# Linux2025一面题

---

# 题目总结

**这套题全面覆盖了指针（含数组指针、指针数组、函数指针、二级指针）、`const` 权限控制、字符串处理、位运算、宏定义、全局 / 静态变量特性、浮点数精度、动态内存分配等知识点。最后还考察了 Linux 基础命令，检验初学者对Linux系统的掌握程度，也考察代码逻辑分析能力。**

# 0.拼命的企鹅

一只企鹅在爬山，每隔一段路都会遇到一块石头。第 1 块石头重量是 a，每往上走一段路，石头重量就会变成上一段的平方。企鹅可以选择把某些石头捡起来，最后把捡到的石头重量相乘。它怎样捡石头，才能得到重量乘积恰好是 a 的 b 次方的石头？（比如 b = 173 时， 要捡哪些石头？）

# 题目分析

- 第1块石头重量为`a`，第2块为$a^2$，第3块为$a^4$，第4块为$a^8$.....指数为2的0次方，2的1次方，2的3次方
- 捡到的石头重量相乘结果需恰好等于`a^b`，指数相加和为`b`
- 石头重量的指数是“2的幂次”，因此可以将`b`转为二进制，0

代表不捡这块石头，1代表捡这块石头，因为二进制最低位是乘2的0次方，所以第n块石头对应指数实际为2的n-1

## 求解步骤

- 第一步：将目标指数b转换为二进制
- 二进制的核心是“除2取余，逆序排列”，得到`b`的二进制：10101101
- 第二步：定位二进制中“1”的位置（对应2的幂次）

  二进制`10101101`从右到左（即从$2^0$到$2^7$）的位值与“1”的位置对应如下：

  “1”对应的2的幂次为：$2^0$、$2^2$、$2^3$、$2^5$、$2^7$。
  对应要捡石头为第一、三、四、六、八块

## 题目反思

- 本质是“二进制拆解”，而非“暴力尝试”

由于石头指数是2的幂次，无需暴力枚举所有组合，转为二进制就行

- 需注意石头序号与指数的对应关系+1

## 1.西邮Linux欢迎你啊

请解释以下代码

```
int main() {
    if (printf("Hi guys! ") || printf("Xiyou Linux ")) {
        printf("%d\n", printf("Welcome to Xiyou Linux 2%d", printf("")));
    }
    return 0;
}
```

## 题目分析

- 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
- 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

## 求解步骤

- 执行if条件判断（外层逻辑）
  - 先计算''||''左侧的`printf("Hi guys! ")`：输出“Hi guys! ”（共7个字符），返回值为7（非0，即“真”）。
  - 因“||”左侧为“真”，触发短路特性，右侧的`printf("Xiyou Linux ")`直接跳过，不执行，进入后续的大括号代码块。
- 执行大括号内的printf嵌套
  按“自右向左”顺序计算参数：
  - 步骤1：计算最右侧的`printf("")`：输出空内容，返回值为0（该值将作为内层printf的参数）
  - 步骤2：计算中间的`printf("Welcome to Xiyou Linux 2%d", 0)`：将步骤1的返回值0代入%d，输出“Welcome to Xiyou Linux 20”（共25个字符），返回值为25（该值将作为外层printf的参数）

- 步骤3：计算最外层的`printf`：将步骤2的返回值25代入`%d`，输出`25`并换行

- 最终输出结果

  综合以上步骤，最终控制台输出为：`Hi guys! Welcome to Xiyou Linux 2025`

## 2.可以和\0组一辈子字符串吗?

你能找到成功打印的语句吗？你能看出每个 if 中函数的返回值吗？这些函数各有什么特点？

```c
int main() {
    char p1[] = { 'W', 'e', 'l', 'c', 'o', 'm', 'e', '\0' };
    char p2[] = "Join us\0", p4[] = "xiyou_linux_group";
    const char* p3 = "Xiyou Linux Group\0\n2025\0";
    if (strcmp(p1, p2)) {
        printf("%s to %s!\n", p1, p2);
    }
    if (strlen(p3) > sizeof(p3)) {
        printf("%s", p3);
    }
    if (sizeof(p1) == sizeof(p2)) {
        printf("%s", p4);
    }
    return 0;
}
```

# 题目分析

核心考点：`strcmp`(函数）比较字符串内容，`sizeof`（运算符）计算变量占用内存字节数，`strlen`（函数）统计字符串有效字符数（不含`'\0'`）

## 求解步骤

- 第一个if语句：`if (strcmp(p1,p2))`

  strcmp[1]判断两者内容不同，`strcmp(p1,p2)`返回非 0，条件为真，执行打印：`Welcome to Join us!`

- 第二个if语句:`if (strlen(p3)>sizeof(p3))`

  `strlen`[2]统计到第一个 '\0' 为止的有效字符数,所以`strlen(p3)=16`

  `p3`是`const char*`，是一个指针变量，在 32/64 位系统中，指针占用内存固定为4 字节 / 8 字节，16大于 4/8，条件为真，因为遇第一个 '\0' 停止读取，打印：`Xiyou Linux Group`

- 第三个if语句:`if (sizeof(p1)==sizeof(p2))`

  计算`sizeof(p1)`,`p1`是字符数组，存储`"Welcome"`（7 个有效字符）+ 1 个 '\0'，共 8 个元素，每个`char`占 1 字节，`sizeof(p1) = 8`

  计算`sizeof(p2)`，`p2`是字符数组，存储`"Join us"`（7 个有效字符）+ 1 个显式 '\0'+1个隐式'\0'，共 9 个元素，`sizeof(p2) = 9`

  8 == 9，条件为假，不打印

## 最终打印

```
Welcome to Join us!
Xiyou Linux Group
```

## 3.数学没问题，浮点数有鬼

这个程序的输出是什么？解释为什么会这样？

```
int main() {
    float a1 = 0.3, b1 = 6e-1, sum1 = 0.9;
    printf("a1 + b1 %s sum1\n", (a1 + b1 == sum1) ? "==" : "!=");
    float a2 = 0x0.3p0, b2 = 0x6p-4, sum2 = 0x0.9p0;
    printf("a2 + b2 %s sum2\n", (a2 + b2 == sum2) ? "==" : "!=");
    return 0;
}
```

# 题目分析

- 十进制浮点数的二进制存储误差
- 十六进制浮点表示法的精确性
- 浮点数直接比较的风险

# 求解步骤

- 比较`a1 + b1`与`sum1`

  `b1=6e-1=0.6`$^3$,此时这三个数均为十进制小数，在这里转为二进制来比较，乘二取整，由于`float`中23位用于小数存储，超过截断，因此存在误差，比较结果为`!=`

- 比较`a2+b2`与`sum2`

  这三个数均为十六进制浮点数，十六进制能精确转换为二进制，是因 16 是 2 的 4 次方，1 位十六进制数可唯一对应 4 位二进制数，转换时仅需按位替换，无精度损失。因此方便观察，我们可将这三个数转为十进制来判断他们是否相等

  - `0x0.3p0` -> 3×16^−1^=3/16=0.1875
  - `0x6p-4` -> 6×2^−4^=6/16=0.375

- `0x0.9p0` -> 9×16^−1^=9/16=0.5625

  比较结果为`==`

# 最终打印

```
a1 + b1 != sum1
a2 + b2 == sum2
```

## 4.不和群的数字

在一个数组中，所有数字都出现了偶数次，只有两个数字出现了奇数次，请聪明的你帮我看看以下的代码是如何找到这两个数字的呢?

```
void findUndercoverIDs(int nums[], int size) {
    int xorAll = 0,id_a = 0,id_b = 0;
    for (int i = 0; i < size; i++) {
        xorAll ^= nums[i];
    }
    int diffBit = xorAll & -xorAll;
    for (int i = 0; i < size; i++) {
        if(nums[i] & diffBit){
            id_a ^= nums[i];
        } else {
            id_b ^= nums[i];
        }
    }
    printf("These nums are %d %d\n", id_a, id_b);
}
```

# 题目分析

本题主要考察异或运算:

任何数与自身异或结果为 0

任何数与 0 异或结果为自身

相同为0,不同为1

# 求解步骤

第一步：计算所有元素的异或结果

```
int xorAll = 0;
for (int i = 0; i < size; i++) {
    xorAll ^= nums[i];
}
```

由于偶数次出现的数字会通过异或两两抵消，结果为 0，最终 `xorAll` 的值为两个目标数字的异或结果

第二步：找到区分两个数字的数位（`diffBit`）

```
int diffBit = xorAll & -xorAll;
```

- `-xorAll` 在二进制中是 `xorAll` 的补码[4]（按位取反加 1），`xorAll & -xorAll` 的作用是找到`xorAll`的最低位的 1，其余位为0
- `diffBit`对应的位是两个目标数字不同的位，可用于将数组分为两组

第三步：分组异或，分离两个目标数字

```
for (int i = 0; i < size; i++) {
    if(nums[i] & diffBit){
        id_a ^= nums[i];
    } else {
        id_b ^= nums[i];
    }
}
```

- 通过 `nums[i] & diffBit` 判断元素在标志位上的值，将数组分为两组，两个目标数字会被分到不同组
- 每组中偶数次出现的数字会抵消为0，最终剩下的就是其中一个目标数字

## 5.会一直循环吗

你了解`argc`和`argv`吗，程序的输出是什么？为什么会这样?

```
int main(int argc, char* argv[]) {
    printf("argc = %d\n", argc);
    while (argc++ > 0) {
        if(argc < 0){
            printf("argv[0] %s\n", argv[0]);
            break;
        }
    }
    printf("argc = %d\n", argc);
    return 0;
}
```

# 核心知识

`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）

## 求解步骤

- 初始：`argc=1`,输出`argc=1`
- 进入`while`循环：

  自增使得`argc`一直大于0,不会进入`if`,不会`break`，陷入无限循环，直到`argc`超出`int`类型的最大范围增到2147483647,下一次自增变为`-2147483648`,此时判断`-2147483648<0`,进入`if`语句，打印argv[0]循环退出，输出`argc=-2147483648`

## 最终输出

```
argc=1
argv[0] (程序名)
argc=-2147483648
```

# 6.const与指针：谁能动？谁不能动？

```
struct P {
    int x;
    const int y;
};

int main() {
    struct P p1 = { 10, 20 }, p2 = { 30, 40 };
    const struct P p3 = { 50, 60 };
    struct P* const ptr1 = &p1;
    const struct P* ptr2 = &p2;
    const struct P* const ptr3 = &p3;
    return 0;
}
```

说说下列操作是否合法，并解释原因：

`ptr1->x = 100`, `ptr2->x = 300`, `ptr3->x = 500`

`ptr1->y = 200`,`ptr1 = &p2`,`ptr2->y = 400`

`ptr2 = &p1`,`ptr3->y = 600`,`ptr3 = &p1`

## 核心知识

指针修饰的变量不可改，谁离得近谁不能改

## 求解步骤

- 结构体内`const`修饰`y`，因此不论`*`在哪`y`的值均不可改
- `P1`,`P2`的`x`值可改，`y`值不可改
- `P3const`修饰结构体，`x`,`y`值均不可改
- `ptr1`指针本身是`const`，不可修改指向
- `ptr2 const`修饰结构体，`ptr2`的指向可改，但指向的结构体成员的值不可改
- `ptr3`指针本身和指向的对象均为`const`，指向和指向的结构体的内容均不可改

## 示例

`ptr1->x = 100`合法

`ptr2->x = 300`非法

`ptr3->x = 500`非法

`ptr1->y = 200`非法

`ptr1 = &p2`非法

`ptr2->y = 400`非法

`ptr2 = &p1`合法

`ptr3->y = 600`非法

`ptr3 = &p1`非法

## 7.指针！数组！

在主函数中定义如下变量:

```
int main() {
    int a[3] = { 2, 4, 8 };
    int(*b)[3] = &a;
    int* c[3] = { a, a + 1, a + 2 };
    int (*f1(int))(int*, int);
    return 0;
}
```

说说这几个表达式的输出分别是什么?

`a`,`*b`,`*b + 1`,`b`,`b + 1`,`* (*b + 1)`,`c`,`sizeof(a)`,`sizeof(b)`,`sizeof(&a)`,`sizeof(f1)`

# 题目分析

考查数组指针、指针数组、函数指针的性质，以及`sizeof`运算符对不同类型变量的计算规则

- 数组名隐式转换为指向首元素的指针
- 数组指针`int(*b)[3]`的解引用
- 指针数组`int*c[3]`的本质
- `sizeof`对数组、指针、函数名的计算规则
- 函数声明的解析

## 求解步骤

- 第一行定义`int`类型数组`a`，包括三个元素
- 第二行定义数组指针`b`，指向整个数组`a`
- 第三行定义指针数组`c`，元素为3个指针，分别指向`a`的3个元素
- 第四行声明函数`f1`,其返回值为函数指针，这个函数指针指向一个参数为`int*`和`int`,返回值为`int`的函数（接收`int`，返回 `int(*)(int*, int)`类型指针）

## 示例

- `a`： `a`数组首元素地址
- `*b`：对`b`进行解引，为数组`a`，也为`a`数组首元素地址
- `*b+1`:`a`数组第二个元素的地址
- `b`:存储的`a`数组的地址，也为`a`数组首元素的地址
- `b+1`:偏移一个数组的大小，比`b`大12个字节
- `*(*b + 1)`:`a`第二个元素的值
- `c`:`c`数组首元素的地址
- `sizeof(a)`:12 数组总字节
- `sizeof(b)`:8 一个指针所占字节数
- `sizeof(&a)`:8 一个数组指针的大小
- `sizeof(f1)`:8 函数指针的大小（函数在sizeof中会被隐式转换为函数指针，在这计算函数指针的字节数）

## 8.全局还是局部！！！

观察程序输出，思考为什么?

```
int g;
int func() {
    static int j = 98;
    j += g;
    return j;
}

int main() {
    g += 3;
    char arr[6] = {};
    arr[1] = func();
    arr[0] = func();
    arr[2] = arr[3] = func() + 1;
    arr[4] = func() + 1;
    printf("%s linux\n",arr);
    return 0;
}
```

# 核心知识

全局变量初始化：未初始化的全局变量`g`默认初始化为 0

静态局部变量特性：函数内的静态变量`j`只初始化一次，生命周期贯穿程序运行全程，值会在多次函数调用间保留

输出要将对应ASCII码转化为对应的字符

# 求解步骤

`g += 3`：`g`从 0 变为3

- `arr[1]` =`func()`

调用`func()`：`j` = 98 +`g` = 98 + 3 = 101对应为'e',`arr[1]=101`

- `arr[0] = func()`

调用`func()`：`j` = 101 + 3 = 104对应为'h',`arr[0]=104`

- `arr[2] = arr[3] = func() + 1`

调用`func()`：`j` = 104 + 3 = 107 对应为'k'

`func()`+ 1 = 108 对应'l'，`arr[2]=108 arr[3]=108`

- `arr[4] = func() + 1`

调用`func()`：`j` = 107 + 3 = 110 对应'n'

`func()`+ 1 = 111 对应'o',`arr[4]=111`

数组`arr`最终：

```
[104('h'), 101('e'), 108('l'), 108('l'), 111('o'), 0('\0')]
```

# 最终输出

```
hello linux
```

## 9.宏函数指针

观察程序结果，说说程序运行的过程：

```
#define CALL_MAIN(main, x) (*(int (*)(int))*main)(x);
#define DOUBLE(x) 2 * x
int (*registry[1])(int);
int main(int argc) {
    if (argc > 2e3) return 0;
    printf("%d ", argc + 1);
    *registry = (int(*)(int))main;
    CALL_MAIN(registry, DOUBLE(argc + 1));
    return 0;
}
```

# 核心知识

- 宏定义解析

`CALL_MAIN(main, x)`：将`main`解析为函数指针数组，取出首元素并强制转换为`int (*)(int)`类型（接收`int`参数、返回`int`的函数指针），然后调用该函数并传入参数`x`

`DOUBLE(x)`：简单文本替换为`2 * x`，注意这里没有括号

- 函数指针数组

`int (*registry[1])(int)`：定义一个包含 1 个元素的数组，元素类型为`int (*)(int)`（函数指针），用于存储指向`main`函数的指针

- 递归调用特性

程序通过函数指针数组`registry`存储`main`函数地址，再通过宏`CALL_MAIN`实现`main`函数的递归调用

# 求解步骤

1.初始参数`argc`的值为 1

2.第一次流程：

- 判断`argc > 2e3`（即argc > 2000），不成立
- 打印`argc + 1`：输出2
- `*registry = (int(*)(int))main`：将`registry`数组的首元素指向`main`函数（存储`main`的函数指针）
- 执行`CALL_MAIN(registry, DOUBLE(argc + 1))`：

- 宏展开：`(*(int (*)(int))*registry)(2 * (argc + 1))`
- 解析：`*registry`是指向`main`的函数指针，强制转换后调用，传入参数`2 * (1 + 1) = 4`
- 效果：递归调用`main(4)`

3.递归调用

每次递归调用的参数为上一次`argc + 1`的 2 倍：

- 第 1 次调用：`argc=1` -> 打印2 -> 调用`main(4)`
- 第 2 次调用：`argc=4` -> 打印5 -> 调用`main(2*(4+1)=10)`
- 第 3 次调用：`argc=10` ->打印`11` -> 调用`main(2*(10+1)=22)`
- 第 4 次调用：`argc=22` ->打印`23` ->调用`main(46)`
- ...

4.终止条件

当某次调用的`argc > 2000`时，返回0，结束

## 10.拼接 排序 去重

本题要求你编写以下函数，不能改动 `main` 函数里的代码。实现对 `arr1`和`arr2` 的拼接、排序和去重。你需要自行定义`result`结构体并使用 `malloc`手动开辟内存。

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

```
    struct result result;
    your_concat(arr1, len1, arr2, len2, result);
    print_result(result);
    your_sort(result);
    print_result(result);
    your_dedup(result);
    print_result(result);
    free(result.arr);
    return 0;
}
```

# 题目分析

# 声明结构体

结构体需包含两个关键信息：存储数据的动态数组指针和数组长度

```
struct result {
    int* arr;  // 存储动态数组的指针
    int len;   // 数组当前长度
};
```

# 定义拼接函数

将`arr1`和`arr2`的所有元素合并到`result`的动态数组中

```
void Your_concat(int arr1[], int len1, int arr2[], int len2, struct result* result) {
    // 计算总长度并分配内存
    result->len = len1 + len2;
    result->arr = (int*)malloc(result->len * sizeof(int));
    if (result->arr == NULL) {
        // 内存分配失败处理（实际可增加报错逻辑）
        result->len = 0;
        return;
    }
    // 复制arr1元素
    for (int i = 0; i < len1; i++) {
        result->arr[i] = arr1[i];
    }
    // 复制arr2元素
    for (int i = 0; i < len2; i++) {
        result->arr[len1 + i] = arr2[i];
    }
```

```
}
```

# 定义排序函数

冒泡排序

```
void Your_sort(struct result* result) {
    if (result->len <= 1) return;  // 空数组或单元素无需排序
    // 冒泡排序（升序）
    for (int i = 0; i < result->len - 1; i++) {
        for (int j = 0; j < result->len - 1 - i; j++) {
            if (result->arr[j] > result->arr[j + 1]) {
                // 交换元素
                int temp = result->arr[j];
                result->arr[j] = result->arr[j + 1];
                result->arr[j + 1] = temp;
            }
        }
    }
}
```

# 定义去重函数

移除排序后数组中的重复元素，更新数组长度并释放多余内存

利用排序后重复元素相邻的特性，遍历数组时只保留与前一个元素不同的值

重新分配内存存储去重后的元素，并释放原内存，确保内存使用合理

```
void Your_dedup(struct result* result) {
    if (result->len <= 1) return;  // 空数组或单元素无需去重
    // 统计不重复元素数量
    int new_len = 1;
    for (int i = 1; i < result->len; i++) {
        if (result->arr[i] != result->arr[i - 1]) {
            result->arr[new_len++] = result->arr[i];
        }
    }
    // 重新分配内存（释放原内存并分配新内存，或使用realloc）
    int* new_arr = (int*)malloc(new_len * sizeof(int));
    if (new_arr != NULL) {
```

```
        // 复制不重复元素到新数组
        for (int i = 0; i < new_len; i++) {
            new_arr[i] = result->arr[i];
        }
        free(result->arr);  // 释放原内存
        result->arr = new_arr;
        result->len = new_len;
    }
}
```

## 11.指针魔法

用你智慧的眼睛，透过这指针魔法的表象，看清其本质：

```
void magic(int(*pa)[6], int** pp) {
    **pp += (*pa)[2];
    *pp = (*pa) + 5;
    **pp -= (*pa)[0];
    *pp = (*pa) + ((*(*pa + 3) & 1) ? 3 : 1);
    *(*pp) += *(*pp - 1);
    *pp = (*pa) + 2;
}
int main() {
    int a[6] = { 2, 4, 6, 8, 10, 12 };
    int* p = a + 1,** pp = &p;
    magic(&a, pp);
    printf("%d %d\n%d %d %d\n%d %d\n",*p,**pp,a[1],a[2],a[3],a[5],p-a);
    return 0;
}
```

# 题目分析

本题考查多级指针、数组指针、解引等

# 求解步骤

- 初始：

  数组 `a = [2, 4, 6, 8, 10, 12]`

  指针`p` 指向 `a[1]` （`p = &a[1]`，`*p = 4`）

二级指针 `pp` 指向`p`（即 `pp` = `&p`，`*pp` = `p`，`**pp` = `4`）

`pa` 是数组指针，指向`a`（`*pa` 等价于 `a`）

- 执行`**pp += (*pa)[2]`
  - `(*pa)[2]` 是 `a[2] = 6`
  - `**pp`是`p`指向的元素（`a[1] = 4`）
  - 运算后：`a[1]`= 4 + 6 = 10
- 执行`*pp = (*pa) + 5`
  - `(*pa) + 5` 是 &a[5]（数组 `a` 第 5 个元素的地址）
  - `*pp`是`p`，因此`p`被修改为 `&a[5]`
  - `p`指向`a[5]`,`*p`=12
- 执行`**pp -= (*pa)[0]`
  - `(*pa)[0]` 是 `a[0] = 2`
  - `**pp`是`p`指向的元素（即 `a[5]` = 12）
  - 运算后：`a[5]` = 12 - 2 = 10
- 执行`*pp = (*pa) + ((*(*pa + 3) & 1) ? 3 : 1)`
  - `*pa + 3`是 `&a[3]`，`*(*pa + 3) = a[3] = 8`
  - 8 `&` 1 = 0（偶数），条件表达式结果为 `1`
  - `(*pa) + 1` 是 `&a[1]`，因此 `*pp`（即 `p`）被修改为 `&a[1]`
  - `p`指向`a[1]`,`*p`=10
- 执行`*(*pp) += *(*pp - 1)`

`*pp`是 `p = &a[1]`，`*pp - 1 = &a[0]`，`*(*pp - 1) =`

`a[0] = 2`

`*(*pp)` 是 `a[1] = 10`

运算后：`a[1]` = 10 + 2 = 12

- 执行`*pp = (*pa) + 2`

  `(*pa) + 2` 是 `&a[2]`，因此 `*pp`（即 `p`）被修改为 `&a[2]`

  此时`p`指向 `a[2]`（`*p = 6`）

# 最终输出

```
6 6
12 6 8
10 2
```

## 12.奇怪的循环

你能看明白这个程序怎样运行吗？试着理解这个程序吧！

```
union data {
    void**** p;
    char arr[20];
};
typedef struct node {
    int a;
    union data b;
    void (*use)(struct node* n);
    char string[0];
} Node;
void func2(Node* node);

void func1(Node* node) {
    node->use = func2;
    printf("%s\n", node->string);
}
void func2(Node* node) {
    node->use = func1;
    printf("%d\n", ++(node->a));
}
int main() {
    const char* s = "Your journey begins here!";
```

```
    Node* P = (Node*)malloc(sizeof(Node) + (strlen(s) + 1) * sizeof(char));
    strcpy(P->string, s);
    P->use = func1;
    P->a = sizeof(Node) * 50 + sizeof(union data);
    while (P->a < 2028) {
        P->use(P);
    }
    free(P);
    return 0;
}
```

# 题目分析

- 联合体的内存共享特性（`union data`中 `p` 和 `arr` 共享 20 字节内存）
- 柔性数组的用法（`string[0]` 用于动态扩展结构体存储字符串）
- 函数指针的切换调用（`use` 交替指向 `func1` 和 `func2`）
- 动态内存分配与字符串复制（`malloc` 分配包含柔性数组的内存）

# 求解步骤

1.`union data`:由一个指针和20字节数组组成，内存大小为最大成员的内存，所以内存大小为20

2.`struct Node`:`int a` 4字节 `union data b`20字节 `void(*use)<font color=black size=5>(Node*)`:函数指针，占8字节 `char string[0]`:柔性数组，不占结构体内存

3.动态分配内存：`sizeof(Node) + 字符串长度+1`（包含`\0`），为柔性数组申请内存储存`s`

`string`复制为`"Your journey begins here!"`

4.函数调用

初始`use`指向`func1`，`a`的初始值为`sizeof(Node)*50 + sizeof(union data)`

- **`func1`**：将`use`切换为`func2`，打印`string`内容
- **`func2`**：将`use`切换为`func1`，打印`++a`）
- **终止条件**：`while (P->a < 2028)`，每次调用`P->use(P)`时，实现函数指针在`func1`和`func2`之间交替

5.`a`的初始值取决于`sizeof(Node)`，而`Node`的大小受内存对齐影响（64 位系统典型情况）：

- `int a`：4 字节（对齐到 4 字节）
- `union data b`：需按 8 字节对齐（因指针成员为 8 字节），对齐为 24 字节
- 函数指针`use`：8 字节（按 8 字节对齐）
- 所以`sizeof(Node)`=4+24+8=36

  a=36*50 + 20 = 1800 + 20 = 1820

  循环需执行的`func2`次数：

  2028 - 1820 = 208次（每次`func2`调用使`a`自增 1）

6.**最终输出**

```
Your journey begins here!
2025
Your journey begins here!
2026
Your journey begins here!
2027
Your journey begins here!
2028
```

## 13.GNU/Linux(选做)

注：嘿！你或许对 Linux 命令不是很熟悉，甚至你没听说过 Linux。但别担心，这是选做题，了解 Linux 是加分项，但不了解也不扣分哦！

1.你知道 `cd` 命令的用法与 `/` `~` `-` 这些符号的含义吗?

2.你知道 Linux 系统如何创建和删除一个目录吗?

3.请问你还懂得哪些与 GNU/Linux 相关的知识呢?

# cd 命令及 / ~ - 的含义

- `cd` 是 Linux 中用于切换当前工作目录的命令，基本用法为 `cd 目录路径`（例如 `cd /home` 切换到 `/home` 目录）
- `/`：表示根目录，是 Linux 文件系统的最顶层目录，所有文件和目录都从根目录开始延伸
- `~`：表示当前用户的主目录（家目录），例如 `root` 用户的主目录是 `/root`，普通用户的主目录通常是 `/home/用户名`。使用 `cd ~` 可快速回到主目录，简化为 `cd` 也能实现相同效果
- `-`：表示上一次所在的目录，使用 `cd -` 可在当前目录和上

一次目录之间切换（类似 “返回” 功能）

## Linux 中创建和删除目录的命令

- 创建目录：使用 `mkdir` 命令，基本用法为 `mkdir 目录名`（例如 `mkdir test` 创建名为 `test` 的目录）。若要创建嵌套目录（如 `a/b/c`），可加 `-p` 参数：`mkdir -p a/b/c`（自动创建父目录）
- 删除目录：使用 `rmdir` 或 `rm` 命令
  - `rmdir` 只能删除空目录，用法：`rmdir 目录名`（若目录非空会报错）
  - `rm` 命令加 `-r` 参数可删除非空目录（递归删除目录及内容），例如 `rm -r test`；若需强制删除（不提示确认），可加 `-f` 参数：`rm -rf test`（使用时需谨慎，避免误删重要文件）

## 其他 GNU/Linux 相关知识

- **文件权限**：Linux 中每个文件 / 目录有读（r）、写（w）、执行（x）权限，分别对应所有者、所属组、其他用户三类角色，可用 `ls -l` 查看，`chmod` 命令修改（例如 `chmod 755 file` 设为所有者可读可写可执行，组和其他用户可读可执行）
- **包管理工具**：不同发行版有不同工具，如 Ubuntu/Debian 用 `apt`（`apt install 软件名`），CentOS/RHEL 用 `yum` 或 `dnf`，Arch Linux 用 `pacman` 等，用于安装、更新、卸载软件

- **进程管理**：`ps` 命令查看当前进程（`ps aux` 显示所有进程），`top` 或 `htop` 实时监控进程资源占用，`kill` 命令终止进程（`kill -9 进程ID` 强制终止）
- **管道与重定向**：`|` 管道可将一个命令的输出作为另一个命令的输入（例如 `ls | grep .txt` 筛选出 `.txt` 文件）；`>` 或 `>>` 重定向输出到文件（`>` 覆盖，`>>` 追加，如 `echo "hello" > test.txt`）
- **发行版差异**：GNU/Linux 有众多发行版，如面向桌面的 Ubuntu、Fedora，面向服务器的 CentOS、Debian，面向安全的 Kali Linux 等，核心均为 Linux 内核，但在软件管理、默认工具等方面有差异

---

1. `strcmp`是字符串处理函数，用来比较两个字符串的内容，通过比较字符串的ASCII码来实现相等返回 0，不相等返回非 0 值(`a > b`返回正，`a < b`返回负）↩
2. `strlen` 是 C 语言标准库 `<string.h>` 中的一个函数，用于计算字符串的长度（即字符串中字符的个数，不包含终止符 `'\0'`）↩
3. `strlen` 是 C 语言标准库 `<string.h>` 中的一个函数，用于计算字符串的长度（即字符串中字符的个数，不包含终止符 `'\0'`）↩
4. 补码是计算机中表示有符号整数的核心编码方式，能将减法运算转化为加法运算。正数的补码 = 其本身的二进制（符号位为

0，其余位为数值），负数的补码 = 其绝对值的二进制 “取反加 1”（符号位为 1，结果为数值部分）↩