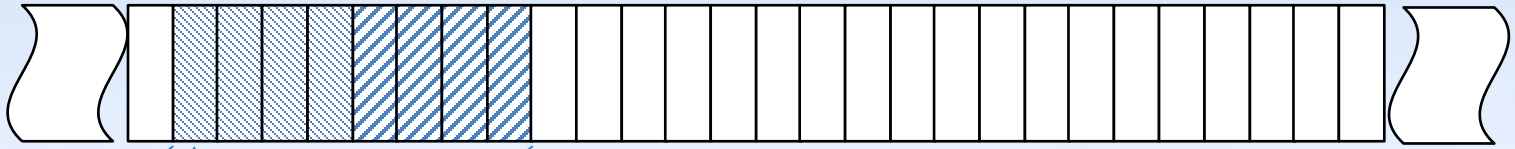
再来观察int e[2]

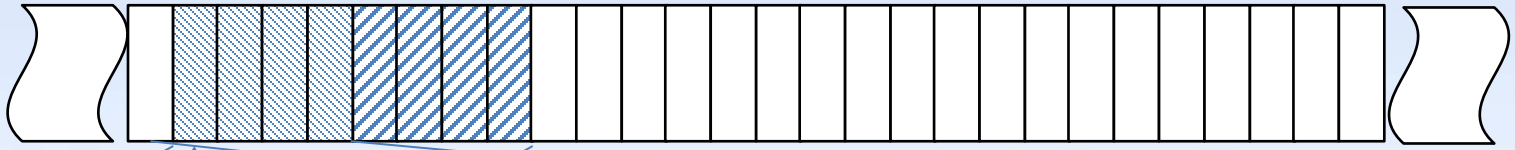


0x0076AB11

A: 0x0076AB11
Obj_T: int[2]
N: e
S: 8
V: 0x0076AB11
V_T: int*

思考1：size为什么是8？
思考2：Object_Type是int[2]
思考3：value为什么是和Address一样？
思考4：Type为什么是int*

e[0]到底是什么？



0x0076AB11

A: 0x0076AB11
Obj_T: int[2]
N: e
S: 8
V: 0x0076AB11
V_T: int*

e的内存六元组

e[0]=*(e+0)

e+0: e的value+sizeof(*e)*0

type保持不变

取值

<0x0076AB11, int*>

e+0 <0x0076AB11, int*>

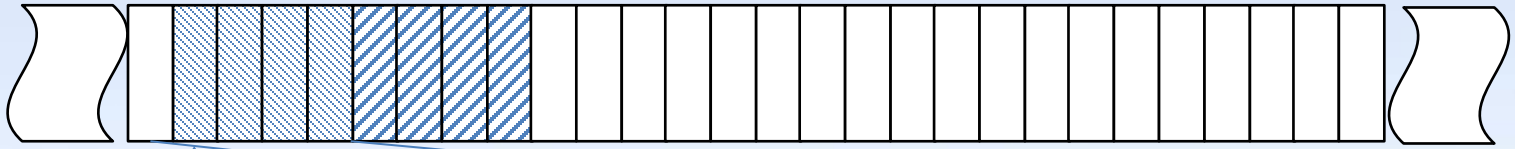
*定位

*定位

A: 0x0076AB11
Obj_T: int
N: N/A
S: 4
V: ?
V_T: int

e/(e+0)的抽象内存六元组

&(e[0])、sizeof(e[0])和e[0]



0x0076AB11

&(e[0]): <0x0076AB11, int*>

sizeof(e[0]): <4, size_t>

e[0]: <?, int>
因为int e[2]; 没有初始化这块内存
如果int e[2] = {1, 2};
e[0]: <1, int>

A: 0x0076AB11
Obj_T: int
N: N/A
S: 4
V: ?
V_T: int

e/(e+0)的内存六元组

e=e+1和e++为什么会报错

```
int main()
{
    int e[2];

    e = e + 1;

    return 0;
}
```

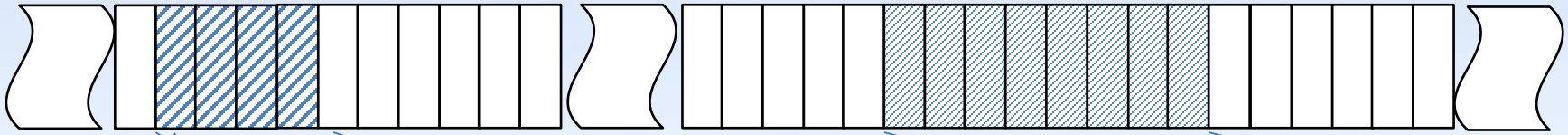
```
=== Build file: "no target" in "no project" (compiler: unknown) ===
In function 'main':
error: assignment to expression with array type
=== Build failed: 1 error(s), 0 warning(s) (0 minute(s), 0 second(s)) ===
```

A: 0x0076AB11
Obj_T: int[2]
N: e
S: 8
V: 0x0076AB11
V_T: int*

数组的Value一定是指向数组第一个元素的地址编号
对于数组e来说，V的值必须和A相等

因此，e并不是一个常量，它的值不能更改是语法限制的
对常量修改的错误应该是：assignment of read-only variable 'e'

int* p=e；p++为什么可以？



0x0046A521

0x0076AB11

```
int main()
{
    int e[2];
    int* p = e;

    p = p + 1;

    return 0;
}
```

A: 0x0046A521
Obj_T: int*
N: p
S: 4
V: 0x0076AB11
V_T: int*

p的内存六元组

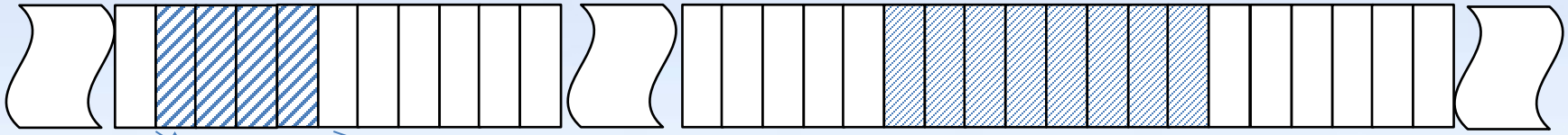
p=e;

<0x0076AB11, int*>

A: 0x0076AB11
Obj_T: int[2]
N: e
S: 8
V: 0x0076AB11
V_T: int*

e的内存六元组

int* p=e；p++为什么可以？



0x0046A521

0x0076AB11

```
int main()
{
    int e[2];
    int* p = e;

    p = p + 1;

    return 0;
}
```

A: 0x0046A521
Obj_T: int*
N: p
S: 4
V: 0x0076AB15
V_T: int*

<0x0076AB15, int*>

p=p+1

p+1

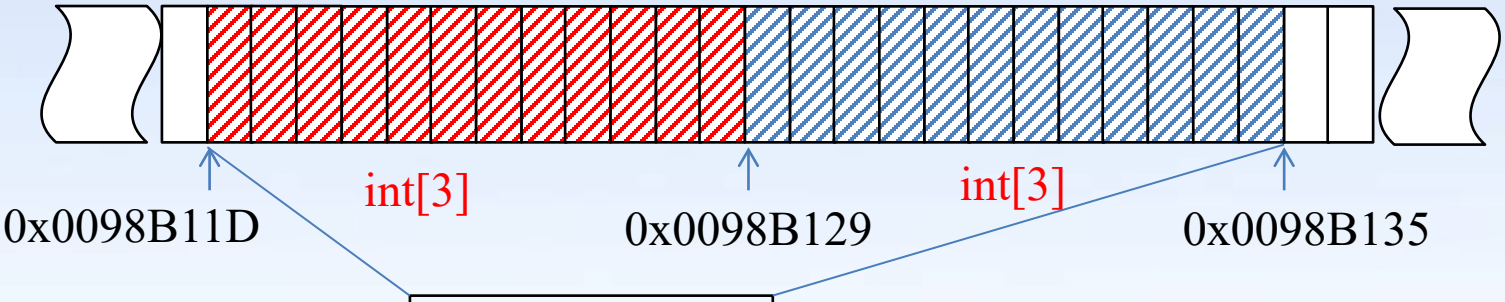
p取值

<0x0076AB11, int*>

p的内存六元组

修改的是p的值
没有修改e的值

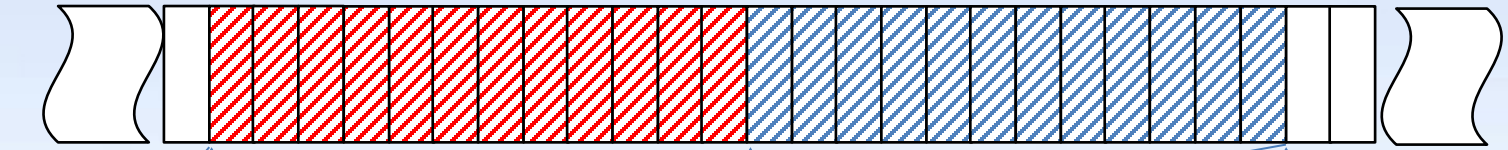
再来观察：int g[2][3]



A: 0x0098B11D
Obj_T: int[2][3]
N: g
S: 24
V: 0x0098B11D
V_T: int(*)[3]

思考1：size为什么是24？
思考2：Object_Type是int[2][3]
思考3：Value为什么是和Address一样？
思考4：Value Type为什么是int(*)[3]

&g的返回值是多少？



A: 0x0098B11D
Obj_T: int[2][3]
N: g
S: 24
V: 0x0098B11D
V_T: int(*)[3]

&

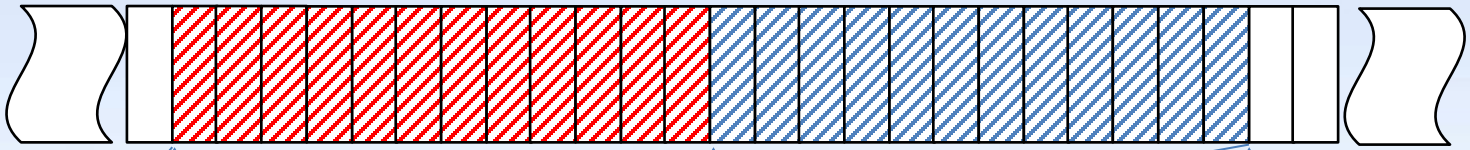
<0x0098B11D, int(*)[2][3]>

如何定义一个变量y=&g

y的类型必须是int(*)[2][3]

int (*y)[2][3] = &g

sizeof(g)的返回值是多少？



A: 0x0098B11D
Obj_T: int[2][3]
N: g
S: 24
V: 0x0098B11D
V_T: int(*)[3]

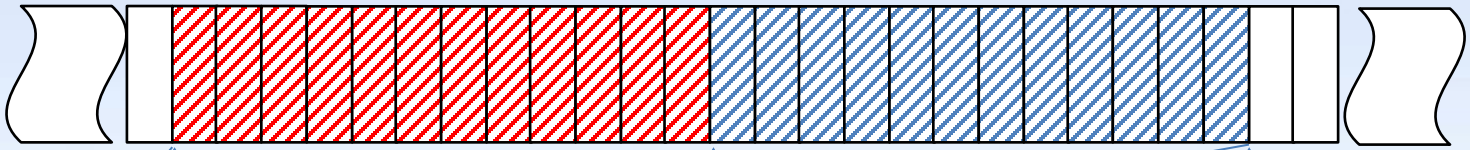
sizeof → <24, size_t>

如何定义一个变量y=sizeof(g)

y的类型必须是size_t

size_t y = sizeof(g)

g的返回值是多少？



A: 0x0098B11D
Obj_T: int[2][3]
N: g
S: 24
V: 0x0098B11D
V_T: int(*)[3]

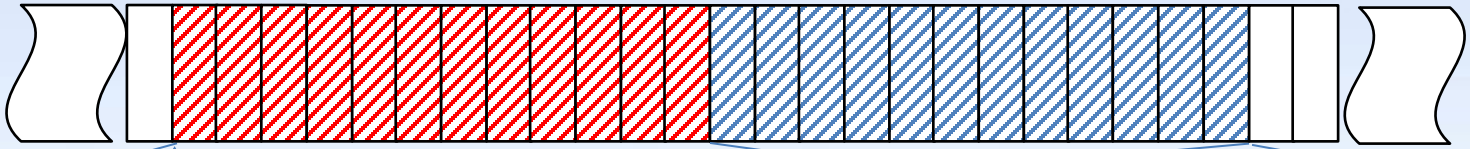
取值 → <0x0098B11D, int(*)[3]>

如何定义一个变量y=g

y的类型必须是int(*)[3]

int (*y)[3] = g

g[1]到底是什么？



0x0098B11D　int[3]　int[3]

A: 0x0098B11D
Obj_T: int[2][3]
N: g
S: 24
V: 0x0098B11D
V_T: int(*)[3]

g的内存六元组

g[1]=*(g+1)

g+1: g的value+sizeof(*g)*1

g+1的type保持不变

取值

<0x0098B11D, int(*)[3]> *定位

g+1<0x0098B129, int(*)[3]>

*定位

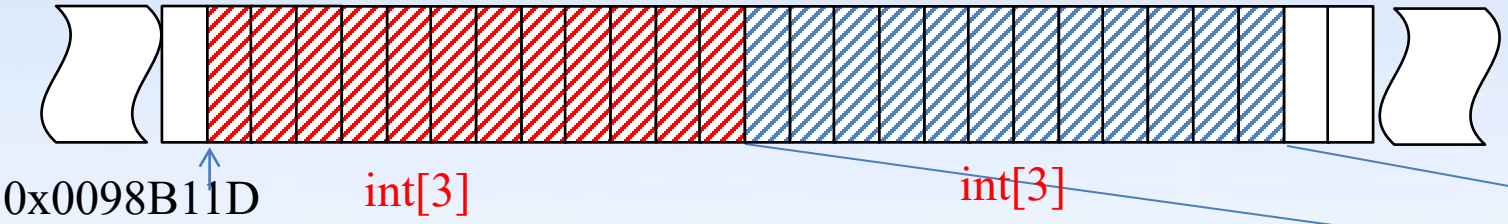
A: 0x0098B11D
Obj_T: int[3]
N: N/A
S: 12
V: 0x0098B11D
V_T: int*

*g的内存六元组

A: 0x0098B129
Obj_T: int[3]
N: N/A
S: 12
V: 0x0098B129
V_T: int*

*(g+1)的内存六元组

&(g[1])、sizeof(g[1])和g[1]



&(g[1]): < 0x0098B129, int(*)[3]>

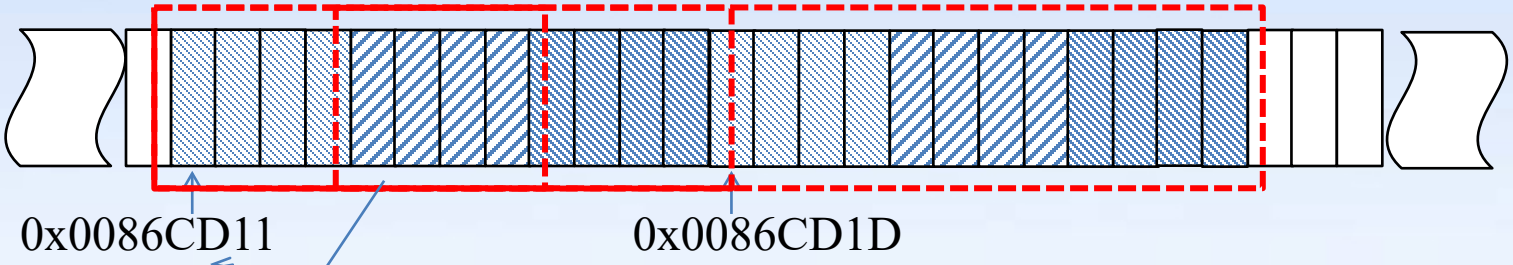
sizeof(g[1]): <12, size_t>

g[1]: <0x0098B129, int*>

A: 0x0098B129
Obj_T: int[3]
N: N/A
S: 12
V: 0x0098B129
V_T: int*

*(g+1)/g[1]的内存六元组

示例3



char d[2][3][4]

&d[0][1]应该返回什么？

<0x0086CD15, char(*)[4]>

1、d是lvalue，定位到24个字节，char[2][3][4]类型

2、d取值，返回值为<0x0086CD11, char(*)[3][4]>

3、d[0]是lvalue，定位到12个字节，char[3][4]类型

4、d[0]取值，返回值为<0x0086CD11, char(*)[4]>

5、d[0][1]是lvalue，定位到4个字节，char[4]类型

6、&d[0][1]取这块内存的地址，返回<0x0086CD15, char(*)[4]>

理解这六个例子

给定一个int a[2][3] = {0};
修改a[1][2]的值

```
a[1][2] = 1;
printf("%d\n", a[1][2]);

(*(&a))[1][2] = 2;
printf("%d\n", a[1][2]);

(&(*a))[1][2] = 3;
printf("%d\n", a[1][2]);

(a+1-1)[1][2] = 4;
printf("%d\n", a[1][2]);
```

这四个赋值语句都成功的修改了a[1][2]

1、为什么？
2、有区别吗？

再来两个

```
1[a][2] = 5;
printf("%d\n", a[1][2]);
1[2][a] = 6;
```

为什么也成功修改了a[1][2]

为什么编译不过了？

这背后的工作机理到底是什么呢？
这几个语句并不是所谓的技巧，帮助理解数组名

复习1：E1[E2]等价于*((E1)+(E2))

The definition of the subscript operator [] is that E1[E2] is identical to (*((E1)+(E2))).

*((E1)+(E2))这个形式非常重要

E1和E2都是表达式

其中一个表达式返回值类型是一个合法的指针类型

另一个表达式返回值类型是一个合法的整数类型

注意：没有要求E1/E2分别对应哪种类型表达式

复习2：Identifier是lvalue

1、对象标识符是一个lvalue

2、lvalue是一个合法的表达式

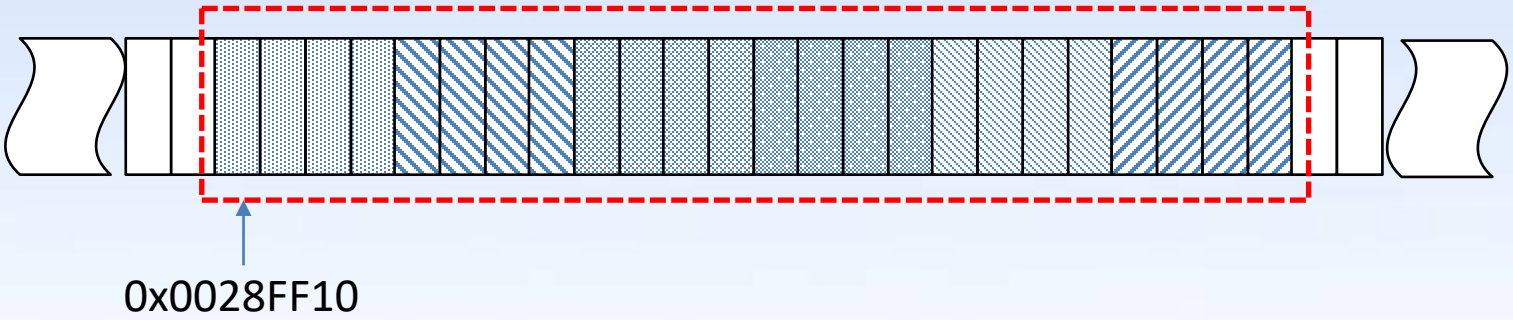
3、lvalue可以用来定位一个对象

给定一个声明语句int a[2][3];

a是一个标识符，是一个lvalue（基础表达式），可以定位那个24字节对象

数组名不是指针，更不是什么特殊指针，没有任何的特殊性

复习3：int a[2][3]分配的这块内存属性



<0x0028FF10, int[2][3], a, 24, 0x0028FF10, int(*)[3]>

复习概念：非数组对象 vs. 数组对象 取值

复习4：表达式的值

表达式都会被Evaluate，返回<Value, Value_Type>

表达式可以由子表达式构成，逐步Evaluate

例如1+2: 1和2是基础表达式（常量是基础表达式），1+2是加法表达式

1: <1 ,int>
2: <2, int>
1+2: <3, int>

复习5：lvalue表达式的值

如果一个表达式exp是lvalue，该表达式有三种evaluate的操作

假设该exp定位的内存：<Address, Object_Type, Name, Size, Value, Value_Type>

1、&exp: <Address, Object_Type对应的指针类型>

2、sizeof(exp): <Size, size_t>

3、exp: <Value, Value_Type>

Lvalue进一步分为可修改的lvalue vs. 不可修改的lvalue

复习6：lvalue表达式在等号两边差别

如果exp是不可修改的lvalue，则不可以出现在等号左边

int a[2][3]; a = NULL; ✗

如果exp是可以修改的lvalue，则可以出现在等号左边

int b; b = 3; ✓

lvalue可以出现在等号右边以及充当任何表达式的子表达式

int c[10];

c = NULL; ✗

c[0] = a[0][0]; ✓

c, c[0], a, a[0], a[0][0]都是lvalue

c作为整个表达式，不能出现在等号左边，c[0]作为一个整个表达式，可以出现在等号左边

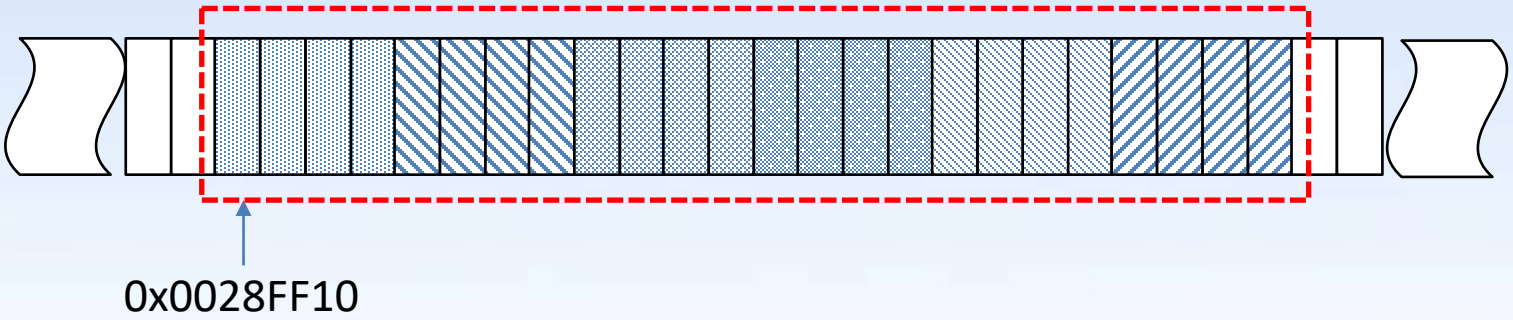
看看C语言标准如何理解数组及其操作

我们来看一个C语言标准中的示例，给定int x[3][5]，如何理解x[i][j]

Here x is a 3 × 5 array of objects of type int; more precisely, x is an array of three element objects, each of which is an array of five objects of type int. In the expression x[i], which is equivalent to (*((x)+(i))), x is first converted to a pointer to the initial array of five objects of type int. Then i is adjusted according to the type of x, which conceptually entails multiplying i by the size of the object to which the pointer points, namely an array of five int objects. The results are added and indirection is applied to yield an array of five objects of type int. When used in the expression x[i][j], that array is in turn converted to a pointer to the first of the objects of type int, so x[i][j] yields an int.

这段描述的文字涉及到的知识点有哪些呢？

int a[2][3]={0},a[1][2]的实际定位过程

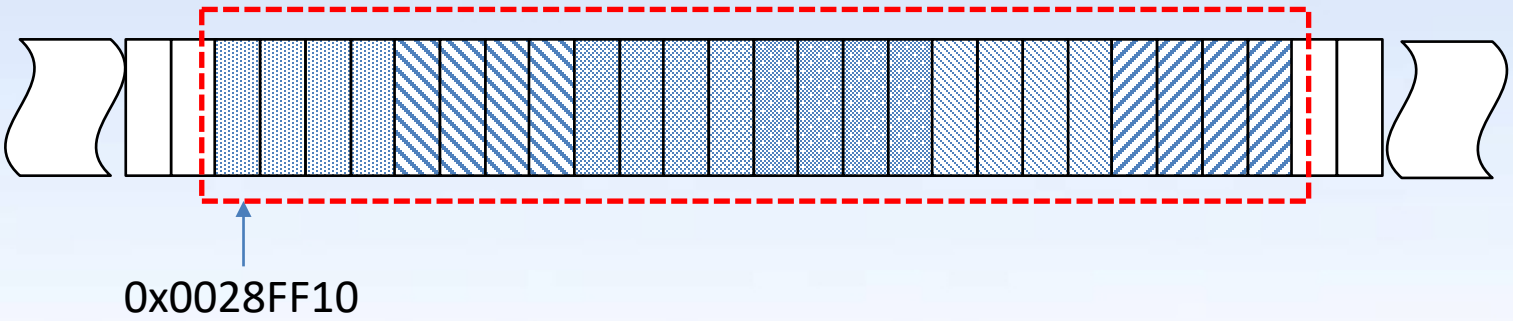


int a[2][3]在内存中分配了一个24字节的连续内存，该对象描述如下：

<0x0028FF10, int[2][3], a, 24, 0x0028FF10, int(*)[3]>

现在这段内存全部都初始化为0了

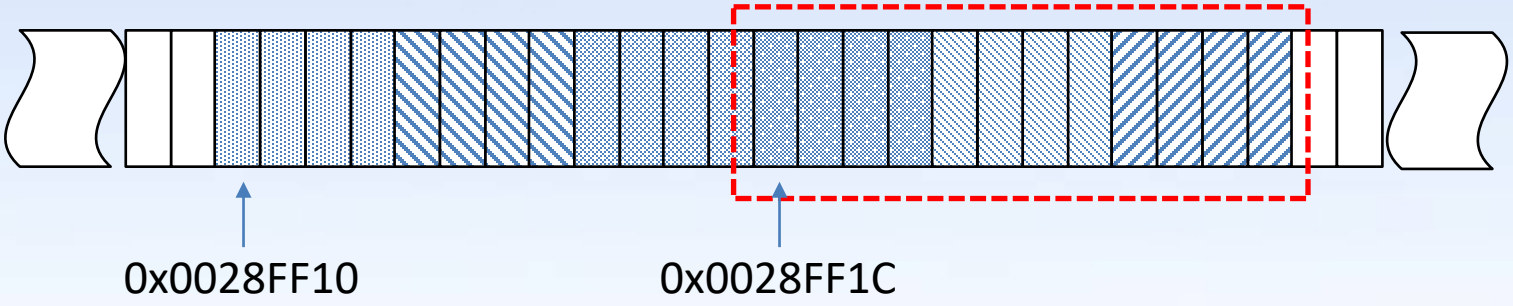
给定a[1][2]这个表达式，先识别a



a[1][2]是一个表达式，这其中，基础表达式有a、1、2

a是一个identifier，是一个lvalue，因此可以定位到这块内存

给定a[1][2]这个表达式，识别a[1]



a是a[1]这个表达式的子表达式，因此表达式a的取值如下：

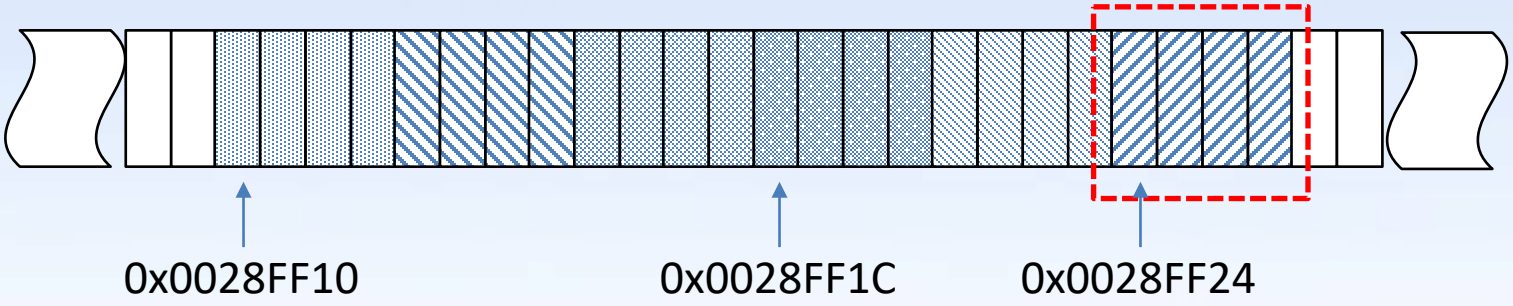
a: <0x0028FF10, int(*)[3]>

a[1]等价于*(a+1)，下面运算a+1，获得值如下：

<0x0028FF10+1*sizeof(*a), int(*)[3]>，即<0x0028FF1C, int(*)[3]>

(a+1)定位的内存：<0x0028FF1C, int[3], 12, N/A, 0x0028FF1C, int>

给定a[1][2]这个表达式，识别a[1][2]



a[1]是a[1][2]这个表达式的子表达式，因此表达式a[1]的取值如下：

a[1]: <0x0028FF1C, int*>

a[1][2]等价于*(a[1]+2)，下面运算a[1]+2，获得值如下：

<0x0028FF1C+2*sizeof(*(a[1])), int*>，即<0x0028FF24, int*>

*(a[1]+2)定位的内存：<0x0028FF24, int, 4, N/A, 0, int>

如何正确地理解数组及其操作

我们来看一个C语言标准中的示例，给定**int x[3][5]**，如何理解**x[i][j]**

Here x is a 3 × 5 array of objects of type int; more precisely, x is an array of three element objects, each of which is an array of five objects of type int. In the expression x[i], which is equivalent to (*((x)+(i))), x is first converted to a pointer to the initial array of five objects of type int. Then i is adjusted according to the type of x, which conceptually entails multiplying i by the size of the object to which the pointer points, namely an array of five int objects. The results are added and indirection is applied to yield an array of five objects of type int. When used in the expression x[i][j], that array is in turn converted to a pointer to the first of the objects of type int, so x[i][j] yields an int.

强调int x[3][5]是一个一维数组，每个元素是一个int[5]

如何正确地理解数组及其操作

我们来看一个C语言标准中的示例，给定int x[3][5]，如何理解x[i][j]

Here x is a 3 × 5 array of objects of type int; more precisely, x is an array of three element objects, each of which is an array of five objects of type int. In the expression x[i], which is equivalent to (*((x)+(i))), x is first converted to a pointer to the initial array of five objects of type int. Then i is adjusted according to the type of x, which conceptually entails multiplying i by the size of the object to which the pointer points, namely an array of five int objects. The results are added and indirection is applied to yield an array of five objects of type int. When used in the expression x[i][j], that array is in turn converted to a pointer to the first of the objects of type int, so x[i][j] yields an int.

x是一个lvalue，定位到了一个对象，该对象类型是int[3][5]，
因此该对象的值是一个指向第一个int[5]元素类型的指针

如何正确地理解数组及其操作

我们来看一个C语言标准中的示例，给定int x[3][5]，如何理解x[i][j]

Here x is a 3 × 5 array of objects of type int; more precisely, x is an array of three element objects, each of which is an array of five objects of type int. In the expression x[i], which is equivalent to (*((x)+(i))), x is first converted to a pointer to the initial array of five objects of type int. Then i is adjusted according to the type of x, which conceptually entails multiplying i by the size of the object to which the pointer points, namely an array of five int objects. The results are added and indirection is applied to yield an array of five objects of type int. When used in the expression x[i][j], that array is in turn converted to a pointer to the first of the objects of type int, so x[i][j] yields an int.

先求x+i的值，也就是要跳过i个x指向的对象大小

如何正确地理解数组及其操作

我们来看一个C语言标准中的示例，给定int x[3][5]，如何理解x[i][j]

Here x is a 3 × 5 array of objects of type int; more precisely, x is an array of three element objects, each of which is an array of five objects of type int. In the expression x[i], which is equivalent to (*((x)+(i))), x is first converted to a pointer to the initial array of five objects of type int. Then i is adjusted according to the type of x, which conceptually entails multiplying i by the size of the object to which the pointer points, namely an array of five int objects. The results are added and indirection is applied to yield an array of five objects of type int. When used in the expression x[i][j], that array is in turn converted to a pointer to the first of the objects of type int, so x[i][j] yields an int.

*(x+i)，即x[i]就是定位一个从x+i地址开始的int[5]的对象

如何正确地理解数组及其操作

我们来看一个C语言标准中的示例，给定int x[3][5]，如何理解x[i][j]

Here x is a 3 × 5 array of objects of type int; more precisely, x is an array of three element objects, each of which is an array of five objects of type int. In the expression x[i], which is equivalent to (*((x)+(i))), x is first converted to a pointer to the initial array of five objects of type int. Then i is adjusted according to the type of x, which conceptually entails multiplying i by the size of the object to which the pointer points, namely an array of five int objects. The results are added and indirection is applied to yield an array of five objects of type int. When used in the expression x[i][j], that array is in turn converted to a pointer to the first of the objects of type int, so x[i][j] yields an int.

x[i]是x[i][j]这个表达式的子表达式，因为x[i]是一个lvalue，
定位一个类型为int[5]的对象，因此x[i]就取值成一个int*的指针

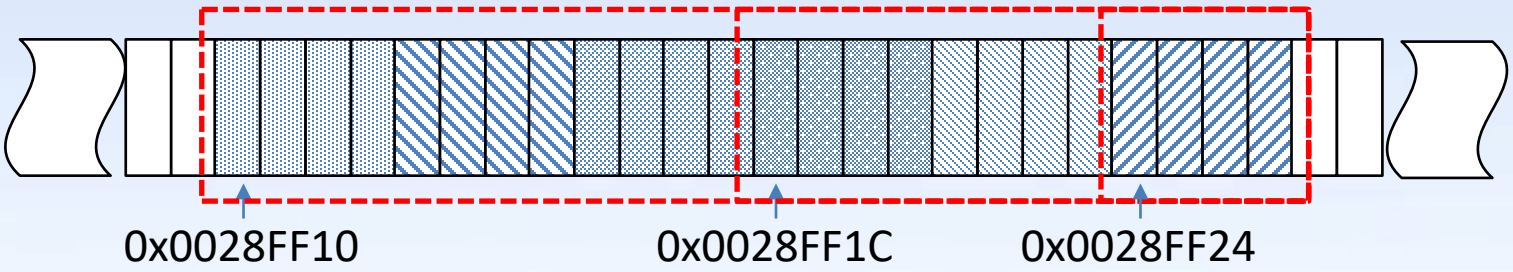
如何正确地理解数组及其操作

我们来看一个C语言标准中的示例，给定int x[3][5]，如何理解x[i][j]

Here x is a 3 × 5 array of objects of type int; more precisely, x is an array of three element objects, each of which is an array of five objects of type int. In the expression x[i], which is equivalent to (*((x)+(i))), x is first converted to a pointer to the initial array of five objects of type int. Then i is adjusted according to the type of x, which conceptually entails multiplying i by the size of the object to which the pointer points, namely an array of five int objects. The results are added and indirection is applied to yield an array of five objects of type int. When used in the expression x[i][j], that array is in turn converted to a pointer to the first of the objects of type int, so x[i][j] yields an int.

继续按照一维数组的方式计算地址偏移量然后定位对象
x[i][j]定位到了一个int类型的对象，其返回值类型就是int

a[1][2]=1的操作过程



1、识别基础表达式，包括a、1（等号左边）、2、1（等号右边）

2、a是lvalue，定位这24个字节内存（对象类型int[2][3]）

3、a是a[1]子表达式，没有跟&和sizeof结合，a取值: <0x0028FF10, int(*)[3]>

4、a[1]等价于*(a+1)，首先计算a+1: <0x0028FF1C, int(*)[3]>

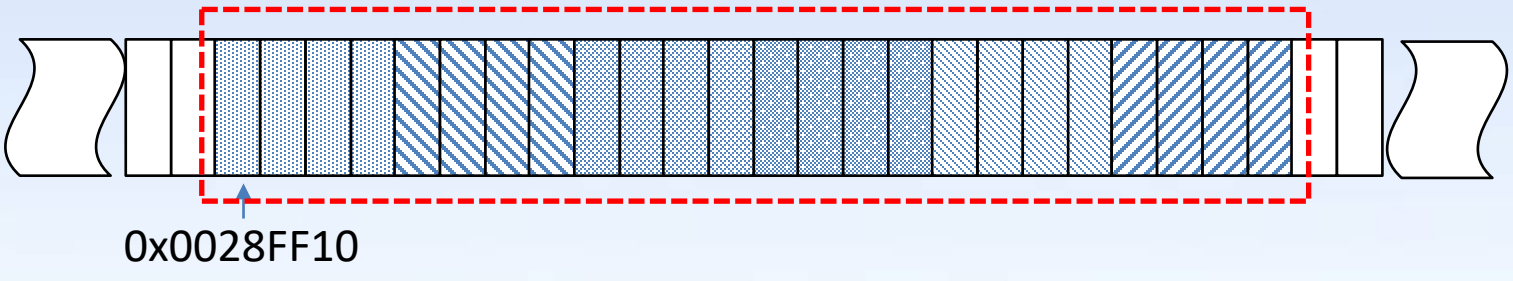
5、观察*(a+1)，这个表达式是lvalue，定位这12个字节内存（对象类型int[3]）

6、a[1]是a[1][2]子表达式，没有跟&和sizeof结合，a[1]取值: <0x0028FF1C, int*>

7、a[1][2]等价于*(a[1]+2)，首先计算a[1]+2: <0x0028FF24, int*>

8、观察*(a[1]+2)，这个表达式是lvalue，定位这4个字节内存（对象类型int）

(*(&a))[1][2] = 2的操作过程



1、识别基础表达式，包括a、1、2（等号左边）、2（等号右边）

2、a是lvalue，定位这24个字节内存（对象类型int[2][3]）

3、a和&结合，&a返回值: <0x0028FF10, int(*)[2][3]>

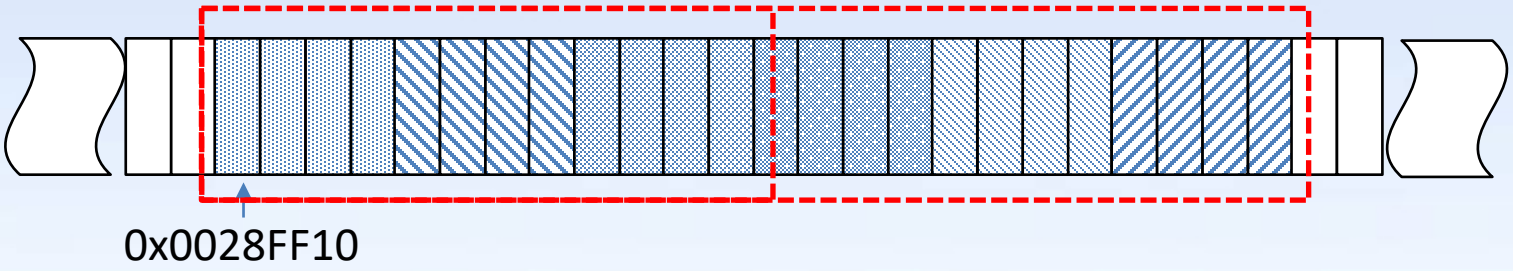
4、观察*(&a)，这个表达式是lvalue，定位这24个字节内存（对象类型int[2][3]）

6、*(&a)是(*(&a))[1]子表达式，没有跟&和sizeof结合， *(&a)取值: <0x0028FF10, int(*)[3]>

后续过程就跟之前a[1][2]一样了

a、*(&a)、 (*(&a))[1]、 (*(&a))[1][2]都是lvalue

(&(*a))[1][2] = 3的操作过程



1、识别基础表达式，包括a、1、2 、3

2、a是lvalue，定位这24个字节内存（对象类型int[2][3]）

3、a是*a的子表达式，没有跟&和sizeof结合，a取值: <0x0028FF10, int(*)[3]>

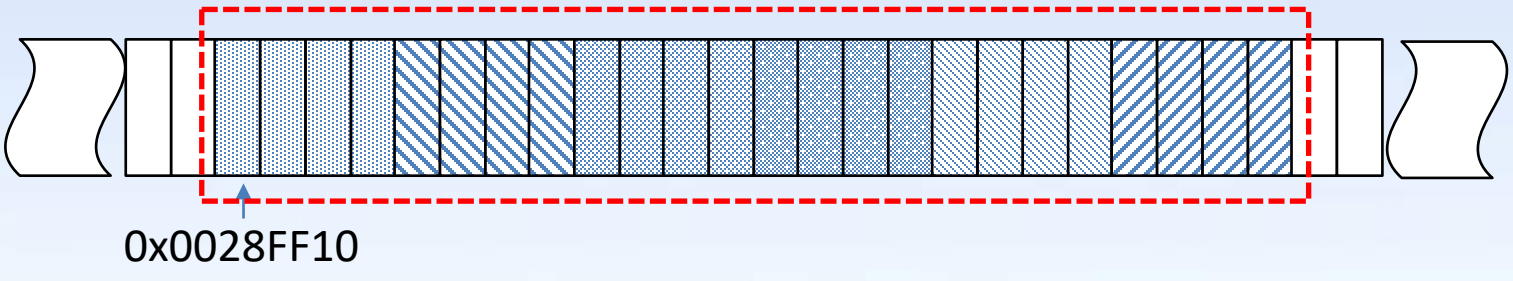
4、*a这个表达式是lvalue，定位这12个字节内存（对象类型int[3]）

6、*a是&(*a)子表达式，&(*a)取值: <0x0028FF10, int(*)[3]>

后续过程就跟之前a[1][2]一样了

a、*a、 (&(*a))[1]、 (&(*a))[1][2]都是lvalue

(a+1-1)[1][2] = 4的操作过程



1、识别基础表达式，包括a、1、1（a+1-1中的两个1）、1、2 、4

2、a是lvalue，定位这24个字节内存（对象类型int[2][3]）

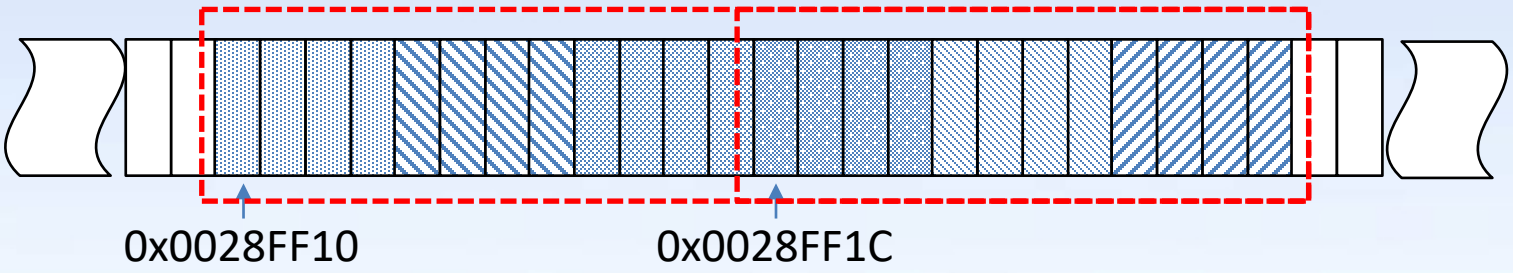
3、a是a+1-1的子表达式，没有跟&和sizeof结合，a取值: <0x0028FF10, int(*)[3]>

4、计算a+1-1表达式的值，结果是<0x0028FF10, int(*)[3]>

后续过程就跟之前a[1][2]一样了

a、(a+1-1)[1]、 (a+1-1)[1][2]都是lvalue

1[a][2] = 5的操作过程



1、识别基础表达式，包括1、a、2、5

2、1是常量表达式，取值<1, int>

3、1[a]等价于*(1+a)， a是lvalue，定位这24个字节内存（对象类型int[2][3]）

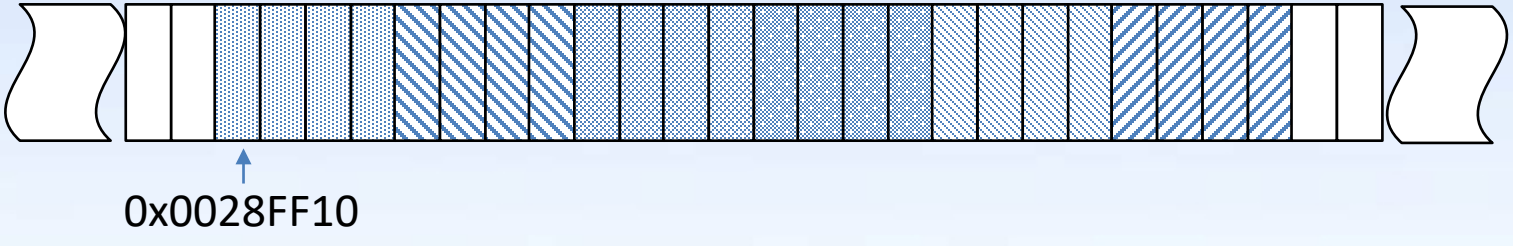
4、计算1+a， 返回值：<0x0028FF1C, int(*)[3]>

5、观察*(1+a)，这个表达式是lvalue，定位这12个字节内存（对象类型int[3]）

后续过程就跟之前a[1][2]一样了

a、1[a] 、1[a][2]都是lvalue

1[2][a] = 6的操作过程



1、识别基础表达式，包括1、2、a、6

2、1、2是常量表达式，取值<1, int>，<2, int>

3、1[2]等价于*(1+2)

4、1+2的返回值是<3, int>，不是一个有效指针类型，无法跟*结合

报错

总结：数组名是什么

数组名是一个Identifier，是一个标识符，是一个lvalue表达式

数组名不是一个指针，也没有特殊性，背后的机制就是表达式的evaluate

理解这个机制，才能理解为什么数组参数第一维大小会丢失

int a[2][3]; a++出错是因为a是无法修改的lvalue，而不是因为a是常量

数组名是lvalue，但不可以放到等号左边

思考题

int n[2][3][4][5]={0}，以下表达式返回值是什么？

&n, &n+1
n, n+1
n[0], n[0]+1
n[0][0], n[0][0]+1
n[0][0][0], n[0][0][0]+1
n[0][0][0][0], n[0][0][0][0]+1

假设变量**n**对应的内存首地址为0x00FF3811
返回值表示为**<Value, Value_Type>**形式

线上同学发邮件**w.rong@buaa.edu.cn**

&n: <0x00FF3811, int(*)[2][3][4][5]>
&n+1: <0x00FF39F1, int(*)[2][3][4][5]>

n:<0x00FF3811, int(*)[3][4][5]>
n+1: <0x00FF3901, int(*)[3][4][5]>

n[0]: <0x00FF3811, int(*)[4][5]>
n[0]+1: <0x00FF3861, int(*)[4][5]>

n[0][0]: <0x00FF3811, int(*)[5]>
n[0][0]+1: <0x00FF3825, int(*)[5]>

n[0][0][0]: <0x00FF3811, int*>
n[0][0][0]: <0x00FF3815, int*>

n[0][0][0][0]: <0, int>
n[0][0][0][0]: <1, int>