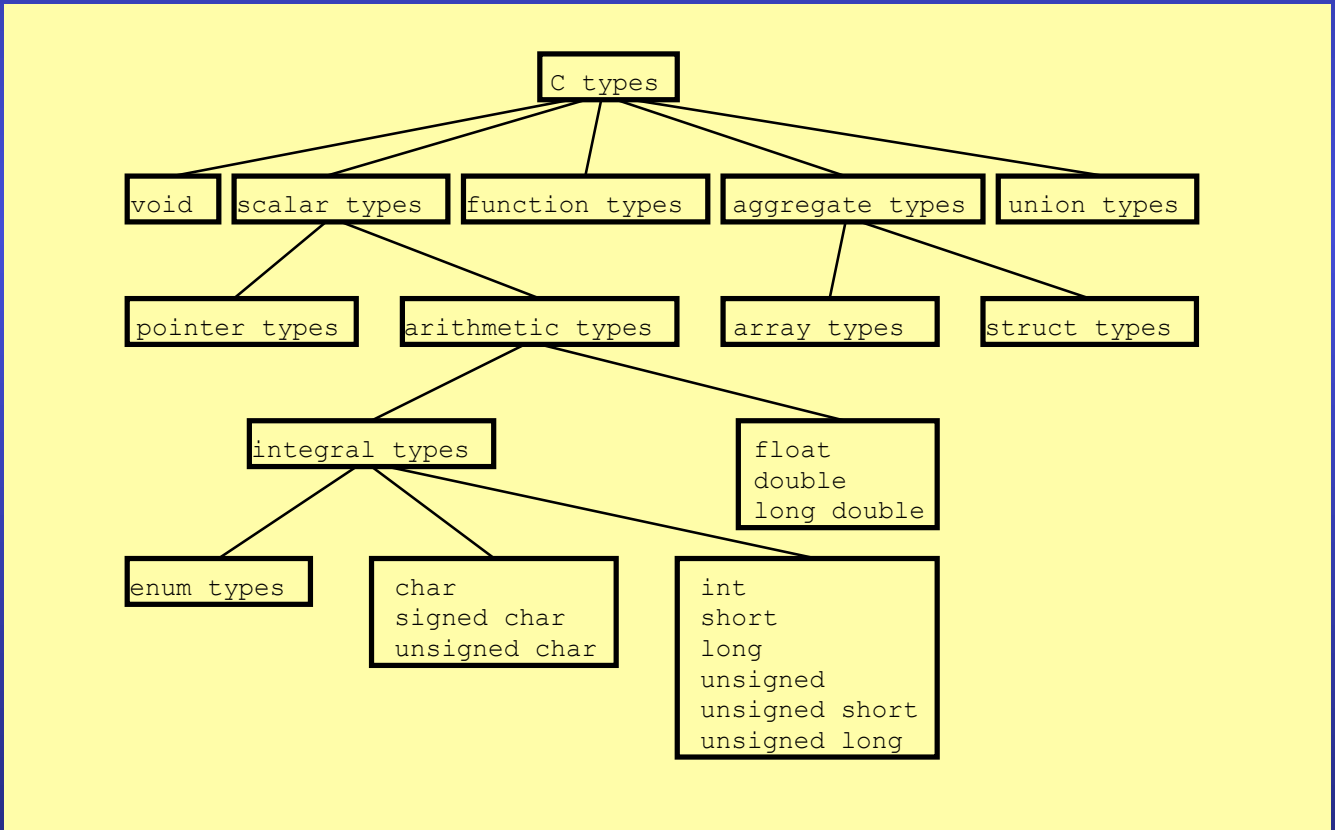
Types, Operators and Expressions

type system

- C is statically typed
  - every variable and every expression has a single definite type that can be deduced by the compiler at compile time



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 class

- 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 class

- 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 not 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**

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

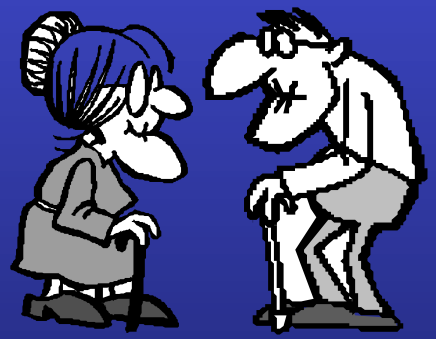
- there are three complex types
  - float complex
  - double complex
  - long double complex
- <complex.h> provides
  - the macro complex for _Complex
  - lots of function declarations

```
#include <complex.h>

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

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

```
#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

conversions

- 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;
```

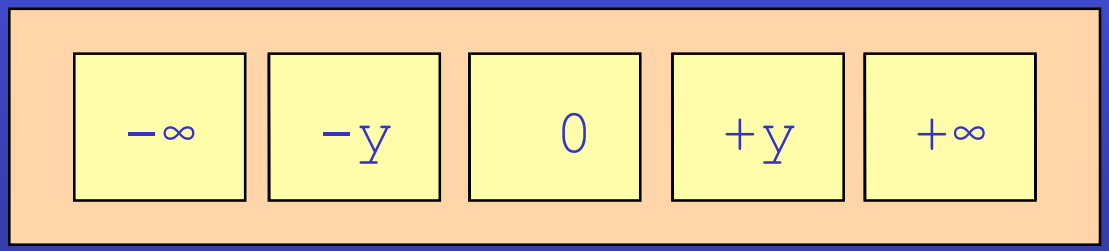
initialization 

```
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*



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

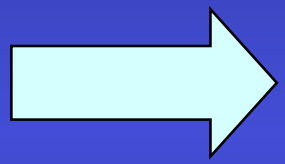
```
upper = 0;
lower = upper;
```

compound assignment

- common assignment patterns are supported natively with compound assignment operators

non-idiomatic

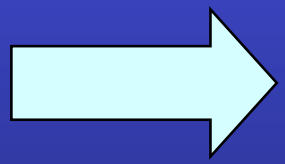
```
lhs = lhs * rhs;
lhs = lhs / rhs;
lhs = lhs % rhs;
```



idiomatic

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

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



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

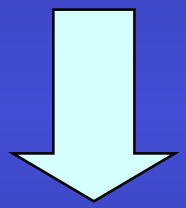
increment/decrement

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

non-idiomatic

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

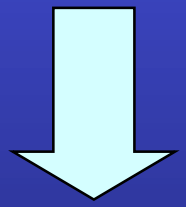
```
lhs = lhs - 1;
```



non-idiomatic

```
lhs += 1;
```

```
lhs -= 1;
```



idiomatic

```
lhs++;
```

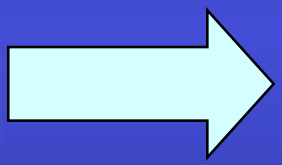
```
lhs--;
```

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 :-)

?

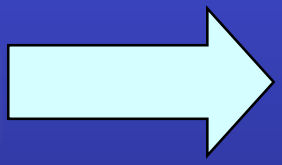
```
prefix = ++m;
```



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

equivalent†

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



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

†except that m is evaluated only once

bit twiddling

- sometimes you want to use an integer because of the bits it comprises
  - ~expression inverts the bits: 0-bit ← ~ → 1-bit
  - left-shift: integer-expression << bit-count
  - right-right: integer-expression >> bit-count

&	0	1
0	0	0
1	0	1

|	0	1
0	0	1
1	1	1

^	0	1
0	0	1
1	1	0

if the bit-count is negative or greater than or equal to the width of the left operand the behaviour is undefined

sequence points

- **a sequence point is…**
  - **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**

> C99: 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 a single object is modified more than once between sequence points the result is *undefined***

sequence points

- sequence points occur…
  - at the end of a full expression
    - a full expression is an expression that is not a sub-expression of a larger expression
  - 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

&& || 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

^ operators

- ^ is the exclusive-or operator
  - no short-circuit behaviour
  - does not have a sequence point
- ! is the logical not operator
  - !true == false
  - !false == true

^	false	true
false	false	true
true	true	false

idiomatic?

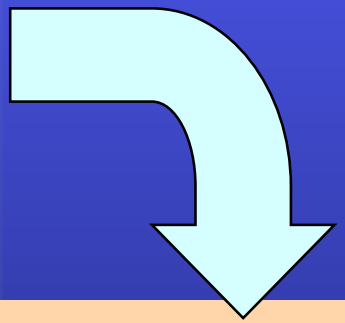
```
count += !!expression
```

what is this doing?

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

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

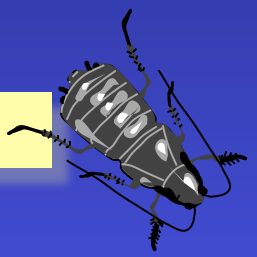


```
void some_func(void)
{
    bool found = search(...);
    printf("%sfound", found ? "" : "not ");
}
```

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);
```

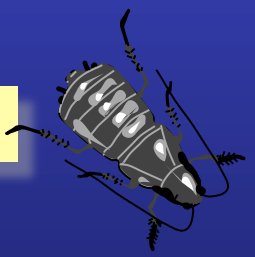


- common in for statements

```
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];
```



exercises

- 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++;
```

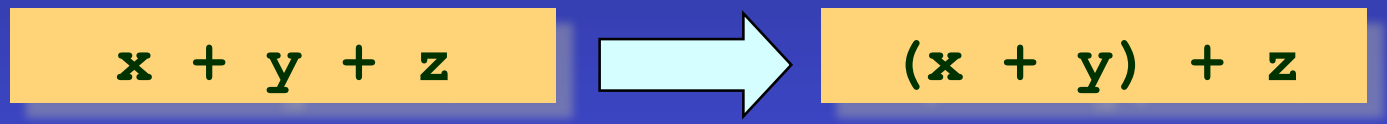
3
```
m = m = 0;
```

precedence

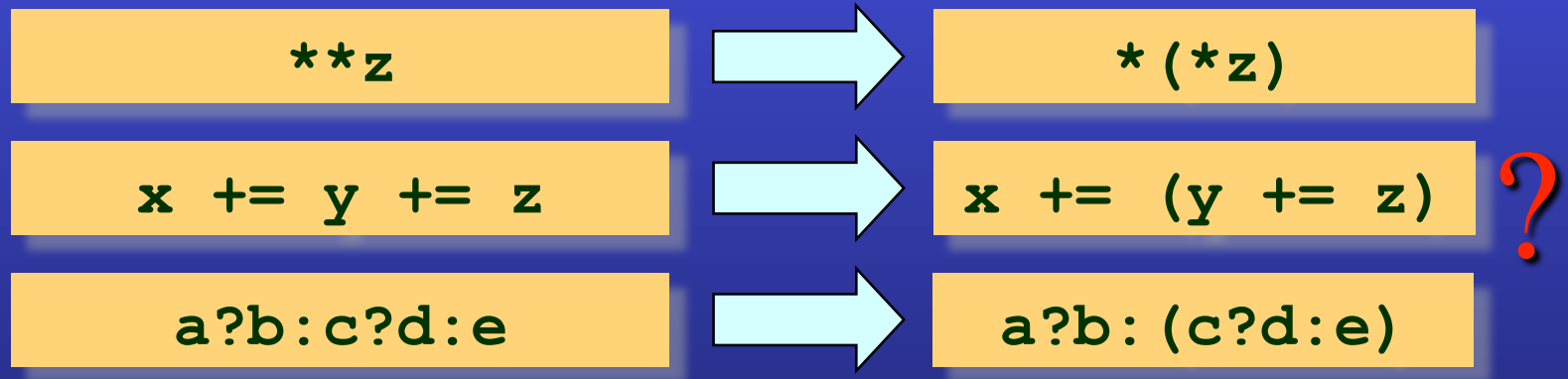
primary	`( )  [ ]  ->  .`
unary	`! ~ + - ++ -- (T) * &`
multiplicative	`* / %`
additive	`+ -`
shift	`<< >>`
relational	`< > <= >=`
equality	`== !=`
bitwise/boolean	`& then ^ then |`
boolean	`&& then || then ?:`
assignment	`= *= /= %= += -= ...`

associativity

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



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



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();
```

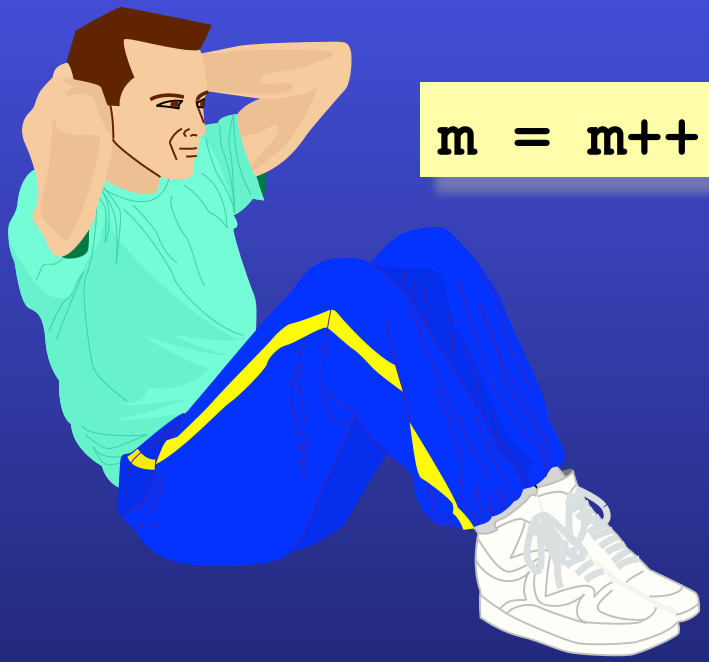
In this example the three functions f( ) and g( ) and h( ) can be called in *any* order

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

?

exercise

- 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