是以二进制补码形式表示整数的。因为二进制补码用来表示整数具有高度的一致性，并且统一了加减法的意义，简化了乘除法运算，甚至直接简化了计算机的硬件结构。

将十进制正整数转换成二进制补码形式的整数的转换方法是采用“除2取余法”，即对被转换的十进制整数除以2，取其余数，并将商再除以2，再取余数，直到商为0。每次除下来的余数按先后构成了从低位到高位的二进制整数。例如：

$35 = 100011_{(2)}$

转换的具体步骤为：

```
35÷2 = 17……1
17÷2 = 8……1
8÷2 = 4……0
4÷2 = 2……0
2÷2 = 1……0
1÷2 = 0……1
```

由于计算机内部表示数的字节单位都是定长的，以 2 的幂次展开，或者 8 位，或者 16 位，或者 32 位……于是，一个二进制数用计算机表示时，位数不足 2 的幂次时，高位上要补足若干个 0。例如，35 以 8 位长和 16 位长在计算机内部表示时的二进制数分别为：

```
00100011
0000000000100011
```

两个十进制整数相加在计算机中是做二进制数加法运算的，例如：

```
35+12 = 00100011+00001100
      = 00101111
      = 101111(2)
      = 47
```

一个十进制负整数，表示成二进制补码形式的整数时，该负整数的对应正整数先转换成二进制数，然后“取补”，规则是“取反加一”，例如，用 8 位长度的二进制形式表示：

```
-15 = -1111(2)
    = -00001111
    = 11110001
```

用二进制补码表示的数中，以最高位是否为 0 判断该数是否为正数。例如：

```
01111110-------正数
10001101-------负数
```

因此，一定长度的二进制补码中总是有一半是正数，一半是负数。例如，8 位二进制补码中有 128 个数最高位为 0，即 128 个正数（其中含 0），另外 128 个数为负数；所能表示的最大正整数是 127，即 01111111，最小非负数是 0，即 00000000；所能表示的最接近于 0 的负整数是–1，即 11111111；绝对值最大的负整数是–128，即 10000000。

由于 0 取补后还是 0，其他数取补后从正数转为负数，或从负数转为正数，所以体现了二进制补码表示形式的一致性。它为计算机进行加减运算带来了设计上的方便。

在二进制补码的运算中，减法相当于取补后相加，如果相加后在最高位有进位，则简单地弃之了事。因此，二进制补码运算在计算机中没有减法。例如：

```
3-5 = 00000011-00000101
    = 00000011+11111011
    = 11111110
    = -00000010(2)
    = -2
```

在二进制补码中，有一种很有用的移位操作，8 位二进制码的左移 1 位操作就是将最高位挤出，最低位补 0。例如，6(00000110)左移 1 位后得到 12(00001100)，即相当于 6 乘 2 等于 12。

由于二进制整数左移 1 位相当于做乘 2 运算，所以，二进制补码的乘法在具体的操作

中都分解成了一系列的左移和加法操作。例如：

```
3×5 = 00000011×00000101
    = 00000011×00000001+00000011×00000100
    = 00000011 左移 0 位 + 00000011 左移 2 位
    = 00000011+00001100
    = 00001111
    = 15
```

同理，二进制整数做除以 2 运算相当于右移 1 位。所以，二进制补码的除法运算在计算机中都分解成了一系列的左、右移和加法操作。例如：

```
13÷3 = 00001101÷00000011
     = (00001100+00000001)÷00000011
     = 00001100÷00000011+00000001÷00000011
     = 00000100 余 00000001
     = 4
```

由于整数除法中结果没有小数，所以其除法也就是抛弃余数的整除法。读者必须明白实际实现的乘除法操作设计比这里描述的要复杂（☞参考文献[7]CH8）。

❑ 3.1.2 整型数表示范围（int Range）

整型的设计有多种形式，按表示的长度分，有 8 位、16 位和 32 位，以后还有 64 位，大型机还有 128 位，随着计算机的发展，整型数的位长也在增加。每一种长度都分为有符号（signed）和无符号（unsigned）两种，并且总是指定一种为默认类型，见表 3-1。

表 3-1 整型分类表

类　型	有符号形式	无符号形式	默　认
8 位	signed char	unsigned char	signed char
16 位	signed short int	unsigned short int	signed short int
32 位	signed int	unsigned int	signed int
32 位	signed long int	unsigned long int	signed long int

例如：

```
unsigned int x = 23;
int y = -67;              //等价于 signed int y = -67;
unsigned int z = -43;     //表示方式有错
```

值得注意的是，默认类型并不属于 C++标准，而是编译器的设定，有些 C++编译器的 char 默认为 unsigned char，而且其长度为 16 位。不过，目前流行的 C++编译器都是按表 3-1 所示默认的。不管怎样，所有的编译器均应满足 C++标准所规定的整数长度关系式：

```
char ≤ short int ≤ int ≤ long int
```

表 3-2 是目前流行的 32 位编译器的各种整数类型表示范围一览表。

表 3-2　整型数表示范围

类　型	字节数	位数	表示范围		解　释
			下　限	上　限	
char	1	8	–128	127	-2^7~(2^7–1)
signed char	1	8	–128	127	-2^7~(2^7–1)
unsigned char	1	8	0	255	0~(2^8–1)
short int	2	16	–32768	32767	-2^{15}~(2^{15}–1)
signed short int	2	16	–32768	32767	-2^{15}~(2^{15}–1)
unsigned short int	2	16	0	65535	0~(2^{16}–1)
int	4	32	–2147483648	2147483647	-2^{31}~(2^{31}–1)
signed int	4	32	–2147483648	2147483647	-2^{31}~(2^{31}–1)
unsigned int	4	32	0	4294967295	0~(2^{32}–1)
long int	4	32	–2147483648	2147483647	-2^{31}~(2^{31}–1)
signed long int	4	32	–2147483648	2147483647	-2^{31}~(2^{31}–1)
unsigned long int	4	32	0	4294967295	0~(2^{32}–1)

❑ 3.1.3　编译器与整型长度（Compiler & int Length）

C++编译器在不同的计算机硬件上的表现是不同的。目前计算机主板上的主流 CPU 是 64 位的，而目前的 C++编译器版本则仍然是 32 位的，软件相对于硬件总是滞后的。所谓 32 位编译器，是指它能将程序源代码编译成最高为 32 位的 CPU 指令系统代码，或者更加直接地说，默认 int 类型的长度是 32 位。C++编译器过去曾是 16 位的，今天是 32 位的，那么自然，明天将是 64 位的。

32 位 C++编译器并非一定只能编译那些 32 位 CPU 指令系统的代码。为了兼容运行环境，32 位 C++编译器可以将代码编译成较低级别的指令系统。32 位 C++编译器还可以表示 64 位整型，C++ 11 标准规定 64 位整型名称为 long long。

例如，在 32 位编译器中，若将代码编译成 16 位机器指令系统，则：

```
int a = 327777;    //错，16 位机器指令表示的有符号整数最大只能为 32767
```

就不正确，而如将代码编译成 32 位机器指令系统，则上述语句就是合理的。为了使编写的程序具有可移植性，在各种机器指令环境下，或者说在各种操作系统环境下运行，都能得到唯一的结果，必须分辨编译器。上面那个定义语句若在低版本编译器编译，就应该写为：

```
long int a = 327777;
```

3.1.4 整数字面值（Integer Literals）

整数用具体的数值表示就是整数字面值。整数字面值遵循文法表示（☞附录 A.5）。

整数可以用十进制、八进制和十六进制数表示。编程时，用非 0 数字开头的数字序列表示十进制数，0 开头的数字序列表示八进制数，0X 或 0x 开头的数字和 ABCDEFabcdef 序列表示十六进制数。例如：

```
int a = 23;
long int b = 02345;
unsigned int c = 0x79fa;
```

整数字面值可以区分类型（长度），如果像上面这样朴素的整数字面值，则默认为 int 型整数，即 signed int 型；如果要表示 unsigned int 或者 long int，则可以在整数字面值后面加 U 或 L，大小写都可以。例如：

```
b = 02345L;          //long
c = 235u + 123u;     //unsigned
```

文法就是语法，C++语言都是由语法规定的。可以参考附录 A，以了解怎样学习文法。

语言的描述要受到实现的限制，即受到计算机发展技术的限制，受到编译器的限制。例如，文法中规定的整数字面值是非 0 数字开头的数字序列。但对序列的长度没有具体说明，其数字序列是递归定义的形式。

事实上，下面超过整数范围描述的字面值在各个计算机中有不同的解释：

```
int a = 12345678901234567890012345678900;
```

存储在 a 空间的值究竟是多少呢？C++标准告诉我们，当整数的表示在整型表示范围内时，任何编译器的理解是一致的，但当其超过了所表示的范围时，不同的编译器有不同的处理方式，因而，上述 a 的值是不可预料的。

例如，在 VC 编译器中，该语句报错，而在 BCB 编译器中，编译能通过，但输出的 a 是一个莫名其妙的数。而对 20 位长度的整数字面值，在 VC 中居然也通过了编译，但与 BCB 一样，其值是荒谬的。那是因为各个编译器对整数字面值（更确切地说，是 C++语言的词法单位）长度限制不同，超过一定的长度，就是错误；没有超过规定的长度，但超过了表示范围，虽合法但不合理。

3.1.5 整数算术运算（Integer Arithmetic Operations）

整数可以进行+、–、*、/、%、<<、>>、<<=、>>=、!、^、<、<=、>、>=、==、^=、&、|、&=、|=、&&、||、&&=、||=、!=、=、+=、–=、*=、/=、%=、++、––、,、?:等操作。其中有些是在整数之间做比较的，有些是在两个整数上面做算术运算的，有些是做位操作（☞CH4.5）的，有些是做赋值操作的。

+、–、*、/、%这五种操作是整数的算术运算。其中，“/”是整除运算，“%”是取余运算。例如，11%5=1。规定除数不能为 0，否则将导致运行错误。值得一提的是，余

数的正负性决定于被除数的正负性，这与整除“/”操作所得结果的“负负得正”不同。例如：

```
11/(-5) = -2        -11/(-5) = 2        //结果符号遵循负负得正原理
11%(-5) = 1         -11%(-5) = -1
```

3.2 整数子类（int Subtypes）

❑ 3.2.1 字符型（char Type）

ASCII 码有 128 个字符，其中，ASCII 值（即字符的整数值）0～31 和 127 为不可见字符。不可见字符也称控制字符。直接表示可见的 ASCII 字符（即字符的字面值）是用单引号括起来的单个字符。例如，'a'，'x'，'?'，'$'等。除了这种形式的字面值外，C++还允许使用一种特殊形式的字面值，即以“\”打头的格式字符，称为转义字符（escape character）。

经常用的不可见字符就是用一个转义符后跟一个专门的字符表示。例如，换行符用 '\n' 表示。有些符号虽可见，但表示上有时与语法发生冲突，也用转义符委婉表示。例如，单引号字符表示为'\''，不能表示为'''；字串"I say "OK""应写为"I say \"OK\""。可见字符在知道其 ASCII 值的前提下也可以用转义字符的形式表示，例如，'A'的 ASCII 码为 65，也可用转义字符 '\101' 或 '\x41' 表示，即表示为转义符后跟去掉前导 0 的八进制或十六进制数。表 3-3 列出了一些转义字符。

表 3-3 C++转义字符

字符形式	整数值	代表符号	字符形式	整数值	代表符号
\a	0x07	响铃 bell	\"	0x22	双引号
\b	0x08	退格 backspace	\'	0x27	单引号
\t	0x09	水平制表符 HT	\?	0x3F	问号
\n	0x0A	换行 return	\\	0x5C	反斜杠字符\
\v	0x0B	垂直制表符 VT	\ddd	0ddd	1~3 位八进制数
\r	0x0D	回车	\xhh	0xhh	1~2 位十六进制数

字符型是针对处理 ASCII 字符而设的。字符型在表示方式和操作上与整数吻合，在表示范围上是整数的子集。它由一字节（8bit）组成，所以只能表示 256 个状态值。由于 ASCII 码有 128 个字符，所以可以用 signed char（即 char）中的所有正数表示所有 ASCII 码值，而负数表示非正常状态，以示区别。由于它可以看作为整数的子集，所以其运算可以参与到整型数中去，只要不超过其范围。例如：

```
char a = 31;
int b = a + '\a';              //31+65=96
```

然而它与整数毕竟还是有区别的，最大的区别是在输出方式上，字符型的输出不是整

数，而是该整数所代表的 ASCII 码字符。例如：

```
int a = 65;
char b = 65;
cout<<a<<" "<<b<<end;   //虽然其值都为 65，但其结果为 65 A
```

值得注意的是，有些地方和机器环境用的不是 ASCII 码。ASCII 码并不是终极标准。因此，为了代码可移植，不要用数字对字符变量进行赋值，应以字符字面量对字符变量进行赋值。例如，对于程序 f0206.cpp，请不要像下面这样编程：

```
//=====================================
//f0301.cpp
//请用字符不用 ASCII 码
//=====================================
#include<iostream>
using namespace std;
//-------------------------------------
int main(){
  for(int i=1; i<=10; ++i){
    cout<<string(10-i,' ');
    for(char ch='A'; ch<=64+2*i; ++ch)    //64+2*i 应写成'A'+2*i-1;
      cout<<ch;
    cout<<endl;
  }
}//=====================================
```

❑ 3.2.2 枚举型（enum Type）

枚举型是对整数区间的自定义类型，用户须为区间中的值取名。例如：

```
enum Week{ Mon, Tue, Wed, Thu, Fri, Sat, Sun };
```

因此，枚举 Week 是一个类型。

定义枚举时，其大括号中的名称代表某个整数值，默认时，第一个名称对应整数 0，第二个对应 1，以此类推。因此，Week 中，Mon=0，Tue=1，…，Sun=6。也可以人为规定。例如：

```
enum Color{ Red=5, Green, Yellow, Blue=20, Orange };
```

则表示 Green 对应整数值 6，Yellow 对应 7，Orange 对应 21。

枚举的整数区间为包含头尾整数取值的最小 2 的幂次值。因此，对 Week 取值范围[0，6]来说，$2^0-1 \leqslant 0$，$6 \leqslant 2^3-1$，所以，Week 的整数区间为$[2^0-1, 2^3-1]$。对 Color 取值范围[4，31]来说，$2^2-1 \leqslant 4$，$31 \leqslant 2^5-1$ 所以，Color 的整数区间为$[2^2-1, 2^5-1]$。枚举变量在该整数范围内取值和运算都是合理的。例如：

```
Week day = 7;
Color color = Orange+day;     //28
```

然而，对于超过枚举范围的赋值行为就是不确定的了。定义枚举时的最大取值不能超过整型的最大值。

枚举定义中大括号中的名字称为枚举符，全体枚举符作为整型数的一个子集，可以直接参加整数所应该享受到的运算。因此，枚举符可以脱离枚举变量的定义而使用。例如：

```
enum Week{ Mon, Tue, Wed, Thu, Fri, Sat, Sun };
if(a==Mon) cout<<"Mon\n";
```

枚举符一旦定义则不能改变。所以它常常代替整数常量使用。这才是语言中设计枚举的真实意图。有时候甚至比整数常量还管用，因为在进入函数调用或其他模块时，常量需要初始化，而枚举却是一种类型，无须定义其实体，便可以直接使用其枚举符。

❑ 3.2.3 布尔型（bool Type）

整数 1 和 0 两个值构成了布尔型的表示范围。相当于：

```
enum bool{ false, true };
```

只有两个整数的类型，其范围偏窄了一些，但是用它表示逻辑的 true 和 false，却可以表达千千万万的真假命题。C++表达式值的大小比较、条件的真假判断，还有一切逻辑推理（运算）的结论都可以用布尔型值表示（☞CH4.4）。

用任何非 0 整数给布尔型变量赋值时，其值都为 1，甚至非整数的其他类型，只要非 0，其值也是 1。因此：

```
bool a = 3;     //a 为 true
bool b = 1;     //b 为 true
bool c = a+b;   //c 为 true (1+1=2，2 为非 0，即 1,其间不做模 2 运算)
bool d = a-b;   //d 为 false (不是 3-1，而是 1-1)
```

布尔型的输出形式可以选择，默认为整数 1 和 0，如果要输出 true 和 false，则可用输出控制符：

```
cout<<boolalpha<<d<<endl;   //输出结果为: false
```

3.3 浮点型（float Type）

现实世界是丰富多彩的，用数学方法描述问题，仅用整数，而且是用计算机所能包含的这么一点局限的整数表示范围实在是太局促了。因此在计算机的基本设计中，还包括浮点数（floating-point number）。浮点数因为内部表示特殊，所以其操作不同于整数，能够表示的大小范围比同样大小空间表示的整数大很多，在两个连续的整数之间也能表示许多较为精细的数值。但是，有得必有失，浮点数的有效位数就不如整数了，即无法表示在若干位数之后的细部。不管怎么说，它还是与整数在对现实问题的抽象描述中互补。

❑ 3.3.1 浮点数表示（Floating-Point Number Representation）

1 十进制浮点数

在十进制数中，通常一个浮点数可以用科学记数法表示。例如，–306.5 可以写成 -0.3065×10^3。其中，–是符号，指数 3 为阶或称阶码，0.3065 是小数部分，其左右端非 0 数字包起来的最长的数字序列称为有效值（significance），这里的有效值是 3065。小数部分也称为尾数，显然 3065 也是尾数（mantissa）。之所以称之为浮点数，是因为它也可表示成 -3.065×10^2，以及 -0.03065×10^4 等，小数点可以左右“浮动”。但不管小数点怎么移动，有效值不变，都是 3065，不过其尾数会随着小数点的移动而变化，或为 065，或为 03065。于是，如果两个浮点数要相加，就先要通过小数点的左右浮动，将阶码对齐，然后进行尾数相加。例如：

$$
\begin{aligned}
&0.0365\times10^3 + 6.78\times10^2\\
=\ &0.365\times10^2 + 6.78\times10^2\\
=\ &7.145\times10^2
\end{aligned}
$$

为了使有效值和尾数能够统一，在数值表示上具有唯一性，在空间表达上更具效率，即以一定长度的尾数表示尽可能多的有效值，有必要将所有浮点数规格化（normalization），即浮点数通过调整阶码，写成小数点前不含有效数字，小数点后第 1 位由非 0 数字表示。例如，–306.5 规格化为 -0.3065×10^3。

2 十进制浮点数转换成二进制浮点数

在计算机内部，浮点数都是以二进制表示的，所以，对于十进制浮点数，要先转换为二进制浮点数。以手工方式操作，转换分两步：整数部分的转换，采用“除 2 取余法”（☞CH3.1.1）；小数部分的转换，采用“乘 2 取整法”，即对被转换的十进制小数乘以 2，取其整数部分（0 或 1）作为二进制小数部分，然后取其小数部分，再乘以 2，又取其整数部分作为二进制小数部分，然后又取其小数部分，再乘以 2，直到小数部分为 0 或者已经取到了足够位数。每次取的整数部分，按先后次序，构成了二进制小数从高位到低位的数字排列。例如：

$$0.8125 = 0.1101_{(2)}$$

转换的具体步骤为：

$$
\begin{aligned}
&0.8125\times2 = 1.625 \quad 0.1\\
&0.6250\times2 = 1.250 \quad 0.11\\
&0.2500\times2 = 0.500 \quad 0.110\\
&0.5000\times2 = 1.000 \quad 0.1101
\end{aligned}
$$

有时候，在转换中，二进制小数的某些位会周而复始地重复，以致无穷。由于计算机的表示是有限的，所以在计算机内，只能截取到某个精度，而在文字描述时，对重复的部分，其两端数字各用一个着重号表示该段数字的重复。例如：

$$0.6 = 0.100110011001\cdots_{(2)} = 0.\dot{1}00\dot{1}_{(2)}$$

转换的具体步骤为：

```
0.6×2     =1.2        0.1
0.2×2     =0.4        0.10
0.4×2     =0.8        0.100
0.8×2     =1.6        0.1001
```

下一步是 0.6×2，又回到了开始转换的第一行。这说明 $0.6=0.100110011001\cdots_{(2)}$是个有无穷循环小数位的二进制浮点数（意味着在计算机内部无法精确表示）。

3 二进制浮点数的尾数及规格化

一旦十进制浮点数转换成二进制浮点数后，就要像十进制数那样，对二进制数规格化，以便用计算机表示。二进制浮点数规格化是通过调整浮点数的阶码使得该数的有效值在 1 与 2 之间，即二进制浮点数的整数部分为 1。例如：

$$0.8125 = 0.1101_{(2)} = 1.101\times 2^{-1}$$

在计算机内部，浮点数是以国际标准 IEEE 754 的形式表示的。该标准将二进制浮点数分成三段，第一段是符号段，它总是占 1 位；第二段是阶码段；第三段是尾数段。例如，在 32 位浮点数（对应 C++的 float 类型）中，符号段占 1 位，阶码段占 8 位，尾数段占 23 位。在 64 位浮点数（对应 C++中的 double 类型）中，阶码段占 11 位，尾数段占 52 位，见图 3-1。

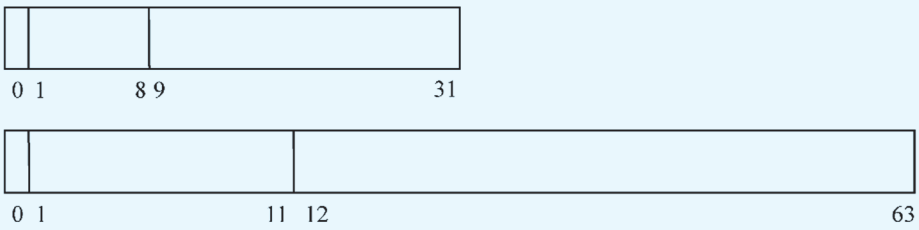

图 3-1　二进制浮点数格式

如果二进制浮点数像十进制浮点数那样规格化，即要求小数点后第 1 位非 0，那么，其小数点后的第 1 位的非 0 值只能为 1，它构成了浮点数尾数的一部分，在计算机内部占去了表示精度的宝贵的 1 位。由于这样表示的结果使得小数第 1 位总是 1，何不将该位挪前，增加 1 位有效位呢？因此，二进制浮点数的规格化不同于十进制浮点数。例如，在 32 位浮点数表示中：

$$0.6 = 0.1001_{(2)}$$
$$= 1.0011,0011,0011,0011,0011,010\times 2^{-1}$$

它比老老实实表示的 23 位尾数 0.1001,1001,1001,1001,1001,101 多了一位精度。而在具体实现中，抹掉规格化的 1，将尾数写入二进制浮点数；又将二进制浮点数的尾数取出，在头上添上 1，重新构成二进制规格化数。因此，23 位尾数加上省略的一位，其精度或有效位却是 24 位。

4 二进制浮点数的阶码

二进制尾数的规格化表示提高了一位精度，但是牺牲了浮点数 0 的常规表示，因为规格化要求小数点前面一位必须是 1，所以，即使机内码整个尾数部分都为 0，其浮点数的有效值也为 1。但是，0 参与浮点数运算是必不可少的，因此就有 0 在浮点数中如何表示的问题。

标准 32 位浮点数（单精度）规定（64 位浮点数，即双精度浮点数，可以此类推），浮点数的阶码为 8 位，阶值在–126～127；另外两个值，–127 和–128 用来表示特殊浮点数。其中–127 表示阶码为–126 的非规格化数，非规格化数就是不做规格化的二进制浮点数，也就是说，有效值不省略小数点前面的 1，只用 23 位尾数。因为 0 乘上任何 2 的阶数都为 0，所以，当非规格化数的尾数全 0 时，该数就是浮点数 0（前提是阶码为–127）。另外，当阶码为–128 时，表示的数是非法操作的数（称其为 NaN，即 Not a Number 之意）或者±∞，这种数一般起因于除 0 操作、0 的 0 次方，或者运算结果超过了浮点数所能表示的范围。C++编译器的策略一般是，运行中碰到这种数时，激活一个异常或者唤醒一个溢出中断而停止运行。

为了使浮点数 0 与整数 0 统一，即位码全 0 表示 0。标准单精度浮点数对所有规格化和非规格化二进制数阶码一律做+127 的偏移操作。而在取出该浮点数时，再做一个–127 的逆操作。例如：

```
35.6 = 100011.1001̇1̇(2)
     = 1.00011100110011001100110×2^5
     = 0,10000100,00011100110011001100110   //机内表示
```

其中用两个逗号将浮点数分隔成三段，第一段 0 表示该数为正数，第二段 10000100 为指数 5 加上 127 所得，第三段是规格化后的 23 位尾数。

5 浮点数字面值及内部表示

浮点数字面值的书写格式在 C++语言的文法中有规范的表示（☞附录 A.6）。浮点数既可以表示为定点方式（非指数方式），例如 35.623，也可以表示成科学记数法（指数方式），例如 0.35623e+02，意即 0.35623 乘上 10 的 2 次方。直接写出的浮点数字面值，默认为 double 型，如果要表示成 float 型，则要在浮点数后面加上字母 F 或 f，如果要表示成 long double 型，则要在浮点数后面加上字母 L 或 l。例如：

```
float f1 = 19.2f;
float f2 = 0.192e+02;           //将 double 数转换为 float
double d1 = 19.2;
double d2 = 0.192e+02f;         //将 float 数转换为 double
long double ld1 = 19.2L;
long double ld2 = 0.192e+02;    //将 double 数转换为 long double
```

三种不同精度的浮点数表示同一个十进制浮点数 19.2，其位码分别为：

```
单精度: 0,10000011,00110011001100110011010
双精度:0,10000000011,0011001100110011001100110011001100110011001100110011
长双精度: 0,100000000000011,1001100110011001100110011001100110011001100110011001100
        11001100110011010
```

注意，单精度浮点数尾数的最后，有一个进位问题。因为舍去的是 1，按四舍五入规则，要进位，所以最后三位的 001 加 1 变成了 010。双精度浮点数尾数的最后，同样有一个仅为问题。因为舍去的是 0，所以舍之不进位。于是就造成了 19.2 这个小数，如果赋给 float 变量，则大于 19.2，如果赋给 double 变量，则小于 19.2。

在理解了计算机内部是如何表示浮点数之后，我们便可以用程序的方法查看单精度浮点数的二进制位码了（双精度和长双精度数类推），见下列程序代码：

```
//=====================================
//f0302.cpp
//浮点数的位码
//=====================================
#include<iostream>
using namespace std;
int main(){
  float f=19.2F;
  int* pa = (int*)&f;
  for(int i=31; i>=0; i--)
    cout<<(*pa>>i & 1)<<(i==31||i==23 ? "-":"");
  cout<<"\n";
}//=====================================
```

窥探浮点数的计算机内部表示，没有专门的数据类型表示其操作，这里采用对存储空间逐位判断的方法查看位码，用到了两种超前技术：一种是指针技术（☞CH3.7）；另一种是位操作（☞CH4.5）。

❑ 3.3.2 浮点型表示范围（Floating-PointType Ranges）

对于单精度浮点数来说，由于阶码有 8 位，可以表示正负指数。当尾数取到全 1 再加上小数点前面的 1，阶码取到最大为 127 时，浮点数取到正负数的最大值：

$$\pm 1.11111111111111111111111 \times 2^{127}$$
$$= \pm(2 - 2^{-23}) \times 2^{127}$$
$$\approx \pm 2^{128}$$
$$\approx \pm 3.4 \times 10^{38}$$

那么，单精度浮点数表示的最接近于 0 的实数是什么呢？其精度又是多少呢？

对于规格化数，当尾数全为 0，阶码为–126 时，最接近于 0 的数，其值为 $2^{-126} \approx \pm 1.2 \times 10^{-38}$。

其有效位数由尾数表示，有 24 位，即精度有二进制的 24 位，又因为 $2^{24} \approx 1.7 \times 10^{7}$，所以精度相当于十进制的 7 位。也就是说，十进制数，阶码在±38 之内，有效位数在 7 之内，

其单精度浮点数都能精确表示。

对于非规格化数，即阶码为–127，表示 2^{-126}，当尾数为 0.00000000000000000000001 时，最接近于 0，其值为 $2^{-23-126}=2^{-149}\approx1.4\times10^{-45}$，但是该数的精度只有 1 位！因此，单精度浮点数最接近于 0 的数由非规格化的特殊浮点数所表示，约为$\pm1.4\times10^{-45}$。非规格化数只能象征性地表示最接近于 0 的数，因为其精度表示差。

到了 long double，浮点标准不再为那 1 位精度优化，不分规格化与否，所以表示范围直接由 $0.11111\cdots1\times2^{16383}\approx5.9\times10^{4931}$ 得到，最近 0 数直接由 $0.0000\cdots1\times2^{-16383}\approx9.1\times10^{-4952}$ 得到。表 3-4 是 C++中浮点类型的一些说明。

表 3-4　浮点类型说明

类别 \ 类型	float	double	long double
说明	单精度	双精度	长双精度
位数	32 位	64 位	80 位
长度	4 字节	8 字节	10 字节
表示范围	$\pm3.4\times10^{38}$	$\pm1.8\times10^{308}$	$\pm5.9\times10^{4931}$
规格化近 0 数（保证精度）	$\pm1.2\times10^{-38}$	$\pm2.2\times10^{-308}$	$\pm9.1\times10^{-4952}$
非规格化近 0 数	$\pm1.4\times10^{-45}$	$\pm4.9\times10^{-324}$	
阶码	8 位	11 位	15 位
尾数	23 位	52 位	64 位
二进制有效位数	24 位	53 位	64 位
十进制有效位数	7 位	15 位	19 位
规格化数阶值范围	–126～127	–1022～1023	–16383～16383
非规格化数阶值	–127	–1023	
NaN 阶值	–128	–1024	–16384

3.4　C-串与 string（C-strings & string）

❑ 3.4.1　C-串（C-strings）

在 C++中，有两种字符串，一种是从 C 沿袭过来的，称为 C-字符串，简称 C-串。C-串是以一个全 0 位（整数 0）字节作为结束符的字符序列。该全 0 字节既是 8 位的整数 0，也是 ASCII 码的 0。C-串还称为 ASCIIZ 串（即 ASCII 字符序列加上尾巴 Zero）。

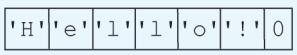

图 3-2　“Hello!”的存储形式

C-串也是字符串字面值，其格式为双引号括起来的字符序列。例如，我们前面用到的“Hello!”。它在空间中的存储形式为图 3-2 所示。

很显然，C-串的空间长度为字符串长度加 1。如果要将 C-串放入字符数组，则元素个数非比字符数多 1 不可。例如：

```
char buffer[7]="Hello!";     //若为 char buffer[6]="Hello!";则为错误!
```

我们知道，字符字面值的类型为 char，那么 C-串又是什么类型呢？

C-串的类型为 char*，说得更精确一点，是 const char*。事实上，所有的字面值类型都是 const 的。char*称为字符指针，它与字符数组虽然类型不同，但操作上是一样的，都表示 C-串的起始地址。

❑ 3.4.2 字符指针与字符数组（char Pointers & char Arrays）

指针是表示内存空间位置的存储实体（☞CH3.7）。字符指针就是所指向的空间位置上的值，当作字符操作的类型。例如：

```
char* str="Hello";
cout<<*str<<endl;   //显示 H
cout<<str<<endl;    //显示 Hello
```

str 是字符指针。*str 是字符指针的间接引用。即，若 str 指向“Hello”的首地址，则*str 表示该地址代表的空间上的值——'H'。

输出字符指针就是输出 C-串。所以输出 str 时，便从'H'字符的地址开始，输出所有字符直到遇到 0。输出字符指针的间接引用，就是输出单个字符。所以输出*str 时，便输出 str 所指向的字符'H'，见图 3-3。

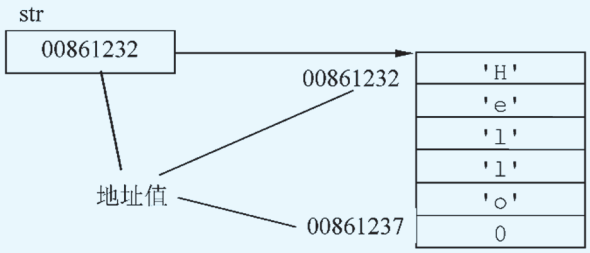

图 3-3 *str 指针示意图

由于 C-串是字符指针，所以比较两个 C-串就是比较两个字符指针，即存储地址的比较。比较两个相同 C-串的时候，会因空间存储位置的不同而不同。而且，分别存储两个相同 C-串的字符数组，其数组比较也会是不相同的。例如：

```
//=====================================
//f0303.cpp
//C-串比较的错误方式
//=====================================
#include<iostream>
using namespace std;
//-------------------------------------
int main(){
  cout<<("join"=="join" ? "" : "not ")<<"equal\n";
```

```
  char* str1="good";
  char* str2="good";
  cout<<(str1==str2 ? "" : "not ")<<"equal\n";
  char buffer1[6]="Hello";
  char buffer2[6]="Hello";
  cout<<(buffer1==buffer2 ? "" : "not ")<<"equal\n";
}//====================================
```

```
E:\ch03>f0303↙
not equal
not equal
not equal
```

还有 C-串的复制问题：

```
char* str1="Hello";
char* str2=str1;  //意味着 str1 与 str2 共享"Hello"空间
```

数组复制干脆告禁：

```
char a1[6]="Hello";
char a2[6]=a1;   //错: 数组是不能复制的
```

为了比较 C-串的字典序大小，在 C 库函数（C++头文件 cstring 或 C 头文件 string.h）中，专门设计了 C-串的比较函数 strcmp。因而 C 库函数为其设计了 strcpy 函数。总之，C 库函数设计了一系列的 C-串库函数，解决 C-串的赋值、复制、修改、比较、连接等问题。例如：

```
//====================================
//f0304.cpp
//C-串操作
//====================================
#include<iostream>
#include<string>
using namespace std;
//------------------------------------
int main(){
  char* s1 = "Hello ";
  char* s2 = "123";
  char a[20];
  strcpy(a, s1);                                          //复制
  cout<<(strcmp(a,s1)==0 ? "" : "not ")<<"equal\n";       //比较
  cout<<strcat(a, s2)<<endl;                              //连接
  cout<<strrev(a)<<endl;                                  //倒置
```

```
    cout<<strset(a, 'c')<<endl;                              //设置
    cout<<(strstr(s1, "ell") ? "" : "not ")<<"found\n";      //查找串
    cout<<(strchr(s1,'c') ? "": "not ")<<"found\n";          //查找字符
}//===================================
```

```
E:\ch03>f0304↙
equal
Hello 123
321 olleH
ccccccccc
found
not found
```

这些库函数的操作，默认在 string.h 的头文件中。

strcpy 读作 string copy，其函数声明为：

```
char* strcpy(char* x1, char* x2);
```

该函数将 x2 字串复制到 x1 所在位置，不论 x1 字串先前是什么内容，复制之后都将被所复制内容所覆盖。

拷贝函数被调用之后，返回 x1 参数的首地址，目的是让调用结果可以直接参加之后的字串操作。

由于 x2 字串的长度可能比 x1 字串空间要长，所以 strcpy 的使用并不安全，需要编程时谨慎考虑。如果 a 字符数组的长度为 3（少于 s1 的长度），则执行 strcpy(a, s1)会让紧挨 a 数组的邻近内存空间也被修改，导致不可预料的运行错误。

f0304.cpp 中的 strcpy 函数调用，将使得 a 字符数组的内容变为：

H	e	l	l	o		0	x	x	x	x	x	x	x	x	x	x	x	x

字串"Hello "覆盖之后，以整数 0 标志字串的结束，其后的空间内容并没有被修改，以 x 表示原来的数据。

strcmp 读作 string compare，其函数声明为：

```
int strcmp(char* x1, char* x2);
```

它表示 x1 串与 x2 串进行字典序比较。如果 x1 小则返回值为负数；如果 x1 大，则返回非 0 正数；如果两者相等，则返回 0。

因为 a 串与 s1 串虽然分处于不同空间，但值相同，将其作为参数调用 strcmp 时，返回值为 0，所以我们看到程序运行结果第一行为 equal。

strcat 读作 string concat，其函数声明为：

```
char* strcat(char* x1, char* x2);
```

它将 x2 字串的内容接在 x1 字串之后，或者说，将 x2 字串复制到 x1 字串结束处。所返回的 x1 字串让处理结果可以直接参加之后的操作。

显然，该调用会引起 x1 字串加长，或者说，结束符 0 的位置移后。在 x1 字串之后所

余自身空间不足以接纳 x2 字串时，调用操作将不安全。

strrev 读作 string reverse，其函数声明为：

```
char* strrev(char* x);
```

它将 x 字串倒过来，并改变原来的存储，同时直接将结果字串返回，以便于其后的操作。所以调用 strrev(a)之后，将字串"Hello 123"变成了"321 olleH"。

strset 读作 string set，其函数声明为：

```
char* strset(char* x, char b);
```

它将 x 整个字串的每个字符都用 b 字符来取代，并将 x 作为结果返回。因此执行了 strset(a,'c')之后，a 字串变成了"ccccccccc"。

strstr 读作 string substring，其函数声明为：

```
int strstr(char* x, char* s);
```

该函数在 x 字串中查找 s 字串。若找到，则返回整数 1，否则返回整数 0。

strchr 读作 string char，其函数声明为：

```
int strchr(char* x, char b);
```

该函数在 x 字串中查找 b 字符。若找到，则返回整数 1，否则返回整数 0。

在头文件 string.h 中，还有其他一些常用的 C-串库函数，通过编译器的帮助功能可以搜索到相关的函数。

❑ 3.4.3 string

string 是 C++的 STL 提供的一种自定义的字串类型，它可以方便地执行 C-串所不能直接执行的一切操作。它处理空间占用问题是自动的，需要多少，用多少，不像字符指针那样，提心吊胆于指针脱钩时的空间游离。它可从 C-串转换得到，还可以从内部“提炼”出 C-串……。string 本身就是针对方便字串操作来设计的。例如：

```
//=====================================
//f0305.cpp
//=====================================
#include<iostream>
#include<string>
#include<algorithm>
using namespace std;
//-------------------------------------
int main(){
  string a, s1 = "Hello ";
  string s2 = "123";
  a = s1;                                          //复制
  cout<<(a==s1 ? "" : "not ")<<"equal\n";          //比较
  cout<<a+=s2<<endl;                               //连接
  reverse(a.begin(), a.end());                     //倒置串
```

```
    cout<<a<<endl;
    cout<<a.replace(0,9,9,'c')<<endl;                    //设置
    cout<<(s1.find("ell")!= -1 ? "" : "not ")<<"found\n"; //查找串
    cout<<(s1.find('c')!= -1 ? "": "not ")<<"found\n";    //查找字符
}//=====================================
```

其运行结果与 f0304.cpp 相同。

需要注意的是，string 资源和 string.h 不是一回事。string 资源是指 string 字串应用，string.h 头文件是指 C-串操作的库函数集合。因为 string 头文件总是自动包含 string.h 头文件，所以，在 C-串应用中，包含了 string 头文件等于包含了 string.h。

定义 string 实体的原始形式与定义整型变量一样。例如：

```
int a = 35, b;
string s = "Hello", t;
```

它定义了一个初始化了的整型 a 变量和一个未初始化的 b 变量。同样也定义了一个初始化了的 string 型 s 实体和一个未初始化的 t 实体。

string 还可以有参数化的定义方式。例如：

```
int a(35);
string s("Hello");
string t(15, 'H');
string u(15);
```

string 型 s 实体的定义依据字串参数而将 s 初始化成"Hello"，就像整型 a 变量以 35 初始化那样。string 型 t 实体初始化中有两个参数，第一个参数表示重复度，第二个参数表示重复字符。依据其意义， t 中以 15 个'H'字符构成一个字串。string 实体定义中的初始化，是依据其参数类型的不同而分别操作的，当只有一个整数参数时，表示重复若干个空格的字串 string 型 u 实体中以 15 个空格字符构成一个字串。

字符指针操作字串时，直接比较字符指针是无法进行字串内容比较的，f0303.cpp 已经说明了这个问题。而 string 型字串可以直接进行字串内容的相等比较。例如代码 f0305.cpp：

```
cout<<(a==s1?"":"not ")<<"equal\n";
```

就能得到"equal"的结果。

string 实体还能用+进行字串拼接操作，用+=进行附加式字串拼接操作。例如代码 f0305.cpp 中的 a+=s2 便是赋值表达式，它使 a 实体和表达式本身都变成了"Hello 123"。

string 类型自身是没有字串逆反操作的，但是通过 C++的 STL 库中的 reverse 函数，便可以实现字串的逆反。reverse 函数包含在资源 algorithm 中，它的两个参数以一头一尾的形式描述一个容器的一个区间，其功能是将该容器中一头一尾区间内的所有元素颠倒位置。string 实体也是一种容器，其一头一尾的标准表示就是对其实体做 begin()和 end()操作。对于 string 实体 a 的内容原来为"Hello 123"，做了 reverse(a.begin(), a.end())函数调用后，其 a 的内容变成了"321 olleH"。由于 reverse 操作不返回颠倒位置后的内容结果，所以为了查看颠倒结果，需要对 string 实体 a 进行单独输出。

从表示 string 实体 a 的头尾位置的 a.begin()和 a.end()操作，我们看到了 STL 中的操作

往往都是一定类型的实体捆绑某个操作（函数调用）的行为。这也是在解释整型实体往往称为变量，因为其只有数值的单纯变化。而 STL 实体往往涉及多个分量变化，而且有诸多捆绑的操作。

string 中有 replace 操作。它可以将由下标规定的字串区间用重复一定个数的字符来替换。a.replace(0,9,9,'c')是表示 a 字串从下标 0 开始，长度是 9 的子串用 9 个'c'字符代替。由于 a 字串本来就 9 个字符长，所以就是将字串中所有 9 个字符用'c'字符代替。因为它返回实体在操作之后的结果，所以直接用 cout 输出将看到替换后的结果。

string 中也有 find 操作。当它以 C-串为参数时，返回是否找到该参数描述字串的逻辑值；当以字符为参数时，返回是否找到该参数描述字符的逻辑值。

string 还有其他的操作，如各种搜索操作、插入操作、取长度、删除字符、删除字串、判断空串等。string 与 C-串具有很好的亲和性，C-串可以直接赋值给 string 实体。string 长度可伸缩，比字符数组灵活得多。由于通过字符指针来操作字串，是人为操纵所指向的字串空间，因此，它会导致许多编程中的低级错误。这些错误的排查，需要各种经验，成为程序员晋级的重要障碍。例如：

```
char* str1;
char* str2 = new char[5];
strcpy(str2, "ugly");
strcpy(str1,str2);          //错: str1 没有空间可储
strcpy(str2, "Hello");      //错: str2 空间不够
str2 = "Hello";             //原来的"ugly"空间脱钩，导致内存泄漏
```

因此，操作字串，string 比之 C-串（字符指针），既优雅又灵活，唯一的缺点是，在大量的字符处理中，性能上略逊于字符指针，所以在 ACM 程序设计竞赛中，多以字符指针处理字串。而在入门阶段，string 之于初尝编程快感，且衔接于后继的抽象编程，C-串之于体验内部存储实现，两者互为参照，缺一则憾。

❑ 3.4.4 string 与 C-串的输入输出（string & C-string I/O）

IO 流对 string 串和 C-串都能完美识别，读入 string 串与读入一行 string 串略有区别，同样，读入 C 字串与读入一行 C-串也略有区别。下面这行文字的输入，可以来自文件，也可以来自键盘，总之是带有回车的有若干空格的字符序列：

```
Hello, How are you? ↙
```

则可通过循环读入单词操作将内容输送到变量中，直到读不到数据（缓冲区读完，并且没有数据接续，则流状态变成 false）：

```
for(string s; cin>>s;)          //用 string 串
  cout<<s<<" ";
cout<<endl;
```

或者：

```
for(char a[10]; cin>>a;)      //用C-串
  cout<<a<<" ";
cout<<endl;
```

cin>>的读入方式总是将前导的空格（所谓空格，即包括空格、回车、水平或垂直制表符等）滤掉，将单词读入，当遇到空格时结束本次输入。

也可以通过 getline 将其一次性地输入：

```
string s;
getline(cin, s);              //string串的读入一行
cout<<s<<endl;
```

或者：

```
char a[40];
cin.getline(a, 40);           //C串的读入一行
cout<<a<<endl;
```

注意两者使用 getline 格式上的差异。getline 总是将行末的回车符滤掉。在本次输入中没有什么影响，但若有许多行，且行中还夹杂着其他类型的数据时，借助于 getline 然后再逐个分解行中各数据是简明的，了解这一点很重要！

如果是逐个字符输入，那应如何呢？见下列代码：

```
for(char ch; (ch=cin.get())!='\n';)
  cout<<ch;
cout<<endl;
```

要注意的是，上面分别用了字符数组和 string 两种操作的方式，边比较，边学习 string 的使用方法。

❑ 3.4.5 string 流（string Streams）

如果一个文件 aaa.txt，有若干行，不知道每行中含有几个整数，要编程输出每行的整数和，该怎么实现？

由于 cin>>不能辨别空格与回车的差异，因此只能用 getline 的方式逐行读入数据到 string 实体中，但在 string 实体中分离若干个整数还是显得有点吃力。一个好的方法是用 string 流：

```
//=====================================
//f0306.cpp
//整行读入再分解读入
//=====================================
#include<iostream>
#include<sstream>
#include<fstream>
using namespace std;
```

```
12 3 45 67 8 9
56 232 12 23
12 1
8
1212 2312
```

aaa.txt

```
//-------------------------------------
int main(){
  ifstream in("aaa.txt");
  for(string s; getline(in, s);){
    int a, sum=0;
    for(istringstream sin(s); sin>>a; sum += a); //用 string 流分解 s 串的整数
    cout<<sum<<endl;
  }
}//=====================================
```

```
E:\ch03>f0306↙
144
323
12
8
3524
```

说实话，该程序编得有点放肆。本该将 istringstream sin(s)单独占一行的，结果非但不然，还将 sum+=a 都缩到循环结构描述的步长部分中去了。这样一来，循环体便为空了，于是，for 循环的描述部分后面加上分号便自成独立的语句，但它确实能够完成累计工作。作为单独的循环，最后的“;”还是不能忘记的！因为程序小，所以可读性还不到受伤害的地步，请读者也来见识一下这种风格。

istringstream 是输入 string 流，它在 sstream 资源中说明。该语句类似文件流操作，只不过创建 sin 流时，其参数为 string 对象。它是将 string 的实体看作是一个输入流，因而，sin>>a 即是从 string 流中输入整数到 a 中，输啊输，一直输到 string 中的最后一个整数！

string 流很有用，有时候要将内容先逐个输出到 string 中，最后才根据计算结果来编排输出格式。这时候，用 string 流就很管用。

由于 string 可以很方便地修改、复制、插入、删除、拼接、比较等，所以在输入/输出流中，还能够进一步编辑流的格式，对于程序处理的统一性、方便性带来了莫大的好处。

getline 函数的返回是流状态，通过其可以判断文件是否还有数据行可读。

在 C++的早些时候，用 C-串流比较多，定义方式与 string 流不同，它在头文件 strstream 中说明，因为使用 C-串流的历史也不长，况且如今 C++标准已经走得很远了，所以不应再回头去用那个迂腐的东西。

3.5 数组（Arrays）

❑ 3.5.1 元素个数（Number of Elements）

数组定义的格式为：

```
类型名 数组名[常量表达式];
```

常量表达式表示数组的元素个数，并不表示最大下标值。例如：

```
int a[5];
```

则表示可以访问的元素为 a[0]～a[4]。但不能访问 a[5]，见图 3-4。

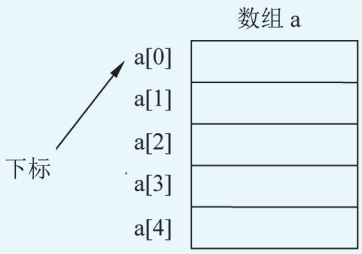

图 3-4　数组下标

常量表达式的值只要是整数或整数子集就行。例如：

```
int a['a'];   //表示 int a[97];
```

数组定义是具有编译确定意义的操作，它分配固定大小的空间，就像变量定义一样的明确。因此元素个数必须是由编译时就能够定夺的常量表达式。下面这样的数组定义有问题：

```
int n=100;
int a[n];   //错: 数组元素个数必须是常量
```

虽然，根据上下文，编译似乎已经知道 n 的值，但编译动作因变量性质而完全不同。变量性质就是具有空间占用的可访问实体，编译每次碰到一个变量名称就对应一个访问空间的操作。因此，int a[n]实际上要在运行时，才能读取变量 n 的值，才能确定其空间大小。这与数组定义的编译时确定意义的要求相违背，因而编译时报错。

而对于下面的定义，却是允许的。因为编译中，常量虽也占空间，甚至也涉及内存访问，但因数据确定，而可以被编译用常数替代。事实上，常量在编译时经常是优化的目标，能省略内存空间访问就应该省略，讲求效率的 C++编译器会乐此不疲：

```
const int n=100;
int a[n];    //ok
```

❑ 3.5.2　初始化（Initialization）

数据的读入一般涉及从其他外部设备中输入的过程，但元素不多而又确定数据值的小数组可以直接在程序中初始化。例如：

```
int iArray[10] = {1,1,2,3,5,8,13,21,34,55};
```

注意上述形式中，大括号中的初始值个数不能多于数组定义的元素个数。初始值不能通过逗号的方式省略，初始值也不能为空。但在总体上，初始值可以少于数组定义的元素个数。例如：

```
int array1[5] = {1,2,3,4,5,6};  //错: 初始值个数太多
int array2[5] = {1,,2,3,4};     //错: 不能以逗号方式省略
int array3[5] = {1,2,3, };      //错: 同上
int array4[5] = {};             //错: 初始值不能为空
int array5[5] = {1,2,3};        //ok
int array6[5] = {0};            //ok
```

只要动用了大括号，就是实施了初始化。对于实施初始化的数组，如果初始值个数不

足方括号中规定的元素个数，则后面的元素值全补为 0。因此，array5[3]、array5[4]为 0，且 array6 的全部元素都为 0。

具有初始化的数组定义，其元素个数可以省略，即方括号中的表达式可以省略。这时候，最后确定的元素个数取决于初始化值的个数。例如：

```
//=====================================
//f0307.cpp
//省略数组定义中方括号内的表达式
//=====================================
#include<iostream>
//-------------------------------------
int main(){
  int a[]={1,2,3,4,5};          //数组 a 有 5 个元素
  //...
  for(int i=0; i<sizeof(a)/sizeof(a[0]); ++i)
    std::cout<<a[i]<<" ";
  std::cout<<"\n";
}//=====================================
```

```
E:\ch03>f0307↙
1 2 3 4 5
```

程序中，用了 sizeof(a)，即 a 数组的字节数。还有 sizeof(a[0])，即第一个元素所占空间字节数，因为是整型数组，所以相当于 sizeof(int)，在 32 位编译器中整型数长度为 4。a 数组有若干个元素，每个元素有 sizeof(a[0])字节数，所以两者相除就是元素个数。这样表示的用意在于可维护性。因为数组元素个数随着初始化中元素的增减而变化，数组的类型随着编程需要可能也会变化。在稍大一点的编程中，输入过程与处理过程往往分离，即中间相隔许多语句，所以，并不一下子能够直观地得到数组元素个数的值，而根据数组名和其元素的信息，获得元素个数的方式，就带有很好的通用性。for 循环的结构描述就无须跟着数组的初始化变动而改动了。

另外，如果没有初始化部分，则数组定义的方括号内的表达式不能省略。例如：

```
int a[];    //错: 元素个数不知道
```

除此之外，字符数组比其他数组有一点书写上的特殊性，它的初始化有以下三种形式：

```
(1) char chs1[6]={"hello"};
(2) char chs2[5]={'h','e','l','l','o'};
(3) char chs3[6]="hello";
```

其中最后一种形式最简单，在上节已经看到过。要注意的是，第二种形式没有 C-串的结束符，但因默认初始化 chs[5]=0，所以数组名可以拿来做 C-串操作。而第 1、3 两种情况的实际字符数应为 6，如果元素个数少于 6，则将编译出错。这样的设计完全是为了满足编程方便。

❑ 3.5.3 默认值（Default Values）

对于没有初始化的数组，分两种情况：一种是全局数组和静态数组（☞CH7.3，CH7.4），也就是在函数外部定义的，或者加上 static 修饰的数组定义，其元素总是全被清 0；另一种是局部数组，就是在函数内部定义的数组，它们的值是不确定的。例如：

```
//=====================================
//f0308.cpp
//探测数组初值
//=====================================
#include<iostream>
using namespace std;
//-------------------------------------
int array1[5]={1,2,3};      //有初始化
int array2[5];              //无初始化
//-------------------------------------
int main(){
  int array3[5]={2};        //有初始化
  int array4[5];            //无初始化
  cout<<"array1: ";
  for(int i=0; i<5; ++i)
    cout<<array1[i]<<" ";
  cout<<"\narray2: ";
  for(int i=0; i<5; ++i)
    cout<<array2[i]<<" ";
  cout<<"\narray3:   ";
  for(int i=0; i<5; ++i)
    cout<<array3[i]<<" ";
  cout<<"\narray4:   ";
  for(int i=0; i<5; ++i)
    cout<<array4[i]<<" ";
  cout<<"\n";
}//=====================================
```

```
E:\ch03>f0308↙
array1: 1 2 3 0 0
array2: 0 0 0 0 0
array3: 2 0 0 0 0
array4: 1245072 845597673 0 0 4198406
```

数组的这种初始化规定方式源于程序运行的空间布局，函数中的局部数组是随着函数调用而创立的，而函数外部的数组是在整个程序运行中起作用的，驻于全局数据区，在运行起始时被初始化为 0（☞CH5.3.1）。

❑ 3.5.4 二维数组（2-D Arrays）

可以定义二维数组：

```
int a[3][5];
```

也可以给二维数组初始化：

```
int a[3][5]={{1,2,3,4,5}, {2,3,4,5,6}, {3,4,5,6,7}};
```

初始化的默认值规则，参照一维数组的初始化规则。
使用二维数组，可以按行、列下标访问对应元素。例如：

```
//=====================================
//f0309.cpp
//二维数组
//=====================================
#include<iostream>
using namespace std;
//-------------------------------------
int main(){
  int array1[2][3]={1,2,3,4,5};     //依次初始化全部 6 个元素，最后一个默认为 0
  int array2[2][3]={{1,2},{4}};     //以维为单元，3 个元素为 1 个单元，共 2 个单元
  cout<<"array1: ";
  for(int i=0; i<2; ++i)
  for(int j=0; j<3; ++j)
    cout<<array1[i][j]<<",";
  cout<<"\narray2: ";
  for(int i=0; i<2; ++i)
  for(int j=0; j<3; ++j)
    cout<<array2[i][j]<<",";
  cout<<"\n";
}//=====================================
```

```
E:\ch03>f0309↙
array1: 1,2,3,4,5,0,
array2: 1,2,0,4,0,0,
```

3.6 向量（Vectors）

❑ 3.6.1 基本操作（Basic Operations）

vector 是向量类型，它是一种对象实体。它可以容纳许多其他类型的相同实体，如若干个整数，所以称其为容器。vector 是 C++STL（标准模板类库）的重要一员，使用它时，

只要包含资源 vector 即可。

vector 可以有四种定义方式：

```
(1) vector<int> a(10);
(2) vector<int> b(10, 1);
(3) vector<int> c(b);
(4) vector<int> d(b.begin(), b.begin()+3);
```

vector<int>是带类型参数的模板类型，尖括号中为元素类型名，它可以是任何合法的数据类型。

第 1 种形式定义了 10 个整数元素的向量，但并没有给出初值，因而，其值是不确定的。

第 2 种形式定义了 10 个整数元素的向量，且给出其每个元素的初值为 1。这种形式是数组望尘莫及的。数组只能通过循环来成批地赋值。

第 3 种形式用另一个现成的 b 向量创建一个 c 向量。

第 4 种形式定义了其值依次为 b 向量中第 0 到第 2 个（共 3 个）元素的向量。

因此，创建向量时，不但可以整体向量复制性赋值，还可以选择其他容器的部分元素来复制定义。特别地，向量还可以从数组获得初值。例如：

```
int a[7]={1,2,5,3,7,9,8};
vector<int> va(a, a+7);
```

上面的第 4 种形式的 b.begin()、b.end()是表示向量的起始元素位置和最后一个元素之外的元素位置。向量元素位置也属于一种类型，称为遍历器。遍历器不单表示元素位置，还可以在容器中前后挪动。每种容器都有对应的遍历器。向量中的遍历器类型为：vector<int>::iterator。因此，如果要输出向量中所有元素，可以有两种循环控制方式：

```
for(int i=0; i<a.size(); ++i)    //第 1 种
  cout<<a[i]<<" ";
for(vector<int>::iterator it=a.begin(); it!=a.end(); ++it)   //第 2 种
  cout<<*it<<" ";
```

第 1 种形式是下标方式，a[i]是向量元素操作，这种形式与数组一样；第 2 种形式是遍历器方式，*it 是指针间接访问形式，它的意义是 it 所指向的元素值。

a.size()是向量中元素的个数，a.begin()表示向量的第一个元素，这种操作方式是一个对象捆绑一个函数调用，表示对该对象进行某个操作。类似这样的使用方式称为调用对象 a 的成员函数，这在对象化程序设计中很普遍（☞CH8.2）。向量中的操作都是通过使用成员函数完成的。它的常用操作有：

```
a.assign(b.begin(), b.begin()+3);   //b 向量的 0~2 元素赋给 a
a.assign(4,2);                      //使 a 向量只含 0~3 元素，且赋为值 2
int x = a.back();                   //将 a 的最后一个元素值赋给整数变量 x
a.clear();                          //a 向量中元素清空（不再有元素）
if(a.empty()) cout<<"empty";        //a.empty()经常作为条件，它判断向量空否
int y = a.front();                  //将 a 的第一个元素值赋给整数变量 y
a.pop_back();                       //删除 a 向量的最后一个元素
a.push_back(5);                     //在 a 向量最后插入一个元素，其值为 5
a.resize(10);                       //将向量元素个数调至 10 个。多则删，少则增补，其值随机
```

```
a.resize(10,2);                 //将向量元素个数调至 10 个。多则删，少则增补，其值为 2
if(a==b) cout<<"equal";         //向量的比较操作还有!=,<,<=,>,>=
```

除此之外，还有元素的插入与删除、保留元素个数和容量观察等操作（☞参考文献[10] CH6.2）。

向量是编程中使用频率最高的数据类型。这不仅是因为数据的顺序排列性在生活中最常见，还因为向量有一些整体赋值、判空和元素添加、搜索等最简单的常规操作。当数据并不复杂时，可以代替其他数据类型而很好地工作。特别是向量可以自动伸展，容量可以自动增大，使得对一些不确定数量的顺序性操作数据在工作上带来了极大的方便。

❑ 3.6.2 添加元素（Adding Elements）

常规数组必须在定义时确定元素个数，之后的使用中不得更改。向量较之数组的优越之处是可以改变元素数量的多少。添加元素是其一种操作。例如，读入一个文件 aaa.txt 的数据到向量中，文件中为一些整数（不知个数）。要判断向量中的元素有多少个两两相等的数对。程序代码如下：

```
//=====================================
//f0310.cpp
//向量操作
//=====================================
#include<iostream>
#include<fstream>
#include<vector>
using namespace std;
//-------------------------------------
int main(){
  ifstream in("aaa.txt");
  vector<int> s;                //无元素的空向量
  for(int a; in>>a;)
    s.push_back(a);
  int pair=0;
  for(int i=0; i<s.size()-1; ++i)
  for(int j=i+1; j<s.size(); ++j)
    if(s[i]==s[j]) pair++;
  cout<<pair<<"\n";
}//====================================
```

```
12 3 45 67 8 9
56 232 12 23
12 1
8
1212 2312
```

aaa.txt

```
E:\ch03>f0310↙
4
```

aaa.txt 文件中第 1～3 行各有一个 12，它们两两相等，因此构成三对满足条件的数。文件中第一行和第四行的 8 也是两两相等，所以共有 4 对两两相等的数，这便是运行的结果。

因为不知道文件中的元素个数，所以无法用数组处理，也无法在向量定义中确定元素个数，但可以先创建一个空向量，然后用添加操作不断往向量中添加元素。向量操作中有

一个性能问题，如果频繁扩展容量，就要显著增加向量操作的负担，因为扩容意味着分配更大空间，复制原空间到现空间，删除原空间。

向量并不是每次扩展都要扩容，向量中预留了一部分未用的元素供扩展之用。一开始若创建一个空向量，则向量中已经含有一些未用的元素，可以用 capacity()查看。如果上述程序面临着大量数据，例如，10 万个整数，这时候，为了保持向量的性能，应该在一开始就规定保留未用元素的数量（☞CH6.6）。

❑ 3.6.3 二维向量（2-D Vectors）

在二维向量中，可以使用 vector 中的 swap 操作交换两个向量。swap 操作是专门为提高两个向量之间互相交换的性能而设计的。如果用一般的 swap：

```
void swap(vector<int>& a, vector<int>& b){
  vector<int> temp = a; a = b; b = temp;
}
```

它要涉及向量的创建、赋值和再赋值，最后还要销毁临时向量。但若用 vector 的 swap 操作，这些工作都可以省掉。只要做微不足道的地址交换工作，岂不美哉？！

例如，文件 aaa.txt 中含有一些行，每行中有一些整数，可以构成一个向量。整个文件可以看成是一组向量，其中每个元素又都是向量，只不过作为元素的向量其长度参差不齐。设计一个排序程序，使得按从短到长的顺序输出每个向量。这时候，程序代码如下：

```
//=====================================
//f0311.cpp
//若干个向量按长短排序
//=====================================
#include<iostream>
#include<fstream>
#include<sstream>
#include<vector>
using namespace std;
//-------------------------------------
typedef vector<vector<int>> Mat;
Mat input();
void mySort(Mat& a);
void print(const Mat& a);
//-------------------------------------
int main(){
  Mat a = input();
  mySort(a);
  print(a);
}//-------------------------------------
Mat input(){
  ifstream in("aaa.txt");
  Mat a;
  for(string s; getline(in, s);){          //循环读入行
    vector<int> b;
```

```
12 3 45 67 8 9
56 232 12 23
12 1
8
1212 2312
```

aaa.txt

```
    istringstream sin(s);
    for(int ia; sin>>ia;)
      b.push_back(ia);                          //分解行中整数存入 b 向量
    a.push_back(b);                             //将 b 向量添加入 a 矩阵
  }
  return a;
}//-----------------------------------
void mySort(Mat& a){
  for(int pass=1; pass<a.size(); ++pass)
  for(int i=0; i<a.size()-pass; ++i)
    if(a[i+1].size()<a[i].size()) a[i].swap(a[i+1]);     //向量中的 swap
}//-----------------------------------
void print(const Mat& a){
  for(int i=0; i<a.size(); ++i){
    for(int j=0; j<a[i].size(); ++j)
      cout<<a[i][j]<<" ";
    cout<<endl;
  }
}//===================================
```

```
E:\ch03>f0311↙
8
12 1
1212 2312
56 232 12 23
12 3 45 67 8 9
```

该程序涉及四个函数，其中 main 函数调用了其他三个函数。这三个函数分别是输入 input，排序 mySort 和输出 print。可以参考函数参数的使用（☞CH5.1.3）。

其中的 input 函数与程序 f0306.cpp 例子很像。只不过这里输入的是向量。由于每行中的整数个数不知道，所以向量的长短也不知道，又由于总的行数不知道，所以二维向量的元素个数也不知道，输入时只能以向量添加元素的方式。

输出 print 是一个两重循环，它按序将二维向量中的每个元素（向量）打印出来。且每打印一个向量，就换行。

mySort 是排序函数，它按向量元素个数的多少进行排序，少的在前，多的在后。使用的是“冒泡排序法”。冒泡排序法在很多程序设计和数据结构的书中都有介绍（☞参考文献[7]CH7.6.1）。排序中所使用的 swap 就是两个向量相互交换的操作，它在 vector 中定义。

用 typedef 来定义 Mat 这个二维向量的名称，以使程序中的名称易记易用。

向量操作散见于本书的各个章节中，且为本书重点描述的数据类型。

3.7 指针与引用（Pointers & References）

C++拥有在运行时获得变量或对象的地址和通过地址操纵数据的能力，这种能力是通过指针发挥的。由于其很多高级操作的内部实现都依赖指针，所以指针不但在过程化程序设计中必不可少，在面向对象程序设计中也必不可少。指针用于数组，用于函数参数，用

于动态内存空间申请和释放等，指针对于成功进行 C++ 编程至关重要。也许指针在其他语言中并不是必要的，但在 C++中，显得很必要。指针在提高性能方面，提升 C++的产业竞争力上，立下了汗马功劳。指针功能是强大的，但又是最危险的。学习指针的时候，我们始终要强调指针的双刃剑作用。

❑ 3.7.1 指针（Pointers）

每个类型都可以定义存储实体，所以就有对应的指针，指针定义的形式为：

```
int* ip;
char *cp;
float*fp;
double * dp;
```

定义中的*可以居左，居右，或居中，ip、cp、fp、dp 都是指针。由于指针本身也是一种实体，因此，甚至指针本身也有对应的指针：

```
int** iip;                  //即整数指针 int* 的指针
```

其中，iip 称为二级整型指针。

一个*只能修饰一个指针，所以：

```
int* ip, iq;
```

则表示 ip 为指针，而 iq 为整型变量。

在一个定义语句中定义两个指针的方法为：

```
int* ip, *iq;
```

其中 ip 和 iq 都是指针。指针的定义，由数据类型后跟星号，再跟指针名组成。指针有变量与常量之分（☞CH3.7.4），不做说明的指针通常指的是指针变量。要弄清一些教科书中对指针的含混描述，就得留意不同情景下的意味。

指针可以赋值，也可以在定义指针时初始化，赋值或初始化的值是同类型实体的地址：

```
int* ip;
int iCount = 18;
int* iPtr = &iCount;      //初始化
ip = &iCount;             //赋值
```

“&”表示实体的地址，由于字面值不认为是具有空间地址的实体，所以不能进行&操作：

```
ip = &23;   //错
```

初学者一不小心，就会混淆语句“int* ip=&iCount;”与“*ip=&iCount;”，前者是带有初始化的指针定义语句；后者是错误的赋值语句，因为*ip 为间接访问（dereference）所指向的整型实体的操作，而&iCount 并非整型实体，左右两边类型不一致，导致错误！

指针指向的实体，可以通过指针的间接访问操作（即在指针变量前加*号的操作）读写该空间的内容。例如：

```
int iCount = 18;
int* ip = &iCount;
*ip = 12;
cout<<*ip<<" "<<iCount<<endl;
```

*ip=12 间接访问操作的结果，把变量 iCount 的值改变了。显示的结果为 12 12。因此，间接访问操作对所指向的实体既可以读也可以写。写就意味着实体的改变，意味着也影响了所关联的变量。

由于指针本身也为具有空间的实体，因此也具有空间地址，也可以被别的指针（二级指针）所操纵。例如，下面通过二级指针的两次间接访问，最终操纵整型实体，见图 3-5。

```
int iCount = 18;
int* ip = &iCount;
int** iip = &ip;
cout<<**iip<<endl;
```

其显示的结果为 18。

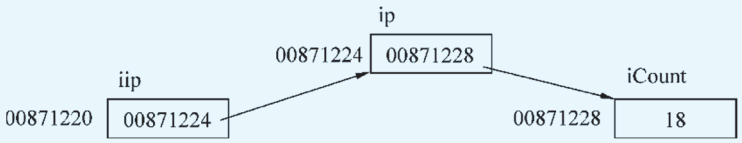

图 3-5　二级指针

初学者要注意，*在不同的地方有不同的含义：

```
int a = 8;
int c = 12 * a;      //乘法操作符
int* ip = &c;        //指针定义
cout<<*ip<<endl;     //指针间接访问
```

间接访问操作只能用在指针上，否则编译报错：

```
cout<<*c<<endl;      //错
```

指针的 0 值不是表示指向地址 0 的空间，而是表示空指针，即不指向任何空间。而指针只有指向具体的实体，才能使间接访问操作具有意义：

```
int* iPtr;
*iPtr = 58;          //错
```

从另一个角度说，指针忘了赋值，比整型变量忘了赋值危险得多。因为这种错误，不能被编译所发现，甚至调试的发现能力也很有限，到了运行时发现，可能已经作为发行版本颁发，而来不及挽回损失了。这种不安全性也是 C++引入引用（☞CH3.7.5）的重要意图所在。

❑ 3.7.2　指针的类型（Pointer Types）

指针是有类型的。给指针赋值，不但必须是一个地址，而且应该是一个与指针类型相符的变量或常量的地址。如，“int* ip;”则 ip 为整型指针；“float* fp;”则 fp 为浮点型指针。

计算机中的内存空间实体是没有类型性的，都是二进制位码。如果一个 int 型变量代表一个 32 位的空间实体，那么这个空间在用该变量访问时，就理解为整型实体，如果一个 float 型变量也代表这同一个 32 位空间实体，则在用该浮点变量访问时，就理解为浮点实体。例如：

```
//=====================================
//f0312.cpp
//空间实体的理解
//=====================================
#include<iostream>
using namespace std;
//-------------------------------------
int main(){
  float f = 34.5;
  int* ip = reinterpret_cast<int*>(&f);
  cout<<"float address: "<<&f<<"=>"<<f<<endl;
  cout<<"  int address: "<<ip<<"=>"<<*ip<<endl;
  *ip = 100;
  cout<<"  int: "<<*ip<<endl;
  cout<<"float: "<<f<<endl;
}//=====================================
```

```
E:\ch03>f0312↙
float address: 1245064=>34.5
  int address: 1245064=>1107951616
  int:100
float:1.4013e-43
```

一个 float 变量的空间实体，被一个整型指针指向，当该指针间接访问时，float 变量的空间实体便现出整型实体的面相，于是 34.5 这个浮点数按二进制补码理解，得出的结果变得怪异了：

$$
\begin{aligned}
34.5 &= 100010.1_{(2)} \\
&= 1.000101\times 2^{5}{}_{(2)} \\
&= 0,10000100,00010100000000000000000
\end{aligned}
$$

结果变成：

$$2^{30} + 2^{25} + 2^{19} + 2^{17} = 1107951616$$

指针的类型性体现在间接访问时，读写所指向的空间实体是以自身的类型规定其操作的。然而，指针的类型性表明不同类型的指针，其类型是不同的，不能相互赋值。例如：

```
int* ip = &f;   //错: float 的地址不能赋给 int 指针
```

但从地址值的本质来说，无非是用二进制表示的整数而已。因此从内存空间位置性而言，不同类型的指针又是统一的，都是二进制的地址。所以不能完全隔绝这种不同地址间的联系。于是，在 C 中，便有了蛮横的强制转换（cast）操作，专门对付这种需要：

```
int* ip = (int*)&f;
```

“(int*)”的意思是说，该地址的空间上不管放着什么数据，都将它作为整型地址看待，甚至该空间放的是函数代码（意义根本不同的二进制代码），它也不管！结果导致程序的极度脆弱，程序员如履薄冰，强制转换成了运行崩溃的梦魇。因此为了换得这种地址转换的灵活性，把所有不小心的误操作的责任统统归咎于程序员了。

那么 C++又是怎么看待转换和如何转换的呢？

首先，主要是因为有了指针（和引用）才有使用转换的需要。

其次，以一种类型的角度去看另一种类型的表示总是怪怪的（整型的二进制码以浮点型看，会被理解成完全不同的数值）。转换操作的目的一是要逃避编译的类型检查，以使不是专门用来完成本任务的模块能够凑合着用一用（例如，函数调用的参数匹配）；二是希望进行这种怪怪的理解。所以转换总是别扭的，其操作需要三思而行，在编程中应确实知道自己在干什么。为此，C++用一种繁杂难记的名字标记这种操作。

最后，转换的目的拓宽了，单纯地址意义下的重解释转换 reinterpret_cast<type>（表达式）是最不讲道理的，见程序 f0312.cpp。除此之外，还有静态转换 static_cast<type>(表达式)（☞CH4.3.3）、动态转换 dynamic_cast<type>(表达式)（☞CH12.7.1）和常量转型 const_cast<type>(表达式)（☞CH12.7.3）。

程序 f0312.cpp 中，两种类型的地址转换形式为：

```
int* ip = reinterpret_cast<int*>(&f);
```

C++好难哦，reinterpret_cast 这么难的保留字，亏他设计得出☺！然而这是一种逆向目标的设计，通过不让程序员产生使用的快感达到少用慎用的目的。C++语言所蕴含的哲理，会令无法深入编程技术的程序员获得最大的利益。

反过来，如果该地址指向的空间以整数指针间接访问的形式赋予 100，则以浮点的眼光去看的时候，发现一切都变了！请读者亲自分析一下运行结果的最后一行。

❑ 3.7.3 指针运算（Pointer Operations）

指针值表示一个内存地址，因此它内部表示为整数，这在显示的时候可以看到。指针所占的空间大小总是等同于整型变量的大小，但它不是整型数，我们重温数据类型的三个要素：数据表示、范围表示、操作集合。指针与整型虽有相同的数据表示，相同的范围表示，但它们具有不同的操作。例如，整型变量不能进行间接访问操作，所以指针与整型不是相同的数据类型。

指针不服从整型数操作规则。例如，两个指针值不能相加。两个指针值相减得到一个整型数，指针值加上或减去一个整数得到另一个指针值等。

指针不能赋予一个整型数。要想获得整型数表示的绝对地址，应将整型数重解释转换为对应指针的类型。例如：

```
int* ip = 1234567;          //错: 不能进行 int 到 int*的直接转换
int* sp = reinterpret_cast<int*>(1234567);   //ok
```

指针的加减整型数的操作大多数用在数组这种连续的又是同类型元素的序列空间中。可以把数组起始地址赋给一指针，通过移动指针（加减整数）对数组元素进行操作。数组名本身就是表示元素类型的地址，所以可以直接将数组名赋给指针。例如：

```
//=====================================
//f0313.cpp
//=====================================
#include<iostream>
using namespace std;
//-------------------------------------
int main(){
  int iArray[6];
  for(int i=0; i<6; ++i)          //数组元素赋值
  iArray[i] = i*2;                //用 ip 指针访问数组元素
  for(int* iP=iArray; iP<iArray+6; iP+=1)
    cout<<iP<<": "<<*iP<<endl;
}//=====================================
```

```
E:\ch03>f0313↙
1245036: 0
1245040: 2
1245044: 4
1245048: 6
1245052: 8
1245056: 10
```

该程序先将数组 iArray 赋初值为 0，2，4，6，8，10；然后将数组起始地址赋给指针 iP，通过指针的循环移动完成输出工作，见图 3-6。C++中的函数表现了 C 的性能特征。在传递以数组为代表的大块数据时，仅仅传递了数据块首地址（☞CH5.2.1），所以，指针的循环移动成为块数据内逐一访问元素的经常性操作。

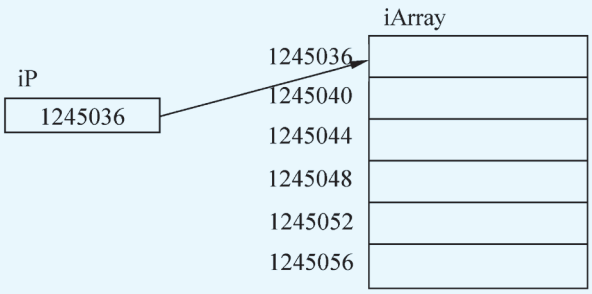

图 3-6　指针指向数组

要留心的是，作为指针 iP 每次循环都只加 1 而不是假想的 4。但是元素的地址却是以 4 字节递增的。

指针的增减是以该类型的实体大小为单位的。即

对 float 指针加 6 实际增加了 24 字节；

对 long int 指针加 5 实际增加了 20 字节；

对 char 指针减 7 实际减少了 7 字节；

对 double 指针减 2 实际减少了 16 字节。

然而，指针的增减操作应受约束，如果数组元素只有 10，而指针获得数组首地址后，进行了+20 等超过数组范围的操作是危险的！可参见进一步的描述（☞CH5.2.2）。

❑ 3.7.4　指针限定（Pointer Restrictions）

一个指针可以操作两个实体，一个是指针值（即地址），一个是间接访问值（即指向

的实体)。于是指针的常量性也分两种：指针常量（constant pointer）和常量指针（pointer to constant）。

指针常量是相对于指针变量而言的，也就是指针值不能修改的指针。

常量指针是指向常量的指针的简称，在一些书上将 pointer to constant 翻译成指针常量就与语意不符了，因为 pointer to constant 的主体是指针而不是常量。定义指针常量还是常量指针就看 const 修饰，若 const 修饰指针本身，则为指针常量，若修饰指针类型（指向的实体类型），则为常量指针。例如：

```
const int a = 78;
int b = 10;
int c = 18;
const int* ip = &a;      //const 修饰指向的实体类型——常量指针
int const* dp = &b;      //等价于上一句——常量指针
int* const cp = &b;      //const 修饰指针 cp——指针常量
const int* const icp = &c;  //常量指针常量
*ip = 87;                //错: 常量指针不能修改指向的实体，*ip 只能作右值
ip = &c;                 //ok: 常量指针可以修改指针值
*cp = 81;                //ok: 指针常量可以修改指向的实体
cp = &b;                 //错: 指针常量不能修改指针值，即使用同一个地址
*icp = 33;               //错: 常量指针常量不能修改指向的实体
icp = &b;                //错: 常量指针常量不能修改指针值
int d = *icp;            //ok
```

见图 3-7，图中阴影部分为指针操作不能修改的内容，但是读取访问总是可以的。

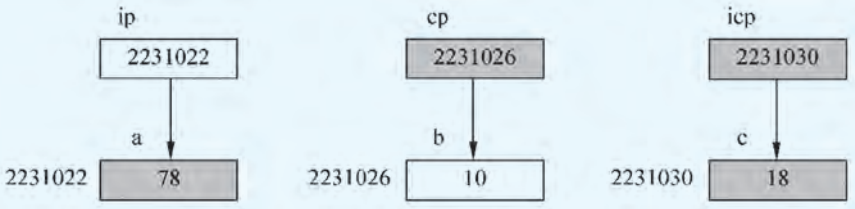

图 3-7　指针限定的各种形态

const 修饰只是限定指针的操作，但不能限定空间上的可改变性。因为一个实体空间可能被不止一个变量所关联，所以实体空间被其他关联变量的改变是可能的。例如：

```
//=====================================
//f0314.cpp
//=====================================
#include<iostream>
using namespace std;
//-------------------------------------
int main(){
  int a = 78, c = 18;
  const int* ip = &a;
  const int* const icp = &c;
  a = 60;  c = 28;
  cout<<"ip =>"<<b<<endl;
```

```
  cout<<"icp=>"<<c<<endl;
}//===================================
```

```
E:\ch03>f0314↙
ip =>60
icp=>28
```

指针的限定在函数参数传递中较常用（☞CH5.2.1）。

❑ 3.7.5 引用（References）

从逻辑上理解，引用是个别名(alias)。当建立引用时，用一个有具体类型的实体去初始化别名，之后，别名便与关联其实体的变量（或对象）享受访问的同等待遇。

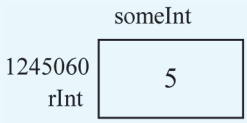

图 3-8 引用

引用定义的形式如下，见图 3-8。

```
int someInt = 5;
int& rInt = someInt;          //初始化
```

注意，格式上可以：

```
int  &rInt = someInt;
int  &  rInt = someInt;
```

引用定义时必须初始化，这是它与指针根本不同的地方。给引用初始化的总是一个内存实体，否则的话，引用就如无根之草，虚无缥缈。初始化的内存实体不是通过一个地址表示，而是通过一个代表该实体的名称表示的，它可以是变量也可以是常量。显然，它也严格要求类型匹配。也就是说，引用的类型与实体的类型应该是严格一致的，否则，编译这一关就通不过。

使用引用，就等于一个实体又多了一个关联的名字。实体的值因而便任由关联的名称（变量或者对象）操作所宰割。因此，修改引用值，就是修改实体值，就是修改对应的变量值，而引用的地址操作也就是所代表的实体地址操作：

```
//=====================================
//f0315.cpp
//引用及其地址
//=====================================
#include<iostream>
using namespace std;
//-------------------------------------
int main(){
  int int1 = 5;
  int& rInt = int1;
  cout<<"&int1: "<<&int1<<"   int1: "<<int1<<endl;
  cout<<"&rInt: "<<&rInt<<"   rInt: "<<rInt<<endl;
  int int2 = 8;
  rInt = int2;
  cout<<"&rInt: "<<&rInt<<"   rInt: "<<rInt<<endl;
}//===================================
```

```
E:\ch03>f0315↙
&int1: 1245064   int1: 5
&rInt: 1245064   rInt: 5
&rInt: 1245064   rInt: 8
```

程序中，rInt 关联了 int1，执行 rInt=int2 后，其值发生变化，但其地址永不改变。

引用与指针的差别，指针可以操纵两个实体，一个是指针值，一个是指向的值，因此指针可以改变关联的实体，即指向的实体。而引用只能操纵一个实体。

一旦引用诞生，就确定了它与一个实体的联系，这种联系是打不破的，直到引用自身的灭亡。从物理实现上理解，引用是一个隐性指针，即引用值是引自所指向的实体。这才是引用（reference）的真意，见图 3-9。

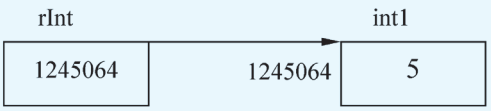

图 3-9　引用实质

引用与实体的关系，看似直接访问，实为指针的间接访问。这幕后的转换工作，是由编译做的。编译将这个特殊的指针 rInt 转换成*rInt 操作，因而，引用不能操作自身的地址值，每次访问 rInt，实际上是在访问所指向的 int1 实体。

引用实现为指针，但它封锁了作为指针实现的地址操作，又将间接访问操作暗中转变为直接操作，使得引用从 reference 的根本实现中抽象出来，对应为 alias 理解了。引用比之指针的直接效果是，使得间接访问操作相对来说更安全了，也就隔绝了万恶之源（☞CH5.3.4）。

引用也可以限定，例如：

```
int int1 = 5;
const int& crInt = int1;
```

则阻止 crInt 做写操作，但这不妨碍实体值可能被修改：

```
int1 = 8;
cout<<crInt<<endl;   //结果为 8
```

引用的限定与指针的限定相似，它们在函数参数传递中大展身手（☞CH5.2.1）。

对象化程序设计中对象参数传递多用引用，这主要是从安全因素着眼的。

3.8　目的归纳（Conclusion）

整型数在计算机内是用二进制补码表示的。一定类型的整数，其位长是确定的。由于采用二进制补码，正、负数得到统一，加、减法得到统一，所以乘、除法也就得以简化，并且使计算机运算部件的设计与实现受益。

char、bool 和 enum 都可以看作是整型的子类，它们表示为二进制补码形式，也就是整型数形式，范围是整型数的子集，它们符合整数的操作规律。只是在各自类型的处理上有各自的特点。

浮点数是为了表示范围比较大、精度相对比较粗的数据而设计的。浮点数的表示、范围和处理与整数完全是两回事。由于浮点数的精度在不同长度上的差异，所以它不能进行

精确比较（☞CH4.2.4）。

C-串是 C++程序中经常要表示的字面值，它是 const char*类型的，它能很方便地转换成 string。用字符指针操纵 C-串各个方面都不如 string 操纵 C-串来得安全和灵活。除非为了兼容旧程序，否则应该放弃字符指针。

数组是语言中的基本设施，可以认为它是低级的，在第三、四部分的内容中，大量涉及抽象编程的地方，很少看到数组的踪迹。然而在许多低层的实现中，在讲求性能的编程中，还是需要使用数组的。

从逻辑上理解，向量也是一系列同类元素在空间上的顺序排列，与数组似乎是一回事。但是，向量升华为一种数据结构，它不但具有方便地扩容、重定尺寸、彼此复制、增删、比较等特点，而且还能够借助遍历器和算法库在其上做搜索、排序、集合、分类等操作，因此，它与数组比较而言，是一种功能齐全的奢侈品。如果不是对性能有苛刻的要求，那绝对应将向量作为首选。

指针可以操纵两个数据实体，一个是地址值，一个是指向的实体。而引用的内部实现虽然也是指针形式，但是编译屏蔽了其地址的操作，所以引用是指针出于安全考虑的替代品。

练习 3（Exercises 3）

1．模仿程序 f0302.cpp，打印整数–1234567 的二进制位码。

2．整数分 long int、int、char、bool，浮点数分 float、double、long double，试分别输出各类型的字节长度和位长，输出形式如：

```
long int: 4 byte 32 bits
```

3．定义一个数组，数据为 6，3，7，1，4，8，2，9，11，5。请创建一个向量，把数组的初值赋给它，然后对该向量求标准差（均方差）：

$$s=\sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i-\overline{x})^2}$$

4．有一些日期，在文件 abc.txt 中，后面加*号的表示要加班的日期，试汇总所有每个月 25 号的天数，如果是加班日，则该天乘 2。

```
Oct. 25 2003
Oct. 26 2003
Sep. 12 2003*
Juy. 25 2002*
App. 25 2004
```

abc.txt

5．编制程序，将输入的一行字符以加密的形式输出，然后将其解密，解密的字符序列与输入的正文进行比较，吻合时输出解密的正文，否则输出解密失败。

加密时，将每个字符的 ASCII 码依次反复加上“4962873”中的数字，并在 32(' ')～122('z')之间做模运算。解密与加密的顺序相反。例如，对于输入正文“the result of 3 and 2

is not 8”，则运行结果为：

```
xqk "zlvyuz" wm#7)gpl'5$ry"vvw$A
the result of 3 and 2 is not 8
```

6．阅读下列程序，写出运行结果（应该知道的遍历数组的五种方法）。

```
//=====================================
#include<iostream>
using namespace std;
//-------------------------------------
int main(){
  int sum[5]={0};                    //存放每种方法的结果
  int iArray[]={1,4,2,7,13,32,21,48,16,30};
  int size = sizeof(iArray)/sizeof(*iArray);
  int* iPtr=iArray;
  for(int n=0; n<size; ++n)          //方法 1
    sum[3] += iPtr[n];

  for(int n=0; n<size; ++n)          //方法 2
    sum[2] += *(iPtr+n);

  for(int n=0; n<size; ++n)          //方法 3
    sum[1] += *iPtr++;               //见 4.6.3 节

  for(int n=0; n<size; ++n)          //方法 4
    sum[0] += iArray[n];

  for(int n=0; n<size; ++n)          //方法 5
    sum[4] += *(iArray+n);

  for(int i=0; i<5; ++i)
    cout<<sum[i]<<endl;
}//=====================================
```

7．试将下列程序中的指针改为引用：

```
//=====================================
#include<iostream>
using namespace std;
//-------------------------------------
void mySwap(int* a, int* b);
//-------------------------------------
int main(){
  int a = 16, b = 48;
  cout<<"a = "<<a<<", b = "<<b<<endl;
  mySwap(&a,&b);
  cout<<"After Being Swapped: \n";
  cout<<"a = "<<a<<", b = "<<b<<endl;
}//-------------------------------------
void mySwap(int* a, int* b){
  int temp = *a;
  *a = *b;
  *b = temp;
}//=====================================
```