

Computer Programming

## Recursion. Assignment. Iteration

Marius Minea

marius@cs.upt.ro

10 October 2016

## 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:

⇒ need same computation, with switched numbers ⇒ 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.

⇒ 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);
}
```

## Recursion: reverse digits in number

Often, problem restated with explicit *partial result* (accumulator)

n	r
1465	empty(0)
146	5
14	56
1	564
empty(0)	5641

What is the result of reverting

*given that*

the end has already been reverted

the resulting number is r

and remaining part is n?

```
unsigned rev2(unsigned n, unsigned r) {  
    return n == 0 ? r : rev2(n/10, 10*r + n % 10);  
}
```

```
// initial reversed part is zero
```

```
unsigned rev(unsigned n) { return rev2(n, 0); }
```

Careful: **return** in base case *must use accumulator*

(else computation is thrown away!)

# Recursion can express arbitrary repetition

*Base case*: are we done? return (result)  
not just: zero or first term

*Recursive case* (not done):  
compute new partial result  
call recursive function with new partial result  
(usually an extra parameter, besides initial input)

## Recursion for computing approximations: square root

Babylonian method:  $a_0 = 1$ ,  $a_{n+1} = \frac{1}{2}(a_n + \frac{x}{a_n})$

*recurrent* sequence of approximations  $\Rightarrow$  recursive solution  
*given* (parameters):  $x$  and the current approximation  
*result* = a satisfactory approximation (precision  $\epsilon$ )

Re-state problem: compute  $\sqrt{x}$  *given current approximation*  $a_n$   
*In recursion, partial result is usually carried as parameter*

Computation:

if precision good  $|a_{n+1} - a_n| < \epsilon$  return *current approximation*  $a_n$   
(base case)

else, return value computed starting from *new approximation*  $a_{n+1}$   
(recursive call)

We no longer need an index  $n$ , and the base case is not  $n = 0$   
(but it's still the case when nothing left to compute)

Can prove: error to  $\sqrt{x}$  is less than distance between last two terms

## Square root by approximation

```
#include <math.h>
// needed for double fabs(double x); (abs. value for reals)

// root of x with error < 1e-6 given approximation a_n
double root2(double x, double an)
{
    return fabs(a_n - x/a_n) < 2e-6 ? a_n
        : root2(x, (a_n + x/a_n)/2);
}
double root(double x) { return x < 0 ? -1 : root2(x, 1); }
```

Two functions:

auxiliary root2 needs two parameters (also approximation)

for user: root defined as required: only one parameter

returns -1 for negative numbers (error code)

Recall: this form is *tail recursion*: recursive call is *last* computation.

Compiler can convert this to *iteration* (efficient).

Reading character-by-character



## 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!

⇒ 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 change in the state of the execution environment is called a *side effect* (e.g., reading, writing, assignment).

**WARNING!** Careful when using functions that have side effects, since they can interact (unexpectedly) through these effects.

⇒ write *side-effect free* functions whenever possible!

## Reading a natural number

The number is read as string of digits; base case: last digit  
Consider  $c_1c_2 \dots c_m$ , and the partial sequences  $c_1, c_1c_2, c_1c_2c_3, \dots$   
We have:  $r_0 = 0, r_k = 10 \cdot r_{k-1} + c_k, (k > 0)$ .

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

- when the char read is not a digit, return accumulated number  $r$
- else, recursive call with  $10 \cdot r + c$ , reading next character

**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'
- $\Rightarrow$  we adjust with `-'0'`:  $6 == '6' - '0'$

## 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)

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(10*r+(c-'0'), 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  $\Rightarrow$  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(10*r + (c-'0')) : r;
}
unsigned readnat(void) { return readnat_r(0); }
```

## 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(10*r + (c-'0'));
    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(10*r + (c-'0'))
    : (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()`

## Declaring variables

A *variable* is an object 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

⇒ 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  $f(x) = \dots$  and  $g(x) = \dots$  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)

## 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:

$expr$	$! expr$	$e_1 \ \&\& \ e_2$	$e_2$	$e_1 \    \ e_2$	$e_2$
0	1	0	$\neq 0$	0	$\neq 0$
$\neq 0$	0	0	0	0	1
		$\neq 0$	0	$\neq 0$	1
negation ! NOT		conjunction && AND		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 divisible by 400 which still are

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

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

## 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

⇒ 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

⇒ can naturally write  $x < y + z \ \&\& \ y < z + x$

## 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  $g$ )

# 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!

<code>n + 1</code>	<code>sqr(x)</code>	<code>toupper(c)</code>	<i>compute</i> , DON'T change!
<code>n = n + 1</code>	<code>x = sqr(x)</code>	<code>c = toupper(c)</code>	<i>change</i>

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

## From recursion to iteration

Recall: reversing a number

	rev( n ,	r )
	<hr/>	
	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*

$n/10, \quad 10*r+n\%10$

*condition* controls repetition (call) or termination (base case)

$n == 0$

one *function* for each repetitive computation

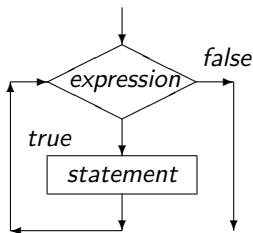
## Iteration. The `while` loop (initial test)

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.

⇒ 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)
```

```
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)
```

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

```
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

```
#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
            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!

## 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

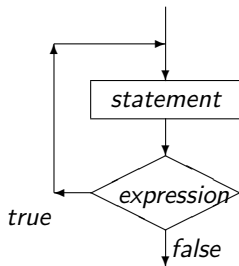
Ex: ignore \ followed by repeated text between braces  
\{text1}{text2}...

```
#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 to 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
```

## 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