C语言程序设计

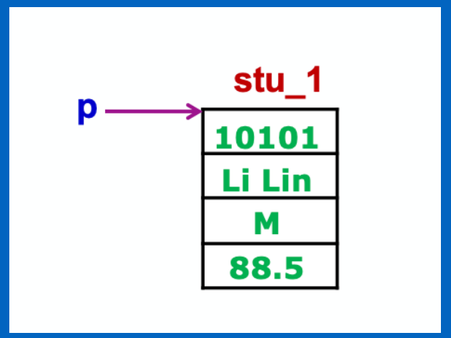
计算机科学与技术学院

结构体指针

- 指向结构体的指针
- 动态内存分配
- 单链表

指向结构体的指针

- 指针变量可以指向结构体变量
- 指向结构体变量的指针变量，其基类型必须与结构体变量的类型相同

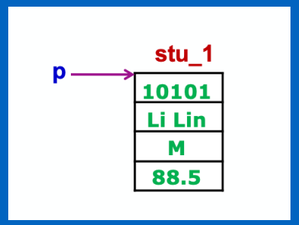



```c
#include <stdio.h>
#include <string.h>
int main(){
struct Student  {
    int   num;
    char  name[20];
    char  sex;
    float score;
};

struct Student stu_1;

//struct Student *p;

//p = & stu_1;

stu_1.num = 10101;
strcpy(stu_1.name,"Li Lin");
stu_1.sex='M';
stu_1.score=88.5;

printf("No.:%ld\n",stu_1.num);
printf("name:%s\n",stu_1.name);
printf("sex:%c\n",stu_1.sex);
printf("score:%f\n",stu_1.score  ) ;

return 0;
}
```

```c
#include <stdio.h>
#include <string.h>
int main(){
struct Student  {
    int   num;
    char  name[20];
    char  sex;
    float score;
};

struct Student stu_1;

struct Student *p;

p = & stu_1;

stu_1.num = 10101;
strcpy(stu_1.name,"Li Lin");
stu_1.sex='M';
stu_1.score=88.5;

printf("No.:%ld\n",(*p).num);
printf("name:%s\n",(*p).name);
printf("sex:%c\n",(*p).sex);
printf("score:%f\n",(*p).score);

return 0;
}
```

结构体指针: -> 运算符

- 为了使用方便和直观，常用p->num来代替(*p).num
  - (*p).name等价于p->name
- 如果p指向一个结构体变量stu_1，以下等价：
  - stu*1*.成员名,如 *stu*1.num
  - (*p).成员名,如 (*p).num
  - p->成员名,如 p->num




```c
#include <stdio.h>
#include <string.h>
int main(){
struct Student  {
    int   num;
    char  name[20];
    char  sex;
    float score;
};

struct Student stu_1;

struct Student *p;

p = & stu_1;

stu_1.num = 10101;
strcpy(stu_1.name,"Li Lin");
stu_1.sex='M';
stu_1.score=88.5;

printf("No.:%ld\n", p->num);
printf("name:%s\n", p->name);
printf("sex:%c\n", p->sex);
printf("score:%5.1f\n", p->score);

return 0;
}
```

结构体指针与结构体数组

- 例：有3个学生的信息，放在结构体数组中，要求输出全部学生的信息
- 解题思路：用指向结构体变量的指针处理
  - 声明struct Student，并定义结构体数组、初始化
  - 定义指向struct Student类型指针p
  - 使p指向数组首元素，输出元素中各信息
  - 使p指向下一个元素，输出元素中各信息
  - 重复上一步，直到输出数组全部元素

有3个学生的信息，放在结构体数组中，要求输出全部学生的信息

```c
#include <stdio.h>
struct Student {
    int num;
    char name[20];
    char sex;
    int age;
};
struct Student stu[3]={
    {10101,"Li Lin",'M',18},
    {10102,"Zhang Fun",'M',19},
    {10104,"Wang Min",'F',20}
};
int main(){
    struct Student *p;
    printf(" No.  Name     sex  age\n");
    for(p=stu;p<stu+3;p++)
        printf("%5d %-20s %2c %4d\n",
            p->num, p->name,p->sex, p->age);
    return 0;
}
```

动态内存分配

- 无论简单变量或数组，在引用前都要被定义
- 预先分配内存

```
#define N  10
int  n, a[10],b[N];
```

- 下面写法是错误的：

```
int n;   scanf("%d",&n);
int a[n];
```

- 但是在实际的编程中，往往所需的内存空间无法预先确定
- 解决方法：
  - 采用动态生成的数组
  - 采用动态的数据结构
- 这2种方式都需要采用C语言的动态内存分配

动态内存分配

- 动态内存分配:在程序的运行过程中，根据需要随时向系统申请内存空间，用完之后，系统收回内存单元
- 申请内存空间函数malloc()
  - `void *malloc (unsigned int size) ;`
  - 功能：在内存的动态存储区中，开辟一个长度为size字节的连续空间，函数的返回值是一个指向该区域的指针（地址）
  - 例:动态申请数组a[n],n的大小从键盘输入

```
int n, *a;
scanf("%d",&n);
a=(int*)malloc(n*sizeof(int));
```

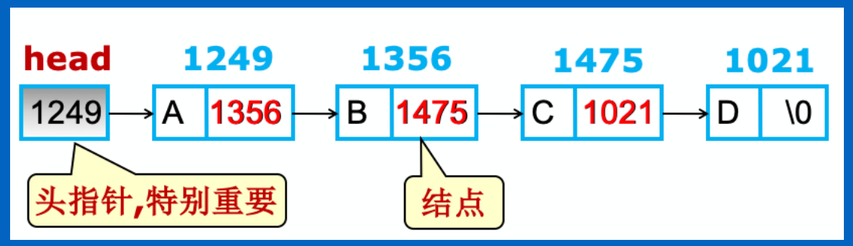
- 释放内存空间free()
  - `void  free (void *p) ;`
  - 功能：释放由指针p所指内存单元的空间
  - 例如：`free(a);`

- 有若干个学生的信息（学生人数由键盘输入），放在结构体数组中，要求输出全部学生的信息
  - 学生成绩保存在结构体数组中
  - 由于成绩记录不确定，所以不能用静态数组
  - 可以采用动态生成的数组


```c
#include<stdio.h>
#include<stdlib.h>
struct student
    int   num;
    char  name[20];
    char  course_name[20];
    float  credit;
    int grade;
    float GP;
};

int main(){
    int n,i;
    struct student *s;
    printf("请输入人数: ");
    scanf("%d",&n);
    s=(struct student*)
        malloc(n*sizeof(struct student));
    for(i=0;i<n;i++)    {
        printf("请输入第%d位成绩: \n",i+1);
            scanf("%d%s%s%f%d%f",
            &s[i].num,
            s[i].name,
            s[i].course_name,
            &s[i].credit,
            &s[i].grade,
            &s[i].GP);
    }

    printf("输出成绩: \n");
    for(i=0;i<n;i++)
        printf("%6d%10s%10s%4.1f%5d%5.1f\n",
        s[i].num,s[i].name, s[i].course_name,
        s[i].credit,s[i].grade,s[i].GP);
    free(s);    //释放空间
    return 0;
}
```

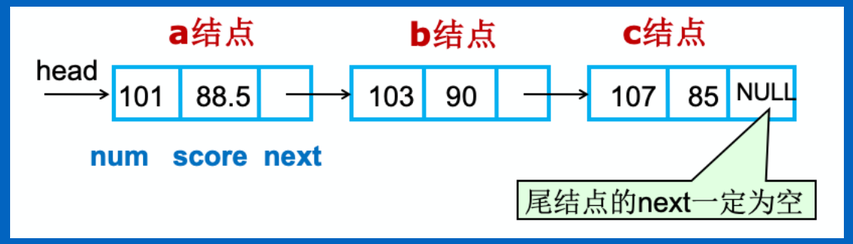
链表

- 数组必须是用一片连续的存储单元来存储一组数据
- 链表是用一组任意的、不连续的存储单元来存储一组数据，且链表的长度不是固定的
- 链表是一种典型的动态数据结构



- 链表的每一个元素称为一个“结点”
- 各结点的地址不连续,可存放在内存中的不同位置
- 要存储每个元素与后继元素的逻辑关系，以便构成“一个结点链着一个结点”的链式存储结构
  - 存储元素本身的信息外
  - 存储后继元素的地址
- 每个结点都应包括两个部分：
  - 用户需要用的数据
  - 下一结点的地址
- 单向链表：只有通过前一个结点才能找到下一个结点

单链表



- 一个单链表：

```
struct Student{
    int num;
    float score;
    struct Student *next;
}a,b,c,*head;

a. num=101;
a.score=88.5;
b. num=103;
b.score=90;
c. num=107;
c.score=85;

head=&a;
a.next=&b;
b.next=&c;
c.next=NULL;
```

单链表：建立动态链表

- 建立动态链表：在程序执行过程中从无到有地建立起一个链表，即一个一个地开辟结点，并输入各结点数据，并建立起前后相链的关系

- 定义3个指针变量：head,p1和rear
  - 用来指向struct Student类型数据
  - head为单链表的头指针
  - p1指向当前结点
  - rear指向当前链表的尾结点

```
struct Student{
    long num;
    float score;
    struct Student *next;
};
struct Student
        *head,*p1,*rear;
```

- 初始状态下，链表为空：

```
head=NULL;
rear=NULL;
```

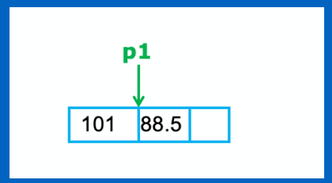
- 用malloc函数开辟一个结点，并使p1指向它


```
p1=(struct Student*)malloc(LEN);
```

单链表：建立动态链表

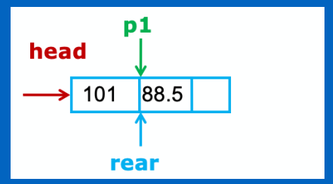
- 建立动态链表：在程序执行过程中从无到有地建立起一个链表，即一个一个地开辟结点，并输入各结点数据，并建立起前后相链的关系

- 读入一个学生的数据给p1所指的结点




```
scanf("%ld,%f",&p1->num,&p1->score);
```

- 如果是第一个结点(head==NULL)
  - 当前结点既是单链表的头结点
  - 也是尾结点



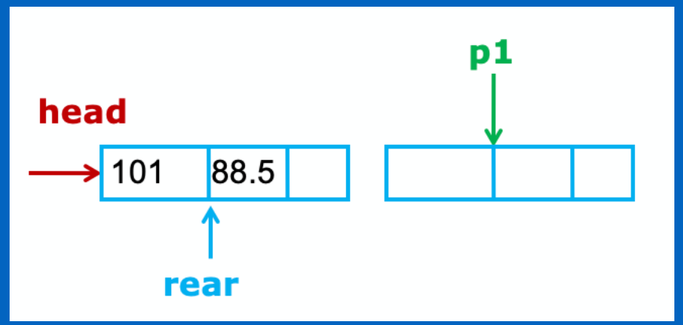

```
head=p1;
rear=p1;
```

单链表：建立动态链表

- 建立动态链表：在程序执行过程中从无到有地建立起一个链表，即一个一个地开辟结点，并输入各结点数据，并建立起前后相链的关系

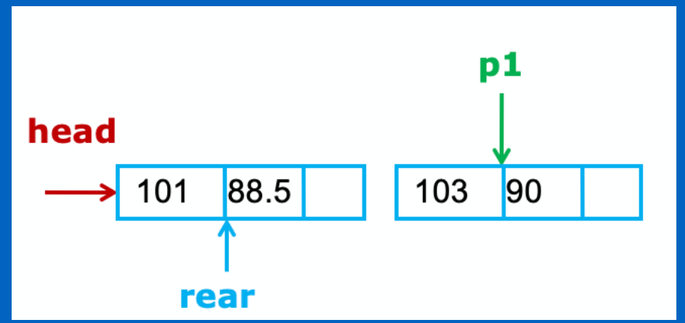
- 开辟一个新结点并使p1指向它

```
p1=(struct Student*)malloc(LEN);
```



- 输入该结点的数据

```
scanf("%ld,%f",&p1->num,&p1->score);
```

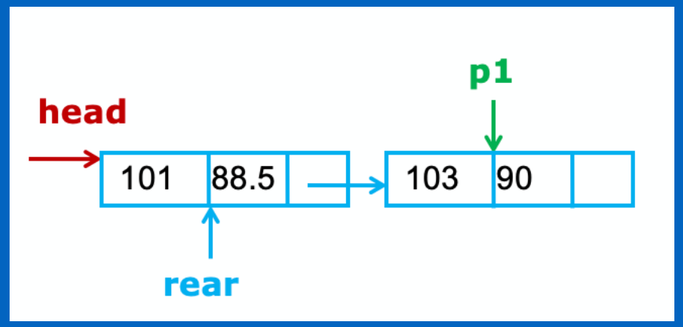


单链表：建立动态链表

- 建立动态链表：在程序执行过程中从无到有地建立起一个链表，即一个一个地开辟结点，并输入各结点数据，并建立起前后相链的关系

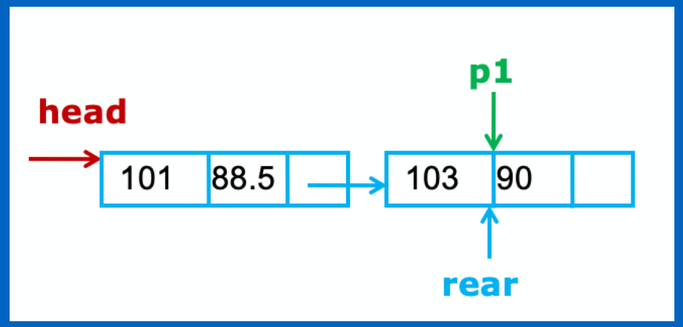
- 链接尾结点(rear)和当前结点(p1)

```
rear->next = p1;
```



- 使rear指向新的尾结点

```
rear = p1;
```



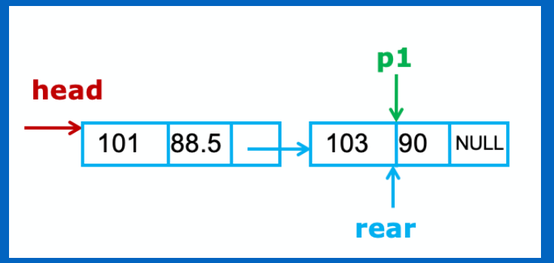
单链表：建立动态链表

- 建立动态链表：在程序执行过程中从无到有地建立起一个链表，即一个一个地开辟结点，并输入各结点数据，并建立起前后相链的关系

- 循环重复刚才的过程
- 如果某一个结点的学号输入为999，循环结束

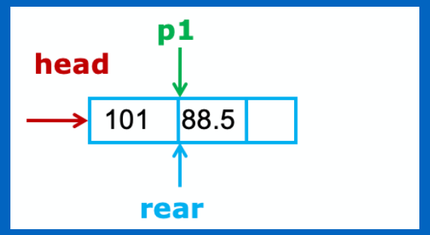
- 最后一个结点的next域置为NULL：

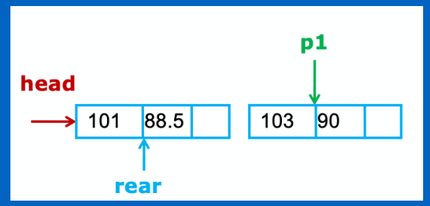
```
rear->next = NULL
```

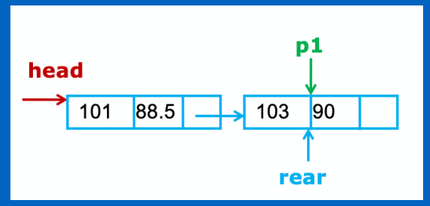


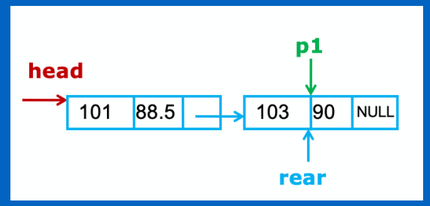
```c
#include <stdio.h>
#include <stdlib.h>
#define LEN sizeof(struct Student)
struct Student{
    long num;
    float score;
    struct Student *next;
};

struct Student *create( ) {
    struct Student *head=NULL,*p1,*rear=NULL;
    p1=(struct Student*) malloc(LEN);
    scanf("%ld,%f",&p1->num,&p1->score);
    while(p1->num != 999){
    if(head == NULL)
        head=p1;
    else
        rear->next=p1;
        rear=p1;
        p1=(struct Student*)malloc(LEN);
        scanf("%ld,%f",&p1->num,&p1->score);
    }
    rear->next=NULL;//最后一个结点的next置为空
    return(head); //返回单链表的头指针
}
```

- 主要过程







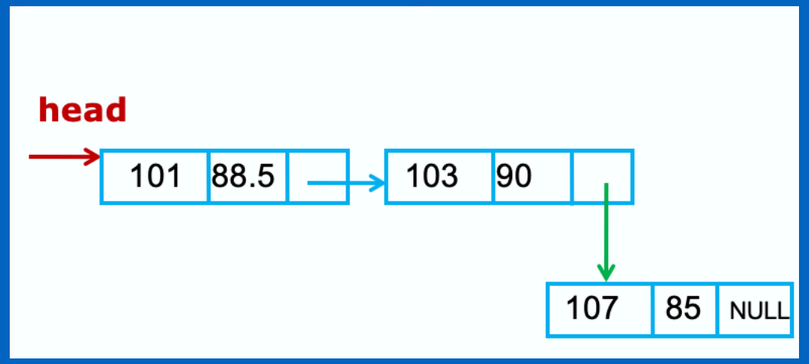


单链表：遍历链表并输出

- 编写一个输出单链表元素的函数print()，它能够从头到尾遍历链表，并输出链表各个元素的值

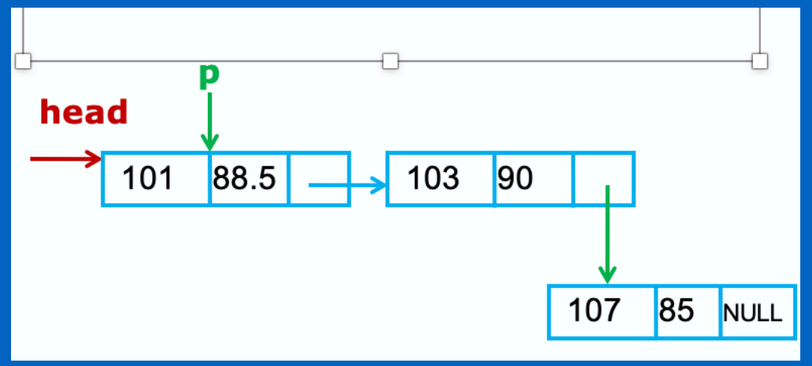
- P指向单链表的第一个结点

```
p = head
```



- 输出p所指的结点

```
printf("%ld %5.1f\n",p->num,p->score);
```

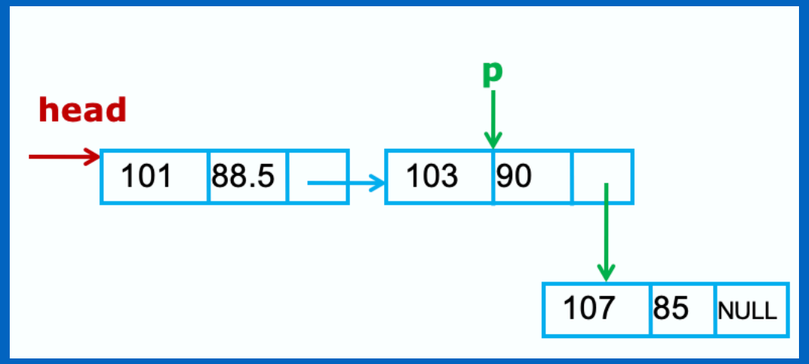


单链表：遍历链表并输出

- 编写一个输出单链表元素的函数print()，它能够从头到尾遍历链表，并输出链表各个元素的值

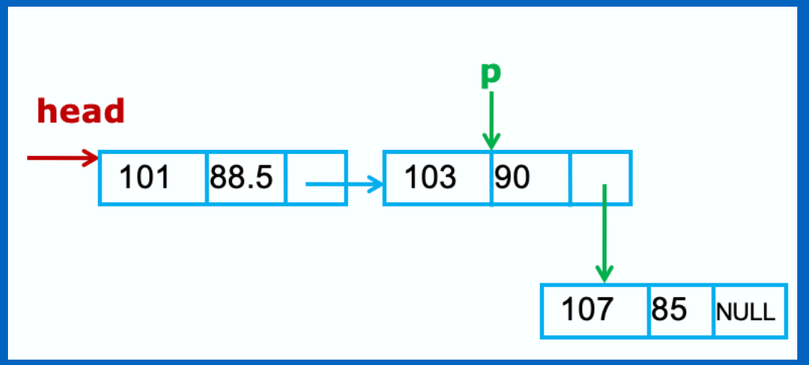
- P移动到后一个结点

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



- 输出p所指的结点

```
printf("%ld %5.1f\n",p->num,p->score);
```

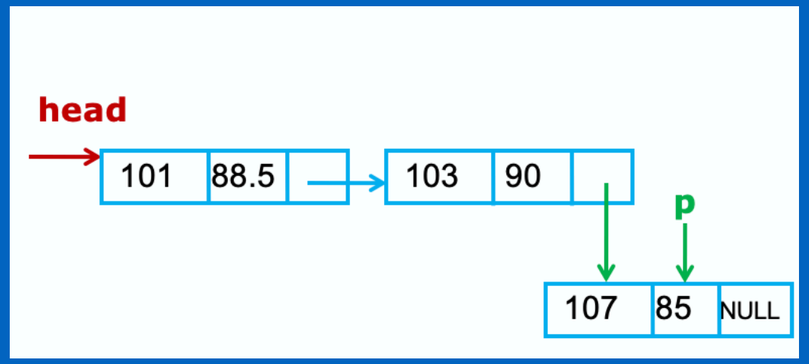


单链表：遍历链表并输出

- 编写一个输出单链表元素的函数print()，它能够从头到尾遍历链表，并输出链表各个元素的值

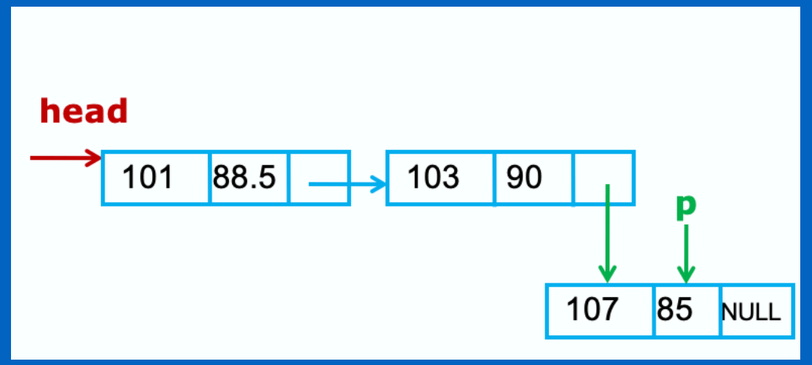
- P再后移一个结点

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



- 输出p所指的结点

```
printf("%ld %5.1f\n",p->num,p->score);
```



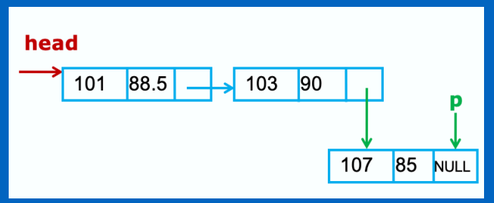
单链表：遍历链表并输出

- 编写一个输出单链表元素的函数print()，它能够从头到尾遍历链表，并输出链表各个元素的值

- P再后移一个结点

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

- 此时 p = NULL,循环结束。

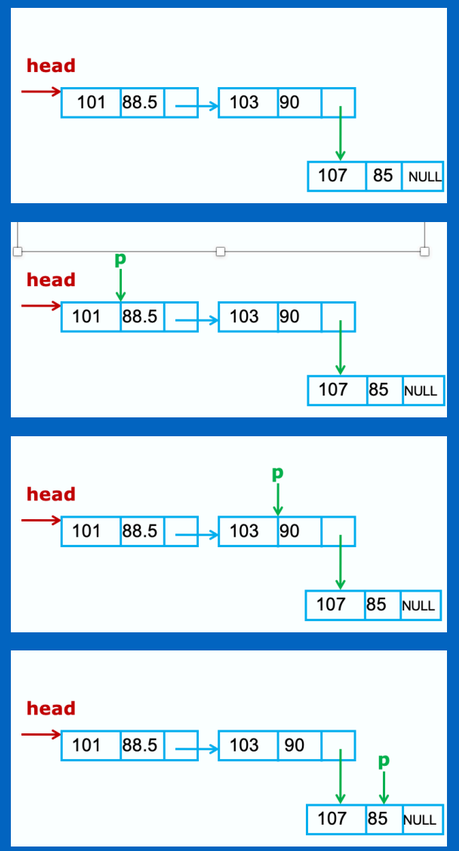


- 遍历步骤小结：
  - 指针p指向单链表的头结点
    p=head;
  - 输出p指向的结点的内容
  - 指针p后移一个结点
  - 重复以上两步，直到指针p为空

单链表：创建和遍历链表

```
void print(struct Student *head){
    struct Student  *p=head;
    printf("\nThese records are:\n");
    while(p!=NULL){
        printf("%ld %5.1f\n",
            p->num,p->score);
        p=p->next;
    }
}

// 函数creat()和print()可以用如下main()函数调用

int main(){
    struct Student *head;
    head = create();
    print(head);
    return 0;
}
```

- 遍历主要过程



后续学习

- C语言：不适于编写较大规模的程序。
- C++是由AT&T Bell(贝尔)实验室的Bjarne Stroustrup博士及其同事于20世纪80年代初在C语言的基础上开发成功的。
- C++保留了C语言原有的所有优点，增加了面向对象的机制。
- C++是由C发展而来的，与C兼容，是C的超集。用C语言写的程序基本上可以不加修改地用于C++。
- C++既可用于面向过程的结构化程序设计，又可用于面向对象的程序设计，是一种功能强大的混合型的程序设计语言。

后续学习

- C#是一种面向对象的编程语言，它将作为Visual Studio中的一部分推出。
- C#既保持了C++中熟悉的语法，并且还包含了大量的高效代码和面向对象特性
- C#语言在保持C/C++灵活性的基础上带来更高效的RAD开发方式。
- 不仅能用于系统级程序的开发，并且更适于WEB应用程序的开发。

C#

```
using System;

class Hello{

    static void Main(){

        Console.WriteLine("Hello, world");

  }

}
```