中山大學
SUN YAT-SEN UNIVERSITY

指针与结构体

中山大学计算机学院

讲课人：潘茂林

目录

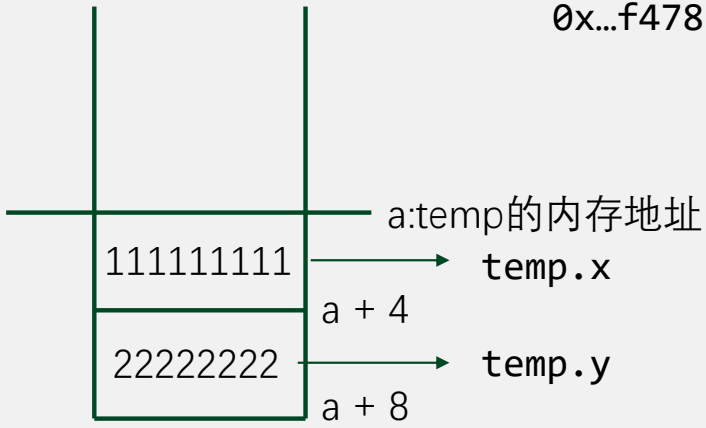
CONTENTS

结构体内存布局

在内存中，结构体**实例变量**分配在一段**连续的内存空间**内，每个成员变量依次被分配在后续的内存位置上。例如: `struct Rational {int x,y;}`

```
struct Rational temp =
{ 0x11111111, 0x22222222 };
```

```
(lldb) x/10xw &temp
0x…f458: 0x11111111 0x22222222 0xefbff498 0x00007ffe
0x…f468: 0x00000001 0x00000000 0xefbff488 0x00007ffe
0x…f478: 0x20640f5d 0x00007fff
```



字宽（word width）与内存中数据

32位计算机指 CPU 加法指令一次完成 32 位数加法。字宽指 CPU 一次从访问内存读或写的位数，32位字宽表示一次读或写4个字节。=32位

对于32位字宽的计算机，每次访问内存的地址总是 4 的倍数。如果将整数放置在非 4 倍的地址时，如右图读 0x55443322，则要么编译器用两条指令完成该整数读写并拼接为一个整数，要么硬件用两个以上微指令周期完成该整数读写。导致CPU处理数据性能大幅下降。

地址	值（byte）
0x…0000	00
	11
	22
	33
0x…0004	44
	55
	66

解决办法就是**对齐**(*alignment*)。例如大于字宽的基本类型数据都按字宽边界对齐，即使浪费一些内存也值得。因此，程序中整数变量地址一般是 4 的倍数。

结构体对齐 - 案例

最大对齐值 默认N=4

```
/*struct layout*/
#pragma pack(1)

typedef struct {
    char a[3];
    int b;
    short c;
} packed_struct;

#pragma pack(4)

typedef struct {
    char a[3];
    int b;
    short c;
} unpacked_struct;
```

```
sizeof packed_struct 9
sizeof unpacked_struct 12
```

为什么产生上述执行结果?

1. 预编译指令 #pragma pack(N) 控制后继定义的类和联合体的**最大对齐值**。字宽 32 默认 $N = 4$。$N \in \{1, 2, 4, 8, 16, 32\}$
2. 基本数据类型成员变量**对齐值**规则:
   - 数据类型所占字节数 >=N，则地址是 N 的倍数。
   - < N 则对齐值是数据类型所占字节数的倍数。例如
     - char 类型对齐值是1，开始地址是1的倍数；
     - short 开始地址是2的倍数;
     - int 开始地址是4的倍数
3. struct 类型对齐值：$\min$(成员中对齐值最大值，N)。除了**地址**是对齐值的倍数，其 size 也必须是对齐值的倍数。

结构体对齐 - 内存案例

```
/*struct layout*/
#pragma pack(1)

typedef struct {
    char a[3];
    int b;
    short c;
} packed_struct;

#pragma pack(4)

typedef struct {
    char a[3];
    int b;
    short c;
} unpacked_struct;
```

地址	值（byte）
a	00
a+1	11
a+2	22
&b	33
	44
	55
	66
&c	77
	88

N=1
struct 对齐值 1
struct size 9

地址	值（byte）

N=2
struct 对齐值 ? 2
struct size ? 10

地址	值（byte）
a	00
a+1	11
a+2	22
	--
&b	33
	44
	55
	66
&c	77
	88
	--
	--

N=4
struct 对齐值 4
struct size 12

共用体对齐 - 内存案例

```
/*struct layout*/
#pragma pack(1)

typedef union {
    char a[3];
    int b;
    short c;
} packed_union;

#pragma pack(4)

typedef union {
    char a[3];
    int b;
    short c;
} unpacked_union;
```

地址			值（byte）		
a	&b	&c	00	33	77
a+1			11	44	88
a+2			22	55	--
			--	66	--

N=1
union 对齐值 1
union size 4

地址			值（byte）		
a	&b	&c	00	33	77
a+1			11	44	88
a+2			22	55	--
			--	66	--

N=4
union 对齐值 4
union size 4

位域定义

位域可以使得结构体支持方便的按位访问

```
struct A {
    unsigned int t: 2;
};

struct A a = { 2 };
a.t = 3;
a.t = 4; // warning: Implicit truncation from 'int' to bit-field
         // changes value from 4 to 0
```

声明含义是，t的位宽为 2 的无符号整数
这里, 0 <= 位宽 <= 整数的位数

位域类型可以是整数，无符号整数，_Bool

位域 - 内存布局

如果几个位域被顺序写在一起，那么他们会被打包拼合在一起。

位域对齐值按整数对齐值，及第一个位域必须按整数对齐。一个包最多是一个整数

```
struct B {
    unsigned t1: 10;
    unsigned t2: 11;
    unsigned t3: 11;
};
```

```
printf("%lu\n", sizeof(struct B));

>>> 4
```

拼合的特性使得位域可以用来表示寄存器一类的结构

位域 - 内存布局案例

匿名位域表示跳过若干位；**0宽度位域**表示当前整数包已满，下个位域分配在新整数单元。**位域不能求地址**，建议与 union 配合获得位域的地址。

```
/*bit field layout*/
struct BF10 {
    int a:4;
} bit_field1;

struct BF11 {
    int a:4, :2, b:5;
} bit_field2;

struct BF20 {
    int a:4, :2;
    int b:27;
} bit_field3;

struct BF21 {
    int a:4, :2, :0;
    int b:4;
} bit_field4;
```

```
union {
    int i;
    struct BF11 bf11;
} u1 = {.i=0x5a};

union {
    int i[2];
    struct BF21 bf21;
} u2 = {.i={0x5a,0x5a}};

int main() {
    printf("bit_field1 size %d\n",sizeof(bit_field1)); 4

    printf("bit_field2 size %d\n",sizeof(bit_field2)); 4

    printf("bit_field3 size %d\n",sizeof(bit_field3)); 8

    printf("bit_field4 size %d\n",sizeof(bit_field4)); 8

    printf("%x,%x\n",u1.bf11.a,u1.bf11.b);
    printf("%x,%x\n",u2.bf21.a,u2.bf21.b);
}
```

自引用结构体

结构体（struct）类型，不能嵌套自身类型，但可以通过成员变量引用自身类型。

例如：struct NODE 中有个 next 指针引用 struct NODE

数组缺陷：

指定大小

```
typedef struct NODE Node;
typedef Node *NODE_PTR;

struct NODE {
    int customerData;
    NODE_PTR next;
};
```

预申明结构体别名

声明结构体指针类型

定义结构体类型

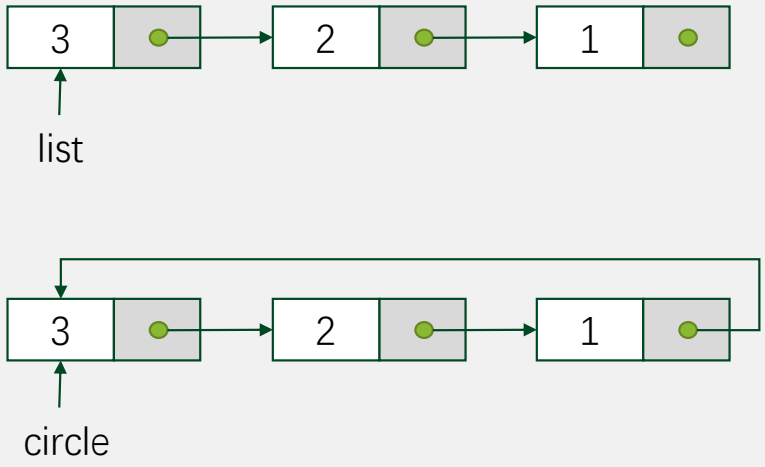
定义自引用成员

用指针实现自引用

结构体类型的成员包含自身引用指针，即自引用结构体。这种类似递归的定义，有哪些用途呢？

自引用结构体的价值

自引用结构体类型，可以表示序列结构数据，甚至更复杂的数据结构。



这些实例对象(变量)通过指针链接起来，同样可以表示序列，字符串等数据。

```c
/*self reference*/
int main() {
    Node list1 = {1,NULL};
    Node list2 = {2,&list1};
    Node list3 = {3,&list2};

    NODE_PTR list = &list3;
    for (;list;list = list->next) {
        printf("%d,",list->customerData);
    }
    printf("\n");

    list1.next = &list3;
    NODE_PTR circle = &list3;
    for (int i=0;i<10;i++,circle=circle->next) {
        printf("%d,",circle->customerData);
    }
    printf("\n");

    return 0;
}
```

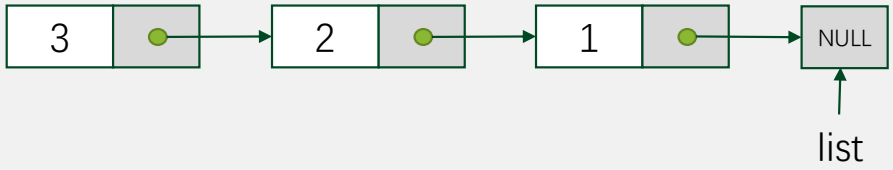
3 2 1

3,2,1,3,2,1,3,2,1,3,

自引用结构体 - 链表及其定义

链表(*LinkedList*)是一种表示序列的数据结构，又可称为**列表**。可递归定义如下：

- $LinkedList_0$ = NULL 是一个空链表
- $LinkedList_1$ = &$Node_1$，Node.next = NULL 是一个元素的链表
- $LinkedList_n$ = &$Node_n$，Node.next = $LinkedList_{n-1}$ 是n个元素的链表



推论：LinkedList -> next 是一个 n-1 个元素的子链表。

链表基本操作 - 创建与释放

链表(*LinkedList*) 基本操作是按前述定义进行创建或释放

```
/*list ops*/
NODE_PTR ListCreate(int i,NODE_PTR li) {
    NODE_PTR np = malloc(sizeof(Node));
    np->customerData = i;
    np->next = li;
    return np;
}

void ListFree(NODE_PTR li) {
    if (li) {
        NODE_PTR p = li->next;
        free(li);
        ListFree(p);
    }
}
```

链表基本操作 - 长度与索引

链表(*LinkedList*) 也可以和字符串一样计算长度或索引访问

```
/*list ops*/
size_t ListLength(NODE_PTR li) {
    if (li) return ListLength(li->next) + 1;
    else return 0;
}

NODE_PTR ListIndex(NODE_PTR li, size_t Idx) {
    if (li && Idx) return ListIndex(li->next,--Idx);
    else return li;
}
```

链表基本操作 - 元素添加与移除

链表(*LinkedList*) 按链表定义可以在链表头部添加和移除元素。

```
/*list ops*/
NODE_PTR ListAdd(NODE_PTR li,int i){
    return ListCreate(i,li);
}

NODE_PTR ListRemove(NODE_PTR li){
    if (li) {
        NODE_PTR p = li->next;
        free(li);
        return p;
    }
    return NULL;
}
```

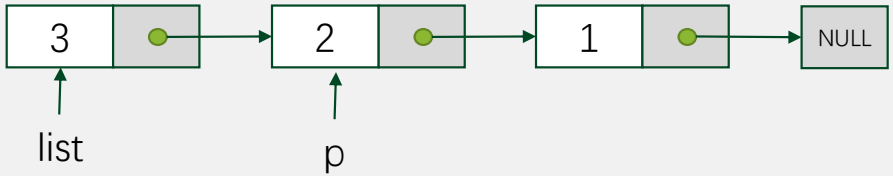
链表操作 - 指定位置元素插入与删除

链表(*LinkedList*) 删除指定位置元素或在指定位置插入!

显然，直接用当前指针p移除子链表首元素导致链表断裂，怎么办?

```
/*ex-list*/
NODE_PTR ListDelete(NODE_PTR *li){
    *li = ListRemove(*li);
    return *li;
}

NODE_PTR ListInsert(NODE_PTR *li, int i){
    ? 参见练习
}
```



这里函数参数是 struct NODE **li？ 指针的指针不作为二维数组，**这儿的作用是?**

链表操作 - 练习

请完成 *ex-list* 上的三个练习。

1. 给出 ListInsert 的函数描述及其实现代码
2. 给出 ListInsert 和 ListDelete 的测试代码
3. 课堂/课后完成链表元素查找和翻转链表操作
   - 链表(*LinkedList*) 元素查找。**思考：链表元素查找可用二分法吗？**
   - 翻转链表。要求:
     - 请给出两种或以上计算方法

链表 vs 数组

链表(*LinkedList*) 和**数组**都可以表示处理序列元素，它们有哪些异同？

	数组	链表	备注
定义	同类元素，连续存放， 固定长度	同类元素，离散存放， 动态长度	节点多个指针域
索引元素	指针算术运算 （双向随机访问）	从头顺序迭代计算 （单向顺序访问）	
计算长度	指针算术运算	顺序迭代计算	
插入删除	在尾部插入或删除方便 其他位置会导致元素迁移	方便任意位置插入和删除	
元素查找	有序序列可二分查找	只能顺序查找	
空间	少	多	

中山大學
SUN YAT-SEN UNIVERSITY

谢 谢

中山大学计算机学院

编制人：课题组