Computer programming

Iterative processing

Marius Minea

marius@cs.upt.ro

14 October 2014

Basic text processing: reading all text

Reading all input, doing nothing
body of while is empty, `;` is the *empty statement*

```
#include <stdio.h>

int main(void) {
  int c;
  while ((c = getchar()) != EOF);
  return 0;
}
```

Reading and printing all input:

```
  int c;
  while ((c = getchar()) != EOF)
    putchar(c);
```

Text processing: pattern starting with given char

If we search for text starting with a given char,
continue checking for text in the `if ( )` that has found that char:
e.g. ignore \ if followed by letters, print rest

```
#include <stdio.h>
int main(void) {
  int c;
  while ((c = getchar()) != EOF) {
    if (c == '\\')    // found backslash
      if (isalpha(c = getchar()))    // if letter
        while (isalpha(c = getchar())); // skip more letters
      else putchar('\\');
    putchar(c);       // print, also after cases above
  }
  return 0;
}
```

This has a slight problem, do you notice ?

Text processing: pattern starting with given char

If string of letters ends with EOF, will also try to print EOF
  −1 converted to code 255 (strange character, ÿ)

When *searching* for a given char, must also test for EOF
When *using* a char (e.g. after a loop), must check it's not EOF

```
#include <stdio.h>
int main(void) {
  int c;
  while ((c = getchar()) != EOF) {
    if (c == '\\') {          // found backslash
      if (isalpha(c = getchar()))    // skip any letters
        while (isalpha(c = getchar()));
      else putchar('\\');            // no letters
      if (c != EOF) putchar(c);      // last char read
    } else putchar(c);        // not backslash
  }
  return 0;
}
```

Finding repeated patterns

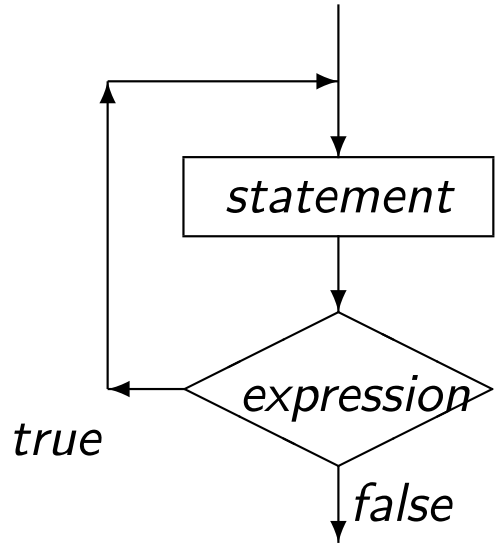
Ex: ignore \ followed by repeated text between braces
`\{text1}{text2}...`

```
#include <stdio.h>
int main(void) {
  int c;
  while ((c = getchar()) != EOF)
    if (c == '\\') {           // found backslash
      while ((c = getchar()) == '{')
        while ((c = getchar()) != '}')
          if (c == EOF) return 1; // incomplete
      if (c != EOF) putchar(c); // char after pattern
    } else putchar(c); // anything else
  return 0;
}
```

Often, it is useful two write functions for parts of the pattern (makes code more manageable)

The `do-while` loop (final test)

```
do
   statement
while ( expression );
```



Sometimes we know that a cycle needs to be executed at least once (we read at least one character, a number has at least one digit)

Like the `while` loop, executes *statement* as long as the expression evaluates to true (nonzero)

Expression is (re)evaluated *after* every iteration

Equivalent with:

```
statement
while ( expression )
   statement
```

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
  $\Rightarrow$ 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*, 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)
```

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

```
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 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

```c
#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)
$\Rightarrow$ 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

$\Rightarrow$ 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