深入Linux C/C++

目录

[环境篇 3](#_Toc3058)

[1. Linux C/C++开发环境 3](#_Toc204)

[C语言篇 3](#_Toc26595)

[2. 基本数据类型 3](#_Toc26459)

[2.1. sizeof 3](#_Toc31143)

[2.2. 基本数据类型 3](#_Toc17369)

[2.3. 符号位于补码 4](#_Toc2158)

[2.4. 浮点数 4](#_Toc7645)

[3. 复合数据类型 5](#_Toc3576)

[3.1. 结构体 5](#_Toc24226)

[3.2. 共用体 5](#_Toc13091)

[3.3. 字节对齐 5](#_Toc9687)

[4. 数组 5](#_Toc29185)

[5. 指针 5](#_Toc481)

[6. 函数 5](#_Toc4507)

[6.1. 函数传参 5](#_Toc3853)

[6.2. 函数指针 5](#_Toc4297)

[6.3. 调用栈 5](#_Toc23949)

[7. const与static 5](#_Toc199)

[8. 类型识读 5](#_Toc15418)

[9. 宏与预编译 5](#_Toc5485)

[C++语言篇 6](#_Toc13926)

[10. 面向对象回顾 6](#_Toc8245)

[11. 继承 6](#_Toc17737)

[12. 内存管理 6](#_Toc5242)

[12.1. 堆内存 6](#_Toc31182)

[12.2. 内存布局 6](#_Toc26254)

[13. 异常和系统调用 6](#_Toc18887)

[13.1. 异常 6](#_Toc25754)

[13.2. 系统调用 6](#_Toc17370)

[应用篇 7](#_Toc24420)

[14. 文件操作 7](#_Toc18329)

[15. 网络编程 7](#_Toc27251)

[附加篇 7](#_Toc2985)

[16. 编程规范 7](#_Toc30250)

# 环境篇

# Linux C/C++开发环境

# C语言篇

# 基本数据类型

## sizeof

C语言中提供了sizeof运算符，用于计算数据类型或者变量的数据长度，sizeof为单目运算符，写作：sizeof(*数据类型*) 或者 sizeof(*变量名*)，比如以下代码。

|  |
| --- |
| printf(“%d\n”, sizeof(char));  printf(“%d\n”, sizeof(long));  char var1 = 5;  long var2 = 6;  printf(“%d\n”, sizeof(var1));  printf(“%d\n”, sizeof(var2)); |

表格 1 sizeof 使用举例

虽然采用sizeof()的形式，但sizeof不是一个被调用的函数，在编译器编译期间可以获知数据类型或者变量的长度，并将sizeof()部分替换为数据类型的长度。sizeof()运算符不仅能获取基本数据类型的长度，也能获取结构体、共用体数据的长度。

## 基本数据类型

C语言支持多种不同的基本数据类型，比如 char、int等，而这些基本的数据类型在不同的编译环境下的长度并不相同(主要受目标计算机字长的影响，通常为16位，32位和64位)。具体的数据类型占用字节数如下：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 类型 | DOS  Turbo C  16bit | Linux  GCC  32bit | Linux  GCC  64bit | Windows  VC  32bit | Windows  VC  64bit |
| char | 1 | 1 | 1 | 1 | 1 |
| unsigned char | 1 | 1 | 1 | 1 | 1 |
| short | 2 | 2 | 2 | 2 | 2 |
| unsigned short | 2 | 2 | 2 | 2 | 2 |
| int | 2 | 4 | 4 | 4 | 4 |
| unsigned int | 2 | 4 | 4 | 4 | 4 |
| long | 4 | 4 | 8 | 4 | 4 |
| unsigned long | 4 | 4 | 8 | 4 | 4 |
| long long |  | 8 | 8 | 8 | 8 |
| unsigned long long |  | 8 | 8 | 8 | 8 |
| float | 4 | 4 | 4 | 4 | 4 |
| double | 8 | 8 | 8 | 8 | 8 |
| void \* | 2 | 4 | 8 | 4 | 8 |

上表中，比如long和unsigned long 数据类型，在32bit和64bit的gcc编译环境下，数据的长度并不相同，在编程的时候需要多加注意。void \*表示C语言中指针数据类型的长度，指针类型的长度全都与计算机的字长相同，这样才能使指针指向任意的地址位置。

值得注意的是，决定的数据类型长度的是编译器而不是运行时的操作系统，比如在32bit gcc编译的程序拷贝到64bit linux下运行，此时long型依然占4个字节。因为64bit linux提供了兼容32bit的运行环境，将64bit的计算机当做32bit计算机来使用。程序的数据长度在编译完成时就已经确定了，并且不会再改变。如果希望long型占用8个字节，则需要使用64bit gcc重新编译。

## 补码与符号位

在C语言的整形数据类型中，都支持有符号数和无符号数，用于表示不同的数据类型，以char型为例，如下表：

|  |  |  |
| --- | --- | --- |
| 数据类型 | 数值范围 | 数值范围(补码表示) |
| char | -128 -- 127 | 10000000 -- 01111111 |
| unsigned char | 0 -- 255 | 00000000 -- 11111111 |

表格 2 char型、unsigned char 型数值范围

以上表的unsigned char 型为例子，各种整形数据类型都有自己的数值范围，当存储的数值超过或者低于这个数值范围的时候就会产生溢出，溢出后剩余的数值仍然在该数据类型描述的区间内。当unsigned char的变量值为255 与1相加时产生溢出以后剩余的值为0，而根据数学这个值应该是256，以此类推，数学上应该为257的值在unsigned char 中表示为1。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 数学数值 | ... | -2 | -1 | 0 | 1 |
| unsigned char | ... | 254 | 255 | 0 | 1 |
| 补码(二进制) | ... | 11111110 | 11111111 | 00000000 | 00000001 |
| 数学数值 | ... | 254 | 255 | 256 | 257 |
| unsigned char | ... | 254 | 255 | 0 | 1 |
| 补码(二进制) | ... | 11111110 | 11111111 | 00000000 | 00000001 |
| 数学数值 | ... | 510 | 511 | 512 | ... |
| unsigned char | ... | 254 | 255 | 0 | ... |
| 补码(二进制) | ... | 11111110 | 11111111 | 00000000 | ... |

表格 3 数学数值存储于unsigned char中

由上表可知，unsigned char型表示了一个0 -- 255 的连续区间(因为unsigned char占用8bit，故一共有2的8次幂个数值)，当任何数学数值保存于unsigned char的时候，都将转化为这个区间的某个数值来表示，否则因为存储空间有限而无法容纳，而这个转化计算就是数学数值对256做求模数。其补码的值与unsigned char 类型表达的值相同。

对于char类型，所表示的依然是 -128 -- 127 的连续区间，所表示数值的个数也是256个，比如下表：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 数学数值 | ... | -256 | -255 | ... | -129 | -128 |
| char | ... | 0 | 1 | ... | 127 | -128 |
| 补码(二进制) | ... | 00000000 | 0000001 | ... | 01111111 | 10000000 |
| 数学数值 | ... | -1 | 0 | 1 | ... | 127 |
| char | ... | -1 | 0 | 1 | ... | 127 |
| 补码(二进制) | ... | 11111111 | 00000000 | 00000001 | ... | 01111111 |
| 数学数值 | 128 | 129 | ... | 255 | 256 | 257 |
| char | -128 | -127 | ... | -1 | 0 | 1 |
| 补码(二进制) | 10000000 | 10000001 | ... | 11111111 | 00000000 | 00000001 |

表格 4 数学数值存储于 char 中

由上表可知，当任意数学数值存储于char型数据的时候，都将转换为-128 -- 127 区间内某个值来表示，转换过程为，将数学的数值与-128的差值对256求模数，再将模数与-128相加的结果。其补码的值为char型表达值对256求模数。

补码为有符号数提供了连续的码值序列，并且补码满足了 “某值补码 + 某值相反数补码 == 0”的要求，比如1的补码值为00000001，-1的补码值为11111111，两个补码直接相加值为00000000 (最高位因为溢出丢失)。因此可以得到公式：“某数的补码 = 0 - 相反数的补码”，考虑到溢出可以将公式转换为：“某数的补码 = (11111111 + 1) - 相反数的补码”，再转换为：“某数的补码 = 11111111 - 相反数的补码 + 1”，因为11111111减其他值得时候，相当于按位取反，因此公式再次转化为：“某数的补码 = 相反数的补码按位取反 + 1”。

在对char 和 unsigned char 型补码的观察中，我们会发现随着数值的递增，补码始终在00000000 -- 11111111 之间连续的循环，但是这些补码在unsigned char型中对应值0 -- 255,0 -- 255,0 -- 255的循环，在char型中对应0 -- 127,-128 -- -1,0 -- 127,-128 -- -1的循环。在进行算符运算过程中，求得的数值无论是多少，如果要存储在char 或者 unsigned char 类型中，都要进行数值对补码的转换。

**关于符号位**：

如果是一个有符号数，比如char型。其值小于0的情况下，即-128 -- -1区间，对应的补码是 10000000 -- 11111111。补码的最高位是1，此时我们这一位为符号位，符号位为1的时候表示值为负数。

思考以下问题，有符号类型的最高位为符号位，其他低位直接存储数值的绝对值，这样人们将数值转换为二进制存储的时候还能方便些，但是会不会有什么问题？答案是肯定的，在这样的数值存储方案中，会出现00000000 和 10000000 两个码值代表0 和 -0，而且其码值的二进制表示将按照11111111 -- 10000000,00000000 -- 01111111 来排列，其二进制表示并不是连续的，非常不方便计算机进行计算。而且也不能满足“某数的补码 + 相反数补码 == 0”的要求。

## 字节序

待补充

## 浮点数

待补充

## 默认数据类型

待补充

# 复合数据类型

## 结构体/共用体

## 数组

## 指针

## 字节对齐

## 位域

比特序

# 函数

## 函数传参

## 函数指针

## 调用栈

# const与static

# 类型识读

# 宏与预编译

# C++语言篇

# 面向对象回顾

# 继承

# 内存管理

## 堆内存

## 内存布局

# 异常和系统调用

## 异常

## 系统调用

# 应用篇

# 文件操作

# 网络编程

# 附加篇

# 编程规范