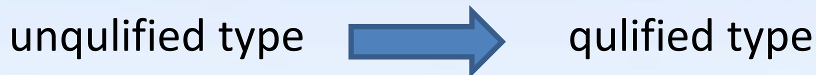
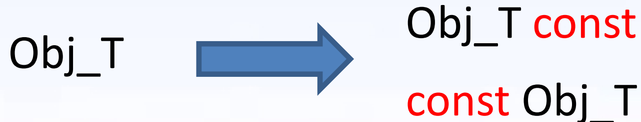
了解Qualified Type：const

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



非数组对象类型



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

为什么？

我们通过例子来进一步理解

const修饰int类型

```
int const a = 10;
```

Obj_T： int const

V_T： int

对象类型是int const，
但值类型是int

A: 0x0061FE10
Obj_T: int const
N: a
S: 4
V: 10
V_T: int
Align: 4

… qualified type, the value has unqualifed version …

对于const, volatile, and restrict，”The qualified or unqualified versions of a type are distinct types that belong to the same type category and have the same representation and alignment requirements”

const修饰int*类型

int a; int* const p = &a;

Obj_T： int* const

V_T： int*

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

对象类型是int* const，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

typedef int* PINT

const修饰int*类型

int a; PINT const p = &a;

Obj_T： PINT const

V_T： PINT

对象类型是PINT const，值类型也是PINT

… qualified type, the value has unqualifed version …

PINT等价于int*

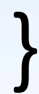
A: 0x0068FE10
Obj_T: PINT const
N: p
S: 4
V: 0x0061FE10
V_T: PINT
Align: 4

回顾一下Pointer Type

给定任何一个Type (T)，都有对应的一个指针类型Pointer to T

1、Object Type

2、Function Type



Referenced Type

Pointer Type

Referenced Type	Pointer Type
int	int*
int[5]	int(*)[5]
int*	int**
int const	int const*

int const也是一个合法的对象，自然也有对应的指针类型

思考题

1、5个int const*对象构成的数组对象类型是什么？

2、这个数组对象对应的指针对象类型是什么？

答案：

1、int const*[5]

2、int const*(*)[5]

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*，值类型是int const*

提示：把int const当作一个整体来看

typedef int const CINT

A: 0x0078FE10
Obj_T: int const*
N: p
S: 4
V: 0x0061FE10
V_T: int const*
Align: 4

int const对应的指针对象类型

int a; CINT* p = &a;

Obj_T： CINT*

V_T： CINT*

对象类型是CINT*，值类型是CINT*

CINT等价于int const

A: 0x0078FE10
Obj_T: CINT*
N: p
S: 4
V: 0x0061FE10
V_T: CINT*
Align: 4

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]
值类型是int const(*)[3]

提示：把int const当作一个整体来看
int const ⇔ CINT

A: 0x0082FB30
Obj_T: int const[2][3]
N: g
S: 24
V: 0x0082FB30
V_T: int const(*)[3]
Align: 4

int const用来构造数组对象类型

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

Obj_T： CINT[2][3]

V_T： CINT(*)[3]

对象类型是CINT[2][3]
元素类型是CINT[3]
值类型是CINT(*)[3]

int const ⇔ CINT

A: 0x0082FB30
Obj_T: CINT[2][3]
N: g
S: 24
V: 0x0082FB30
V_T: CINT(*)[3]
Align: 4

const直接限定数组对象类型呢？

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

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

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

等价于先观察到int const，继而观察到int const[2][3]

int const类型对象赋值

`int const a = 10;`　　`a = 20;`　✕

定位内存

A: 0x0061FE10
Obj_T: int const
N: a
S: 4
V: 10
V_T: int
Align: 4

试图修改对象a对应的内存都会导致编译错误

如果一个lvalue定位的对象，其对象类型为const限定类型，则该lvalue为不可修改左值

注意：a不是常量

int* const类型对象赋值

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

定位p

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

p对应的内存对象类型为int* const

p是不可修改的lvalue

int* const类型对象赋值(续)

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

✓

定位p

<20, int>

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

取值

<0x0061FE10, int*>

*定位

A: 0x0061FE10
Obj_T: int
N: N/A
S: 4
V: 20
V_T: int
Align: 4

类型匹配/赋值

int const*类型对象赋值

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

定位p内存

A: 0x0068FE10
Obj_T: int const*
N: p
S: e
V: 0x0068FE10 V: NULL
V_T: int const*
Align: 4

<NULL, int const*>;

提示：int const*是int const的指针类型

int const*类型对象赋值(续)

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

定位p内存

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

取值

<0x0061FE10, int const*>

*定位

A: 0x0061FE10
Obj_T: int const
N: N/A
S: 4
V: 10
V_T: int
Align: 4

*p定位对象的类型为int const
*p为不可修改的lvalue

提示：int const*是int const的指针类型

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*

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

p = NULL;

*p = 20;

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: 0x0082FB10
Obj_T: int const[2]
N: e
S: 8
V: 0x0082FB10
V_T: int const*
Align: 4

取值

<0x0082FB10, int const*>

*定位

A: 0x0082FB10
Obj_T: int const
N: N/A
S: 4
V: 1
V_T: int
Align: 4

e[0]定位对象的类型为int const
e[0]为不可修改的lvalue

提示：int const*是int const的指针类型

复习：不可修改左值

int const e[2] = {1,2};

e=NULL; e对应的对象类型为数组对象类型，不可修改左值

e[0] = 20; e[0]对应的对象类型为const限定类型，不可修改左值

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

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

无歧义

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

```
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;	const int* p = &a;		const int* p = &a;
	typedef const int CINT; CINT* p = &a;	typedef int* PINT; const PINT p = &a;	
P = NULL;	P = NULL;	P = NULL;	P = NULL;
*p = 20;	*p = 20;	*p = 20;	*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 O的形式

int const a=1; int* p=(int*)&a; *p=10;

a是不可修改左值，所以a=10会出现编译错误

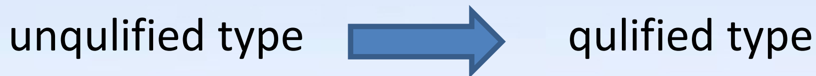
&a返回值类型应该int const*

强制转换成int*并赋值给p

*p=10会发生什么？

这是一个未定义行为

了解Qualified Type：volatile



给定一个对象类型Obj_T和对象名称O，声明volatile对象的语法为

Obj_T volatile O 或 volatile Obj_T O

int volatile a 或 volatile int a

Obj_T volatile/volatile Obj_T是一个新的类型

volatile修饰int类型

int volatile a = 10;

Obj_T： int volatile

V_T： int

对象类型是int volatile，但值类型是int

... qualified type, the value has unqualifed version ...

A: 0x0061FE10
Obj_T: int volatile
N: a
S: 4
V: 10
V_T: int
Align: 4

int volatile的含义

```
int volatile a = 10;
```

A: 0x0061FE10
Obj_T: int volatile
N: a
S: 4
V: 20
V_T: int
Align: 4

volatile修饰内存的值可能会以未知的方式发生变化

An object that has volatile-qualified type may be modified in ways unknown to the implementation

例如：代码没有进行修改，但值变成了20

谁改的呢？

volatile的应用场景示例

1、Memory-Mapped Input/Output (MMIO)端口对应的对象

2、异步中断函数访问的对象

MMIO场景下的应用

MMIO中，内存和I/O设备共享同一个地址空间。

会给各种I/O设备预留出相应的地址区域

一个地址可能访问内存，也可能访问某个I/O设备

1、假设0x12340000是某一个I/O设备对应的映射地址

2、假设这个地址开始的I/O设备对应的对象是int类型

3、将这个地址开始的I/O设备对应对象值定义成MYNUM

```
#define MYNUM (*(int volatile*)0x12340000)
```

MYNUM蕴含的对象类型是int volatile

MYNUM这个值就可能因为硬件的行为而改变
这种改变对于程序来说是不可知的（Unknown）

正常场景 vs. MMIO场景

```
int MYNUM = 10;

while (MYNUM > 20) {
    do something
}
```

MYNUM的类型是int，除初始化外没有代码对其赋值

编译器可能会将这个while条件语句完全优化掉，因为MYNUM现在是10，不可能超过20

```
#define MYNUM (*(int volatile*)0x12340000)
MYNUM = 10;

while (MYNUM > 20) {
    do something
}
```

MYNUM现在类型是int volatile，除了初始化赋值10之外也没有其他代码对其赋值

编译器不会对MYNUM>20这个表达式做任何优化，因为MYNUM有被修改的潜在可能

volatile的对象的evaluate

volatile对象如果要evaluate，都必须去访问对应的内存

Volatile accesses to objects are evaluated strictly according to the rules of the abstract machine.

… abstract machine in which issues of optimization are irrelevant

while (MYNUM > 20)

每次while条件判断都会读MYNUM对应的内存（而不会是通过缓存或者其他优化方法）获得值和20比较

volatile的对象一定会evaluate吗?

```
#define YOURNUM (*(unsigned int volatile*)0x12340004)

while (YOURNUM < 0) {
    do something
}
```

编译器也可以忽略这个while循环，因为YOURNUM是无符号整数，取值一定不会小于0

C语言标准规定如果编译器能推断出一个表达式无效，也可以选择不evaluate这个表达式，即使这个表达式包括volatile的对象

volatile使用时注意事项

if((MYNUM = 2+3) == 5)

if的条件判断表达式结果一定是真吗？

赋值表达式的值是表达式执行完成后等号左边对象的值

不强制要求通过访问左边对象来获得表达式的值，即使这个对象是volatile的也不强制

1、如果通过访问左边对象获得值，则可能是5，也可能是硬件正好刚刚又进行一次修改的值

2、如果不是通过访问左边对象来获得值，比如通过缓存刚才计算的值，则结果就是5

int volatile对应的指针对象类型

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

p: Obj_T是int volatile*, V_T是int volatile*

提示：把int volatile当作一个整体来看

int* q = (int*) &a; *q = 20;有什么问题？

这是一个undefined behavior
试图通过non-volatile-qualified的类型
来访问一个volatile修饰的内存

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

const + volatile

```
int const volatile a = 10;
```

对象a不能被程序显示赋值和修改

但可以被未知方式（例如：硬件）隐式修改

A: 0x0061FE10
Obj_T: int const volatile
N: a
S: 4
V: 10
V_T: int
Align: 4

const修饰的对象为什么不是常量

```
typedef int volatile VINT

VINT const x = 10;
```

x的值一定不会变吗？

不可修改的lvalue，是面向程序员而言

volatile使用时注意事项

int a = MYNUM + MYNUM

这个表达式有问题吗？

An access to an object through the use of an lvalue of volatile-qualified type is a volatile access

A volatile access to an object, …., are all side effects

这个表达式中读取两次MYNUM，两个side effect，有什么问题呢？

进一步深入了解表达式的side effect

给定int i = 1;以下表达式都等于多少？

```
i + i++;
i++ + i++;
++i + i++;
++i + ++i
++i + ++i + ++i
...
```

这些表达式的在C语言标准中都被定义为**Undefined Behavior**

在实际工程中，不能使用这样的表达式

Evaluation of Expression

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）

Value Computation: 返回值Value和返回值类型Value_Type，记为<V, V_T>

Side Effect: 状态的改变（changes in the state of the execution environment）

Evaluation of Expression

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

帮助我们去理解语法

2、 An actual implementation need not evaluate part of an expression if it can deduce that its value is not used and that no needed side effects are produced

帮助我们去理解优化

求值和副作用示例

```
int a = 0; int b = 1; int c = 2;

a;
```

求值	<0, int>
副作用	N/A

Value computation for an lvalue expression includes determining the identity of the designated object.

求值和副作用示例

int a = 0; int b = 1; int c = 2;

a = b;

b：

求值	<1, int>
副作用	N/A

a=b：

求值	<1, int>
副作用	a的值变成1

问题：a=b和b的求值是否有先后顺序要求？ **有**

An assignment expression has the value of the left operand after the assignment

求值和副作用示例

```
int a = 0; int b = 1; int c = 2;

a = b;
```

b：

求值	<1, int>
副作用	N/A

a=b：

求值	<1, int>
副作用	a的值变成1

问题：a的值什么时候改的？

The side effect of updating the stored value of the left operand is sequenced after the value computations of the left and right operands.

求值和副作用示例

int a = 0; int b = 1; int c = 2;

a = b + c;

b:	求值	<1, int>
	副作用	N/A
c:	求值	<2, int>
	副作用	N/A
b+c:	求值	<3, int>
	副作用	N/A
a=b+c:	求值	<3, int>
	副作用	a的值变成3

问题：b、c、b+c的求值是否有先后顺序要求？

b和c的求值没有先后要求
但必须先于b+c的求值

求值和副作用示例

int a = 0; int b = 1; int c = 2;

a = b++;

b++：
求值 <1, int>
副作用 b的值加1

a=b++：
求值 <1, int>
副作用 a的值变成1

问题：b++的副作用和a=b++的求值是否有先后顺序要求？ 无

Sequenced Before

Sequenced Before是一种非对称、可传递的成对Evaluation之间的关系

A sequenced before *B*的含义是：

*A*的evaluation在*B*的evaluation之前

与*A*相关的Valuation Computation和side effects全部在
与*B*相关的Valuation Computation和side effects之前

C语言定义了一系列sequence point来规范sequenced before这种行为

A sequence point *B*保证*A* sequenced before *B*

C语言中有哪些Sequence Point呢？

1、Function Designator和实参的evaluation，和实际函数调用执行之间

2、在&&、||、逗号运算符分隔的前后两个表达式之间

3、?:三目运算表达式中，?之前表达式以及之后执行的表达式之间

4、两个full expression之间，full expression包括例如：

表达式语句（分号）、if、while、do、for，return等控制条件表达式等

5、库函数调用之前

6、printf/scanf、fpritnf/fscanf、sprint/sscanf等一系列输入输出中按转换说明符执行完转换动作之后

7、bsearch，qsort等比较函数调用之前和之后以及调用比较函数和对象移动之间

我们先来看最简单的

两个full expression之间一定会有一个sequence point

Sequence Point前后的表达式执行顺序是确定的

int i=1; i++; i++; 此时i一定等于3

表达式语句（分号）

Sequence Points之间的表达式内执行顺序

Sequence Points之间表达式内执行顺序是如何规定的呢？

除了显式的语法规定，表达式内子表达式的evaluation之间顺序关系没有约定

The grouping of operators and operands is indicated by the syntax. Except as specified later, side effects and value computations of subexpressions are unsequenced

```
a = b + c;
```

a=b+c的求值必须在b+c求值之后，b+c的求值必须在b和c的求值之后，但b和c的求值没有顺序要求

++i + ++i有什么问题？

```
int main()
{
    int a = 1;
    int i = 2;

    a = ++i + ++i;

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

    return 0;
}
```

a=?

Sequence Points a = ++i + ++i Sequence Points

++i + ++i有什么问题？

x86-64 clang 17.0.1 Compiler options...

```
1  main:                                   # @main
2          push    rbp
3          mov     rbp, rsp
4          sub     rsp, 16
5          mov     dword ptr [rbp - 4], 0
6          mov     dword ptr [rbp - 8], 1
7          mov     dword ptr [rbp - 12], 2
8          mov     eax, dword ptr [rbp - 12]
9          add     eax, 1
10         mov     dword ptr [rbp - 12], eax
11         mov     ecx, dword ptr [rbp - 12]
12         add     ecx, 1
13         mov     dword ptr [rbp - 12], ecx
14         add     eax, ecx
15         mov     dword ptr [rbp - 8], eax
16         mov     esi, dword ptr [rbp - 8]
17         lea     rdi, [rip + .L.str]
18         mov     al, 0
19         call    printf@PLT
20         xor     eax, eax
21         add     rsp, 16
22         pop     rbp
23         ret
```

初始：a=1, i=2　　　a=7

1、初始化对象i的值为2

2、对象i的值放到寄存器eax中

3、eax寄存器的值加 1

4、eax寄存器的值放回对象i中

5、对象i的值放到寄存器ecx中

6、ecx寄存器的值加1

7、ecx寄存器的值放回对象i中

8、将ecx寄存器的值加到eax寄存器中

9、将eax寄存器的值放到对象a中

++i + ++i有什么问题？

x86-64 gcc 13.2 | Compiler options...

```
1   .LC0:
2           .string "a=%d\n"
3   main:
4           push    rbp
5           mov     rbp, rsp
6           sub     rsp, 16
7           mov     DWORD PTR [rbp-4], 1
8           mov     DWORD PTR [rbp-8], 2
9           add     DWORD PTR [rbp-8], 1
10          add     DWORD PTR [rbp-8], 1
11          mov     eax, DWORD PTR [rbp-8]
12          add     eax, eax
13          mov     DWORD PTR [rbp-4], eax
14          mov     eax, DWORD PTR [rbp-4]
15          mov     esi, eax
16          mov     edi, OFFSET FLAT:.LC0
17          mov     eax, 0
18          call    printf
19          mov     eax, 0
20          leave
21          ret
```

初始：a=1, i=2

a=8

1、初始化对象i的值为2

2、对象i的值加1

3、对象i的值加1

4、对象i的值放到寄存器eax中

4、eax+eax再放到寄存器eax中

5、eax寄存器的值放到对象a中

C语言标准是怎么规定这种情况的呢？

对1个标量对象(Scalar Object)，如果

- 产生两次副作用且两次副作用没有先后顺序要求，或
- 产生的副作用和同样标量对象取值之间没有先后顺序要求

其结果是undefined behavior

If a side effect on a scalar object is unsequenced relative to either a different side effect on the same scalar object or a value computation using the value of the same scalar object

int i=1; i = ++i + 1;

++i的副作用将i的值+1，i=++i+1的副作用是将i的值设置为++i+1的值

++i的求值结果是2， i=++i+1 的求值结果是3

但是，如果++i的副作用发生在i=++i+1的副作用之后呢？

C语言标准是怎么规定这种情况的呢？

对1个标量对象(Scalar Object)，如果

- 产生两次副作用且两次副作用没有先后顺序要求，或
- 产生的副作用和同样标量对象取值之间没有先后顺序要求

其结果是undefined behavior

If a side effect on a scalar object is unsequenced relative to either a different side effect on the same scalar object or a value computation using the value of the same scalar object

```
int i=1; a[i++] = i;
```

如果i++的副作用发生在i的求值之前

思考题

```
int a = 3;
a += a -= a * a;

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

现在a等于几了？

函数调用中的Sequence Point

1、Function Designator和实参的evaluation，和实际函数调用执行之间

意味着Function designator和所有实参的evaluate先后顺序不定

```c
int main(void) {
    static int a = 0;
    printf("output order is %d %d\n",
           a++, a++);

    return 0;
}
```

MinGW clang 16.0.2

```
Program returned: 0
Program stdout
output order is 0 1
```

x86-64 gcc 13.2

```
Program returned: 0
Program stdout
output order is 1 0
```

这产生了未定义行为

对a产生两次副作用且两次副作用没有先后顺序要求

函数调用中的Sequence Point

1、Function Designator和实参的evaluation，和实际函数调用执行之间

意味着Function designator和所有实参的evaluate先后顺序不定

```c
int foo(void) {
    static int a = 0;

    return a++;
}

int main(void) {
    printf("output order is %d %d\n",
            foo(), foo());

    return 0;
}
```

MinGW clang 16.0.2

```
Program returned: 0
Program stdout
output order is 0 1
```

x86-64 gcc 13.2

```
Program returned: 0
Program stdout
output order is 1 0
```

什么是Function designator？

逗号运算符相关的Sequence Point

```c
int main(void) {
    int a=0;

    int b = (a++, a++);

    printf("%d", b);

    return 0;
}
```

逗号运算符分隔的前后两个

表达式之间有sequence point

```c
int main(void) {
    static int a = 0;
    printf("output order is %d %d\n",
            a++, a++);

    return 0;
}
```

两个a++之间的逗号不是逗号运算符

&&、||运算符相关的Sequence Point

```
int main(void) {
    int a=0;

    a++ && a++;

    return 0;
}
```

&&、||运算符会带来

sequence point

三目运算符相关的Sequence Point

```
int main(void) {
    int a=0;

    a++ ? a++ : a++;

    return 0;
}
```

三目运算符会在?前面的表达式和

后面2选1的表达式之间插入一个sequence point

自增/自减运算符的谨慎使用

因为自增/自减表达式同时会带来Valuation Computation和Side Effect

使用上要格外谨慎

GJB 5369−2005

4.8.2 推荐类……31
4.8.2.1 避免使用“+=”或“-=”操作符……31
4.8.2.2 谨慎使用“++”或“--”操作符……31

5.6.1.5 准则 R−1−6−5

禁止在运算表达式中或函数调用参数中使用＋＋或−−操作符。

回顾两次volatile access

int a = MYNUM + MYNUM

这个语句中，对MYNUM进行了两次访问

对同一个MYNUM对象有两次副作用，且顺序没有要求