

## 02\_数据与存储

### 基本字符


本质上 C 语言程序就是使用 "基本字符" 按照语言的 "规定形式 ( C语言标准规定的语法规则 )" 写出来的字符序列。

基本字符包括：

1. 数字字符：0-9
2. 大小写拉丁字母：A-Za-z
3. 一些可打印的字符 ( ASCII码表 ) ，例如：标点符号、括号、运算符等，  
包括：~ ! % & \* ( ) \_ - + = { } [ ] : ; " ' < > , . ? /  
| \
4. 其他字符：空格符、制表符 \t 和换行符 \n 等字符，也被称为空白字符
5. 中文字符等其他字符只能用在注释和字符串

程序中的一段段基本字符构成程序里的各种名字、运算符、各种数据和其他字符。

ASCII 字符集

(低位字符)		(高位字符)							
		Hex	0	1	2	3	4	5	6
	0	NUL	DLE	Space	0	@	P	`	p
	1	SOH	DC1	!	1	A	Q	a	q
	2	STX	DC2	"	2	B	R	b	r
	3	ETX	DC3	#	3	C	S	c	s
	4	EOT	DC4	\$	4	D	T	d	t
	5	ENQ	NAK	%	5	E	U	e	u
	6	ACK	SYN	&	6	F	V	f	v
	7	Bell	ETB	'	7	G	W	g	w
	8	BS	CAN	(	8	H	X	h	x
	9	HT	EM	)	9	I	Y	i	y
	A	LF	SUB	*	:	J	Z	j	z
	B	VT	ESC	+	;	K	[	k	{
	C	FF	FS	,	<	L	\	l	
	D	CR	GS	-	=	M	]	m	}
	E	SO	RS	.	>	N	^	n	~
	F	SI	US	/	?	O	_	o	DEL

ASC II 码表（7 单位码）

ASC II 码是根据美国国家标准化协会（ANSI）制定，它给出了 128 个字符的 3 种不同进制的 ASC II 码值。  
其中

①传输控制类有 10 个 ASC II 字符分别代表 10 种用于数据传输控制的功能码，其助记符及功能如下：

- SOH: 01H, 报文标题的开始控制符;

ETX: 03H, 正文结束控制符;

EOT: 04H, 报文传送结束控制符;

ACK: 06H, 接收确认控制符;

DLE: 10H, 数据链转移控制符, 在本字符后有限个非控制字符, 可使控制类字符个数及控制功能得到扩充;

SYN: 16H, 同步通讯控制符。通常, SYN 同步字符后面总是跟随报头标题信息或数据信息。
- STX: 02H, 正文开始控制符;

ETB: 17H, 分组结束控制符;

ENQ: 05H, 询问查询控制符;

NAK: 15H, 接收否认控制符;

②设备控制类有 4 个控制符, 即

- DC1-通讯中表示为开始传送 Xon (CTRL+Q 键);

DC3-通讯中表示为停止传送 Xoff (CTRL+S 键);

DC2、DC4-与 DC1 类似常由厂商规定。

③格式形成类有 6 个字符, 用以控制打印或显示字符的位置。

- BS: 退格控制符;

HT: 横向制表符;

VT: 纵向制表符;

LF: 换行符;

ASCII 码

ASCII 字符代码表 一																									
高四位		ASCII非打印控制字符												ASCII 打印字符											
低四位	0000						0001						0010		0011		0100		0101		0110		0111		
	0						1						2		3		4		5		6		7		
	+进制	字符	ctrl	代码	字符解释	+进制	字符	ctrl	代码	字符解释	+进制	字符	+进制	字符	+进制	字符	+进制	字符	+进制	字符	+进制	字符	ctrl		
0000	0	0	BLANK	^@	NUL	空	16	▶	^P	DLE	数据链路转意	32		48	0	64	@	80	P	96	`	112	p		
0001	1	1	☺	^A	SOH	标题开始	17	◀	^Q	DC1	设备控制 1	33	!	49	1	65	A	81	Q	97	a	113	q		
0010	2	2	☹	^B	STX	正文开始	18	↕	^R	DC2	设备控制 2	34	"	50	2	66	B	82	R	98	b	114	r		
0011	3	3	♥	^C	ETX	正文结束	19	!!	^S	DC3	设备控制 3	35	#	51	3	67	C	83	S	99	c	115	s		
0100	4	4	♦	^D	EOT	传输结束	20	¶	^T	DC4	设备控制 4	36	\$	52	4	68	D	84	T	100	d	116	t		
0101	5	5	♣	^E	ENQ	查询	21	♢	^U	NAK	反确认	37	%	53	5	69	E	85	U	101	e	117	u		
0110	6	6	♠	^F	ACK	确认	22	■	^V	SYN	同步空闲	38	&	54	6	70	F	86	V	102	f	118	v		
0111	7	7	●	^G	EEL	脱转	23	↑	^W	ETB	传输块结束	39	'	55	7	71	G	87	W	103	g	119	w		
1000	8	8	□	^H	BS	退格	24	↑	^X	CAN	取消	40	(	56	8	72	H	88	X	104	h	120	x		
1001	9	9	○	^I	TAB	水平制表符	25	↓	^Y	EM	媒体结束	41	)	57	9	73	I	89	Y	105	i	121	y		
1010	A	10	☐	^J	LF	换行/新行	26	→	^Z	SUB	替换	42	*	58	:	74	J	90	Z	106	j	122	z		
1011	B	11	♂	^K	VT	垂直制表符	27	←	^[	ESC	转意	43	+	59	;	75	K	91	[	107	k	123	{		
1100	C	12	♀	^L	FF	换页/新页	28	└	^\ F5	文件分隔符	44	,	60	<	76	L	92	\	108	l	124				
1101	D	13	♫	^M	CR	回车	29	↔	^] G5	组分隔符	45	-	61	=	77	M	93	]	109	m	125	}			
1110	E	14	♪	^N	SO	移出	30	▲	^6	RS	记录分隔符	46	.	62	>	78	N	94	^	110	n	126	~		
1111	F	15	☒	^O	SI	移入	31	▼	^~	US	单元分隔符	47	/	63	?	79	O	95	_	111	o	127	Δ Back space		

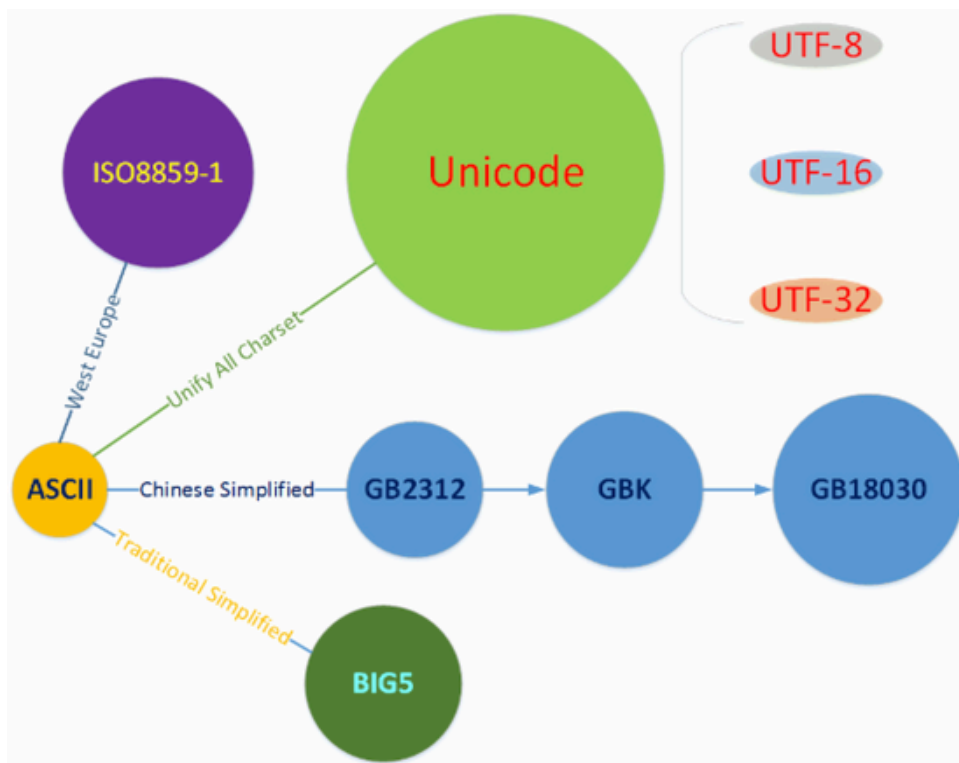
注：表中的ASCII字符可以用:ALT + “小键盘上的数字键” 输入

注：表中的ASCII字符可以用:ALT + “小键盘上的数字键”输入

## ASCII 码

!"#\$%&'()\*+,-./0123456789:;<=>?  
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^\_`abcdefghijklmnopqrstuvwxyz{|}~

转义字符	意义	ASCII码值（十进制）
\a	响铃(BEL)	007
\b	退格(BS)，将当前位置移到前一个列	008
\f	换页(FF)，将当前位置移到下页开头	012
\n	换行(LF)，将当前位置移到下一行开头	010
\r	回车(CR)，将当前位置移到本行开头	013
\t	水平制表(HT)（跳到下一个TAB位置）	009
\v	垂直制表(VT)	011
\\	代表一个反斜线字符\"	092
\'	代表一个单引号（撇号）字符	039
\"	代表一个双引号字符	034
\?	代表一个问号	063
\0	空字符(NUL)	000
\ddd	1到3位八进制数所代表的任意字符	三位八进制
\xhh	十六进制所代表的任意字符	十六进制



## 💡 字面量

在计算机中，字面量（literal）是用于表达源代码中一个固定值的表示法（notation）。

几乎所有的编程语言都具有对基本值的字面量表示，例如：整数、浮点数以及字符串。在很多语言中也对布尔、字符、数组以及对象的值也支持字面量表示。

字面量的种类：

1. 整型，整型数值，默认是 int 类型。例如：1,2,3,4,5
2. 浮点型，浮点数类型，默认是double类型。例如：3.14
3. 字符串型，包含在双引号中的字符序列，被称为字符串。例如："Hello World!"
4. 字符型，包含在单引号中的基本字符，被称为字符。例如：'a'

对字面量的理解：

这些基本字符，都是人类可以理解的内容。而计算机是以二进制识别和存储数据的。

在计算机中，-1 存储的值为 0xffffffff。在C语言中，int a=-1和 int a=0xffffffff 是等价的，但是0xffffffff 对于人类而言不是很好理解，在编译时，编译器先将-1替换成0xffffffff在进行运算。字符串也是同样的道理，“Hello” 人类很容易理解，但是换成二进制则不方便人类理解值的含义。

## 💙 字面常量

字面常量 (literal constant)，所谓的字面，就是我们在程序中直接以值的形式来操作、表现。

所谓的常量，是指在程序执行过程中其值不可以改变。

与字面量是一回事，只是特别强调其不可改变性。

# 标识符和关键字

## 关键字

关键字就是被C语言赋予了特殊的含义，用做特定功能的字符序列。

ANSI C (C90) 标准，一共提供 32 个关键字。主要有数据类型关键字 (12)，控制语句关键字 (12)，存储类型关键字 (5)、其他关键字 (3)

1. 数据类型关键字：char, enum, double, long, float, int, short, signed, struct, unsigned, union, void
2. 控制语句关键字：break, case, continue, default, do, else, for, goto, if, return, switch, while
3. 存储类型关键字：auto, extern, register, static, const
4. 其他关键字：sizeof, typedef, volatile

C99 标准增加了5个关键字：inline, restrict, \_Bool, \_Complex, \_Imaginary

C11 标准增加了7个关键字：\_Alignas, \_Alignof, \_Static\_assert, Noreturn, \_Thread\_local, \_Generic

## [C 关键字](#)

# 标识符

在程序中，用来标识对象和实体的符号称为标识符 ( identifier )。

在计算机编程语言中，标识符是用户编程时使用的名字。用于给变量、常量、函数、语句块等命名，以建立名称和使用之间的关系。标识符通常由字母和数字以及下划线构成。

在C语言程序中，标识符用字母和数字以及下划线的连续序列表示，其中不能以出现空白字符，要求第一个字符必须是字母（下划线也被作为字母看待），对于C99后的标准中会使用"\_"作为关键字的开头。为了避免与标准中的关键字混淆。所以尽量避免是用"\_"作为标识符首字母。

标识符的规范：

1. 字母、数字以及下划线
2. 数字不可以作为首字符
3. 严格区分大小写
4. 不允许使用系统关键字

关于标识符的长度，C99和C11允许使用更长 的标识符名，但是编译器仅识别前63个字符。（会忽略超出的字符）

关于标识符的命名建议：

1. 见名知意，可以提高程序的阅读型。比如：sex、age、year、month、total、name、title 等等
2. 不要仅通过大小写来区分标识符。例如：name、Name 容易混淆
3. 对于常量（符号常量、只读常量、枚举常量）使用大写字母命名，合成词使用下划线分隔单词

4. 避免使用下划线作为首字符（因为C语言标准中使用一些下划线开头的标识符）

## 语句和复合语句

### 语句

C 语言程序中描述计算过程的基本单位是语句（statement），一个语句是由分号结束的（符合语法规则的）一段字符。语句表示在程序执行过程中要做的一些操作，这个被称为语句的语义。

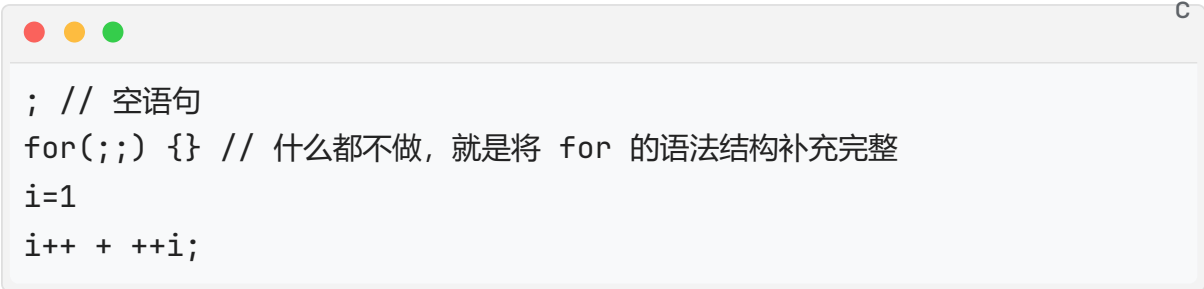
语句必须符合C语言的语法（C语言标准规定的语法规则），我们学习编程语言，本质上就是学习的是语义以及语法，使用语义描述程序流程（思路），使用语法实现程序设计（编程）。

例如：

语义：printf 函数的用于向终端输出格式字符串。

语法：printf 声明在 stdio.h（标准输入输出头文件）中，printf(格式字符串，参数1，参数2，.....)，格式字符串中的占位符由 %d、%c、%p、%s、%% 等等

**空语句**是只有一个分号而没有其他字符。在执行时什么都不做，也可以作为填充，有时候需要用它将程序的语法结构补充完整。



```
; // 空语句
for(;;) {} // 什么都不做，就是将 for 的语法结构补充完整
i=1
i++ + ++i;
```

### 复合语句

复杂的计算过程需要通过多条基本语句按照特定的顺序执行才能实现。为了描述复杂的计算过程，编程语言提供了控制结构（顺序结构、选择结构、循环结构），以实现对语句的执行过程（流程）控制。描述计算流程的最基本结构是复合结构（也被称为复合语句）。复合结构的形式以一对花括号作为定界符，



在括号内可以写任意多条语句。当程序执行时，复合结构中的各条语句按照书写顺序依次执行。（复合结构的语义）。

空复合结构是仅拥有一对花括号，不包含任何语句。执行时什么都不做，立即结束，相当于空语句。

## 数据编码

高级语言把程序能够处理的基本数据分成一些集合，属于同一集合的数据具有同样性质：采用统一的书写形式，在具体实现中采用同样的编码方式，对它们能做同样的操作。语言中每一个具有这样性质的数据集称为一个类型（type）。程序里能写的、执行中能处理的每个基本数据都属于某个基本类型。在具体语言系统里，各种基本类型都有规定的表示（编码）方式和表示范围。基本类型有各自的名字，称为类型名。基本类型名由一个或几个标识符（它们都是关键字）构成。

## 二进制计数法

在人类的历史长河中，已经养成了使用十进制进位计数法的习惯（可能是人类只有10根手指）。表示字符：0、1、2、3、4、5、6、7、8、9。在此基础上采用“逢10进1”的进位原则用数码序列表示一般的自然数。数字序列上的每一位都可以取这十个数码。在此基础上再引进负数、小数等数的概念和相应的记法形式。

## 十进制

进位计数法中，进位记数法中，每一个数位上的数码都有相应的位权。在十进制数的小数点左侧，个位数的位权是  $10^0$ ，十位数的位权是  $10^1$ ，百位数的位权是  $10^2$ ，依次向左类推。而在十进制数的小数点右侧，十分位的位权是  $10^{-1}$ ，百分位的位权是  $10^{-2}$ ，依次向右类推。进位记数法中把数字的左侧数位称为高位，右侧数位称为低位，任何进位制数的值都可以表示成从高到低（从左向右）按位权展开的和式，例如十进制数“234.15”的值就等于  $2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0 + 1 \times 10^{-1} + 5 \times 10^{-2}$ 。


类似地，可以采用其它进位记数法，例如二进制、八进制和十六进制的记数规则是：

二进制数使用 0 和 1 这两个数码，进位原则是“逢2进1”

八进制数使用的数码是 0、1、2、3、4、5、6、7，进位原则是 "逢8进1"

十六进制数使用的数码是 0、1、2、3、4、5、6、7、8、9、A、B、C、D、E、F 这十六个数码，进位原则是 "逢16进1"

## 二进制



```
0000 0000
0000 0001
0000 0010
0000 0011
0000 0100
0000 0101
0000 0110
0000 0111
0000 1000
0000 1001
0000 1010
0000 1011
0000 1100
0000 1101
0000 1110
0000 1111
0001 0000
0001 0001
0001 0010
0001 0011
0001 0100
0001 0101
0001 0110
0001 0111
0001 1000
0001 1001
0001 1010
0001 1011
0001 1100
0001 1101
```

```
0001 1110
0001 1111
```

## 八进制



## 十六进制



## 计算机的存储和计算

计算机是用电子器件构造起来的电子设备，数据的存储和计算采用二进制的表示形式。在硬件层面，电路中使用"低电位"和"高电位"两种状态，它们可以分别表示一个二进制数位。一连串"低电平/高电平"脉冲也可以表示一个多数位的二进制数。而且通过精巧的电路结构，可以实现二进制数的各种运算。在软件层面，使用抽象的0和1表示两种状态。目前不必关心计算机的底层物理构成，只需从抽象的二进制来理解计算机内部的数据即可。

一个二进制数位也称为 1 个位 (bit，也称为"比特")，只能表示0和1，需要使用更多的二进制数位才能表示丰富的数据。在计算机中，通常把连续的8个二进制数位作为一组看待和处理。因此，就把8个二进制位称为1个字节 (byte)。目前大多数计算机存储器是以字节为单位组织起来的。字节是计算机领域最重要的一种数据单位，计算机存储器容量大小、计算机处理的数据量的大小等，都是用其中包含的字节数来度量。常用的数据单位如下：

1. 1KB = 1024B
2. 1MB = 1024KB
3. 1GB = 1024MB
4. 1T = 1024GB

## 定点整数的表示和整数类型

计算机中各种数据都是用二进制存储，因此需要以合适的格式来存储来存储不同的数据。自然数是正整数（或者说不带正负号，是无符号整数），因此可以直接以二进制数表示。例如，二进制数 10101，存储在一个字节（计算机中总是以字节为单位存储数据）中时为 00010101（多余的 高位补 0）。容易理解，用一个字节直接存储二进制无符号整数时，能表示的整数最小值是 0，最大值是 +255，用两个字节直接存储时，能表示的整数范围是[0, 65535]。

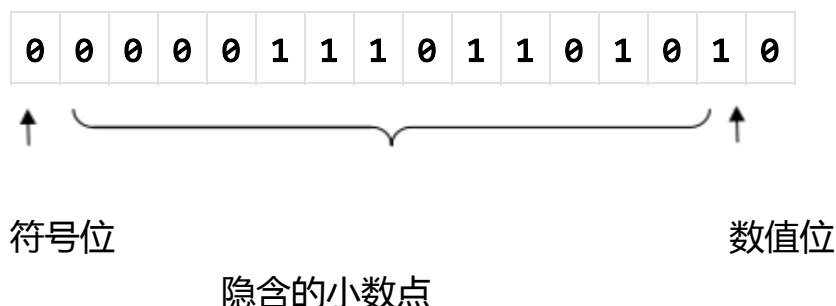
对于带正负号的整数，如何表示其正负号？一种简单的方法是把正负号和绝对值分开来看待：把正负号视为  $(-1)^s$ ， $s$  称为符号（sign），可以取 0 和 1 以表示正号和负号。把  $s$  存储在最高位，把绝对值存储在后续数位，中间的多余数位补 0。例如：

二进制正整数  $+10101 = (-1)^0 \times 10101$ ，存储 在一个字节中为 00010101（最高位的 0 表示正号，后面写绝对值 10101，中间多余的两个数位补 0）。

二进制负整数  $-10101 = (-1)^1 \times 10101$ ，存储在一个字节中为  $10010101$ （最高位的 1 表示正号）。

用一个字节以原码表示法存储带正负号的整数时，能表示的整数最大值是 +127，最小值是 -127。而用一个字节以补码表示法存储带正负号的整数时，表示范围是 -128 ~ +127。如果想表示更大范围的整数，就需要使用连续的多个字节。

例如：计算机系统使用以 2 个字节存储整数，则十进制整数 +1898 的二进制数真值为 +11101101010，在机内以二进制原码定点数表示时，存储情况如下图所示：



定点整数：总是默认小数点固定地隐含在所有数位的最后。

## 原码

### 💙 原码

整数 $x$ 的原码其数符位 $0$ 表示正， $1$ 表示负；其数值部分就是 $x$ 绝对值的二进制表示

例如：

$[+1]$ 原码= $00000001$ ； $[-1]$ 原码= $10000001$

$[+127]$ 原码= $01111111$ ； $[-127]$ 原码= $11111111$

关于八位二进制，由于第一位是符号位，所以从负到正为  
 $11111111 \sim 01111111$

故原码范围为 $-127$ 到 $127$ ，关于为什么 $01111111$ 表示 $127$ ，我们知道八位， $2^7=10000000$ ，表示的是 $128$ ，注意几次方就有几个 $0$

因此对于 $01111111$ ，加一个就变成了 $10000000$ （ $128$ ），故 $01111111$ 表示 $127$ ；

因此原码的取值范围为 $-127 \sim 127$

原码中有正 $0$ 与负 $0$ ， $[+0]$ 原码= $00000000$ ； $[-0]$ 原码= $10000000$

## 反码

### 💙 反码

整数 $x$ 的反码对于正数，与原码相同；对于负数，符号位为 $1$ ，数值位为 $x$ 的绝对值取反

例如：

$[+1]$ 反码= $00000001$ ； $[-1]$ 反码= $11111110$

$[+127]$ 反码= $01111111$ ； $[-127]$ 反码= $10000000$

反码中 $0$ 也有正 $0$ 和负 $0$ ， $[+0]$ 反码= $00000000$ ； $[-0]$ 反码= $11111111$

因此反码的取值范围也是 $-127 \sim 127$

## 补码

### ❤ 补码

整数x的补码对于正数与反码、原码相同；对于负数，数符位为1，其数值位x的绝对值取反最右加1，也就是反码加一

例如：

[+1]补码=00000001；[-1]补码=11111111

[+127]补码=01111111；[-127]补码=10000001

注意的是，0的补码唯一

即[+0]补码=[-0]补码=00000000

我此刻可以发现，对比原码和反码，我们发现补码中少了一个0的编码，就是10000000（在原码和反码中表示-0）这个编码

因此在补码中，将这个多出来10000000进行扩充，用它来表示-128

因此补码的取值范围位-128~127

C 语言提供了以二进制定点整数的形式编码表示的多个整数类型，以满足实际应用中的不同需要。

- 普通整数类型（简称为整数类型或整型）的类型名是 `int`。
- 短整数类型（简称为短整型），类型名为 `short int`，可简写为 `short`。
- 长整数类型（简称为长整型），类型名为 `long int`，可简写为 `long`。
- 长长整数类型（简称为长长整型），类型名为 `long long int`，可简写为 `long long`。

整数类型的差别在于它们可能采用不同的二进制编码位数。C 语言没有规定各种整数的二进制编码位数。没有规定各种整数类型的表示范围，只对它们的相对关系有如下规定：

- `int` 类型的表示范围大于或等于 `short`

- long 类型的表示范围大于或等于 int
- long long 类型的表示范围大于 long

### 64位程序上C语言整数数据类型的典型取值范围

C 数据类型	最小值	最大值
char	-128	127
unsigned char	0	255
short	-32 768	32 767
unsigned short	0	65 535
int	-2 147 483 648	2 147 483 647
unsigned int	0	4 294 967 295
long	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
unsigned long	0	18 446 744 073 709 551 615

各个整数类型的表示范围取决于操作系统和编译器。C 语言的整型数据类型的保证的取值范围。以下是C语言标准要求这些数据类型必须至少具有这样的取值范围。

C 数据类型	最小值	最大值
char	-127	127
unsigned char	0	255
short	-32 767	32 767
unsigned short	0	65 535
int	-32 767	32 767
unsigned int	0	65 535
long	-2 147 483 647	2 147 483 647
unsigned long	0	4 294 967 295

```
#include <stdio.h>
#include <limits.h>
#include <stdint.h>
int main() {
    // 整数的数据类型
    // short int 短整型
    // int 整型
    // long int 长整型
    // long long int 长长整型
}
```

```

// 1. C 只是提供类型名称, 没有提供具体的长度
// 类型的具体长度与操作系统和编译器有关
printf("%d\n", sizeof(int)); // 4 字节 * 8 = 32 位
printf("%d\n", sizeof(short int)); // 2 字节 * 8 = 16 位
printf("%d\n", sizeof(long int)); // 4 字节 * 8 = 32 位
printf("%d\n", sizeof(long long int)); // 8 字节 * 8 = 64 位

printf("int 类型最小值: %d\n", INT_MIN);
printf("int 类型最大值: %d\n", INT_MAX);
printf("short 类型最小值: %d\n", SHRT_MIN);
printf("short 类型最大值: %d\n", SHRT_MAX);
printf("long 类型最小值: %d\n", LONG_MIN);
printf("long 类型最大值: %d\n", LONG_MAX);

printf("%d\n", sizeof(int8_t)); // 1 字节 * 8 = 8 位
printf("%d\n", sizeof(int16_t)); // 2 字节 * 8 = 16 位
printf("%d\n", sizeof(int32_t)); // 4 字节 * 8 = 32 位
printf("%d\n", sizeof(int64_t)); // 8 字节 * 8 = 64 位
return 0;
}

```

## 浮点数的表示和浮点数类型

### LaTeX

LaTeX (【读音：雷泰克斯】)，是一种基于TEX的排版系统，由美国计算机科学家莱斯利·兰伯特在20世纪80年代初期开发，利用这种格式系统的处理，即使用户没有排版和程序设计的知识也可以充分发挥由TEX所提供的强大功能，不必一一亲自去设计或校对，能在几天，甚至几小时内生成很多具有书籍质量的印刷品。对于生成复杂表格和数学公式，这一点表现得尤为突出。因此它非常适用于生成高印刷质量的科技和数学、物理文档。这个系统同样适用于生成从简单的信件到完整书籍的所有其他种类的文档。

## 二进制小数



理解浮点数的第一步是考虑含有小数值的二进制数字。首先，让我们来看看更熟悉的十进制表示法。十进制表示法使用如下形式的表示：

$$d_n d_{n-1} \cdots d_2 d_1 . d_{-1} d_{-2} \cdots d_{-n-1} d_{-n}$$

其中每个十进制数 $d_i$ 的取值范围是 $0 \sim 9$ 。这个表达描述的数值 $d$ 定义如下：

$$d = \sum_{i=-n}^m 10^i \times d_i$$

### 💙 公式解读

从 $i = -n$  到  $i=m$  的所有  $10^i \times d_i$  的和， $d_i$ 是每一项的系数， $10^i$  是每一项的基数， $i$  是从  $-n$  到  $m$  的索引。

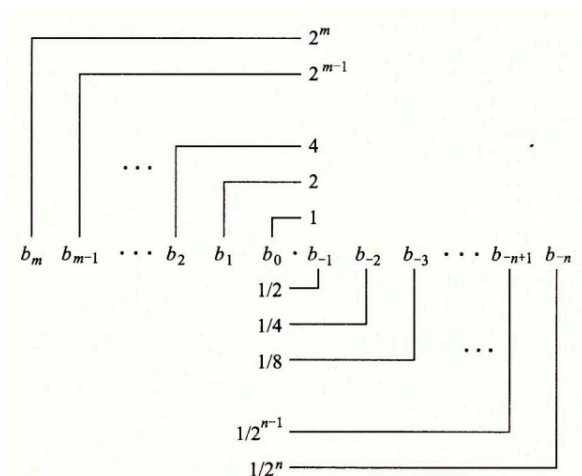
数字权的定义与十进制小数点符号('.')相关，这意味着小数点左边的数字的权是10的正幂，得到整数值，而小数点右边的数字的权是10的负幂，得到小数值。例如： $12.34_{10}$  的表示数字：

$$1 \times 10^1 + 2 \times 10^0 + 3 \times 10^{-1} + 4 \times 10^{-2} = 12 \frac{34}{100}$$

类似，考虑一个形如  $b_n b_{n-1} \cdots b_2 b_1 . b_{-1} b_{-2} \cdots b_{-n-1} b_{-n}$  的表示法，其中每个二进制数字，或者称为位， $b_i$  的取值范围是0和1，如下面的表达式：

$$b = \sum_{i=-n}^m 2^i \times b_i$$

如图：



符号 ‘.’ 现在变为了二进制的点，点左边的位的权是 2 的正幂，点右边的位的权是 2 的负幂。例如： $101.11_2$  表示数字：

$$1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} = 4 + 0 + 1 + \frac{1}{2} + \frac{1}{4} = 5\frac{3}{4}$$

从等式  $b = \sum_{i=-n}^m 2^i \times b_i$  可以看出，二进制小数点向左移动一位相当于这个数被 2 除。例如： $101.11_2$  表示数  $5\frac{3}{4}$ ，而  $10.111_2$  表示数  $2 + 0 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} = 2\frac{7}{8}$ 。类似，二进制小数点向右移动一位相当于将该数乘以 2。例如： $1011.1_2$  表示数  $8 + 0 + 2 + 1 + \frac{1}{8} = 11\frac{1}{8}$ 。

假定我们仅考虑有限长度的编码，那么十进制表示法不能准确地表达像  $\frac{1}{3}$ 、 $\frac{5}{7}$  这样的数。类似，小数的二进制表示法只能表示那些能够被写成  $x \times 2^y$  的数。其他的值只能够被近似地表示。例如，数字  $\frac{1}{5}$  可以用十进制小数 0.20 精确表示。不过，我们并不能把它准确地表示为一个二进制小数，我们只能近似地表示它，增加二进制表示的长度可以提高表示的精度：

表示	值	十进制
$0.0_2$	$\frac{0}{2}$	$0.0_{10}$
$0.01_2$	$\frac{1}{4}$	$0.25_{10}$
$0.010_2$	$\frac{2}{8}$	$0.25_{10}$
$0.011_2$	$\frac{3}{16}$	$0.1875_{10}$
$0.0110_2$	$\frac{6}{32}$	$0.1875_{10}$

表示	值	十进制
$0.01101_2$	$\frac{13}{64}$	$0.203125_{10}$
$0.011010_2$	$\frac{26}{128}$	$0.203125_{10}$
$0.0110011_2$	$\frac{51}{256}$	$0.19921875_{10}$

### ♥ 精度问题

对于一些特殊的数值 $\frac{1}{3}$ 、 $\frac{5}{7}$ ，在有限的长度内只能近似的表示它。增加长度可以提高表示的精度。

## 4.3.2、IEEE 浮点表示

### ♥ IEEE

电气和电子工程师协会 (IEEE, 读作 "eye-triple--ee" 【读音：爱崔宝呢呢】) 是一个包括所有电子和计算机技术的专业团体。它出版刊物，举办会议，并且建立委员会来定义标准，内容涉及从电力传输到软件工程。另一个 IEEE 标准的例子是无线网络的 802.11 标准。-----  
摘自《深入理解计算机系统 - 2.4 浮点数》

IEEE 浮点标准用  $V = (-1)^s \times M \times 2^E$  的形式来表示一个数：

$$V = (-1)^s \times M \times 2^E$$

- 符号 (sign) 决定这个数是负数 ( $s=1$ ) 还是整数 ( $s=0$ )，而对于数值 0 的符号位解释作为特殊情况处理。
- 尾数 (significand)  $M$  是一个二进制小数，它的范围是  $1 \sim 2 - \epsilon$  (表示 1 至  $2 - \epsilon$  之间的数，其中  $\epsilon$  可以被视为无限接近于零的数，但不等于零) 或者  $0 \sim 1 - \epsilon$
- 阶码 (exponent)  $E$  的作用是对浮点数加权，这个权重是 2 的  $E$  次幂 (可能是负数)。

将浮点数的位表示划分为三个阶段，分别对这些值进行编码：

- 一个单独的符号位s直接编码为**符号**s。 --- **符号位**
- k 位的阶码字段  $exp = e_{k-1} \cdots e_1 e_0$  编码**阶码**E。 --- **(指数位 - 偏置位)**
- n 位小数字段  $frac = f_{n-1} \cdots f_1 f_0$  编码**尾数**M, 但是编码出来的值也依赖于阶码字段的值是否等于 0。 --- **(1 + 小数位)**

下面给出了将这三个字段装进字中两种最常见的格式。在单精度浮点数格式 (C语言中的float), s、exp、frac 字段分别为s=1位、k=8位、n=23位, 得到一个32位的表示。在双精度浮点格式 (C语言中double) 中, s、exp、frac 字段分别为s=1位、k=11位、n=52位, 得到一个64位的表示。

s(31)	exp(30:23)	frac(22:0)
-------	------------	------------

s(63)	exp(62:52)	frac(51:0)
-------	------------	------------

给定位 (给定位是指在有限的位数下如何表示浮点数) 表示, 根据 **exp (阶码、指数位)** 的值, 被编码的值可以分成三种不同的情况 (最后一种情况有两个变种)。以下是单精度浮点数格式的情况。

### 1. 规格化的

s	$\neq 0 \& \neq 255$	f
---	----------------------	---

### 2. 非规格化的

s	0 0 0 0 0 0 0 0	f
---	-----------------	---

### 3. 第三种情况的两个变种

#### 1. 无穷大

s	1 1 1 1 1 1 1 1 1 0
---	---

#### 2. NaN

s	1 1 1 1 1 1 1 1 1	$\neq 0$
---	-------------------	----------

## ⚠ 单精度浮点数的分类

单精度浮点数值分类（阶码的值决定了这个数是规格化的、非规格化的或特殊值）

## 💙 情况1：规格化的值

当  $\text{exp}$  的位模式既不全为 0（数值 0），也不全为 1（单精度浮点数值为 255），双精度数值为 2047 时，都属于这种情况。在这种情况下，阶码字段被解释为以偏置形式表示的有符号整数。也就是说，阶码的值是  $E = e - \text{Bias}$ ，其中  $e$  是无符号数，其位表示为  $e_{k-1} \cdots e_1 e_0$ ，而  $\text{Bias}$  一个等于  $2^{k-1} - 1$ （单精度是 127，双精度是 1023）的偏置值。由此产生指数的取值范围，对于单精度是  $-126 \sim +127$ ，而对于双精度是  $-1022 \sim +1023$ 。

小数字段  $\text{frac}$  被解释为描述小数值  $f$ ，其中  $0 \leq f < 1$ ，其二进制表示为  $0.f_{n-1}f_{n-2} \cdots f_0$ ，也就是二进制小数点在最高有效位的左边。尾数定义为  $M = 1 + f$ 。有时，这种方式也叫做隐含的以 1 开头的（implied leading 1）表示，因为我们可以把  $M$  看成一个二进制表达式为  $1.f_{n-1}f_{n-2} \cdots f_0$  的数字。既然我们总是能够调整阶码  $E$ ，使得尾数  $M$  在范围  $1 \leq M < 2$  之中（假设没有溢出），那么这种表示方法是一种轻松获得一个额外精度位的技巧。既然第一位总是等于 1，那么我们就不需要显式地表示它。

## 💡 为什么采用偏置位？

### 1. 简化计算

- 偏置位的存在使得指数成为无符号数
- 有符号数的溢出判断比无符号数更为复杂

### 2. 表示小于 1 的数字

- 如果不加偏置将指数定义为无符号数，是没有办法表示小于 1 的数字

计算方式： $2^{k-1} - 1$  其中  $k$  是指数位数

## 💙 情况2：非规格化的值

当阶码域为全0时，所表示的数是非规格化形式。在这种情况下，阶码值是  $E = e - Bias$ ，而尾数的值是  $M = f$ ，也就是小数字段的值，不包含隐含的开头的1。非规格化数有两个用途。首先，它们提供了一种表示数值0的方法，因为使用规格化数，我们必须总是使  $M \geq 1$ ，因此我们就不能表示0。实际上，+0.0 的浮点表示的位模式为全0：符号位是0，阶码字段全为0（表明是一个非规格化值），而小数域也全为0，这就得到  $M=f=0$ 。令人奇怪的是，当符号位为1，而其他域全为0时，我们得到值 -0.0。根据IEEE的浮点格式，值 +0.0 和 -0.0 在某些方面被认为是不同的，而在其他方面是相同的。非规格化数的另外一个功能是表示那些非常接近于0.0的数。它们提供了一种属性，称为逐渐溢出(gradual underflow)，其中，可能的数值分布均匀地接近于0.0。



### 对于非规格化的值为什么要这样设置偏置值

使阶码的值为  $1 - Bias$  而不是简单的  $-Bias$ ，提供了一种从非规格化值平滑转换到规格化值的方法。



### 情况3：特殊值

最后一类数值是当指阶码全为1的时候出现的。当小数域全为0时，得到的值表示无穷，当  $s=0$  时是  $+\infty$ ，或者当  $s=1$  时是  $-\infty$ 。当我们把两个非常大的数相乘，或者除以零时，无穷能够表示溢出的结果。当小数域为非零时，结果值被称为"NaN"，即"不是一个数(Not a Number)"的缩写。一些运算的结果不能是实数或无穷，就会返回这样的NaN值，比如当计算  $\sqrt{-1}$  或  $\infty - \infty$  时。在某些应用中，表示未初始化的数据时，它们也很有用处。

描述	位表示	指数			小数		值		
		$e$	$E$	$2^E$	$f$	$M$	$2^E \times M$	$V$	十进制
0	0 0000 000	0	-6	$\frac{1}{64}$	$\frac{0}{8}$	$\frac{0}{8}$	$\frac{0}{512}$	0	0.0
最小的非规格化数	0 0000 001	0	-6	$\frac{1}{64}$	$\frac{1}{8}$	$\frac{1}{8}$	$\frac{1}{512}$	$\frac{1}{512}$	0.001953
	0 0000 010	0	-6	$\frac{1}{64}$	$\frac{2}{8}$	$\frac{2}{8}$	$\frac{2}{512}$	$\frac{1}{256}$	0.003906
	0 0000 011	0	-6	$\frac{1}{64}$	$\frac{3}{8}$	$\frac{3}{8}$	$\frac{3}{512}$	$\frac{3}{512}$	0.005859
	$\vdots$								
最大的非规格化数	0 0000 111	0	-6	$\frac{1}{64}$	$\frac{7}{8}$	$\frac{7}{8}$	$\frac{7}{512}$	$\frac{7}{512}$	0.013672
最小的规格化数	0 0001 000	1	-6	$\frac{1}{64}$	$\frac{0}{8}$	$\frac{8}{8}$	$\frac{8}{512}$	$\frac{1}{64}$	0.015625
	0 0001 001	1	-6	$\frac{1}{64}$	$\frac{1}{8}$	$\frac{9}{8}$	$\frac{9}{512}$	$\frac{9}{512}$	0.017578
	$\vdots$								
	0 0110 110	6	-1	$\frac{1}{2}$	$\frac{6}{8}$	$\frac{14}{8}$	$\frac{14}{16}$	$\frac{7}{8}$	0.875
	0 0110 111	6	-1	$\frac{1}{2}$	$\frac{7}{8}$	$\frac{15}{8}$	$\frac{15}{16}$	$\frac{15}{16}$	0.9375
	0 0111 000	7	0	1	$\frac{0}{8}$	$\frac{8}{8}$	$\frac{8}{8}$	1	1.0
	0 0111 001	7	0	1	$\frac{1}{8}$	$\frac{9}{8}$	$\frac{9}{8}$	$\frac{9}{8}$	1.125
	0 0111 010	7	0	1	$\frac{2}{8}$	$\frac{10}{8}$	$\frac{10}{8}$	$\frac{5}{4}$	1.25
	$\vdots$								
	0 1110 110	14	7	128	$\frac{6}{8}$	$\frac{14}{8}$	$\frac{1792}{8}$	224	224.0
最大的规格化数	0 1110 111	14	7	128	$\frac{7}{8}$	$\frac{15}{8}$	$\frac{1920}{8}$	240	240.0
无穷大	0 1111 000	—	—	—	—	—	—	$\infty$	—

图 2-35 8 位浮点格式的非负值示例( $k=4$  的阶码位的和  $n=3$  的小数位。偏置量是 7)



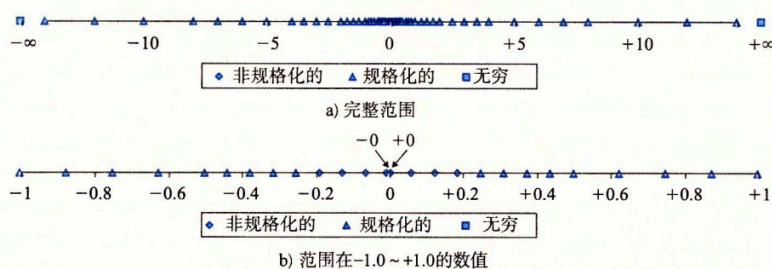
## 非规格化数存在的原因？

### 1. 表示接近0的数

- 规格化浮点数能表示的最小量级  $1 \sim 2^{-E}$

### 2. 平滑过渡

- 可以平滑地过渡到规格化浮点数，使得这两大类浮点数的衔接不至于“突兀”



## 示例 5.5 的浮点数存储格式

### IEEE 754 单精度浮点数格式

- 符号位 (Sign bit) : 1位，表示正负号。0表示正数，1表示负数。

- 指数 (Exponent) : 8位, 用于表示指数。
- 尾数 (Fraction/Mantissa) : 23位, 表示小数部分。

## 步骤 :

### 1. 符号位 :

- 5.5是正数, 所以符号位为 0 。

### 2. 转换成二进制 :

- 整数部分 : 5的二进制是 101 。
- 小数部分 : 0.5的二进制是 0.1 ( 因为0.5在二进制中等于2的-1次方, 即 0.1 ) 。
- 因此, 5.5的二进制表示是 101.1 。

### 3. 规格化 :

- 将小数点左移直到小数点前的数变为1 ( 这在IEEE 754中是隐含的1 ) :  $1.011 * 2^2$ 。
- 这里的指数为2。

### 4. 指数 ( Exponent ) :

- IEEE 754单精度浮点数的指数偏移量为127, 所以实际存储的指数值为  $2 + 127 = 129$  。
- 129的二进制表示为 10000001 。

### 5. 尾数 ( Fraction ) :

- 由于小数点前是隐含的1, 我们只存储小数点后的部分, 即 011 。
- 因为尾数部分有23位, 我们需要在 011 后面补0, 直到满23位 : 01100000000000000000000 。

## 最终存储格式

- 符号位 : 0



- 指数：10000001
- 尾数：011000000000000000000000

因此，5.5在IEEE 754单精度浮点数格式中的存储方式是：

```
0 10000001 011000000000000000000000
```

## 变量

### 💙 什么是变量

- 在程序运行的过程中，其值可以改变。
- 内存中的一块空间（大小），空间内的内容（数据）可以在相同的类型内变化。

## 变量规则

- 变量名必须符合标识符规范。
- 变量在使用前必须先定义，定义变量前必须有相应的数据类型。

```
#include <stdio.h>
int main() {
    // int 123; // 报错，变量名必须符合标识符规范
    printf("%d\n", number); // 报错，因为number变量未定义先试用了
    int number = 1;
    return 0;
}
```

### 💙 变量理解

1. 变量在编译时为其分配内存空间
2. 可以通过变量名和地址访问内存

### 💡 对于变量

对于变量，在C语言中可以将其理解为 4 个维度的描述

1. 变量名
2. 内存地址：虚拟地址
3. 数据类型：决定了内存空间的大小
4. 数据的值：内存中存放数据（二进制数据）的字符集合

## 变量的声明与赋值

数据类型 变量名; // 声明变量是语句，所以必须以分号结尾

```
int weight;
```

```
int width,height; // 一次声明多个变量
// 等价于：
int width;
int height;
```

变量声明时，就为它分配内存空间，但是不会清除内存里面原先的数值。变量就会是一个随机的值，也可能是编译器设置的0或Null。所以对于变量一定要先赋值在使用。

```
int age; // 变量的声明
age = 18; // 变量的赋值
```

```
int age = 18;
```

```
int a = 1, b = 2; // 多种相同类型变量的赋值，可以写在同一行
```

```
int a, b, c, d, e;
a = b = c = d = e = f; // 连续赋值
```

声明变量后，不要忘记初始化赋值！定义变量时编译器并不一定会清空这块内存，所以值可能是无效的数据，运行程序，会异常退出。

## 变量的作用域

### 💙 块级作用域

在其定义所在的代码块中有效。

代码块外部不可见。

### 💙 文件作用域

在其定义所在的文件内中有效（从声明位置到文件结束有效）。

同一作用域内，不能定义重名的变量。

```
#include <stdio.h>
int main() {
    int m=10;
    if (m==10) {
        int n=20;
        printf("%d %d\n", m, n); // 10 20
    }
    printf("%d\n", m); // 10
    printf("%d\n", n); // 超出作用域，报错
    return 0;
}
```

```
#include <stdio.h>
int main() {
    // 代码块1 - start
```

```

{
    int a = 1;
    // 代码块2 - start
    {
        int b = 2;
        // 代码块3 - start
        {
            printf("a=%d, b=%d", a, b); // a=1, b=2
        } // 代码块3 - end
    } // 代码块2 - end
} // 代码块1 - end
return 0;
}

```

## 数据类型

### 💙 数据类型的作用

数据类型决定变量占用内存空间中的大小以及表示范围

## 整数类型

C 语言标准规定了以下几类整型：

- short [int] 短整型
- int 整型
- long [int] 长整型
- long long [int] 长长整型

同时，每种类型都可以被 signed 和 unsigned 修饰。

- signed 表示有符号位
- unsigned 表示无符号位

默认使用 signed 修饰符号性。

数据类型	符号性	占用空间	取值范围
short [int]	signed	2byte = 16bit	-32,768 ~ 32,767 ( $-2^{15} \sim 2^{15}-1$ )

数据类型	符号性	占用空间	取值范围
short [int]	unsigned	2byte = 16bit	$0 \sim 65,535$ ( $0 \sim 2^{16}-1$ )
int	signed	4byte = 32bit	$-2,147,483,648 \sim 2,147,483,648$ ( $-2^{31} \sim 2^{31}-1$ )
int	unsigned	4byte = 32bit	$0 \sim 4,294,967,296$ ( $0 \sim 2^{32}-1$ )
long [int]	signed	4byte   8byte	
long [int]	unsigned	4byte   8byte	
long long [int]	signed	8byte = 64bit	$-9,223,372,036,854,775,808 \sim 9,223,372,036,854,775,807$ ( $-2^{63} \sim 2^{63}-1$ )
long long [int]	unsigned	8byte = 64bit	$0 \sim 18,446,744,073,709,551,616$ ( $0 \sim 2^{64}-1$ )

## C 语言标准规定

- 未规定各种类型占用的存储空间的长度
- 规定了 `short int <= int <= long int <= long long int`
- 具体大小由编译系统自行决定
- `sizeof` 是测量数据类型或变量长度的运算符
- `short` 至少应为 2 字节
- `long int` 至少应为 4 字节