

C 语言入门教程

Grant Lee

2025 年 11 月 20 日

目录

1 C 语言程序的基本结构	4
1.1 最小示例 HelloWorld	4
1.2 #include 与头文件	4
1.3 main 函数（程序入口）	4
1.4 语句块与分号	5
1.5 printf 与常用占位符	5
1.6 从源代码到可执行文件：将自然语言翻译为机器语言	5
2 C 语言的变量与常量	8
2.1 变量与常量的基本定义	8
2.2 命名规则与规范	12
2.3 作用域与生命周期	12
3 从字面量（Literal）到数据类型	14
3.1 何为魔法数字？	14
3.2 定义与概念	15
3.3 字面量的具体区分方式	15
3.4 浮点数的后缀	19
3.5 总结	19
4 数据类型：数据在计算机中的表现形式	20
4.1 小白需掌握的一些入门知识	20
4.2 整数类型（Signed Integers）	21
5 数据类型——浮点类型（Floating-Point Numbers）	23
5.1 基本概念与名词解释	23
5.2 二进制浮点数	25
5.3 正规化的必要性	25
5.4 总结	25
5.5 示例 1：0.15625（可精确表示）	30
5.6 示例 2：-7.75（带符号）	31
5.7 示例 3：0.1（无法精确表示）	32
5.8 总结	33
5.9 字符类型（Character Types）	34
5.10 布尔类型（Boolean）	35
5.11 枚举类型（Enumerations）	35
5.12 指针类型（Pointers）	35

5.13 复合与派生类型	35
5 为什么我们需要数据类型	33
5.1 计算机与数学	33
5.2 数据类型的本质：对比特模式的解释	34
5.3 数据类型决定存储格式与运算规则	35
5.4 有限位宽带来的问题：溢出、截断与精度损失	36
5.5 小结	40
5.6 整数的存储与表示	41

4 数据类型：数据在计算机中的表现形式

4.1 小白需掌握的一些入门知识

在真正进入类型列表之前，先牢牢记住几个基础事实：

- ① 计算机底层只处理二进制的 0 与 1；
- ② 一位二进制称为一个比特（bit），8 个比特构成一个字节（byte），也是 C 语言中最基本的存储单位。
- ③ 所有变量不过是连续若干个字节的比特序列，类型只是告诉编译器应该如何解释这些比特。

本章仅描述各种类型在内存中的表现形式，即具体的比特布局与取值范围；关于“为何如此设计”留待其它章节。所以各位读者大可以先阅读后面的部分，等需要用到时再回头细看本章内容。

4.1.1 原码、反码与补码的概念

- **原码：**原码是最直观的表示方法，它直接用二进制数表示一个数，包括正负号。在原码中，最高位（最左边的位）是符号位，0 表示正数，1 表示负数。其余位表示数值本身。例如，十进制数 +5 的原码表示为 0000 0101，而 -5 的原码表示为 1000 0101。
- **反码：**正数的反码与原码相同；负数的反码是“符号位不变、其余位按位取反（0 变 1，1 变 0）”。例如，十进制数 -5 的反码表示为 1111 1010。在现代计算机中，反码主要用于过渡推导。
- **补码：**正数的补码与原码完全一致；负数的补码是在反码的基础上加 1。对于正数，其补码与其原码相同。对于负数，其补码是其反码加 1。补码的一个重要特性是，任何数的补码加上该数本身，结果总是 0。现代 CPU 的整数都存储为补码。

4.1.2 从十进制到补码：以 -42 为例

下面通过一个 8 位整数的完整示例，展示如何将十进制的负数 -42 转换为内存中的补码形式：

- ① 写出绝对值的二进制（原码）：

先对 42 进行十进制到二进制的转换：

$$42 = 32 + 8 + 2 = 2^5 + 2^3 + 2^1$$

因此 42 的二进制为 00101010（8 位，高位补零）。

负数 -42 的原码即在最高位标记符号位 1，其余保持不变：10101010。

② 求反码：

保持符号位不变，其余 7 位按位取反：

$$10101010 \xrightarrow{\text{取反}} 11010101$$

③ 得到补码：

在反码的基础上加 1：

$$11010101 + 1 = 11010110$$

这个 11010110 就是 -42 在 8 位补码表示下最终存入内存的比特模式。

4.2 整数类型 (Signed Integers)

4.2.1 有符号整数 (Signed Integers)

C 语言的带符号整数采用补码 (two's complement) 形式：最高位是符号位，其余位代表数值部分，所有位共同组成一个固定宽度的比特模式。

对于占 4 字节的 int 而言，一共有 32 个比特，记作 $b_{31}b_{30}\dots b_0$ 。最高位 b_{31} 是符号位，其余 31 位表示数值部分，它们共同决定唯一的比特模式。先把所有比特当作无符号整数：

$$U = \sum_{i=0}^{31} b_i \times 2^i \quad (\text{补码形式})$$

若符号位为 0，则补码表示的值就是 U ；若符号位为 1，则需要减去 2^{32} 才能得到负值：

$$\text{数值} = \begin{cases} U, & b_{31} = 0 \\ U - 2^{32}, & b_{31} = 1 \end{cases}$$

典型的 int 型数据的 32 位补码示例如下：

表 5：典型的 int 型数据的 32 位补码示例

二进制补码 (32 位)	说明	表示的值
0000...0000	全 0	0
1111...1111	全 1	-1
1000...0000	符号位为 1，其余为 0	-2^{31} (最小值)
0111...1111	符号位为 0，其余为 1	$2^{31} - 1$ (最大值)

下面列出了常见的有符号整数类型及其取值范围：

表 6：常见有符号整数类型的取值范围

类型	字节数	最小值	最大值
<code>short</code>	2	-2^{15}	$2^{15} - 1$
<code>int</code>	4	-2^{31}	$2^{31} - 1$
<code>long</code>	4 或 8	-2^{31} 或 -2^{63}	$2^{31} - 1$ 或 $2^{63} - 1$
<code>long long</code>	8	-2^{63}	$2^{63} - 1$

4.2.2 无符号整数（Unsigned Integers）

无符号整数和有符号整数的比特位宽度相同但不设专门的符号位——所有位都用于表示数值。对于宽度为 n 的无符号整数，其数值按二进制权展开得到：

$$U = \sum_{i=0}^{n-1} b_i \times 2^i,$$

因此取值范围为 0 到 $2^n - 1$ 。由于无符号整数的补码和原码相同，所以其实内存中保存的比特模式就是该无符号整数的二进制表示，按无符号规则直接解释。

下面列出了常见的无符号整数类型及其取值范围：

表 7：常见无符号整数类型的取值范围

类型	字节数	取值范围
<code>unsigned short</code>	2	$[0, 2^{16} - 1]$
<code>unsigned int</code>	4	$[0, 2^{32} - 1]$
<code>unsigned long</code>	4 或 8	$[0, 2^{32} - 1]$ 或 $[0, 2^{64} - 1]$
<code>unsigned long long</code>	8	$[0, 2^{64} - 1]$

5 数据类型——浮点类型 (Floating-Point Numbers)

C 提供 `float`、`double` 和 `long double` 三个主要的浮点类型，它们在现代平台上几乎都遵循 IEEE 754 浮点标准。浮点数通过科学计数法近似实数，既能表示极大值也能表示极小值，但尾数位数有限，只有有限精度。

5.1 基本概念与名词解释

在正式开始深入 IEEE 754 浮点数的细节之前，先在节 5.1 对这些基本概念做一个大致阐释，方便后续理解。

5.1.1 浮点数内存布局

IEEE 754 浮点数在内存中由三部分组成：

- **符号位 (sign):** 1 bit，决定浮点数的正负。若 $s = 0$ 则为正， $s = 1$ 则为负。
- **指数域 (exponent field):** 用于存储被偏置的指数 E 。`float` 占 8 位，`double` 占 11 位。
注意，指数域 = 硬件格式中的编码，这完全是人为规定的，是 IEEE-754 标准约定的编码规则，不完全等同于数学中的指数本身。
- **尾数域 (fraction/mantissa):** 存储有效数字的小数部分 f ，`float` 占 23 位，`double` 占 52 位。

以下面这个表格的形式或许更容易理解 IEEE 754 浮点数的内部内存空间的分配布局：

表 8: IEEE 754 浮点数的内存布局

部件	含义	float(32 bit)	double(64 bit)
符号位 (Sign)	表示正负号	1 bit	1 bit
指数域 (Exponent)	表示科学计数法中的指数部分	8 bits	11 bits
尾数域 (Mantissa/Fraction)	表示有效数字	23 bits	52 bits

IEEE 754 中，正规化浮点数的数学表示形式为：

$$(-1)^s \times (1.f)_2 \times 2^e$$

5.1.2 何为正规化浮点数

正规化浮点数 (Normalized Floating-Point Number) 的核心思想是：有效数字（尾数，Significand 或 Mantissa）的最高位必须是非零，并保持在规定的范围内。

在 IEEE 754 浮点数标准中，只要指数域 (exponent field) 既不全为 0、也不全为 1，浮点数就会被认定为 正规化浮点数 (normalized number)。这类浮点数被强制写成统一的科学计数法形式，以便把有限的存储位宽尽可能用于记录有效数字。

(1) 十进制正规化浮点数类比

阅读完上面这两段话的你或许会感到吃力，不用担心，下面我们通过对比中学时代学过的十进制科学计数法来帮助理解正规化浮点数的概念。十进制的写法可以概括为

$$\text{形如 } d_0.d_1d_2 \dots \times 10^k, \quad 1 \leq d_0 \leq 9,$$

为了唯一表示，我们规定其中 d_0 是介于 1 到 9（因为这里基数为 10）之间的首位有效数字。这就叫“正规化”。例如十进制数 13.25 可以写成

$$13.25 = 1.325 \times 10^1,$$

其中，1.325 称为尾数 (mantissa) 或有效数 (significand)；

10^1 中的 1 称为指数 (exponent)。

这就是一个正规化浮点数的例子，首位 d_0 必须是非零，且必须处于基数 radix 所决定的范围内。此例中 $d_0 = 1$ 、小数部分为 0.325、指数为 $k = 1$ 。若写成 0.1325×10^2 ，虽然数值相同，却违反了“首位非零”的规范化规则。

(2) 二进制正规化浮点数

有了前面的示例，再理解 C 语言中的浮点数就容易多了。我们已经知道，计算机在底层全部采用二进制进行存储和计算，而 C 语言的浮点数同样遵循这一原则，其科学计数法的基数 (base) 为 2。

因此，在采用二进制科学计数法表示浮点数时，也需要进行所谓的“正规化”(Normalization)。那么很显然，正规化之后，二进制浮点数的最高位只能是 1。既然这样，干脆就限定最高位恒为 1 吧！

IEEE 754 标准就规定这最高位不必显式存储，称为“隐含位”(implicit bit 或 hidden bit)。IEEE 754 于是在存储时把这一位视作 隐含位，只记录后续的小数位：

$$\text{二进制正规化形式 } 1.b_1b_2 \dots \times 2^e, \quad b_i \in \{0, 1\},$$

这样无论是哪一个二进制实数，正规化后的尾数都固定以‘1.’开头，等价写法就是常见的 $1.f$ 。仍以 13.25 为例，它的二进制展开为 $1.10101_2 \times 2^3$ ，只需把 10101 存入尾数域，并保存符号位和偏置后的指数，就得到了唯一的浮点编码。

5.2 二进制浮点数

IEEE 754 浮点标准使用二进制表示形式:

$$x = (-1)^s \times 1.\text{fraction} \times 2^e.$$

二进制浮点数的正规化要求尾数满足:

$$1 \leq m < 2,$$

因此尾数的最高位总是 1, 称为隐含的 leading 1。

5.3 正规化的必要性

如果不对尾数进行规范化, 同一个数可以有无数种表示方式, 例如:

$$50 = 5 \times 10^1 = 50 \times 10^0 = 0.5 \times 10^2,$$

这会导致比较困难、硬件实现复杂。

正规化的目的在于让浮点数表示 唯一且标准化。

5.4 总结

正规化浮点数的尾数范围如下:

进制	浮点形式	尾数正规化区间
十进制	$m \times 10^e$	$1 \leq m < 10$
二进制	$1.f \times 2^e$	$1 \leq m < 2$

因此我们只需要把小数部分 10101 (实际的有效数字) 写入尾数域, 把最高位的 1 看作 隐含位 (hidden bit 或 implicit leading 1), 就能在不额外占用位宽的情况下完成存储。

在 C 语言中的浮点数表示中, 尾数域大致就是这样

在二进制浮点数中, 尾数的最高有效位被固定为 1, 这样正规化浮点数的有效数字就可以统一写作

$$1.f,$$

其中 f 即尾数域中存放的二进制小数。

对于正规化浮点数, 其真实指数值为

$$e = E - \text{Bias},$$

其中 E 为指数域存储的无符号整数, Bias 为偏置常数: 对 float (32 位) 来说 $\text{Bias}_{\text{float}} = 127$, 对 double (64 位) 来说 $\text{Bias}_{\text{double}} = 1023$ 。

综上，一个 IEEE 754 正规化浮点数的数值可表示为

$$(-1)^s \times 2^e \times (1.f).$$

这种“尾数前恒有隐含 1”且指数有限的表示方式，使正规化浮点数在相同存储位宽下能够提供更高的精度，因为最高有效位无需存储，从而节省出更多尾数位用于表示小数部分。

todo: 在这里插入一段关于对正规化浮点数的表述，不然小白不知道这是什么

在 IEEE 754 标准中，只要指数域既不是全 0 也不是全 1，该浮点数就被视作正规化浮点数 (*normalized number*)。此时真实指数满足

$$e = E - \text{Bias}, \quad \text{Bias}_{\text{float}} = 127, \quad \text{Bias}_{\text{double}} = 1023$$

同时尾数域默认有一个隐含的最高位 1，因此有效数字被解释为 $1.f$ 。综合起来，一个正规化浮点数的数值可写作

$$(-1)^s \times 2^e \times (1.f)$$

这也是浮点数能够在给定位宽下获得最大精度的原因，因为最高位的 1 不必显式存储，节省出更多位给小数部分。

当你对这些感到疑惑的时候，请放心，这是正常的，因为浮点数的表示方式本身就比较复杂。接下来我们将逐步解释这些新出现的概念，并通过具体示例来帮助理解。

(1) 指数域：偏置 (Bias) 的本质与指数

首先，由上文，我们已知 IEEE 754 中，正规化浮点数的数学表示形式为：

$$(-1)^s \times (1.f)_2 \times 2^e$$

其中，指数 e 称为实际指数 (actual exponent)，表示科学计数法中 2 的次幂。然而，从内存布局来看，浮点数并不直接存储 e ，指数域里真正存储的是经过偏置处理后的值 E ，称为存储指数 (stored exponent)。那么，这 e 和 E 有什么区别呢？又为什么需要偏置呢？这就要从指数域的存储方式说起。

指数域就是浮点数内部专门为“指数”分配的固定 bit 空间，其由若干固定的二进制位组成，所以本身只能存储无符号整数；然而，科学计数法中的实际指数 e 可以为负，如：

$$1.23 \times 2^{-3}.$$

为了解决指数域不能直接表示负数的问题，IEEE 754 引入了偏置 (Bias) 的概念，使得存储值 E 与实际指数 e 的关系为：

$$E = e + \text{Bias}.$$

对于单精度浮点数 (float)，偏置为：

$$\text{Bias} = 127.$$

例如，对于单精度浮点数 (float)，当实际指数为 $e = -3$ 时，有：

$$E = -3 + 127 = 124,$$

其存储到指数域中的二进制形式为：

$$01111100_2.$$

这种机制保证了指数域永远为非负数，使其能使用无符号二进制进行存储。例如：

$$e = 0 \Rightarrow E = 127,$$

$$e = -1 \Rightarrow E = 126,$$

$$e = -4 \Rightarrow E = 123,$$

$$e = 2 \Rightarrow E = 129.$$

最终浮点数的数值由下式决定：

$$(-1)^s \times (1.f)_2 \times 2^{E-\text{bias}}$$

(2) 尾数域：隐含位与有效数字

同样的，由上文，我们已知 IEEE 754 中，正规化浮点数的数学表示形式为：

$$(-1)^s \times (1.f)_2 \times 2^e$$

这其中尾数域 f 用于表示有效数字的小数部分，是浮点数精度的来源。以单精度浮点数为例，总共 32 位被固定划分为：

1 bit sign + 8 bits exponent + 23 bits fraction.

尾数域由 23 位二进制小数组成，可写为：

$$f = 0.b_1 b_2 \dots b_{23}, \quad b_i \in \{0, 1\}.$$

浮点数的数值可以分解为：

$$(-1)^s \times \text{有效数字 (significand)} \times \text{指数项 } 2^e.$$

其中，有效数字完全由尾数域与其前方的隐含位共同决定。尾数域本质上是一个二进制小数，用来描述有效数字在 $[1, 2)$ 或 $[0, 1)$ 区间内的细节刻度，因此尾数域的位数越多、精度越高。

接下来，我们将在此基础上讨论为何 IEEE 754 要采用“规格化表示”，以及规格化数中出现的“隐含位”如何提升尾数域的有效精度。

在 IEEE 754 单精度浮点数 (32 bit) 中，数值由三个部分组成：符号位 s 、指数域 E 和尾数域 f 。对于正规化数，其值由下式给出：

$$(-1)^s \times (1.f)_2 \times 2^{E-\text{bias}},$$

其中 $\text{bias} = 127$ ，尾数域 f 由 23 位二进制小数组成。

1. 尾数域的含义

尾数域 f 表示为：

$$f = 0.b_1 b_2 \dots b_{23}, \quad b_i \in \{0, 1\},$$

而真正参与计算的有效数字为：

$$1.f = 1 + \sum_{i=1}^{23} b_i 2^{-i}.$$

对于正规化浮点数，IEEE 754 规定小数点左侧的最高位恒为 1，称为“隐含位 (hidden bit)”，因此无需存储。这使得单精度浮点数在尾数部分实际上提供了 24 位有效二进制精度。

2. 尾数域的作用与精度

隐藏最高位后，尾数域所能表达的有效数字范围为：

$$1 \leq 1.f < 2,$$

因此在区间 $[1, 2)$ 内，相邻两个可表示浮点数之间的间距 (ULP, unit in the last place) 约为：

$$\text{ULP} \approx 2^{-23}.$$

这意味着尾数域的 23 位二进制小数决定了浮点数的精度，约等价于十进制 7 位有效数字。

3. 示例：尾数域如何表达小数

例如：

- 数值 $1.5 = (1.1)_2$, 其尾数部分为 $f = 0.1_2$, 故尾数域编码为 1000...0。
- 数值 $1.25 = (1.01)_2$, 尾数部分为 $f = 0.01_2$, 尾数域编码为 0100...0。

在这两种情况下，有效数字均按 $1.f$ 的形式参与浮点数计算。

4. 非正规化数中的尾数域

当指数域全为 0 且尾数域非零时，浮点数为“非正规化数 (Subnormal Number)”。此时不再存在隐含位，其值为：

$$(-1)^s \times (0.f)_2 \times 2^{1-\text{bias}}.$$

这种设计实现了“渐进下溢 (gradual underflow)”，使得数值从最小正规化数平滑过渡至 0，避免出现表示范围的突然中断。

5. 尾数域总结

尾数域是浮点数中负责存储有效数字的小数部分的 23 个二进制位：

- 对正规化数：有效数字为 $1.f$, 包含隐藏的最高位 1。
- 对非正规化数：有效数字为 $0.f$, 无隐含位。
- 尾数域决定浮点数的精度，其最低位决定了最小可区分增量 (ULP)。

整体可将浮点数视为：

$$\text{数值} = \text{符号} \times \text{有效数字 (尾数)} \times 2^{\text{的幂 (指数)}}.$$

数域由 f 的 23 位二进制小数组成, 可记为 $b_1 b_2 \dots b_{23}$, 其数值可理解为

$$1.f = 1 + \sum_{i=1}^{23} b_i \times 2^{-i}, \quad b_i \in \{0, 1\}.$$

通过把小数点固定在最高位 1 的右侧 (即“尾数正规化”), 乘上 2^{E-127} 就能得到浮点数的真实大小。这样 $1.f$ 的取值范围始终位于 $[1, 2 - 2^{-23}]$, 配合指数域就能覆盖相当宽的数量级。

把隐含位算在内, 单精度共提供 24 位有效数字, 大约相当于 7 位十进制有效数字。每向右多一位, 数值的分辨率就减半, 因此第 i 位代表 2^{-i} 的权重; 当我们把二进制小数写成 010000... 这种形式时, 其实就是在说明尾数当前比 1.0 大了一个 2^{-2} 。尾数的最末位对应的最小增量通常称为 1 ULP (unit in the last place), 在单精度中约等于 2^{-23} , 这也是浮点计算误差常用的比较尺度。

由于大多数十进制小数无法被有限二进制尾数精确表示, IEEE 754 定义了多种舍入模式, 默认为 round-to-nearest, ties-to-even。运算结果在写回尾数域时, 如果多出的比特高于 23 位, 就需要根据该模式决定是否对尾数加 1, 再配合可能发生的进位去调整指数。理解“尾数域 + 舍入”的配合有助于分析诸如累加误差、比较操作失效等常见的浮点陷阱。其中最高位的 1 不存储于内存中, 称为隐含位 (hidden bit)。

5.4.1 规格化数 (Normalized Numbers)

指数域既不全为 0, 也不全为 1, 此时浮点数为正规化数。其表示为:

$$(-1)^s \times (1.f) \times 2^{E-\text{bias}}$$

5.4.2 非规格化数 (Subnormal Numbers)

当指数域全为 0 时, 为非规格化数, 其表示为:

$$(-1)^s \times (0.f) \times 2^{1-\text{bias}}$$

尾数不含隐含位 1。

5.5 示例 1: 0.15625 (可精确表示)

十进制转二进制

$$0.15625 = 0.00101_2$$

科学计数法

$$0.00101_2 = 1.01_2 \times 2^{-3}$$

因此 $e = -3$, 尾数 $f = 0.01$ 。

符号位

数为正, $s = 0$ 。

指数加偏置

$$E = -3 + 127 = 124 = 01111100_2$$

尾数域

$$f = 01000000000000000000000000000000$$

最终 IEEE 754 表示

$$0 | 01111100 | 010000000000000000000000$$

十六进制为:

0x3E200000

5.6 示例 2: -7.75 (带符号)

十进制转二进制

$$7.75 = 111.11_2$$

科学计数法

$$111.11_2 = 1.1111_2 \times 2^2$$

因此 $e = 2$, 尾数 $f = 0.1111$ 。

符号位

负数, $s = 1$ 。

指数加偏置

$$E = 2 + 127 = 129 = 10000001_2$$

尾数域

$$f = 11110000000000000000000000000000$$

最终 IEEE 754 表示

$$1 | 10000001 | 11110000000000000000000000000000$$

十六进制为:

0xC0F80000

5.7 示例 3: 0.1 (无法精确表示)

十进制转二进制

$$0.1 = 0.00011001100110011\dots_2 \quad (\text{无限循环})$$

科学计数法

$$0.000110011\dots_2 = 1.100110011\dots_2 \times 2^{-4}$$

因此 $e = -4$ 。

符号位

正数, $s = 0$ 。

指数加偏置

$$E = -4 + 127 = 123 = 01111011_2$$

尾数域 (截断为 23 位)

$$f = 10011001100110011001101$$

最终 IEEE 754 表示

$$0|0111011|10011001100110011001101$$

十六进制为:

0x3DCCCCCD

5.8 总结

本节通过三个具有代表性的浮点数转换示例 (可精确表示、带符号、不可精确表示), 完整展示了 IEEE 754 浮点数的构造方式, 包括符号位、偏置指数、隐含位、尾数域等核心概念的具体化过程。

IEEE 754 布局 每个浮点值在内存中由以下三部分构成:

- 符号位 s (1 位), 决定数值正负, 并允许存在 ± 0 ;
- 阶码 e (`float` 为 8 位、`double` 为 11 位), 采用偏置表示, 偏置分别是 127 和 1023;
- 尾数 f (`float` 为 23 位显式尾数, `double` 为 52 位), 正规值隐式包含最高位 1, 即 $1.f$ 。

正规数的值由

$$(-1)^s \times 1.f \times 2^{e-\text{bias}}$$

给出: 符号位决定整体符号; 尾数 f 被解释为二进制小数 (例如比特 $b_1 b_2 \dots$ 代表 $1 + b_1 2^{-1} + b_2 2^{-2} + \dots$); 阶码存储的是偏置后的指数, `float` 的偏置为 127, 即实际指数是 $e - 127$ 。例如 `float` 值 1.5 的比特模式为 $s = 0$ 、 $e = 127$ (对应指数 0) 和 $f = 0.1_2$, 代入即可得到 $1.1_2 \times 2^0 = 1.5$ 。如果阶码为 0, 则不再假设最高位 1, 形成次正规数: 值为 $(-1)^s \times 0.f \times 2^{1-\text{bias}}$, 能平滑连接到 0。除了正规数, 还存在 $\pm\infty$ 以及 NaN, 每种情况都占用特定的位模式。C 允许实现提供更宽的 `long double` (例如 x87 80 位扩展精度或 128 位双二进制精度), 因此要以 `sizeof` 或 `long double` 相关宏确认实际宽度。

数值类别与特殊值 IEEE 754 将浮点结果划分为：

- **正规数**: 阶码在 $(0, E_{\max})$ 范围, 可用最大尾数精度表示;
- **次正规数**: 阶码为 0, 尾数没有隐含最高位, 用来平滑接近 0 时的下溢;
- **零**: 阶码和尾数都为 0, 但符号位仍然保留, 区分 $+0$ 和 -0 ;
- **无穷大**: 阶码全为 1, 尾数为 0, 表示溢出或除以 0 的结果;
- **NaN**: 阶码全为 1, 尾数非 0, 表示未定义或非法结果, quiet NaN 会在表达式中传播, signaling NaN 会触发异常。

类型	字节数	近似范围	有效数字
<code>float</code>	4	$\pm 3.4 \times 10^{38}$	7–8 位十进制有效位
<code>double</code>	8	$\pm 1.7 \times 10^{308}$	15–16 位
<code>long double</code> (实现相关)	16	$10^{\pm 4932}$ 量级	18–19 位或更多

精度、舍入与比较 浮点运算按照舍入模式进行, 缺省模式为“最接近且偶数优先”。有限尾数意味着不是所有十进制小数都能精确表示, `0.1f` 就是一个无限循环的二进制小数。`float.h` 中提供 `FLT_EPSILON`、`DBL_EPSILON` 等常量描述机器精度, 还提供 `FLT_MIN/FLT_MAX` 等范围边界。比较两个浮点数时应优先比较差值是否在容忍误差内, 而非直接使用 `==`。

工程实践建议

- **类型选择**: 除非明确需要节约内存或与硬件接口, 优先使用 `double`, 它提供良好的可移植性; `long double` 可在高精度算法中减少累积误差。
- **转换**: 整数向浮点转换可能溢出为 $\pm\infty$, 浮点向整数转换会舍弃小数部分, 若超出整数范围则产生未定义行为, 应结合 `fma`、`trunc`、`llround` 等函数控制过程。
- **异常与状态**: IEEE 754 定义上溢、下溢、除以零、无效操作和不精确五种异常, C 的 `fenv.h` 可检测或设置浮点环境, 但不同平台支持度不同。

5.9 字符类型 (Character Types)

`char` 始终占 1 字节, 其比特模式可以按不同方式解释:

- `char` (实现定义为有符号或无符号);
- `signed char`: 补码, 范围 $[-128, 127]$;
- `unsigned char`: 无符号, 范围 $[0, 255]$ 。

将 ASCII 或 UTF-8 的单个字节写入 `char` 时, 就是直接存放对应的 8 位编码单元。

5.10 布尔类型 (Boolean)

C99 引入的 `_Bool` 以及 `<stdbool.h>` 提供的 `bool` 也占 1 字节。任何非零比特模式被解释为真，零值表示假；编译器在赋值时会把结果归一化为 0 或 1 存放。

5.11 枚举类型 (Enumerations)

`enum` 的存储形式是某种整型（通常是 `int`），编译器会选择能够表达所有枚举常量的最小宽度。每个枚举值最终就是一个整型比特模式，与所选整型的表示保持一致。

5.12 指针类型 (Pointers)

指针保存的是内存地址，长度为实现相关（32 位平台常为 4 字节，64 位平台常为 8 字节）。不同指针类型在内存中的比特模式相同，差异仅体现在编译器如何解释该地址所指向的数据。

5.13 复合与派生类型

- **数组：**元素的比特模式在内存中顺序排列，总长度等于元素字节数乘以元素个数。
- **结构体：**成员按声明顺序逐一存放，必要时插入对齐所需的填充字节。每个成员占用与其类型一致的存储形式。
- **联合：**所有成员共享同一块内存，长度等于最长成员或最大对齐要求，读取时按具体成员类型解释当前比特模式。

本章概述了 C 语言各类数据在内存中的布局方式：整数采用补码、无符号整数直接映射、浮点遵循 IEEE 754、字符占单字节、布尔存 0/1、枚举与指针映射为整型或地址值，复合类型由其成员或元素的具体表示拼接而成。