17 Conversions, casts, and operator precedence

17.1 Arithmetic expressions

- These are expressions combining numbers using $+, -, *, /, \%$ and parentheses.
- There are rules of 'precedence' which follow the old conventions of algebra. Expressions are evaluated from left to right, parenthesised expressions are evaluated first, and multiplication and division come before addition and subtraction (BODMAS).
- Numeric types are `double, int, char,` also `float, long, short` which don't concern us now. Expressions can contain a mixture of different types.

Conversion.

- An integer-valued expression can be converted to a double-precision expression with the same value. Example

```
  double x = 1;
  printf("%f\n",x);
will print 1.000000
```

- A double-precision expression can (if within integer range) be converted to an integer value, rounded towards zero: positive doubles get rounded down and negative doubles get rounded up. Example.

```
  int x = 1.23, y = -1.23; printf("%d %d\n", x,y); prints
1 -1
```

- Where `int, double, char` occur in the same expression, `char` is converted to `int` or `double` and `int` is converted to double.

Char to int conversion. On intel machines at least, `char`s are regarded as 8-bit 2s complement integers! The trouble is that if a `char` has face value between 128 and 255, it is regarded as negative, and if assigned to an int, the int will be negative. This can be a **nuisance**. It can be avoided by using the `unsigned char` type, which is not covered in this module.[1]

17.2 Types of constants

Here are some constants. They have an implicit type.

```
'\n'          char
'a'           char
-45           int
"hello"       character string
1.23          double
2.00          double
```

[1] The usual ascii characters are ≤ 127. I'm not sure about the others.

17.3 Casts

An example of a cast is

```
(int) 1.23
```

The `(int)` is called a *cast* and it 'casts' the expression into an integer, i.e., the double expression is converted to an int.

17.4 Casts and pointers

A typical example of pointer usage:

```
char * x = (char*) calloc(1, 100);
```

This `(char*)` is a cast, but the address is unchanged. The cast does not change the address, but it changes its meaning. The bizarre example below is to illustrate this.

```
#include <stdio.h>
main()
{ int num = 260;
  int * x = & num;
  printf("x: %d\n", x[0]);
  printf("x cast to char*: %d\n", ((char*)x)[0]);
}
% a.out
x: 260
x cast to char*: 4
```

17.5 Routines

The argument list in a routine specifies the types of the arguments. When the routine is called, the arguments in the calling routine are automatically converted to the correct types.

The `printf()` routine does a lot of conversions. For example, floats are automatically converted to doubles, and chars to ints. (This conversion is invisible since ints are little endian.) You can print a char variable with `%d` or `%c` format; you will get an answer for each, though the `%d` gives its ascii rather than printed value.

17.6 Assignment operators

The operators `x++, ++x, x--, --x, x=y, x += y,` etcetera have the property that they do something and also return a value.

So

```
The value of x++ is x and its effect is to add 1 to x.
The value of ++x is x+1 and its effect is to add 1 to x.
x--, --x, similar.
```

```
The value of x=y is that of y and its effect is to
assign this value to x.
the value of x+=y is x+y and its effect is to assign
that value to x.
etcetera.

Assignments are evaluated right to left!  As a result,
x = y = z = 0;
sets x, y, and z, all to zero.
```

17.7 & operator and * operator

Given

```
int * x;
       the expression
*x
       is the int value  stored at x.
*x and x[0] are identical.

&x is, of course, the address of the variable x.
```

17.8 Operator precedence

Note. This is hopefully a correct transcription of rules from internet sources. The only purpose of these rules is to avoid too many parentheses. It is unwise to rely on one's mastery of these rules. ***If in doubt, add parentheses.***

C applies the BODMAS rules for arithmetic expressions. Parenthesised expressions are evaluated first, then `*,/,%`, left to right, then `+,-` left to right.

C is full of operators, and they have carefully defined precedence rules for the order of evaluation. The following list covers the operators taught in this module.

1. Highest precedence, ***left to right***. They have the same precedence, with the left-to-right rule for breaking ties.

   () Function call (previously forgotten),
   [] (i.e., accessing array element),
   Postfix increment/decrement `x++, x--`

2. ***Right to left:*** Prefix increment/decrement `++x, --x`,
   ! (logical negation, previously forgotten)
   Casts,
   `*p` (the value stored at location `p`),
   `&` (address),
   `sizeof()`.

3. Left to right: $*, /, \%$ multiplication, division, remainder modulo

4. Left to right: $+, -$ addition, subtraction

5. Left to right: $<, <=, >=, >$ logical relations

6. Left to right: $==$, $!=$ logical relations

7. Left to right: && logical AND

8. Left to right: || logical OR

9. ***Right to left:*** $=$, $+=$, $-=$, etcetera Assignment and assignment operators

Examples.

Disambiguate the following expressions by inserting parentheses, and say whether the expression is meaningful (legal), assuming the variables have suitable types.

```
  (i) while (   *x++!='\0'   )..
 (ii) a = b = c == 0
(iii) a = b == c = 0
 (iv) a = b = c == d && e || f || g
```

```
  (i) while (   *x++!='\0'   )..
       while (  ( *(x++) ) != '\0' )..
This is correct
 (ii) a = b = c == 0
        a = ( b = ( c == 0 ))
This is correct
(iii) a = b == c = 0
        a = ( ( b == c ) = 0 )
Illegal. You cannot assign a value to an expression
(b==c).
 (iv) a = b = c == d && e || f || g
      a = ( b = ((( ( c == d ) && e  )|| f )|| g ) )
This is correct.
```