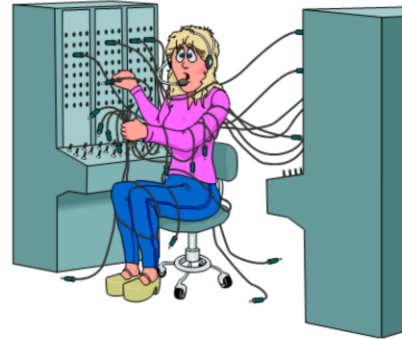




Expressions

- **Objective**
 - To process scalar data
- **Contents**
 - Arithmetic expressions
 - Assignment expressions
 - Compound assignment expressions
 - Increment expressions
 - Decrement expressions
 - Type conversion
 - Bit operators
 - Evaluation Points - Introduction
- **Summary**
- **Practical**
 - A small program which illustrates some of C's operators



This is the second of the two chapters covering the basic language syntax. The objective of this chapter is to cover code statements. This entails discussion on operators, expression evaluation and type conversion during data manipulation. Boolean expressions and their relevant operators are discussed. There is no boolean type, in ANSI C, but there is in C99 (covered in the Advanced course).


QACPROG

Arithmetic Expressions

- **Binary**
 - +** **Addition**
 - **Subtraction**
 - *** **Multiplication**
 - /** **Division**
 - %** **Modulus**
- **Unary**
 - **Minus**
 - +** **Plus**

`lhs + rhs`
`lhs - rhs`
`lhs * rhs`
`lhs / rhs`
`lhs % rhs`

`+ operand`
`- operand`



There is nothing exciting here! However, here are a few minor points.

The division operator takes its lead from the data. If both data items are integral (`ints` or `chars`), then integer division is performed, else floating-point division is performed.

The modulus operator can only be used with integral operands, and it generates a remainder on integer division.

Expressions Have a Value

- **Expressions allow a *value* to be formed from...**
 - Constants, which are literal *values*
 - Variables, which are names of *values*
 - Operators, which create new *values*
 - Subexpressions, to form more complex expressions

```
...  
int count = 3;  
int j = 4;  
...  
printf("%d\n", 5);  
printf("%d\n", count);  
printf("%d\n", count*j - 7);  
...
```

Level 1 expression:
a constant;
evaluates to 5

Level 2 expression:
a variable;
evaluates to value in count

Level 3 expression:
involves operators;
evaluated according to
precedence rules

Again, C does not perform anything really surprising. An expression is defined as a combination of variables, constants, operators and subexpressions.

An expression always has a value with associated type, although this type could be void. The examples shown above illustrate three levels of expression complexity: the constant, the variable, and finally, the full works! All three expressions are of type `int`. It is vital to know the type of an expression and how to use it properly in the subsequent code. Automatic and programmer-controlled type conversion may take place within expression evaluation. This is covered later in the chapter.

There are two types of expression: those used as statements and those used as subexpressions. The former are made into statements simply by placing the `;` terminator at the end. The latter are used to control looping and decision-making constructs, as arguments to functions and as initialisers for data declaration.

Assignment Expressions

- **Assignments assign!**
 - The right hand side is evaluated and the value is stored in the variable
- **In C, assignment is an expression**
 - This means the whole assignment itself has a value...
 - The value of the expression

assignment
operator =

statement
terminator;

variable = expression;

```
...
int i;
int count = -3;
int j = 3;
...
i = 5;
i = count;
i = count * j - 7;
...
printf("%d\n", i = 5);
...
```

Assignment in C is an expression with a side effect on the left hand side operand and a result of that same operand. The value in the left hand is overwritten with the value from the right hand side — quite a side effect! This differs from most other languages where assignment is a statement that has no result.

An *lvalue* is a variable or object that has an address and may be used as the target of assignment. For instance, in the slide above `count` is an *lvalue*, but the literal value 5 is not. An *lvalue* is so called because it is the *value* appearing on the *left* hand side of the assignment.

Operator associativity defines how the operands of binary operators will group: this will be either to the left or to the right. In the case of the assignment operators it is to the right. You can imagine the idea of associativity being related to direction of grouping, i.e. if you were to put parentheses around the expressions the associativity would tell you the default way of grouping them. In the multiple assignment below

```
x = y = 0;
```

`x` and `y` both receive the value 0, is an example illustrating assignment's right associativity. The implicit grouping is

```
x = (y = 0);
```

The assignment to `y` is an expression. It has a value: zero. This value is then assigned to `x`. The assignment to `x` is an expression. It has a value: zero. This value is discarded. The result of assignment is an *rvalue*. An *rvalue* is so called because it is a value that can only appear on the *right* hand side of an assignment. This means that the following will not compile in C (it will in C++, but that is another story...):

```
(x = y) = 0;
```

Compound Assignment

- A convenient short hand for some assignments

```
int stein = 1;
int pint = 1;
```

```
stein = stein + pint;
```



```
stein += pint;
```

- Use any arithmetic operator

```
lhs = lhs + rhs;
lhs = lhs - rhs;
lhs = lhs * rhs;
lhs = lhs / rhs;
lhs = lhs % rhs;
```



```
lhs += rhs;
lhs -= rhs;
lhs *= rhs;
lhs /= rhs;
lhs %= rhs;
```

*Compound assignment is an assignment!
It has a result, which is usually ignored.*



The expression `a += b` can be read as "a is incremented by the value of b". Therefore, in the code above, the variable `stein`, initially at 1 is incremented by the value of `pint`, which is 1. After the assignment the value of `stein` becomes 2 and the value of `pint` remains 1.

Compound assignment operators are more succinct than the long hand approach found in other languages. C programmers tend to write expressions like this:

```
total += subtotal;
geometric *= progression;
```

Rather than like this:

```
total = total + subtotal;
geometric = geometric * progression;
```

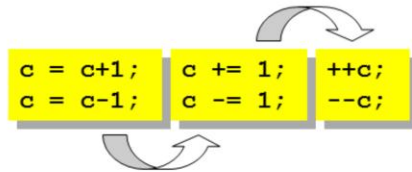
Often the resulting machine code generated by the compiler will be the same, but once you are familiar with the syntax the compound form proves easier to read and less error prone. In later chapters we will see that the expression on the left hand side can be more complex than a simple numeric variable. Duplicating complex expressions is a potential source of errors. For more complex data types the resulting long hand version can also be less efficient.

Here are the compound assignment operators for the arithmetic operators encountered so far:

+	+=	<i>addition</i>
-	-=	<i>subtraction</i>
*	*=	<i>multiplication</i>
/	/=	<i>division</i>
%	%=	<i>modulo</i>

Increment and Decrement

- **Have special operators in C**
 - Use ++ to increment a variable by 1
 - Use -- to decrement a variable by 1



- **May be used as *prefix* operators**

- Result is new value
after
operation is performed

```
result = ++pre_inc;
```

```
pre_inc += 1;  
result = pre_inc;
```

- **May be used as *postfix* operators**

- Result is old value
before
operation is performed

```
result = post_inc++;
```

```
result = post_inc;  
post_inc += 1;
```

The most common value by which a variable is increased or decreased is 1. C already provides a fairly brief way of doing in this in the form of compound assignment operators:

```
value += 1; // increase value by 1  
value -= 1; // decrease value by 1
```

The idea of "the next one" or "the previous one" is so common that for convenience increment and decrement operators exist in the language: ++ and --. When the increment or decrement expression is the only part of a statement there is effectively no difference between using the prefix or postfix forms of the operator:

```
int value = 0;  
printf("%d\n", value); // prints 0  
++value;  
printf("%d\n", value); // prints 1  
value++;  
printf("%d\n", value); // prints 2
```

When used as part of a more complex expression the position of the operator becomes important. Like the assignment operators, the increment and decrement operators return a value. The prefix form increments the value first before returning the result, whereas the postfix uses the value before the increment as its return:

```
int value = 0;  
printf("%d\n", ++value); // prints 1  
printf("%d\n", value--); // prints 1  
printf("%d\n", value);   // prints 0
```

Exercise: What is Displayed?

1 `int index = 0;`
`printf("%d\n", index++);`
`printf("%d\n", --index);`

2 `int first, last = 60;`
`printf("%d\n", first = last % 11);`
`printf("%d\n", first /= 2);`



1. In the first case `index` is incremented after its value is used, so 0 is displayed. In the second case `index` is decremented before it is used, so it goes from 1 back down to 0. So 0 is then displayed.
2. In the first case `last % 11` is assigned to `first`, which becomes 5 and this value is then displayed. In the second case `first` receives half of its value which, as this is integer division, is 2. So 2 is then displayed.

These examples are deliberately complicated to test your understanding.

Question 2 is much more understandable as:

```
int last = 60;
int first = last % 11;
printf("%d\n", first);
```


Conversions

- **Implicit conversions occur between all arithmetic types!**

```
int    lower_a = 'a';
double level   = 0;
```

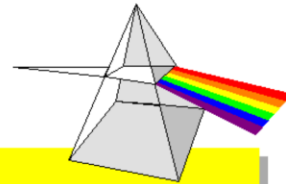
Widening conversions
are always safe

```
int     vat    = 17.5;
float   pi     = 3.14159265359;
unsigned wrap   = -1;
```

Narrowing conversions
can result in loss of
precision and range

- **Explicit conversions are called casts**
 - **(type)** is the conversion operator syntax

```
int         slice = (int)pi;
unsigned long pig  = (unsigned long) (pi * 9.98);
```



A narrowing conversion occurs when coercing the value from a type into a type that has a smaller range and precision. Such conversions are not always well defined and are potentially "lossy". For instance, assigning a double to an int may result in truncation of any fractional part and possible loss of range.

Widening is always safe and describes the conversion from one type to another with a larger range and precision. Therefore initialising a double from a float or int, or an int from a short, is OK.

It is safe to initialise any arithmetic type from literals as the compiler checks these and may warn about suspicious conversions. For instance, when initialising a short from an int literal the compiler can check at compile time whether to warn against a suspicious conversion. The same is true of initialising float from double literals, which is why the F or f suffix is not often used.

Within an expression the terms are implicitly converted to the widest type, if necessary. For instance if an int is added to a double, the int value is first converted implicitly to a double before the addition is performed:

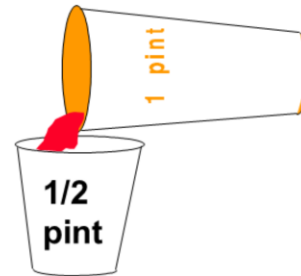
```
const int i = 3;
printf("%d", i / 2);           // prints 1
printf("%f", i / 2.0);        // prints 1.5
printf("%f", (double)i / 2);  // prints 1.5
```

The term *type coercion* is sometimes used to describe casting. The best option of all is not to need them; view casts with suspicion.

Assignment Revisited

- What if the operands do not have the same type?
 - Implicit conversions will occur!
 - Narrowing conversions are machine dependent

```
...
int    i = 42;
char   c = 'Q';
float  f = 3455.3F;
double d = 193443.34233;
...
i = c;
c = i;
f = d;
...
```



What conversions occur in this program?

```
int i = 42;
char c = 'Q';
float f = 3455.3F;
double d = 193443.34233;
```

There are no conversions in any of these statements. They all initialise a variable with a literal of the equivalent type.

```
i = c;
```

In this statement an implicit widening conversion occurs. The `char c` is widened to an `int` which is then assigned to the `int i`.

```
c = i;
```

In this statement an implicit narrowing conversion occurs. The `int i` is narrowed to a `char` which is then assigned to the `char c`. This will probably cause the compiler to generate a warning.

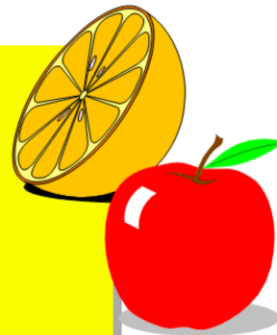
```
f = d;
```

In this statement an implicit narrowing conversion occurs. The `double d` is narrowed to a `float` which is then assigned to the `float f`. This too will probably cause the compiler to generate a warning.

Arithmetic Revisited

- **What if the operands do not have the same type?**
 - The narrower type is converted to the wider type
 - Then the operator proceeds

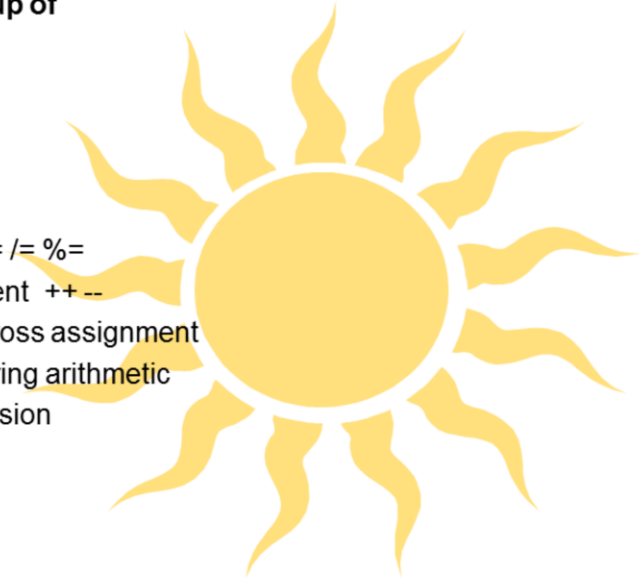
```
...
short  rate = 15;
long   number = 40L;
float  factor = 4.5F;
double value = 10.2;
...
number = number / rate;
factor  = factor * rate;
factor  = value + rate * factor;
...
```



*What conversions occur
in this program?*

Summary

- **A C expression is made up of**
 - Literals (e.g. 42)
 - Variables (e.g. result)
 - Operators (e.g. +)
- **Operators**
 - Arithmetic + - * / %
 - Assignment = += -= *= /= %=
 - Increment and decrement ++ --
 - Conversion: implicit across assignment
 - Conversion: implicit during arithmetic
 - Cast: an explicit conversion



Common Pitfalls

Operators

- Writing `=op` instead of `op=` for compound assignment
- Forgetting that integer division produces an integer
- Attempting to divide by zero
- Attempting to concatenate two string literals using `+`

```
value *= rate;
...
value =+ rate;
```



```
no_half = 5 / 2;
```



```
danger = x / y ;
```



```
"foo" + "bar"
"foo" "bar"
```



Precision

- Losing it!



```
int    vat = 17.5;
unsigned wrap = -1;
```



The syntax of a compound assignment is `=op` and not `op=`. For example compound multiplication is `*=` and not `*=`. There is a nasty gotcha here because writing `=+` instead of `+=` will still compile. The reason is that:

```
value =+ rate;
```

is parsed by the compiler as:

```
value = +rate;
```

In other, words, a simple assignment with the expression on the right hand side being formed from the unary plus operator.

Integer division will produce an integer result. In C, 5 divided by 2 is 2 and not 2.5.

The C standard states that any attempt to divide by zero causes undefined behaviour. In other words all bets are off! Don't do it.

If string literals are being concatenated they need only be adjacent to each other:

```
"foo" "bar"
```

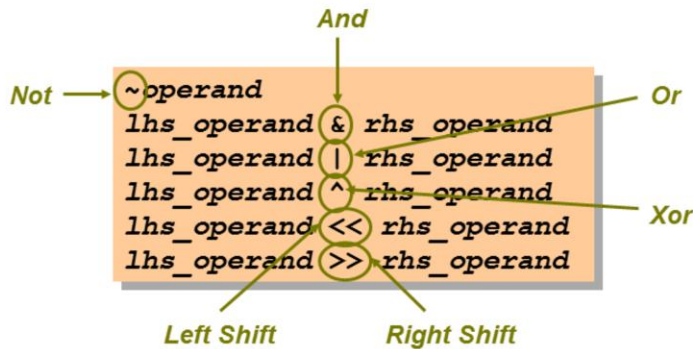
The compiler will sometimes warn against lossy conversions:

```
int vat = 17.5;
```

Floating point numbers suffer a certain amount of rounding and loss of precision in calculations. This is often not enough to be significant, except when attempting to compare two values for equality (see *Common Pitfalls* in the *Making Decisions* chapter).

Bitwise Operators

- Integer bit representation can be manipulated directly



- Compound assignment operators exist for each of the binary operators shown

Bit operators are covered in more detail later

C also supports bit manipulation operators:

<code>~</code>		<i>bitwise not (one's complement)</i>
<code>&</code>	<code>&=</code>	<i>bitwise and</i>
<code> </code>	<code> =</code>	<i>bitwise or (inclusive or)</i>
<code>^</code>	<code>^=</code>	<i>bitwise xor (exclusive or)</i>
<code><<</code>	<code><<=</code>	<i>bitshift left</i>
<code>>></code>	<code>>>=</code>	<i>bitshift right</i>

The `~` operator flips all the bits in a representation:

```
unsigned char low  = 0x0f; /* 00001111 */
unsigned char high = ~low; /* 11110000 */
```

The *and* and *or* operators apply to each bit in the representation, so that `a & b` will result in an integer whose representation of each corresponding bit in `a` and `b` *anded* together. The `&` operator is the bitwise *and* operator, `|` is the bitwise *inclusive or* ("one or other or both"), and `^` is the bitwise *exclusive or* ("one or other but not both").

The `<<` and `>>` operators may be used for left and right shifting the bits in an integer by a specified number of places:

```
unsigned char low    = 0xf; /* 00001111 */
unsigned char left   = low << 2; /* 00111100 */
unsigned char right  = low >> 2; /* 00000011 */
```

For signed integers it is platform specific whether a right shift is sign preserving (*arithmetic shift*) or zero filling (*logical shift*). For this reason unsigned integers are normally used when manipulating bits.

Again, bit manipulation is discussed in more detail in the More on Data Types chapter.