Contents

1 Expressions and Evaluation Semantics

1.1 Expressions everywhere

A lot of C code consists of *expressions*. For example, an assignment has a left-hand-side expression, which determines the object whose value will be assigned (or "l-value"), and a right-hand-side expression, which determines the value that will be assigned (or "r-value") to the left-hand-side object. And the whole assignment is itself an expression whose value (or "result") is its left-hand-value.

1.1.1 Objects, L-values, and R-values

An *object* is a named region of storage. 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] = '?';
```

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

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

However, the right hand side of an assignment can 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');
```

1.2 Evaluation semantics: 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, here the result 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 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);
```

1.3 Evaluation order

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();
```

What is the order of invocation of the three functions?
And what about this case:

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

What is the order of invocation of the three functions?
The answer is, *unspecified!*

1.3.1 Sequence points

The evaluation order of expressions is sometimes unspecified. 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:

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

Here the order of evaluation of `f(i)` and `g(i)` is unspecified, so neither `f(i)` is *sequenced before* `g(i)`, nor `g(i)` is *sequenced before* `f(i)`. However, both `f(i)` and `g(i)` are sequenced before the evaluation of the assignment operator, which has the side effect of changing the value of `i`. So, in other words, 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 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 *unsequed* relative to each other.

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--;
```

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

And again:

```
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?

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