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   *K. N. King: chapter 4 and 20*
- Part 2 – Control Structures: Selection Statements and Loops
  - Statements and Coding Styles
  - Selection Statements
  - Loops
  - Conditional Expression   *K. N. King: chapters 5 and 6*
- Part 3 – Assignment HW 01

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)		
Decimal	123 450932	
Hexadecimal	0x12 0xFAFF	(starts with `0x` or `0X`)
Octal	0123 0567	(starts with `0`)
unsigned	12345U	(suffix `U` or `u`)
long	12345L	(suffix `L` or `l`)
unsigned long	12345ul	(suffix `UL` or `ul`)
long long	12345LL	(suffix `LL` or `ll`)

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

Rational literals

- `double` – by default, if not explicitly specified to be another type;
- `float` – suffix `F` or `f`;

```
float f = 10.f;
```

- `long double` – suffix `L` or `l`.

```
long double ld = 10.1l;
```

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.**

```
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');
```

- Text string is a sequence of characters enclosed in quotation marks.
  `"A string with the end of line \n"`.

  - String literals separated by white spaces are joined to single one.
  `"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

'w'	'o'	'r'	'd'	'\0'

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

Literals – Enumeration

- By default, values of the enumerated type starts from 0 and each other item increases the value about one, values can be explicitly prescribed.

```
1  enum {
2      WHITE,
3      BLACK,
4      RED,
5      GREEN,
6  };
```

```
1  enum {
2      ERROR_OK = 0, // EXIT_SUCCESS
3      ERROR_INPUT = 100,
4      ERROR_RANGE = 101
5  };
```

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.

```
enum { WHITE = 0, BLACK, RED, GREEN, BLUE, NUM_COLORS };

for (int color = WHITE; color < NUM_COLORS; ++color) {
   ...
}
```

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`;
    None or more type quantifiers are allowed.
  - **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 = 100l; //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 writting 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 ? :.

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 _.   *Or we can use CamelCase.*
  - Each variable is defined at a new line.   *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 ;.

Basic Arithmetic Expressions

- For an operator of the numeric types `int` and `double`, the following operators are defined.

  Also for `char`, `short`, and `float` numeric types.

  - Unary operator for changing the sign −;
  - Binary addition + and subtraction −;
  - Binary multiplication * and division /.
- For integer operator, there is also
  - Binary module (integer reminder) **%**.
- 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.

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 divison
14 printf("a / b = %i\n", result);

16 result = a + b * c; // priority of the operators
17 printf("a + b * c = %i\n", result);
```

Example – Arithmetic Operators 2/2

```c
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
```

Arithmetic Operators

- Operands of arithmetic operators can be of any arithmetic type.

  *The only exception is the operator for the integer reminder % defined for the **int** type.*

*	Multiplication	x * y	Multiplication of x and y
/	Division	x / y	Division of x and y
%	Reminder	x % y	Reminder from the x / y
+	Addition	x + y	Sum of x and y
-	Subtraction	x - y	Subtraction x and y
+	Unary plus	+x	Value of x
-	Unary minus	-x	Value of $-x$
++	Increment	++x/x++	Incrementation before/after the evaluation of the expression x
--	Decrement	--x/x--	Decrementation before/after the evaluation of the expression x

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 reminder, 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.

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 writting 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
i = 1; a = 9;	1	9
a = i++;	2	1
a = ++i;	3	3
a = ++(i++);	Not allowed! Value of i++ is not the l-value	

    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.

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.

$<$	Less than	x < y	1 if x is less than y; otherwise 0
$<=$	Less than or equal	x <= y	1 if x is less then or equal to y; otherwise 0
$>$	Greater than	x > y	1 if x is greater than y; otherwise 0
$>=$	Greater than or equal	x >= y	1 if x is greater than or equal to y; otherwise 0
$==$	Equal	x == y	1 if x is equal to y; otherwise 0
$!=$	Not equal	x != y	1 if x is not equal to y; otherwise 0

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.

&&	Logical AND	x && y	1 if x and y is not 0; otherwise 0.
||	Logical OR	x || y	1 if at least one of x, y is not 0; otherwise 0.
!	Logical NOT	!x	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.

Example – Short-Circuiting Behaviour 1/2

```c
#include <stdio.h>
#include <stdlib.h>

int fce_a(int n);
int fce_b(int n);

int main(int argc, char *argv[])
{
   if (argc > 1 && fce_a(atoi(argv[1])) && fce_b(atoi(argv[1])) )
   {
      printf("Both functions fce_a and fce_b pass the test\n");
   } else {
      printf("One of the functions does not pass the test\n");
   }
   return 0;
}

int fce_a(int n)
{
   printf("Calling fce_a with the argument '%d'\n", n);
   return n % 2 == 0;
}

int fce_b(int n)
```

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

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!

Example – Bitwise Expressions

```c
#include <inttypes.h>

uint8_t a = 4;
uint8_t b = 5;

a     dec: 4 bin: 0100
b     dec: 5 bin: 0101
a & b dec: 4 bin: 0100
a | b dec: 5 bin: 0101
a ^ b dec: 1 bin: 0001


a >> 1 dec: 2 bin: 0010
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
&	Address	&x	Pointer to x
*	Indirection	*p	Variable (or function) addressed by the pointer p.
[]	Array subscripting	x[i]	*(x+i) – item of the array x at the position i.
.	Structure/union member	s.x	Member x of the struct/union s.
->	Structure/union member	p->x	Member x of the struct/union addressed by the pointer p.

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

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

```
int a = 10;
printf("%lu %lu\n", sizeof(a), sizeof(a + 1.0));
```
                                                    lec02/sizeof.c

- Example of the **comma** operator.

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

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.,

```
int i;
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
  $$(x \text{ op } y) \text{ op } z = x \text{ op}(y \text{ op } z), \text{ for each } x, y, z \in \boldsymbol{S}.$$
- 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

  ⟨**variable**⟩ = ⟨**expression**⟩

  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.

  ```
  int i = 320.4; // implicit conversion from 'double' to 'int' changes value from
      320.4 to 320 [-Wliteral-conversion]

  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.*

Compound Assignment

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

  ⟨**variable**⟩ = ⟨**variable**⟩ ⟨**operator**⟩ ⟨**expression**⟩

- can be written as

  ⟨**variable**⟩ ⟨**operator**⟩ = ⟨**expression**⟩

  Example

  ```
  int i = 10;                int i = 10;
  double j = 12.6;           double j = 12.6;

  i = i + 1;                 i += 1;
  j = j / 0.2;               j /= 0.2;
  ```

- Note that the assignment is an expression.

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

  ```
  int x, y;

  x = 6;
  y = x = x + 6;
  ```

Assignment Expression and Assignment Statement

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

  ```
  robot_heading = -10.23;
  robot_heading = fabs(robot_heading);
  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.

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 behaves 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 happened 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

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.

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

```c
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

- It supports clarity and readability of the source code.

  https://www.gnu.org/prep/standards/html_node/Writing-C.html

- Formatting of the code is the fundamental step.
- 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!

  Notice, sometimes it can be better to start from scratch.

- Recommend coding style.

```c
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  }
```

- Use English, especially for identifiers.
- Use nouns for variables.
- Use verbs for function names.

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

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/7EmboKQH8lM
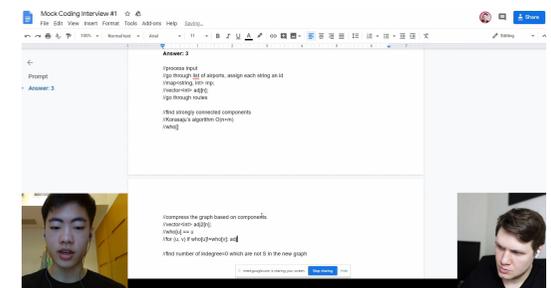
Google Coding Interview with a High School Student
https://youtu.be/qz9tKlF431k

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;
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.

```c
int get_sum_of_even_numbers(int from, int to)
{
    if (from < to) {
        int sum = 0;
        for (int number = from; number <= to; ++number) {
            if (number % 2 == 0) {
                sum += number;
            }
        } // end for loop
        return sum;
    } else {
        return 0;
    }
}
```

We aim to have a more readable form.

```c
int get_sum_of_even_numbers(int from, int to)
{
    if (from > to) return 0;
    int sum = 0;
    for (int number = from; number <= to; ++number) {
        sum += filter_odd(number);
    } // end for loop
    return sum;
}
```

Extraction (new function definition).

```c
int filter_odd(int number);

int get_sum_of_even_numbers(int from, int to)
{
    if (from < to) {
        int sum = 0;
        for (int number = from; number <= to; ++number) {
            sum += filter_odd(number);
        } // end for loop
        return sum;
    } else {
        return 0;
    }
}

int filter_odd(int number)
{
    if (number % 2 == 0) {
        return number;
    }
    return 0;
}
```

- 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).

```c
int filter_odd(int number);

int get_sum_of_even_numbers(int from, int to)
{
    if (from > to) {
        return 0;
    }
    int sum = 0;
    for (int number = from; number <= to; ++number) {
        sum += filter_odd(number);
    } // end for loop
    return sum;
}

int filter_odd(int number)
{
    if (number % 2 == 0) {
        return number;
    }
    return 0;
}
```

Final cleanup.

```c
int filter_odd(int number);

int get_sum_of_even_numbers(int from, int to)
{
    if (from > to) return 0;

    int sum = 0;
    for (int number = from; number <= to; ++number) {
        sum += filter_odd(number);
    } // end for loop
    return sum;
}

int filter_odd(int number)
{
    return (number % 2 == 0) ? number : 0;
}
```

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

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.

```c
int max;
if (a > b) {
    if (a > c) {
        max = a;
    }
}
```

```c
int max;
if (a > b) {
    ...
} else if (a < c) {
    ...
} else if (a == b) {
    ...
} else {
    ...
}
```

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 constant_1: statements_1; **break;**
      case constant_2: statements_2; **break;**
      ...
      case constant_n: statements_n; **break;**
      default: statements_{def}; **break;**
  }
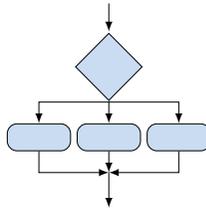
  where *constants* are of the same type as the *expression* and statements_i is a list of statements.
- Switch statements can be nested.
  Semantics: First the expression value is calculated. Then, the statements under the same value are executed. If none of the branch is selected, statements_{def} under default *branch as performed (optional).*

The switch Statement – Example

```c
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;
}
```

```c
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.

Example

```c
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
  Branch 3

- 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 can be a part of the syntax.
    ```c
    1  for (int i = 0; i < 5; ++i) {
    2      ...
    3  }
    ```
  - **while** – controlling variable out of the syntax
    ```c
    1  int i = 0;
    2  while (i < 5) {
    3      ...
    4      i += 1;
    5  }
    ```
- The **do** loop tests the controlling expression after the first loop is performed.
  ```c
  1  int i = -1;
  2  do {
  3      ...
  4      i += 1;
  5  } while (i < 5);
  ```
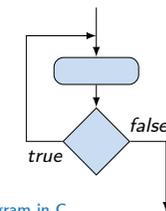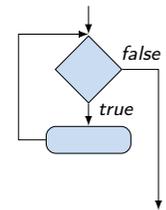
The **for** Loop

- The basic form has four parts (three expressions and a single statement).

$$\text{for (expr}_1\text{; expr}_2\text{; expr}_3\text{) statement}$$

- All expr_i are expressions and typically they are used for
  1. expr_1 – initialization of the controlling variable (side effect of the assignment expression);
  2. expr_2 – Test of the controlling expression;
  3. If expr_2 !=0 the `statement` is executed; Otherwise the loop is terminated.
  4. expr_3 – updated of the controlling variable (performed at the end of the loop

- Any of the expressions expr_i can be omitted.
- **break** statement – force termination of the loop.
- **continue** – force end of the current iteration of the loop.

  The expression expr_3 is evaluated and test of the loop is performed.

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

$$\text{for (;;) \{...\}}$$

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 ()`
- Examples

```c
int i;
for (i = 0; i < 20; ++i) {
    if (i % 2 == 0) {
        continue;
    }
    printf("%d\n", i);
}
```
lec02/continue.c

```c
for (int i = 0; i < 10; ++i) {
    printf("i: %i ", i);
    if (i % 3 != 0) {
        continue;
    }
    printf("\n");
}
```
lec02/demo-continue.c

```
$ 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
```

The break Statement – Force Termination of the Loop

- The program continues with the next statement after the loop.
- Example in the `while` loop.
```c
int i = 10;
while (i > 0) {
    if (i == 5) {
        printf("i reaches 5, leave the loop\n");
        break;
    }
    i--; // or -i; or i -= 1; or i = i - 1;
    printf("End of the while loop i: %d\n", i);
}
```
lec02/break.c

- Example in the `for` loop.
```c
for (int i = 0; i < 10; ++i) {
    printf("i: %i ", i);
    if (i % 3 != 0) {
        continue;
    }
    printf("\n");
    if (i > 5) {
        break;
    }
}
```
lec02/demo-break.c

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

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.
- It can be used only within a function block.

```c
 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
        represents statement (there must be an address to be jump at).
13  return -1;
```
lec02/goto.c

Nested Loops

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

```c
for (int i = 0; i < 3; ++i) {
    for (int j = 0; j < 3; ++j) {
        printf("i-j: %i-%i\n", i, j);
        if (j == 1) {
            break;
        }
    }
}
```

```
1  i-j: 0-0
2  i-j: 0-1
3  i-j: 1-0
4  i-j: 1-1
5  i-j: 2-0
6  i-j: 2-1
```

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

```c
for (int i = 0; i < 5; ++i) {
    for (int j = 0; j < 3; ++i) {
        printf("i-j: %i-%i\n", i, j);
        if (j == 2) {
            goto outer;
        }
    }
}
outer:
;
```

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

lec02/demo-goto.c

Example – isPrimeNumber() 1/2

```c
#include <stdbool.h>
#include <math.h>

_Bool isPrimeNumber(int n)
{
    _Bool ret = true;
    for (int i = 2; i <= (int)sqrt((double)n); ++i) {
        if (n % i == 0) {
            ret = false;
            break;
        }
    }
    return ret;
}
```

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.

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

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

```c
for (int i = 2, maxBound = (int)sqrt((double)n);
        i <= maxBound; ++i) {
    ...
}
```

- Or, we can declare maxBound as a constant variable.

```c
_Bool ret = true;
const int maxBound = (int)sqrt((double)n);
for (int i = 2; i <= maxBound ; ++i) {
    ...
}
```

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

Conditional Expression – Example Greatest Common Divisor

```c
int getGreatestCommonDivisor(int x, int y)
{
    int d;
    if (x < y) {
        d = x;
    } else {
        d = y;
    }
    while ( (x % d != 0) || (y % d ! = 0)) {
        d = d - 1;
    }
    return d;
}
```

- The same with the conditional expression $expr_1$? $expr_2$: $expr_3$ can be as follows.

```c
int getGreatestCommonDivisor(int x, int y)
{
    int d = x < y ? x : y;
    while ( (x % d != 0) || (y % d ! = 0)) {
        d = d - 1;
    }
    return d;
}
```

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

  - Assesment of the input values.

- Deadline: 16.03.2024, 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.

- Peform 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.

```c
#include <stdio.h> //for putchar()
#include <stdlib.h> //for atoi()

enum {
    ERROR_OK = 0,
    ERROR_INPUT = 100,
    ERROR_RANGE = 101
};

#define MIN_VALUE 11
#define MAX_VALUE 67

#define LINES 3

// Print line of the with n using character
//   in c and space; with k continuous
//   characters c followed by space.
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.

- Peform 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.

```c
...
int main(int argc, char *argv[])
{
    int ret = ERROR_OK;
    int n = argc > 1 ? atoi(argv[1]) : 27; //
      convert argv[1] or use default value

    ret = n % 2 == 0 ? ERROR_INPUT : ret; //
      ensure n is odd number
    if (!ret &&
          (n < MIN_VALUE || n > MAX_VALUE)) {
        ret = ERROR_RANGE; //ensure n is in the
      closed interval [MIN_VALUE, MAX_VALUE]
    }
    ...
    return ret;
}
```

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.

- Peform 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.

```c
// print a line with n characters with the
    pattern:  k-times c, then space.
// the line ends by new line character '\n'.
void print(char c, int n, int k);

int main(int argc, char *argv[])
{ ...
    if (!ret) { // only if ret == ERROR_OK
        for (int l = 1; l <= LINES; ++l) {
            print('*', n, l); // print l x '*'
        }
        print('*', n, n); // print n x '*'
        for (int l = LINES; l > 0 ; --l) {
            print('*', n, l); // print l x 'x'
        }
    }
    return ret;
}
```

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.

- Peform 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.

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

- 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.

- Peform 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.

```c
void print(char c, int n, int k)
{
    int i, j;
    for (i = j = 0; i < n; ++i, ++j) {
        if (j == k) {
            putchar(' ');
            j = 0;
        } else {
            putchar(c);
        }
    }
    putchar('\n');
}
```

- 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
	. −>		Structure/union member
2	++	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	*Cast*
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	Conditional
15	=		Assignment
	+=, -=		additive
	*=, /=, %=	R→L	multiplicative
	<<=, >>=		bitwise shift
	&=, ^=, |=		Bitwise AND, XOR, OR
15	,	L→R	Comma

K. N. King: Page 735

http://en.cppreference.com/w/c/language/operator_precedence