

## Integer variables and expressions

Comments and questions to [John Rowe](#).

As we mentioned last week, some things, such as people, are conventionally considered to be considered as coming only in units of integers, not fractions. It is meaningless to say "the average class has 28.3 pupils", whilst it may well be the case that "classes have an average of 28.3 pupils". Similarly, bank accounts are traditionally considered to contain a whole number of pence, or cents (possibly a negative one!).

Almost every C program will have such integer variables and expressions, which have generic type `int`. We have met `int` before as the return type of the `main()` function. We should emphasise that integer variables and expressions are only allowed to have integer values. Apart from a little discussion of what happens when we try to give them non-integer values, they are fairly simple and obvious.

The number of bytes used for integers is much less standardised than for floating-point numbers. Four bytes for an `int` is common and we shall occasionally use it in our examples, but if it matters to you be sure to check.

A four-byte `int` can store integer values roughly in the range  $\pm 2$  billion ( $2^{31}$ ). Since that is not always enough C also has `long int` (typically eight bytes, giving a range of roughly  $\pm 8$  billion billion or  $2^{63}$ ) which can simply be abbreviated to `long` for convenience.

C also defines a specialised integer type called `char` which is used for storing printable characters such as 'a', 'b', 'c', etc. We shall meet this type in a later lecture.

One bit of the integer is used to indicate whether it's positive or negative. We may add the qualifier "unsigned" before any integer type to regain the extra bit in which case the value are always positive.

## Examples where we would need to use a long

The following examples assume a four-byte `int` and an eight-byte `long`.

### Bank accounts

Bank accounts are considered to contain a whole number of cents. A four-byte `int` would only allow balances of  $\pm 2$  billion cents or pence, i.e. approximately  $\pm \$20,000,000$ . Whilst this is probably enough for you or me, it wouldn't be enough for a medium-sized company. Therefore we would need to use a `long`.

### Bytes within a file

If we wish to number the bytes in a file `0, 1, 2...` etc. then using an `unsigned int` would limit us to a maximum file length of 4GiB. Indeed, it's not that long ago that this was the case. Using an `unsigned long int` allows a file length of several exabytes.

A similar problem related to computer memory or RAM. Computer operating systems and applications like to be able to number their memory by bytes (byte number zero, byte number 1, byte number 2, etc.) Older 32-bit processors therefore limited applications to using 4GiB of RAM. For that reason, all modern PC processors are 64-bit.

It's good to know that  $2^{10}$  is approximately equal to 1000. It's also good to know that it's not exactly equal to 1000. Use the prefixes **Kibi**, **Mibi**, **Gibi**, etc. if it's important to be precise. (Which is not very often!)

## Declaring and using integer variables

Integer variables are declared in the obvious manner:

```
int j, k;
long bigint;

j = -2;
bigint = 123456789123456789;
k = 4 * j - 11;
```

Notice that the specific types are called `int` and `long`, the word "integer" is a generic term, there is no variable type of this name.

### Initialisation

Variables of all types (not just integers) can be *initialised* at the same time as being declared so we could have written the above code as:

C also defines `short` and `long long` but all that's guaranteed is that `long long` is at least as good as `long`. They are often the same.

```
int j = -2, k;
long bigint = 123456789123456789;

k = 4 * j - 11;
```

Notice that `k` was left *uninitialised* (which is a fancy word for random!) when it was declared. It would have been Very Bad News to have tried to use the value of `k` without explicitly setting it first.

Uninitialised variables of all types are **not automatically set to zero**.

## Integer constants

As shown above, these work as we would expect but anything with a decimal point or an exponent is treated as a `double` even if its value is an integer. This means that `1000.0` and `1e3` are both `doubles` with value "one thousand point zero". This only really matters when we are dealing with division, which we discuss in the next section.

A minor warning: for historical reasons, integer constants starting with zero are treated as octal (base eight), not decimal. So `017` is fifteen, not seventeen.

Something that is occasionally useful is that integer constants starting with "0x" are treated as *hexadecimal* ("hex") or base sixteen, using the letters "a-f" for ten to fifteen. This is useful as sixteen is 2 to the power 4, i.e. four bits so any byte can be represented by two hexadecimal "digits". Hex is used to specify colours on the Web, for example.

## Integer division and remainders

**For the rest of our lecture notes we shall assume that variables `j` and `k` have integer type and that `x` is a double.**

What happens when we divide one integer by another, e.g.  $5/2$ ? Is it an integer or a fraction? (Answer: it's an integer, but `C` gives us the choice of both.)

### Why are we interested?

In the "class size" issue at the top of the page, the average of a sum of integers (pupils in a class) was a fraction (28.3). So there clearly are occasions where we wish to divide two integers to obtain a fraction.

On the other hand, when trying to divide 13 people equally between 4 cars the "solution" of 3.25 people per car is unlikely to be popular! Indeed, somebody will probably say "only a computer could have come up with that answer".

The answer in this case is to say "thirteen divided by four is three, remainder one", i.e. to take the integer quotient  $13/4$  to be the whole number less than or equal to the exact result (3) and to use the "remainder" of 1 to say how many cars will need to take an extra person.

Thus, whenever we dealing with the division of integers we need to decide whether our algorithm requires the operation to be treated in an integer or floating-point way. `C` gives us the chance to do either integer division, which always results in an integer, or to convert the integers to their floating-point values.

## Integer expressions

Any arithmetic expression involving only integers is treated as an integer expression, for example:

$$(j + 7) / (k * 6 - j * 4 + 12)$$

will always yield an integer. For simplicity we will use the example of the division of two integer variables below, but exactly the same rules apply to any integer expressions.

## Integer division of positive expressions

This is straight-forward and is by far the most common case of integer division. (You don't often want to divide minus-thirteen people between four cars!) **In this section we shall consider only positive values of `j` and `k`**, negative values will be discussed in the next section.

If both `j` and `k` are positive the result of the division `j/k` is the smallest integer less than or equal to the true, floating-point value of `j/k`. This can be thought of as "the number of whole `ks` in `j`".

```
7/2  equals 3    (with remainder 1)
13/5 equals 2    (with remainder 3)
9/3  equals 3    (with remainder 0)
```

**Note that this is not necessarily the nearest integer**, for example  $13.0/5.0$  equals 2.6 which is nearer to 3 than to 2 but  $13/5$  equals 2, not 3.

This can equally be thought of as "discarding the fraction" or rounding towards zero.

## % is the remainder operator

When we want to find the remainder we can use the expression  $j \% k$  "j remainder k" (as above, k must not be zero), defined by:

$j \% k$  equals  $j - (j / k) * k$

e.g. using the same examples as above:

```
7 % 2 equals 1
13 % 5 equals 3
9 % 3 equals 0
```

This means that  $j \% k$  is always in the range 0 to  $k-1$ .

Between them integer division and the remainder operator enable programs such as:

```
/*
 * Convert a total number of seconds to minutes and seconds
 */
#include <stdio.h>

int main() {
    int totaltime, seconds, minutes;

    totaltime = 429;

    minutes = totaltime / 60;
    seconds = totaltime % 60;

    printf(" %i seconds in total, is %i minutes and %i seconds.\n",
           totaltime, minutes, seconds);
    return 0;
}
```

The output is:

```
429 seconds in total, is 7 minutes and 9 seconds.
```

You will observe the use of "**%i**" to print an integer. Strangely, "**%d**" (for decimal) is a synonym for **%i** (the "d" does **not** stand for "double!"). Use "**%ld**" or "**%li**" to print a long.

## Integer division involving negative values

As mentioned above, division involving negative integers is very rarely used so you don't need to read this section if you don't want to. But should we ever need to know:

We know that  $5/2$  equals 2 but should  $-5/2$  be -2 or -3?

- i.  $-5/2$ , equals -2 rounds towards zero and preserves the equality  $-j/k$  equals  $-(j/k)$
- ii.  $-5/2$  equals -3 preserves the fact that "integer"  $j/k$  is always less than or equal to "floating-point"  $j/k$ , giving a positive remainder  $j \% k$ .

Early versions of C allowed the compiler to choose either option (!); that choice has now been removed and the rule is now option (i), negative numbers round towards zero:

```
-5/2 equals 5/-2 equals -2
-j/k equals j/-k equals -(j/k)

-5 % 2 equals 5 % -2 equals -1
-j % k equals j % -k equals -(j % k)
```

The fact that negative-integer division and remainder used not to be uniquely defined shows you how rare it is.

## Floating-point to integer conversion (also rare)

C adopts exactly the same policy when dealing with statements like:

```
x = 1.82;
j = x;
```

the whole point of integers is that they are not allowed to have fractional values so `j` is assigned the value **1**. Again note that **this is not necessarily the nearest integer**.

Negative values are also handled in the same way as integer division, rounding towards zero:

```
x = -1.82;
j = x;
```

gives `j` the value of -1;

Finally, if we do ever find ourselves converting floating-point expressions to integers we must be aware that rounding errors can tip the result by one from the mathematically correct result when that result is exactly, or close to, an integer.

## Converting integers to floating point

### Automatic (implicit) conversion

When evaluating expressions with two different arithmetic types C just "does the right thing" and converts things for us. For example, in the expression:

```
j * x;
```

Since `j` is an integer and `x` a double the value of `j` is automatically converted to a double before the multiplication. (The variable `j` itself is unchanged of course.)

Integer to floating-point conversions and assignments are also common and again they "just work":

```
j = 12;
x = j;
```

just sets `x` to 12.

### Explicit conversion

Given the previous discussion on integer division we can see that the following code which tries to calculate an average is incorrect:

```
int number, total;
double average;

...

average = total/number;          /* Wrong! */
```

as `total` and `number` are both integers and that therefore the expression `total/number` is also an integer. The way to divide two integers in a floating-point way is to explicitly convert one or both of the integers to a double. I would normally suggest the divisor. Any integer expression can be converted to a double expression of the same value by the simple, if slightly unintuitive, method of writing "`(double) expression`":

```
int number, total;
double average;

...

average = total/((double) number);
```

We can put any valid type inside the parentheses to convert an expression to a different type.

Thus the expression `(double) number` means "the value of `number` treated as a double, not an integer". If `number` had the value 7 then `(double) number` is treated as if it were 7.0. Since we are now dividing an integer by a double, the integer expression on the top is also converted to a double and all is well.

The extra parentheses on the bottom are optional, we could have simply written:

```
average = total/(double) number;
```

**Thirty-second discussion:** turn to your neighbour and discuss whether you think it is clearer with or without the redundant parentheses.

## Tips and warnings concerning integer division

Integer division is simple but we don't want it to occur by accident, when we were expecting floating-point division.

### It depends on the expression, not the context

If as humans we look at the statement:

```
x = j/k;
```

we might be tempted to look at the left-hand side, see it's a `double` and treat the right-hand side `j/k` as a floating-point division. (This is a particular danger because we see the `x` first.) Computers don't work that way - they follow fixed rules. The right-hand side is evaluated as an integer and only after that does the compiler look at the left-hand side.

### Always use decimal points for **double** constants with integer values

If we do this all the time, even in expressions not involving integer division, we will avoid mistakes like:

```
x = y + 2/3;          /* Oops! */
```

```
x = y + 2.0/3.0;     /* Good! */
```

### Don't deliberately put integer division inside floating-point expressions.

It's only safe inside parentheses:

```
x = y * (j/k);      /* (j/k) is treated as an integer */
```

And I would suggest avoiding it even then. It's not that the rules for when something gets treated as integer or floating-point are complicated (they're not), **it's just giving ourselves another chance to go wrong.**

## Complex numbers

The include file `complex.h` allows us to conveniently use complex numbers using the **double complex** type with `I` being the square root of minus one. Complex versions of most of the `math.h` functions are also inside `complex.h`, all with a `c` in front of the name such as `csqrt()`, `csin()`, `casin()`, `cexp()`, etc.

The functions `creal()`, `cimag()` give the real and imaginary parts respectively with `conj()`, `cabs()` and `carg()` also available.

Once we've done that complex arithmetic goes just as we would expect, although we have to split expressions into their real and imaginary parts for printing:

```
/*
 * Demonstrate complex arithmetic and the complex square root.
 */
#include <stdio.h>
#include <complex.h>

int main() {
    double complex x, y, sqrtx;

    x = 1.2 + 3.4 * I;

    y = -13.7 * x + 5.6 * x * x - 12.7 * I;
    sqrtx = csqrt(x);

    printf("x is %g %g * I\n", creal(x), cimag(x));
    printf("y is %g %g * I\n", creal(y), cimag(y));
    printf("sqrt(x) is %g %g * I\n", creal(sqrtx), cimag(sqrtx));

    return 0;
}
```

Complex numbers weren't available in early versions of C as they are highly specialised and only used by mathematicians, scientists and engineers.

The imaginary constant is a **capital I**; the C convention is that variables have lower case names and constants upper case.

The output is:

```
x is 1.2 +3.4 * I
y is -73.112 -13.584 * I
sqrt(x) is 1.55009 +1.09671 * I
```

The format `%+g` in `printf()` means "always print a plus if the number is positive".

Observe that we could have written the last `printf()` function as

```
printf("sqrt(x) is %g %+g * I\n", creal(csqrt(x)), cimag(csqrt(x)));
```

as **the arguments to functions are always the value of expressions**, we never "pass a variable" to a function in C.

## Sensible variable names

Choosing a sensible variable name is a compromise between clarity and compactness. For example, if variables are called `number_of_atoms` and `maximum_number_of_atoms`, that's not just a lot to type but any arithmetic expressions involving them and other similarly-named variables become very hard to understand. This is particularly a problem for scientific programs with large numbers of complicated arithmetic expressions.

One answer can be to develop a **consistent** sort-hand, for example using the prefixes "n" for "number of" and "max" for maximum yields `n_atoms` and `max_atoms` or `natoms` and `maxatoms` depending on your taste.

On the other hand, avoid short but unhelpful variable names such as in the following example.

### Example

Suppose we are unfortunate enough to have to deal with masses in both kilograms (kg) and imperial pounds (lb). (Naturally we should avoid this situation if at all possible!)

A poor choice would be something like:

```
double mass1; /* Kilograms */
double mass2; /* Pounds */
```

When we come to use these variables we will constantly be wondering which is which and referring back to the comment where the variables were declared. Much better would be:

```
double mass_kg, mass_lb;
```

which doesn't even need a comment.

Try to write your program so that it explains itself without needing any comments, and only use comments inside of functions for situations where that isn't possible.

## A preview of if statements

We shall be dealing with this more thoroughly in a later lecture, but it's useful to introduce the subject a little more gently.

Any statement can have the construction `if (expression)` put in front of it in which case the sub-statement following the `if()` is only executed if the expression is true:

```
if ( x > 0 ) printf("x is positive\n");
```

It is conventional to put the sub-statement on a separate line indented from the original:

```
if ( x > 0 )
    printf("x is greater than zero\n");
```

but **notice that there is no semi-colon after the `if()` as it is all one statement.**

This is very important:

```
x = -1.4;
if ( x > 0 ); /* Subtle but horrible mistake */
    printf("x is greater than zero\n");
```

```
printf("x is greater than zero\n");
```

In this buggy code **the `if()` statement terminates with the semicolon**. Therefore the `printf()` will always occur. (Notice that the indentation has no effect.)

The meaning of `>` should be fairly obvious, C also has `<` (less than), `>=` (greater than or equal to: notice no space) and `<=` (less than or equal to), as well other operators which we shall discuss next week.

## else

The word **else** has the meaning of "otherwise". Here we use the "less than" operator:

```
if ( x < 0 )
    printf("x is less than zero\n");
else
    printf("x is greater than or equal to than zero\n");
```

## Using { ... } blocks

One useful feature of C is that we can put a `{ ... }` block wherever we can put a single statement. There is no semi-colon after the closing `}`, just like there isn't one after the `{ ... }` block that forms the `main()` function.

So, we could have chosen to write the above as:

```
if ( x > 0 ) {
    printf("x is greater than zero\n");
}
else {
    printf("x is less than or equal to than zero\n");
}
```

Whilst it's not necessary in this case, **don't hesitate to use `{ }` if you think it's clearer**. Remember the megaprinciple!

**Sixty-second discussion.** Turn to your neighbour and discuss which of the above two is clearer.

We can, of course, put more than one statement in a `{ ... }` block:

```
if ( x > 0 ) {
    printf("x is greater than zero\n");
    y = sqrt(x);
}
else {
    printf("x is less than or equal to than zero\n");
}
```

This concludes our introduction to the `if()` statement, we shall have a more thorough discussion later but in the mean time **be sure to indent them properly**.