Linux2023

题目总结

0.鼠鼠我啊，要被祸害了

有1000瓶水，其中有一瓶有毒，小白鼠只要尝一点带毒的水，24小时后就会准时死亡。至少要多少只小白鼠才能在24小时内鉴别出哪瓶水有毒?

解析

至少需要10只，原理是利用二进制编码,每只老鼠对应1位二进制数，0000000000~1111100111可覆盖1000瓶水

- 给1000瓶水编号0-999，转换为10位二进制数（如第5瓶=0000000101）
- 第i只老鼠（1-10）喝所有“第i位二进制为1”的水（如第3只喝编号0000000100、0000000101等的水）
- 24小时后，死亡老鼠记为1，存活记为0，组成10位二进制数，对应编号即为毒水瓶（如老鼠3和5死亡->0000000101->第5瓶有毒）

1.先预测一下~

按照函数要求输入自己的姓名试试~

```
char *welcome() {
    // 请你返回自己的姓名
}
int main(void) {
    char *a = welcome();
    printf("Hi, 我相信 %s 可以面试成功!\n", a);
    return 0;
}
```

解析

此时输出：

```
Hi, 我相信 (null) 可以面试成功!
```

因此需要分配内存：

```
char *welcome() {
    // 1. 分配动态内存（假设姓名最长20字符，+1存结束符'\0'）
```

```
    char *name = (char *)malloc(21 * sizeof(char));
    if (name == NULL) { // 检查内存分配是否成功
        printf("内存分配失败！\n");
        exit(1); // 分配失败则退出程序
    }
    scanf("%s", name); // 输入不含空格的姓名（含空格需用fgets）

    return name; // 返回动态内存地址（函数结束后内存不释放）
}

int main(void) {
    char *a = welcome();
    printf("Hi, 我相信 %s 可以面试成功!\n", a);

    free(a); // 释放动态内存，避免内存泄漏
    a = NULL; // 指针置空，防止野指针

    return 0;
}
```

·输入的姓名是字符串常量

·指针指向逻辑：`char *a` 接收的是字符串常量的首地址，`printf`通过该地址遍历字符，直到遇到 `'\0'` 结束符

`main`函数调用 `welcome()`，函数返回字符串常量“姓名”的首地址

指针 `a` 指向该地址，`printf`通过 `%s` 格式化输出地址对应的字符串

1.动态内存分配：用 `malloc` 在堆区分配内存，而非栈区（栈区变量在函数结束后会销毁，返回栈地址会导致野指针）

2.内存安全：

◦ 检查内现存分配是否成功；

用 `free` 释放内存，防止内存泄漏；

◦ 释放后指针置空，防止出现野指针；

2.欢迎来到Linux兴趣小组

有趣的输出，为什么会这样子呢~

```
int main(void) {
    char *ptr0 = "Welcome to Xiyou Linux!";
    char ptr1[] = "Welcome to Xiyou Linux!";
    if (*ptr0 == *ptr1) {
      printf("%d\n", printf("Hello, Linux Group - 2%d", printf("")));
    }
    int diff = ptr0 - ptr1;
    printf("Pointer Difference: %d\n", diff);
}
```

解析

分析条件

1.`ptr0` 与 `ptr1` 的本质差异

• `char *ptr0` = "字符串常量":
字符串常量存储在 只读数据区，`ptr0` 是指向该区域的指针（只读，不可修改内容）

• `char ptr1[]` = "字符串常量":
字符串常量被 拷贝到栈区，`ptr1` 是栈数组的首地址（可修改数组内容）

• 两者指向 不同内存区域，因此 `ptr0 - ptr1` 是两个不同内存地址的差值（非0）

2.`*ptr0 == *ptr1` 的判断逻辑

• `*ptr0` 取 `ptr0` 指向地址的第一个字符（`'W'`），`*ptr1` 取 `ptr1` 指向地址的第一个字符（也是`'W'`），因此条件为真，执行`if`内代码

`printf` 执行顺序：从内到外（先执行最内层，返回成功输出的字符个数）

- 最内层 `printf("")`：
  输出空字符串，返回值为 0
- 中层 `printf("Hello, Linux Group - 2%d", 0)`：
  代入参数0，输出字符串 `Hello, Linux Group - 20`，共输出 23个字符（返回值为23）
- 外层 `printf("%d\n", 23)`：
  输出中层`printf`的返回值23，换行
- `diff`为一个非零值，输出`Pointer Difference:(一个非零整数)`

```
Hello, Linux Group - 2023
Pointer Difference: 1431668036
```

3.一切都翻倍了吗

请尝试解释一下程序的输出。
请谈谈对sizeof()和strlen()的理解吧。
什么是sprintf()，它的参数以及返回值又是什么呢?

```
int main(void) {
    char arr[] = {'L', 'i', 'n', 'u', 'x', '\0', '!'}, str[20];
    short num = 520;
    int num2 = 1314;
    printf("%zu\t%zu\t%zu\n", sizeof(*&arr), sizeof(arr + 0),
           sizeof(num = num2 + 4));
    printf("%d\n", sprintf(str, "0x%x", num) == num);
    printf("%zu\t%zu\n", strlen(&str[0] + 1), strlen(arr + 0));
}
```

解析

逐步分析

1,`printf("%zu\t%zu\t%zu\n", sizeof(*&arr), sizeof(arr + 0), sizeof(num = num2 + 4))`

• `sizeof(*&arr)` -> 7： `&arr` 取数组首地址,`*&arr` 解引用，等价于 `arr`,`sizeof(arr)` 计算数　　组总字节数：7个`char` × 1字节 = 7

• `sizeof(arr + 0)` -> 8：`arr` 自动转换为数组首地址,为指针类型，64位系统中指针占8字节

• `sizeof(num = num2 + 4)` -> 2：`sizeof` 仅计算表达式结果的类型大小，不执行赋值运算,表达式结果类型为 `short`（`num` 是`short`类型），`short`占2字节，与 `num2+4` 的值无关

2.`printf("%d\n", sprintf(str, "0x%x", num) == num)`

• `sprintf(str, "0x%x", num)`：将 `num`（值为520，十六进制 0x208）格式化为字符串 `"0x208"` 存入 `str`

• `sprintf` 返回值：输出的字符个数（`"0x208"` 共5个字符，返回5),5 == 520 为假，输出 0

3.`printf("%zu\t%zu\n", strlen(&str[0] + 1), strlen(arr + 0))`

• `strlen(&str[0] + 1)` ->4：`&str[0]` 是 `str` 首地址，`+1` 指向第二个字符`'x'`,`strlen` 从 `'x'` 开始统计，到 `'\0'` 结束（字符串为 `"x208"`），共4个字符

• `strlen(arr + 0)` -> 5：`arr + 0` 等价于 `arr`（数组首地址），`arr` 内容为 {`'L','i','n','u','x','\0','!'`},`strlen` 统计到 `'\0'` 为止（忽略 `'\0'` 后的 `'!'`），共5个字符

最终输出

```
7       8       2
0
4       5
```

sizeof和strlen

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'

sprintf

1.功能

将格式化的数据写入字符数组（而非直接输出到屏幕，区别于 `printf`）

2.参数（共3个及以上，可变参数）

`int sprintf(char *str, const char *format, ...);`

• `char *str`：目标字符数组（需提前分配足够内存，避免溢出）；

• `const char *format`：格式化字符串（同 `printf`，如 `%d`、`%x`、`%s`）；

• `...`：可变参数（需格式化的数据，如整数、字符串等）

3.返回值

• 成功：返回写入数组的字符个数（不含末尾自动添加的`'\0'`）；

• 失败：返回负数（如内存溢出、格式错误）

4.关键注意事项

• 必须确保`str`内存足够（如示例中`str[20]`可容纳格式化后的字符串）；

• 不会自动检查内存边界，需手动控制长度（避免缓冲区溢出漏洞）

4.奇怪的输出

程序的输出结果是什么？解释一下为什么出现该结果吧~

```
int main(void) {
    char a = 64 & 127;
    char b = 64 ^ 127;
    char c = -64 >> 6;
    char ch = a + b - c;
    printf("a = %d b = %d c = %d\n", a, b, c);
    printf("ch = %d\n", ch);
}
```

解析

1.计算`a = 64 & 127`

64 的二进制：`01000000`

127 的二进制：`01111111`

按位与（`&`）：对应位均为 1 则为 1，否则为 0

`01000000 & 01111111 = 01000000`（即十进制 64）

2.计算`b = 64 ^ 127`

按位异或（`^`）：对应位不同则为 1，相同则为 0

`01000000 ^ 01111111 = 00111111`（即十进制 63）

3.计算`c = -64 >> 6`

-64 的二进制(补码)：

原码：`11000000`（最高位为符号位 1，表示负数）

反码：`10111111`（符号位不变，其他位取反）

补码：`11000000`（反码 + 1，因 - 64 是特殊值，补码仍为 `11000000`

右移 6 位（`>> 6`）：

对于负数，右移为符号位保持不变，剩余高位补`1->11111111`，是-1的补码，`c = -1`

4.计算`ch = a + b - c`

`64 + 63 - (-1) = 64 + 63 + 1 = 128`

但 `char` 范围为 -128~127，128 超出范围，发生溢出：

128 的二进制表示为 `10000000`（8 位）。

在 8 位有符号 `char` 中，最高位为符号位（1 表示负数），而 `10000000` 恰好是 `-128` 的补码（特殊值）

最终输出

```
a = 64 b = 63 c = -1
ch = -128
```

5.乍一看就不想看的函数

“人们常说互联网凛冬已至，要提高自己的竞争力，可我怎么卷都卷不过别人，只好用一些奇技淫巧让我的代码变得高深莫测。”

这个func()函数的功能是什么？是如何实现的？

```
int func(int a, int b) {
    if (!a) return b;
    return func((a & b) << 1, a ^ b);
}
int main(void) {
    int a = 4, b = 9, c = -7;
    printf("%d\n", func(a, func(b, c)));
}
```

解析

一、`func()` 函数核心功能

`func(a, b)` 的功能是计算两个整数 `a` 和 `b` 的和，本质是用 位运算模拟加法

二、位运算模拟加法

加法的本质是“无进位相加”和“处理进位”两步，`func` 通过递归实现这两个步骤的循环，直到无进位为止

1.关键位运算的作用

- `a ^ b`（异或）：实现 无进位相加

异或规则“相同为0，不同为1”，恰好对应二进制加法中“无进位时的结果”(如 `1^1=0`对应 `1+1`的无进位部分，`1^0=1` 对应 `1+0`的结果)

• `(a & b) << 1`（与运算 + 左移1位）：计算 进位值

与运算规则“全1为1，否则为0”，只有 `1&1` 会产生进位，左移1位后得到该位的进位值（如 `1&1=1`，左移1位为 10，对应`1+1`的进位 2）

2.递归终止条件

`if (!a) return b`;

当 `a`为0时，两数之和为b,返回b

三、代码分析

1.`func(b,c):`

• 9的二进制（32位）：`00000000 00000000 00000000 00001001`

• -7的二进制（32位补码）：`11111111 11111111 11111111 11111001`(原码取反+1）

- 计算进位：`（a & b)<<1`

  `a & b:00000000 00000000 00000000 00001001 & 11111111 11111111 11111111 11111001 = 00000000 00000000 00000000 00001001(18)`

- 计算无进位和：`a ^ b`

  `00000000 00000000 00000000 00001001 ^ 11111111 11111111 11111111 11111001 = 11111111 11111111 11111111 11110000(-16)`

  递归调用：`func(18,-16)`一直调用，直到`a = 0`,最终当`a`

`= 0`时，b的补码为`00000000 00000000 00000000 00000010`（2）

2.`func(a,func(b,c))`:

和步骤一一样不断递归调用，结果为6

```
6
```

6.自定义过滤

请实现filter()函数：过滤满足条件的数组元素。

提示：使用函数指针作为函数参数并且你需要为新数组分配空间。

```
typedef int (*Predicate)(int);
int *filter(int *array, int length, Predicate predicate,
            int *resultLength); /*补全函数*/
int isPositive(int num) { return num > 0; }
int main(void) {
    int array[] = {-3, -2, -1, 0, 1, 2, 3, 4, 5, 6};
    int length = sizeof(array) / sizeof(array[0]);
    int resultLength;
    int *filteredNumbers = filter(array, length, isPositive,
                                  &resultLength);
    for (int i = 0; i < resultLength; i++) {
      printf("%d ", filteredNumbers[i]);
    }
    printf("\n");
    free(filteredNumbers);
    return 0;
}
```

解析

要实现`filter()`函数，需完成以下步骤：

遍历原数组，通过函数指针判断元素是否满足条件；

统计满足条件的元素个数，作为新数组的长度；

为新数组分配内存，存储满足条件的元素；

通过结果长度通过指针传出，返回新数组地址

```
typedef int (*Predicate)(int);
// 过滤数组中满足条件的元素
int *filter(int *array, int length, Predicate predicate, int *resultLength) {
    // 第一步：统计满足条件的元素个数
    *resultLength = 0;
    for (int i = 0; i < length; i++) {
        if (predicate(array[i])) {  // 通过函数指针判断条件
            (*resultLength)++;
        }
    }
 // 第二步：为新数组分配内存
    int *result = (int *)malloc(*resultLength * sizeof(int));
    if (result == NULL) {
        *resultLength = 0;
        return NULL;
    }
 // 第三步：将满足条件的元素存入新数组
    int index = 0;
    for (int i = 0; i < length; i++) {
        if (predicate(array[i])) {
            result[index++] = array[i];
        }
    }
 return result;
}
```

7.静...态...

如何理解关键字static?

static与变量结合后有什么作用?

static与函数结合后有什么作用?

static与指针结合后有什么作用?

static如何影响内存分配?

解析

一、关键字`static`

`static`是C语言中用于控制变量/函数的生命周期、作用域、链接属性的关键字，核心是“静态化”——延长生命周期、限制访问范围，且仅初始化一次

二、`static`与变量结合（分全局/局部变量）

1. 静态全局变量(定义在全局作用域)

• 作用：限制作用域为当前源文件（默认全局变量作用域是整个程序，其他文件可通过`extern`访问；加`static`后，其他文件无法访问）

• 示例：

```
// file1.c
static int g_val = 10; // 静态全局变量，仅file1.c可见
// file2.c
extern int g_val; // 错误：无法访问file1.c的静态全局变量
```

2.静态局部变量（定义在函数内）

• 作用：

◦ 生命周期延长为整个程序运行期间（默认局部变量存储在栈区，函数结束后销毁；静态局部变量存储在静态数据区，程序启动时初始化，结束时释放）；

◦ 仅初始化一次（多次调用函数，不会重复初始化，保留上一次的值）；

◦ 作用域仍为当前函数（函数外无法访问）

• 示例：

```
int count() {
    static int num = 0; // 仅初始化1次，存储在静态数据区
    num++;
    return num;
}
```

调用`count()`3次，返回1、2、3（普通局部变量会返回1、1、1）

三、`static`与函数结合（静态函数）

• 作用：限制函数的链接属性为内部链接，仅当前源文件可调用（默认函数是外部链接，其他文件可通过`extern`声明后调用）

• 核心价值：避免不同文件中同名函数冲突

• 示例：

```
// file1.c
static void func() { printf("静态函数\n"); } // 仅file1.c可调用
// file2.c
extern void func(); // 错误：无法访问file1.c的静态函数
```

四、`static`与指针结合（静态指针）

• 本质：`static`修饰的是指针变量本身，而非指针指向的内容——指针变量存储在静态数据区（生命周期延长），作用域受定义位置限制（全局->当前文件，局部->当前函数），`static`不影响指针的指向

• 示例：

```
// 静态全局指针：仅当前文件可见，生命周期为整个程序
static int *p1;
void test() {
    static int a = 10;
    p1 = &a; // 指针指向静态局部变量（合法，a不会销毁）
// 静态局部指针：仅test()内可见，生命周期为整个程序
static int *p2 = &a;
(*p2)++; // 可修改指向的内容，a变为11
}
```

五、`static`对内存分配的影响

C程序内存分区中，`static`修饰的变量（静态全局/局部变量）统一存储在 静态数据区,而非栈区或堆区，具体规则：

1,分配时机：程序启动时（`main`函数执行前）完成分配，程序结束时由操作系统统一释放（无需手动管理）；

2.初始化规则：未显式初始化时，默认被初始化为0（或`NULL`，针对指针）,因静态数据区在程序加载时会被操作系统清零；

3.与其他分区的区别：

◦ 栈区：存储普通局部变量，函数结束后销毁，默认初始化随机值；

◦ 堆区：存储`malloc`分配的变量，需手动`free`，默认初始化随机值；

8.救命！指针！

数组指针是什么？指针数组是什么？函数指针呢？用自己的话说出来更好哦，下面数据类型的含义都是什么呢？

```
int (*p)[10];
const int* p[10];
int (*f1(int))(int*, int);
```

解析

一、核心概念

1.数组指针（指向数组的指针）

本质是指针，指向一个数组

2.指针数组（存储指针的数组）

本质是数组，数组里的每个元素都是指针

3.函数指针（指向函数的指针）

本质是指针，指向一个函数，通过它能找到对应函数并调用

二、题目分析

1.`int (*p)[10]`：数组指针

优先级：`()` > `[]`，`p`是一个指针，指针`p`指向的是一个包含10个`int`元素的数组；

2.`const int* p[10]`：指针数组

优先级：`[]` > `*`，`p`是一个包含10个元素的数组,数组的每个元素是“指向`const int`的指针”,指针可指向不同`const int`变量，但不能通过指针修改变量值`p`是一个数组，里面装了10个指针，每个指针都指向一个“不能修改的`int`变量”

3.`int (*f1(int))(int*, int)`：函数声明（返回函数指针的函数）

优先级：`()` > `*`，`f1`是一个函数，接收1个`int`类型参数；

`(*f1(int))`：`f1`的返回值是一个指针；

`(int*, int)`和`int`：该指针指向一个“接收`int*`和`int`参数、返回`int`的函数”

`f1`是一个函数（入参是`int`），调用后会返回一个函数指针，这个指针能指向另一个“接收指针和`int`、返回`int`的函数”

9.咋不循环了

程序直接运行，输出的内容是什么意思?

```
int main(int argc, char* argv[]) {
    printf("[%d]\n", argc);
    while (argc) {
      ++argc;
    }
    int i = -1, j = argc, k = 1;
    i++ && j++ || k++;
    printf("i = %d, j = %d, k = %d\n", i, j, k);
    return EXIT_SUCCESS;
}
```

解析

- `argc` 是命令行参数的个数（包含程序自身路径），程序直接运行时无额外参数，因此`argc=1`（仅程序名这1个参数），`printf` 输出 1
- 初始 `argc=1`，循环中 `++argc` 持续执行, 实际 `argc` 是 `int` 类型，递增到 `INT_MAX` 后溢出为 `INT_MIN`，最终 `argc=0` 时循环终止，最终 `argc=0` `i=-1`, `j=argc=0`, `k=1`
- `i++ && j++ || k++`

  先算 `i++`：先使用 `i=-1`（非0，为真），再 `i` 自增为 0；

因 `&&` 左侧为真，需计算右侧 `j++`： `j=0`（为假），再 `j` 自增为 1；

此时 `i++ && j++` 结果为真 `&&` 假 = 假，因 `||` 左侧为假，需计算右侧 `k++`：先使用 `k=1`（为真），再 `k` 自增为 2；

最终逻辑表达式结果为假 `||` 真 = 真，

最终输出：

```
[1]
i = 0, j = 1, k = 2
```

10.到底是不是TWO

```
#define CAL(a) a * a * a
#define MAGIC_CAL(a, b) CAL(a) + CAL(b)
int main(void) {
  int nums = 1;
  if(16 / CAL(2) == 2) {
    printf("I'm TWO(/>ω<)/\n");
  } else {
    int nums = MAGIC_CAL(++nums, 2);
  }
  printf("%d\n", nums);
}
```

解析

宏定义的本质：文本替换（无运算符优先级处理）

- `#define CAL(a) a * a * a`：直接替换参数，不添加括号；
- `#define MAGIC_CAL(a, b) CAL(a) + CAL(b)`：嵌套替换，先替换为 `a*a*a` + `b*b*b`

第一部分：`if(16 / CAL(2) == 2)`

- 宏替换过程：`CAL(2)` → `2*2*2`，表达式变为 `16 / 2*2*2`；

• 运算优先级：除法和乘法同级，从左到右执行 → `(16/2)*2*2` =`8*2*2` = 32,为假，执行else

• `int nums = MAGIC_CAL(++nums, 2);`

◦ 此处 `nums` 是 局部变量，覆盖外层全局`nums`；

◦ 宏替换：`MAGIC_CAL(++nums, 2)` ->`CAL(++nums) + CAL(2)` ->`++nums * ++nums * ++nums + 2*2*2`=16但这个局部nums仅在else里有效，else外仍为全局nums=1；

最终输出

```
1
```

11.克隆困境

试着运行一下程序，为什么会出现这样的结果?

直接将s2赋值给s1会出现哪些问题，应该如何解决？请写出相应代码。

```
struct Student {
    char *name;
    int age;
};

void initializeStudent(struct Student *student, const char *name,int age) {
    student->name = (char *)malloc(strlen(name) + 1);
    strcpy(student->name, name);
    student->age = age;
}
int main(void) {
    struct Student s1, s2;
    initializeStudent(&s1, "Tom", 18);
    initializeStudent(&s2, "Jerry", 28);
    s1 = s2;
    printf("s1的姓名: %s 年龄: %d\n", s1.name, s1.age);
    printf("s2的姓名: %s 年龄: %d\n", s2.name, s2.age);
    free(s1.name);
```

```
    free(s2.name);
    return 0;
}
```

解析

此时输出

```
s1的姓名: Jerry 年龄: 28
s2的姓名: Jerry 年龄: 28
```

结构体成员name是一个指针，当执行s1=s2时，发生的是浅拷贝，将s2的name的地址复制给s1的name，导致两个指针指向同一个地方，姓名输出一样，并且后面free会对这一块内存释放两次，导致程序崩溃，并且，第一个指针释放后另一个指针变成悬挂指针，访问该指针会出错

改正

```
struct Student {
    char *name;
    int age;
};

void initializeStudent(struct Student *student, const char *name,int age) {
    student->name = (char *)malloc(strlen(name) + 1);
    strcpy(student->name, name);
    student->age = age;
}
void copy(struct Student *dest,const struct Student *src){//表示将src中的数据复制到dest中
free(dest->name);
dest->name = (char *)malloc(strlen(src->name)+1);
strcpy(dest->name,src->name);
dest->age = src->age;
}
int main(void) {
    struct Student s1, s2;
    initializeStudent(&s1, "Tom", 18);
    initializeStudent(&s2, "Jerry", 28);
   copy(&s1,&s2);
    printf("s1的姓名: %s 年龄: %d\n", s1.name, s1.age);
    printf("s2的姓名: %s 年龄: %d\n", s2.name, s2.age);
    free(s1.name);
    free(s2.name);
```

```
    return 0;
}
```

12.你好，我是内存

作为一名合格的C-Coder，一定对内存很敏感吧~来尝试理解这个程序吧!

```
struct structure {
    int foo;
    union {
      int integer;
      char string[11];
      void *pointer;
    } node;
    short bar;
    long long baz;
    int array[7];
};
int main(void) {
    int arr[] = {0x590ff23c, 0x2fbc5a4d, 0x636c6557, 0x20656d6f,
                 0x58206f74, 0x20545055, 0x6577202c, 0x6d6f636c,
                 0x6f742065, 0x79695820, 0x4c20756f, 0x78756e69,
                 0x6f724720, 0x5b207075, 0x33323032, 0x7825005d,
                 0x636c6557, 0x64fd6d1d};
    printf("%s\n", ((struct structure *)arr)->node.string);
}
```

解析

结构体中，共用体在`int foo`之后，`int foo`占4字节内存，所以，共用体`node`的起始地址相较于结构体起始地址偏移4字节，数组`arr`的每个元素是4字节，`node`对应`arr`索引从1开始，`arr[0]=foo`,`arr[1]`及以后对应`node`

`node.string`对应字节为11,但字符串输出要超过数组定义长度，直到读取到`\0`，因为`printf`只认`\0`作结束符，不关心数组长度

x86架构默认小端序，将最低有效字节存储在内存最低地址

`arr[1]`拆分成`0x4d`,`0x5a`,`0xbc`,`0x2f`

对应ASCII分别为`M`，`Z`，`0xbc`,`/`

`arr[2]`拆分成`0x57`,`0x65`,`0x6c`,`0x63`

对应ASCII分别为`W`,`e`,`l`,`c`

`arr[3]`拆分成`0x6f`,`0x6d`,`0x65`,`0x20`

对应ASCII分别为`o`,`m`,`e`,

`arr[4]`拆分成`0x74`,`0x6f`,`0x20`,`0x58`

对应ASCII分别为`t`,`o`,,`X`

`arr[5]`拆分成`0x55`,`0x50`,`0x54`,`0x20`

对应ASCII分别为`U`,`P`,`T`,

`arr[6]`拆分成`0x2c`,`0x20`,`0x77`,`0x65`

对应ASCII分别为`,`,,`w`,`e`

`arr[7]`拆分为`0x6c`,`0x63`,`0x6f`,`0x6d`

对应ASCII分别为`l`,`c`,`o`,`m`

`arr[8]`拆分为`0x65`,`0x20`,`0x74`,`0x6f`

对应ASCII分别为`e`,,`t`,`o`

`arr[9]`拆分为`0x20`,`0x58`,`0x69`,`0x79`

对应ASCII分别为,`X`,`i`,`y`

`arr[10]`拆分为`0x6f`,`0x75`,`0x20`,`0x4c`

对应ASCII分别为`o`,`u`,,`L`

`arr[11]`拆分为`0x69`,`0x6e`,`0x75`,`0x78`

对应ASCII分别为`i`,`n`,`u`,`x`

`arr[12]`拆分为`0x20`,`0x47`,`0x72`,`0x6f`

对应ASCII分别为,`G`,`r`,`o`

`arr[13]`拆分为`0x75`,`0x70`,`0x20`,`0x5b`

对应ASCII分别为`u`,`p`,,`[`

`arr[14]`拆分为`0x32`,`0x30`,`0x32`,`0x33`

对应ASCII分别为`2`,`0`,`2`,`3`

`arr[15]`拆分为`0x5d`,`0x00`,`0x25`,`0x78`

对应ASCII分别为`]`,`\0`……

拼接`node.string`的有效字符:

`Welcome to XUPT , welcome to Xiyou Linux Group [2023]`

从连续字符W开始算起有效字符为`welcome`

最终输出:

```
Welcome to XUPT , welcome to Xiyou Linux Group [2023]
```

13.GNU/Linux(选做)

一、cd命令用法

cd（change directory）用于切换当前工作目录，核心语法简洁，常用场景如下：

• 切换到绝对路径：cd /home/user（从根目录开始定位目标目录）；

• 切换到相对路径：cd ../docs（从当前目录向上一级再进入 docs 目录）；

• 切换到用户主目录：cd 或 cd ~（直接回到当前登录用户的家目录，如 /home/ubuntu）；

• 切换到上一次工作目录：cd -（快速在两个目录间切换）；

• 切换到当前目录的子目录：cd test（直接进入当前目录下的 test 文件夹）

二、/ . ~符号含义

• /（根目录符号）：Linux 文件系统的最顶层目录，所有文件和目录都挂载在 / 下（如 /etc、/usr 均为根目录的子目录），绝对路径必须以 / 开头；

• .（当前目录符号）：表示当前所在的工作目录，例如 ./script.sh 表示执行当前目录下的 script.sh 脚本；

• ~（主目录符号）：等价于当前登录用户的家目录路径，普通用户通常为 /home/用户名，root 用户为 /root，例如 cd ~/Downloads 直接进入下载目录

三、常用Linux核心知识拓展

1. 文件权限管理：用 chmod 命令修改权限（如 chmod 755 file 赋予读/写/执行权限），rwx 分别对应读（4）、写（2）、执行（1）；
2. 文件查看命令：cat（查看完整文件）、less（分页查看大文件）、grep（搜索文件内容，如 grep "error" log.txt）；
3. 进程管理：ps aux 查看所有进程，kill -9 进程ID 强制终止进程；
4. 包管理工具：Debian 系（Ubuntu）用 apt（如 apt install nginx），RedHat 系（CentOS）用 yum（如 yum install gcc）；
5. 目录结构特点：/etc 存储系统配置文件，/usr 存放系统软件，/var 存储日志和缓存文件