函数类型和函数指针类型

```
int a;
```

声明了一个int类型的对象a

Object Type

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

声明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)

函数类型

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对象类型（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(Function Type)

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

利用typedef进一步理解Function Type

typedef int (FUNC)(int, int)

Referenced Type

FUNC

int(int, int)

Pointer Type

FUNC*

int(*)(int, int)

函数指针也是一种合法的指针

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)：对象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];
```

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

思考题

```
int* foo(int* p, int* q)
{
    // do something
    return NULL;
}
```

该函数的函数类型是什么？
对应的指针对象类型是什么？

答案：
Function Type: int*(int*, int*)
Pointer Type: int*(*)(int*, int*)

函数与函数指针：挑战一下

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

```
int (*p)(int, int);
```

```
p = func;
p = &func;
p = *func;
p = **func;
p = ***func;
```

✓

```
func(2, 3);
(*func)(2, 3);
(**func)(2,3);
(&func)(2, 3);
```

✓

```
p(2, 3);
(*p)(2, 3);
(**p)(2, 3);
(&p)(2, 3);
```

✓ ✗

Logically Inconsistent?

函数类型七元组

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

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

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

函数类型“七元组”

我们借用对象七元组的概念
设计了一个函数七元组来描述函数

1、A：函数入口地址（假设是0x00419850)
2、Func_T：函数类型
3、N：函数名
4、S：函数类型没有大小（sizeof无效）
5、V：值，和A一样
6、V_T：Func_T对应的指针对象类型
7、Align：函数类型无对齐要求（alignof无效）

Function Designator

lvalue是可以定位一个对象的表达式

简单复习一下lvalue

对象标识符、数组下标运算表达式、*exp，String Literal、Compound Literal、指向结构体/联合体的lvalue.member，指向结构体/联合体指针表达式->member

同样的，Function Designator是可以定位一个函数的表达式

Function Designator

C语言中有两种Function Designator

1、Function Identifier（函数标识符）

2、如果一个表达式exp，其rvalue类型是函数指针类型，则*exp是一个合法的Function Designator

我们先来看第一种，函数标识符

Function Identifier

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

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

1、func是一个函数标识符，是基础表达式

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

Function Identifier

func定位的函数

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

func; → **对func做evaluate**

sizeof(func); → **不合法**

typeof(func);
&func; } **对func不做evaluate**

这些func相关的表达式是如何evaluate的呢？

typeof(func)不是表达式

当func被evaluate的时候

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

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

函数七元组

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

func;

<Value, Value_Type>

func: <0x00419850, int(*)(int, int)>

int (*x)(int, int) = func;

sizeof(func)

INVALID

函数不是对象，没有大小

typeof(func)

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

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

函数七元组

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

typeof(func);

等价于Func_T

typeof(func)等价于int(int, int)

typeof(func)不是表达式

typeof(func) x; ✗

typeof(func)* x; ✓

&func

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

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

函数七元组

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

&func;

<Address, Func_T*>

&func: <0x00419850, int(*)(int, int)>

注意rvalue的类型Func_T*

int (*x)(int, int) = &func;

func vs. &func

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

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

函数七元组

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

&func;
<Address, Func_T*>

func;
<Value, Value_Type>

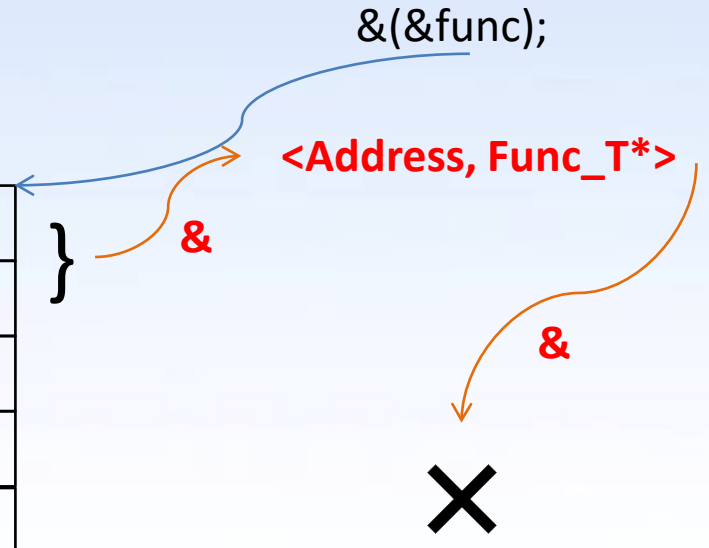
Evaluate的结果都是
<0x00419850, int(*)(int, int)>

&(&func)

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

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

函数七元组

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



&func不是lvalue，也不是function designator

alignof(func), alignof(typeof(func))

INVALID

Function Designator不能跟alignof结合

函数类型不能跟alignof结合

Function Designator

C语言中有两种Function Designator

1、Function Identifier（函数标识符）

2、如果一个表达式exp，其rvalue类型是函数指针类型，则*exp是一个合法的Function Designator

我们再来看第二种，*exp作为Function Designator

func被evaluate的rvalue是函数指针

*func;

函数七元组

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

<0x00419850, int(*)(int, int)>

*定位

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

func和*func表达式定位的函数是一样的

*exp的规则

1、Value -> Address

2、V_T -> Referenced Type as Funct_T

&func被evaluate的rvalue是函数指针

*(&func);

函数七元组

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

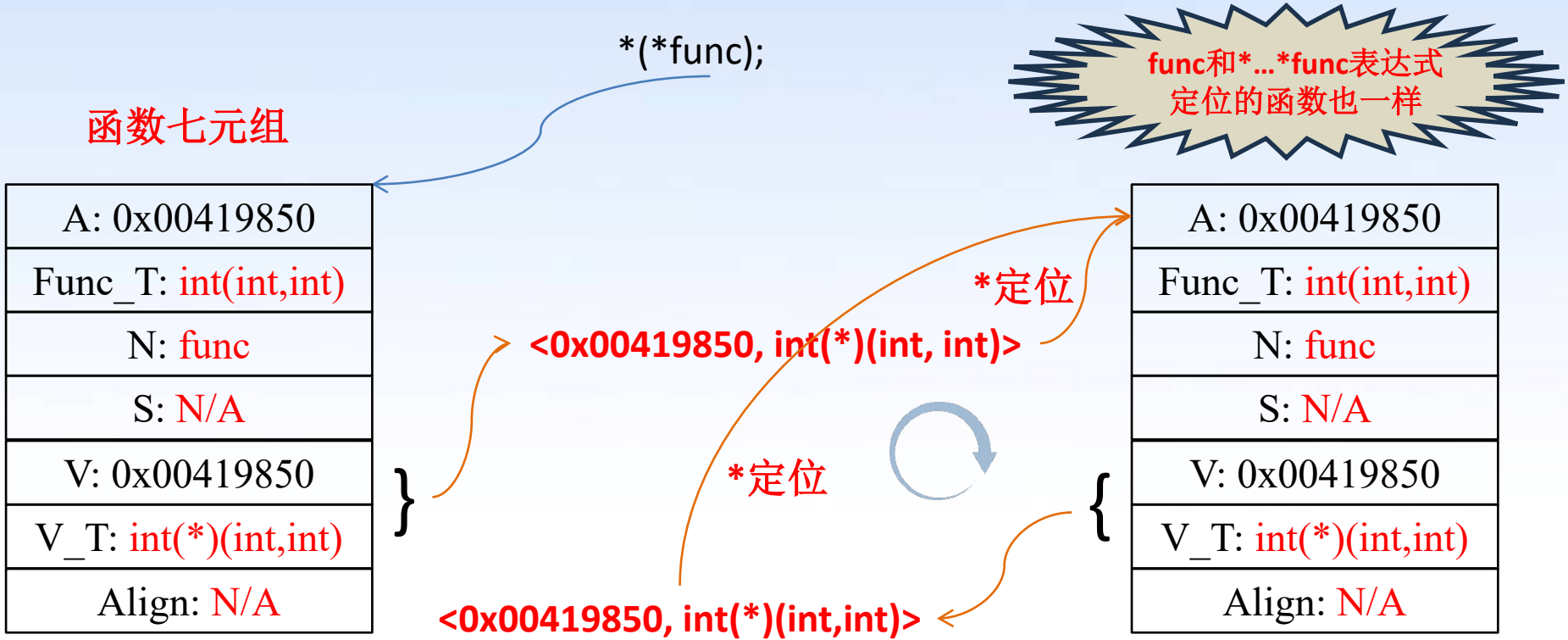
<0x00419850, int(*)(int, int)>

*定位

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

func和*(&func)表达式
定位的函数也一样

*func被evaluate的rvalue是函数指针



函数指针对象

int (*p)(int, int) = func;

A: 0x0098FE10
Obj_T: int(*)(int, int)
N: p
S: 4
V: 0x00419850
V_T: int(*)(int, int)
Align: 4

<0x00419850, int(*)(int, int)>

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

函数指针对象

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

int (*p)(int, int);
```

```
p = func;

p = *func;

p = **func;

p = ***func;

p = &func;
```

func, *func, **func, ***func这些表达式都是Function Designator

Function Designator没有跟typeof，&结合的时候，如果被evaluate，则值为指向该函数的指针

&func表达式返回值也是指向该函数的指针

func，*func, **…func和&func结果一样，但evalute取值的路径不同**

函数指针对象

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

int (*p)(int, int);
```

p = func; ✓

p = *func; ✓

p = **func; ✓

p = ***func; ✓

p = &func; ✓

p = &&func; ✗

函数指针对象

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

int (*p)(int, int) = func;
```

对象p

A: 0x0098FE10
Obj_T: int(*)(int, int)
N: p
S: 4
V: 0x00419850
V_T: int(*)(int, int)
Align: 4

&p;

<Address, Obj_T*>

&

<0x0098FE10, int(**)(int, int)>

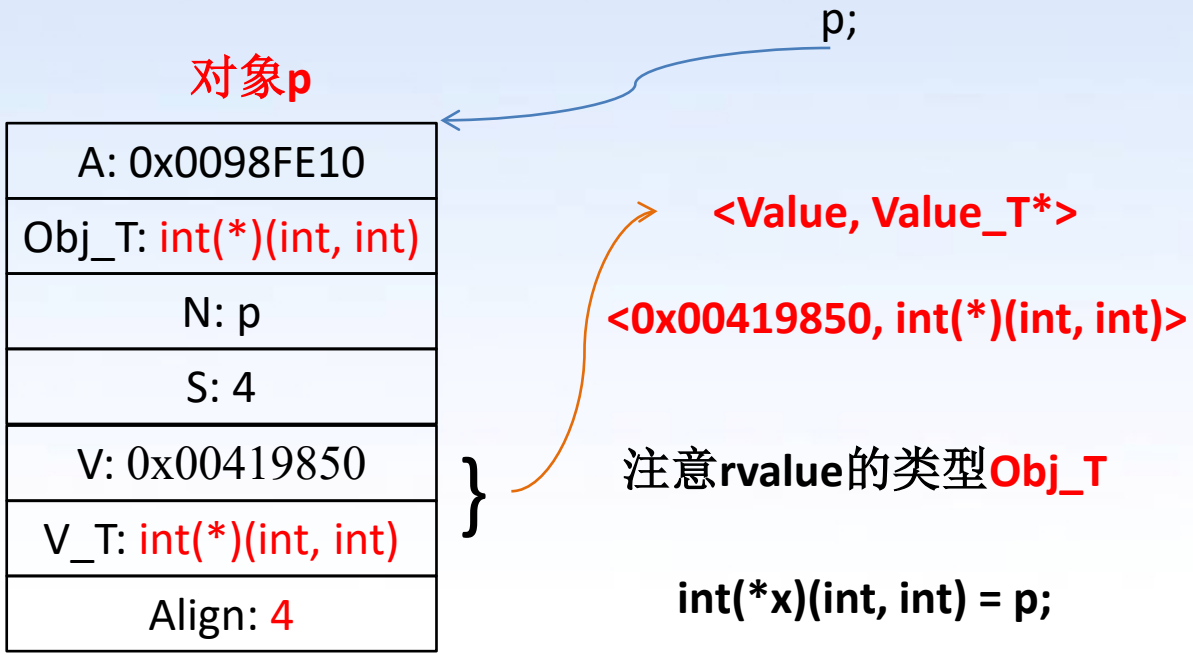
注意rvalue的类型Obj_T*

int(*)(int, int)是合法的对象类型

int(**x)(int, int) = &p;

函数指针对象

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

int (*p)(int, int) = func;
```



*p, **p, ***p…

**…*p;

**…*p定位的函数都一样

对象p

A: 0x0098FE10
Obj_T: int(*)(int, int)
N: p
S: 4
V: 0x00419850
V_T: int(*)(int,int)
Align: N/A

<0x00419850, int(*)(int, int)>

*定位

*定位

<0x00419850, int(*)(int,int)>

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

总结一下：函数和函数指针

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

int (*p)(int, int) = func;
```

func, *func, **func, ***…***func表达式都是Function Designator

*p, **p, ***p, ***…***p这些表达式也都是Function Designator

Function Designator进行evaluate之后值为指向func的指针

p和&func不是Function Designator，但evaluate的值为指向func的指针

但是函数标识符func本身不是函数指针
Evaluate是这些诡异现象的核心机制

函数调用(Function Call)

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

int main()
{
   exp(arg1, arg2);
   return 0;
}
```

如何调用函数func呢？

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

1、对exp表达式evaluate之后rvalue是指向func函数的指针，且
2、arg1和arg2进行evaluate之后rvalue类型符合func函数对应参数的对象类型，则

exp(arg1, arg2)称为函数func的函数调用

函数调用(Function Call)

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

```
func(2, 3);

(*func)(2, 3);
(**func)(2,3);

(&func)(2, 3);
```

```
p(2, 3);

(*p)(2, 3);
(**p)(2, 3);

(&p)(2, 3);
```

func, *func, **func, &func, p, *p, **p这些表达式evaluate的结果都是

<0x00419850, int(*)(int,int)>

&p这个表达式evaluate的结果是

<0x0098FE10, int(**)(int, int)>

思考题

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

以下语句是否合法，为什么？

1、func = NULL;

2、(1?func:NULL)(2, 3);

3、&func(2, 3);

答案：

1、错，func不是lvalue

2、正确，1?func:NULL这个表达式返回值是指向func函数的指针

3、错误，&func(2, 3)是&(func(2, 3)), func(2, 3)返回的是rvalue，不是合法的lvalue，注意与(&func)(2, 3)的区别。

&exp是一元表达式，func(2, 3)是后缀表达式，后缀表达式优先级比一元表达式高

理解函数调用中参数的Pass by Value

参数传递的到底是什么？

all expressions are evaluated as specified by the semantics

函数中的实参也要按要求进行evaluate

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

非数组对象作为函数参数

```
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，获得的是对象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，放到对象pa对应的内存

main函数里面：
a: <10, int>

func函数里面：
pa: <10, int>

数组对象作为函数参数

```
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：

e: <0x0082FB10, int*> ⟶ **pe: <** 0x0082FB10, int*>

g: <0x0082FB30, int(*)[3]> ⟶ **pg: <** 0x0082FB30, int(*)[3]>

func1和func2函数里面：

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

数组中第一维信息的丢失是C语言
所有数组类型对象的取值机制造成的

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0x0082FB30

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
0x0082FB30

pg的内存七元组

A: 0x0036AA20
Obj_T: int**
N: pg
S: 4
V: 0x0082FB30
V_T: int**
Align: 4

取值

<0x0082FB30, int**>

*定位

*pg的内存七元组

A: 0x0082FB30
Obj_T: int*
N: N/A
S: 4
V: 0/NULL
V_T: int*
Align: 4

*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
假设对象e对应的内存首地址为0x0061FDF0

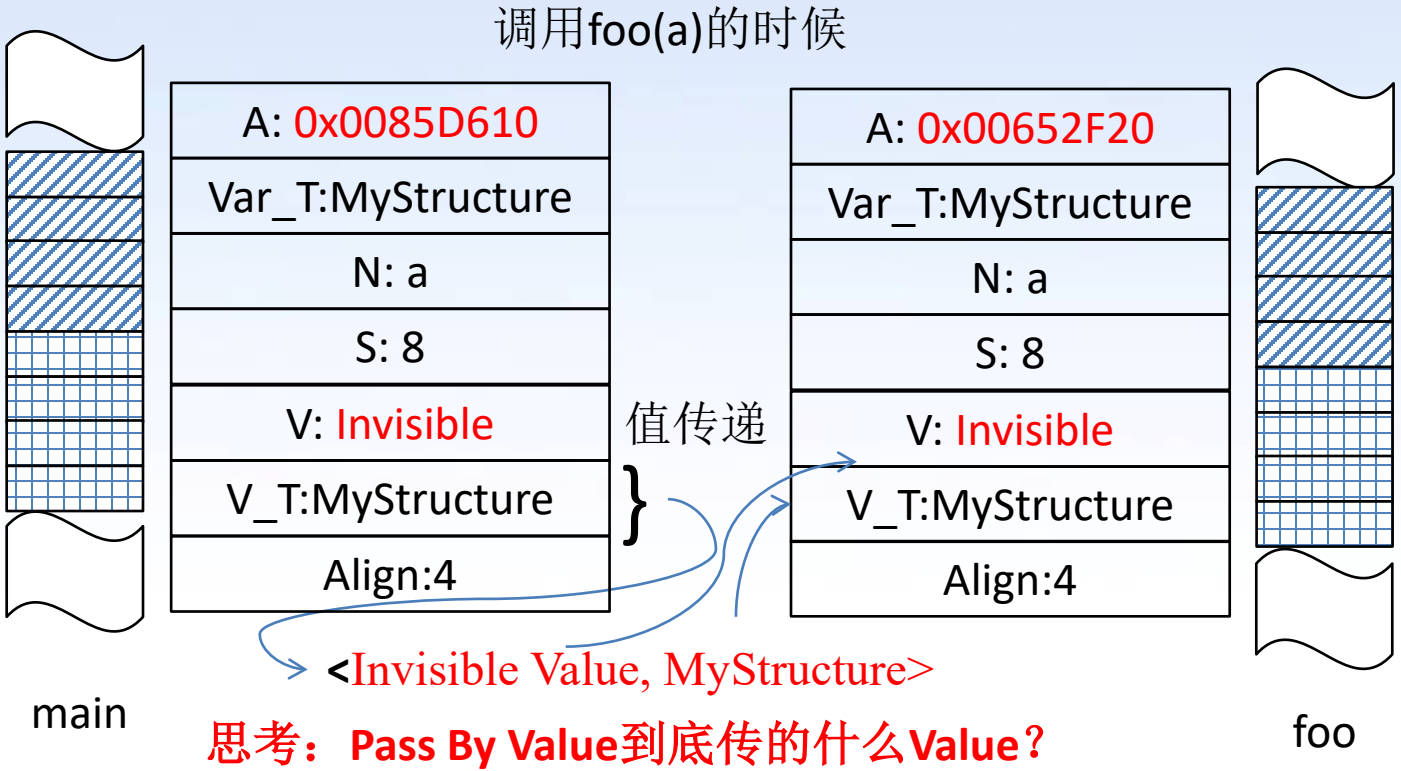
答案

1、main函数中，e的返回值：< 0x0061FDF0, int*>
2、e+1的返回值：< 0x0061FDF4, int*>，传递给pe
3、形参pe的值：< 0x0061FDF4, int*>
4、(pe+1)的值：< 0x0061FDF8, int*>
5、(pe+1)[1]等价于*((pe+1)+1)，(pe+1)+1的值为：< 0x0061FDFC, int*>
6、(pe+1)[1]定位的是0x0061FDFC开始的4个字节

e[3] = 1;

结构体作为函数参数

```
typedef struct _MyStructure{
    int a;
    int b;
} MyStructure;

int foo(MyStructure a)
{
    a.a++;
}

int main()
{
    MyStructure a;

    a.a = a.b = 0;

    foo(a);

    printf("a.a=%d\n", a.a);

    return 0;
}
```



关于结构体lvalue的问题

```
typedef struct _MyStruct {
    int a;
    int b;
} MyStruct;

MyStruct foo()
{
    MyStruct m;

    m.a = 5;
    m.b = 10;

    return m;
}
```

对于foo函数中的对象m

1、m是lvalue吗？
2、m.a是lvalue吗？
3、m.b是lvalue吗？

m、m.a、m.b都是lvalue
&m, &m.a, &m.b都是合法的

A postfix expression followed by the . operator and an identifier designates a member of a structure or union object. The value is that of the named member, and is an lvalue if the first expression is an lvalue.

A postfix expression followed by the-> operator and an identifier designates a member of a structure or union object. The value is that of the named member of the object to which the first expression points, and is an lvalue.

关于结构体lvalue的问题（续）

```
typedef struct _MyStruct {
    int a;
    int b;
} MyStruct;

MyStruct foo()
{
    MyStruct m;

    m.a = 5;
    m.b = 10;

    return m;
}
```

```
int main()
{
    MyStruct m = foo();

    printf("a=%d, b=%d\n", foo().a, foo().b);

    return 0;
}
```

foo()是lvalue吗？　　不是

If f is a function returning a structure or union, and x is a member of that structure or union, f().x is a valid postfix expression but is not an lvalue.

限定符在函数参数中的作用

```
void func(int* p)
{
    *p = 1;
}

int main()
{
    int a;
    func(&a);
    return 0;
}
```

main函数中的a现在被成功修改为1了

如果不允许func函数中对a的值进行修改呢？

```
void func(int const* p)
{
    *p = 1;
}
```

限定符在函数参数中的作用

```
void func(int const* p)
{
    *p = 3;
}

int main()
{
    int e[2]={1, 2};

    func(e);

    return 0;
}
```

int const*对于数组对象作为参数非常有效

了解restrict限定符

char str[100] = “hello”;
strcpy(str, str);
strcpy(str, str+1);

这两个字符串拷贝的语句会导致未定义行为，为什么？

If copying takes place between objects that overlap, the behavior is undefined.

```
#include <string.h>
char *strcpy(char * restrict s1, const char * restrict s2);
```

了解restrict

restrict也是一种限定符，只能用来修饰指针类型

T* restrict O; 注意这里restrict放在T*的右边

int* restrict p; ✓

vs.

restrict int* p; ✗

typedef int* PINT;

PINT restrict p; vs. restrict PINT p;

可以参考此前介绍const限定符的内容

restrict修饰int*类型

int a; int* restrict p = &a;

Obj_T： int* restrict

V_T： int*

对象类型是int* restrict，表示值类型也是int*

… qualified type, the value has unqualifed version …

int* restrict* q = &p;

typedef int* PINT;
typedef PINT restrict RPINT;
RPINT* q = &p;

A: 0x0045B810
Obj_T: int* restrict
N: p
S: 4
V: 0x0028FF10
V_T: int*
Align: 4

了解Based On

T* restrict O;

标识符O能够定位一个对象Obj（对象类型为T* restrict的内存块）

1、Obj的值可以用来定位一个数组中的元素对象（指针总是蕴含着数组访问）

例如*O, *(O+1), O[n]

2、给定指针表达式**E**，如果Obj值修改，用于定位一个新的数组中各元素对象（即便这个新的数组对象各元素对象是Obj原始值可以定位的数组各元素对象的拷贝），表达式E的值也会被修改，则

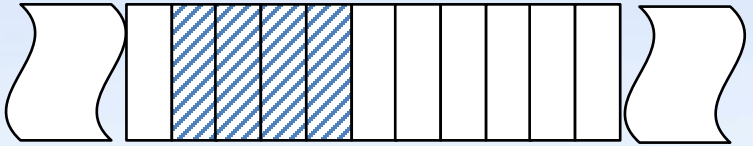
E **Based on** Obj

了解Based On：示例

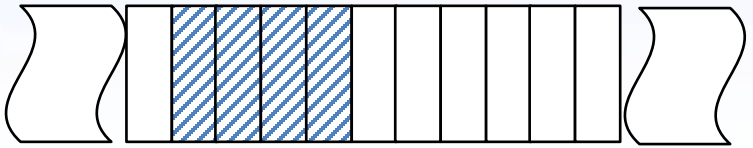
```
char e[4]={0};
char g[4]={0};

char* restrict q = e;
```

q, q+1, q+2这些表达式
based on对象q

因为执行q = g;之后
q, q+1, q+2的值也会改变



对象e



对象g

了解Based On：示例

int* restrict p;

标识符p定位一个对象int* restrict类型的对象

不失一般性，如果把这个对象称为对象p

表达式p+n的值是跟对象p的值紧密相关，对象p的值修改，p+n的值必然修改

表达式p based on 对象p

表达式p+1 based on 对象p

表达式&(p[n])的值也跟对象p的值紧密相关，对象p的值修改，p[n]这个左值表达式的地址必然也会修改

表达式&(p[n]) based on 对象p

表达式p vs. 对象p

了解Based On

int** restrict p;

不失一般性，这里用p指代p指向的对象

指针表达式p based on 对象p
指针表达式p+1 Based on 对象p

指针表达式p[0], p[1]不是based on p
指针表达式p[0], p[0]+1是based on p[0]指向的对象
指针表达式p[1], p[1]+1是based on p[1]指向的对象

E depends on **P** itself rather than on the value of an object referenced indirectly through P.

了解Based On：示例

```
char e[4]={0};
char* g[4]={e, e+1, e+2, e+3};
char* h[4]={e, e+1, e+2, e+3};

char** restrict q = g;
```

q[0], q[1], q[2]这些表达式
不是based on对象q

因为执行q = h;之后
q[0], q[1], q[2]的值不会改变

restrict的作用

在一个Block里面

如果有一个左值表达式L，其地址&L是Based on一个对象P

该左值表达式L其定位的对象假设为X，如果X对象的值被修改

有另一个左值表达式M能访问X，则M的地址也必须based on对象P

int* restrict p; 标识符p定位的对象，不失一般性称为**对象p**
左值表达式p[2]（也就是L）的地址（&(p[2]) 或p+2）是Based on对象p
p[2]这个左值表达式定位的这个对象称为X
如果对象X的值会被修改
任何访问对象X的其他左值表达式M，&M必须based on对象p
例如(p+1)[1]也能访问X，这个表达式的地址也是based on对象p

restrict的示例

```
int foo(int* restrict p, int* restrict q)
{
    *p = 10;
    *q = 20;

    return (*p+*q);
}

int main()
{
    int a = 10;

    printf("%d\n", foo(&a, &a));

}
```

1、在foo函数中，标识符p和q分别对应一个对象，不失一般性，称为对象p和对象q

2、*p和*q是两个左值表达式，其地址分别是&(*p)和&(*q)，分别based on对象p和q

3、*p定位的对象假设为X，如果*q要访问的对象也是X，则&(*q)表达式必须也based on p

4、&(*q)是based on q，因此编译器就能推断出*p和*q定位的这两个对象肯定不是同一个

5、这个例子中*p和*q都定位了main函数中同样的对象a，因此是未定义行为

restrict的示例

```
int foo(int* restrict p, int* restrict q)
{
    *p = 10;
    *q = 20;

    return (*p+*q);
}
```

```
int foo(int* p, int* q)
{
    *p = 10;
    *q = 20;

    return (*p+*q);
}
```

foo函数中*p和*q对应不同对象由谁保证？

由程序员保证

区别在哪呢？

主要就是为了编译器优化的效率问题，如果*p和*q肯定不是一个对象，则可以优化的空间就大大增加了

之前举例的strcpy是系统库函数，效率极为重要，因此引入restrict关键字加强优化

程序员保证restrict正确使用的示例

```
void f(int n, int * restrict p, int * restrict q)
{
      while (n--  >  0)
            *p++ = *q++;
}
```

```
void g(void)
{
      extern int d[100];
      f(50, d + 50, d); // valid
      f(50, d +  1, d); // undefined behavior
}
```

在f(50, d+50, d)这个函数调用中
d[50]到d[99]对象会被修改，且based on p based on q的表达式不会访问d[50]到d[99]
因此是确定性行为

在f(50, d+1, d)这个函数调用中
d[1]到d[50]对象会被修改，且based on p based on q的表达式会访问其中d[1]到d[49]
导致未定义行为

程序员保证restrict正确使用的示例

```
void h(int n, int * restrict p, int * restrict q, int * restrict r)
{
      int i;
      for (i = 0; i  <  n; i++)
            p[i] = q[i] + r[i];
}
```

假设a和b是不相交的两个数组，h(100, a, b, b);会不会导致未定义行为呢？

不会

虽然q和r都访问b数组中的对象，但是由于没有对b数组元素进行修改，因此不会造成未定义行为

程序员保证restrict正确使用的示例

```
{
      int * restrict p1;
      int * restrict q1;
      p1 = q1; // undefined behavior
      {
            int * restrict p2 = p1; // valid
            int * restrict q2 = q1; // valid
            p1 = q2;                // undefined behavior
            p2 = q2;                // undefined behavior
      }
}
```

在同一个scope里面，将两个restrict修饰的指针进行赋值，会导致未定义行为

在outer-to-inner的情况下，将outer的restricted指针赋值给inner的restrict指针是合法的

总结一下restrict

restrict也是一种限定符，只能用来修饰指针类型

T* restrict O

An object that is accessed through a restrict-qualified pointer has a special association with that pointer. This association requires that all accesses to that object use, directly or indirectly, the value of that particular pointer.

The intended use of the restrict qualifier is to promote optimization

A translator is free to ignore any or all aliasing implications of uses of restrict

指针总是蕴含着数组访问！！！