Zusammenfassung C

March 23, 2020

Inhalt

1 Introduction

C is a programming language, probably most known for being used to write and extend the UNIX Operating System.

It was originally developed by Dennis M. Ritchie in 1972 and remains one of the most used high level programming languages today (next to Java).

A 'C'-File should have the file-extension `.c` and must contain at least the following lines:

```c
#include <stdio.h>
int main(){
}
```

2 Hello World

The codeblock to the right shows a simple 'Hello World' program. All it does it print out 'Hello World' on the console, once compiled and run.

```c
#include <stdio.h>
int main(){
    printf("Hello World");
    return 0;
}
```

The first line of this program `#include <stdio.h>` tells the C-compiler to include the stdio header file before compiling the actual file.

The `int main()` is the main-method. The program execution starts there.

`return 0` terminates the main method and returns the value 0, usually interpreted as 'Everything worked fine'

To run this file, you first have to compile it. Assuming you saved the file as `hello.c` and run Linux, you just have to open a console and run `gcc hello.c`. This command uses the GNU-C-compiler to compile your C-file. It will generate a file called `a.out` (you can change this by specifying a file name with the `-o`-Parameter, e.g. `gcc hello.c -o hello.compiled`).

3 Identifiers

An identifier is a name for a variable or a function. C allows basically all alphanumerical names, as long as they start with a letter or an underscore ('_'). So `abc123` would be a legal variable name, as would be `_11abc`. However `1_ab#d` would not be.

Important: C is case sensitive, so variable `variable` is not the same as `VaRiaBlE`.

I'm saying 'basically', because C has reserved some words for language-specific things. These words are:

auto	else	long	switch	break	enum
register	typedef	case	extern	return	
union	char	float	short	unsigned	
const	for	signed	void	continue	goto
sizeof	volatile	default	if	static	while
do	int	struct	_Packed	double	

4 Data Types

C uses different data types for different things. The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted.

C-Types can be classified as follows:

Basic Types: Arithmetic Types ('You can do math with them'). Are further classified into
a: Integer Types
b: Floating-point Types

Enumerated Types: Arithmetic Types, used to define variables that can only assign certain integer values. (i.e enums)

Void Types: No value

Derived Types: Further divided into
a: Pointer Types
b: Array Types
c: Structure Types
d: Union Types
e: Function Types

4.1 Integer Types

Tabelle 1: Integer Types, their storage size and value range

Type	Storage Size	Value range
char	1 byte	-128 - 127 / 0 - 255
unsigned char	1 byte	0 - 255
signed char	1 byte	-128 - 127
int	2-4 bytes	-32'768 - 32'767 -2'147'483'648 to 2'147'483'647
unsigned int	2-4 bytes	0 - 65'535 0 - 4'294'967'295
unsigned short	2 bytes	0-65'5535
short	2 bytes	-32768 - 32767
long	8 bytes	-9'223'372'036'854'775'808 - 9'223'372'036'854'775'807
unsigned long	8 bytes	0 - 18'446'744'073'709'551'615

Alternatively, you can use the `sizeof` to get the size of a type (e.g. `sizeof(int)`) or just the stored values in the header-files (e.g. `LONG_MAX`)

4.2 Floating-point Types

Tabelle 2: Integer Types, their storage size and value range

Type	Storage Size	Value range	Precision
float	4 byte	1.2E-38 - 3.4E+38	6 decimal places
double	8 byte	2.3E-308 - 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

4.3 Void Types

Function returns as void: The function does not return anything.

Function arguments as void: The function does not take any parameters.

Pointers to void: A pointer can point to a memory-address, but it does not have a type associated to it. You can store at that address whatever you want, given there's enough space for it.

5 Variables

Variables are basically just a name given to a memory-address, so it can be accessed more easily. Each variable has to be assigned to a specified type (char, int, float, double or void).
Variables can be defined, as in most other programming languages, with the '=' operator. (`int i = 3`). Multiple variables can be separated with a comma (`int i, j, k`).
If a C-program consists of multiple files, the compiler can be assured of the existence of a given variable with the `extern` keyword. It might not have been initialized yet, but it will, so an address-space must be allocated.

```c
#include <stdio.h>

// Variable declaration:
extern int a, b;
extern int c;
extern float f;

int main () {

   /* variable definition: */
   int a, b;
   int c;
   float f;

   /* actual initialization */
   a = 10;
   b = 20;

   c = a + b;
   printf("value of c : %d \n", c);

   f = 70.0/3.0;
   printf("value of f : %f \n", f);

   return 0;
}
```

5.1 Lvalues and Rvalues

lvalue: Refers to a memory-address. Can be used on both sides of an assignment (`a=b`)

rvalue: Refers to the data value at a memory-address. Can only be used on the right-hand side of an assignment (`a=20`)

5.2 Constants and Literals

Constants, also called Literals are read-only variables. They are defined once and cannot be changed by the program. This can e.g. be useful for physical constants. The gravity will most likely not change during the runtime of the program. Constants usually refer to the rvalue of the actual value of the constant (rvalue), whereas literals are usually the address of said value (lvalue).

Constants can be defined as hexadecimal (prefix `0x`), octal (prefix `0`) or decimal (no prefix) values. They can also be unsigned (suffix `U`) or long (suffix `L`).

Constants can also be Float (`3.14159`, `314159E-5L`) or even chars. C already defines some char literals, which are preceded by \ (e.g. `\t` or `\n`).
Constants can either be defined with the `#define` preprocessor, or alternatively with the `const` keyword.

```c
#define PI 3.14159

int main(){
    const float G = 9.81;
    printf("Gravity: %f \n", G);
    printf("Pi: %f", PI);
}
```

6 Storage Classes

Storage classes define the visibility, scope and lifetime of a variable or function.

auto: Default - Only visible within the function (i.e. local variables)

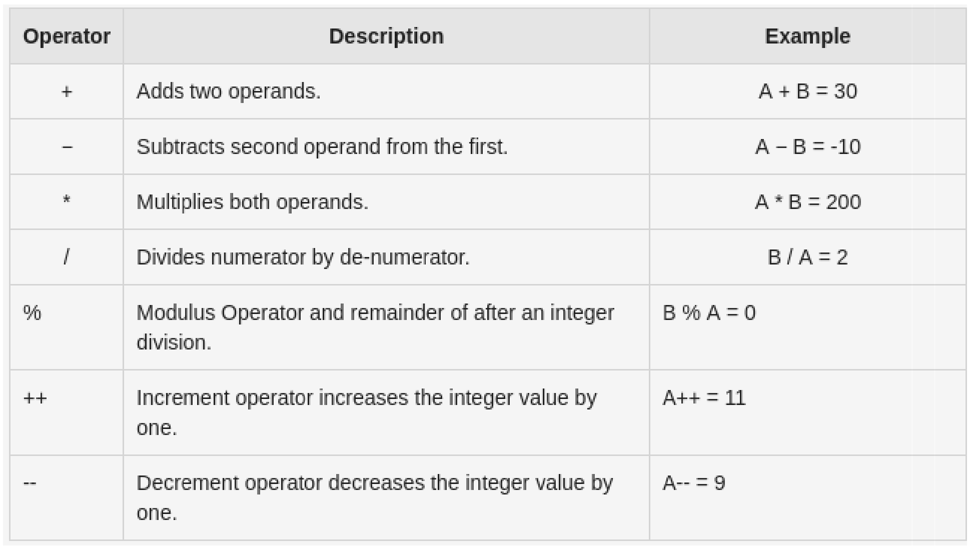
register: Local variables which are to be stored in a register instead of memory. It can be accessed faster, but its size is limited to the register size (usually one word).

static: Global variable - Is kept alive throughout the whole program-execution. Alternatively, it can also mean that it is not destroyed and re-initiated after every function call.

extern: Variable visible to **all** program files.

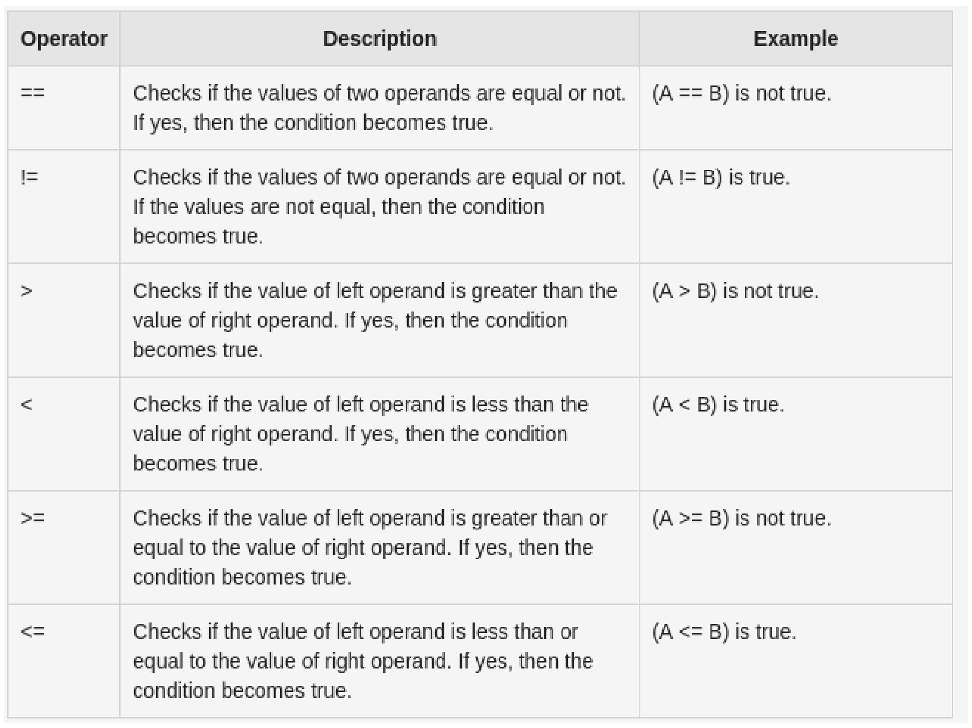
7 Operators

The following tables/images show the different operators supported by C.



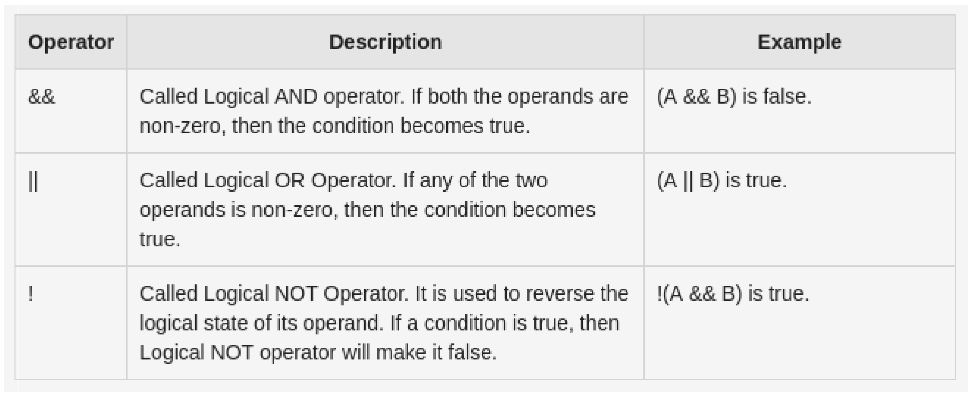
Operator	Description	Example
+	Adds two operands.	A + B = 30
−	Subtracts second operand from the first.	A − B = -10
*	Multiplies both operands.	A * B = 200
/	Divides numerator by de-numerator.	B / A = 2
%	Modulus Operator and remainder of after an integer division.	B % A = 0
++	Increment operator increases the integer value by one.	A++ = 11
--	Decrement operator decreases the integer value by one.	A-- = 9

Abb. 7.1: Arithmetic Operators (Assume A holds 10 and B holds 20)



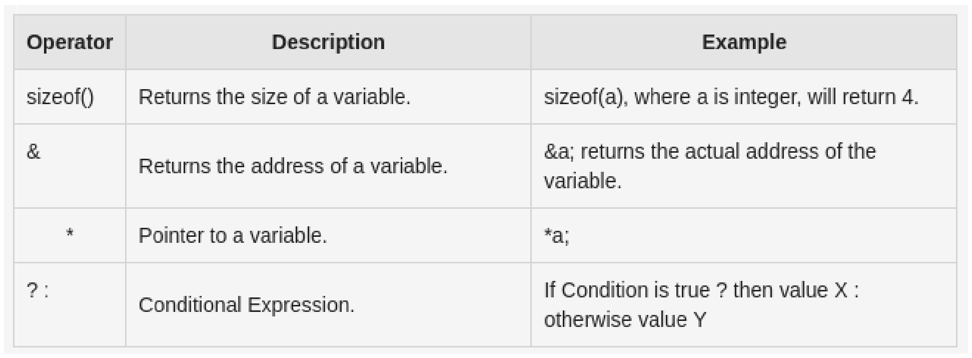
Operator	Description	Example
==	Checks if the values of two operands are equal or not. If yes, then the condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not. If the values are not equal, then the condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand. If yes, then the condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand. If yes, then the condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand. If yes, then the condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true.	(A <= B) is true.

Abb. 7.2: Relational Operators (Assume A holds 10 and B holds 20)



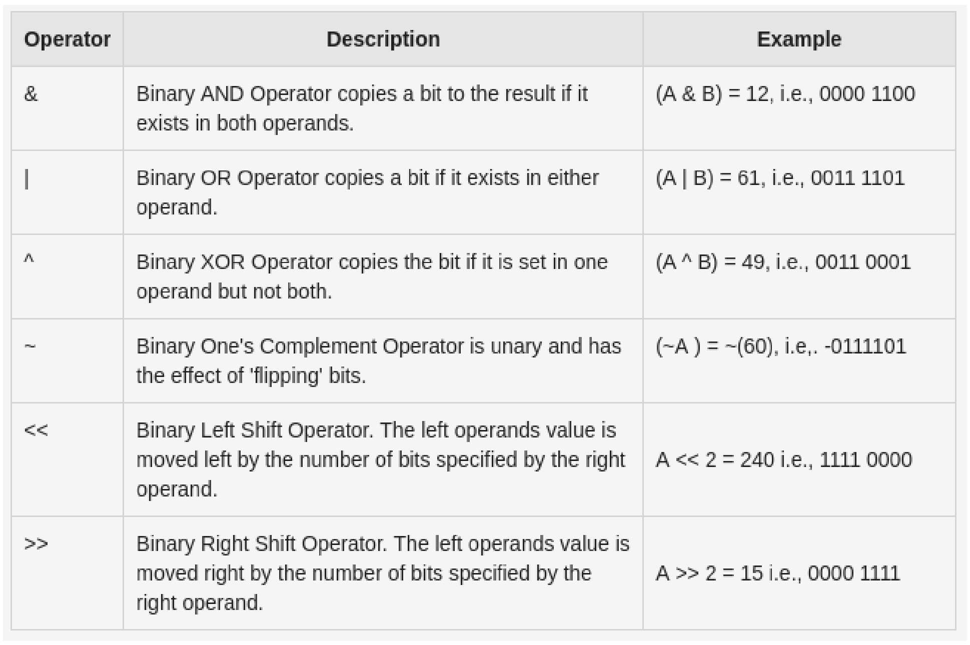
Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.	!(A && B) is true.

Abb. 7.3: Logical Operators (Assume A holds 1 and B 0)



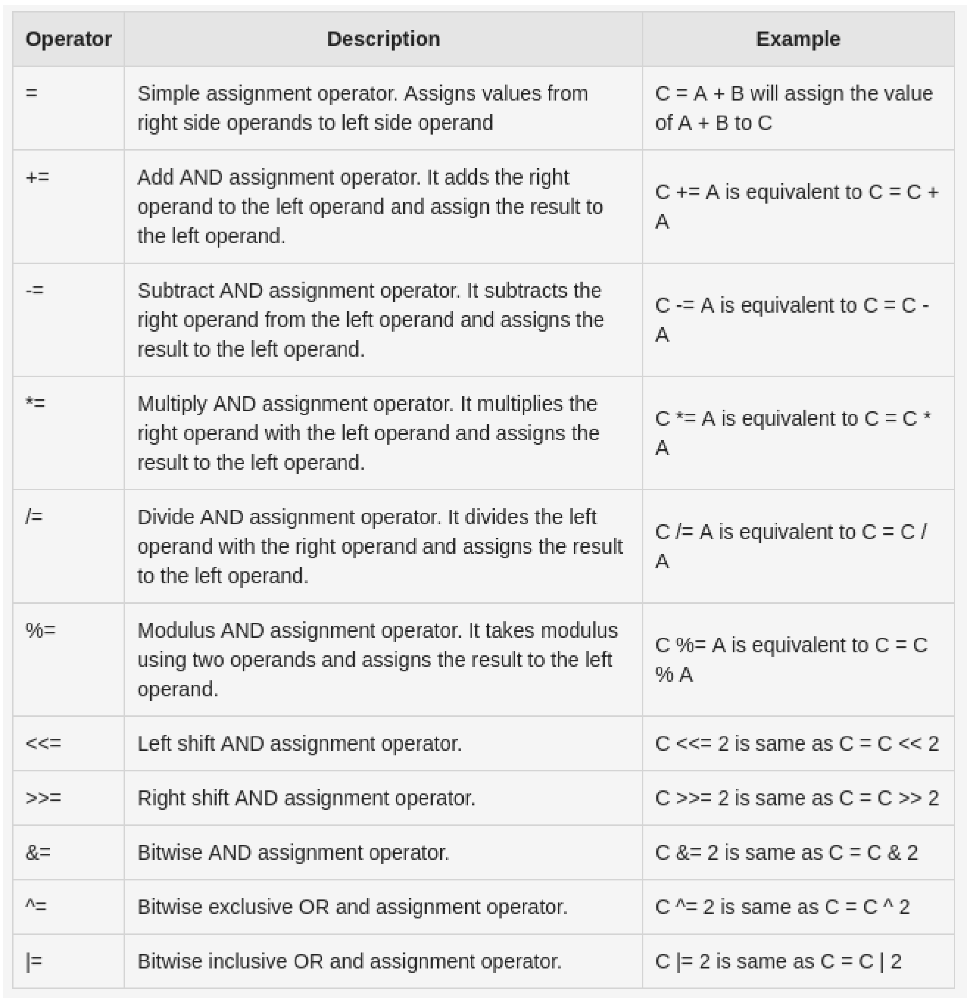
Operator	Description	Example
sizeof()	Returns the size of a variable.	sizeof(a), where a is integer, will return 4.
&	Returns the address of a variable.	&a; returns the actual address of the variable.
*	Pointer to a variable.	*a;
? :	Conditional Expression.	If Condition is true ? then value X : otherwise value Y

Abb. 7.4: Miscellaneous Operators



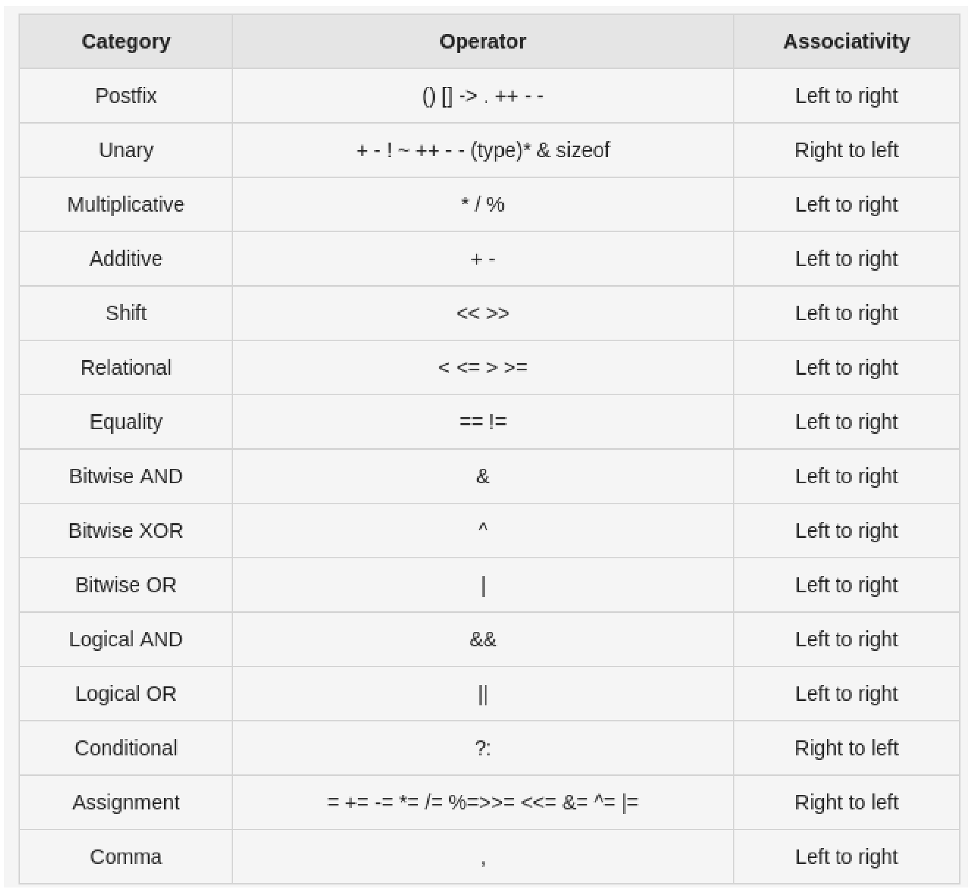
Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) = 12, i.e., 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) = 61, i.e., 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) = 49, i.e., 0011 0001
~	Binary One's Complement Operator is unary and has the effect of 'flipping' bits.	(~A) = ~(60), i.e,. -0111101
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 = 240 i.e., 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 = 15 i.e., 0000 1111

Abb. 7.5: Bitwise Operators (Assume A holds 60 and B 13)



Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand	C = A + B will assign the value of A + B to C
+=	Add AND assignment operator. It adds the right operand to the left operand and assign the result to the left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand.	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand.	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	Bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	Bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

Abb. 7.6: Assignment Operators



Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right

Abb. 7.7: Operator Precedence (Top to Bottom as importance)

8 Decision-Making

As most other programming languages, C can choose one or another path depending on how or if a condition is met:

if: Boolean expression followed by one or more statements, which are only executed if the boolean expression is true

```
if(PI ==  3.14159 ){
    printf("Pi is set correctly");
}
```

if. . . else: An if statement can have an optional 'else' block, which will only be executed if the boolean expression is false.

```
    if(PI ==  3.14159 ){
        printf("Pi is set correctly");
    } else {
#define PI 3.14159
        printf("Pi has been set to the correct value");
    }
```

nested ifs: If. . . else statements can also be nested, so that if one statement holds true, the next one gets checked etc.

switch: A variable is being tested against a list of predefined values.

```
switch(grade) {
case 'A':
    printf("Excellent! \n");
    break;
case 'B':
    printf("Good!");
    break;
default:
    printf("Invalid grade");
}
```

nestd switch: Same as if-statements, switch statements can be nested as well.

?:-Operator: Shorthand for if. . . else. `Exp1` is checked. If it is true, `Exp2` is checked and becomes the value of the entire expression. If `Exp1` is false, `Exp3` is checked and becomes the value of the entire expression

```
/* if-else */
if(a > b){
    result = x;
} else {
    result = y;
}
```

```
/* same thing with ?: */
result = a > b ? x : y;
```