

# Sequence Points

“to get a deeper understanding of the language”



Deep C - a 3 day course  
Jon Jagger & Olve Maudal

```
#include <stdio.h>

int a[] = { 0,2,4,6,8 };
int b[] = { 0,2,4,6,8 };

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```

Here is what I get on my machine:

```
$ clang ... main.c
error: multiple unsequenced modifications to 'i' [-Werror,-Wunsequenced]
    int n = v[i++] - v[i++];
               ^      ~~~
```

```
#include <stdio.h>

int a[] = { 0,2,4,6,8 };
int b[] = { 0,2,4,6,8 };

int main(void)
{
    int i = a[0] + 1;
    int n = i + a[++i] + b[++i];
    printf("%d\n", n);
}
```

```
$ gcc ... main.c
$ ./a.out
12
```

```
$ icc ... main.c
$ ./a.out
13
```

```
$ clang ... main.c
$ ./a.out
11
```



```
$ gcc ... main.c
$ ./a.out
12
$ gdb ./a.out
(gdb) set disassembly-flavor intel
(gdb) disassemble main
```

12

# gcc

...

```
mov ecx,DWORD PTR [ebp-0xc]
add ecx,0x1
mov DWORD PTR [ebp-0xc],ecx
mov ecx,DWORD PTR [ebp-0xc]
mov ecx,DWORD PTR [eax+ecx*4+0x115]
```

i + a[++i] + b[++i];  
4

```
mov edx,DWORD PTR [ebp-0xc]
add ecx,edx
```

i + a[++i] + b[++i];  
6

```
mov edx,DWORD PTR [ebp-0xc]
add edx,0x1
mov DWORD PTR [ebp-0xc],edx
mov edx,DWORD PTR [ebp-0xc]
mov edx,DWORD PTR [eax+edx*4+0x135]
```

i + a[++i] + b[++i];  
6

```
add ecx,edx
```

i + a[++i] + b[++i];

...

12

```

$ gcc ... main.c
$ ./a.out
13
$ gdb ./a.out
(gdb) set disassembly-flavor intel
(gdb) disassemble main

```

13

# icc

...

```

mov  edx,0x1
add  edx,DWORD PTR [ebp-0x18]
mov  DWORD PTR [ebp-0x18],edx

```

$i + a[++i] + b[++i];$   
2

```

mov  ecx,0x1
add  ecx,DWORD PTR [ebp-0x18]
mov  DWORD PTR [ebp-0x18],ecx

```

$i + a[++i] + b[++i];$   
3

```

shl  edx,0x02
lea  ebx,[eax+0xc2]
add  ebx,edx
mov  edx,DWORD PTR [ebx]

```

$i + a[++i] + b[++i];$   
4

```

add  edx,DWORD PTR [ebp-0x18]

```

$i + a[++i] + b[++i];$   
7

```

shl  ecx,0x2
lea  ebx,[eax+0xd6]
add  ebx,ecx

```

$i + a[++i] + b[++i];$   
6

```

add  edx,DWORD PTR [ebx]

```

$i + a[++i] + b[++i];$

...

13

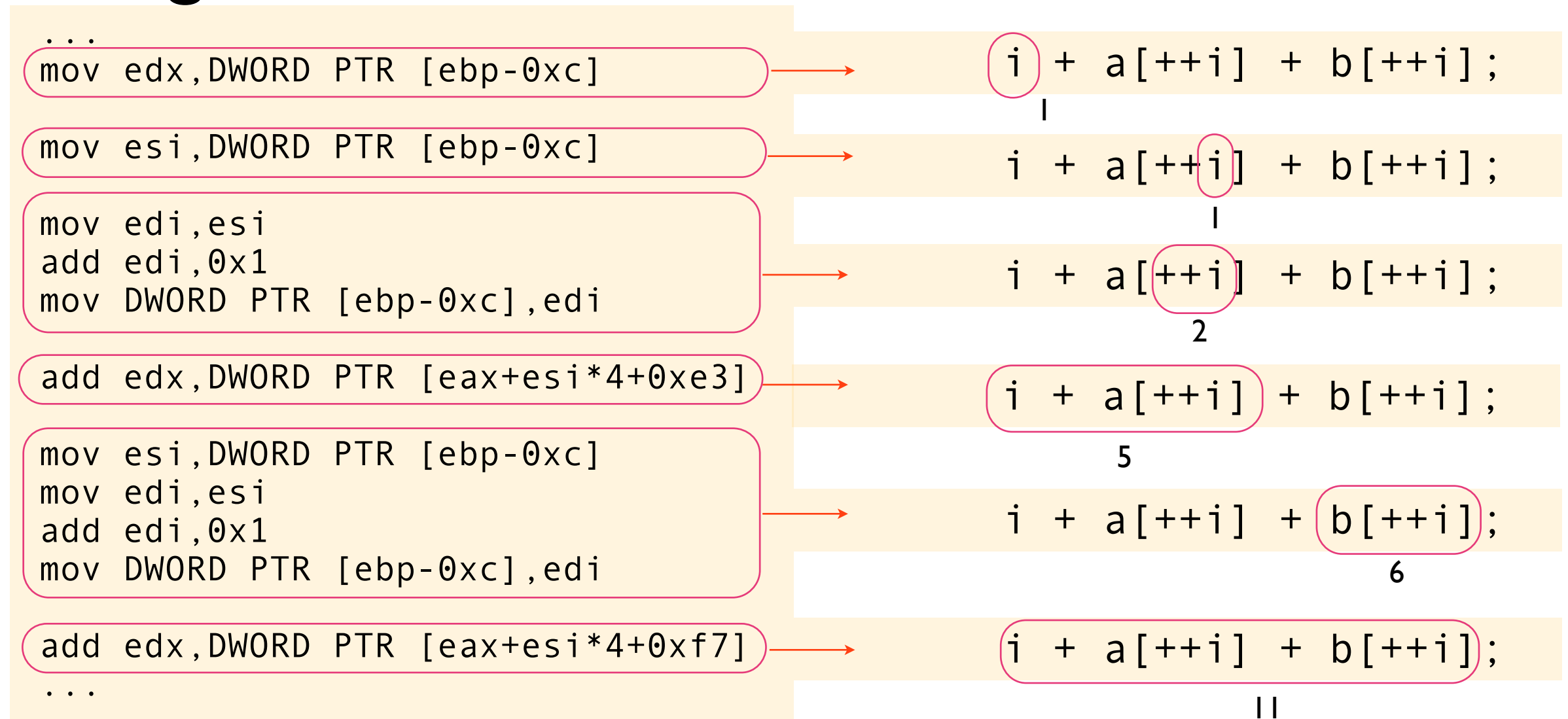
```

$ clang ... main.c
$ ./a.out
11
$ gdb ./a.out
(gdb) set disassembly-flavor intel
(gdb) disassemble main

```

11

# clang



### 5.1.2.3 Program execution. Paragraph 3

*Sequenced before* is a ... relation between evaluations executed by a single thread, which induces a partial order among those evaluations.

The presence of a *sequence point* between the evaluations of expressions A and B implies that every value computation and side effect associated with A is sequenced before every value computation and side effect associated with B.

If A is not sequenced before or after B, then A and B are *unsequenced*.

In other words:

A sequence point is a point (in time) in the program's execution sequence when all previous side effects will have already taken place and when all subsequent side-effects will not yet have taken place.

*Only* sequence points govern this sequencing!

## 6.5 Expressions. Paragraph 2

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, the behaviour is *undefined*.

## 6.5 Expressions. Paragraph 3

Except as specified later, side effects and value computations of subexpressions are *unsequenced*.

In other words, by default, subexpressions are *unsequenced*.





# Sequence Points occur...

- After evaluation of the function designator and the actual arguments but before the actual function call (6.5.2.2 paragraph 10)

- After evaluation of first operand of these operators.

&& logical-and (6.5.13 paragraph 4)

|| logical-or (6.5.14 paragraph 4)

?: ternary (6.5.14 paragraph 4)

, comma (6.5.17 paragraph 2)

- At the end of a full declarator.

6.7.5. Declarators. paragraph 3.

A full declarator is a declarator that is not part of another declarator.

- At the end of a full expression.

6.8 Expressions. paragraph 4.

A full expression is an expression that is not a sub-expression of another expression or declarator.

# Full Declarator?

## 6.7.5. Declarators. paragraph 3.

A full declarator is a declarator that is not part of another declarator.

This comma is a punctuation and *not* a comma operator

`int x = a++, y = a++;`

`int`

declarator

`x = a++`

`,`

declarator

`y = a++`

this `a++` is *sequenced before* this `a++`

# Full Expression?

## 6.8 Expressions. paragraph 4.

A full expression is an expression that is not a sub-expression of another expression or declarator.

```
return n % 2 == 0;
```

n is an expression  
n is not a full expression

```
return n % 2 == 0;
```

n % 2 is an expression  
n % 2 is not a full expression

```
return n % 2 == 0;
```

n % 2 == 0 is an expression  
n % 2 == 0 is a full expression

```
return n % 2 == 0;
```

return n % 2 == 0;  
is a statement



```
n = n++
```

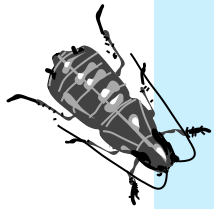
undefined?



```
n + n++
```

undefined?





`n = n++`

undefined

[n =]  
(side effect on n)  
is *unsequenced* relative to  
[n++]  
(different side effect on n)



`n + n++`

undefined

[n++]  
(side effect on n)  
is *unsequenced* relative to  
[n]  
(value computation using the value of n)

```
f(n, n++);
```

undefined?





```
f (n , n++ ) ;
```

undefined\*

[n++]

(side effect on n)

is *unsequenced* relative to

[n]

(value computation using the value of n)

\* the comma in f(n,n++) is a punctuator, not an operator

```
if (n++ & n)
{
    ...
}
```

undefined?



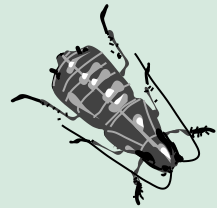
```
if (n++ && n)
{
    ...
}
```

undefined?





```
if (n & n++)  
{  
    ...  
}
```



# undefined

n++ is  
*unsequenced*  
relative to n

```
if (n++ && n)  
{  
    ...  
}
```

# not undefined\*

n++ is  
*sequenced before*  
n

\* but still very poor style

in this code fragment it is "intuitively obvious"  
that  $n$  incremented before being used  
as an argument to `func()`



```
if (n++ < 10) {  
    func(n);  
}
```

How do you *know* this?



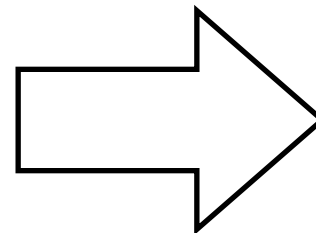
For sure?



because `n++ < 10` is the *full-expression*  
controlling the if statement

and there is a sequence-point at the  
end of each *full-expression*

**if** ( *expression* )  
*statement*



**if** ( *full-expression* )  
*statement*

```
if (n++ < 10)
{
    func(n);
}
```

`n++` is  
*sequenced before*  
`n`

```
int x = n, n++;
```

undefined?



```
int x = (n, n++);
```

undefined?



```
int x = n, n++;
```

doesn't compile!

```
int x = (n, n++);
```

not undefined

n is  
*sequenced before*  
n++

```
int f(int * p, int * q)
{
    return (*p) - (*q)++;
}
```

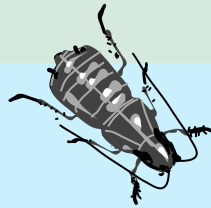
**undefined?**



```
int f(int * p, int * q)
{
    return (*p) - (*q)++;
}
```

```
int eg(int n)
{
    return f(&n, &n);
}
```

**undefined**



n is *unsequenced*  
relative to n++

```
int eg(int n, int m)
{
    return f(&n, &m);
}
```

**not undefined!**

p and q point to  
different variables

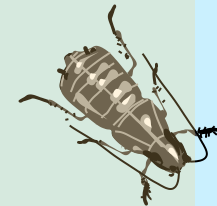
```
void swap_trick(int a, int b)
{
    a ^= b ^= a ^= b;
    ...
}
```

**undefined?**





```
void swap_trick(int a, int b)
{
    a ^= b ^= a ^= b;
    ...
}
```



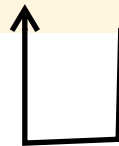
## undefined

side effect on a is *unsequenced* relative to different side effect on a  
side effect on b is *unsequenced* relative to different side effect on b

# Precedence?

- precedence occurs at compile time
- it governs what operator an operand binds to
- you determine *what* happens

```
int n = a() * b() + c();
```



- order of evaluation occurs at run time
- it is governed *only* by sequence points
- compiler determines *how* it happens



```
int reg1 = c();  
int reg2 = a();  
int reg3 = b();  
reg2 * reg3 + reg1
```

```
int reg1 = a();  
int reg2 = b();  
int reg3 = c();  
reg1 * reg2 + reg3
```

# Summary

- precedence is not the same as order of evaluation
- only sequence points govern order of evaluation
  - before function call
  - end of full declarator
  - operators `&&` `||` `?:` `,`
  - end of full expression