Linux2024

题目总结

本套题考察了变量作用域、指针、数组、结构体、联合体、宏定义、动态内存、字符串处理、位运算、浮点数精度、内存对齐、const 修饰指针等内容

0.聪明的吗喽

一个小猴子边上有 100 根香蕉，它要走过 50 米才能到家，每次它最多搬 50 根香蕉，（多了就拿不动了），它每走 1 米就要吃掉一根，请问它最多能把多少根香蕉搬到家里（提示：他可以把香蕉放下往返走，但是必须保证它每走一米都能有香蕉吃。也可以走到 n 米时，放下一些香蕉，拿着 n 根香蕉走回去重新搬 50 根。）

解析

1.100根香蕉超单次搬运上限（50根），得先往返搬一段路，把香蕉变少；

2.每往返1米要吃3根（去1根、回1根、再去1根），搬17米后，吃了51根，剩49根（刚好≤50根，不用再往返）；

3.剩下50-17=33米，单次搬49根，每米吃1根，到家剩49-33=16根

1.西邮Linux欢迎你啊

请解释以下代码的运行结果。

```
int main() {
    unsigned int a = 2024;
    for (; a >= 0; a--)
        printf("%d\n", printf("Hi guys! Join Linux - 2%d", printf("")));
    return 0;
}
```

解析

- 函数参数求值顺序：C语言规定函数参数遵循“自右向左”的求值顺序
- 逻辑运算符"||"的短路特性：当"||"左侧表达式结果为“真”时，右侧表达式会被跳过，不再执行

  逻辑运算符“&&”的短路特性：当“&&”左侧为真时还要继续判断右侧，只有当“&&”两侧表达式结果均为真时才会返回1
- `printf`函数的返回值：`printf`函数的返回值是其成功输出的字符个数，而非输出内容本身；若输出空字符串（`""`），返回值为0
- `unsigned int`[1]的取值范围：无符号整数仅存储非负整数（0 ~ 4294967295），不存在负数。当`a=0`时执行`a--`，会溢出为最大值`4294967295`，而非`-1`，循环会无限执行下去，这是因为无符号整数遵循**模运算规则**[2],`unsigned`（无符号）类型的模运算核心规则是 “结果恒为非负"
- 最内层`printf`输出空字符串，返回0,中间层`printf`输出字符串`Hi guys ! Join Linux - 20`共24个字符，返回24,最外层`printf`输出24并换行。

  单次循环最终输出`Hi guys ! Join Linux - 2024`,循环会无限次输出这行代码

2.眼见不一定为实

输出为什么和想象中不太一样?

你了解 `sizeof()` 和 `strlen()` 吗? 他们的区别是什么?

```c
int main() {
    char p0[] = "I love Linux";
    const char *p1 = "I love Linux\0Group";
    char p2[] = "I love Linux\0";
    printf("%d\n%d\n", strcmp(p0, p1), strcmp(p0, p2));
    printf("%d\n%d\n", sizeof(p0) == sizeof(p1), strlen(p0) == strlen(p1));
    return 0;
}
```

解析

核心知识

对比维度	sizeof	strlen
本质	运算符	库函数
计算内容	计算变量/类型占用的内存字节数	计算字符串中的有效字符数，不包含末尾的'\0'

代码分析

• `strcmp`[^3]：`strcmp`比较到字符串结束符`\0`为止，`p0`和`p1`的有效内容（`"I love Linux"`）完全相同，故返回`0`

• `strcmp(p0, p2)`：`p2`末尾的`\0`是字符串默认结束符，与`p0`的有效内容一致，比较结果为0

• `sizeof(p0) == sizeof(p1)`：`p0`是字符数组，`sizeof`计算数组总字节数，含隐式`\0`，共13字节；`p1`是指针，`sizeof`计算指针大小（32位系统4字节，64位系统8字节），两者不相等，结果为0

• `strlen(p0) == strlen(p1)`：`strlen`统计`\0`前的字符数，`p0`和`p1`的有效字符数均为12（`"I love Linux"`），长度相等，结果为1

最终输出

```
0
0
0
1
```

3.1.1-1.0！=0.1

为什么会这样，除了下面给出的一种方法，还有什么方法可以避免这个问题?

```c
int main() {
    float a = 1.0, b = 1.1, ex = 0.1;
    printf("b - a == ex is %s\n", (b - a == ex) ? "true" : "false");
    int A = a * 10, B = b * 10, EX = ex * 10;
    printf("B - A == EX is %s\n", (B - A == EX) ? "true" : "false");
}
```

解析

代码分析

`float`存储十进制小数，十进制小数转为二进制乘2取整，只有23位有效小数位，会截断，`a`,`b`,`ex`三个数只能存储为近似值，最终输出`b - a == ex is false`，下面一行将浮点数放大10倍转为整数，无误差，最终输出`B - A == EX is true`

如何避免

- 使用绝对值，通过比较两个浮点数差值的绝对值是否小于极小值
- 使用`double`类型，`doublie`存储位数更多，32位，精度更高

4.听说爱用位运算的人技术都不太差

解释函数的原理，并分析使用位运算求平均值的优缺点。

```
int average(int nums[], int start, int end) {
    if (start == end)
        return nums[start];
    int mid = (start + end) / 2;
    int leftAvg = average(nums, start, mid);
    int rightAvg = average(nums, mid + 1, end);
    return (leftAvg & rightAvg) + ((leftAvg ^ rightAvg) >> 1);
}
```

解析

函数的原理

1.递归求区间平均值

• 终止：当区间 [`start`, `end`] 只有1个元素时，直接返回该元素（单个元素的平均值为自身）

• 将区间二分（`mid = (start+end)/2`），分别递归计算左半区间 [`start`, `mid`] 和右半区间 [`mid+1`, `end`] 的平均值（`leftAvg`、`rightAvg`），最后合并两个子区间的平均值得到原区间结果

2.合并平均值

两个整数的平均值 = 两数共有的二进制位 + 两数不同的二进制位右移1位（即除以2）

`leftAvg & rightAvg`：提取两数二进制中 都为1的位（这些位相加后除以2，结果仍为自身，无需改变）

`leftAvg ^ rightAvg`：提取两数二进制中 不同的位（这些位相加后为1，除以2等价于右移1位）

位运算求平均值的优缺点

优点：

- 避免溢出：直接用 `(a+b)/2` 时，若 `a` 和 `b` 接近 `int` 最大值（如 `2^31-1`），`a+b` 会溢出为负数；
- 效率更高：位运算（`&`、`^`、`>>`）是CPU直接支持的底层操作，比加法+除法（除法需多周期）执行速度更快
- 代码简洁

缺点：

- 仅适用于整数：位运算针对二进制位操作，无法直接用于浮点数平均值计算，需额外处理小数部分
- 可读性差

5.全局还是局部！！！

先思考输出是什么，再动动小手运行下代码，看跟自己想得结果一样不一样 >-<

```
int i = 1;
static int j = 15;
int func() {
    int i = 10;
    if (i > 5) i++;
    printf("i = %d, j = %d\n", i, j);
    return i % j;
}
int main() {
    int a = func();
    printf("a = %d\n", a);
    printf("i = %d, j = %d\n", i, j);
    return 0;
}
```

解析

本题涉及考点为全局变量、局部变量和静态局部变量

区别	全局变量	局部变量	静态局部变量
存储位置	全局/静态存储区	栈区	全局/静态存储区
作用域	贯穿整个过程	仅在代码块中	仅在代码块中
默认初始化	未赋值默认初始化为0，仅初始化一次	未赋值为随机值，栈区垃圾数据	未赋值默认初始化为0,且仅初始化一次
重复定义	不可重复定义	可在不同汉函数里定义	可在不同函数里定义

代码分析

- 全局`i`：作用域为整个程序，但`func`内定义了局部`i`，会覆盖全局`i`，初始10，`i>5`成立后自增为11
- 局部`i`：仅在`func`函数里覆盖全局`i`的值
- `static int j`：静态全局变量，初始值15且全程不变
- `main`调用`func`：打印`func`内局部`i`=11和`j`=15，返回11%15=11，赋值给`a`
- 打印`a`=11，再打印全局`i`=1和静态`j`=15

  最终输出：

```
i = 11, j = 15
a = 11
i = 1, j = 15
```

6.指针的修罗场：改还是不改，这是个问题

指出以下代码中存在的问题，并帮粗心的学长改正问题。

```c
int main(int argc, char **argv) {
    int a = 1, b = 2;
    const int *p = &a;
    int * const q = &b;
    *p = 3, q = &a;
    const int * const r = &a;
    *r = 4, r = &b;
    return 0;
}
```

解析

代码分析

`const int *p`:指针`p`指向可改，但指向的变量的值不可改

`int *const q`:指针`q`指向不可改，但指向的变量的值可改

`const int *const r`指针`r`指向和指向的变量的值均不可改

`*p = 3`和`q = &a`和`*r = 4`和`r = &b`均错误

改正

```c
int main(int argc, char **argv) {
    int a = 1, b = 2;
    const int *p = &a;
    int * const q = &b;
    p = &b;
    *q = 3;
    const int * const r = &a;
    printf("a=%d, b=%d\n", a, b);
    return 0;
}
```

7.物极必反?

你了解 `argc` 和 `argv` 吗，这个程序里的 `argc` 和 `argv` 是什么?

程序输出是什么？解释一下为什么。

```c
int main(int argc, char *argv[]) {
    while (argc++ > 0);
    int a = 1, b = argc, c = 0;
    if (--a || b++ && c--)
        for (int i = 0; argv[i] != NULL; i++)
            printf("argv[%d] = %s\n", i, argv[i]);
    printf("a = %d, b = %d, c = %d\n", a, b, c);
    return 0;
}
```

解析

`argc` 和 `argv` 是用于处理命令行参数的标准方式，通常出现在 `main` 函数的参数列表

- `argc`

整数类型，表示程序运行时接收的命令行参数总数（包含程序名）

- `argv`

字符串数组，存储的命令行参数

`argv[0]` 是程序名

`argv[1]` 到`argv[argc-1]` 是用户输入的实际参数

`argv[argc]`为`NULL`（作为数组结束标志）

代码分析

自增/自减运算符：前置（`--a`）先改值再用，后置（`b++`）先用值再改

短路求值："||"左侧为真则右侧不执行，"&&"左侧为假则右侧不执行

`argc`初始为1,`argc`后置自增，进入`while`循环，argc++一直大于0,出不去while循环，直到增到最大值2147483647,这时再自增，溢出为-2147483648,再判断一次while的条件，argc再自增1,退出while循环，此时b=-2147483647

条件判断表达式拆解（`--a || b++ && c--`）

1.运算符优先级与结合性

- 优先级：`--`（前置）> `&&` > `||`；`&&`和`||`均为左结合
- 执行顺序：先算`--a`，再根据结果短路判断后续表达式

2.分步计算

- `--a`：前置自增，`a`从1变为0，表达式值为0（假）
- 因`||`左侧为假，需计算右侧`b++ && c--`：
- `b++ && c--`：左侧为真（非0），右侧为假（0），整体值为0（假）

最终表达式：`0 || 0 = 0`（假），因此`if`条件不成立

- 因`if`条件为假，`for`循环（遍历`argv`打印）不执行，但b还是自增了1变成-2147483646
- 最终打印`a`（0）、`b`、`c`

最终输出

```
a = 0, b = -2147483646, c = -1
```

8.指针？数组？数组指针？指针数组？

在主函数中定义如下变量：

```c
int main() {
    int a[2] = {4, 8};
    int(*b)[2] = &a;
    int *c[2] = {a, a + 1};
    return 0;
}
```

说说这些输出分别是什么？

```c
a, a + 1, &a, &a + 1, *(a + 1), sizeof(a), sizeof(&a)
*b, *b + 1, b, b + 1, *(*b + 1), sizeof(*b), sizeof(b)
c, c + 1, &c, &c + 1, **(c + 1), sizeof(c), sizeof(&c)
```

解析

- 题目分析

  `a`是一个包含两个int整数的数组

  `b`是一个数组指针，指针指向a数组

  `c`是一个指针数组，包含两个指针，分别指向a的两个元素

- 输出结果

`a`	**a首元素地址**
`a + 1`	a第二个元素地址
`&a`	整个数组地址，a的首元素地址
`&a + 1`	下一个数组地址，指向a之后的连续8字节内存
`*(a + 1)`	8,解引后为a第二个元素值
`sizeof(a)`	8 ，2 * 4 = 8
`sizeof(&a)`	8 ，数组指针（64位系统）
`*b`	a数组本身
`*b + 1`	a第二个元素地址
`b`	数组a的地址
`b + 1`	下一个数组地址，指向a之后的连续8字节内存
`*(*b + 1)`	8 ， 二次解引得到`a[1]`的值

a	a首元素地址
sizeof(*b)	8，2 * 4 = 8
sizeof(b)	指针（64位系统）
c	首元素地址
c + 1	数组第二个元素地址，指向c[1]
&c	整个指针数组的地址
&c + 1	下一个指针数组的地址，步长为8
**(c + 1)	8， 一次解引得a + 1，二次解引得到a[1]的值
sizeof(c)	8，2 * 4 = 8
sizeof(&c)	8， 数组指针

9.嘻嘻哈哈，好玩好玩

在宏的魔法下，数字与文字交织，猜猜结果是什么?

```
#define SQUARE(x) x *x
#define MAX(a, b) (a > b) ? a : b;
#define PRINT(x) printf("嘻嘻, 结果你猜对了吗, 包%d滴\n", x);
#define CONCAT(a, b) a##b

int main() {
    int CONCAT(x, 1) = 5;
    int CONCAT(y, 2) = 3;
    int max = MAX(SQUARE(x1 + 1), SQUARE(y2))
    PRINT(max)
    return 0;
}
```

解析

- 宏定义:

  `#define SQUARE(x) x * x`简单文本替换

  `#define MAX(a, b) (a > b) ? a : b;`:如果a>b的话，返回a，否则返回b

`#define PRINT(x) printf("嘻嘻，结果你猜对了吗，包%d滴\n", x)`:简单文本替换

`#define CONCAT(a, b) a##b`:是 C 语言中带参数的宏定义，核心功能是通过 `##`（预处理器连接符）将宏的两个参数直接拼接成一个符号（变量名、函数名、常量名等），且拼接发生在代码编译前的预处理阶段

- 逐步分析：

`int CONCAT(x, 1) = 5`：x1=5

`int CONCAT(y, 2) = 3`： y2=3

`int max = MAX(SQUARE(x1 + 1), SQUARE(y2))`:`SQUARE(x1 + 1)` 替换为 `x1 + 1 * x1 + 1`，而非预期的 `(x1 + 1) * (x1 + 1)`,结果为11,3*3=9,11>9返回11

输出`嘻嘻，结果你猜对了吗，包11滴`

10.我写的排序最快

写一个 `your_sort` 函数，要求不能改动 `main` 函数里的代码，对 `arr1` 和 `arr2` 两个数组进行升序排序并剔除相同元素，最后将排序结果放入 `result` 结构体中。

```
int main() {
    int arr1[] = {2, 3, 1, 3, 2, 4, 6, 7, 9, 2, 10};
    int arr2[] = {2, 1, 4, 3, 9, 6, 8};
    int len1 = sizeof(arr1) / sizeof(arr1[0]);
    int len2 = sizeof(arr2) / sizeof(arr2[0]);

    result result;
    your_sort(arr1, len1, arr2, len2, &result);
    for (int i = 0; i < result.len; i++) {
        printf("%d ", result.arr[i]);
    }
```

```
    free(result.arr);
    return 0;
}
```

解析

定义结构体

```
#include <stdio.h>
#include <stdlib.h>
typedef struct {
    int *arr;
    int len;
} result;
```

排序去重

```
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
void bubble_sort(int *arr, int len) {
    for (int i = 0; i < len - 1; i++) {
        for (int j = 0; j < len - 1 - i; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(&arr[j], &arr[j + 1]);
            }
        }
    }
}
int remove_duplicates(int *arr, int len) {
    if (len == 0) return 0;
    int unique_idx = 0;
    for (int i = 1; i < len; i++) {
        if (arr[i] != arr[unique_idx]) {
            unique_idx++;
            arr[unique_idx] = arr[i];
        }
    }
    return unique_idx + 1;
}
void your_sort(int arr1[], int len1, int arr2[], int len2, result *res) {
    int *copy1 = (int *)malloc(len1 * sizeof(int));
    int *copy2 = (int *)malloc(len2 * sizeof(int));
    for (int i = 0; i < len1; i++) copy1[i] = arr1[i];
    for (int i = 0; i < len2; i++) copy2[i] = arr2[i];
```

```
    bubble_sort(copy1, len1);
    bubble_sort(copy2, len2);
    int len1_unique = remove_duplicates(copy1, len1);
    int len2_unique = remove_duplicates(copy2, len2);
    int i = 0, j = 0, k = 0;
    int total_len = 0;
    int temp_i = 0, temp_j = 0;
    while (temp_i < len1_unique && temp_j < len2_unique) {
        if (copy1[temp_i] < copy2[temp_j]) {
            total_len++;
            temp_i++;
        } else if (copy1[temp_i] > copy2[temp_j]) {
            total_len++;
            temp_j++;
        } else {
            total_len++;
            temp_i++;
            temp_j++;
        }
    }
    total_len += (len1_unique - temp_i) + (len2_unique - temp_j);
    res->arr = (int *)malloc(total_len * sizeof(int));
    res->len = total_len;
    while (i < len1_unique && j < len2_unique) {
        if (copy1[i] < copy2[j]) {
            res->arr[k++] = copy1[i++];
        } else if (copy1[i] > copy2[j]) {
            res->arr[k++] = copy2[j++];
        } else {
            res->arr[k++] = copy1[i];
            i++;
            j++;
        }
    }
    while (i < len1_unique) res->arr[k++] = copy1[i++];
    while (j < len2_unique) res->arr[k++] = copy2[j++];
    free(copy1);
    free(copy2);
}
```

11.猜猜我是谁

在指针的迷宫中，五个数字化身为神秘的符号，等待被逐一揭示。

```c
int main() {
    void *a[] = {(void *)1, (void *)2, (void *)3, (void *)4, (void *)5};
    printf("%d\n", *((char *)a + 1));
    printf("%d\n", *(int *)(char *)a + 1);
    printf("%d\n", *((int *)a + 2));
    printf("%lld\n", *((long long *)a + 3));
    printf("%d\n", *((short *)a + 4));
    return 0;
}
```

解析

一、数组a的储存:

`a`是`void*`类型的数组，包括五个元素`(void *)1, (void *)2, (void *)3, (void *)4, (void *)5`

这步包括整数到指针的转换：当将整数强制转换为`void*`时，编译器会将整数视为一个 64 位的内存地址值,由于题目给的整数较小（1~5），转换后的 64 位指针值为 “高位补 0，低位为整数本身”

`(void*)1` 存储为 `0x0000000000000001`,以此类推

二、逐行输出分析：

- `*((char *)a + 1)`

  `(char*)a`：将数组首地址转换为`char*`(每次偏移1字节)

  `(char*)a + 1`：从首地址偏移 1 字节，指向`a[0]`的第 2 个字节（`a[0]`的内存为`01 00 00 00 00 00 00 00`，小端存储下低地址是`01`，偏移 1 字节后指向`00`）

  解引用后的值为`0`

- `*(int *)(char *)a + 1`

`(char*)a`转换为`int*`：指向`a[0]`的前 4 字节（`01 00 00 00`，小端存储对应`int`值`1`）。

`*(int*)(char*)a`取值为`1`，加 1 后为`2`

- `*((int *)a + 2)`

  `(int*)a`：将数组首地址视为`int*`（每次偏移 4 字节）

  `(int*)a + 2`：偏移`2×4=8`字节，指向`a[1]`的前 4 字节（`a[1]`值为 2，前 4 字节对应`int`值`2`）

  解引用后的值为`2`

- `*((long long *)a + 3)`

  `(long long*)a`：将数组首地址视为`long long*`（每次偏移 8 字节）

  `(long long*)a + 3`：偏移`3×8=24`字节，指向`a[3]`（数组第 4 个元素，值为 4）

  解引用后的值为`4`

- `*((short *)a + 4)`

  `(short*)a`：将数组首地址视为`short*`（每次偏移 2 字节）。

  `(short*)a + 4`：偏移`4×2=8`字节，指向`a[1]`的前 2 字节（`a[1]`值为 2，前 2 字节对应`short`值`2`）。

  解引用后的值为`2`。

答案

```
0
2
2
4
2
```

12.结构体变小写奇遇记

计算出 `Node` 结构体的大小，并解释以下代码的运行结果。

```c
union data {
    int a;
    double b;
    short c;
};
typedef struct node {
    long long a;
    union data b;
    void (*change)( struct node *n);
    char string[0];
} Node;
void func(Node *node) {
    for (size_t i = 0; node->string[i] != '\0'; i++)
        node->string[i] = tolower(node->string[i]);
}

int main() {
    const char *s = "WELCOME TO XIYOULINUX_GROUP!";
    Node *P = (Node *)malloc(sizeof(Node) + (strlen(s) + 1) * sizeof(char));
    strcpy(P->string, s);
    P->change = func;
    P->change(P);
    printf("%s\n", P->string);
    return 0;
}
```

解析

一、计算`Node`结构体的大小

需考虑内存对齐规则[3]

1. 分析`Node`结构体的成员

- **`long long a`**：`long long`占8 字节，对齐系数为8
- **`union data b`**：联合体大小为其最大成员的大小

  `int`4`double`8`short`2,联合体内存为其最大成员double大小，8字节，对齐系数为8
- **void (*change)(struct node *n)**：函数指针占8字节，对齐系数为8
- **`char string[0]`**：柔性数组[4]，不占用结构体本身的内存空间，用于后续动态扩展

2. 按内存对齐规则计算结构体总大小

- 成员`a`（8 字节）：从偏移 0 开始，占用 0~7 字节（满足 8 字节对齐）
- 成员`b`（8 字节）：最大对齐系数 8，下一个可用偏移为 8
- 成员`change`（8 字节）：对齐系数 8，下一个可用偏移为 16
- 柔性数组`string`：不占空间

结构体总大小为前 3 个成员的总占用空间24，且 24 是最大对齐系数8的整数倍，故`Node`结构体的大小为24 字节

二、代码运行结果分析

- **动态内存分配**：

`malloc(sizeof(Node) + (strlen(s) + 1) * sizeof(char))`[5]分配的内存包括：`Node`结构体本身 + 字符串`s`的长度（为柔性数组分配空间）

- **字符串复制**：

`strcpy(P->string, s)`将常量字符串`s`（`"WELCOME TO XIYOULINUX_GROUP!"`）复制到柔性数组中

- **函数指针赋值**：`P->change = func`使函数指针指向`func`函数
- **函数调用与字符串转换**：

`P->change(P)`调用`func`函数，遍历`string`中的每个字符，通过`tolower`[6]将大写字母转换为小写字母

答案

`Node`结构体的大小为 24 字节

```
welcome to xiyoulinux_group!
```

13.GNU/Linux(选做)

`ls`命令的用法与`/`、`.`、`~`的含义

- **`ls`命令**：用于列出当前目录（或指定目录）中的文件和文件夹。常见用法包括：
  - `ls`：列出当前目录的可见文件 / 文件夹；
  - `ls -l`：以详细列表形式显示（包含权限、所有者、大

小、修改时间等信息）；
    - `ls -a`：显示所有文件（包括以 `.` 开头的隐藏文件）；
    - `ls /path/to/dir`：列出指定目录（如 `ls /home`）的内容。
- **符号含义**：
    - `/`：表示 Linux 系统的根目录，是所有文件和目录的起点（类似 Windows 的 “此电脑” 根节点）；
    - `.`：表示当前所在的目录（例如 `ls .` 等价于 `ls`）；
    - `~`：表示当前登录用户的家目录（例如普通用户通常是 `/home/用户名`，root 用户是 `/root`）。

权限 `rwx` 的含义

Linux 中，文件 / 目录的权限通过 `r`（读）、`w`（写）、`x`（执行）三个字符表示，分别对应三类对象：**所有者（user）**、**所属组（group）**、**其他用户（others）**。例如 `rwxr-xr--` 表示：

- 所有者（user）：`rwx` 拥有读、写、执行权限；
- 所属组（group）：`r-x` 拥有读和执行权限，无写权限；
- 其他用户（others）：`r--` 仅拥有读权限。

对于目录而言，`x` 权限表示 “进入该目录” 的权限（若无 `x`，即使有 `r` 也无法查看目录内容）。

其他 GNU/Linux 相关知识

- **文件系统结构**：除了根目录 `/`，常见的重要目录有 `/bin`

（基础命令）、`/etc`（系统配置文件）、`/var`（动态数据如日志）、`/tmp`（临时文件）等。

- **命令行工具**：如 `cd`（切换目录）、`pwd`（显示当前路径）、`cp`（复制）、`mv`（移动 / 重命名）、`rm`（删除）、`grep`（文本搜索）、`sudo`（临时获取管理员权限）等。
- **用户与组**：Linux 是多用户系统，通过 `useradd`/`usermod`/`userdel` 管理用户，`groupadd` 等管理组，权限隔离严格。
- **包管理**：不同发行版（如 Ubuntu 用 `apt`，CentOS 用 `yum`/`dnf`）通过包管理器安装 / 更新软件，替代手动编译。
- **进程管理**：`ps` 查看进程，`kill` 终止进程，`top`/`htop` 实时监控系统资源。

类型	范围	原因
int	-2147483648到2147483647	1位用来当符号位，31位数
unsigned int	0到4294967295	32位全用来存储数值，无符号位

---

1. unsigned:`unsigned` 是 C 语言中的**无符号类型修饰符**，用于修饰整数类型（如 `int`、`char`、`long` 等），表示该类型的变量**只存储非负整数**（取值范围从 0 开始），没有符号位（正负之分）扩展非负整数的取值范围计算机中整数的存储会占用固定位数（如 `int` 通常占 4 字节 = 32 位），其中 1 位默认作为 “符号位”（0 表示正数，1 表示负数），剩下的位存储数值。`unsigned` 会 “取消符号位”，让所有位都用于存储数值，因此： 无符号类

型的最小值固定为 0； 最大值比对应的有符号类型大一倍（所有位都存数值）以 4 字节 `int` 和 `unsigned int` 为例，取值范围对比：↩

2. unsigned 模运算本质是无符号数的取余运算，结果始终为 非负整数，结果非负性：无论被除数正负（若实际传入负数，会先转为无符号数），结果均 ≥ 0 且 < |m|↩

3. **内存对齐规则**每个数据类型有默认的 “对齐系数”(通常等于其自身大小，如`char`为 1 字节，`int`为 4 字节，`double`为 8 字节等)，可通过`#pragma pack(n)`手动指定对齐系数（`n`为 2 的幂，如 1、2、4、8)，此时实际对齐系数为数据类型自身大小与`n`的较小值。成员对齐规则：结构体 / 联合体中，每个成员的起始地址相对于结构体首地址的偏移量，必须是该成员 “对齐系数” 的整数倍,若前一个成员占用的内存未满足当前成员的对齐要求，会自动填充空白字节。整体对齐规则：结构体 / 联合体的总大小，必须是其所有成员中最大对齐系数的整数倍,若总大小不满足，会在末尾填充空白字节↩

4. 主要作用是动态扩展结构体的内存空间，不占用结构体本身的内存，必须放到结构体的最后，动态分配结构体内存时必须包含柔性数组的内存↩

5. `malloc` 是 C 语言标准库 `<stdlib.h>` 中用于动态分配内存的函数，允许程序在运行时根据需要申请内存空间，常用于处理大小不确定的数据,如字符串。动态数组，结构体↩

6. tolower`是 C 语言标准库`<ctype.h>` 中的一个函数，用于将大写字母转换为对应的小写字母，非字母字符则保持不变↩