

# Writing Program in C

## Expressions and Control Structures (Statements and Loops)

Jan Faigl

Department of Computer Science  
Faculty of Electrical Engineering  
Czech Technical University in Prague

Lecture 02

B3B36PRG – Programming in C

## Overview of the Lecture

- Part 1 – Expressions
  - Expressions – Literals and Variables
  - Expressions – Operators
  - Associativity and Precedence
  - Assignment
- Part 2 – Control Structures: Selection Statements and Loops
  - Statements and Coding Styles
  - Selection Statements
  - Loops
  - Conditional Expression
- Part 3 – Assignment HW 01

*K. N. King: chapter 4 and 20*

*K. N. King: chapters 5 and 6*

## Part I

### Part 1 – Expressions

## Expressions

- **Expression** prescribes calculation value of some given input.
- Expression is composed of **operands**, **operators**, and **brackets**.
- Expression can be formed of
  - literals,
  - variables,
  - constants,
  - unary and binary operators,
  - function call,
  - brackets.
- The order of operation evaluation is prescribed by the operator **precedence** and **associativity**.

```
10 + x * y // order of the evaluation 10 + (x * y)
10 + x + y // order of the evaluation (10 + x) + y
```

*\* has higher priority than +  
+ is associative from the left-to-right*

- The evaluation order can be prescribed by **fully parenthesized expression**.

*Simply: If you are not sure, use brackets.*

## Literals – Integer and Rational

- Integer values are stored as one of the integer type (keywords): `int`, `long`, `short`, `char` and their `signed` and `unsigned` variants. *Further integer data types are possible.*
- Rational numbers (data types `float` and `double`) can be written with floating point – `13.1`; or with mantissa and exponent – `31.4e-3` or `31.4E-3`. *Scientific notation*
- Floating point numeric types depends on the implementation (usually as IEEE-754-1985).

Integer literals (values)		Rational literals
Decimal	123 450932	<ul style="list-style-type: none"> <li><code>double</code> – by default, if not explicitly specified to be another type;</li> <li><code>float</code> – suffix <code>F</code> or <code>f</code>; <code>float f = 10.f;</code></li> <li><code>long double</code> – suffix <code>L</code> or <code>l</code>. <code>long double ld = 10.11;</code></li> </ul>
Hexadecimal	0x12 0xFAFF (starts with <code>0x</code> or <code>0X</code> )	
Octal	0123 0567 (starts with <code>0</code> )	
<code>unsigned</code>	12345U (suffix <code>U</code> or <code>u</code> )	
<code>long</code>	12345L (suffix <code>L</code> or <code>l</code> )	
<code>unsigned long</code>	12345ul (suffix <code>UL</code> or <code>ul</code> )	
<code>long long</code>	12345LL (suffix <code>LL</code> or <code>ll</code> )	

Without suffix, the literal is of the type `int`.

## Literals – Enumeration (Constants of the Enumerated Type)

- The default values of the enumeration type constants start from 0 and the values can be explicitly prescribed.
- Without specifying a value, the next element has a value one higher than the previous one.
- Enumeration type constants can have the same values.

```

1 enum {          1 enum {          1 enum {
2   WHITE,        2   SPADES = 10,        2   NUMBER_1 = 1,
3   BLACK,        3   CLUBS, // the value is 11  3   NUMBER_2 = 2,
4   RED,          4   HEARTS = 15,        4   NUMBER_3 = 1,
5   GREEN,        5   DIAMONDS = 13        5   NUMBER_4, // the value is 2
6 };              6 };              6 };
    
```

*The enumeration values are usually written in uppercase.*

- Type – enumerated constant is the `int` type.
  - Value of the enumerated literal can be used in loops.

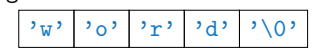
```

1 enum { WHITE = 0, BLACK, RED, GREEN, BLUE, NUM_COLORS };
3 for (int color = WHITE; color < NUM_COLORS; ++color) {
4     ...
5 }
    
```

`lec02/enum.c`

## Literals – Characters and Text Strings

- Character literal is single (or multiple) character in apostrophe.
  - `'A'`, `'B'` or `'\n'`
- Value of the single character literal is the ASCII code of the character.
  - `'0' ~ 48`, `'A' ~ 65`  
*Value of character out of ASCII (greater than 127) depends on the compiler.*
- Type of the character constant (literal).
  - Character constant is the `int` type.**
- Text string is a sequence of characters enclosed in quotation marks.
  - String literals separated by white spaces are joined to single one.
  - "A string with the end of line `\n`".
  - "A string literal" "with the end of the line `\n`" is concatenate into "A string literal with end of the line `\n`"
- String literal is stored in the array of the type `char` terminated by the null character `'\0'`. A string literal "word" is stored as



*The size of the array must be +1 item longer to store `\0`!*

```

char c = '8'; // Letter of the digit 8
int v = c - '0'; // Conversion to int
           value 8

char a = '0'; // Test a letter is upper
           case
_Bool upper = (a >= 'A' && a <= 'Z');
char i = '5'; // Test a letter is a digit
_Bool digit = (i >= '0' && i <= '9');
    
```

## Variable Definition

- The variable definition has a general form **declaration-specifiers variable-identifier;**
- Declaration specifiers are following.
  - Storage classes:** at most one of the `auto`, `static`, `extern`, `register`;
  - Type quantifiers:** `const`, `volatile`, `restrict`;
  - Type specifiers:** `void`, `char`, `short`, `int`, `long`, `float`, `double`, `signed`, `unsigned`.  
In addition, `struct` and `union` type specifiers can be used. Finally, own types defined by `typedef` can be used as well. *How many keywords are covered?*

```

float f = 10.1f; // float variable initialized by float literal
const double pi = 3.14; //const double variable initialized to 3.14
unsigned char v = 255; //one byte integer variable with the full range 0..255
const unsigned long l = 1001; //constant long integer variable initialized by long literal
int i; // i variable of the common C integer type int that is not initialized
    
```

## Operators

- Operators are selected characters (or sequences of characters) dedicated for writing expressions.
- Five types of **binary operators** can be distinguished.
  - Arithmetic** operators – additive (addition/subtraction) and multiplicative (multiplication/division);
  - Relational** operators – comparison of values (less than, greater than, ...);
  - Logical** operators – logical **AND** and **OR**;
  - Bitwise** operators – bitwise **AND**, **OR**, **XOR**, bitwise shift (left, right);
  - Assignment operator** **=** – a variables (l-value) is on its left side.
- Unary operators
  - Indicating positive/negative value: **+** and **-**.  
*Operator – modifies the sign of the expression.*
  - Modifying a variable : **++** and **--**.
  - Logical negation: **!**.
  - Bitwise negation: **~**.
- Ternary operator – conditional expression **? :**.

## Basic Arithmetic Expressions

- For an operator of the numeric types **int** and **double**, the following operators are defined.
  - Unary operator for changing the sign **-**;
  - Binary addition **+** and subtraction **-**;
  - Binary multiplication **\*** and division **/**.
- For integer operator, there is also
  - Binary module (integer remainder) **%**.
- If both operands are of the same type, the results of the arithmetic operation is the same type.
- In a case of combined data types **int** and **double**, the data type **int** is converted to **double** and the results is of the **double** type.

*Implicit type conversion.*

## Variables, Assignment Operator, and Assignment Statement

- Variables are defined by the type and name.
    - Name of the variable is in lowercase.
    - Multi-word names can be written with underscore **\_**.
    - Each variable is defined at a new line.  
*Or we can use **CamelCase**. That is our coding style choice.*
- ```
int n;
int number_of_items;
int numberOfItems;
```
- Assignment is setting the value to the variable, i.e., the value is stored at the memory location referenced by the variable name.
  - Assignment operator  $\langle \text{l-value} \rangle = \langle \text{expression} \rangle$   
*Expression is literal, variable, function calling, ...*
    - The side is the so-called **l-value – location-value, left-value**  
*It must represent a memory location where the value can be stored.*
    - Assignment is an expression and we can use it everywhere it is allowed to use the expression of the particular type.
  - Assignment statement is the assignment operator **=** and **;**.

## Example – Arithmetic Operators 1/2

```
1 int a = 10;
2 int b = 3;
3 int c = 4;
4 int d = 5;
5 int result;
7 result = a - b; // subtraction
8 printf("a - b = %i\n", result);
10 result = a * b; // multiplication
11 printf("a * b = %i\n", result);
13 result = a / b; // integer division
14 printf("a / b = %i\n", result);
16 result = a + b * c; // priority of the operators
17 printf("a + b * c = %i\n", result);
19 printf("a * b + c * d = %i\n", a * b + c * d); // -> 50
20 printf("(a * b) + (c * d) = %i\n", (a * b) + (c * d)); // -> 50
21 printf("a * (b + c) * d = %i\n", a * (b + c) * d); // -> 350
    lec02/arithmetric_operators.c
```

### Example – Arithmetic Operators 2/2

```

1 #include <stdio.h>
3 int main(void)
4 {
5     int x1 = 1;
6     double y1 = 2.2357;
7     float x2 = 2.5343f;
8     double y2 = 2;
10    printf("P1 = (%i, %f)\n", x1, y1);
11    printf("P1 = (%i, %i)\n", x1, (int)y1);
12    printf("P1 = (%f, %f)\n", (double)x1, (double)y1);
13    printf("P1 = (%.3f, %.3f)\n", (double)x1, (double)y1);
15    printf("P2 = (%f, %f)\n", x2, y2);
17    double dx = (x1 - x2); // implicit data conversion to float
18    double dy = (y1 - y2); // and finally to double
20    printf("(P1 - P2)=(%.3f, %0.3f)\n", dx, dy);
21    printf("|P1 - P2|^2=%.2f\n", dx * dx + dy * dy);
22    return 0;
23 }

```

lec02/points.c

### Integer Division

- The results of the division of the operands of the `int` type is the integer part of the division.  
*E.g., 7/3 is 2 and -7/3 is -2*
- For the integer remainder, it holds  $x\%y = x - (x/y) * y$ .  
*E.g., 7 % 3 is 1      -7 % 3 is -1      7 % -3 is 1      -7 % -3 is -1*
- **C99**: The result of the integer division of negative values is the value closer to 0.
  - It holds that  $(a/b)*b + a\%b = a$ .  
*For older versions of C, the results depends on the compiler.*

### Arithmetic Operators

- Operands of arithmetic operators can be of any arithmetic type.  
*The only exception is the operator for the integer remainder % defined for the int type.*
- |    |                |                      |                                                                |
|----|----------------|----------------------|----------------------------------------------------------------|
| *  | Multiplication | <code>x * y</code>   | Multiplication of x and y                                      |
| /  | Division       | <code>x / y</code>   | Division of x and y                                            |
| %  | Reminder       | <code>x % y</code>   | Reminder from the x / y                                        |
| +  | Addition       | <code>x + y</code>   | Sum of x and y                                                 |
| -  | Subtraction    | <code>x - y</code>   | Subtraction x and y                                            |
| +  | Unary plus     | <code>+x</code>      | Value of x                                                     |
| -  | Unary minus    | <code>-x</code>      | Value of -x                                                    |
| ++ | Increment      | <code>++x/x++</code> | Incrementation before/after the evaluation of the expression x |
| -- | Decrement      | <code>--x/x--</code> | Decrementation before/after the evaluation of the expression x |

### Implementation-Defined Behaviour

- The C standard deliberately leaves parts of the language unspecified.
- Thus, some parts depend on the implementation, such as compiler, environment, or computer architecture.  
*E.g., Reminder behavior for negative values and version of the C prior C99.*
- The reason for that is the focus of C on efficiency, i.e., match the hardware behavior.
- Having it in mind, it is best to avoid writing programs that depend on implementation-defined behavior.

*K.N.King: Page 55*

*That is one example of difference in writing programs that seem to be working and functional and a program that is correct.*

## Unary Arithmetic Operators

- Unary operator ( `++` and `--` ) change the value of its operand.

*The operand must be the l-value, i.e., an expression that has memory space, where the value of the expression is stored, e.g., a variable.*

- It can be used as **prefix** operator, e.g., `++x` and `--x`;
- or as **postfix** operator, e.g., `x++` and `x--`.
- In each case, the **final value of the expression is different!**

| int i; int a;              | value of i                                          | value of a |
|----------------------------|-----------------------------------------------------|------------|
| <code>i = 1; a = 9;</code> | 1                                                   | 9          |
| <code>a = i++;</code>      | 2                                                   | 1          |
| <code>a = ++i;</code>      | 3                                                   | 3          |
| <code>a = ++(i++);</code>  | <b>Not allowed! Value of i++ is not the l-value</b> |            |

*For the unary operator `i++`, it is necessary to store the previous value of `i` and then the variable `i` is incremented. The expression `++i` only increments the value of `i`. Therefore, `++i` can be more efficient.*

## Logical operators

- Operands can be of arithmetic type or pointers.
- Resulting value **1** means **true**, **0** means **false**.
- In the expressions `&&` (Logical AND) and `||` (Logical OR), the left operand is evaluated first.

- If the results is defined by the left operand, the right operand is not evaluated.

**Short-circuiting behavior** – it may speed evaluation of complex expressions in runtime.

|                         |             |                             |                                                  |
|-------------------------|-------------|-----------------------------|--------------------------------------------------|
| <code>&amp;&amp;</code> | Logical AND | <code>x &amp;&amp; y</code> | 1 if x and y is not 0; otherwise 0.              |
| <code>  </code>         | Logical OR  | <code>x    y</code>         | 1 if at least one of x, y is not 0; otherwise 0. |
| <code>!</code>          | Logical NOT | <code>!x</code>             | 1 if x is 0; otherwise 0.                        |

- Operands `&&` a `||` have the **short-circuiting behavior**, i.e., the second operand is not evaluated if the result can be determined from the value of the first operand.

## Relational Operators

- Operands of relational operators can be of arithmetic type, pointers (of the same type) or one operand can be `NULL` or pointer of the `void` type.

|                    |                       |                        |                                                   |
|--------------------|-----------------------|------------------------|---------------------------------------------------|
| <code>&lt;</code>  | Less than             | <code>x &lt; y</code>  | 1 if x is less than y; otherwise 0                |
| <code>&lt;=</code> | Less than or equal    | <code>x &lt;= y</code> | 1 if x is less then or equal to y; otherwise 0    |
| <code>&gt;</code>  | Greater than          | <code>x &gt; y</code>  | 1 if x is greater than y; otherwise 0             |
| <code>&gt;=</code> | Greater than or equal | <code>x &gt;= y</code> | 1 if x is greater than or equal to y; otherwise 0 |
| <code>==</code>    | Equal                 | <code>x == y</code>    | 1 if x is equal to y; otherwise 0                 |
| <code>!=</code>    | Not equal             | <code>x != y</code>    | 1 if x is not equal to y; otherwise 0             |

## Example – Short-Circuiting Behaviour 1/2

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int fce_a(int n);
5 int fce_b(int n);
6
7 int main(int argc, char *argv[])
8 {
9     if (argc > 1 && fce_a(atoi(argv[1])) && fce_b(atoi(argv[1])))
10    {
11        printf("Both functions fce_a and fce_b pass the test\n");
12    } else {
13        printf("One of the functions does not pass the test\n");
14    }
15    return 0;
16 }
17
18 int fce_a(int n)
19 {
20    printf("Calling fce_a with the argument '%d'\n", n);
21    return n % 2 == 0;
22 }
23
24 int fce_b(int n)
25 {
26    printf("Calling fce_b with the argument '%d'\n", n);
27    return n > 2;
28 }

```

lec02/demo-short\_circuiting.c

## Example – Short-Circuiting Behaviour 2/2 – Tasks

- In the example `lec02/demo-short_circuiting.c`
- Test how the logical expressions (a function call) are evaluated.
  - Identify what functions `fce_a()` and `fce_b()` are implementing.
  - Rename the functions appropriately.
  - Identify the function headers and why they have to be stated above the main function.
  - Try to split implementation of the functions to a separate module.

## Bitwise Shift Operators

- Bitwise shift operators shift the binary representation by a given number of bits to the left or right.
  - Left shift – Each bit shifted off a zero bit enters at the right.
  - Right shift – Each bit shift off.
    - A zero bit enters at the left – for positive values or unsigned types.
    - For negative values, the entered bit can be either 0 (logical shift) or 1 (arithmetic shift right). Depends on the compiler.
- Bitwise shift operators **have lower precedence than the arithmetic operators!**
  - `i << 2 + 1` means `i << (2 + 1)`

Do not be surprised – parenthesized the expression!

## Bitwise Operators

- Bitwise operators treat operands as a series of bits.
 

*Low-Level Programming – A programming language is low level when its programs require attention of the irrelevant. K.N.King: Chapter 20.*

|    |                            |                           |                                         |
|----|----------------------------|---------------------------|-----------------------------------------|
| &  | Bitwise AND                | <code>x &amp; y</code>    | 1 if x and y is equal to 1 (bit-by-bit) |
|    | Bitwise inclusive OR       | <code>x   y</code>        | 1 if x or y is equal to 1 (bit-by-bit)  |
| ^  | Bitwise exclusive or (XOR) | <code>x ^ y</code>        | 1 if only x or only y is 1 (bit-by-bit) |
| ~  | Bitwise complement (NOT)   | <code>~x</code>           | 1 if x is 0 (bit-by-bit)                |
| << | Bitwise left shift         | <code>x &lt;&lt; y</code> | Shift of x by y bits to the left        |
| >> | Bitwise right shift        | <code>x &gt;&gt; y</code> | Shift of x by y bits to the right       |

## Example – Bitwise Expressions

```

1  #include <inttypes.h>
3  uint8_t a = 4;
4  uint8_t b = 5;
6  a      dec: 4 bin: 0100
7  b      dec: 5 bin: 0101
8  a & b dec: 4 bin: 0100
9  a | b dec: 5 bin: 0101
10 a ^ b dec: 1 bin: 0001
12 a >> 1 dec: 2 bin: 0010
13 a << 1 dec: 8 bin: 1000
    
```

lec02/bits.c

## Operators for Accessing Memory

Here, for completeness, details in the further lectures.

- In C, we can directly access the memory address of the variable. *We need in `scanf()`!*
- The access is realized through a pointer. *It is an integer value, typically `long`.*

*It allows great options and also understand data representation and memory access models.*

| Operator           | Name                   | Example              | Result                                                                                  |
|--------------------|------------------------|----------------------|-----------------------------------------------------------------------------------------|
| <code>&amp;</code> | Address                | <code>&amp;x</code>  | Pointer to <code>x</code>                                                               |
| <code>*</code>     | Indirection            | <code>*p</code>      | Variable (or function) addressed by the pointer <code>p</code> .                        |
| <code>[]</code>    | Array subscripting     | <code>x[i]</code>    | <code>*(x+i)</code> – item of the array <code>x</code> at the position <code>i</code> . |
| <code>.</code>     | Structure/union member | <code>s.x</code>     | Member <code>x</code> of the struct/union <code>s</code> .                              |
| <code>-&gt;</code> | Structure/union member | <code>p-&gt;x</code> | Member <code>x</code> of the struct/union addressed by the pointer <code>p</code> .     |

*It is not allowed an operand of the `&` operator is a bit field or variable of the register class, because it has to be addressable memory space.*

*Operator of the indirect address `*` allows to access to the memory using pointers.*

## Other Operators

| Operator            | Name             | Example                | Result                                                                                             |
|---------------------|------------------|------------------------|----------------------------------------------------------------------------------------------------|
| <code>()</code>     | Function call    | <code>f(x)</code>      | Call the function <code>f</code> with the argument <code>x</code> .                                |
| <code>(type)</code> | Cast             | <code>(int)x</code>    | Change the type of <code>x</code> to <code>int</code> .                                            |
| <code>sizeof</code> | Size of the item | <code>sizeof(x)</code> | Size of <code>x</code> in bytes.                                                                   |
| <code>? :</code>    | Conditional      | <code>x ? y : z</code> | Do <code>y</code> if <code>x != 0</code> ; otherwise <code>z</code> .                              |
| <code>,</code>      | Comma            | <code>x, y</code>      | Evaluate <code>x</code> and then <code>y</code> , the result is the result of the last expression. |

- The operand of `sizeof()` can be a type name or expression.

```
1 int a = 10;
2 printf("%lu %lu\n", sizeof(a), sizeof(a + 1.0));
```

`lec02/sizeof.c`

- Example of the **comma** operator.

```
1 for (c = 1, i = 0; i < 3; ++i, c += 2) {
2     printf("i: %d c: %d\n", i, c);
3 }
```

## Cast Operator

- Changing the variable type in runtime is called type cast.
- Explicit cast is written by the name of the type in `()`, e.g.,

```
1 int i;
2 float f = (float)i;
```

- Implicit cast is made automatically by the compiler during the program compilation.
- If the new type can represent the original value, the value is preserved by the cast.
- Operands of the `char`, `unsigned char`, `short`, `unsigned short`, and the bit field types can be used everywhere where it is allowed to use `int` or `unsigned int`.  
*C expects at least values of the `int` type.*
  - Operands are automatically cast to the `int` or `unsigned int`.

## Operators Associativity and Precedence

- Binary operation `op` is **associative** on the set `S` if  
 $(x \text{ op } y) \text{ op } z = x \text{ op } (y \text{ op } z)$ , for each  $x, y, z \in S$ .
- For not associative operators, it is required to specify the order of evaluation.

- Left-associative – operations are grouped from the left.

*E.g.,  $10 - 5 - 3$  is evaluated as  $(10 - 5) - 3$ .*

- Right-associative – operations are grouped from the right.

*E.g.,  $3 + 5^2$  is 28 or  $3 \cdot 5^2$  is 75 vs  $(3 \cdot 5)^2$  is 225.*

- The assignment is right-associative.

*E.g.,  $y=y+8$ .*

*First, the whole right side of the operator `=` is evaluated, and then, the results are assigned to the variable on the left.*

- The order of the operator evaluation can be defined by the **fully parenthesized expression**.

## Simple Assignment

- Set the value to the variable.

*Store the value into the memory space referenced by the variable name.*

- The form of the assignment operator is

$$\langle \text{variable} \rangle = \langle \text{expression} \rangle$$

*Expression is literal, variable, function call, ...*

- C is statically typed programming language.

- A value of an expression can be assigned only to a variable of the same type.

*Otherwise the type cast is necessary.*

- Example of the implicit type cast.

```
1 int i = 320.4; // implicit conversion from 'double' to 'int' changes value from 320.4 to 320
   [-Wliteral-conversion]
3 char c = i;   // implicit truncation 320 -> 64
```

- C is type safe only within a limited context of the compilation, e.g., for `printf("%d\n", 10.1);` a compiler reports an error.

- In general, C is not type safe. *In runtime, it is possible to write out of the allocated memory space.*

## Assignment Expression and Assignment Statement

- The statement performs some action and it is terminated by `;`

```
1 robot_heading = -10.23;
2 robot_heading = fabs(robot_heading);
3 printf("Robot heading: %f\n", robot_heading);
```

- Expression has **type and value**.

```
23      int type, value is 23
14+16/2 int type, value is 22
y=8     int type, value is 8
```

- Assignment is an expression and its value is assigned to the left side.

- By adding **the semicolon**, the assignment expression becomes the assignment statement.

## Compound Assignment

- A short version of the assignment to compute a new value of the variable from itself:

$$\langle \text{variable} \rangle = \langle \text{variable} \rangle \langle \text{operator} \rangle \langle \text{expression} \rangle$$

- can be written as

$$\langle \text{variable} \rangle \langle \text{operator} \rangle = \langle \text{expression} \rangle$$

### Example

```
1 int i = 10;           1 int i = 10;
2 double j = 12.6;     2 double j = 12.6;
4 i = i + 1;           4 i += 1;
5 j = j / 0.2;         5 j /= 0.2;
```

- Note that the assignment is an expression.

*The assignment of the value to the variable is a side effect.*

```
1 int x, y;
3 x = 6;
4 y = x = x + 6;
```

## Undefined Behaviour

- There are some statements that can cause **undefined behavior** according to the C standard.

- `c = (b = a + 2) - (b - 1);`
- `j = i * i++;`

- The program may behave differently according to the used compiler, but may also not compile or may not run; or it may even crash and behave erratically or produce meaningless results.

- It may also happen if variables are used without initialization.

- Avoid statements that may produce undefined behavior!**

*A further detailed example of undefined behavior and code optimization with its analysis is in Lecture 09.*



## Part II

## Part 2 – Control Structures: Selection Statements and Loops

## Coding Style

- It supports clarity and readability of the source code.  
[https://www.gnu.org/prep/standards/html\\_node/Writing-C.html](https://www.gnu.org/prep/standards/html_node/Writing-C.html)
- Formatting of the code is the fundamental step.  
*Setup automatic formatting in your text editor.*
- Appropriate identifiers.  
*Setup automatic formatting in your text editor.*
- Train yourself in coding style even at the cost of slower coding!
- Readability and clarity is important, especially during debugging!
- Recommend coding style.  
*Notice, sometimes it can be better to start from scratch.*
  - Use English, especially for identifiers.
  - Use nouns for variables.
  - Use verbs for function names.

```

1 void function(void)
2 { /* function block start */
3     for (int i = 0; i < 10; ++i) {
4         //inner for-loop block
5         if (i == 5) {
6             break;
7         }
8     }
9 }

```

*Lecturer's preference: indent shift 3, space characters rather than tabular.*

## Statement and Compound Statement (Block)

- Statement is terminated by ;  
*Statement consisting only of the semicolon is empty statement.*
- Block consists of sequences of declarations and statements.
- ANSI C, C89, C90:** Declarations must be placed prior other statements.  
*It is not necessary for C99.*
- Start and end of the block is marked by the curly brackets { and }.
- A block can be inside other block.

```

void function(void)
{ /* function block start */
  /* inner block */
  for (i = 0; i < 10; ++i)
  {
    //inner for-loop block
  }
}

```

```

void function(void) { /* function block
start */
  /* inner block */
  for (int i = 0; i < 10; ++i) {
    //inner for-loop block
  }
}

```

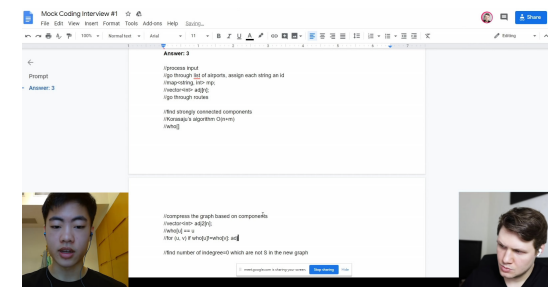
*Notice the coding styles.*

## Coding Style – Code Clarity and Readability

- There are many different coding styles.
- Inspire yourself by existing recommendations and by reading representative source codes.



Clean Code - Uncle Bob / Lesson 1  
<https://youtu.be/7EmboKQH81M>



Google Coding Interview with a High School Student  
<https://youtu.be/qz9tK1F431k>

<http://users.ece.cmu.edu/~eno/coding/CCodingStandard.html>;  
<https://www.doc.ic.ac.uk/lab/cplus/cstyle.html>;  
[http://en.wikipedia.org/wiki/Indent\\_style](http://en.wikipedia.org/wiki/Indent_style);  
<https://google.github.io/styleguide/cppguide.html>;  
<https://www.kernel.org/doc/Documentation/process/coding-style.rst>

## Compound Command and Nesting 1/2

Four nested levels.

```

1 int get_sum_of_even_numbers(int from, int to)
2 {
3     if (from < to) {
4         int sum = 0;
5         for (int number = from; number <= to; ++number) {
6             if (number % 2 == 0) {
7                 sum += number;
8             }
9         } // end for loop
10        return sum;
11    } else {
12        return 0;
13    }
14 }
    
```

We aim to have a more readable form.

```

1 int get_sum_of_even_numbers(int from, int to)
2 {
3     if (from > to) return 0;
4     int sum = 0;
5     for (int number = from; number <= to; ++number) {
6         sum += filter_odd(number);
7     } // end for loop
8     return sum;
9 }
    
```

Extraction (new function definition).

```

1 int filter_odd(int number);
2
3 int get_sum_of_even_numbers(int from, int to)
4 {
5     if (from < to) {
6         int sum = 0;
7         for (int number = from; number <= to; ++number) {
8             sum += filter_odd(number);
9         } // end for loop
10        return sum;
11    } else {
12        return 0;
13    }
14 }
15
16 int filter_odd(int number)
17 {
18     if (number % 2 == 0) {
19         return number;
20     }
21     return 0;
22 }
    
```

■ Using **extraction** and **inversion** techniques, we reduce the nesting depth. <https://youtu.be/CFRhGnuXG-4>

## Compound Command and Nesting 2/2

Inversion (substitution of the input value conditions).

```

1 int filter_odd(int number);
2
3 int get_sum_of_even_numbers(int from, int to)
4 {
5     if (from > to) {
6         return 0;
7     }
8     int sum = 0;
9     for (int number = from; number <= to; ++number) {
10        sum += filter_odd(number);
11    } // end for loop
12    return sum;
13 }
14
15 int filter_odd(int number)
16 {
17     if (number % 2 == 0) {
18         return number;
19     }
20     return 0;
21 }
    
```

■ Using **extraction** and **inversion** techniques, we reduce the nesting depth.

Final cleanup.

```

1 int filter_odd(int number);
2
3 int get_sum_of_even_numbers(int from, int to)
4 {
5     if (from > to) return 0;
6     int sum = 0;
7     for (int number = from; number <= to; ++
8         number) {
9         sum += filter_odd(number);
10    } // end for loop
11    return sum;
12 }
13
14 int filter_odd(int number)
15 {
16     return (number % 2 == 0) ? number : 0;
17 }
    
```

<https://youtu.be/CFRhGnuXG-4>

## Control Statements

- Selection Statement
  - Selection Statement: `if ()` or `if () ... else`
  - Switch Statement: `switch () case ...`
- Control Loops
  - `for ()`
  - `while ()`
  - `do ... while ()`
- Jump statements (unconditional program branching)
  - `continue`
  - `break`
  - `return`
  - `goto`

## Selection Statement – if

- `if (expression) statement1; else statement2`
- For `expression != 0` the `statement1` is executed; otherwise `statement2`.  
*The statement can be the compound statement.*
- The `else` part is optional.
- Selection statements can be nested and cascaded.

Why You Shouldn't Nest Your Code – <https://youtu.be/CFRhGnuXG-4>.

```

1 int max;
2 if (a > b) {
3     if (a > c) {
4         max = a;
5     }
6 }
    
```

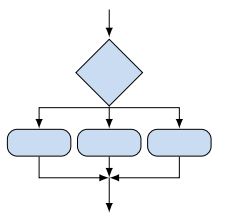
```

1 int max;
2 if (a > b) {
3     ...
4 } else if (a < c) {
5     ...
6 } else if (a == b) {
7     ...
8 } else {
9     ...
10 }
    
```

## The switch Statement

- Allows to branch the program based on the value of the expression of the enumerate (integer) type, e.g., `int`, `char`, `short`, `enum`.
- The form is

```
switch (expression) {
    case constant1: statements1; break;
    case constant2: statements2; break;
    ...
    case constantn: statementsn; break;
    default: statementsdef; break;
}
```



where *constants* are of the same type as the *expression* and *statements<sub>i</sub>* is a list of statements.

- Switch statements can be nested.
  - Semantics: First the expression value is calculated. Then, the statements under the same value are executed. If none of the branch is selected, statements<sub>def</sub> under default branch as performed (optional).*

## The switch Statement – Example

```
switch (v) {
    case 'A':
        printf("Upper 'A'\n");
        break;
    case 'a':
        printf("Lower 'a'\n");
        break;
    default:
        printf("It is not 'A' nor 'a'\n");
        break;
}

if (v == 'A') {
    printf("Upper 'A'\n");
} else if (v == 'a') {
    printf("Lower 'a'\n");
} else {
    printf("It is not 'A' nor 'a'\n");
}
```

lec02/switch.c

## The Role of the break Statement

- The statement `break` terminates the branch. If not presented, the execution continues with the statement of the next `case` label.

```
1 int part = ?
2 switch(part) {
3     case 1:
4         printf("Branch 1\n");
5         break;
6     case 2:
7         printf("Branch 2\n");
8     case 3:
9         printf("Branch 3\n");
10        break;
11    case 4:
12        printf("Branch 4\n");
13        break;
14    default:
15        printf("Default branch\n");
16        break;
17 }
```

- part ← 1    Branch 1
- part ← 2    Branch 2
- part ← 3    Branch 3
- part ← 4    Branch 4
- part ← 5    Default branch

lec02/demo-switch\_break.c

## Loops

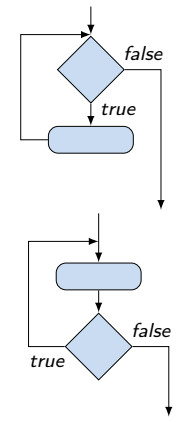
- The `for` and `while` loop statements test the controlling expression before the enter to the loop body.

- `for` – initialization, condition, change of the controlling variable is a part of the syntax.
 

```
1 for (int i = 0; i < 5; ++i) {
2     ...
3 }
```
- `while` – controlling variable is out of the syntax
 

```
1 int i = 0;
2 while (i < 5) {
3     ...
4     i += 1;
5 }
```
- The `do` loop tests the controlling expression after the 1st loop is performed.
 

```
1 int i = -1;
2 do {
3     ...
4     i += 1;
5 } while (i < 5);
```



## The for Loop

- The basic form has four parts (three expressions and a single statement).  
`for (expr1; expr2; expr3) statement`
- All `expri` are expressions and typically they are used for
  - `expr1` – initialization of the controlling variable (side effect of the assignment expression);
  - `expr2` – Test of the controlling expression;
  - If `expr2 != 0` the `statement` is executed; Otherwise the loop is terminated.
  - `expr3` – updated of the controlling variable (performed at the end of the loop)
- Any of the expressions `expri` can be omitted.
- `break` statement – force termination of the loop.
- `continue` – force end of the current iteration of the loop.

*The expression `expr3` is evaluated and test of the loop is performed.*

- An infinity loop can be written by omitting the expressions.

```
for (;;) {...}
```

## The continue Statement

- It transfers the control to the evaluation of the controlling expression.
- The `continue` statement can be used inside the body of the loops.
  - `for ()`
  - `while ()`
  - `do...while ()`

```

1 int i;
2 for (i = 0; i < 20; ++i) {
3     if (i % 2 == 0) {
4         continue;
5     }
6     printf("%3d", i);
7 }
8 putchar('\n');

1 for (int i = 0; i < 10; ++i) {
2     printf("i: %i ", i);
3     if (i % 3 != 0) {
4         continue;
5     }
6     printf("\n");
7 }

$ clang demo-continue.c
$ ./a.out
i:0
i:1 i:2 i:3
i:4 i:5 i:6
i:7 i:8 i:9
lec02/demo-continue.c

```

## The break Statement – Force Termination of the Loop

- The program continues with the next statement after the loop.
- `break` example in the `while` loop.

```

1 int i = 10;
2 while (i > 0) {
3     if (i == 5) {
4         printf("i reaches 5, leave the loop\n");
5         break;
6     }
7     i--; // or -i; or i -= 1; or i = i - 1;
8     printf("End of the while loop i: %d\n", i);
9 }

1 $ clang break.c
2 $ ./a.out
3 End of the while loop i: 9
4 End of the while loop i: 8
5 End of the while loop i: 7
6 End of the while loop i: 6
7 End of the while loop i: 5
8 i reaches 5, leave the loop
lec02/break.c

```

- `continue` and `break` example in the `for` loop.

```

1 for (int i = 0; i < 10; ++i) {
2     printf("i: %i ", i);
3     if (i % 3 != 0) {
4         continue;
5     }
6     printf("\n");
7     if (i > 5) {
8         break;
9     }
10 }

$ clang demo-break.c
$ ./a.out
i:0
i:1 i:2 i:3
i:4 i:5 i:6
lec02/demo-break.c

```

## The goto Statement

- `goto` allows transfing the control to the defined label. *It can be used only within a function body.*
- Syntax `goto label;`
- The `goto` can jump only outside of the particular block, it jumps to a statement. *Add empty statement ; if necessary.*
- Label is not a statement!

```

1 int test = 3;
2 for (int i = 0; i < 3; ++i) {
3     for (int j = 0; j < 5; ++j) {
4         if (j == test) {
5             goto loop_out;
6         }
7         fprintf(stdout, "Loop i: %d j: %d\n", i, j);
8     }
9 }
10 return 0;
11 loop_out:
12 fprintf(stdout, "After loop\n"); // goto can jump to a label that
13 return -1;
    represents statement (there must be an address to be jumped at).
lec02/goto.c

```

## Nested Loops

- The **break** statement terminates the inner loop.

```

1 for (int i = 0; i < 3; ++i) {
2     for (int j = 0; j < 3; ++j) {
3         printf("i-j: %i-%i\n", i, j);
4         if (j == 1) {
5             break;
6         }
7     }
8 }

```

```

i-j: 0-0
i-j: 0-1
i-j: 1-0
i-j: 1-1
i-j: 2-0
i-j: 2-1

```

- The outer loop can be terminated by the **goto** statement.

```

1 for (int i = 0; i < 5; ++i) {
2     for (int j = 0; j < 3; ++j) {
3         printf("i-j: %i-%i\n", i, j);
4         if (j == 2) {
5             goto outer;
6         }
7     }
8 }
9 outer:
10 ;

```

```

i-j: 0-0
i-j: 0-1
i-j: 0-2

```

lec02/demo-goto.c

## Example – isPrimeNumber() 1/2

```

1 #include <stdbool.h>
2 #include <math.h>
3
4 _Bool isPrimeNumber(int n)
5 {
6     _Bool ret = true;
7     for (int i = 2; i <= (int)sqrt((double)n); ++i) {
8         if (n % i == 0) {
9             ret = false;
10            break;
11        }
12    }
13    return ret;
14 }

```

lec02/demo-prime.c

- Once the first factor is found, call **break** to terminate the loop.

*It is not necessary to test other numbers.*

## Example – isPrimeNumber() 2/2

- The value of `(int)sqrt((double)n)` is not changing in the loop.

```

1 for (int i = 2; i <= (int)sqrt((double)n); ++i) {
2     ...
3 }

```

- We can use the **comma operator** to initialize the `maxBound` variable.

```

1 for (int i = 2, maxBound = (int)sqrt((double)n);
2     i <= maxBound; ++i) {
3     ...

```

- Or, we can define `maxBound` as a constant variable.

```

1 _Bool ret = true;
2 const int maxBound = (int)sqrt((double)n);
3 for (int i = 2; i <= maxBound ; ++i) {
4     ...
5 }

```

*E.g., Compile and run demo-prime.c: clang demo-prime.c -lm; ./a.out 13.*

## Conditional Expression – Example Greatest Common Divisor

```

1 int getGreatestCommonDivisor(int x, int y)
2 {
3     int d;
4     if (x < y) {
5         d = x;
6     } else {
7         d = y;
8     }
9     while ( (x % d != 0) || (y % d != 0) ) {
10        d = d - 1;
11    }
12    return d;
13 }

```

- The same with the conditional expression `expr1 ? expr2 : expr3` can be as follows.

```

1 int getGreatestCommonDivisor(int x, int y)
2 {
3     int d = x < y ? x : y;
4     while ( (x % d != 0) || (y % d != 0) ) {
5         d = d - 1;
6     }
7     return d;
8 }

```

lec02/demo-gcd.c

# Part III

## Part 3 – Assignment HW 01

## HW 01 – Assignment

### Topic: ASCII art

Mandatory: **2 points**; Optional: none; Bonus : none

- **Motivation:** Have a fun with loops and user parametrization of the program.
- **Goal:** Acquire experience using loops and inner loops.
- **Assignment** <https://cw.fel.cvut.cz/wiki/courses/b3b36prg/hw/hw01>
  - Read parameters specifying a picture of small house using selected ASCII chars. [https://en.wikipedia.org/wiki/ASCII\\_art](https://en.wikipedia.org/wiki/ASCII_art)
  - Assesment of the input values.
- **Deadline:** 15.03.2025, 23:59 AoE.

*AoE – Anywhere on Earth.*

## Summary of the Lecture

## Topics Discussed

- Expressions
  - Operators – Arithmetic, Relational, Logical, Bitwise, and others
  - Operator Associativity and Precedence
  - Assignment and Compound Assignment
  - Implementation-Defined Behaviour
  - Undefined Behaviour
- Coding Styles
- Select Statements
- Loops
- Conditional Expression
  
- **Next:** Data types, memory storage classes, function call

## Part V

### Appendix

### Coding Example – Assignment

- Implement a program that prints the pattern with seven lines.
- The default width  $n$  is 27 characters or it is read as the first program argument (if given).
- The width  $n$  needs to be odd number, or the program returns 100.
- It holds  $11 \leq n \leq 67$ , or the program returns 101.
- On success, the program prints seven lines and returns 0.
- Avoid “magic numbers” in the program whenever possible.

```

1 * * * * *
2 ** ** ** **
3 *** *** **
4 *****
5 *** ** **
6 ** ** **
7 * * * * *
```

- Convert program `argv[1]` by `atoi()`, if given.
- Decompose the program into printing  $7 \times$  line.
- Implement the program infrastructure first.
- Then, focus on logic to particular lines controlled by a suitably designed **expressions**.

### Coding Example – Implementation Strategy 1/4

- Define return (error) values to make the code clean (0, 100, 101), e.g., using `enum`.
- Define valid range  $\langle 11, 67 \rangle$ , e.g., using `#define`.
- Ensure accessing passed arguments to the program only if they are passed to the program.
- Ensure the number of lines  $n$  is a valid value or set the error program return value.
- Perform any operation only if arguments (values) are valid.
- Split printing 7 lines into two for loops, with one print line call between the loops.
- Implement a function to print the line pattern.

```

1 #include <stdio.h> //for putchar()
2 #include <stdlib.h> //for atoi()
3
4 enum {
5     ERROR_OK = 0,
6     ERROR_INPUT = 100,
7     ERROR_RANGE = 101
8 };
9
10 #define MIN_VALUE 11
11 #define MAX_VALUE 67
12
13 #define LINES 3
14
15 // Print line of the with n using character in c
16 // and space; with k continuous characters c
17 // followed by space.
18 void print(char c, int n, int k);
```

### Coding Example – Implementation Strategy 2/4

- Define return (error) values to make the code clean (0, 100, 101), e.g., using `enum`.
- Define valid range  $\langle 11, 67 \rangle$ , e.g., using `#define`.
- Ensure accessing passed arguments to the program only if they are passed to the program.
- Ensure the number of lines  $n$  is a valid value or set the error program return value.
- Perform any operation only if arguments (values) are valid.
- Split printing 7 lines into two for loops, with one print line call between the loops.
- Implement a function to print the line pattern.

```

1 ...
2 int main(int argc, char *argv[])
3 {
4     int ret = ERROR_OK;
5     int n = argc > 1 ? atoi(argv[1]) : 27; //
6     // convert argv[1] or use default value
7     ret = n % 2 == 0 ? ERROR_INPUT : ret; // ensure
8     // n is odd number
9     if (!ret &&
10         (n < MIN_VALUE || n > MAX_VALUE)) {
11         ret = ERROR_RANGE; //ensure n is in the
12         // closed interval [MIN_VALUE, MAX_VALUE]
13     }
14     ...
15     return ret;
16 }
```

## Coding Example – Implementation Strategy 3/4

- Define return (error) values to make the code clean (0, 100, 101), e.g., using enum.
- Define valid range  $\langle 11, 67 \rangle$ , e.g., using #define.
- Ensure accessing passed arguments to the program only if they are passed to the program.
- Ensure the number of lines  $n$  is a valid value or set the error program return value.
- Perform any operation only if arguments (values) are valid.
- Split printing 7 lines into two for loops, with one print line call between the loops.
- Implement a function to print the line pattern.

```

1 // print a line with n characters with the
  pattern: k-times c, then space.
2 // the line ends by new line character '\n'.
3 void print(char c, int n, int k);
4
5 int main(int argc, char *argv[])
6 { ...
7     if (!ret) { // only if ret == ERROR_OK
8         for (int l = 1; l <= LINES; ++l) {
9             print('* ', n, l); // print l x '* '
10        }
11        print('* ', n, n); // print n x '* '
12        for (int l = LINES; l > 0; --l) {
13            print('* ', n, l); // print l x '* '
14        }
15    }
16    return ret;
17 }
```

## Coding Example – Implementation Strategy 4/4

- Define return (error) values to make the code clean (0, 100, 101), e.g., using enum.
- Define valid range  $\langle 11, 67 \rangle$ , e.g., using #define.
- Ensure accessing passed arguments to the program only if they are passed to the program.
- Ensure the number of lines  $n$  is a valid value or set the error program return value.
- Perform any operation only if arguments (values) are valid.
- Split printing 7 lines into two for loops, with one print line call between the loops.
- Implement a function to print the line pattern.

```

1 void print(char c, int n, int k)
2 {
3     for (int i = 0; i < n; ++i) {
4         putchar( (i+1) % (k+1) ? c : ' ');
5     }
6     putchar('\n');
7 }
```

- The line consists of  $n$  characters; so  $n$  characters has to be printed.
- Space is placed after each  $k$  characters of  $c$ .
- Multiple of  $k$  can be detected by the remainder after division, the operator `%`.
- We need to handle  $i$  starts from 0.
- The space is every  $(k+1)$ -th character.

## Coding Example – Implementation Strategy 4(b)/4

- Define return (error) values to make the code clean (0, 100, 101), e.g., using enum.
- Define valid range  $\langle 11, 67 \rangle$ , e.g., using #define.
- Ensure accessing passed arguments to the program only if they are passed to the program.
- Ensure the number of lines  $n$  is a valid value or set the error program return value.
- Perform any operation only if arguments (values) are valid.
- Split printing 7 lines into two for loops, with one print line call between the loops.
- Implement a function to print the line pattern.

```

1 void print(char c, int n, int k)
2 {
3     int i, j;
4     for (i = j = 0; i < n; ++i, ++j) {
5         if (j == k) {
6             putchar(' ');
7             j = 0;
8         } else {
9             putchar(c);
10        }
11    }
12    putchar('\n');
13 }
```

- Use extra counter  $j$  for space as every  $k$ -th printed character.
- Enjoy comma operator to increment  $j$  within the `for` loop.

## Summary of the Operators and Precedence 1/3

| Precedence | Operator | Associativity | Name                     |
|------------|----------|---------------|--------------------------|
| 1          | ++       | L→R           | Increment (postfix)      |
|            | --       |               | Decrementation (postfix) |
|            | ()       |               | Function call            |
|            | []       |               | Array subscripting       |
| 2          | . ->     |               | Structure/union member   |
|            | ++       | R→L           | Increment (prefix)       |
|            | --       |               | Decrementation (prefix)  |
|            | !        |               | Logical negation         |
|            | ~        |               | Bitwise negation         |
|            | - +      |               | Unary plus/minus         |
|            | *        |               | Indirection              |
|            | &        |               | Address                  |
|            | sizeof   |               | Size                     |



## Summary of the Operators and Precedence 2/3

| Precedence | Operator     | Associativity | Name                       |
|------------|--------------|---------------|----------------------------|
| 3          | ()           | R→L           | <i>Cast</i>                |
| 4          | *, /, %      | L→R           | Multiplicative             |
| 5          | + --         |               | Additive                   |
| 6          | >>, <<       |               | Bitwise shift              |
| 7          | <, >, <=, >= |               | Relational                 |
| 8          | ==, !=       |               | Equality                   |
| 9          | &            |               | Bitwise AND                |
| 10         | ^            |               | Bitwise exclusive OR (XOR) |
| 11         |              |               | Bitwise inclusive OR (OR)  |
| 12         | &&           |               | Logical AND                |
| 13         |              |               | Logical OR                 |

## Summary of the Operators and Precedence 3/3

| Precedence | Operator   | Associativity | Name                 |
|------------|------------|---------------|----------------------|
| 14         | ? :        | R→L           | <i>Conditional</i>   |
| 15         | =          |               | Assignment           |
|            | +=, -=     |               | additive             |
|            | *=, /=, %= | R→L           | multiplicative       |
|            | <<=, >>=   |               | bitwise shift        |
|            | &=, ^=,  = |               | Bitwise AND, XOR, OR |
| 15         | ,          | L→R           | <i>Comma</i>         |

K. N. King: Page 735

[http://en.cppreference.com/w/c/language/operator\\_precedence](http://en.cppreference.com/w/c/language/operator_precedence)