A Deeper Look at Expressions in C

A lot of C code consists of *expressions*. Expressions are everywhere, really. For example, consider an assignment

```
i = 7 + j;
```

The sum on the right-hand side (`7 + j`) is an expression. That is clear. The expression is defined by the addition operator (`+`) and its two operands, and the value of the expression is the value of `j` plus 7. All of this is pretty obvious. However, `j` could be considered an expression, and even `7`, and `i`. And the assignment as a whole is also an expression.

An assignment expression is defined by the assignment operator (`=`) and its two operands (or arguments): a left-hand-side expression that determines the object whose value will be assigned, and a right-hand-side expression that determines the value that will be assigned to the left-hand-side object. And the whole assignment is itself an expression whose value (or "result") is the value being assigned, meaning the result of the right-hand-side expression.

We now take a closer look at expressions in order to better understand the semantics of the evaluation of expressions, which is a fundamental part of the semantics of the language.

1 Objects, L-values, and R-values

An *object* is a named region of storage, in memory. An object can therefore store a value, which can be read and written. Examples are a variable of a basic type such as this one:

```
int i;
```

This defines an `int` object corresponding to variable `i`. Notice that variable `i` has *static* storage, so this object exists throughout the execution of the program.

An *l-value* is an *expression* referring to an object. An obvious example of an lvalue expression is an identifier:

```
i = 7;
```

Here is a slightly more complex example:

```
char s[100];

s[i] = 'a';
```

Here the expression `s[i]` is an l-value. That is, it is an expression that refers to an object.

And here's another one, just a bit more complicated—but you got the idea, right?

```
struct S {
    int x;
    char s[10];
};

struct S * fun2(int);

(fun2(i + 2) + 5)->s[i] = '?';
```

The left-hand side of an assignment must be an l-value, as in all the examples above. An l-value can also be used on the right-hand side of an assignment, as below:

```
s[i] = i;
```

However, the right hand side of an assignment may also denote a value that does not correspond to an object. The most obvious example is a *literal value*:

```
i = 7;
```

But many other expressions evaluate to a "value without a corresponding object". These are called *r-values*. Here's an example:

```
i = (fun2(i + 2) + 5)->s[i] * i;
```

So, both l-values and r-values are *expressions*, but the main difference is that an l-value refers to an *object*, which also has a value, while an r-value refers to a value *without a corresponding object*.

Therefore, another thing you can do with an l-value that you can not do with an r-value is to apply the address-of operator (`&`).

```
/* CORRECT: the & operator applies to l-values */
int * p = &i;
char * c = &((fun2(i + 2) + 5)->s[i]);
/* INCORRECT: these are not l-values */
int * p_bad = &(i + 2);
char * c_bad = &(i > 0 ? 'x' : 'y');
```

In conclusion, each operator in the C language defines constraints on its operands. As we have seen, the assignment operator (`=`) requires that its first (left-hand) operator is an l-value; the address-of operator (`&`), which is a unary operator, requires that its operand be an l-value, and the same is true for the prefix and postfix increment and decrement operators (`++`, `--`); instead, the addition operator (binary `+`) only requires that its operands be r-values.

2 Operator precedence and associativity

See http://en.cppreference.com/w/c/language/operator_precedence

Operator precedence determines the semantics (i.e., the exact interpretation) of an expression. For example:

```
int i = 5;
int j = 10;

i = i + j * 2;

printf("i = %d\n", i);
```

Here the result is 25 (not 30), since the multiplication operator (`*`) has a higher precedence than the addition operator (`+`). Of course, you can use parenthesized expressions to force a certain evaluation precedence. So, the output of the code below is 30, not 25:

```
int i = 5;
int j = 10;

i = (i + j) * 2;

printf("i = %d\n", i);
```

When operators have the same precedence, the semantics is defined by the associativity. For example, the `+` and `-` operators, when used as binary operators (subtraction and addition), have the same precedence, and are left-associative:

```
int i = 10;
int j = 2;
i = i - j + 4;
```

So this is equivalent to this:

```
i = ((i - j) + 4); /* == 12 */
```

as opposed to this:

```
i = (i - (j + 4)); /* == 6 */
```

Other operators associate from right to left. For example:

```
int i = 3;
int j = 5;

i = j = 7;
printf("i = %d, j = %d\n", i, j);
```

where the assignment expression is equivalent to this:

```
i = (j = 7);
```

Pop quiz 1: what happens here? And *why?*

```
int i = 3;
int j = 5;

(i = j) = 7;
printf("i = %d, j = %d\n", i, j);
```

Pop quiz 2: what happens here? And *why?*

```
int i = 3;
int j = 5;

i = -j--;

printf("i = %d, j = %d\n", i, j);
```

Pop quiz 3: what happens here? And *why?*

```
int i = 3;
int j = 5;

i = ---j;

printf("i = %d, j = %d\n", i, j);
```

3 Evaluation order

note: evaluation for an expression/subexpression = depends on computations of values and side effects

Notice that the semantics of an expression, as determined by the precedence and associativity of operators, has nothing to do with the *order of evaluation* of the operands (subexpressions). Consider for example the following code:

```
i = f() - g() + h();
```

side effects: function calls (), assignment operator = (storing r-value into l-value—> changing object in memory!)

What is the order of invocation of the three functions?

And what about this case:

| = sequence point

=
L-value evaluation | side effect
r-value evaluation |

there is sequencing between evaluation and side effect, but not in the computation of the result of the assignment

```
A[f()] = B[g()] =  C[h()];
```

What is the order of invocation of the three functions?

The answer is, *unspecified!*

**= operator:
computations are sequenced before the storing, but the side effect/s of the expressions which go into the assignment operator are not sequenced**

Notice that the *evaluation* of an expression (or subexpression) might include the computations of values, as well as the initiation of side-effects.

3.1 Sequence points

The evaluation order of expressions is sometimes unspecified. One reason for this is to give maximum flexibility to the compiler to perform aggressive code optimization.

For example, the order in which the arguments of a function call are evaluated is unspecified. This means that *correct code must never rely on a particular order.*

However, obviously, in many cases the order of evaluation is indeed fully specified and therefore unambiguous.

More generally, in some cases an evaluation *A* is *sequenced before* another evaluation *B*, in other cases it's the opposite, and yet in other cases it is *neither*, that is, neither A is squenced before B nor B is sequenced before A. In this latter case, we also say that *A* and *B* are *unsequenced* relative to each other.

Here is an example:

```
A[i] = A[i+1] * A[i+2];
```

Here the evaluations of the two operands of the multiplication operator are *unsequenced*. This means that the expressions `A[i+1]` and `A[i+2]` may be evaluated in any order. In fact, the same is true for the subexpressions that make up the array indexes. In fact, even the evaluation of the two operands of the assignment operator are unsequenced. This means that the determination of the destination object (l-value) and the determination of the value to be assigned are unsequenced. However, those two evaluation are *sequenced before* the side effect of storing the value of the right operand (r-value) into the object of the left operand (l-value). This means that the value of `i` used in evaluating `f(i)` and `g(i)` is the initial value of `i`, which is *later* replaced by the value of the expression `f(i)+g(i)`.

Here is another, related example:

```
i = f(i) + g(i);
```

i used in f and g is the i before it is assigned

as in the previous case, the evaluation of the operands of the addition operator are unsequenced. So, neither `f(i)` is *sequenced before* `g(i)`, nor `g(i)` is *sequenced before* `f(i)`. However, in this case, both evaluations include the invocation of a function, `f(i)` and `g(i)`, respectively, and since the executions of functions do not overlap, `f(i)` is sequenced either before or oafter `g(i)`, but we do not know which. In this case we say that the two evaluations (function executions) are *indeterminately sequenced*.

**easiest way to put a sequence point is by putting a semicolon ;
&& and || operators do the trick as well
Left hand side of && gets evaluated first, then if it's true, right hand side gets evaluated as well**

Here is yet another example:

```
if (i > 0 || f(i) == 1 || g(i) == 2) {
    /* ... */
}
```

Here the evaluation is completely specified. In fact, the evaluation of `i > 0` is sequenced before the evaluation of `f(i) == 1`, which is sequenced before `f(i) == 2`. Furthermore, since the evaluation of logical operators *always* uses a short-cut semantics, `f(i) == 1` is evaluated *only if* `i > 0` evaluates to *false*, and `g(i) == 2` is evaluated *only if* `f(i) == 1` also evaluates to *false*.

So, what determines the sequencing of evaluations? Many contstructs do that. In an abstract sense, there are points in the program (flow) that induce sequencing. We call them *sequence points*. So, everything before a sequence point is *sequenced before* anything after that sequence point.

Example: the end of a full expression statement. In simpler terms, a semicolon:

```
i = g(j) + h(j);
j = f(i);
```

Here `g(j)` and `h(j)` are *sequenced before* `f(i)`, although `g(j)` and `h(j)` are unsequenced. Also, the assignment to `i` in the first expression statement is *sequenced before* the evaluation of `f(i)`.

There is also a sequence point between the evaluation of function parameters (as well as the function designator), and the actual call. So, for example

```
j = f(g(j) + h(j));
```

here the evaluation of the parameter to function `f` is *sequenced before* the actual call of `f`, so the invocations of `g` and `h` are both *sequenced before* the invocation of `f`. However, the invocations of `g` and `h` are *indeterminately sequenced*.

Other syntactic structures introduce sequence points. The details are mostly intuitive.

So, what happens when two evaluations are *unsequenced*? There are good and bad cases. Good cases are perfectly fine expressions (indeed most non-trivial expressions) in which unsequenced expressions do not interfere with each other, and the semantics of the program is unambiguous anyway.

Bad cases are those in which unsequenced expressions have interrelated side-effects, which leads to ambiguities in the interpretation of the evaluation. For example:

```
j = i++ - i--;
```

i++ and i— both have side effects

What is the value of `j`? And what is the value of `i`?

expressions with 2 or more side effects on the same object, that are not properly sequenced, cause undefined behaviour

And again:

that's why the expression to the left creates undefined bahaviour, while
j = i++ - l++ does not, because
in the last expression, the side effects by the suffix increment and suffix decrement affect different objects
(i++ affects i but not l, while l++ affects l but not i)

```
int i = 0;
j = A[++i] + B[++i];
```

Is `j == A[1] + B[2]`? or perhaps `j == A[2] + B[1]`? Or `j == A[1] + B[1]`?

The answer: in all these cases the behavior is *undefined*. And it should be clear, because they are all ambiguous cases.

However, notice that there are other, perhaps more subtle cases in which the behavior is also undefined. For example:

```
int i = 0;
i = ++i;
a[i++] = i;
```

These two (latter) statements might look innoquous. In the first expression statement–you might think–the `i` gets its own increment, which changes the value of `i` to `i == 1`, and then in the following assignment statement, you assign `A[1] = 1` and then increment `i`. However, there two statements alre invoke *undefined behavior*.

This is the precise wording in the definition of the C language:

> If a side effect on a scalar object is unsequenced relative to either a different side effect on the same scalar object or a value computation using the value of the same scalar object, the behavior is undefined. If there are multiple allowable orderings of the subexpressions of an expression, the behavior is undefined if such an unsequenced side effect occurs in any of the orderings.

So, the key point is that a side effect on an object should not be unsequenced relative to another side effect on the same object, or relative to a *use* of the value of the same object.

Pop quiz: do the following two statements invoke undefined behavior? Why?

```
A[i] = i;
i = i + 1;
```

No,different objects

Pop quiz: does the following statement invoke undefined behavior? Why?

```
A[i] = i++;
```

yes

Pop quiz: does the following statement invoke undefined behavior? Why?

```
A[++i] = A[++j] + 1;
```

No

Pop quiz: does the following statement invoke undefined behavior? Why?

```
i = j++ + k++;
```

Pop quiz: does the following statement invoke undefined behavior? Why?

```
A[i] = A[j]++ + A[k]++;
```

Pop quiz: does the following statement invoke undefined behavior? Why?

```
int *p, *q;
/* ... */
*p = *q++;
```

Pop quiz: does the following statement invoke undefined behavior? Why?

```
int *p, *q;
/* ... */
*p = (*q)++;
```

Author: Antonio Carzaniga
Created: 2019-05-21 Tue 23:13
Emacs 25.2.2 (Org mode 8.2.10)
Validate