

程序设计基础

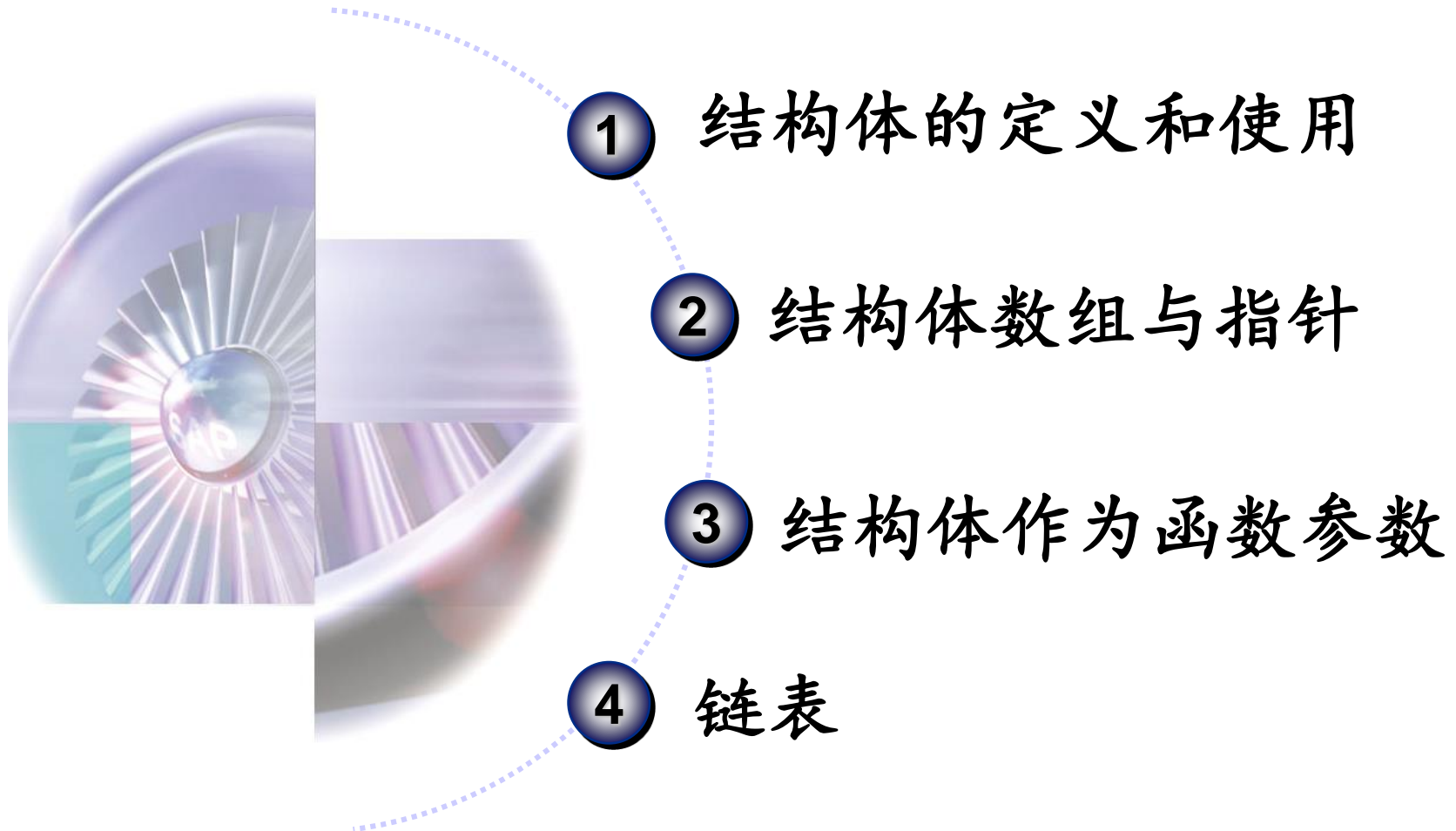
Fundamental of Programming

清华大学软件学院

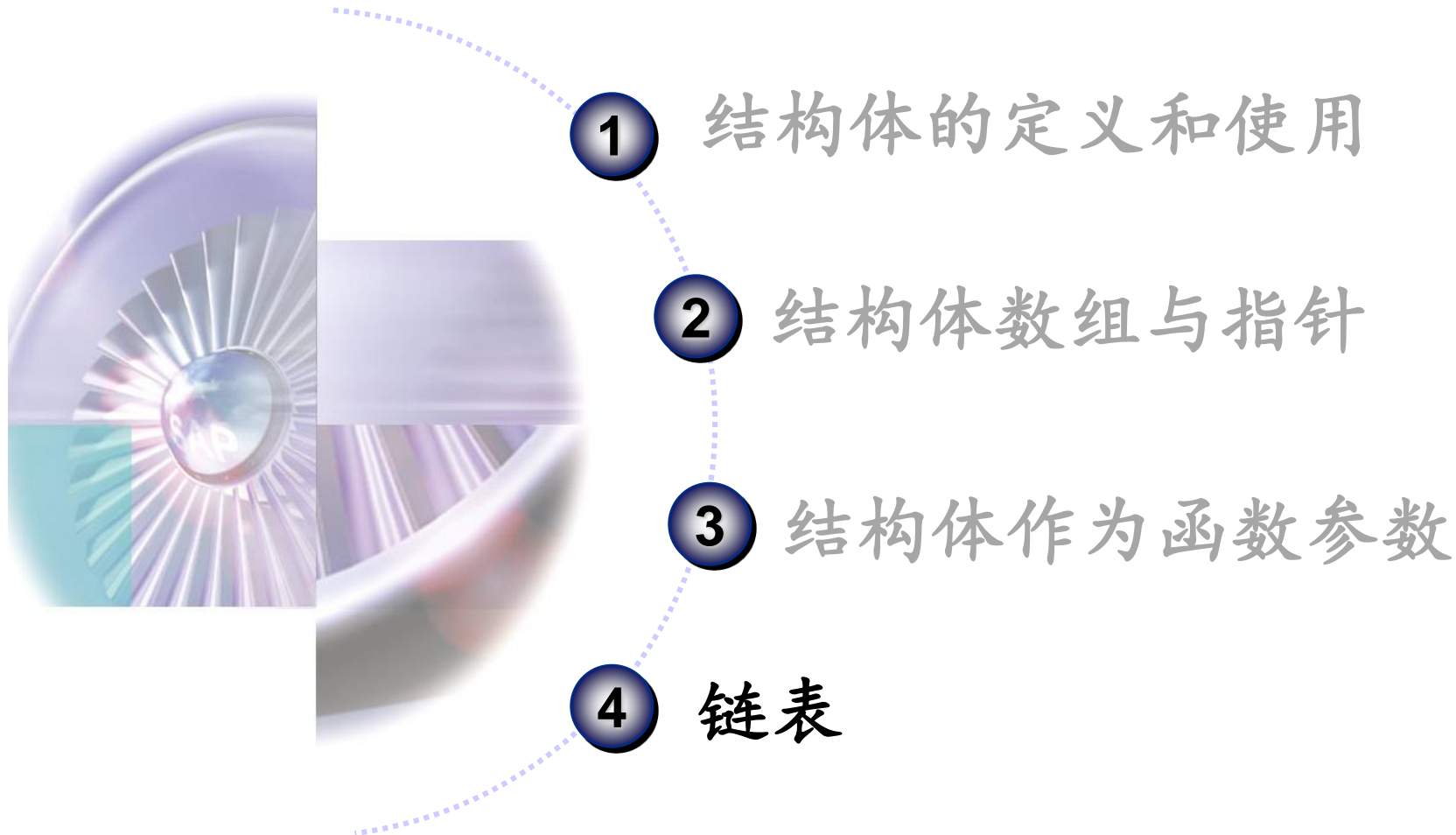
刘玉身

liuyushen@tsinghua.edu.cn

Lecture 7: 结构体



Lecture 7: 结构体



why链表？

姓名	体力	智力	武力	魅力	运气
新君主	84	91	90	90	85
随从	80	80	80	80	80
关羽	95	88	95	85	80
鲁肃	70	90	85	80	85
夏侯惇	90	80	92	75	80
...					

1. 单个人物

2. 所有人物

需求分析

- 目标：把若干个结构体变量组织在一起
- 可能会显示部分/全部变量；
- 可能会动态增加新变量；
- 可能会动态减少变量。

结构体数组？

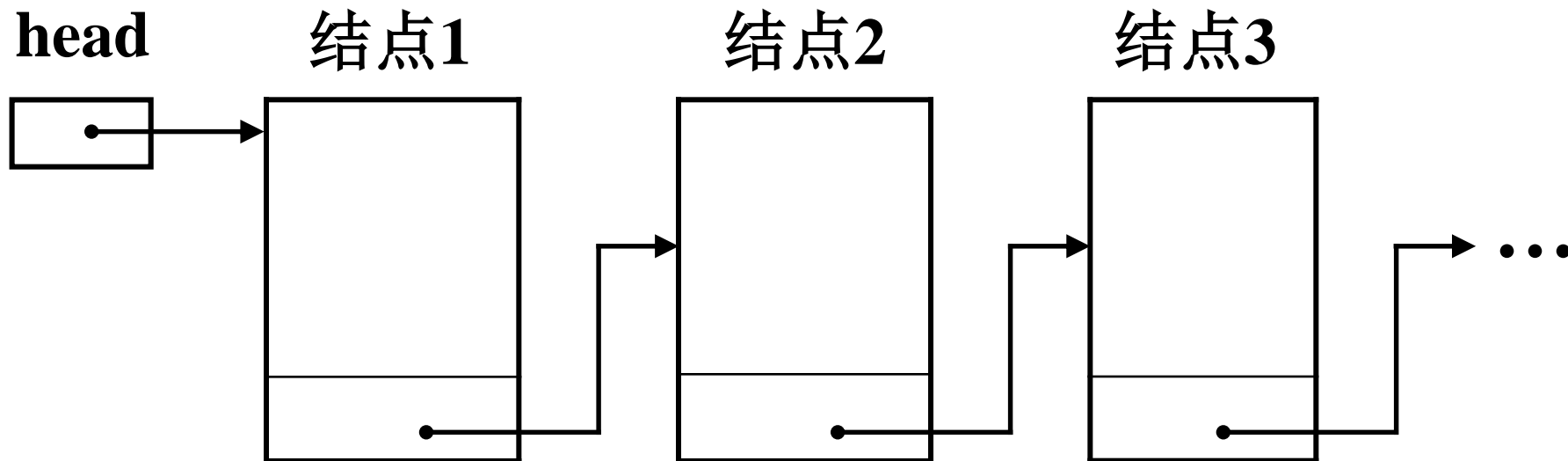
什么是链表？



如何在程序中来描述一列火车？

- 如何来描述火车头？
- 如何来描述每一节车厢？
- 如何来描述各节车厢之间的链接关系？


需要引入一种新的数据结构：链表。链表中的每个元素称为一个“结点”，每个结点包括两部分：一为数据；二为下一结点的起始地址。另外，链表还有一个头指针**head**，指向首结点。





如何实现链表结构？可定义如下结构体类型：

```
struct  Train
{
    char  Num[8];
    char  Name[10];
    int   Weight;
    char  From[20];
    char  To[20];
    struct Train *next;
};
```



数据域

指针域

创建链表

在程序中为链表的每一个结点动态地分配相应的存储空间，并把它们链接成一个链表的形式。

【问题描述】

创建一个人物链表，并输入每一个结点的各种描述信息（姓名、体力、智力、武力、魅力、运气等），直到用户输入的人物姓名为`#`，表示链表结束。

```
struct General
```

```
{
```

```
    char Name[20];           // 姓名
```

```
    int Body;                // 体力
```

```
    int Intelligence;        // 智力
```

```
    int Power;               // 武力
```

```
    int Charisma;            // 魅力
```

```
    int Luck;                // 运气
```

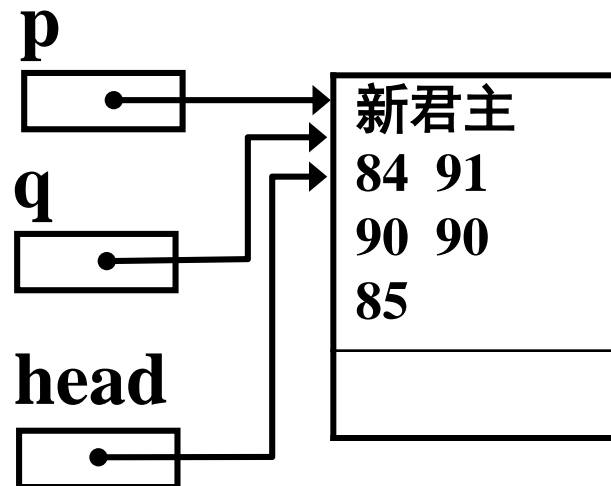
```
    struct General *next;
```

```
};
```

```
struct General *head, *p, *q;
```

创建链表的过程可归纳为如下三个步骤

- ① 基本思路是将一个一个的结点添加至链表中。首先用指针 p 来申请一个结构体变量的内存空间，并且装入用户输入的各种描述信息，然后将指针 q 和 head 都指向它。如下图：



链表的第一个 结点建成

- (1) p, q, head 和结点分别存放在哪？
- (2) 编程实现上述步骤
- (3) 如何访问动态分配的结构体变量？

② **后继结点的创建**：若用户输入的姓名不为空，则要构建第二个结点。先用 p 来申请一段内存空间，并装入用户输入的各种描述信息，然后把第一个结点的next指针去指向它，从而建立两个结点之间的链接关系。最后再把 q 指向新的结点。如下图：

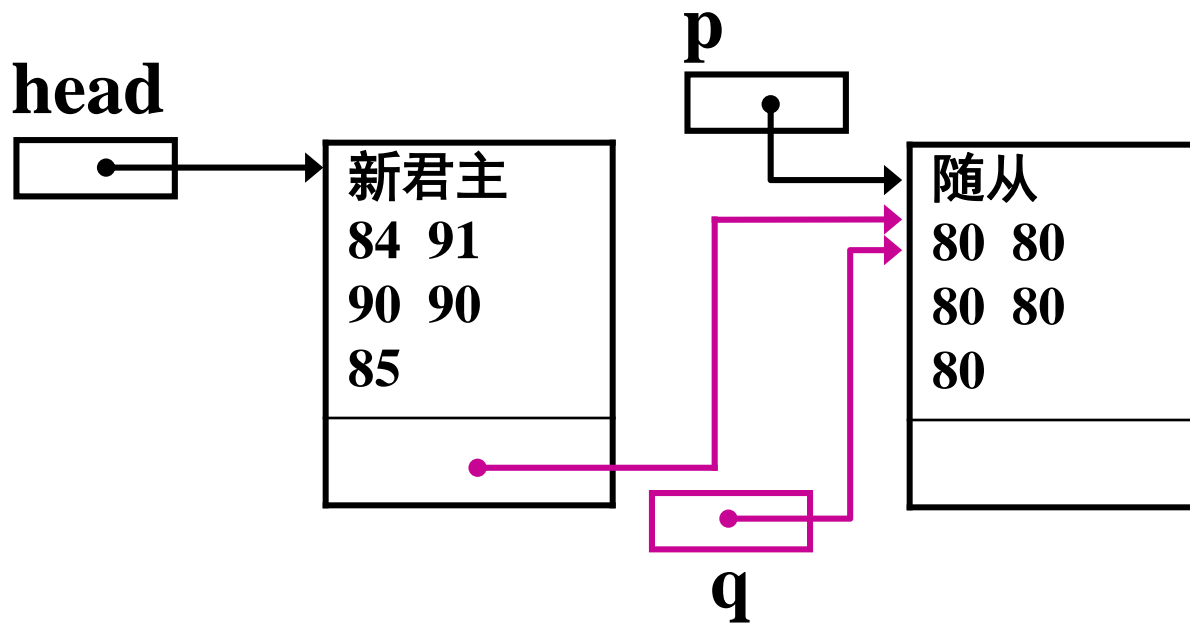
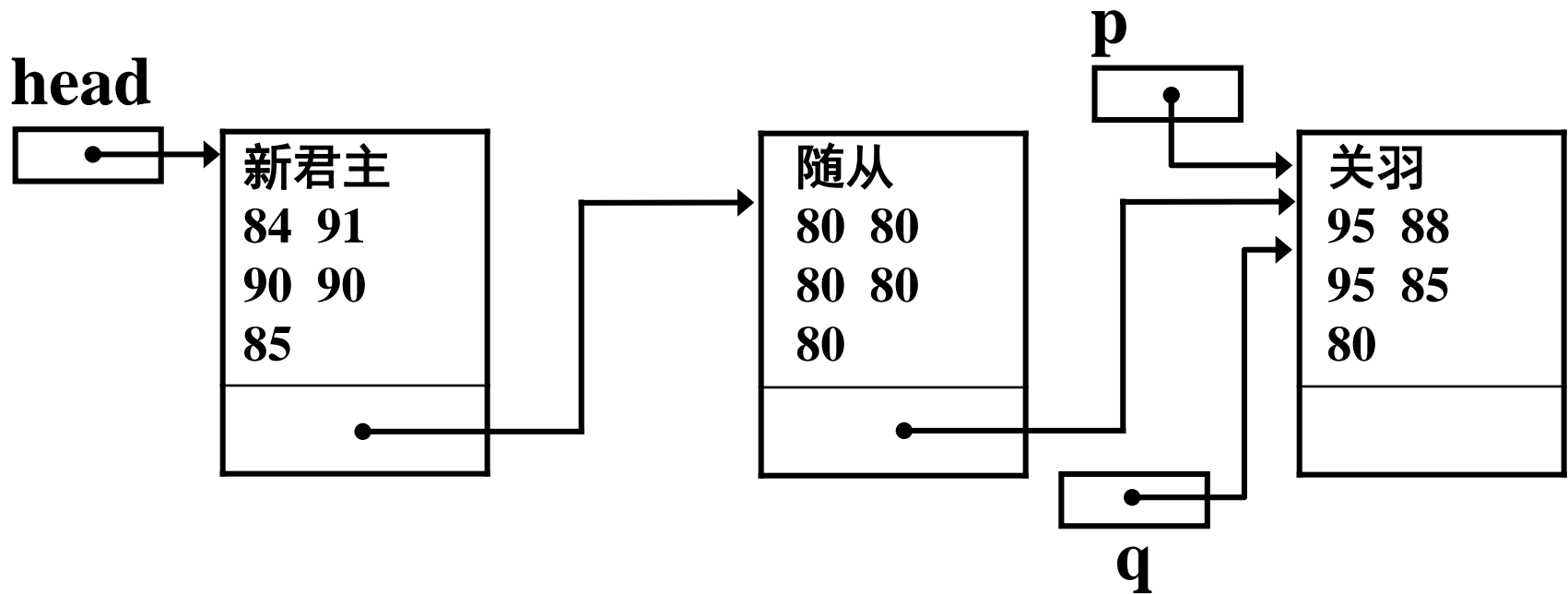


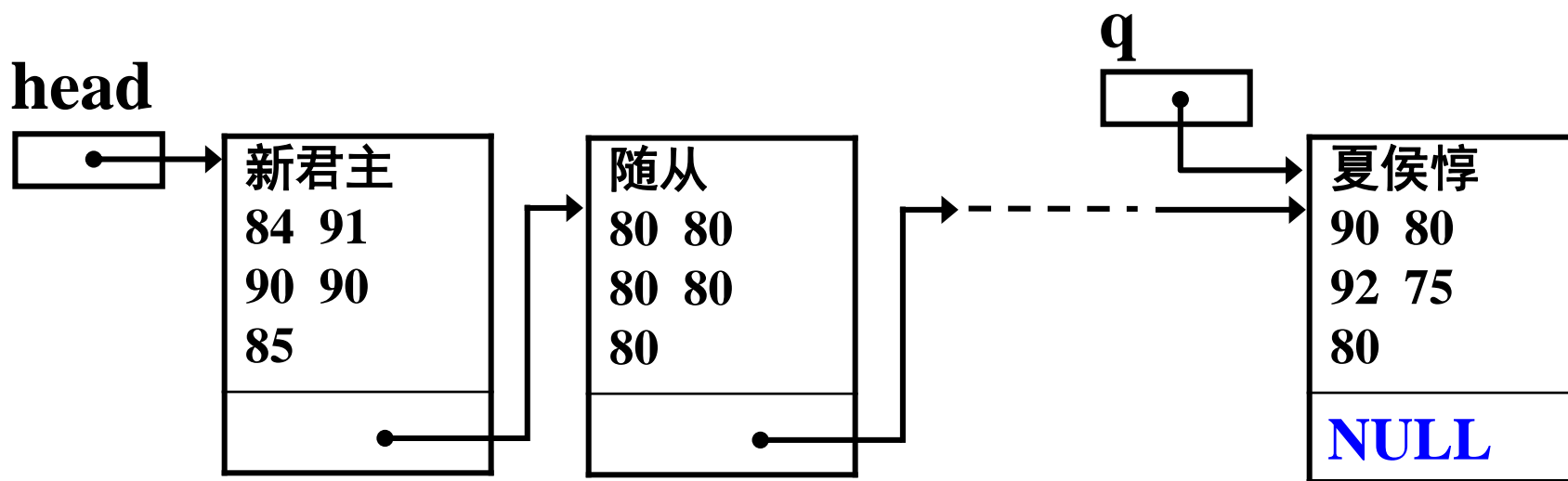
图 链表的第二个结点建成

如何编程实现这种链接关系？

第三个结点加入链表的过程：



③ 链表创建过程的结束：如果用户输入的姓名为空，意味着链表创建过程的结束，此时指针 q 所指向的就是链表的最后一个结点，所以要把该结点的next指针赋值为NULL，表示这里已是链尾。如下图：



```
struct General *CreateList( )
```

```
{
```

```
    char name[20];    struct General *head, *p, *q;
```

```
    head = p = q = NULL;
```

```
    while( 1 ){
```

```
        printf("输入人物姓名: ");
```

```
        scanf("%s", name);
```

```
        if(name[0] == '#') break;
```

```
        p = (struct General*)malloc(sizeof(struct General));
```

```
        strcpy(p->Name, name);
```

```
        // 输入该结点的其他信息
```

```
        scanf("%d %d %d %d %d", &p->Body, &p->Intelligence,
```

```
                &p->Power, &p->Charisma, &p->Luck);
```

```
        if (head == NULL) // 新建的是首结点
```

```
            { head = p;    q = p; }
```

```
        else // 不是首结点
```

```
            { q->next = p;  q = p; }
```

```
    }
```

```
    if (q != NULL) q->next = NULL;
```

```
    return(head);
```

```
}
```

代码测试:

1. 空链表情形
2. 单结点情形
3. 双结点情形
4. 多结点情形

函数结束后链表是否还在?

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
struct General
{
    char Name[20];    // 姓名
    int Body;         // 体力
    int Intelligence; // 智力
    int Power;        // 武力
    int Charisma;     // 魅力
    int Luck;         // 运气
    struct General *next;
};
struct General *CreateList( );
void print_list(struct General *head);
```

```

int main ()
{
    struct General *head;
    head = CreateList();
    print_list(head);
    return 0;
}

void print_list(struct General *head)
{
    struct General *p = head;
    while(p != NULL)
    {
        printf("%-6s %d %d %d %d %d\n", p->Name, p->Body,
            p->Intelligence, p->Power, p->Charisma, p->Luck);
        p = p->next;
    }
}

```

```

新君主 84 91 90 90 85
随从   80 80 80 80 80
关羽   95 88 95 85 80
夏侯惇 90 80 92 75 80
#

```

```

新君主 84 91 90 90 85
随从   80 80 80 80 80
关羽   95 88 95 85 80
夏侯惇 90 80 92 75 80

```

QuickWatch

Expression: `(((*head).next)).next).next`

Value:

Reevaluate Add Watch

Name	Value	Type
head	0x003b3690 {Name=0x003b3690	General *
Name	0x003b3690 "新君主"	char [20]
Body	84	int
Intelligence	91	int
Power	90	int
Charisma	90	int
Luck	85	int
next	0x003b5fa8 {Name=0x003b5fa8	General *
Name	0x003b5fa8 "随从"	char [20]
Body	80	int
Intelligence	80	int
Power	80	int
Charisma	80	int
Luck	80	int
next	0x003b6010 {Name=0x003b6010	General *
Name	0x003b6010 "关羽"	char [20]
Body	95	int
Intelligence	88	int
Power	95	int
Charisma	85	int
Luck	80	int
next	0x003b6078 {Name=0x003b6078	General *
Name	0x003b6078 "夏侯惇"	char [20]
Body	90	int
Intelligence	80	int
Power	92	int
Charisma	75	int
Luck	80	int
next	0x00000000 {Name=0x00000000	General *

Close Help

【例1】

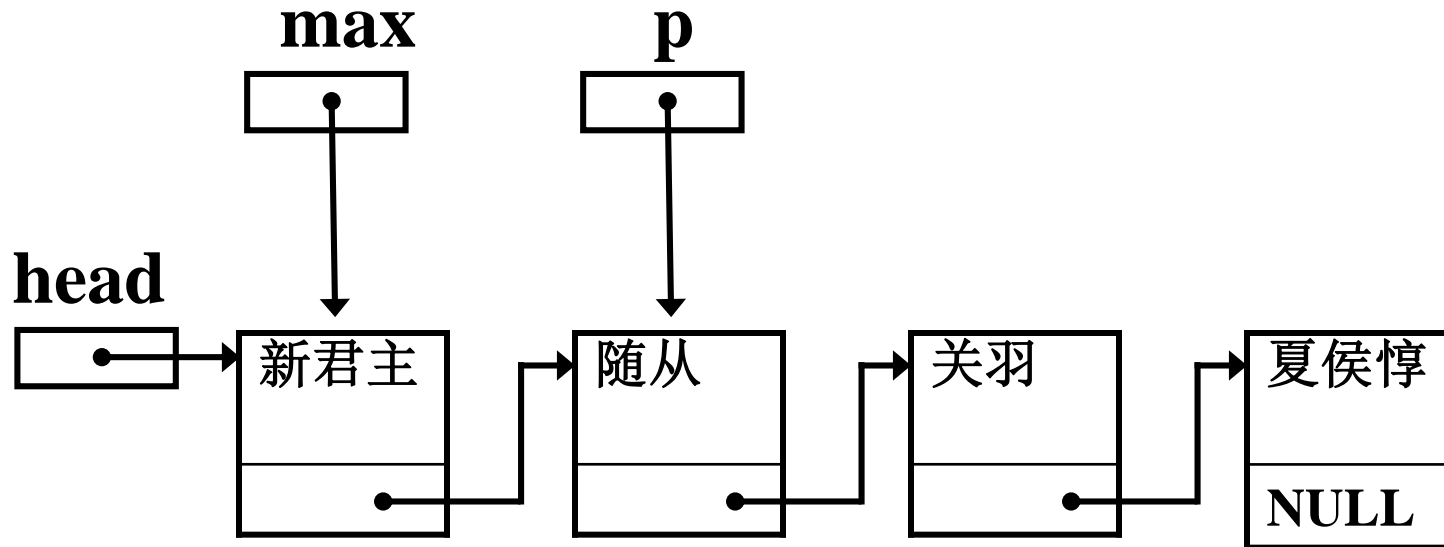
大战在即，新君主需要选拔一名武力最强的将领，率领先锋部队，挺进中原。

请编写一个程序，帮助新君主完成此次选拔任务。

问题分析

1. “武力最强”：求最大值/最小值编程模式。
2. 如何针对链表进行改进？

```
struct General *FindMostPowerful(struct General *head)
{
    struct General *p, *max;
    if(head == NULL)    return NULL;
    max = head; // max指向武力最强的结点
    p = head->next;
    while(p != NULL)
    {
        if(p->Power > max->Power)    max = p;
        p = p->next; ← 能否 p++;
    }
    printf("%s, %d, %d, %d, %d, %d\n",
           max->Name, max->Body, max->Intelligence,
           max->Power, max->Charisma, max->Luck);
    return max;
}
```

```
while(p != NULL)  
{  
    if(p->Power > max->Power) max = p;  
    p = p->next;  
}
```

```

int main ()
{
    struct General *head, *max, *p;
    head = CreateList();
    print_list(head);

    max = FindMostPowerful(head);
    if (max != NULL)
    {
        p = max;
        printf("武力最强的是: %s %d %d %d %d %d\n",
            p->Name, p->Body, p->Intelligence, p->Power,
            p->Charisma, p->Luck);
    }
    return 0;
}

```

```

新君主 84 91 90 90 85
随从   80 80 80 80 80
关羽   95 88 95 85 80
夏侯惇 90 80 92 75 80
#

```

```

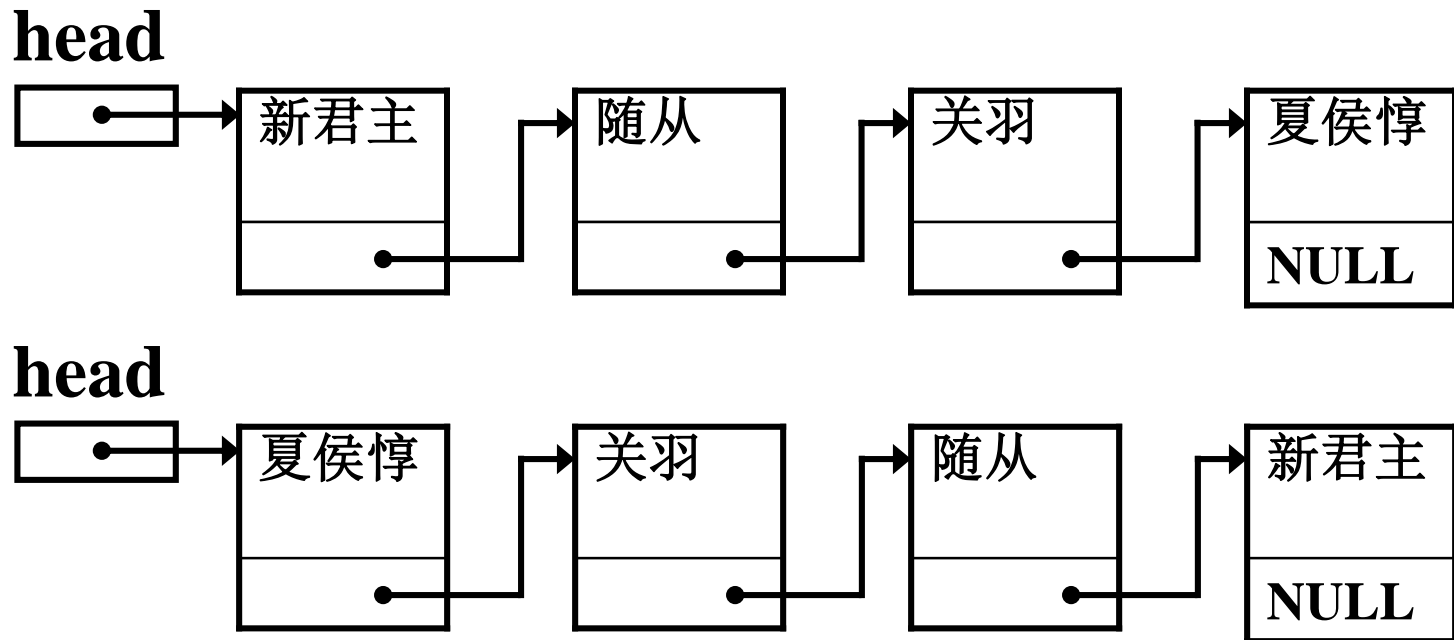
武力最强的是:
关羽   95 88 95 85 80

```

链表反转

【例2】

编写一个程序，将一条链表的各个结点反转过来。



"level"

l	e	v	e	l	'\0'
----------	----------	----------	----------	----------	-------------

str 0 1 2 3 4 5

1. str[0] ↔ str[4]: 'l' ↔ 'l';

2. str[1] ↔ str[3]: 'e' ↔ 'e';

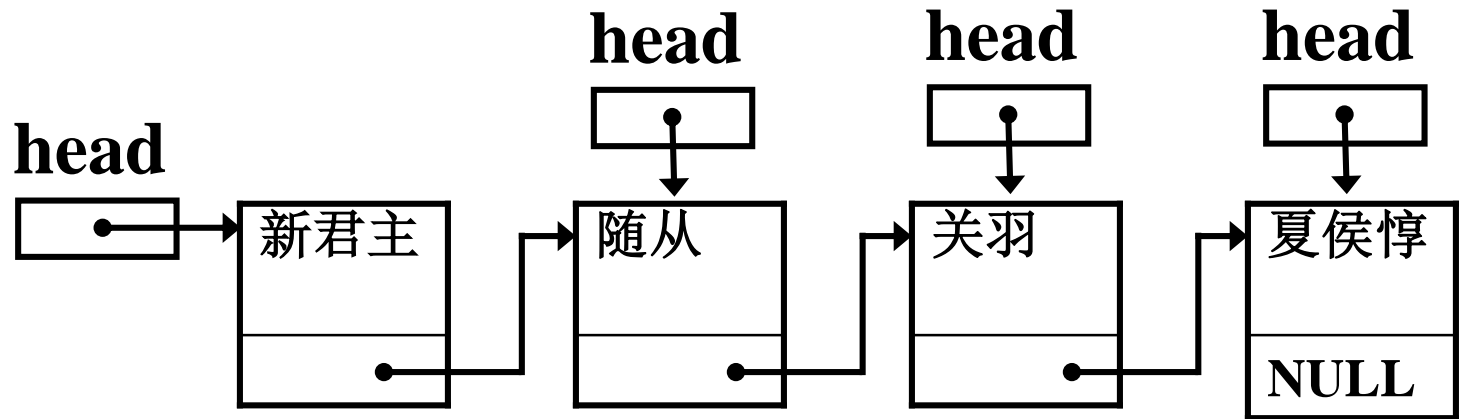
3. str[2] ↔ str[2]

4. 完成!

能否应用于链表?

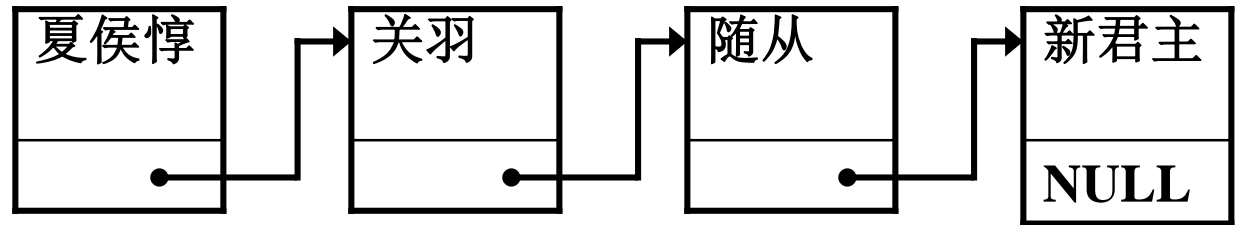
算法设计

1. 基本思路：仿照创建链表的过程，但是逆序排列。
2. 逐一访问每一结点，把它加入到新链表中。



基本思路:

1. 结点数据域不变
2. 结点指针域改变
3. 新链表头处理
4. 尾结点处理



```

struct General *ReverseList( struct General *head1)
{
    struct General *head2, *p;

    if(head1 == NULL)    return NULL;
    head2 = NULL;
    while(head1 != NULL)
    {
        p = head1->next;
        head1->next = head2; // 新增结点处理
        head2 = head1;      // 新链表头处理
        head1 = p;
    }
    return(head2);
}

```

head1: 原链表头结点
head2: 新链表头结点

```
int main ()
{
    struct General *head, *head2;
    head = CreateList();
    print_list(head);

    head2 = ReverseList(head);
    printf("反转链表: \n");
    print_list(head2);
    return 0;
}
```

```
新君主 84 91 90 90 85
随从   80 80 80 80 80
关羽   95 88 95 85 80
夏侯惇 90 80 92 75 80
#
```

```
反转链表:
夏侯惇 90 80 92 75 80
关羽   95 88 95 85 80
随从   80 80 80 80 80
新君主 84 91 90 90 85
```

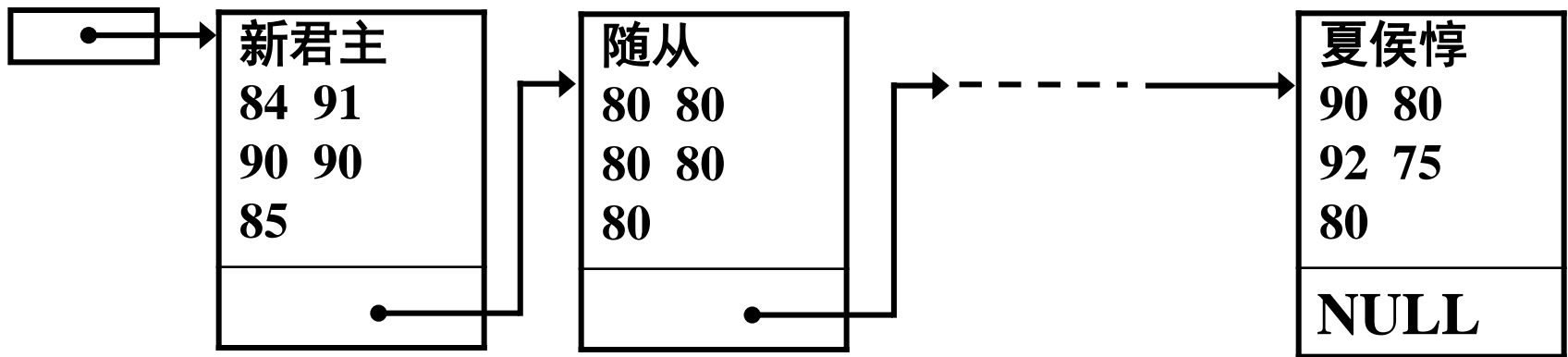

链表排序

【例3】

新君主想全面了解众将领的各项技术指标，请编写一个程序，能根据某个单项指标，对所有将领进行排序。

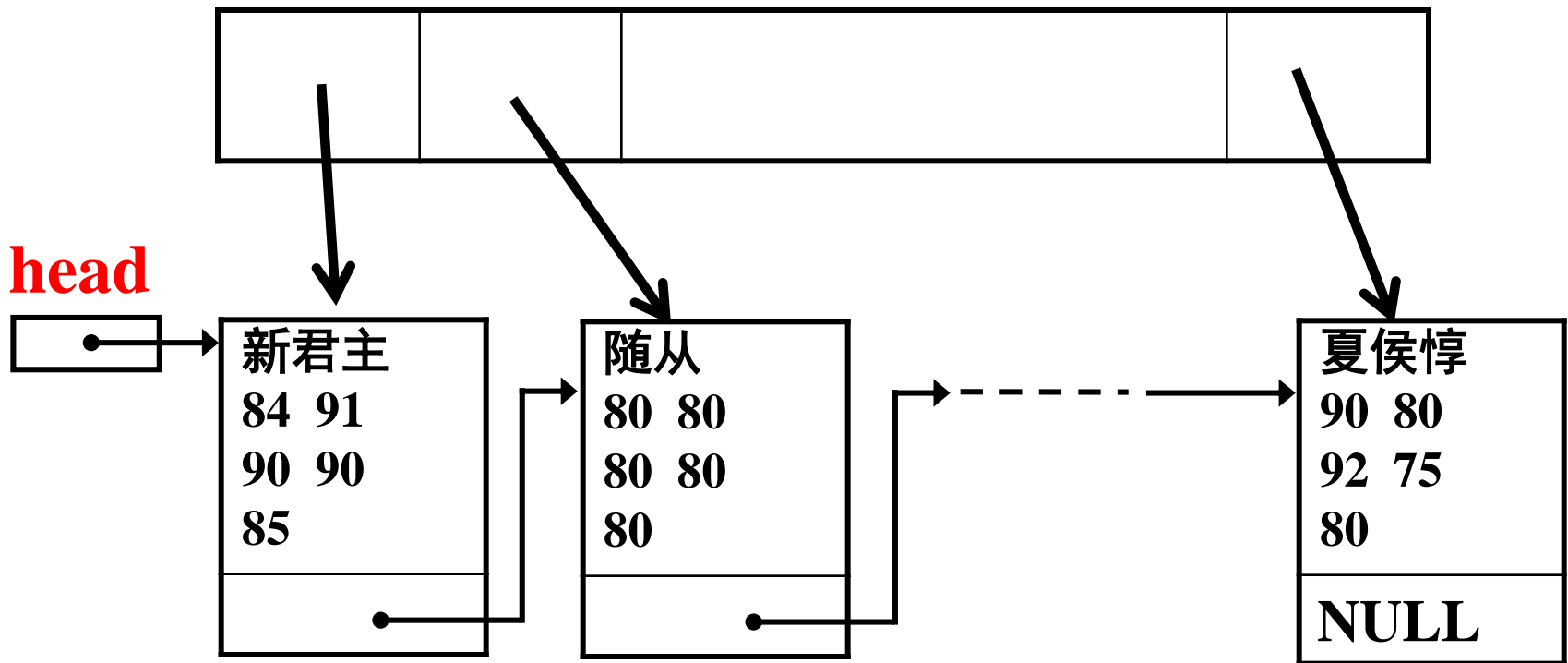
姓名	体力	智力	武力	魅力	运气
新君主	84	91	90	90	85
随从	80	80	80	80	80
关羽	95	88	95	85	80
鲁肃	70	90	85	80	85
夏侯惇	90	80	92	75	80
...					

head



a[0]	8	8	8	8	8	8	1
a[1]	9	9	9	9	9	1	8
a[2]	2	2	2	2	1	9	9
a[3]	4	4	4	1	2	2	2
a[4]	3	3	1	4	4	4	4
a[5]	1	1	3	3	3	3	3

初始值 1,3互换 1,4互换 1,2互换 1,9互换 1,8互换 一遍扫描完成



1. 把链表中每个“结点”看成数组的一个元素;
2. 计算链表长度 `int list_length(node *head);`
3. 调用冒泡法对链表排序: 只交换结点中的数据域, 不交换结点中的指针(next)

数据域只有一个值的实现

```
node *list_sort(node *head)
```

```
{  
    node *p;    int i, j, n, temp;  
    n = list_length(head); // 计算链表长度  
    if(head == NULL || head->next == NULL) // 空结点 or 单结点  
        return head;  
    for(j = 1; j < n; j++) { // 冒泡排序，外层循环  
        p = head;  
        for(i = 0; i < n - j; i++) { // 内层循环  
            if(p->data > p->next->data) { // 把小的数往上冒  
                temp = p->data; // 交换数据域  
                p->data = p->next->data;  
                p->next->data = temp;  
            }  
            p = p->next;  
        }  
    }  
    return head;  
}
```

```
typedef struct student  
{  
    int data;  
    struct student *next;  
} node;
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
.....
struct General *CreateList( );
void print_list (struct General *head);
struct General *list_sort (struct General *head);
int list_length (struct General *head);
void list_data_copy (struct General *Dst, struct General *Src);
int main()
{
    struct General *head, *head2;
    head = CreateList();
    print_list (head);

    head2 = list_sort (head);
    printf('按武力排序(从大到小):\n');
    print_list(head2);
    return 0;
}
```

```

struct General *list_sort (struct General *head)
{
    int i, j, n;
    struct General *p, temp;
    n = list_length (head); // 计算链表长度
    if(head == NULL || head->next == NULL)
        return head;
    for(j = 1; j < n; j++) { // 冒泡排序，外层循环
        p = head;
        for(i = 0; i < n - j; i++) { // 内层循环
            if(p->Power > p->next->Power) { // 把小的数往上冒
                list_data_copy (&temp, p);
                list_data_copy (p, p->next);
                list_data_copy (p->next, &temp);
            }
            p = p->next;
        }
    }
    return head;
}

```


int list_length (struct General *head) // 链表长度

```
{  
    struct General *p = head;  
    if (p == NULL) return 0;  
    int n = 0;  
    while(p) {  
        n++;  
        p = p->next;  
    }  
    return n;  
}
```

新君主	84	91	90	90	85
随从	80	80	80	80	80
关羽	95	88	95	85	80
夏侯惇	90	80	92	75	80

// 复制链表数据域

void list_data_copy (struct General *Dst, struct General *Src)

```
{  
    strcpy(Dst->Name, Src->Name);  
    Dst->Body = Src->Body;  
    Dst->Intelligence = Src->Intelligence;  
    Dst->Power = Src->Power;  
    Dst->Charisma = Src->Charisma;  
    Dst->Luck = Src->Luck;  
}
```

按武力排序(从小到大):

随从	80	80	80	80	80
新君主	84	91	90	90	85
夏侯惇	90	80	92	75	80
关羽	95	88	95	85	80

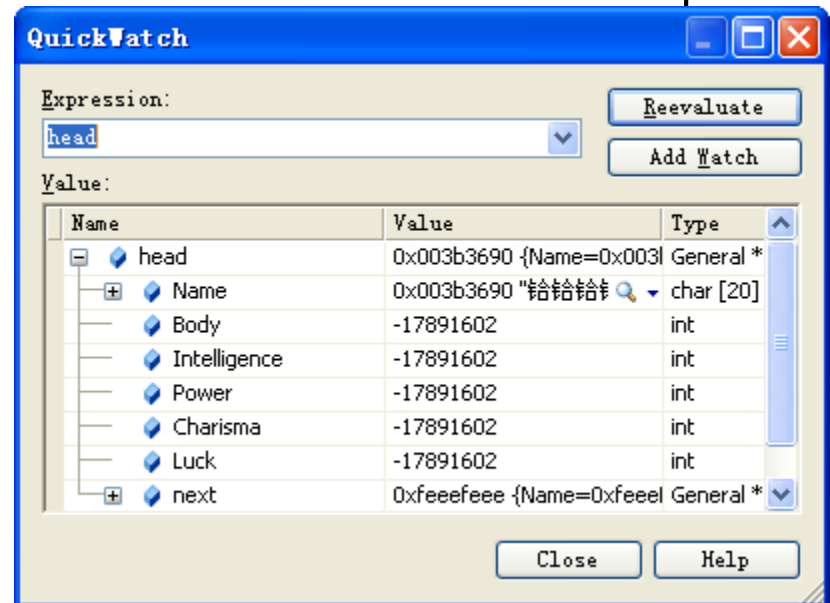
链表的释放

- 对于静态链表，它们所占用的内存空间是由系统自动来分配和释放的；
- 对于动态链表，必须由程序员自己来进行内存的分配与释放。

```

void Destroy(struct General *head)
{
    struct General *p, *q;
    p = head;
    while(p != NULL)
    {
        q = p;
        p = p->next;
        free(q);
    }
}

```



head = NULL?

链表的其他操作

➤ **插入**链表结点

➤ **删除**链表结点

➤
.....

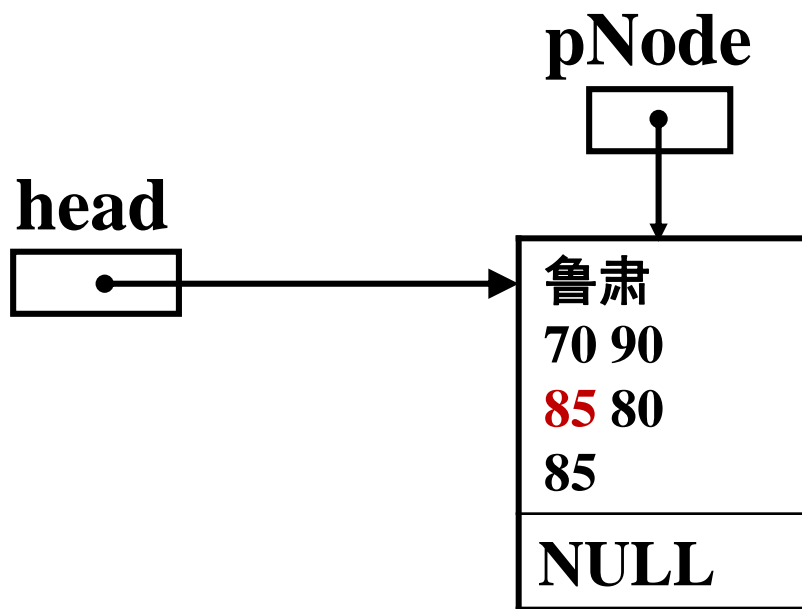
插入链表结点

原则：

- 插入操作不应破坏原有链接关系；
- 需要插入的这个结点应该把它放在合适的位置上，也就是说，应该有一个插入位置的查找过程；
- 假设已有链表按着升序(↑)排列，分成三成情况考虑。

第一种情况：链表为空，即 head = NULL

待插入的 pNode 结点就是链表中的第一个结点。



head = pNode;

第二种情况：

pNode 结点的 **Weight** 值小于等于链表首结点的 Weight 值，即 $\text{pNode} \rightarrow \text{Weight} \leq \text{head} \rightarrow \text{Weight}$
这时要将 pNode 结点插入到首结点的前面：

执行的语句：

```
pNode->next = head;  
head = pNode;
```

第三种情况：

即pNode结点的 Weight 要大于首结点的 Weight 值，这时肯定地说 pNode 结点要插入到首结点之后，但究竟插入到哪里需要先找到正确的位置。我们设指针q和指针p分别指向相邻的两个结点，q 在前 p 在后（即q更靠近首结点）。

首先让q = head，让 p = head->next，然后让它们顺序往后移动，每次移动一个结点。当满足：

$q \rightarrow \text{Weight} < \text{pNode} \rightarrow \text{Weight} \leq p \rightarrow \text{Weight}$
时，pNode 就插在 q 与 p 之间。

移动指针:

q = p;

p = p->next;

插入结点:

pNode->next = p;

q->next = pNode;



这两个语句是否可以互换?

如果新结点的Weight大于所有结点？

```
while(p != NULL)
{
    if(pNode->Weight <= p->Weight)
        break;
    else
    {
        q = p;
        p = p->next;
    }
}
```

在这种情形下，当循环语句结束后，指针q是指向链表的尾结点，而指针 $p = \text{NULL}$ 。

```

struct General *Insert (struct General *head, struct General *pNode)
{
    struct General *p, *q;
    // 第一种情形，链表为空
    if(head == NULL)
    {
        head = pNode;
        return head;
    }
    // 第二种情形，新结点的Weight小于等于首结点
    if(pNode->Power <= head->Power)
    {
        pNode->next = head;
        head = pNode;
        return head;           // 或直接 return pNode;
    }
}

```

注意：在第一、二种情形，返回的head不同于原始链表head

// 第三种情形，循环地查找正确的插入位置

```
q = head;
p = head->next;
while(p != NULL)
{
    if(pNode->Power <= p->Power)
        break;
    else
    {
        q = p;
        p = p->next;
    }
}
```

// 将pNode结点插入在正确的位置(q和p之间)

```
pNode->next = p;
q->next = pNode;
return head;
```

```
}
```

三种情况概括为：

1. 链表为空；
2. 首结点之前；
3. 首结点之后；

单结点链表插入情况呢？

注意：在第三种情形，返回的head与原始链表head一致

..... // 与链表排序头文件一致

```
struct General *Insert (struct General *head, struct General *pNode);
```

```
int main()
```

```
{
```

```
    struct General *head, *head2, *head3;
```

```
    head = CreateList();
```

```
    print_list(head);
```

```
    head2 = list_sort(head);
```

```
    printf("按武力排序(从小到大):\n");
```

```
    print_list(head2);
```

```
    struct General *pNode = CreateList(); // 待插入新结点“鲁肃”
```

```
    print_list(pNode);
```

```
    head3 = Insert(head2, pNode); // 插入新结点, 返回新head3
```

```
    printf("插入结点后的链表:\n");
```

```
    print_list(head3);
```

```
    return 0;
```

```
}
```

插入一个新结点测试: (“鲁肃”, power = 85)

```
新君主 84 91 90 90 85
随从   80 80 80 80 80
关羽   95 88 95 85 80
夏侯惇 90 80 92 75 80
#
```

```
鲁肃   70 90 85 80 85
#
```

按武力排序(从小到大):

```
随从   80 80 80 80 80
新君主 84 91 90 90 85
夏侯惇 90 80 92 75 80
关羽   95 88 95 85 80
```

插入结点后的链表:

```
随从   80 80 80 80 80
鲁肃   70 90 85 80 85
新君主 84 91 90 90 85
夏侯惇 90 80 92 75 80
关羽   95 88 95 85 80
```

重新插入一个新结点测试: (“鲁肃”, power = 98)

鲁肃 70 90 98 80 85
#

插入结点后的链表:
随从 80 80 80 80 80
新君主 84 91 90 90 85
夏侯惇 90 80 92 75 80
关羽 95 88 95 85 80
鲁肃 70 90 98 80 85

删除链表结点

假设新君主要从队伍当中删除一名人员。

这里就需要用到链表结点的删除技术。

情形一、待删除的是首结点

free(head); ?

```
p = head;  
head = p->next;  
free(p);
```

情形二、待删除的不是首结点

```
q->next = p->next;  
free(p);
```

如何找到待删除的结点？

.....

```
p = head;
```

```
while((p != NULL) && strcmp(p->Name,“随从”))
```

```
{
```

```
    q = p;
```

```
    p = p->next;
```

```
// 把指针p往后移动一个结点
```

```
}
```

```

struct General *Delete (struct General *head, char *name)
{
    struct General *p, *q;
    if(head == NULL) { printf("空链表"); return NULL; }
    p = head;
    while((p != NULL) && strcmp(p->Name, name)) {
        q = p;
        p = p->next; // 把指针p往后移动一个结点
    }
    if(p != NULL) {
        if(p == head)
            head = p->next; // 删除的是首结点
        else
            q->next = p->next; // 删除的是中间结点
        free(p);
    }
    return(head);
}

```

若删除的是尾结点呢？

..... // 与创建链表头文件一致

```
struct General *Delete (struct General *head, char *name);
```

```
int main()
```

```
{  
    struct General *head, *head2;  
    head = CreateList();  
    print_list(head);
```

```
新君主 84 91 90 90 85  
随从    80 80 80 80 80  
关羽    95 88 95 85 80  
夏侯惇 90 80 92 75 80  
#
```

```
    char * name = "随从";  
    head2 = Delete (head, name);  
    printf("删除\"%s\"结点后链表:\n", name);  
    print_list(head2);
```

```
删除“随从”结点后链表:  
新君主 84 91 90 90 85  
关羽    95 88 95 85 80  
夏侯惇 90 80 92 75 80
```

```
    name = "夏侯惇";  
    head2 = Delete (head2, name);  
    printf("删除\"%s\"结点后链表:\n", name);  
    print_list(head2);  
    return 0;
```

```
删除"夏侯惇"结点后链表:  
新君主 84 91 90 90 85  
关羽    95 88 95 85 80
```

```
}
```

注意: 删除结点后, 返回新head2

链表 vs. 数组

- 一批类型相同的数据用数组存储的问题：
 - 静态数组：必须指定数组的元素个数，此后无法更改数组大小，可能造成空间不足或浪费
 - 动态数组：空间不会浪费或不足，但需要连续存储空间
 - 数组中插入/删除元素需大量移动元素，效率低
- 链表结构：
 - 数组元素逻辑上相邻物理地址上也相邻
 - 链表结构其数据元素作为一个个结点的数据域，结点中另有指针域存储逻辑上相邻元素地址

链表总结

- 链表结构的优点:

- 系统不必为应用程序分配一组连续的空间，可以充分利用系统的零散空间；有一个元素就生成一个节点，**空间不浪费**
- 如果内存足够大，理论上数据容量不受限制
- **插入/删除**等操作不必通过移动元素实现，**效率高**

- 单向链表基本操作:

- 基础：建立、遍历、查找、释放
- **难点：插入、删除、反向、排序**
- 链表 = 动态内存分配 + 结构 + 指针

链表总结2

- 链表是程序设计中一种重要**动态数据结构**，它是动态地进行存储分配的一种结构。
- **动态性**体现为：
 - 链表中的元素个数可以根据需要增加和减少，不像数组，在声明之后就固定不变；
 - 元素的位置可以变化，即可以从某个位置删除，然后再插入到一个新的地方；
 - 链表中的元素称为“结点”，每个结点包括两个域：数据域和指针域。单向链表通常由一个头指针(head)，用于指向链表头，有一个尾结点，该结点的指针部分指向一个空结点(NULL)

环形链表

环形链表：一种特殊的链表，其尾结点的 **next** 指针，又指向了链表的首结点，从而形成了一个圆环。

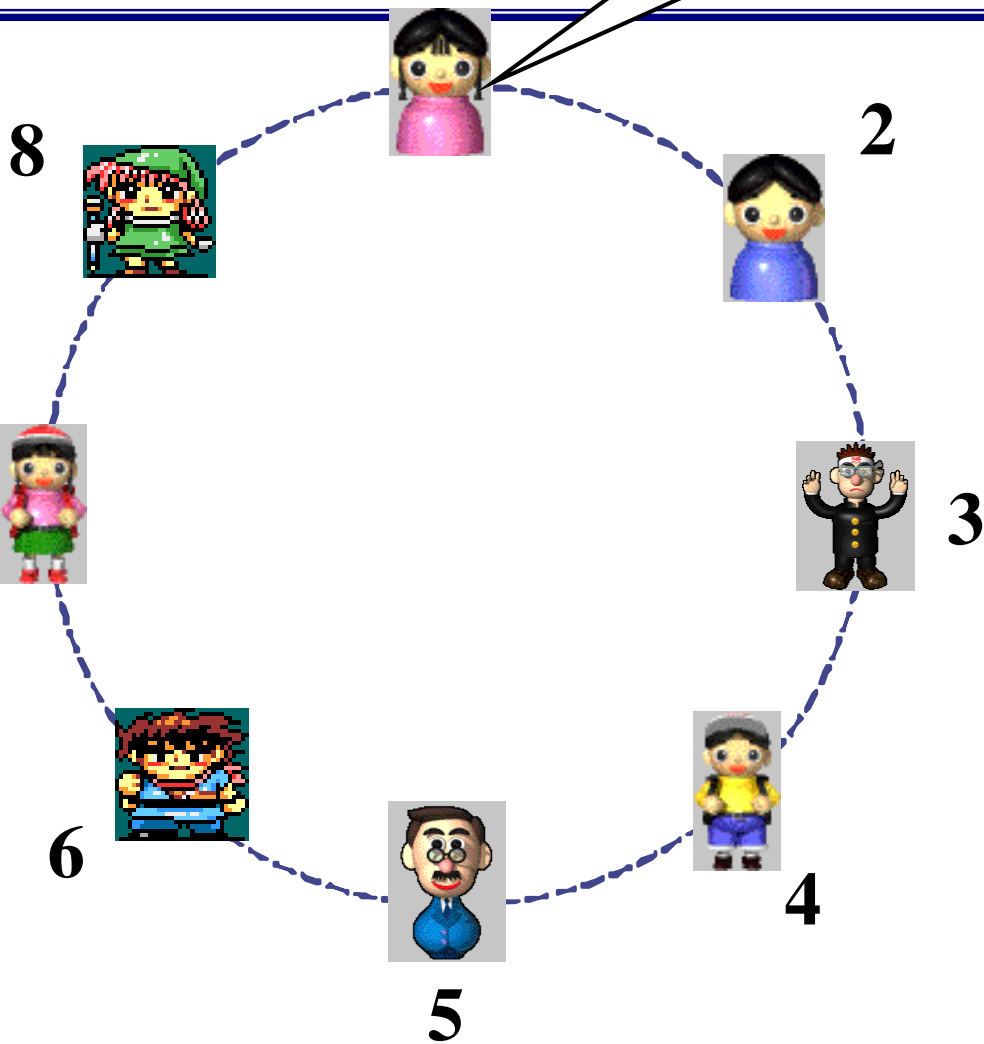
从环形链表的任何一个结点出发，都可以遍历整个的链表。

问题描述：

学校给高一（三）班分配了一个名额，去参加奥运会的开幕式。每个人都争着要去，可是名额只有一个，怎么办？班长想出了一个办法，让班上的所有同学围成一圈，按照顺时针方向进行编号。然后随便选定一个数 m ，并且从1号同学开始按照顺时针方向依次报数， $1, 2 \dots m$ ，凡报到 m 的同学，都要主动退出圈子。然后不停地按顺时针方向逐一让报出 m 者出圈，最后剩下的那个人就是去参加开幕式的人。

演示: $n=8, m=3$

起始位置



讨论!

退出圈子的顺序: 3 6 1 5 2 8 4

数据结构

// 定义一个名为STUDENT的结构体类型

struct STUDENT

{

int number; // 表示同学的编号

struct STUDENT *next; // 指向下一位同学

};

struct STUDENT *head, *tail, *p, *prev;

int n, m, i, j;

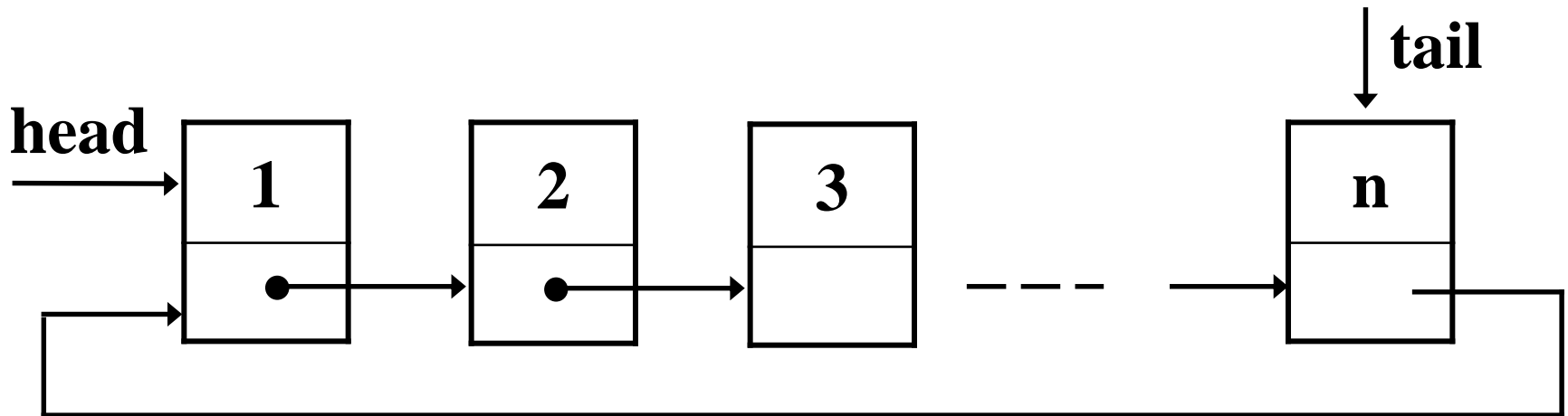
基本思路

- 模块1: 输入学生的个数 n 和不吉利的数字 m , 然后验证它们的有效性;
- 模块2: 创建一个环形链表, 模拟众同学围成一圈的情形;
- 模块3: 进入循环淘汰环节, 模拟从1到 m 报数, 让 $n-1$ 个同学逐一退出圈子的过程;
- 模块4: 输出结果。

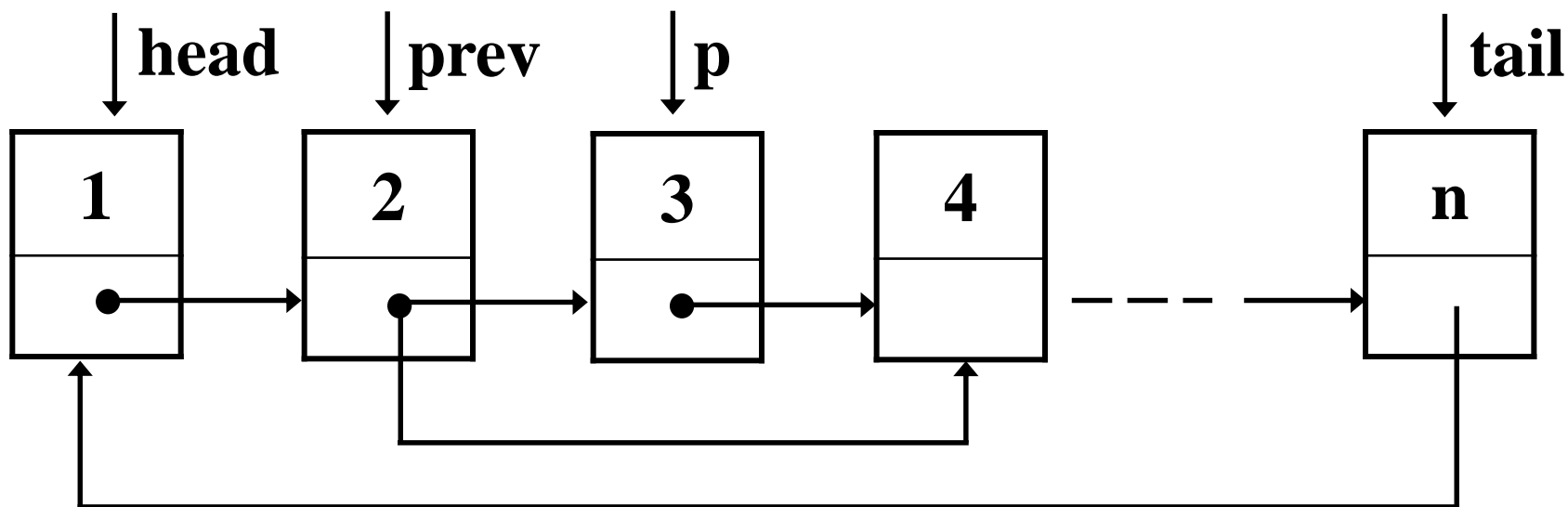
模块2，创建环形链表

- (1) 动态地为每一个结点分配内存空间，并顺序地进行编号；
- (2) 把各个结点按照编号顺序链接成一条环形链表；
- (3) 用**head**指向链表的第一个结点，用 **tail** 指向链表的最后一个结点。

模块2，创建环形链表



模块3，循环淘汰环节



假设 $m = 3$

参考程序：

略.....

附：参考程序框架

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
// 定义一个名为STUDENT的结构体类型
```

```
struct STUDENT
```

```
{
```

```
    int number;
```

```
// 表示同学的编号
```

```
    struct STUDENT *next;
```

```
// 指向下一位同学
```

```
};
```

```
struct STUDENT *CreateList (int n, struct STUDENT *&tail);
```

```
struct STUDENT *Select (struct STUDENT *head,  
                        struct STUDENT *tail, int m);
```

```
int main()
{
    struct STUDENT *head, *tail, *p;

    int n, m;
    printf("请输入总人数: ");
    scanf("%d", &n);
    printf("请输入间隔数: ");
    scanf("%d", &m);

    head = CreateList (n, tail);

    p = Select (head, tail, m);
    printf("参加的人是: %d\n", p->number);

    return 0;
}
```

请输入总人数: 8
请输入间隔数: 3

退出的人是: 3
退出的人是: 6
退出的人是: 1
退出的人是: 5
退出的人是: 2
退出的人是: 8
退出的人是: 4

参加的人是: 7

// 创建一个环形链表

```
struct STUDENT *CreateList (int n, struct STUDENT *&tail)
{
    .....
}
```

// 删除退出人的节点，更新环形链表

```
struct STUDENT *Select (struct STUDENT *head, struct
                        STUDENT *tail, int m)
{
    .....
}
```

补充: C++ 的“引用”

- C语言没有“引用”，只有C++才有“引用”
 - 引用就是某个目标变量的“别名” (alias)，对引用的操作与对变量直接操作效果完全相同
- 将“引用”作为函数参数
 - 传递引用给函数与传递指针的效果是一样的
 - 使用引用传递函数的参数，在内存中并没有产生实参的副本，它是直接对实参操作

1. 值传递

```
#include <stdio.h>
void swap(int x, int y)
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}
int main()
{
    int a = 4, b = 6;
    swap(a, b);
    printf("%d %d", a, b);
}
```

4 6

2. 地址传递

```
#include <stdio.h>
void swap(int *x, int *y)
{
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
int main()
{
    int a = 4, b = 6;
    swap(&a, &b);
    printf("%d %d", a, b);
}
```

6 4

3. 引用传递

```
#include <stdio.h>
void swap(int &x, int &y)
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}
int main()
{
    int a = 4, b = 6;
    swap(a, b);
    printf("%d %d", a, b);
}
```

6 4

地址传递 vs. 引用传递

- 地址传递

- 使用指针作为函数的参数虽然也能达到与使用引用的效果，但是，在被调函数中同样要给形参分配存储单元
- 且需要重复使用“*指针变量名”的形式进行运算，这容易产生错误且程序的阅读性较差
- 另一方面，在主调函数的调用点处，必须用变量的地址作为实参

- 而“引用”更容易使用，更清晰。

```
struct STUDENT *CreateList (int n, struct STUDENT *&tail);
```

思考题（1）

编写一个基于动态链表的“**长整数加法运算器**”，来实现任意长度的两个整数的**加法**运算。

具体要求：

（1）必须用**线性链表**的形式来存储一个长整数，例如：对于整数135，可以创建一条线性链表，该链表包含三个结点，分别用来存储1、3、5这三个数字。考虑到输入整数的长度是任意的（不超过100位），因此，为了减少内存空间的浪费，在程序中必须采用动态链表的方法，即每一个链表结点都是根据需要动态创建的；

（2）只考虑两个正整数的加法，无须考虑负整数的情形；

(3) 为了增强程序的可读性，应采用多函数的形式来实现，至少应包含如下的函数：创建链表(CreatList)、加法函数(AddList)、打印链表(DisplayList)等；

(4) 提示：由于本题处理的整数的长度是任意的，可能会超出long的取值范围，所以应该通过字符串的方式来处理输入输出。

本题的程序实现不能调用C++标准库函数<list>来实现，否则将被扣分。

以下是一次模拟运行的结果：

样例输入：

1234567890

135

样例输出：

1234568025

Lecture 8 - Summary

- **Topics covered:**
 - **Defining and using Structures**
 - **Functions and Structures**
 - **Initializing Structures. Compound Literals**
 - **Arrays of Structures**
 - **Structures Containing Structures and/or Arrays**
 - **Enumerated Data Types**
 - **The typedef Statement**
 - **List: create, delete, insert, update, find, sort**