18 Casts and mixed arithmetic expressions

18.1 Pointers

What made C unique is the way arrays and pointers were made almost synonymous.

```
  int *a, **b, c, d[10], e[10][10];
  char  *list[10];

a is pointer to int (address of an int)

list is an array of pointers to char

a[2] is the integer stored at a+8, which may be a random part of memory

The value of d is its starting address.
  a = d;
is ok; then a[0] and d[0], a[1] and d[1], ... are identical;
they   share the same memory area.

Again,
  a = e[5];
is ok; a is (the address of) the sixth row of e.
```

18.2 Conversions

- When assigning a value to a variable, the types should match, with some exceptions in the case of numeric types (arithmetic types).
- Thus

  ```
  main()
  {
    int i;
    int a[3];
    i = a;
  }
  ```

  will compile, but with a warning:

  ```
  a.c:5:5: warning: assignment makes integer from pointer
  without a cast [enabled by default]
  ```

  on the other hand

  ```
  main()
  {
  ```

```
  char i;
  double a;
  i = a;
}
```

compiles with no warnings. The assignment involves *type conversion.*

- In C, `char, short, int, long, float, double` are arithmetic types and they are interchangeable.

  There are also **unsigned** versions. Unsigned short has the range 0 to 65535.

- **Subroutines.** C will perform type conversions in subroutine and function arguments.

- Type conversion applied to pointers is peculiar (and can produce very odd effects). When pointer types are converted, the actual *value* of the pointer is unchanged, but C *treats it differently.*

```
int a[10];
int * b = a;
char * c = b;

a, b, c have the same value, but
a[1], b[1], and c[1] are not the same.
```

- **printf()** performs a lot of *default type conversion,* or rather promotion. `char, short` are automatically promoted to `int` and `float` to doubles.

  Thus

```
printf("%c %d", 'a', 'a');
```

  is perfectly ok.

18.3 Expressions of mixed arithmetic type

An expression can be a combination of other expressions using $+, -, *, /, \%$. Where two subexpressions of different types are combined,

- Surprisingly, `chars` are promoted to ints.

  This can be problematic if there is 'sign extension.' A char with a face value > 127 has 'high order bit 1' and on some machines (including the maths machines, which have Intel processors) it is converted to a negative integer.

  Sign extension can be prevented by using the data type

  unsigned char

- ints are converted to doubles.

- Floats are *always* converted to doubles in any arithmetic calculation.

```
1 - 2.3 - 4 is evaluated from left to right
1 - 2.3 is -1.3
-1.3 - 4  is -5.3

1 + 2/3  becomes 1 + 0 then 1
1 + 2.0/3 = becomes 1 + 0.666667 then 1.666667
```

18.4 Casts

- When assigning a value to a variable, the types should match, with some exceptions in the case of numerical values.
- The type of a variable is clear: it has to be declared. For the type of an expression, it is not clear. However, the types of constants are generally easy to recognise.

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

- A double value can be assigned to an int, and vice-versa. Conversion of int to double is direct, that is, 3 becomes 3.0. Double to ints are **rounded towards zero,** so 1.23 becomes 1 and -1.23 becomes -1.
- An expression can be converted to another type using **casts**. The syntax is

```
   (  ... type .... )   expression
```

such as

```
   ( double ) 1;
```

- Also, expressions can have subexpressions of different types. For example,

```
  1 + 2.34
```

is 3.34.

Example.

```
cat cast1.c
#include <stdio.h>
main()
{
  int x[2] = {60000,70000};
  char * y= (char*) x;
  printf("%d %d\n",x[0],y[0] );
  printf("%d %d\n",x[1],y[1] );
}
```

```
prompt% gcc cast1.c (no complaints)
prompt% a.out
60000 96
70000 -22

int 60000 hex  60 ea 00 00 (little endian)
int 70000 hex  70 11 01 00 (little endian)

ea hex = 234 -> -22 2s complement 8 bit
```