Computer programming

Iteration. Side effects. Recursive text processing

Marius Minea

marius@cs.upt.ro

17 October 2016

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`
see later: also bitwise assignment operators `>>` `<<` `&` `^` `|`
use them: shorter, 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
```

⇒ same effect as *statements*, not same value in *expressions*

Side effects and sequence points

C standard defines *sequence points*, they define a *partial order* between evaluations (order specified for some but not all pairs).

All side effects must complete before crossing a sequence point.
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 (getchar() == '*' && ((c = getchar()) == '/'))
```

  if first comparison fails, second char is not read
  (c has previous / uninitialized value)
  ⇒ 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:

* except: `continue` statement, see later

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

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*, e.g. `i = 0`
Since C99: *init-clause* can also be a *declaration*, e.g. `int i = 0`
   scope of declared identifiers is loop body only

$\Rightarrow$ *USE* loop scope for counters, if they are 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

```c
#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)`~~

(compiler may optimize some, but not always)

If needed, precompute upper bound:

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

Example: rewrite, starting every word with uppercase

```c
#include <ctype.h>
#include <stdio.h>
int main(void) {
  int c;
  while((c = getchar()) != EOF) {
    if (!isspace(c)) {        // first non-space
      putchar(toupper(c));    // print uppercase if letter
      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 *update expression* in `for` and to *test* in `do` or `while`

```
void printfact(unsigned n) { // print prime factors of n
  for (unsigned d = 2; d*d <= n; d += 1 + d % 2) {
    if (n % d != 0) continue; // not divisible; next d
    unsigned exp = 1;
    while ((n /= d) % d == 0) ++exp;
    printf ("%u", d);   // write current factor
    if (exp > 1) printf("^%u", exp); // write exponent
    if (n > 1) putchar('*'); else return;
  }
  printf("%u", n); // 0, 1 or remaining prime
}
```

Use `continue` sparingly (appears much less often than `break`)
can make code clearer, if decision to skip is early, and loop is long
otherwise, a simple `if` may be cleaner and clearer.

The goto statement

Syntax: goto *somelabelname* ;

Jumps to statement with given label, only inside same function.

Any statement can be labeled with *somelabelname* :

Discouraged (unstructured code); ok to jump out of several loops.

```
#include <ctype.h>
#include <stdio.h>
int main(void)        // count chars, words, lines
{
  unsigned nc = 0, nw = 0, nl = 0;
  for (int c; (c = getchar()) != EOF; ++nc) {
    if (!isspace(c))  // word start
      for (++nc, ++nw; !isspace(c = getchar()); ++nc)
        if (c == EOF) goto outloop;  // exit both loops
    if (c == '\n') ++nl; // c isspace here; ++nc in for
  }
  outloop: printf("%u lines, %u words, %u chars\n", nl, nw, nc);
  return 0;
}
```

The switch statement: example

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

```
#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

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 *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

Example: Parsing expressions

Expression *syntax*: rigorously defined by a *grammar*
frequent notation: Backus-Naur form (BNF)

Writing code: one function for each defined notion (*nonterminal*)

Prefix expressions (no parantheses/precedence needed)
expr ::= *number* | *operator expr expr*

Postfix expressions
expr ::= *number* | *expr expr operator*

Left recursive ⇒ both variants start alike, can't decide choice
⇒ rewrite grammar:

expr ::= *number restexpr*
restexpr ::= ϵ | *expr operator restexpr*
ϵ is usual notation for empty string

Parsing usual (infix) expressions

Simplest attempt: does not deal with precedence or parantheses:

$\mathit{expr} ::= \mathit{number} \mid \mathit{expr}\ \mathit{operator}\ \mathit{expr} \mid (\ \mathit{expr}\ )$

$\Rightarrow$ distinguish additive/multiplicative expressions/operators

$\mathit{expr} ::= \mathit{term} \mid \mathit{expr} + \mathit{term} \mid \mathit{expr} - \mathit{term}$
$\mathit{term} ::= \mathit{factor} \mid \mathit{term} * \mathit{factor} \mid \mathit{term}\ /\ \mathit{factor}$
$\mathit{factor} ::= \mathit{number} \mid (\ \mathit{expr}\ )$

which variant to choose? *expr* or *term*? $\Rightarrow$ rewrite:

$\mathit{expr} ::= \mathit{term}\ \mathit{restexpr}$
$\mathit{restexpr} ::= \epsilon \mid + \mathit{term}\ \mathit{restexpr} \mid - \mathit{term}\ \mathit{restexpr}$
$\mathit{term} ::= \mathit{factor}\ \mathit{restterm}$
$\mathit{restterm} ::= \epsilon \mid * \mathit{factor}\ \mathit{restterm} \mid /\ \mathit{factor}\ \mathit{restterm}$
$\mathit{factor} ::= \mathit{number} \mid (\ \mathit{expr}\ )$

Writing code from recursive definitions

One *function* for each *nonterminal*

Function structure determined by computation (*data flow*)

expr ::= *term restexpr*
`restexpr` needs previous `term` ⇒ gets it as parameter

```
int expr(void) { return restexpr(term()); }
```

restexpr ::= ϵ | + *term restexpr* | - *term restexpr*
restexpr is right-recursive write as *tail-recursive* function

```
int restexpr(int t1) {
  int c = getchar();
  if (c == '+') return restexpr(t1 + term()); else ...
}
```

or rewrite as loop within `expr()`, *accumulate* expression value

```
int expr(void) {
  int c, e = term();
  for (;;) {  // use break; to stop
    if ((c = getchar()) == '+') e += term; else ...
}
```