

第三次C语言培训

Vidar-Team



VIDAR TEAM

今天我们讲什么

- “陷阱”
- 连接
- 数据结构



今天我们讲什么

- “陷阱”
 - 词法陷阱
 - 语法陷阱
 - 语义陷阱
- 连接
- 数据结构



1. 词法陷阱



VIDAR TEAM

1.1

=

=



VIDAR TEAM

1.1

```
int a,b;  
printf(“%d”,b=2);  
  
a=b=2;
```



VIDAR TEAM

1.1

```
while(c==' ' || c == '\n')  
    ;
```

```
' ' || c == '\n'
```



VIDAR TEAM

1.2

|

&

||

&&



VIDAR TEAM

1.2

|

$$1 \mid 1 == 1$$

$$1 \mid 0 == 1$$

$$0 \mid 0 == 0$$



VIDAR TEAM

1.2

&

$1 \& 1 == 1$

$1 \& 0 == 0$

$0 \& 0 == 0$



VIDAR TEAM

1.2

flag

D	C	B	A
0	0	0	0
8	4	2	1

#define A 1 0001

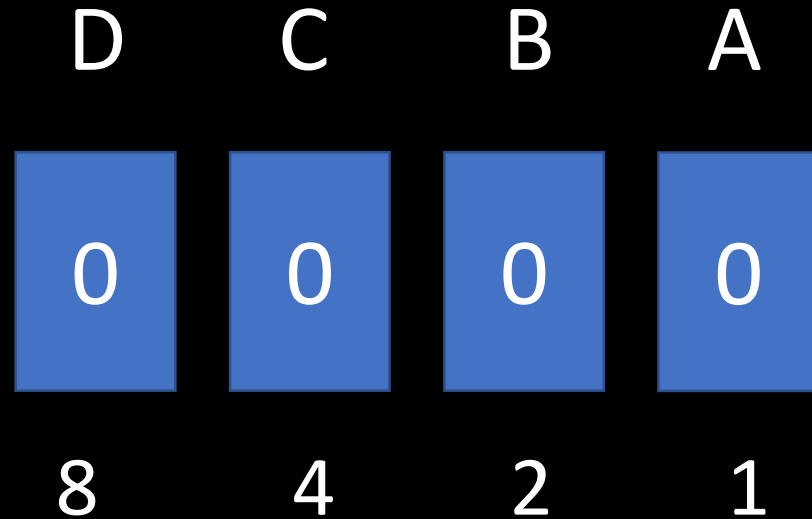
#define B 2 0010

#define C 4 0100

#define D 8 1000

1.2

flag



flag |= A;

flag &= ~A;

→ “0001 &= 1110”

flag == 0000

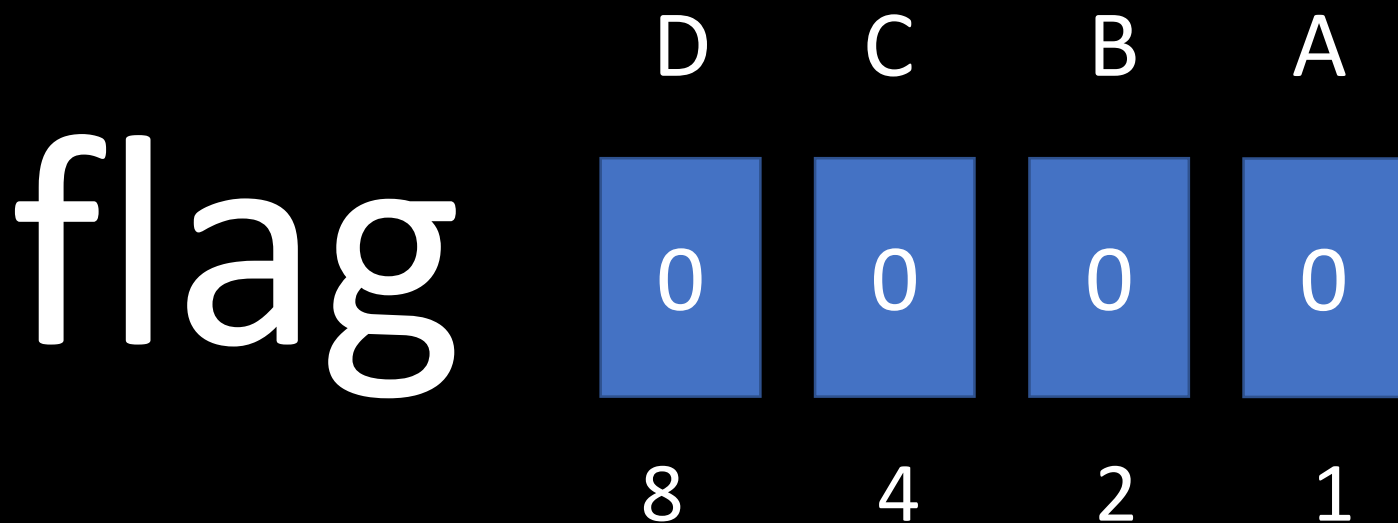
“0000 |= 0001”

flag == 0001



VIDAR TEAM

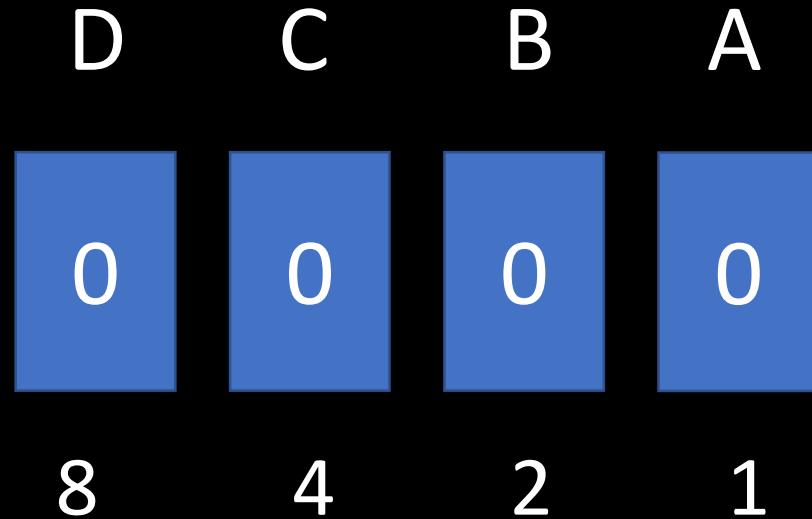
1.2



那么我们有办法知道A是打开了
还是没打开呢？

1.2

flag



flag & A;

"0001 &= 0001"

"0000 &= 0001"



VIDAR TEAM

1.3 词法分析中的“贪心法”

a---b; 单字符符号: / * =
...

a -- - b; 多字符符号: /* ==
...

我们怎么判断符号？ 编译器怎么判断符号？



VIDAR TEAM

1.3 词法分析中的“贪心法”

a---b; a/*p;

a -- - b; a/*p;

a/(*p);



1.4 字符与字符串

‘a’

“a”



VIDAR TEAM

1.4 字符与字符串

‘a’



“a”



VIDAR TEAM

2. 语法陷阱



VIDAR TEAM

2.1 理解一下这个先XD

```
(* (void(*) ()) 0) ();
```



2.1 算了先看看这个

```
int a;
```

```
int *a;
```



2.1 算了先看看这个 (数组+指针

```
int a[10];
```

```
int *a[10];
```

```
int (*a)[10];
```



2.1 函数指针

```
void a();
```

```
void (*a) ();
```

```
(void (*) ())a;
```



2.1

(void (*)())a;

(void ()())a;



VIDAR TEAM

2.1 现在能看懂了嘛？

```
(* (void(*) ()) 0) ();
```

为啥对0解引用？ 最后的括号又是啥？



VIDAR TEAM

2.1 再来看看这个

```
void a();
```

```
a();
```

```
a;
```



2.1 再来看看这个

```
1  #include<stdio.h>
2
3  void a()
4  {
5      ;
6  }
7
8  int main()
9  {
10     printf("%p", a);
11 }
```

Output:

00401410

a;

取a函数的地址



VIDAR TEAM

2.1 这次真的懂了吧?

```
(* (void(*) ()) 0) ();
```



2.2 分号

； 空语句



2.2 分号

粗心写上的分号

```
#include<stdio.h>
#include<math.h>
int main()
{
    double r,x,y;
    scanf("%lf",&r);
    for(y=r;y>=-r;y--);
    {
        for(x=-r;x<=r;x++)
        {
            if(x*x+y*y <= r*r+0.1 && x*x+y*y >= r*r-0.1)
            {
                printf("*");
            }
            else{
                printf(" ");
            }
        }
        printf("\n");
    }
    return 0;
}
```



2.2 怎样才能杜绝没必要的bug

规范代码格式!!!

使用VS Code 中的自带格式化 (Alt + Shift + F)

2.2 怎么才能杜绝不必要的bug

```
6  int main()  
7  {  
8      for(int i=0; i< 5;i++);  
9      for(int j=0;j< 6;j++)  
10     {  
11         printf("%d%d\n",i,    j)  
12     ;  
13     }  
14 }
```



2.2 怎么才能杜绝没必要的bug

```
3  int main()  
4  {  
5      for (int i = 0; i < 5; i++)  
6          ;  
7      for (int j = 0; j < 6; j++)  
8      {  
9          printf("%d%d\n", i, j);  
10     }  
11 }
```

非常明显



VIDAR TEAM

2.2 怎样才能杜绝没必要的bug

当然

最重要的还是要有良好的习惯



VIDAR TEAM

3. 语义陷阱



VIDAR TEAM

3.1

数组与指针



VIDAR TEAM

3.1 数组

- 1.C语言中只有一维数组
- 2.数组的大小必须在编译期就作为一个常数确定下来
- 3.数组的元素可以是任何类型的对象
(例如数组



3.1 数组

对于一个数组，我们只能够做两件事：

1. 确定该数组的大小，以及获得指向该数组下标为0的元素的指针
2. 其他有关数组的操作，实际上都是通过指针进行的



3.1 数组

```
int a[10];
```

```
int *p = a + 1;
```

数组名会被当做指向该数组下标为0的指针



VIDAR TEAM

3.1 数组

```
p = &a;
```

大多数早期版本的C语言实现中并没有所谓“数组的地址”这一概念



3.2

二维数组



VIDAR TEAM

3.2001 从类型的角度取考虑

```
int M [3] [4];
```



3.2001

```
int M [3] [4];
```

```
int (M [3]) [4];
```



VIDAR TEAM

3.2002 现在来理解一下

```
printf("%d",M[1][2]);
```

发生了什么？



VIDAR TEAM

3.2003 语法糖

$M[1][2];$

$*(M+1)[2];$

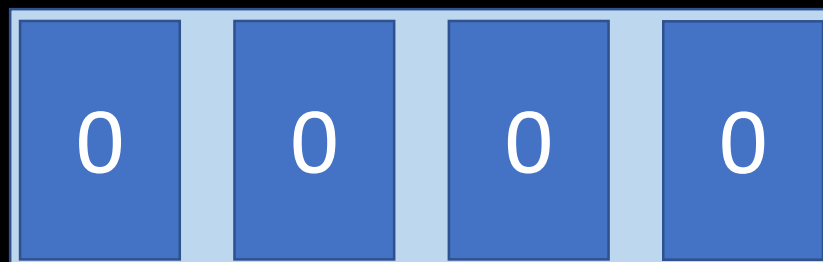
$*(*(M+1)+2);$



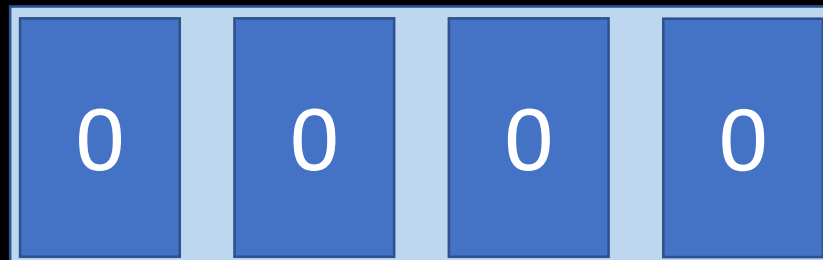
3.2004 难道它们在内存里也是嵌套的?

等等, 内存长这样?

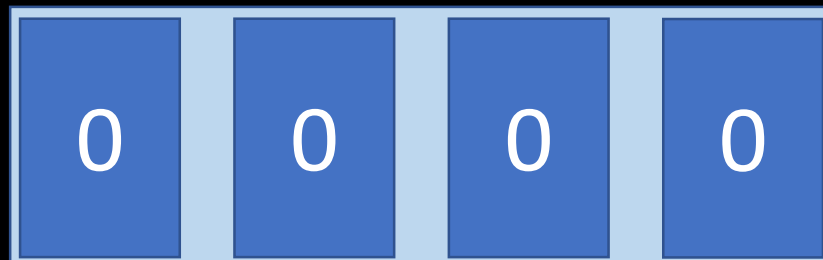
*M M[0]



*(M+1) M[1]

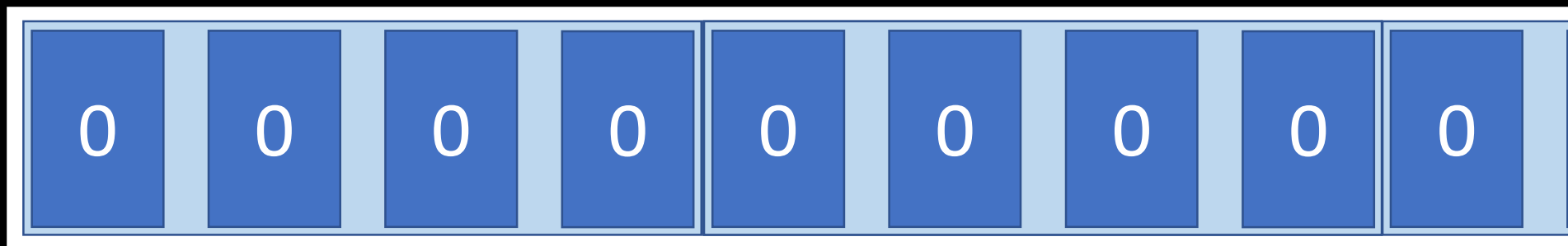


*(M+2) M[2]



[0] [1] [2] [3]

3.2004



$M[0][0]$

$M[1][0]$

$M[2][0]$

$M[0][4]$

M是什么?

$*(*M + 0)$

$*(*(M + 1) + 0)$

$*(*M + 4)$

3.3

为main函数提供返回值



3.3

返回值0代表程序执行成功
非0代表失败

如果一个程序没有return 0; 会怎么样?

可能会返回某个“垃圾”整数

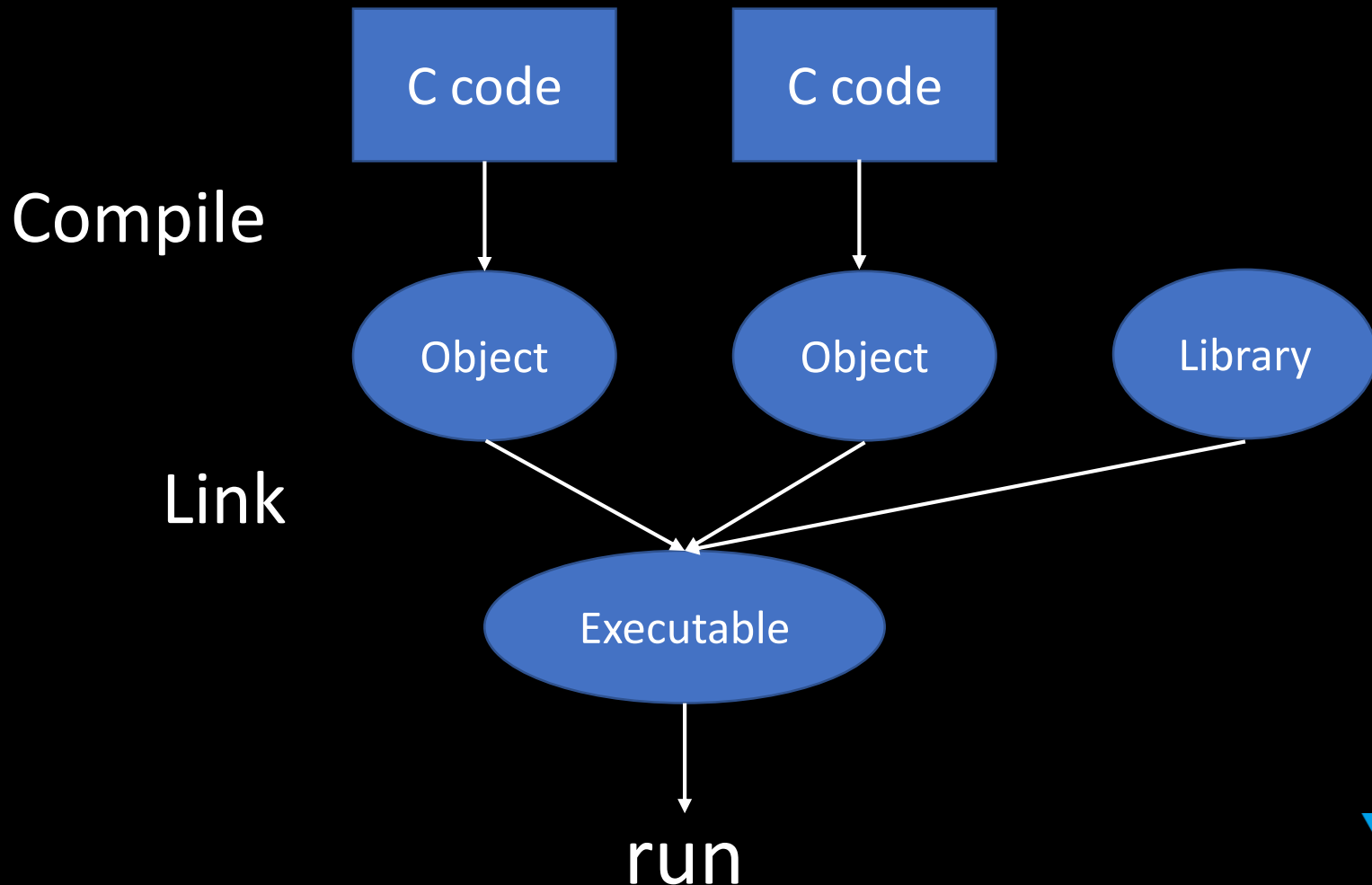


4. 连接



VIDAR TEAM

4.1 分别编译 (Separate Compilation)



4.2 定义与声明

```
int a = 7;
```

定义

```
extern int a;
```

声明

extern说明是一个对外部变量a的引用



VIDAR TEAM

4.2

```
void func();
```

声明

```
void func()
```

定义

```
{
```

```
    printf("hello world");
```

```
}
```



VIDAR TEAM

4.3 命名冲突与static

```
static int a;
```

```
int a;
```

static 也适用于函数



4.3 头文件

每个外部对象只在一个地方声明

这个声明一般就在头文件中

需要用到该对象的所有模块都应该包括这个头文件



5. 数据结构 (简单



VIDAR TEAM

5.1 结构体-分清“类型”与变量

先来看结构体的三种写法



VIDAR TEAM

5.1 结构体-分清“类型”与变量

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

struct Node 是一种类型



5.1 结构体-分清“类型”与变量

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

struct Node 是一种类型
但node是一个变量
不建议这么写!!!



5.1 结构体-分清“类型”与变量

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



两种需求糅合到一起了



5.1 结构体-分清“类型”与变量

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

Node 是一种类型，没有实体存在



5.1 结构体-分清“类型”与变量

```
typedef struct Node {  
    int data = 0;  
    struct Node *next = 0;  
}Node;
```

没有这种写法，结构体只负责描述内存排布的方式

--by oyeye



VIDAR TEAM

5.1 结构体-分清“类型”与变量

那我们就来看看内存排布的方式是怎样的吧！



5.1 结构体-内存排布

'a' == 97 == 0x61 ==

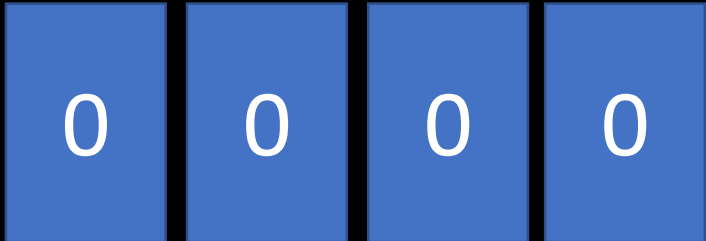
'a'

一个字节 == 8位二进制数
== 2位十六进制数



VIDAR TEAM

5.1 结构体-内存排布

int a == 

一般来说是4个字节
也就是32位二进制数



5.1 结构体-内存排布

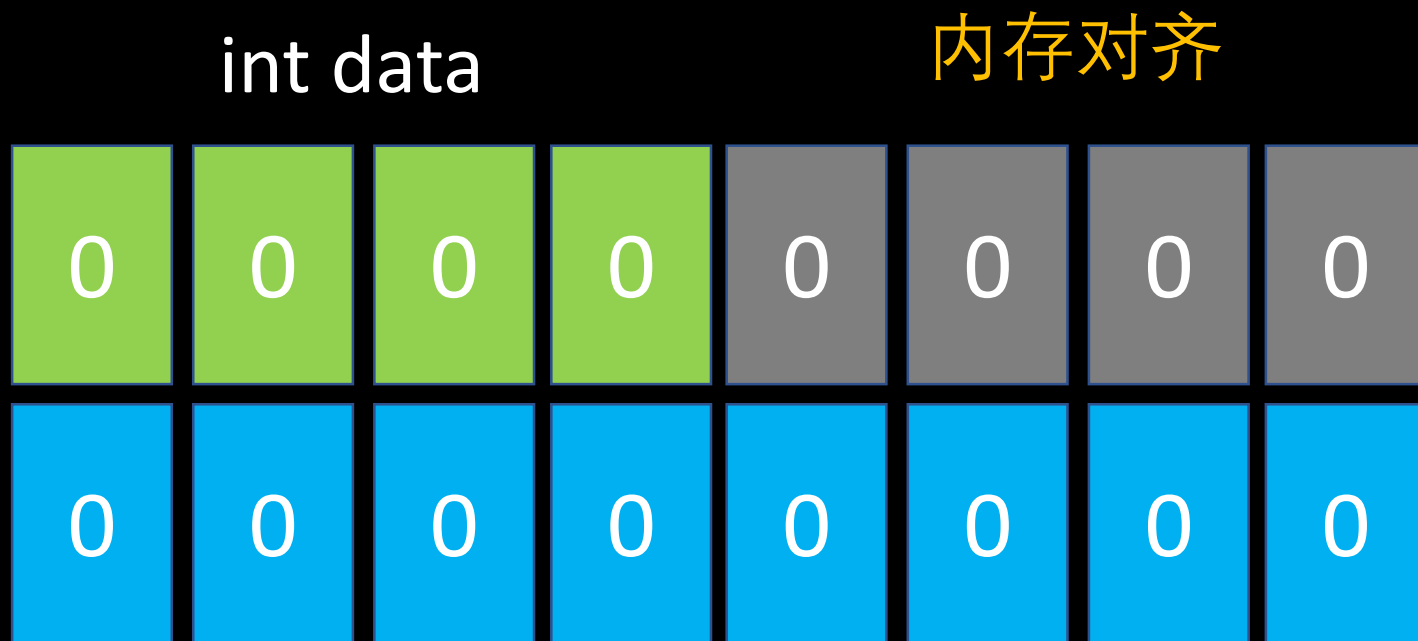
`int *a ==` 

64位的程序是8个字节
也就是64位二进制数



5.1 结构体-内存排布

有没有发现这玩意和什么我们之前讲过的什么东西很像？

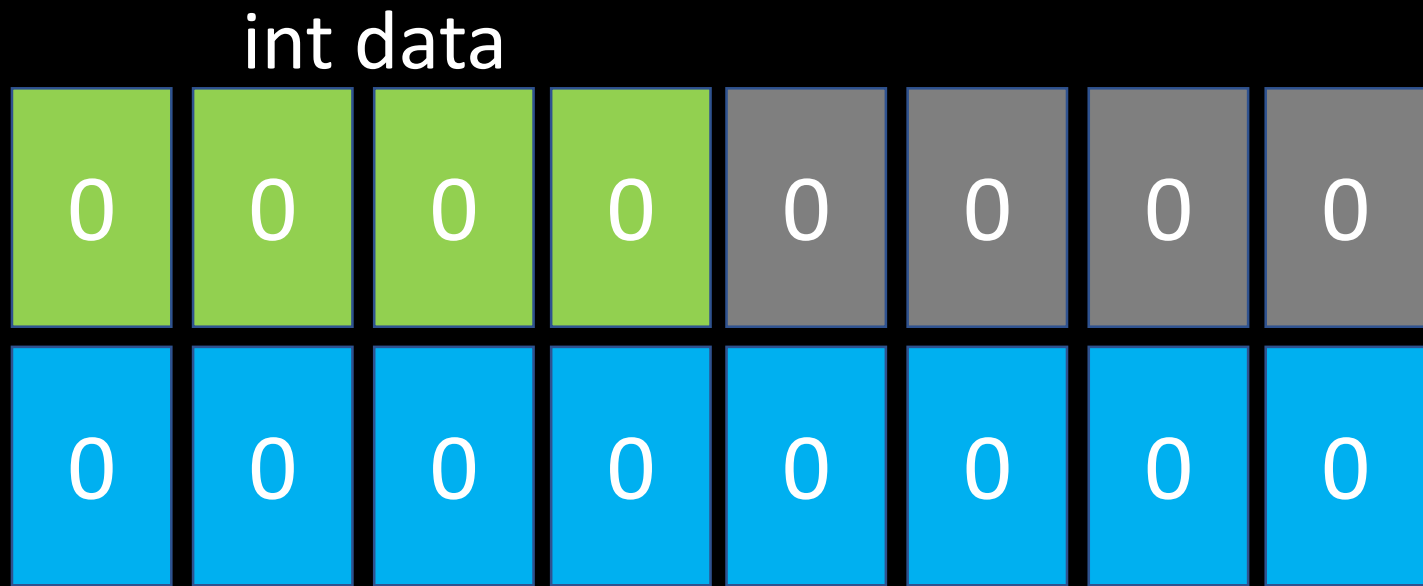


struct Node *next

5.1 结构体-内存排布

Node a;

`a.data == (int *)a[0] == (int *)*a`



struct Node *next

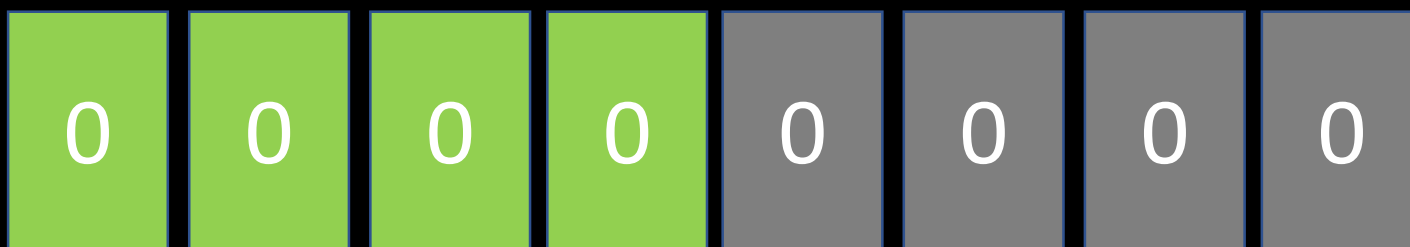
5.2 联合（共用体）

```
typedef union Student{  
    char name[4];  
    char *largeName;  
} Student;
```



5.2 联合-内存排布

char name[4]



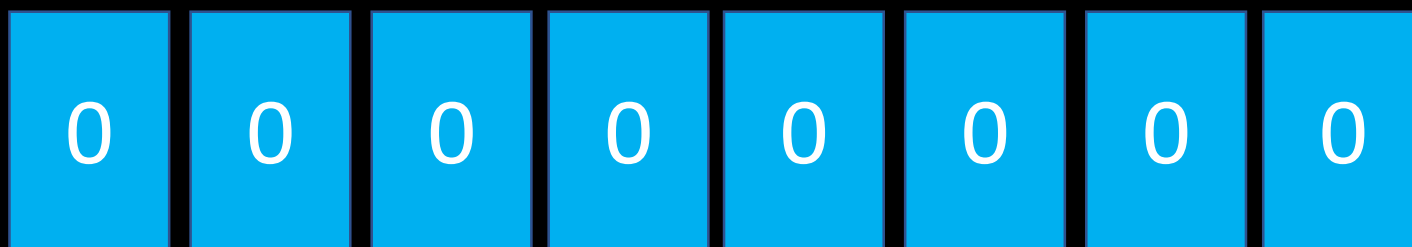
使用name时



VIDAR TEAM

5.2 联合-内存排布

`char *largeName`



使用`largeName`时



VIDAR TEAM

5.3 单向链表

数组的缺点？

提一个需求， 删除数组下标为0的元素



5.3 单向链表

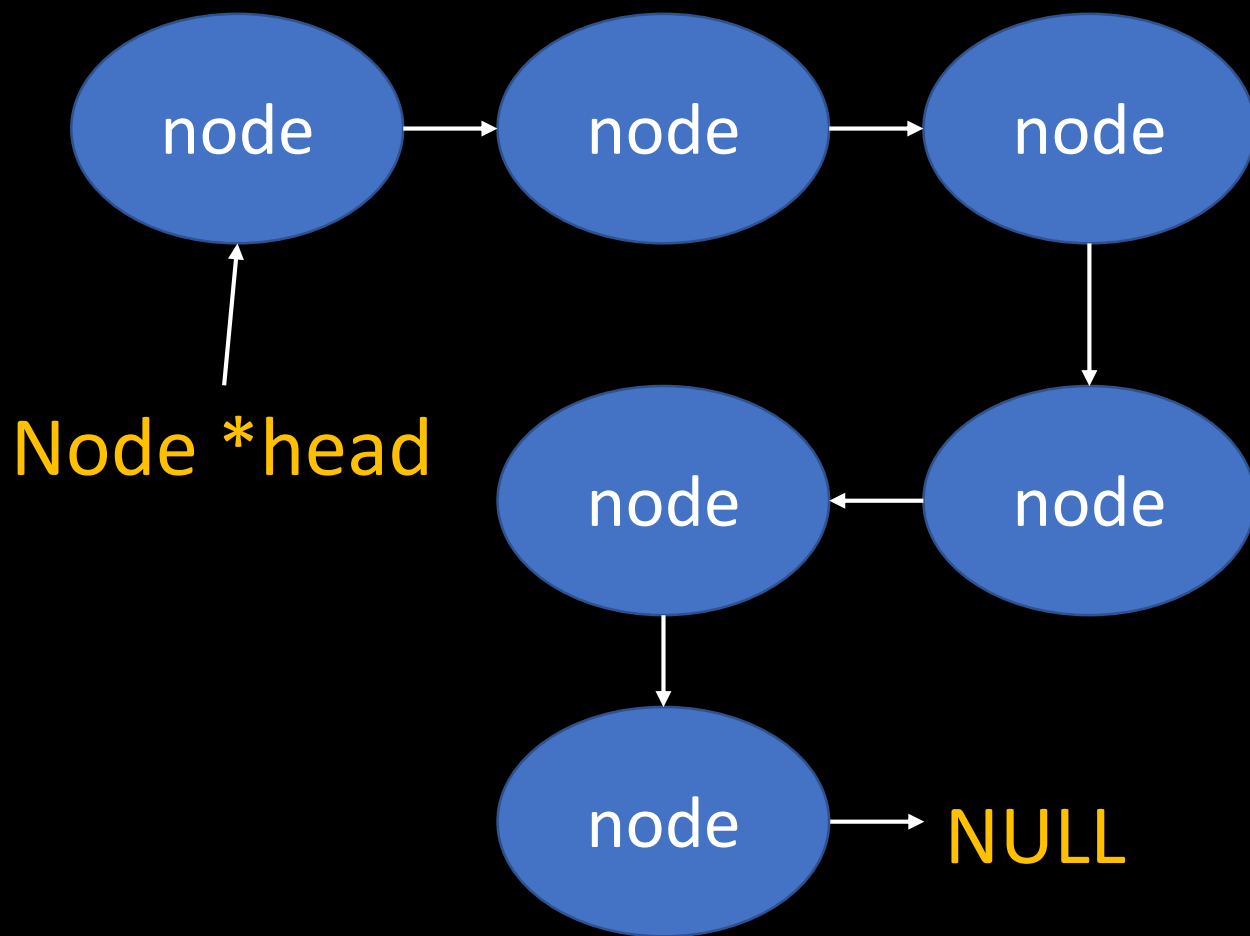
```
struct Node {  
    int data;  结点中存放的值  
    struct Node *next; 指向下一个结点  
};
```

一个结点



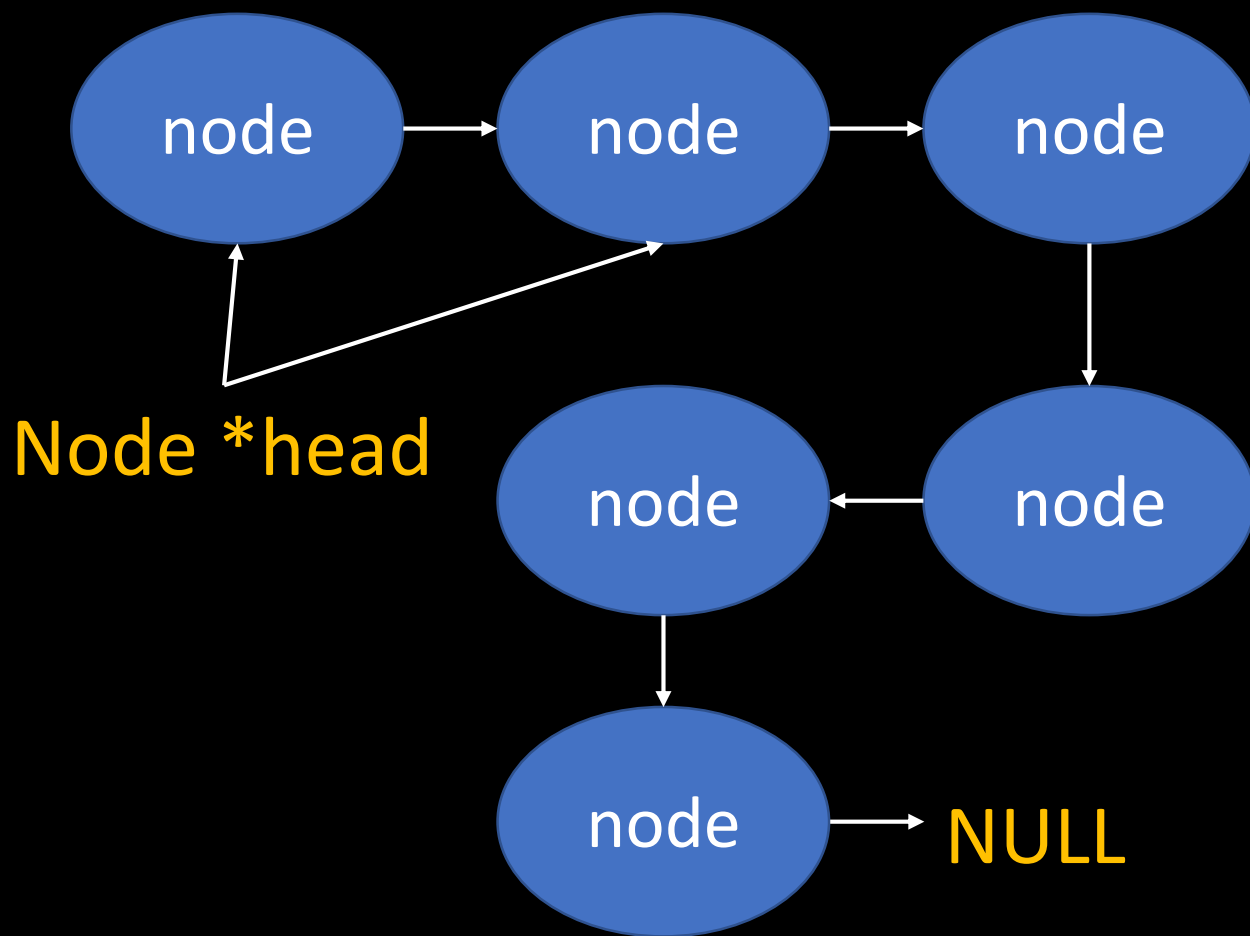
5.3 单向链表

怎么删除第一个结点呢?



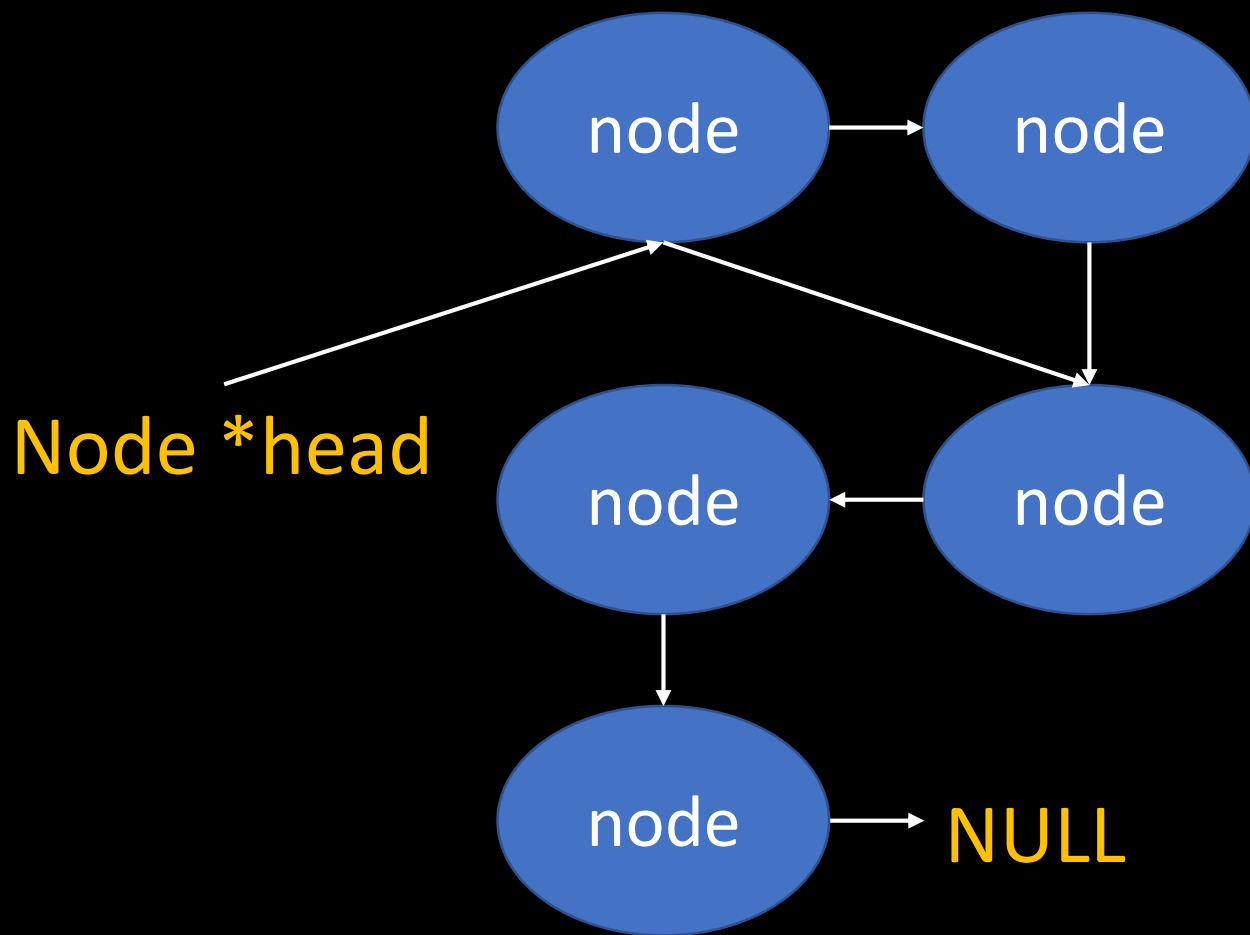
5.3 单向链表

怎么删除第二个结点呢?



5.3 单向链表

怎么在第一个结点后添加一个结点呢?



5.3 单向链表

