深入了解表达式的evaluation

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 （理解**pass by value**中的**value**）

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*

Sequence Point

1、函数名和实参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

Sequence Points之间的表达式内执行顺序

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

Sequence Points之间表达式内执行顺序呢？

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

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

Side effect 1 (SE1)：i的值+1，Value Computation 1 (VC1): 获得i的值

Side effect 2 (SE2)：i的值+1，Value Computation 2 (VC2): 获得i的值

SE1 -> VC1 -> SE2 -> VC2：a的值就是5

SE1 -> SE2 -> VC1 -> VC2：a的值就是6

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

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

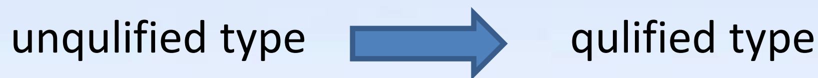
GJB 5369−2005

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。

关于volatile



给定一个对象类型Obj_T和变量名称N，声明volatile变量的语法为

Obj_T volatile N 或 volatile Obj_T N

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，但表示值类型是int

… qualified type, the value has unqualifed version …

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

a对应的内存六元组

int volatile的含义

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

A: 0x0028FF11
Obj_T: int volatile
N: a
S: 4
V: 10 20
V_T: int

a对应的内存六元组

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的类型是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）获得表示值和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

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

对同一个标量对象的副作用，先后没有次序约定
这是一个undefined behavior

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 = &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: 0x0047B521
Obj_T: int volatile*
N: p
S: 4
V: 0x0028FF11
V_T: int volatile*

p对应的内存六元组

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

a对应的内存六元组

volatile和多线程

volatile和多线程没有必然联系
多线程下共享对象的奇怪表现不是unkown，而是没有合理的访问控制导致的

在某些特定的应用场景中，通过volatile可能刚好可以实现需求，但这同时也可能会为软件未来的迭代更新埋下隐患

int const volatile a;

一个线程更新a的值，一个或多个线程读a的值的情况
解决了不使用缓存优化，所有线程都可以读到最新值

但有两个以上线程更新a的值，就不一定可行了

2个线程同时更新volatile对象场景

read-modify-write

假设两个线程A和B都执行a=a+1;

1、读入a的值
2、将a的值+1
3、将结果写入a

假设a初始为10

假设并发顺序是A1、A2、A3、B1、B2、B3　　a的值是12

假设并发顺序是A1、B1、A2、A3、B2、B3　　a的值是11

C语言中多线程的工具

如果一个表达式试图修改一段内存中的值，其他表达式试图读或写同一块内存，就构成了一个冲突（Conflict）

The library defines a number of atomic operations and operations on mutexes that are specially identified as synchronization operations. These operations play a special role in making assignments in one thread visible to another.

理解原子操作和锁相关的概念对于多线程的学习至关重要

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*

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（指针总是蕴含着数组访问）

例如*O, *(O+1), O[n]

2、给定一个指针表达式E，如果Obj的值修改，指向一个新的对象（这个新的对象是Obj原来指向对象的拷贝），表达式E的值也会被修改，则

E **Based on** Obj

了解Based On：示例

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.

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

```c
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！！！

了解size、padding和alignment

1、了解对象的size、padding和alignment的基本概念

2、通过结构体的实际应用来加深size、padding和alignment的理解

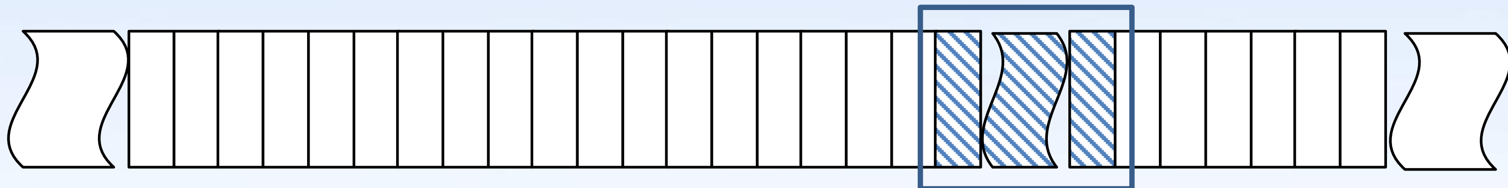
回顾一下：1 byte一定等于8 bit？

int的一定占4个字节吗？

声明一个char* p，(int*)p会有问题吗？

size的内涵

给定一个完全对象类型*T*，声明一个变量*O*（即*T O*），其含义是什么呢？



分配了一系列字节，这段内存的对象类型为*T*，且用*O*这个变量名可以定位

注意：*O*不是对象（*O*是标识符，是表达式），这段内存才是对象

sizeof(*T*) vs. sizeof(*O*)

sizeof(*T*)返回值的含义是为一个类型*T*的对象分配空间需要多少个字节

sizeof(*O*)返回值的含义是用变量名*O*来定位的那个对象共占用了多少个字节

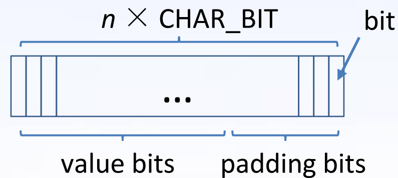
理解padding：无符号整数

假设一个无符号整数类型*T*，一共占用*n* × CHAR_BIT个bit位

sizeof(*T*)返回值为<*n*, size_t>

这个*n*×CHAR_BIT个bit分为以下两个部分：

1、value bits
2、padding bits



示意图，不意味着必须这么摆放

假设value bits的个数是*N*，该无符号整数对象表值范围为0 ~ 2^N-1
这个*N*值就被称为这个无符号整数类型*T*的宽度（width）

unsigned char类型不允许有padding bits，其他无符号整数类型可以没有padding bits

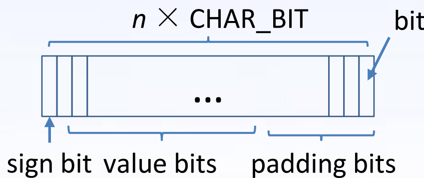
理解padding：有符号整数

假设一个有符号整数类型*T*，一共占用$n \times$ CHAR_BIT个bit位

sizeof(*T*)返回值为<*n*, size_t>

这个$n \times$CHAR_BIT个bit分为以下三个部分：

1、sign bit
2、value bits
3、padding bits



示意图，不意味着必须这么摆放

假设sign bit+value bits的个数是*N*，该有符号整数对象表值范围为$-(2^{N-1}) \sim 2^{N-1}-1$
这个*N*值就被称为这个有符号整数类型*T*的宽度（width）

signed char类型不允许有padding bits，其他有符号整数类型可以有padding bits

char和int在size和padding上的区别

C语言规定：

1、sizeof(unsigned char)/sizeof(signed char)恒等于1

2、unsigned char/signed char类型不允许有padding bits

结论：signed/unsigned char的大小一定是1个byte，且没有padding

例如对于一个unsigned char类型的对象，其取值范围为 0 to $2^{CHAR_BIT} - 1$

对于司空见惯的int类型，思考一下下面两个问题

1、sizeof(int)=?

2、int类型允许包含padding bits吗？

深入了解unsigned int/int

对于int类型，思考一下下面两个问题

1、sizeof(int)=?
2、int类型允许包含padding bits吗？

C语言规定，int类型的sign bit+value bits长度必须大于等于16（新的INT_WIDTH宏）

相应的，INT_MAX必须大于等于32767，INT_MIN必须小于等于-32768

试试你自己电脑上编译器的INT_MAX等于多少？

不能假设sizeof(int)一定等于4，有些系统（例如Turbo C）中sizeof(int)就等于2

目前主流编译器的int类型都没有padding bits，但是依然不能假设所有int没有padding bits

了解intN_t/uintN_t

C语言标准规定编译器可以定义一类1）没有padding和2）确定宽度的整数类型
Exact-width integer types，形如intN_t，uintN_t

例如int16_t：意味着这个有符号整数类型没有padding bits且width恰好等于16

C语言标准不要求编译器必须提供intN_t类型

但如果编译器提供了宽度为8，16，32和64，且没有padding的整数类型
则应该通过typedef提供相应的intN_t类型

例如某平台int类型width是32且没有padding bit
则应该通过typedef int int32_t定义出int32_t类型

了解int_least*N*_t/uint_least*N*_t

C语言标准规定编译器需要定义一类宽度至少是某个N值的整数类型

Minimum-width integer types，形如int_least*N*_t，uint_least*N*_t

例如int_least16_t：意味着这个有符号整数类型的width大于等于16

以下类型是所有编译器都必须定义的

int_least8_t，int_least16_t，int_least32_t，int_least64_t

uint_least8_t，uint_least16_t，uint_least32_t，uint_least64_t

注意：如果编译器定义了int*N*_t，那么int_least*N*_t和int*N*_t一样

了解int_fast*N*_t/uint_fast*N*_t

C语言标准规定编译器需要定义一类宽度至少是某个N值且处理速度最快的整数类型 Fastest minimum-width integer types，形如int_fast*N*_t，uint_fast*N*_t

例如int_fast16_t：意味着这个有符号整数类型的width大于等于16，且处理速度最快

以下类型是所有编译器都必须定义的

int_fast8_t，int_fast16_t，int_fast32_t，int_fast64_t
uint_fast8_t，uint_fast16_t，uint_fast32_t，uint_fast64_t

注意：这个fast并不保证所有情况下处理速度都是最快，编译器可以简单选择满足符号要求和宽度要求的整数类型来进行typedef