南京大学 计算机科学与技术系
Department of Computer Science & Technology, NJU

Chapter 8

结构与链表及联合

刘奇志

- **结构的基本概念**
  - 结构类型的构造
  - 结构变量的定义与初始化
  - 结构的操作
    - **含有指针成员的结构及其操作**
- **用指针操纵结构**
  - **用指针操纵含有指针成员的结构**
  - **链表的构造与操作**
- 结构数组与名表
- 基于结构数组的信息检索程序
- 联合

问题的提出

- 描述一位学生的信息：

num	name	F/M	age	score	addr
181220999	Hans	M	19	395	Nanjing

- 数组？

（数组类型用于表示 固定多个 同类型 的数据群体）

结构(struct)类型

元素、属性

- 结构类型用于表示由固定多个 类型可以不同 的成员所构成的数据群体。
  - 固定多个
  - 类型可以不同的数据群体（含义可以不同的相关信息）
  - 成员间在逻辑上没有先后次序关系，其说明次序仅影响成员的存储安排，不影响操作，成员有成员名
  - 相当于其他高级语言中的“记录”
- 数组类型：
  - 固定多个
  - 同类型（相同意义的相关信息）
  - 元素间在逻辑上有先后次序关系，按序连续存储，元素有下标

结构类型的构造

```
struct Student
{
  int number;   //成员
  char name;    //成员
  int age;      //成员
};

typedef struct
{
  int number;
  char name;
  int age;
}Student;
```

一种 tag

宣布组成的
成员名称和成员类型

```
struct Date
{
   int month;
   int day;
   int year;
};
```

注意：构造结构类型时，
花括号中至少要定义一个成员。
除void类型和本结构类型外，
结构成员可以是其他任意的类型。

- 结构类型标识符与其成员或其他变量可重名，
- 结构成员和其他变量也可以重名，
- 同一个结构的各个成员不能重名。

- 即使两个结构类型中的成员类型、名称、顺序都完全一致，它们也是不同的结构类型。不同结构类型的成员可以重名，不过结构类型标识符不能重名。

- 如果在函数内部构造结构类型，则该函数之外此结构类型不可用。一般把结构类型的构造放在文件头部，也可以把结构类型的构造放在头文件中。

结构变量的定义

- 可以用构造好的结构类型来定义结构变量。
  - 比如，

```
struct Student s1, s2;
struct Date d1, d2;
```

  - 前面的struct可以省略

- 也可以在构造结构类型的同时直接定义结构变量，
  - 比如，

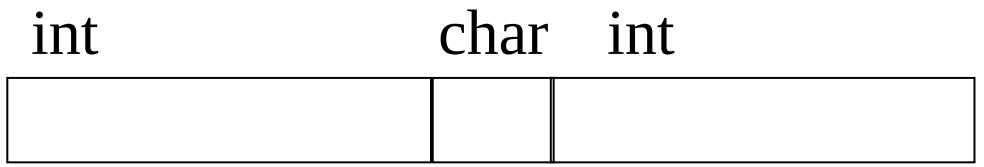
```
struct Student
{       int number;
        char name;
        int age;
} s1, s2;
```

```
struct Employee
{
  int number;
  char name;
  struct Date
  {
     int   year;
     int   month;
     int   day;
  } birthday ; //其他结构类型的变量可以作为本结构类型的成员
} e ;
```

```
struct Date
{   int   year;
    int   month;
    int   day;
};
struct Employee
{   int number;
    char name;
    Date birthay;
} e ;
```

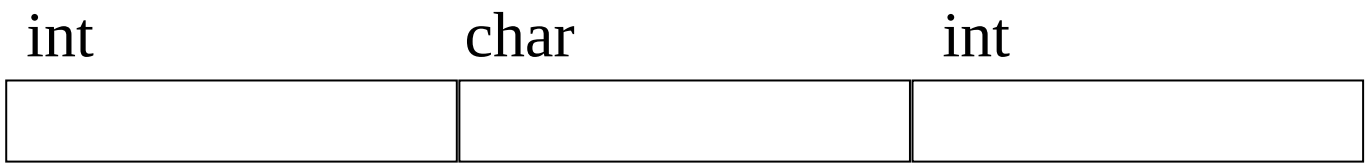
- 系统按构造时的顺序为各个成员分配空间

```
struct Student
{
   int number;
   char name;
   int age;
} s ;
```



（a）

- 系统往往以字为单位给结构变量分配空间



（b）

（c）

- 结构变量一般不加register修饰

结构变量的初始化

- 可以在定义结构变量的同时，给各个成员赋值，即结构变量的初始化。
  - 比如，

```
Student s1, s2 = {1220001, 'T', 19};
Employee e = {1160007, 'J', {1996, 12, 26}};
```

  - 注意：在构造一个结构类型时，不能对其成员进行初始化，因为构造类型时，编译器不分配存储空间。比如，

```
struct Student
{
 int number = 1220001;      //此处的初始化是错误的
 char name;
 int age;
};
```

C++11新标准允许给一个默认值

```
struct
{  char  name[20];
   struct Date
   {     int   year;
         int   month;
         int   day;
    }birthday;
} s={"Joe", {1996,12,14}};
```

```
char  *name;
```

```
s.name = "Joe";
```
✗

```
s.name = "Joe";
```

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

```
scanf("%s", s.name);
```
✗

结构的操作

- **对结构的操作常常是通过成员操作符操作结构变量的成员完成的。访问成员的格式为：**
  - <结构变量名> . <成员名>

    ```
    s2.age = 19;
    ```

  - 点号是成员操作符，它是双目操作符，具有1级优先级，结合性为自左向右。
  - 由于对成员的访问不是按次序，而是按名称访问，所以，构造结构类型时成员的排列顺序无关紧要。

- 如果某成员类型是另一个结构类型，则可以用若干个成员操作符访问最低一级的成员。比如，

```
e.birthday.year = 1996;
```

```
struct Employee
{
  int number;
  char name;
  struct Date
  {
     int   year;
     int   month;
     int   day;
  } birthday ;
} e ;
```

- 结构类型可以含有**指针成员**，对指针成员的操作方法与其他类型成员类似。比如，

```
struct
{
  int no;
  int *p;
} s ;

s.no = 1001;
s.p = &s.no;
```

赋值操作

- 相同结构类型的不同变量之间可以直接相互赋值，其实质是两个结构变量相应的存储空间中的所有成员数据直接拷贝。比如，

```
Employee e1, e2;
e1 = e2;
```

  - 或者，

```
typedef Employee Employ
Employee e1;
Employ e2;
e1 = e2;
```

- 不同结构类型的结构变量之间不能相互赋值，

下面a、b两个结构变量不可以相互赋值：

```
struct
{     int no;
      char name;
} a ;

struct
{     int no;
      char name;
} b ;
a = b;  ×
```

结构类型数据作为函数参数/返回值

- 可作为参数传给函数：默认参数传递方式为**值传递**

  （实参和形参都是结构变量名，类型相同；但实参和形参代表两个不同的结构变量，运行时分配不同的存储空间。）

- 函数也可以返回一个

  结构类型的值（结构 型函数）

例 验证结构类型参数的传值传递方式。

```
void myFun(Stu s1)
{ s1.name = 'J';
  s1.score = 100.0;
  printf("%c: %d", s1.name, s1.score);
}

int main( )
{ struct Stu stu1;
  stu1.name = 'T';
  stu1.score = 90.0;
  printf("%c: %d", stu1.name, stu1.score);
  myFun(stu1);
  printf("%c: %d", stu1.name, stu1.score);
  return 0;
}
```

```
enum FeMale {F, M};
struct Stu
{
  int id;
  char name;
  FeMale s;
  int age;
  float score;
};
```

程序结果会显示:

```
T: 90.0
J: 100.0
T: 90.0
```

用指针操纵结构

重点

- 将某结构类型变量的地址赋给基类型为该结构类型的指针变量，则可以用这个指针变量操纵该结构变量的成员，这时成员操作符写成箭头形式（->），而不是点形式（.）。比如，

```
struct
{
  int no;
  float score;
} s, *ps ;
ps = &s;
ps -> no = 1001;          //相当于(*ps).no或s.no
ps -> score = 90.0;       //相当于(*ps).score或s.score
```

重点

- **如果有成员是另一结构类型的指针变量，则可以用若干个箭头形式的成员操作符访问最低一级的成员。比如，**

```
struct
{
  Student *p1;
  float score;
} s, *pps ;
pps = &s;
pps -> p1 = &s1;
pps -> p1 -> number = 1220001;
pps -> p1 -> name = 'Q';
pps -> p1 -> age = 19;
pps -> score = 85;
```

```
struct Student
{
      int number;     //成员
      char name;      //成员
      int age;  //成员
} s1;
```

重点

- **为了提高程序的效率，函数间传递结构类型的数据时，实参可以用结构变量的地址，形参用相同结构类型的指针。**

传值方式-效率不高

- 例 用指针变量操纵结构。

```
int main( )
{
    struct Date d1;
    scanf("%d%d%d", &d1.year, &d1.month, &d1.day);
    Days(d1);

    return 0;
}
```

d1


```
void Days(struct Date d2)
```

d2


```
struct Date
{
    int year;
    int month;
    int day;
    int yearday;
};
```

```
void Days(struct Date d2)
{
    int monthtable[ ][13]= {
        {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
        {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}};

    int i, leap = 0;
    d2.yearday = d2.day;

    if( ((d2.year %4 == 0) && (d2.year %100 != 0))
                          || (d2.year % 400 == 0) )
        leap = 1;

    for(i = 1; i < d2.month; i++)
        d2.yearday += monthtable[leap][i];
    printf("所输入的日期是该年的第几天: %d", d2.yearday);
}
```

传址方式-提高效率

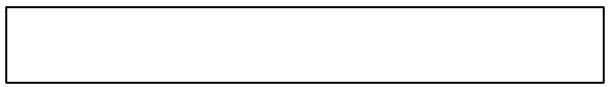
- 例 用指针变量操纵结构。

d1


```
int main( )
{
    struct Date d1;
    scanf("%d%d%d", &d1.year, &d1.month, &d1.day);
    Days(&d1);

    return 0;
}
```

```
void Days(struct Date *p)
```

p



```
struct Date
{
    int year;
    int month;
    int day;
    int yearday;
};
```

```
void Days(struct Date *p)
{
    int monthtable[ ][13]= {
        {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
        {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}};

    int i, leap = 0;
    p -> yearday = p -> day;

    if( ((p -> year %4 == 0) && (p -> year %100 != 0))
                             || (p -> year % 400 == 0) )
        leap = 1;

    for(i = 1; i < p -> month; i++)
        p -> yearday += monthtable[leap][i];
    printf("所输入的日期是该年的第几天: %d", p -> yearday);
}
```

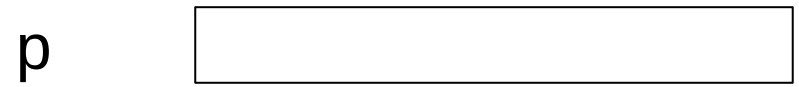
```
Days(&d1);
```

传址方式-提高效率、“返回”结果

- 例 用指针变量操纵结构。


```
int main( )
{
    struct Date d1;
    scanf("%d%d%d", &d1.year, &d1.month, &d1.day);
    Days(&d1);
    printf("所输入的日期是该年的第几天：%d", d1.yearday);
    return 0;
}
```

```
void Days(struct Date *p)
```




```
struct Date
{
    int year;
    int month;
    int day;
    int yearday;
};
```

```
void Days(struct Date *p)
{
     int monthtable[ ][13]= {
          {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
          {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}};

     int i, leap = 0;
     p -> yearday = p -> day;

     if( (p -> year %4 == 0) && (p -> year %100 != 0))
                              || (p -> year % 400 == 0) )
          leap = 1;

     for(i = 1; i < p -> month; i++)
          p -> yearday += monthtable[leap][i];
}
```

- 如果不需要通过参数返回数据，则可以用const避免函数的副作用。比如，

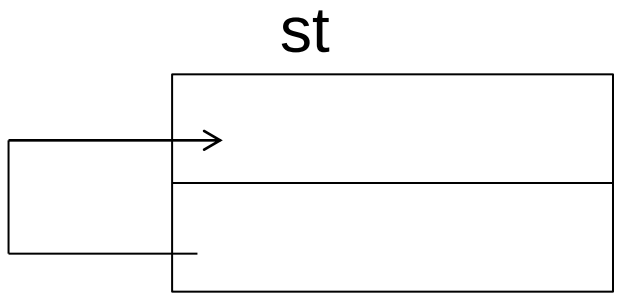
```
void G(const Date *p)
{
  …
  p -> day = 20;   //会出错，因为不能改变p所指向的数据
  …
}
```

- 函数也可以返回一个结构变量的地址。

重难点

- 结构类型不可以含有本结构类型成员。
- 结构类型可以含有**本结构类型的指针成员**。比如，

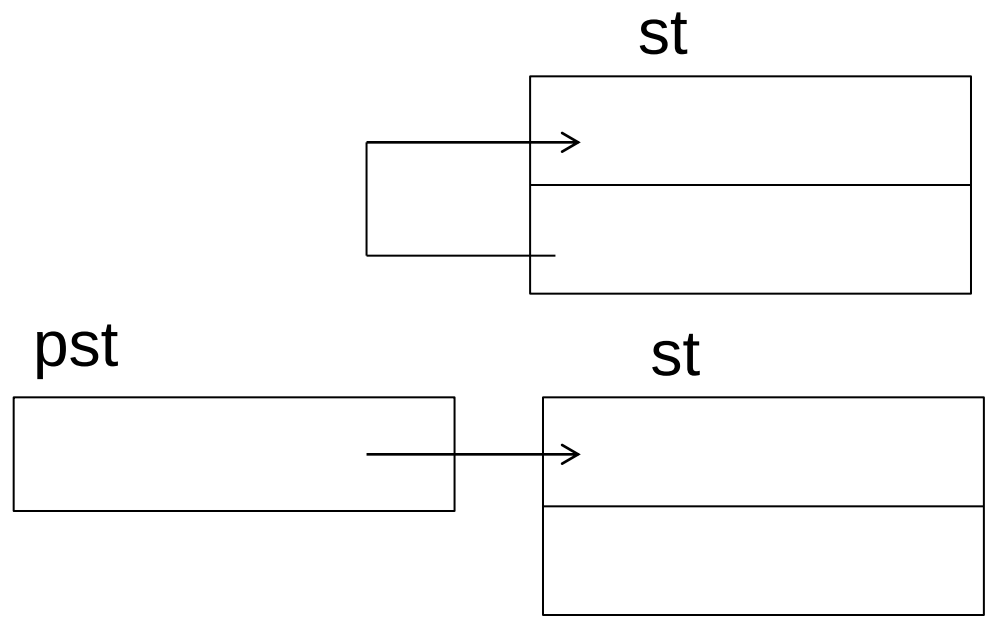
```
struct Stup
{
  int no;
  Stup *p0;
} st;

st.p0 = &st;
```



重难点

- 结构类型不可以含有本结构类型成员。
- 结构类型可以含有**本结构类型的指针成员**。比如，

```
struct Stup
{
  int no;
  Stup *p0;
} st;
```

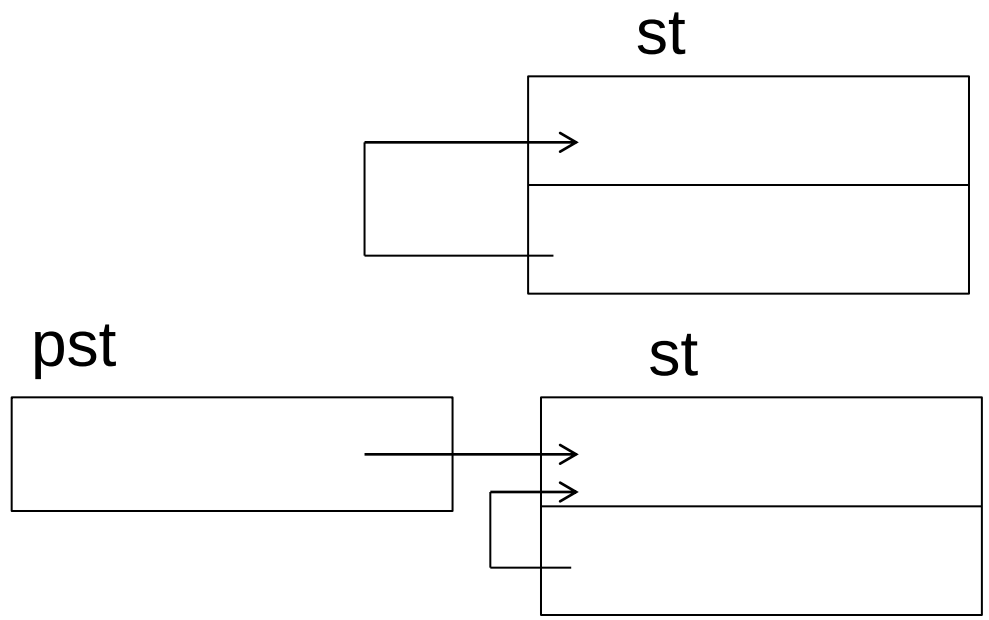



```
st.p0 = &st;
Stup *pst = &st;
```

重难点

- 结构类型不可以含有本结构类型成员。
- 结构类型可以含有**本结构类型的指针成员**。比如，

```
struct Stup
{
  int no;
  Stup *p0;
} st;
```



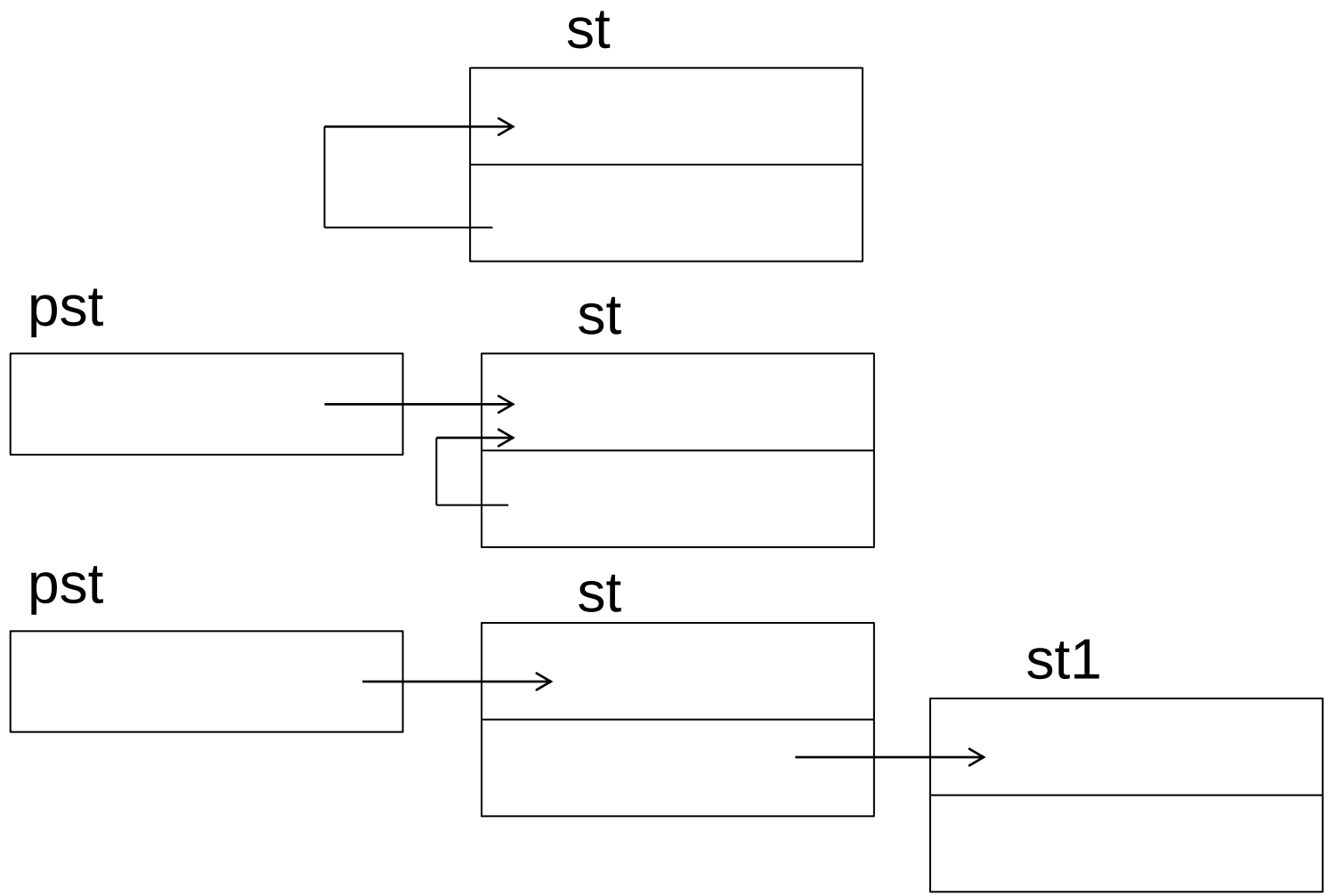

```
st.p0 = &st;
Stup *pst = &st;
pst -> p0 = &st;
```

重难点

- 结构类型不可以含有本结构类型成员。
- 结构类型可以含有**本结构类型的指针成员**。比如，

```
struct Stup
{
  int no;
  Stup *p0;
} st;
```

```
st.p0 = &st;
Stup *pst = &st;
pst -> p0 = &st;
Stup st1;
pst -> p0 = &st1;
```



链表的构造与操作

重难点

- 链表的建立
- 链表中节点的插入与删除
- 整个链表的输出与删除
- 基于链表的排序与检索程序

问题的提出

……

3）对输入的若干个正整数进行排序（先输入各个正整数，最后输入一个结束标志 -1）

可能需要大量数据搬迁

可能找不到足够的连续空间（(int *)malloc(sizeof(int)*max_len)）

4-1）输入一个数，插入到已经排好序的若干个数当中，保持原来的大小顺序

4-2）删除一个数，保持原来的大小顺序

？

？

- **数组的缺陷：**
  - 数组的长度一般在定义前就已确定，所占内存空间在其生存期始终保持不变，即使可变，由于数组元素的有序性，删除一个元素可能会引起大量数据的移动而降低效率，插入一个元素不仅可能会引起大量数据的移动，还可能会受数组所占内存空间大小的限制。

- **链表**
  - 实际应用中，常常需要表示一种在程序运行期间元素个数可以随机增加或减少、所占内存空间大小可动态变化、数据元素在逻辑上连续排列而物理上并不占用连续存储空间的数据群体，C语言用链表（list）来表示这种数据群体。

链表的建立

- 链表不是一种数据类型，而是一种通过程序将若干个同类型的数据（节点）链接起来的数据结构。链表中的每个节点是一个结构类型的动态变量，这种结构类型的若干个成员中至少有一个指针类型的成员，其构造形式为：

```
struct Node
{
  int data;   //存储数据
  Node *next;//存储下一个节点的地址
};
```

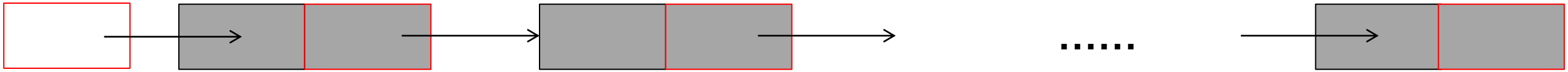
单向链表

- 注意，结构类型的成员不能是本结构类型的变量，但可以是本结构类型的指针变量。正是通过指针变量得以将若干个节点链接起来，从而完成链表的建立。可见，链表中的各节点不必存放在连续的内存空间中。

- `int a[10];`

- `(Node *)malloc(sizeof(int)*10);`

- 链表




```
struct Node
{
        int data;
        Node *next;
};
```

产生一个新节点

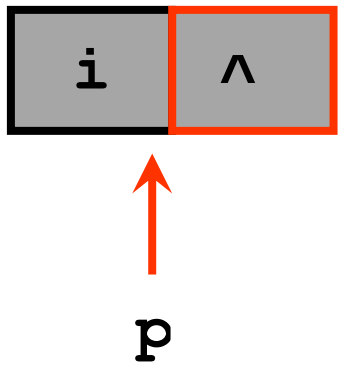
```
(Node *)malloc(sizeof(Node));
```

```
Node *p = (Node *)malloc(sizeof(Node));
```

```
scanf("%d", &p->data); // p->data = i;
```

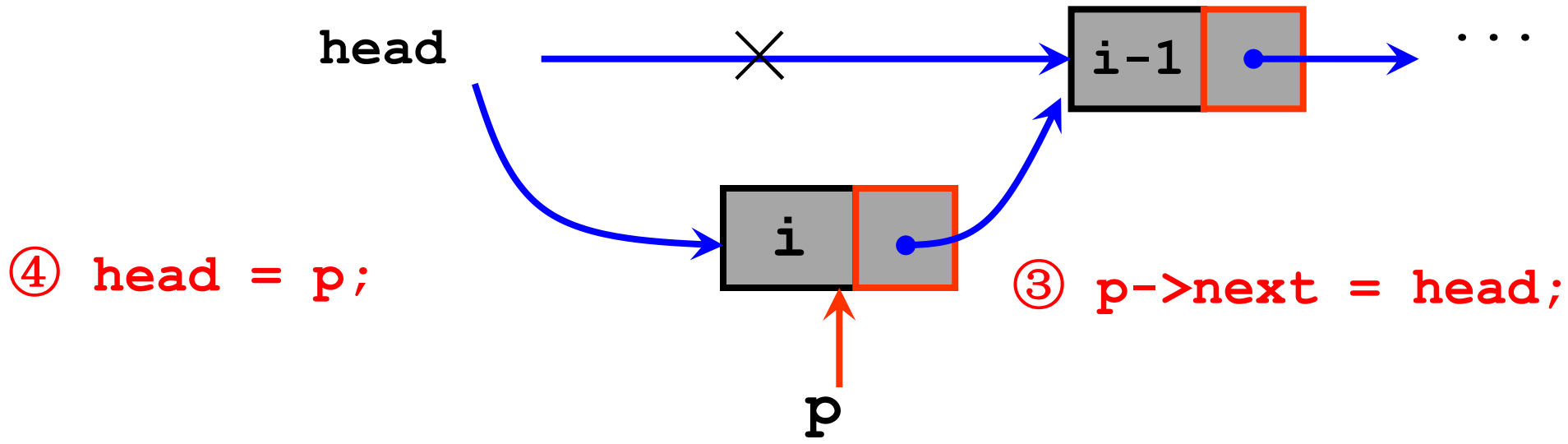
```
p->next = 0; // p->next = NULL;
```

```
struct Node
{
        int data;
        Node *next;
};
```



（1）头部插入节点方式建立链表

① `for(i=0; i<n; i++)`



② `p = (Node *)malloc(sizeof(Node));`
`p->data = i;`

指针变量　赋值　地址

```
#define N 10
Node *InsCreate( )
{
  Node *head = NULL;

  for(int i = 0; i < N; i++)
  {  Node *p = (Node *)malloc(sizeof(Node));//创建新节点= new Node;

     p -> data = i; //也可给新节点的数据成员输入值
     p -> next = head;    //head的值赋给新节点的指针成员
     head = p; //将p这个指针变量的值赋给指针变量head
  }
  return head;
}
```

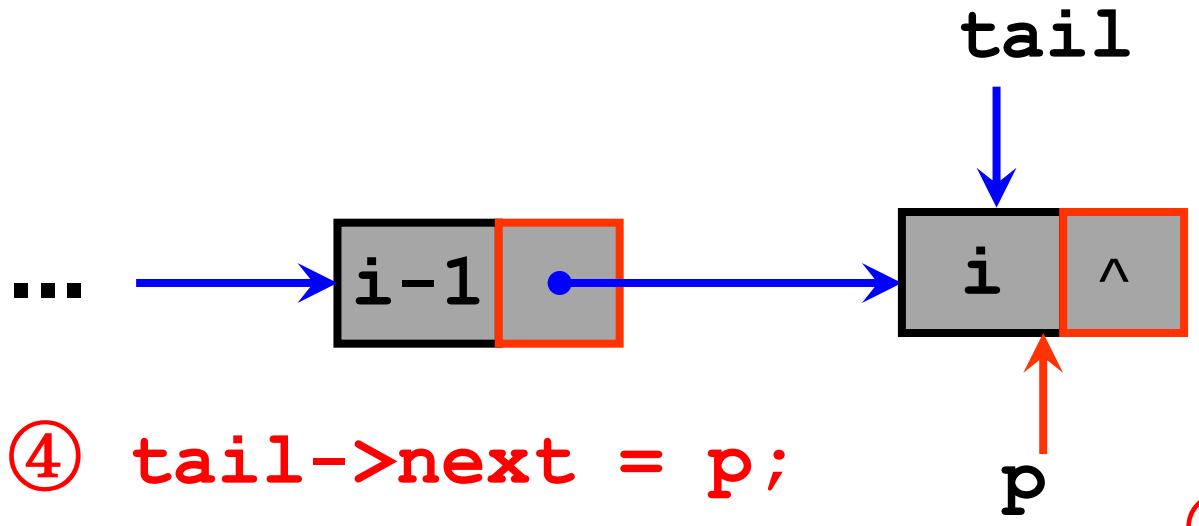
这样，第十个节点成为链表的头节点，只要知道头节点的地址（存于head中），就可以访问链表中的所有节点。

指针类型返回值：一般用来返回一组数据 链表

```
int main( )
{
      Node *h = InsCreate( );
      PrintList(h);
      ……
```

(2) 尾部追加节点方式建立链表

① `for(i=0; i<n; i++)`



⑤ `tail = p;`

④ `tail->next = p;`

③ `p->next = NULL;`

② `Node *p = (Node *)malloc(sizeof(Node));`
`p->data = i;`

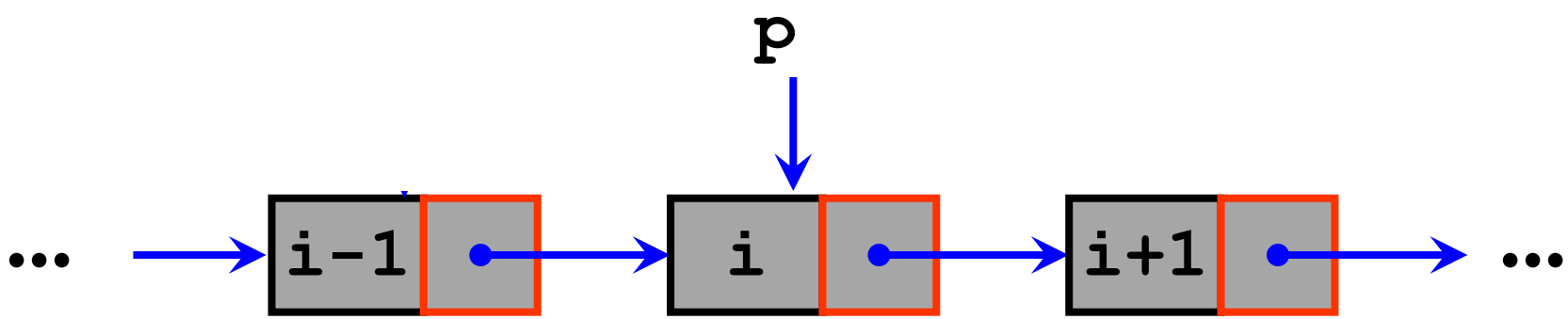
```
#define N 10
Node *AppCreate( )
{ Node *head = NULL, *tail = NULL;
  for(int i = 0; i < N; i++)
  {  Node *p = (Node *)malloc(sizeof(Node)); //创建新节点
     p -> data = i;          //给新节点的数据成员输入值
     p -> next = NULL;       //给新节点的指针成员赋值
     if(head == NULL)        //已有链表为空链表的情况
          head = p;
     else                    //已有链表不空的情况
          tail -> next = p;
     tail = p;
  }
  return head; }
```

整个链表的输出(链表的遍历)

等价于while(p != NULL)

① while(p)

③ p = p->next;



② printf("%d ", p->data);

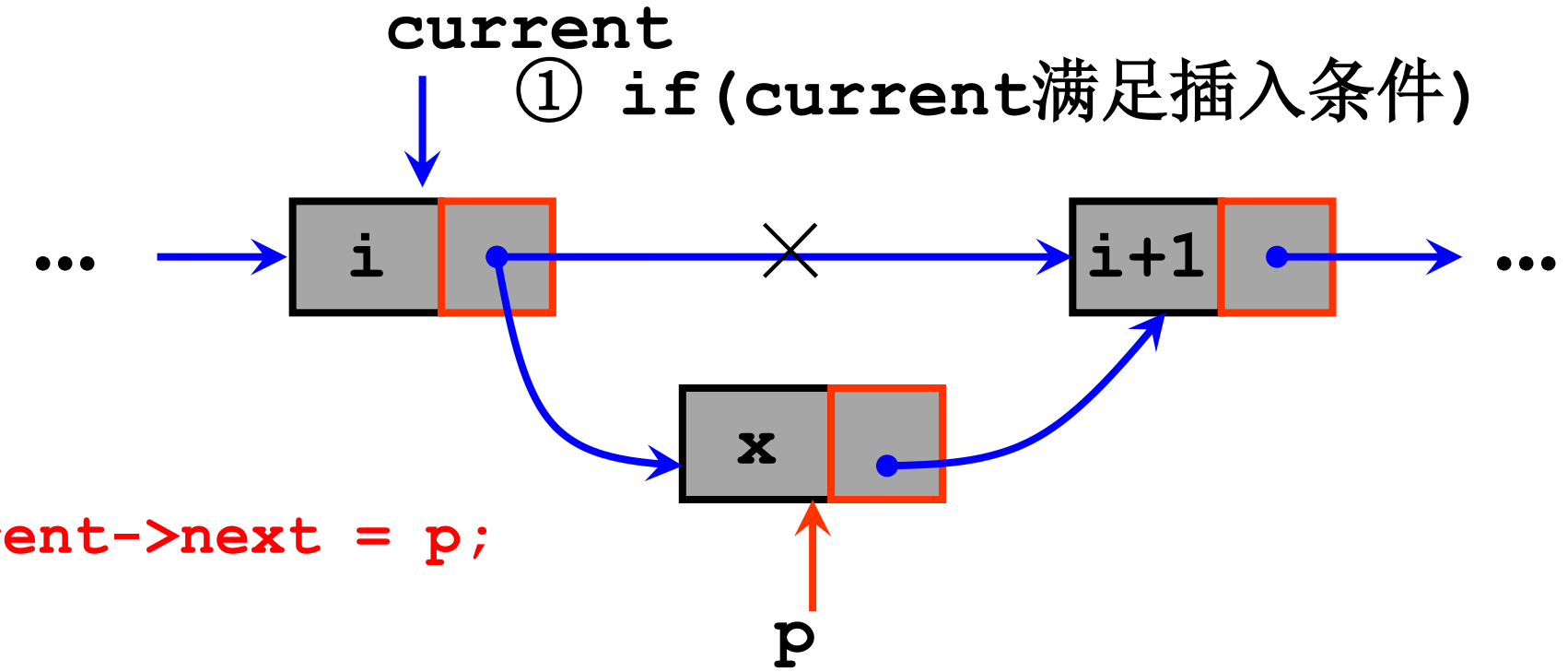
- 假设原链表首节点的地址存于head中，则输出整个链表的实现方式为：

```
void PrintList(Node *head)
{ if(!head)     //如果是空链表
     printf("List is empty. \n");
  else
  {
     while(head)     //遍历链表，等价于while(head != NULL)
     {    printf("%d, ", head -> data);
          head = head -> next;
     }
     printf("\n");
  }
}
```

如果写成 while(head -> next)

链表中插入节点

- 链表中的各个节点在物理上并非存储于连续的内存空间，所以在链表中插入或删除一个节点不会引起其它节点的移动。下面假设原链表首节点的地址存于head中，在第i（i>0）个节点后插入一个节点。



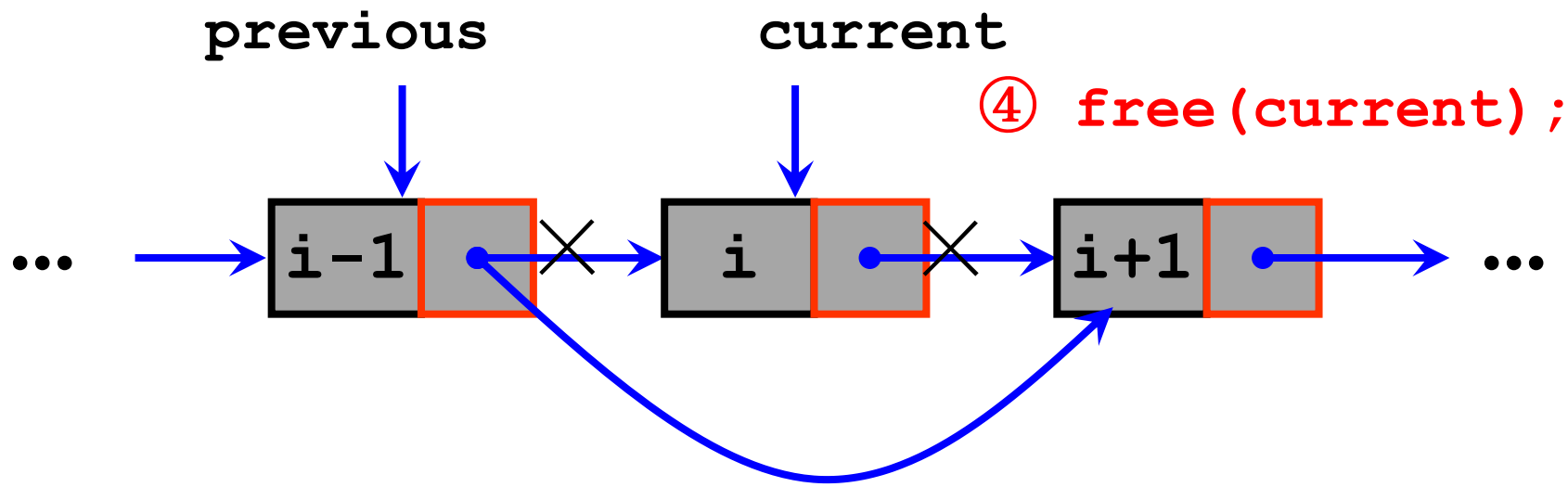
```
void InsertNode(Node *head, int i)
{
  Node *current = head;  // current指向第一个节点
  int j = 1;
  while(j < i && current -> next != NULL)     //查找第i个节点
  {   current = current -> next;
      j++;
  } //循环结束时，current指向第i个节点或最后一个节点（节点数不够i时）

  …

}
```

```
void InsertNode(Node *head, int i)
{
  …

  if(j == i)    // current指向第i个节点
  {  Node *p = (Node *)malloc(sizeof(Node)); //创建新节点
     scanf("%d", &p -> data);
     p -> next = current ->next;
                 //让第i+1个节点链接在新节点之后
     current -> next = p; //让新节点链接在第i个节点之后
  }
  else     //链表中没有第i个节点
     printf("没有节点: %d \n", i);
}
```

删除第i（i>0）个节点

① if(previous->next满足删除条件)

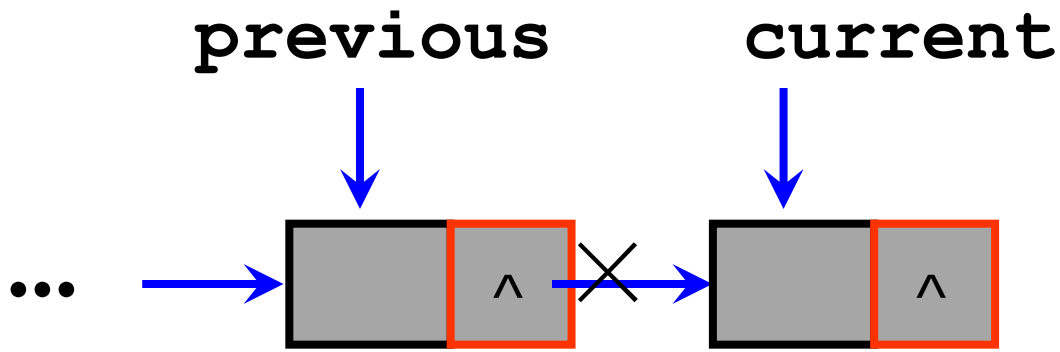
② current = previous->next;

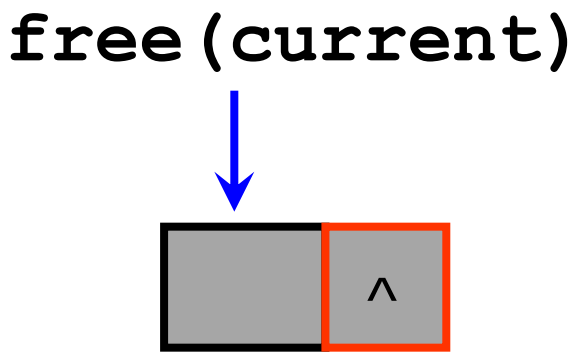


```
Node *DeleteNode(Node *head, int i)
{
  if(i == 1)    //删除头节点
  {   Node *current = head;     // current指向头节点
      head = head->next;  // head指向新的头节点
      free(current); //释放删除节点的空间
  }
  else
  {
      ……
  }
  return head;
}
```

```
else
{   Node *previous = head;    // previous指向头节点
    int j = 1;
    while(j < i-1 && previous -> next != NULL)
    {    previous = previous -> next;
         j++;
    }    //查找第i-1个节点
    if(previous -> next != NULL)  //链表中存在第i个节点
    {    Node *current = previous -> next;
                    // current指向第i个节点
         previous -> next = current -> next;
                    //让待删除节点的前后两个节点相链接
         free(current); //释放第i个节点的空间
    }
    else //链表中没有第i个节点
```

- 如果只考虑删除链表中的最后一个节点，则：





```
Node *DeleteLastNode(Node *head)
{
  Node *previous = NULL, *current = head;
  while(current -> next != NULL)
  {   previous = current;
      current = current -> next;
  }   //查找最后一个节点，current指向它，previous指向倒数第二个节点
  if(previous != NULL) //存在倒数第二个节点
      previous -> next = NULL; //让最后一个节点与倒数第二个节点断开
  else     //链表中只有一个节点
      head = NULL;    //让唯一的节点与头指针断开
  free(current);      //释放删除节点的空间
  return head;
}
```

- 假设原链表首节点的地址存于head中，则输出整个链表的实现方式为：

```
void PrintList(Node *head)
{ if(!head)     //如果是空链表
     printf("List is empty. \n");
  else
  {
     while(head)     //遍历链表，等价于while(head != NULL)
     {    printf("%d, ", head -> data);
          head = head -> next;
     }
     printf("\n");
  }
}
```

如果写成 while(head -> next) 则不能输出尾节点！！

整个链表的删除

- 链表中的每个节点都是动态变量，所以在链表处理完后，最好用程序释放整个链表所占空间，即删除链表。
- 假设原链表首节点的地址存于head中，则删除整个链表的程序为：

```
void DeleteList(Node *head)
{ while(head)
      //遍历链表，如果写成while(head -> next)则不能删除尾节点！！
  {   Node *current = head;
      head = head -> next;
      free(current);
  }
}
```

难点2

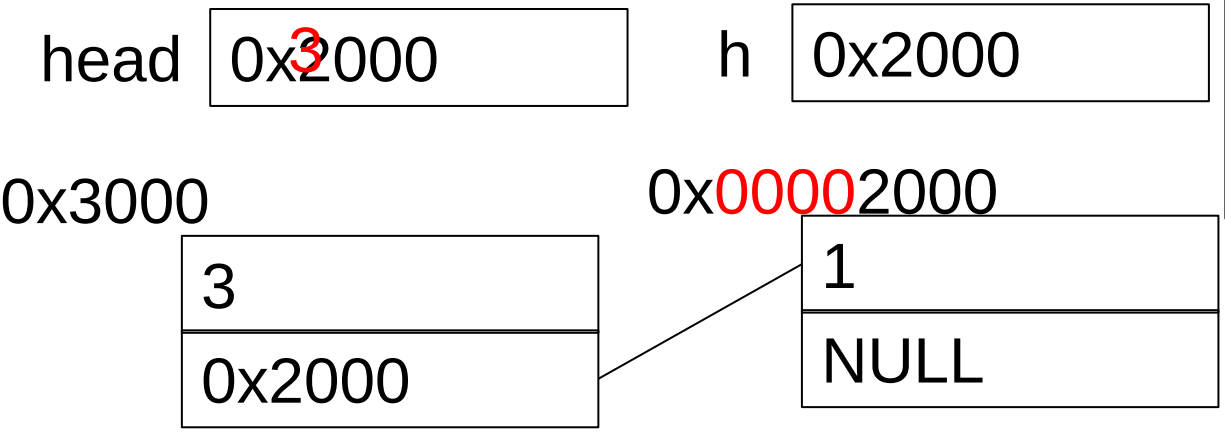
- 上述程序中的形参head与实参（即使变量名也是head）是不同的指针变量，形参的值有可能发生改变，所以要通过return语句返回给调用者。如果利用函数的副作用返回其值，则形参需定义成二级指针！！

参数为指针的传值调用

```
int main()
{
  Node *h = new Node;
  h-> data = 1;
  h-> next = NULL;
  InsOneNode(h);
  …

  return 0;
}
```

已有链表头部插入1个结点?

```
void
Node *InsOneNode(Node *head)
{
      Node *p = new Node;
      p -> data = 3;
      p -> next = head;//并未取值
      head = p;//并未取值

      return head;
}
```

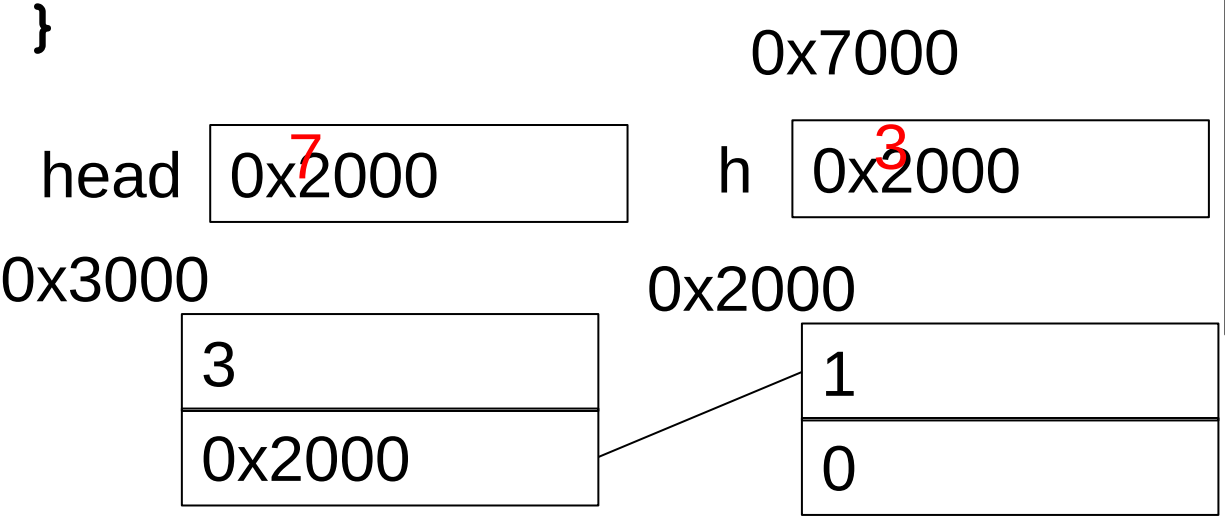


改为传址调用

```
int main()
{
  Node *h = new Node;
  h-> data = 1;
  h-> next = NULL;
  InsOneNode(&h);
  …

  return 0;
}
```

已有链表头部插入1个结点?

```
void
Node *InsOneNode(Node **head)
{
      Node *p = new Node;
      cin >> p -> data;
      p -> next = *head;
      *head = p;

      return head;
}
```



改为引用

```
int main()
{
  Node *h = new Node;
  h-> data = 1;
  h-> next = NULL;
  InsOneNode(h);
  …

  return 0;
}
```

已有链表头部插入1个结点?

void

```
Node *InsOneNode(Node *&head)
{
      Node *p = new Node;
      cin >> p -> data;
      p -> next = head;
      head = p;

      return head;
}
```

```
Node *InsertBeforeKeyNode(Node *h, int key)
{    Node *p = (Node *)malloc(sizeof(Node));
     scanf("%d", &p -> data);
     if(h != NULL)
     {    Node *current = h;
          Node *previous = NULL;
          while(current != NULL && current -> data != key )
          {    previous = current;
               current = current -> next;
          }
          if(current != NULL && previous != NULL)
          {    p -> next = current;
               previous -> next = p;
          }
          else if(current != NULL && previous == NULL)
          {    p -> next = current;
               h = p;
          }//头部插入
     }    return h; }
```

又比如，在某节点前插入新节点

顺序！
短路规则

```
void InsertBeforeKeyNode(Node **h, int key)
{    Node *p = (Node *)malloc(sizeof(Node));
     scanf("%d", &p -> data);
     if(h != NULL)
     {    Node *current = *h;
          Node *previous = NULL;
          while(current != NULL && current -> data != key )
          {    previous = current;
               current = current ->next;
          }
          if(current != NULL&& previous != NULL)
          {    p -> next = current;
               previous -> next = p;
          }
          else if(current != NULL && previous == NULL)
          {    p -> next = current;
              *h = p;
          }
     }
```

又比如，在某节点前插入新节点

再比如

```
Node *DeleteNode(Node *head) //删除头节点
{     Node *current = head;     // current指向头节点
      head = head->next;  // head指向新的头节点
      free(current); //释放删除节点的空间
      return head;
}

void DeleteNode(Node **head) //删除头节点
{     Node *current = *head;    // current指向头节点
     *head = *head->next; // *head指向新的头节点
      free(current); //释放删除节点的空间
      //return head;
}
```

链表操作中需要注意的几个问题

- 注意考虑几个特殊情况下的操作
  - 链表为空表 (head==NULL)
  - 链表只有一个节点
  - 对链表的第一个节点进行操作
  - 对链表的最后一个节点进行操作
  - 最后一个节点的next指针应为NULL
  - 操控链表的指针是否已经指向了链表末尾

基于链表的排序

- 基于链表的排序一般会涉及两个节点数据成员的比较和交换操作，以及节点的插入等操作。

例 用链表实现N个数的插入法排序。

…

```
#define N 10
struct Node
{ int data;
  Node *next;
};
```

```
extern Node *AppCreate( );
extern Node *ListSort(Node *head);
extern void PrintList(Node *head);
extern void DeleteList(Node *head);
```

```
int main( )
{
      Node *head = AppCreate( );      //建立链表，程序略
      PrintList(head);                //输出链表，程序略
      PrintList(ListSort(head));      //输出排序之后的链表
      DeleteList(head);               //删除链表，程序略
      return 0;
}
```

```
Node *ListSortInsert(Node *head, Node *p); // 插入一个节点
Node *ListSort(Node *head) //插入法排序函数
{  if(head == NULL)
        return NULL;
   if(head -> next == NULL)
        return head;
   Node *cur = head -> next;//指向第二个节点
   head -> next = NULL; //将头节点脱离下来，作为已排序队列
   while(cur)//将后面的节点依次插入已排序队列
   {    Node *prev = cur;
        cur = cur -> next;
        head = ListSortInsert(head, prev);
   }
   return head;          }
```

```
Node *ListSortInsert(Node *head, Node *p)//插入一个节点
{  if(p -> data < head -> data)
   {     p -> next = head;
         head = p;
         return head;
   }//插入头部
   Node *cur = head;
   Node *prev; //用cur和prev操纵已排序队
   while(cur)
   {     if(p -> data < cur -> data)
               break;
         prev = cur;
         cur = cur -> next;
   }//查找合适的位置，在prev后插入
         p -> next = prev -> next;
         prev -> next = p;
         return head;
   }
```

基于链表的信息检索

- 一般不适合用折半查找法。

例 基于链表的顺序查找程序。

…

```
struct NodeStu
{ int id; //学号
  float score; //成绩
  NodeStu *next;
};
```

```
extern NodeStu *AppCreate( );
extern float ListSearch(NodeStu *head);
extern void DeleteList(NodeStu *head);
```

```
int main( )
{ NodeStu *head = AppCreateStu( ); //建立链表
  int x = 181220999;
  float y = ListSearch(head, x);
                  //在链表中查找指定id对应的score值
  if(y < 0)
      printf("没有找到! \n");
  else
      printf("%s同学的成绩为: %f \n", x, y);

  return 0;
}
```

```
float ListSearch(NodeStu *head, int x)
{
  NodeStu *p;
  for(p = head; p != NULL; p = p->next)
     if(p -> id == x)     //遍历链表，查找id为x的节点
        break;
  if(p != NULL)       // if(p) 找到了
     return p -> score;
  else
     return -1.0;
} //p没有在for语句里定义，因为…
```

for(p = head; p && p -> id != x ; p = p->next) ;

结构类型数组

- **结构数组可用于表示二维表格。比如，名表：**

```
Stu stu_array[5]; //定义了一个一维结构数组
Stu stu_array[5] = { {1001, 'T', 'M', 20, 90.0}, …,
                     {1005, 'L', 'F', 81.0} }; //初始化
```

```
#define N 5
enum FeMale {F, M};
struct Stu
{   int id;
    char name;
    FeMale s;
    int age;
    float score;
};
```

stu_array[5]

结构变量

一维数组

	num	name	s	age	score
s[0]	1001	T	M	20	90.0
s[1]	1002	K	F	19	89.0
s[2]	1003	M	M	19	95.5
s[3]	1004	J	M	18	100.0
s[4]	1005	L	F	18	81.0

例 基于结构数组的数据顺序查找。

```
float Search(Stu stu_array[], int count, int id)
{   int i;
    for(i = 0; i < count; i++)
    {    if(id == stu_array[i].id)
             break;
    }
    if(i >= count)
         return -1.0;
    else
         return stu_array[i].score;
}
```

```
#define N 5
enum FeMale {F, M};
struct Stu
{
    int id;
    char name;
    FeMale s;
    int age;
    float score;
};
```

```
for(i = 0; i < count; i++)
{   if(id == stu_array[i].id)
        return stu_array[i].score;
}
    return -1.0;
```

- **例 基于结构数组的数据折半查找。** 先按某一列排序

```
void  BublSort(Stu stu_array[ ], int count)
{
    int i, j;
    for(i = 0; i < count-1; i++)
        for(j = 0; j < count-1-i; j++)
            if(stu_array[j].id > stu_array[j+1].id)
            {
                Stu temp;
                temp = stu_array[j];
                stu_array[j] = stu_array[j+1];
                stu_array[j+1] = temp;
            }
}
```

```
float BiSearchR(Stu stu_array[], int first, int last, int id)
{
    if(first > last)
        return -1.0;
    int mid = (first + last) / 2;
    if(id == stu_array[mid].id)
        return stu_array[mid].score;
    else if(id > stu_array[mid].id)
        return BiSearchR(stu_array, mid + 1, last, id);
    else
        return BiSearchR(stu_array, first, mid - 1, id);
}
```

- **结构类型数组也可以用指针来操纵，操纵方法和用指针操纵其他基本类型数组的方法类似。比如，**

```
Student stu[10], *psa;
psa = stu;
```

联合（union）类型

```
union myType
{
  int i;
  char c;
  double d;
};
```

与结构类型类似，
联合类型由程序员构造而成，
构造时需要用到关键字union。

联合变量的初始化、成员的操作方式
也与结构变量类似。

```
myType v;     //定义了一个myType类型的联合变量v
```

- 与结构变量不同的是，系统采用覆盖技术按需要占用内存单元最多的成员为联合变量分配内存

```
struct B
{   int i;
    char c;
    double d;
};
B b;
```


```
union A
{   int i;
    char c;
    double d;
};
A v;
```

- **对于上述联合变量v，在程序中可以分时操作其中不同数据类型的成员。比如，**

```
v.i = 12;                    //以下只操作变量v的成员i
……
v.c = 'X';                   //以下只操作变量v的成员c
……
v.d = 12.95; //以下只操作变量v的成员d
……
```

- **当给一个联合变量的某成员赋值后，再访问该变量的另外一个成员，将得不到原来的值。比如，**

```
v.i = 12;
printf("%f", v.d);                    //不会输出12
```

- **即可以分时把v当作不同类型的变量来使用，但不可以同时把v当作不同类型的变量来使用。**

- **联合类型使程序呈现出某种程度的多态性。这种多态性的好处是：在提高程序灵活性的同时，可以实现多种数据共享内存空间。比如，**

```
union Array
{
    int int_a[100];
    double dbl_a[100];
};
Array buffer;
… buffer.int_a …    //使用数组int_a，只有一半存储空间闲置
……
… buffer.dbl_a …    //使用数组dbl_a，没有存储空间闲置
```

如果不使用联合类型，例如，

```
int int_a[100];
double dbl_a[100];
… int_a …     //使用数组int_a（dbl_a所占的存储空间闲置）
……
… dbl_a …     //使用数组dbl_a（int_a所占的存储空间闲置）
……
```

- **如果是联合类型的数组，则其每一个元素的类型可以不同。比如，**

```
enum Grade{UNDERGRAD, MASTER, PHD, FACULTY};     //职级
union Performanc
{  int nPaper;      //已发表论文篇数
   double gpa;      //GPA
}; //业绩类型
struct Person
{  char id[20];
   char name[20];
   enum Grade grd;
   union Performanc pfmc;//每个人业绩属性的类型可以不同
};
```

```
const int N = 800;                    //人员的个数
void input(Person prsn[ ], int num);
int main( )
{  Person prsn[N] = {{0,0,0,0}};
   input(prsn, N);
   double maxgpa = 0.1;
   int maxMaster = 0;
   int maxPhd = 0;
   int maxFaculty = 0;
```

```
for(int i = 0; i < N; i++)
{       if(prsn[i].grd == UNDERGRAD
                && prsn[i].pfmc.gpa > maxgpa)
            maxgpa = prsn[i].pfmc.gpa;
        if( prsn[i].grd == MASTER
                && prsn[i].pfmc.nPaper > maxMaster)
            maxMaster = prsn[i].pfmc.nPaper;
        if( prsn[i].grd == PHD
                && prsn[i].pfmc.nPaper > maxPhd)
            maxPhd = prsn[i].pfmc.nPaper;
        if( prsn[i].grd == FACULTY
                && prsn[i].pfmc.nPaper > maxFaculty)
            maxFaculty = prsn[i].pfmc.nPaper;
}
```

```
    for(int i = 0; i < N; i++)
        if(prsn[i].grd == UNDERGRAD
                && prsn[i].pfmc.gpa == maxgpa)
                printf( "本科生获奖者: %s \n",  prsn[i].name);
    //...
    return 0;
}
```

```
void input(Person prsn[ ], int num)
{  int g = 0;
   for(int i = 0; i < num; i++)
   {    printf( "输入人员的编号与姓名: \n");
        scanf("%s%s", prsn[i].id, prsn[i].name);
        printf( "输入0~3分别代表本科、硕士、博士生和教师: ");
        scanf("%d", &g);
        switch(g)
        {    case 0: prsn[i].grd = UNDERGRAD;
                     printf( "输入学分绩: ");
                     scanf("%lf", &prsn[i].pfmc.gpa);
                     break;
             case 1: prsn[i].grd = MASTER;
                     printf( "输入已发表论文篇数: ");
                     scanf("%d", &prsn[i].pfmc.nPaper);
                     break;          …… ……
```

小结

- 结构是一种派生数据类型，用来描述多个不同类型的数据群体
- 基于结构类型的数据结构——链表
- 基于链表的排序和检索算法的程序实现方法
- 联合是一种与结构类似的派生数据类型，与结构类型的不同点，仅在于存储方式（系统对联合类型的成员采用了覆盖存储技术），可以在实现多态性程序的同时节约内存空间，程序的多态性可以提高程序的可读性。

- 要求：
  - 掌握结构变量的定义、初始化和操作方法
  - 掌握链表的特征及其创建、删除、插入节点、删除节点等方法
    - 一个程序代码量≈200行
  - 了解联合类型的特点和应用
  - 继续保持良好的编程习惯

Thanks !