Computer Programming

Introduction

Alexandru IOVANOVICI
based on notes of Marius Minea

iovanalex@cs.upt.ro

http://bit.do/uptcp2019

September 2020

Course goals

Learn programming fundamentals

no prior knowledge needed
for those who know, hopefully learn more

Know one language well

imperative programming in C
some insight into alternatives

Write clean, correct, secure code

handle errors
test your code
think of corner cases

Last year

Some students failed

13 students failed the lab

17 students failed the midterm

(out of 52 attending)

52% passed the subject

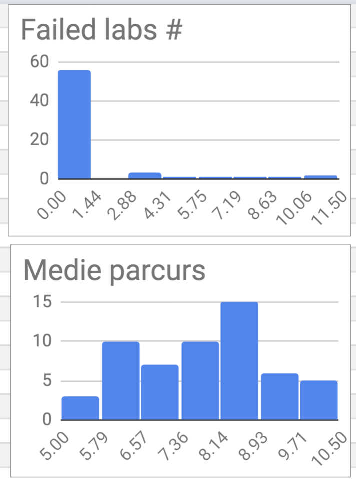


Figure: Grades

This year... administrative stuff

SARS-CoV-2/COVID19 Grading

Exam: midterm (code on PC)+final exam (quiz+code on PC)

Lab grade: 4-5 grades over 14 weeks;

Activity: lecture participation and interventions, QA

Assignments/homeworks: deadlines and online followup

Office hours

Discuss problems you have with the subject or code you have tried and did not make it work

Code of conduct: Don't steal, cheat or lie

If you discuss a solution with a colleague you still can't have the same code with his... you are both humans;

Use *official* email and/or cv.upt.ro platform: traceability.

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 (ver.: C90, C99, C11, C18 - current, C2x)

Why use C?

versatile: direct access to data representation, freedom in working with memory, good hardware interface

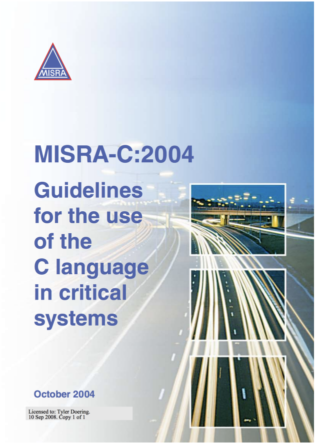
mature, large code base (libraries for many purposes)

efficient: good compilers that generate compact, fast code

WARNING: very easy to make *errors* !

MISRA C: Why C?

- Only available language (ASM?)
- high speed low-level I/O
- smaller and less RAM-intensive code
- porting/reusing code from other projects
- auto-generated code

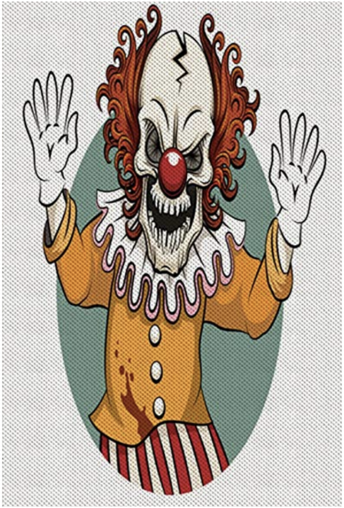



```
main:
 .LFB0:
 .cfi_startproc
 endbr64
```

MISRA C: Language insecurities

- The programmer makes mistakes
- The programmer misunderstands the language
- The compiler doesn't do what the programmer expects
- The compiler contains errors
- Run-time errors

The use of C for safety-related systems!!!



50000ft. view

- Functions. Input/Output operations. Character level stream processing
- Bit-wise operations and internal representation of data
- Recursion
- Arrays and Pointers
- Strings... as arrays of characters ;)
- Packed data: structures
- Dynamic memory allocation
- Files
- Testing, debugging and safe programming techniques.

Prove us you know about all/some of those and we'll talk...

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.

Functions are the core of programming

Programs are *structured* into functions (methods, procedures)

Splitting into functions helps *manage complexity*
NOT one huge piece of code

Functions can be *reused*, making development efficient

Functions are core for the *functional programming* paradigm
computation is function *evaluation*, not ~~assignment~~

Functions are core to defining what is *computable*
(recursive functions, lambda calculus)

Functions in mathematics and C

Squaring for integers:

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

$sqr(x) = x \cdot x$

function type | function name | parameter type and name

```
int sqr(int 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: `( ) ; { }`

Functions in C vs. other languages

concrete syntax: detail
(keywords, punctuation)

vs.

abstract syntax: essence
(language elements/concepts)

function type | function name | parameter type and name

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

Essence:

- *names*: function, parameter(s)
- *types*: of parameter(s) and return value
  - cannot omit (some languages: can infer types)
  - one precise type (some languages: polymorphism, overloading)
- *expression* (what is computed)

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 $\star$ operator is now defined on a different set (type)
Need different name to distinguish from `sqr` in the same program

`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 math: $\mathbb{Z} \subset \mathbb{R}$, both are *infinite*, $\mathbb{R}$ is dense/uncountable.

In C: **`int`**, **`float`**, **`double`** are *finite!*
both have *limited range*, reals have *finite precision*

Important to remember this! (overflows, precision loss)
default math functions use **`double`**, you should too!

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`

`1.0` and `1.` are equivalent, same for `0.1` and `.1`

Mathematical operators

`+` `-` `*` `/`

Multiplication must be written explicitly !
we can't write 2*x*, but `2 * x` (or `x * 2`)

Some operators have different meanings for integers and reals
and different results!

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`

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

$\Rightarrow$ sign of remainder is same as sign of dividend.

Some terminology... to be known and used

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: $a \cdot x^2 + b \cdot x + c = 0$

```
double discrim(double a, double b, double 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.

  must give type for each parameter, even if types are the same

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

```
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 int, interpreted as exit status by operating system: 0 = successful termination, $\neq 0$ is an error code

`return 0;` at the end of main is optional (if end brace is reached, 0 is returned by default; still most programs have it explicit).

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

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

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

```
#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; 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 code is executed)

Printing

We cannot print a number like this: ~~`printf(5)`~~

We can write `printf("5")` but this means printing a *string*
(although the effect is the same: one character printed)

The first argument of `printf` must always be a string
with or without format specifiers (special characters)

Understanding how functions work

Two distinct things:

function *definition*: `int sqr(int x) { ... }`

function *call*: `sqr(2)`, `sqr(a)`, etc.

Function definitions use *names* (of parameters, variables, etc.)

Function calls work with *values* (2, the *value* of a, etc.)
(they do *not* compute with symbolic expressions)

Understanding the function call

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

```
#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
```

C uses call by value

In C, function arguments are passed *by value*.

There is *NO* such concept as *reference* !!!

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

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

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

This type of argument passing is named *call by value*.

Function call example

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.

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))`

⇒ in C, a function computes with *values*, never with *expressions*