Types, Operators and Expressions

behaviour

- *implementation-defined*
  - the construct is not incorrect; the code must compile; the compiler must document the behaviour
- *unspecified*
  - the same as implementation-defined except the behaviour need not be documented
- *undefined*
  - the standard imposes no requirements ; anything at all can happen ; all bets are off! ; klaxon

examples:
signed integer right shift → implementation-defined
function argument evaluation order → unspecified
signed integer overflow → undefined

auto storage

- an identifier declared inside a function[†]
  - has automatic storage class - it's storage is reserved each time the function is called
  - has local scope
  - has an ***indeterminate*** initial value

reading an indeterminate value causes undefined behaviour

```
int outside;

int function(int value)
{
    int inside;
    ...
    static int different;
}
```

automatic storage class
local scope
no default initial value

[†] unless declared with the static keyword

static storage

- an identifier declared outside a function[†]
  - has static storage class - its storage is reserved before main starts
  - has file scope
  - has a default initial value

```
int outside;

int function(int value)
{
    int inside;
    ...
    static int different;
}
```

static storage class
default initial value is zero

[†] or declared inside the function with the static keyword

integers

- come in various flavours
  - also as signed or unsigned
- min-max values are <u>not</u> precisely defined
  - int typically corresponds to the natural word size of the host machine; the fastest integer type
  - for exact size on your computer use <limits.h>

type	*min bits*	*min limit*
char	8	2^7-1 (127)
short	16	$2^{15}-1$ (32767)
int	16	$2^{15}-1$ (32767)
long	32	$2^{31}-1$
long long	64	$2^{63}-1$

<stdint.h> and <inttypes.h>

- provide specific kinds of integers

c99

some examples

type	*meaning*
int16_t	signed int, exactly 16 bits
uint16_t	unsigned int, exactly 16 bits
int_least32_t	signed int, at least 32 bits
uint_least32_t	unsigned int, at least 32 bits
int_fast64_t	signed int, fastest at least 64 bits
uint_fast64_t	unsigned int, fastest at least 64 bits

floating point

- come in three flavours
  - float, double, long double
- again their limits are not precisely defined
  - double corresponds to the natural size of the host machine ; the fastest floating point type (but much much slower than integers)
- can be determined in code via <float.h>
  - e.g. DBL_EPSILON (max 10^{-9})
  - e.g. DBL_DIG (min 10)
  - e.g. DBL_MIN (min 10^{-37})
  - e.g. DBL_MAX (min 10^{+37})
- not represented with absolute precision

complex

- there are three complex types
  - float complex
  - double complex
  - long double complex
- <complex.h> provides
  - the macro complex for _Complex
  - lots of function declarations

```
#include <complex.h>

void eg(double complex z)
{
    double real = creal(z);
    double imag = cimag(z);
    ...
}
```

c99

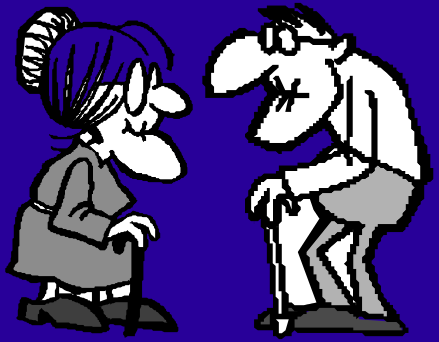
booleans

- <stdbool.h> provides three macros
  - bool for _Bool
  - false for 0
  - true for 1
  - the size of the bool type is not defined
- any integer value can be converted to a bool
  - zero is interpreted as false
  - *any* non-zero is interpreted as true

c99

```
#include <stdbool.h>

bool love = true;
bool teeth = false;
```



characters

- the char type represents a single byte
  - the smallest addressable unit of memory
  - usable as a single character or a very small int

escaped chars	*meaning*
`'\n'`	newline
`'\t'`	tab
`'\b'`	backspace
`'\r'`	carriage return
`'\f'`	form feed
`'\\'`	backslash
`'\''`	single quote

sizeof

- sizeof is a unary operator
  - use is sizeof(type) or sizeof expression
  - common for dynamic memory allocation
- result is number of bytes as a size_t
  - size_t is a typedef for an unsigned integer
  - capable of holding the size of any variable

```
type * var = malloc(sizeof(type));
```

```
type * var = malloc(sizeof *var);
```

this version is slightly better. why?

- literals for simple types are const!
  - their types can be specified

type	*suffix*	*example*
long int	L or l	42L
unsigned	U or u	42U
float	F or f	42F
long double	L or l	42.0L

- variables can be const!
  - useful for naming magic numbers

```
const double pi = 3.141592;
```

```
pi += 4.22;
```

compile time error ✗

- C is *very* liberal in its conversions
  - a widening conversion never loses information
  - a narrowing conversion may lose information

```
double mass = 0;
```

int → double

```
int bad_pi = 3.141592;
```

double → int

- an explicit conversion is called a cast
  - syntax is (type)expression
  - (void) is sometimes used to make discard explicit

```
int cast = (int)mass;
```

```
(void)printf("%i", cast);
```

arithmetic

- the usual arithmetic operators
  - + – * /
  - % is the remainder operator
  - note that integer / integer == integer

```
bool is_even(int value)
{
    return value % 2 == 0;
}
```

- overflow
  - undefined for signed integers
  - well defined for unsigned integers
  - infinities, NaN's, <fenv.h> for floating point
- divide by zero
  - undefined

initialization != assignment

- initialization occurs at declaration
- assignment occurs after declaration

```
int count;
count = 0;
```

assignment ?

```
int count = 0;
```

initialization - better ✓

```
const int answer = 42;
```

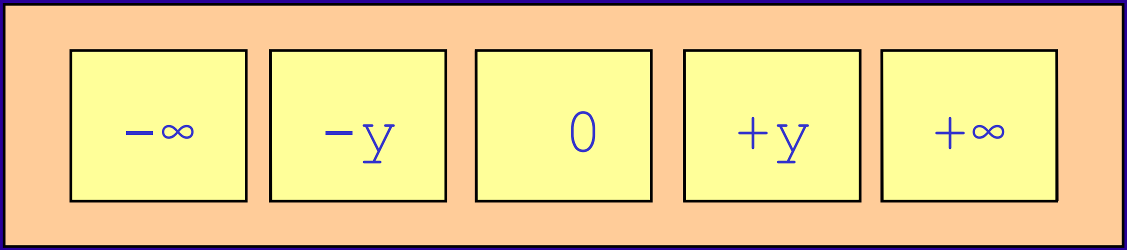
initialization ✓

```
const int answer;
answer = 42;
```

compile-time error ✗

comparison

- == != operators test for equality or identity
  - don't use == != on floating point operands
  - don't use == != on boolean literals
- < <= > >= operators test relational ordering
  - works all numeric types (but NaNs are unordered)
  - rarely useful on *char*
  - but '0' .. '9' are sequential



floating point ordering

simple assignment

assignment is an expression

- so it has an outcome – the value of the rhs
- assignment also has a significant side effect!

```
int lower;
int upper;

lower = 0;
printf("%d", lower = 0);

lower = upper = 0;
printf("%d", lower = upper = 0);
```

same as

```
upper = 0;
lower = upper;
```

common assignment patterns are supported natively with compound assignment operators

non-idiomatic

```
lhs = lhs * rhs;
lhs = lhs / rhs;
lhs = lhs % rhs;
```

idiomatic

```
lhs *= rhs;
lhs /= rhs;
lhs %= rhs;
```

```
lhs = lhs + rhs;
lhs = lhs - rhs;
```

```
lhs += rhs;
lhs -= rhs;
```

?

✓

inc/dec

- adding/subtracting one is supported directly
  - ++ is the increment operator
  - -- is the decrement operator

non-idiomatic

```
lhs = lhs + 1;
```

```
lhs = lhs - 1;
```

?

non-idiomatic

```
lhs += 1;
```

```
lhs -= 1;
```

?

idiomatic

```
lhs++;
```

```
lhs--;
```

✓

prefix-postfix

++ and -- come in two forms

- result of ++var is var *after* the increment
- result of var++ is var *before* the increment
- no other operators behave like this :-)

```
prefix = ++m;
```

```
m = m + 1;
prefix = m;
```

?

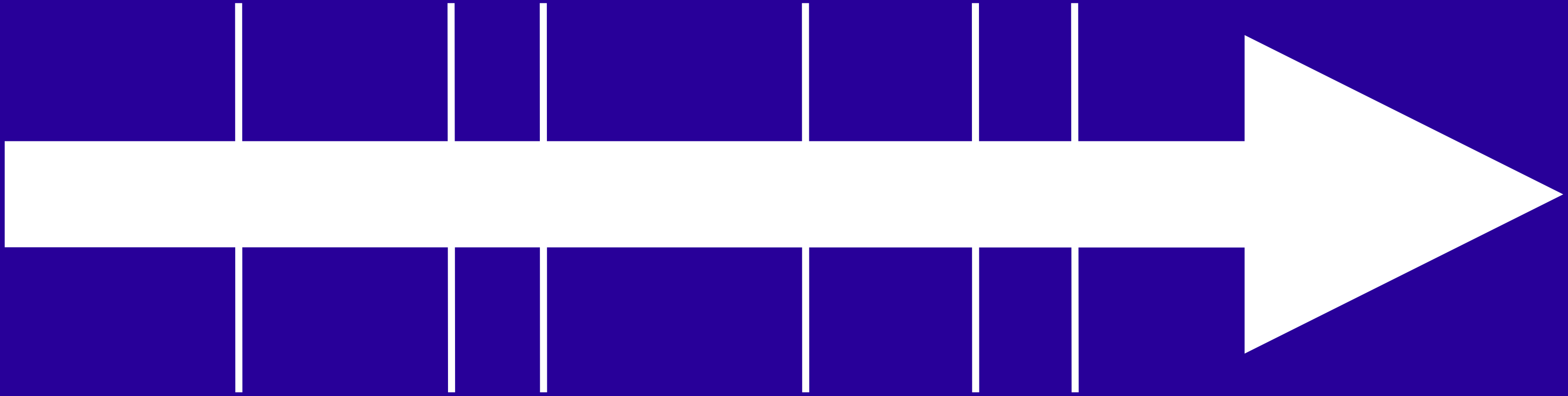
equivalent[†]

```
postfix = m++;
```

```
postfix = m;
m = m + 1;
```

[†]except that m is evaluated only once

- a sequence point is…
  - a point in the program's execution sequence where all previous side-effects ***shall*** have taken place and where all subsequent side-effects ***shall not*** have taken place



- 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 (6.8p2)
  - after the first operand of these operators
    - && logical and
    - || logical or
    - ?: ternary
    - , comma
  - after evaluation of all arguments and function expression in a function call
    - note that the comma used for separating function arguments is not a sequence point
  - at the end of a full declarator

sequence points rule 1

C Standard: 6.5 Expressions
Between the previous and next sequence point an object *shall* have its stored value modified at most *once* by the evaluation of an expression.
…

in other words, if an object is modified more than once between sequence points the result is *undefined*

sequence points rule 2

C Standard: 6.5 Expressions

...

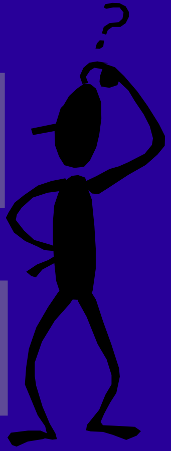
Between the previous and next sequence point…the prior value *shall* be read only to determine the value to be stored.

in other words, if an expression reads the value of a modified object more than once between sequence points the result is *undefined*

sequence points

```
n = n++
```

```
n + n++
```



Are these expressions undefined?

&& || operators

- lhs && rhs
  - if lhs is false the rhs is not evaluated
  - if lhs is true, ***sequence point***, rhs is evaluated
- lhs || rhs
  - if lhs is true the rhs is not evaluated
  - if lhs is false, ***sequence point***, rhs is evaluated

&&	false	true
false	false	false
true	false	true

||	false	true
false	false	true
true	true	true

ternary operator

- the only operator with three arguments
  - a ? b : c → if (a) b; else c;
  - *sequence point* at the ?
  - an expression rather than a statement
  - useful in macros and to avoid needless repetition

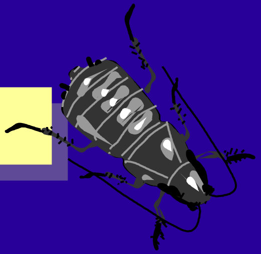
```c
void some_func(void)
{
    bool found = search(...);
    if (found)
        printf("found");
    else
        printf("not found");
}
```

```c
void some_func(void)
{
    bool found = search(...);
    printf("%sfound", found ? "" : "not ");
}
```

comma operator

- lhs , rhs
  - lhs is evaluated and the result is discarded
  - *sequence point* at the comma
  - rhs is evaluated and is the result

```
int last = (2, 3, 4, 5, 6);
```

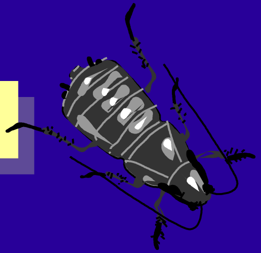


- sometimes seen in for statements

```
for (octave = 0, freq = 440;
      is_audible(freq);
    ++octave, freq *= 2) ...
```

does this access an element of a 2-d array?

```
int element = matrix[row,col];
```



- in these statements…
  - where are the sequence points?
  - how many times is m modified?
  - which ones are undefined?

1
```
f(++m * m++);
```

2
```
m = m++;
```

3
```
m = m = 0;
```

precedence

primary	`( ) [ ] -> .`
unary	`! ~ + - ++ -- (T) sizeof + - * &`
multiplicative	`* / %`
additive	`+ -`
shift	`<< >>`
relational	`< > <= >=`
equality	`== !=`
bitwise/boolean	`&` *then* `^` *then* `|`
boolean	`&&` *then* `||` *then* `?:`
assignment	`= *= /= %= += -= ...`

associativity

- rule 1
  - except for assignment all binary operators are left-associative

```
x + y + z      →   (x + y) + z
```

- rule 2
  - unary operators, assignment and ?: are right-associative

```
**z            →   *(*z)
x += y += z    →   x += (y += z)  ?
a?b:c?d:e      →   a?b:(c?d:e)
```

evaluation order

- very important
  - precedence controls operators not operands
  - order of evaluation of operands is unspecified
  - only sequence points guarantee evaluation order

```
int x = f() + g() * h();
```

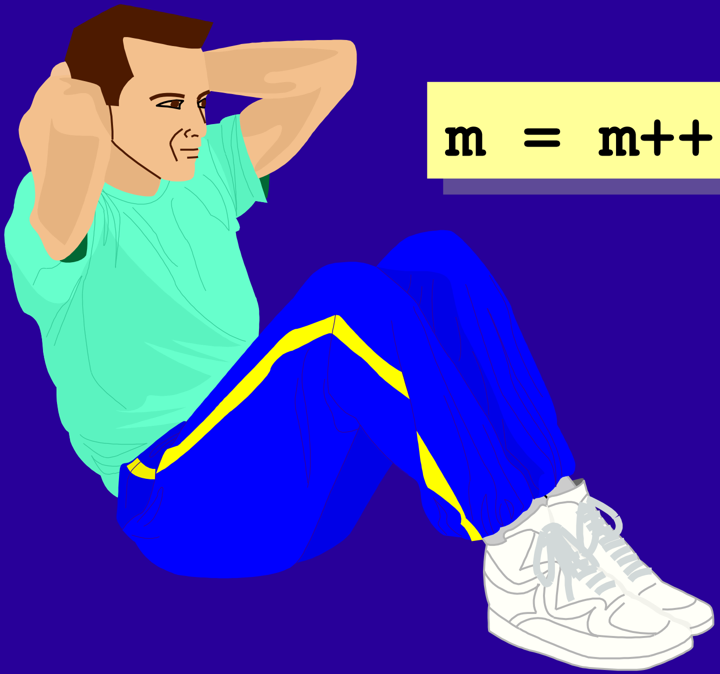
In this example the three functions f( ) and g( ) and h( ) can be called in *any* order

```
int v1 = f();
int v2 = g();
int v3 = h();
int x = v1 + v2 * v3;
```

?

- in the given statement…
  - where are the sequence points?
  - what are the operators?
  - what is their relative precedence?
  - how many times is m modified between sequence points?
  - is it undefined?

```
m = m++, m;
```



summary

- know your enemy!
  - undefined vs unspecified vs imp-defined
  - local variables do not have a default value
  - a char is the smallest addressable unit of memory
  - many integer types have minimum sizes
  - integers can be interpreted as true/false
  - integer arithmetic overflow can be undefined
  - type conversions are implicit and liberal!
  - initialisation != assignment
  - assignment is an expression
  - sequence points knowledge is vital
  - precedence controls operators not operands
  - order of evaluation between sequence points is unspecified
  - strive for simplicity

The standard may limit the set of allowable behaviours from which a compiler implementer must choose their implementation defined behaviour. Or alternatively the standard may impose no particular requirements.

Annex J of C99 lists all the implementation-defined, unspecified, and undefined behaviours (and also the locale-specific behaviours). There are 190 documented undefined behaviours!

To be 100% accurate, if an indeterminate value is read from an object that has unsigned char type the behaviour is *unspecified* rather than *undefined* (this is because the unsigned char type does not support trap representations).

The default value of a static variable is the value the variable would have if assigned zero. Note that this is not necessarily the same as all-bits zero - assigning a pointer the value zero assigns it the null pointer and the null-pointer is not necessarily all-bit zero.

Note that it is possible to declare a variable outside a function with the static keyword. However, whether or not the static keyword is used the variable has static storage class because of its location (outside a function). Using the static keyword on a variable declared outside a function affects its linkage (covered later).

The `char` type can be manipulated as a very small integer. `short` is a shorthand for `short int`. `long` is a shorthand for `long int`. The limit shown is both the positive and negative limit. In other words, the smallest integer value an `int` variable can hold is `-32767` and the largest integer value an `int` variable can hold is `+32767`.

Be aware that the comma is an operator in C so in code you should always write `32767` and never `32,767`. Also be aware that the decimal separator in C is a point, so that `32.767` is also not equivalent to `32767`.

The limit of the unsigned versions is simply 2 to the power of the min-bits size minus one. For example the min-bits size of a `short` is 16 and $2^{16} - 1$ is `65535` which is the largest value an `unsigned short int` variable can hold.

The standard header `<limits.h>` provides specific values for the sizes on your particular compiler-machine combination. For example `INT_MAX` is the maximum value for a `signed int`.

The long long type was introduced in C99. It's literal suffix is `LL` or `ll` (`LL` is preferred since lowercase ell often looks very similar to a one). C++ compilers are not currently required to support the `long long` type (but that will change).

<stdint.h> is new in C99.

Which types with exact sizes (i.e., for which values of N) are provided is implementation-defined.

All C99 compilers must provide the leastN_t and fastN_t types for N=8, 16, 32, and 64.

The headers also provide macros such as INT16_MIN which is the smallest 16 bit signed integer value.

Prior to C89 all implementations were required to convert all values of type float to type double. The sizes of the three floating point types are not required to be different. The only safe assumption is that the values representable as a float are a subset of those representable as double which are a subset of the those representable as long double.

C99 now allows binary exponents in hexadecimal floating point constants.

The values defined in <float.h> use FLT_, DBL_ and LDBL_ as prefixes for limits related to float, double and long double respectively.

DIG is the number of decimal digits of precision.

EPSILON is the minimum value x (greater than 0.0) such that 1.0 + x != x.

The complex and imaginary types are new in C99.

The complex type is expressed differently in C++, where it is an ordinary library type rather than a special language type:

```
double complex v; /* C */
complex<double> v; // C++
```

<stdbool.h> is new in C99.

<stdbool.h> provides a macro bool for the type _Bool, and for true (which expands to the decimal constant 1) and for false (which expands to the decimal constant 0).

Comparing explicitly against true is particularly error prone when you remember that any non-zero value should be interpreted as true. Clearly the true macros must use a specific non-zero value. Because of this it is more idiomatic to use != false rather than == true.

However, the best style of all is to just drop the use of == != false and true and simply use the expression in a boolean context. For example, rather than writing if (expression() != false) it is better to just write if (expression()). This reads better and is a higher level, less solution focused piece of code. This leaves true and false as direct literals used to initialize bool variables.

The character mapping is not specified in the standard. One on machine the character 'x' might be represented by its ASCII encoding whereas as on another machnie it might be represented by its EBCDIC encoding (and on a third machine by yet another encoding). You are however guaranteed that the encoding of the decimal digits is ascending. That is, '0' + 3 == '3'.

Note that character constants in C are of type int. This means in C that sizeof('x') == sizeof(int). Note however that in C++ sizeof('x') == sizeof(char) == 1. Note further than even in C sizeof(char) is 1.

C99 introduced a notation for universal-characters using the syntax \u hex-digit{4} or \U hex-digit{8}. A universal-character may even appear in the midst of an identifier.

The parentheses in sizeof(expression) are optional. In other words you can write sizeof expression or sizeof(expression). The parentheses are required when taking the sizeof a type.

The second code fragment is slightly better than the first code fragment. This is because if the type is written only once. It avoids errors where the type is changed in one place but not in the other.

Note that the standard does not tell what specific kind of integer size_t is a typedef of.

In a sizeof expression is one of the few places in C where the name of an array does not decay into a pointer to its first element. A common idiom for finding the size of an array is to divide the size of the whole array by the size of one of its elements.

```
const size_t element_count = sizeof array / sizeof array[0];
```

Note that in sizeof expression the expression is not evaluated at runtime (unless you are taking the size of a variable length array), it solely used at compile to determine the type of the expression. Thus sizeof array[n] will not cause undefined behaviour if n is out of bounds.

Although C is statically typed, it is only weakly checked, supporting some quite counterintuitive and loose conversions. Widening conversions are unsurprising and implicit, but there are a number of narrowing conversions (conversions that could lose precision) that are also implicit. There are also some cross-type conversions that are in principle unsafe but are not required to be diagnosed as problems, such as free and easy conversions between all pointer types.

It is also common to convert between numeric types and their string representations. For example 42 ←→ "42".

To convert an int to a string use sprintf/snprintf() from <stdio.h>

To convert a string to an long use strtol() from <stdlib.h>

Conversion to or from an integer type whose value is outside the range that can be represented causes undefined behaviour. For example:

```
        char c = 99999999999999999999999999999;
```

In C99 integer division and integer remainder are defined to perform truncation towards zero. For example 22 / -7 == -3

Pre C99 it was implementation defined whether truncation was towards zero or towards infinity. For example (towards infinity) 22 / -7 == -4

Explicitly comparing to true or false is generally considered poor style:

```
 if (in_range == true) ...
```

Consider: in_range == true is a boolean expression, which is itself true or false. So why not write:

```
    if ((in_range == true) == true) ...
```

Clearly this is ridiculous. The idiomatic (and "higher level") usage is to write:

```
    if (in_range) ...
```

Using == != to compare floating point values is bad practice:
1. Floating point operands can be NaN's. NaN's are never equal to other values, not even to themselves!

2. Floating point arithmetic has a finite number of significant digits and is subject to rounding errors. It is very easy for two expressions that, mathematically, should be equal to end up being be very slightly different. One way to see if two floating point operands are "the same" is to use a threshold that is relative to the magnitude of the numbers being compared. For example:

   |a – b| / sqrt(a^2 + b^2 + epsilon^2) < epsilon

gives a colloquial meaning of "a and b agree to about N significant digits" when epsilon == 1e-N.

NaNs are not ordered: when using the < <= > or >= operator if either operand is NaN then the result is *always* false.

C is unusual in that assignment is an expression. This means that an assignment expression as a whole has a value. This value can be used as part of a larger expression (as shown in the printf calls). However, the statement printf("%d", lower = 0); is better as two separate statements: lower = 0; printf("%d", lower);

Chaining assignments together can be useful where you wish to stylistically express the idea that the variables necessarily have the same value. For example:

```
lower = 0;
upper = 0;
```

indicates that lower and upper *coincidentally* have the same value (and if the value for one was changed it would not necessarily mean the value for the was also changed).

In contrast:

```
lower = upper = 0;
```

indicates that lower and upper *necessarily* have the same value.

The value of an assignment expression is the value (and type) of the left hand side. This is important if the type of the right hand side is not the same as the type of the left hand side and there is a conversion.

The compound operators (such as *=) are effectively equivalent to the expanded versions on the right hand side. However, the equivalence is only approximate. Note in particular that the lhs expression only occurs once in the right hand column whereas it occurs twice in the left hand column. When using a compound assignment if the left hand side contains any side effects the side effects will only take place once.

```
array[func(x)] = array[func(x)] + 42;  // func will be called twice :-(

array[func(x)] += 42;  // func will be called only once :-)
```

You should use the patterns in the right hand column. The patterns in the left hand column are not considered idiomatic in C.

C is renowned for being a terse language. Writing lhs += 1 or lhs -= 1 is still too much typing! Instead C provides the ++ and -- operators. Both these operators are considered idiomatic and are a useful way of removing what would otherwise be a magic number 1. Note that C++ took its name from the ++ operator.

Given that assignment is an expression (and so has an outcome) it is not that surprising that the difference between these two forms is the value of the whole expression itself. In both the cases ++m and m++ the variable m is incremented by one. The difference is only in the result of the whole expression.

```
int m = 42;
printf("%i", m++);          // m == 43, prints 42

int m = 42;
printf("%i", ++m);          // m == 43, prints 43
```

It is advisable to not use the increment and decrement operators as parts of a larger expression.

Sequence points are points of stability.

Annex C of C99 details the sequence point model.

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."

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.

Yes, the expression is undefined since neither = nor ++ introduce a sequence point. Therefore the expression will attempt to modify the same object (n) twice (once for the = and once more for the ++) between two sequence points.

Note further that the expression ((*p)++ = (*q)++) could also be undefined if p and q point to the same object.

Yes, the expression is undefined, since the sub-expression n reads the value of n but not to determine the value to be stored in n.

Similarly the expression (index[n++] = n) is also undefined.

Both expressions result in undefined behaviour.

The expression (n = n++) is undefined since neither = nor ++ introduce a sequence point. Therefore the expression will attempt to modify the same object (n) twice (once for the = and once more for the ++) between two sequence points.

Similarly the expression ((*p)++ = (*q)++) could also be undefined if p and q point to the same object.

The expression (n + n++) is also undefined, since the value of the modified object n is read twice between sequence points.

Similarly the expression (index[n++] = n) is also undefined.

The && and || operators do not not necessarily evaluate their right hand side operand. If they can determine the outcome from just the left hand side expression then the right hand side is not evaluated. For example in the following example, if dee() returns false then dum() is not called. This is known as short-circuit evaluation.

```
if (dee() && dum()) …
```

The result of the ternary operator is an rvalue and not an lvalue. This means the ternary operator cannot be used on the left hand side of an assignment:

```
(expression() ? a : b) = 42; // won't compile
```

The comma operator has the lowest precedence of all operators and is left associative.

The expression matrix[row,col] does not access an element of a 2-d array. The compiler sees the expression as matrix[(row,col)] and the sub expression in the parentheses is a comma operator – row is evaluated and its value discarded, then col is evaluated is the result of the parenthesised sub expression. In other words, the whole expression is equivalent to row,matrix[col]

They are all undefined. They all attempt to modify m twice between sequence points.

The only operators not shown on the above table is the comma operator which has the lowest precedence of all and sizeof which is a unary operator, and compound literals which has precedence between postfix ++ -- and prefix ++ --

Occasionally the precedence is not as expected and parentheses must be used. For example, the bitwise logical operators is below the equality operators. This means that the following expression:

```
if (x & MASK == 0) …
```

parses as:

```
if (x & (MASK == 0)) …
```

which is almost certainly not what was intended.

The unspecified evaluation order is a defining characteristic of C and C++. In contrast, the evaluation order in Java and C# is strictly left to right.

There is a sequence point at the comma, and a sequence point at the end of the full expression (effectively at the semi-colon).

The operators are the assignment operator, the increment operator, and the comma operator.

The increment operator has a higher precedence than the assignment operator which in turn has higher precedence than the comma operator.

m is modified twice, once by the increment operator, once by the assignment.

Yes, it is undefined.

The comma operator has the lowest precedence so the expression binds as follows:

```
(m = m++) , m
```

The sequence point introduced by the comma does not help up, since we have already attempted to modify m twice by the time we get to the sequence point.

Prior to C89 all implementations were required to convert all values of type float to type double. The sizes of the three floating point types are not required to be different. The only safe assumption is that the values representable as a float are a subset of those representable as double which are a subset of the those representable as long double.

C99 now allows binary exponents in hexadecimal floating point constants.

The values defined in <float.h> use FLT_ DBL_ and LDBL_ as prefixes for limits related to float, double, and long double respectively.

DIG is the number of decimal digits of precision.

EPSILON is the minimum value x (greater than 0.0) such that 1.0 + x != x