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

![Type System Diagram]

- **C types**
  - void
  - scalar types
  - function types
  - aggregate types
  - union types
  - pointer types
  - arithmetic types
  - array types
  - struct types
  - integral types
    - char
    - signed char
    - unsigned char
  - float
    - double
    - long double
  - enum types
  - int
    - short
    - long
    - unsigned
    - unsigned short
    - unsigned long
• **implementation-defined behaviour**
  - the construct is not incorrect; the code must compile; the compiler must document the behaviour

• **unspecified behaviour**
  - the same as implementation-defined except the behaviour need not be documented

• **undefined behaviour**
  - 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
• 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*

```c
int outside;

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

† unless declared with the static keyword
• 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

```c
int outside;

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

† 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>`

<table>
<thead>
<tr>
<th>type</th>
<th>min bits</th>
<th>min limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>8</td>
<td>$2^7 - 1$ (127)</td>
</tr>
<tr>
<td>short</td>
<td>16</td>
<td>$2^{15} - 1$ (32767)</td>
</tr>
<tr>
<td>int</td>
<td>16</td>
<td>$2^{15} - 1$ (32767)</td>
</tr>
<tr>
<td>long</td>
<td>32</td>
<td>$2^{31} - 1$</td>
</tr>
<tr>
<td>long long</td>
<td>64</td>
<td>$2^{63} - 1$</td>
</tr>
</tbody>
</table>
- `<stdint.h>` and `<inttypes.h>`
  - provide specific kinds of integers

**Some Examples**

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int16_t</code></td>
<td>signed int, exactly 16 bits</td>
</tr>
<tr>
<td><code>uint16_t</code></td>
<td>unsigned int, exactly 16 bits</td>
</tr>
<tr>
<td><code>int_least32_t</code></td>
<td>signed int, at least 32 bits</td>
</tr>
<tr>
<td><code>uint_least32_t</code></td>
<td>unsigned int, at least 32 bits</td>
</tr>
<tr>
<td><code>int_fast64_t</code></td>
<td>signed int, fastest at least 64 bits</td>
</tr>
<tr>
<td><code>uint_fast64_t</code></td>
<td>unsigned int, fastest at least 64 bits</td>
</tr>
</tbody>
</table>
• 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 (min $10^{-5}$)
   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
• there are three complex types
  - float complex
  - double complex
  - long double complex

• `<complex.h>` provides
  - the macro complex for `_Complex`
  - lots of function declarations

```c
#include <complex.h>
void eg(double complex z)
{
    double real = creal(z);
    double imag = cimag(z);
    ...
}
```
• `<stdbool.h>` provides three macros
  - bool for `_Bool`
  - false for 0
  - true for 1
  - the size of the bool type is *unspecified*

• any integer value can be converted to a bool
  - zero is interpreted as false
  - *any* non-zero is interpreted as true

```c
#include <stdbool.h>

bool love = true;
bool teeth = false;
```
The char type represents a single byte

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

<table>
<thead>
<tr>
<th>escaped chars</th>
<th>meaning</th>
</tr>
</thead>
</table>
| '
'           | newline         |
| '\t'         | tab             |
| '\b'         | backspace       |
| '\r'         | carriage return |
| '\f'         | form feed       |
| '\\'         | backslash       |
| `'`           | single quote    |
**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

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

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

this version is slightly better. why?
• literals for simple types are const!
  • their types can be specified

<table>
<thead>
<tr>
<th>type</th>
<th>suffix</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>long int</td>
<td>L or l</td>
<td>42L</td>
</tr>
<tr>
<td>unsigned</td>
<td>U or u</td>
<td>42U</td>
</tr>
<tr>
<td>float</td>
<td>F or f</td>
<td>42F</td>
</tr>
<tr>
<td>long double</td>
<td>L or l</td>
<td>42.0L</td>
</tr>
</tbody>
</table>

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

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

```c
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

```c
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

```c
int cast = (int)mass;
(void)printf("%i", cast);
```
• the usual arithmetic operators
  - + – * /
  - % is the remainder operator
  - note that integer / integer == integer

```cpp
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**

```c
int count;
count = 0;  // assignment
```

```c
int count = 0;  // initialization - better
```

```c
const int answer = 42;  // initialization
```

```c
const int answer;
answer = 42;  // compile-time error
```
• **== != 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:

- $-\infty$
- $-y$
- 0
- $+y$
- $+\infty$
• assignment is an expression
  • so it has an outcome – the value of the rhs
  • assignment also has a significant side effect!

```c
int lower;
int upper;

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

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

same as
```c
upper = 0;
lower = 0;
```
• common assignment patterns are supported natively with compound assignment operators

non-idiomatic

```c
lh = lh * rhs;
lh = lh / rhs;
lh = lh % rhs;
lh = lh + rhs;
lh = lh - rhs;
```

idiomatic

```c
lh *= rhs;
lh /= rhs;
lh %= rhs;
lh += rhs;
lh -= rhs;
```
• 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++;  // Correct
lhs--;  // Correct
```
++ 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 :-)
• 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

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

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 a single object is modified more than once between sequence points the result 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 (6.8p2)
- 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
• **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

<table>
<thead>
<tr>
<th></th>
<th>false</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>false</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>
• ^ is the exclusive-or operator
  - no short-circuit behaviour
  - does not have a sequence point

• ! is the logical not operator
  - !true == false
  - !false == true

<table>
<thead>
<tr>
<th></th>
<th>true</th>
<th>false</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
</tbody>
</table>
• the only operator with three arguments
  • $a ? b : c \rightarrow \text{if } (a) \text{ } b; \text{ else } c;$
  • \textit{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 ");
}
```
• **lhs , rhs**
  - *sequence point* at the comma
  - lhs is evaluated and the result is discarded
  - 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];
```
<table>
<thead>
<tr>
<th>precedence</th>
<th>operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>primary</td>
<td>( ) [ ] -&gt; .</td>
</tr>
<tr>
<td>unary</td>
<td>! ~ + - ++ -- (T) * &amp;</td>
</tr>
<tr>
<td>multiplicative</td>
<td>* / %</td>
</tr>
<tr>
<td>additive</td>
<td>+ -</td>
</tr>
<tr>
<td>shift</td>
<td>&lt;&lt; &gt;&gt;</td>
</tr>
<tr>
<td>relational</td>
<td>&lt; &gt; &lt;= &gt;=</td>
</tr>
<tr>
<td>equality</td>
<td>== !=</td>
</tr>
<tr>
<td>bitwise/boolean</td>
<td>&amp; then ^ then</td>
</tr>
<tr>
<td>boolean</td>
<td>&amp;&amp; then</td>
</tr>
<tr>
<td>assignment</td>
<td>= *= /= %= += -= ...</td>
</tr>
</tbody>
</table>
• **Rule 1**
  - Except for assignment, all binary operators are left-associative

  \[ x + y + z \] \rightarrow \[ (x + y) + z \]

• **Rule 2**
  - Unary operators, assignment, and `?:` are right-associative

  \[ **z \] \rightarrow \[ *(*z) \]
  \[ x += y += z \] \rightarrow \[ x += (y += z) \]
  \[ a?b:c?d:e \] \rightarrow \[ a?b: (c?d:e) \]
• very very very very very very important
  - precedence controls operators not operands
  - order of evaluation of operands is unspecified
  - only sequence points guarantee evaluation order

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

```c
int v1 = f();
int v2 = g();
int v3 = h();
int x = v1 + v2 * v3;
```
in these statements…
- where are the sequence points?
- what are the operators?
- what is their relative precedence?
- how many times is \( m \) modified between sequence points?
- which ones are *undefined*?

1. \( f(++m * m++) \);
2. \( m = m++; \)
3. \( m = m = 0; \)
4. \( m = m++, m; \)
• 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
This course was written by Jon Jagger

Expertise: Agility, Process, OO, Patterns
Training+Designing+Consulting+Mentoring

Jon Jagger
jon@jaggersoft.com
www.jaggersoft.com