Computer Programming

Recursion. Character input

Marius Minea

marius@cs.upt.ro

30 September 2014

Review

We solve a (computational) problem by writing a *function*.

the *answer* to the problem = the function *result*
produced with the statement `return` *expression* ;
(mandatory, else the function won't return a result!)
if no return value (e.g., just print) declare function type `void`

the *input data* = the function *parameters*
(based on which the result is computed)

Review: the purpose of a function

Computes a value

```
double discrim(double a, double b, double c)
{
  return b*b - 4*a*c;
}
```

Produces an effect (e.g. prints a message)

```
void myerr(int code) // void type: returns nothing
{
  printf("error code %d\n", code);
}
```

effect + value (computes + writes: several statements)

```
int sqr(int x)
{
  printf("Computing the square of %d\n", x);
  return x * x;
}
```

Review: structure of a simple program

```c
#include <stdio.h>    // if we need to read/write
#include <math.h>     // if we use math functions

// function definition: third side of a triangle
double thirdside(unsigned a, unsigned b, double phi)
{
  // the expression contains 2 function calls: cos, sqrt
  return sqrt(a*a + b*b - 2*a*b*cos(phi));
}

int main(void)
{
  // function call with values for its arguments
  printf("third side: %f\n", thirdside(3, 5, atan(1)));
  return 0;
}
```

Program structure: separating concerns

passing an argument is NOT *reading* from input

computing a value is NOT *writing* it

A function will typically NOT ask for input.
The smallest functions will *receive arguments* and *return results*

This allows them to be composed and used anywhere.

A function will typically NOT print its result, just return it.
(printing is inflexible: may want different format, language, etc.)

We might write wrapper functions that ask for input, then call the computation function.

We might also write display functions that get a value and print it.

Recursion: power by repeated squaring

Recursion = reduction to a *simpler* case of the *same* problem

Base case is simple enough for direct computation
(can / need no longer be reduced)

$$x^n = \begin{cases} 1 & n = 0 \\ (x^2)^{n/2} & n > 0 \text{ even} \\ x \cdot (x^2)^{n/2} & n > 0 \text{ odd} \end{cases}$$

```
double pow2(double x, unsigned n)
{
  return n == 0 ? 1
                : n % 2 == 0 ? pow2(x*x, n/2)
                             : x * pow2(x*x, n/2);
}
```

Let's follow the recursive calls

```
#include <stdio.h>

double pow2(double x, unsigned n)
{
  printf("base %f exponent %u\n", x, n);
  return n == 0 ? 1
                : n % 2 == 0 ? pow2(x*x, n/2)
                             : x * pow2(x*x, n/2);
}
int main(void)
{
  printf("5 to the 6th = %f\n", pow2(5, 6));
  return 0;
}
```

Each call halves the exponent $\Rightarrow 1 + \lceil \log_2 n \rceil$ calls

`pow2(5, 6)` $\rightarrow$ `pow2(25, 3)` $\rightarrow$ `pow2(625, 1)`

Elements of a recursive definition

1. *Base case*: no recursive call
= simplest case, defined directly
e.g. in sequences: initial term x_0 of the recurrence
the empty list (for a list of elements)

A missing base case is an *ERROR* $\Rightarrow$ recursion never stops!

2. the *recurrence relation*
defines a notion using a simpler case of the same notion

3. Proof/argument that recursion stops in a finite number of steps
(e.g. a nonnegative measure that decreases on each application
– for recurrent sequences: the index (smaller in the definition body but ≥ 0)
– for recursive objects: size (component objects are smaller)

Are the following definition recursive and correct ?

? $x_{n+1} = 2 \cdot x_n$
? $x_n = x_{n+1} - 3$
? $a^n = a \cdot a \cdot \ldots \cdot a$ (n times)
? a sentence is a sequence of words
? a sequence is the concatenation of two smaller sequences
? a string is a character followed by a string

A recursive definition must be *well formed* (conditions 1-3)

something cannot be defined only in terms of itself

one can only use other notions which are already defined

computation has to stop at some point

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 ϵ)

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

```c
#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 a_n)
{
  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)

Recursion in numbers: sequences of digits

A natural number (in base 10) can be defined/viewed recursively:
a number is a single digit
or: last digit preceded by *another number* (in base 10)

We can find the two parts using integer division (with remainder)

$n = 10 \cdot (n/10) + n\%10$	$1457 = 10 \cdot 145 + 7$
the last digit of n is $n\%10$	$1457\%10 = 7$
the number remaining in front is $n/10$	$1457/10 = 145$

Problems with a simple recursive solution:
sum of a number's digits
number of digits; largest/smallest digit, etc.

Solution: always *follow the structure of the recursive definition*
base case: give result for single-digit number
recurrence: combine last digit with result for the remaining number before it

How many digits in a number?

1, if number < 10
else, one digit more than the number without its last digit ($n/10$)

```
unsigned ndigits(unsigned n)
{
  return n < 10 ? 1 : 1 + ndigits(n / 10);
}
```

Alternative: use an *accumulator* for the digits already counted
– start counting from 1 (surely has one digit)
– if the number is single-digit, return the digits already counted
– else, count for $n/10$, accumulating current digit in parameter

```
unsigned ndigs2(unsigned n, unsigned r)
{
  return n < 10 ? r : ndigs2(n / 10, r + 1);
}
```

Need function with only one parameter: wrap auxiliary function (called with starting value 1: single-digit number)

```
unsigned ndig(unsigned n) { return ndigs2(n, 1); }
```

Largest digit in a number

base case: single-digit number (digit is also max)
else, max of last digit and result for the remaining number

```
unsigned max(unsigned a, unsigned b) { return a > b ? a : b; }
unsigned maxdigit(unsigned n)
{
  return n < 10 ? n : max(n%10, maxdigit(n/10));
}
```

Variant with accumulator: maximal digit seen so far: `md`
– if the number is zero, return the maximum so far: `md`
– else, continue with maximum of last digit and previous max

```
unsigned maxdig2(unsigned n, unsigned md)
{
  return n == 0 ? md : maxdig2(n/10, max(md, n%10));
}
unsigned maxdig(unsigned n) { return maxdig2(n/10, n%10); }
```

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` $\geq$ 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 betweeen (single) *quotes* `' '`
They are *integer values*. In expressions: implicitly converted to `int`
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

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)

`getchar()` needs to return `int`, not `char` to also encompass `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

Writing a character: `putchar`

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

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

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

Reading a natural number

The number is read as string of digits; base case: last digit

Consider $c_1c_2 \ldots c_m$, and the partial sequences c_1, c_1c_2, $c_1c_2c_3$, ...

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;
}
```

As a final solution, we write a function without auxiliary parameters
`unsigned readnat(void) { return readnat_rc(0, getchar()); }`
Note: no error checking; consumes first character that is not a digit

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).

Combining functions that have side effects requires a lot of care, since they also interact through these effects.
⇒ write side-effect free functions whenever possible!

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.

`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); }
```

Reading an integer

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()`

Comma , is the *sequencing operator* for expressions: *expr1* , *expr2*
expr1 is evaluated, ignored; the expression's result is that of *expr2*

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

About variables

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)

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)