第17章

指针的高级应用

内容

- 动态存储分配与释放
- 链表

动态存储分配

- C语言数据结构，如数组，通常大小固定
  - 一旦程序运行，大小不可改变
  - 要改变，必须修改程序再次运行，不方便
- C支持动态存储分配：
  - 在程序执行期间分配内存单元的能力
  - 按需扩大或缩小内存
- 适用于所有类型的数据，但主要用于字符串、数组和结构
- 便于生成表、树或其他数据结构

内存分配函数

- 动态存储分配主要通过调用内存分配函数来实现
- stdlib.h头文件中声明了三个内存分配函数：
  - malloc()：分配内存块，不对初始化内存块。
  - calloc()：分配内存块，对内存块进行清零。
  - realloc()：调整先前分配的内存块大小。
- 都返回void *类型（“通用型”指针）

空指针

- 内存分配时，如果找不到满足需要的足够大内存块，返回空指针(null pointer)
  - 空指针：“不指向任何地方的指针”
  - 地址全0：区别于有效指针的特殊值
  - NULL：一个宏，在多个头文件里都有定义，stdio.h、stdlib.h、string.h等
- 内存分配函数调用后，需判断返回值（赋值给某指针变量）是否为空指针
- 可将malloc调用和返回值判断是否为空结合起来：

```
if ((p = malloc(10000)) == NULL) {
    /* allocation failed; take appropriate action */
}
```

空指针

- 指针测试真假的方法和数的测试一样。所有非空指针都为真，而只有空指针为假
- 常见写法：

```
if (p == NULL) …
```

或者

```
if (!p) …
if (p != NULL) …
```

或者

```
if (p) …
```

动态分配字符串

- 动态内存分配对字符串操作非常有用
- 之前用字符数组存储字符串
  - char reminders[MAX_REMIND][MSG_LEN+3];
  - #define MAX_REMIND ？？
- 通过动态分配，可按需添加日程项
  - malloc(MSG_LEN+3);

使用malloc函数为字符串分配内存

- malloc函数原型：

```
void *malloc(size_t size);
```

- 分配size个字节的内存块，并返回指向该内存块首地址的指针。
  - size_t：无符号整数

使用malloc函数为字符串分配内存

- 给n个字符的字符串分配内存空间

```
char * p;
p = malloc(n + 1);
```

- 明确分配空间用处:

```
p = (char *) malloc(n + 1);
```

  - 非必须

- ps: malloc不清零或初始化所分配空间
- 分配后可调strcpy函数对字符串设置

```
strcpy(p, "abc");
```

使用动态存储分配编写字符串函数

- 需求：编写函数实现连接两个字符串获得新串
  - 1、测量新串长度
  - 2、调malloc分配新串内存
  - 3、复制字符串到串
  - 4、调strcat拼接第二个串

使用动态存储分配编写字符串函数

```
char *concat(const char *s1, const char *s2)
{
    char *result;

    result = malloc(strlen(s1) + strlen(s2) + 1);
    if (result == NULL) {
        printf("Error: malloc failed in concat\n");
        exit(EXIT_FAILURE);//严厉处理方式
    }
    strcpy(result, s1);
    strcat(result, s2);
    return result; //堆中分配，不属函数局部
}
```

使用动态存储分配编写字符串函数

- concat函数调用方式：
  - p = concat("abc", "def");
  - p指向了新字符串“abcdef”
  - 思考该字符串中字符是否可改
- 使用动态分配的存储空间须小心
  - 内存空间需有借有还：动态分配空间不再需要时，须调用free函数释放空间

```
free(void *);
```

  - 静态分配，编译器分配和回收
- 否则，程序最终会耗尽内存空间

温故而知新——动态分配

- 方法

```
type *p;
p = (type *)malloc(n * sizeof(type));//单数据项、数组
if(p==NULL) {
    /* allocation failed; take appropriate action
*/
}
```

- 使用：当type类型变量、数组、字符串
- 释放：不再使用时
  - free(p);

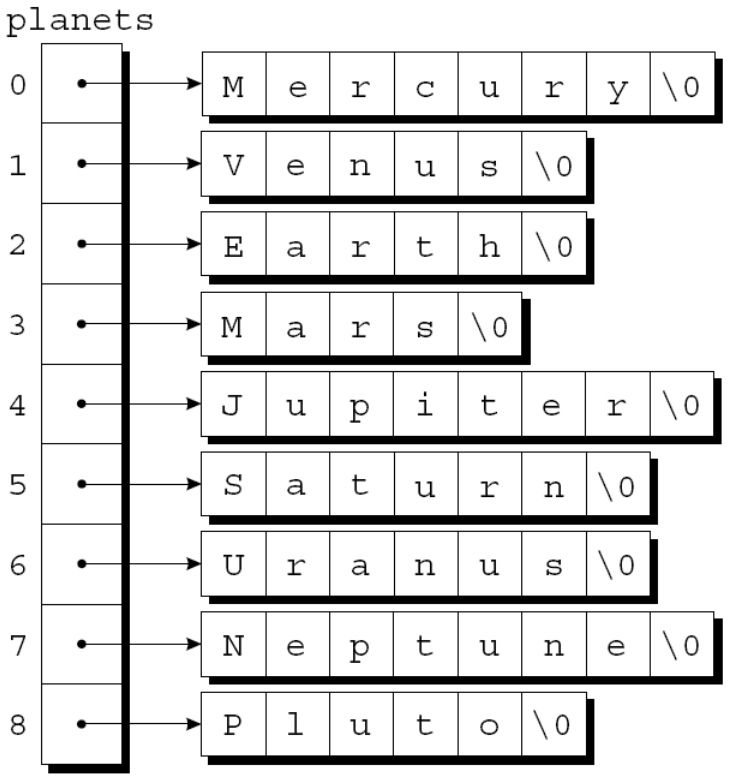
日程表（动分改进版）

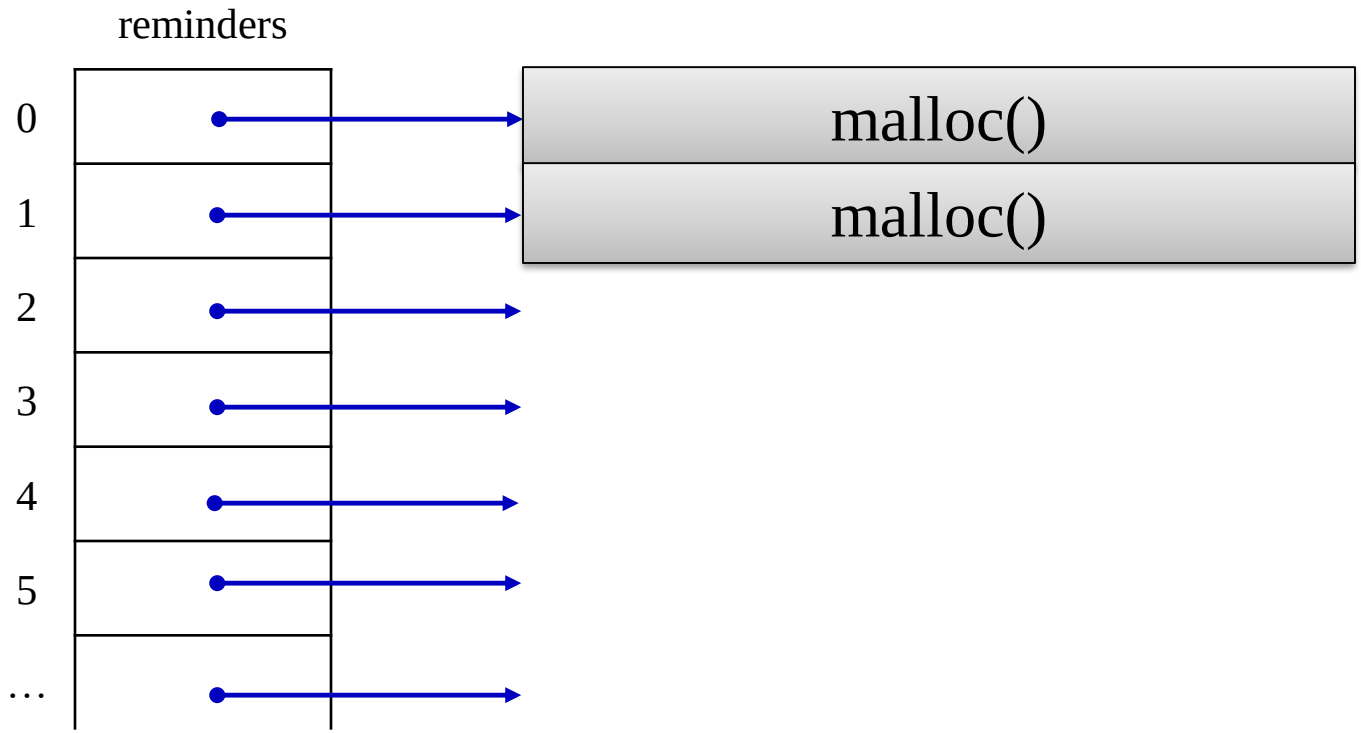
- 老版remind.c用二维字符数组造表
- 改版remind2.c，一维字符指针数组造表
  - 每个元素指针，指向动态分配内存，存放日程字符串
  - 类似13章行星名字数组
  - 只需改变程序8行内容（红色的部分）：
- 动态分配主要好处：
  - 1、按需分配
    - 日程数量动态增加
    - 每条日程长度动态确定
  - 2、插入日程时让位只需修改指针，不需实际移位已有日程条目，

日程表（动分改进版）

- 老版remind.c用二维字符数组造表
- 改版remind2.c用一维字符指针数组造表

5 Saturday class
5 6:00 - Dinner with Marge and Russ
7 10:30 - Dental appointment
24 Susan's birthday
26 Movie - "Chinatown"




reminders
0
1
2
3
4
5
…
malloc()
malloc()

remind2.c

```
/* Prints a one-month reminder list (dynamic string
 version) */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_REMIND 50 /* maximum number of reminders */
#define MSG_LEN 60 /* max length of reminder message */

int read_line(char str[], int n);
int main(void)
{
  char *reminders[MAX_REMIND];//指针数组大小仍固定
  char day_str[3], msg_str[MSG_LEN+1];
  int day, i, j, num_remind = 0;
```

```
for (;;) {
    if (num_remind == MAX_REMIND) {
      printf("-- No space left --\n");
      break;
    }//查满

    printf("Enter day and reminder: ");
    scanf("%2d", &day);
    if (day == 0)
      break;
    sprintf(day_str, "%2d", day);
    read_line(msg_str, MSG_LEN);//新日程

    for (i = 0; i < num_remind; i++)
      if (strcmp(day_str, reminders[i]) < 0)
        break;//定位
    for (j = num_remind; j > i; j--)
      reminders[j] = reminders[j-1];//改指针指向让位
```

```
    reminders[i] = malloc(2 + strlen(msg_str) + 1);
      if (reminders[i] == NULL) {
        printf("-- No space left --\n");
        break;
    }

      strcpy(reminders[i], day_str);
      strcat(reminders[i], msg_str);

      num_remind++;
    }

    printf("\nDay Reminder\n");
    for (i = 0; i < num_remind; i++)
      printf(" %s\n", reminders[i]);

    return 0;
}
```

每条日程
按需分配

```
int read_line(char str[], int n)
{
  int ch, i = 0;

  while ((ch = getchar()) != '\n')
    if (i < n)
      str[i++] = ch;
  str[i] = '\0';
  return i;
}
```

动态分配数组

- 动态分配数组和动态分配字符串类似。
- 有时用calloc分配并初始化。
- 有时用realloc按需对数组进行“扩展”或“缩减”。

使用malloc给数组分配存储空间

- 假设需n个整数的数组，程序执行时确定n值。

```
int *a;//抓手、句柄
a = malloc(n * sizeof(int));
```

- a：返回的数组地址，可作数组名

```
for (i = 0; i < n; i++)
    a[i] = 0;
```

calloc 函数

- calloc函数是malloc的另一种替代。原型:

```
void *calloc(size_t nmemb, size_t size);
```

  - nmemb：数组元素个数，size：每个元素字节数
  - 如要求空间无效，返回空指针
  - 分配后，把所有位设置为0

- 为n个整数的数组分配存储空间:

```
a = calloc(n, sizeof(int));
```

- 以1作为第一个参数，可为任何类型的数据项分配空间:

```
a = calloc(1, sizeof(int));
struct point { int x, y; } *p;
p = calloc(1, sizeof(struct point));
```

realloc函数

- 调整一个动态分配的数组的大小，原型:

```
void *realloc(void *ptr, size_t size);
```

  - ptr：malloc、calloc或realloc已分配的内存块
  - Size：新内存块的尺寸。
- 特点
  - 不会对添加进内存块的字节进行初始化。
  - 如不能按要求扩大内存块，返回空指针，不改变原有内存块数据。
  - 如以空指针作为第一个实际参数，作用与malloc函数一样。
  - 如果realloc函数被调用时以0作为第二个实际参数，那么它会释放掉内存块。

realloc函数

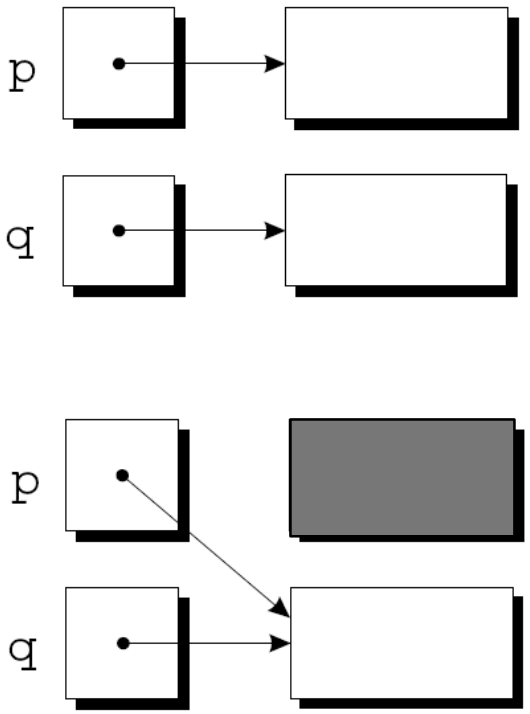
- 我们希望realloc能具有有效的合理性：
  - 当要求缩减尺寸时，realloc应该在原先的内存块上进行缩减
  - 当扩大尺寸时，也不应该对其进行移动；
- 如果无法扩大内存块，realloc会在别的地方开辟一块空间，然后把旧的内容复制过去。
- 一旦realloc函数返回，请一定要对指向内存块的所有指针进行更新，因为它可能会使内存块移动到了其他地方。

释放存储空间

- malloc等函数分配的内存块都来自堆(heap)
- 过于频繁分配可能会耗尽堆
- 更糟的是，程序可能分配了内存块，然后又丢失了对这些块的记录，因而浪费了空间。

释放存储空间

- 例如:
  - p = malloc(...);
  - q = malloc(...);
  - p = q;



释放存储空间

- 不可再被访问的内存块被称为垃圾（garbage）
- 有垃圾的程序存在内存泄漏（memory leak）问题。
- C没有垃圾收集器（garbage collector），需程序自己回收垃圾
  - 调用free函数释放不需要的内存

free 函数

- Free函数原型:
  - void free(void *ptr);
- 示例:

```
p = malloc(…);
q = malloc(…);
free(p);//p指向内存释放，但p还在
p = q;
```

“悬空指针”问题

- free(p)释放p指向的内存块，但不会改变p本身，p成为悬空指针（dangling pointers）
- 修改p指向的内存是严重的错误。

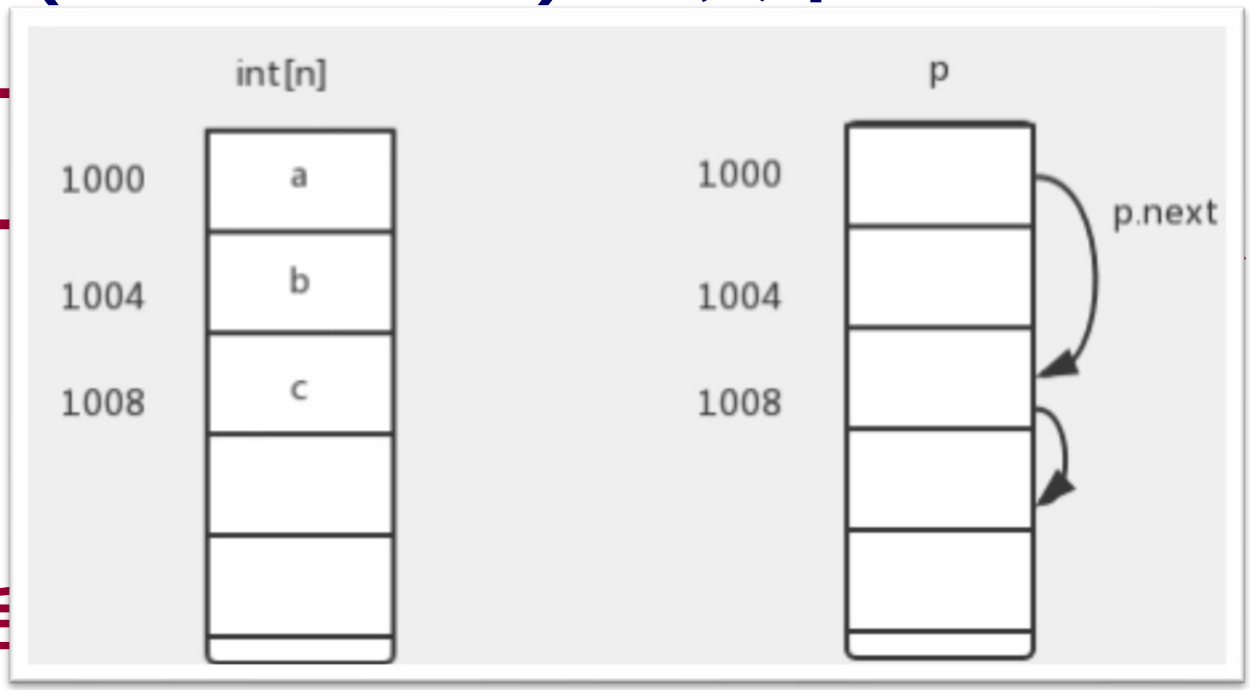
```
char *p = malloc(4);
free(p);
strcpy(p, "abc");    /*** WRONG ***/
```

- 悬空指针很难发现：因为几个指针可能指向相同的内存块
- 在释放内存块后，全部的指针都悬空了。

链表

链表

- 动态存储分配对建立表、树、图和其他链式数据结构是特别有用的
- 链表（Linked Lists）：火车
  - 由一
  - 每一 的指针：
  - 最后



链表

- 比数组更灵活
  - 允许按需扩大和缩小：容易在链表中插入和删除结点
- 另一方面，失去数组的“随机访问”能力:
  - 下标访问数组任意元素耗时相同，eg，mp3播放
  - 访问链表结点用时不同：结点距离链表头近，访问快；结点靠近链表尾，访问慢。eg, 磁带机

创建链表——1、声明结点类型

- 声明链表结点的结构

```
struct node {//任意结构标记名皆可
    int value;/*data stored in node */
    struct node *next;
};//node结构类型申明无法使用typedef方式
```

创建链表——2、声明表头

- 声明表头指针，指向链表第一个结点（记录链表开始位置）：
  - `struct node *first = NULL;//链表初始值为空。`

创建链表——3、创建结点

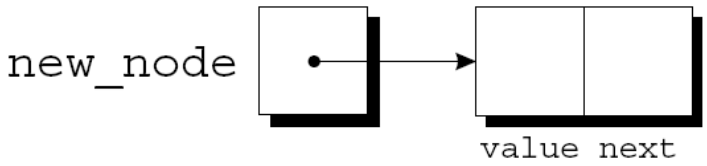
- 逐个创建结点，然后将其加入链表中。步骤：
  - 为结点分配内存单元；
  - 把数据存储到结点中；
  - 把结点插入到链表中。

创建链表——3.1创建结点

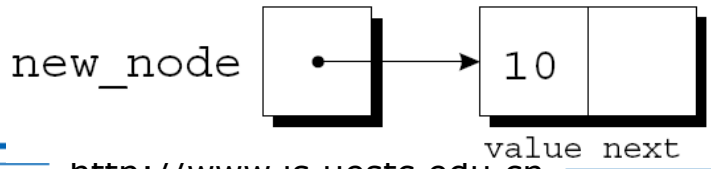
- 声明临时指针 (到该结点插入链表中为止)，用于动态分配结点内存:

```
struct node *new_node;
new_node = malloc(sizeof(struct node));
```

- 设置数据



(*new_node).value = 10;或右箭头选择运算符new_node->value = 10;



创建链表——3.2在链表中插入结点

- 链表好处之一
  - 可以在表中的任何位置添加结点。
- 表头插入
  - 作为新表头结点
- 链表中间插入
  - 有序链表

在链表的开始处插入结点

- 待插结点指向原表头结点

```
new_node->next = first;
```

- 表头指针指向该新结点:

```
first = new_node;
```

在链表的开始处插入结点

- 创建并设置新结点

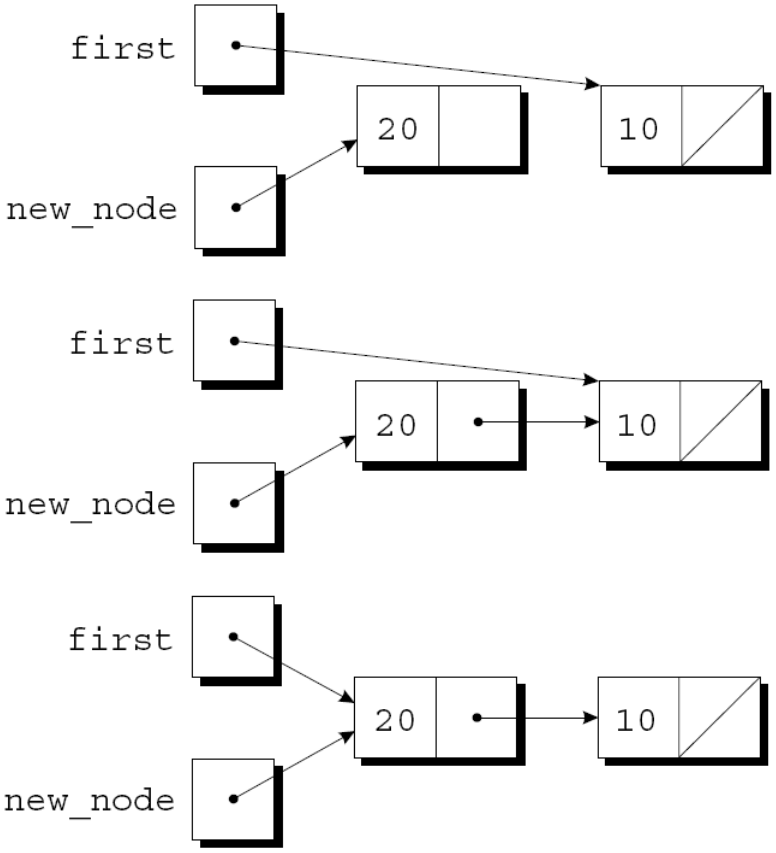
```
new_node =  malloc(sizeof(struct node));
new_node->value = 20;
```

- 抓住后链

```
new_node->next = first;
```

- 作为表头

```
first = new_node;
```



在链表的开始处插入结点

- 函数实现将整数n插入到list所指向的链表中:

```
struct node *add_to_list(struct node *list, int n){
  struct node *new_node;
  new_node = malloc(sizeof(struct node));
  if (new_node == NULL) {
    printf("Error: malloc failed in add_to_list\n");
    exit(EXIT_FAILURE);
  }
  new_node->value = n;
  new_node->next = list;
  return new_node;
}
```

在链表的开始处插入结点

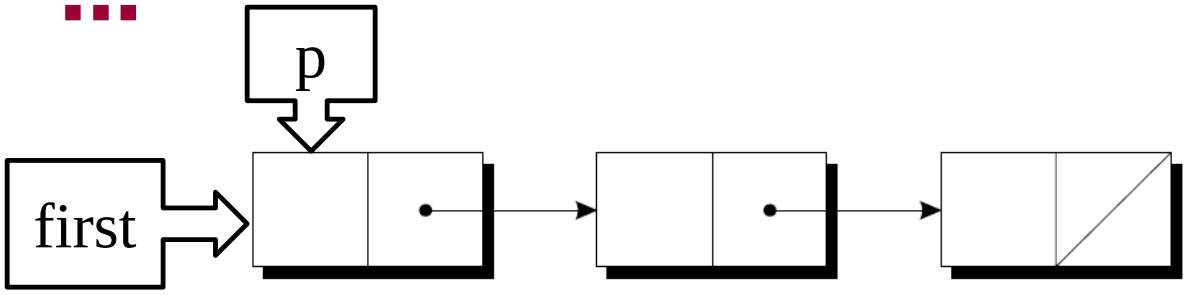
- 调用
  - `first = add_to_list(first, 10);`
  - `first = add_to_list(first, 20);`
- 返回新结点的指针，更新 first

搜索链表

- 从链表头结点开始，循环搜索链表结点，直到尾结点（指针为NULL）
  - 可用while语句，但for语句常常是首选。
- 使用指针 p 来跟踪“当前”结点：

```
for (p = first; p != NULL; p = p->next)
```

…



搜索链表

- 函数实现：查找数据为n的结点，找到则返回指向该结点的指针；否则返回NULL。

```
struct node *search_list(struct node *list, int n)
{
  struct node *p;

  for (p = list; p != NULL; p = p->next)
    if (p->value == n)
      return p;
  return NULL;
}
```

搜索链表

- 直接用list自身来搜索和跟踪：

```
struct node *search_list(struct node *list, int
n)
{
  for (; list != NULL && list->value != n;
          list = list->next)
      return list;
  return NULL;
}
```

- list是原始链表指针的副本，函数不会改变其值

从链表中删除结点

- 使用链表可以轻松删除不需要的结点。
- 删除一个结点分为3步:
  - 定位要删除的结点;
  - 前一结点“绕过”删除结点指向后一结点;
  - free删除结点;
- 需要操作删除结点和前一结点
  - 前结点指针、当前指针

从链表中删除结点

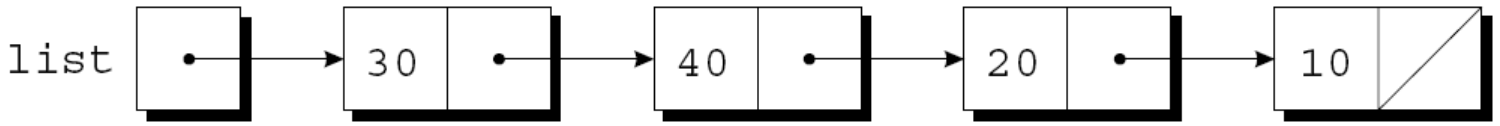
- 使用“追踪指针”方法：
  - 指向前一结点的指针（prev）
  - 指向当前结点的指针（cur）
- 假设要删除list链表的数据为n的结点，循环如下：

```
for (cur = list, prev = NULL;
     cur != NULL && cur->value != n;
     prev = cur, cur = cur->next)  ;//空循环体
```

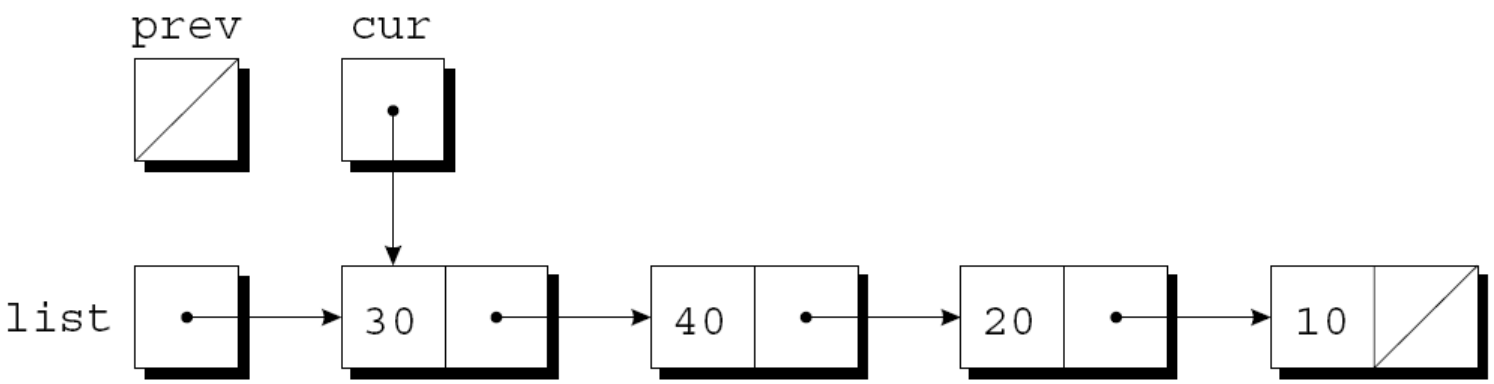
- 循环结束，cur指向需删除结点，prev指向它之前的结点。

从链表中删除结点

- 假设list 如下图所示，n 为 20:

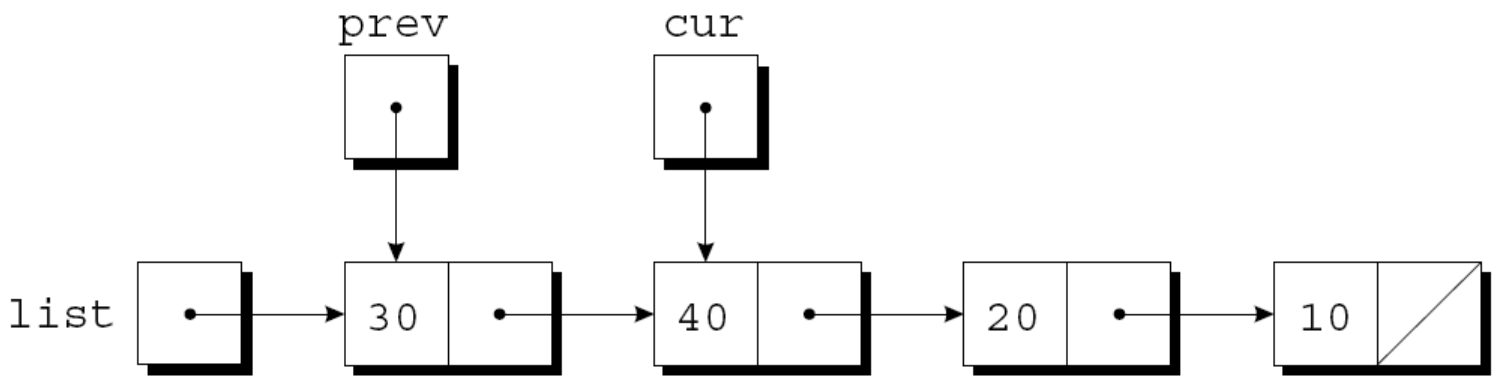


- 在执行完 cur = list, prev = NULL后，情形为：



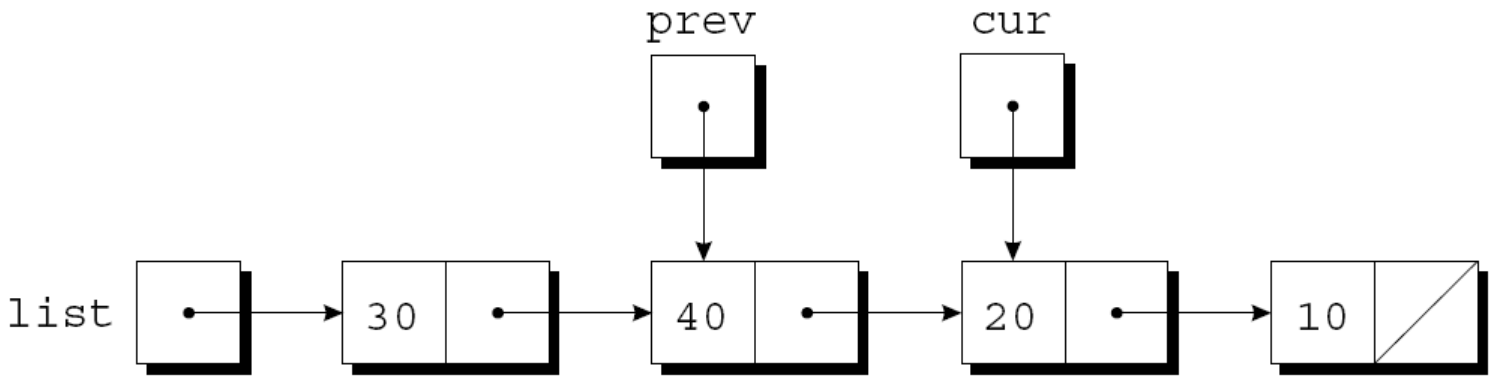
从链表中删除结点

- 首次循环
  - cur != NULL && cur->value != n 为真
- 循环继续
  - prev = cur, cur = cur->next



从链表中删除结点

- 第三次循环，cur指向包含20的结点
  - cur->value != n 为假，循环结束；
  - 删除结点找到。



温故而知新——创建链表

- 1、声明链表结点的结构

```
struct node {//任意结构标记名皆可
        int value;/*data stored in node */
        struct node *next;
};//node结构类型申明无法使用typedef方式
```

- 2、声明表头指针，指向链表第一个结点（记录链表开始位置）：
  - struct node *first = NULL;//链表初始值为空。
- 3、逐个创建结点，然后将其加入链表中。步骤：
  - 为结点分配内存单元；
  - 把数据存储到结点中；
  - 把结点插入到链表中。

温故而知新——链头插入结点

- 创建并设置新结点

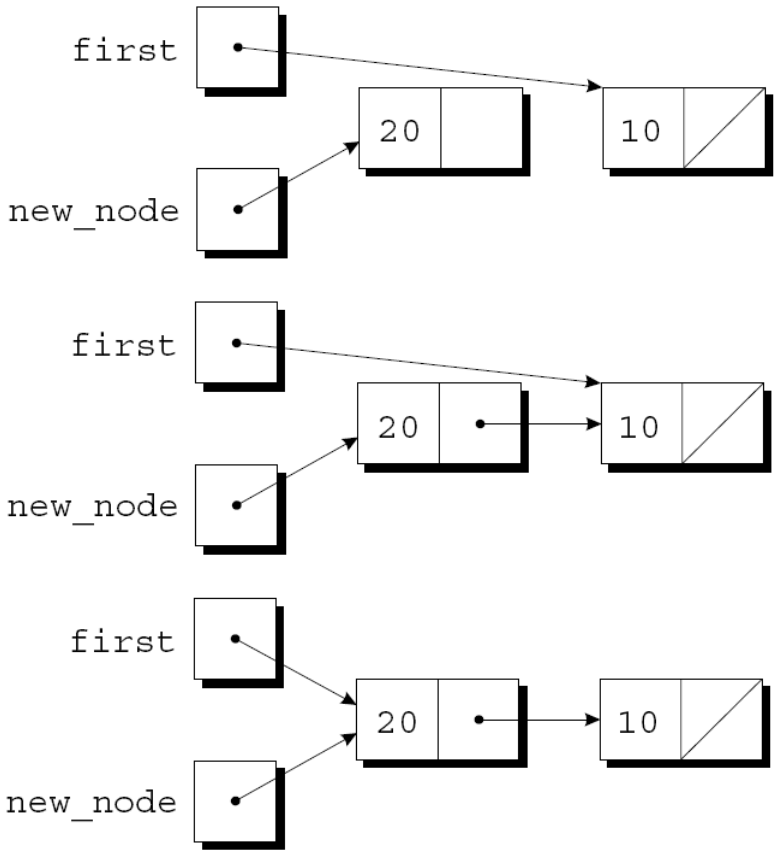
```
new_node =  malloc(sizeof(struct node));
new_node->value = 20;
```

- 抓住后链

```
new_node->next = first;
```

- 作为表头

```
first = new_node;
```



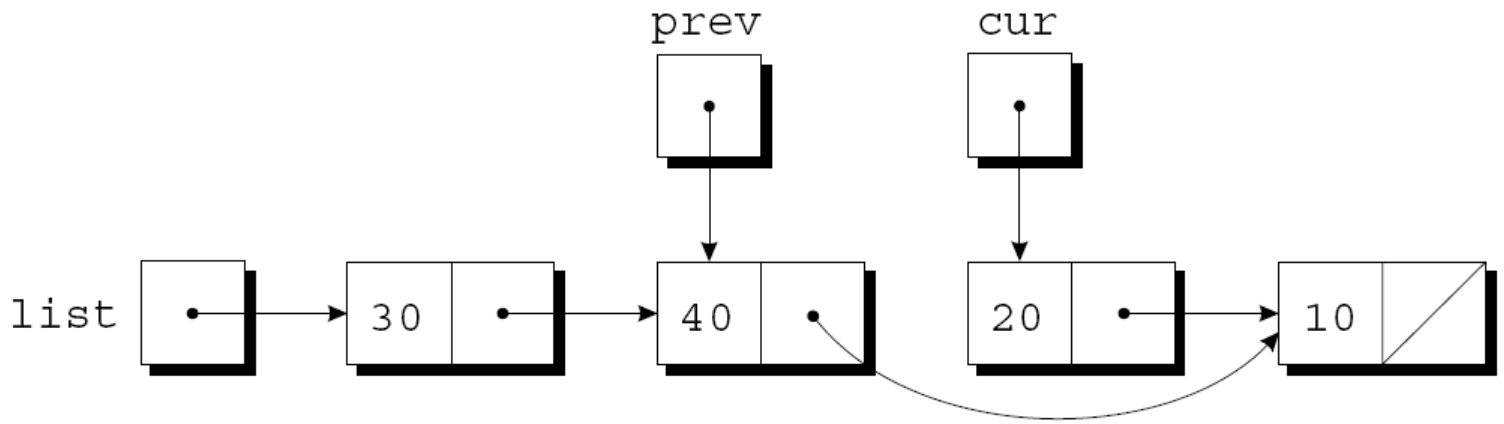
从链表中删除结点

- 搜索定位待删（数据为n）结点，循环如下：

```
for (cur = list, prev = NULL;
     cur != NULL && cur->value != n;
     prev = cur, cur = cur->next)  ;//空循环体
```

- 绕过删除结点

```
prev->next = cur->next;
```



- 释放删除结点内存

```
free(cur);
```

从链表中删除结点

```
struct node *delete_from_list(struct node *list, int n)
{
  struct node *cur, *prev;

  for (cur = list, prev = NULL;
       cur != NULL && cur->value != n;
       prev = cur, cur = cur->next)     ;

  if (cur == NULL)
    return list;                  /* n was not found */
  if (prev == NULL)
    list = list->next;            /* n is in the first node */
  else
    prev->next = cur->next;       /* n is in some other node */
  free(cur);
  return list;
}
```

有序链表

- 结点有序的链表（按结点中的数据排序）。
- 插入结点困难一些（不再始终把结点放置在链表的开始处）。
- 但是搜索会更快（在到达期望结点应该出现的位置后，就可以停止查找了）。

程序：维护零件数据库(改进版)

- 零件数据库程序链表版。
- 使用链表有两个好处：
  - 不需要事先限制数据库大小
  - 容易按零件编号对数据库排序（升序）
- 原版零件数据库无序

程序：维护零件数据库(改进版)

- 声明结点，part结构包含新成员——指向下一结点的指针：

```
struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
  struct part *next;
};
```

- 链表头指针

```
struct part *inventory = NULL;
```

程序：维护零件数据库(改进版)

- 新版大多数函数类似于原版
- find_part 和 insert 函数变复杂了

程序：维护零件数据库(改进版)

- find_part的搜索循环如下:

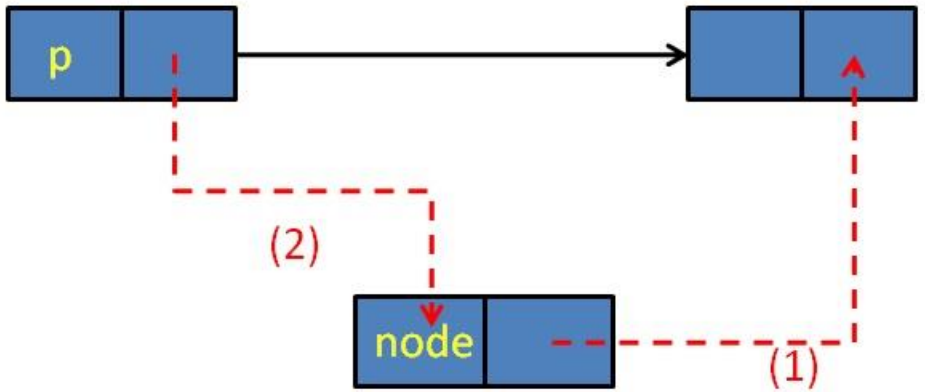
```
for (p = inventory;
     p != NULL && number > p->number;
     p = p->next)    ;//查找定位，空循环
```

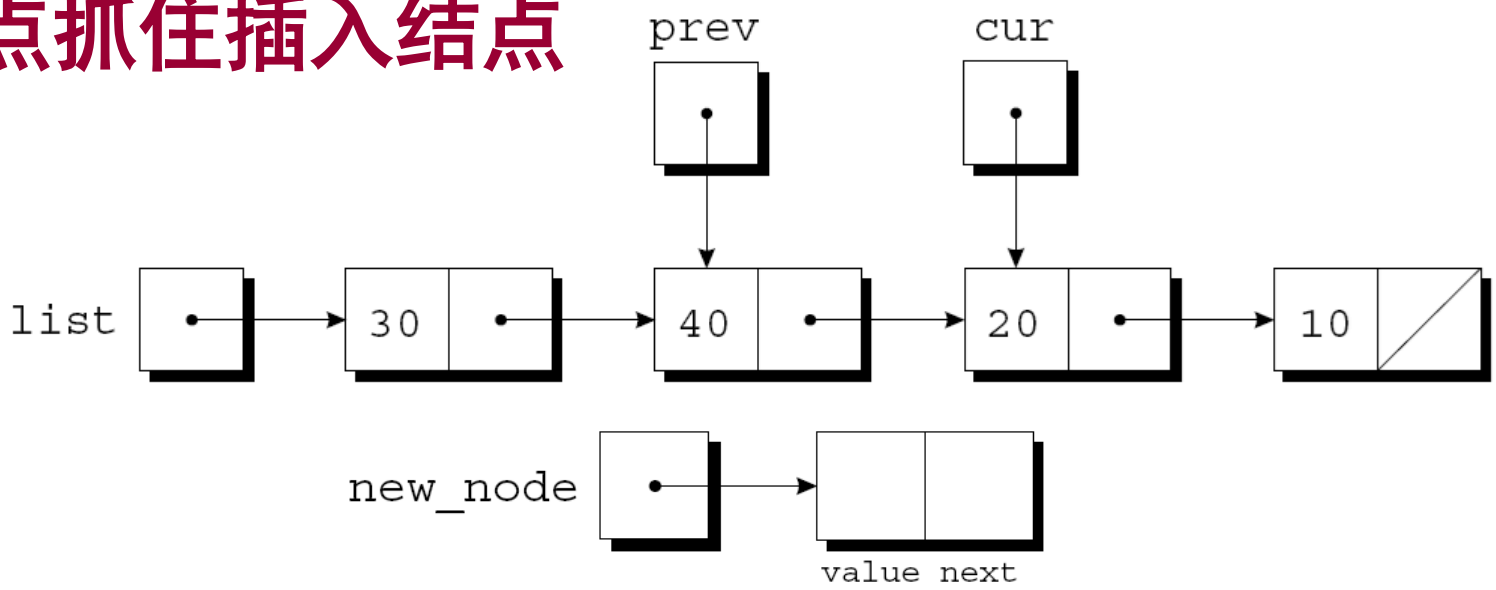
- 循环结束，需判断零件是否找到:

```
if (p != NULL && number == p->number)
     return p;
```

程序：维护零件数据库(改进版)

- 零件插入
  - 搜索定位
  - 抓住后链
  - 前结点抓住插入结点





程序：维护零件数据库(改进版)

- 搜索定位

```
for (cur = inventory, prev = NULL;
     cur != NULL && new_node->number > cur->number;
     prev = cur, cur = cur->next);
```

- 循环终止

```
if(cur != NULL && new_node->number == cur->number)
```

  - 成立，则零件已经在表里，不用插入
  - 否则，插入新结点到prev和cur指向的结点之间

inventory2.c

```
/* Maintains a parts database (linked list version) */

#include <stdio.h>
#include <stdlib.h>
#include "readline.h"
#define NAME_LEN 25

struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
  struct part *next;
};

struct part *inventory = NULL;   /* points to first part */

struct part *find_part(int number);
void insert(void);
void search(void);
void update(void);
void print(void);
```

```
/*************************************************************
*main: Prompts the user to enter an operation code,         *
*       enters an illegal code.                              *
*************************************************************/
int main(void)
{
  char code;

  for (;;) {
    printf("Enter operation code: ");
    scanf(" %c", &code);
    while (getchar() != '\n');  /* skips to end of line */
```

```
    switch (code) {
      case 'i': insert();
                break;
      case 's': search();
                break;
      case 'u': update();
                break;
      case 'p': print();
                break;
      case 'q': return 0;
      default:  printf("Illegal code\n");
    }
    printf("\n");
  }
}
```

```
/*****************************************************
* find_part: Looks up a part number in the inventory      *
*            number is not found, returns NULL.           *
*****************************************************/
struct part *find_part(int number)
{
  struct part *p;

  for (p = inventory;
       p != NULL && number > p->number;
       p = p->next)
    ;
  if (p != NULL && number == p->number)
    return p;
  return NULL;
}
```

```
/*********************************************************
 * insert: Prompts the user for information about a new  *
*********************************************************/
void insert(void)
{
  struct part *cur, *prev, *new_node;

  new_node = malloc(sizeof(struct part));
  if (new_node == NULL) {
    printf("Database is full; can't add more parts.\n");
    return;
  }

  printf("Enter part number: ");
  scanf("%d", &new_node->number);
```

//构造新零件结点

```
  for (cur = inventory, prev = NULL;
       cur != NULL && new_node->number > cur->number;
       prev = cur, cur = cur->next)
    ;//定位
  if (cur != NULL && new_node->number == cur->number) {
    printf("Part already exists.\n");
    free(new_node);
    return;
  }

  printf("Enter part name: ");
  read_line(new_node->name, NAME_LEN);
  printf("Enter quantity on hand: ");
  scanf("%d", &new_node->on_hand);//完善零件信息

  new_node->next = cur;
  if (prev == NULL)
    inventory = new_node;
  else
    prev->next = new_node;
}
```

改进：新零件号查找，不存在再构造新零件

```
/*************************************************
 * search: Prompts the user to enter a part number, then  *
*************************************************/
void search(void)
{
  int number;
  struct part *p;

  printf("Enter part number: ");
  scanf("%d", &number);
  p = find_part(number);
  if (p != NULL) {
    printf("Part name: %s\n", p->name);
    printf("Quantity on hand: %d\n", p->on_hand);
  } else
    printf("Part not found.\n");
}
```

```
/*************************************************
 * update: Prompts the user to enter a part number.       *
*         database.                                       *
 *************************************************/
void update(void)
{
  int number, change;
  struct part *p;

  printf("Enter part number: ");
  scanf("%d", &number);
  p = find_part(number);
  if (p != NULL) {
    printf("Enter change in quantity on hand: ");
    scanf("%d", &change);
    p->on_hand += change;
  } else
    printf("Part not found.\n");
}
```

```
/*************************************************
 * print: Prints a listing of all parts in the database,  *
 *        showing the part number, part name, and         *
 *        quantity on hand. Part numbers will appear in   *
 *        ascending order.                                *
 *************************************************/
void print(void)
{
  struct part *p;
  printf("Part Number   Part Name                    "
         "Quantity on Hand\n");
  for (p = inventory; p != NULL; p = p->next)
    printf("%7d       %-25s%11d\n", p->number, p->name,
           p->on_hand);
}
```

第17章
~完~