

C 语言入门教程

Grant Lee

2025 年 11 月 13 日

目录

| | | |
|----------|---------------------------|-----------|
| 1 | C 语言程序的基本结构 | 3 |
| 1.1 | 最小示例 HelloWorld | 3 |
| 1.2 | #include 与头文件 | 3 |
| 1.3 | main 函数（程序入口） | 3 |
| 1.4 | 语句块与分号 | 4 |
| 1.5 | printf 与常用占位符 | 4 |
| 1.6 | 从源代码到可执行文件：将自然语言翻译为机器语言 | 4 |
| 2 | C 语言的变量与常量 | 7 |
| 2.1 | 变量与常量的基本定义 | 7 |
| 2.2 | 命名规则与规范 | 11 |
| 2.3 | 作用域与生命周期 | 11 |
| 3 | 数据类型 | 13 |
| 3.1 | 整数的存储与表示 | 13 |
| 4 | 从字面量（Literal）到数据类型 | 16 |
| 4.1 | 定义与概念 | 16 |
| 4.2 | 字面量与常量的区别 | 16 |
| 4.3 | 延伸说明 | 17 |
| 4.4 | 总结 | 17 |

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

```
1
2  #include <stdio.h>           // 1) 预处理指令: 包含头文件
3
4  int main(void) {             // 2) 主函数: 程序从这里开始执行
5      printf("Hello, World!\n"); // 3) 调用库函数进行输出
6      return 0;                // 4) 返回状态码 0: 正常结束
7  }
```

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

```
1 int main(void)
2 // 或
3 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

```
1 #include <stdio.h>
2 int main(void) {
3     int a = 42; double x = 3.14159; char c = 'A';
4     printf("a=%d, x=%.2f, c=%c\n", a, x, c); // 输出 a=42, x=3.14, c=A
5     return 0;
6 }
```

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 \Rightarrow 预处理（展开 `#include/#define`） \Rightarrow 编译（生成 `.o/.obj`） \Rightarrow 链接（合并库与目标文件） \Rightarrow 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

```
1 int a = 10;  
2 float b = 3.14;  
3 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

```
1 #include <stdio.h>
2
3 #define PI 3.14          //经过指定后，本文件中所有PI都会被替换为3.14。
4 #define AREA(r) (PI * (r) * (r))    //定义函数宏，用于计算圆面积。
5
6 int main() {
7     float r = 2.0;
8     printf("Area = %.2f\n", AREA(r));
9     return 0;
10 }    //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) 常量变量 `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。

常量变量具有类型信息，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

```
1 #include <stdio.h>
2
3 const int    MAX_CONN = 1024;
4 const float PI      = 3.14159f;
5
6 int main(void) {
7     MAX_CONN = 2048; // 编译报错：只读对象不可修改
8     printf("%f\n", PI);
9     return 0;
10 }
```


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。

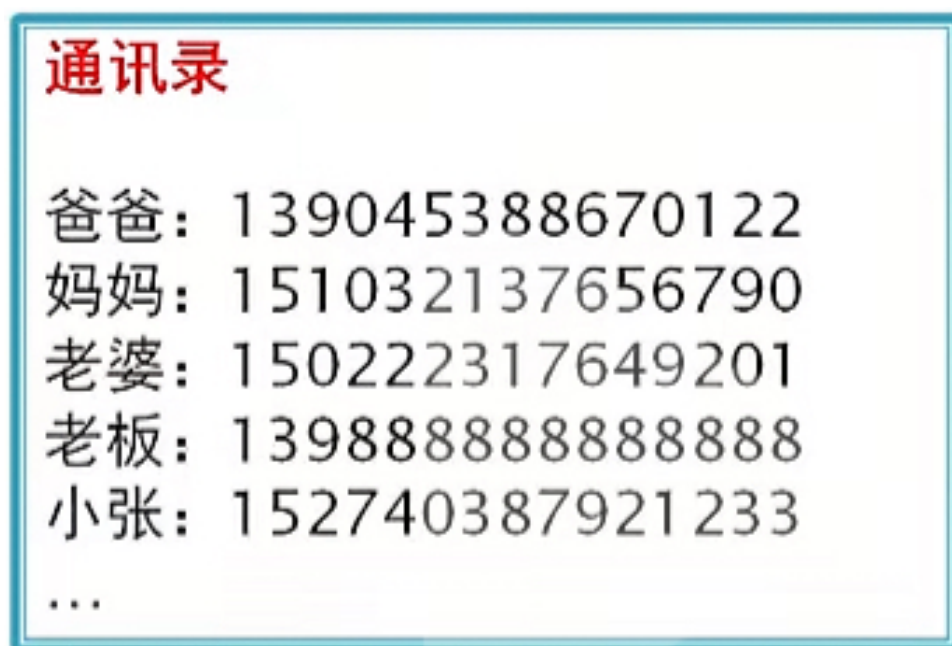


图 1: 生活中的枚举数据: 通讯录

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

```
1 enum Color { RED, GREEN, BLUE };
2 // 数值若不自定义, 则依次为 0,1,2
3 enum Weekday{ MON=1, TUE, WED, THU, FRI, SAT, SUN };
4 //若自定义了起始值, 则后续依次递增1
5 enum State { ST_INIT=-5, ST_RUN=8, ST_ERR=9, ST_STOP=100 };
6 //也可自定义每个枚举值
7
8 //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

```
1 #include <stdio.h>
2
3 enum Color { RED, GREEN, BLUE }x; // 定义枚举类型 Color 并声明变量 x
4
5
6 enum Weekday { MON, TUE, WED, THU, FRI, SAT, SUN };
7 enum Weekday this_month; // 声明枚举类型 Weekday 变量 this_month
8 //这与上面Color的声明方式是等价的。
9
10 int main(void) {
11     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
12     this_month = TUE; // 给枚举变量 this_month 赋值
13     printf("Color: %d, Weekday: %d\n", x, this_month);
14
15     return 0;
16 }
```

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

```
1 void foo() {  
2     int x = 0;           // 局部变量  
3     static int y = 0;    // 静态局部变量  
4     x++;  
5     y++;  
6     printf("x=%d, y=%d\n", x, y);  
7 }
```

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 数据类型

为什么在计算机运算时要指定数据的类型呢？在数学中，数值是不分类型的，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 $0.33333333\cdots$ （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。

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，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。

3.1 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。

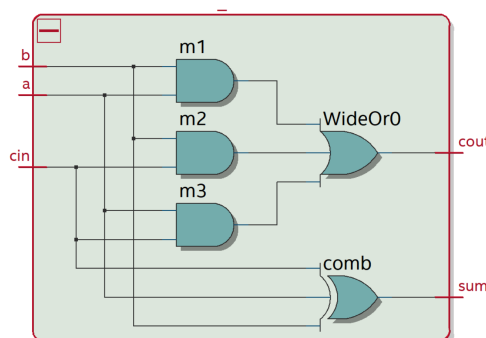


图 2：加法器

这就好比一个时钟，如图 3 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）。

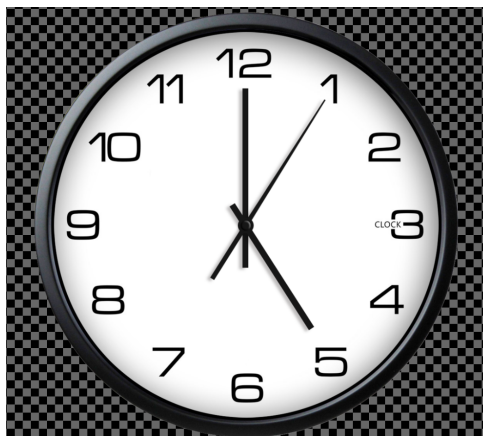


图 3: 时钟

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 4: 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 | $-3.4 \times 10^{-38} \sim +3.4 \times 10^{38}$ (精度: 7 ~ 8 位有效数字) |
| double | 双精度浮点型 | 8 | $-1.7 \times 10^{-308} \sim +1.7 \times 10^{308}$ (精度: 15 ~ 16 位有效数字) |
| long double | 长双精度浮点型 | 8 或 16 | 16 位: $-1.2 \times 10^{-4932} \sim 1.2 \times 10^{4932}$ (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 | $-128 \sim 127$ |
| unsigned char | 无符号字符型 | 1 | $0 \sim 255$ |
| 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 5: C 语言数据类型（典型字节数）

4 从字面量 (Literal) 到数据类型

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

```
1 #include <stdio.h>
2
3 int main(void) {
4     int x = 42;           // 42 是整型字面量
5     char c = 'A';         // 'A' 是字符字面量
6     double pi = 3.1416;   // 3.1416 是浮点字面量
7     printf("%s\n", "Hi"); // "Hi" 是字符串字面量
8     return 0;
9 }
```

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。

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、十六进制也是经常使用，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

4.1 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 |

4.2 字面量与常量的区别

| 名称 | 是否存在于语法中 | 是否有标识符 | 示例 | 说明 |
|------------------------|----------|--------|--------------------------------|----------|
| 字面量 (<i>literal</i>) | 是 | 否 | 3.14, 'A', "abc" | 直接出现在代码中 |
| 常量 (<i>constant</i>) | 是 | 是 | <code>const int a = 10;</code> | 存储在命名空间中 |

因此, 10 是一个字面量, 而在 `const int a = 10;` 中的 a 是一个常量。

4.3 延伸说明

- C 语言仅定义了四类基本字面量: 整型、浮点型、字符和字符串。
- C++ 对字面量进行了扩展, 例如用户自定义字面量 `42_km`。
- 在 C17 标准第 §6.4 节中, “literal” 一词被正式用于描述这些常量。

4.4 总结

在 C 语言标准中, “字面量” (literal) 是正式术语, 表示直接出现在源代码中的常量值, 如 10、'A'、"abc" 等。它们是语法结构的一部分, 而非常量变量。