Lec19: Linked data structures & C to LC3

C 语言到 LC-3 示例：寻找绝对值

目标: 编写一个 C 函数，计算一个 32 位整数的绝对值，并将其转换为 LC-3 代码 。

C 语言函数:

```c
int32_t find_abs (int32_t num)
{
    int32_t abs_value;
    // 使用条件运算符判断 num 是否大于等于 0
    // 如果是，abs_value = num
    // 如果否，abs_value = -num
    abs_value = (0 <= num ? num : -num);
    return abs_value;
}
```

- 函数签名:

  ```c
  int32_t find_abs (int32_t num)
  ```

  - 函数名称: `find_abs`
  - 输入参数: `int32_t num` (一个 32 位整数)
  - 返回值: `int32_t` (计算出的绝对值)
- 核心逻辑:

  使用三元条件运算符

  ```c
  (condition ? value_if_true : value_if_false)
  ```

  来计算绝对值

  - 测试条件: `0 <= num` (或者 `num >= 0`)
  - 如果为真 (num 是非负数): `abs_value` 被赋值为 `num` 。
  - 如果为假 (num 是负数): `abs_value` 被赋值为 `-num` 。

任务流程:

1. 让用户输入一个数字。
2. 调用 `find_abs` 函数将数字转换为其绝对值。
3. 以十六进制格式打印结果。

内存分配与数据结构

结构体定义 (struct flightType):

```
struct flightType {
    char ID[7];        // 飞机ID（最多6个字符 + '\0'）
    int altitude;      // 海拔（米）
    int longitude;     // 经度（十分之一度）
    int latitude;      // 纬度（十分之一度）[cite: 13]
    int heading;       // 航向（十分之一度）
    double airSpeed;   // 空速（公里/小时）
};
```

变量声明:

```
int x;
struct flightType plane; // 声明一个 flightType 类型的结构体变量 plane
int y;
```

内存布局示例: (假设变量按声明顺序分配)

- `y` (int)
- `plane.ID[0]` 到 `plane.ID[6]` (char array)
- `plane.altitude` (int)
- `plane.longitude` (int)
- `plane.latitude` (int)
- `plane.heading` (int)
- `plane.airSpeed` (double) - *注意：double 类型的大小可能因系统而异*
- `x` (int)

链式数据结构 (Linked Lists)

1. 链表节点的内存表示

节点结构体定义 (node):

```
typedef struct Node {
    char symbol;       // 存储的字符数据
    struct Node *next; // 指向下一个节点的指针
} node;
```

示例代码:

```
int main() {
    node N3, N2, N1; // 声明三个节点变量

    N1.symbol = 'E';
    N2.symbol = 'C';
    N3.symbol = 'E';

    N1.next = &N2;     // N1 指向 N2
    N2.next = &N3;     // N2 指向 N3
    N3.next = NULL;    // N3 是链表末尾，指向 NULL [cite: 18]
}
```

内存表示示例: (假设 N1, N2, N3 的起始地址分别为 x6000, x6002, x6004)

变量	地址	值	符号	描述
N1	x6000	'E'	N1.symbol	N1 存储的字符
	x6001	x6002	N1.next	N1 指向下个节点 N2 的地址
N2	x6002	'C'	N2.symbol	N2 存储的字符
	x6003	x6004	N2.next	N2 指向下个节点 N3 的地址
N3	x6004	'E'	N3.symbol	N3 存储的字符
	x6005	NULL	N3.next	N3 是末尾，next 指针为 NULL

2. 打印链表 (Recursive C Function)

函数定义:

```
void print_list (node *cursor) {
    if (cursor == NULL) { // 基本情况：如果链表为空或到达末尾
        /* List empty; do nothing */
        return; // 直接返回 [cite: 21, 25]
    } else { // 递归情况：链表非空
        /* Print and recurse */
        printf("%c", cursor->symbol); // 打印当前节点的数据 [cite: 21]
        print_list(cursor->next);    // 递归调用，处理下一个节点 [cite: 21]
    }
}
```

- 这是一个递归函数，用于遍历并打印链表中的所有字符。
- **cursor**: 一个指向当前节点的指针。
- **基本情况**: 当 cursor 为 NULL 时，表示已到达链表末尾或链表本身为空，递归结束。

- **递归步骤**: 打印当前 `cursor` 指向节点的数据 (`cursor->symbol`)，然后递归调用 `print_list` 处理链表的剩余部分 (`cursor->next`)。

3. 打印链表 (LC-3 实现)

这部分详细展示了如何将上述 `print_list` C 函数转换为 LC-3 汇编代码，并涉及了函数调用约定（Call Convention）、栈帧（Stack Frame）的建立和销毁。

LC-3 主程序段 (部分):

代码段

```
.ORIG x3000
MAIN
    LD R5, RSTACK      ; 加载栈底指针到 R5 (Frame Pointer)
    LD R6, RSTACK      ; 加载栈顶指针到 R6 (Stack Pointer)
    LD R0, HEAD        ; 加载链表头指针到 R0
    STR R0, R6, #0     ; 将头指针（参数）压入栈中
    ADD R6, R6, #-1    ; 移动栈顶指针
    JSR PRINT_LIST     ; 调用打印链表的子程序
    ADD R6, R6, #2     ; 调用者清理栈（弹出返回值和参数）
    HALT

HEAD   .FILL x4004     ; 链表头指针的内存地址（示例）
RSTACK .FILL x7000     ; 运行时栈的起始地址（示例）
```

`PRINT_LIST` 子程序 (LC-3 汇编 - 概述):

1. Callee Setup (被调用者建立栈帧):
    - 保存返回地址 (R7) 到栈中。
    - 保存调用者的帧指针 (R5) 到栈中。
    - 设置新的帧指针 (R5)。
2. 函数逻辑实现:
    - **检查 `cursor == NULL`:** 加载参数 (链表节点指针) 到寄存器 (如 R1) 。如果指针为 0 (NULL)，则跳转到结束部分 (TEAR_DOWN) 。
    - **`printf("%c", cursor->symbol)`:** 如果指针非空，加载 `cursor->symbol` (节点数据) 到 R0 ，然后调用 `OUT` TRAP 指令打印字符 。
    - `print_list(cursor->next)` (递归调用准备):
        - 加载 `cursor->next` (下一个节点的指针) 到寄存器 (如 R1) 。
        - 将 `cursor->next` 作为参数压入栈中 。
        - 使用 `JSR PRINT_LIST` 进行递归调用 。
        - **Caller Teardown (调用者清理递归调用的栈):** 弹出递归调用的参数 。

3. Callee Teardown (被调用者销毁栈帧):
    - 恢复调用者的帧指针 (R5) 。
    - 恢复返回地址 (R7) 。
    - 调整栈顶指针 (R6) 。
    - 使用 `RET` 指令返回到调用者 。

运行时栈 (Runtime Stack) 示例:

- 课件通过图示展示了在调用 `PRINT_LIST` 时，参数 (链表头指针) 如何被压入栈中 。栈从高地址向低地址增长。

内存数据示例 (链表节点):

地址	值 (十六进制)	值 (十进制)	解释
x4004	x0045	69	'E' (节点1数据)
x4005	x4006	16390	指向节点2的地址 (x4006)
x4006	x0043	67	'C' (节点2数据)
x4007	x4008	16392	指向节点3的地址 (x4008)
x4008	x0045	69	'E' (节点3数据)
x4009	x400A	16394	指向节点4的地址 (x400A)
x400A	x0032	50	'2' (节点4数据)
x400B	x400C	16396	指向节点5的地址 (x400C)
x400C	x0032	50	'2' (节点5数据)
x400D	x400E	16398	指向节点6的地址 (x400E)
x400E	x0030	48	'0' (节点6数据)
x400F	x0000	0	NULL (链表末尾)

链表练习

合并两个已排序的链表:

1. **任务:** 给定两个已排序的链表，合并它们并返回合并后链表的头指针 。
    - **朴素方法 (Naive):** 使用数组。时间复杂度 O((n+m)log(n+m))，空间复杂度 O(n+m)。
    - **较好方法 (Better):** 使用递归合并。时间复杂度 O(n+m)，空间复杂度 O(n+m) (递归栈)。
    - **高效方法 (Efficient):** 使用迭代合并 (原地合并)。时间复杂度 O(n+m)，空间复杂度 O(1) 。
    - **原地合并步骤概述:** 维护两个指针分别指向两个链表的当前节点，比较节点值，将较小的节点链接到结果链表中，并移动相应指针，直到一个链表遍历完，然后将另一个链表的剩余部分链接到结果链表末尾。
2. 反转单向链表:

- **迭代法 (Iterative):** 推荐方法。时间复杂度 O(n)，空间复杂度 O(1)。需要维护 `prev`, `current`, `next` 三个指针来逐个反转节点的指向。
- **递归法 (Recursive):** 时间复杂度 O(n)，空间复杂度 O(n) (递归栈)。
- **栈法 (Stack):** 时间复杂度 O(n)，空间复杂度 O(n)。将所有节点压入栈，然后依次弹出构建新链表。

3. 实现双向链表:
   - 双向链表节点包含指向前一个节点 (`prev`) 和后一个节点 (`next`) 的指针。

双向链表操作：删除指定位置的节点

步骤:

1. **遍历:** 找到要删除的节点，记为 `curr` 。
2. 调整指针 (核心):
   - **如果 curr 不是头节点:** 将 `curr` 的前一个节点的 `next` 指针指向 `curr` 的后一个节点 (`curr->prev->next = curr->next`) 。
   - **如果 curr 不是尾节点:** 将 `curr` 的后一个节点的 `prev` 指针指向 `curr` 的前一个节点 (`curr->next->prev = curr->prev`) 。
3. **释放内存:** 释放 `curr` 节点占用的内存 。

更多链表问题资源

- 斯坦福大学 Nick Parlante 的《Linked list problems》提供了更多链表相关的练习题 。