Some Details of C and C Environment

Timo Karvi

October, 2013

Some Details of C and C Environment

In this last lecture, we look at

- Bit orders and storage alignment,
- Advanced aspects of C operators, and
- Some aspects of numerical calculations.

Little and big endian systems I

Suppose we have the definition

```
    unsigned int n = 257;
```

Normally *n* takes 4 bytes. In this case, two bytes contain the number 1 and the other two only zero bits. Is the situation in these Linux machines

100	101	102	103
1	1	0	0

(little endian)

or

100	101	102	103
0	0	1	1

(big endian)?

We can test this by defining

Little and big endian systems II

```
unsigned int n = 257;
uint8_t* u_ptr;

u_ptr = &n;

printf("Mem addr of n: %lu\n", u_ptr);
printf("First byte: %d  ", *u_ptr);
printf("Second byte: %d \n ", *(u_ptr+1));
```

If 1 and 1 are printed, the system is little endian. If the system is big endian, maybe zeros are printed or the system crashes (segmentation fault).

Bit order in a byte I

Assume the definitions

```
   uint8_t byte = 1;
```

Is the byte now

```
         1000 0000
```

or

```
         0000 0001 ?
```

We can examine this with shift operations. If the first were the case, then

```
         byte >> 1;
```

would produce the value 2. If the latter, then

```
         byte << 1;
```

would produce 2. It turns out that the second alternative happens in Linux machines.

Storage alignment I

Consider the following structure definition:

```
typedef struct {
    short Data1;
    short Data2;
    short Data3;
    short Data4;
} Data;
```

Then `sizeof(Data) = 8`. Let's consider another structure

```
typedef struct {
    char Data1;
    short Data2;
    int Data3;
    char Data4;
} MixedData;
```

Storage alignment II

The size of MixedData is 8, but the actual size is 10. So padding bytes are added by the compiler.

In Linux systems, malloc reserves memory blocks so that the number of the first byte is divisible by four. However, it seems possible to store an integer so that it starts from a byte not divisible by four. This kind of integer position may, however, slow down an execution.

Operators

We deal with the following:

- Assignment combinations.
- Lazy evaluation.
- Evaluation order and side-effect operators.
- Conditional operator.

Operators: Assignment

All the assignment-combination operators have the same very low precedence and associate from right to left. Consider for example the expression

```
t /= n -= m *= k += 7;
```

The evaluation starts with $k = k + 7$ and continues from right to left. So $+$ is made before $*$ even if in normal expressions the precedence of multiplication is higher than the precedence of addition.

Operators: Lazy evaluation I

- With lazy evaluation, when we skip, we skip the right operand.
- This is not confusing, when the right operand is only a simple variable. However, sometimes it is an expression with several operators.
- Consider an example:

  ```
  y = a < 10 || a >= 2 * b && b != 1;
  ```

  The left operand is `a<10` and the right operand is `a>=2*b && b!=1`.
- If $a = 7$, we skip all after $||$ and y get the value 1. If $a = 17$ and $b = 20$ the expression after && is not evaluated and y becomes 0.
- A usual assumption is that && is executed first, because it has higher precedence. Although precedence controls the construction of the parse tree, precedence simply is not considered when the tree is evaluated.

Operators: Lazy evaluation II

- Sometimes lazy evaluation can substantially improve the efficiency of a program. A much more important use of skipping is to avoid evaluating parts of an expression that would cause machine crashes or other kinds of troubles.
- Assume that we want to divide a number x, compare the answer to a minimum value, and do an error procedure if the answer is less than the minimum. But it is possible for x to be 0 and that must be checked. We can avoid a division-by-0 error and do the computation and comparison in one expression by using a *guard* before the division:

  ```
  if (x != 0) && total / x < minimum) do_error();
  ```
- The skipping may happen in the middle of an expression. Consider

  ```
  y = a < 0 || a > b && b > c || b > 10;
  ```

  for $a = 3$ and $b = 17$. The subexpression $b > c$ is not evaluated, all the other are and y becomes 1.

Operators: Evaluation Order and Side Effects I

- When used in isolation, the increment and decrement operators are convenient and relatively free of complication. When side-effect operators are used in long, complex expressions, they create the kind of complexity that fosters errors. If such an operator is used in the middle of a logical expression, it may be executed sometimes but skipped at other times.
- A second problem with side-effect operators relates to the order in which the parts of an expression are evaluated. Recall that the evaluation order has nothing to do with precedence order. We have stated that logical operators are executed left to right. This is also true of two other kinds of sequencing operators: the conditional operator ?...: and the comma (for example, for $(i = 1,\ \ j = 1; ...)$). Most other operators can be evaluated right-side first or left-side first.

Operators: Evaluation Order and Side Effects II

- This leads to one important warning: If an expression contains a side-effect operator that changes the value of a variable V, *do not use* V anywhere else in the expression. The side-effect could happen either before or after the value of V is used elsewhere in the expression and the outcome is unpredictable.

Operators: Conditional Operator I

- Even if the evaluation in an expression with the conditional operator starts by calculating true or false value, the value of the entire conditional operator, in general, will not be true or false.
- If the condition contains any postincrement operators, the increments must be done before evaluating the true clause or the false clause. Therefore, it is "safe"to use postincrement in the condition.