Computer Programming

Introduction. Recursion

Marius Minea

marius@cs.upt.ro

http://www.cs.upt.ro/~marius/curs/cp/

24 September 2013

The C programming language

developed in 1972 at *AT&T Bell Laboratories* by Dennis Ritchie
together with the UNIX operating system and its tools
   (C first developed under UNIX, then UNIX was rewritten in C)
Brian Kernighan, Dennis Ritchie: The C Programming Language (1978)

Mature language, but still evolving
   ANSI C standard, 1989 (American National Standards Institute)
   then ISO 9899 standard (versions: C90, C99, C11 - current)

The C programming language

developed in 1972 at *AT&T Bell Laboratories* by Dennis Ritchie
together with the UNIX operating system and its tools
    (C first developed under UNIX, then UNIX was rewritten in C)
Brian Kernighan, Dennis Ritchie: The C Programming Language (1978)

Mature language, but still evolving
    ANSI C standard, 1989 (American National Standards Institute)
    then ISO 9899 standard (versions: C90, C99, C11 - current)

Why use C?
    versatile: direct access to data representation, freedom in
working with memory, good hardware interface
    mature, large code base (libraries for many purposes)
    efficient: good compilers that generate compact, fast code
WARNING: very easy to make *errors* !

Computations, functions, and programs

A program
  reads *input data*
  processes them (through (mathematical) *computations*)
  writes (produces) *results*

Computations, functions, and programs

A program
  reads input data
  processes them (through (mathematical) *computations*)
  writes (produces) *results*

In mathematics, computations are expressed by *functions*:
  we *know* predefined functions (sin, cos, etc.)
  we *define* new functions (for the given problem)
  we *combine* functions into more complex computations

In programming, we use functions in a similar way.

Think about functions you can *reuse* !

Splitting into functions helps *manage complexity* !

Functions in mathematics and C

Squaring for integers:

$$sqr : \mathbb{Z} \to \mathbb{Z}$$

$$sqr(x) = x \cdot x$$

function function parameter
type   name   type and name

```c
int sqr(int x)
{
    return x * x;
}
```

Functions in mathematics and C

Squaring for integers:

	function type	function name	parameter type and name

$$sqr : \mathbb{Z} \to \mathbb{Z}$$

```
int sqr(int x)
{
```

$$sqr(x) = x \cdot x$$

```
    return x * x;
}
```

A function *definition* contains:
  the function *header*, specifying: the type (range) of function values (int), function name (`sqr`) and parameters (the integer `x`)
  the function *body*, within { }: here, the (`return`) *statement*, with an *expression* that gives the function value from its parameters

There are precise *rules* for writing in the language (the *syntax*):
  language elements are written in a given *order*;
  separators are used to precisely delimit them: ( )   ;   { }

Another function

Squaring for *reals*:

$$sqrf : \mathbb{R} \to \mathbb{R}$$

$$sqrf(x) = x \cdot x$$

```
float sqrf(float x)
{
  return x * x;
}
```

Another function domain and range (reals) $\Rightarrow$ a different function
  even the * operator is now defined on a different set (type)
To distinguish it from sqr in the same program, it needs a
different name.

Another function

Squaring for *reals*:

$$sqrf : \mathbb{R} \rightarrow \mathbb{R}$$

$$sqrf(x) = x \cdot x$$

```
float sqrf(float x)
{
    return x * x;
}
```

Another function domain and range (reals) $\Rightarrow$ a different function
  even the * operator is now defined on a different set (type)

To distinguish it from sqr in the same program, it needs a
different name.

int and float denote *types*.

A *type* is a *set of values* together with a *set of operations* allowed
for these values.

For reals, it is preferable to use the type double (double precision)
  (used by library functions: sin, cos, exp, etc.)

Integers and reals

Numeric types differ in C and mathematics.

In mathematics: $\mathbb{Z} \subset \mathbb{R}$, both are infinite, $\mathbb{R}$ is uncountable.

In C: int, float, double are finite (have limited range);
reals have finite precision.

Important to remember this! (overflows, precision loss)

The type of numeric *constants* depends on their writing
  2 is an integer, 2.0 is a real
  scientific notation for reals: 1.0e-3 instead of 0.001
  writing 1.0 or 1. is equivalent, same for 0.1 and .1

Mathematical operators

`+ - * /`

Multiplication is written explicitly !
we can't write $2x$, but $2 * x$ (or $x * 2$)

Some operators have different meanings (and results!) for integers and reals:

Integer division has an *integer result* !!! (division with remainder)
 7 / 2 is 3, but 7.0 / 2.0 is 3.5
−7 / 2 is −3, likewise −(7 / 2)
(integer division truncates towards zero)

Mathematical operators
`+ - * /`

> Multiplication is written explicitly !
> we can't write 2*x*, but 2 * x (or x * 2)

Some operators have different meanings (and results!) for integers and reals:

Integer division has an *integer result* !!! (division with remainder)
 7 / 2 is 3, but 7.0 / 2.0 is 3.5
-7 / 2 is -3, likewise -(7 / 2)
(integer division truncates towards zero)

The *modulo* operator % is only defined for integers.

9 / 5 = 1	9 % 5 = 4	9 / -5 = -1	9 % -5 = 4
-9 / 5 = -1	-9 % 5 = -4	-9 / -5 = 1	-9 % -5 = -4

The sign of the remainder is the same as the sign of the dividend.

Rule for integer division: a = a / b * b + a % b

Some terminology

Keywords: have a predefined meaning (cannot be changed)
Examples: statements (`return`), types (`int`, `float`, `double`)

Some terminology

Keywords: have a predefined meaning (cannot be changed)
Examples: statements (`return`), types (`int`, `float`, `double`)

Identifiers (e.g. `sqr`, `x`) chosen by the programmer to name functions, parameters, variables, etc.

An identifier is a sequence of characters comprised of letters (upper and lower case), underscore `_` and digits which does not start with a digit and is not a keyword.
Examples: `x3`, `a12_34`, `_exit`, `main`, `printf`, `int16_t`

Some terminology

Keywords: have a predefined meaning (cannot be changed)
Examples: statements (`return`), types (`int`, `float`, `double`)

Identifiers (e.g. `sqr`, `x`) chosen by the programmer to name functions, parameters, variables, etc.

An identifier is a sequence of characters comprised of letters (upper and lower case), underscore _ and digits which does not start with a digit and is not a keyword.
Examples: `x3`, `a12_34`, `_exit`, `main`, `printf`, `int16_t`

Constants
integer: `-2`; floating point: `3.14`; character: `'a'`, string: `"a"`

Some terminology

Keywords: have a predefined meaning (cannot be changed)
Examples: statements (`return`), types (`int`, `float`, `double`)

Identifiers (e.g. `sqr`, `x`) chosen by the programmer to name functions, parameters, variables, etc.

An identifier is a sequence of characters comprised of letters (upper and lower case), underscore `_` and digits which does not start with a digit and is not a keyword.
Examples: `x3`, `a12_34`, `_exit`, `main`, `printf`, `int16_t`

Constants
integer: `-2`; floating point: `3.14`; character: `'a'`, string: `"a"`

Punctuation signs, with various meanings:
  `*` is an operator
  `;` terminates a statement
  parantheses `( )` around an expression or function parameters
  braces `{ }` group declarations or statements

Functions with several parameters

Example: the discriminant of a quadratic equation:
$a \cdot x^2 + b \cdot x + c = 0$

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

Between the parantheses ( ) of the function header there can be arbitrary comma-separated parameters, each with its own type.

Function call (function evaluation)

So far, we have only *defined* functions, without using them.

The value of a function can be *used* in an expression.

Syntax: like in mathematics: *function(param, param, · · · , param)*

Example: in the discriminant, we could use the `sqrf` function:

```
    return sqrf(b) - 4 * a * c;
```

Function call (function evaluation)

So far, we have only *defined* functions, without using them.

The value of a function can be *used* in an expression.

Syntax: like in mathematics: *function(param, param, ⋯, param)*

Example: in the discriminant, we could use the `sqrf` function:

```
    return sqrf(b) - 4 * a * c;
```

Or, using the previously defined `sqr` function we can define:

```
int cube(int x)
{
  return x * sqr(x);
}
```

Function call (function evaluation)

So far, we have only *defined* functions, without using them.

The value of a function can be *used* in an expression.

Syntax: like in mathematics: *function(param, param, $\cdots$, param)*

Example: in the discriminant, we could use the `sqrf` function:

```
    return sqrf(b) - 4 * a * c;
```

Or, using the previously defined `sqr` function we can define:

```c
int cube(int x)
{
  return x * sqr(x);
}
```

IMPORTANT: In C, any identifier must be *declared before use*
(we must know what it represents, including its type)
$\Rightarrow$ The above examples assume that `sqrf` and `sqr` are defined
before `discrim` and `cube` respectively in the program.

A first C program

```c
int main(void)
{
  return 0;
}
```

The smallest program: it does not do anything!

Any program contains the *main* function and is executed by calling it at program start. In main, other functions may be called.

Here, main does not have any parameters (void)
  void is a keyword for the empty type (without any element)

main returns an integer, interpreted as exit status by the operating system:
$0 =$ successful termination, $\neq 0$ is an error code

A commented program

```c
/* This is a comment */
int main(void) // comment to end of line
{
  /* This is a comment spanning several lines
     usually, the program code would be here */
  return 0;
}
```
Programs may contain comments, placed between /* and */
or starting with // until (and excluding) the end of the line
Comments are stripped by the preprocessor.
They have no effect on code generation or program execution.

A commented program

```c
/* This is a comment */
int main(void) // comment to end of line
{
  /* This is a comment spanning several lines
     usually, the program code would be here */
  return 0;
}
```

Programs may contain comments, placed between /* and */
or starting with // until (and excluding) the end of the line
Comments are stripped by the preprocessor.
They have no effect on code generation or program execution.

Programs *should be* commented
    so a reader can understand (including the writer, at a later time)
    as documentation (may specify functionality, restrictions, etc.)
    explain function parameters, result, local variables
    specify preconditions, postconditions, error behavior

Printing (writing)

```c
#include <stdio.h>
int main(void)
{
  printf("hello, world!\n"); // prints a text
  return 0;
}
```

printf (from "print formatted"): a standard library function

    is NOT a *statement* or a *keyword*

    is called here with one string parameter

    string constants are written with double quotes " "

\n denotes the newline character

Printing (writing)

```c
#include <stdio.h>
int main(void)
{
  printf("hello, world!\n"); // prints a text
  return 0;
}
```

printf (from "print formatted"): a standard library function
  is NOT a *statement* or a *keyword*
  is called here with one string parameter
  string constants are written with double quotes " "
\n denotes the newline character

The first line is a *preprocessing directive*, it includes the stdio.h
header file which contains the *declarations* of the standard
input/output functions

Declaration = type, name, parameters: needed to use the function

Implementation (compiled object code): in a *library* which is linked
at compile-time, loaded at execution time

Printing numbers

```c
#include <math.h>
#include <stdio.h>
int main(void)
{
  printf("cos(0) = ");
  printf("%f", cos(0));
  return 0;
}
```

```c
#include <stdio.h>
int sqr (int x) { return x * x; }
int main(void)
{
  printf("2 times -3 squared is");
  printf("%d", 2 * sqr(-3));
  return 0;
}
```

To print the value of an expression, `printf` takes two arguments:
– a character string (format specifier):
  `%d` or `%i` (*decimal* integer), `%f` (*floating point*)
– the expression, whose type must be compatible with the specified one (programmer must check ! compiler may warn or not)

Sequencing: in function, statements are executed in textual order
But: `return` statement ends function execution (no further statement is executed)

Functions defined by cases

$$abs : \mathbb{Z} \to \mathbb{Z} \qquad abs(x) = \begin{cases} x & x \geq 0 \\ -x & \text{otherwise } (x < 0) \end{cases}$$

The function value is not given by a *single* expression, but by one of two different expressions (x or -x), depending on a condition $(x \geq 0)$

$\Rightarrow$ need a language construct that to *decide* which expression to evaluate, based on a *condition* (true/false)

The conditional operator ? :

Syntax of *conditional expression*: *condition* ? *expr1* : *expr2*
– if the condition is true, only *expr1* is evaluated, its value becomes the result of the entire expression
– if the condition is false, only *expr2* is evaluated and its value becomes the value of the expression

```c
int abs(int x)
{
  return x >= 0 ? x : -x;    // unary minus operator
}
```

Comparison operators: == (equality), != (different), <, <=, >, >=

IMPORTANT! The equality test in C is == and not simple = !!!

Note: abs exists as standard function, declared in `stdlib.h`

Functions defined by several cases

$$sgn : \mathbb{Z} \to \{-1, 0, 1\} \qquad sgn(x) = \begin{cases} -1 & x < 0 \\ 0 & x = 0 \\ 1 & x > 0 \end{cases}$$

The conditional operator has only *one* condition, and *two* branches
But: either of the expressions can be arbitrarily complex
$\Rightarrow$ must decompose the decision based on the value of x
$\Rightarrow$ *decompose into smaller subproblems*: key in problem solving

We rewrite the function with a single decision at any given point:

$$sgn(x) = \begin{cases} \text{if } x < 0 & -1 \\ \text{else } (x \geq 0) & \begin{cases} \text{if } x = 0 & 0 \\ \text{else } (x > 0) & 1 \end{cases} \end{cases}$$

Writing the case-based function in C

$$sgn(x) = \begin{cases} \text{if } x < 0 & -1 \\ \text{else } (x \geq 0) & \begin{cases} \text{if } x = 0 & 0 \\ \text{else } (x > 0) & 1 \end{cases} \end{cases}$$

```
int sgn (int x)
{
  return x < 0 ? -1
             : x == 0 ? 0 : 1;
}
```

We can group arbitrarily many conditional operators   ?   :
expr1 and *expr2* can be in turn conditional expressions
A correctly written expression has a : for any ?
(think of : as linking a *pair* of answers)

Decomposing into simpler problems

The minimum of two numbers is easily written:

```
double min2(double x, double y)
{
  return x < y ? x : y;
}
```

For the minimum of *three* numbers, the comparisons multiply:

$$
min3(x, y, z) = \begin{cases} \text{if } x < y & \begin{cases} \text{if } x < z & \mathbf{x} \\ \text{else } (x \geq z) & \mathbf{z} \end{cases} \\ \\ \text{else } (x \geq y) & \begin{cases} \text{if } y < z & \mathbf{y} \\ \text{else } (y \geq z) & \mathbf{z} \end{cases} \end{cases}
$$

Decomposing into simpler problems

The minimum of two numbers is easily written:

```
double min2(double x, double y)
{
  return x < y ? x : y;
}
```

For the minimum of *three* numbers, the comparisons multiply:

$$
min3(x, y, z) = \begin{cases} \text{if } x < y & \begin{cases} \text{if } x < z & \mathbf{x} \\ \text{else } (x \geq z) & \mathbf{z} \end{cases} \\[2ex] \text{else } (x \geq y) & \begin{cases} \text{if } y < z & \mathbf{y} \\ \text{else } (y \geq z) & \mathbf{z} \end{cases} \end{cases}
$$

We notice the structure of `min2` is repeated $\Rightarrow$ can do it simpler:
The result is the minimum between the minimum of the first two
numbers and the third. $\Rightarrow$ just apply `min2` twice!

```
double min3(double x, double y, double z)
{
  return min2(min2(x, y), z); // or min2(x, min2(y,z))
}
```

Understanding the function call

This program computes $x^6 = (x \cdot x^2)^2$

```c
#include <stdio.h>
int sqr(int x)
{
  printf("the square of %d is %d\n", x, x*x);
  return x * x;
}
int main(void)
{
  printf("2 to the 6th is %d\n", sqr(2 * sqr(2)));
  return 0;
}
```

What is the order of printed statements ?

```
the square of 2 is 4
the square of 8 is 64
2 to the 6th is 64
```

Understanding the function call

In C, parameters to functions are passed *by value*.

all function arguments are *evaluated* (their value is computed)

values are assigned to the *formal parameters* (names from the function definition)

only then function is *called* and executes with these values

Understanding the function call

In C, parameters to functions are passed *by value*.
   all function arguments are *evaluated* (their value is computed)
   values are assigned to the *formal parameters* (names from the function definition)
   only then function is *called* and executes with these values

The program starts executing `main`. The first statement:
```
printf("2 to the 6th is %d\n", sqr(2 * sqr(2)));
```
Before doing the call, `printf` needs the *values of its arguments*
  first argument: the value is known (a *string constant*)
  second argument: need to call `sqr(2 * sqr(2))`
    BUT: the outer `sqr` also needs the value of its argument
    `2 * sqr(2)` $\Rightarrow$ need to call `sqr(2)` first
$\Rightarrow$ call order: first `sqr(2)`, then `sqr(8)`, then `printf`

Errors in understanding function evaluation

C does NOT do the following (other languages might...)

Functions do NOT start execution without computer arguments
  `printf` would print `2 to the 6th is `, then need the value
  it would call the outer `sqr` that writes ` the square of`,
then would need x
  it would call `sqr(2)`, write `the square of 2 is 4`, return 4,
etc.

Errors in understanding function evaluation

C does NOT do the following (other languages might...)

Functions do NOT start execution without computer arguments
  `printf` would print `2 to the 6th is`, then need the value
  it would call the outer `sqr` that writes `the square of`,
then would need x
  it would call `sqr(2)`, write `the square of 2 is 4`, return 4,
etc.

Function parameters are NOT substituted with expressions
`printf` would call the outer `sqr` with the *expression* `2 * sqr(2)`
`sqr(2)` would be called twice for `(2*sqr(2))*(2*sqr(2))`

$\Rightarrow$ in C, a function computes with *values*, never with *expressions*

Recursion

Recursion: definition, examples

From mathematics, we know recurrence relations for *sequences*:

arithmetic sequence: $\begin{cases} x_0 = b & \text{(i.e.: } x_n = b \text{ for } n = 0) \\ x_n = x_{n-1} + r & \text{for } n > 0 \end{cases}$

Example: $1, 4, 7, 10, 13, \ldots$ ($b = 1$, $r = 3$)

Recursion: definition, examples

From mathematics, we know recurrence relations for *sequences*:

arithmetic sequence: $\begin{cases} x_0 = b & \text{(i.e.: } x_n = b \text{ for } n = 0) \\ x_n = x_{n-1} + r & \text{for } n > 0 \end{cases}$

Example: $1, 4, 7, 10, 13, \ldots$ ($b = 1$, $r = 3$)

geometric sequence: $\begin{cases} x_0 = b & \text{(i.e.: } x_n = b \text{ for } n = 0) \\ x_n = x_{n-1} \cdot r & \text{for } n > 0 \end{cases}$

Example: $3, 6, 12, 24, 48, \ldots$ ($b = 3$, $r = 2$)

x_n is not computed *directly*, but *step by step*, using x_{n-1}.

Recursion: definition, examples

From mathematics, we know recurrence relations for *sequences*:

arithmetic sequence: $\begin{cases} x_0 = b & \text{(i.e.: } x_n = b \text{ for } n = 0) \\ x_n = x_{n-1} + r & \text{for } n > 0 \end{cases}$

Example: $1, 4, 7, 10, 13, \ldots$ ($b = 1$, $r = 3$)

geometric sequence: $\begin{cases} x_0 = b & \text{(i.e.: } x_n = b \text{ for } n = 0) \\ x_n = x_{n-1} \cdot r & \text{for } n > 0 \end{cases}$

Example: $3, 6, 12, 24, 48, \ldots$ ($b = 3$, $r = 2$)

x_n is not computed *directly*, but *step by step*, using x_{n-1}.

A notion is *recursive* if it is *used in its own definition*.

Exercise: write recurrences for: C_n^k, Fibonacci sequence, $\ldots$

Recursion: definition, examples
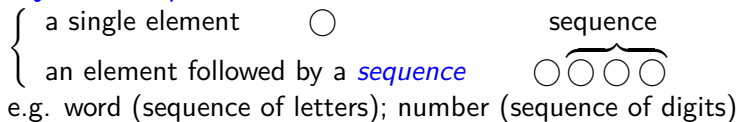
Recursion is fundamental in computer science:
it reduces a problem to a simpler case of the *same* problem

objects: a *sequence* is

$\begin{cases} \text{a single element} \\ \text{an element followed by a } \textit{sequence} \end{cases}$

sequence

e.g. word (sequence of letters); number (sequence of digits)
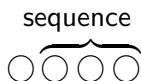
Recursion: definition, examples

Recursion is fundamental in computer science:
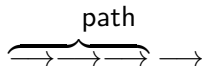it reduces a problem to a simpler case of the *same* problem

objects: a *sequence* is

$\begin{cases} \text{a single element} \\ \text{an element followed by a } \textit{sequence} \end{cases}$

e.g. word (sequence of letters); number (sequence of digits)

actions: a *path* is

$\begin{cases} \text{a step} \\ \text{a } \textit{path} \text{ followed by a step} \end{cases}$

e.g. traversing a path in a graph

Recursion: definition, examples

Recursion is fundamental in computer science:
it reduces a problem to a simpler case of the *same* problem

objects: a *sequence* is
$\begin{cases} \text{a single element} \\ \text{an element followed by a } \textit{sequence} \end{cases}$

$\bigcirc$ sequence

$\bigcirc\overbrace{\bigcirc\bigcirc\bigcirc}$

e.g. word (sequence of letters); number (sequence of digits)

actions: a *path* is
$\begin{cases} \text{a step} \quad \longrightarrow \\ \text{a } \textit{path} \text{ followed by a step} \end{cases}$

path

$\overbrace{\longrightarrow\longrightarrow\longrightarrow} \longrightarrow$
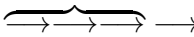
e.g. traversing a path in a graph

An *expression*:
$\begin{cases} \text{number (7)} \\ \text{identifier (x)} \\ \textit{expression} + \textit{expression} \\ \textit{expression} - \textit{expression} \\ (\textit{ expression }), \text{ etc} \end{cases}$

Example: power function

$$x^n = \begin{cases} 1 & n = 0 \\ x \cdot x^{n-1} & \text{otherwise } (n > 0) \end{cases}$$

```c
#include <stdio.h>
double pwr(double x, unsigned n)
{
 return n==0 ? 1 : x * pwr(x, n-1);
}
int main(void)
{
  printf("-2 raised to 3 = %f\n", pwr(-2.0, 3));
  return 0;
}
```

Example: power function

$$x^n = \begin{cases} 1 & n = 0 \\ x \cdot x^{n-1} & \text{otherwise } (n > 0) \end{cases}$$

```c
#include <stdio.h>
double pwr(double x, unsigned n)
{
 return n==0 ? 1 : x * pwr(x, n-1);
}
int main(void)
{
  printf("-2 raised to 3 = %f\n", pwr(-2.0, 3));
  return 0;
}
```

`unsigned`: type of nonnegative integers (natural numbers)

The *header* of `pwr` is a *declaration* of the function
so it can be used in its own function body (recursive call)

Even if we write `pwr(-2, 3)`, `-2` (int) will be *converted* to float
(the type declared for each parameter is known)

The mechanism of a recursive call

The `pwr` function does two computations:
– a *test* (n == 0 ? *base case* ?) if so, return 1
– else, a multiply; the right operand requires a *new recursive call*

```
pwr(5, 3)
      call↓↑125
          5 * pwr(5, 2)
                call↓↑25
                    5 * pwr(5, 1)
                          call↓↑5
                              5 * pwr(5, 0)
                                    call↓↑1
                                        1
```

The mechanism of a recursive call

In the recursive computation of the power function:

Evert call makes *a new call*, until the base case it reached

Every call executes *the same code*, but with *other data*
(own values for parameters)

When reaching the base case, all started calls are still *unfinished*
(each has to perform the multiplication with the result of the call)

Returning is done *in opposite order* of the calls
(call with exponent 0 returns, then the one with exponent 1, etc.)