Struct, Union, Enum
Linked List

王慧妍

why@nju.edu.cn

南京大学 软件学院 计算机软件研究所

struct, union, enum

struct

- 结构变量
  - 可能具有不同类型的值（成员）的集合

```
struct {
    int id;
    char name[N];
    double score;
}stu1, stu2;
```

  - struct.c
  - 初始化
    - {1, “Allen”, 98.5};
    - {.name = “Su”, .score = 88.0, .id = 2};
    - {.name = “Su”, 88.0, .id = 2};
    - {.name = “Su”, .score = 88.0};
    - 除初始化外不允许整体赋值

struct成员

- 结构体成员作用域仅在当前结构体，具有独立的name space
  - 多结构体成员重名，互不冲突

- 成员访问通过“.”操作访问
  - `stu1.name`
  - `stu2.score`
  - `&stu1.name`

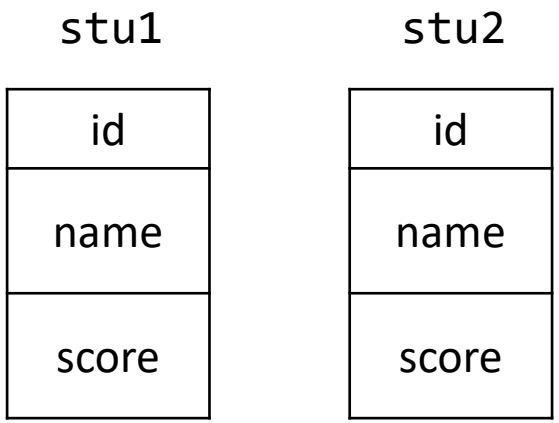


表 2-1　C 语言运算符优先级表（由上至下，优先级依次递减）

运 算 符	结 合 性
() [] -> .	自左向右
! ~ ++ -- - (type) * & sizeof	自右向左
* / %	自左向右
+ -	自左向右
<< >>	自左向右
< <= > >=	自左向右
== !=	自左向右
&	自左向右

结构类型的名字

- 结构标记

```
struct record {
    int id;
    char name[N];
    double score;
};
struct record stu1, stu2;
```

- typedef类型定义

```
typedef struct {
            int num;
            long score;
            char id;
        } record2;
record2 stu2
```

结构类型作为参数和返回值

- rectangle.c
- `void Print(struct record stu);`
  - 值传递：结构体会复制

> “If a larger structure is to be passed to a function, it is generally more efficient to pass a pointer than to copy the whole structure.”
>
> —— K & R (p.131)

  - 区分结构体名称和数组名称

结构体的操作

- 结构体变量赋值
  - `struct record stu1, stu2;`
  - `stu1 = stu2;`
  - 结构变量的名字不是一个地址
- 兼容的结构体变量可以相互赋值
  - 惊喜：内部数组也被赋值了
- 不同的结构体变量不能==或!=比较

事情开始变得有趣了

- 结构类型给我们定义一种C语言能够理解的自定义的类型
- struct可以嵌套
  - 匿名结构：有成员无标记，未命名的结构类型×✓

```
struct tag
{
    int i;
    struct m{
        int j, k;
    };
    struct
    {
        char c;
        float f;
    };
    struct
    {
        double d;
    }s;
} t;
```

```
t.i = 2000;     ✓
t.j = 5;        ×
t.k = 6;        ×
t.c = 'A';      ✓
t.f = 0.2;      ✓
t.s.d = 20.2;   ✓
```

嵌套的结构体和数组

- StuSort.c

```
struct Name
{
    char firstname[10];
    char midname[10];
    char lastname[10];
};
```

```
struct Stu
{
    Struct Name name;
    char sex;
    int age;
};
```

```
struct Stu stus[10];
```

struct内存对齐原则

- 每个结构体的成员:
  - 第一个成员都位于偏移为0的位置
  - 以后每个数据成员的偏移量min(#pragma pack()指定的数，这个数据成员的自身长度) 的倍数**
    - 一般情况下不指定#pragma pack()默认长度32位系统是4字节，64位系统是8字节

- 在数据完成各自对齐之后，结构体本身也要对齐
  - 结构体的大小要和该结构体内最大元素大小的倍数，不够补齐

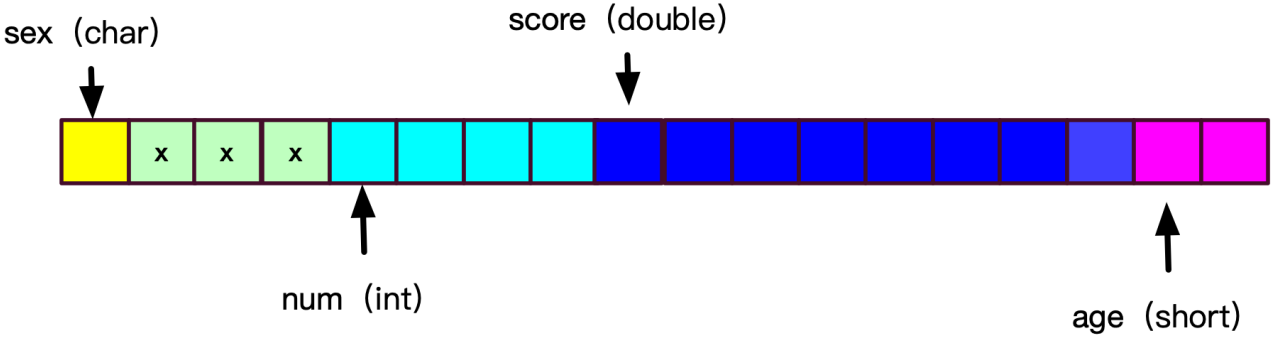
- 结构体内套用结构体
  - 遵循两个原则，首先第一点就是要将嵌套结构体内要内存对齐，然后就是嵌套结构体的起始位的偏移量必须是嵌套结构体内的占用最大内存属性的倍数就好，其他都一样

struct内存对齐原则

在没有指定#pragma pack()情况下

char只占用第一个字节所以
直接存到偏移为0的位置

同样的偏移量要是自身的倍数，但是
偏移量刚好是自身的倍数所以不用补齐



因为偏移量要是自身长度的倍数
所以要冲第五号位置开始存储
（int前面三个位置就是需要补齐的内存），
然后int自身长度有4字节

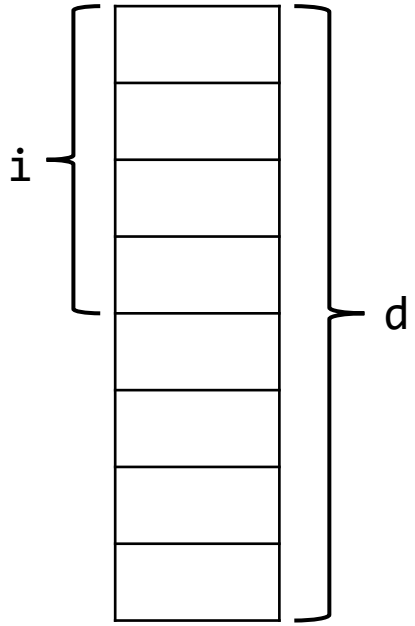
同样的偏移量要是自身的倍数，但是
偏移量刚好是自身的倍数所以不用补齐

最后还有一条规则是：结构的的长度要是结构体内最大元素尺寸的倍数，不难看出最大的是double 8字节，而我们成员对齐之后发现长度只有18字节，所以要补齐再补6个字节结果就是24字节刚好是8的倍数

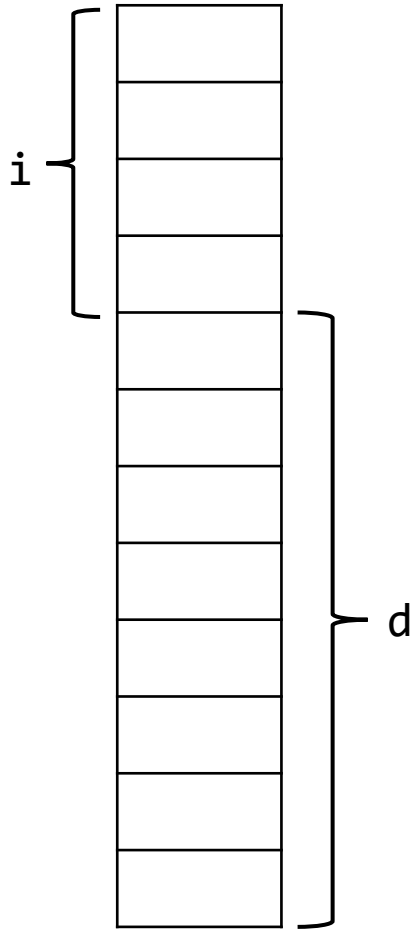
union

- 和struct的不同点：
  - 成员是否共用同样的内存空间
  - struct-union.c

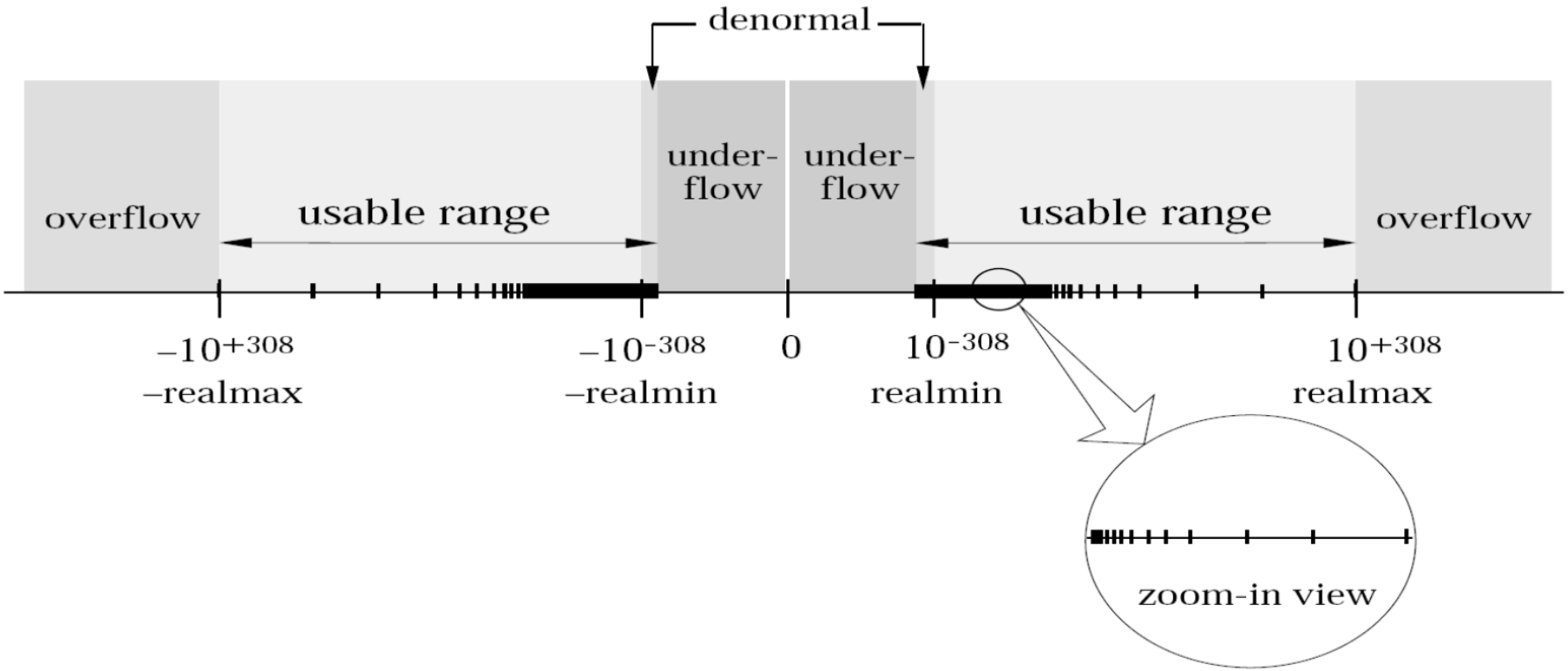
```
union{int i; double d};
```

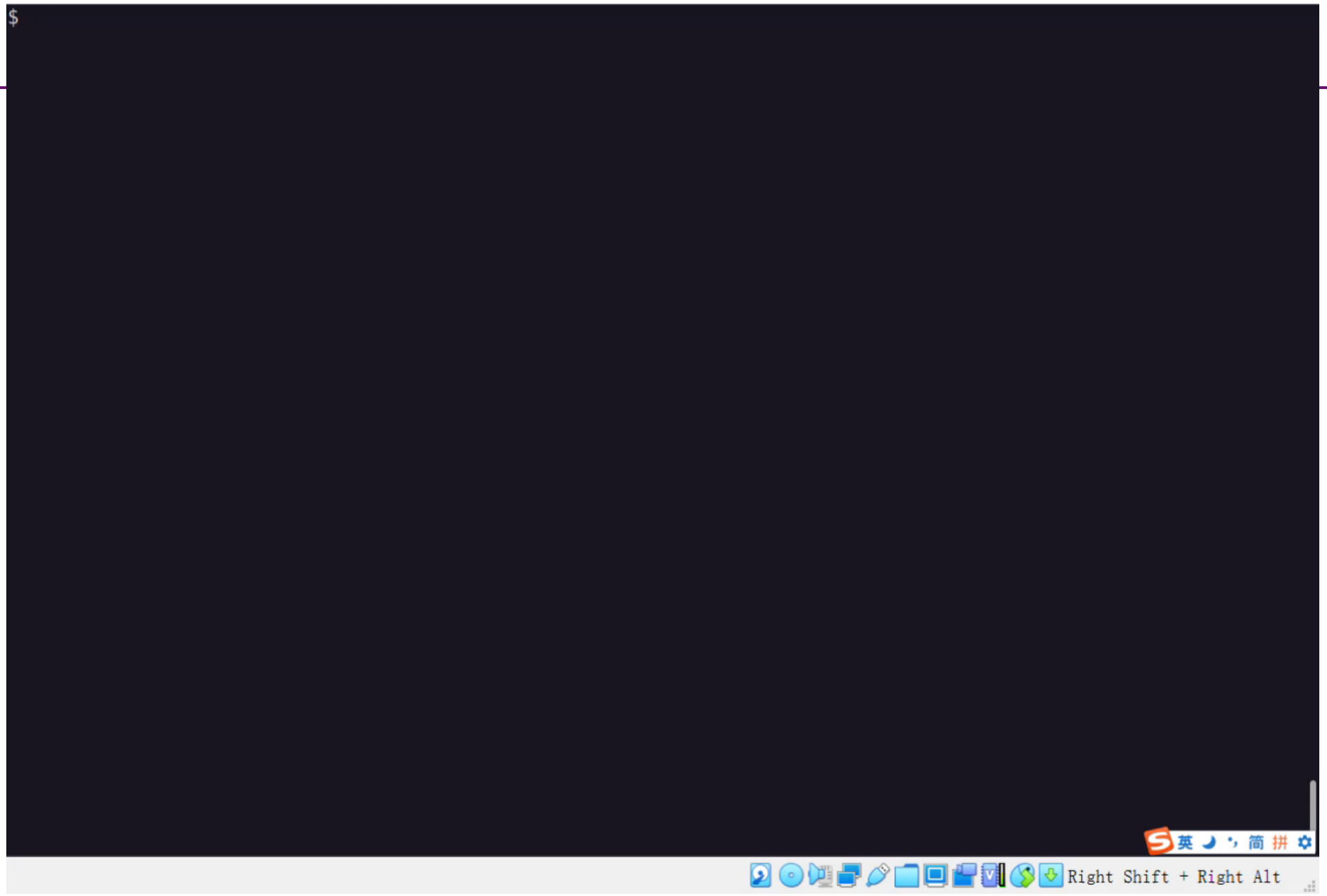



```
struct{int i; double d};
```



Floating Point Number Line




管理 控制 视图 热键 设备 帮助
$
Right Shift + Right Alt

Bit field

- 位域
  - 用一个字节中的不同二进制位来表达不同信息
  - 变量名：位数
  - Bitfield.c

```
typedef union inst {
   struct { u8 rs : 2, rt: 2,  op: 4; } rtype;
   struct { u8 addr: 4,        op: 4; } mtype;
} inst_t;
```

```
1  struct tcphdr {
2      __be16    source;
3      __be16    dest;
4      __be32    seq;
5      __be32    ack_seq;
6  #if defined(__LITTLE_ENDIAN_BITFIELD)
7      __u16    res1:4,
8          doff:4,
9          fin:1,
10         syn:1,
11         rst:1,
12         psh:1,
13         ack:1,
14         urg:1,
15         ece:1,
16         cwr:1;
17 #elif defined(__BIG_ENDIAN_BITFIELD)
26         syn:1,
27         fin:1;
28 #else
29 #error    "Adjust your <asm/byteorder.h> defines"
30 #endif
31     __be16    window;
32     __sum16    check;
33     __be16    urg_ptr;
34 };
```

enum

- 枚举类型
  - 由程序员列出的，具有常量的值(起始默认为0)
  - 常量符号化
  - `enum {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday};`
  - `typedef enum {False, True} Bool;`
  - `enum {RED=20, YELLOW, GREE=5};`
  - `enum {RED, YELLOW, GREEN, NumOfColors};`
  - enum.c
  - month-enum.c

```
struct Number{
    enum{INT_TYPE, DOUBLE_TYPE} kind;
    union{
        int i;
        double d;
    }num;
};
```

struct和指针

- 一些有趣的事情
  - struct-p-func.c

```
struct {
    int a;
    char b;
    float c;
} x[10], *p;
```

```
struct {
        char *name;
        char *des;
        int (*handler)(char *);
    } cmd_table[] = {
        {"i", "Print info", cmd_i},
        {"q", "Exit", cmd_q},
        {"c", "Continue", cmd_c}
    };
```

结构体的内存示意图

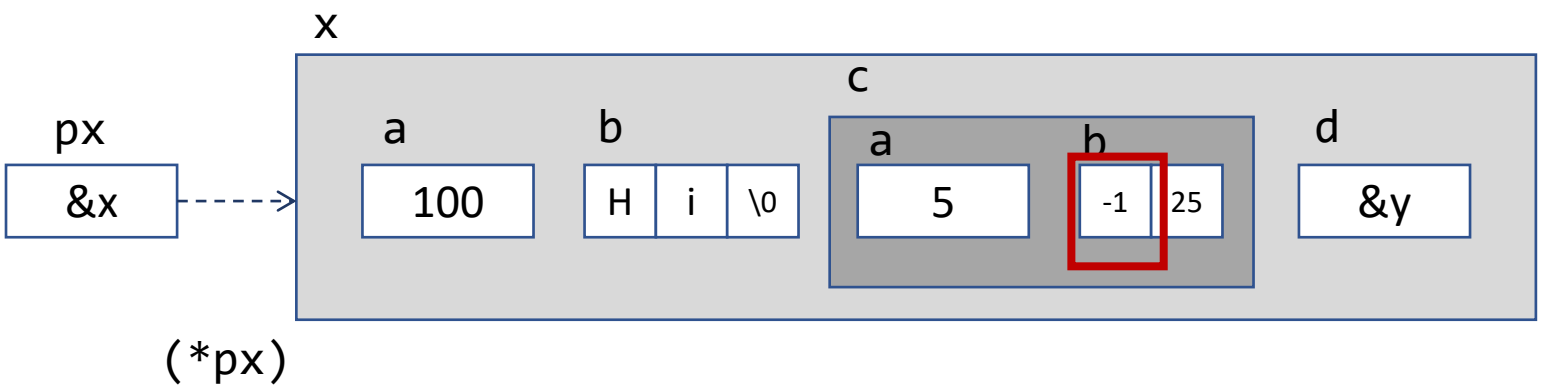
C Operator Precedence - cppreference.com

```
typedef struct {
    int a;
    short b[2];
} EX2;
```

```
typedef struct EX{
    int a;
    char b[3];
    EX2 c;
    struct EX *d;
} EX;
```

```
EX x;
EX *px = &x;
```

```
*px->c.b
```



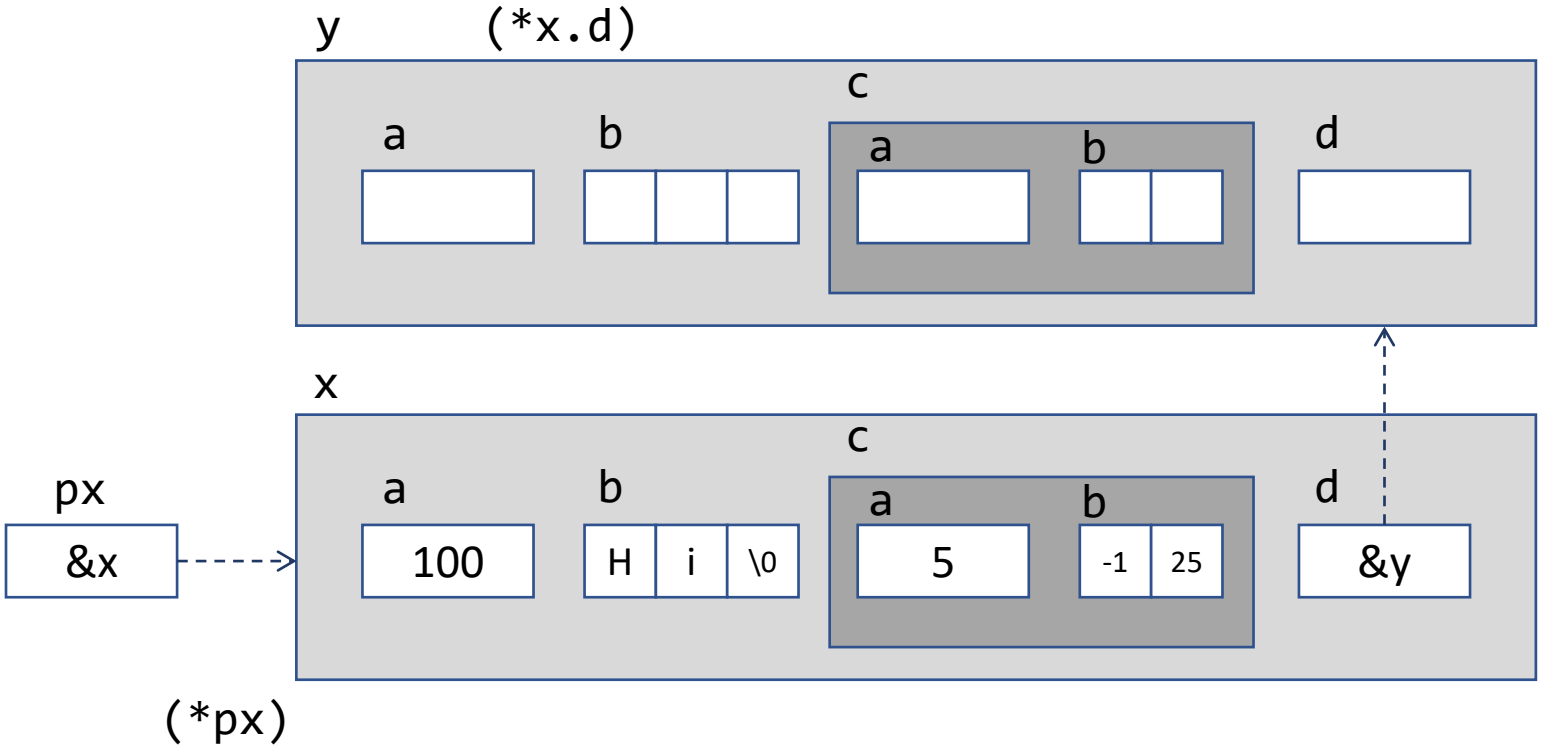

```
EX x = {100, {'H', 'i', '\0'}, {5, {-1, 25}}, 0};
```

```c
typedef struct {
    int a;
    short b[2];
} EX2;
```

```c
typedef struct EX{
    int a;
    char b[3];
    EX2 c;
    struct EX *d;
} EX;
```

```c
EX x;
EX *px = &x;
```

```c
EX y;
x.d = &y;
```




```c
EX x = {100, {'H', 'i', '\0'}, {5, {-1, 25}}, 0};
```

struct自引用

- 不同自引用的例子（分别合法么？）
  - 什么含义?

```
struct SELF{
    int a;
    struct SELF b;
    float c;
};
```

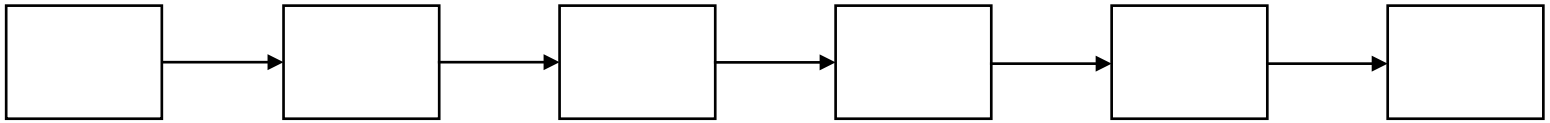
```
struct SELF{
    int a;
    struct SELF *b;
    float c;
};
```

```
typedef struct {
    int a;
    SELF *b;
    float c;
} SELF;
```

```
typedef struct SELF_TAG {
    int a;
    struct SELF_TAG *b;
    float c;
} SELF;
```

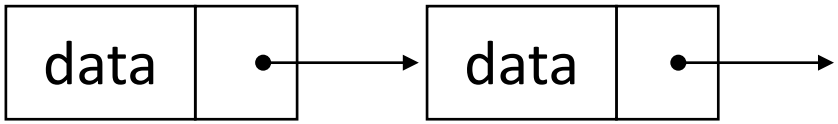
Linked List

- 链表：内存非连续的线性结构



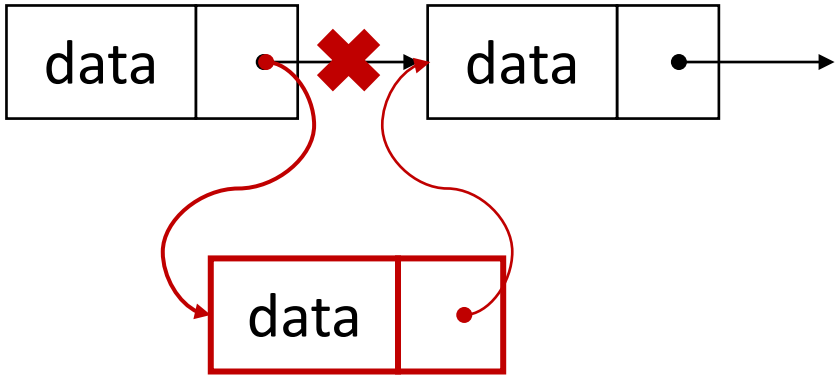
- 链表节点
  - 存储有价值的数据
  - 指向下一个链表节点的指针

  - `struct node {int data; struct node *next;};`

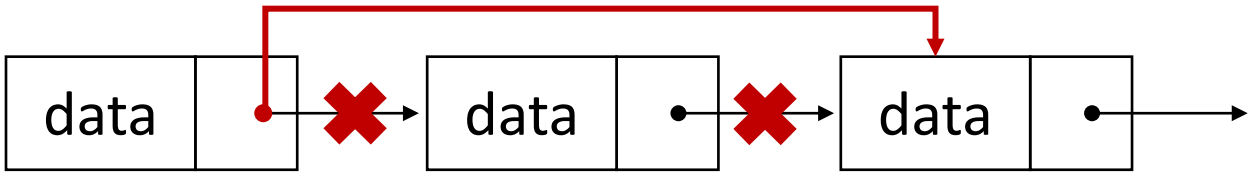


链表的节点插入和删除

- 新链表节点的插入



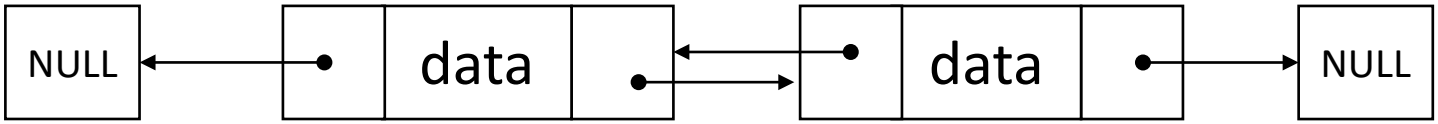
- 已有链表节点的删除



单向链表和双向链表

- 单向链表
  - `struct node {int data; struct node *next;};`

- 双向链表
  -
```
struct node {
  struct node * prev;
  int data;
  struct node *next;
};
```

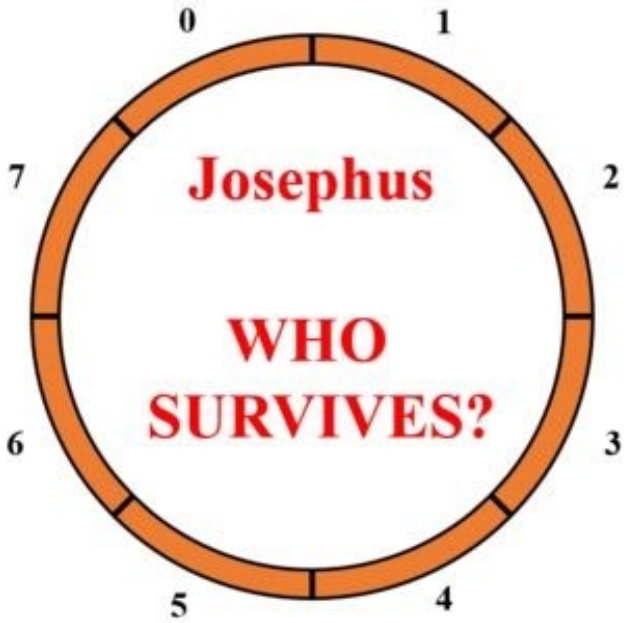


循环链表

- 与单向链表相比，区别在于
  - 尾节点的指针成员不再指向NULL，而是指向首节点

约瑟夫问题

- 由来
  - 在罗马人占领乔塔帕特后，39 个犹太人与Josephus 及他的朋友躲到一个洞中，39个犹太人决定宁愿死 也不要被敌人抓到，于是决定了一个自杀方式。41个人排成一个圆圈，由第1个人开始报数，每报数 到第3人该人就必须自杀，然后再由下一个重新报数，直到所有人都自杀身亡为止。Josephus要他的朋友先假装遵从，他将朋友与自己安排在第16个与第31个位置，于是逃过了这场死亡游戏 。



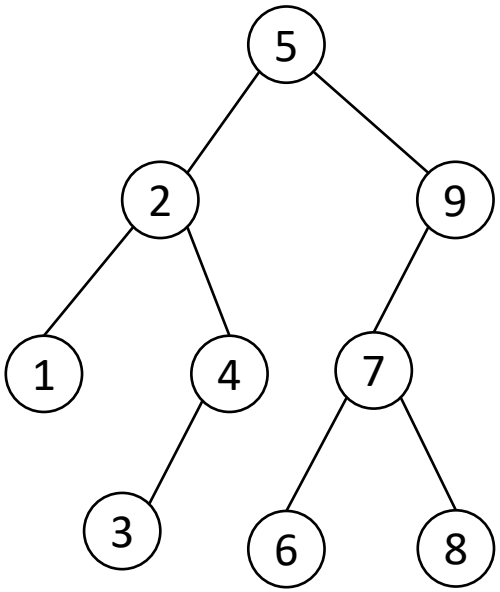
约瑟夫问题

- 用数组实现约瑟夫环?

- 试试用循环链表来实现约瑟夫环?
  - Joseph.c

更复杂点场景：链式结构实现二叉树

- 二叉搜索树(binary search tree)
  - 每个节点有其左孩子和右孩子
  - 额外属性：左孩子比当前节点值小，右孩子比当前节点值大

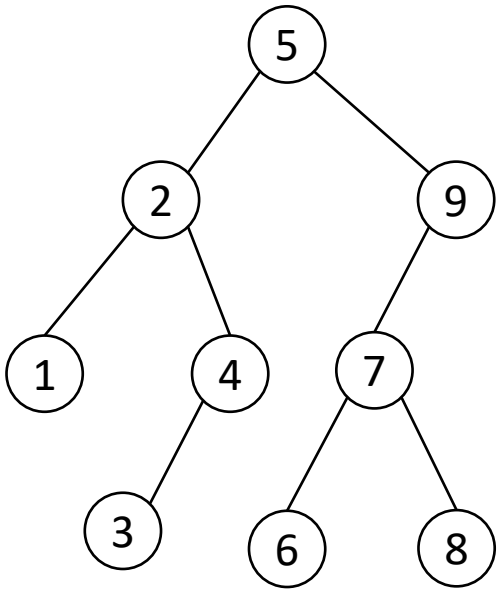



```
struct BNode {
    int value;
    struct BNode *left;
    struct BNode *right;
} *root;
```

End

更复杂点场景：链式结构实现二叉树

- 二叉树的遍历
  - 前序pre-order
    - 521439768
  - 中序in-order
    - 123456789
  - 后序post-order
    - 134268795



- BST.c

各种树

- 二叉树BST
- 平衡二叉树AVL
- 红黑树
- B树，B+树