

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

Expressions – Literals and Variables

Expressions – Operators

Associativity and Precedence

Assignment

Part I

Part 1 – Expressions

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 1 / 64

Expressions – Literals and Variables Expressions – Operators Associativity and Precedence Assignment

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**.

Always, if you are not sure, use brackets.

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 5 / 64

Expressions – Literals and Variables Expressions – Operators Associativity and Precedence Assignment

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_SUCCCESS
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) {
    ...
}
```

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 8 / 64

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 2 / 64

Expressions – Literals and Variables Expressions – Operators Associativity and Precedence Assignment

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">double – by default, if not explicitly specified to be another type;float – suffix F or f;long double – suffix L or l.
Hexadecimal	0x12 0xFAFF (starts with 0x or 0X)	<ul style="list-style-type: none">float f = 10.f;long double ld = 10.11;
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 `int`.

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 6 / 64

Expressions – Literals and Variables Expressions – Operators Associativity and Precedence Assignment

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 = 1001; //constant long integer variable initialized by long literal
int i; // i variable of the common C integer type int that is not initialized
```

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 9 / 64

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 3 / 64

Expressions – Literals and Variables Expressions – Operators Associativity and Precedence Assignment

Literals – Characters and Text Strings

- Character literal is single (or multiple) character enclosed in apostrophe.
- Text string is a sequence of characters enclosed in quotation marks.

'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');
```

- 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**!

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 7 / 64

Expressions – Literals and Variables Expressions – Operators Associativity and Precedence Assignment

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 **?:**.

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 11 / 64

Expressions – Literals and Variables Expressions – Operators Associativity and Precedence Assignment

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

$$(l\text{-value}) = \langle \text{expression} \rangle$$

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 `;`.

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 12 / 64

Expressions – Literals and Variables Expressions – Operators Associativity and Precedence Assignment

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 `/`.

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

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 13 / 64

Expressions – Literals and Variables Expressions – Operators Associativity and Precedence Assignment

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);
```

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 14 / 64

Expressions – Literals and Variables Expressions – Operators Associativity and Precedence Assignment

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
```

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 15 / 64

Expressions – Literals and Variables Expressions – Operators Associativity and Precedence Assignment

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.

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

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 16 / 64

Expressions – Literals and Variables Expressions – Operators Associativity and Precedence Assignment

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.

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 17 / 64

Expressions – Literals and Variables Expressions – Operators Associativity and Precedence Assignment

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.

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 18 / 64

Expressions – Literals and Variables Expressions – Operators Associativity and Precedence Assignment

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>	Not allowed!	Value of i++ is not the l-value

For the unary operator `++`, 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.

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 19 / 64

Expressions – Literals and Variables Expressions – Operators Associativity and Precedence Assignment

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

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 20 / 64

Expressions – Literals and Variables Expressions – Operators Associativity and Precedence Assignment

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

- Operands **&&** and **||** 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.

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 21 / 64

Expressions – Literals and Variables Expressions – Operators Associativity and Precedence Assignment

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)

```

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 22 / 64

Expressions – Literals and Variables Expressions – Operators Associativity and Precedence Assignment

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.

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 23 / 64

Expressions – Literals and Variables Expressions – Operators Associativity and Precedence Assignment

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 & 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 << y</code>	Shift of x by y bits to the left
>>	Bitwise right shift	<code>x >> y</code>	Shift of x by y bits to the right

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 24 / 64

Expressions – Literals and Variables Expressions – Operators Associativity and Precedence Assignment

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!**

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 25 / 64

Expressions – Literals and Variables Expressions – Operators Associativity and Precedence Assignment

Example – Bitwise Expressions

```

#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`

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 26 / 64

Expressions – Literals and Variables Expressions – Operators Associativity and Precedence Assignment

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

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 27 / 64

Expressions – Literals and Variables Expressions – Operators Associativity and Precedence Assignment

Other Operators

Operator	Name	Example	Result
()	Function call	<code>f(x)</code>	Call the function f with the argument x.
(type)	Cast	<code>(int)x</code>	Change the type of x to int.
sizeof	Size of the item	<code>sizeof(x)</code>	Size of x in bytes.
? :	Conditional	<code>x ? y : z</code>	Do y if x != 0; otherwise z.
,	Comma	<code>x, y</code>	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));
```
- Example of the **comma** operator.


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

`lec02/sizeof.c`

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 28 / 64

Expressions – Literals and Variables Expressions – Operators Associativity and Precedence Assignment

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**.
 - Operands are automatically cast to the **int** or **unsigned int**. *C expects at least values of the int type.*

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 29 / 64

Expressions – Literals and Variables Expressions – Operators Associativity and Precedence Assignment

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**.

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 31 / 64

Expressions – Literals and Variables Expressions – Operators Associativity and Precedence Assignment

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.


```
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.*

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 33 / 64

Expressions – Literals and Variables Expressions – Operators Associativity and Precedence Assignment

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

```
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;
```

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 34 / 64

Expressions – Literals and Variables Expressions – Operators Associativity and Precedence Assignment

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.

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 35 / 64

Expressions – Literals and Variables Expressions – Operators Associativity and Precedence Assignment

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.

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 36 / 64

Statements and Coding Styles Selection Statements Loops Conditional Expression

Part II

Part 2 – Control Structures: Selection Statements and Loops

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 37 / 64

Statements and Coding Styles Selection Statements Loops Conditional Expression

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.

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 39 / 64

Statements and Coding Styles Selection Statements Loops Conditional Expression

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.


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


 - Lecturer's preference: indent shift 3, space characters rather than tabular.
 - Use English, especially for identifiers.
 - Use nouns for variables.
 - Use verbs for function names.

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 40 / 64

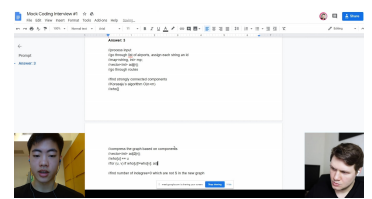
Statements and Coding Styles Selection Statements Loops Conditional Expression

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>

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 41 / 64

Statements and Coding Styles Selection Statements Loops Conditional Expression

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 }

```

Extraction (new function definition).

```

1 int filter_odd(int number);
2
3 int get_sum_of_even_numbers(int from, int to)
4 {
5     int sum = 0;
6     if (from < to) {
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 }

```

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 }

```

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

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C <https://youtu.be/CFRbGnuXG-4> 42 / 64

Statements and Coding Styles Selection Statements Loops Conditional Expression

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 }

```

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
7     int sum = 0;
8     for (int number = from; number <= to; ++number) {
9         sum += filter_odd(number);
10    } // end for loop
11    return sum;
12 }
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.

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C <https://youtu.be/CFRbGnuXG-4> 43 / 64

Statements and Coding Styles Selection Statements Loops Conditional Expression

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`

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 44 / 64

Statements and Coding Styles Selection Statements Loops Conditional Expression

Selection Statement – if

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

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

```

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

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

```

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 46 / 64

Statements and Coding Styles Selection Statements Loops Conditional Expression

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_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, `statementsdef` under default branch as performed (optional).

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 47 / 64

Statements and Coding Styles Selection Statements Loops Conditional Expression

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

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 48 / 64

Statements and Coding Styles Selection Statements Loops Conditional Expression

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

```

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         break;
9     case 3:
10        printf("Branch 3\n");
11        break;
12    case 4:
13        printf("Branch 4\n");
14        break;
15    default:
16        printf("Default branch\n");
17        break;
18 }

```

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

lec02/ame-switch_break.c

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 49 / 64

Statements and Coding Styles Selection Statements Loops Conditional Expression

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.


```

1 for (int i = 0; i < 5; ++i) {
2     ...
3 }

```
 - `while` – controlling variable 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 first loop is performed.


```

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

```

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 51 / 64

Statements and Coding Styles Selection Statements Loops Conditional Expression

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 (;;) {...}

```

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 52 / 64

Statements and Coding Styles Selection Statements Loops Conditional Expression

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

```

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

```

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
1:0
1:1 i:2 i:3
1:4 i:5 i:6
1:7 i:8 i:9

lec02/continue.c

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 53 / 64

Statements and Coding Styles Selection Statements Loops Conditional Expression

The break Statement – Force Termination of the Loop

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

```

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.

```

for (int i = 0; i < 10; ++i) {
    printf("i: %i ", i);
    if (i % 3 != 0) {
        continue;
    }
    printf("\n");
    if (i > 5) {
        break;
    }
}
  
```

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

lec02/demo-break.c

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 54 / 64

Statements and Coding Styles Selection Statements Loops Conditional Expression

The goto Statement

- `goto` allows transferring 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.

```

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     represents statement (there must be an address to be jump at).
14 return -1;
  
```

lec02/goto.c

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 55 / 64

Statements and Coding Styles Selection Statements Loops Conditional Expression

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 }
  
```

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.

```

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

lec02/demo-goto.c

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 56 / 64

Statements and Coding Styles Selection Statements Loops Conditional Expression

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.

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 57 / 64

Statements and Coding Styles Selection Statements Loops Conditional Expression

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 declare `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.

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 58 / 64

Statements and Coding Styles Selection Statements Loops Conditional Expression

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

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 60 / 64

Part III

Part 3 – Assignment HW 01

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 61 / 64

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.

Jan Faigl, 2024 B3B36PRG – Lecture 02: Writing your program in C 62 / 64

Summary of the Lecture

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 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.

```

// 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;
}
    
```

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

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.

```

#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 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.

```

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.
    
```

Part V Appendix

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.

```

...
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 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.

```

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