第3章　数 据 类 型

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（☞参考文献[5]）对它进行研究。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、数据抽象中得到更好的运用，从而通过数学和计算机的手段来解决问题。

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。

以数据类型来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，而且还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**）和双精度浮点（**double**）。

本章除了介绍整型和浮点型两大基本数据类型外，还介绍系统提供的自定义类型——string 字串。这些都是初学中常用的数据类型。

要学好编程，必须理解数据类型，学会数据说明；要学习新的程序设计方法，就要学会自定义数据类型，而这就必须先了解语言的内部数据类型。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

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首先提供了整数数据类型，即整型。它规定了整数的内外部表示形式、表示范围以及整数的运算（操作）。

3.1.1 内部表示原理

1．十进制数转为二进制数

人们首先将十进制正整数转换成二进制整数。因为这种转换是从人类表达走向计算机理解的必要环节。

转换的方法是采用“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顺序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

各步骤的余数自底向上列成一个 01 串——100011，即所求的二进制数。

2．二进制的计算机器

◆ 概述

一个二进制数相当于一串机械开关，一系列的机械开关构成了电子线路，就可以用来做二进制数的逻辑运算了。由数学理论，逻辑数学可以表达任何计算。计算机逻辑就这样随着硬件工艺水平的提高，逐步复杂、逐步发展起来了。

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 8 位，或 16 位，或 32 位，于是，一个二进制数用计算机表示时，位数不足定长字节的位数时，高位上要补足若干个 0。例如，35，以一个字节的 8 位长和两个字节的 16 位长在计算机内部表示时，其二进制数分别为：

00100011
0000000000100011

在计算机还很原始的时候，专家们研究用怎样的二进制数表示十进制数比较有效，既要容易转换又要便于计算。

◆ 原码

计算机最初有一种原码表示方法，它是将若干字节长的位串第一位表示为符号，其他作为二进制数的数字位。例如，8 位二进制数表示-5，则其第一位为 1，其余 7 位表示成 0000101，凑起来就是 10000101。

但是专家们发现，原码中 0 有正、负之分，00000000 为正 0，10000000 为负 0，不同

的计算可能得到不同的 0 值，表达不唯一，而且对于异号加法与同号减法要做两个数的大小比较，它们都将使计算机内部多出一些比较电路，导致处理的复杂性增加。

◆ 反码

同样，反码是另一种二进制数表达方式，它将正数与负数表示为相反数。因此正数有多少，负数也有多少。相反数是指全部位取反，即 0 变 1，1 变 0。例如，正数 5 表示成 00000101，则相反数即 111111010，即为-5。

结果发现它同样有正 0 与负 0 两个 0，令计算中渗入了额外的逻辑判断，也导致复杂性的增加。

两种方法从表达上说应该是明确的，判断正负性只要看第一位就知道。但是在计算逻辑上不够简明，故只应用在很少的场合。

3．补码

◆ 方法

补码的方法是将正数与负数表示为互补数，即两者相加去掉进位构成全 0 数。

在操作上，将正数取反加 1 即成负数，负数取反加 1 即成正数。取反加 1 称为取补操作。取反加 1 即在反码的基础上再加 1。例如，正数 15 表示成 00001111 的 8 位二进制数，取补（取反加 1）而成-15：

```
-15＝-1111(2)
   ＝-00001111
   ＝11110000 + 00000001 //取反加 1
   ＝11110001
```

◆ 0 的特征

粗看之下，与反码差别不大，但是仔细一看却发现，0 的表示唯一了：

```
-0 = -00000000
   = 11111111+ 00000001          //取反加 1
   = 100000000
   = 00000000                    //舍去进位
   = 0
```

◆ 统一加减法

由于 0 取补后还是 0，所以体现了二进制补码中 0 表示形式的唯一性和一致性。更重要的是，减法可以通过加上一个负的减数来计算，而负的减数通过取补而得到加数。减法通过减数的取补而成为加法，加减法便统一了。

加减法的统一使二进制数的操作种类减少，最后当乘除法归并到一系列的加减法之后，发现二进制数的计算操作就只有加法了。

◆ 最高位判正负性

补码同样也能通过最高位来判断数的正负性。例如：

```
01111110-------正数
```

10001101-------负数

补码不失其表达的直观性，又容易转换和便于计算，它还统一了加减法的意义，简化了操作种类，因此而带来了许多设计优化。

因此，整数类型采用二进制数的补码形式作为其内部表示的形式。

4．算术运算原理

◆ 加法

两个十进制整数相加在计算机中是作二进制数加法运算的。例如：

35+12＝00100011+00001100
　　＝00101111
　　＝$101111_{(2)}$
　　＝47

◆ 减法

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

如果计算结果是负数，则在取出去输出时，取补还原成负数。

◆ 乘法

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。

由于二进制整数做左移 1 位相当于乘 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＝11111101 ×00000101
　　＝11111101 ×00000100+ 11111101 ×00000001
　　＝11111101 左移 2 位 + 11111101 左移 0 位
　　＝11110100 + 11111101
　　＝11110001
　　＝$-00001111_{(2)}$
　　= -15

◆ 除法

同理，二进制整数做除以 2 的运算相当于右移 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
　　＝00000100 + 00000001
　　＝4 余 1
　　＝4

由于整数除法中，结果没有小数，所以这里的除法也就是抛弃余数的整除法。读者必须明白实际实现的除法是按一定的算法来决定移位次数从而决定商，操作过程还会做些优化。

3.1.2　整型长度与范围

1．有符号与无符号数

C++中的整型数有两种类型修饰。一种是有符号数，用二进制补码表示；另一种是无符号数，用原始二进制码表示。

◆ **有符号数**

有符号数就是二进制数的补码形式。通过判断其最高位，能够知道该数表达的是正数还是负数。其所对应的类型为默认的 **int**，或者 **signed int**。

◆ **无符号数**

无符号数则纯粹为二进制数，没有符号，没有正负性，全部表示正数。因此，可以判定 8 位二进制无符号数的表示范围为 0 到 255。其中最大数 255 所对应的二进制数为 11111111。无符号数所对应的类型为 **unsigned int**。例如：

```
unsigned int x = 23;   //正确
unsigned int z = -43;  //警告: 无符号数非负，招惹调试问题
```

有符号数与无符号数都是整数类型，当需要进行非负整数运算时，用无符号整数，可以获得多一倍的正数值表达能力。但是，正数表达能力多一倍在计算机中微不足道，用更长的整型数表示整数可能会更有效。相反，整数的正负性表示在计算过程中却十分重要。所以，整数多用有符号整型数，特殊情况才用无符号整型数。C++默认的整型 **int** 即为有符号整型。

有符号数与无符号数在计算机内部表示本质上是一致的，也就是说，操作都是相同的，只是在值的表现（输入/输出）、表示范围、合法性判断上形成了区别。

◆ **补码正负数各半**

补码能够通过最高位判断出整数的正负性，因此，一定位数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，另外 128 个数为负数。正数所能表示的范围为从 0 到 127，最大正整数是 127，即 01111111。负数所能表示的范围是-1 到-128，最小负数是-128，即 10000000。

值得关注的是：

（1）负数值越小（绝对值越小），所表现的二进制数绝对值却越大，即-1 为 11111111，

而-128 却为 10000000。

（2）所有负数，都有正数与之对应。例如，00000001(1)对应 11111111(-1)，01111111(127)对应 10000001(-127)。

（3）-128 这个数是独特的。-128 没有正数与之对应。-128 的取补后仍然是其自身。一切位数（8 位、16 位、32 位、64 位）的二进制补码都有这个特性。

2．整型长度

◆ **长度分类表**

整型的设计有多种长度，有 8 位（例如 **char**），16 位（例如 **short int**），32 位（**int** 或 **long int**），64 位（例如 **long long int**）。随着计算机技术的发展，整型数的位长也在发展。

无论在什么编译器上工作，**short int** 能够保证定义 16 位整型，**long int** 能够保证定义 32 位整型，**long long int** 能够保证定义 64 位整型。而用 **int** 描述的整型，一般表示当前编译器中默认的整型长度。例如，在 32 位编译器中 **int** 代表 32 位整型，而在 16 位编译器中则代表 16 位整型，或许在以后的 64 位编译器中，**int** 将会代表 64 位整型。不管怎么说，标准 C/C++保证下列整型长度的关系成立：

```
char[①]≤short int≤long int≤long long int
```

每一种整型长度都对应有符号（**signed**）和无符号（**unsigned**）两种。并且一般的编译器总是指定默认整型 **int** 为 **signed**。如表 3-01 所示。

表 3-01　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**	**char**
16 位	**signed short int**	**unsigned short int**	**short int**
32 位[②]	**signed int**	**unsigned int**	**int**
32 位	**signed long int**	**unsigned long int**	**long int**
64 位	**signed long long int**	**unsigned long long int**	**long long int**

例如：

```
int y = -67;            //等价于 signed int
```

关于 64 位整型数的表示，C/C++中有一种表示 64 位整型数的扩展类型——**__int64**，它与 **long long int** 所表示的整数位数是相同的，许多编译器都实现了相同的操作。**long long int** 类型作为标准的时间比较晚，在这之前都是使用**__int64**，所以，虽然它不是标准类型，但是许多编译器考虑到历史的原因，都兼容了它。

◆ **编译器与整型长度**

C++编译器在不同的计算机硬件上的表现是不同的。目前计算机上的主流 CPU 逐渐成为 64 位，而目前的主流 C++编译器版本则仍然是 32 位的，软件相对于硬件总是滞后。

① 在一些大型机的 C++编译器中，**char** 默认为 16 位。

② 32 位编译器默认 **int** 为 32 位。

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。C++程序的指令集受制于 C++的编译器。也就是说，如果 C++编译器是 32 位的，则机器哪怕是 128 位的，其 C++程序的运行指令还是 32 位的。

32 位编译器，其 int 类型的长度是 32 位的。所以：

```
int a = 393876;          //正确，但是在 16 位编译器上却有问题
long int b = 393876;     //正确
```

32 位的 **int** 能够表示超过 2^{16} 的整数，或者说，**int** 与 **long int** 是一致的。在本书出版的年份，无论哪个计算机商家，主流编译器还都是 32 位的。

同理，16 位编译器生成 16 位机器指令程序，所实现的整型是 16 位的。所以：

```
int a = 393876;          //错误：16 位有符号整数最大只能表示为 32767
long int b = 393876;     //正确
```

16 位的编译器将整型实现为 16 位，所以 **int** 与 **long int** 不一致。使用低版本的编译器时，编程中为了整型数表示的可移植性，应该将 **int** 改为 **long int**。

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。一般来说，为了兼容运行环境，C++编译器有选择编译成各种机器系统指令的能力，32 位的编译器可以编译比自己级别低的流行指令系统。例如，BCB 编译器可以通过编译指令的设置，而将程序编译成低于 32 位的机器指令。人们总是趋向使用高端机和高端操作系统，所以从用途来看，编译成较低长度机器指令或许只是为了能在某种低端机上运行。

3．表示范围

表 3-02 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 3-02　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~$(2^{32}-1)$
long long int	8	64	-9223372036854775808	9223372036854775807	-2^{64}~$(2^{64}-1)$
signed long long int	8	64	-9223372036854775808	9223372036854775807	-2^{64}~$(2^{64}-1)$
unsigned long long int	8	64	0	18446744073709551615	0~$(2^{64}-1)$

3.1.3 整型数

整型数在外部表示中分两种情况：

（1）在编程时，表示在代码中的整数。这种整数也称为整型字面值（literal），它是 C++编译器能够接受的整数。

（2）通过输入语句读入的整数，也就是 C++能够接受（识别）和表达的整型数。这种整数在输入时，受到读入整型变量类型的约束。表示范围大的整型，能够接受更长的整型数值。

至于通过输出语句显示的整数，输出时，数值长度上将会按整数类型的表示范围，格式上将会按输出时规定的格式控制。

1．整型字面值

◆ **十进制数**

在整型数表示范围（-2147483648～2147483647）内的整数，默认为 **int** 型数。例如：

```
int a = 23;            //整型字面值
```

整型数一般是指有符号整数。

整型数有长度之分，有长整型 **long int** 和长长整型 **long long int**，分别表示 32 位和 64 位整型数。

◆ **八进制数**

八进制整型数以 0 引导，以 0～7 的范围表示各位数字。例如：

```
int b = 0235;          //八进制字面值
```

如果用数字 8 或 9 表示八进制数，例如，0238，则对编译器来说不可忍受，将报告错误。

八进制数的数值转换成十进制数值，可以用各位数字分别乘上 8 的对应幂次来计算。例如，$0235 = 2\times8^2 + 3\times8^1 + 5 = 128 + 24 + 5 = 157$。

八进制整数没有负数表示，也就是说，八进制数是无符号整数。

八进制数与十进制数一样有各种长度的分别，有长整型和长长整型的表示。

◆ **十六进制数**

十六进制整型数以 0x 或 0X 引导，以 0～9 和 A～F（或者 a～f）的范围表示各位数字。其中 a～f 分别表示 10，11，12，13，14，15。例如：

```
unsigned int c = 0x7fa;    //十六进制字面值
```

十六进制数的数值转换成十进制数值，可以用各位数字分别乘上 16 的对应幂次来计算。例如，0x7fa 等价于 $7\times16^2 + 15\times16^1 + 10 = 1792 + 240 + 10 = 2042$。

十六进制数没有负数表示。与八进制数一样，有各种长度的十六进制数。

◆ **无符号数**

整型数分为有符号数与无符号数。字面值后面跟 U 或者 u 表示无符号数，否则为有符

号数。例如：

```
long long int x = 65530U * 65530U;
```

整数表示成无符号数之后，则按无符号数规则进行操作计算（☞CH3.1.5 混合计算）。

该 x 变量定义初始化为两个无符号数（65530<2^{16}）的积（其值介于 2^{31} 和 2^{32} 之间）。此处如果不用无符号整数，则结果只能是有符号数。根据数值大小与表示范围，其积会计算溢出，而得到一个负数。

◆ **长整型**

后缀 L 和 l 或者 LL 和 ll 用于区分长整型（32bit）和长长整型（64bit）。在 32 位编译器下的 C++语言，其长整型与整型的内部表示是相同的。

有时候，为了表示更高精度的整数，非得用长长整型数描述不可。例如：

```
long long int x = 12345678901234567 8LL;  //正确
long long int y = 1234567890123456 78;    //错误: 整型数超范围
```

长长整型有有符号与无符号之分，在整型数后用 U 表示无符号数。长度描述 LL 和无符号性描述 U，次序可以任意，也就是说：12345ULL 等价于 12345LLU。

12345 与 12345ULL 虽然在数值上是相等的，但是在内部表示上占有空间的大小不同，一个为 32 位，一个为 64 位。而且操作意义也不同，一个是有符号操作，一个是无符号操作。

◆ **长度限制**

语言的描述要受到实现的限制，即受到计算机技术发展的限制，受到编译器的限制。例如，文法中规定的整数字面值（☞CH5.1.2 整型数文法），虽然为非 0 数字开头的递归性定义的数字序列，但对序列的长度没有具体说明。正像标识符长度受机器限制那样，整数字面值也受到长度的限制。例如，下列超过整数范围描述的字面值在各个计算机中有不同的反应：

```
int x = 12345678901234567 8LL;
int a = 12345678901234567 89001234567890;
```

将超过 **int** 整型变量 x 表达范围的合法的长长整型数赋值，将会导致 x 变量值不确定。因为在类型转换时，可能截断高位，保留低位，也可能截断低位，保留高位，从而使数值变化不确定。

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；但当其超过了整型所表示的范围时，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。

一般来说，编译器首先将其当做名字接受，因而先进行长度检查。例如，在 VC 编译器中，该语句受到标识符长度限制（≤30）而报错。而在 BCB 编译器中编译能通过，说明标识符长度限制没有超过，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 中便能通过编译，说明通过了长度检查，但与 BCB 一样，其值是荒谬的。

2．整型数的读入

读入整数的识别的程度，受到数据类型的限定。例如，**int** 型变量接受 8 位之内的十进

制整数，long long int 型变量接受 15 位以内的十进制整数。无符号整型变量接受无符号数，有符号整型变量接受有符号整数。例如，对于下列数字串：

```
123
12345
1234567890123
```

则对于下列流输入语句，将正确接受：

```
int a;
long int b;
long long int c;
cin>>a>>b>>c;  //a 为 123，b 为 12345，c 为 1234567890123
```

但是对于“cin>>c>>b>>a;”输入操作，则不能得到 a 的合理值，因为流输入操作在试图接受变量 a 的输入时，发现数字串超过整型数的表示范围。这时候，a 变量的值将不可预料，不同的编译器将作出不同的输入处理。或许输入处理只涉及如何截断数字串，可能影响之后的数据读入起点，而不会因此中断程序运行。

3.1.4 整型操作

1．算术操作

整数可以进行+，-，*，/，%的算术运算。进行算术操作时，必须注意结果是否溢出。整数的算术运算的溢出其实也是有规律可循的（☞CH3.1.5 去掉进位规则）。

其中“/”的操作是整除运算，得到的结果为商（整数）；“%”的操作是取余运算，得到的结果为余数（整数）。例如，9/5 得到 1，而 9%5 得到 4。“/”和“%”都涉及除法操作，都规定除数不能为 0，否则将导致运行错误。

值得一提的是，取余操作结果的正负性取决于被除数的正负性，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
11%（-5）=1    -11%（-5）=-1
```

2．关系与逻辑操作

◆ **逻辑值与整数值互通**

整数可以进行“!”，“&&”，“||”的逻辑操作。逻辑操作的结果值要么为 **true** 要么为 **false**，没有其他可能。**true** 和 **false** 与 1 和 0 对应并且相容，所以逻辑值可以看做整数值 0 和 1。将逻辑值与整数值等同看待，这是 C/C++处理逻辑值的一个通融设计，它带来了许多计算优化和精彩的代码精简。因为整个逻辑学的完备计算体系，统统可以搬到 C++程序设计中来，给编程带来多学科交叉描述、理解，甚至找到解决问题的通同性，从中也折射出相关的哲学意义。

逻辑值与整数值互通，必须将整数值也看做逻辑值：任何非 0 数（包括正数负数）等价于逻辑真，即当一个非 0 整数作为逻辑值运算时，其逻辑值为 1。显然，整数值 0 便对应逻辑值 0，这是一对一的对应。整数值 0 即逻辑值 0。

◆ **关系操作**

整数可以做“<”，“<=”，“>”，“>=”，“==”和“!=”的关系操作。关系操作的结果是逻辑值，要么 0 要么 1，没有其他可能。所以可以将关系操作的结果：

（1）作为条件判断；

（2）作为逻辑值参加逻辑运算；

（3）作为整数值参加整数运算。

◆ **逻辑操作**

逻辑运算中，“!”表示“逻辑非”操作，也就是说，任何非 0 整数值经过“!”操作，都变成了 0，而整数 0 经过“!”操作变成了 1。“&&”操作是“逻辑与”操作，“||”操作是“逻辑或”操作，与数理逻辑中的“非”、“与”、“或”操作对应。例如，23&&5 即 1&&1 得 1，0||1 得 1。

逻辑学有真值逻辑。例如，A，B 代表某个逻辑值，则!A&&B||A&&!B||A&&B 等价于 A||B，所以下列两条语句等价：

```
if( !A&&B||A&&!B||A&&B) x=12;
if(A||B) x=12;
```

从中可以看出编程中利用逻辑表达式的真值等价性，可以进行计算的优化。

3．位操作

整数可以进行“<<”，“>>”的移位操作和“&”，“|”，“^”，“~”的按位逻辑操作。

在做移位操作时，将整数的内部二进制数表示看做一个 01 序列串。对于左移 1 位操作，即将最左边的 0 挤走，而在右端添上 0。右移则反之。例如：

```
int a = 12;      //a 为 00000000000000000000000000001100
int b = a<<1;    //b 为 00000000000000000000000000011000
int c = a>>3;    //c 为 00000000000000000000000000000001
```

b 的值为 24，二进制数左移 1 位相当于做乘 2 运算。

c 的值为 1，二进制数右移 1 位相当于做除 2 运算。因为原来右边的 3 位都被挤走，而整数的左边补 0。

移位操作在处理负数的右移时，挤走若干位，但却补上若干位 1 而不是 0，这是考虑到有符号数的符号一致性。例如：

```
int a = -12;     //a 为 11111111111111111111111111110100
int b = a<<1;    //b 为 11111111111111111111111111101000
int c = a>>3;    //c 为 11111111111111111111111111111110
```

这时候 b 的值为-24，而 c 的值为-2。

位逻辑操作是按位做逻辑操作。“&”表示“位与”操作；“|”表示“位或”操作；“^”

表示“位异或”操作，也就是不进位加；而“~”表示“位反”操作。位反操作是一元操作符。例如：

```
int a = 12;     //a 为 00000000000000000000000000001100
int b = 5;      //b 为 00000000000000000000000000000101
int c = a|b;    //c 为 00000000000000000000000000001101
int d = a&b;    //d 为 00000000000000000000000000000100
int e = a^b;    //e 为 00000000000000000000000000001001
int f = ~a;     //f 为 11111111111111111111111111110011
```

可得 c 为 13，d 为 4，e 为 9，f 为-13。

4．赋值操作

赋值操作除了直接赋值的“=”操作之外，还有累积性赋值操作。例如， +=，-=，*=，/=，%=分别表示累加、累减、累乘、累除，累取模操作；还有^=，&=，|=分别表示对位异或、位与、位或操作的结果进行赋值。再有<<=和>>=表示对左移和右移的结果进行赋值。

累积性赋值操作的意义是清楚的。例如，“a+=6;”表示“a=a+6;”。

因为赋值操作涉及一个变量实体的值修改，所以左边必须是一个可以被修改的实体。（☞CH5.3.4）。

3.1.5 整数问题

1．周而复始性

◆ **去掉进位规则**

讨论整数在计算机内只有 8 位的形式，假想其类型为 int8，并默认其为 **signed**，分别创建 **signed** 和 **unsigned** 的变量如下：

```
int8 a1=255，a2=3;
unsigned int8 b1=255, b2=3;
```

显然，a1、a2 的表示范围为–128～127，而 b1、b2 的表示范围为 0～255。有符号和无符号整型的表示范围之间有一个交集。整数 255 超过了 a1 的表示范围，如果显示 a1，会是什么数呢？如果 b1+b2 则又得到什么呢？

试想一下时钟的变化规律。9 点钟过了 5 个小时，我们一般会说 2 点，而不说 14 点。即使说 14 点，那也是特指下午的 2 点。否则，9 点过了 50 个小时，说成 59 点就是笑话了。9 点过了 50 个小时是几点呢？只要 59 去掉若干个 12 就能得出结果是 11 点。因此根据走过的时数，可以得到钟点数的计算为：

（起始钟点 + 走过的时数）取 12 的余数

二进制 8 位补码运算中，255+3 也类似钟点规律，简单地抛弃最高位的进位（取 2^8＝256 的余数），而获得结果为 2。

因为 255 没有超过 int8 的内部表示范围，所以其值就直接赋给了 a1，即使一个合法整数超过了 int8 的表示范围，也可以通过取 256 的模而获得 a1 的赋值。例如，a1=257，其实 a1 真实的赋值是 1。

因为 a1 的类型是有符号整数，当 a1 在输出时，就得到-1 了。因为最高位是 1，在输出时将会通过取补，表现有符号类型；而 b1 的类型是无符号数，当输出 b1 时，就得到 2 了。它也是自身类型合适的输出。

◆ **混合计算**

由于变量 a1、a2、b1、b2 都是 8 位整型，所以彼此之间的计算，都在 8 位整型数中展开。无符号类型和有符号类型之间的混合计算，不管在不在两个类型的交集中计算，编译器都将会报告警告信息。为了安全计算，需在两者的交叉区域中进行，以确保不会溢出其外。例如，string 类型中描述字串长度的 length()操作是 **unsigned** 整型结果。因此在写下列 for 循环控制时，会看到一则警告：

```
string s="HelloWorld";
for(int i=0; i<s.length(); i++) //警告: 有符号与无符号数混合运算了
//…
```

因为有符号数变量 i 是从 0 开始增量变化直至一个非 0 正整数，没有取负数的可能，有符号与无符号数混合比较运算始终在其交叉区间中进行，所以这个循环控制的区间描述是安全的。

任何其他的整型，都有相似的规律。无符号数和有符号数在计算时都是以 2 的位数幂为模（例如，int 型以 2 的 32 次幂为模）来存放计算结果的。如果有相互的混合运算，并不因为溢出而显示错误，只在最后根据其有符号或无符号类型来表示其输出。有符号类型输出时将先判断该整数最高位是否为 1，以决定是否输出一个负数。

2．表示范围的影响

◆ **整数局限性**

整数的表示范围决定了其表示的局限性。例如，阶乘的计算：

```
//=====================================
//X0301.cpp
//阶乘计算
//=====================================
#include<iostream>
using namespace std;
int main(){
  for(int i=1,t=1; i<18; t*=i++)
    cout<<i<<""<<t<<"\n";
}//=====================================
```

得到其运行结果：

```
1 1
2 2
```

```
3 6
4 24
5 120
6 720
7 5040
8 40320
9 362880
10 3628800
11 39916800
12 479001600
13 1932053504
14 1278945280
15 2004310016
16 2004189184
17 -288522240
```

阶乘本可以按规则一直计算下去，但是由于整型数表达范围的限制，使得其在 13!的时候就不正确了，甚至表面上看，13!，14!，15!，16!都是一个个用肉眼无法立即判断的大正数，直到在 17!出现负数了，才确认计算结果溢出了。回过头一个个排查，才发现早在13!的时候，结果就已经错了。

上述错误的发现毕竟是直观的，但要等到发生好几个错误，明显观察到情况不对之后。

◆ **输出整数二进制位码**

我们修改一下程序，用 64 位和 32 位整型数同时输出阶乘计算结果来观察整数中的内在规律：

```
//=======================================
//X0302.cpp
//阶乘计算修改版
//=======================================
#include<iostream>
#include<iomanip>
using namespace std;
int main(){
  long long int t=1;
  for(int i=1,u=1; i<18; t*=i,u*=i)
  {
    cout<<setw(2)<<i++<<" ";
    for(int j=63; j>=0; j--)
      cout<<(t>>j&1);

    cout<<"\n"<<setw(18)<<t<<setw(14)<<u<<"   ";
    for(int j=31; j>=0; j--)
      cout<<(u>>j&1);
    cout<<"\n";
  }
```

```
}//====================================
```

得到其运算结果：

```
 1 00000000000000000000000000000000000000000000000000000000000000000000000000000001
                 1             1    00000000000000000000000000000001
 2 00000000000000000000000000000000000000000000000000000000000000000000000000000010
                 2             2    00000000000000000000000000000010
 3 00000000000000000000000000000000000000000000000000000000000000000000000000000110
                 6             6    00000000000000000000000000000110
 4 00000000000000000000000000000000000000000000000000000000000000000000000000011000
                24            24    00000000000000000000000000011000
 5 00000000000000000000000000000000000000000000000000000000000000000000000001111000
               120           120    00000000000000000000000001111000
 6 00000000000000000000000000000000000000000000000000000000000000000000001011010000
               720           720    00000000000000000000001011010000
 7 00000000000000000000000000000000000000000000000000000000000000000001001110110000
              5040          5040    00000000000000000001001110110000
 8 00000000000000000000000000000000000000000000000000000000000000001001110110000000
             40320         40320    00000000000000001001110110000000
 9 00000000000000000000000000000000000000000000000000000000000000010110001001100000
            362880        362880    00000000000001011000100110000000
10 00000000000000000000000000000000000000000000000000000000000001101110101111100000000
           3628800       3628800    00000000001101110101111100000000
11 00000000000000000000000000000000000000000000000000000000000100110000100010101000000000
          39916800      39916800    00000100110000100010101000000000
12 00000000000000000000000000000000000000000000000000000000000111001000110011111100000000000
         479001600     479001600    00011100100011001111110000000000
13 00000000000000000000000000000000000000000000000000000000000101110011001010001100110000000000
        6227020800    1932053504    01110011001010001100110000000000
14 00000000000000000000000000000000000000000000000000000000001010001001100001110110010100000000000
       87178291200    1278945280    01001100001110110010100000000000
15 00000000000000000000000000000000000000000000000000000000010011000001110111011101110101100000000000
     1307674368000    2004310016    01110111011101110101100000000000
16 00000000000000000000000000000000000000000000000000000001001100000111011101110111010110000000000000000
    20922789888000    2004189184    01110111011101011000000000000000
17 00000000000000000000000000000000000000000000000000000101000011011111110111011101100110110000000000000000
   355687428096000    -288522240    11101110110011011000000000000000
```

◆ **代码解读**

代码中 t 是 64 位整型变量，初始化为 1，用来求正确的阶乘值以与 32 位整型数作比较。外层循环中的 u 是 32 位整型变量，同样初始化为 1，用来求阶乘值，但因为表示范围有限，会发生错误。t 与 u 都在循环的步长变化部分中计算下一个阶乘值，注意 t 与 u 的表达式之间用逗号隔开而不是分号。**for** 循环描述中的分号只能用两个，且只能用来分隔循环描述，不允许他用。逗号可将多个表达式并成一个表达式，这里只是对 t 和 u 求值。

由于将阶乘的计算（t 和 u 的计算）放到了循环步长变化部分去做了，所以循环事实上的主体是输出处理，而不是计算。循环角色的这种灵活多变性，是编程语言所赋予的特征。富于创造性地应用让编程充满乐趣，也将获得更多的代码灵巧性。

内循环中的两个 **for** 循环分别是 64 位和 32 位按二进制输出整型数，并通过输出格式控制进行上下个位数对齐，以便进行观察。整型的二进制数输出中，用到了“移位”操作和“位与”操作。这里取整型数某一位的操作，是将该位移到个位，再做与 1 的“位与”操作，屏蔽其他各位，同时得到该位 1 或 0 的值。

移位操作和位与操作有优先级（☞CH5.2.5 优先级）的区分。移位操作优先级高，事实上也希望先做移位后做位与，所以 t>>j&1 直接作了表达。但是作为流输出的<<操作，结合性或优先级高于后面的移位和位与操作，所以需要将取整数某位的表达式用括号括起来。

◆ **影响算法**

通过运行结果比较，看到在 i 等于 13 时，64 位整数的二进制码在高 32 位上开始不为全 0。也就是说，32 位整数在计算 13!时，中间结果开始溢出。所得到的输出，实际上是低 32 位二进制数值。13 后面的所有结果都一样。至于在 17!阶乘时出现负值，那是因为低 32 位计算中正好使 32 位最高位为 1，而所有其他超过 32 位的进位都被切除。这就是整型数在处理中间结果溢出的内在规律。

也就是说，计算 18!，需要用 64 位整型。如果要计算更大数的阶乘，则需要用其他的数据表达手段。

算法总是描述一般的方法，而编程总是要考虑语言中数据描述的局限性。

3．中间结果溢出

溢出的情况在整数类型的计算中如何表现呢？例如，在 **short int**（2 字节长）中计算时：

```
short int a=234;
short int b=456;
short int c=6;
cout<<a*b/c;        //结果 17784 正确
```

但要看到，虽然 **short int** 为 16 位整型，但由于运算过程是在编译器所选定的计算机运算器即 CPU 的字长中进行的。如果编译器指定的指令系统是 32 位，那么就会放在 32 位的整数寄存器中进行运算，所以中间运算结果 234×456＝106704 不会溢出，最后结果就能正确表示。

选择编译代码的机器指令（16 位抑或 32 位），也就选择了整数运算过程在哪一个精度上进行。上述例子若生成 16 位机器代码，则结果将不正确：

```
a*b/c =234*456/6
      =106704/6
      =1,1010000011010000/0000000000000110  //头上一位为进位
      =1010000011010000/0000000000000110
      =-24368/6
      =-4061
```

因为在 16 位的机器上运算，将抛弃 a*b 的 16 位上的进位。即使 a、b、c 都是 **int** 型整数，而非 **short int**，结果同样错误，因为 16 位编译器的 **int** 为 16 位长。只有当 a、b、c 是 32 位 int 型整数时，结果才正确。

下列程序的运行结果，虽然乘除法的交换律成立，但因其中的一个有中间结果溢出而致错：

```
//=====================================
//X0303.cpp
//int 的中间结果溢出
//=====================================
#include<iostream>
using namespace std;
//-------------------------------------
int main(){
  int a = 100000;
  int b = 100000;
  int c = 1000;
  cout<<a*b/c<<"\n";
  cout<<a*(b/c)<<"\n";
}//=====================================
```

其运行结果为：

```
1410065
10000000
```

3.2　整 型 子 类

3.2.1　字符型

子类的概念首先是与父类具有相同的“处事原则”（具有相同的操作，此外允许个别操作有自身特点），其次是具有相同的基本表示。

字符类型简称字符型。因为它的存储和计算模式也是二进制补码，所有的操作与整型一样，包括+，-，*，/，%，<<，>>，|，&，~，^，++，--等，只不过字符值的表示范围比整型数小而已。

字符值的取值范围取决于字符值的表示长度，通常由编译器决定。标准 C++没有规定字符型的长度，通常所见的编译器是用 8 位（1 个字节）表示字符型长度的，也是因为在我们的学习环境中，没有针对专业用途的大型机以及属下的 C++编译器，本书所描述的字符型（不加 **signed** 或者 **unsigned** 修饰）由一个字节（8bit）构成。字符值根据有符号数与无符号数不同而取范围-128～127 和 0～255。字符值除了表示以规定范围的整型数外，还

可以表示成专门的字符字面值。

字符在输入/输出时的表现，与整型数不同，这是字符型的操作区别于整型的地方。字符还是字串的元素，字符型是字串处理的基础。

1．字面值

字符字面值简称字符，而字符值是字符所对应的整数值。字符型的每个字符不管有形无形（可见与不可见）都对应一个整数，以此方式与整数操作保持兼容和互通。

C++的语言文字即语言规则所允许的文字，需要用编译器进行理解或检查，其取自 ASCII 码表的可见字符集的一个子集。C++语言随着发展或许会允许编译器识别更多的字符甚至多国文字。

程序运行中当做数据的输入、输出、处理的字符，即字符型所用的字符集，远比 C++语言文字所用字符集要广。当有特殊字符处理需要时，还可以使用宽字符型。这里所述的字符型所用的字符集，为常用的 ASCII 码表所列字符。

ASCII 码字符构成字面值，它直接用单引号括起来以表示字符。例如，'a'，'x'，'?'，'$'等字符。

ASCII 码有 128 个字符，其中 ASCII 码值从 0 到 31 和码值 127 为不可见字符。不可见字符也称控制字符。ASCII 码表的可见字符可以通过下列程序打印出来：

```
//=====================================
//X0304.cpp
//输出 ASCII 码可见字符
//=====================================
#include<iostream>
#include<iomanip>
using namespace std;
//-------------------------------------
int main(){
  for(char c=32; c<=126; c++)
    cout<<setw(5)<<int(c)<<" "<<c<<((c%8)==7 ? "\n":"");
}//=====================================
```

其运行结果为：

```
   32    33 !   34 "   35 #   36 $   37 %   38 &   39 '
   40 (   41 )   42 *   43 +   44 ,   45 -   46 .   47 /
   48 0   49 1   50 2   51 3   52 4   53 5   54 6   55 7
   56 8   57 9   58 :   59 ;   60 <   61 =   62 >   63 ?
   64 @   65 A   66 B   67 C   68 D   69 E   70 F   71 G
   72 H   73 I   74 J   75 K   76 L   77 M   78 N   79 O
   80 P   81 Q   82 R   83 S   84 T   85 U   86 V   87 W
   88 X   89 Y   90 Z   91 [   92 \   93 ]   94 ^   95 _
   96 `   97 a   98 b   99 c  100 d  101 e  102 f  103 g
  104 h  105 i  106 j  107 k  108 l  109 m  110 n  111 o
  112 p  113 q  114 r  115 s  116 t  117 u  118 v  119 w
  120 x  121 y  122 z  123 {  124 |  125 }  126 ~
```

for 语句通过字符变量 c 从 32 到 126 的循环，这正好是全部 ASCII 码的可见字符（包括空格字符）。程序每次输出一个 ASCII 码（整数）和其对应的字符，并控制每行输出 8 个 ASCII 码和所对应的字符。例如，大写字母字符'A'表示整数 65，而小写字母字符'a'却表示整数 97。而且看到空格也是一个字符，其对应的 ASCII 码为 32。

一开始可能不习惯的是，数字字符'3'的 ASCII 码是 51，所以若用数字字符值与整数进行运算，例如，'5'*2 得到的是 106 而不是 10。这也正好说明数字与字符值的不同性。

2．转义符

◆ **概念**

编程是输入一些符合语言文法的文字字符，那都是些可见的 ASCII 字符。编程应该能够接受一些不可见的 ASCII 码控制字符，以便满足按指定格式输出的需要。换行符“\n”作为不可见字符，前面的章节已经在用。

除了那些可见 ASCII 字符之外，C++还允许使用一种特殊形式的字面值，用以表达不可见的 ASCII 字符。这种特殊形式的字面值以“\”（称为转义符）开头，后跟一个格式字符，称为转义字符（escape character）。例如，换行符就是一个转义字符，其格式字符为 n。

◆ **转义字符表**

使用转义字符的目的是在编程中描述不可见字符。并不是每个控制字符都有其格式字符，是有些常用的控制字符才有其转义形式。但是可以通过 ASCII 码，描述所有的控制字符，方式就是在转义符后跟十六进制 ASCII 码。表 3-03 描述了常用的转义字符与 ASCII 码十六进制数的对应，并描述了该字符所控制的动作。

表 3-03　C++转义字符

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

◆ **解释**

有时候需要单独输出转义符“\”自身。由于字符“\”已被用作为转义符，所以“\”后面跟上一个字符可能会被看成是一个控制字符的转义字符。为了解决在代码中单独表示字符“\”的问题，用双写字符（即“\\”）的形式来表示单独的“\”字符。例如，"or\\and"与"or\and"完全是两个意思，前者表示字串“or\and”，后者表示字串“or+响铃+nd”。

字串用双引号括起来，字符用单引号括起来，这是C/C++语言的规则。可是，为了描述字串中所含有的双引号，或者单独描述一个单引号字符，我们就陷入了困境。例如，字串中含有双引号，"I say, "good""就不能正常理解。为了解决这个问题，在描述双引号和单引号字符时，用转义符来描述，即“\"”和“\'”，而在作为字串的边界或字符的边界时，则无须转义，直接表示。

转义字符的双字符特征，决定了一对单引号除了括起一个字符，表示一个字符字面值外，还可以括起含两个字符的转义字符，来表示独立的字符字面值。例如，“\n”只是一个字符字面值。

在转义字符表中没有出现的控制字符，也有可能会需要描述，这时候就会用到转义符后跟ASCII码（整数）的形式。例如，要描述ASCII码为6的控制字符，可以写成“\06”。

ASCII码的可见字符也可以用转义字符的形式表示。例如，'A'的ASCII码为65，65的八进制数为101，十六进制数为41，所以可用两种数制来表示字符'A'的转义形式：“\0101”或“\x41”。

3．输入/输出

◆ 输入

若用字符变量去读一个字符，那么这个变量值就是该字符所对应的ASCII码值。

因此，如果用字符变量去读一个数字字符，那么这个变量值就是该数字字符的ASCII码值；如果用整型变量去读一个数字字符，那么，这个变量值就是这个数字值。例如：

```
char a;
cin>>a;              //输入6得到字符'6'即54
int x;
cin>>x;              //输入6得到整数6
cout<<a*2<<"\n";     //输出108
cout<<x*8<<"\n";     //输出48
```

数字字符与数字值（整数值），在字符或字串处理中经常需要转换。例如，一个string字串变量赋有一个数字串"123"，希望通过下标访问每个字符，转换为整数123。则有：

```
string a="123";
int x = (a[0]- '0')*100+(a[1]- '0')*10+a[2]- '0';  //x得到123
```

a[0]字符为'1'，'1'-'0'便得到1，同样，'2'-'0'得到2，'3'-'0'得到3，因此通过上述表达式计算，就可以将字串"123"转换成整数123了。

◆ 输出

按字符型的长度为1字节算，它只能表示256个状态值。由于ASCII码有128个字符，所以可以用所有正数表示所有ASCII码值，而负数便可用来表示各种非正常状态。当然在进行存储和计算的时候，字符值可以是所有正负状态之一。例如：

```
int c = '\a';
char b = 5;
char a = 10;
```

```
a -= c;
b -= c;
```

则 c 应该为 7（从转义字符表得）；计算结果，a 应该得到值 3，这个值对应的 ASCII 码是一个不可见字符；b 应该得到值-2，这个值不在 ASCII 码所能表示的范围内。a 和 b 这种取值，如果以字符打印，因为没有对应 ASCII 字符，因而达不到输出效果。当然，有时候显示的是一个中文字，如果系统中安装了汉字系统的话。

整数作为抽象的数字形式表现，无所谓取值的对与错。而字符如果要进行输出表现，其值只能为正数，以代表特定的 ASCII 码。而且若是不可见字符的话，还必须保证有控制意义。因为字符型的输出并不是输出整数，而是该字符值所代表的 ASCII 码字符。例如：

```
int  a = 67;
char b = 67;
cout<<a<<""<<b<<"\n";  //值虽都为 67,但其结果一为整数 67，一为字符 C
```

这就是 CH2.3.1 中的问题“字符三角形”的解答中，为什么输入一个字符给字符变量，而输出时显现的是字符而不是整数的原因了。

cout 流能区别类型。如果是整型，则输出的是整数，如果是字符型，则输出的是 ASCII 码（整数）所对应的字符。

4．可移植性

值得注意的是，有些地方和机器环境用的不是 ASCII 码（ASCII 码并不是语言文字所使用字符集的终极标准），结果，字符型的整数值可能对应另一种码表。因此，为了代码可移植，不要用整数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，下面代码输出一个腰长为 10，每行依字母序展开的三角形：

```
//=====================================
//X0305.cpp
//字母三角形
//=====================================
#include<iostream>
using namespace std;
//-------------------------------------
int main(){
  for(int i=1; i<=10; ++i)
  {
    cout<<string(10-i, ' ');
    for(char ch='A'; ch<'A'+2*i-1; ++ch) //输出 ABC…共 2i-1 个字符
      cout<<ch;
    cout<<'\n';
  }
}//=====================================
```

其运行结果为：

```
          A
         ABC
        ABCDE
       ABCDEFG
      ABCDEFGHI
     ABCDEFGHIJK
    ABCDEFGHIJKLM
   ABCDEFGHIJKLMNO
  ABCDEFGHIJKLMNOPQ
ABCDEFGHIJKLMNOPQRS
```

代码中的内循环，完成输出 2i-1 个字母字符的工作，其循环变量 ch 是字符型的。ch 的取值为从'A'到'A'+2*i-1。如果将'A'写成 65，也可以工作，但是，并不一定在任何计算机环境上运行都能得到同样的结果。原因是字符'A'的整数码 65 并不构成计算机字符表示的标准。所以循环如果写成下列这样，其可移植性就差：

```
for(char ch=65; ch<64+2*i; ++ch) //输出 ABC…共 2i-1 个字符
  cout<<ch;
```

3.2.2 枚举型

1．枚举型

枚举是对整数区间取若干个整数值从而框定整数子集的一种自定义机制。所匡定的整数子集称为枚举类型，简称枚举型。定义枚举型时以关键字 **enum** 引导，枚举型的名称可以自定义，程序员还可以对框定的整数子集中的整数值取名。例如，下列是一个枚举型定义：

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

enum 为关键字，引导枚举的类型定义。其枚举型的名称为 Week。一对花括号中各个值即是从整型数中枚举出来的若干个值。这便定义了一个整型数的子集类型，类名称为 Week，其全部值为 Mon，Tue，Wed，Thu，Fri，Sat，Sun。分别表示整数 0，1，2，3，4，5，6。

2．枚举符

◆ **概念**

枚举的值必须用名字表示，所取的名字要防止与其他变量名或类型名冲突。

枚举值所用的名字即为枚举符。

在定义枚举型时，需要用花括号具体描述整型数的子集。描述的方式，便是用枚举符来表示枚举值。枚举符是自定义的名字。在一个枚举类型定义中，一个枚举符只能代表一个整数值。一个整数也最多只能由一个枚举符表示。枚举符与整数是一一对应的关系。

◆ **枚举值**

上面定义的 Week 枚举型，定义了 Mon，Tue，Wed，Thu，Fri，Sat，Sun 这么几个枚

举符。因为没有指定整数，所以就默认为：第一个枚举符对应整数 0，第二个枚举符对应 1，以此类推。因此，在 Week 中，Mon=0，Tue=1…Sun=6，一共由 7 个整数值构成了一个整数子集，并且这个整数子集中的每个元素都有相应的枚举符。

名字（枚举符）与整数的对应可以不默认，也就是人为规定。例如：

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

则在 Color 枚举型中，名字 Red 被指定对应整数 5，Green 则顺延对应整数 6，Yellow 对应 7。之后的 Blue 又人为指定为整数 20，则 Orange 便顺延对应整数 21。也就是说，没有指定整数值的枚举符，按上一项的值，依次默认下推。所以 Red 指定整数 5 之后，Green 没有指定值，便默认为 6。

3．枚举实体

◆ **枚举变量**

有了枚举型，便可以定义枚举变量。枚举变量一般都取该枚举型整数子集内的枚举符。枚举变量与整型数的操作兼容。例如：

```
Week day=Sun;
Color color=Green + 1;  //Yellow
```

一般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。

◆ **枚举操作**

值得注意的是，枚举型虽然是类型，但在枚举型定义中已经规定了若干个代表整数值的枚举符。枚举符可以直接代表对应的整数值参加整数计算。因而枚举变量的定义并不是必须的。例如：

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

4．枚举设计意图

◆ **定义枚举型**

枚举型定义的并不是一个具有闭运算的自成体系的像整型这样的子代数。编程中已经有了整数类型，便不需要这样的子代数了。枚举型定义的是若干个与整型数对应的枚举符。通过名字指认的整数值，能够剥离整数的抽象性质。例如，“d = Mon;”与“d = Green;”与“d =1;”这三者通过枚举符的使用与否，能够区分变量 d 操作的意义，即从中可以知道 d++是在日期上消逝一天，还是在颜色上加浓一些，还是仅仅做数量上的抽象变化。

◆ **看重枚举符**

枚举符一旦定义则在程序运行的全程中不能改变，所以它常常代替整型常量使用，这才是设计枚举的真实目的。也就是说，寥寥几个枚举符，代表几个经常使用的整数，可以脱离枚举型变量定义而与其他整数进行比较和运算。

有时候枚举符甚至比整型常量还管用。因为常量通常需要初始化的形式，需要实体空间。所以，模块之间切换（例如，函数调用）时，其值也就需要传递。而枚举符是在编程中确定与整数的对应，在编译中（而不是在运行中）确认，并在运行之前枚举符已经转换为整数，因此在运行中，枚举符代表的整数无须去对应一个可存放的内存实体。

3.2.3 布尔型

1．概念

整数 1 和 0 两个值构成了 **bool** 型的表示范围。相当于自定义了枚举型：

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

注意，**true** 和 **false** 都是关键字，可以认为是 C++语言定义的枚举符。因而程序员如果再定义这两个名字，就会与关键字发生冲突。

bool 型是只有两个整数的类型，其范围窄了一些，但是用它表示逻辑 **true** 和 **false**，却可以表达千千万万的真假命题和逻辑表达式的值。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）的结论都用 **bool** 型值来表示。

2．布尔值操作

◆ **整数与布尔值对应**

用任何非 0 整数给 **bool** 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(与 1 对应)
bool c = a+a;   //c 为 true ( 1+1=2，2 表示为逻辑值即为 1)
bool d = a-c;   //d 为 false (值为 1 的 a - 值为 1 的 b，得 0)
```

◆ **布尔值的逻辑操作**

bool 型变量值的“&&”操作、“||”操作、“!”操作所得的结果，也是逻辑值。例如对于上面的 **bool** 型变量定义：

```
c = a && d;
d = !a;
if( a||d ) cout<<"OK\n";
```

◆ **布尔值的输出**

bool 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<<"\n";   //输出结果为 false
```

3．布尔型的意义

◆ **逻辑的专门表达性**

编程无时无刻不在与真假条件或 0、1 状态打交道。我们经常会设计一个函数或者一

个表达式，通过计算来获得真假的结论，以便决定后续工作进行的方式。因此，布尔型值的描述带给我们一种专门区别于数据处理和数值计算的数据类型。

◆ **成就布尔代数思想**

布尔型虽然其中只有两个值 **true** 和 **false**，但它其实蕴含了二进制数的计算思想，也是一个布尔代数体系的缩影。

从布尔代数体系知道，理论上，布尔代数通过自己的封闭运算系统，可以表达一切数学模型、演算和定义一切数学问题。所以，布尔类型不但带给我们一种表达真假判断的特定性，还带给我们一套逻辑计算系统。只要能够建立起一套构造计算的方法，就能够用来辅助解决单纯靠数值计算遇到的复杂问题。

布尔代数体系与计算机内部的构造密切相关，与计算表达密切相关。其机械开关的二态性和逻辑可运算性，使得计算机可以被看做是一台布尔代数的演算机器，能够进行充分复杂的计算。

◆ **整数的布尔化**

在 C/C++中，布尔值与整型值是相容的。也就是说，**true** 和 **false** 可以等价于整数 1 和 0。更进一步，整数可以作为逻辑值参加运算。例如，“a=3&&5; ”则 a 的赋值为 1。

整数与逻辑值的通同性，使得整数与逻辑值融为一体，逻辑值成为整数的子集，编程中自然成全了许多谓词演算（即返回布尔值的函数）的设计，而许多逻辑计算又可以演变成为二进制整数的数据处理，从而又导致许多代码的优化。

3.3 浮　点　型

3.3.1　浮点型数

1．编程中的描述

◆ **浮点型数概述**

浮点型数就是浮点字面值，简称浮点数。浮点数在表示的时候，有直接表示和科学表示两种形式。

浮点数按表示能力，或按长度又分 **float**，**double**，**long double** 三种。

浮点数在表示绝对值大小方面的能力比整数强很多，所以很多宏观和微观数据的计算处理需要浮点数。但是浮点数因为大小表示能力强了，其精度表示就比同等长度的整型数差一些了，而且一般认为，浮点数的运算性能比整型数差一些。

除此之外，浮点数还有近似表达的问题，所以编程精确输出浮点数问题、判断浮点数之间的相等问题要复杂一些；计算机浮点数内部表达原理比整型数也要复杂一些；计算过程中，计算机要判断处理溢出和无穷大问题，而且运行错误的情况分类也要比整型数复杂一些。

浮点数的表示分输入识别的浮点数、输出表达的浮点数以及编程中表达的浮点数

三类。

◆ **直接表示**

编程中对浮点数的直接表示就是描述一个有限长度的十进制小数。

因为直接表示的小数一定带有小数点，所以区别于整数。小数点前若无有效数，可以省略数字。但是小数点后若无有效数，则不能省略数字 0，否则就成了非法表示。如果连小数点也省略，就成了整型数。用整型数赋值给浮点型变量，就会导致编译器的类型转换工作。例如：

```
double a = .92;
double c = 9.;          //错误: 应为 9.0,不能省略 0
double b = 9;           //正确: 编译器将把整型数 9 转换成浮点数
```

这些规则细节适合于三种不同的浮点型。

◆ **科学表示法**

用科学记数法表示浮点数需要在小数描述的基础上增加阶码描述。

- 小数与阶码中间必须要有字母 E 或 e 将两数隔开；
- 小数可以是整数，也可以是直接表示的小数；
- 小数作为数值表示的主体，不能省略；
- 阶码是整数，可以有前导 0，可以有正负号，但不能是小数。

例如：

```
double a = E15          //错误:  不允许省略尾数部分，简直是变量名了
double b = 1e0.5        //错误:  阶码应为整数
double c = .e05         //错误:  尾数应为合法小数，不能仅以“."表示
double d = .2e000105    //正确
```

◆ **浮点数后缀**

浮点字面值是有类型区别的，它们由后缀决定：

（1）后缀为 F 或 f：表示单精度（**float**）；

（2）后缀默认（无后缀）：表示双精度（**double**）；

（3）后缀为 L 或 l：表示长双精度（**long double**）。

例如：

```
float f1 = 19.2f;           //float 型字面值
double d1 = 19.2;           //double 型字面值
long double ld1 = 19.2L;    //long double 型字面值
```

用大写和小写字母作为后缀效果是一样的。显然，如果将高精度的数赋给低精度类型的变量，将引起精度丢失；反之，如果将低精度的数赋给高精度类型的变量，则高精度变量也只有低精度的数值（☞CH3.3.5 精度丢失）。

要注意这两个浮点数的表示：12345F、12345.0L。

12345 虽然是整型数，但是后面加了 F 标记，所以表示为 **float** 型数。

一个数尾部加了 L 标记，在整型数中表示长整型，在浮点型数中表示长浮点型，所以

L 标记之前的数为整型或浮点型，就决定了该数为长整型或长浮点型数。为了表示长浮点型数，后缀之前的数必须为小数，以示与整型数的区别。

对于用科学表示法表示的浮点字面值，其后缀 F 或 L 表示与用浮点数直接表示的意义相同。

◆ **精确性的类型制约**

小数的直接描述的准确性受到类型的制约。例如：

```
//=====================================
//X0306.cpp
//test float format
//=====================================
#include<iostream>
#include<iomanip>
using namespace std;
//-------------------------------------
void print(float f){
  cout<<f<<"\n";
}//-------------------------------------
void print(double f){
  cout<<f<<"\n";
}//-------------------------------------
int main(){
  cout<<fixed<<setprecision(8);
  print(123456.7890123);
  print(123456.7890123f);
}//=====================================
```

得到的输出结果为：

```
123456.78901230
123456.78906250
```

前者是将浮点数以 **double** 类型输出，后者是以 **float** 类型输出，它们都设定为固定小数输出形式，小数精度设置为 8 位。从中可以看出两者在表示能力上的差异。

2．输入描述

◆ **精度与类型的关系**

浮点数在输入时不接受（识别）尾部后面的 F 或 L 标记，完全依赖于浮点类型。精度高的浮点型接受（识别）位数多的浮点数。例如对于下列输入内容：

```
1234567
1234567
1234567.890123456
1234567.890123456
```

```
12345.67E-58
12345.67E-58
```

识别读入浮点数的程序：

```
//=====================================
//X0307.cpp
//test float format2
//=====================================
#include<iostream>
#include<iomanip>
using namespace std;
//-------------------------------------
void print(float f){
  cout<<f<<"\n";
}//-------------------------------------
void print(double f){
  cout<<f<<"\n";
}//-------------------------------------
int main(){
  cout<<fixed<<setprecision(8);
  float f;
  double d;
  cin>>f>>d;
  print(f);print(d);
  cin>>f>>d;
  print(f);print(d);
  cout<<scientific;
  cin>>f>>d;
  print(f);print(d);
}//=====================================
```

得到下面的结果：

```
1234567.00000000
1234567.00000000
1234567.87500000
1234567.89012346
0.00000000e+00
1.23456700e-54
```

◆ **代码解读**

程序先是设置了定点输出格式，观察 **float** 与 **double** 型数显示的效果。结果，对于输入的 7 位十进制数得到了同样的数值，但是对于有效位数达 15 位的两个数，**float** 型输出的表现明显不如 **double** 了。特别是在识别科学表示法的两个数中，因为精度跟不上，**float**

型表示只能显示 0，而 **double** 型表示就可以输出准确的输入值。

3．输出描述

◆ **输出按格式控制的规则**

输出浮点数与在代码中描述浮点数不同。

代码中描述的浮点数（浮点字面值）通过编译成为内部浮点数。它需要语言的浮点数规范，以让编译器能理解，准确无误地转换成内部浮点数。

但输出浮点数是将内部浮点数按输出格式（宽度、空位及填充字符）、有效位数、小数精度等规定进行显示。

输出浮点数时，若浮点数值有效位偏离小数点较远，将自动按科学表示法输出。

按固定格式输出时，浮点数的有效位不论偏离小数点多远，总是会将整个小数按格式全部输出示人。

按科学表示法输出时，没有后缀表示（加 F 表示 **float** 等），尾数部分也没有省略形式，阶码总是标记有正负号，阶码位数按精度不同而取 2 到 4 位不等。而且，小数部分总是按十进制数的规格化输出，即小数点前只保留一位，且该位非 0。小数点的真实位置通过阶码表示。

注意：十进制数的规格化在输出时与输入识别时不同。输入时，要求统一为只有小数点、尾数和阶码，没有整数部分，因而可以省略小数点前的那位数。当然规格化数的尾数第一位应为非 0 数字。

◆ **字面值与输出对照**

描述一个浮点字面值，将其输出，虽也为科学表示法，但其格式或许大相径庭。例如：

```
float f = 0.001256123456789E025;
double d = 0.1256123456789e0023;
cout<<f<<"\n"<<d<<"\n";
cout<<125612345.6789e0014<<"\n";
cout<<scientific<<setprecision(10)<<f<<"\n"<<d<<"\n";
cout<<125612345.6789e0014<<"\n";
```

得到的输出形式为：

```
1.25612e+22
1.25612e+22
1.25612e+22
1.2561234041e+22
1.2561234568e+22
1.2561234568e+22
```

前面三行结果表示一个有效位偏离小数点很远而自动以科学表示法输出。精度不同的变量，直接字面值都不影响输出结果。后面三行结果是指定用一定精度的科学表示法输出。从中看出，**float** 型精度表示不够，勉强凑足输出位数，但与 **double** 型数已经有所区别。

3.3.2 十进制小数

1．实数与计算机

◆ 用计算机表示小数

十进制小数表现了数学中的实数数系，人们研究高等数学离不开它，科学计算也离不开它。为了让计算机有更强的数值表现能力，十进制小数必须表示成计算机表示形式不可。

十进制小数要表示成计算机认可的形式，最主要的影响因素是宏观数字和微观数字的表达，因为它们偏离实数数系中的 0 实在太远了。如果硬来，那计算机不知道要用多少位二进制数才能表示一个真实的小数呢。

◆ 定点数表示的失败

计算机中所表示的数都是 01 序列，或者说是由二进制的位拼凑而成。要在一个固定长度的 01 序列中表示一个小数，专家们尝试用定点数的方法。

定点数是确定某一个小数位置，也就是说，固定整数部分和小数部分的长度。例如，32 位定点数用 1 位表示符号，16 位表示整数，15 位表示小数：

符号	整数	小数
X	XXXXXXXXXXXXXXXX	XXXXXXXXXXXXXXX

结果发现它的表示范围仅仅只有 2^{16}，即几万，小数精度也只有大约 5 位十进位数。

定点数若论精度，甚至还不能描述 7 位小数（3.1415926）的 π 值；论大小，连地球半径（按米算，有 6371004 米）都无法表示。定点数的表示离科学计算的目标太远，因而早早地被放弃了。

◆ 浮点数

浮点数的方法，就是专门腾出一部分固定长度的位数用来表示小数点的偏移（阶码），余下的另一部分用来表示小数部分（尾数）。这种表示方式就是实现了用计算机来表示的科学记数法。

尾数的长度决定精度的大小，阶码的长度决定一个数的数量级。专家们发现通过科学确定尾数与阶码的长度，可以既保证一定范围的精度，又能够对应某个数量级的空间描述。

这种方法所表示的数，当参加计算发生值的改变后，可以很容易地调整其小数点和阶码，使其规格化而有利于进一步的计算。这种具有小数点“浮动（左右偏移）”的特点，称其为浮点数。或者说，浮点数是科学记数法在计算机中的实现。

2．十进制小数的自身表示

◆ 科学表示法原理

十进制小数通常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，幂次（指数）3 称为阶或阶码，3065 是小数点之后的部分，也称为小数部分，更正式一点，称为尾数。其左右端非 0 数字的最长的数字序列称为有效值（significance）。无论是-306.5 还是-0.3065×10^{3}，其有效值都是 3065。对于后者，3065 恰好也是尾数（mantissa）。

相对实数来说，小数表示实数的区域实在有限，写上十几位数已经很了不起了，但是用科学记数法，用指数的形式，哪怕表示一个天文数字也可以八九不离十。用科学记数法的最大优点是可以表示一个大数，其有效位值偏离小数点左边很远，或者可以表示一个小数，其有效位值偏离小数点右边很远。指数表示数幂的形式这么好，因此被大量地用作科学计算。相应的表示形式也被称为科学记数法了。

用科学记数法来表示一个小数，其表示方式并不是唯一的，因为它也可表示成 -3.065×10^{2} 以及 0.03065×10^{4} 等等，其小数点可以左右偏移。但不管小数点怎么移动，这个数的有效值还是不变，都是 3065。只不过其尾数会随着小数点的移动而变化，或为 065，或为 03065。

用科学记数法表示的小数进行计算是很方便的。利用其小数点的左右偏移（浮动），使两数的阶码对齐，对尾数做加减法，就是两个小数的加减操作了。例如：

$$
\begin{aligned}
&0.0365\times10^{3} + 6.78\times10^{2}\\
=\ &0.365\times10^{2} + 6.78\times10^{2} \qquad \text{//向后一项对齐}\\
=\ &7.145\times10^{2}
\end{aligned}
$$

或者也可以向前者对齐：

$$
\begin{aligned}
&0.0365\times10^{3} + 6.78\times10^{2}\\
=\ &0.0365\times10^{3} + 0.678\times10^{3} \quad \text{//向前一项对齐}\\
=\ &0.7145\times10^{3}
\end{aligned}
$$

科学记数法的小数做乘除法，甚至不需要对齐阶码。做乘法时，小数相乘，阶码相加；做除法时，小数相除，阶码相减。对计算结果再按需要做一下阶码调整就可以了。例如：

$$
\begin{aligned}
&0.0365\times10^{3}\div(6.78\times10^{2})\\
=\ &0.00538\times10^{1} \ \text{//小数相除,阶码相减}\\
=\ &0.538\times10^{-1} \quad \text{//阶码调整}
\end{aligned}
$$

◆ **规格化**

为了让十进制数表示法唯一，也是让尾数与有效值统一，更希望用有限的位数表示尽可能多的有效位数，有必要将十进制小数规格化（normalization）。

十进制数的规格化是将科学表示法的小数通过阶码调整，写成小数点后第一位为第一位有效数字的方式，即小数点前只有唯一一个数字 0，小数点后第一位即排列有效数的表示方式。例如，-306.5 规格化后为 -0.3065×10^{3}。因为规格化要求小数点后的第一个数字非 0，这就使-306.5 不能写成 3.065×10^{2} 或者 0.003065×10^{5}。

规格化保证了尾数与有效位数的统一，也就保证了浮点数表示的唯一性。

小数规格化之后，便使其有效位最大化，从而可最大程度地表达小数的精度，有利于在运算中尽量保留计算的精度。

◆ **转成二进制小数**

在计算机内部，小数是以二进制位序列表示的。为了让计算机表示十进制小数，要先将其转换为二进制小数。转换的方式是将十进制数分为整数部分和小数部分分别转换。

整数部分的转换，采用“除 2 取余法”。小数部分的转换，一种典型的做法是采用“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）作为转换成的二进制小数部分的第一位；再取其余下的十进制数小数部分，再乘以 2，又取其整数部分作为二进制小数部分第二位；然后再取余下的小数部分，再乘以 2，直到余下的小数部分为 0 或者已经取到了二进制小数足够的位数。每次取的整数部分（或为 0 或为 1）按先后次序，构成了二进制小数部分从高位到低位的数字排列。例如，十进制小数 0.8125，转换成二进制小数的具体步骤为：

十进制小数乘 2		取整部分	余下的小数部分	二进制小数部分
0.8125×2	＝1.625	1	0.625	0.1
0.6250×2	＝1.250	1	0.250	0.11
0.2500×2	＝0.500	0	0.500	0.110
0.5000×2	＝1.000	1	0.000	0.1101

最后得：

$0.8125 = 0.1101_{(2)}$

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。例如，将 0.6 转换成二进制小数，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＝$0.100110011001\ldots_{(2)}$，十进制数 0.6 是个有无穷循环小数位的二进制小数。

在数学上将尾数中重复部分的数字序列两端各用一个着重号标记。例如，将转换后的二进制小数表示成：

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

计算机的表示是有限的。在计算机内，要表示上述无限循环的二进制数，只能截取到某一位，至多再考虑一下“0 舍 1 入”，也就是说，只能表示到某个精度。所以一个确定值的十进制小数转换成二进制小数，可能无法精确表示，而只能得到一个近似数。

3.3.3 32 位浮点数

1．浮点数表示

◆ 浮点数标准

浮点数的构造，是把整数部分和小数部分都看成是数字部分，称为有效数字或尾数，然后再用一个阶码（幂次）来表示数值的规模。完整的浮点数由符号位、阶码位和尾数位这三个部分组成。

32 位浮点数是计算机中比较小型而又比较典型的小数构造。搞清 32 位浮点数的构造

就能了解浮点数表示的概貌。

浮点数的表示原则，是希望用确定长的位数表示尽可能多的有效位数（精度）和尽可能大的数域范围（幂次）。32 位浮点数构造，按国际标准浮点数表示（即 IEEE 国际标准 754），是由 1 位符号位，8 位阶码位和 23 位尾数位组成的。

符号位用 1 位，以 0、1 两个状态表示一个数的正负性。它与整数补码表示中的高位符号判定是一致的，所以用第一位而不是用其他位来表示浮点数的正负性。

◆ **规格化**

十进制小数转换成二进制小数时，其表示方式是不统一的。为了使二进制小数的表示唯一，就要像十进制数规格化那样对二进制小数进行规格化。这是最终能够表示成计算机浮点数的必然途径。

二进制小数的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，手工转换 8.8125 为二进制规格化数：

$$8.8125 = 1000.1101_{(2)} = 1.0001101\times 2^3$$

要注意的是，32 位二进制规格化数，不像十进制数规格化那样要求小数点后第一位必须为非 0，而是要求小数点前面的一位必须为 1。

◆ **多出一位精度**

如果二进制浮点数像十进制浮点数那样规格化，即要求小数点后第一位非 0，那么，其小数点后第一位的非 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 1 位，并使得规格化之后的小数第一位永远为 1。

实际上，永远为 1 的那一位是没有必要的。因而规格化二进制数时，只是采用了十进制数规格化的原理，也就是固定 1 位非 0 数字的位置，其值由阶码去调整。但是根据二进制数的每位数字只有 0 和 1 的特点，固定的那 1 位一定是 1，就可以采用全部尾数挪前 1 位的办法将这 1 位省出来（将最高位从尾数中挤到小数点前面去）。因此，二进制小数在规格化后的尾数值加上整数部分总是小于 2 而又不小于 1。

◆ **0 舍 1 入**

不同精度的浮点数在尾数表达上都有一个舍入问题。就十进制来说，是四舍五入；就二进制浮点数来说，是 0 舍 1 入。因为二进制的 1 对于十进制数来说，实际上已经过半。例如，观察浮点数 19.2F 的 0 舍 1 入性：

$10011.0011001100110011001100101101001\cdots$
$= 1.00110011\dot{0}0\dot{1}\times 2^4$
$= 1.00110011001100110011001\ 1\cdots\times 2^4$　　//32 位浮点数
$= 1.00110011001100110011010\times 2^4$　　//32 位浮点数尾数“1 入”

于是得到浮点数的一个表示，尤其关注其尾数部分：

0,10000011,00110011001100110011010

浮点数尾数的最后，在浮点数尾部，其最后三位加上待抛弃的那一位，是 001,1。这个待抛弃的 1，作为二进制数中的“过半”数，实施进位的结果，尾数最后三位就变成了 010(001+1)了。

在任何浮点数做存储操作的时候都会面临舍入操作，所以，这也让我们看到了二进制浮点数的数值只能维持其邻域的模糊性，而不能保证其绝对精确性。

◆ **规格化实例**

例如，十进制数 0.6，对于一个近似的二进制数表示，转换成规格化二进制数为：

$$0.6 = 0.\dot{1}00\dot{1}_{(2)} = 1.0011,0011,0011,0011,0011,010\times 2^{-1}$$

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。

计算机在存储 32 位规格化浮点数的尾数时，抹掉小数点前面的 1；而取出该数时，则在其取出尾数的基础上再加上 1。因此 23 位尾数，加上省略的 1 位，其精度（这里的精度与有效位概念是一致的）为 24 位。

2．阶码设计

◆ **阶码与尾数的平衡**

至于阶码位，则可多可少。阶码位越多，尾数位就越少。例如，阶码若用 5 位表示，则尾数就可以用 26 位，加上符号位，拼成 32 位。阶码的 5 位只能表示 2^5=32 个幂次方值，按正负各半，可以表示二进制小数的位移 16 位，表示能力小于 10^5，即只能使十进制小数偏移约 5 个小数点位置。这对表示大数不利，但对表示高精度数有利。例如，表示常数 π 比较适合，但表示大富翁的上百亿存款就不够了。阶码如果用 10 位表示，则尾数只能用 21 位了。阶码的 10 位可以表示 2^{10}=1024 个状态，按正负各半，可以表示约 2^{512} 次方（10^{154}）大的数，但是尾数 21 位只能表示 6 位精度了。一个有 100 多位的大数，却只有 6 位有效数，只适合表示向领导汇报的数字，不适合进行科学计算了，因为精度丢失未免太大了点。

一个符合精度要求的数（要求较长的尾数），却不能在某个区间表示（阶码长度不够），那就不能做正常计算；而一个绝对值很大的数（要求较长的阶码），却只有很小的精度（尾数长度不够），这个数就不大有用。在科学计算中，既要很强调精度，又要兼顾各种区间的数值表示，所以有效位数和阶码的长度在设计中是一对矛盾。经过专家论证，所定出的 IEEE 国际标准 754 的浮点数规范，是取得了有效位数与尾数长度之间的一个平衡，以利于用 32 位数最充分地表达一个小数。

8 位阶码意味着 2^8=256 个阶码状态，可以表示 2^{128}（10^{38}）的数值范围，23 位尾数相当于 2^{23}（≈10^7）可以表示约 7 位精度。

◆ **阶码设置**

二进制小数规格化之后，尾数的精度从 23 位提升到 24 位，但是牺牲了浮点数 0 的规格化表示。因为规格化要求小数点前面 1 位必须是 1，所以，即使整个尾数部分都为 0，其浮点数的有效值也为 1，结果规格化的浮点数都非 0。为了能表示二进制浮点数中的 0，除了规格化表示之外，还保留了非规格化表示形式。

标准 32 位浮点数（单精度）规定，浮点数的阶码为 8 位（表示为有符号整数，范围在-128~127），规格化数的阶码范围在-126～127。所以规格化数的阶码最小为-126，而不是-128。

当阶码值为-127 时，表示阶码为-126 的非规格化（denormalized）数。所以，非规格化数只能表示非常靠近 0（含 0）的小数，不能表示幂次方为正的大数。

当阶码值为-128 时，用来表示特殊浮点数。

32 位浮点数在计算机内以规格化数表示，还是以非规格化数表示，还是以特殊浮点数表示，是由机器自动决定的。作为一般的计算结果，都表示以规格化数；当在计算时，遇到一个非常接近 0 的数，以至于用规格化数难以表示时，机器便会自动以非规格化数表示；当遇到一个非法计算，或者计算溢出时，便会以特殊浮点数表示。

◆ **0 与非规格化**

非规格化数就是不做规格化的二进制小数。它用阶码值-127 来表示固定的 2^{-126}，相当于一种定点数。其尾数也不隐含整数 1，所以有效值不省略小数点前面的 1，只用 23 位尾数，于是就可以表示全 0 的尾数了。0 乘以 2^{-126} 次方为 0。所以，当非规格化数（阶码为-127）的尾数全 0 时，该数就是浮点数 0（即 0,10000001,00000000000000000000000）。

如果浮点数 0 也是由全 0 位码实现，那么浮点数也可以当做逻辑数操作了：0 代表 **false**，非 0 代表 **true**，这将给语言带来数系之间更广泛的包容。

为了使浮点数 0 与整数 0（32 位全 0 位码）统一，就将浮点数的 0 也用全 0 位码表示。这就需要对所有浮点数的原阶码值作偏移——存储浮点数时，对所有阶码作+127 的偏移操作。这样一来，浮点数 0 的阶码-127 加上 127 就偏移为 0，加上符号位与尾数都为 0，便得到全 0 的位码了。

任何浮点数从存储中取出时，其阶码再做一个-127 的逆操作，就将该数又还原为能读懂的浮点数了。这在理解上并不难，就好比对于一个慢了 10 小时的钟，看见的是 1 点钟，对所有钟点都加上 10 个小时去看，认定为是 11 点钟一样的。

3．规格化数

◆ **表示形式**

存储为规格化浮点数时，需要额外做两件事：一件是获得 24 位尾数，并剔除第一位存入；另一件是对阶码做+127 操作，再存入。

例如，十进制数 35.6 转换为二进制数，存入到计算机中时，如下过程描述：

```
35.6 = 100011.1001(2)                                  //转换成二进制数
     = 1.00011100110011001100110×2^5                   //二进制数规格化
     = 0,00000101,1.00011100110011001100110            //准备机内表示
     = 0,10000100,00011100110011001100110              //阶码+127，尾数剔除 1
     = 01000010000011100110011001100110                //最终机内表示
```

其中机内表示中的 2 个逗号将浮点数分隔成三段，第一段的 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。

如果需要输出而将其取出，则做逆操作：阶码减去 127 得到 5，尾数加上 1 得到 1.00011100110011001100110，又还原为本来的数值了。如果仅仅是变量赋值参与计算，则原浮点数格式不变。

◆ **最大数**

非规格化数都是指数幂次在-127 的，即非常靠近 0 的数。比最微细规格化数还要小的数才用非规格化数表示。32 位浮点数能表示的远离 0 的最大数，则一定得用规格化数来描述。

当阶码最大（127），尾数也最大（24 位全 1）时，浮点数的绝对值达到最大。即：

```
0,11111110,11111111111111111111111          //机内码
= 0,01111111,1.11111111111111111111111      //取出时,阶码-127，尾数+1
= 1.11111111111111111111111 ×2^127          //折合二进制数
= (2-2^-24)×2^127
≈ 2^128
≈ ±3.4×10^38
```

注意其机内码阶码是加上了 127 的值。表 3-04 浮点类型说明中的 float 型数的表示范围即为 32 位浮点数的最大值。

如果浮点数为负，则为最大浮点数负值-3.4×10^{38}。

◆ **规格化近 0 数**

对于规格化数，其最小、最接近 0 的数值是多大呢？对于一个 24 位精度的不小于 1 的二进制数，应取其阶码最小值（-126），尾数最小值（23 位尾数全 0），这时候其内部表示为：

```
0,00000001,00000000000000000000000          //机内码
= 0,10000010,1.00000000000000000000000      //取出时,阶码-127，尾数+1
= 1.0×2^-126                                //折合二进制数
≈±1.2×10^-38
```

注意机内阶码-126+127 故为 1。见表 3-04 的规格化近 0 数。

4．非规格化数

◆ **近 0 数**

因为尾数可以取到比 1 更小，阶码与规格化的最小阶码一致，所以非规格化数能够比规格化数表示更接近 0 的数。当尾数取全 0 时，则该浮点数为 0；当尾数的 23 位取 22 个 0，最后 1 位为 1 时，其尾数的值相当于 2^{-23}，再迭加上阶码的值，则其数值为：

```
0,00000000,00000000000000000000001          //机内码
= 0,10000001,00000000000000000000001        //取出时阶码-127
= 2^-23×2^-126                              //折合二进制数
= 2^-149
≈ ±1.4×10^-45
```

因为尾数中只有一个非 0 位，有效位只有 1 位，所以，非规格化数的精度比规格化数要差。越靠近 0 的数，精度越差。

◆ **近规格化数**

非规格化小数最大也只能表示尾数全 1，阶码为-126 的数，此时，非规格化数最大，最接近规格化数。即：

```
0,00000000,11111111111111111111111          //机内码
= 0,10000001,11111111111111111111111        //取出时阶码-127
= 0.11111111111111111111111×2^-126          //折合二进制数
= (1-2^-24)×2^-126  //<2^-126
```

$\approx 2^{-126}$

$\approx 1.2\times10^{-38}$

相当于对机内码做二进制数运算，令规格化近 0 数减 1 得到最大的非规格化数。所以非规格化数的最大数正好与规格化近 0 数衔接上。非规格化数的阶码用-127 表示 2 的-126 次幂，在技术上恰好与规格化数衔接。

5．特殊浮点数

◆ **概说**

当阶码为-128 时，表示的数是特殊数。

特殊数有两种，一种是±∞，用 Inf 表示；一种是无意义数（Not a Number），用 NaN 表示。

特殊数不是通过赋值得到的，而是计算机在运行过程中因操作异常所致。

当浮点运算上下溢出时，得到±∞的特殊数。∞表示为尾数全 0，阶码为-128。当然经过+127 操作之后，实际的机内码为全 1。最简单的±∞可以从±1.0 除以 0.0 得到。+∞符号位为 0，-∞符号位为 1，以示区别。

无意义数在数学上是没有讨论价值的数，例如，0.0 除以 0.0，或者非实数等。当阶码为-128，尾数第一位为 1 时，便为无意义数，或者称非数。尾数只要第一位为 1，其他任何尾数组合都是无意义数。

当浮点数运算得到一个特殊数时，如果再做计算，便将会得到运行错误。

这与整数计算略有不同。整数计算中，如果某数除以 0，将立即得到一个运行错误，使程序中断运行。而浮点数计算中得到一个特殊数时，只要不再以该结果继续计算下去，程序仍将继续运行。

◆ **测试代码**

下面是产生浮点特殊数的测试代码。

```
//=====================================
//X0308.cpp
//test special float format
//=====================================
#include<iostream>
#include<iomanip>
using namespace std;
//-------------------------------------
void print(float f){
  int* p = (int*)&f;
  for(int i=31,a=*p; i>=0; i--)
    cout<<(a>>i & 1);
  cout<<"  "<<f<<"\n";
}//-------------------------------------
int main(){
  cout<<scientific<<setprecision(18)<<uppercase;
```

```
    print(0.0f);                //正 0
    print(-0.0f);               //负 0
    cout<<(-0.0f==0.0f ? "-0.0==0.0\n":"-0.0!=0.0\n");  //正负 0 比较

    print(1.0f/0.0f);           //正无穷 Inf
    print(-1.0f/0.0f);          //负无穷-Inf
    print(0.0f/0.0f);           //0 除 0 得到无意义数 NaN

    float f = 1.23567e+14f;
    int* pf=(int*)&f;
    print(f);
    (*pf)++;                    //格式化微小数的尾数增量
    print(f);
    (*pf)++;
    print(f);
    f *= 5.67E+57f;             //上溢导致无穷大 Inf
    print(f);
    f = 1.2345e-33f;
    f *= 5.621e-8f;             //微细化导致产生非规格化数直至 0
    print(f);
}//=====================================
```

得到结果为：

```
10000000000000000000000000000000  -0.00000000000000000E+00
00000000000000000000000000000000  0.00000000000000000E+00
-0.0==0.0
01111111100000000000000000000000  Inf
11111111100000000000000000000000  -Inf
01111111110000000000000000000000  NaN
01010110111000001100010001011110  1.23566992507954 7904E+14
01010110111000001100010001011111  1.23567006251850 1376E+14
01010110111000001100010001100000  1.23567019995745 4848E+14
01111111100000000000000000000000  Inf
00000000000000001100000101101111  6.93908990090595 5328E-41
```

◆ **代码解释**

代码是对 32 位 **float** 类型的浮点数进行的操作。输出采用 print 函数调用的方式。函数在执行输出的时候，是将 32 位浮点内码与浮点值一并输出，这样便于观察和比较。

通过整型数的位操作，可以访问内存字节所表示的二进制数中的每一位。

要将浮点数的内码逐位输出，需先将浮点数的内存空间看做 32 位的二进制整数，然后对其实施位操作，来获得其每一位的值。因为 C/C++浮点数作为数据类型的实体本身是不能做位操作的，只有整型数可以做位操作。

函数 print 中，为了逐位输出浮点数的二进制内码，先采用整型指针指向浮点数内存空

间，然后通过整型指针的间接访问（☞CH6.1.4 间接访问）获得该空间的 32 位整型数。再对整型数做位操作，从而逐位输出之。

整型指针定义为 **int*** p，初始化为浮点变量的地址&f。因为指针也有类型的分别，所以在初始化时，要先将浮点变量的地址转换成整型实体地址，即**(int*)**&f。

获得指针指向的整数值，使用指针的间接访问操作*p，因而使用 **int** a=*p。

位操作>>是对整型数进行右移操作。a>>1 表示整型变量 a 右移 1 位，若 a 原先为 5，则右移 1 位后变成了 2（☞CH3.1.4 位操作）。

位操作&表示逐位与操作。a>>i & 1 表示整型变量 a 在经过 i 位的右移后（将右边第 i+1 位移到了右边第 1 位），再与 1 做位与操作，从而得到结果非 0 即 1，该结果实际上便是右边第 i+1 位的 01 值。输出这个值即是输出右边第 i+1 位的值。如果 i 循环从 31 到 0，则完成从最高位（右边第 32 位）到右边第 1 位逐位输出。

◆ **结果解读**

程序首先将+0 与-0 输出，以展示其差别。随即又输出其比较的结果是相等。浮点数不是二进制补码，所以存在正负 0 的差别，但是正负 0 在比较判断中是忽略其符号位的。

程序然后输出正负无穷大，可以看出计算机表达无穷大用的特殊符号。要注意的是，不同的编译器，表达特殊数的方式是不同的。

当执行 0.0/0.0 之后，便获得了一个 NaN 特殊数。

接着对浮点数的规格化最小微粒的增量进行了展示。先赋一个规格化小数，然后对尾数加 1（这个加 1 是通过整型指针的间接访问完成的），然后输出之。比较与前一个浮点数的差别，可以看到最小微粒的增量，其实也是一个相当的数量值。

后面便是展示浮点数的上溢。上溢得到一个正无穷大。同理，上溢也可能从负方向得到负无穷大。

最后，当浮点数计算得到充分接近 0 的数时，便会自然地以非规格化数表示。若再小下去，最后便会得到 0。

3.3.4　三种浮点型

1．浮点类型

计算机中的浮点数是科学记数法的标准二进制表示形式，它通过阶码的变动，具有了浮动的数据区间和一定精度之实数的表达能力，是一种相对定点数和整数能够更好地完成科学计算任务的数据表达方式。

对一些科学计算来说，32 位浮点数虽然比 32 位整型数表示的范围要宽，但是总的精度却比不上 32 位整型数了，加之 32 位浮点数在科学计算中，表示范围还是显得有限，便需要更高精度和范围的浮点数处理。C/C++迎合了这种要求，提供了三种标准浮点型数的支持。

float 表示 32 位浮点型；**double** 表示 64 位浮点型；**long double**[①]表示 80 位浮点型。

① long double 数只有在更高 C++版本中才有完整实现。在 VC 6，BCB6，G++4 中，虽然都实现了 80 位长，但实际最大阶值仍为 double 型阶值，只是提高了一些精度而已。

double 关键字虽然没有一点浮点的意思，但是它的双倍性词意，正是要表达在单精度的 **float** 之上，具有双倍精度的浮点型之意；同样，**long double** 表示加长的或者扩展的双精度浮点型，而不是在双精度的基础上的再加倍（128 位）之意。

在双精度浮点型（**double**）中，有 1 位符号，11 位阶码，52 位尾数，共 64 位，占 8 个字节；在长双精度浮点型（**long double**）中，有 1 位符号，15 位阶码，64 位尾数，共 80 位，占 10 个字节，如图 3-01 所示。

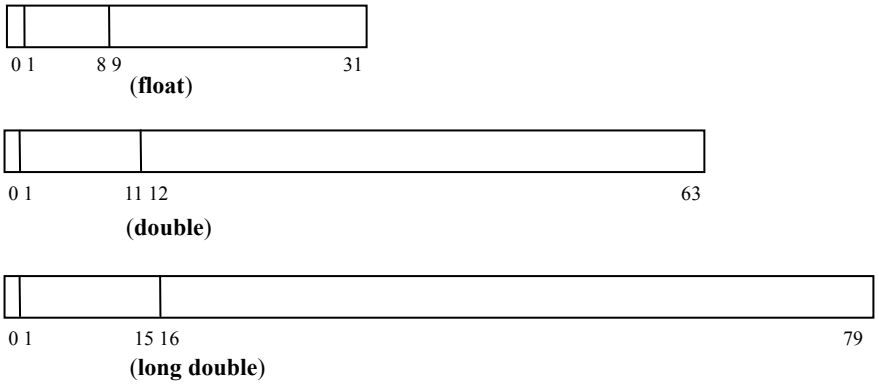

图 3-01　各种浮点数的内部表示形式

2．双精度数表示

双精度浮点数的表示原理与单精度浮点数（32 位）相同，只是阶码位数不同，尾数位数不同，所表示的精度要比 **float** 大不仅仅是一倍。

双精度浮点数的规格化数的阶码表示范围为-1022～1023，而-1023 表示非规格化数，-1024 表示特殊浮点数。类似地，浮点数的阶码在存入操作时要做加 1023 操作，取出时做减 1023 操作。尾数的位数虽然只有 52 位，但是与 32 位浮点数同样的原因，规格化数的精度却有 53 位，有 1 位整数位在规格化时省出来了。

例如，对于双精度浮点型变量定义：“**double d** = 35.8125;”可得 d 的内部表示为：

$1.0001111010000000000000000000000000000000000000000000\times 2^{5}$

= 0,10000000100,0001111010000000000000000000000000000000000000000000

其中阶码 10000000100 为 5+1023 所得。

按照这个原理，双精度浮点型的最大数，是当阶码为 1023，尾数为全 1 时所得。即机内码：

0,11111111110,1111111111111111111111111111111111111111111111111111

规格化近 0 数，是当阶码为-1022，尾数为全 0 时所得。即机内码：

0,00000000001,0000000000000000000000000000000000000000000000000000

而非规格化最大数，是当阶码为-1023，尾数为全 1 时所得。这个数与规格化近 0 数几乎相等。若非规格化数加上 $2^{-52-1022}$（2^{-1074}），则完全相等。其机内码为：

0,00000000000,11111111111111111111111111111111111111111111111111111111

非规格化近 0 数，是当阶码为-1023，尾数为 51 个全 0 加上最后一个 1。即机内码：

0,00000000000,0000000000000000000000000000000000000000000000000000000000000001

如表 3-04 所示。

3．长双精度数表示

长双精度浮点型是比双精度浮点型更高一级的浮点型，精度已经高到不稀罕那尾数节省下来的那一位。更何况规格化中的那一位不是白白节省的，而是在存储和取用时，用减去和加入操作的代价去换的。对于短浮点数，很计较精度，很看重这省出的一位。但是对于长浮点数，精度已经很高了，比起精度更看重其操作性能。这时候，就不省这一位了。长双精度数的国际标准正是这样做的。

或许，现代人再来设计 **float** 的内部结构，不会省那一位，因为有后面两种浮点数类型可做选择，正好在小精度计算中也强调一下性能。

长双精度浮点数的正规数的阶码表示范围为-16383~16383，并且没有非规格化数表示的必要，-16384 表示特殊浮点数。当然，阶码的飘移操作还是有的，也就是说，浮点数在作存入操作时，阶码要加 16383，作取出操作时要减 16383。所以同样对于 35.8125 这个浮点数，长双精度浮点型数内部是这样表示的：

1.00011110100000000000000000000000000000000000000000000000000000×2^5
= 0,　　//符号
100000000000100,　　//阶码 5+16383
1000111101000000000000000000000000000000000000000000000000000000　//尾数

其中的阶码是在 5 的基础上做了+16383 的偏移所得。从中可以看到，尾数头上那一位没有省略。

长双精度浮点数所能表示的最大数，是在阶码为 16383，尾数为全 1 时达到的。即机内码：

0,　　//符号
111111111111110,　　//阶码 16383+16383
11111111111111111111111111111111111111111111111111111111111111111 //尾数

长双精度浮点数所能表示的近 0 数（不分规格化与非规格化数）是在阶码为-16383，尾数为全 0 加上最后一个 1 时达到的。即机内码：

0,　　//符号
000000000000000,　　//阶码-16383+16383
0000000000000000000000000000000000000000000000000000000000000001　//尾数

但是，这样的表示不是很有用，在不损失精度的前提下，也就是尾数仍然保持有 64 位有效，这时候的近 0 数应在阶码-16383，尾数全 1 时达到。也就是将其看做规格化数，为了保证精度而已，这在表 03-04 中都可以看到。

4．浮点数表示范围

所有浮点型的表示范围，我们用表 3-04 来说明。

表 3-04　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	$\pm3.4\times10^{38}$	$\pm1.8\times10^{308}$	$\pm1.2\times10^{4932}$
规格化近 0 数(精度保证)	$\pm1.2\times10^{-38}$	$\pm2.2\times10^{-308}$	$\pm6.7\times10^{-4932}$
非规格化近 0 数(损精度)	$\pm1.4\times10^{-45}$	$\pm4.9\times10^{-324}$	$\pm4.0\times10^{-4951}$
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	18 位
规格化数阶值范围	-126～127	-1022～1023	-16383～16383
非规格化数阶值	-127	-1023	
阶码偏移	127	1023	16383
NaN 阶值	-128	-1024	-16384

3.3.5　浮点数问题

1．精度丢失

◆ 范围溢出

浮点数的上溢是指绝对值够大，大到浮点型没有办法表示而趋无穷大。上溢可能是正无穷大∞，也可能是负无穷大-∞。

浮点数的下溢是指绝对值趋向于 0。

在运行中，产生的上溢作为特殊浮点数看待，产生该数则免不了会导致计算结果的异常。产生下溢也会导致计算无法进行下去。例如：

```
float f = 35.8125e300F;              //该数无穷大
double d = 35.8125e1000;             //该数无穷大
long double ld = 35.8125E5000L;      //该数无穷大
float f2 = 3.2e-28;
f2 *= 3.2e-28;                       //产生上溢，得到负无穷大
d = -3.9e-300;
d *= 123e-40;                        //产生-0 下溢
```

◆ 类型与精度

如果浮点数的类型表示错了，会导致数值的精度受损。例如：

```
long double ld1 = 19.2;
long double ld2 = 19.2L;
```

则 ld1 与 ld2 变量在其内部表示的尾数是不同的：

```
ld1: 10011001100110011001100110011001100110011001100110011001100000000000
ld2: 10011001100110011001100110011001100110011001100110011001100110011010
```

可以看到，ld1 的尾数因为是由双精度数赋值而来，在表达长度以外的地方以 0 填充，显然精度不及 ld2。

◆ **精度丢失程度**

由于参加运算的数据类型可能先要转换成另外的数据类型（由低精度向高精度转）进行中间运算，然后再转换回到低精度上去，因此有一个从高精度数向低精度数转换的问题，转换中可能会引起精度丢失。例如，下列程序说明高精度类型的数赋值给低精度数据类型的实体，造成了不同程度的精度丢失。

```
//=====================================
//X0309.cpp
//精度丢失程度
//=====================================
#include<iostream>
#include<iomanip>
using namespace std;
//-------------------------------------
int main(){
  cout<<setprecision(9)<<fixed;
  //在小数区域显示精度丢失的程度
  float f=7.0/3;         //用 double 精度值给 float 变量赋值,引起精度丢失
  int a=7.0/3;           //用 double 精度值给 int 变量赋值,引起精度丢失
  cout<<"double :"<<7.0/3<<"\n";    //按 double 精度正常输出
  cout<<"float  :"<<f<<"\n";
  cout<<"integer:"<<a<<"\n\n";
  //在整数区域显示精度丢失的程度
  f = 700000000.0/3;
  a = 700000000.0/3;
  cout<<"double :"<<700000000.0/3<<"\n";
  cout<<"float  :"<<f<<"\n";
  cout<<"integer:"<<a<<"\n";
}//=====================================
```

其运行结果为：

```
double :2.333333333
float  :2.333333254
integer:2

double :233333333.333333343
```

```
float  :233333328.000000000
integer:233333333
```

将 double 型数 2.33333…转换为 **float** 数的时候，因为 **float** 的有效位数有限，所以精度丢失，计算没有办法在 **double** 层次上进行，只能在 **float** 的单精度上进行。显示该 **float** 变量的值 2.333333254。这个数是 32 位浮点数在转换为十进制浮点数时保留 9 位小数的结果，显然没有 **double** 数精度高。

同样将 **double** 型数赋给 **int** 型变量，截去了所有小数部分，引起更大的精度丢失。然而引起更大精度丢失的原因不是因为整型数的精度比 **float** 浮点数小，而是表现精度值的区域不在整数部分。

接下去的运行结果说明了这一点。将一个在整数区域充分大的 **double** 型数分别赋给 **float** 浮点型变量和 **int** 整型变量，观察两者的精度丢失情况。可以看到 **float** 在整数部分中已经丢失精度，而 **int** 变量至少保留了整数部分的精度。

因为 **float** 的表示区域大于整型，而又与整型有同样长度的数据表示，所以其有效位少于整型数。相当于其在数轴上的数值分布更加稀疏，所以实际精度表示不及整型。但是，在 0 到 1 区间，整型数不具有表现力，这时候，**float** 型数便体现出“更高”的精度了。

2．浮点数比较

◆ **浮点数不精确**

浮点数可以进行比较操作，但是浮点数由于两个原因而使比较的不精确：

（1）是浮点数类型在精度表示能力上的问题引起。例如，两个 9 位小数的十进制数，前 8 位相同，但表示成 **float** 型浮点数之后，两个浮点数没有差异。

（2）是十进制数转换成二进制浮点数的不精确性引起。例如，十进制数 0.6 在浮点数表示中只能近似表示。

这些原因导致了浮点数在相等性比较中失败。例如用下列程序说明：

```
//=====================================
//X0310.cpp
//精度误差影响比较
//=====================================
#include<iostream>
using namespace std;
//-------------------------------------
int main(){
  float f1 = 7.123456789;
  float f2 = 7.123456785;
  cout<<(f1!=f2 ? "not same\n" : "same\n");

  float g = 0.6;
  double d = 0.6;
  cout<<(g==d ? "same\n" : "not same\n");
}//=====================================
```

其运行结果为：

```
same
not same
1
```

当你满怀信心地读完程序后，想象出的运行结果却与真实的运行结果大相径庭，你有何感想？

◆ **代码分析**

首先，由于十进制单精度浮点数的有效位数约为 7，两个前 7 位相等而后面不同的数有可能在计算机中表示为同一浮点数，因而判断 f1 和 f2 为不相等失败。

在编程计算中，f1 可能来自一个 **double** 型计算结果，而 f2 可能来自另一个不同的 **double** 型计算结果，当它们用 float 型表示时，由于精度受限而不能分辨出其差异，这就是错误结果的根源。如果要分辨值的差异，则应该用表示能力更强的 **double** 或 **long double** 来表示这两个数。

其次，由于 **float** 和 **double** 的精度不同，比较操作总是先要将两个相容的类型化成相同的类型再进行比较，所以相同的浮点数初始化给不同浮点类型的变量，可能表示不同的浮点数，判断 g 与 d 相等居然也会遭到失败。

◆ **解决方法一**

为了避免这类问题，在单一计算中请统一使用一种浮点型（建议使用 **double**），而不要混用不同精度的浮点数。对 C++来说，**float** 已是昔日黄花，除了为一些 C 程序作过渡，在新编的程序中很少有大的用处。

在 32 位计算机中，浮点计算装置被默认设置成 **double**，若用 **float** 型计算，反而需要来去转换的时间开销。况且 **double** 的精度完全覆盖了 **float**，混进 **float** 型只会添乱。按程序运行实测，许多 **double** 的算术计算并不比 **float** 慢。

另外，不要用不同的数据类型与浮点型数进行比较。例如，用整型数与浮点型数进行相等性比较，那是不合适的。

◆ **解决方法二**

虽然用同一个浮点型，两个数值相等性的比较也要慎重。因为两个“相等”的浮点数可能来自不同的计算。

由于浮点数在计算机内实际上是一个近似表示（☞CH3.3.1），在手工计算看来为正确的结果，在计算机中运算未必能得出正确的结果。例如：

```
//=====================================
//X0311.cpp
//浮点数的比较
//=====================================
#include<iostream>
#include<iomanip>
#include<cmath>
using namespace std;
//-------------------------------------
int main(){
```

```
    double d1=123456789.9*9;                          //手工计算结果 1111111109.1
    double d2=1111111109.1;                           //手工计算结果 1111111109.1
    cout<<setprecision(9)<<fixed<<d1<<"\n"<<d2<<"\n";    //实际显示不同

    cout<<(d1==d2 ? "same\n" : "not same\n");            //直接比较法失败
    cout<<(abs(d1-d2)<1e-05 ? "same\n" : "not same\n"); //邻域比较法正确
}//====================================
```

其运行结果为：

```
1111111109.100000143
1111111109.099999905
not same
same
```

十进制数在转换为内部浮点数时，由浮点数的构成原理决定了由无穷尾数而带来的不精确性。上述代码中的 d1 和 d2 变量的值都来自精确十进制数或者通过其计算本应相等，却在计算机内部表示中不等。通过较长精度的显示格式，将该问题的实质暴露了出来。

显然，浮点数的比较是比较浮点数的内部表示，而不是一定格式的外在输出形式，所以，直接比较两个来自不同计算的浮点型数，并不能权威地得到正确结果。

使用浮点数进行相等（==）和不相等（!=）比较的操作通常是有问题的。浮点数的相等比较一般总是使用两者相减的值是否落在 0 的邻域中来判断的。

需要注意的是邻域大小的控制，一般在 10^{-2}～10^{-5}。这对计算机来说，约为二进制数的 7 位到 13 位。在这个范围若出现差别，已经足够说明两个浮点数的不等性了。请体会上面代码中所示的邻域比较技术。

3.4 C 字 串

一连串的字符构成字串。若将字串看成是集合，则字符为其元素。只不过字串元素是按顺序挨个排列的。

我们所使用的字串从直观上认识，就是双引号括起来的字符序列。为了对字串深化运用，就需要对其进行读写操作，因而自然地就要将字串像变量那样存储起来。

在 C++中，有两种字串，一种是从 C 沿袭过来的 C 字串，另一种是 C++的 STL 资源提供的 string 类型的字串。它们都很常用。

3.4.1 C 串表示

1. C 串结构

◆ C 串称呼

C 字串有时候直接称 C 串。它在存储结构中表现为字符序列，所以经常用字符数组表

示 C 串。

比较特别的是，C 串以 0（一个全 0 位字节）作为字符序列的结束符。该全 0 字节既是 8 位[①]的整数 0，也是 ASCII 码的 0。所以即使算上结束符，C 串还是全部由 ASCII 码构成，因而又称 ASCIIZ 串（即 ASCII 字符序列加上尾巴 Zero）。

◆ C 串存储

C 串在编程中书写下来的形式称为字串字面值，其格式为双引号括起来的字符序列。

例如，我们将"Hello!"这一字串字面值与字串的存储结构作一个对应，则得到如图 3-02 所示的存储表示。

(a)　(b)

图 3-02　"Hello!"的存储结构

图中（a）为字串字面值的各字符表示，（b）为字串真正存储的各字符所对应的 ASCII 码值（☞CH3.2.1 字面值）。很显然，C 串的空间存储长度为字串长度加 1。

在谈论字串存储的时候，我们才区分是否是 C 串。在讨论字串字面值的时候，虽然其存储都是 C 串的形式，但是，更多的是描述其文字值及其使用。

2．C 串数组表示

◆ 数组表示

数组是表示一定类型的实体连续空间。数组需要定义数组名和数组元素个数。必要时，还需要初始化。数组将在第 4 章展开描述。

如果要将字串放入字符数组，则元素个数非比字符数多 1 不可。例如：

```
char buffer[7] = { 'H', 'e', 'l', 'l', 'o', '!',0 };  //尾部的 0 是必需的
```

考虑到字串字面值的表达，C/C++为字符数组定义提供了字串初始化的特殊形式，也即下面两种字串的字符数组定义与上面等价：

```
char chs1[]={"Good"};        //chs1 得到 5 个字符元素
char chs2[]="hello!";        //与 buffer 长度相同，都是 7
```

这两种字符数组初始化字串的形式，可以有花括号，也可以不用花括号，直接将字串写出来。保留字串的自然形式不拆分给编程带来了方便。字符数组 chs2 虽然在初始化中给出的是 6 个字符长的字串，但是，实际初始化的却有 7 个字符。

要注意的是，字符数组存储的是字符元素，当数组的字符元素序列之尾部跟上 0 时，也可以当做字串用。所以：

```
char chs3[6]="Hello!";           //错误:  虽该字串长为 6,但却需要 7 个元素存储
char chs4[6]={ 'H', 'e', 'l', 'l', 'o', '!'}; //正确: 但不构成字串
char chs5[10] = { 'G', 'o', 'o', 'd',0, 'A' }; //正确: 但字串长为 4
```

① 字符的 8 位不是标准 C++的说法，因为字符类型的字节长度并不是由语言决定的。

◆ 输入字串

用字符数组存储 C 串，便可以输出。字符数组也可以承载新的 C 串，通过给输入设备一个字符数组名就可以了。例如：

```
char a[10];
cin>>a;
```

需要注意的是，上例输入的字串最长只能为 9，因为考虑到结束符占 1 个字符。如果需要输入更长的字串，则需要增大字符数组的元素个数。程序在运行中，如果输入长度超过了定义的数组大小，则将产生运行错误。

输入字串的前提是输入的字串有地方存放，并且存放得下。保存字串，一般是将字串输入到字符数组，它比较经济、有效。

◆ 输出字串

字串的输出是用字符数组名来操作的。例如：

```
cout<<chs2<<"\n";
```

就可以获得"Hello!"字串的输出。

这与字符输出是不同的。字符是字符数组的元素，要输出数组中的字符，可以访问数组元素，也就是下标方式的访问。例如，输出其中的'H'字符和'!'字符，可以：

```
cout<<chs2[0]<<chs2[5]<<"\n";
```

所以，输出字符用的是数组元素下标访问，而输出字串用的却是数组名。

用字符数组名代表字串，用于字串输出是 C/C++的特殊操作。它的前提是，字串在字符数组的存储表达中有自己的 0 结束标记。输出字符数组名即是输出字串，中间若遇 0 便结束，并不是输出全部数组元素（不一定构成字串）。例如：

```
cout<<chs4<<"\n";  //错误：运行时因为结束符不明而在字串尾可能带有乱符
cout<<chs5<<"\n";  //正确：只输出 Good
```

字符数组与 C 串结下了不解之缘。除了字符数组，其他数组名都不能被输出设备所理解。例如，对于整数数组 a，下列代码会惹编译器报错：

```
cout<<a<<"\n";      //错误：不认识的类型
```

◆ 数组名不宜作变量

用字符数组名输出字串，给了我们两种合理的猜测：

一是字符数组名表示数组空间的首地址。用表示字串首地址的数组名来输出字串，可以利用字串所含有的自结束标记，被输出设备成功识别其边界而输出；

二是数组名当做字串变量。输出变量值，就把存储着的字串输出了。

如果 C/C++按第二个猜想实现，那么对于字符数组 chs5 的定义（10 个元素），应该获得另一种结果：

```
Good?A????
```

上面结果中的问号表示 ASCII 码的 0。作为 0 的字符值虽然内码为整数 0，但由其字

符的不可见属性而表现为没有显示。

更进一步，对于另一种数组，例如整数数组：

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

来说，是否可以通过输出语句：

```
cout<<a<<"\n";
```

而得到全部 5 个整数的依序连贯输出呢？答案是：不行。

原因是 C 数组是一个连续的内存空间块，访问元素的操作指令并不去检查数组的边界（如果老是检查边界的话，就不会被评为挖空心思以性能取胜的语言，也许 C++就不会去继承 C 了）。那么在输出过程中怎么知道数组的尽头呢？何况输出全体元素的格式要求千变万化，这种输出设计于编程、于计算又有什么意义呢？

因此，数组名作为变量也只有字符数组才有意义，而且通过输出数组名来达到输出字串的目的也要依靠字串本身的结束标记。

既然输出数组名操作只是专门针对字串输出，而且依赖于字串结束符，那么显然针对第一种猜测——数组名当做字串的地址去实现，更为合理，事实如此。

3．字符指针

◆ **定义形式**

用字符数组表示字串，能够成功地将字串以 C 串形式存储起来，以便作字串的进一步操作。当需要输出的时候，只要给出字串的起始位置（字符数组名），因其含有结束标记，就能被输出设备识别而输出。

但是，如果字串的挪来挪去，仅仅用作输出，就不需要将字串存入数组，用一个字符指针，也能表示（指向）字串的起始位置，因而直接把字串赋值给字符指针就行了：

```
char* str = "Hello";//正确
```

在 C/C++中，如果在字符数组初始化定义中出现字串字面值，那么就将字面值填入字符数组中。而如果字串字面值出现在其他场合，那么字面值就作为只读内容存储在独立的空间中。

◆ **初始化**

上述字符指针 str 定义中出现的字串字面值"Hello"，就是被存储在另辟蹊径的空间中。其初始化的意义则是将指针 str 指向这块字串空间的起始地址，如图 3-03 所示。'

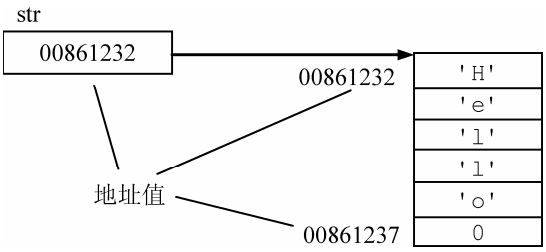

图 3-03　指向字串的指针

指针操作的详细解说将在第 6 章展开。这里需要知道的是说明指针类型的方式是在类型名后加上星（*）号。在指针定义中，初始化的值就是类型实体的地址（也可以看做指针值）。在这里，指针 str 是字符型的，所以所承载的值也应是字符的地址。初始化

的值（字串"Hello"）就代表这片存储空间的起始地址。

图 3-03 表示了一个字符指针 str 正指向一块字串空间的起始位置（00861232），也就是说，指针 str 值为 00861232，这个数字串是二进制数表示的地址值。虽然从计算机的角度来说，程序和数据没有差别，机器中有的只是些抽象的 0 和 1 的状态而已，但是从程序员的角度看，地址是地址，不能当做整型数来操作。

◆ **字面值属性**

按照我们对字面值的理解，字串字面值是在编程中描述的确定量，不管它占不占空间，都不应该对其实施修改。而指针的类型若是字符常量，则可以抑制指针对其进行修改访问。所以，将独立存储的字串字面值之起始地址作初始化赋给一个指针，更确切的表示应为：

```
const char* str = "Hello!";
```

事实上，所有的字面值类型都是 **const** 的。

字符指针作为一种类型，虽然与字符数组不同，但在输入/输出字串的表示上是一样的，都表示字串的起始地址。由于 C 串自身有结束符标志，所以，给定 C 串的首地址便可以识别和操作 C 串。也就是说，可以用字符指针来输出字串：

```
cout<<str<<"\n";
```

我们在这里首次接触字符指针，了解了指针是一个地址的表示，指针的类型（在这里是字符类型）也是指向实体的类型；而且字串的输出，用字符数组的数组名也好，用字符指针也好，都是因为它们表示的是字串的起始地址。这些似乎也在告诉我们，数组名与指针肯定有联系（☞CH6.3.3 指针与数组的差别）。

3.4.2 C 串操作

1. 字符数组的局限

◆ **字符数组能做到的**

数组的本质是同类实体多个单元的存储空间块。所以，用数组名表示它自己这个空间块的起始地址是合理的。数组的下标相当于从数组起始空间开始的元素位置偏移。

字符数组可以直接初始化为字串，以字符数组名的方式可以输入/输出字串，字符数组也就只能行使元素的下标访问之权利了，下标访问也只能对元素（字符）读出和修改（替换）而已。例如：

```
char a[10]="Hello";     //字串初始化
cin>>a;                 //输入一个字串
cout<<a;                //输出整个字串
cout<<a[0];             //输出 a 字串的第一个元素
```

◆ **字符数组不能做到的**

然而，代表字符地址的数组名与真实的字串还是有差别的。例如，比较两个字串的大小时，地址比较便不能代替字串比较：a 与"Hello"字串不能直接比较，因为数组名与字串

字面值都是字串地址。

由于数组名牢牢地捆绑着数组所拥有的空间，所以，任何把数组名看做变量，想要修改它的企图都是意欲对数组空间的重组或者对数组空间的破坏：

```
int a[10], b[10];
a = b;    //错误: 不能让数组 a 放弃自身空间而去关联数组 b 空间
```

这就牵连到存储有字串的字符数组不能直接进行赋值操作。也就是说，字串不能通过数组名赋值的方式从一个地方拷贝到另一个地方：

```
char ch1[10]="hello";   //正确: 字符数组初始化的特许形式
char ch2[10] = ch1;     //错误: ch1 是一个字符指针，不能初始化
```

何况字串的替换操作（ch1="Good";）涉及数组元素存储的重组，根本就是非法的。

字串的查找操作涉及遍历所有的数组元素，数组根本无法应承这些字串的操作要求。

甚至连获取字串长度的操作（“**sizeof**(ch1);”只表示数组空间大小，不能表示字串长度），对于只能进行下标访问的数组来说，也显得无能为力。

◆ C 串函数方案

C/C++意识到从语言的角度，也就是从字符数组本身不能解决这个 C 串操作的问题。但是数组的内在结构却是 C/C++得以发挥性能优势的良好基础，于是专门设计了一个字串操作函数集合来进行 C 串的操作。关于 C 串操作的函数，全部集中在头文件 string.h 中。

这里只选择一些重要的字串操作，分为修改性操作和非修改性操作来描述。其他以 C 串为基础的操作，可以通过下标访问、拷贝等一些字串操作的组合来解决。

2．非修改性操作

◆ 取长度

很多时候需要将字串的长度作为计算和循环的依据，或者以此来安排输出的位置。而表示字串的数组名仅仅是字符的地址而已，没有字串长度信息，但是可以通过字串起始地址，找到结束位置，从而获得长度信息。以此原理来获得字串长度的库函数声明为：

```
int strlen(const char* s);
```

函数名 strlen 从 string length 中拼凑而来。参数 s 是一个字符指针，表示字串的起始位置，所给定的字串是以 C 串结构存储的。无论字串存储在哪里，是否可修改，都可以通过该库函数获得字串长度。例如：

```
char* s = "abcdefg";          //指针 s 指向一个字串字面值
char a[10] = "12345";         //字符数组存储一个字串
for(int i=strlen(s); i>=0; i--)
  s[i] += 1;
cout<<"There are "<<strlen(s)<<" characters in "<<s<<"\n";
for(int i=strlen(a); i>=0; i--)
  a[i] += 1;
cout<<"There are "<<strlen(a)<<" characters in "<<a<<"\n";
```

代码中第一个 **for** 循环将每个字符的 ASCII 码加 1，从而得到字串 bcdefgh，循环的次数就是以指针指向的字串的长度 7 为依据。

第二个 **for** 循环将每个数字字串的 ASCII 码加 1，从而得到字串 23456，循环的次数以存储的字串长度 5 为依据，而不是以数组空间长度 10 为依据。

后面的输出语句则是对字符数组施行 strlen 操作。

字串长度为字串中的字符个数，不包括 C 串的结束符。例如，strlen(s)将使 for 语句循环 7 次。

代码中所给定的字串有来自字符数组的，也有来自字符指针的。但给出的必须是 C 串结构的字串，否则所得到的字串长度信息将不正确。例如：

```
string t = "hello";
cout<<strlen(t)<<"\n";  //错误: strlen 的参数不是 C 串，而是 string 字串
```

◆ **比较 C 串**

字串依赖于自身的结束符，而使数组名能在输入/输出场合代表字串自己。但是数组名只是字串地址而无法以此进行字串的比较。例如：

```
char buf1[6]="Hello";
char buf2[6]="Hello";
if(buf1==buf2)           //错误:  字串地址比较而不是字串值比较
  cout<<"equal\n";       //等价于 printf("equal\n");
else
  cout<<"not equal\n";  //等价于 printf("not equal\n");
```

由于数组 buf1 和 buf2 代表不同的空间区域，所以，无论初始化的内容是什么字串，以空间地址比较做依据的 if 语句都将输出“not equal”。

C 串比较操作的 C/C++的库函数声明为：

```
int strcmp(const char* a, const char* b);
```

函数 strcmp 即为 string compare 之意。参数 a、b 为以字符地址所代表的字串。函数 strcmp 的功能是对 a、b 字串逐个字符进行 ASCII 值的比较，以确定是相等或者大于或者小于。例如：

```
char a[8]="hello";
char b[7]="good";
char c[6]="goods";
char d[5]="good";
cout<<strcmp(a, b)<<"\n";   //>0 的整数值
cout<<strcmp(b, c)<<"\n";   //<0 的整数值
cout<<strcmp(b, d)<<"\n";   //0
```

则 strcmp(a,b)的结果为大于 0，strcmp(b,c)的结果为小于 0，strcmp(b,d)的结果为 0。也就是说：

（1）当字串 1 大于字串 2 时，返回值大于 0

（2）当字串 1 小于字串 2 时，返回值小于 0

（3）当字串 1 等于字串 2 时，返回值等于 0

因为是以 ASCII 码为比较依据的，所以'h'大于'g'而使字串 a 大于字串 b，0 小于's'而使字串 b 小于字串 c，两个完全相等的字串 b 和 d，而使其比较结果为 0。

从上面的数组定义中看到，字串比较与数组的空间大小无关。实施字串比较时，只要是符合存储结构要求的 C 串，就会得到合理的比较结果。

当字串 1 与字串 2 不相等时，函数 strcmp 所返回的值，并没有交代是大于 0 或者小于 0 的什么值，这与库函数的内部实现相关。也就是说，返回值会因不同的系统或版本而不同。它只是保证字串 1 大于/小于字串 2 时，一定返回一个正/负整数。所以编程中，也不要拿该返回值去做精确比较。比较代码可以下列这样的方式来写：

```
if(strcmp(a,b)==0)
  cout<<a<<"="<<b<<"\n;    //等价于 printf("%s=%s\n, a, b);
else if(strcmp(a,b)>0)
  cout<<a<<">"<<b<<"\n";   //等价于 printf("%s>%s\n, a, b);
else
  cout<<a<<"<"<<b<<"\n";   //等价于 printf("%s<%s\n, a, b);
```

◆ **查找 C 串**

用查找字串中子串的找到与否来决定程序接下来的行为，需要具有查找字串能力。C/C++的对 C 串实施查找的库函数声明为：

```
char *strstr(char *s1, const char *s2);
```

函数名 strstr 即为在 string 中查找 string 之意。函数 strstr 的功能为在字串 s1（第一参数）中查找字串 s2（第二参数）。如果找到，则说明母串 s1 中含有子串 s2，返回该子串在 s1 中的字符地址（指针），否则返回空指针（0）。

例如，在字串"International"中查找字串"nation"的下标位置，则有代码：

```
char *str1 = "International", *str2 = "nation";
char* p=strstr(str1, str2);
cout<<"String is: "<<str1<<"\nThe substring is: "<<str2<<"\n";
if(p!=0) cout<<"The position of substring is at: "<<p-str1<<"\n";
else   cout<<"The substring was not found\n";
```

若执行这些语句，则其结果为：

```
String is: International
The substring is: nation
The position of substring is at: 5
```

返回值为字符元素的位置（指针），它还不是下标位置。下标位置是数组元素中的相对位置。为了得到下标位置，要将字符位置减去字串起始位置（p-str1），即两个地址相减得到位移值。

如果在字串 s1 中找不到字串 s2，则返回的是空指针。空指针一般用 0 表示。于是是

否找到字串，只要判断 p!=0 即可。

因为参数 s1 可能为字符数组中存储的字串，也可能为字符指针指向的不可修改字串，所以，返回的字符位置的类型（**char***）令该子串处于可读又可写的状态可能会造成错误。所以 C++专门用另一个相关函数来补充：

```
const char *strstr(const char *s1, const char *s2);
```

该函数是个同名函数，但是参数类型和返回类型都不同。如果调用函数的参数是不可修改的字符指针形式，例如：

```
char* p = strstr("international","nation");
```

则编译器会自动匹配这个函数而不是前面的函数，以此来调校返回值处于正确的读写状态。编译器的这种能力源于 C++的函数重载（☞CH7.5.2）。

◆ **查找字符**

与查找字串中子串需求的同样，希望能在字串中查找字符。C/C++的对 C 串实施字符查找的库函数声明为：

```
char* strchr(char* s, int c);
char* strrchr(char* s, int c);
const char* strchr(const char* s, int c);
const char* strrchr(const char* s, int c);
```

函数名 strchr 即为在 string 中查找 character 之意。函数 strchr 的功能为在字串 s 中查找字符 c，如果找到，则返回找到的位置，否则，返回空指针（0）。考虑到可移植性，字串结束符也是字符之一（也许并不用 0 表示）等等因素，决定了参数中的字符 c 用 **int** 型表示。由于 **char** 类型与 **int** 是兼容的，所以完全可以用字符型变量作为参数传递给函数。

函数名 strrchr 即为在 string 中逆向（reverse）查找 character 之意。所以该函数的功能除了查找方向为从字串尾开始而不是从头开始外，其余都与 strchr 一样。

例如，在字串"What you really think."中找字符't'。因为顺着找，得到下标位置 3，倒着找，得到下标位置 16，则可以写出代码：

```
char str[25]="What you really think.";
char c = 't';
cout<<"String is: "<<str<<"\n"; //printf("String is: %s\n",str);
char* p = strchr(str, c);
char* rp = strrchr(str, c);
if(p!=0){
  cout<<"find '"<<c<<"' in the forward direction at: "<<p-str<<"\n";
  cout<<"find '"<<c<<"' in the backward direction at: "<<rp-str<<"\n";
}else  cout<<"The character was not found\n";
```

若执行这些语句，则其结果为：

```
String is: What you really think.
find 't' in the forward direction at: 3
```

```
find 't' in the backward direction at: 16
```

由于第一参数可能处于不可写状态，所以返回的字符位置应具有读写约束，这就是我们又一次看到了函数重载 strchr 和 strrchr 表示法的原因。

3．修改性操作

◆ **拷贝 C 串**

C 串的复制强调数据保护——从源数据中复制出数据，不破坏源数据就达到目的了。因为拷贝是强调从源数据所复制的一个副本，所以复制出的数据作为拷贝过程的返回数据。字串拷贝操作的函数声明为：

```
char* strcpy(char* dest, const char* src);
```

函数名 strcpy 即为 string copy 之意。它涉及三个字符指针，一个是指针 src，它是源字串的地址，一个是指针 dest，它是目标字串的地址。函数的功能是将源字串取出放入目标字串地址所指示的空间中，并返回该目标空间首地址。例如：

```
char a[20] = "fresh air";
char* s = "hello";
strcpy(a, s);
cout<<a<<"\n"; //后两句可以直接写成: cout<<strcpy(a,s)<<"\n";
```

结果，字符数组 a 的字串内容变成了"hello"。因为 strcpy 所执行的是将 s 地址的字串拷贝到数组 a 中，a 中原来的内容就被覆盖了。

函数 strcpy 是一个表达式，作为复制字串结果，所以后两个语句可以合并为一条语句。

要注意的是，目标字串空间大小是由程序员人为把握的，一定要记得目标空间应大于源字串的空间，否则，因拷不下而造成数组空间溢出的运行错误就要崩掉了。为了在编程中有更稳妥的手段，库函数中还提供有 strncpy 的函数：

```
char* strncpy(char* dest, const char* src, int n);
```

该函数比 strcpy 多用了一个额外的参数 n，说明目标字串空间（dest 所代表的空间）最多能接受的字符数。例如：

```
char b[8]="hello";
char* s="abcdefghij";
strncpy(b, s, 7);
```

考虑到 C 串的结束符，在编程时，数组 b 只有 8 个字符空间，如果存储字串，长度最多为 7，所以，strncpy 的第 3 参数设置为 7。如果用 strcpy，遇到上面这么长的字串 s，数组 b 将全部充满字母字符而无法构成 C 串，其后果是修改了数组 b 之后的空间，造成不可预料的错误。

值得注意的是，源字串描述的是常量指针类型，它可以是数组名，也可以是字符指针。但是目标地址应该是字符数组名，如果是指针类型，则必须是带有存储空间的地址，要可以存放字串。

◆ 拼接 C 串

运行结果为文字说明的时候经常会有选择性。例如，一会儿输出"equal"，一会儿又要输出"not equal"，这中间可能就需要字串的拼接操作。

C/C++的 C 串拼接操作的库函数声明如下：

```
char* strcat(char* a, const char* b);
```

函数名 strcat 即为 string concatenation（拼接）之意。函数的功能为将字串 b 拼接到字串 a 的后面。也就是说，字串 b 的内容拷贝到去掉字串 a 的结束符之后。例如：

```
char a[]="I do";
char b[]="n't";
cout<<strcat(a,b)<<"\n";    //等价于 printf("%s\n",strcat(a,b));
```

将得到"I don't"的输出结果。

如果要将一个字符拼接到字串上，则应该将该字符放入字符数组中，表示成 C 串的形式。否则将是一个编译错误。例如，想要拼接's'到字串"good"上的操作：

```
char c[]="good";
char d[]={'s',0};           //等价于 char d[]="s";
strcat(c,d);                //正确:等价于 strcat(c,"s");
strcat(c, 's');             //错误:第二参数的类型不应是字符，应是字符指针
```

◆ 倒置

判断一些字串中的字符排列是否对称，将 C 串倒置并与原串比较会很省事。C/C++对 C 串的倒置操作使用下述函数声明：

```
char *strrev(char *s);
```

倒置操作会对原字串作修改，以获得倒置的结果值。因此，参数 s 必须具有数组式的存储空间，能被修改。例如，将字串"string"倒过来：

```
char ss[] = "string";
cout<<"Before strrev(): "<<ss<<"\n";
cout<<"After strrev(): "<<strrev(ss)<<"\n";
```

则执行上述语句，会得到下列结果：

```
Before strrev(): string
After strrev(): gnirts
```

◆ 批量字符置值

有时候需要获得重复字符的字串，或者设置成重复字符的字串，那么，批量设置字符的操作便有需要了。C/C++对批量字符设置操作的库函数声明为：

```
char* strset(char* s, int c);
void* memset(char* s, int c, int n);
```

函数名 strset 是将 string“set”的意思，意即字串 s 中所有字符都设置为给定的字符 c

参数值。函数名 memset 是 memory set 之意思，意即 s 指向的内存块中 n 个字节都设置为给定的 c 字节值。

字串 s 为 C 串结构的数组存储形式的字串空间，strset 函数将字串的全部元素统统设置成字符 c 的值。它以字串结束符作为设置操作的结束。而 memset 则不分字串结束符，将 s 起始的 n 个字符统统置成 c 值。两个函数都来自同一个头文件 string.h。例如：

```
char ss[10] = "12345";
char symbol = 'c';
cout<<"Before strset(): "<<ss<<"\n";
strset(ss, symbol);
cout<<"After  strset(): "<<ss<<"\n";
memset(ss, symbol, 9);
cout<<"After  memset(): "<<ss<<"\n";
```

将会得到结果：

```
Before strset(): 12345
After  strset(): ccccc
After  memset(): ccccccccc
```

批量字符设置 memset 可用于给字符数组赋值，以使数组全部元素取同样的值。但如果要用作字串，则要注意结束符不能被覆盖。利用字符地址即字串的原理，可以重新来审视代码 X0208.cpp，使其简化：

```
//=====================================
//X0312.cpp
//字符三角形(C 串版)
//=====================================
#include<stdio.h>
#include<string.h>
//-------------------------------------
int main(){
  char sp[30]="                            ";      //预制 29 个空格字串
  char ch[60]={0}, c;                              //单设字串结束符
  for(int n; scanf("%c%d",&c,&n)!=EOF; ){          //循环输入字符和整数
    memset(ch, c, 59);                             //ch 数组批量 c 值设置
    for(int i=1; i<=n; i++)
      printf("%s%s\n",&sp[29-n+i],&ch[60-2*i]); //n-i 个空和 2i-1 个字符
  }
}//=====================================
```

勾画字符三角形的循环中，需要第 i 行输出 n-i 个空格和 2i-1 个输入的字符。

因为最初的初始化已经准备了 29 个空格的字串 sp，在循环输入字符和三角形高度的循环中，一开始也设置了重复字符 c 变量 59 次的字串 ch（构成 59 个相同字符的字串）。所以，倒数 sp 字串 n-i 个位置开始的字串，便是 n-i 个空格的字串，即&sp[29-(n-i)]，其中&表示该 sp 数组元素的地址；然后，倒数 ch 字串 2i-1 个位置开始的字串，便是 2i-1 个重

复字符的字串，即&ch[59-(2*i-1)]，或者&ch[60-2*i]。

当然，因为 30≤n，所以，sp 数组的 29 个字符的字串长度总是小于 n-i 的最大值，ch 数组的 59 个字符字串长度也总是小于 2i-1 的最大值。

3.5 string 字串

3.5.1 string 串

1．string 类

◆ 头文件

string 是一种数据类型（type），它最初属于第三方提供的 STL（标准模板库）资源，后来并入 C++系统。使用 string 类型需要包含 string 头文件。需要注意的是，C++包含头文件都不用后缀.h。事实上，包含 string 与包含 string.h 是有区别的：使用 string 类型，包含 string；使用 C 串库函数，包含 string.h（或者用 C++头文件 cstring）。

在第 1 章就已经看到过它的使用了：

```
string s = "Hello";  //像变量那样初始化
```

◆ 身份评价

称 string 为字串类，其所产生的 string 实体就是 string 串了。string 串作为另一种字串表示，与 C 串的处理形成对照。string 是通过类机制（☞《C++程序设计教程详解——对象化编程》）产生的一种类型，其所定义的实体按理应称为对象，它拥有自己（串）的操作和内部表达。

由于单纯处理字串，作为一个简单概念称 string 实体为变量没有什么不妥。但若硬是在本书的过程化编程实践中强调对象性质，反而显得古板和生涩。

我们知道，作为内部类型的实体，会拥有一些现成的操作。例如，整型数有加、减、乘、除操作。string 类是一个成熟的自定义类型，它不是内部类型，但作为数据类型也拥有一些操作。依靠那些操作，可以方便地对字串进行许多 C 串所不能直接执行的操作。

string 串的存储空间是由 C++自动分配的，需要多少用多少。不像字符数组那样，需要事先估算好字符数，更不像字符指针那样，成天提心吊胆于可能指向错误的位置或者空位置。string 串可以从 C 串字面值赋值得到，显示了与 C 串强大的互操作性。string 类型本身就是 C++针对字串操作而设计的，所以，在代码编写上比 C 串的函数处理要直观和方便得多。

2．string 串定义

◆ 两种初始化方式

变量的定义有两种等价的初始化方式：

```
int a=3;    //C 方式
int a(3);   //C++方式
```

所以下列定义可以在两种方式之间互通：

```
string s1("hello");             //即 string s1="hello";
string s2(s1);                  //即 string s2=s1;
```

◆ 基于 C 串的 C++初始化

C++初始化方式的意义在于，当被初始化的实体依赖于不止一个数据时，便需要以括号的形式进行初始化。string 类定义规定了初始化的多参数形式，所以，除了以 C 串直接初始化 string 变量之外，其他的 string 变量初始化定义都是 C++方式的。下列包含了一些需要学习领会的 string 定义：

```
char* s = "HelloWorld";         //C 串
char c = 'A';                   //字符
int n = 6;
string a = "good";              //string 串初始化
string s3("HelloWorld", 6);     //取 C 串(第一参数)的前 6(第二参数)个字符
string s4(s, n);                //同上,即 string s4="HelloW";
string s5(s1,3,2);              //即 string s5="lo";
string s6(n, c);                //即 string s6="AAAAAA";
```

s3 的定义是取"HelloWorld"字串的前面 6 个字符为其字串。

s4 的定义形式与 s3 相同。但是其意义在于，括号中初始化的参数可以是变量，即 string 串的内容可以通过计算来产生。

s5 的定义中有三个参数，第一个参数表示主体字串（s1="hello"），第二个参数表示位置（位置是下标表示，所以是第 4 个位置在第二个'l'上），第三个参数表示长度。

s6 的定义中有两个参数。虽然 s4 的定义也是两个参数，但是参数的类型不同，意义就不同了：s4 的第一个参数是 C 串地址 s，第二个参数是整数 n，根据 string 类设计，规定为字串 s 的前 n 个字符；s6 的第一个参数是整数 n，第二个参数是字符 c，规定为 n 个重复字符 c（'A'），即"AAAAAA"。

字串字面值如果按 C 串存储起来，就是 C 串结构。如果赋给 string 作初始化用，所存储的就不是 C 串的空间结构了。string 串的内部表示比 C 串结构要复杂一些。我们学习 string 的时候，只强调其操作的实用性，先避免接触类机制方面的内容。

3．无名 string 串

◆ 概念

C 串既表示了字面值的外部形式，也表示了内部存储形式，经常就直呼为字符串，或者字串。

string 类型专门处理字串，所以其所定义的变量也称为字串变量。作为新生代的 string 串，也经常直呼为字串。

由于经常希望字串能做直观方便的处理，所以有必要将 C 串表示成 string 类型。在 C

串上直接加上 string 括号：string("Hello")就成为 string 串了。这样表示的 string 串实体，没有变量名与之对应，称为无名 string 串。

◆ 定义

string 的初始化形式，如果去掉变量名就成为无名 string 串了。例如：

```
string("good");
string("HelloWorld", 6);    //string("HelloW")串
string(s, n);               //string("HelloW")串
string(s1,3,2);             //string("lo")串
string(n, c);               //string("AAAAAA")串
```

无名 string 串有好几种产生方式。有多少种定义方式，就有多少种产生无名 string 串的方式。

打造无名 string 串，而不是字串变量，都是因为只需一次性使用。例如：

```
string x = string("Hello")+' '+string("World");      //拼接"Hello"和"World"
cout<<string(70,'W')+'\n';                           //输出一行 70 个'W'字符
```

◆ 应用

string(n,c) 的形式，能简化重复字符输出的循环。例如，X0208.cpp 的程序中，内循环体要做的是：

（1）输出 n-i 个空格；

（2）输出 2i-1 个 c 变量字符值；

（3）输出换行。

于是可以表示成：

```
cout<<string(n-i,' ');          //n-i 个空格的 string 串
cout<<string(2*i-1, c);     //2i-1 个 c 值字符的 string 串
cout<<"\n";
```

再根据流的接续特性，或者 string 的加法操作，可合并成一条语句：

```
cout<<string(n-i,' ')<<string(2*i-1,c)<<"\n";  //or
cout<<string(n-i,' ')+string(2*i-1,c)+"\n";
```

于是便可对 X0208.cpp 进行下列的简化：

```
//=====================================
//X0313.cpp
//字符三角形(string 版)
//=====================================
#include<iostream>
using namespace std;
//-------------------------------------
int main()
{
  char c;
```

```
  for(int n; cin>>c>>n; )
    for(int i=1; i<=n; ++i)
      cout<<string(n-i,'')+string(2*i-1,c)+"\n";
}//====================================
```

X0208.cpp 是用三重循环结构实现的，到了这里变成了二重循环，而且代码更加简单，容易阅读。所利用的就是无名 string 串的重复字符的表示形式。它甚至比 C 语言版的代码 X0312.cpp 的程序还要简单、可读。

4．操作方式

◆ **直接方式**

定义一个 string 串实体之后，该实体就和变量性质相似，赋值可以直接赋（=），比较（==）可以直接比，拼接（+）可以直接拼，甚至可以做累拼（+=）。做这些操作时，还有很大的包容度。也就是说，赋值操作可以赋字符、C 串和 string 串；拼接操作可以拼字符、C 串和 string 串。同时 string 串还具有数组下标操作的特点，可以访问串中的元素。例如，下列操作都是合法的：

```
string s = "string";    //初始化
string t = "using";     //初始化
s = t;                  //赋值: s = "using"
t += ' ';               //拼接字符: t = "using "
t += "namespace";       //拼接 C 串: t = "using namespace"
if(t==s)                //相等比较
  cout<<t+s;            //拼接 t 串与 s 串
else
  cout<<s[2]<<"\n"      //下标访问,"using"中的第三个字符,输出'i'
if(t<s)                 //小于比较
  cout<<"end\n";
```

所以，直接操作常用的有上面这些：=操作，==操作，+操作，+=操作，还有下标[]操作。

◆ **成员方式**

自定义的类型中，一些操作是用操作符表示的。例如，string 类中的+、=、<、[]等操作，它们看上去像直接操作方式。另一些操作是通过函数来描述的，对实体进行操作，就是实体去调用所定义的函数，构成成员操作方式。

在自定义类型中定义的函数操作一般都打上了类型的烙印，要想实施操作，就必须捆绑操作主体。例如，对于一个 string 串变量 s 获取其长度的操作形式：

```
string s = "Hello";
cout<<s.length()<<"\n";    //串 s 捆绑 length 函数,构成成员方式操作,输出为 5
```

因为所做的操作是有关 string 类的，所以操作主体 s 必须是 string 类的，通过点（.）操作与函数 length()捆绑。所捆绑的函数总是呈调用状。也就是说，函数名后必须附括号，这就是我们看到的 s.length()操作。

length()是 string 类的函数成员。string 类在定义时，定义了一些函数成员，也定义了一些实体成员。只要是成员，都能通过点操作符访问。

◆ **成员函数**

从函数的性质来说，函数有普通函数和专属于类型的成员函数，例如，string 类中的 length 函数。所以，专属于类型的函数也称成员函数。

有些成员函数没有参数，就像 length()函数那样，只要捆绑 string 串实体，就能返回该实体的串长信息。

一个类，只有定义了成员函数，才能使用成员形式的调用操作，而且，必须按声明形式匹配参数个数、类型和顺序。其规则与普通函数调用是一样的。所以，捆绑成员函数的操作不是随意的，需要查看调用声明。

string 类的成员函数声明在其头文件中都有，通过 C++集成环境中的帮助可以查看说明。

◆ **容器属性**

string 串作为字符集合而归为容器类。所谓容器，是指存放一定类型（字符型）元素集合的存储结构。string 通过下标操作访问字符元素。

作为容器都是有始有终的，string 也不例外，具有关于容器的标准起始和终结位置的成员函数描述。string 串 s 的起始与终结位置分别为：s.begin()和 s.end()。

begin()和 end()函数是一种统一的表示容器起始和终结的方式，用于 STL 的算法。例如，STL 中有一个逆反算法（函数）reverse，它将全部元素按顺序颠倒。作用于 string 串时，便使用字串的起始和终结两个位置参数。按下列方式调用：

```
reverse(s.begin(), s.end());
```

其调用的结果是字串 s 的内容颠倒了。

5．替换操作

成员捆绑操作主体的形式是自然而然的。哪个主体在做操作，做什么操作，目的很明确，也一目了然。习惯了这种方式，就觉得 STL 标准库开始有亲和力了。

◆ **操作要求**

字串有时候希望能做替换操作，这种操作可以删除字串中的一些字符或插入一些字符，可以对 string 串直接赋值，还可以对全体字符做批量字符置值操作。

替换（replace）操作，可以拿 string 串或 string 串中的子串，也可以拿 C 串或者 C 串的前几个字符，甚至还可以拿批量字符来对主体 string 串的子集作替换。例如对于如下操作：

```
string t = "XYZ";  //string 串 t
char* u = "WXVJ";  //C 串 u
字串"1234567890"中的"45678"(起始下标+长度)用 t 值替换，得到"123XYZ90"
字串"1234567890"中的"45678"(起始下标+长度)替换成 t 中的"XY"，得到"123XY90"
字串"1234567890"中的"45678"(起始下标+长度)替换成 u，得到"123WXVJ90"
字串"1234567890"中的"45678"(起始下标+长度)替换成 u 中的"WXV"，得到"123WXV90"
字串"1234567890"中的"45678"(起始下标+长度)替换成 3 个'A'，得到"123AAA90"
字串"1234567890"(位置区间)替换成 t，得到"XYZ"//即赋值
```

字串"1234567890"(位置区间)替换成 t 中的"YZ"，得到"YZ"
字串"1234567890"中的"4567890"(位置区间)替换成 u，得到"123WXVJ"
字串"1234567890"中的"45678"(位置区间)替换成 0 个字符'A'//即删除 45678
字串"1234567890"中的'5'位置处替换成 t，得到"12345XYZ67890" //即插入

◆ **函数声明**

则根据以下 string 成员操作声明：

```
string& replace(int pos, int n, const string& s);
string& replace(int pos1, int n1, const string& s,int pos2, int n2);
string& replace(int pos, int n1, const char* s,int n2);
string& replace(int pos, int n, const char* s);
string& replace(int pos, int n1, int n2, char c);
string& replace(iterator i1, iterator i2, const string& s);
string& replace(iterator i1, iterator i2, const string& s,int pos, int n);
string& replace(iterator i1, iterator i2, const char* s, int n);
string& replace(iterator i1, iterator i2, const char* s);
string& replace(iterator i1, iterator i2, int n, char c);
```

◆ **代码**

能够得到相应的代码：

```
//=====================================
//X0314.cpp
//string 替换操作
//=====================================
#include<iostream>
#include<string>
using namespace std;
int main(){
  string t = "XYZ";     //string 串
  char* u = "WXVJ";     //C 串

  //将"1234567890"赋给 x1,x2,x3,x4,x5,x6,x7,x8,x9,xa 变量
  string s = "1234567890";
  string x1(s),x2(s),x3(s),x4(s),x5(s),x6(s),x7(s),x8(s),x9(s),xa(s);

  //字串"1234567890"中下标为 3,长度为 5 的"45678"替换成 t，得到"123XYZ90"
  cout<<x1.replace(3,5,t)<<"\n";

  //字串"1234567890"中的"45678"替换成 t 中下标为 0,长度为 2 的"XY"，得到"123XY90"
  cout<<x2.replace(3,5,t,0,2)<<"\n";

  //字串"1234567890"中的"45678"替换成 u，得到"123WXVJ90"
  cout<<x3.replace(3,5,u)<<"\n";

  //字串"1234567890"中的"45678"替换成 u 中前 3 个字符的"WXV"，得到"123WXV90"
```

```
  cout<<x4.replace(3,5,u,3)<<"\n";

  //字串"1234567890"中的"45678"替换成 3 个'A'，得到"123AAA90"
  cout<<x5.replace(3,5,3,'A')<<"\n";

  //字串"1234567890"中从开始到结束位置，替换成 t，得到"XYZ"   //即赋值
  cout<<x6.replace(x6.begin(), x6.end(), t)<<"\n";

//字串"1234567890"中下标 0,长度为 10 的串，替换成 t 中下标 0 长度 2 的"XY"，得到"XY"
  cout<<x7.replace(0,10,t,0,2)<<"\n";

  //字串"1234567890"中的第 4 个位置到结束的"4567890"替换成 u，得到"123WXVJ"
  cout<<x8.replace(x8.begin()+3, x8.end(), u)<<"\n";

  //字串"1234567890"中的"45678"替换成 0 个字符'A'  //即删除 45678
  cout<<x9.replace(x9.begin()+3, x9.begin()+8, 0, 'A')<<"\n";

  //字串"1234567890"中的下标 5 长度 0 的串替换成 t，得到"12345XYZ67890"
  cout<<xa.replace(5,0,t)<<"\n";
}//====================================
```

◆ **代码解释**

被捆绑的操作主体是 string 串，例如代码中的 x1，x2，x3 等等。

描述主体被替换掉的部分有两种方式：一为起始下标和长度，它们用整数表示，下标和长度都应该是合理的数值，如果下标或者长度为负，则将引起运行错误；另一为起始和结束位置，它们都是容器中的一种位置类型，称为 iterator（遍历器）。起始位置用 s.begin()，向后推进 3 个字符则用 s.begin()+3，结束位置一律用 s.end()。

替换的字串则可以是 string 串、C 串、若干重复字符：

（1）描述 string 串或为整体（一个 string 串参数），或为起始下标和长度方式（一个 string 串参数，两个整型参数），两个整型参数值必须合适；

（2）描述 C 串或为整体（一个 C 串参数），或为字串从开始数起若干字符（一个 C 串参数，两个整型参数）；

（3）描述重复字符则由一个重复度和一个字符表示，一个整型参数，一个字符型参数。

编译器根据参数类型、个数和顺序，总是能分清重载函数（☞CH7.5.2 重载识别）。如果分不清，那一定是程序员搞错了。

6．查找操作

◆ **find 操作**

查找操作是基于字串中的各种搜索的需要。string 类中有一个 find 成员函数，可以完成一般的搜索工作。find 成员函数有几种形式：

```
int find(const string& str, int pos)const;
int find(const string& str)const;
```

```
int find(const char* s, int pos, int n)const;
int find(const char* s, int pos)const;
int find(const char* s)const;
int find(char c, int pos)const;
int find(char c)const;
```

该成员函数可以用 string 串去搜，也可以用 C 串去搜，也可以用字符去搜，所以该成员函数的第一个参数或为 string 串，或为 C 串，或为字符。

被搜索的操作主体可以从头搜，也可以从第二参数规定的某个位置开始搜。

如果子串是 C 串，除了规定母串从第二参数规定的位置搜之外，还可以由第三参数规定 C 串的搜索长度。所以，搜索 find 操作可以有一个参数形式、两个参数形式，甚至三个参数的形式。

所有的 find 成员函数都返回搜索到的下标位置。如果找不到，则返回-1（非标准）或者 string 类中专门描述找不到的变量 npos，意即为 no position。

例如：

```
string s ="abcdefghijklmijn";
string str1 = "ijk";
char* str2 = "ijk";
char  c ='k';
cout<<s.find(str1)<<"\n";          //结果为 8
cout<<s.find(str1, 9)<<"\n";       //母串第 10 个位置起搜,结果为 npos
cout<<s.find(str2)<<"\n";          //结果为 8
cout<<s.find(str2, 5)<<"\n";       //结果为 8,从母串第 6 个位置起搜
cout<<s.find(str2, 9, 2)<<"\n";    //结果为 13,从母串第 10 个位置起搜,子串长度 2
cout<<s.find(c)<<"\n";             //结果为 10
cout<<s.find(c, 12)<<"\n";         //结果为 npos,从母串第 12 个位置起搜
```

◆ **存在性搜索**

string 串除了 find 操作之外，还有另外几个搜索操作：

```
int find_first_not_of ( … )const;
int find_first_of( … )const;
int find_last_not_of( … )const;
int find_last_of( … )const;
```

这些搜索的功能是搜索母串的字符在子串中是否存在。_of 是搜存在，not_of 是搜不存在，first 表示从前面开始搜，last 表示从后面开始搜。

find_first_not_of 表示搜第一个不在子串（字符）中的字符元素位置。也就是说，它并不是在母串中搜完整的子串，而是搜母串字符在子串中是否存在。也就是说，依次搜母串中的字符，若母串中第一个字符在子串中存在，便搜母串第二个字符……搜啊搜，一发现母串中某字符在子串中不存在（不匹配）就停下来，该字符就是找到的位置。

而 find_first_of 与 find_first_not_of 正好相反：母串中第一个字符不在子串中，便搜母串第二个字符……搜啊搜，一发现母串中某字符在子串中（匹配）就停下来，该字符位置

就是找到位置。

对于 find_last_not_of 和 find_last_of 的操作，则母串中的字符都是从后面开始对子串进行搜索。例如，搜索串为"bcigm"，则：

```
string s ="abcdefghijklmijn";
string str = "abcigm";
cout<<s.find_first_not_of(str)<<"\n";  //结果为 3,'d'在子串中不存在
cout<<s.find_first_of(str)<<"\n";      //结果为 0,'a'在子串中存在
cout<<s.find_last_not_of(str)<<"\n";   //结果为 15,'n'在子串中不存在
cout<<s.find_last_of(str)<<"\n";       //结果为 13,'i'在子串中存在
cout<<s.find_first_of('i')<<"\n";      //结果为 8
cout<<s.find_first_not_of('f')<<"\n";  //结果为 0
cout<<s.find_last_of('i')<<"\n";       //结果为 13
cout<<s.find_last_not_of('f')<<"\n";   //结果为 15
```

与 find 操作类似，其参数都有 string 串、C 串和字符三种形式，可以从操作主体的头上开始，也可以从中间某个位置开始搜。

string 的其他操作比较容易理解，可以查看 STL 手册或者 C++开发环境的帮助信息。

3.5.2 string 流

1．串流

◆ **串流说明**

C++提供了 STL 库中的 string 串流，简称 string 流。也就是说，将 string 串当做输入/输出流设备。当做流设备，取流设备的名字，行使流操作来存取 string 串。

string 串由头文件 string 支持，而 string 流却是由头文件 sstream 支持，使用 string 流时需要包含 sstream 头文件。

string 流与文件流一样，也是一种 C++的类型，具有定义和使用的一般特征。

◆ **串输入流**

string 输入流的类型名是 istringstream。定义字串输入流的时候需要用 string 串对流初始化，因为输入流操作需要有读取内容的载体，即从 string 串中流出。例如：

```
string s("Hello How are you?");//需要包含头文件 string
istringstream s_in(s);          //s 作为初始化的值,s_in(任取的名)为串输入流名
```

取名是为了助记，s_in 表示串输入流设备。有了这个定义，便可以做下列操作：

```
char ch[10];
string x;
s_in>>ch>>x;            //从串流 s_in 中读入数据
cout<<ch<<x<<"\n";      //得到输出结果 HelloHow
```

串流 s_in>>ch 时，读入一个单词；执行 s_in>>x 时，又读入一个单词。虽然这是不同的类型，但只要标准流能辨认，串流同样能辨认。

◆ 串输出流

串输出流的类型名与串输入流类型名对应，将 i 换成 o，即 ostringstream。定义串输出流的时候，无须 string 串的初始化，串输出流将会在内部自建一个 string 串作为流入数据存放设备。定义和操作可以做下列操作：

```
ostringstream s_out;        //s_out 为串输出流名
char ch[10]="Hello ";
string x="How are you?";
s_out<<ch<<x;               //将"Hello "加上"How are you?"送串输出流 s_out
cout<<s_out.str()<<"\n";    //取串输出流中的 string 串,即"Hello How are you?"
```

用 s_out.str()操作，可以得到串输出流的内容。用标准输出流显示之，得到"Hello How are you?"。

串输入流的使用可以使本来复杂的输入操作简单化。

2．问题——求和

基本描述

对若干群整数，求和。

输入描述

输入数据中有若干行，每行有若干([1,30])整数([0,1000])，整数之间以空格隔开。

输出描述

对应每行整数，输出其和。

样本输入

```
12 3 45 67 8 9
56 232 12 23
```

样本输出

```
144
323
```

问题中有若干行数据，对不同行的数据需要分别求和。行数只是示意性的，不超过 30。输入的整数值也不大，没有计算问题，但是必须判断行尾，才能分辨每组数据而求和。

样本输入数据中，间隔的空格不一定只有 1 格，多少不一，甚至行首也有空格，所以给逐个字符读入的处理方式带来了一定的难度。

3．分析输入结构

◆ 次数控制的输入模式

如果数据个数含在输入数据之中，例如，每组数据的第一个整数 n 是后继的整数个数，并且当 n 为 0，则输入结束。即表示为：

```
5
12 3 45 67 8 9
```

```
4
56 232 12 23
0
```

或者这样排列：

```
5 12 3 45 67 8 9
4 56 232 12 23
0
```

那么，循环处理就会很方便：

```
for(int n; cin>>n && n!=0; )    //处理若干组数据,以读入 n 为 0 而结束
{
  int sum=0;
  for(int a; n--; sum+=a)            //循环控制执行 n 次
    cin>>a;
  cout<<sum<<"\n";
}
```

但是问题描述中的数据，以一行作为处理单位，并不指出行中元素个数，所以以依赖次数控制的循环处理结构便无法发挥作用。

◆ **处理一行数据**

如果输入数据仅有一行，该行中包含了若干个整数，以输入结束作为循环结束。那就很好办：

```
int sum=0;
for(int a; cin>>a; sum+=a);      //循环控制到数据输入结束
cout<<sum<<"\n";
```

这是一个很容易写的过程。

◆ **多行数据输入方式**

对于有多行数据的输入，如果能将每行分解成上面这个独立的处理过程，那就好了。

我们在 CH3.5.2 中看到，cin 的>>操作虽然可以分辨每个整数读入，但是它不能辨别空格与回车的差异,因此只能用 getline 的方式先以行为单位逐行读入数据到 string 变量中，然后在 string 串中再想办法：

```
for(string s; getline(cin,s); ){
  int sum=0;
  for(int a; (a=readIntFrom(s))>0; sum+=a);//readIntFrom(s)<0 为读完数据
    cout<<sum<<"\n";
}
```

为了容易理解，将读入一行中的整数的过程设计成函数 readIntFrom。当函数返回大于 0 的数时，该整数便是需要读入的数，同时说明行中整数还在读；当小于 0 时，说明行中数据已经读完，据此判断一行数据处理是否结束。

函数 readIntFrom 是以 string 串或 C 串（即一行数据）为参数的。在 string 变量中分离

若干个整数的循环处理，如果借助于下标访问字符元素，则需要将字符序列逐个判断出来，以空格为间隔分开，再将每个字符序列逐位转换成数字，数字再拼合还原成整数。这样的过程还是显得有点吃力，这在后面的 C 方法中还会描述。

4．串流使用

我们可以通过读入字符行 getline 操作，将一行数据存放在 string 串中，然后将该 string 串当做流设备，照着流设备循环读数据就行了。这个循环流输入的过程，也是处理一行数据的过程。于是以读入字符行的结束与否作为全部数据处理的循环的结束控制，构成了下列代码：

```
//=====================================
//X0315.cpp
//整行读入再分解读入
//=====================================
#include<iostream>
#include<sstream>
#include<string>
using namespace std;
//-------------------------------------
int main(){
  for(string s; getline(cin, s); ){     //读入字符行,若无数据,则结束
    istringstream sin(s);               //将字串当做串流设备 sin
    int sum=0;
    for(int a; sin>>a; sum+=a);         //从串流中循环读数
    cout<<sum<<"\n";
  }
}//=====================================
```

虽然读入一行的操作 getline 是字串读入的形式，但是将它作为 string 串输入流设备 sin 之后便具有了流操作的能力，能够辨别各种数据类型。所以，在一行数据处理中，从流设备 sin 中循环读入整数的操作变得很自然而且有效，代码也很简短。

sin>>a 即是从 string 流中输入整数到 a 中，循环输入的同时累加，一直到串流 sin 中输完最后一个数。

由于 string 串可以很方便地进行修改、拷贝、插入、删除、拼接、比较等，所以在串输入流之前或者串输出流之后，还能够进一步编辑流的格式，对于程序处理的统一性和方便性带来了莫大的好处。

getline 函数的返回与 cin 一样，是流状态，通过它可以判断数据输入是否结束。

5．背时的 C 串流

在 C++早些的时候，用 C 串流比较多。C 串流以 C 串为依托而不是以 string 串为依托，其定义方式与 string 流相似，流操作功能也相似，但毕竟是一组完全不同的操作。string 流的头文件是 sstream，而 C 串流却在头文件 strstream 中说明。C 串流在 C++的发展中如过

眼烟云，使用 C 串流的历史很短。如今 C++标准也已经走得很远了，所以不应再回头去用那个背时的东西。

6．C 方法

在 C 语言中，使用 C 串来存储数据行，然后单独处理一个数据行，识别和读入每个整数。设计一个识别和读入整数的函数 readIntFrom 来代替 C++版本中从串流读入整数的操作。这样便可以保持计算处理的框架不变：

```
//=====================================
//X0316.cpp
//整行读入再分解读数(C 语言版)
//=====================================
#include<stdio.h>
//-------------------------------------
int readIntFrom(char* s){
  static int n=0;                          //n 为下标,新行总是从 0 开始处理
  while(s[n]==' ') n++;                    //读数前先滤空格
  if(s[n]==0){ n=0; return -1; }           //字串结束,则重置新行状态
  int x=0;
  while(s[n]>='0' && s[n]<='9')            //读入若干数字位的整数
    x = x*10+s[n++]-'0';
  return x;                                //返回读入的整数
}//-------------------------------------
int main(){
  char s[200];                             //存放一行数据
  while( gets(s)!=0 ){                     //数据行读到 s 中,若无,则结束
    int sum=0,a;
    while( (a=readIntFrom(s))>=0 )         //>=0 作为成功读入标志
      sum+=a;
    printf("%d\n", sum);
  }
}//=====================================
```

字符数组 s 的大小设成 200，是考虑到一行数据大致上最大的可能长度。大小略微多一点没有关系，因为字串是以 0 结束的，不影响数据处理，但是多太多又占了内存空间，只是适当多一些而已。

函数 readIntFrom 以负数-1 作为数据读空的标志，从而分辨一行处理的结束。因为数据行并不是想像中的一个整数隔一个空格，最后一个整数之后就紧挨一个换行，而是行首可能有空格，空格可能有连续若干个，整数之间可能有若干空格，换行之前可能也有空格。所以，自处理输入函数必须先滤去空格，然后再判断结束标志，断定为数字符后，再对数字进行循环求值。

C 语言中有一个从 C 串中进行格式输入的操作 sscanf，本可以用它从字串中读入整数的，但是它不会记忆上次读入的位置，所以不能像串流那样担当循环读数的工作。

C 语言方法因为要满足处理框架的简单性（main 函数的描述），分解出了函数 readIntFrom。而 readIntFrom 的函数定义因为没有以字串为设备的连续格式输入（能够识别整数），而使读数操作变成为更琐碎的处理过程。但是，其处理性能并不一定因此而下降，相反，该程序相比 X0323.cpp 来说，也并不落于下风。

我们崇尚的编程是能像 C++这样的简捷而让机器去复杂去，因而我们着力在如何简单化编程，又不失去好的性能。也就是说，在学习中，既要结合编程与测试，也要知道工作内情和性能。

3.6 输 入 方 式

3.6.1 基本输入

1．流输入

学习编程中的输入控制主要是为了能满足输入一些常规的文本数据，用于编程运行和测试。所以，只要能顺利识别各种基本数据类型的字面值（整数、浮点数、字串、字符等）和空格、换行、输入（文件）结束符就能做到如意。在此前提下，学习如何进行输入，效果会更好。至于许多其他特殊的设备和格式的输入功能，在进入到实质性开发阶段时会有更自然的输入操作控制。

C++用 cin 的>>操作识别各种内部数据类型，包括 C 串、string 串。例如：

```
int a;
double b;
char s[100];
string str;              //需要 string 头文件
cin>>a>>b>>s>>str;       //正确：能识别整型、浮点型、C 串和 string 串
```

即使输入字符，也能从 cin 的>>中得到，只不过所得到的字符值不会是空格或换行字符，因为读入字符是以空格来间隔各个实体的。例如，对于输入字串"a b　c"，用下列循环去获得输入：

```
for(char c; cin>>c; )
  cout<<c<<"\n";
```

则只能得到输出：

```
a
b
c
```

结果中没有任何空格，说明 cin>>c 并没有接受空格字符。

2．C 格式输入

对于 C 语言来说，用 scanf 函数的格式参数也能识别一些内部类型，包括 C 串。例如：

```
int a;
double b;
char s[100];
scanf("%d%lf%s", &a,&b,s);     //正确：识别整型、浮点型、C 串
```

scanf 的格式参数，如%d 能识别整型数，%lf 能识别双精度浮点数，但是 scanf 并不能规范地识别 string 串，并且也永远不能识别自定义的其他数据类型。不能识别就意味着不能通过这个函数读入相关类型的数据，意味着这些类型必须要专门写输入函数去对付数据的读入。一种语言这么对待新生代数据类型的输入，被人抨击就十分正常了。这也是 C++用自定义的类机制来开拓 C 之后，又用 IO 流来改造 C 语言的 IO 设备的重要原因。流输入设备 cin 能通过操作符重定义（☞《C++程序设计教程详解——对象化编程》）来识别任何自定义类型。

所有上述的输入操作都以空格来间隔一个识别体。也就是说，读入的字串中不可以有空格或换行符。例如，输入数据为“I am a student.”，通过输入操作：

```
char s[100];
scanf("%s", s);
```

s 在读入“I”之后遇到空格，便只能得到字串"I"，而不能得到整个句子。不过，scanf 可以让字符型输入读入空格：

```
for(char c; scanf("%c",&c)!=EOF; )
   printf("%c", c);
printf("\n");
```

便能得到“I am a student.”整个句子，而不是不带空格的"Iamastudent."字串。

3.6.2 文件输入

1．缓冲原理

计算机要操作文件，总是借助于缓冲设备。任何磁盘文件（硬件存储的数据）都可以对应一个运行中的虚拟文件设备。如果该文件作为输入用，则该虚拟文件设备负责将磁盘文件数据的需要部分（很小一块）读入到缓冲区，以供程序读入之用。

设置缓冲区的目的是同步读与取的操作。例如，工地上需要许多砖块，汽车所运来的砖块先用吊车卸下（犹如放入缓冲区），然后再被使用者装小车取走（犹如读数据）。如果直接由工人从汽车上取砖块，就会延迟汽车离开，影响运输总量，从而影响工程进度。如果使用者直接到砖块仓库取，不用汽车运输，则取的人太多，还有放砖块的人，会搞乱仓库管理。何况取砖的人需要来回走很长的路，工作量加大，不利于工作。因而，设置堆放场地（缓冲区）能充分发挥汽车运输能力，同时，工人在没有砖块的情况下，也可以先从

事别的工作，从而不慌不忙地保证工程的进度。其付出的代价则仅仅是开辟出一个砖块堆放场（缓冲区）。

设备工作速度相对于 CPU 的计算速度来说要慢很多。当设备数据慢慢倒入缓冲区时，操作系统或许可以让别的不需要该缓冲数据的程序先运行一阵子，等到缓冲区满了再通知本程序可以接着运行了，从而使读取数据工作流畅，而不需要等待了。当有文件设备在程序中一起工作的时候，依赖于缓冲区就能使系统兼顾别的程序，工作效率得以提高，这也是对文件读入读出操作的普遍做法。

缓冲设备的另一个特点是设备中有一个缓冲区的读取位置，根据位置便可随时发现缓冲区是为空或者为满。有了读取位置，便为连续读入数据创造了条件，不会发生读一个数据后下一个数据就找不到的情况。

标准设备都是带有缓冲区的设备。所谓标准，就是其操作可以当做其他设备操作的共同规范；所谓输入缓冲设备，就是锁定了输入源，便可以连续地从中输入数据了。

2．文件流

文件设备可以捆绑到标准设备，从而成为带有缓冲区的设备。C++中，将文件捆绑到对应的输入文件流设备之后，所有的操作就是流操作，所读入的数据为从文件中读入的数据，于是就等于是在操作文件。所以，它是从文件读入数据的一种手段。

要做这样的文件操作，需要文件流头文件 fstream，表示 file stream。例如，文件 text.txt 中有内容：

```
1 2 3 4 5 6
7 8 9
```

则要将该文件中所有数据加起来输出的计算操作为：

```
//=====================================
//X0317.cpp
//文件操作
//=====================================
#include<fstream>
#include<iostream>
using namespace std;
//-------------------------------------
int main(){
  ifstream fin("text.txt");
  int sum=0;
  for(int a; fin>>a; )
    sum += a;
  cout<<sum<<"\n";
}//=====================================
```

文件流可以规定其为输入型还是输出型。输入文件流是一种类型，称为 ifstream，意即 input file stream。定义该类型的一个实体 fin，用文件名”text.txt”对其进行初始化，就达

到了将磁盘文件捆绑到程序中的文件流设备的目的。之后，对文件流的操作就是对磁盘文件的操作。代码中 for 循环的输入操作用文件流 fin 代替了标准流 cin，以达到将文件数据读入程序进行计算的目的。

用文件流来操作文件很方便，而且，编辑文本文件在操作系统中也很方便，这使得许多程序调试所要用的输入数据可以预先存放在文件中。特别是当数据量很大时，用标准输入流在键盘上操作的工作量会很大，此时用文件流操作的办法来驾驭数据输入便很妥当。若将输入文件流定义成 cin（覆盖了标准输入流名），则可以通过注释文件流定义语句的办法而使程序的输入方便地从文件操作切换到标准输入操作。例如：

```
//=====================================
//X0318.cpp
//文件流覆盖标准流
//=====================================
#include<fstream>
#include<iostream>
using namespace std;
//-------------------------------------
int main(){
  //ifstream cin("text.txt");  //注释掉则是标准流操作,反之是文件流操作
  int sum=0;
  for(int a; cin>>a; )
    sum += a;
  cout<<sum<<"\n";
}//=====================================
```

当程序中多处出现流输入操作时，不需要修改流名 cin 便可以切换文件操作与标准输入操作，从而显现这种技巧所带来的输入操作的灵活性。

C++的流方法用的是覆盖法，也就是原来的标准输入流仍然存在，定义一个 cin 的文件流，同名同姓而覆盖了原 cin 流，导致原来的 cin 失效，而整篇都是文件流操作。一旦文件流退出历史舞台，则原 cin 又处于活动状态了。定义同名同姓的实体定义按道理是非法的，但是标准流是在 STL 标准库中定义，而文件流是在程序主函数中定义，所以没有构成直接冲突，只是以近的 cin 覆盖了远处的 cin。

3．C 文件格式输入

同样的计算要求，用 C 的方法也可对文件进行操作，但是它不是将文件与文件流设备捆绑，而是将文件与标准输入设备捆绑。因为标准输入设备已经存在，调用函数固定，要使文件操作像标准输入设备那样操作，必须使文件等同于标准输入设备。因此，调用一个 freopen 函数将标准输入设备重新打开，从键盘中脱钩，与文件挂钩，实际上就是将文件与标准输入设备捆绑：

```
//=====================================
//X0319.cpp
//文件设备
```

```
//=====================================
#include<stdio.h>
//-------------------------------------
int main(){
  freopen("text.txt","r",stdin); //注释掉是键盘输入,反之是文件输入
  int sum=0;
  for(int a; scanf("%d", &a)!=EOF; )
    sum += a;
  printf("%d\n",sum);
}//=====================================
```

scanf()与 printf()函数都是标准设备的操作，因为输入设备的重新捆绑，使得 scanf 输入数据来自文件 text.txt，而输出仍然面向控制台窗口。

freopen 函数有三个参数，第一参数为所捆绑的文件名 text.txt，第二参数为读写方式，"r"表示读方式，第三参数为缓冲设备，这里用标准输入设备 stdin。于是，所有从标准输入设备中读数据的操作都是针对文件操作。文件操作的种种参数，在初学时暂时不要去深究，这些功能够用就行，不要搞得复杂化了。

3.6.3　控制台输入

控制台输入仅用于 console 应用编程。控制台设备的输入函数直接接受键盘输入。不用缓冲设备的标准输入而直接用控制台输入函数来输入，适合于控制台窗口程序的对话式运行。例如，下面用到的 getch()、cputs()、clrscr()函数等。使用控制台函数须包含头文件 conio.h。

```
//=====================================
//X0320.cpp
//控制台输入
//=====================================
#include<conio.h>  //getch(), cputs(), clrscr()
//-------------------------------------
int main(){
  char* arr[]={ "good\n\r", "better\n\r", "best\n\r" };
  for(int choice; 1; ){
    cputs("1-----display good\r\n");
    cputs("2-----display better\r\n");
    cputs("3-----display best\r\n");
    cputs("0-----exit\r\n");
    cputs("Enter your choice:");
    if((choice=getch()-'1')!=-1);    //get a console character
      break;
    clrscr();                        //清屏 clear screen
    switch(choice){
      case 1: case 2: case0: cputs(arr[choice]); break;
```

```
        default: cputs("you entered a wrong key.\a\r\n");
      }
    }
}//==================================
```

该程序全部使用控制台输入/输出。

getch()函数是从控制台（键盘）读入一个字符。因为是字符而不是整数，所以数字字符与整数之间有一个'0'的偏差，例如，'3'-3='0'，所以'3'与 3 相差'0'即整数 48，这个偏差需要在开关语句跳转之前将其减去。代码中的 if 语句做的就是这个操作。该函数在键盘按下的瞬间，即得到了该字符。同样，getche()函数（此处没有用到）也是从控制台读入一个字符，它们之间的差别为 getche()还将该字符回显到控制台上，而 getch()读入之后，输入字符并不回显。

clrscr()函数是将控制台窗口清屏，将当前显示位置置于左上。

cputs()函数以 C 串为参数，直接将该字串显示在控制台窗口的当前位置上。由于对控制台是纯字符操作，所以，需要对控制台坐标位置进行控制。换行只是换行，还需要回车('\r')控制才能重新在下一行的起点显示新的字串。控制符'\a'为机内喇叭鸣“嘟”声。对错误操作以鸣叫声警示，是控制台编程中的一种原始手段。

代码中用了字符指针类型的数组 arr，用三个 C 串给予其初始化，这是对成组的字串字面值使用的一种示范。

用控制台输入/输出从性能上讲，因为它更接近物理设备，所以比缓冲设备的输入/输出要好，但由于它只与控制台发生关系，不与缓冲式设备发生关系，所以无法与其他标准设备的数据进行联络。

3.6.4 标准键盘输入

1．键盘数据发送

Console 应用程序的运行调试往往使用标准输入设备来输入数据，验证程序的正确性。因为标准输入设备可以重定向（☞CH8.3.1 重定向）到任何设备，所以可以在测试时从系统指定的设备读入数据。

编程运行中，我们经常面临用标准输入的默认设备（键盘）去调试程序的问题。

无论是 C 语言中的标准输入设备 stdin，还是 C++中的标准输入流 cin 设备，它们默认时都与键盘设备捆绑，所以在运行中对 stdin 或者 cin 的操作就能够接受键盘输入。

标准输入设备行使键盘输入（控制台输入）时有一个缺点，就是需要人为键盘发送输入数据到输入设备的缓冲区中去，典型的操作就是按下“回车”，否则输入数据将逗留在控制台上迟迟不能被输入变量所接受。例如：

```
char c;
scanf("%c", &c);   //相当于 cin>>c;
```

这样的代码如果编进程序，在键盘执行输入时虽然键入了十七八个字符，但是程序执行还是无动于衷，一直要等到用户按下了“回车”才将键入的所有字符中的第一个字符移

交给变量 c。同时，其他的字符连带换行符都存储在设备的缓冲区中，以备再次输入之用。流输入亦如此。

2．数据结束标志

标准输入的这个缺点也导致了输入数据的循环需要人为控制其结束。例如，求一组整数（需要输入）的和，对于代码 X0324.cpp 来说，为了求 1 2 3 4 5 6 7 8 9 这九个数据的和，便在键盘上录入了这九个数据，按下“回车”之后，程序仍然处于输入等待状态，等到按下 Ctrl+Z 之后，不！还得按“回车”，发送这个标志着数据输入结束或者文件尾的 EOF（End of File）控制字符时，程序才得以显示结果 45。注意，在 VC 平台中，需要在按下“回车”后，再按任意键，才能看到运行结果。

但如果是由文件输入，即使像这样需要全部数据输入之后才能接着计算以求结果的程序也没有这个问题。因为文件数据中的结尾处都有 EOF 符，数据读完后，紧接着再读就读到一个 EOF，从而导致像 X0323.cpp 那样的代码中 for 循环的结束，顺畅地得到输出结果。

当然，有许多程序的输入数据与运行结果之间，一起一落，输入一个数据，立即计算获得一个结果，输出之后，又输入一个数据，获得一个结果，这样的计算结构便没有键盘输入人为发送 Ctrl+Z 的必要。

3.6.5　读入行

1．读行格式

有些时候输入数据需要以行为单位进行计算处理，这就需要输入时能识别换行符。但是由基本输入的知识，输入流 cin 的>>操作是以空格（包括换行、制表符）来间隔各个数据的，也就是说，空格与换行对于 cin 流的>>操作是没有区分的，或者说，不能单独识别换行符，所以需要新的流操作来解决识别换行符问题。

在 C++中 getline 操作可以将一行字串读入 string 串中。格式为：

```
istream getline(istream& is, string s);
istream getline(istream& is, string s, char delim);
```

第一种形式是读入一行字串到 s 中，当输入字符为换行符时，读入过程结束。string 变量获得的一行字串并不包括结尾的换行符，也就是说，getline 虽然遇到了换行，但存储到 string 实体中去的时候，却把换行符给抹掉了。

第二种形式也是读入一行字串到 s 中，但有一个第三参数——当遇到 delim 字符时，读入过程结束。也就是说，getline 不但可以识别换行符，还可以识别任何别的字符。

2．识别换行

识别换行的读行 getline，不需要有第三参数。例如，文件 tt.txt 的内容为：

```
Read them in one place with the Reader,
where keeping up with your favorite
```

```
websites.
```

读入文件内容，将其在控制台窗口输出，可以如下编程：

```
//=====================================
//X0321.cpp
//读行
//=====================================
#include<fstream>
#include<string>
#include<iostream>
using namespace std;
//-------------------------------------
int main(){
  ifstream cin("tt.txt");
  for(string s; getline(cin, s); )
    cout<<s<<"\n";
}//=====================================
```

代码中每读入一行，就输出一行，并且输出在读入过程中每次都被抹掉的换行，从而将文件内容完整地输出。

3．识别指定字符

如果要以字符'u'为行结束标记来输出文件内容，则可以写成：

```
//=====================================
//X0322.cpp
//读行(以'u'为结束)
//=====================================
#include<fstream>
#include<string>
#include<iostream>
using namespace std;
//-------------------------------------
int main(){
  ifstream cin("tt.txt");
  for(string s; getline(cin, s, 'u'); )//遇到'u'则行结束,到文件尾则循环结束
    cout<<s<<"\n";
}//=====================================
```

若运行之，则得到下列结果：

```
Read them in one place with the Reader,
where keeping
p with yo
r favorite
websites.
```

这个时候，输出的字串 s 中不会含有字符'u'，但可能会含有换行符。

我们看到文件中只含有两个字符'u'，但是输出却有 5 行，其中有两次读入的字串中含有换行符，造成了输出字串 s 的操作在多行中完成。

4．C 方式

对应 C 语言，其读入一行 C 串的操作为 gets，可以用下列代码完成同样的文件输出任务：

```
//=====================================
//X0323.cpp
//读行(C 语言版)
//=====================================
#include<stdio.h>   //gets(), puts(), freopen()
//-------------------------------------
int main(){
  freopen("tt.txt","r",stdin);
  for(char s[100]; gets(s); )
    puts(s);
}//=====================================
```

假定每行字串不超过 100 个字符，所以 for 循环创建了一个长为 100 的字符数组。虽然 gets 函数读入一行，滤去了换行，但是 puts 函数在输出时，又将换行给补了回去，所以，程序运行能够一个字符不差地将文件内容输出。

3.6.6　读入字符

1．C++方式

有些时候，需要将字符（包括空格换行等字符）逐个读入，参与计算，显然根据基本输入的知识，用 cin 的>>操作是无能为力的。

C++用 cin 的成员函数 get 操作，可以获得缓冲设备中的任何字符。例如，输出文件内容，也可以用下列代码：

```
//=====================================
//X0324.cpp
//读字符
//=====================================
#include<iostream>
#include<fstream>
using namespace std;
//-------------------------------------
int main(){
  ifstream cin("tt.txt");
  for(char c; (c=cin.get())!=EOF; )  //读入一个字符给字符变量 c
```

```
    cout<<c;
}//=====================================
```

c=cin.get()是赋值表达式，它获得输入设备中的一个字符给变量 c。同时这个表达式的值也是这个 c 的值，将其与 EOF 比较，构成循环结束的条件，便可以将文件中所有的字符读个遍，将其逐个输出，就是文件原文了。

2．C 方式

C 语言中的读字符操作，可以用 getchar()函数直接从标准输入设备中读，也可以用 getc()函数从选定的缓冲设备中读。例如，将文件内容搬到控制台窗口中去：

```
//=====================================
//X0325.cpp
//输入字符
//=====================================
#include<stdio.h>
//-------------------------------------
int main(){
  freopen("tt.txt","r",stdin);
  for(char c; (c=getchar())!=EOF; )  //等价于 getc(stdin)
    putchar(c);
}//=====================================
```

这种读字符的方式比标准格式输入的下列代码

```
//=====================================
//X0326.cpp
//输入格式字符
//=====================================
#include<stdio.h>
//-------------------------------------
int main(){
  freopen("tt.txt","r",stdin);
  for(char c; scanf("%c",&c)!=EOF; )
    printf("%c",c);
}//=====================================
```

性能要好一些，也比 C++的流字符输入操作要好一些。在编程中，若是对读字符的性能很计较，可以拿 X0325.cpp 所用的方法一试。

3.7 目 的 归 纳

要解决具体问题，必须首先要学会用数据类型来描述问题中的具体事物。

C++提供了两大内部数据类型：整型和浮点型。

3.7.1　整型数

整型是所有内部结构为二进制数表示方式的总代表。其类型的变化形式有 **bool**、**char**、**short int**、**long int**、**long long int** 以及它们各自的无符号整型。甚至 **enum** 枚举也是按整型原理构造枚举符的值。**long long int** 是 64 位整型，随着计算机硬件的发展，它在简单编程中也将会经常出现。在这之前，还有**__int64** 这个过渡类型也在用，虽然它不是标准 64 位整型，但考虑到其历史性，各个编译器都兼容了它。

在编程中，人们大量使用整型数，这是由计算机内部构造的特殊性所决定的。人们用整型数表示 ASCII 码，表示数组下标，表示开关 **switch** 语句的入口编号，表示枚举值，表示循环变量，表示字串长度，表示内存空间地址，表示错误编号，表示设备编号，表示申请内存空间的数量，表示各个实体的空间大小，等等。甚至逻辑值与逻辑值表达式运算都表示以整型数。在问题解决过程中，更多地表示以各类数据值。

字符型在编译的类型识别上有别于整型，表现在输入/输出中尤为明显。字符也是字串的组成元素，在 string 类型和 C 串的下标访问中频繁使用。字符值作为整型数，相容于整型数的计算。

整型数在计算机内部所对应的二进制数，很容易通过其移位、位与、位或等位操作，截取一个整数的任何一位二进制信息，改变其整数值，从而进行更细微的编程计算与控制。

计算机的世界是二进制数的世界，决定了整型数在编程中被高频率地使用。

3.7.2　浮点型

浮点型数都符合国际 IEEE754 和 854 标准。它们有最早的 **float** 型和现在的主流 **double** 型以及长浮点型 **long double**。其差别在位数上。

float 为 32 位。对于 32 位编译器来说，C++程序都被默认编译成按 64 位的 **double** 型操作。使用 **float** 时，要先在 CPU 中转换成 **double** 进行计算，完成后再被转换回来，所以 **float** 相对 **double** 来说，无论精度和运行速度都有点背时了。

浮点数多用于科学计算，可以通过输入/输出控制来表示长度、有效位数和精度。**long double** 在低版本编译器中的实现并不完善，存在一些表达上的错误，所以在十分计较精确度的场合，建议在更高版本的 C++编译器中使用 **long double**。

3.7.3　C 串

C 串是 C/C++语言中表示字串的最基本方式。C 串由若干字符组成，结束于 0 整数值，在空间上占用比字符数多 1 个的字节数。C 串在编程中用双引号括起一些字符来表示。C 串在类型上不同于字符，不能与字符进行比较，由于 C 串实际上是一种字符指针(☞CH3.4.1 字符指针)，所以字串的比较也无法通过 C 串的相等与否来完成。

3.7.4 string

C++有两种字串——C 串和 string 串。两种字串可以相互转换。例如，可以将 string 串表示的数字串转换成 C 串后，调用 atoi 函数将其转变成整型数；可以将 C 串转换成 string 串后，进行字串的搜索（find）和拼接（+）操作。

string 串的操作比较直接，而 C 串的操作往往要通过 C 库函数调用来完成；string 串比 C 串具有更多、更全面的操作；输入/输出流对 C 串和 string 串都能识别和控制，但是 C 库函数 scanf 和 printf 却只能操作 C 串。string 串比 C 串在编程上会更加优雅和安全，所以初学者一方面要了解 C 串，另一方面还应多选择 string 进行字串表达和计算。

3.7.5 输入方式

数据输入的对象不外乎字串、整数和浮点数。数据输入的逻辑设备除了控制台之外，还有文件设备和字串模拟设备，它们都可以定义为流输入设备，这便使得输入操作简单化和规范化了。

流操作读取整型数（包括字符）和浮点数时，按类型为单位识别；读取字串时，可以选择按单词读取（cin）或按行读取（getline）。

3.8 练　　习

A. 1!到 n!的和(1174)

- **基本描述**

求 1!+2!+3!+4!+…&+n!的结果。

- **输入描述**

输入数据含有一些正整数 n（1≤n≤12）。

- **输出描述**

对于每个 n，输出计算结果。每个计算结果应占独立一行。

- **样本输入**

```
3 6
```

- **样本输出**

```
9
873
```

B．等比数列 (1175)

◆ 基本描述

已知 q 与 n，求等比数列之和:

$1+q+q^2+q^3+q^4+\cdots+q^n$

◆ 输入描述

输入数据含有一些数据对，每对数据含有一个整数 n(1≤n≤20)和一个小数 q(0<q<2)。

◆ 输出描述

对于每组数据 n 和 q，计算其等比数列的和，精确到小数点后 3 位，每个计算结果应占单独一行。

◆ 样本输入

```
6 0.3 5 1.3
```

◆ 样本输出

```
1.428
12.756
```

C．平均数 (1179)

◆ 基本描述

求若干个整数的平均数。

◆ 输入描述

输入数据含有一些数据组,每组数据由一个整数 n(n≤50)开头,表示后面跟着 n 个整数。

◆ 输出描述

对于每组数据，输出其平均数，精确到小数点后 3 位，每个平均数应占单独一行。

◆ 样本输入

```
3 6 5 18
4 1 2 3 4
```

◆ 样本输出

```
9.667
2.500
```

D．级数求和 (1180)

◆ 基本描述

求下列级数的和:

$$1+x-\frac{x^2}{2!}+\frac{x^3}{3!}-...(-1)^{n+1}\frac{x^n}{n!}$$

◆ 输入描述

输入数据含有一些浮点数 x(0.0≤x≤40.0)。

◆ 输出描述

对于每个 x，将所有大于 10^{-6} 的数据项汇总，输出其级数和，精确到小数点后 6 位，每个 x 对应的级数计算结果应占单独一行。其格式见样本输出。

◆ 样本输入

```
2.1 3.2
```

◆ 样本输出

```
x=2.100000,sum=1.877544
x=3.200000,sum=1.959238
```

E．整数内码 (1183)

◆ 基本描述

将十进制整数的内部 32 位二进制码输出。

◆ 输入描述

输入数据中含有一些整数 n（$-2^{31}<n<2^{31}$）。

◆ 输出描述

对于每个 n，输出其对应的 32 位二进制码与原整数，中间空 2 格，每个结果对应一行输出。

◆ 样本输入

```
5 12 -12
```

◆ 样本输出

```
00000000000000000000000000000101  5
00000000000000000000000000001100  12
11111111111111111111111111110100  -12
```

F．算菜价 (1700)

◆ 基本描述

妈妈每天都要出去买菜，但是回来后也懒得数兜里的钱，到底花了多少钱真是一笔糊涂帐。现在好了，作为好儿子（女儿）的你可以给她用程序算一下了，呵呵。

妈妈买每样菜都要付钱，付钱都是采用四舍五入的方法，把零头去掉。例如，买 2 个鸡腿，每个 4.2 元，付 8 元钱。妈妈回来时，根据菜篮子里菜的品种和数量，花了多少钱呢？

◆ 输入描述

输入含有一些数据组，每组数据包括菜种（字串），数量（可能为小数）和单价（可能为小数，单位为人民币元），因此，每组数据的菜价就是数量乘上单价啊。菜种、数量和单价之间都以空格隔开的。

◆ 输出描述

输出一个精度为角的菜价总量。

◆ 样本输入

```
青菜 1  2
萝卜 2  1.52
鸡腿 2  4.2
```

◆ 样本输出

```
13.4
```

G. 去掉双斜杠注释 (1216)

◆ 基本描述

将 C++程序代码中的双斜杠注释去掉。

◆ 输入描述

输入数据中含有一些符合 C++语法的代码行。需要说明的是，为了方便编程，规定双斜杠注释内容中不含有双引号。

◆ 输出描述

输出不含有双斜杠注释的 C++代码， 原语句行格式不变，行尾也不应有空格。

◆ 样本输入

```
//======================
//simplest program
//======================
#include<iostream>
using namespace std;
//----------------------
int main(){
  cout<<"hello world!\n";
}//======================
```

◆ 样本输出

```
#include<iostream>
using namespace std;
int main(){
  cout<<"hello world!\n";
}
```