

Objects

Users care about how easy objects are to use and how hard objects are to misuse. This is called Affordance.

Users don't want to know or care about how hard objects are to make*. This is called Ignorance/Apathy.

Users care about only what they care about! This is called Selfishness.

(in)consistency

3

<stdio.h>

```
int fsetpos(FILE *, ...);  
int fprintf(FILE *, ...);  
int fscanf(FILE *, ...);
```

```
int fputc(..., FILE *);  
int fputs(..., FILE *);  
size_t fwrite(..., FILE *);
```

```
int main(int argc, char * argv[])...
```

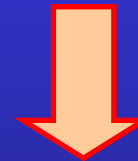
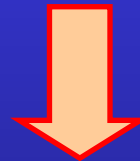
```
char * fgets(char * s, int n, FILE *);
```



- 7.14.1.1 The signal function
 - ♦ if the request can be honoured, the signal function returns the value of func for the most recent successful call to signal for the specified signal sig.

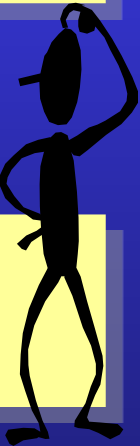
<signal.h>

```
void (* signal(int sig, void (*func)(int)))(int);
```



```
typedef void (*signal_handler)(int sig);
```

```
signal_handler signal(int sig, signal_handler func);
```



- 6.5.2.5 Compound literals
 - ♦ para 6 – if the compound literal occurs outside the body of a function, the object has static storage duration; ...

```
int * p = (int[]){1,2,3,4};
```

as-if

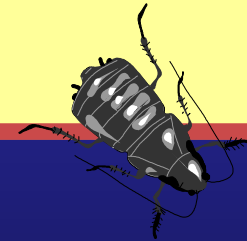
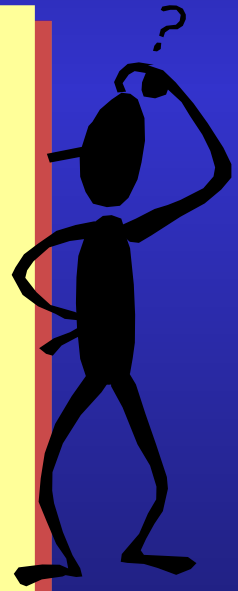
```
int $[] = { 1,2,3,4 };  
int * p = $;
```

- 6.5.2.5 Compound literals

- ◆ para 6 – if the compound literal occurs outside the body of a function, the object has static storage duration;

otherwise it has automatic storage duration associated with the enclosing *block*.

```
int eg(int i, int j)
{
    int * p;
    if (i == j)
        p = (int[]){ i, i };
    else
        p = (int[]){ j, j };
    return *p;
}
```



6.8.4 Selection statements

- ♦ para 3 – A selection statement is a block whose scope is a strict subset of the scope of its enclosing block. Each associated substatement is also a block whose scope is a strict subset of the scope of the selection statement.

```
void f(void)
{
    if (...)
        ...
    else
        ...
}
```

blocks

```
void f(void)
{
    {
        if (...)
        { ... }
        else
        { ... }
    }
}
```



c89 blocks != c99 blocks

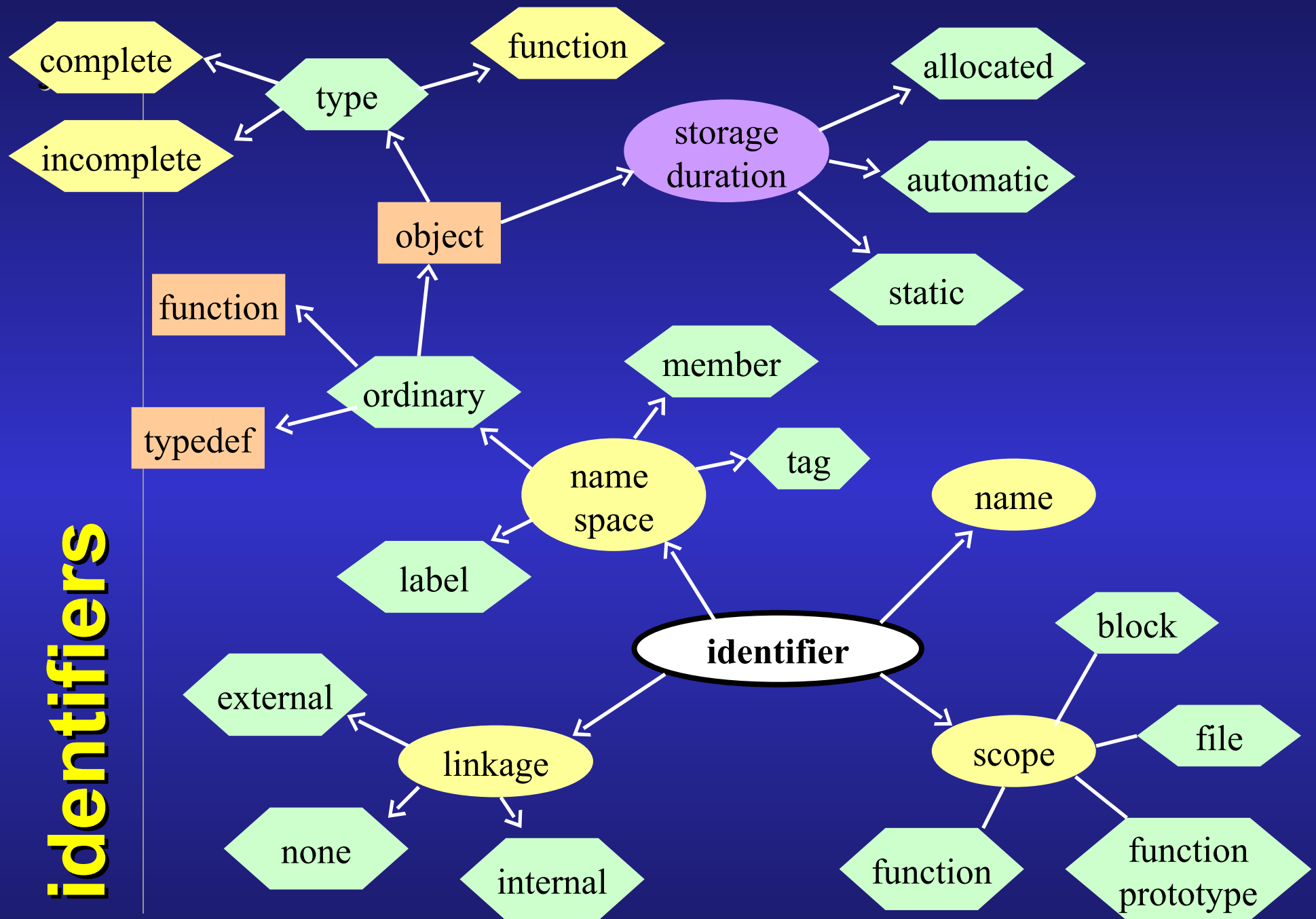
block scope

compound literals

```
int eg(int i, int j)
{
    int *p;
    {
        if (i == j)
        {
            int $[] = { i, i };
            p = $;
        }
        else
        {
            int $[] = { j, j };
            p = $;
        }
    }
    return *p;
}
```



identifiers



scope-linkage

```
/* file scope */

typedef int typedef__no_linkage;

struct stag__no_linkage { int member__no_linkage; };
union  utag__no_linkage { int member__no_linkage; };
enum   etag__no_linkage {      member__no_linkage, };

static int object__internal_linkage;
extern int object__external_linkage;
        int object__external_linkage;

static void function__internal_linkage(void) {}
extern void function__external_linkage1(void) {}
        void function__external_linkage2(void) {}
```

```
void function_prototype_scope(  
    /* typedef illegal here */  
  
    struct stag__no_linkage { int member__no_linkage; },  
    union  utag__no_linkage { int member__no_linkage; },  
    enum   etag__no_linkage {      member__no_linkage, },  
  
    int object__no_linkage  
);
```

```
void function_scope(void)  
{  
    label__no_linkage: ;  
}
```

```
void block_scope(void)
{
    typedef int typedef__no_linkage;

    struct stag__no_linkage { int member__no_linkage; };
    union  utag__no_linkage { int member__no_linkage; };
    enum   etag__no_linkage {      member__no_linkage, };

        int object__no_linkage1;
    static int object__no_linkage2;
    extern int object__external_linkage;

    /* illegal - 6.7.1 paragraph 5 (pedantic) */
    static void function__internal_linkage(void);

    extern void function__external_linkage(void);
        void function__external_linkage(void);
}
```

- 6.5 Expressions
 - ♦ para 2 - Between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression.



A sequence point is a point of stability; a point in the program's execution sequence where all previous side-effects will have taken place and where all subsequent side-effects will not have taken place.

If a single object is modified more than once between sequence points the behaviour is undefined.

- **sequence points occur...**
 - ◆ **at the end of a full expression**
 - a full expression is an expression that is not a sub-expression of another expression or declarator (6.8 p4)
 - ◆ **after the first operand of these operators**
 - && logical and (6.5.13 p4)
 - || logical or (6.5.14 p4)
 - ?: ternary (6.5.15 p4)
 - , comma (6.5.17 p2)
 - ◆ **after evaluation of all arguments and function expression before a function call (6.5.2.2 p10)**
 - ◆ **at the end of a full declarator (6.7.5 p3)**

6.8 Statements and blocks

- ♦ para 4 – A full expression is an expression that is not part of another expression or declarator.

```
int function(int value)
{
    int inside = ;
    ;
    if (  ) ...
    switch (  ) ...
    while (  ) ...
    do (  ) ...
    for ( ; ;  ) ...
    return ;
}
```

• an initializer;

• the expression in an expression statement;

• the controlling expression in a selection statement...;

• the controlling expression of a while or do statement;

• each of the (optional) expressions in a for statement;

• the (optional) expression in a return statement;

- when is a comma a sequence point?

```
int x    = f( o++ , o++ );
```

```
int x    =      o++ , o++;
```

```
int x[] = { o++ , o++ };
```



- when it's an operator but not when it's a punctuation

```
int x    = f( o++ , o++ );
```



```
int x    =      o++ , o++;
```

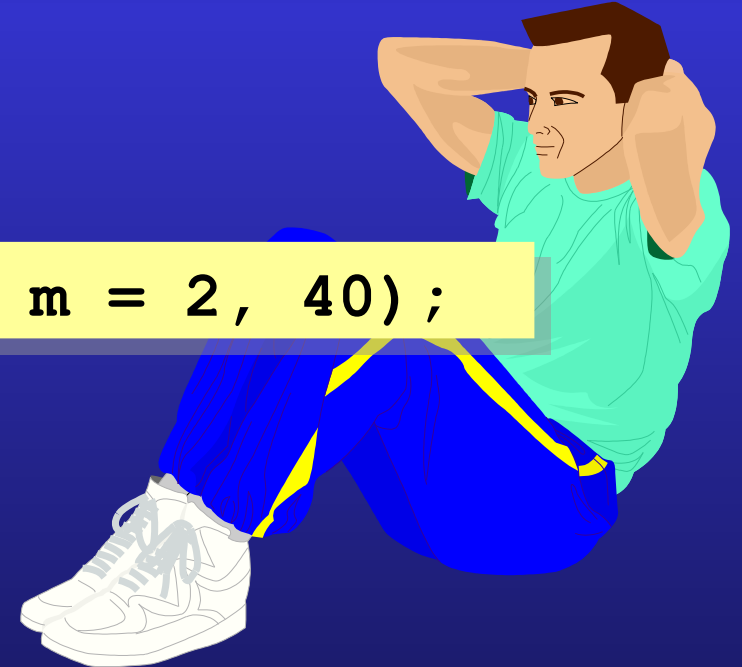


```
int x[] = { o++ , o++ };
```



- in this very tricky statement...
 - ◆ where are the sequence points?
 - ◆ what are the operators?
 - ◆ what is their relative precedence?
 - ◆ what is affected by what?
 - ◆ how many times is m modified between sequence points?
 - ◆ is it undefined?

```
(10, m = 1, 20) + (30, m = 2, 40) ;
```



- what does this print?

```
#include <stdio.h>

int glob = 0;

int f(int value, int * ptr)
{
    if (value == *ptr)
    {
        printf("value == *ptr\n");
    }
    if (value != *ptr)
    {
        printf("value != *ptr\n");
    }
    return 1;
}

int main(void)
{
    return (glob = 1) + f(glob, &glob);
}
```



representation?

A general principle of coding

Use as little representation information as you can.

Use as little representation information as you can.

6.2.6 Representations of types

6.2.6.1 General

para 4 – Values stored in non-bit-field objects of any other object type consist of $n \times \text{CHAR_BIT}$ bits, where n is the size of that type, in bytes. The value may be copied into an object of type `unsigned char [n]` (e.g., by `memcpy`); the resulting set of bytes is called the object representation of the value.



To gain access to the bits comprising the representation of an object you must use the address of the object as a pointer to an `unsigned char`. This type alone guarantees access to all bits in all bytes.

6.2.6 Representations of types

6.2.6.2 Integer types

para 1 – For unsigned integer types other than unsigned char, the bits of the object representation shall be divided into two groups; value bits and padding bits...If there are N value bits, each bit shall represent a different power of 2 ... using a pure binary representation; this shall be known as the value representation. The value of any padding bits are unspecified.

Suppose `CHAR_BIT == 8` and `sizeof(int) == 4`

This does not guarantee that `int` is a 32 bit integer type! Use `INT_MAX` to determine if `int` is a 32 bit integer.

`x == y; → memcmp(&x, &y, sizeof(x)) == 0`
true or false? ←

?

Objects

Users care about how easy objects are to use and how hard objects are to misuse. This is called Affordance.

Users don't want to know or care about how hard objects are to make*. This is called Ignorance/Apathy.

Users care about only what they care about! This is called Selfishness.

Is the only type that guarantees you the ability to read the object-representation of an object. Important when copying data from one location to another. Worth mentioning.

This could be mentioned on the <limits.h> slide.

- Click to add an outline

<stdio.h>

```
int fsetpos(FILE *, ...);  
int fprintf(FILE *, ...);  
int fscanf(FILE *, ...);
```

```
int fputc(..., FILE *);  
int fputs(..., FILE *);  
size_t fwrite(..., FILE *);
```

```
int main(int argc, char * argv[])...
```

```
char * fgets(char * s, int n, FILE *);
```



- 7.14.1.1 The signal function
 - ♦ if the request can be honoured, the signal function returns the value of func for the most recent successful call to signal for the specified signal sig.

<signal.h>

```
void (* signal(int sig, void (*func)(int)))(int);
```

?



```
typedef void (*signal_handler)(int sig);
```

```
signal_handler signal(int sig, signal_handler func);
```




- 6.5.2.5 Compound literals
 - ♦ para 6 – if the compound literal occurs outside the body of a function, the object has static storage duration; ...

```
int * p = (int[]){1,2,3,4};
```

as-if

```
int $[] = { 1,2,3,4 };  
int * p = $;
```



- 6.5.2.5 Compound literals

- ♦ para 6 – if the compound literal occurs outside the body of a function, the object has static storage duration:

otherwise it has automatic storage duration associated with the enclosing *block*.

```
int eg(int i, int j)
{
    int * p;
    if (i == j)
        p = (int[]){ i, i };
    else
        p = (int[]){ j, j };
    return *p;
}
```



- 6.8.4 Selection statements

- ♦ para 3 – A selection statement is a block whose scope is a strict subset of the scope of its enclosing block. Each associated substatement is also a block whose scope is a strict subset of the scope of the selection statement.

```
void f(void)
{
    if (...)
        ...
    else
        ...
}
```



blocks

```
void f(void)
{
    {
        if (...)
        { ... }
    }
    else
    { ... }
}
```

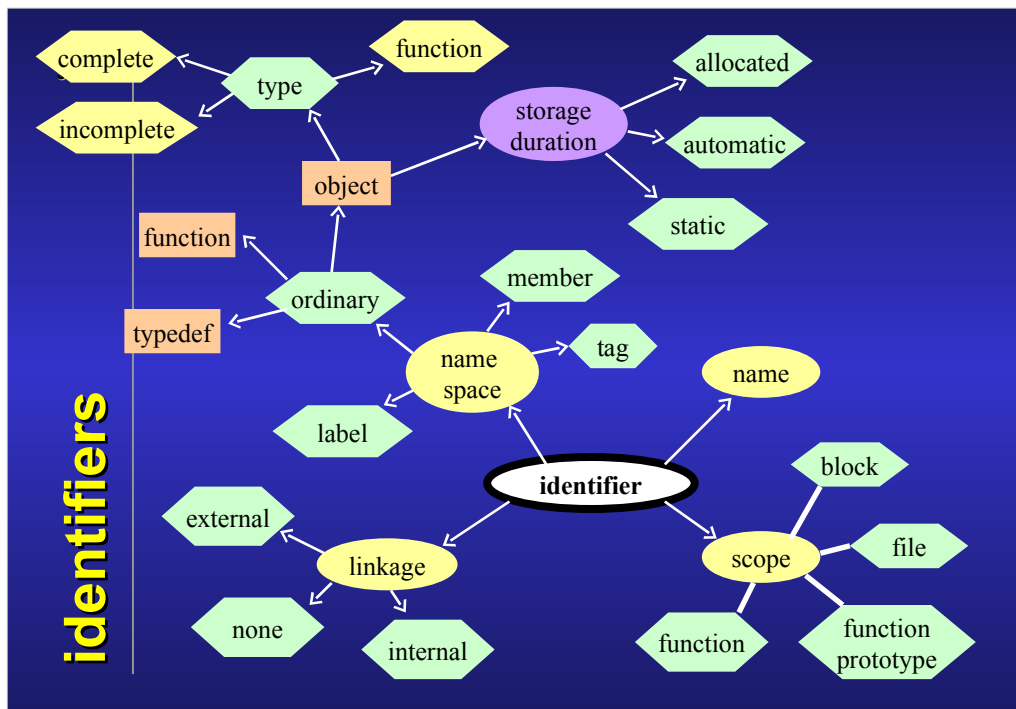


c89 blocks != c99 blocks

- Click to add an outline

```
int eg(int i, int j)
{
    int *p;
    {
        if (i == j)
        {
            int $[] = { i, i };
            p = $;
        }
        else
        {
            int $[] = { j, j };
            p = $;
        }
    }
    return *p;
}
```





Copied from Derek's book (6.2.1)

- Click to add an outline

```
/* file scope */  
  
typedef int typedef__no_linkage;  
  
struct stag__no_linkage { int member__no_linkage; };  
union utag__no_linkage { int member__no_linkage; };  
enum etag__no_linkage { member__no_linkage, };  
  
static int object__internal_linkage;  
extern int object__external_linkage;  
int object__external_linkage;  
  
static void function__internal_linkage(void) {}  
extern void function__external_linkage1(void) {}  
void function__external_linkage2(void) {}
```


- Click to add an outline

```
void function_prototype_scope(  
    /* typedef illegal here */  
  
    struct stag__no_linkage { int member__no_linkage; },  
    union  utag__no_linkage { int member__no_linkage; },  
    enum   etag__no_linkage {      member__no_linkage, },  
  
    int object__no_linkage  
);
```

```
void function_scope(void)  
{  
    label__no_linkage: ;  
}
```

- Click to add an outline

```
void block_scope(void)
{
    typedef int typedef__no_linkage;

    struct stag__no_linkage { int member__no_linkage; };
    union utag__no_linkage { int member__no_linkage; };
    enum etag__no_linkage { member__no_linkage, };

    int object__no_linkage1;
    static int object__no_linkage2;
    extern int object__external_linkage;

    /* illegal - 6.7.1 paragraph 5 (pedantic) */
    static void function__internal_linkage(void);

    extern void function__external_linkage(void);
    void function__external_linkage(void);
}
```

- 6.5 Expressions

- ♦ para 2 - Between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression.



A sequence point is a point of stability; a point in the program's execution sequence where all previous side-effects will have taken place and where all subsequent side-effects will not have taken place.

If a single object is modified more than once between sequence points the behaviour is undefined.

Annex C of C99 details the sequence point model.

The relevant sentence from clause 6.5 of the standard has a second sentence "Furthermore, the prior value shall be read only to determine the value to be stored."

The C99 definition of a side-effect is from 5.1.2.3 Program Execution – "Accessing a volatile object, modifying an object, modifying a file, or calling a function that does any of those operations are all side-effects."

- **sequence points occur...**
 - ♦ at the end of a full expression
 - a full expression is an expression that is not a sub-expression of another expression or declarator (6.8 p4)
 - ♦ after the first operand of these operators
 - && logical and (6.5.13 p4)
 - || logical or (6.5.14 p4)
 - ?: ternary (6.5.15 p4)
 - , comma (6.5.17 p2)
 - ♦ after evaluation of all arguments and function expression before a function call (6.5.2.2 p10)
 - ♦ at the end of a full declarator (6.7.5 p3)

The Standard also specifies that there is a sequence point immediately before a library function returns.

Examples of an expression where a single object is modified more than once between sequence points are:

```
++m * m++
m = ++m;
```

It may well be that on a particular compiler the behaviour of the above expression can be rationalized. But the result is undefined. You cannot assume the behaviour will be the same if, for example, you upgrade your compiler, or port your code to a new computer, or compile with different compile-time settings.

A simple function call names the called function directly. However it is possible to write expressions more complicated than simple names which evaluate to the address of a function. For example, consider the statement `af[t](1,2)`; where `af` is an array of function pointers, `t` is an integer variable, so `af[t]` is a function pointer.

6.8 Statements and blocks

- ♦ para 4 – A full expression is an expression that is not part of another expression or declarator.

```
int function(int value)
{
    int inside =  ;
     ;
    if (   ) ...
    switch (   ) ...
    while (   ) ...
    do (   ) ...
    for (  ;  ;   ) ...
    return  ;
}
```

- an initializer;
- the expression in an expression statement;
- the controlling expression in a selection statement...;
- the controlling expression of a while or do statement;
- each of the (optional) expressions in a for statement;
- the (optional) expression in a return statement;

- when is a comma a sequence point?

```
int x = f( o++ , o++ );
```

```
int x = o++ , o++;
```

```
int x[] = { o++ , o++ };
```



- when it's an operator but not when it's a punctuation

```
int x = f( o++ , o++ );
```



```
int x = o++ , o++;
```



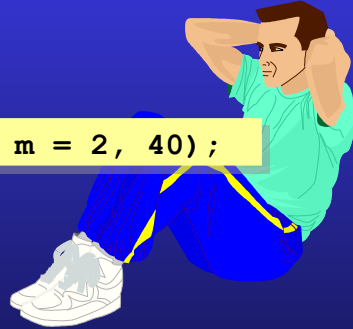
```
int x[] = { o++ , o++ };
```



another exercise

- in this very tricky statement...
 - ♦ where are the sequence points?
 - ♦ what are the operators?
 - ♦ what is their relative precedence?
 - ♦ what is affected by what?
 - ♦ how many times is m modified between sequence points?
 - ♦ is it undefined?

```
(10, m = 1, 20) + (30, m = 2, 40) ;
```



The answer is yes, it is undefined. The comma operator creates a sequence point and the statement contains four uses of the comma operator. These four comma operators guarantee

- a) the expression 10 occurs before the expression m=1
- b) the expression m=1 occurs before the expression 20
- c) the expression 30 occurs before the expression m=2
- d) the expression m=2 occurs before the expression 40

However, the comma operator does not guarantee that the expression m=1 occurs before the expression m=2.

One way to visualize this is to imagine the compiler reordering the subexpressions into the following three groups with a sequence point between the first and second group and between the second and third group, but no sequence points inside any individual group:

(10 and 30)

(m=1 and m=2)

(20 and 40)

- what does this print?

```
#include <stdio.h>

int glob = 0;

int f(int value, int * ptr)
{
    if (value == *ptr)
    {
        printf("value == *ptr\n");
    }
    if (value != *ptr)
    {
        printf("value != *ptr\n");
    }
    return 1;
}

int main(void)
{
    return (glob = 1) + f(glob, &glob);
}
```



A general principle of coding

Use as little representation information as you can.

Use as little representation information as you can.

6.2.6 Representations of types**6.2.6.1 General**

para 4 – Values stored in non-bit-field objects of any other object type consist of $n \times \text{CHAR_BIT}$ bits, where n is the size of that type, in bytes. The value may be copied into an object of type unsigned char [n] (e.g., by memcpy); the resulting set of bytes is called the object representation of the value.



To gain access to the bits comprising the representation of an object you must use the address of the object as a pointer to an unsigned char. This type alone guarantees access to all bits in all bytes.

Is the only type that guarantees you the ability to read the object-representation of an object. Important when copying data from one location to another. Worth mentioning.

This could be mentioned on the <limits.h> slide.

If you wish to gain access to the bits comprising the representation of an object you must use the address of the object as a pointer to an unsigned char. This type alone in C guarantees access to all bits in all bytes.

6.2.6 Representations of types

6.2.6.2 Integer types

para 1 – For unsigned integer types other than unsigned char, the bits of the object representation shall be divided into two groups; value bits and padding bits...If there are N value bits, each bit shall represent a different power of 2 ... using a pure binary representation; this shall be known as the value representation. The value of any padding bits are unspecified.

Suppose CHAR_BIT == 8 and sizeof(int) == 4

This does not guarantee that int is a 32 bit integer type! Use INT_MAX to determine if int is a 32 bit integer.

x == y; → memcmp(&x, &y, sizeof(x))==0
true or false?

?

It's false.