Expressions - Chapter Summary

1 Binary and Unary Operators

Operators are used to modify and test variables, constants and expressions, which are collectively known as operands. Binary operators take two operands; unary operators act on a single operand.

2 Arithmetic Operators

2.1. Addition, Subtraction, Multiplication and Division: - +, -, * and /.

The + operator adds the value on its right to the value on its left. The - operator subtracts the value on its right from that on its left. The * operator multiplies the value on its left by the value on its right. The / operator divides the value on its left by the value on its right. Examples:

```
a + b           /* adds b to a */
c - 10          /* subtracts 10 from c */
x + (y * z)     /* multiplies y by z and then adds the resulting value to x */
distance/time   /* divides distance by time (to give velocity?) */
```

Note that when integers are divided and the answer is not a whole number, the answer is always rounded down to the integer below.

```
main()
{
     int distance, time;

     distance = 15;
     time = 4;
     printf("\nVelocity is %d.\n", distance/time);
}
```

In the above example, the true velocity should be 3.75, but when the program is compiled and run, it will print out:

```
Velocity is 3.
```

3.75 has been rounded down to 3. The fraction has been discarded. This problem would have been avoided if floating-point variables had been used instead of integers, and %f instead of %d in the printf statement.

2.2 Modulus: %

When used with integers, the modulus operator % yields the remainder when the value at its left is divided by the value to its right. Examples:

```
11 % 3      /* yields the value 2, the remainder when 11 is divided by */
```

2.3 Unary Plus and Minus

When used as unary operators, the plus and minus operators + and - multiply the values to their right by +1 and -1 respectively.

The unary minus operator changes the sign of the value to its right. For example, if x is positive, -x is negative, and -(-7) has the value of +7.

The unary plus operator has two main uses: to make explicit the sign of an operand where this improves legibility (as in the case of +7 in the above example), and to raise the precedence of one of two expressions, which would otherwise have equal precedence, and where there would thus be no other way of ensuring that one expression was evaluated before the other. If two expressions have equal precedence, qualifying one with a unary plus tips the balance in favour of that expression being evaluated first. This is important if the evaluation of one expression affects the value of the other expression of equal precedence.

3 Expressions

An expression consists of any combination of constants, variables and operators.

Every expression has a value. If the expression consists of constants only, the value of the expression is the value of that constant. If it consists of a variable, the value of the expression is the value currently assigned to that variable.

If it includes operators, its value is determined by enacting the operations determined by the various operators according to their order of precedence, as explained in Section 5 below.

4 Assignment Operator: =

The assignment operator = causes the expression on its right to be evaluated and then stores the resulting value in the variable on its left.

For instance, assuming that `test` is an integer variable, the assignment statement

```
test = 4 * 3 - 7;
```

assigns the value of the expression `4 * 3 - 7`, which evaluates to 5, to the variable `test`.

Note that the assignment statement, like all simple statements in C, ends in a semicolon `;`. This is referred to as the statement terminator.

Note also that whereas in some programming languages data items may be assumed to have an initial value of zero, when a variable is declared in C, it will have an unpredictable initial value, unless a specific value is used either to initialise it during declaration or to assign to in the code section. Unless either of these is made, the data item will have whatever value was last stored in the area of memory that the program has now allotted to it.

If a data item of any type is to have an initial value, be it zero or anything else, this value must be explicitly assigned at the start of the code block, or initialised within the declaration statement:

```
main()
{
     int count = 0, age = 18;
     char answer = 'y';
     float vat = 0.15F;
}
```

5 Precedence

The order in which operators act affects the values of the expressions in which they occur.

Consider the following expression: `20 - 10 % 7`. If the subtraction operator were to act first, the value of the expression would be `3`: `20` minus `10` gives `10`, and the remainder when `10` is divided by `7` is `3`.

However, if the modulus operator acts first, the expression evaluates to `17`: the remainder when `10` is divided by `7` is `3`, and `20` minus `3` is `17`.

Such ambiguity cannot be allowed in a computer programming language. That is why the order in which operators take effect is strictly defined in C according to a hierarchy known as precedence. A table of operator precedence is given in Appendix A of the course notebook.

In fact, the modulus operator has a higher precedence than the subtraction operator, so the expression `20 - 10 % 7` actually does evaluate to `17`.

Note that brackets, which have highest priority, can be used to change the value of expressions. If the expression were to be rewritten `(20 - 10) % 7`, the section in the brackets would be evaluated first, and the overall expression would now have the value `3`.

It is often helpful to use brackets when writing expressions to avoid errors of precedence and to make the order of evaluation more explicit and hence more easily legible.

6 Increment and Decrement Operators: ++ and --

The increment operator ++ increases the value of a variable by one. The decrement operator `--` decreases the value of a variable by one.

Statements such as

```
count = count + 1;
time = time - 1;
```

may thus be written more succinctly as:

```
count++;
time --;
```

The increment and decrement operators may either be placed before the variable in what is known as prefix mode, or after the variable in what is known as postfix mode. In prefix mode, the operators cause the variable to be incremented or decremented by one before the next operator is applied to that variable. In postfix mode, the increment and decrement operators cause the variable to incremented or decremented by one after the next operator is applied to that variable.

The choice of mode for the increment and decrement operators may affect the value of the expressions in which they are used. Consider the following example:

```
main()
{
     int  prefix, postfix,
          test1=10, test2=10,
          check1, check2;

     prefix = --test1;
     check1 = test1;
     postfix = test2--;
     check2 = test2;

     printf("\nThe value of prefix is %d.", prefix);
     printf("\nThe value of check1 is %d.", check1);
     printf("\nThe value of postfix is %d.", postfix);
     printf("\nThe value of check2 is %d.", check2);
}
```

When this program is compiled and run, it will print out:

```
The value of prefix is 9.
The value of check1 is 9.
The value of postfix is 10.
The value of check2 is 9.
```

Although the variables `test1` and `test2` both have the initial value of `10`, the value of `test1` is decremented before its value is assigned to the variable prefix, whereas the

value of `test2` is only decremented after its initial value has been assigned to the variable postfix. The final value of `check2` shows that the value of `test2` really has been decremented by one.

Always check that increment and decrement operators are written in the appropriate mode for the context.

7 Assignment Operators

As well as the = operator, C has additional arithmetic assignment operators: `+=`, `-=`, `*=`, `/=` and `%=`, together with the five bit-assignment operators, which are covered in the Further Data Types chapter.

The addition assignment operator += adds the current value of the variable on its left to the value of the expression its right and then assigns this new value to the variable on its left. Thus, a statement such as

```
minutes = minutes + 60;
```

can be written more succinctly as:

```
minutes += 60;
```

The other four arithmetic assignment operators work in similar fashion:

```
minutes -= 60;      /* equivalent to minutes = minutes - 60;*/
minutes *= 60;      /* equivalent to minutes = minutes * 60;*/
minutes /= 60;      /* equivalent to minutes = minutes/60;*/
minutes %= 60;      /* equivalent to minutes = minutes % 60;*/
```

8 Mixing and Converting Data Types

8.1 Mixing Data Types

In most applications, it is important that data types are not mixed, and that, for instance, integer values are not assigned to floating-point variables and vice versa. There are times, however, when such mixing is advantageous, and C is in fact very flexible in allowing this to take place.

8.2. Type Conversion in Assignments

Type conversion occurs automatically across assignment statements. When, for example, a floating-point value is assigned to an integer variable, the value is converted to an integer.

However, because different amounts of memory are apportioned to different types of data, there may not be sufficient memory allocated to the new type of data to take all the information contained in the old. There is no problem when a narrow object, that is to say, an item of a type allocated a small amount of memory, is assigned to a wide object of a type allocated a larger amount of memory. When it is the other way round, and a wide object is allocated to a narrow object, some of the information in the original object will be lost. The nature and extent of the information lost is not defined by the language, however, but is entirely dependent on the system being used.

For this reason, great care must always be taken when converting data from one type to another.

8.3 Arithmetic with Mixed Types

When an operator acts on data of different types, the value of the narrower type is always promoted to the wider type before the operation proceeds. Consider the following example:

```
main()
{
     short sh;
     char ch = 'G';
     float fl = 3.5E2F;

     sh = fl + ch;
}
```

Assuming that ASCII is being used, the value of the expression in the final line, `fl + ch`, may be calculated as follows: the numerical value of the variable `ch`, namely the ASCII code for `G`, which is `71`, will be converted to a floating-point value, in this case `71.0`, before being added to the value of `fl`, which is `3.5E2`, or `350.0` in decimal notation. This gives a final value of `421.0`.

Note, however, that this floating-point value is assigned to an integer variable of type `short int`. The value that is actually assigned to `sh` will depend entirely on the system being used.

8.4 Explicit Type Conversions using the cast operator: ( )

A cast forces the conversion of an expression to the type specified in the cast, just as if the value of the expression had been assigned to a variable of that type. A cast is written by placing the name of the type to be converted to in brackets in front of the expression, thus:

```
(type) expression
```

An example would be:

```
(int) (2 * 3.142 * radius)
```

Even if `radius` in this example is an integer variable, the expression `(2 * 3.142 * radius)` includes a floating-point number, so would evaluate to a floating-point value, which the integer cast would then explicitly convert to an integer value.

The process of making an explicit type conversion by means of a cast is known as coercion. As will be seen in subsequent chapters, specific functions require arguments and return values of specific types, and coercion can be useful for ensuring that they are always passed the appropriate types of data, and that the values they return are converted to the type required.

9 Relational Operators

C's relational operators assess the truth of a relational statement about two expressions (i.e. they are binary operators), and return the value `1` if the statement is true or `0` if it is false.

The expression on the left is greater than the expression on the right.

```
test = x > y * z;
```

The value of the variable `test` will be `1` if the value of the variable `x` is greater than the value of `y * z`, or `0` if the value of `x` is less than or equal to `y * z`.

The six relational operators available are:

`>`	is greater than
`<`	is less than
`==`	is equal to
`!=`	is not equal to
`>=`	is greater than or equal to
`<=`	is less than or equal to

Always remember to distinguish between the relational operator == and the assignment operator =. They are easy to confuse, and, although obvious when spotted, errors involving the use of the relational operator instead of the assignment operator and vice versa are notoriously difficult to trap.

Remember too that the operator `!=` ("is not equal to") returns a `1` when it is not true that the operands on either side are of equal value.

10 Truth and Falsity

When testing for truth or falsity in C, zero is defined as false and non-zero as true.

The unary `NOT` operator `!` returns `1` if it is true that its operand is false, and zero if its operand is not false, i.e. true.

In the following example, the variable `test` will have the value `1`, since, by definition, not zero is true.

```
test = !0;
```

11 Logical Operators

In addition to the unary NOT operator, C has two binary logical operators:

`&&` logical AND

`||` logical OR

The logical AND operator `&&` will return a `1` for true if both the expressions to the left and right of the operator are true, i.e. non-zero, and yield a `0` if either expression is false, i.e. zero. The logical OR operator `||` will yield a `1` if either the expression to the left of the operator or that to its right are true, i.e. non-zero, and a `0` only if both expressions are false.

```
main()
{
     int test1, test2, a = 3, b = 1, c =7, d = 2;

     test1 = (a > b) && (c >= d);
     test2 = (d < b) || (a < c);
}
```

In the above example, the variable `test1` will be assigned the value `1`, because both `(a > b)` and `(c >= d)` are true. The variable `test2` will also be assigned the value `1`, because although the expression `(d < b)` is false, `(a < c)` is true, and only one expression needs to be true for the logical OR to return a true value.