C 语言入门教程

Grant Lee

2025 年 11 月 17 日

目录

1 C 语言程序的基本结构

1.1 最小示例 HelloWorld

代码 1: 最小 C 程序示例

```

  #include <stdio.h>          // 1) 预处理指令：包含头文件

int main(void) {             // 2) 主函数：程序从这里开始执行
    printf("Hello, World!\n"); // 3) 调用库函数进行输出
    return 0;                // 4) 返回状态码 0：正常结束
}
```

1.2 #include 与头文件

- `#include` ：#include 是预处理指令，在编译开始前，编译器会把被包含的文件内容“复制粘贴”进来。
- `stdio.h`：`stdio.h` 中定义了常用的输入输出函数（如 printf, scanf）。如果事先不在预处理阶段声明定义，后续却在代码中使用了 stdio.h 中的函数，那么编译器就会报错。
- 尖括号 `<...>` 在系统路径查找；引号 `"my.h"` 先在当前目录再到系统路径查找。

1.3 main 函数（程序入口）

代码 2: 常见 main 函数入口示例

```
int main(void)
// 或
int main(int argc, char *argv[])
```

在 C 语言中，main() 是程序的唯一入口函数，程序的执行从这里开始。

- 接受：main 函数的参数只能是 void 或命令行参数。究其原因是因为 main 函数是由系统启动代码调用的，系统启动时，系统只知道程序名、命令行参数和环境变量。
- 返回值：在 C 语言标准（C99、C11、C17、C23）中明确规定，main 函数必须返回一个 int 类型的值，不能是 char、float、double、void 等其他类型，用来表示程序的执行状态。

① return 0: 表示正常结束。

② return 非 0: 表示异常结束，具体值可自定义以表示不同错误类型。

1.4 语句块与分号

- 花括号 { ... } 构成语句块（复合语句），可包含声明与可执行语句。
- 大多数 C 语句以分号 ; 结尾：表达式语句、声明（含初始化）、return 等。

1.5 printf 与常用占位符

代码 3: printf 与占位符示例

```
#include <stdio.h>
int main(void) {
    int a = 42; double x = 3.14159; char c = 'A';
    printf("a=%d, x=%.2f, c=%c\n", a, x, c); // 输出 a=42, x=3.14, c=A
    return 0;
}
```

1.6 从源代码到可执行文件：将自然语言翻译为机器语言

C 程序从源代码到可执行文件一般经历四个主要阶段：**预处理**、**编译**、**汇编**和 **链接**。下面以 gcc 为例分别说明，并给出对应命令。为便于工程化，还补充多文件与库的常见用法及常见问题。

（1）预处理（Preprocessing） 主要工作：展开 #include、宏替换（#define）、条件编译（#if/#ifdef 等），并去除注释，得到纯 C 源。

```
gcc -E hello.c -o hello.i
```

产物：.i 文件（预处理后的 C 代码）。

（2）编译（Compilation） 对 .i 做词法/语法/语义分析与优化，生成目标架构的汇编代码。

```
gcc -S hello.i -o hello.s        % 也可直接对 .c 使用 -S
```

产物：.s 文件（汇编源码）。

（3）汇编（Assembly） 把汇编转为可重定位的目标文件，包含代码段、数据段与符号表。

```
gcc -c hello.s -o hello.o        % 也可 gcc -c hello.c -o hello.o
```

产物：.o 文件（目标文件）。

（4）链接（Linking） 将若干 .o 与所需库合并，进行符号解析与重定位，加入启动代码，生成可执行文件（Linux 为 ELF，Windows 为 PE）。

```
gcc hello.o -o hello
```

（5）装载与运行（Loading & Running） 操作系统加载可执行文件到内存，先进入运行时入口 `_start`，再调用用户的 `main()`；`main` 返回值作为进程退出码。

（6）多文件与库的基本用法

```
gcc -c main.c -o main.o
gcc -c util.c -o util.o
gcc main.o util.o -o app              % 链接生成可执行文件

ar rcs libmylib.a foo.o bar.o         % 生成静态库
gcc main.o -L. -lmylib -o app2        % 使用静态库（库放在对象文件之后）

gcc -fPIC -c foo.c -o foo.o
gcc -shared -o libmylib.so foo.o      % 生成共享库
gcc main.o -L. -lmylib -o app3        % 运行时需能找到共享库
```

（7）常见问题速查

- 头文件找不到：添加包含路径 `-Ipath`。
- 未定义引用（`undefined reference`）：缺少对应对象文件或库，或库链接顺序在前；应将 `-lxxx` 放在使用到该库的对象文件之后。
- 库找不到：添加库路径 `-Lpath`，并确认库名（`libxxx.a`/`libxxx.so` 对应 `-lxxx`）。
- 位宽/架构不匹配：确认 `-m32`/`-m64` 或交叉编译器是否正确。

（8）嵌入式特例 在无操作系统的嵌入式环境中，仍有编译与链接步骤，但会使用启动文件与链接脚本生成固件映像（如 `.hex`/`.bin`），`main()` 通常为永不返回的主循环。

构建流程：

`.c` 源文件 ⇒ 预处理（展开 #include/#define）⇒ 编译（生成 `.o`/`.obj`）⇒ 链接（合并库与目标文件）⇒ 可执行文件

命令行示例（以 gcc 为例）：

```
gcc hello.c -o hello   # 生成可执行文件
./hello                # 运行（Windows: hello.exe）
```

2 C 语言的变量与常量

2.1 变量与常量的基本定义

2.1.1 变量

变量（Variable）是程序在运行过程中用于存储数据的命名空间。每个变量在内存中都有一个存储单元，用于保存值，其内容可以在程序执行过程中改变。你可以把它暂时理解为一个装满数据的容器或者是一个可以住人的房间。例如：

代码 4: 变量声明示例

```
int a = 10;
float b = 3.14;
char c = 'A';
```

2.1.2 常量

在 C 语言程序设计中，常量（Constant）是指程序在运行过程中，其值不可改变的量。合理地定义常量可以提升代码的可读性和可维护性。本文对比三种常用常量定义方式：`#define` 宏常量、`const` 常量与 `enum` 枚举常量，并给出实践建议。

(1) 符号常量 `#define`

`#define` 在预处理阶段进行文本替换，指定用一个符号名称来代替一个常量。它不参与类型检查，也不分配存储空间。适合表达编译期已知且简单的固定值。

代码 5: 使用 #define 定义宏常量

```
#include <stdio.h>

#define PI 3.14       //经过指定后，本文件中所有PI都会被替换为3.14。
#define AREA(r) (PI * (r) * (r))      //定义函数宏，用于计算圆面积。

int main() {
    float r = 2.0;
    printf("Area = %.2f\n", AREA(r));
    return 0;
}       //简单理解：只是个文本替换而已。
```

在程序编译之前，预处理器会将代码中的所有 `PI` 替换为 3.14，所有 `AREA(r)` 替换为 (3.14 * (r) * (r))。

(2) 常变量 `const`

`const` 是 C 语言中的一个关键字，表示“只读（只可读，不可改）”。它可以用来修饰变量、指针、函数参数等，使其在程序运行过程中不可被修改。

常变量具有类型信息，占用内存空间，只是其值不可通过该标识符修改。适合表达接口常量、配置值等。

代码 6: 使用 const 定义常量变量

```
#include <stdio.h>

const int   MAX_CONN = 1024;
const float PI       = 3.14159f;

int main(void) {
    MAX_CONN = 2048; //  编译报错：只读对象不可修改
    printf("%f\n", PI);
    return 0;
}
```

要注意一件事情: 修饰一个变量，只是使其在语义上为“只读”。

也就是说编译器虽然会禁止通过该常变量名字修改，但并不保证存储区域物理上不可修改。const 变量仍然占内存，只是不可通过其标识符修改。

(3) enum 枚举常量

enum 定义一组自定义的整数常量，往往是一组有着共同特性的数据集合或是状态标签。这里在展开讨论这一常量之前，先讲解一下枚举的概念。

枚举

枚举（Enumeration）是一种用户自定义的数据类型，它由一组自定义标签的整数常量组成。每个枚举成员都对应一个整数值（可以为负数），默认情况下，从 0 开始递增，也可以手动指定起始值或各个具体值。枚举的主要目的是为了提高代码的可读性和可维护性。

比如春夏秋冬这四个数据就是一组枚举数据，它们可以用一个枚举类型来表示；再比如生活中的通讯录，就是一个非常典型的枚举数据，如图 1 所示。

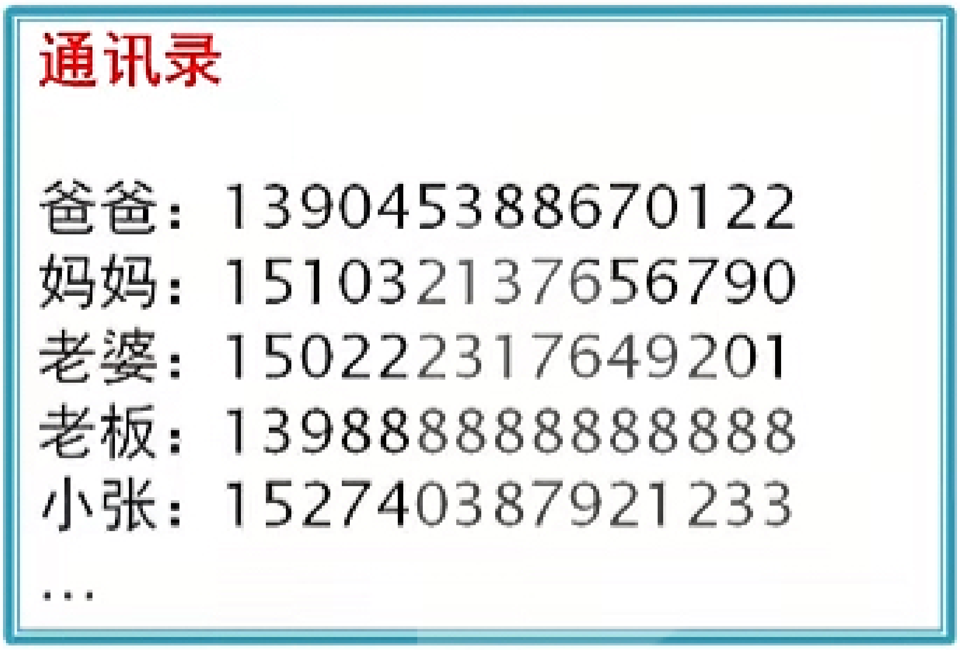


图 1: 生活中的枚举数据：通讯录

电话号码 11 位数字，按道理来说，一共有 10^{11} 种可能性，但实际上，我们的通讯录中并不会存入这么多电话号码，而是只会有其中离散的一部分，比如你亲人、朋友、同事的号码。此时当我们需要拨打电话时，只能从通讯录这几个离散的数据里面挑选。

枚举的基本规则如代码 7所示

代码 7: 使用 enum 基本规则

```
enum Color  { RED, GREEN, BLUE };
// 数值若不自定义，则依次为 0,1,2
enum Weekday{ MON=1, TUE, WED, THU, FRI, SAT, SUN };
//若自定义了起始值，则后续依次递增1
enum State { ST_INIT=-5, ST_RUN=8, ST_ERR=9, ST_STOP=100 };
//也可自定义每个枚举值

//注意：枚举数据可以为负数，但必须为整数。
```

说完了基本规则，让我们来探讨一下枚举类型变量使用方法，让我们通过一个示例来加深理解。

代码 8: 使用 enum 定义变量

```
#include <stdio.h>

enum Color { RED, GREEN, BLUE }x;  // 定义枚举类型 Color 并声明变量 x



enum Weekday { MON, TUE, WED, THU, FRI, SAT, SUN };
enum Weekday this_month;  // 声明枚举类型 Weekday 变量 this_month
//这与上面Color的声明方式是等价的。

int main(void) {
    x = RED;  // 给枚举变量 x 赋值  //枚举变量的赋值只能在枚举成员中选择
    this_month = TUE;  // 给枚举变量 this_month 赋值
    printf("Color: %d, Weekday: %d\n", x, this_month);

    return 0;
}
```

在上面的示例中，我们定义了两个枚举类型，相信大家已经理解了枚举的基本用法。接下来我们来接着探讨 enum 的常量特性。

`enum` 常量是编译期符号，它不同于 `const` 常量，也不同于 `#define` 宏常量在预处理阶段就进行文本替换。enum 常量在编译期被替换为对应的整数值，不占用任何内存空间，且只能表示整数类型。枚举变量一旦被声明之后，其值只能在枚举成员中选择，不能赋予其他整数值。这一安全特性使得枚举常量非常适合表示离散的状态或类别，可以有效提高代码的安全性和可读性。

2.2 命名规则与规范

2.2.1 命名规则

对于字符的组成，变量和常量遵循相同的规则。变量名只能由字母、数字和下划线组成，且首字符不能为数字（虽然语法上允许首字符为 _ ，但在实际编程和标准库约定中有一些潜在风险）

- 区分大小写，如 `age` 与 `Age` 是不同变量；
- 不得与关键字（`int`, `return` 等）同名；

2.2.2 代码命名规范

（1） 变量命名规范：变量命名通常使用小写字母和下划线 (例如 `total_count`)，或者在某些语言中使用驼峰式命名。（例如 `totalCount` ）

（2） 常量命名规范: 常量通常采用全大写字母，单词之间用下划线分隔。这是为了让开发者在看到常量时，能够立刻识别它们是固定值，而不是可变的数据。例如：`MAX_SIZE`、`PI`、`BUFFER_LIMIT`。

这种约定使得代码更加一致，并且提高了可读性和维护性。

2.3 作用域与生命周期

2.3.1 变量

（1） 局部变量：定义在函数或语句块中，仅在该范围内有效。每次进入函数时该变量都会被重新创建并初始化，生命周期仅限于函数调用期间。当函数或语句块执行结束后，变量随即被销毁，内存空间被回收；

（2） 全局变量：定义在所有函数外部，全文件范围内可访问。其生命周期从程序开始运行到程序结束，在此期间始终占据固定的内存空间。全局变量可被同一文件内的所有函数访问（若需跨文件访问，可使用 extern 声明）。

（3） 静态变量（`static`）：使用关键字 static 声明。它的作用域视其属于局部变量还是全局变量而定，但其生命周期贯穿整个程序执行过程。

也就是说，即使该变量为在函数或语句块中定义的局部变量，其在函数调用结束后也不会被销毁，而是保留其上一次的值；下次进入函数时会继续使用原来的值，而不是重新初始化。

示例

代码 9: 局部变量与静态局部变量示例

```
void foo() {
    int x = 0;           // 局部变量
    static int y = 0;    // 静态局部变量
    x++;
    y++;
    printf("x=%d, y=%d\n", x, y);
}
```

连续调用 `foo()` 三次的输出为：

```
x=1, y=1
x=1, y=2
x=1, y=3
```

由于局部变量在函数调用时被创建，并在函数结束后自动销毁，因此每次进入函数时，x 都会重新初始化为 1。而静态局部变量 y 的生命周期贯穿整个程序运行过程，因此它的值在每次函数调用后都会保留并继续增加。

2.3.2 常量

（1） 局部常量：当常量在函数内部使用 `const` 声明时，它的作用域仅限于该函数或代码块。这样的常量在函数调用结束后被销毁；

（2） 全局常量：当常量在函数内部使用 const 声明时，它的作用域仅限于该函数或代码块。这样的常量在函数调用结束后被销毁。全局变量可被同一文件内的所有函数访问（若需跨文件访问，可使用 extern 声明）。

（3） 静态常量（`static`）：与前面静态变量相似，如果常量声明为 static，它的生命周期会贯穿整个程序运行期间，即使它的作用域限于声明的文件或函数。

3 从字面量（Literal）到数据类型

3.1 何为魔法数字？

首先来解释一下什么是字面量。在 C 语言中，字面量指的是直接出现在源代码中的常量值，无需标识符即可使用。说得大白话一点，就是代码中一个又一个数据。如代码 10 所示，代码中出现的一个又一个常量数字如 42、'A'、3.1416、Hi 就是字面量。

代码 10: C 语言中字面量的示例

```c
#include <stdio.h>

int main(void) {
    int x = 42;           // 42 是整型字面量
    char c = 'A';         // 'A' 是字符字面量
    double pi = 3.1416;   // 3.1416 是浮点字面量
    printf("%s\n", "Hi"); // "Hi" 是字符串字面量
    return 0;
}
```

在我们日常生活中，我们每天都会接触大量的数据，比如购物、计数。但我们从来不会纠结于数据类型或是字面量这些乱七八糟的东西。这是因为在日常生活中，我们普遍使用**十进制**，所以对于我们彼此交流的时候，从来都不会纠结于进制、类型问题。

但在计算机中可不是这样，首先在计算机底部存储数字使用二进制，而其他八进制、十六进制等进制也是常见的表示方式，再加上由于计算机存储数据的方式不同又诞生了浮点数、整型数、字符。所以当你冷不丁地给计算机抛出一个数字 42 时, 计算机并不能理解这个数字到底是什么进制、类型，如代码 11 所示，计算机并不能理解这是十进制的 42，还是八进制的 34，还是十六进制的 68。它更无法理解这到底是 int、double 还是 float。

计算机此时只能按照默认处理规则进行处理，默认情况下这个 42 将被理解为十进制、int 类型的数据。但如果你的本意并不是这样，而是想表达其他进制或类型的数据，那么计算机就会误解你的意思。那么在工程场景下，往往就会引发严重的错误。

代码 11: 魔法数字实例

```
42
```

计算机遇到这种情况时，它是无法理解的，因为它不知道这个数字到底是什么进制、类型。这就好像一个麻瓜突然看到一个魔法师念咒语一样（来自哈利波特），它根本无法理解魔法师到底在说什么。对于这种情况，我们形象地称为魔法数字（Magic Number）。

3.2 定义与概念

在 **C 语言标准**（如 ISO/IEC 9899:2018, C17）中，“字面量”（*literal* 或 *literal constant*）是一个正式存在的术语。它指的是 **直接出现在源代码中的常量值**，无需标识符即可使用。

字面量类型	示例	说明
整型字面量 (integer literal)	`123`, `0x7B`, `010`	十进制、十六进制、八进制整数
浮点型字面量 (floating-point literal)	`3.14`, `1e-3`	双精度浮点数
字符字面量 (character constant)	`'A'`, `'\n'`	单个字符，本质为 `int` 类型
字符串字面量 (string literal)	`"Hello"`	以 `\0` 结尾的 `char` 数组

3.3 字面量的具体区分方式

C 语言里面通过“前缀（`prefix`）”和“后缀（`suffix`）”来对字面量（`literal`）进行类型和进制的区分。本节总结了 C 语言中通过前缀（`Prefix`）与后缀（`Suffix`）来区分字面量的进制与类型的方式。内容适用于 C89/C99/C11/C23 标准。

3.3.1 整数：通过前缀区分进制

整数字面量可使用前缀（`prefix`）确定进制，如 表 1 所示。

表 1: 整数前缀及其含义

前缀	进制	示例
(无前缀)	十进制	`123`
`0`	八进制	`0123`（十进制 83）
`0x` / `0X`	十六进制	`0x123`（十进制 291）
`0b` / `0B`（C23）	二进制	`0b1010`（十进制 10）

注意：

- 以 `0` 开头的整数默认按八进制解析（如 `0123` $\neq$ `123`）。所以在 C 语言中避免使用前导零，以免引起歧义。这算是一个非常经典的错误原因呢。
- C89/C99/C11 没有二进制前缀，直到 C23 才正式加入 `0b` 作为二进制前缀，有些编译器（如 GCC）较早支持。
- C 语言中只有十六进制浮点数可以带前缀，其他进制的浮点数一律不允许带前缀。（具体见后文）。

3.3.2 整数：通过后缀区分类型

整数字面量的类型可由后缀（Suffix）确定。表 2 汇总了常见后缀。

表 2: 整数类型后缀

后缀	类型	示例	中文含义
(无后缀)	int	123	整型（默认类型）
u / U	unsigned int	123U	无符号整型
l / L	long int	123L	长整型
ul / UL	unsigned long	123UL, 123LU	无符号长整型
ll / LL	long long int	123LL	长长整型
组合使用	unsigned long long 等	123ULL, 123LLU	无符号长长整型

如代码 12所示，不同的类型其实意味着**存储空间**和**取值范围**的不同。所以当你由于不同的应用场景需要表示不同范围的整数时，可以使用相应的后缀来指定类型。如果这个范围不匹配，编译器会报错或发出警告。

代码 12: 后缀示例

```

123         // int（默认类型）              -2147483648 ～ 2147483647（32位系统）
123U        // unsigned int                0 ～ 4294967295

123L        // long int                    -2147483648 ～ 2147483647（32位系统）
123LU       // unsigned long int           0 ～ 4294967295（32位系统）

123LL       // long long int           -9223372036854775808 ～ 9223372036854775807
123LLU      // unsigned long long int      0 ～ 18446744073709551615


//以上给出的数据范围基于常见的32位和64位系统,
//实际范围基于编译器、CPU 架构和操作系统选择的数据模型而异。
```

注意

作为整数字面量后缀：llu 和 ull 完全等价，可以随便使用，没有任何问题。但在 printf 格式字符串中：必须用%llu，不能用%ull。所以我推荐大家统一记忆 llu。

3.3.3 浮点数：无前缀（Prefix 不可用）

浮点数**不能使用前缀**来表示进制，不能像整数那样写成八进制、二进制形式：

- `012.3` （非法，浮点不能为八进制）
- `0b1.01` （非法，C 不支持二进制小数）
- `0x3.14` （非法的普通十六进制形式）

因此：浮点字面量不允许使用任何前缀。但凡事总有例外，在 C 语言中，十六进制浮点数就能表示，但其他进制的浮点数一律不允许带前缀。也就是说

C 语言中浮点数只能有两种存在方式，第一种是默认的十进制，第二种是十六进制。

使用前缀表示十六进制浮点常量（C99 及以后）

最开始，浮点字面量**不能**写成 `012.3`（八进制）、`0b1.01`（二进制）、`0x3.14`（十六进制）等形式。

但是，从 C99 标准开始，C 语言**新增**了一类字面量：十六进制浮点常量（hexadecimal floating constant）。这一类浮点常量是**允许**使用 `0x` 或 `0X` 前缀的，只是语法与整数常量不同，因此很多教材在入门阶段会直接略去不讲。

(1) 十六进制浮点常量的一般形式

十六进制浮点常量的大致结构可以写成:

$$\underbrace{\texttt{0x} \ / \ \texttt{0X}}_{\text{十六进制前缀}} \quad \underbrace{\text{十六进制数字（可带小数点）}}_{\text{significand}} \quad \underbrace{\texttt{p} \ / \ \texttt{P} + \text{十进制指数}}_{\text{以 2 为底的指数}} \quad \underbrace{\text{浮点后缀}(\texttt{f},\texttt{F},\texttt{l},\texttt{L})}_{\text{类型说明}}$$

更口语一点的记忆方式：

`0x 十六进制小数部分 p 十进制整数指数 [可选后缀]`

其数值等价于：

$$\pm(\text{十六进制有效数字}) \times 2^{\text{指数}}$$

(2) 若干合法示例

表 3: 十六进制浮点常量示例及其对应值

字面量	对应的十进制值（大致含义）
`0x1.2p3`	$(1+2/16)\times 2^3 = 1.125\times 8 = 9.0$
`0x9A.8p-1`	$(154+8/16)\times 2^{-1} = 154.5/2 = 77.25$
`0x1.921fb6p1f`	≈ 3.1415927 （`float` π）

下面用代码形式再演示一次（需要支持 C99 及以上标准的编译器，例如 `gcc -std=c11`）：

代码 13: 十六进制浮点常量示例

```c
#include <stdio.h>

int main(void) {
    double b = 0x1.8p1;   // 3.0
    double c = 0x1.fp2;   // 7.75
    float  d = 0x.ap-3f;  // 0.078125f

    printf("b = %f\n", b);
    printf("c = %f\n", c);
    printf("d = %f\n", d);
    return 0;
}
```

(3) 为什么 `0x3.14` 仍然是“非法写法”？

在上一小节我们给出例子：`0x3.14` “非法的普通十六进制形式”。这个结论在 C99 之后**依然成立**，原因是：

- 对整数常量来说，形如 `0x3.14` 带小数点，本来就不是合法的“整数十六进制字面量”；
- 对十六进制浮点常量来说，`0x3.14` 也**不完整**，因为缺少必须的 `p` 或 `P` 指数部分。

也就是说，`0x3.14` 只是一个“十六进制有效数字”，**不是**完整的“十六进制浮点常量”。如果把它补写完整，例如：

`0x3.14p0`, `0x3.14p+0`, `0x3.14p4`

这些才是标准允许的、真正合法的十六进制浮点字面量。

(4) 对“浮点数不能用前缀”说法的精确修正

因此，可以更严谨地改写原结论：

- **浮点常量**（如 `3.14`、`1e-3`）不能使用 `0`、`0x`、`0b` 等进制前缀；
- 从 C99 起，C 语言单独引入了**十六进制浮点常量**，其写法必须带有 `0x`/`0X` 前缀，并以 `p`/`P` 引出二进制指数。

在入门阶段，为了避免一次性给出过多语法细节，很多教材会只讲“无前缀的十进制浮点常量”，并用“浮点数不能用前缀”这样简化的说法帮助初学者形成直觉。而在更高阶段学习标准（C99 及以后）时，就需要把十六进制浮点常量这一部分补充进来。

3.4 浮点数的后缀

浮点字面量只能用后缀确定类型，如表 4 所示。

表 4: 浮点数字面量类型与后缀

后缀	类型	示例	字节数（典型）	精度（有效数字）	中文含义
(无后缀)	double	`3.14`	8 字节	约 15-16 位	双精度浮点数 (默认)
`f` / `F`	float	`3.14f`	4 字节	约 6-7 位	单精度浮点数
`l` / `L`	long double	`3.14L`	10-16 字节	约 18-33 位	扩展精度浮点数

示例：

代码 14: 浮点数后缀示例

```c
#include <stdio.h>

int main(void) {
3.14f         // float
2.718L        // long double
1e-3          // double（默认类型）
}
```

3.5 总结

- 整数字面量：**前缀区分进制，后缀区分类型**。
- 浮点字面量：**无前缀，只能用后缀区分类型**。
- C99 起提供十六进制浮点字面量（如 `0x1.2p3`）。

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：

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

③ 得到补码：

在反码的基础上加 1：

$$\texttt{11010101} + 1 = \texttt{11010110}$$

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 $b_{31}b_{30}\ldots b_0$。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
`short`	2	-2^{15}	$2^{15}-1$
`int`	4	-2^{31}	$2^{31}-1$
`long`	4 或 8	-2^{31} 或 -2^{63}	$2^{31}-1$ 或 $2^{63}-1$
`long long`	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。

下面列出了常见的有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
`unsigned short`	2	$[0, 2^{16}-1]$
`unsigned int`	4	$[0, 2^{32}-1]$
`unsigned long`	4 或 8	$[0, 2^{32}-1]$ 或 $[0, 2^{64}-1]$
`unsigned long long`	8	$[0, 2^{64}-1]$

4.3 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。

4.3.1 基本概念与名词解释

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。
- **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：

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

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：

$$\begin{aligned} e = 0 &\Rightarrow E = 127, \\ e = -1 &\Rightarrow E = 126, \\ e = -4 &\Rightarrow E = 123, \\ e = 2 &\Rightarrow E = 129. \end{aligned}$$

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

在讨论“规格化数（Normalized Number）”或“隐含位（Hidden Bit）”之前，首先需要了解 IEEE 754 浮点数的基本组成方式。该标准将一个二进制浮点数拆分为三个部分：符号位（sign）、指数域（exponent）与尾数域（fraction 或 mantissa），其结构如下：

$$\underbrace{s}_{\text{符号位}} \quad \underbrace{E}_{\text{指数域}} \quad \underbrace{f}_{\text{尾数域}}$$

其中：

- 符号位 s 表示正负号；
- 指数域 E 用于表示 2^e 中的指数部分；
- **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：

$$1 \text{ bit sign} + 8 \text{ bits exponent} + 23 \text{ bits fraction}.$$

尾数域由 23 位二进制小数组成，可写为：

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

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：

$$f = 0.b_1b_2 \dots b_{23}, \qquad b_i \in \{0,1\},$$

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：

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

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 $1000\dots0$。
- 数值 $1.25 = (1.01)_2$，尾数部分为 $f = 0.01\_2$，尾数域编码为 $0100\dots0$。

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 $b_1 b_2 \ldots b_{23}$，其数值可理解为

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

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）。

4.3.2 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：

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

4.3.3 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。

4.4 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

$$f = \texttt{01000000000000000000000}$$

最终 IEEE 754 表示

$$0 \,|\, 01111100 \,|\, 01000000000000000000000$$

十六进制为：

$$\texttt{0x3E200000}$$

4.5 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

$$f = \texttt{11110000000000000000000}$$

最终 IEEE 754 表示

$$1 \,|\, 10000001 \,|\, 11110000000000000000000$$

十六进制为：

`0xC0F80000`

4.6 示例 3：`0.1`（无法精确表示）

十进制转二进制

$$0.1 = 0.00011001100110011\ldots_2 \quad \text{(无限循环)}$$

科学计数法

$$0.000110011\ldots_2 = 1.100110011\ldots_2 \times 2^{-4}$$

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）

$$f=\texttt{10011001100110011001101}$$

最终 IEEE 754 表示

$$0\,|\,01111011\,|\,10011001100110011001101$$

十六进制为：

$$\texttt{0x3DCCCCCD}$$

4.7 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 $b_1b_2\ldots$ 代表 $1+b_12^{-1}+b_22^{-2}+\ldots$）；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
`float`	4	$\pm 3.4 \times 10^{38}$	7–8 位十进制有效位
`double`	8	$\pm 1.7 \times 10^{308}$	15–16 位
`long double`	16（实现相关）	$10^{\pm 4932}$ 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。

4.8 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。

4.9 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。

4.10 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。

4.11 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。

4.12 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。

5 为什么我们需要数据类型

5.1 计算机与数学

在数学世界里，我们习惯于假定：

- 整数可以无限大、无限小；运算可以无限次进行，而不会" 装不下"。
- 小数可以具有无限精度，如 π；
- 数值的运算是绝对准确的，例如：78 与 97 之和为 175，1/3 的值是 0.33333333⋯⋯（循环小数）。

数学是一门研究抽象问题的学科，数和数的运算都是抽象的。而在计算机中，数据是存放在存储单元中的，它是具体存在的。而且，存储单元是由有限的字节构成的，每一个存储单元中存放数据的范围是有限的，不可能存放“无穷大”的数，也不能存放循环小数。

所以在真实的计算机系统中，造成这些都不成立的根本原因是**硬件资源是有限的**：

- 内存容量有限，只能分配有限数量的比特来表示一个值；
- 寄存器宽度有限，例如常见 CPU 的通用寄存器是 32 位或 64 位；
- 加法器、乘法器等算术单元只能处理固定位宽的输入；

例如用 C 程序计算和输出 1/3：程序 `printf("%f" , 1.0/3.0);` 得到的结果是 0.333333，只能得到 6 位小数，而不是无穷位的小数。

因此，如果想要" 无限精度" 整数或小数，就只能通过**软件模拟**来实现（例如 Python 的大整数、GMP 等库），而非依赖硬件原生支持。

从这个角度看，**数据类型本质上就是在有限资源约束下，对数学世界做出的一个妥协方案**：不同类型对应不同的位宽、不同的范围、不同的精度以及不同的运算规则。数据类型不是单纯用来帮助程序员分类数据的标签，而是**对内存中比特模式的一种解释方式**。

5.2 数据类型的本质：对比特模式的解释

计算机的底层存储介质只认识 0 和 1。数据是具体存在的，存放在存储单元中。例如，下面这 4 个字节的二进制内容（为了阅读方便，用空格分组）：

```
0100 0001 0100 0010 0100 0011 0100 0100
```

在不同语境下，它可能表示完全不同的东西：

表 9: 同一比特模式在不同类型解释下的含义示例

解释方式	含义
char[4]	字符串 "ABCD"
int32_t	一个 32 位有符号整数（某个十进制值）
uint32_t	一个 32 位无符号整数（更大的十进制值）
float	一个 IEEE 754 单精度浮点数

需要注意的是：**比特本身并没有变化，变化的只是”如何解释这些比特”的规则，**也就是数据类型。

这意味着：

- **没有类型就没有意义**：底层的 0 和 1 无法自动告诉计算机”我是哪种数据”；
- **编译器必须依赖类型**来决定存储方式、对齐方式，以及在表达式中使用什么样的运算规则。

换句话说，类型是 CPU 与编译器之间关于”这些比特该怎么看、怎么算”的协议。因此，在计算机中，数据必须有类型，不同类型的数据在存储单元中所占的字节数不同，表示的数据范围也不同，数据的运算规则也不同。

理解数据类型的本质，是写好 C 语言以及其他底层语言的根基。

5.3 数据类型决定存储格式与运算规则

数据类型并不是一个"仅供人参考的标签"，它直接决定两件事，存储格式和运算规则。

5.3.1 存储格式（Representation）

类型会决定：

- 占用的字节数（例如典型平台上 `int` 为 4 字节、`char` 为 1 字节）；
- 在内存中的实际布局，以及是否需要对齐；
- 是否有符号（`signed` vs `unsigned`）；
- 对于浮点数，如何拆分为符号位、阶码和尾数（IEEE 754）。

5.3.2 运算规则（Operations）

类型还会决定表达式计算时的具体运算规则：

- 使用整数加法器还是浮点运算单元（FPU）；
- 是按补码规则进行加减，还是作为无符号数做模 2^n 运算；
- 运算结果是否可能溢出，溢出之后会发生什么；
- 表达式中不同类型之间如何进行整型提升和通常算术转换。

例如下面这个 C 代码片段：

```c
char a = 100;
char b = 100;
char c = a + b;

printf("%c\n", c);       //结果为
printf("%d",c);          //结果为-56
```

在表达式 `a + b` 中，`a` 和 `b` 会先被整型提升为 `int`，实际以 `int` 进行运算。但最终赋值给 `char c` 时，会发生截断：只保留结果的低 8 位，从而可能得到和数学直觉完全不同的值。

因此，**理解类型对于"值是如何被存储和计算的"至关重要**。

5.4 有限位宽带来的问题：溢出、截断与精度损失

由于类型对应的位宽是有限的，必然会带来一系列问题，最典型的包括：

5.4.1 整数溢出（Integer Overflow）

计算机中的整数类型具有固定的位数，例如：

表 10: 常见整数类型的位数与表示范围

类型	位数（一般情况）	可表示范围
`short`	16	$-32768 \sim +32767$
`int`	32	$-2^{31} \sim 2^{31}-1$
`unsigned int`	32	$0 \sim 2^{32}-1$
`long`	32/64	$-2^{31} \sim 2^{31}-1$ (或 $-2^{63} \sim 2^{63}-1$)

若表达式的结果超过了该类型所能表示的范围，就会发生 **整数溢出（Integer Overflow）**。

(1) 无符号整数溢出：定义良好（Well-defined）

以 `unsigned int` 为例，它只能存储 $0 \sim 4294967295$（即 $0 \sim 2^{32}-1$）。当执行：

```
unsigned int x = 4294967295;
x = x + 1;
```

其二进制加法为：

$$11111111111111111111111111111111_2 + 1 = 1\,00000000000000000000000000000000_2$$

由于只有 32 位，最高位溢出被丢弃, 剩下：

$$00000000000000000000000000000000_2 = 0$$

因此：

$$4294967295 + 1 \equiv 0 \pmod{2^{32}}$$

也就是说执行下面的代码，得到的结果为 0。

```
unsigned int x = 4294967295;
printf("%u\n", x + 1);
```

(2) 有符号整数溢出

以 int 为例，在 32 位系统中其范围为 $-2^{31} \sim 2^{31}-1$（即 $-2147483648 \sim 2147483647$）。当执行：

```c
int x = 2147483647;
printf("%d\n", x + 1);
```

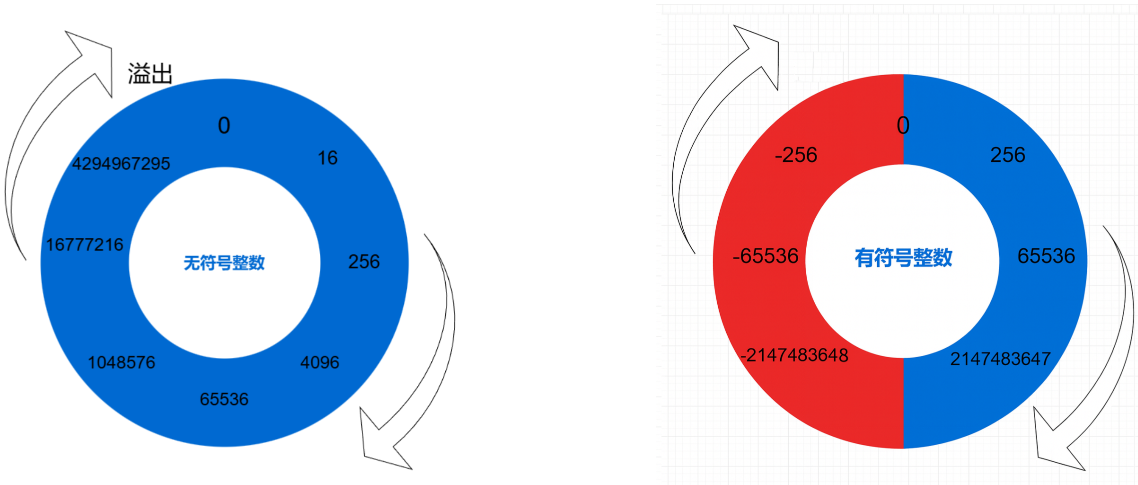
其二进制补码加法为：

$$01111111111111111111111111111111_2 + 1 = 10000000000000000000000000000000_2$$

该值按补码解释为 -2147483648。
这也就是为什么上面的代码执行之后的结果为-2147483648

(3) 图解整数溢出原理

如图 2 所示，下图展示了整数溢出的原理。当无符号整数超过其最大值时，会回绕到 0；而有符号整数溢出时，则会从最大正数翻转为最小负数，这与补码的存储方式直接相关。



(a) 无符号整数溢出示意图　(b) 有符号整数溢出示意图

图 2: 整数溢出示意图

(3) 无符号溢出和有符号溢出的区别

在讨论溢出行为之前，可以先给出一个简单而关键的论断：

> **无符号整数的溢出是标准规定的确定行为；有符号整数的溢出是标准未规定的未定义行为。所以不能完全保证符合预期输出**

在 C 语言中，有符号整数与无符号整数在发生溢出时具有本质上的区别。首先，无符号整数（`unsigned`）的溢出属于标准规定的行为。C 标准明确要求无符号整数采用模 2^n 算术，其中 n 为类型的比特宽度。也就是说，无符号整数的取值范围始终保持在 $[0, 2^n - 1]$ 之间，并在超过最大值时按模 2^n 回绕。例如，一个 32 位的 `unsigned int` 的最大值为 `0xFFFFFFFF`；当其再加 1 时，其结果将变为 `0x00000000`。这种行为在不同的平台、编译器和优化等级下均完全一致，因此被称为定义良好的行为（well-defined behavior）。

无符号溢出的“环结构”

以 8 位无符号整数为例：

$$0 \to 1 \to \cdots \to 254 \to 255 \to 0 \to \cdots$$

整数在 $0 \sim 255$ 之间构成一个模 256 的环。

与此不同的是，有符号整数（如 `int`、`long` 等）的溢出属于未定义行为（Undefined Behavior, UB）。所谓未定义行为，并不是指结果随机，而是 C 标准根本不对溢出之后的结果给出任何保证。编译器在遇到可能导致有符号整数溢出的代码时有完全的自由：它可以产生某个具体值，也可以忽略该表达式，甚至可以在优化阶段假定溢出永远不会发生，从而删除某些看似可执行的代码路径。

出现这种设计的原因，一方面是因为历史上不同处理器使用不同的整数表示方式（如补码、反码和符号-幅度），导致其溢出行为并不一致；另一方面，将其定义为未定义行为可以让编译器进行更激进的优化，例如认为 $i + 1 > i$ 永远成立。

综上所述，无符号整数的溢出是可预测且具有标准保证的，而有符号整数的溢出则是不受标准约束、具有潜在不可预测性的行为。在实际编程中，应避免任何可能导致有符号整数溢出的情况，否则程序的语义可能会偏离标准规定，并引入难以察觉的错误。

简要比较

类型	溢出行为	标准规定	后果
`unsigned`	模 2^n 回绕	定义良好	可预测、可依赖
`signed`	无标准规定	未定义行为（UB）	不可预测、不可依赖

5.4.2 浮点精度损失（Floating-Point Precision Loss）

浮点数基于 IEEE 754 标准，在有限的位宽内同时兼顾数值范围与精度。以单精度浮点数为例，它由 1 位符号、8 位阶码和 23 位尾数组成，每一个数值都被解释为

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

其中 `bias` 为 127。由于只有 23 个显式尾数位（再加上隐藏的 1），任何无法在 24 位二进制小数中终止的十进制小数都必须被**截断或舍入**为附近的某个值。

0.1 到底存储了什么？ 以 0.1_{10} 为例，它的二进制展开为：

$$0.1_{10} = 0.0001100110011\ldots_2$$

该序列会无限循环 `0011`，无法被有限位宽完整表示。单精度浮点数只能保留前 23 位并进行舍入，所以 `float` 版本的 `0.1f` 实际存储的是十六进制 `0x3dcccccd`，对应的数学值约为：

$$0.100000001490116119384765625$$

同理 `0.2f` 变为 `0x3e4ccccd`。这些值在写入内存的一瞬间就已经带上误差，后续的所有运算都只是在误差基础上继续推导。

为什么 $0.1 + 0.2 \neq 0.3$？

```c
float a = 0.1f;
float b = 0.2f;
printf("%.10f\n", a + b);        // 0.3000000119
printf("%a\n", a + b);           // 0x1.333336p-2
```

编译器会把源代码常量转换为最近似的浮点值。`a` 和 `b` 相加后，结果需要再次在 24 位有效数字内舍入，于是得到 `0x1.333336p-2`（略大于 0.3）。这不是 CPU 算错，而是数学上的“真值”在映射到有限精度时被迫变化了。

误差传播的典型场景

- **累计误差**：将 `0.1f` 循环累加 100 次，理论值为 10，但由于每次都引入约 1e-8 的误差，最终会得到 9.999999 或 10.000001。
- **灾难性消除（catastrophic cancellation）**：当两个非常接近的浮点数相减时，阶码对齐会导致有效位大量被抵消，只剩下噪声。例如 `double x = 1e8; double y = x + 1;`，`y - x` 得到的仍是 0。
- **比较陷阱**：直接做 `if (a == b)` 在浮点世界几乎总是错误的，因为“相等”只有在比特级完全相同时才成立。更合理的做法是比较差值是否小于某个阈值。

工程实践中的常见对策

- **规避精度要求**：涉及货币或需要十进制精度的场景使用整数（以分为单位）或专用的 decimal 库。
- **控制运算顺序**：先累加小数再加大数可以减少阶码对齐造成的有效位丢失；在求和、积分类问题中可以使用 Kahan Summation 等补偿算法。
- **设置容差**：比较时使用 `fabs(a - b) < eps` 的形式，并根据量级选择合适的绝对或相对误差。
- **了解舍入模式**：IEEE 754 默认为“舍入到最接近，遇到中点取偶”（Round to nearest, ties to even）；必要时可以通过 `fesetround` 更改舍入策略。

只有真正理解浮点数是“有限精度的有理数近似”，才能在算法、图形、数值计算等场景中写出可靠的 C 代码。

5.4.3 截断（Truncation）

当将一个“位宽较大的类型”赋值给“位宽较小的类型”时，会发生截断：直接丢弃高位，只保留低位。例如：

```c
int x = 1000;
char y = x;   // 截断为低 8 位
```

若不理解这个过程，很容易在类型转换或结构体内存布局中引入难以察觉的 Bug。

5.4.4 混合类型与隐式转换

再看一个常见的“惊喜”示例：

```c
unsigned int a = 1;
int b = -2;
printf("%u\n", a + b);
```

在这个表达式中，`a` 是无符号整数、`b` 是有符号整数。根据 C 语言的*通常算术转换*规则，`b` 会被转换为无符号类型，然后再参与运算，最终得到一个非常“反直觉”的巨大无符号数。

这些细节都与“数据类型如何被解释和提升”紧密关联。

5.5 小结

本节可以归纳为以下几点：

- 在计算机里，**数据类型的本质是对内存中比特模式的一种解释方式**；

- 类型决定了值的存储格式（占多少字节、如何布局、是否有符号）以及运算规则（使用哪种算术单元、是否回绕、如何提升和转换）；
- 硬件资源有限，决定了我们不可能进行“无限精度的数学运算”，从而不可避免地引入溢出、截断和精度损失等问题；
- 理解数据类型，是理解后续补码、字节序、整型提升、浮点误差等全部内容的基础。

从工程实践的角度看，**类型不是语法装饰，而是程序正确性和可移植性的基石**。只有真正搞懂“类型到底在替我们做什么”，才能在 C 语言中写出既高效又可靠的代码。

因此，在计算机中，数据必须有类型，不同类型的数据在存储单元中所占的字节数不同，表示的数据范围也不同，数据的运算规则也不同。在展开讨论 C 语言的数据类型之前，我们先深入了解一下整数与浮点数的存储原理。

5.6 整数的存储与表示

加法和减法是计算机中最基本的运算，计算机时时刻刻都离不开它们，所以它们由硬件直接支持。与此同时，早期计算机电路为了提高加减法的运算效率，硬件电路需要设计得尽量简单，可是这样一来减法电路就势必要被阉割掉，因为减法电路的设计要比加法复杂得多。因此困扰早期计算机设计者的一个重要问题开始浮现：

如何用加法电路来实现减法运算？

要想要实现减法效果，实际上就是在思考怎样才能得到数字 0。如果说 a+b=0，那么 b 就等于-a。此时的 b 也就是我们想要的减法效果。显然，从纯数学的角度来说，这个问题是无解的，因为不可能有两个正数相加等于 0。但是不要忘了这里不是纯粹的数学世界，我们发现计算机的物理电路还有一个重要的特性：**加法器的自然进位溢出**。换而言之，我们可以通过一直加，直到数字溢出回到 0，从而实现减法的效果。

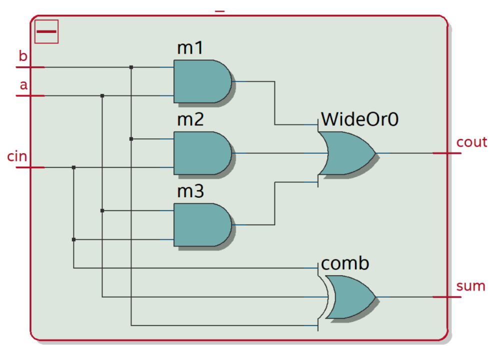


图 3: 加法器

这就好比一个时钟，如图 4所示，时钟的时针此时指向 5 点钟，如果我们想让这个时钟归 0（也就是指向 12 点钟）。那么一共有两种办法，一种是逆时针转动 5 个小时（这其实就是对应了负数），另一种是顺时针转动 7 个小时（而这就是加法溢出）。

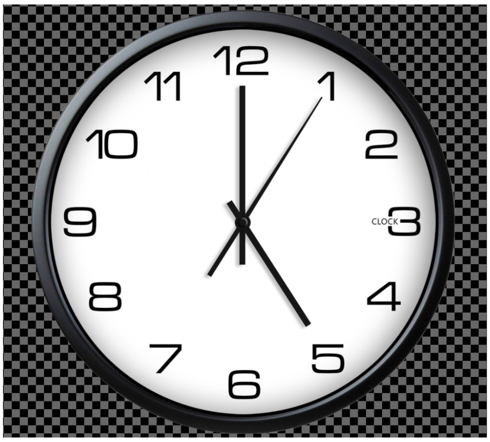


图 4: 时钟

原码反码补码

了解完以上原理，我们就可以引出整数的存储表示方法了。在计算机中，整数使用**补码**来表示。补码的出现，彻底解决了计算机中加减法运算的问题。从此计算机不再关注数字的正负，而只需要关注数字的二进制位即可。这样一来，计算机的加法器就可以直接进行加减法运算。

① 原码：最高位为符号位，0 表示正数，1 表示负数。其余位表示数值的大小。

② 反码：正数的反码与原码相同，负数的反码是将原码中除符号位外，其他位取反。

③ 补码：正数的补码与原码相同，负数的补码是将反码加 1。

图 5: 原码、反码与补码的表示

C 语言的数据类型系统为所有数据（无论变量或常量）提供统一的分类标准，包括：

整型	说明	长度（字节）	取值范围
short	短整型	2	[-32768, +32767]
int	整形	4	[-2147483648, 2147483647]
long	长整型	4 或 8	4 个字节：[-2147483648, 2147483647]
			8个字节：[-2^63 , +2^63-1]
long long	超长整形	8	[-2^63 , +2^63-1]
unsigned short	无符号短整型	2	[0, 65535]
unsigned int	无符号整形	4	[0, 4294967295]
unsigned long	无符号长整型	4 或 8	4 个字节：[0, 4294967295]
			8个字节：[0, 2^64 - 1]
unsigned long long	无符号超长整形	8	[0, 2^64 - 1]

浮点型	说明	长度（字节）	取值范围
float	单精度浮点型	4	-3.4*10^-38 ~ +3.4*10^38 （精度：7 ~ 8 位有效数字）
double	双精度浮点型	8	-1.7*10^-308 ~ +1.7*10^308 （精度：15 ~ 16位有效数字）
long double	长双精度浮点型	8 或 16	16位：-1.2*10^-4932 ~ 1.2*10^4932 （精度：18 ~ 19位有效数字）

字符型	说明	长度（字节）	取值范围
char	字符型	1	-128 ~ 127
unsigned char	无符号字符型	1	0 ~ 255

布尔型	说明	长度（字节）	取值范围
_BOOL	布尔型	1	1 或者 0
bool	布尔型	1	true 或 false

图 6: C 语言数据类型（典型字节数）