简单回顾一下：Evaluation of Expression

In the abstract machine, all expressions are evaluated as specified by the semantics.

表达式需要根据语法的规定来进行evaluate

Evaluation的内涵

给定一个表达式（Expression)，Evaluation过程包括：

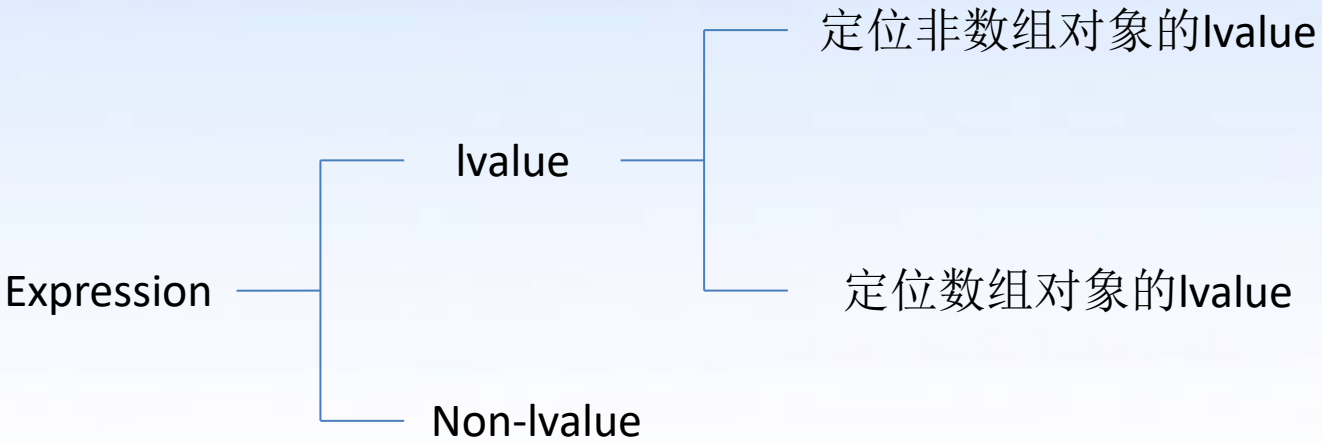
1、Value Computation （求值）

2、Initiation of Side Effect（确定副作用）

表达式的求值会有一个rvalue，以及rvalue的类型
rvalue在是表达式求值的结果

Side Effect:也就是会对某个对象的内容进行修改

不同表达式Evaluate的异同



定位非数组对象的lvalue的Evaluate规则

给定一个能定位非数组对象的lvalue表达式exp，如果这个表达式

1、跟sizeof结合，例如：sizeof(exp)，或sizeof exp

2、跟typeof结合，例如：typeof(exp)，typeof_unqual(exp)

3、跟&结合，例如：&exp

4、跟一元运算符++/--和后缀运算符++/--结合，例如：++exp/--exp/exp++/exp—

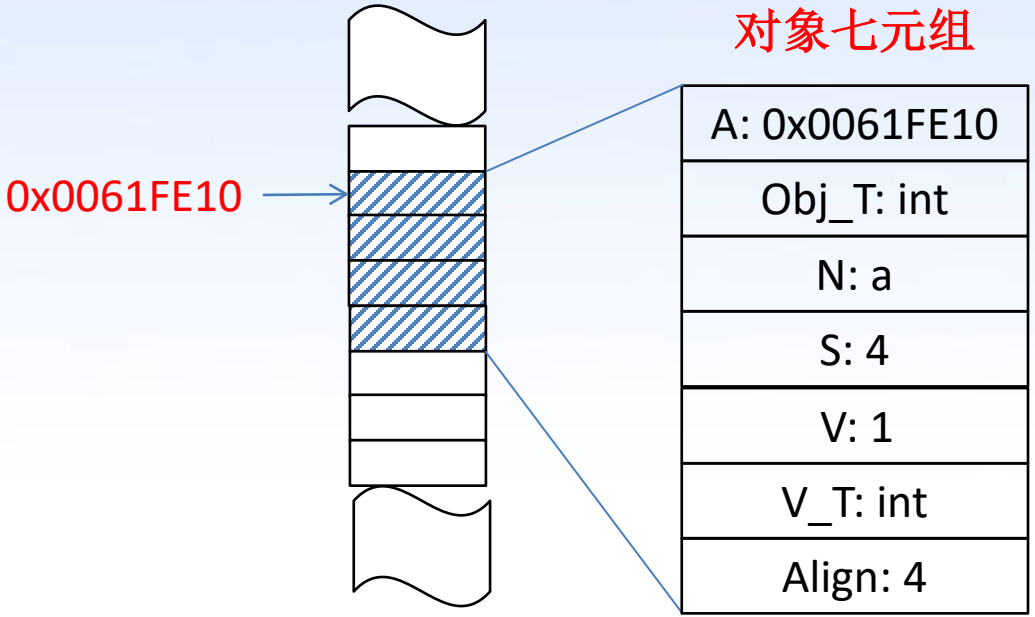
5、如果lvalue定位的对象类型是结构体/联合体，跟.结合，例如：exp.

6、出现在赋值运算符的左侧，例如：exp =

除了以上6种情况，这个表达式都要做evaluate

定位非数组对象lvalue的evaluate示例

int a = 1;



a; → 对a做evaluate

sizeof(a);

typeof(a);

&a;

a++;

a--;

++a;

--a;

a = 2;

对a不做evaluate

这些a相关的表达式是如何evaluate的呢？

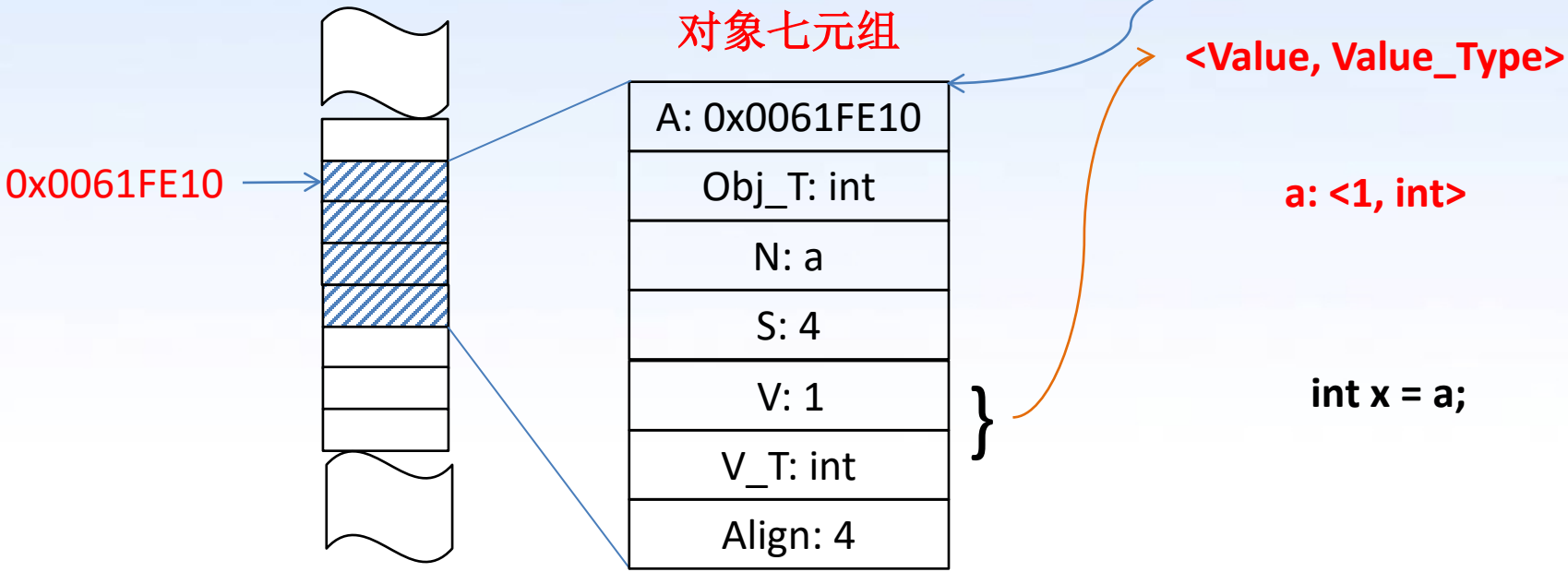
typeof(a)不是表达式

结构体/联合体的.操作后续讲解

当a被evaluate的时候

int a = 1;

a;

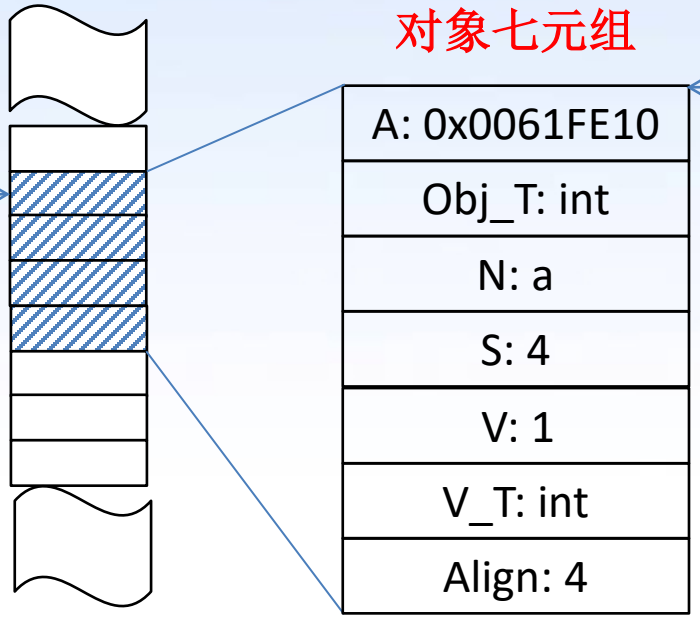


sizeof(a)或sizeof a

int a = 1;

sizeof(a)

对象七元组



0x0061FE10

<size, size_t>

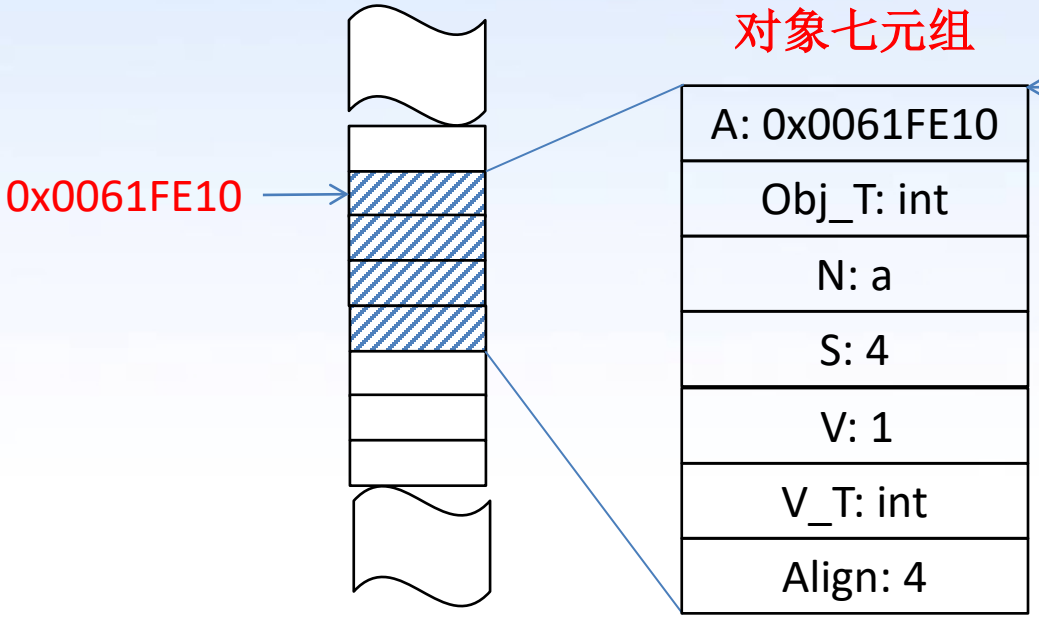
sizeof(a): <4, size_t>

注意rvalue的类型size_t

size_t x=sizeof(a);

typeof(a)

int a = 1;



typeof(a)

等价于**Obj_T**

typeof(a)等价于int

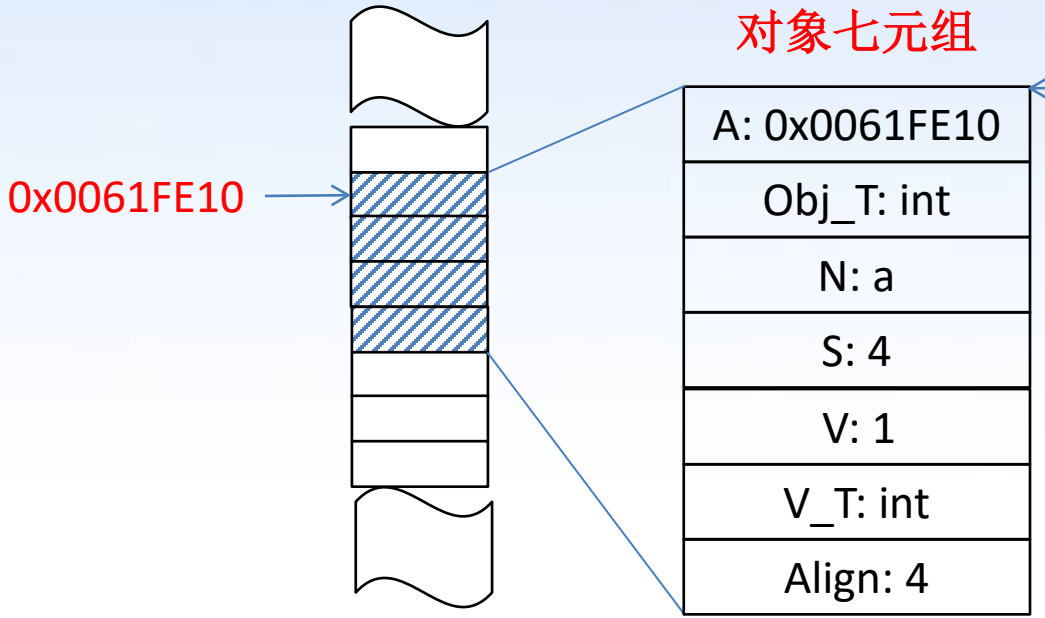
typeof(a)不是表达式

typeof(a) x; 等价于 **int x;**

sizeof(typeof(a)); 等价于 **sizeof(int);**

&a

int a = 1;



&a

<Address, Obj_T*>

&a: <0x0061FE10, int*>

注意rvalue的类型Obj_T*

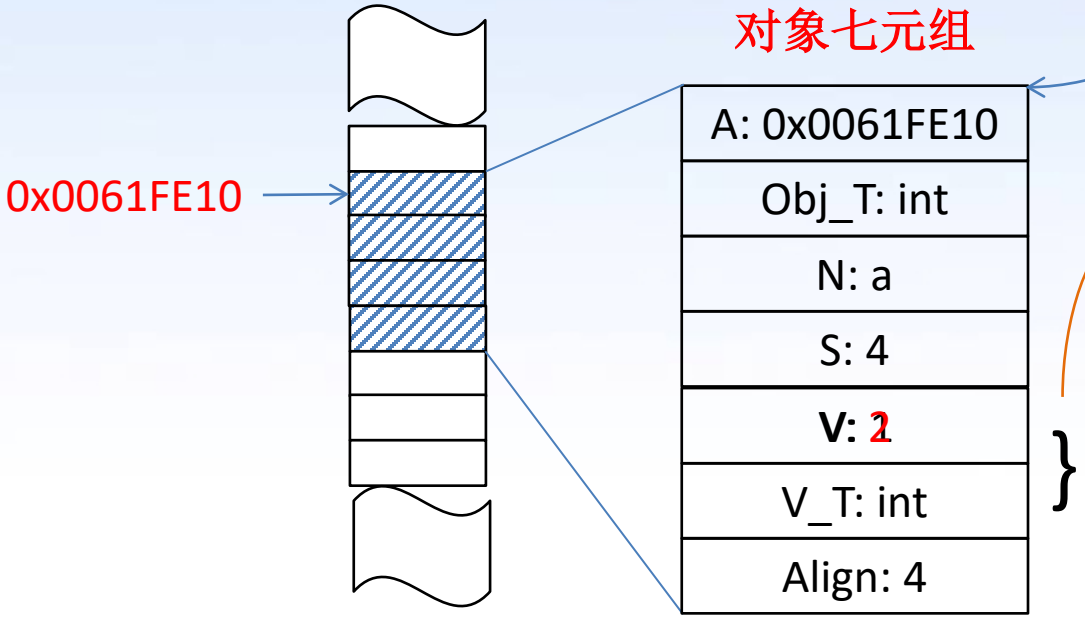
int* x=&a;

++a

int a = 1;

++a

对象七元组



<Value incremented by 1, V_T>

++a: <2, int>

++的语义和V_T相关

该表达式有副作用
即将Value+1

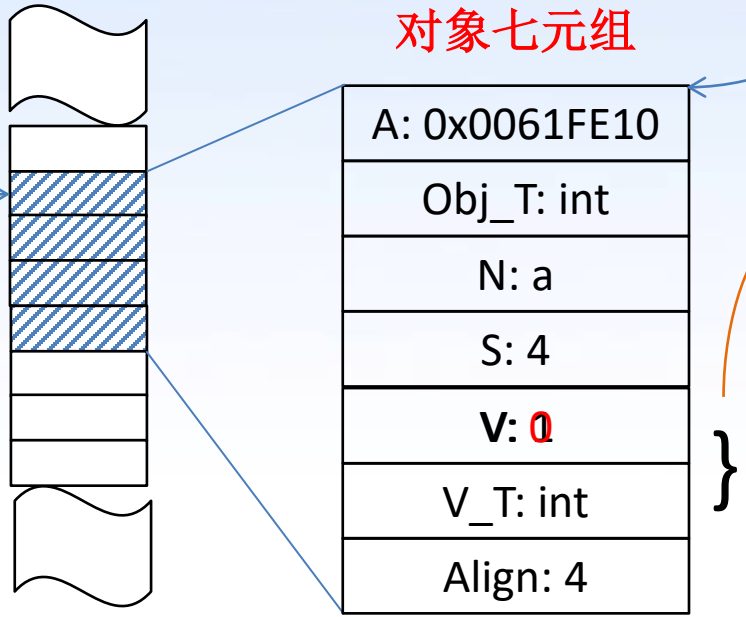
int x = ++a;

--a

int a = 1;

--a

对象七元组



<Value decremented by 1, V_T>

--a: <0, int>

--的语义和V_T相关

该表达式有副作用

即将Value-1

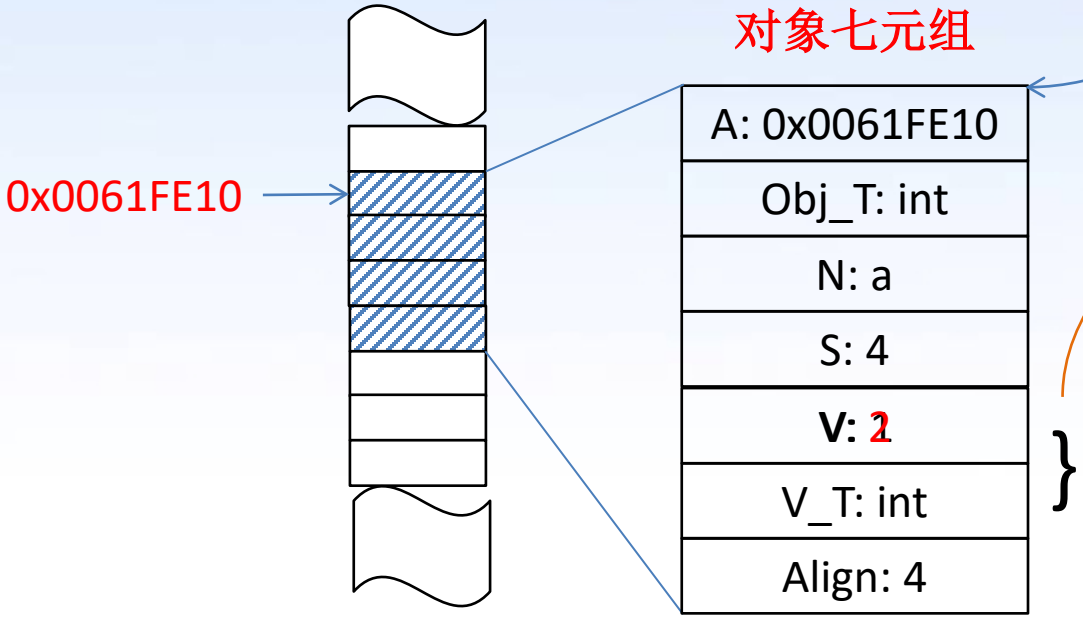
int x = --a;

a++

int a = 1;

a++

对象七元组



<Value, V_T>

a++: <1, int>

++的语义和V_T相关

该表达式有副作用
即将Value+1

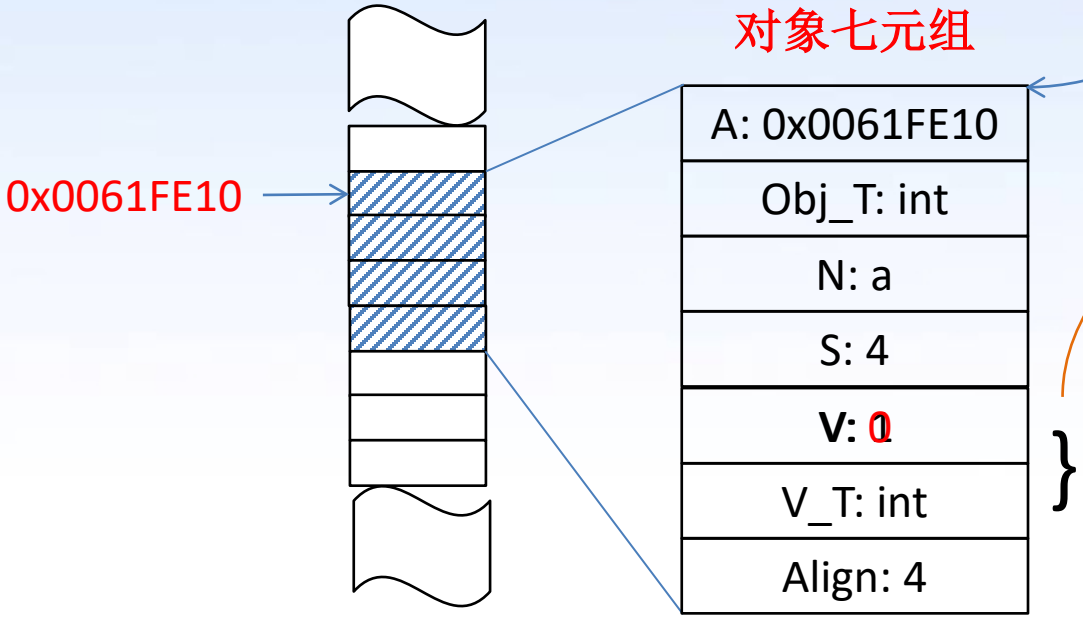
int x = a++;

a--

int a = 1;

a--

对象七元组



<Value, V_T>

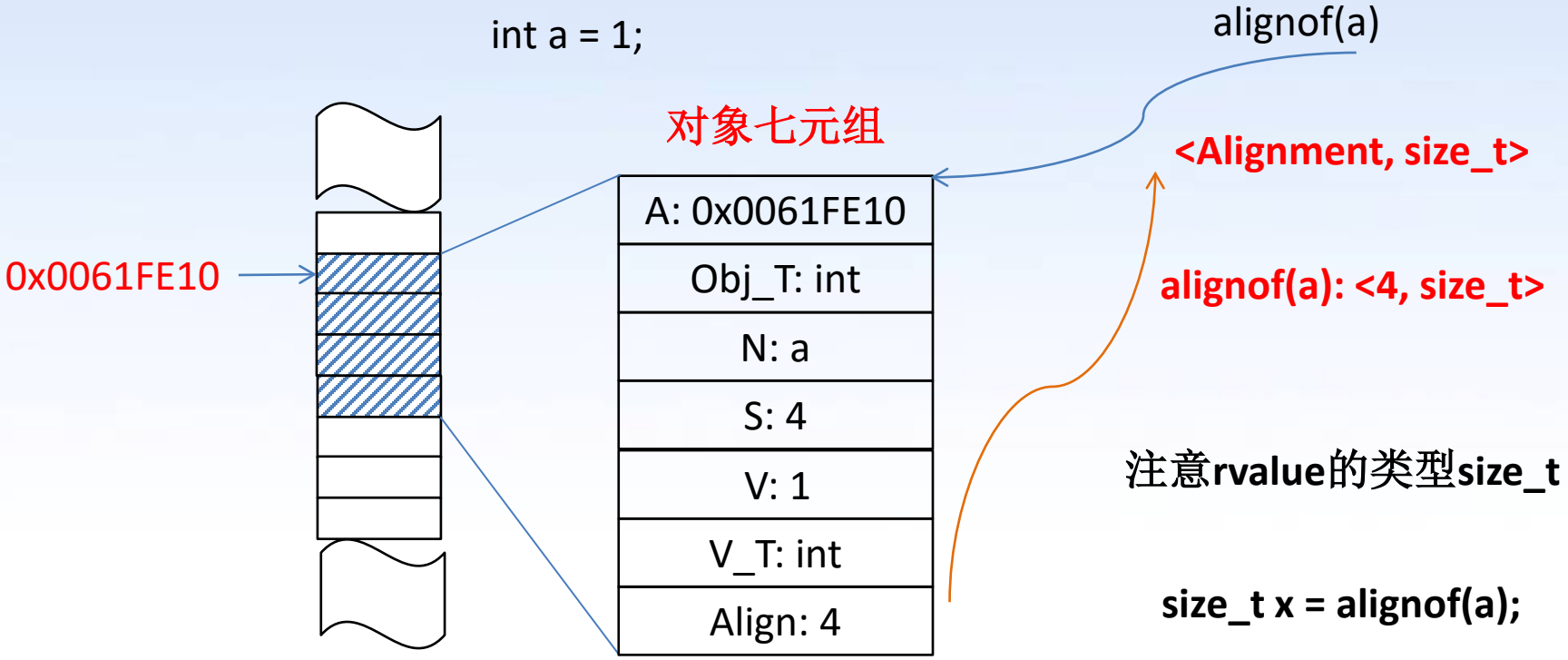
a--: <1, int>

--的语义和V_T相关

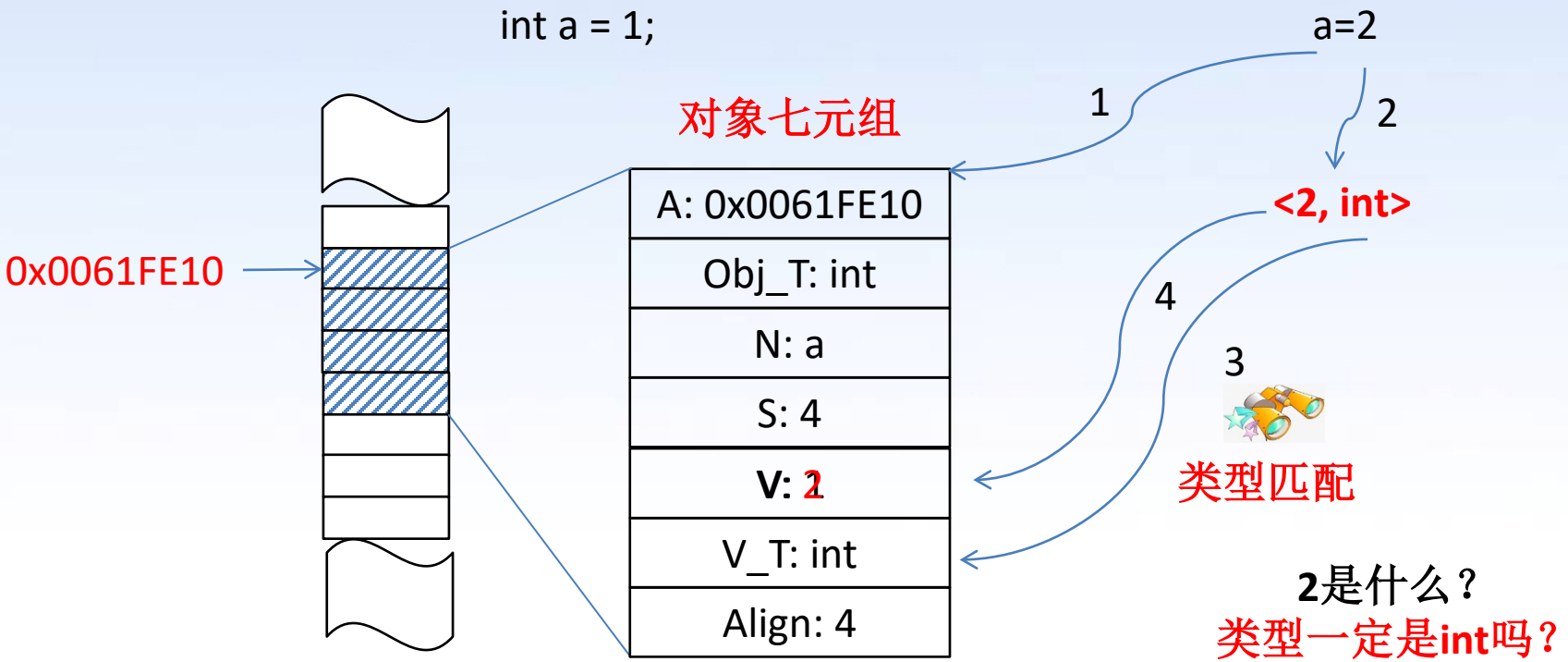
该表达式有副作用
即将Value-1

int x = a--;

alignof(a)或alignof a（标准并不支持）



a=2


int a = 1;
a=2
对象七元组
1
2
A: 0x0061FE10
Obj_T: int
N: a
S: 4
V: 1
V_T: int
Align: 4
0x0061FE10
<2, int>
4
3
类型匹配
2是什么？
类型一定是int吗？

整数常量

Suffix	Decimal Constant	Octal, Hexadecimal or Binary Constant

none	`int`
	`long int`
	`long long int`
		`int`
		`unsigned int`
		`long int`
		`unsigned long int`
		`long long int`
		`unsigned long long int`

`u` or `U`	`unsigned int`
	`unsigned long int`
	`unsigned long long int`
		`unsigned int`
		`unsigned long int`
		`unsigned long long int`

`l` or `L`	`long int`
	`long long int`
		`long int`
		`unsigned long int`
		`long long int`
		`unsigned long long int`

Both `u` or `U` and `l` or `L`	`unsigned long int` `unsigned long long int`	`unsigned long int` `unsigned long long int`
`ll` or `LL`	`long long int`	`long long int` `unsigned long long int`
Both `u` or `U` and `ll` or `LL`	`unsigned long long int`	`unsigned long long int`
`wb` or `WB`	`_BitInt(N)` where the width `N` is the smallest `N` greater than 1 which can accommodate the value and the sign bit.	`_BitInt(N)` where the width `N` is the smallest `N` greater than 1 which can accommodate the value and the sign bit.
Both `u` or `U` and `wb` or `WB`	`unsigned _BitInt(N)` where the width `N` is the smallest `N` greater than 0 which can accommodate the value.	`unsigned _BitInt(N)` where the width `N` is the smallest `N` greater than 0 which can accommodate the value.

The type of an integer constant is the first of the corresponding list in which its value can be represented.

2这个表达式rvalue的类型为什么是int

2147483648这个表达式的rvalue类型是什么？

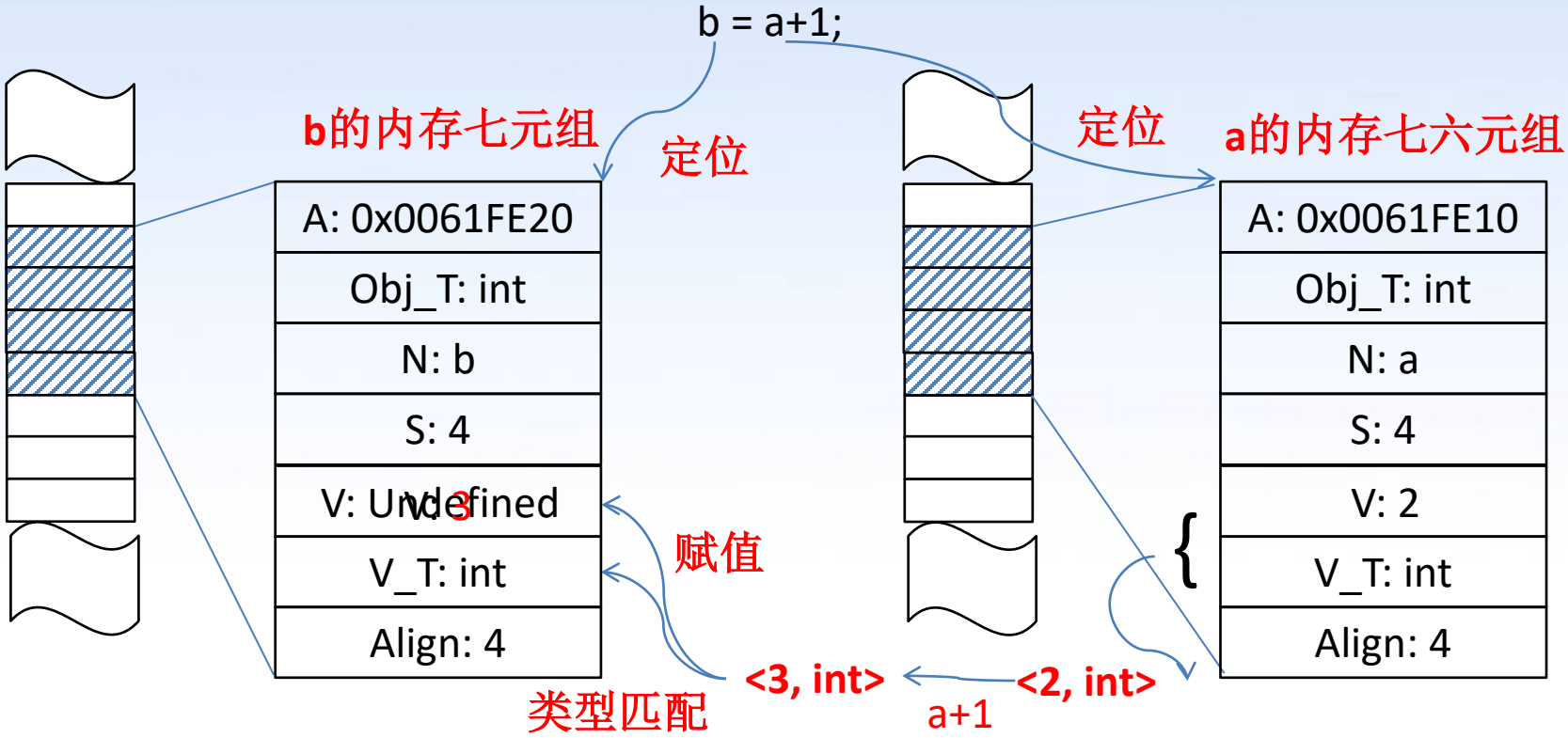
2L这个表达式的rvalue的类型是什么?

看一个例子：int b；b=a+1；

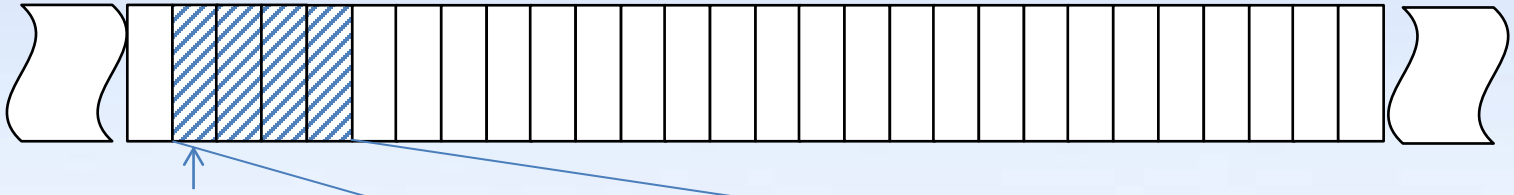
观察这个表达式 b = a+1;

lvalue有？	a，b
基础表达式有？	a，b，1
哪些表达式做evaluate？	a，1，a+1，b=a+1
表达式b为什么不做evaluate?	b是定位非数组对象的lvalue，在等号左边
哪个表达式有副作用	b = a + 1;

看一个例子：int b；b=a+1；



定位结构体对象的lvalue表达式的evaluate



0x0062FE10

```
struct m_struct {
    int a;
} h = {1};
```

A: 0x0062FE10
Obj_T: struct m_struct
N: h
S: 4
V: Defined
V_T: struct m_struct
Align: 4

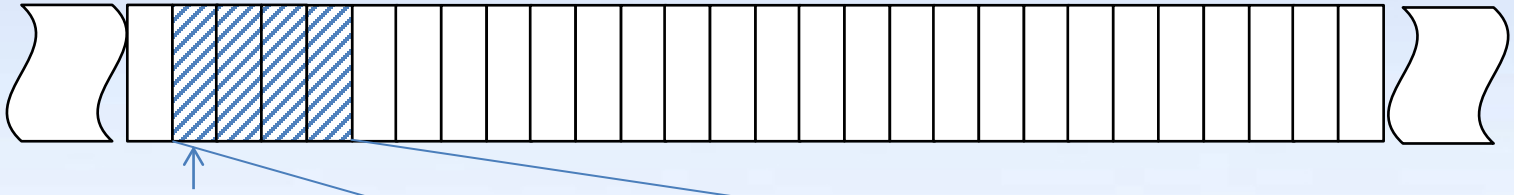
h; 做evaluate，值是什么？

sizeof(h); <4, size_t>
typeof(h); strut m_struct
&h; <0x0062FE10, struct m_struct*>

h++/--, ++/--h 是否可行？

h = xxx; h能否放在等号左边？

定位结构体对象的lvalue表达式的evaluate



0x0062FE10

```
struct m_struct {
    int a;
} h = {1};
```

观察h.a

A: 0x0062FE10
Obj_T: int
N: h.a
S: 4
V: 1
V_T: int
Align: 4

h.a; h做不做evaluate，为什么？

sizeof(h.a); <4, size_t>
typeof(h.a); int
&h.a; <0x0062FE10, int*>

h.a++/--, ++/--h.a 是否可行？

h.a = xxx; h.a能否放在等号左边？

定位数组对象的lvalue的Evaluate规则

给定一个能定位数组对象的lvalue表达式exp，如果这个表达式

1、跟sizeof结合，例如：sizeof(exp)，或sizeof exp

2、跟typeof结合，例如：typeof(exp)，typeof_unqual(exp)

3、跟&结合，例如：&exp

4、如果lvalue定位的是一个string literal，且用于初始化一个字符数组，例如：

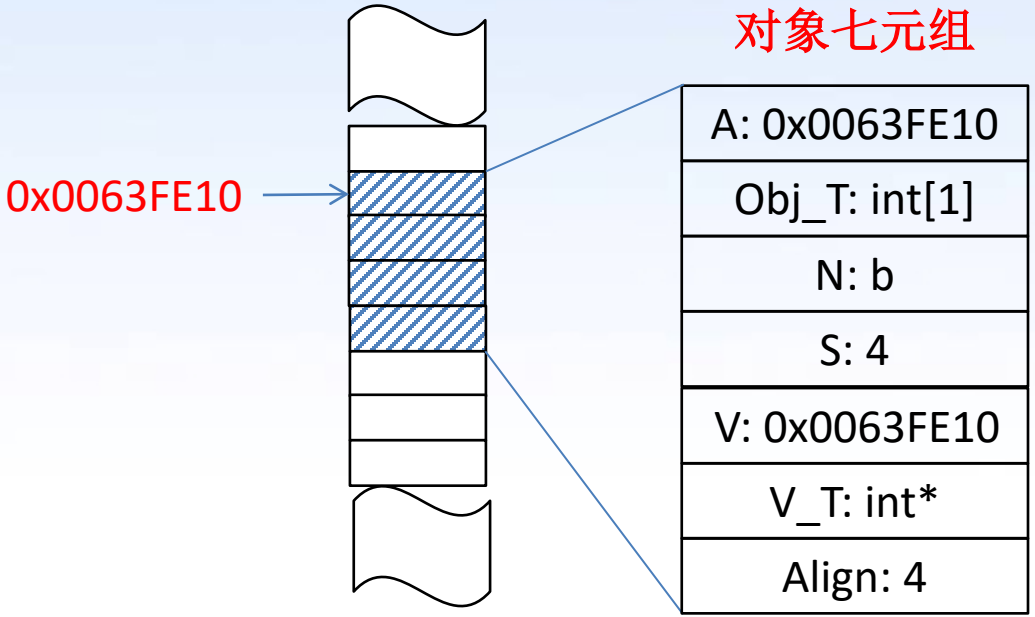
char str[] = “hello”，这个hello是一个lvalue，并初始化字符数组str

除了以上4种情况，这个表达式都要做evaluate

++/--， =为什么不考虑？

定位数组对象lvalue的evaluate示例

int b[1] = {1};



b; → 对b做evaluate

sizeof(b);
typeof(b);
&b;
→ 对b不做evaluate

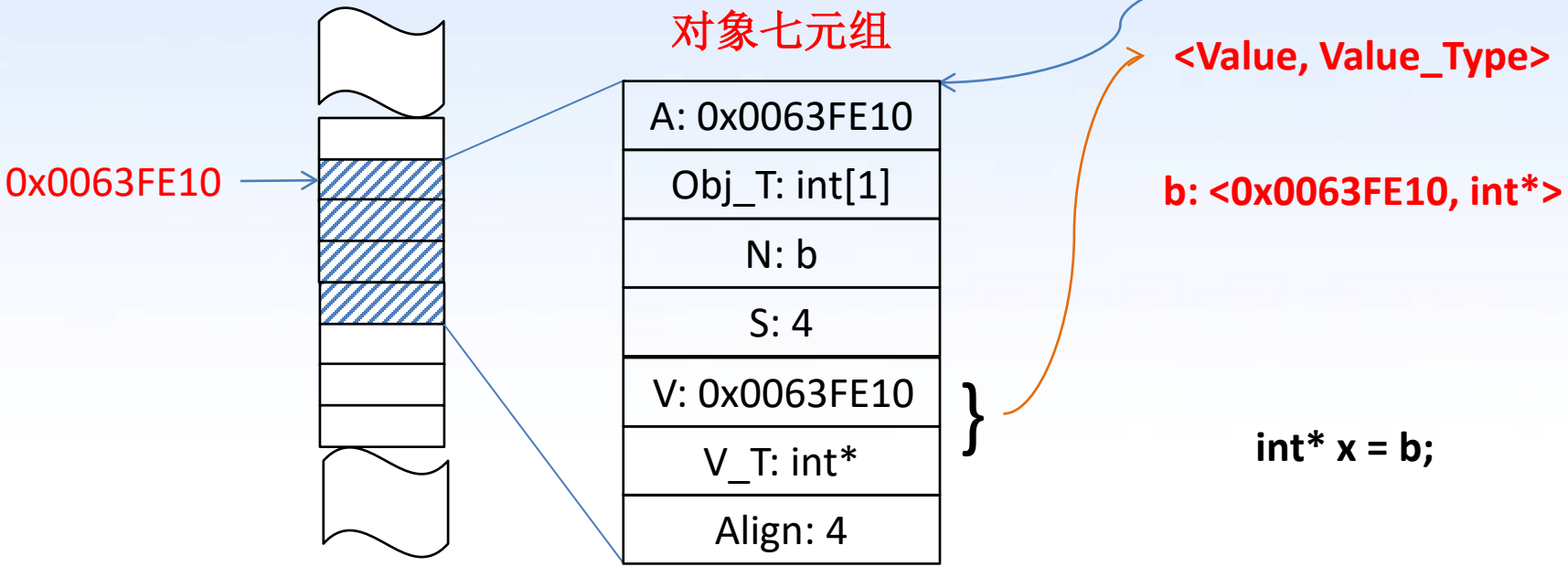
这些b相关的表达式是如何evaluate的呢？

typeof(b)不是表达式

当b被evaluate的时候

int b[1] = {1};

b;

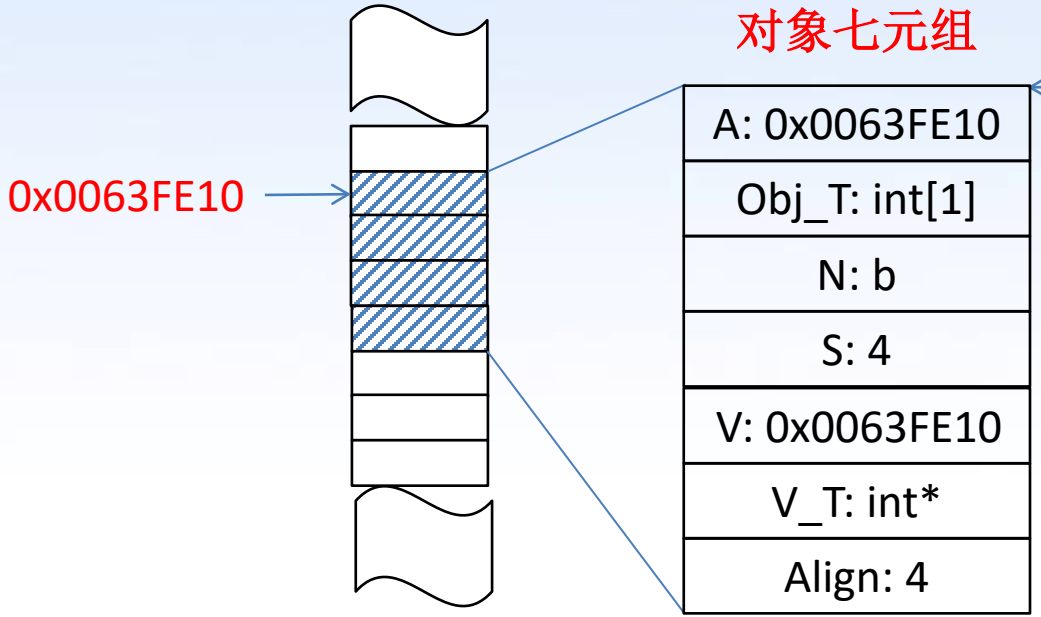


sizeof(b)或sizeof b

int b[1] = {1};

sizeof(b)

对象七元组



<size, size_t>

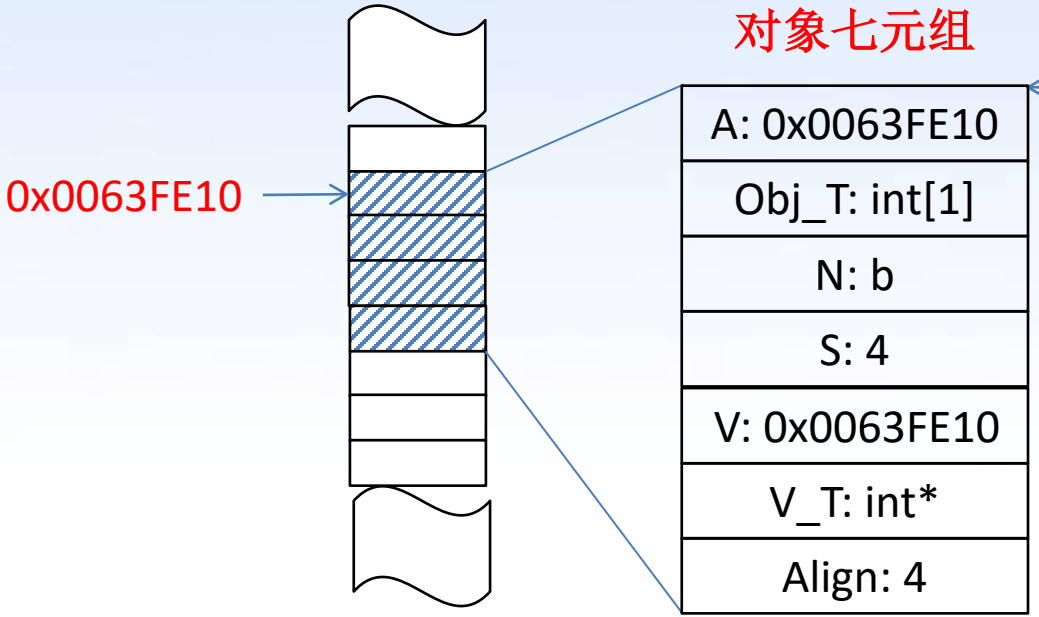
sizeof(b): <4, size_t>

注意rvalue的类型size_t

size_t x = sizeof(b);

typeof(b)

int b[1] = {1};



typeof(b)

等价于**Obj_T**

typeof(b)等价于int[1]

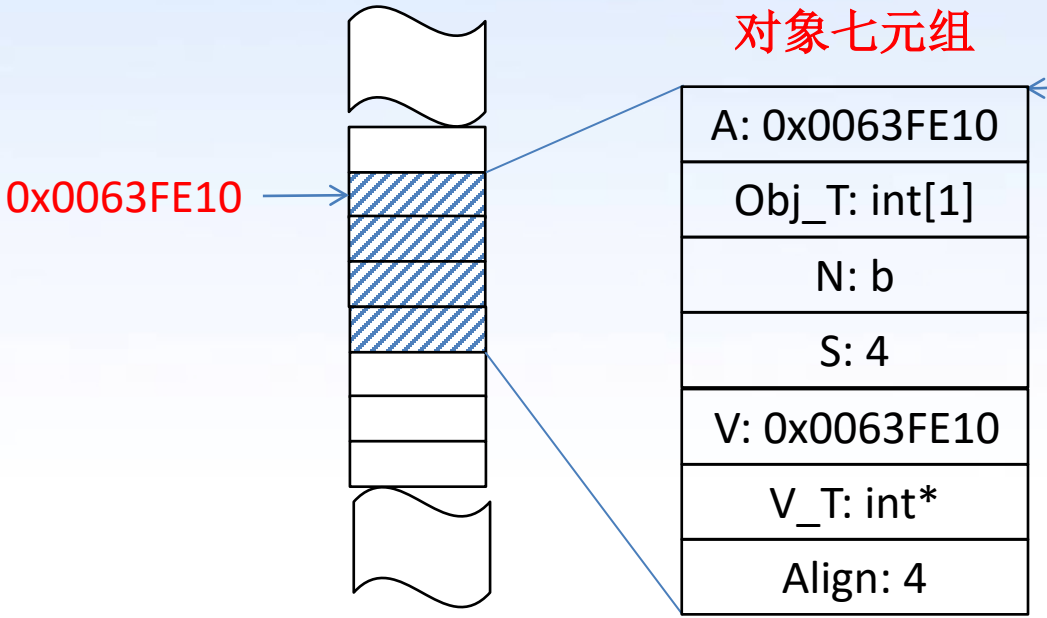
typeof(b)不是表达式

typeof(b) x; 等价于 **int x[1];**

sizeof(typeof(b)); 等价于 **sizeof(int[1]);**

&b

int b[1] = {1};



&b

<Address, Obj_T*>

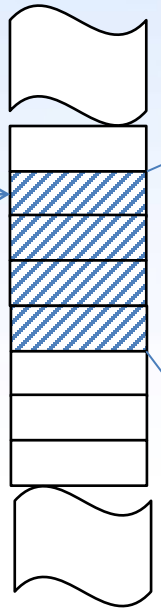
&a: <0x0063FE10, int(*)[1]>

注意rvalue的类型Obj_T*

int (*x)[1] = &b;

alignof(b)或alignof b（标准并不支持）

int b[1] = {1};



0x0063FE10

对象七元组

A: 0x0063FE10
Obj_T: int[1]
N: b
S: 4
V: 0x0063FE10
V_T: int*
Align: 4

alignof(b)

<Alignment, size_t>

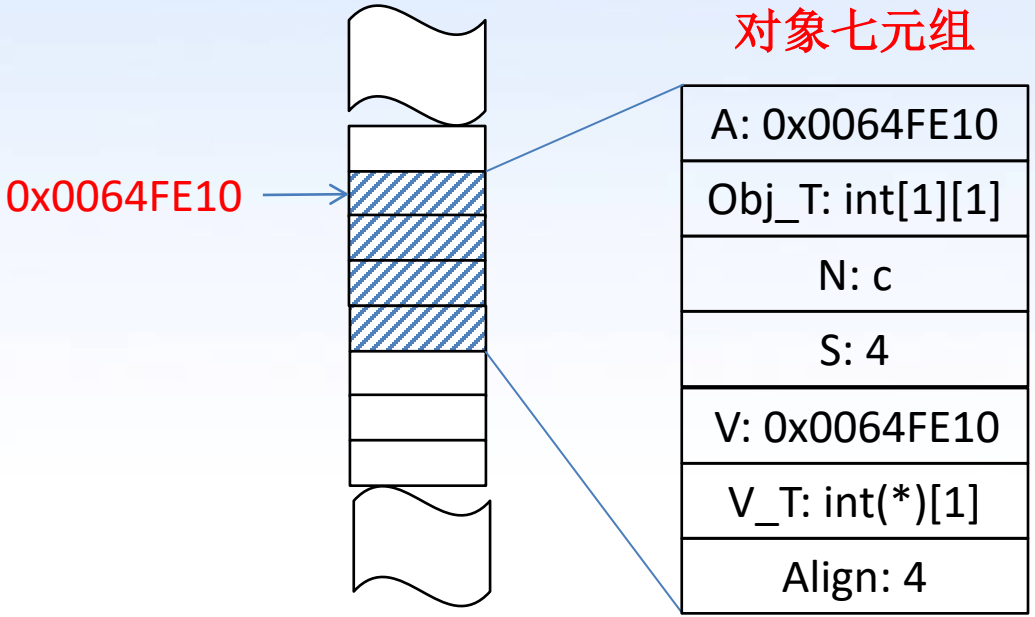
alignof(b): <4, size_t>

注意rvalue的类型size_t

size_t x = alignof(b);

定位高维数组对象lvalue的evaluate示例

int c[1][1] = {1};



c; → 对c做evaluate

sizeof(c);
typeof(c);
&c;
→ 对c不做evaluate

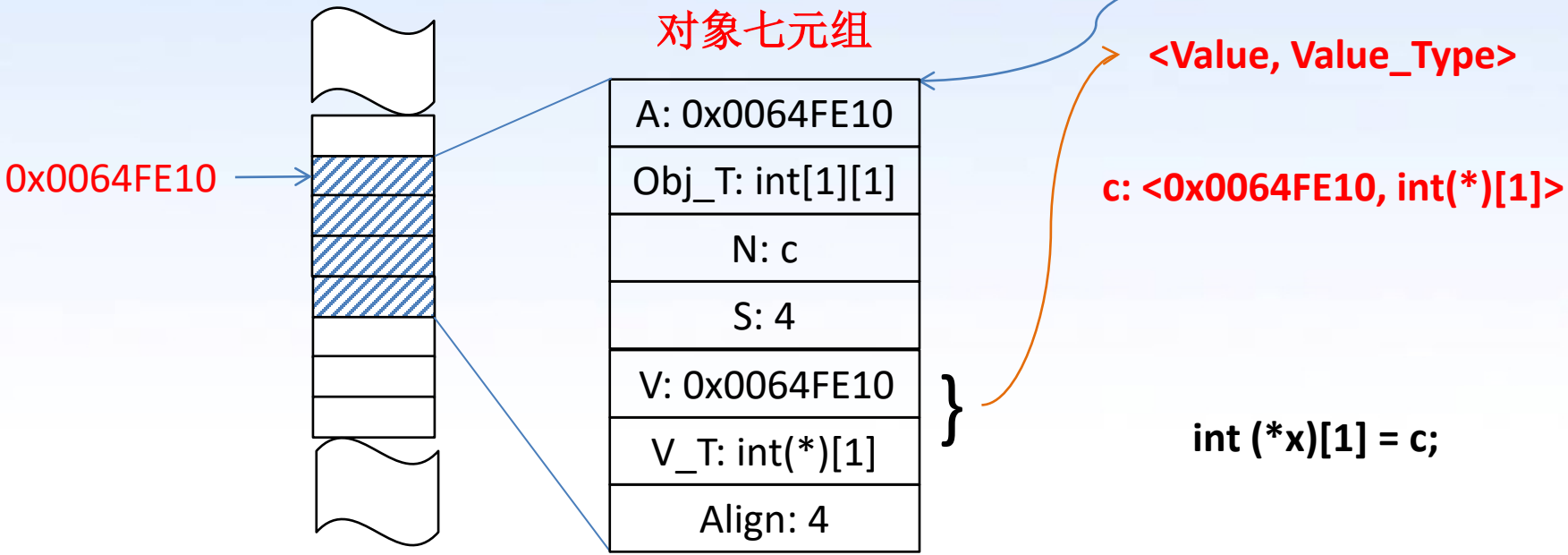
这些c相关的表达式是如何evaluate的呢？

typeof(c)不是表达式

当c被evaluate的时候

int c[1][1] = {1};

c;

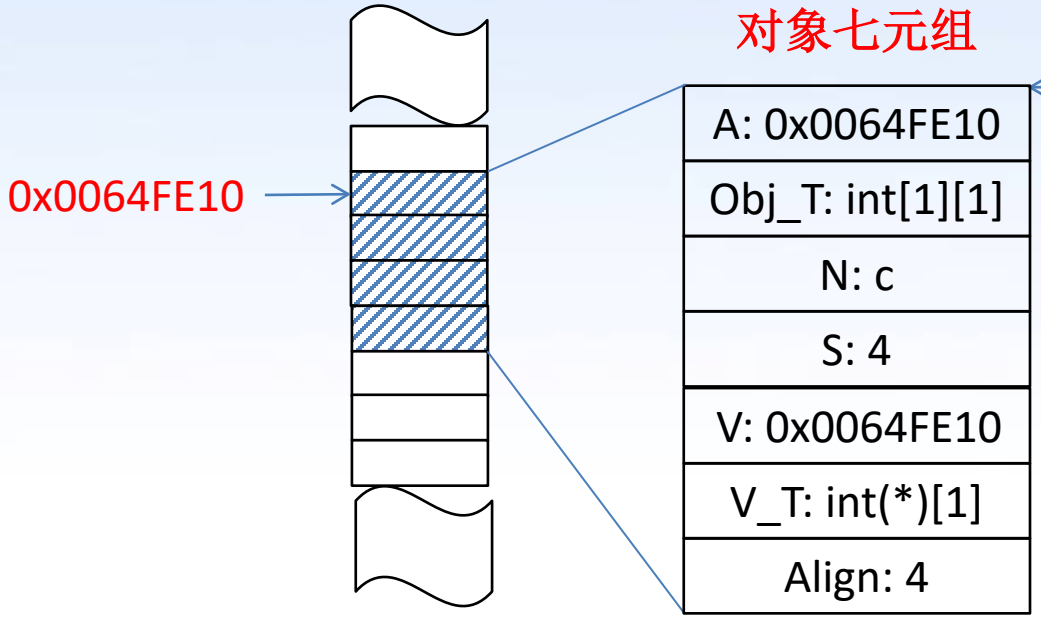


sizeof(c)或sizeof c

int c[1][1] = {1};

sizeof(c)

对象七元组



<size, size_t>

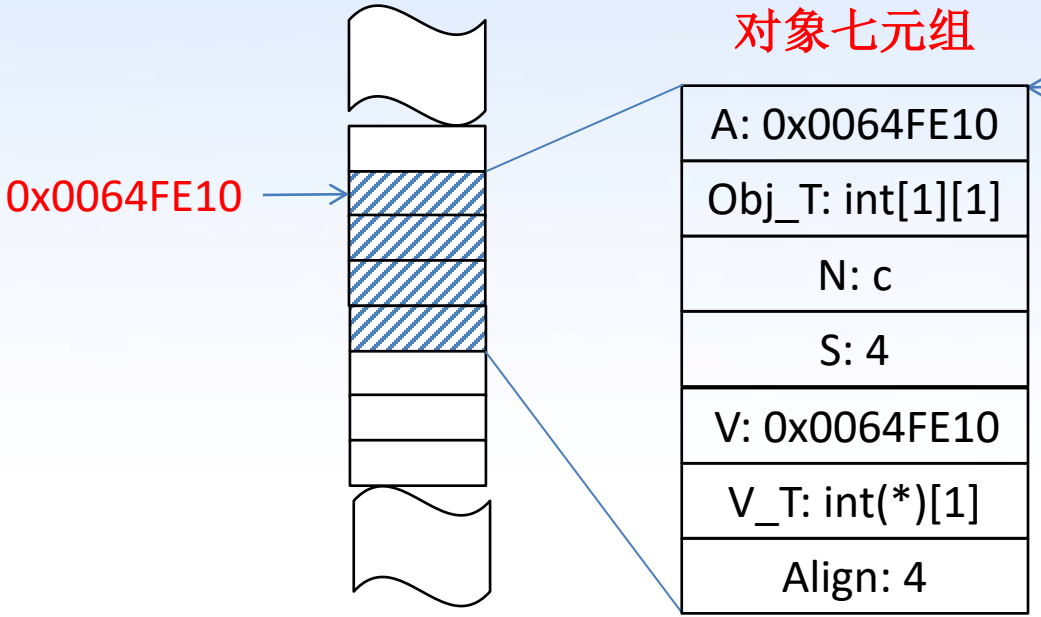
sizeof(b): <4, size_t>

注意rvalue的类型size_t

size_t x = sizeof(c);

typeof(c)

int c[1][1] = {1};



typeof(c)

等价于**Obj_T**

typeof(c)等价于int[1][1]

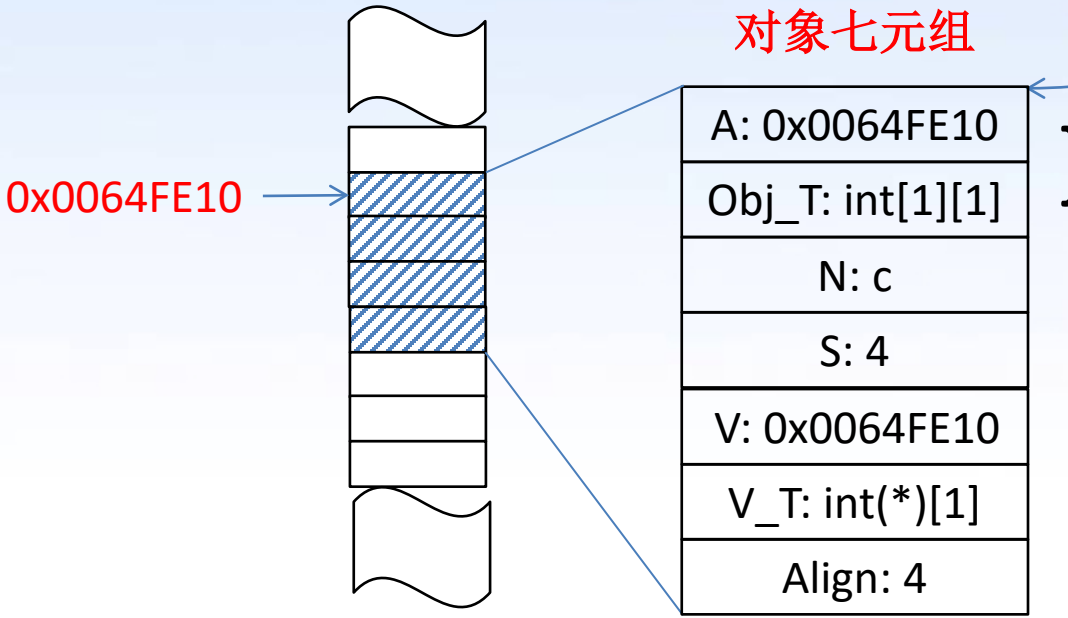
typeof(c)不是表达式

typeof(c) x; 等价于 **int x[1][1];**

sizeof(typeof(c)); 等价于 **sizeof(int[1][1]);**

&c

int c[1][1] = {1};



&c

<Address, Obj_T*>

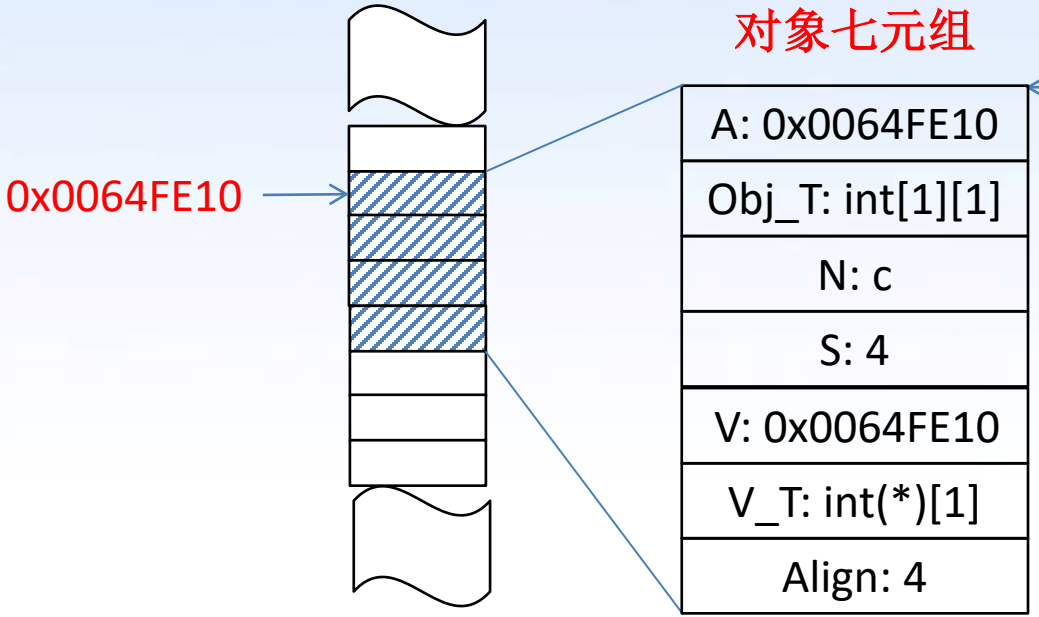
&a: <0x0064FE10, int(*)[1][1]>

注意rvalue的类型Obj_T*

int (*x)[1][1] = &c;

alignof(c)或alignof c（标准并不支持）

int c[1][1] = {1};



alignof(c)

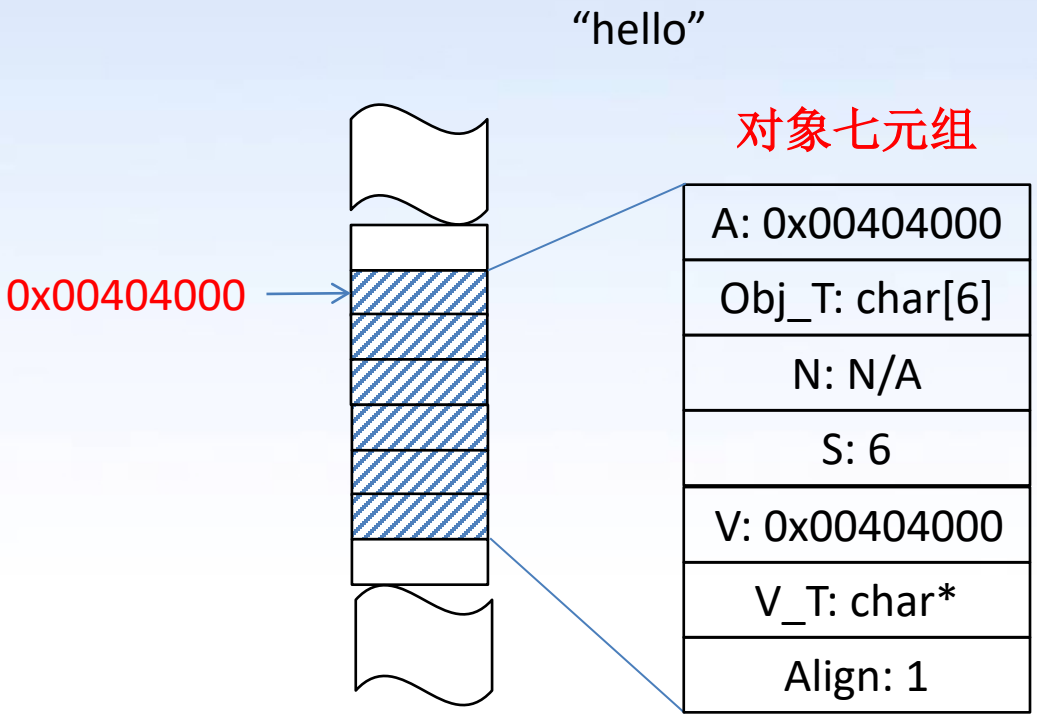
<Alignment, size_t>

alignof(b): <4, size_t>

注意rvalue的类型size_t

size_t x = alignof(c);

定位字符串对象lvalue的evaluate示例



“hello”; → 对”hello”做evaluate

sizeof(“hello”);

typeof(“hello”);

&”hello”;

char str[]=“hello”

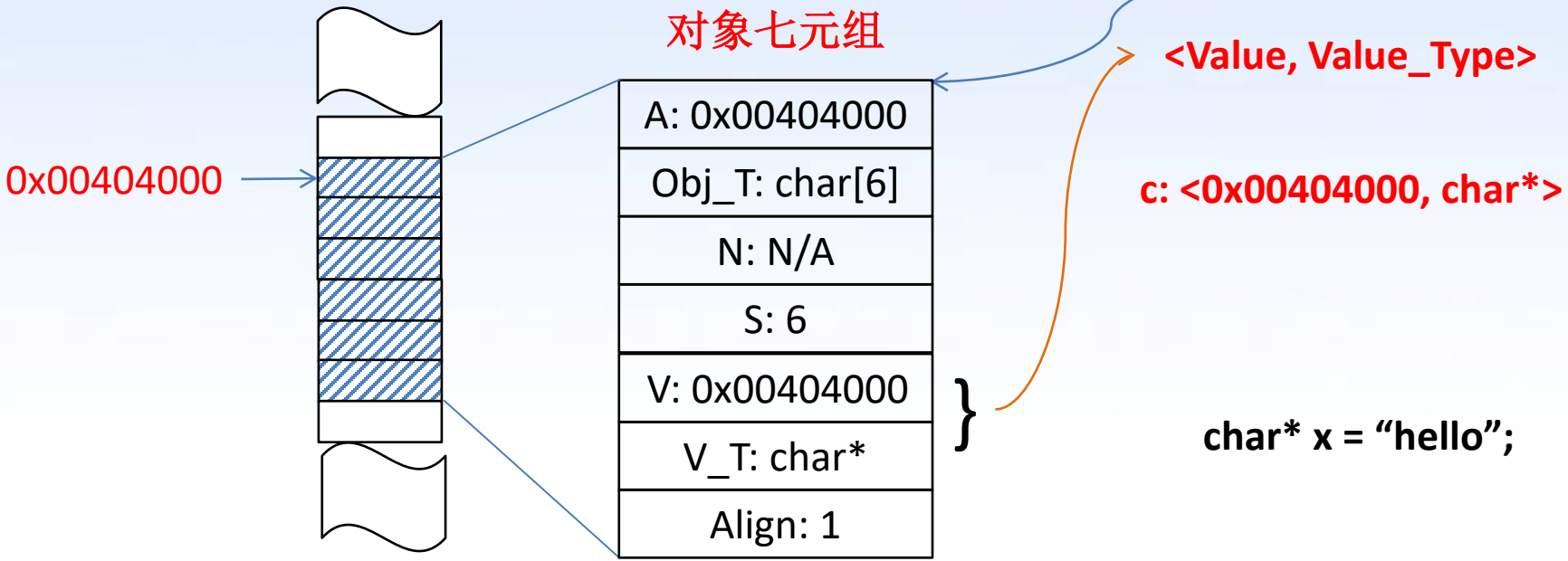
对”hello”不做evaluate

这些”hello”相关的表达式是如何evaluate的呢？

typeof(“hello”)不是表达式

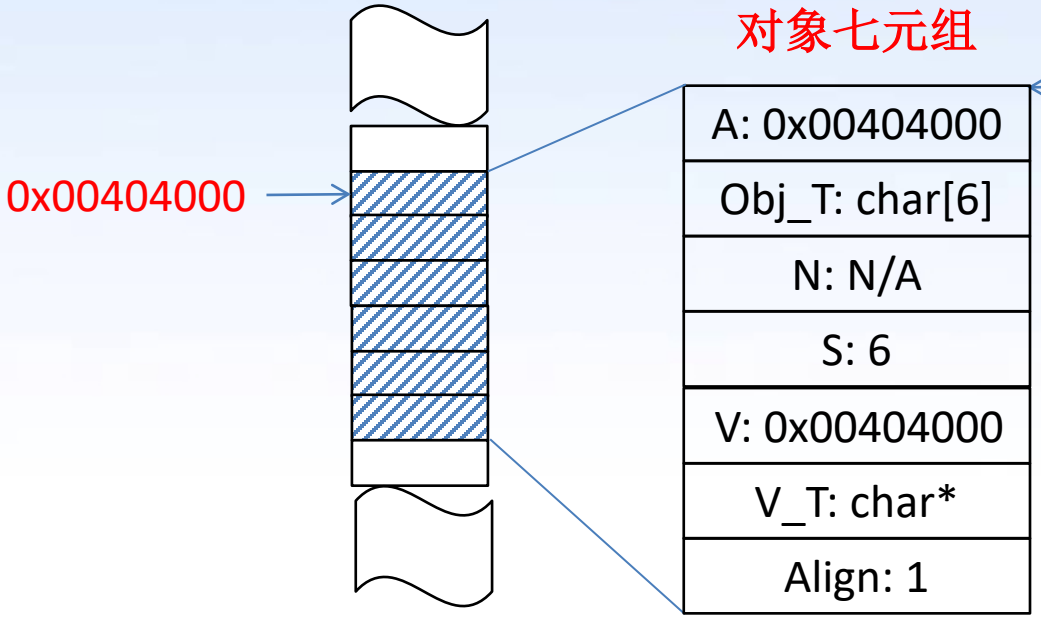
当“hello”被evaluate的时候

“hello”;



sizeof(“hello”)或sizeof “hello”

sizeof(“hello”)



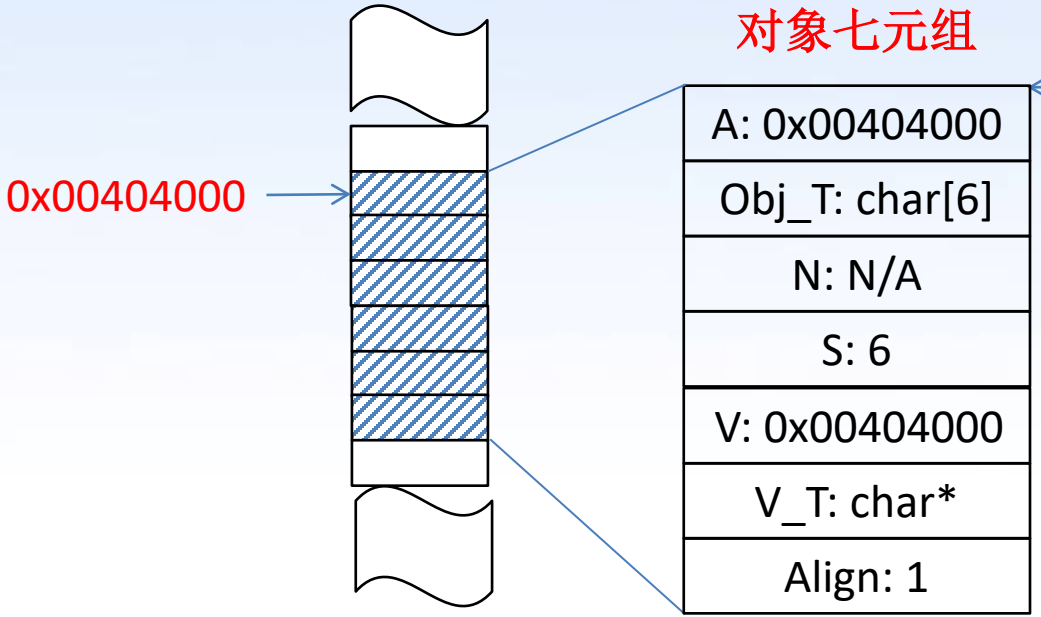
<size, size_t>

sizeof(b): <6, size_t>

注意rvalue的类型size_t

size_t x = sizeof(“hello”);

typeof(c)



typeof("hello")不是表达式

typeof("hello") x;
等价于
char x[6];

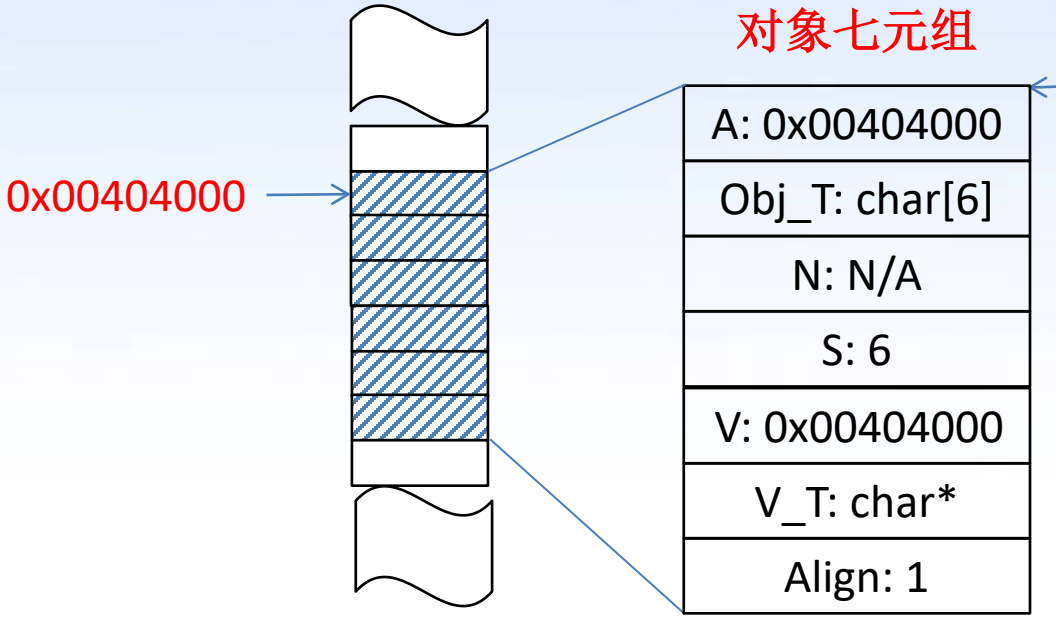
sizeof(typeof("hello"));
等价于
sizeof(char[6]);

& “hello”

&”hello”

对象七元组

<Address, Obj_T*>

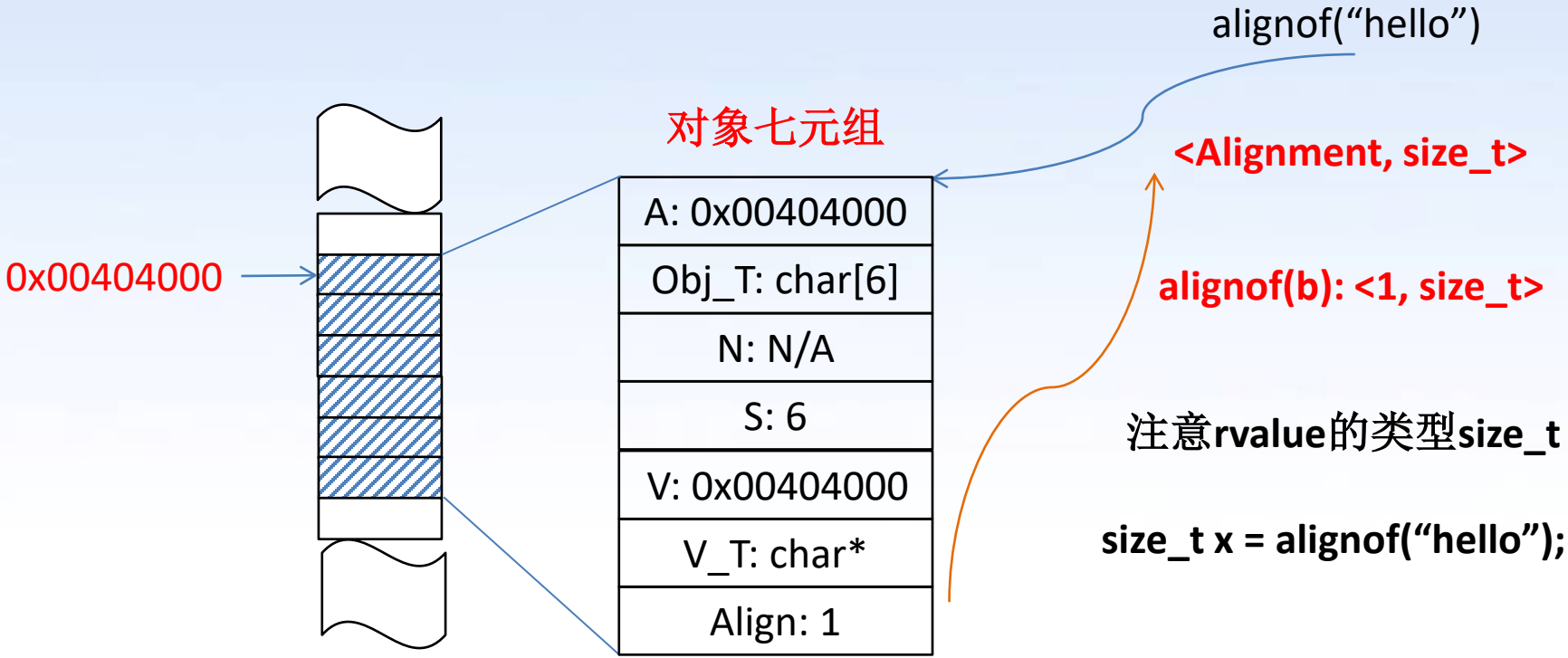


&a: <0x00404000, char(*)[6]>

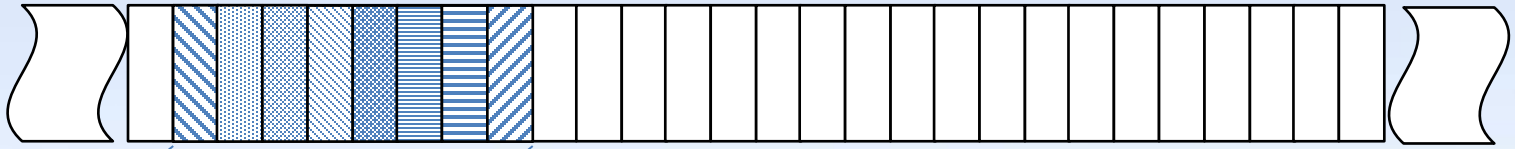
注意rvalue的类型Obj_T*

char (*x)[6] = &”hello”;

alignof(“hello”) （标准并不支持）



字符数组char str[8]



0x0065FE10

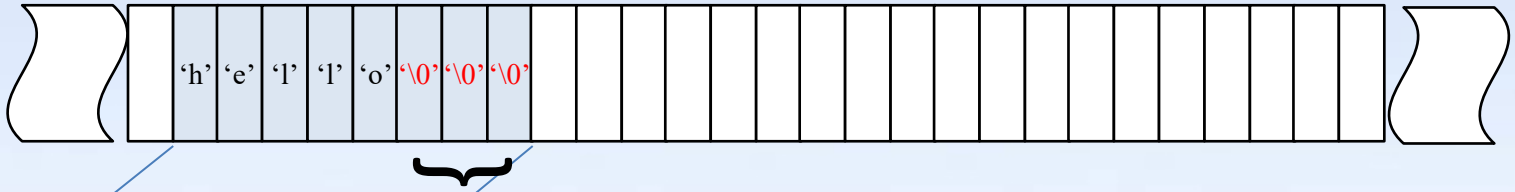
A: 0x0065FE10
Obj_T: char[8]
N: str
S: 8
V: 0x0065FE10
V_T: char*

1、char[8]也是一个数组对象类型

2、拥有和int[2]、float[2][3]等数组完全一样的性质

3、目前str对应的8个字节没有初始化

初始化char str[8] = {'h', 'e', 'l', 'l', 'o'}



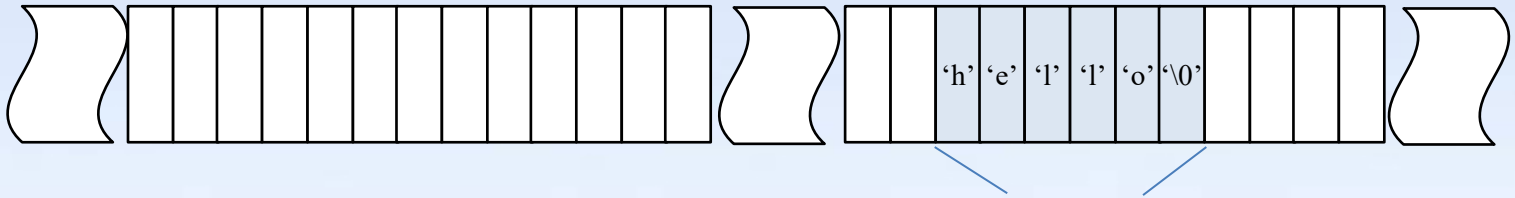
0x0065FE10

A: 0x0065FE10
Obj_T: char[8]
N: str
S: 8
V: 0x0065FE10
V_T: char*

1、 'h', 'e', 'l', 'l', 'o'用于初始化前5个byte

2、后面3个'\0'（null character）是自动填充的

初始化char str[8] = “hello”



“hello”在静态存储区分配了相应的空间

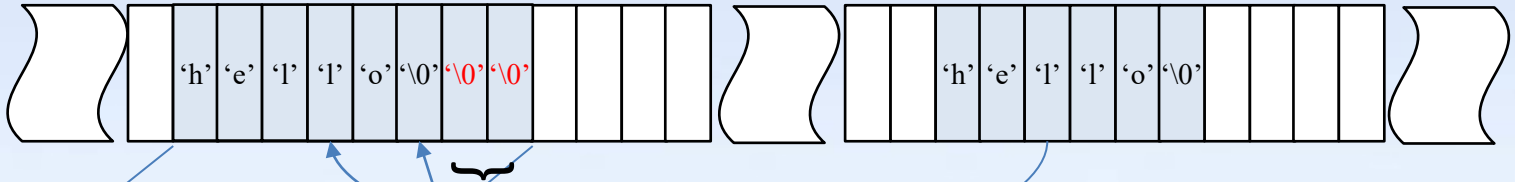
双引号修饰的字符串之后都有一个隐藏字符’\0’

双引号修饰的字符串是具有静态存储周期
(static storage duration)

C语言4种对象存储周期

static, automatic, allocated, thread

初始化char str[8] = “hello”



0x0065FE10

A: 0x0065FE10
Obj_T: char[8]
N: str
S: 8
V: 0x0065FE10
V_T: char*

1、 字符数组可以用String Literal进行初始化

2、第6个’\0’ 是”hello”这个String Literal自身带的

3、后面2个’\0’（null character）是自动填充的

思考题

char* p = “hello”;

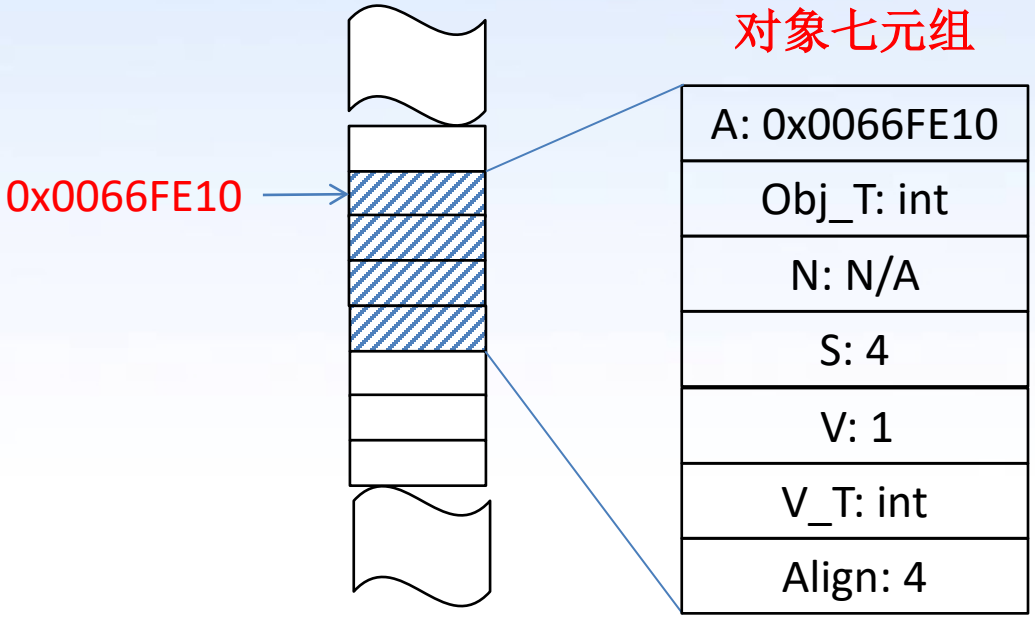
char str[] = “hello”;

这两个hello哪一个需要evaluate？

答案：
第一个需要evaluate，其值赋给对象p。
第二个是用于初始化字符数组str，不进行evaluate

对(int){1}这个lvalue进行evaluate

int a = 1;



(int){1}; → 对(int){1}做evaluate

sizeof((int){1});

typeof((int){1});

&(int){1};

(int){1}++;

(int){1}--;

++(int){1};

--(int){1};

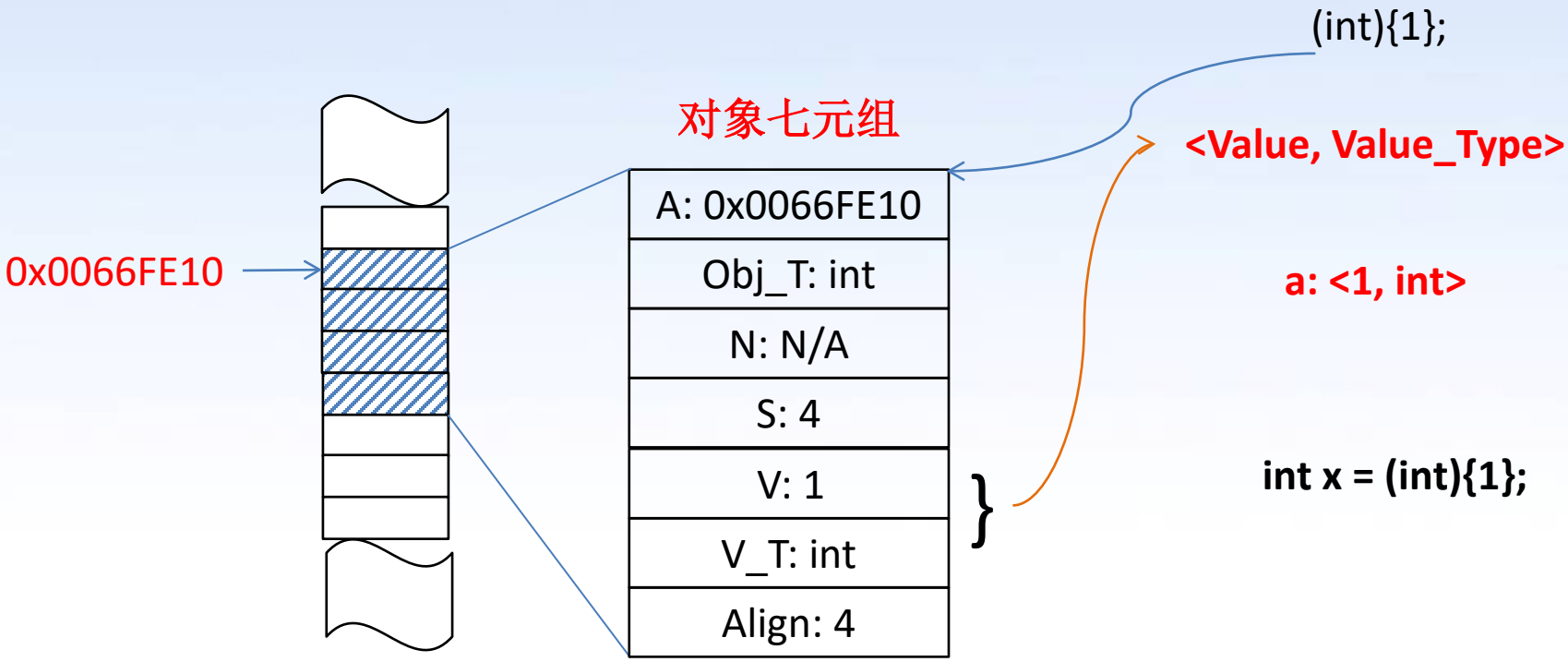
(int){1} = 2;

对(int){1}不做evaluate

这些(int){1}相关的表达式是如何evaluate的呢？

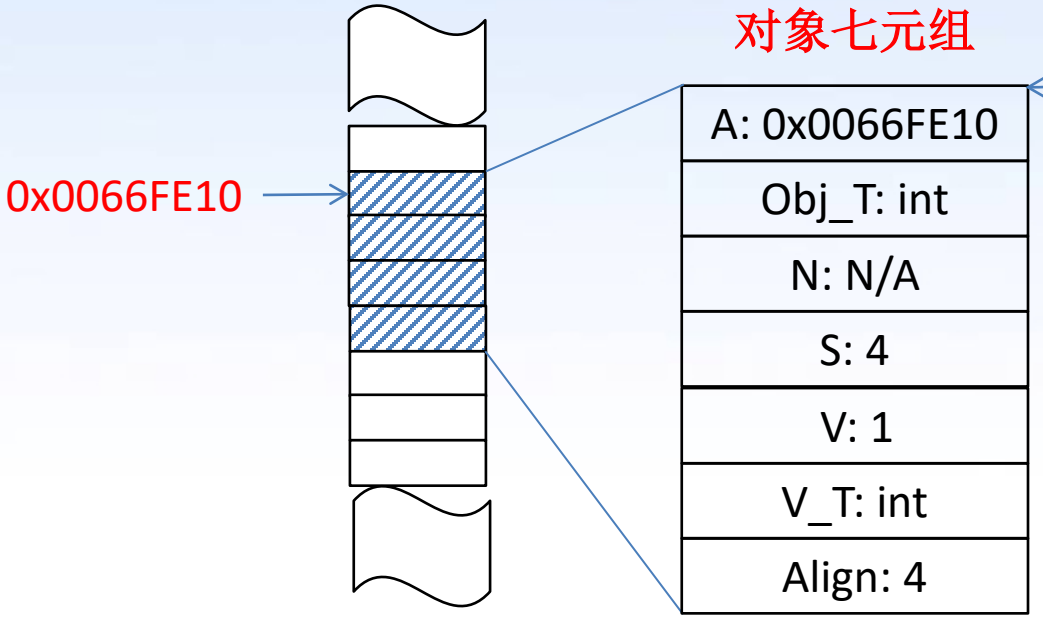
typeof((int){1})不是表达式

当(int){1}被evaluate的时候



sizeof((int){1})或sizeof (int){1}

sizeof((int){1})



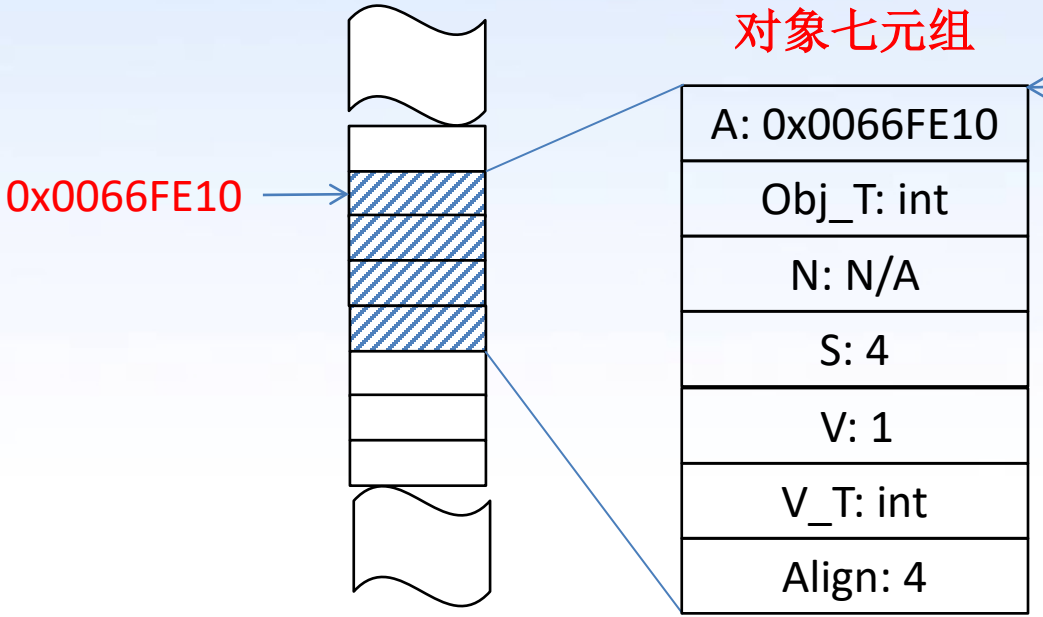
<size, size_t>

sizeof(a): <4, size_t>

注意rvalue的类型size_t

size_t x=sizeof((int){1});

typeof((int){1})



typeof((int){1})

等价于**Obj_T**

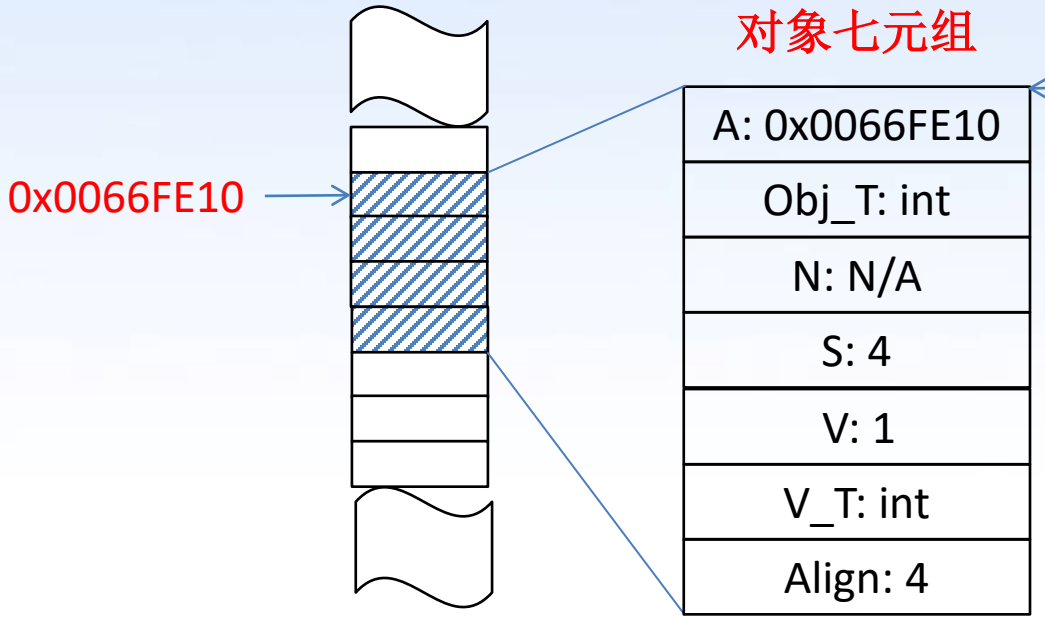
typeof((int){1})等价于int

typeof((int){1})不是表达式

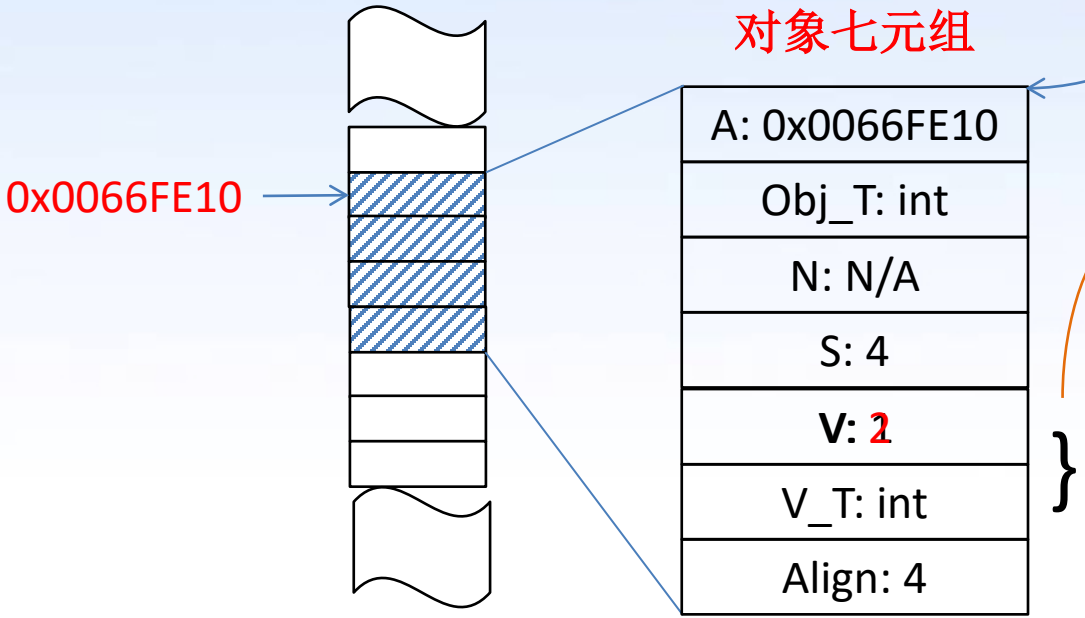
typeof((int){1}) x; 等价于 **int x;**

sizeof(typeof((int){1})) 等价于 **sizeof(int);**

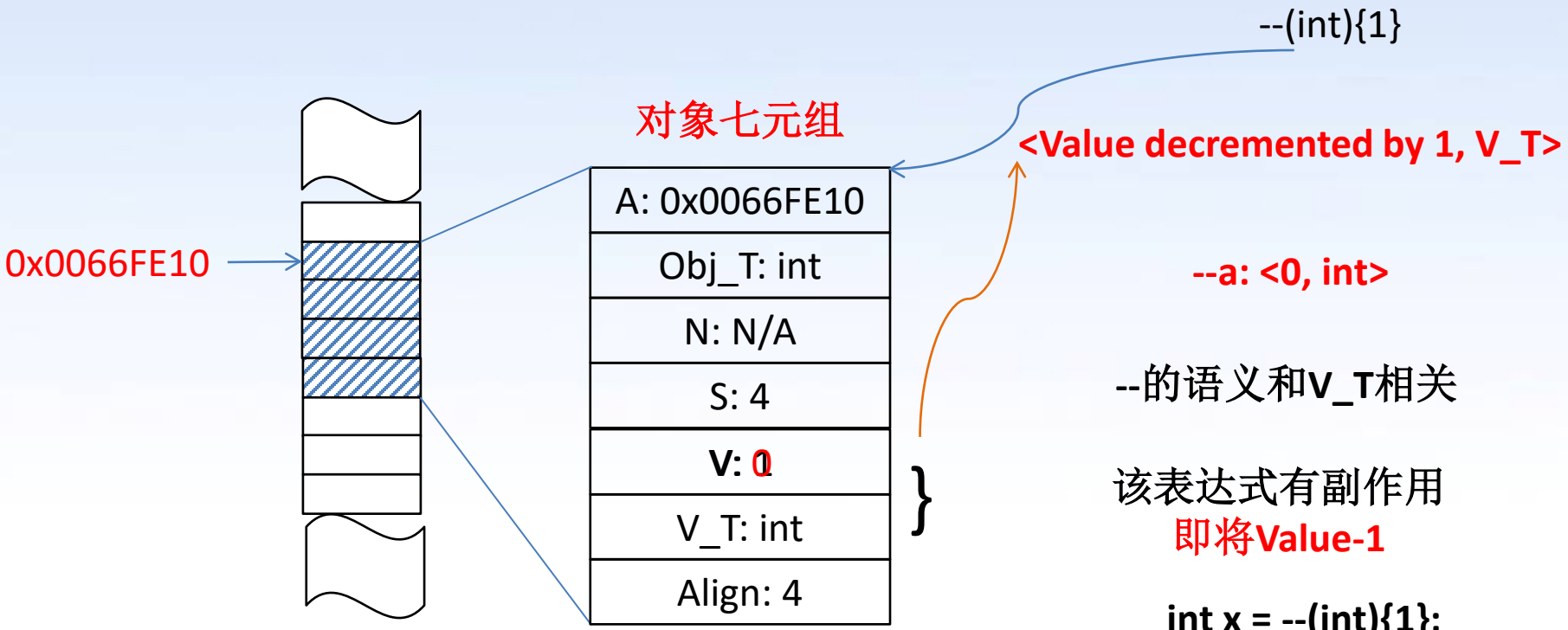
&(int){1}


&(int){1}
对象七元组
<Address, Obj_T*>
0x0066FE10
A: 0x0066FE10
Obj_T: int
N: N/A
S: 4
V: 1
V_T: int
Align: 4
}
&a: <0x0065FE10, int*>
注意rvalue的类型Obj_T*
int* x=&(int){1};

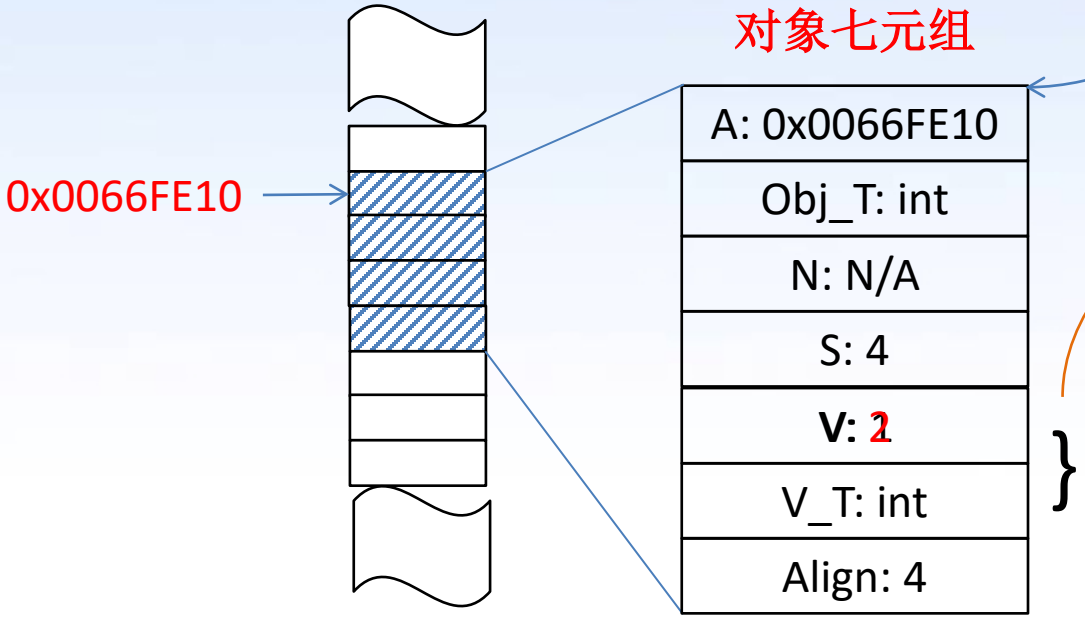
++(int){1}


++(int){1}
对象七元组
<Value incremented by 1, V_T>
0x0066FE10
A: 0x0066FE10
Obj_T: int
N: N/A
S: 4
V: 1 2
V_T: int
Align: 4
}
++a: <2, int>
++的语义和V_T相关
该表达式有副作用
即将Value+1
int x = ++(int){1};

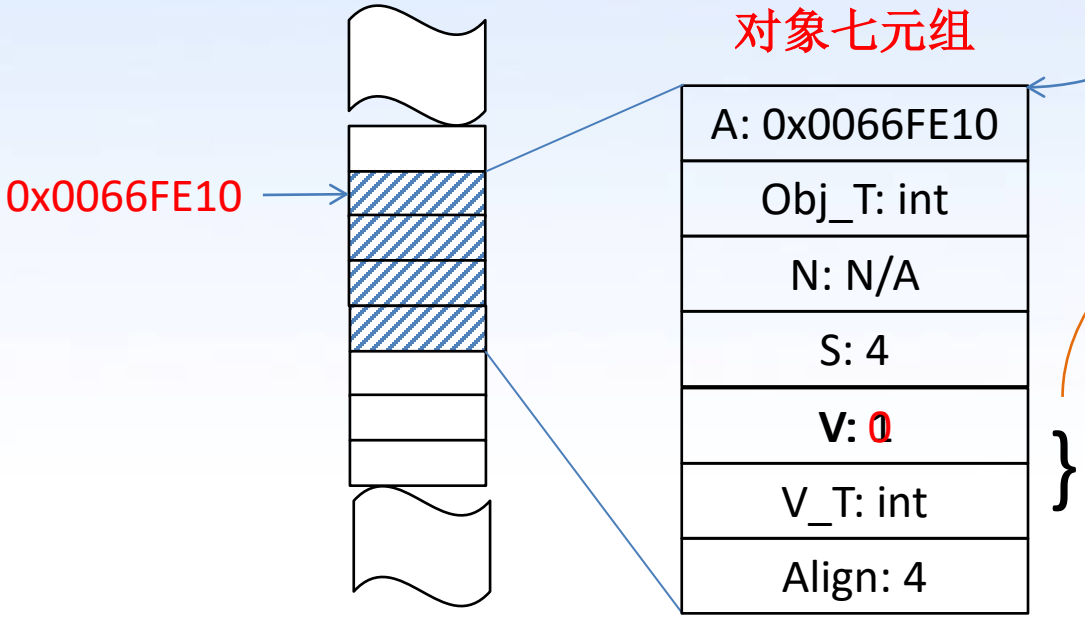
--(int){1}


--(int){1}
对象七元组
<Value decremented by 1, V_T>
0x0066FE10
A: 0x0066FE10
Obj_T: int
N: N/A
S: 4
V: 0 1
V_T: int
Align: 4
}
--a: <0, int>
--的语义和V_T相关
该表达式有副作用
即将Value-1
int x = --(int){1};

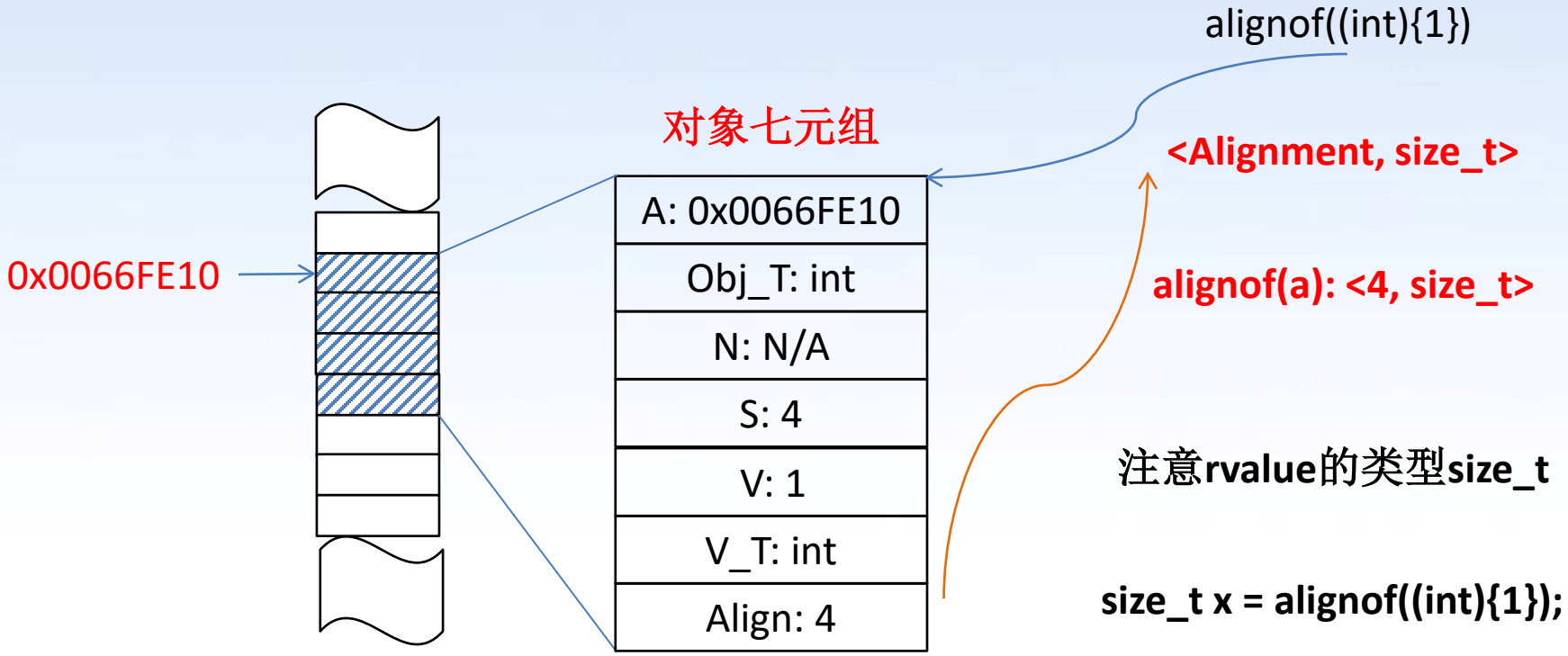
(int){1}++


(int){1}++
对象七元组
0x0066FE10
A: 0x0066FE10
Obj_T: int
N: N/A
S: 4
V: 1
V_T: int
Align: 4
<Value, V_T>
a++: <1, int>
++的语义和V_T相关
该表达式有副作用
即将Value+1
int x = (int){1}++;

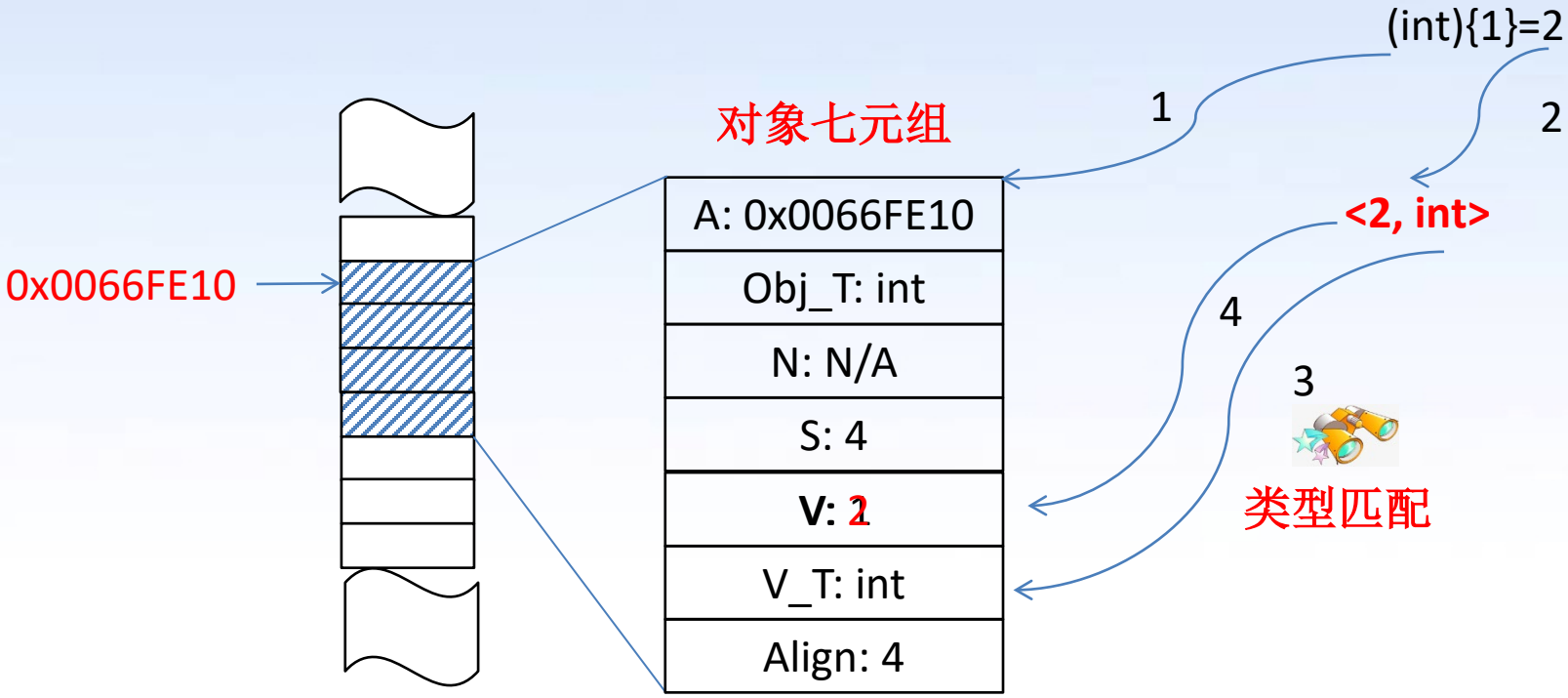
(int){1}--


(int){1}--
对象七元组
0x0066FE10
A: 0x0066FE10
Obj_T: int
N: N/A
S: 4
V: 0
V_T: int
Align: 4
}
<Value, V_T>
a--: <1, int>
--的语义和V_T相关
该表达式有副作用
即将Value-1
int x = (int){1}--;

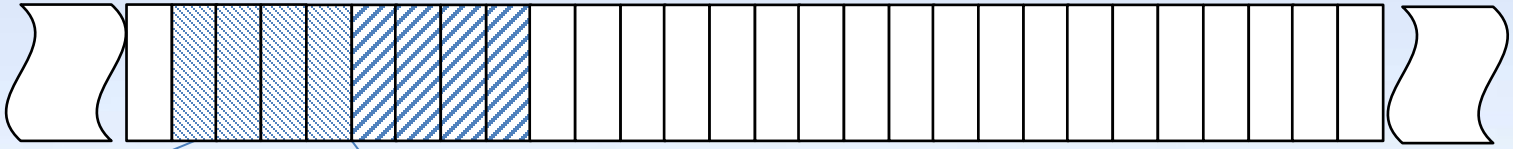
alignof((int){1}) （标准并不支持）



(int){1}=2


(int){1}=2
1
2
对象七元组
A: 0x0066FE10
Obj_T: int
N: N/A
S: 4
V: 1
V_T: int
Align: 4
0x0066FE10
<2, int>
4
3
类型匹配

对(int[1]){1}这个lvalue相关表达式进行evaluate



对象七元组

A: 0x0067FE10	<0x0067FE10,int(*)[1]>	int (*x)[1] = &(int[1]){1};
Obj_T: int[1]		
N: N/A	<4, size_t>	size_t x = sizeof(int[1]){1};
S: 4		
V: 0x0067FE10	< 0x0067FE10, int*>	int* x = (int[1]){1};
V_T: int*		
Align: 4	<4, size_t>	size_t x = alignof(int[1]){1};

思考一下

以上的示例中，“hello”定位的对象都是同一个，正确吗？

以上的示例中，(int){1}定位的对象都是同一个，正确吗？

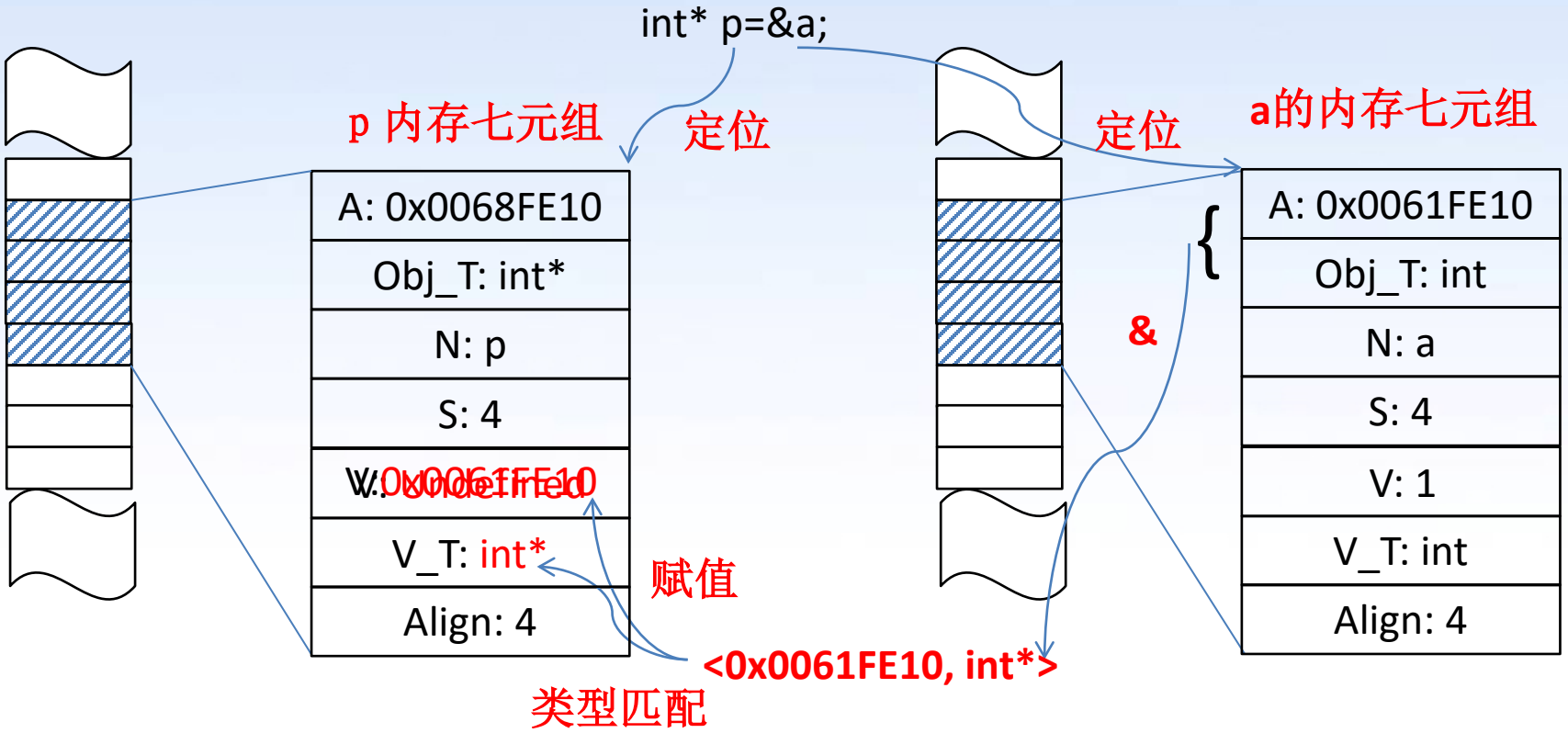
以上的示例中，(int[1]){1}定位的对象都是同一个，正确吗？

关注定位指针类型对象的lvalue

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

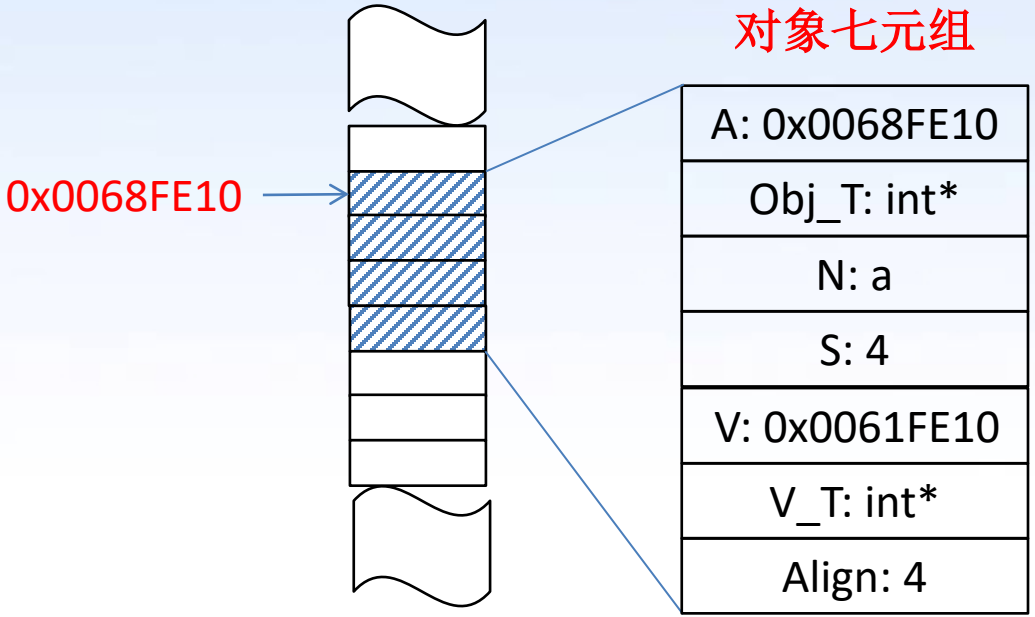
对象标识符p能够定位一个对象，其对象类型为int*

int a=1;int* p=&a发生了什么？



对表达式p进行evaluate示例

int* p = &a;



p; → 对p做evaluate

sizeof(p);

typeof(p);

&p;

p++;

p--;

++p;

--p;

p = NULL;

对p不做evaluate

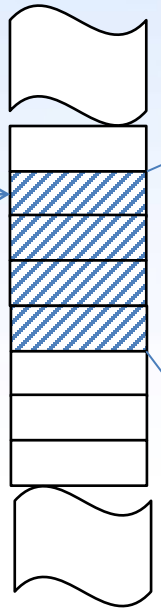
这些p相关的表达式是如何evaluate的呢？

typeof(p)不是表达式

当p被evaluate的时候

int* p = &a;

p;



0x0068FE10

对象七元组

A: 0x0068FE10
Obj_T: int*
N: p
S: 4
V: 0x0061FE10
V_T: int*
Align: 4

<Value, Value_Type>

p: <0x0061FE10, int*>

int* x = p;

理解*p：*exp形式的lvalue

通过*exp来定位对象

假设一个表达式exp的evaluate之后rvalue为<Value, Value_Type>，如果Value_Type是一个对象指针类型，则可以用*exp的方式来定位到Value对应字节编号开头的一段内存

用“*”来间接内存定位

int a=10; int* p=&a;

为什么*p可以间接定位内存？

*p的时候发生了什么？

1、p是一个对象标识符，定位对象p

2、p没有跟sizeof、typeof、sizeof、++/--结合，也不是在等号左边，要做evaluate，获得表达式p的rvalue<V, V_T>，即**<0x0061FE10, int*>**

3、根据<V, V_T>定位*p对象，规则如下：

1）V的值是*p对应对象的Address(A)

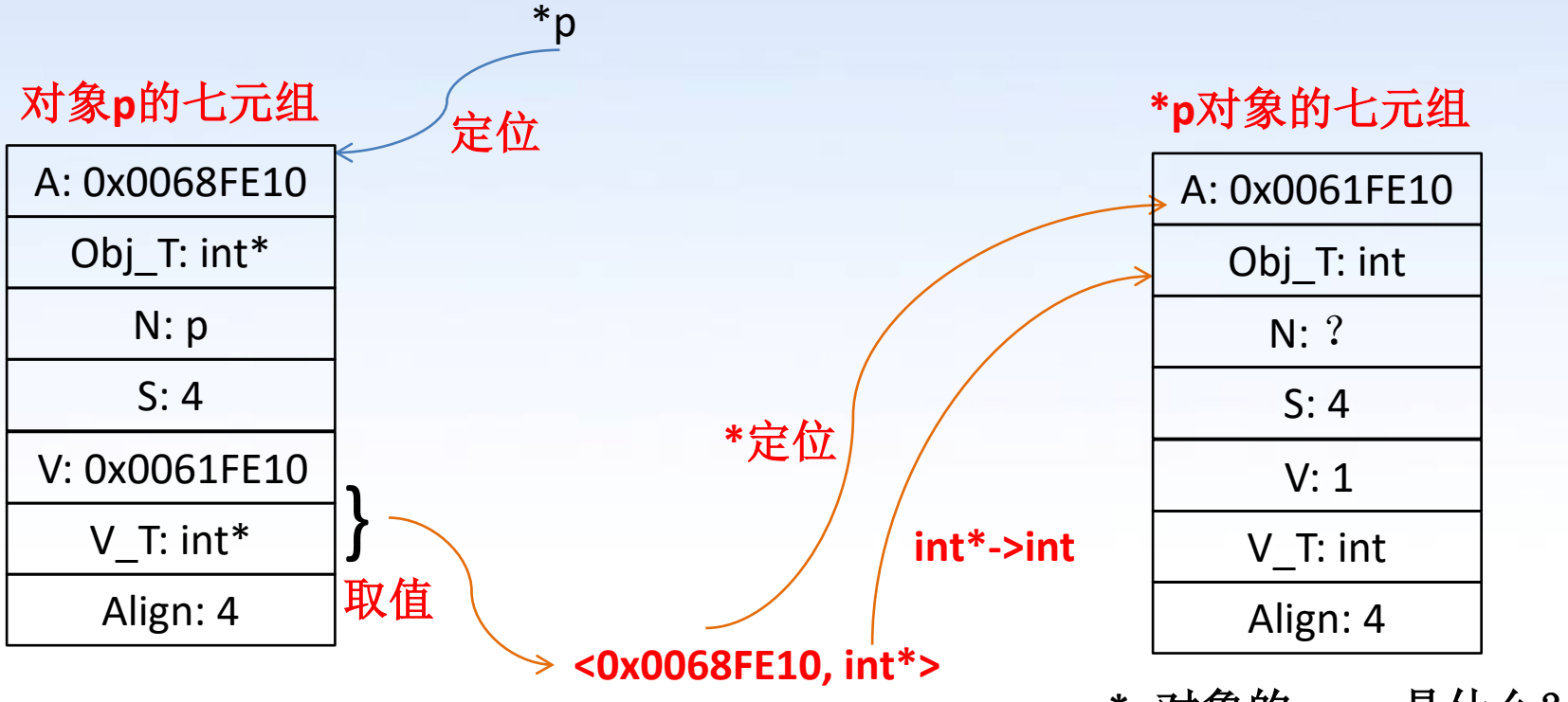
2）V_T对应的Referenced Type就是*p定位对象Object_Type(Obj_T)

3）*p内存七元组其他属性根据V_T依次确定

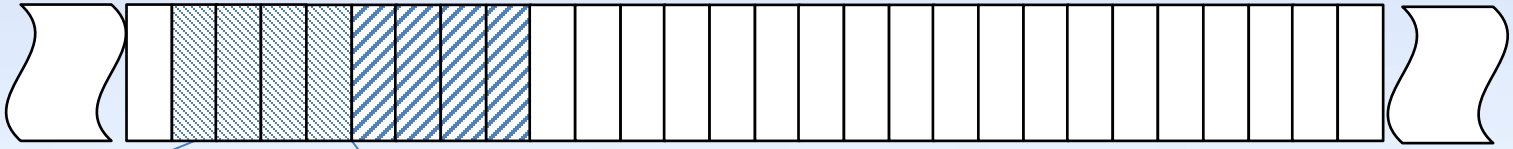
对象 p 的内存七元组

A: 0x0068FE10
Obj_T: int*
N: p
S: 4
V: 0x0061FE10
V_T: int*
Align: 4

int a=1; int* p=&a; *p发生了什么？



*p这个lvalue相关的表达式进行evaluate



对象七元组

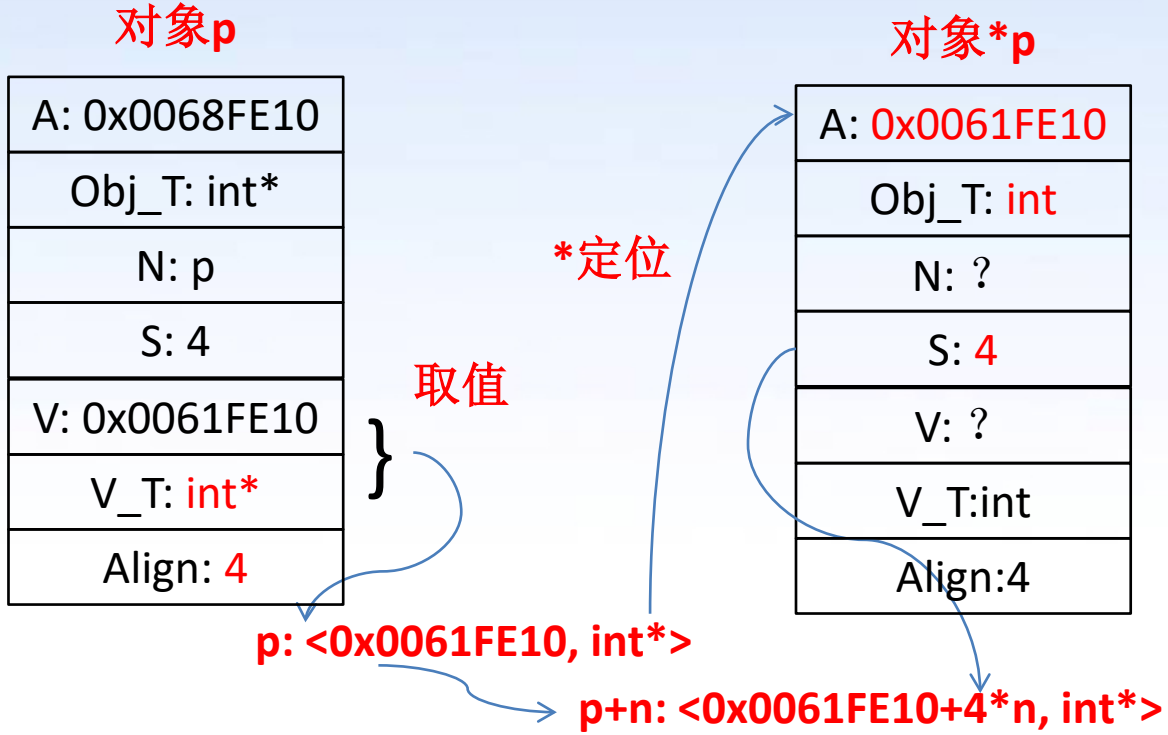
A: 0x0061FE10	<0x0061FE10,int*>	int* p = &(*p);
Obj_T: int		
N: N/A	<4, size_t>	size_t x = sizeof(*p);
S: 4		
V: 1	< 1, int>	int x = *p;
V_T: int		
Align: 4	<4, size_t>	size_t x = alignof(*p)

p+n是什么意思？

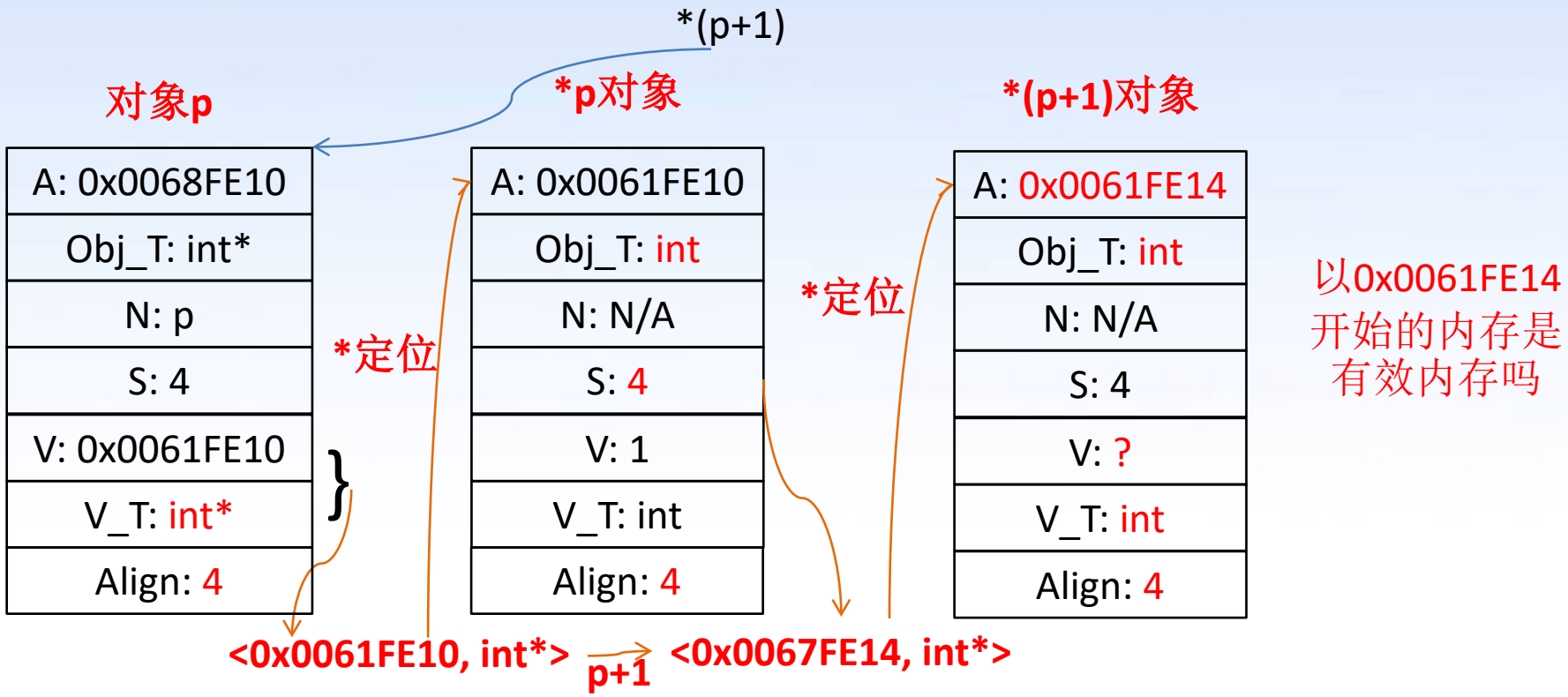
```
int a = 10;
int* p=&a;
```

p+n=？

1、定位p的对象
2、p取值<value, value_type>
3、计算sizeof(*p)
4、p+n的value=
p的value+sizeof(*p)*n
5、p+n的Value_Type不变



*(p+1)定位的对象有什么问题？



回顾一下*p和*(p+1)

int* p;

*p;

1、p是一个表达式exp，该返回值类型是int*，合法指针类型
2、exp的返回值<Value, Value_Type>
3、*exp定位一个对象M
4、M的Address的exp的Value
5、M的Obj_T是exp的Value_Type的Referenced Type

p是一个表达式

*(p+1);

1、p+1是一个表达式exp，该返回值类型是int*，合法指针类型
2、exp的返回值<Value, Value_Type>
3、*exp定位一个对象M
4、M的Address的exp的Value
5、M的Obj_T是exp的Value_Type的Referenced Type

p+1是一个表达式

*(exp+n) <=> exp[n]

任何一个表达式exp，只要这个表达式返回值类型是一个对象指针类型，则

*(exp+n) <=> exp[n]

int* p;

*p = *(p+0)	p[0]
*(p+1)	p[1]
*(p+2)	p[2]
...	...
*(p+n)	p[n]

exp1[exp2] <=> exp2[exp1]

给定两个表达式，只要其中一个表达式evaluate后rvalue的类型是一个有效的对象指针类型，而另一个表达式evaluate后rvalue的类型是一个合法的整数类型，则这两个表达式就可以用[]的方式进行对象定位，但C语言并没有规定哪一个必须放在[]里面。

int* p;

p[0]	0[p]
p[1]	1[p]
p[2]	2[p]
...	...
p[n]	n[p]

思考题

1、int* p;
(p+1)[2]和p[3]的写法是否一样？

答案: *(exp+n) <=> exp[n]

1、*(p+1+2)，将p看作exp
等价于*(p+3)，即p[3]

2、*(p+1+2)，将p+1看作exp
等价于*((p+1)+2)，即(p+1)[2]

因此，(p+1)[2]和p[3]等价