02｜程序基石：数据与量值是如何被组织的？

2021-12-08 于航

《深入C语言和程序运行原理》 课程介绍 >

讲述：于航

时长 17:27 大小 15.99M

你好，我是于航。

从这一讲开始，我们就进入到了“C 核心语法实现篇”的学习。在这一模块中，我们将围绕 C 语言的七大类核心语法，深入探寻隐藏在语法背后，程序代码的实际执行细节。

C 语言让我们能够用一种可移植、结构化，且具有人类可理解语义的方式，构建我们的程序。C 标准中详细描述了 C 语言在语法和语义两个层面的基本内容，但对于实现者，比如编译器来说，如何将这些语法和语义体现在具体的**机器指令**（汇编代码）上，标准并未给出详细规定。

所以问题来了：**在这层抽象背后，一个 C 程序中的各类语法结构，究竟是如何映射到机器能够识别的不同指令上的？**相信学完这一模块，你就能找到这个问题的答案，从而对程序

的运行有更细粒度的把控。

今天，我们就从最基本的数据和量值开始。相信你在第一次接触编程这个概念时就会了解到，一个完整的程序是由“算法”与“数据结构”两部分组成的。其中，算法代表程序会以怎样的具体逻辑来运行；数据结构代表程序运行过程中涉及数据的具体组织方式。而在一门编程语言中，数据便是以不同类型“量值”的形式被组织在一起，并交由算法进行处理的。所以我们可以说，**数据和量值是程序运行的基石。**

今天，我们先从日常使用 C 语言时最直观的编码方式开始，介绍 C 语言中的量值和数据。然后，由源代码的“表象”到计算机内部，我们来看看数据在计算机中存储时是如何被编码的。最后，我们再来一起看下，程序中的各类数据究竟被存放在哪里。

C 语言中的量值与数据

量值可以被粗略地分为变量（variable）与常量（constant），其中变量是指值可以在整个应用程序的生命周期中被多次改变的量；而常量则与之相反，在被定义后便无法被再次修改。作为一种高级语言，C 语言为我们提供了可用于定义常量与变量的语法。那么，首先我们就来看看不同的量值在 C 语言中是怎样体现的。

变量

C 语言为我们提供了众多的语言关键字（keyword）以用来定义相应类型的数据。比如在下面这个例子中，我们通过以下几步成功定义了多个变量：

1. 使用 `int` 等关键字，来指定数据的具体类型；
2. 为该数据设置一个名称；
3. 通过 “=” 赋值运算符为该数据设定具体的值。

复制代码

```
int x = -10;  // 定义一个整型变量；
char y = 'c';  // 定义一个字符变量；
double z = 2.0;  // 定义一个双精度浮点变量；
```

这里的变量具有三部分信息，即变量对应的名称、所表示数据的具体类型，以及当前的数据值。接下来，我们围绕着 C 变量的类型、大小及符号性三个方面来详细地看一看。

C 语言提供了众多的关键字，可用来指定变量的类型，这些类型均以字节作为单位，来表示变量可容纳数据的最大宽度。例如，char 类型的数据仅占用 1 个字节，而 long long 类型则至少占用 8 个字节。除了最常见的用于表示数值的类型外，C90 与 C99 标准还提供了 void（空类型）、_Bool（布尔型）、_Complex（复数类型）等类型关键字，以用于指定其他非数值类型。

当然，**C 语言中变量类型占用的具体字节大小，还与程序运行所在的硬件体系结构紧密相关**，这也是 C 语言与其他高级编程语言有所不同的地方。

C 语言最初被设计时，高效性就是设计者考虑的一个主要因素。因此 C 标准委员会在考虑语言设计时，会参考来自于底层硬件体系的某些因素。比如，C 标准中规定，int 类型的大小为执行环境架构体系所建议的自然大小。所谓自然大小，可以简单理解为：对于该大小的数据，硬件体系能够以最高的效率进行处理。因此，硬件体系不同，对应的自然大小便也不同，这也就意味着同一种 C 变量类型在不同硬件体系上可能会有着不同的大小。

而对于 Rust 和 Java 这些语言来说，它们的语言标准中直接规定了各类型的具体大小。编译器作为编程语言与硬件体系之间的抽象层，它可以确保上层类型在被编译到机器指令时，不会给程序的实际运行带来可观测的差异。当然，保持完全不变的类型大小的代价是一定的性能开销，只是在大多数情况下，这部分开销并不可观。

除了可以为变量指定不同的数据类型外，同大多数其他静态类型语言类似，在 C 语言中，整型变量本身还需区分它们的“符号性（signedness）”。简单来说，其实就是两种情况：若类型仅可以存放正数，则为无符号（unsigned）类型；若正负数都可以存储，则为有符号（signed）类型。

比如下面这行代码中，我们定义了一个无符号整型变量：

复制代码

```
1 unsigned int ux = 10;
```

符号性上的区别有利于程序对某些特定的场景需求进行优化。比如，在编写一个票务系统时，每张票对应的编号只可能为正整数，因此在使用 C 语言编写程序时，便可将票编号对应的变量定义为无符号类型。这样，对于同样的整数类型，由于不用存储对应的符号位，便可以存放更多的正整数，其可表示的正整数范围会更大。

常量

说完了变量在 C 语言中的体现，我们再来看看常量。在 C 语言中，通过内联方式直接写到源代码中的字面量值一般被称为“常量”。

我们在前面提到过常量的一个性质，即“它们被定义后无法被再次修改”。这也就意味着，这些常量数据无法灵活地被开发者操控，它们只能在程序最开始出现的地方发挥作用。比如在前面定义变量的一系列代码中，出现的 “-10”、“2.0” 等数字值便是常量。这些值在被拷贝并赋值给相应的变量后便结束了使命。

这个时候可能有同学想问：用 const 关键字按照与定义变量相同语法定义的量，不也是常量吗？它与字面量常量有什么区别呢？

这是一个非常棒的问题。一般来说，我们会按照下面的方式使用 const 关键字：

复制代码

```
1 const int vx = 10;
2 const int* px = &vx;
```

通常来说，在 C 语言中，使用 const 关键字修饰的变量定义语句，表示对于这些变量，我们无法在后续的程序中修改其对应或指针指向的值。因此，我们更倾向于称它们为“只读变量”，而非常量。当然，在程序的外在表现上，二者有一点是相同的：其值在第一次出现时便被确定，且无法在后续程序中被修改。

只读变量与字面量常量的一个最重要的不同点是，使用 const 修饰的只读变量不具有“常量表达式”的属性，因此无法用来表示定长数组大小，或使用在 case 语句中。常量表达式本身会在程序编译时被求值，而只读变量的值只能够在程序实际运行时才被得知。并且，编译器通常不会对只读变量进行内联处理，因此其求值不符合常量表达式的特征。

误用只读变量和常量会导致编译错误，下面这段代码展示了这类错误：

复制代码

```
1 #include <stdio.h>
2 int main(void) {
3   const int vx = 10;
4   const int vy = 10;
```

```
  int arr[vx] = {1, 2, 3};  // [错误1] 使用非常量表达式定义定长数组；
  switch(vy) {
    case vx: {  // [错误2] 非常量表达式应用于 case 语句；
      printf("Value matched!");
      break;
    }
  }
}
```

数据的存储形式

上面，我们介绍了数据在编程语言中的体现方式，这是程序员能够接触到数据的最初位置。随着源代码被编译，数据的实际使用形式开始变得不透明起来。接下来我们就一起看看，在计算机内部，数据是以怎样的形式被存放的。

对于大多数计算机而言，通常其内部会使用补码（Two’s-complement）的格式来存放有符号整数，使用直接对应的二进制位格式来存放无符号整数，使用 IEEE-754 标准编码格式来存放浮点数，也就是小数。**实际上，计算机在看待数据时，并不会区分其符号性，而符号性的差异仅体现在计算机指令操作数据时的具体使用方式上。**

在接下来的内容中，我不会过多讲解这三种编码方式的基本概念，而是会带你看看它们都有哪些值得关注的特征。因为无符号整数的存储格式较为直接和简单，因此我们主要来看下补码和 IEEE-754 这两种编码方式。

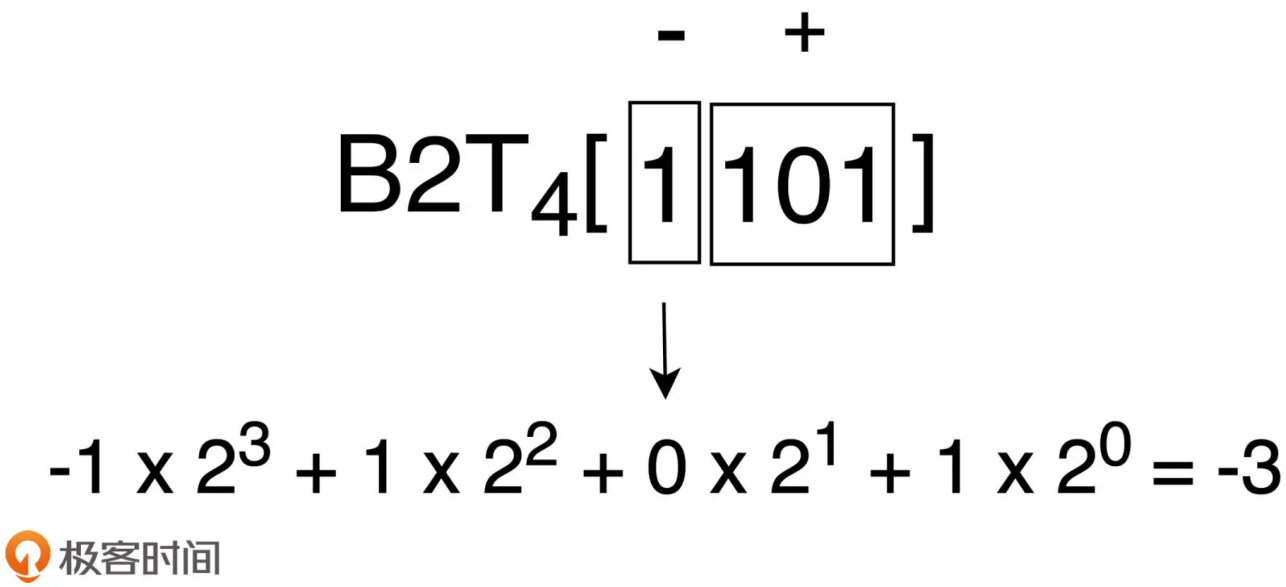
补码

我们先来看补码的几个特点。使用补码来存放有符号整数的一个优点是，CPU 在针对有符号数进行加减法计算时，不需要由于加数的符号性不同而采用多个底层加法电路，这样便可减轻电路设计的负担，另一方面也可以降低 CPU 的物理尺寸。

一个补码所表示的实际数值，由其负权重位的值与正权重位的值求和而来，其中负权重位对应于最高有效位（MSB）的符号位，即该位的二进制值在计算时按负值累加。其余各位一起对应正权重位，即这些位对应的二进制值在计算时按正值累加。那具体该怎样计算呢？我们来看一个简单的例子。

假设我们有一组补码 “1101”，那么应该如何得到它对应的有符号整数值呢？按照顺序，我们首先计算得到该补码对应负权重位的值为 -8，而正权重位的值为 5，因此该补码对应

的实际值为 -3（-8+5）。具体计算步骤可以参考下图（图中的 B2T 表示 “Binary to Two’ s-complement” ，即“二进制转补码” ）：



在计算负权重位时，其权重应取负值，正权重位取正值。通过上面的计算过程，你可以清楚地看到，对于一个 4 位补码，它可以表示的最大值与最小值分别是多少。计算最大值时，符号位置 0，其他位均置 1，可以得到能表示的最大值 7。计算最小值时，符号位置 1，其他位均置 0，可以得到最小值 -8。负整数的值可表示范围比正整数多 1 个，这也是所有有符号整数的一个重要特征。

到这里，我们了解了补码的基本计算方式。那我要向你提出一个小问题：补码的英文名称是 Two’ s-complement ，可直译为“对数字 2 的补充” ，那为什么会叫这个名字呢？你可以先停下来思考一下，然后再来看看我的理解：

首先，我们来计算一下有符号整数 3 对应的四位补码，可以得到一个二进制序列“0011” 。将该二进制序列与上述 -3 对应的二进制序列相加，通过进位可以得到序列“10000” ，该序列可以表示无符号正整数 16。

因此，我们可以得到这样一个结论：对于非负数 x，我们可以用 $2^w - x$ 来计算 $-x$ 的 w 位表示。套用在上述的例子中，可以得到“在四位补码的情况下，对于非负数 3，可以用无符号数 13 (即 $16 - 3$) 的位模式来表示有符号数 -3 的位模式”这个结论，即两者位模式相同。而补码的英文名称正是对 x、$-x$ 和 2^w 三者之间的关系进行的总结。

我们在前面提到过，计算机不会区分数据的符号性，符号性的差异仅由计算机指令如何使用数据而定。比如在 C 语言中，当对某类型变量进行强制类型转换时，其底层存储的数据并不会发生实质的变化，而仅是程序对如何解读这部分数据的方式发生了改变。比如下面这个例子：

复制代码

```
#include <stdio.h>
int main(void) {
  signed char x = -10;
  unsigned char y = (unsigned char)x;
  printf("%d\n", y);  // output: 246.
  return 0;
}
```

其中，有符号整型变量 x 会按照位模式 1111 0110 的补码形式存放有符号数 -10，而如果将该序列按照无符号整数的位模式进行解读，则可得到如程序运行输出一样的结果，即无符号整数值 246。 总之，程序在进行强制类型转换时，不会影响其底层数据的实际存储方式。

在 C 语言中，关于数据使用还有一个值得注意的问题：**变量类型的隐式转换**（Implicit Type Conversion）。C 语言作为一种弱类型语言，其一大特征就是在某些特殊情况下，变量的实际类型会发生隐式转换。

在下面这个例子中，定义的两个变量 x 与 y 分别为有符号整数和无符号整数，且变量 x 的值明显小于变量 y，但程序在实际运行时却会进入到 x >= y 的分支中，这就是因为发生了变量类型的隐式转换。

复制代码

```
#include <stdio.h>
int main(void) {
  int x = -10;
  unsigned int y = 1;
  if (x < y) {
    printf("x is smaller than y.");
  } else {
    printf("x is bigger than y.");   // this branch is picked!
  }
  return 0;
}
```

实际上，在上面的代码中，程序逻辑在真正进入到条件语句之前，变量 x 的类型会首先被隐式转换为 `unsigned int` ，即无符号整型。而根据数据类型的解释规则，原先存放有 -10 补码的位模式会被解释为一个十分庞大的正整数，而这个数则远远大于 1。

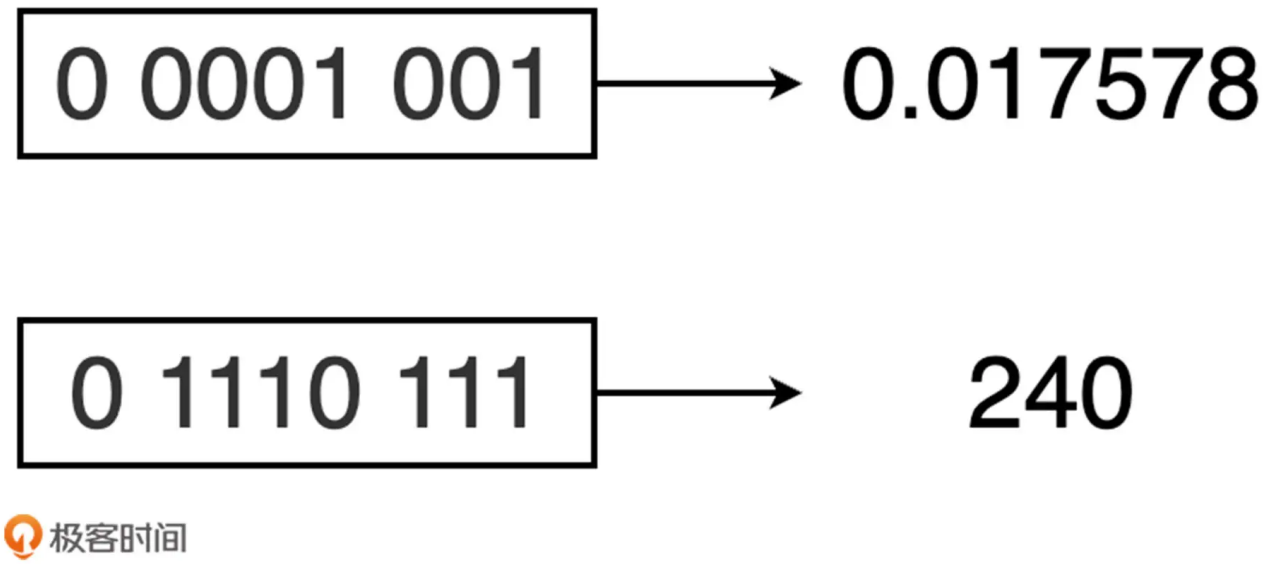
IEEE-754

我们上面主要介绍了有符号整数的补码，以及它在和无符号整数交互时的一些特性。而对于浮点数类型，大多数计算机体系会选择使用 IEEE-754 标准，作为其编码格式。

IEEE-754 是一个被众多硬件浮点计算单元（FPU）采用的浮点数标准，这个标准解决了浮点数在硬件实现上的很多问题，使其更具可移植性。

对于 IEEE-754，一个值得介绍的特点是它对浮点数的存储格式设计，使得计算机可以简单地使用对于整数的排序函数，来对浮点数进行排序。

举个例子，对于无符号数的二进制序列来说，`0010` 的值肯定要小于`1000`（2 < 8）。这对计算机来说很好判断。而对基于 IEEE-754 编码的 8 位浮点数（4 位阶码位，3 位小数位）二进制序列 `0 0001 001` 和 `0 1110 111` 来说，判断其大小也同样十分简单。除去最左侧的符号位外，直接将其余各位当作无符号整数序列值进行比较，所得结果同样适用于对应的浮点数序列。



当然，同整数一样，C 语言在对浮点数进行类型转换时（无论隐式还是显式），也都不会对底层存放的浮点数据进行改动，而只是将对应位序列的解释方式从浮点数改为了其他方

式。在 C 语言中，双精度浮点类型 double 具有作为隐式类型转换的最高优先级。当在一个表达式中存在该类型的变量时，计算机会首先将其他参与变量均转换为该类型，然后再进行表达式求值。

数据的存储位置

了解了数据的基本存储形式，我们再来看看数据会被存放在哪里。

在 C 语言中，通过不同的语法形式，我们可以定义具有不同数据类型的变量，这些变量按照其定义所在位置，可以被划分为局部变量、全局变量。进一步地，通过添加 static 关键字，可以将变量标记为静态类型，以延长变量的生存期，并限定其可见范围为当前编译单元，即当前所在源文件；通过添加 register 关键字，还可以建议编译器将变量值存放到寄存器中，以提升其读写性能。

对于上面提到的这些变量形式，其可能的数据存放位置均不尽相同。根据变量定义时使用的不同语法形式，我总结了变量数据的可能存放位置，如下表所示：

C 语法形式	数据存放位置	说明
int x = 10; / static int y = 10;	.data	初始化的全局变量或静态变量
void foo() { int x = 10; }	寄存器、栈	局部变量
register int x = 10;	寄存器、栈	被 register 关键字修饰的变量
calloc(256);	堆	申请的堆内存
int x = 0; / static int y;	.bss	初始值为 0 的全局或静态变量

极客时间

需要注意的是，表格里和这一讲后面提到的以 “.” 作为开头的标识，都指代对应的 Section 结构。这些结构，我会在第四个模块中为你详细介绍，这里你可以先有个整体感知。

接下来我将具体介绍这些变量数据的可能存放位置。先来看初始化的全局变量和静态变量，这类变量的值具有与应用程序同样长的生命周期，其值通常会被存放到进程 VAS（Virtual Address Space，虚拟地址空间）内的 .data 中。

VAS 我同样会在第四个模块中详细介绍，这里先不展开了。你可以先这样简单理解：应用程序在被正常加载和运行前，需要首先将应用程序代码，及其相关依赖项的数据映射到内存中的某个位置，这段包含有应用程序正常运行所必备数据的内存段即进程的 VAS。像 .data 等以 “.” 开头作为标记的 Section 结构，都代表着该内存段中的某个具体位置，这些 Section 结构都为应用的正常运行提供了各方面支持。

局部变量是我们在编写程序时最常使用的一种变量形式。一般来说，这些变量将被存放在寄存器或应用程序 VAS 的栈内存中，具体使用哪种方式则依赖于编译器的选择。

除此之外，未初始化的全局变量和静态变量，以及直接通过 `malloc`、`calloc` 等标准库函数创建的内存块中所包含的数据，其存放位置也有所不同。它们被分别存放到进程 VAS 的 .bss 以及堆内存中。这部分内容我也会在第四个模块中详细介绍。

最后，不同类型常量数据的存储方式也会有所不同。如下表所示，由于常量本身的不可变特征，它们会按照数据的大小和类型被选择性存放到进程 VAS 的 .rodata 以及 .text 中。其中，.rodata 用于存放只读（Read-only Data）数据，而 .text 通常用于存放程序本身的代码。

C 语法形式（注意高亮部分的常量）	数据存放位置	说明
foo("**Hello, Geektime!**");	.rodata	内联的长字符串
foo(**100**);	.text	内联的数字值

极客时间

一般的规律是，若内联的常量值较大，则会被单独存放到 .rodata 中保存，否则会直接内联到应用程序代码中，作为机器指令（比如最常见的 `mov` 指令）的字面量参数。

总结

好了，讲到这里，今天的内容也就基本结束了。最后我来给你总结一下。

这一讲，我主要介绍了 C 语言中与量值和数据相关的基本语法形式、数据实际存储时的具体编码方式，以及数据在程序运行过程中的实际存储位置等相关知识。

在 C 语言中，我们可以通过多种语法形式来控制一个变量的属性，比如变量的类型、生存期、值存储位置等。数值类型变量所具有的符号性为我们进一步精细化程序逻辑提供了可能。

在计算机内部，数据以二进制比特位的形式进行存放和使用。根据类型，它们会被选择性地特殊编码为相应的补码或 IEEE-754 格式，C 语言仅决定了如何从程序逻辑方面解释和使用这些数据，而不会对数据怎样存储产生影响。

走入底层，不同的 C 变量定义语法形式决定了数据不同的存放位置，而寄存器、栈、堆，以及各类存在于进程 VAS 中的 Section 结构都可能成为数据的存放地点。

和变量相比，常量则显得“轻巧”很多，它们无法在程序运行过程中被灵活修改，其数据存放位置也失去了更多的可能性。

思考题

最后，我们来一起做个思考题吧。

C 语言中的一个常用类型 `size_t` 通常被用在哪些地方？它是整数类型吗？是有符号数还是无符号数？欢迎在评论区留下你的答案。

今天的课程就结束了，希望可以帮助到你，也欢迎你把这节课分享给你的朋友或同事，我们一起交流。

上一篇 01 | 快速回顾：一个 C 程序的完整生命周期

下一篇

精选留言 (12)

写留言

一步 置顶

2021-12-09

我在 mac 环境下 可以使用 const 修饰的只读变量来指定数组长度和 switch case 的值，正常运行了
Apple clang version 13.0.0 (clang-1300.0.29.3)
Target: arm64-apple-darwin21.1.0
Thread model: posix

展开

作者回复: 这是一个很好的发现！
你可以试着在编译指令中添加 “-pedantic -Werror” 这两个选项，然后看看结果会有什么不同？实际上 Clang 在某些情况下会采用名为 “gnu-folding-constant” 的 GNU 扩展来编译 C 代码，但这并不是 C 标准中的内容。

3

jack123 置顶

2021-12-08

typedef unsigned int size_t;
具体类型还要看目标机器上的定义，
不过在一般机器上，size_t被定义成无符号整型
在一些常见C语言的函数的返回值是size_t
比如strlen，sizeof，...

展开

作者回复: 这个回答很棒！

1

pedro

2021-12-08

讲的好，拍案叫绝！

展开

2

qinsi
2021-12-10

```
#include <stdio.h>

int main(int argc, char *argv[]) {
        const float f = 16777214;
        for (int i = 0; i < 10; i++) {...
```

展开


qinsi
2021-12-10

```
#include <stdio.h>

int main(int argc, char* argv[]) {
        const int a[] = { 1, 2, 3 };
...
```

展开


观弈道人
2021-12-09

补码那一段表示看不懂~

展开

作者回复: 哪里没有看懂呢？

共 3 条评论 >

石天兰爱学习
2021-12-09

老师讲的很好，努力学习中，打卡

展开


=
2021-12-09

size_t是无符号整型。
它通常被用于循环中的变量声明、sizeof()的返回值类型、return返回值类型、malloc()分配空间大小的表示、数组大小的表示等。

ppm
2021-12-09

第一次了解到rodata数据段？

展开

柴
2021-12-08

size_t：一般用于指示数据类型的大小，无符号整型

展开

作者回复: 正解！

共 2 条评论

dog_brother
2021-12-08

思考题答案
同样用于下标和大小，是无符号整数类型。

作者回复: Bingo！

I
2021-12-08

size_t 是无符号整型，常用于计数

展开

作者回复: 没错的。