Operators and Assignments

C has a wide range of operators that make simple math easy to handle. The list of operators grouped into precedence levels is as follows:

Primary expressions

An identifier (or variable name) is a primary expression.

A constant is a primary expression. Its type depends on its form and value.

A string literal is a primary expression.

A parenthesized expression is a primary expression. Its type and value are those of the non-parenthesized expression.

Unary expressions

The following expressions are all unary expressions:

The increment or decrement operators followed by a unary expression is a unary expression. The value of the expression is the value of the unary expression *after* the increment or decrement. These operators only work on integers and pointers.

The following operators followed by a cast expression are unary expressions:

```
Operator     Meaning
========     =======
   &         Address-of; value is the location of the operand
   *         Contents-of; value is what is stored at the location
   -         Negation
   +         Value-of operator
   !         Logical negation ( (!E) is equivalent to (0==E) )
   ~         Bit-wise complement
   ++        Increment the value by one
   --        Decrement the value by one
   sizeof()  Find the size of
```

The keyword `sizeof` followed by a unary expression is a unary expression. The value is the size of the type of the expression in bytes. The expression is not evaluated.

The keyword `sizeof` followed by a parenthesized type name is a unary expression. The value is the size of the type in bytes.

Cast operators

Writing data types like `(int)` is called a cast expression.

A cast expression is a unary expression.

The type name has the effect of forcing the cast expression into the type specified by the type name in parentheses. For arithmetic types, this either does not change the value of the expression, or truncates the value of the expression if the expression is an integer and the new type is smaller than the previous type.

An example of casting a float as an int:

```
float pi = 3.141592;
int truncated_pi = (int)pi; // truncated_pi == 3
```

An example of casting a char as an int:

```
char my_char = 'A';
int my_int = (int)my_char; // On machines which use ASCII as the character set, my_int
== 65
```

Multiplicative and additive operators

In C, simple math is very easy to handle. The following operators exist: **+** (addition), **-** (subtraction), ***** (multiplication), / (division), and **%** (modulus); You likely know all of them from your math classes - except, perhaps, modulus. It returns the **remainder** of a division (e.g. 5 % 2 = 1). (Modulus is not defined for floating-point numbers, but the *math.h* library has an *fmod* function.)

Care must be taken with the modulus, because it's not the equivalent of the mathematical modulus: (-5) % 2 is not 1, but -1. Division of integers will return an integer, and the division of a negative integer by a positive integer will round towards zero instead of rounding down (e.g. (-5) / 3 = -1 instead of -2). However, it is always true that for all integer a and nonzero integer b, `((a / b) * b) + (a % b) == a`.

There is no inline operator to do exponentiation (e.g. 5 ^ 2 is **not** 25 [it is 7; **^** is the exclusive-or operator], and 5 ** 2 is an error), but there is a power function *pow()* in the math.h header file.

The mathematical order of operations does apply. For example (2 + 3) * 2 = 10 while 2 + 3 * 2 = 8. Multiplicative operators have precedence over additive operators.

```
#include <stdio.h>
int main()
{
    int i = 0, j = 0;
    /* while i is less than 5 AND j is less than 5, loop */
    while( (i < 5) && (j < 5) )
    {
        /* postfix increment, i++
         *     the value of i is read and then incremented
         */
        printf("i: %d\t", i++);
        /*
         * prefix increment, ++j
         *     the value of j is incremented and then read
         */
        printf("j: %d\n", ++j);
    }
    printf("At the end they have both equal values:\ni: %d\tj: %d\n", i, j);
    return 0;
```

```
}
```

will display the following:

```
i: 0    j: 1
i: 1    j: 2
i: 2    j: 3
i: 3    j: 4
i: 4    j: 5
At the end they have both equal values:
i: 5    j: 5
```

The shift operators (which may be used to rotate bits)

Shift functions are often used in low-level I/O hardware interfacing. Shift and rotate functions are heavily used in cryptography and software floating point emulation. Other than that, shifts can be used in place of division or multiplication by a power of two.

shift left `<<`

The `<<` operator shifts the binary representation to the left, dropping the most significant bits and appending it with zero bits. The result is equivalent to multiplying the integer by a power of two.

unsigned shift right

The unsigned shift right operator, also sometimes called the logical right shift operator. It shifts the binary representation to the right, dropping the least significant bits and prepending it with zeros. The `>>` operator is equivalent to division by a power of two for unsigned integers.

signed shift right

The signed shift right operator, also sometimes called the arithmetic right shift operator. It shifts the binary representation to the right, dropping the least significant bit, but prepending it with copies of the original sign bit. The `>>` operator is not equivalent to division for signed integers.

In C, the behavior of the `>>` operator depends on the data type it acts on. Therefore, a signed and an unsigned right shift looks exactly the same, but produces a different result in some cases.

Relational and equality operators

The relational binary operators `<` (less than), `>` (greater than), `<=` (less than or equal), and `>=` (greater than or equal) operators return a value of 1 if the result of the operation is true, 0 if false.

The equality binary operators `==` (equals) and `!=` (not equals) operators are similar to the relational operators except that their precedence is lower.

Bitwise operators[edit]

The bitwise operators are `&` (and), `^` (exclusive or) and `|` (inclusive or). The `&` operator has higher precedence than `^`, which has higher precedence than `|`.

The values being operated upon must be integral; the result is integral.

One use for the bitwise operators is to emulate bit flags. These flags can be set with OR, tested with AND, flipped with XOR, and cleared with AND NOT. For example:

```
/* This code is a sample for bitwise operations.  */
#define BITFLAG1    (1)
#define BITFLAG2    (2)
#define BITFLAG3    (4) /* They are powers of 2 */

unsigned bitbucket = 0U;    /* Clear all */
bitbucket |= BITFLAG1;      /* Set bit flag 1 */
bitbucket &= ~BITFLAG2;     /* Clear bit flag 2 */
bitbucket ^= BITFLAG3;      /* Flip the state of bit flag 3 from off to on or
                               vice versa */
if (bitbucket & BITFLAG3) {
  /* bit flag 3 is set */
} else {
  /* bit flag 3 is not set */
}
```

Logical operators

The logical operators are `&&` (and), and `||` (or). Both of these operators produce 1 if the relationship is true and 0 for false. Both of these operators short-circuit; if the result of the expression can be determined from the first operand, the second is ignored. The `&&` operator has higher precedence than the `||` operator.

`&&` is used to evaluate expressions left to right, and returns a 1 if *both* statements are true, 0 if either of them are false. If the first expression is false, the second is not evaluated.

```
int x = 7;
int y = 5;
if(x == 7 && y == 5) {
    ...
}
```

Here, the `&&` operator checks the left-most expression, then the expression to its right. If there were more than two expressions chained (e.g. `x && y && z`), the operator would check `x` first, then y (if `x` is nonzero), then continue rightwards to z if neither x or y is zero. Since both statements return true, the `&&` operator returns true, and the code block is executed.

```
if(x == 5 && y == 5) {
    ...
}
```

The && operator checks in the same way as before, and finds that the first expression is false. The && operator stops evaluating as soon as it finds a statement to be false, and returns a false.

`||` is used to evaluate expressions left to right, and returns a 1 if *either* of the expressions are true, 0 if both are false. If the first expression is true, the second expression is not evaluated.

```
    /* Use the same variables as before. */
    if(x == 2 || y == 5) { // the || statement checks both expressions, finds that the
latter is true, and returns true
        ...
    }
```

The `||` operator here checks the left-most expression, finds it false, but continues to evaluate the next expression. It finds that the next expression returns true, stops, and returns a 1. Much how the `&&` operator ceases when it finds an expression that returns false, the `||` operator ceases when it finds an expression that returns true.

It is worth noting that C does not have Boolean values (true and false) commonly found in other languages. It instead interprets a 0 as false, and any nonzero value as true.

Conditional operators

The ternary `?:` operator is the conditional operator. The expression `(x ? y : z)` has the value of `y` if `x` is nonzero, `z` otherwise.

Example:

```
int x = 0;
int y;
y = (x ? 10 : 6); /* The parentheses are technically not necessary as assignment
                     has a lower precedence than the conditional operator, but
                     it's there for clarity.  */
```

The expression `x` evaluates to 0. The ternary operator then looks for the "if-false" value, which in this case, is 6. It returns that, so `y` is equal to six. Had `x` been a non-zero, then the expression would have returned a 10.

Assignment operators

The assignment operators are `=`, `*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `&=`, `^=`, and `|=`. The `=` operator stores the value of the right operand into the location determined by the left operand, which must be an lvalue (a value that has an address, and therefore can be assigned to).

For the others, `x op= y` is shorthand for `x = x op (y)`. Hence, the following expressions are the same:

```
    1. x += y     -     x = x+y
    2. x -= y     -     x = x-y
    3. x *= y     -     x = x*y
    4. x /= y     -     x = x/y
    5. x %= y     -     x = x%y
```

The value of the assignment expression is the value of the left operand after the assignment. Thus, assignments can be chained; e.g. the expression `a = b = c = 0;` would assign the value zero to all three variables.

Comma operator

The operator with the least precedence is the comma operator. The value of the expression `x, y` will evaluate both `x` and `y`, but provides the value of `y`.

This operator is useful for including multiple actions in one statement (e.g. within a for loop conditional).

Here is a small example of the comma operator:

```
int i, x;      /* Declares two ints, i and x, in one declaration.
                  Technically, this is not the comma operator. */

/* this loop initializes x and i to 0, then runs the loop */
for (x = 0, i = 0; i <= 6; i++) {
    printf("x = %d, and i = %d\n", x, i);
}
```