Pass by Value

1. 理解任何表达式都有Value
2. 参数传递的到底是什么？

all expressions are evaluated as specified by the semantics

Evaluation of an expression in general includes both value computations and initiation of side effects

函数参数：实参 vs 形参

主函数

```
int main()
{
    func(exp);
}
```

函数调用

被调用函数

```
int func(Argument_Type Argument_Name)
{
    // do something
}
```

1、获得实参exp的返回值<V, V_T>

2、V写入形参Argument_Name对应的内存

V_T和Argument_Type必须适配

C语言参数传递的机制

Pass By Value

非数组变量作为函数参数

```
void func(int pa)
{
    // do something
}

int main()
{
    int a = 10;

    func(a);

    return 0;
}
```

1、func(a)中实参是a这个表达式，获得的是变量a对应内存的表示值，也就是<10, int>

2、将10传递给pa，也就是将10转换32位0/1串，放到变量pa对应的内存

main函数里面：
a: <10, int>

func函数里面：
pa: <10, int>

数组变量作为函数参数

```
void func1(int* pe)
{
    //do something
}

void func2(int (*pg)[3])
{
    //do something
}

int main()
{
    int e[2];
    int g[2][3];

    func1(e);
    func2(g);

    return 0;
}
```

main函数里面： → **func1和func2函数里面：**

e: <0x0076AB11, int*> → **pe: <**0x0076AB11, int*>

g: <0x0098B11D, int(*)[3]> → **pg: <**0x0098B11D, int(*)[3]>

形参中：**int[]= int*， int[][3]= int(*)[3]**

数组中第一维信息的丢失是C语言
所有数组类型变量内存的取值机制造成的

进一步思考二维数组的形参形式

```
void func(int g[][3])
{
    // do something
}

int main()
{
    int g[2][3] = {0};
    func(g);
    return 0;
}
```

```
void func(int* g, int row, int col)
{
    // do something
}

int main()
{
    int g[2][3] = {0};
    func((int*)g, 2, 3);
    return 0;
}
```

```
void func(int* g, int size)
{
    // do something
}

int main()
{
    int g[2][3] = {0};
    func((int*)g, 6);
    return 0;
}
```

使用int g[][3]作为形参，数字3需要写在参数上，扩展性较弱

int row, int col vs. int size

int** pg当形参能行吗？

```
void func(int** pg)
{
    pg[0][0] = 1;
}

int main()
{
    int g[2][3] = {0};
    func((int**)g);
    return 0;
}
```

注意：g[2][3]={0}, 假设g对应内存首地址为0x0098B11D

pg[0][0] = 1或**pg=1会有什么问题？

int** pg当形参能行吗？

```
void func(int** pg)
{
    pg[0][0] = 1;
}

int main()
{
    int g[2][3] = {0};
    func((int**)g);
    return 0;
}
```

注意g[2][3]={0}
g对应内存首地址
0x0098B11D

pg的内存六元组

A: 0x0036AA31
Obj_T: int**
N: pg
S: 4
V: 0x0098B11D
V_T: int**

取值

<0x0098B11D, int**>

*定位

*pg的内存六元组

A: 0x0098B11D
Obj_T: int*
N: N/A
S: 4
V: 0/NULL
V_T: int*

*pg的Value=0怎么来的？

<NULL, int*>

*定位

思考题

```
void func(int* pe)
{
    (pe+1)[1] = 1;
}

int main()
{
    int e[10] = {0};
    int i;

    func(e+1);

    for(i=0; i<10; i++)
        printf("e[%d]=%d\n", i, e[i]);

    return 0;

}
```

对左边这段代码，请问e[?]现在等于1
假设变量e对应的内存首地址为0x0076AB11

答案

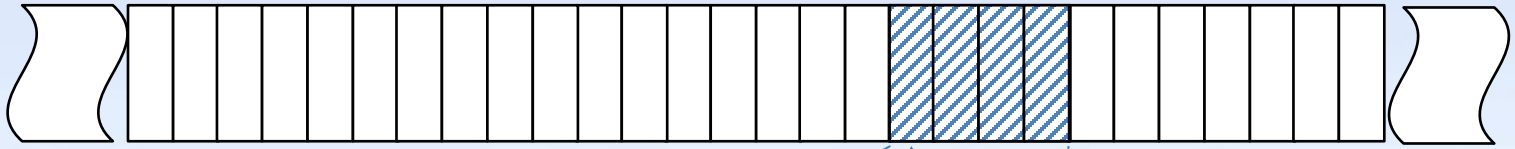
1、main函数中，e的返回值：<0x0076AB11, int*>
2、e+1的返回值：<0x0076AB15, int*>，传递给pe
3、形参pe的值：<0x0076AB15, int*>
4、(pe+1)的值：<0x0076AB19, int*>
5、(pe+1)[1]等价于*((pe+1)+1)，(pe+1)+1的值为：<0x0076AB1D, int*>
6、(pe+1)[1]定位的是0x0076AB1D开始的4个字节

e[3] = 1;

了解malloc

1. 如何确定malloc分配内存后返回的指针类型
2. malloc(sizeof(int)) vs. malloc(sizeof(int)*1)
3. malloc(sizeof(char)*8) vs. malloc(sizeof(char[4])*2)
4. 了解利用实际应用中malloc分配高维数组的方法

访问malloc分配的内存



0x00351728

malloc(4);

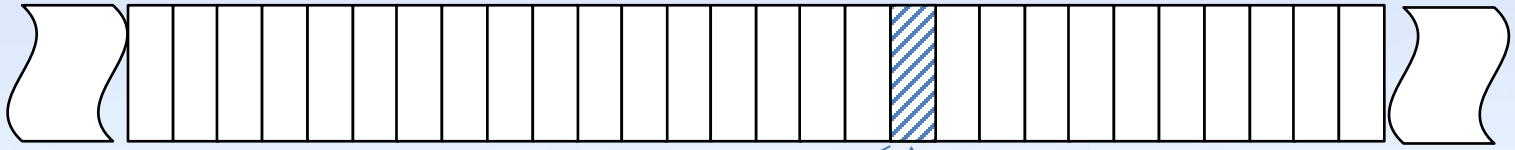
A: 0x00351728
Obj_T: N/A
N: N/A
S: 4
V: N/A
V_T: N/A

这块内存没有变量名称

无法通过变量名进行定位使用

malloc(4)分配的的内存六元组

访问malloc分配的内存



0x00351728

```
void* p = malloc(4);
```

p: <0x00351728, void*>

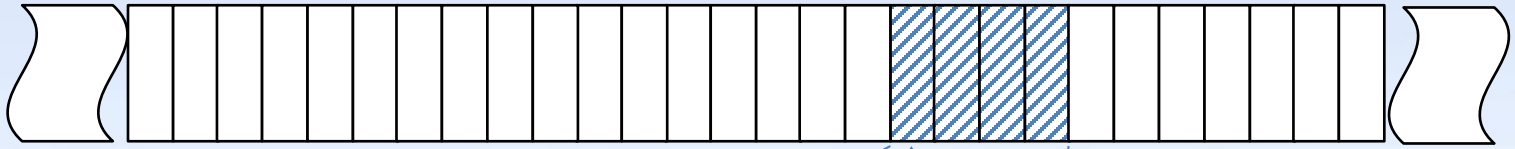
*p定位

A: 0x00351728
Obj_T: ?
N: N/A
S: ?
V: ?
V_T: ?

malloc返回的值需要强制转换成一个有意义的对象类型

*p无法定位有效内存

访问malloc分配的内存



0x00351728

int* p = (int*)malloc(4);

p: <0x00351728, int*>

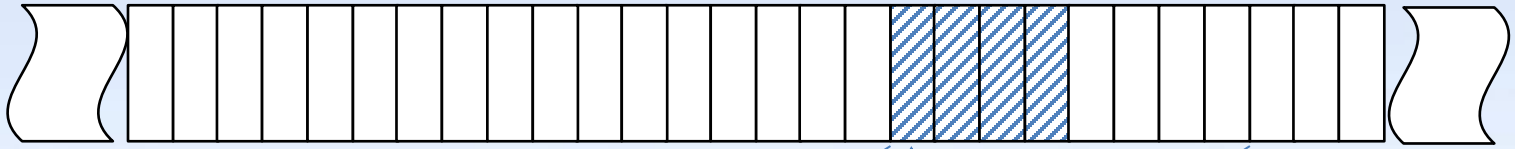
*p = 10;

*p定位

A: 0x00351728
Obj_T: int
N: N/A
S: 4
V: Undefined → V: 10
V_T: int

*p的内存六元组

访问malloc分配的内存



0x00351728

```
double* p = (double*)malloc(4);
```

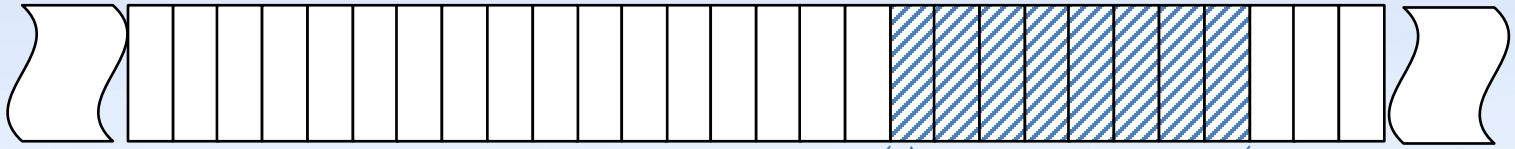
p: < 0x00351728, double*>

*p定位

A: 0x00351728
Obj_T: double
N: N/A
S: 8
V: ?
V_T: double

*p的内存六元组

访问malloc分配的内存



0x00351728

double* p = (double*)malloc(sizeof(double));

p: < 0x00351728, double*>

*p定位

利用sizeof计算待分配空间大小

A: 0x00351728
Obj_T: double
N: N/A
S: 8
V: ?
V_T: double

*p的内存六元组

理解malloc(sizeof(Obj_T)*N)

malloc(sizeof(int)*10)如何理解

int a[10];

变量a对应10个连续int组成的内存块，这块内存的表示值类型是int*

int* p = a;

malloc(sizeof(int)*10)申请10个连续int空间

语义可视为申请一个int[10]空间，则该空间表示值类型应该为int*

int* p = (int*)malloc(sizeof(int)*10);

malloc的形式化定义

假设一个对象类型Obj_T，malloc申请N个Obj_T大小的内存可形式化定义为

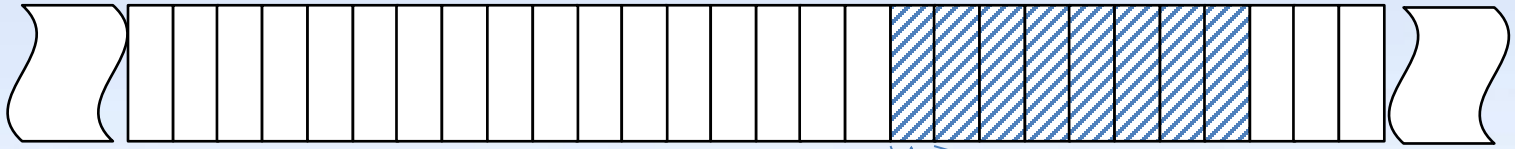
Obj_T* p = (Obj_T*)malloc(sizeof(Obj_T)*N)

这是最常见的分配一维数组的方法

可视为分配了一个Obj_T[N]对象类型空间

int* p = (int*)malloc(sizeof(int)*10)

访问malloc分配的内存（续）



0x00392B11

```
char* p = (char*)malloc(sizeof(char)*8);
```

p: < 0x00392B11, char*>

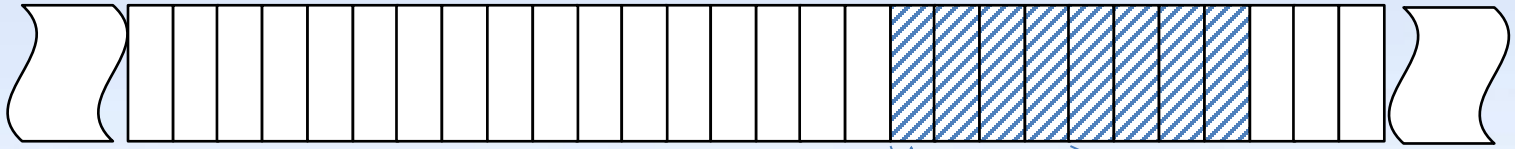
*定位

A: 0x00392B11
Obj_T: char
N: N/A
S: 1
V: ?
V_T: char

*p/p[0]的内存六元组

可视为分配一个char[8]类型变量的内存

访问malloc分配的内存（续）



0x00392B11

char (*p)[4] = (char(*)[4])malloc(sizeof(char[4])*2);

p: < 0x00392B11, char(*)[4]>

*定位

A: 0x00392B11
Obj_T: char[4]
N: N/A
S: 4
V: 0x00392B11
V_T: char*

*p/p[0]的内存六元组

可视为分配一个char[2][4]类型的空间

同样申请8个字节，区分指针类型的差异

char* vs. char(*)[4]

利用malloc直接分配高维数组的缺点

int (*p)[3][4] =(int(*)[3][4])malloc(sieof(int[3][4])*2);

指针类型为int(*)[3][4]，数字3和4需要写死在程序中，工程扩展性较差

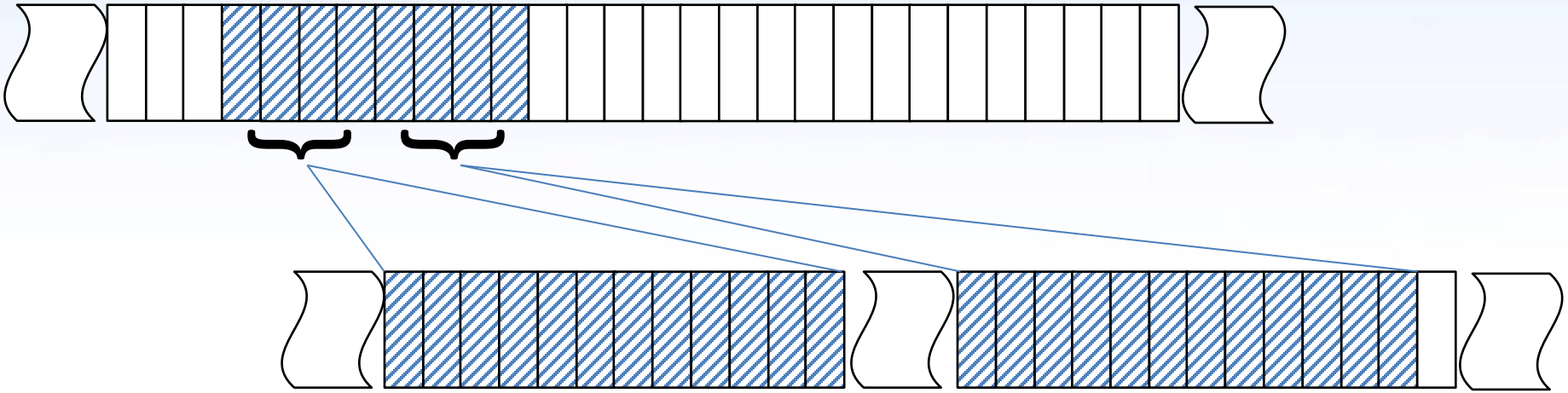
malloc更常用于分配一维数组

更常见malloc分配二维数组

```
int** pp = (int**)malloc(sizeof(int*)*2);

for(int i=0; i<2; i++) {
    pp[i] = (int*)malloc(sizeof(int)*3);
}
```

pp[i][j]保持二维数组形式



sizeof(int) vs. sizeof(int)*1

思考：

int* p = (int*)malloc(sizeof(int))

vs.

int* p = (int*)malloc(sizeof(int)*1)

malloc(sizeof(int))隐含的语义是分配了一个int[1]类型的空间

malloc分配的内存需要释放

1、通过变量声明分配出来的内存不需要释放

2、通过malloc分配出来的内存需要用free进行释放

```
int* p = (int*)malloc(sizeof(int)*4);
free(p);
```

执行了多少次malloc，就需要执行多少次free

Memory Leak是一个恒久的挑战！！！

思考题

利用malloc分配240个字节，如何定义指针变量p，让该240字节类型视为

1、char[240]

2、int[6][10]

3、int[3][4][5]

答案

1、char* p = (char*)malloc(sizeof(char)*240);

2、int (*p)[10] = (int(*)[10])malloc(sizeof(int[10])*6);

3、int (*p)[4][5] = (int(*)[4][5])malloc(sizeof(int[4][5])*3);

思考题

void* p = malloc(32);

1、int* q = (int*)p;
2、char* r = (char*)p;
3、int (*s)[4] = (int(*)[4])p;
4、char (*t)[2][4] = (char(*)[2][4])p;

请给出q+1, r+1, s+1, t+1的值，假设P的值是0x006E1410

答案
1、0x006E1414；2、0x006E1411；3、0x006E1420；4、0x006E1418

关于const

对象类型可以附加**const**, volatile, restrict等限定符（qualifiers)

unqulified type → qulified type

非数组对象类型

Obj_T → Obj_T const

const Obj_T

Obj_T const/const Obj_T
也是一种对象类型

数组对象类型

qualifiers do not have any direct effect on the array type itself

Obj_T const/const Obj_T
用来构造数组类型

我们先来看Obj_T const的方式

const修饰int类型

int const a = 10;

Obj_T： int const

V_T： int

对象类型是int const，但表示值类型是int

… qualified type, the value has unqualifed version …

A: 0x0028FF11
Obj_T: int const
N: a
S: 4
V: 10
V_T: int

const修饰int*类型

int a; int* const p = &a;

Obj_T： int* const

V_T： int*

A: 0x0046A521
Obj_T: int* const
N: p
S: 4
V: 0x0028FF11
V_T: int*

对象类型是int* const，表示值类型也是int*

… qualified type, the value has unqualifed version …

提示：把int*当作一个整体来看

int const对应的指针对象类型

int a; int const* p = &a;

Obj_T： int const*

V_T： int const*

对象类型是int const*，表示值类型是int const*

提示：把int const当作一个整体来看

A: 0x0046A521
Obj_T: int const*
N: p
S: 4
V: 0x0028FF11
V_T: int const*

int const用来构造数组对象类型

int const g[2][3] = {1, 2, 3, 4, 5, 6};

Obj_T：int const[2][3]

V_T：int const(*)[3]

对象类型是int const[2][3]
元素类型是int const[3]
表示值类型是int const(*)[3]

提示：把int const当作一个整体来看

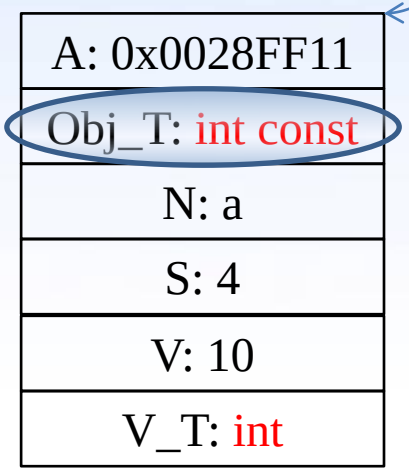
A: 0x0098B11D
Obj_T: int const[2][3]
N: g
S: 24
V: 0x0098B11D
V_T: int const(*)[3]

int const类型内存赋值

int const a = 10;

a = 20; ✕

定位内存



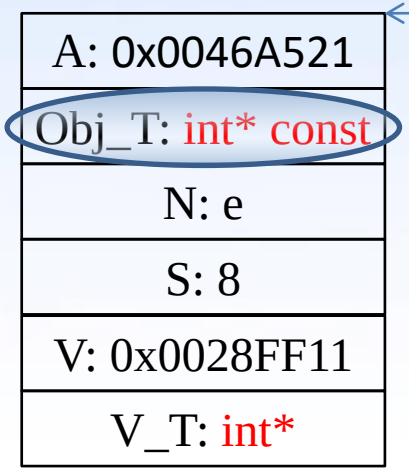
a的内存六元组

试图修改变量a对应的内存
都会导致编译错误

int* const类型内存赋值

int a = 10; int* const p = &a; p = NULL; ✗

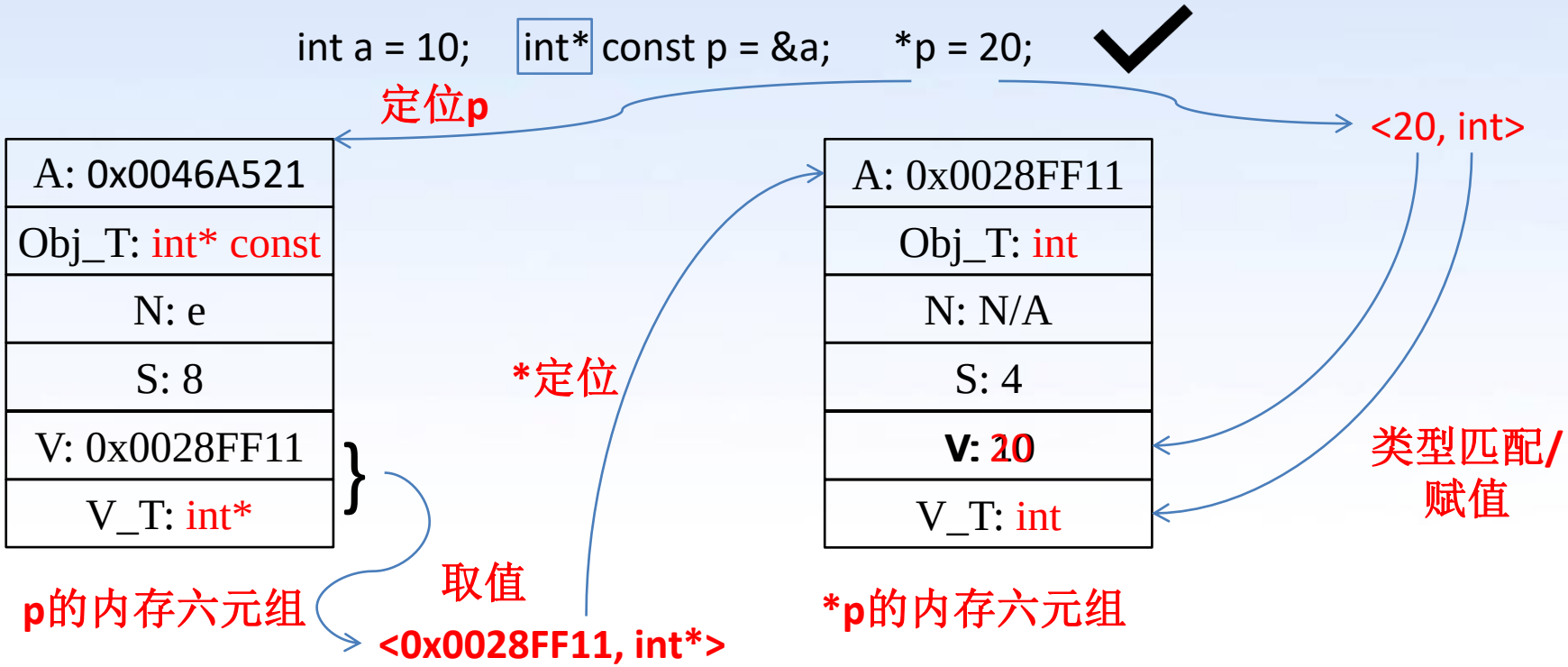
定位p



p的内存六元组

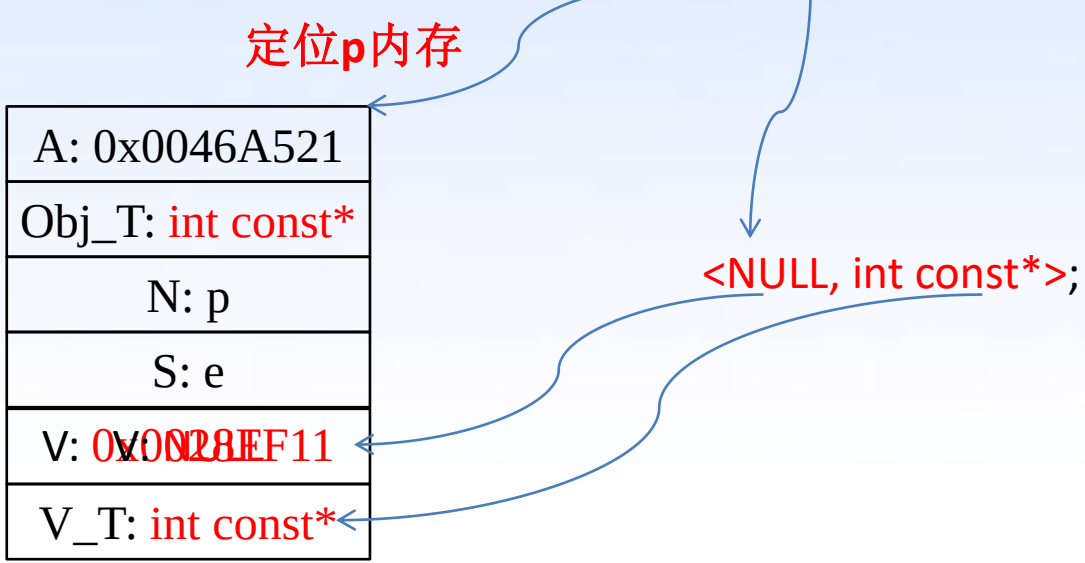
p对应的内存对象类型为int* const
试图修改变量p对应的内存都会导致编译错误

int* const类型内存赋值(续)



int const*类型内存赋值

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



p的内存六元组

提示：int const*是int const的指针类型

int const*类型内存赋值(续)

int a = 10; int const* p = &a; *p = 20; ✗

定位p内存

A: 0x0046A521
Obj_T: int const*
N: p
S: 4
V: 0x0028FF11
V_T: int const*

取值

<0x0028FF11, int const*>

*定位

A: 0x0028FF11
Obj_T: int const
N: N/A
S: 4
V: 10
V_T: int

试图修改*p对应的内存
都会导致编译错误

e的内存六元组

提示：int const*是int const的指针类型

*p的内存六元组

int* const vs. int const*

int a = 10;

int* const p = &a;

const修饰的是int*

typedef int* PINT;
PINT const p = &b;

*p = 20; ✓

p = NULL; ✗

int const* p = &a;

const修饰的是int

typedef int const CINT;
CINT* p = &b;

*p = 20; ✗

p = NULL; ✓

int const* const

int a = 10;

int const* const p = &a;

1、in const中const修饰int

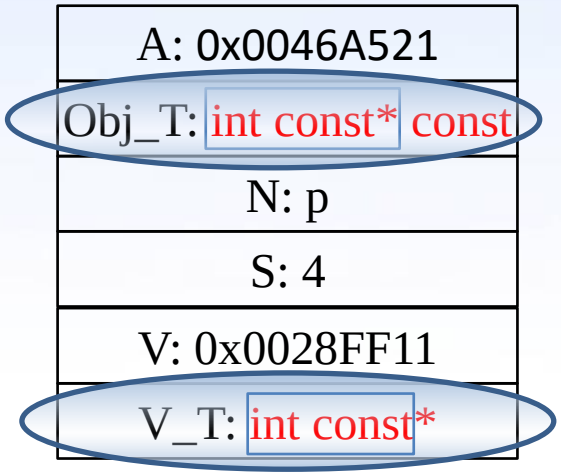
typedef int const CINT;

2、CINT*是CINT的指针类型

typedef CINT* PCINT;

3、PCINT const p = &a;

const修饰PCINT，也就是int const*



p = NULL;

*p = 20;

p的内存六元组

int const构造的数组对象类型赋值

int const e[2] = {1,2};　　e[0] = 20; ✕

定位e内存

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

取值

<0x0076AB11, int const*>

*定位

A: 0x0076AB11
Obj_T: int const
N: N/A
S: 4
V: 1
V_T: int

试图修改e[0]对应的内存
都会导致编译错误

e的内存六元组　　提示：int const*是int const的指针类型　　*e/e[0]的内存六元组

Obj_T const vs. const Obj_T

形式化定义上，Obj_T const N = const Obj_T N，但应用上是否有差异呢？

示例：Obj_T为int

int const a = 10;

无歧义

typedef int const CINT;

CINT a;

const int a = 10;

无歧义

typedef const int CINT;

CINT a;

a = 20; ✕

Obj_T const vs. const Obj_T

Obj_T const是一种对象类型，指向该数据类型的指针类型为Obj_T const*

const Obj_T是一种对象类型，指向该数据类型的指针类型为const Obj_T*

int const* p = &a;

无歧义

typedef int const CINT;

CINT* p = &a;

const int* p = &a;

无歧义

typedef const int CINT;

CINT* p = &a;

p = NULL; ✓

*p = 20; ✗

Obj_T const vs. const Obj_T

示例：Obj_T为int*

int* const p = &a;

无歧义

typedef int* PINT;

PINT const a;

p = NULL; ✗

*p = 20; ✓

const int* p = &a;

如何理解？

把const int看作一个整体？
还是把int*看作一个整体？

Obj_T const vs. const Obj_T

int* const p = &a;

P = NULL; ✗

*p = 20; ✓

const int* p = &a;

typedef const int CINT;
CINT* p = &a;

P = NULL; ✓

*p = 20; ✗

typedef int* PINT;
const PINT p = &a;

P = NULL; ✗

*p = 20; ✓

const int* p = &a;

P = NULL; ✓

*p = 20; ✗

Obj_T const vs. const Obj_T

int const* const p = &a;

p = NULL; ✗

*p = 20; ✗

const const int* p = &a;

p = NULL; ✓

*p = 20; ✗

const const int还是一个const int

Obj_T const vs. const Obj_T

int const* const p = &a;

p = NULL; ✗

*p = 20; ✗

typedef const int CINT;
typedef CINT* PCINT;

const PCINT p = &a;

p = NULL; ✗

*p = 20; ✗

C语言标准中对const位置的摆放没有明确语义规范
推荐使用Obj_T const N的形式