Computer Programming

Characters. Assignment. Iteration

Marius Minea

marius@cs.upt.ro

9 October 2017

Write functions to reuse code!

Copy-paste is bad! Any later change must be done twice.
Code used multiple times should be put in a *function*

Example: median of three numbers.
After comparing two numbers, we know smaller and larger one:
$\Rightarrow$ need same computation, with switched numbers $\Rightarrow$ function!

```
double min(double x, double y) { return x < y ? x : y; }

// know hi <= lo, nothing about c
double med_aux(double lo, double hi, double c) {
  return c < lo ? lo : min(c, hi);
}

double med(double a, double b, double c) {
  return a < b ? med_aux(a, b, c) : med_aux(b, a, c);
}
```

Compiler will optimize and inline function calls if needed.

Recursion: reformulating problem

Anything that a function needs becomes a *parameter*!

Fibonacci sequence: $F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}, n > 1$.
Naive translation is inefficient, recomputes same terms.
$\Rightarrow$ values computed and still needed become parameters

```
// n = number of addition steps still needed
unsigned fib3(unsigned n, unsigned last, unsigned prev) {
  return n > 0 ? fib3(n - 1, last + prev, last) : last;
}

// initial call: last = 0, prev = 1 (as if index = -1)
unsigned fib(unsigned n) { return fib3(n, 0, 1); }

// or: special case for 0 and 1, then n - 1 additions
unsigned fibo(unsigned n) {
  return n < 1 ? n : fib3(n - 1, 1, 0);
}
```

Characters. ASCII code

ASCII = American Standard Code for Information Interchange

Characters are represented as a *numeric code* = index in this table

e.g. `'0'` == 48, `'A'` == 65, `'a'` == 97, etc.

```
       0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
-----------------------------------------------------
0x0   \0                        \a \b \t \n \v \f \r
0x10:
0x20:     !  "  #  $  %  &  '  (  )  *  +  ,  -  .  /
0x30:  0  1  2  3  4  5  6  7  8  9  :  ;  <  =  >  ?
0x40:  @  A  B  C  D  E  F  G  H  I  J  K  L  M  N  O
0x50:  P  Q  R  S  T  U  V  W  X  Y  Z  [  \  ]  ^  _
0x60:  `  a  b  c  d  e  f  g  h  i  j  k  l  m  n  o
0x70:  p  q  r  s  t  u  v  w  x  y  z  {  |  }  ~
```

Prefix `0x` denotes *hexazecimal constants* (in base 16)

Characters $<$ `0x20` (space): *control characters*

digits; uppercase letters; lowercase letters: 3 contiguous sequences

ASCII: only up to `0x7f` (127); then national chars, multi-byte, etc.

The character type

The standard type `char` is used to represent characters.

`char` is an *integer type* with smaller range than `int` or `unsigned`
$\Rightarrow$ can be stored in a byte (`CHAR_BIT` ≥ 8 bits)

`char` can be `signed char` (at least `-128` to `127`)
or `unsigned char` (at least `0` to `255`). Both are included in `int`.

Character constants are written between (single) *quotes* `' '`
They are *integer values*, *implicitly converted* to `int` in expressions.

Digits, lowercase letters and uppercase letters are *consecutive* $\Rightarrow$

```
'7' == '0' + 7   '5' - '0' == 5   'E' - 'A' == 4   'f' == 'a' + 5
```

Escape sequences (textual representation) for special chars:

`'\0'`	null	`'\n'`	newline
`'\a'`	alarm	`'\r'`	carriage return
`'\b'`	backspace	`'\f'`	form feed
`'\t'`	tab	`'\''`	single quote
`'\v'`	vertical tab	`'\\'`	backslash

Writing a character: `putchar`

Declaration, in `stdio.h` : `int putchar(int c);`
Call (sample use): `putchar('7')`

Writes an `unsigned char` (given as `int`); returns its value, or `EOF` (constant -1) on error

```
#include <stdio.h>
int main(void)
{
  putchar('A'); putchar(':'); // writes A then :
  putchar(getchar());         // prints character read
  return 0;
}
```

Chars are just ints (stored in one byte).

`'A'` is just another way of writing 65 .

Printing a number digit by digit

```c
#include <stdio.h>

void printnat(unsigned n) { // recursive, digit by digit
  if (n >= 10)              // if it has several digits
    printnat(n/10);         // write first part
  putchar('0' + n % 10); // always write last digit
}

int main(void)
{
  printnat(312);
  return 0;
}
```

Reading a character: `getchar()`

Declaration, in `stdio.h` : `int getchar(void);`
Call (use): `getchar()` without parameters, but with `()`

Returns an `unsigned char` converted to `int`,
or the value `EOF` (negative int, usually -1) if no char could be read
(e.g., at end-of-file)

The character read is *consumed* from input (no longer available);
next call to `getchar()` returns *next* character (not the same!)

`getchar()` needs to return `int`, not `char` to also include `EOF`
(negative, different from any `unsigned char`)
When typing, characters are *echoed*, and placed in a *buffer*.
They are available to `getchar()` only after typing *Enter*.

WARNING! We have NO CONTROL over input data!
$\Rightarrow$ program must *validate* (check) them, and handle errors

Side effects

Pure computation has no other effect: this program prints nothing!

```
int sqr(int x) { return x * x; }
int main(void) { return sqr(2); }
```

Repeatedly calling the *same function* (in mathematics, or examples `sqr`, `pwr`, etc.) with the same parameters gives the *same result*.

Output (`printf`) produces a *visible* (and irreversible) *effect*.
Input with `getchar()` returns a *different* character on each call; the character is *consumed*.

A *side effect* is a change in the state of the execution environment e.g., *reading*, *writing*, *assignment*.

WARNING! Careful when using functions that have side effects, since they can interact (unexpectedly) through these effects.
$\Rightarrow$ write *side-effect free* functions whenever possible!

Reading a natural number, digit by digit

Digits are included one by one in the result: assume `1475<Enter>`

result	next char
0	`'1'`
1	`'4'`
14	`'7'`
147	`'5'`
1475	`'\n'`

If current (partial) number is r
and *value* of next digit is d,
next number is $r' = 10 * r + d$

Redefine the problem: Define a function that computes the number from the already read part `r` and the current character `c`:
– when the char read is not a digit, return accumulated number `r`
– else, recursive call with `10*r + c - '0'` and next char read

WARNING! `getchar()` returns the *character code* (e.g. ASCII), NOT the *value of the digit*

when typing 6, `getchar()` does NOT return 6, but `'6'`
⇒ we adjust subtracting `'0'`: 6 is `'6' - '0'`

Reading a natural number (cont.)

`ctype.h` has declarations of functions for classifying characters: `isalpha`, `isalnum`, `isdigit`, `isspace`, `islower`, `isupper`, etc. They take a character as parameter and return true (nonzero) or false (zero) (the character is of the stated type, or not)
Also: case mapping: `tolower`, `toupper` (return transformed value).

Redefined problem: Define a function that computes the number from the already read part *r* and the current digit *c*:

```
#include <ctype.h>
#include <stdio.h>
unsigned readnat_rc(unsigned r, int c) {
  return isdigit(c) ? readnat_rc(c-'0' + 10*r, getchar()) : r;
}
```

The new char read is passed as argument to the next recursive call.

Initially, we start from number 0 and the first character read:

```
unsigned readnat(void) { return readnat_rc(0, getchar()); }
```

Note: no error checking; consumes first character that is not a digit

From parameters to variables

So far, we've written functions that work with their parameters
Parameters are *bound* at call time to the values of the arguments.

Sometimes, we repeatedly need to work with values that are obtained *within* a function ⇒ need to also bind these to a name.

We *declare* a (local) *variable* and *initialize* it with a value.

This is *not assignment*, we still don't need to *change* the value!

`readnat` can read the char `c` rather than get it as parameter:

```
unsigned readnat_r(unsigned r)
{
  int c = getchar();
  return isdigit(c) ? readnat_r(c-'0' + 10*r) : r;
}
unsigned readnat(void) { return readnat_r(0); }
```

Exercise: trace by hand the calls if the input is `143<Enter>`

Dealing with the extra character

Reading a number stops at the first non-digit (or `EOF`).
That char is not part of the number and should not be consumed.

```
int ungetc(int c, FILE *stream); //declaration
```

puts a character c back into a given input stream (file).

For now, we use standard input: `ungetc(c, stdin) //call`

```
unsigned readnat_r(unsigned r) {
  int c = getchar();
  if (isdigit(c)) return readnat_r(c - '0' + 10*r));
  else { ungetc(c, stdin); return r; }
}
unsigned readnat(void) { return readnat_r(0); }
```

We could also use *comma* as sequencing operator for expressions:

```
  return isdigit(c) ? readnat_r(c - '0' + 10*r)
       : (ungetc(c, stdin), r);
```

The expression before the comma is *evaluated*, its value is *ignored*, the value of the entire expression is that of the *second part*.

Reading an integer (possibly signed)

We now read an integer, with an optional sign

```
int readint(void)
{
  int c = getchar();
  return c == '-' ? - readnat()
    : c == '+' ? readnat() : (ungetc(c, stdin), readnat());
}
```

If c is not a sign, it may be the first digit of the number
`ungetc(c, stdin)` puts c back into standard input
it will be returned again on the next read, e.g. with `getchar()`

Our `readnat` does not handle errors (on non-digit, it returns 0).
We could return a special value (e.g. `INT_MIN`), and have the caller check (this would reduce the range of useful values by 1).

Simple arithmetic expressions

How do we evaluate this expression?

$$\underbrace{3 * 8 - (7 - 2) * 4}_{} + \underbrace{2 * 3}_{}$$

Find and mark the *last* operation to perform

$$\begin{array}{ll} & 3 * 8 - (7 - 2) * 4 \\ + & 2 * 3 \end{array} \qquad \begin{array}{ll} & \textit{expr1} \\ + & \textit{expr2} \end{array}$$

We've made the *recursive structure* explicit!

Instead of highlighting, we can *write the operator first*:

$$+\ \textit{expr1}\ \textit{expr2}$$

just like function notation:

$$\textit{add}(\textit{expr1}, \textit{expr2})$$

Prefix expressions

$+$ *expr1 expr2* instead of *expr1* $+$ *expr2*

We've just *recursively* defined *prefix expressions*

$$expr = \begin{cases} num \\ + \quad expr \quad expr \\ - \quad expr \quad expr \\ * \quad expr \quad expr \end{cases}$$

implicit in definition: *expr* on the right is in the same format

With the operator always first, no need for parantheses!

Working with prefix expressions

prefix	usual (infix) notation
$* \; 3 \; 8$	$3 * 8$
$* \; \underbrace{- \; 7 \; 2} \; 4$	$(7 - 2) * 4$
$- \; \underbrace{* \; 3 \; 8} \; \underbrace{* \; \underbrace{- \; 7 \; 2} \; 4}$	$3 * 8 - (7 - 2) * 4$

We'll transcribe the rules into code:

read and evaluate an expression that is

num		
$+$	*expr*	*expr*
$-$	*expr*	*expr*
$*$	*expr*	*expr*

expr will be a *recursive function*: read input, return a value

expr on right-hand side: recursive calls for subexpressions

From syntax to code

$$
\begin{array}{lllll}
\textit{expr} = & \textit{num} & & & \\
 & + & \textit{expr} & \textit{expr} \\
 & - & \textit{expr} & \textit{expr} \\
 & * & \textit{expr} & \textit{expr}
\end{array}
$$

First character decides which definition variant (branch) to follow

read first character (non-whitespace)
if digit
  read number, return value
else if known operator
  read_eval expression
  read_eval expression
  apply operator, return value

Code directly follows the structure of the definition!

Logical expressions in C

The *condition* in the `if` statement or the `? :` operator is usually a *relational expression*, with a *logical value*: `x != 0`, `n < 5`, etc.
The C language was conceived without a special boolean type
since C99, `stdbool.h` has type `bool`, `false` (0) and `true` (1)

A value is considered *true* when *nonzero* and *false* when *zero*
(when used as a condition in `? :` , `if` , `while` etc.)
⇒ condition must have *scalar* type (integer, floating point, enum)

Comparison operators (`==` `!=` `<` etc.)
return the *integer* values 1 (for *true*) or 0 (for *false*)
⇒ suitable for direct use as conditions

Library functions often return zero or nonzero (NOT zero or one!)
only compare `if (isdigit(c))` (nonzero), don't compare to 1 !

Logical operators

With logical operators, we can write complex decisions:

expr	! *expr*
0	1
$\neq 0$	0

negation ! NOT

e_1 && e_2	e_2: 0	e_2: $\neq 0$
e_1: 0	0	0
e_1: $\neq 0$	0	1

conjunction && AND

e_1 || e_2	e_2: 0	e_2: $\neq 0$
e_1: 0	0	1
e_1: $\neq 0$	1	1

disjunction || OR

Reminder: logical operators produce 1 for *true*, 0 for *false*
An integer is interpreted as *true* if *nonzero*, and as *false* if 0

Example: leap year

Years divisible by 4 are leap years
except those divisible by 100 which are not
except those divisibile by 400 which still are

Can't directly translate like this
(can't write exception case after normal case already handled)
$\Rightarrow$ need to reverse order:

```
int isleap(unsigned yr)
{
  if (yr % 400 == 0) return 1;
  if (yr % 100 == 0) return 0;
  return yr % 4 == 0;
}
```

What test order for fewest checks on average? (years equally likely)

Example: leap year

A year is a leap year if

it is divisible by 4 **and**
it is **not** divisible by 100 **or** it is divisible by 400

```
int isleap(unsigned yr)     // 1: leap year, 0: not
{
  return yr % 4 == 0 && (!(yr % 100 == 0) || yr % 400 == 0);
}
```

`!(yr % 100 == 0)` is equivalent with `(yr % 100 != 0)`

Precedence of logical operators

The *unary logical operator* `!` (logical negation): highest precedence

`if (!found)` same as `if (found == 0)` (zero is false)

`if (found)` same as `if (found != 0)` (nonzero is true)

Relational operators: lower precedence than arithmetic ones

$\Rightarrow$ we can naturally write `x < y + 1` for `x < (y + 1)`

Precedence: `>` `>=` `<` `<=` , then `==` `!=`

Binary logic operators: `&&` (AND) evaluated before `||` (OR)

have lower precedence than relational operators

$\Rightarrow$ can naturally write `x < y + z && y < z + x`

Careful with logical operators!

Check your logic! Mistaking `&&` and `||` is a common error.

DON'T write code like ~~`if (c >= 'A'|| c <= 'z') ...`~~

- Lots of possible errors:
  - easy to get && and || wrong (what happens above?)
  - not all chars between `'A'` and `'z'` are letters
  - can write < or > instead of <=, etc.
- Hard to read (quality check will reject it)

Much simpler and safer with `isupper`, `islower`, `isalpha`, etc.

Correct code is designed to be simple:

- use `if (isdigit(c)) ...` (asks: is it a digit?)
- NOT ~~`isdigit(c) == 1`~~
- functions from `ctype.h` don't return 1, just something *nonzero*

Short-circuit evaluation

Logical expressions are evaluated *left to right*
(in general, for other operators, evaluation order is *unspecified*)
Evaluation stops (*short-circuit*) when the result is known:
for `&&`, when the left argument is false (right is not evaluated)
for `||`, when the left argument is true

```
if (p != 0 && n % p == 0)
  printf("p divides n");

if (p != 0)                 // only if nonzero
  if  (n % p == 0)          // test the remainder
    printf("p divides n");
```

⇒ Be careful when writing compound tests!
⇒ Avoid side-effects in compound tests (or place them first)

Precedence and *evaluation order* are different notions!
`2 * f(x) + g(x)` : multiplication before addition (precedence)
Unspecified which part of sum is evaluated first (`f` or `g`)

Declaring variables

A *variable* is an obiect with a *name* and a *type*.
It stores values (other than function arguments) needed later

parameters: for values given to the function (by the caller)
variables: for (auxiliary) values computed in the function

Variable declaration: for one or more variables of the *same type*:

```
double x;
int a = 1, b, c;
```

`a` is initialized with 1, the other variables are not

WARNING! Variables declared locally in a block (function) are *NOT initialized* by default!

When we declare a variable, we should know why we need it
$\Rightarrow$ good practice to *initialize* it immediately with the needed value

A function body `{ }` is a sequence of *declarations* and *statements*
since C99, declarations and statements can appear in any order
(in previous standards: first all declarations, then statements)

Variables: scope and lifetime

The *scope* of an identifier (e.g., variable) is the program region where it is *visible* (can be used)

Function *parameters and variables declared in functions* have the function *body as scope* $\Rightarrow$ are *not visible outside* the function

Thus, parameter names for different functions do not conflict
like in mathematics, we can have $f(x) = \ldots$ and $g(x) = \ldots$
same for local variables

The *storage duration* or *lifetime* of an object (e.g., variable) is the part of program execution during which storage is reserved for it.

Local variables have *automatic* storage duration:
they are automatically created on each call and *destroyed on return*
(they do not exist between calls, thus do not preserve their value)

Assignment

In recursive functions we don't need to change variable values
  a programming style typical for (pure) *functional languages*
Recursive calls create *new parameter instances* with *new values*.

In *imperative programming*, we use:
variables to represent objects used in solving the problem
  (current character; partial result; number left to process)
assignment, to give a *new value* to a variable
  (to express a computation step in the program)

Syntax: *variable* = *expression*
    Everything is an *assignment expression*.

Effect: 1. The expression is evaluated
    2. the value is *assigned* to the variable
      and becomes the value of the entire expression.

Example: `c = getchar()` `n = n-1` `r = r * n`

Assignment (cont'd)

May appear in other expressions:

```
if ((c = getchar()) != EOF) ...
```

May be chained: `a = b = x+3` `a` and `b` get the same value

Any *expression* (function call, assignment) with `;` is a *statement*

```
printf("hello");   c = getchar();   x = x + 1;
```

A variable changes value *only by assignment*!
NOT in other expressions, or by passing as parameter!

`n + 1` `sqr(x)` `toupper(c)` *compute*, DON'T change!

`n = n + 1` `x = sqr(x)` `c = toupper(c)` *change*

WARNING! `=` assignment `==` comparison.

From recursion to iteration

Recall: reversing a number

rev(`n` ,	`r`)
rev(465 ,	empty(0))
rev(46 ,	5)
rev(4 ,	56)
rev(0 ,	564)

We have done repeated processing through *recursion*

- on each call, *new values* for *parameters*
  - `n/10`, `10*r+n%10`
- *condition* controls repetition (call) or termination (base case)
  - `n == 0`
- one *function* for each repetitive computation

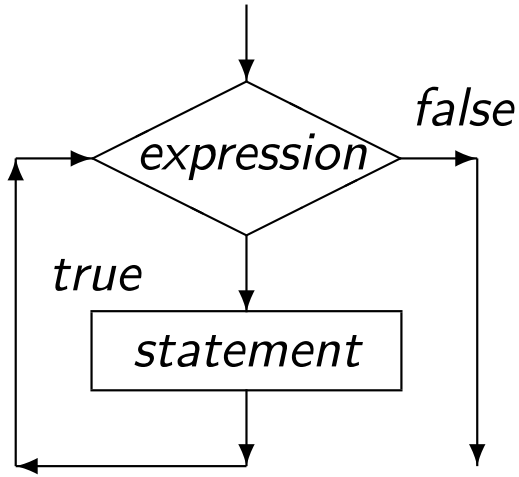
Iteration. The `while` loop (initial test)

Expresses the repetition of a statement, guarded by a condition:

Syntax:

```
while ( expression )
  statement
```

!!! Expression must be between parantheses ( )



Semantics: evaluate expression. If it is true (nonzero):

(1) execute statement (loop *body*)

(2) go back to start of `while` (evaluate expression)

Else (if condition is false/zero), don't execute anything.

$\Rightarrow$ body executes repeatedly, as long as (while) condition is true

Iteration and recursion

We can define iteration (the while loop) recursively:

```
while ( expression )
  statement
```

is the same as

```
if ( expression ) {
  statement
  while ( expression )
    statement
}
```

Recursion is fundamental. It can express any iteration.

Rewriting recursion as iteration

```
unsigned fact_r(unsigned n,
                unsigned r) {
  return n > 0
      ? fact_r(n - 1, r * n)
      : r;
}
// called with fact_r(n, 1)
```

```
unsigned fact_it(unsigned n) {
  unsigned r = 1;
  while (n > 0) {
    r = r * n;
    n = n - 1;
  }
  return r;
}
```

```
int pow_r(int x, unsigned n,
                 int r) {
  return n > 0
       ? pow_r(x, n-1, x*r)
       : r;
}
// called with pow_r(x, n, 1)
```

```
int pow_it(int x, unsigned n) {
  int r = 1;
  while (n > 0) {
    r = x * r;
    n = n - 1;
  }
  return r;
}
```

Rewriting recursion as iteration

Easier if function is written by accumulating a partial result (*tail recursion*)

Stop test and initial result value are the same as in recursion

Recursion creates *new instances* of parameters for each recursive call, with new values dependent on the old ones:

ex. `n * r`, `n - 1`, `x * r`, etc.

Iteration *updates (assigns)* values to variables in each iteration, using the same rules/expressions

Ex. `r = n * r`, `n = n - 1`, `r = x * r`

Both variants return the accumulated result

Caution! Recursion and iteration both repeat a processing step
$\Rightarrow$ in a problem we use one or the other, rarely both

Reading a number iteratively, digit by digit

```c
#include <ctype.h>    // for isdigit()
#include <stdio.h>    // for getchar(), ungetc(), stdin
unsigned readnat(void)
{
  unsigned r = 0;     // accumulates result
  int c;              // character read
  while (isdigit(c = getchar())) // while digit
    r = 10*r + c - '0';          // build number
  ungetc(c, stdin);   // put back char != digit
  return r;
}
int main(void) {
  printf("number read: %u\n", readnat());
  return 0;
}
```

Basic text processing: reading all text

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

Caution! Do not write `;` after `while` unintentionally!

```
#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);
```

Basic text processing: finding a char

```
int c;
while ((c = getchar()) != EOF)
  if (c == '.') puts("Found period!");
```

Often, we search for more text of some sort after that character. looking for the first word:

```
#include <stdio.h>
int main(void) {
  int c;
  while ((c = getchar()) != EOF)
    if (c == '.') {   // found period
      while (isspace(c = getchar())); // skip whitespace
      // print next word made of letters. spot the bug?
      while (isalpha(c = getchar())) putchar(c);
  }
  return 0;
}
```

Reading character by character: filters

Function that reads and prints up to a specified character
returns that character or EOF if reached before that char

```
int printto(int stopchar)    // up to what char ?
{
  int c;
  while ((c = getchar()) != EOF && c != stopchar)
    putchar(c);
  return c;
}
```

DON't forget ( ) `(c=getchar())!=EOF` (assign, then compare)

```
int skipto(int stopchar)     // ignore up to stopchar
{
  int c;
  while ((c = getchar()) != EOF && c != stopchar);
  return c;
}
```

`;` after `while` means an empty loop body. *Don't use by mistake!*

ERRORS with characters and loops

NO! ~~`char c = getchar();`~~ YES: `int c = getchar();`

If `char` is `unsigned char`, `c` will never compare equal to `EOF (-1)`

⇒ will never leave a `while (c != EOF)` loop

If `char` is `signed char`, reading byte 255 becomes `-1 (EOF)`

⇒ a valid char (code 255) will be taken as `EOF` (early stop)

NO! ~~`while (!EOF)`~~ `EOF` is a nonzero constant (`-1`)

thus the condition is always false, the loop is never entered!

YES: `while ((c = getchar()) != EOF)` and careful with the `()` !

NO! ~~`while (c = getchar() != EOF)`~~

`!=` has higher precedence, its result (`0` or `1`) is assigned to `c`

NO! ~~`int c = getchar(); if (c < 5) puts("failed exam");`~~

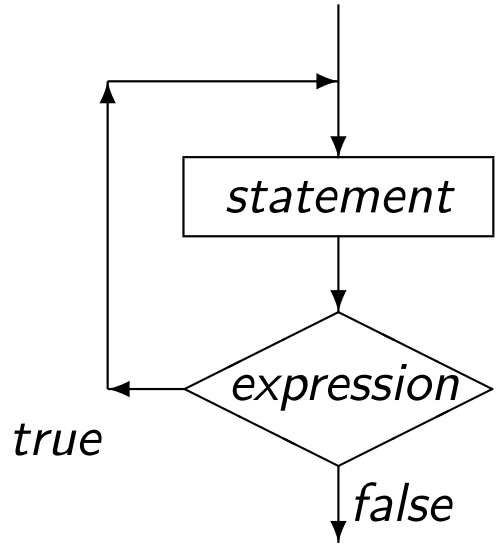
`c` is ASCII code, not value of a one-digit number. Need `c-'0'`

NO! ~~`while ((c = getchar()) != '\n')`~~ may loop forever!

YES: `while ((c = getchar()) != '\n' && c != EOF)` will exit!

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
```

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