Computer programming

Iterative processing. Bitwise operators

Marius Minea
marius@cs.upt.ro

15 October 2013

Assignment operators

We've used the simple assignment: *lvalue* = *expression*
lvalue: variable; also: array element; pointer dereference

Compound assignment operators: `+=` `-=` `*=` `/=` `%=`
`x += expr` is a shorthand for `x = x + expr`
also for bitwise assignment operators `>>` `<<` `&` `^` `|`
use them: shorter, but also makes intent of transformation clearer

Increment/decrement operators prefix/postfix: `++` `--`
`++i` increments `i`, expression value is value *after* assignment
`i++` increments `i`, expression value is value *before* assignment

both have same *side effect* (assignment) but different *value*

```
int x=2, y, z; y = x++; /* y=2,x=3 */; z = ++x; // x=4,z=4
```

Side effects and sequencing points

The C standard defines *sequence points*, which constrain the evaluation order. Examples of sequence points are (Annex C)
– between evaluating the function designator (function expression) and arguments, and the actual call
– between evaluating first and second arguments for `&&`, `||`, `,`
– between evaluating the first operand in `? :` and the second/third

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. C standard, 6.5 Expressions

Thus, `i = i++` or `a[i] = i++` are *undefined!*

Caution with multiple side effects!

Even when order of side effects is well defined, use with caution!

DON’T write: `return i++;`
  assignment to `i` is useless, since the function returns
  obscures intent (should it be `return i;` or `return i+1;` ?

DON’T: `c = toupper(c); return c;` DO: `return toupper(c);`

DON’T read multiple characters in an expression:

```
if ((c1 = getchar()) == '*' && ((c2 = getchar()) -= '/'))
```

  if first comparison fails, second char is not read
  ⇒ hard to reason about program behavior

The `break` statement

Exits the *immediately enclosing* switch or loop statement

Used if we don't want to continue the remaining processing

Usually: `if (`*condition*` ) break;`

```
#include <ctype.h>
#include <stdio.h>
int main(void) {      // count words in input
  unsigned nrw = 0;
  while (1) {         // infinite loop, exit with break
    int c;
    while (isspace(c = getchar())); // consume spaces
    if (c == EOF) break;     // done
    nrw = nrw + 1;           // else: start of word
    while (!isspace(c = getchar()) && c != EOF); // word
  }
  printf("%u\n", nrw);
  return 0;
}
```

The `for` statement

```
for (init-clause ; test-expr ; update-expr)
   statement
```

is equivalent* with:

```
init-clause;
while (test-expr) {
   statement
   update-expr;
}
```

* except: `continue` statement, see later

Any of the 3 parts in `(...)` may be missing, but semicolons stay
If *test-expr* is absent, it is considered *true* (infinite loop)

Before C99: *init* part could only be an *expression*
Since C99: *init-clause* can also be a *declaration*
   scope of declared identifiers is loop body only

⇒ *USE* loop scope for counters, if not needed later
(scope of identifiers should only be as much as needed)

WARNING! The semicolon `;` is the *empty statement*
DO NOT use after closing `)` of `for` unless for empty body!

Counting with for loops

```
#include <stdio.h>
int main(void)
{
  unsigned n = 5;
  while (n--)  // from n-1 to 0: n-- != 0, postdecrement
    printf("loop 1: n = %d\n", n);
  n = 5;       // reinitialize after countdown to 0
  for (int i = 0; i < n; ++i) // from 0 to n-1
    printf("loop 2: counter %d\n", i);
  for (int i = 1; i <= n; ++i) // from 1 to n
    printf("loop 3: counter %d\n", i);
  for (int i = n; i > 0; --i) // from n to 1
    printf("loop 4: counter %d\n", i);
  for (int i = n; i--;)      // from n-1 to 0, postdecr.
    printf("loop 5: counter %d\n", i);
  return 0;
}
```

Counting with `for` loops

If direction does not matter, this is shortest:

```
    for (int i = n; i--;)
```

also easier to compare to zero

Warning: test expression is computed *every* time
⇒ *avoid needless computation*, e.g.

```
for (int i = 0; i < strlen(s); ++i)
```

If needed, precompute upper bound:

```
for (int i = 0, len = strlen(s); i < len; ++i)
```

(if lucky, compiler may optimize for you, but not always)

Example: rewrite, starting every word with uppercase

```
#include <ctype.h>
#include <stdio.h>
int main(void) {
  int c;
  while((c = getchar()) != EOF) {
    if (!isspace(c)) {        // first letter
      putchar(toupper(c));    // print uppercase
      while ((c = getchar()) != EOF) { // still word?
        putchar(c);           // print even if space
        if (isspace(c)) break; // but then exit
      }
    } else putchar(c);
  }
  return 0;
}
```

The `continue` statement

jumps to the *end of the loop body* in a `while`, `do` or `for` loop
i.e. to the *test*, in `while` and `do` loops
and to the *update expression* in a `for` loop

```
for (int d = 2; ; ++d) { // write prime factors of n
  if (n % d != 0) continue; // not divisible; next d
  int exp = 0;
  do exp++;                // count exp while d is factor
  while ((n /= d) % d == 0);
  printf ("%d^%d", d, exp);  // write current factor
  if (n > 1) putchar('*') else break;
}
```

Use sparingly.
can make code clearer, if decision to skip is early, and loop is long
otherwise, a simple `if` may be cleaner/clearer.

The switch statement: example

```
#include <stdio.h>
int main(void)
{
  int a = 3, b = 4, c, r;
  switch (c = getchar()) {
    case '+': r = a + b; break; // end switch
    case '-': r = a - b; break;
    case 'x': c = '*';  // continue onto next branch
    case '*': r = a * b; break;
    case '/': r = a / b; break;
    default: fputs("Unknown operator\n", stderr);
             return 1;
  }
  printf("Result: %d %c %d = %d\n", a, c, b, r);
  return 0;
}
```

The `switch` statement

Used for multiple branches depending on an *integer value*
can be clearer/more efficient than multiple `if` ... `else`

Syntax: `switch (` *integer-expression* `)` *statement*
statement is a *block* with multiple statements, some *labeled*:
`case` *value*: *statement*

The integer expression is evaluated.
If the statement has a `case` label with that value, jump to it
Otherwise, if there is a `default`, label, jump to it
Else, do nothing (goes on to next statement after `switch`)

A statement may have *several* labels (flow jumps to same code)
`case` *val1*: `case` *val2*: *statement*

Normal statement sequencing applies: flow does does *not stop* at the next `case` label (it's just a label)
⇒ to exit `switch` statement, use `break;` statement (*don't forget!*)

`switch` vs. `if` … `else`

A multiple `if` … `else` statement will do *multiple* tests
(until one succeeds)

A `switch` statement may be implemented using a *jump table*:
the expression is evaluated and used as index in a table of addresses
⇒ can be more efficient if range of possible values is limited
(also: compiler may limit range of values to 1023, cf. standard)

More importantly: a `switch` may be *easier to read*

But: *be careful* not to forget `break` where needed!

Writing and testing loops

We should consider:
  what variable changes in each iteration ?
  what is the loop continuation/stopping condition ?

Don't forget update of variable that controls loop
  (otherwise will loop forever)

What do we know on exiting the loop ? The loop condition is *false*.
  we consider this as we reason further about the program

We inspect/check/test the program:
  mentally, running it "pencil and paper" on simple cases
  then with increasingly complex tests, including corner cases

What use are bitwise operators ?

To access the internal representation of data (e.g., numbers)
and represent/encode/process some types of data efficiently

A *set* (of integers): represented by a bit for each possible element
(1 = is member; 0 = is not member of set)
⇒ sets of small integers: using an int (`uint32_t`, `uint64_t`)
 (fixed-width integer types defined in `stdint.h`)

Set operations:
 intersection = bitwise AND
 union = bitwise OR
 adding an element: setting the corresponding bit

The *current date* can be represented using bits:
day: 1-31 (5 bits); month: 1-12 (4 bits)
year: 7 bits suffice for 1900 to 2027
⇒ need operations to extract day/month/year from a 16-bit value
(e.g. `uint16_t`)

Bitwise operators

Can *only* be used for *integer* operands!

<table>
<tr><th></th><th></th><th></th></tr>
<tr><td><code>&</code></td><td>bitwise AND</td><td>(1 only if both bits are 1)</td></tr>
<tr><td><code>|</code></td><td>bitwise OR</td><td>(1 if at least one of the bits is 1)</td></tr>
<tr><td><code>^</code></td><td>bitwise XOR</td><td>(1 if exactly one of the bits is 1)</td></tr>
<tr><td><code>~</code></td><td>bitwise complement</td><td>(opposite value: $0 \leftrightarrow 1$)</td></tr>
<tr><td><code><<</code></td><td colspan="2">left shift with number of bits in second operand
vacated bits are filled with zeros; leftmost bits are lost</td></tr>
<tr><td><code>>></code></td><td colspan="2">right shift with number of bits in second operand
vacated bits filled with zero if number is unsigned or nonnegative
else implementation-dependent (usually repeats sign bit)
$\Rightarrow$ use only <code>unsigned</code> for portable code!</td></tr>
</table>

All operators work with *all bits* of operands at the same time
they *don't change operands*, just give a result (like usual `+`, `*`, etc.)

Properties of bitwise operators

`n << k` has value $n \cdot 2^k$ (if no overflow)
`n >> k` has value $n/2^k$ (integer division) for unsigned/nonnegative
`1 << k` has 1 only in bit `k` ⇒ is 2^k for `k < 8*sizeof(int)`
⇒ use this, *not* `pow` (which is floating-point!)
`~(1 << k)` has 0 only in bit k, rest are 1

`0` has all bits 0, `~0` has all bits 1 (= -1, since it's a signed int)
`~` preserves signedness, so `~0u` is unsigned (`UINT_MAX`)

`&` with 1 preserves a bit, `&` with 0 is always 0
`n & (1 << k)` *tests* (is nonzero) bit k in `n`
`n & ~(1 << k)` *resets* (makes 0) bit k in the result

`|` with 0 preserves a bit, `|` with 1 is always 1
`n | (1 << k)` *sets* (to 1) bit k in the result

`^` with 0 preserves value, `^` with 1 flips value
`n ^ (1 << k)` *flips* bit k in result

Again, *none of these have side effects*, they just produce results.

Creating and working with bit patterns (masks)

& with 1 preserves	& with 0 resets
| with 0 preserves	| with 1 sets

Value given by bits 0-3 of `n`: AND with $0\ldots01111_{(2)}$ `n & 0xF`
Reset bits 2, 3, 4: AND with ~$0\ldots011100_{(2)}$ `n &= ~0x1C`
Set bits 1-4: OR with $11110_{(2)}$ `n |= 0x1E` `n |= 036`
Flip bits 0-2 of `n`: XOR with $0\ldots0111_{(2)}$ `n ^= 7`
⇒ choose fitting operator and *mask* (easier written in hex/octal)

Integer with all bits 1: `~0` (signed) or `~0u` (unsigned)
k rightmost bits 0, rest 1: `~0 << k`
k rightmost bits 1, rest 0: `~(~0 << k)`
`~(~0 << k) << p` has k bits of 1, starting at bit p, rest 0
`(n >> p) & ~(~0 << k)`: `n` shifted `p` bits, reset all except last `k`
`n & (~(~0 << k) << p)`: reset all except `k` bits starting at bit `p`