附录 A　计数系统

人们使用很多计数系统来表示数字。有些计数系统（如罗马数字）不适合用于算术运算；而印度计数系统经过改进，并传入到欧洲后变成了阿拉伯计数系统，这种数字方便了数学、科学和商业计算。现代的计算机计数系统是基于占位符概念的，使用了最先出现在印度计数系统中的零。然而，这种原理被推广到其他计数系统。因此，虽然在日常生活中使用的是下一节将介绍的十进制，但计算领域通常使用八进制、十六进制和二进制。

A.1　十进制数

数字的书写方式是基于 10 的幂数。例如，对于数字 2468，2 表示 2 个 1000，4 表示 4 个 100，6 表示 6 个 10，8 表示 8 个 1。

$2468=2\times1000+4\times100+6\times10+8\times1$

一千是 $10\times10\times10$ 或 10 的 3 次幂，用 10^3 表示。使用这种表示法，可以这样书写上述关系：

$2468=2\times10^3+4\times10^2+6\times10^1+8\times10^0$

因为这种数字表示法是基于 10 的幂，所以将它称作基数为 10 的表示法或十进制表示法。可以用任何数作基数。例如，C++允许使用基数 8（八进制）和基数 16（十六进制）来书写整数（请注意，10^0 为 1，任何非零数的 0 次幂都为 1）。

A.2　八进制整数

八进制数是基于 8 的幂的，所以基数为 8 的表示法用数字 0-7 来书写数字。C++用前缀 0 来表示八进制表示法。也就是说，0177 是一个八进制值。可以用 8 的幂来找到对应的十进制值：

八　进　制	十　进　制
177	$=1\times8^2+7\times8^1+7\times8^0$
	$=1\times64+7\times8+7\times1$
	$=127$

由于 UNIX 操作系统常使用八进制来表示值，因此 C++和 C 提供了八进制表示法。

A.3　十六进制数

十六进制数是基于 16 的幂的。这意味着十六进制的 10 表示 16+0，即 16。为表示 9-16 值，需要其他一些数字，标准的十六进制表示法使用字母 a-f。C++接受这些字符的大写和小写版本，如表 A.1 所示。

表 A.1 十六进制数

十六进制数	十 进 制 值
a 或 A	10
b 或 B	11
c 或 C	12
d 或 D	13
e 或 E	14
f 或 F	15

C++使用 0x 或 0X 来指示十六进制表示法。因此 0x2B3 是一个十六进制值，可使用 16 的幂来得到对应的十进制值。

十六进制	十 进 制
0x2B3	$=2\times16^2+11\times16^1+3\times16^0$
	$=2\times256+11\times16+3\times1$
	$=691$

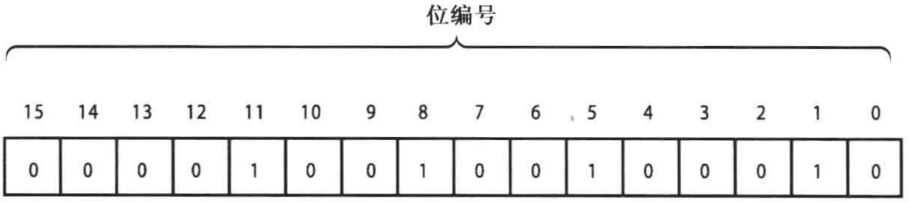
硬件文档常使用十六进制来表示诸如内存单元和端口号等值。

A.4 二进制数

不管是使用十进制、八进制，还是十六进制表示法来书写整数，计算机都将它存储为二进制值（即基数为 2）。二进制表示法只使用两个数字——0 和 1。例如，10011011 就是二进制数。但 C++没有提供二进制表示法来书写数字的方式。二进制数是基于 2 的幂。

二 进 制	十 进 制
10011011	$=1\times2^7+0\times2^6+0\times2^5+1\times2^4+1\times2^3+0\times2^2+1\times2^1+1\times2^0$
	$=128+0+0+16+8+0+2+1$
	$=155$

二进制表示法与计算机内存完全对应，在内存中，每个单元（位）都可以设置成开或关。只是将关表示为 0，将开表示为 1。内存通常是以字节为单位组织的，每个字节包含 8 位（正如第 2 章指出的，C++字节并非一定是 8 位，但本附录采用常见的做法，用字节表示八位组）。字节中的位被编号，对应于相关的 2 的幂。这样，最右侧的位编号为 0，然后是 1，依此类推。例如，图 A.1 表示一个 2 字节的整数。



$$\begin{aligned}\text{值} &= 1\times2^{11}+1\times2^8+1\times2^5+1\times2^1\\ &= 2048+256+32+2\\ &= 2338\end{aligned}$$

图 A.1 2 字节整数值

A.5　二进制和十六进制

十六进制表示法常用于提供更为方便的二进制数据（如内存地址或存储位标记设置的整数）视图。这样做的原因是，每个十六进制位对应于 4 位。表 A.2 说明了这种对应关系。

表 A.2　十六进制数和对应的二进制数

十六进制位	对应的二进制数
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

要将十六进制值转换为二进制，只需将每个十六进制位替换为相应的二进制数即可。例如，十六进制 0xA4 对应于二进制数 10100100。同样，可以轻松地将二进制值转换为十六进制，方法是将每 4 位转换为对应的十六进制位。例如，二进制值 10010101 将被转换为 0x95。

Big Endian 和 Little Endian

奇怪的是，都使用整数的二进制表示的两个计算平台对同一个值的表示可能并不相同。例如，Intel 计算机使用 Little Endian 体系结构来存储字节，而 Motorola 处理器、IBM 大型机、SPARC 处理器和 ARM 处理器使用 Big Endian 方案（但最后的两种系统可配置成使用上述任何一种方案）。

术语 Big Endian 和 Little Endian 是从“Big End In”和“Little End In”（指内存中单词（通常为两个字节）的字节顺序）衍生而来的。在 Intel 计算机（Little Endian）中，先存储低位字节，这意味着十六进制值 0xABCD 在内存中将被存储为 0xCD 0xAB。Motorola（Big Endian）计算机按相反的顺序存储，因此 0xABCD 在内存中被存储为 0xAB 0xCD。

这些术语最先出现在 Jonathan Swift 编写的《Gulliver’s Travels》一书中。为讽刺众多政治斗争的非理性，Swift 杜撰了假想国中两个喜欢争论的政治派别：Big Endians 和 Little Endians，前者坚持认为从大的一头打破鸡蛋更合理，而后者坚持认为从小的一头打破鸡蛋更合理。

作为软件工程师，应了解目标平台的词序，它会影响对通过网络传输的数据的解释方式以及数据在二进制文件中的存储方式。在上面的例子中，二字节内存模式 0xABCD 在 Little Endian 计算机上表示十进制值 52651，而在 Big Endian 计算机上表示十进制值 43981。

附录 B　C++保留字

C++保留了一些单词供自己和 C++库使用。程序员不应将保留字用作声明中的标识符。保留字分三类：关键字、替代标记（alternative token）和 C++库保留名称。

B.1　C++关键字

关键字是组成编程语言词汇表的标识符，它们不能用于其他用途，如用作变量名。表 B.1 列出了 C++关键字，其中以粗体显示的关键字也是 ANSI C99 标准中的关键字，而以斜体显示的关键字是 C++11 新增的。

表 B.1　　C++关键字

alignas	alignof	asm	**auto**	bool
break	**case**	catch	**char**	char16_t
char32_t	class	**const**	const_cast	constexpr
continue	decltype	**default**	delete	**do**
double	dynamic_cast	**else**	**enum**	explicit
export	**extern**	false	**float**	**for**
friend	**goto**	**if**	**inline**	**int**
long	mutable	namespace	new	noexcept
nullptr	operator	private	protected	public
register	reinterpret_cast	**return**	**short**	**signed**
sizeof	**static**	static_assert	static_cast	**struct**
switch	template	this	thread_local	throw
true	try	**typedef**	typeid	typename
union	**unsigned**	using	virtual	**void**
volatile	wchar_t	**while**		

B.2　替代标记

除关键字外，C++还有一些运算符的字母替代表示，它们被称为替代标记。替代标记也被保留，表 B.2 列出了替代标记及其表示的运算符。

表 B.2　　C++保留的替代标记及其含义

标　记	含　义
and	&&
and_eq	&=

续表

标　记	含　义
bitand	&
bitor	|
compl	~
not	!
not_eq	!=
or	||
or_eq	|=
xor	^
xor_eq	^=

B.3　C++库保留名称

编译器不允许程序员将关键字和替代标记用作名称。还有另一类禁止使用（但并非绝对不能用）的名称——保留名称，它们是保留给C++库使用的名称。如果您将这种名称用作标识符，后果将是不确定的。也就是说，可能导致编译器错误、警告、程序不能正确运行或根本不会导致任何问题。

C++语言保留了库头文件中使用的宏名。如果程序包含某个头文件，则不应将该头文件（以及该头文件包含的头文件，依此类推）中定义的宏名用作其他目的。例如，如果您直接或间接地包含了头文件<climits>，则不应将CHAR_BIT用作标识符，因为它已被用作该头文件中一个宏的名称。

C++语言保留了以两个下划线或下划线和大写字母打头的名称，还将以单个下划线打头的名称保留用作全局变量。因此，程序员不能在全局名称空间使用诸如__gink、__Lynx和_lynx等名称。

C++语言保留了在库头文件中被声明为链接性为外部的名称。对于函数，这包括函数的特征标（名称和参数列表）。例如，假设有如下代码：

```
#include <cmath>
using namespace std;
```

则函数特征标tan（double）被保留。这意味着您的程序不应声明一个原型如下所示的函数：

```
int tan(double);  // don't do it
```

该原型确实与库函数tan( )的原型不同，因为后者的返回类型为double，但特征标部分确实相同。然而，定义下面的原型是可以的：

```
char * tan(char *);  // ok
```

这是因为虽然其名称与库函数tan( )相同，但特征标不同。

B.4　有特殊含义的标识符

C++社区讨厌新增关键字，因为它们可能与现有代码发生冲突。这就是标准委员会改变关键字auto的用法，并赋予其他关键字（如virtual和delete）新用法的原因所在。C++11提供了另一种避免新增关键字的机制，即使用具有特殊含义的标识符。这些标识符不是关键字，但用于实现语言功能。编译器根据上下文来判断它们是常规标识符还是用于实现语言功能：

```
class F
{
    int final;                          // #1
public:
...
    virtual void unfold() {...} = final;  // #2
};
```

在上述代码中，语句#1 中的 final 是一个常规标识符，而语句#2 中的 final 使用了一种语言功能。这两种用法彼此不会冲突。

另外，C++还有很多经常出现在程序中，但不被保留的标识符。这包括头文件名、库函数名和 main（必不可少的函数的名称，程序从该函数开始执行）。只要不发生名称空间冲突，就可将这些标识符用于其他目的，但没有理由这样做。也就是说，完全可以编写下面的代码，但常识告诉您不应这样做：

```
// allowable but silly
#include <iostream>
int iostream(int a);
int main ()
{
    std::
cout << iostream(5) << '\n';
    return 0;
}

int iostream(int a)
{
    int main = a + 1;
    int cout = a -1;
    return main*cout;
}
```

附录 C　ASCII 字符集

计算机使用数字代码来存储字符。ASCII 码是美国最常用的编码，它是 Unicode 的一个子集（一个非常小的子集）。C++使得能够直接表示大多数字符，方法是将字符用单引号括起，例如'A'表示字符 A。也可以用前面带反斜杠的八进制或十六进制编码来表示单个字符，例如，'\012'和'\0xa'表示的都是换行符（LF）。这种转义序列还可放在字符串中，如“Hello, \012my dear”。

表 C.1 列出了以各种方式表示的 ASCII 字符集。在该表中，当被用作前缀时，^字符表示使用 Ctrl 键。

表 C.1　　ASCII 字符集

十　进　制	八　进　制	十 六 进 制	二　进　制	字　　符	ASCII 名称
0	0	0	00000000	^@	NUL
1	01	0x1	00000001	^A	SOH
2	02	0x2	00000010	^B	STX
3	03	0x3	00000011	^C	ETX
4	04	0x4	00000100	^D	EOT
5	05	0x5	00000101	^E	ENQ
6	06	0x6	00000110	^F	ACK
7	07	0x7	00000111	^G	BEL
8	010	0x8	00001000	^H	BS
9	011	0x9	00001001	^I，tab	HT
10	012	0xa	00001010	^J	LF
11	013	0xb	00001011	^K	VT
12	014	0xc	00001100	^L	FF
13	015	0xd	00001101	^M	CR
14	016	0xe	00001110	^N	SO
15	017	0xf	00001111	^O	SI
16	020	0x10	00010000	^P	DLE
17	021	0x11	00010001	^Q	DC1
18	022	0x12	00010010	^R	DC2
19	023	0x13	00010011	^S	DC3
20	024	0x14	00010100	^T	DC4
21	025	0x15	00010101	^U	NAK
22	026	0x16	00010110	^V	SYN
23	027	0x17	00010111	^W	ETB
24	030	0x18	00011000	^X	CAN
25	031	0x19	00011001	^Y	EM
26	032	0x1a	00011010	^Z	SUB
27	033	0x1b	00011011	^[，Esc	ESC
28	034	0x1c	00011100	^\	FS

续表

十 进 制	八 进 制	十六进制	二 进 制	字 符	ASCII 名称
29	035	0x1d	00011101	^]	GS
30	036	0x1e	00011110	^^	RS
31	037	0x1f	00011111	^_	US
32	040	0x20	00100000	空格	SP
33	041	0x21	00100001	!	
34	042	0x22	00100010	"	
35	043	0x23	00100011	#	
36	044	0x24	00100100	$	
37	045	0x25	00100101	%	
38	046	0x26	00100110	&	
39	047	0x27	00100111	'	
40	050	0x28	00101000	(	
41	051	0x29	00101001	)	
42	052	0x2a	00101010	*	
43	053	0x2b	00101011	+	
44	054	0x2c	00101100	'	
45	055	0x2d	00101101	-	
46	056	0x2e	00101110	.	
47	057	0x2f	00101111	/	
48	060	0x30	00110000	0	
49	061	0x31	00110001	1	
50	062	0x32	00110010	2	
51	063	0x33	00110011	3	
52	064	0x34	00110100	4	
53	065	0x35	00110101	5	
54	066	0x36	00110110	6	
55	067	0x37	00110111	7	
56	070	0x38	00111000	8	
57	071	0x39	00111001	9	
58	072	0x3a	00111010	:	
59	073	0x3b	00111011	;	
60	074	0x3c	00111100	<	
61	075	0x3d	00111101	=	
62	076	0x3e	00111110	>	
63	077	0x3f	00111111	?	
64	0100	0x40	01000000	@	
65	0101	0x41	01000001	A	
66	0102	0x42	01000010	B	
67	0103	0x43	01000011	C	
68	0104	0x44	01000100	D	
69	0105	0x45	01000101	E	

续表

十　进　制	八　进　制	十六进制	二　进　制	字　　符	ASCII名称
70	0106	0x46	01000110	F	
71	0107	0x47	01000111	G	
72	0110	0x48	01001000	H	
73	0111	0x49	01001001	I	
74	0112	0x4a	01001010	J	
75	0113	0x4b	01001011	K	
76	0114	0x4c	01001100	L	
77	0115	0x4d	01001101	M	
78	0116	0x4e	01001110	N	
79	0117	0x4f	01001111	O	
80	0120	0x50	01010000	P	
81	0121	0x51	01010001	Q	
82	0122	0x52	01010010	R	
83	0123	0x53	01010011	S	
84	0124	0x54	01010100	T	
85	0125	0x55	01010101	U	
86	0126	0x56	01010110	V	
87	0127	0x57	01010111	W	
88	0130	0x58	01011000	X	
89	0131	0x59	01011001	Y	
90	0132	0x5a	01011010	Z	
91	0133	0x5b	01011011	[	
92	0134	0x5c	01011100	\	
93	0135	0x5d	01011101	]	
94	0136	0x5e	01011110	^	
95	0137	0x5f	01011111	-	
96	0140	0x60	01100000	'	
97	0141	0x61	01100001	a	
98	0142	0x62	01100010	b	
99	0143	0x63	01100011	c	
100	0144	0x64	01100100	d	
101	0145	0x65	01100101	e	
102	0146	0x66	01100110	f	
103	0147	0x67	01100111	g	
104	0150	0x68	01101000	h	
105	0151	0x69	01101001	i	
106	0152	0x6a	01101010	j	
107	0153	0x6b	01101011	k	
108	0154	0x6c	01101100	l	
109	0155	0x6d	01101101	m	
110	0156	0x6e	01101110	n	

续表

十 进 制	八 进 制	十六进制	二 进 制	字 符	ASCII 名称
111	0157	0x6f	01101111	o	
112	0160	0x70	01110000	p	
113	0161	0x71	01110001	q	
114	0162	0x72	01110010	r	
115	0163	0x73	01110011	s	
116	0164	0x74	01110100	t	
117	0165	0x75	01110101	u	
118	0166	0x76	01110110	v	
119	0167	0x77	01110111	w	
120	0170	0x78	01111000	x	
121	0171	0x79	01111001	y	
122	0172	0x7a	01111010	z	
123	0173	0x7b	01111011	{	
124	0174	0x7c	01111100	|	
125	0175	0x7d	01111101	}	
126	0176	0x7e	01111110	~	
127	0177	0x7f	01111111	Del	

附录 D　运算符优先级

运算符优先级决定了运算符用于值的顺序。C++运算符分为 18 个优先级组，如表 D.1 所示。第 1 组中的运算符的优先级最高，第 2 组中运算符的优先级次之，依此类推。如果两个运算符被用于同一个操作数，则首先应用优先级高的运算符。如果两个运算符的优先级相同，则 C++使用结合性规则来决定哪个运算符结合得更为紧密。同一组中运算符的优先级和结合性相同，不管是从左到右（表中 L-R）还是从右到左（表中 R-L）结合。从左到右的结合性意味着首先应用最左边的运算符，而从右到左的结合性则意味着首先应用最右边的运算符。

表 D.1　　C++运算符的优先级和结合性

运　算　符	结　合　性	含　义
优先级第 1 组		
::		作用域解析运算符
优先级第 2 组		
（表达式）		分组
()	L-R	函数调用
()		值构造，即 type(expr)
[ ]		数组下标
->		间接成员运算符
.		直接成员运算符
const_cast		专用的类型转换
dynamic_cast		专用的类型转换
reinterpret_cast		专用的类型转换
static_cast		专用的类型转换
typeid		类型标识
++		加 1 运算符，后缀
- -		减 1 运算符，后缀
优先级第 3 组（全是一元运算符）		
!	R-L	逻辑非
~		位非
+		一元加号（正号）
-		一元减号（负号）
++		加 1 运算符，前缀
- -		减 1 运算符，前缀
&		地址
*		解除引用（间接值）
()		类型转换，即(type)expr
sizeof		长度，以字节为单位
new		动态分配内存
new [ ]		动态分配数组
delete		动态释放内存
delete [ ]		动态释放数组

续表

运 算 符	结 合 性	含 义
优先级第 4 组		
.*	L-R	成员解除引用
->*		间接成员解除引用
优先级第 5 组（全是二元运算符）		
*	L-R	乘
/		除
^		模（余数）
优先级第 6 组（全是二元运算符）		
+	L-R	加
-		减
优先级第 7 组		
<<	L-R	左移
>>		右移
优先级第 8 组		
<	L-R	小于
<=		小于或等于
>=		大于或等于
>		大于
优先级第 9 组		
= =	L-R	等于
!=		不等于
优先级第 10 组（一元运算符）		
&	L-R	按位 AND
优先级第 11 组		
^	L-R	按位 XOF（异或）
优先级第 12 组		
|	L-R	按位 OR
优先级第 13 组		
&&	L-R	逻辑 AND
优先级第 14 组		
||	L-R	逻辑 OR
优先级第 15 组		
:?	R-L	条件
优先级第 16 组		
=	R-L	简单赋值
* =		乘并赋值
/=		除并赋值
%=		求模并赋值
+=		加并赋值
-=		减并赋值
&=		按位 AND 并赋值
^=		按位 XOR 并赋值
|=		按位 OR 并赋值
<<=		左移并赋值
>>=		右移并赋值

续表

运　算　符	结　合　性	含　义
优先级第 17 组		
throw	L-R	引发异常
优先级第 18 组		
,	L-R	将两个表达式合并成一个

有些符号（如*或&）被用作多个运算符。在这种情况下，一种形式是一元（一个操作数），另一种形式是二元（两个操作数），编译器将根据上下文来确定使用哪种形式。对于同一个符号可以两种方式使用的情况，表 D.1 将运算符标记为一元组或二元组。

下面介绍一些优先级和结合性的例子。

对于下面的例子，编译器必须决定先将 5 和 3 相加，还是先将 5 和 6 相乘：

```
3 + 5 * 6
```

*运算符的优先级比+运算符高，所以它被首先用于 5，因此表达式变成 3 +30，即 33。

对于下面的例子，编译器必须决定先将 120 除以 6，还是先将 6 和 5 相乘：

```
120 / 6 * 5
```

/和*的优先级相同，但这些运算符从左到右结合的。这意味着首先应用操作数（6）左侧的运算符，因此表达式变为 20*5，即 100。

对于下面的例子，编译器必须决定先对 str 递增还是先对 str 解除引用：

```
char * str = "Whoa";
char ch = *str++;
```

后缀++运算符的优先级比一元运算符*高，这意味着加号运算符将对 str 进行操作，而不是对*str 进行操作。也就是说，将指针加 1，使之指向下一个字符，而不是修改被指针指向的字符。不过，由于++是后缀形式，因此将在将*str 的值赋给 ch 后，再将指针加 1。因此，上述表达式将字符 W 赋给 ch，然后移动指针 str，使之指向字符 h。

下面是一个类似的例子：

```
char * str = "Whoa";
char ch = *++str;
```

前缀++运算符和一元运算符*的优先级相同，但它们是从右到左结合的。因此，str（不是*str）将被加 1。因为++运算符是前缀形式，所以首先将 str 加 1，然后将得到的指针执行解除引用的操作。因此，str 将指向字符 h，并将字符 h 赋给 ch。

注意，表 D.1 在“优先级”行中使用一元或二元来区分使用同一个符号的两个运算符，如一元地址运算符和二元按位 AND 运算符。

附录 B 列出了一些运算符的替代表示。