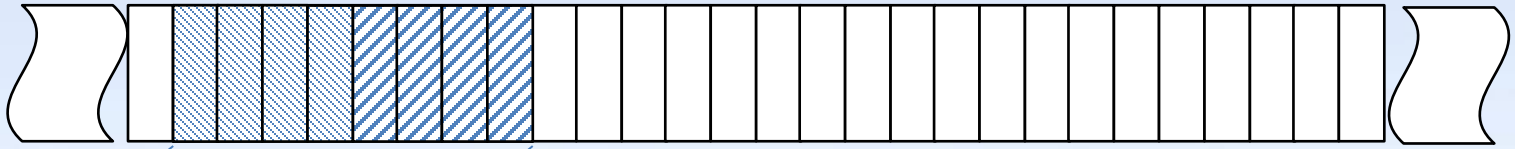
了解字符串

1. 了解字符数组及其的初始化方法
2. 了解String Literal和String的区别
3. 通过字符串了解存储空间的生命周期
4. 掌握字符串对应内存的不同操作细微差别
5. sizeof(“hello”) vs. sizeof(“hello”+1-1)
6. “hello”[0]的语义

回顾数组对象类型int e[2]



0x0076AB11

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

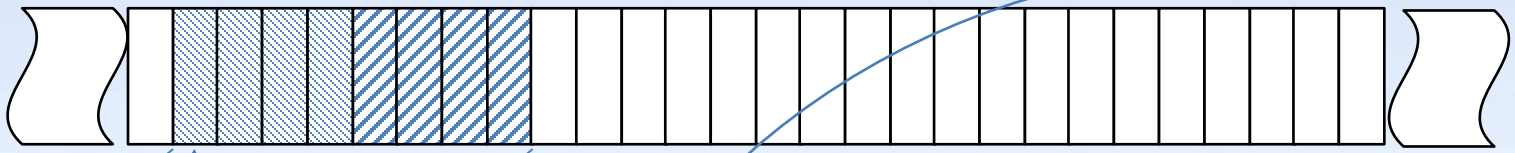
1、对象类型是int[2]，变量名是e

2、表示值是第一个元素的首地址

3、表示值类型为第一个元素对应的指针类型

4、目前e对应的8个字节没有初始化

数组初始化int e[2] = {1};



0x0076AB11

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

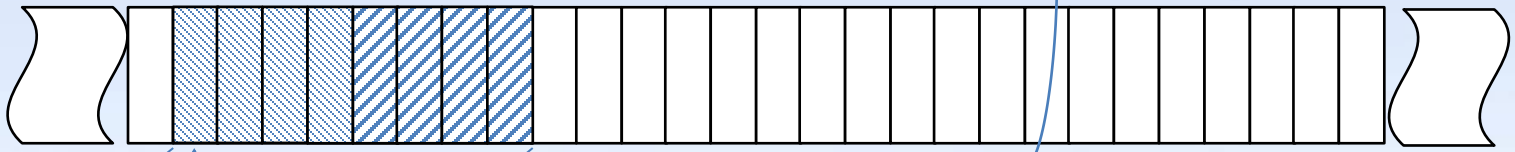
<1, int>　　<0, int>

1、数组变量只能通过初始化进行整体赋值

2、1用来初始化数组中第一个元素，后面自动用0初始化

2、数组对应内存表示值不会发生任何变化

数组声明及初始化int e[] = {1, 2}



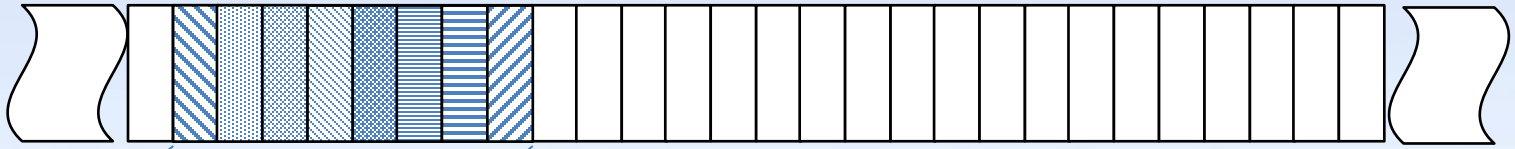
0x0076AB11

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

<1, int> <2, int>

1、声明的时候如果不指定数组长度，则数组长度等于初始化列表中元素的个数

字符数组char str[8]



0x0097BC11

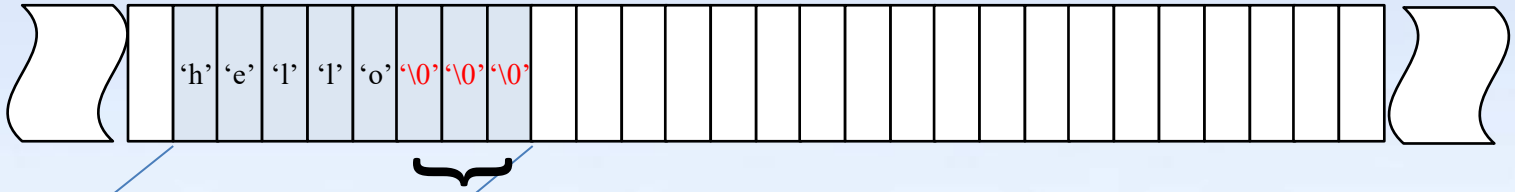
A: 0x0097BC11
Obj_T: char[8]
N: str
S: 8
V: 0x0097BC11
V_T: char*

1、char[8]也是一个数组变量

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'}



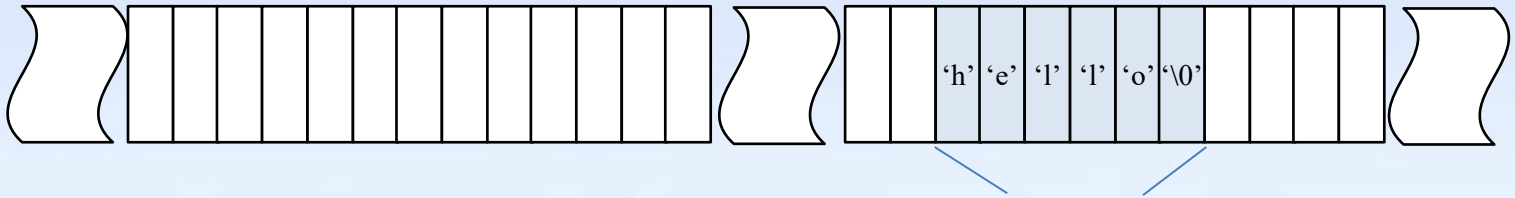
0x0097BC11

A: 0x0097BC11
Obj_T: int[8]
N: str
S: 8
V: 0x0097BC11
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

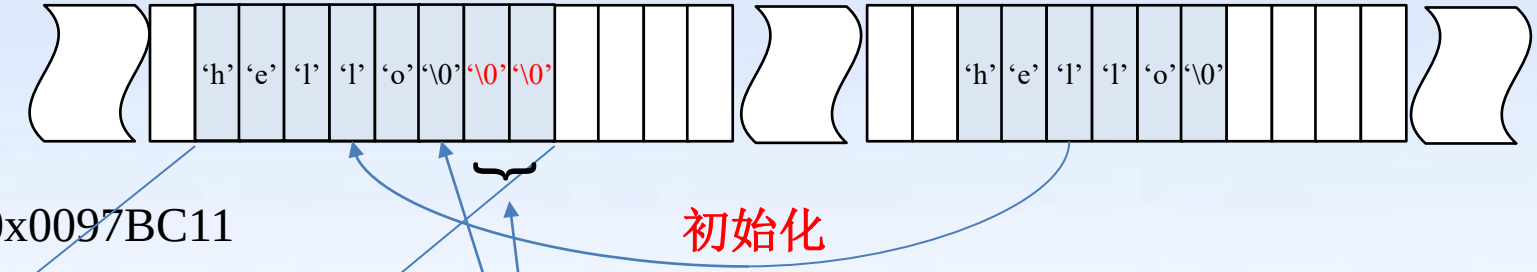
双引号修饰的字符串之后都有一个隐藏字符’\0’

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”



A: 0x0097BC11
Obj_T: char[8]
N: str
S: 8
V: 0x0097BC11
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

String Literal vs. String

A character string literal is a sequence of zero or more multibyte characters enclosed in double-quotes

例如：“hello\0world”

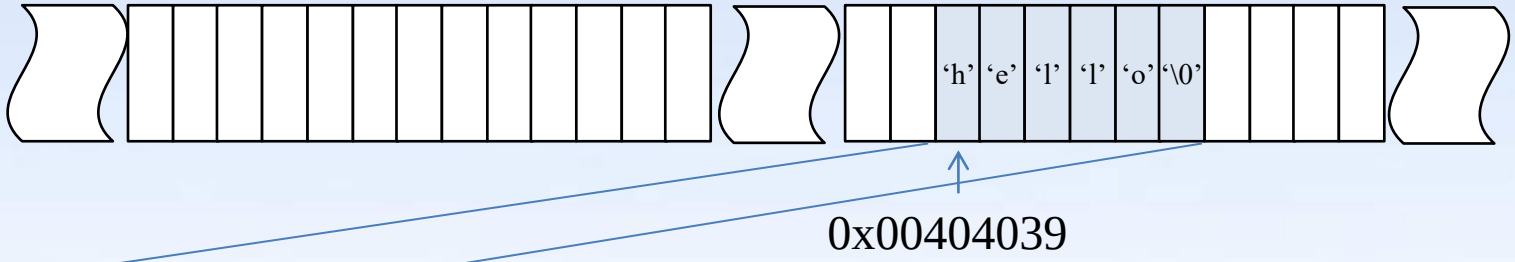
A string is a contiguous sequence of characters terminated by and including the first null character

例如：“hello world”

最后都有一个隐藏的’\0’，双引号内也嵌入’\0’的称之为String Literal

不失一般性，可以统称为字符串

“hello”分配以及定位内存



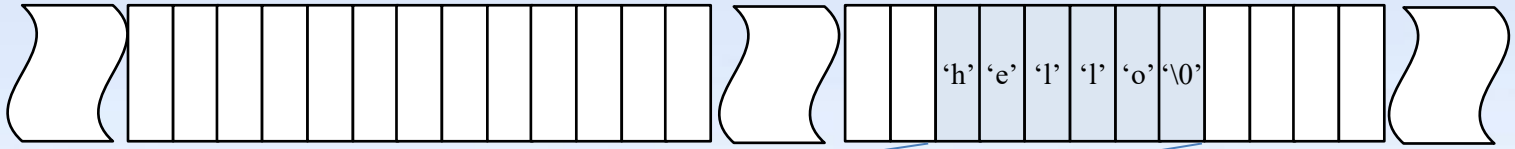
A: 0x00404039
Obj_T: char[6]
N: N/A
S: 6
V: 0x00404039
V_T: char*

”hello”的含义包括：

1、在静态区找到合适的空间，存放'h', 'e', 'l', 'l', 'o', '\0'

2、”hello”本身可以定位到这块内存

“hello”本身就可以用于定位这块内存



0x00404039

”hello”的内存六元组

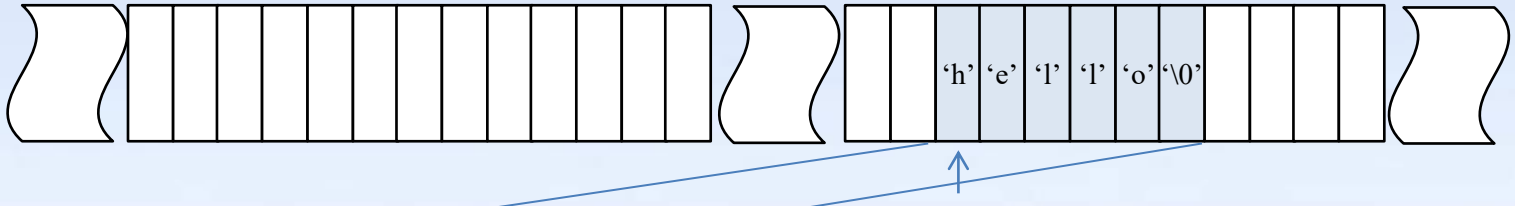
A: 0x00404039
Obj_T: char[6]
N: N/A
S: 6
V: 0x00404039
V_T: char*

<0x00404039,char(*)[6]>　　char (*p)[6] = &”hello”;

<6, size_t>　　size_t c = sizeof(”hello”);

<0x00404039, char*>　　char* q = ”hello”;

sizeof(“hello”) vs. sizeof(“hello”+1-1)



”hello”的内存六元组

A: 0x00404039
Obj_T: char[6]
N: N/A
S: 6
V: 0x00404039
V_T: char*

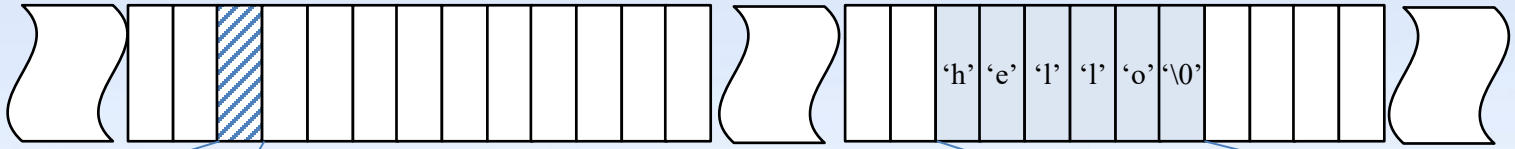
sizeof(”hello”);

结果是6：定位”hello”对应内存，获得这块内存的大小

sizeof(”hello”+1-1);

结果是4：”hello”跟+1结合，”hello”这块内存的表示值类型是char*，+1和-1操作不会改变表达式返回值类型

char c = “hello”[0];



A: 0x0039BD16
Obj_T: char
N: c
S: 1
V: Undefined
V_T: char

c的内存六元组

<'h', char>

A: 0x00404039
Obj_T: char
N: N/A
S: 1
V: 'h'
V_T: char

”hello”[0]的内存六元组

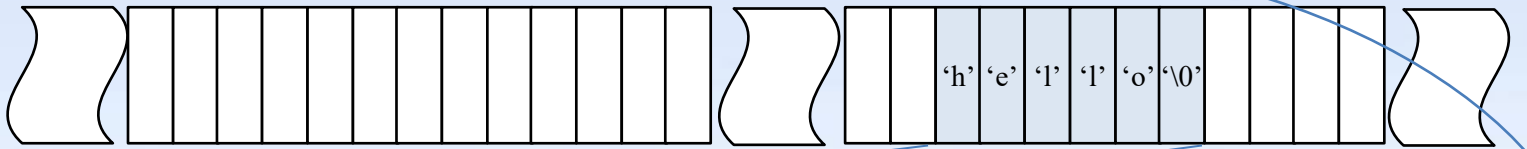
*定位

<0x00404039, char*>

A: 0x00404039
Obj_T: char[6]
N: N/A
S: 6
V: 0x00404039
V_T: char*

”hello”的内存六元组

“hello”[0] = 'e';



A: 0x00404039
Obj_T: char[6]
N: N/A
S: 6
V: 0x00404039
V_T: char*

”hello”的内存六元组

<0x00404039, char*>

*定位

这是一个未定义行为

Undefinede Behavior

A: 0x00404039
Obj_T: char
N: N/A
S: 1
V: 'h'
V_T: char

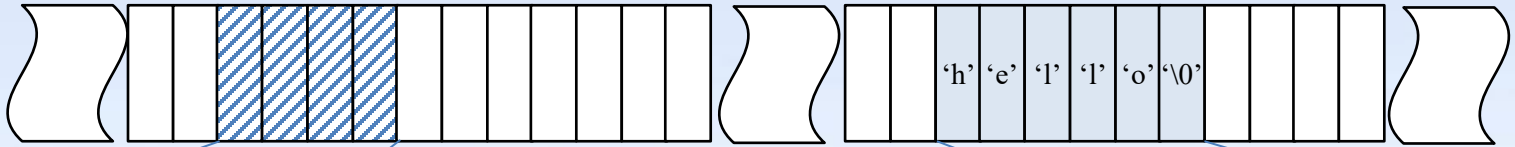
”hello”[0]的内存六元组

<'e', char>

赋值

?

char* p = “hello”;



0x00404039

A: 0x0076AB21
Obj_T: char*
N: p
S: 4
V: 0x00404039
V_T: char*

p的内存六元组

<0x00404039, char*>

赋值

A: 0x00404039
Obj_T: char[6]
N: N/A
S: 6
V: 0x00404039
V_T: char*

”hello”的内存六元组

char* p = “hello” 同时 char* q = “hello”

```
int main()
{
    char* p = "hello";
    char* q = "hello";

    printf("p=%x, q=%x\n", p, q);

    return 0;
}
```

```
C:\Users\wahaha\Desktop\test\a.exe
p=404000, q=404000

Process returned 0 (0x0)   execution time : 0.021 s
Press any key to continue.
```

注意：p和q的值是一样的

sizeof(String Literal) vs strlen(String Literal)

“hello\0world”

sizeof(“hello\0world”)：12
包括中间那个\0以及最后那个\0

strlen(“hello\0world”)：5
字符串的长度是从开始到第一个\0为止（不包括\0)

“hello world”

sizeof(“hello world”)：12

strlen(“hello world”)：11
字符串的长度是从开始到第一个\0为止（不包括\0)

Compound Literal

(type-name){Initializer-list}

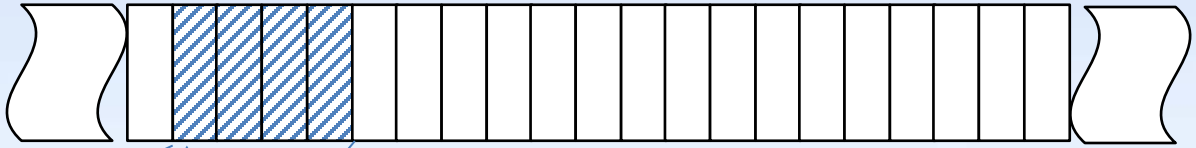
声明定义一个匿名(unnamed)的对象（Object Type）

该对象的对象类型是type-name

该对象由Initializer-list进行初始化

Provides an unnamed object whose value is given by the initializer list.

(int){1}



0x0061FE0C

A: 0x0061FE0C
Obj_T: int
N: N/A
S: 4
V: 1
V_T: int

1、对象类型是int

2、初始化的值为1

(int){1}在函数外，是static storage duration

否则是automatic storage duration

如何访问这块内存呢？

(int){1}是lvalue

(int){1} = 5

(int){1}的内存六元组

A: 0x0061FE0C
Obj_T: int
N: N/A
S: 4
V: 5
V_T: int

<5, int>

赋值

automatic storage duration

(int){1}是合法modifiable lvalue

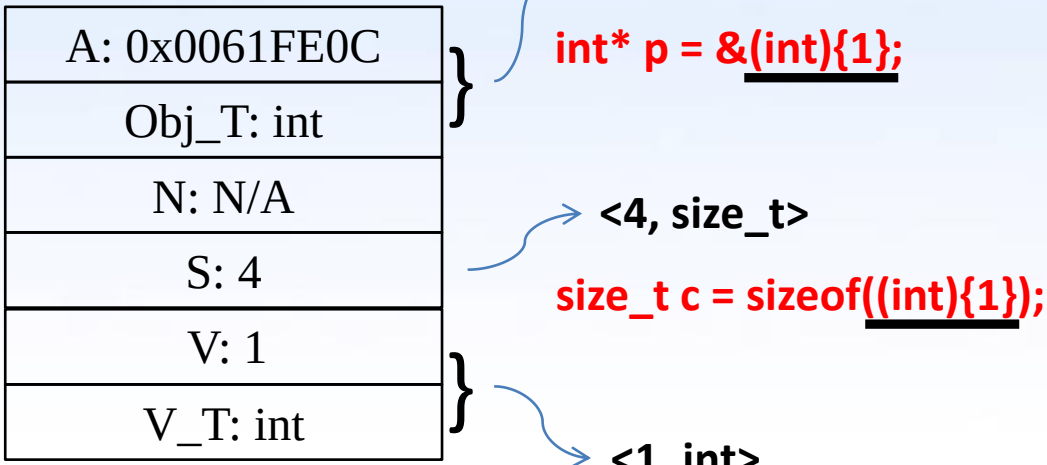
static storage duration

(int){1} = 5; ✗

不是合法的全局变量始化

观察（int）{1}

(int){1}的内存六元组




```
int* p = &(int){1};
size_t c = sizeof((int){1});
int a = (int){1};
```

问题在哪?

每个(int){1}都各分配4个字节

printf("%x, %x", &(int){1}, &(int){1});

同一作用域（int）{1}保持一份

```
int* p = &(int){1};
int* q = &(int){1};
int* r = &(int){1};

printf("%x, %x, %x\n", p, q, r);
```

```
61fdfc, 61fe00, 61fe04

Process returned 0 (0x0)
Press any key to continue.
```

```
for(int i=0; i<3; i++)
{
    int* p = &(int){1};

    printf("%x\n", p);
}
```

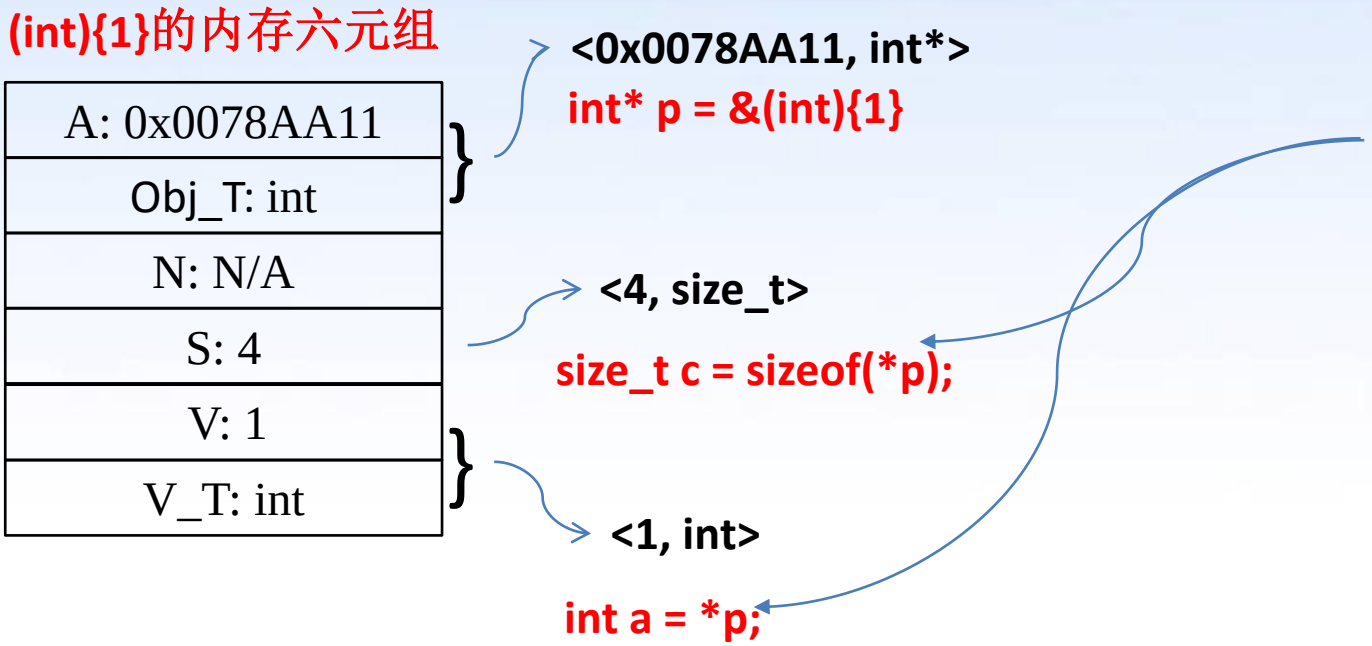
```
61fe0c
61fe0c
61fe0c

Process returned 0 (0x0)
Press any key to continue.
```

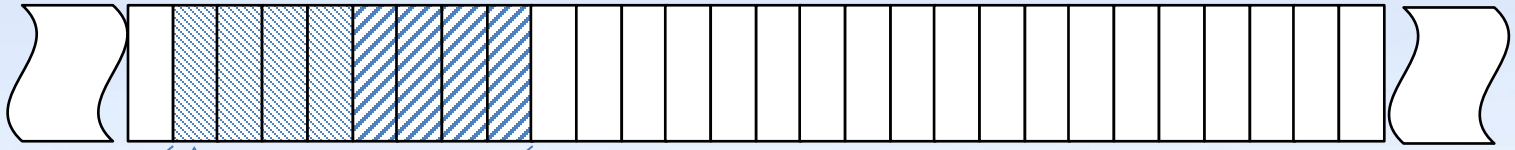
在一个作用域内，同样的Compound Literal只有一个

Each compound literal creates only a single object in a given scope

观察（int）{1}的各种视角



(int[2]){1, 2}



0x0061FE18

A: 0x0061FE18
Obj_T: int[2]
N: N/A
S: 8
V: 0x0061FE18
V_T: int*

<1, int> <2, int>

1、对象类型是int[2]

2、2个int元素的值分别是1，2

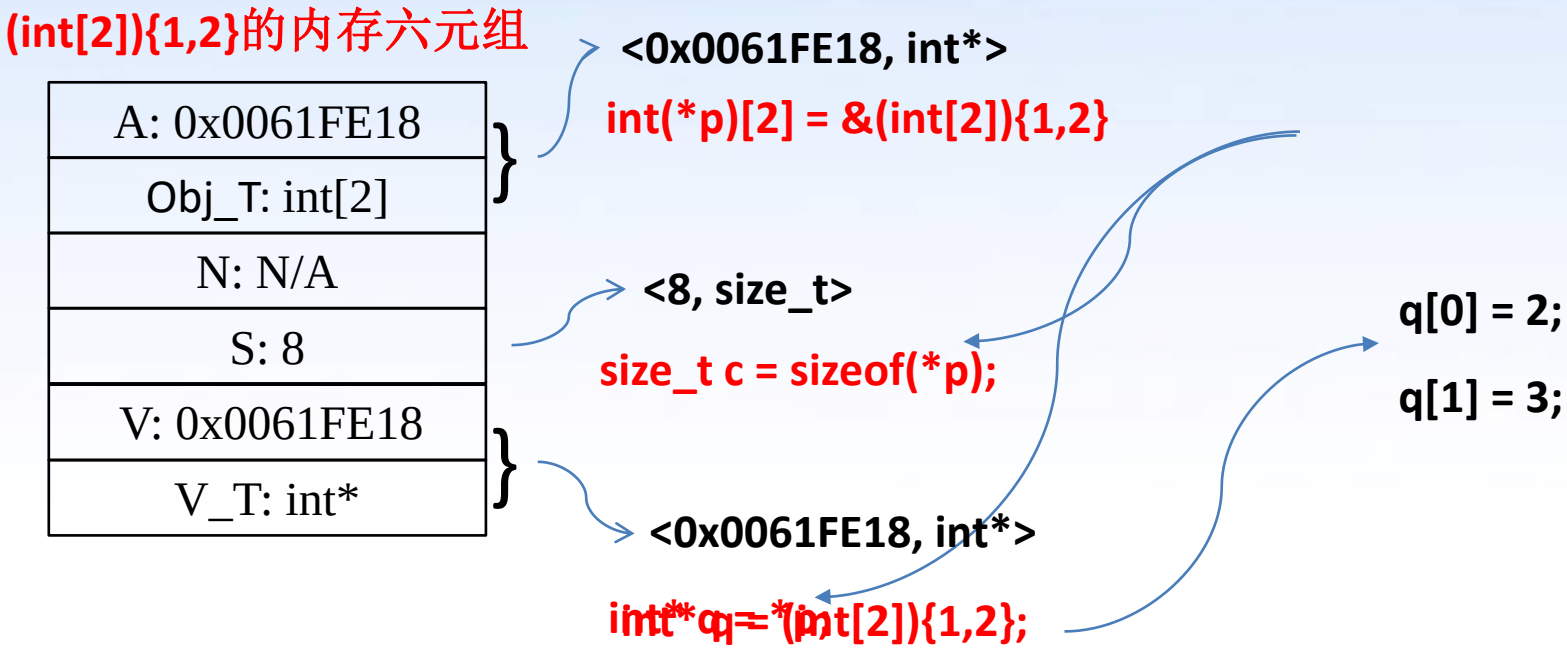
(int[]){1,2}

数组如果不指定长度

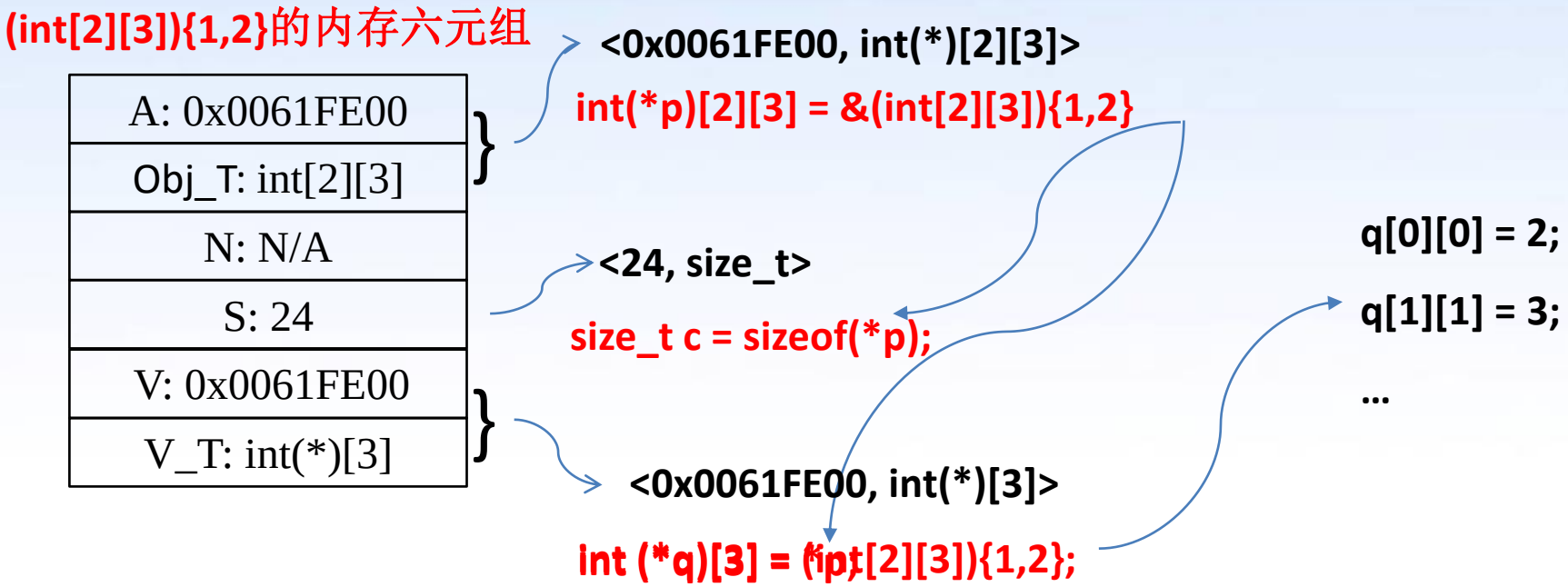
就是初始化列表元素的个数

如何访问这块内存呢？

观察（int[2]）{1, 2}的各种视角



观察更高维数组的Compound Literal



Compount Literal作为参数

```
struct student{
    int ID;
    char name[256];
};
```

```
int main()
{
    func((struct student){1220, "Zhang San"});

    return 0;
}
```

```
void func(struct student s)
{
    printf("%s\n", s.name);
}
```

```
Zhang San

Process returned 0 (0x0)
Press any key to continue.
```

Literal的含义是什么？

进一步了解Literal的含义及在C和C++的细微差异

先来看C语言中的Literal

C语言有两个Literal修饰的概念

String Literal 和 Compound Literal

Literal这个单词的含义：

1、being the most basic meaning of a word or phrase, rather than an extended or poetic meaning

2、that follows the original words exactly

3、lacking imagination

为什么需要String Literal

因为String在C语言中有特殊语义规范

“hello” => 占据6个字节，每个字节分别为 ‘h’, ‘e’, ‘l’, ‘l’, ‘o’, ‘\0’

双引号引导的一串字符，有且只有最后一个字符是\0，被称之为字符串（String）

“hello\0world” => ‘h’, ‘e’, ‘l’, ‘l’, ‘o’, ‘\0’, ‘w’, ‘o’, ‘r’, ‘l’, ‘d’, ‘\0’

\0本身也是一个有效字符，因此中间还包含多个\0的字符串
被称之为String Literal，也就是按字面去理解的字符串

String一定是String Literal，反之则不一定

为什么需要Compound Literal

(int[5]){1,2,3,4,5}

这是一种合法的后缀表达式，在C语言标准中命名为Compound Literal

Literal在这里强调按字面理解的type + initializer的组合形式

C语言中的常量

C语言包括5种常量（constant)

整数常量、浮点数常量、枚举常量、字符常量

integer-constant
floating-constant
enumeration-constant
character-constant
predefined-constant false, true, nullptr

An identifier declared as an enumeration constant has type int.

Compound/String Literal vs. 常量

	常量 （以10为例）	String Literal （以”hello”为例）	Compound Literal （以(int[2]){1,2}为例）
是否为lvalue	否，&10编译错误	是，&”hello”合法	是，&(int[2]){1,2}合法
对应内存是否能修改	否 非lvalue，自然不能修改	形式上可以，实则未定义 例1：“hello”[0] = ‘a’; 编译警告，运行出错 例2：char* p=“hello”; p[0] = ‘a’; 无编译警告，运行出错	是 例1：(int[2]){1,2}[0] = 3; 无编译警告，运行正确 例2：int* p = (int[2]){1,2}; p[0] = 3; 无编译警告，运行正确
值是什么	<10，int>	<Address, char*> char* p = “hello”;	<Address, int*> int* p = (int[2]){1,2};

const修饰的对象 vs. 常量

const int a = 10;

a是一个标识符（identifier），也是一个基础表达式，但是一个lvalue

const修饰的对象如果没有被volatile同时修饰，则：
1、可以把这个对象放到只读区域（read-only region of storage）
2、如果没有对这个对象有取地址的操作，也可以不用真的为该对象分配内存

const volatile int b
b对应的内存虽然由const修饰，但仍然可能会被修改

const修饰的对象和常量是不同的概念体系

C++中的Literal

integer-literal	5,15
character-literal	‘a’, ‘b’, ‘c’
floating-literal	10.3
string-literal	“hello”
boolean-literal	true, false
pointer-literal	nullptr
user-defined-literal	123_KM

C++对Literal的解释
The term “literal” generally designates, in this document, those tokens that are called “constants” in ISO C.

C vs C++

	C	C++
5	integer-constant	integer-literal
10.2	floating-constant	floating-literal
‘a’	character-constant	character-literal
"hello"	string literal	string-literal

String Literal在C/C++中的区别

以”hello”为例

“hello”[0] = ‘a’;	C：（Undefined Behavior)	C++：编译出错
“hello”的值	C：值类型是char* char* p = "hello";	C++：值类型是const char* const char* p = "hello";

C++中“hello”的返回值类型是const char*
对应Literal在C++中具有常量的含义

C++为什么没有Compound Literal

```
int* p = (int[5]){1,2,3,4,5};
```

```
&(int[5]){1,2,3,4,5};
```

C  C++ 

```
error: taking address of temporary
```

```
(int[5]){1,2,3,4,5}[0] = 6;
```

C  C++ 

这块内存可以修改
但Literal有常量的含义

```
(int){1} = 2;
```

C  C++ 

```
error: using temporary as lvalue
```

这还是一个合理的后缀表达式，但在
C++中没有Compound Literal这个概念了

C/C++中sizeof('a')的区别

1、对象类型的大小

sizeof(type_name)

例如：

sizeof(int)的结果为	4
sizeof(int[10])的结果为	40
sizeof(int*)的结果为	4

2、表达式返回值类型的大小

sizeof(expression) 或 sizeof expression

例如：给定int a和int b[10]

sizeof(a)的结果为	4
sizeof(b)的结果为	40
sizeof(b+1-1)的结果为	4

sizeof用于char的疑惑

1、对象类型的大小

sizeof(type_name)

例如：

sizeof(char)的结果为 1

2、表达式返回值类型的大小

sizeof(expression) 或 sizeof expression

例如：给定char c = ‘a’;

sizeof(c)的结果为 1

sizeof(c++)的结果为 1

sizeof(‘a’)的结果是?

‘a’是什么类型？

‘a’, ‘b’, ‘c’在C语言中视为Integer Character Constant

An integer character constant has type int

sizeof(‘a’)的结果是4

sizeof(‘a’)在C和C++环境下表现不一致
C++中，sizeof(‘a’)返回是1

ISO/IEC JTC1SC22/WG14正在讨论是否要发起修改提案

Object Type vs. Function Type

```
int a;
```

声明定义了一个int类型的变量a

Object Type

```
int func(int a, int b)
{
    // do something
    return 0;
}
```

声明定义了一个返回值类型为int，参数数量为2，类型均为int类型的函数func

Function Type

函数类型(Function Type)

自己总结

函数类型说明了：1、函数返回值类型；2、参数数量和类型

Type-name(argument1-type, argument2-type…, argumentn-type)

func函数的函数类型是int(int, int)

函数类型也有与之对应的指针类型（pointer to function returning type）

给定一个函数类型：1、返回值类型为int；2、参数数量为2，类型均为int

int(int, int)

Referenced Type

int(*)(int, int)

Pointer Type

函数类型六元组

A: 0x00419850
Func_T: int(int,int)
N: func
S: N/A
V: 0x00419850
V_T: int(*)(int,int)

func函数类型
“六元组”

我们借用Object Type六元组的概念
设计一个Function Type六元组描述函数

1、A就是函数入口地址（假设是0x00419850)
2、Func_T就是函数类型
3、N是函数标识符
4、函数类型没有大小（sizeof无效）
5、表示值V和A一样
6、V_T是Func_T对应的指针

函数调用(Function Call)

```
int func(int a, int b)
{
    // do something
    return 0;
}

int main()
{
   func(10, 20);
   return 0;
}
```

1、func是一个标识符，是基础表达式，称为函数标识符（Function Designator)

2、函数标识符意味着函数类型

func → int(int, int)

函数标识符+(参数)：函数调用(Function Call)

func(10, 20)

A postfix expression followed by parentheses () containing a possibly empty, comma-separated list of expressions is a function call.

函数标识符（Function Designator）

```
int func(int a, int b)
{
    // do something
    return 0;
}

int main()
{
    func = null; ✕
    int(*p)(int, int) = func;
    int(*q)(int, int) = &func;
    return 0;
}
```

1、func不能充当lvalue，lvalue必须是Object Type

2、func函数标识符这个基础表达式的值：

<函数的地址，函数类型对应的指针类型>

func函数的函数类型对应的指针类型是int(*)(int, int)

A function designator with type "function returning type" is converted to an expression that has type "pointer to function returning type".

3、&func返回的是指向func函数的指针：

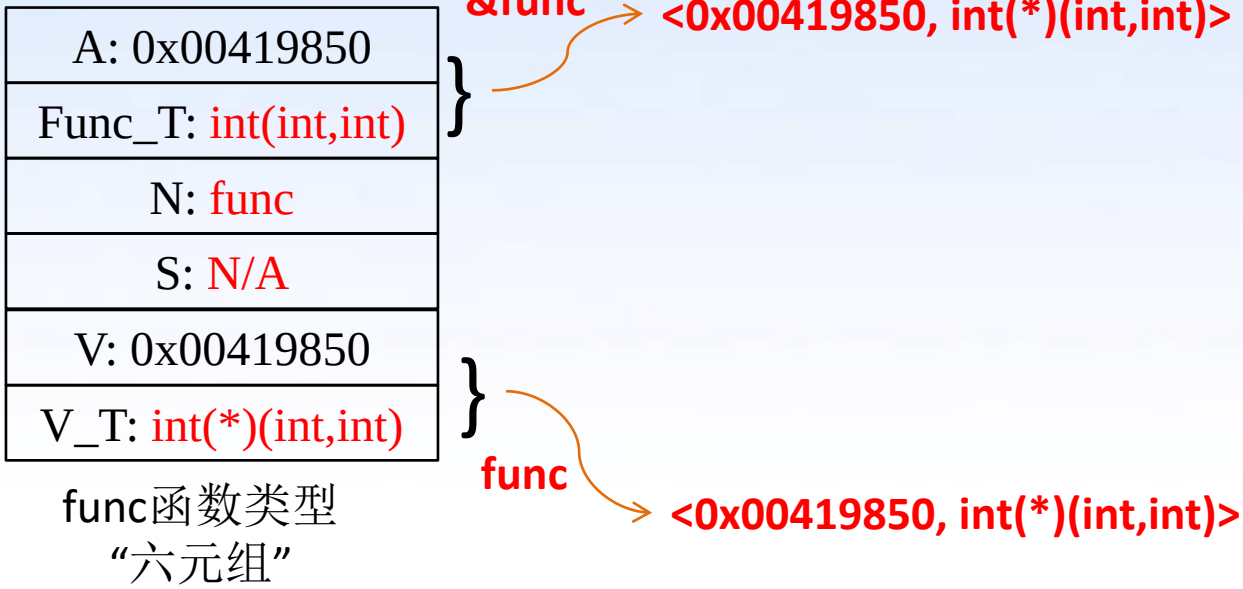
<函数的地址，函数类型对应的指针类型>

The result is a pointer to the function designated by its operand.

函数标识符（Function Designator）

```
int func(int a, int b)
{
    // do something
    return 0;
}

int main()
{
   int(*p)(int, int) = func;
   int(*q)(int, int) = &func;
   return 0;
}
```



函数指针也是一种指针

```
int func(int a, int b)
{
    // do something
    return 0;
}

int main()
{
   int(*p)(int, int) = func;
   int(*q)(int, int) = &func;
   int(**r)(int, int) = &p;
   return 0;
}
```

int(*)(int, int)也是一种指针类型，也有对应的指针类型

int(*)(int, int) ⟺ int(**)(int, int)

int(*s[5])(int, int)：变量s的类型是一个数组类型

5个元素，每个元素类型是int(*)(int, int)

```
typedef int(*FUNC)(int,int);
FUNC s[5];
```

*操作符引导函数指针变量的问题

```
int func(int a, int b)
{
    // do something
    return 0;
}

int main()
{
    int(*p)(int, int) = func;
    int(*q)(int, int) = p;
    int(*r)(int, int) = *p;
    return 0;
}
```

1、If the operand points to a function, the result is a function designator;

*p中，*引导的oprand是一个指向函数的指针
所以*p的结果就是函数标识符func

2、func函数标识符这个基础表达式的值：

<函数的地址，函数类型对应的指针类型>

指针类型即是int(*)(int, int)

*操作符引导函数指针变量的问题