EBNF ISO/IEC Standard:

Verwendung	Zeichen
Definition	=
Aufzählung	,
Endezeichen	;
Alternative	|
Option	[...]
Optionale Wiederholung	{ ... }
Gruppierung	(...)
Anführungszeichen, 1. Variante	" ... "
Anführungszeichen, 2. Variante	' ... '
Kommentar	(* ... *)
Spezielle Sequenz	? ... ?
Ausnahme	-

Beispiel: -(3-(4-5))*(3+4*5)/6

```
digit      = '0' | '1' | '2' | ... | '9'.
number     = digit {digit}.

factor     = number
           | "(" expression ")"
           | "-" factor.

term       = factor { "*" factor | "/" factor }.

expression = term { "+" term | "-" term }.
```

Operator Präzedenzen:

Precedence	Operator	Description	Associativity
17	::	Scope resolution	Left-to-right
16	a++ a--	Suffix/postfix increment and decrement	
	type() *type*{}	Functional cast	
	a()	Function call	
	a[]	Subscript	
	. ->	Member access	
15	++a --a	Prefix increment and decrement	Right-to-left
	+a -a	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(*type*)	C-style cast	
	*a	Indirection (dereference)	
	&a	Address-of	
	sizeof	Size-of[note 1]	
	co_await	await-expression (C++20)	
	new new[]	Dynamic memory allocation	
	delete delete[]	Dynamic memory deallocation	
14	.* ->*	Pointer-to-member	Left-to-right
13	a*b a/b a%b	Multiplication, division, and remainder	
12	a+b a-b	Addition and subtraction	
11	<< >>	Bitwise left shift and right shift	
10	<=>	Three-way comparison operator (since C++20)	
9	< <=	For relational operators < and ≤ respectively	
	> >=	For relational operators > and ≥ respectively	
8	== !=	For relational operators = and ≠ respectively	
7	&	Bitwise AND	
6	^	Bitwise XOR (exclusive or)	
5	|	Bitwise OR (inclusive or)	
4	&&	Logical AND	
3	||	Logical OR	
2	a?b:c	Ternary conditional[note 2]	Right-to-left
	throw	throw operator	
	co_yield	yield-expression (C++20)	
	=	Direct assignment (provided by default for C++ classes)	
	+= -=	Compound assignment by sum and difference	
	*= /= %=	Compound assignment by product, quotient, and remainder	
	<<= >>=	Compound assignment by bitwise left shift and right shift	
	&= ^= |=	Compound assignment by bitwise AND, XOR, and OR	
1	,	Comma	Left-to-right

Range of values:

Ermittelt kleinster/grösster wert des Datentyps T:

```
#inclue<limits>
std::numeric_limits<T>::min();
```

Type	Size in bits	Format	Value range: Approximate	Value range: Exact
character	8 char	signed		-128 to 127
		unsigned		0 to 255
	16	unsigned		0 to 65535
	32	unsigned		0 to 1114111 (0x10ffff)
integer	16 short int	signed	± 3.27 · 10^4	-32768 to 32767
		unsigned	0 to 6.55 · 10^4	0 to 65535
	32 long int = int	signed	± 2.14 · 10^9	-2,147,483,648 to 2,147,483,647
		unsigned	0 to 4.29 · 10^9	0 to 4,294,967,295
	64 long long int	signed	± 9.22 · 10^{18}	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
		unsigned	0 to 1.84 · 10^{19}	0 to 18,446,744,073,709,551,615
floating point	32 float	IEEE-754	• min subnormal: ± 1.401,298,4 · 10^{-45} • min normal: ± 1.175,494,3 · 10^{-38} • max: ± 3.402,823,4 · 10^{38}	• min subnormal: ±0x1p-149 • min normal: ±0x1p-126 • max: ±0x1.fffffep+127
	64 double	IEEE-754	• min subnormal: ± 4.940,656,458,412 · 10^{-324} • min normal: ± 2.225,073,858,507,201,4 · 10^{-308} • max: ± 1.797,693,134,862,315,7 · 10^{308}	• min subnormal: ±0x1p-1074 • min normal: ±0x1p-1022 • max: ±0x1.fffffffffffffp+1023

ASCII Tabelle:

Dec	Hex	Oct	Chr	Dec	Hex	Oct	HTML	Chr	Dec	Hex	Oct	HTML	Chr	Dec	Hex	Oct	HTML	Chr
0	0	000	NULL	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	Start of Header	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	Start of Text	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	End of Text	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	End of Transmission	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	Enquiry	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	Acknowledgment	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	Bell	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	Backspace	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	Horizontal Tab	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	Line feed	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	Vertical Tab	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	Form feed	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	Carriage return	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	Shift Out	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	Shift In	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	Data Link Escape	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	Device Control 1	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	Device Control 2	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	Device Control 3	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	Device Control 4	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	Negative Ack.	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	Synchronous idle	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	End of Trans. Block	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	Cancel	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	End of Medium	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	Substitute	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	Escape	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	File Separator	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	Group Separator	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	Record Separator	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	Unit Separator	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	Del

std::**vector**

class template

<vector>

```
template < class T, class Alloc = allocator<T> > class vector; // generic template
```

Vector

Vectors are sequence containers representing arrays that can change in size.

Member functions

(constructor)	Construct vector (public member function)
(destructor)	Vector destructor (public member function)
operator=	Assign content (public member function)

Iterators:

begin	Return iterator to beginning (public member function)
end	Return iterator to end (public member function)
rbegin	Return reverse iterator to reverse beginning (public member function)
rend	Return reverse iterator to reverse end (public member function)
cbegin (c++0)	Return const_iterator to beginning (public member function)
cend (c++0)	Return const_iterator to end (public member function)
crbegin (c++0)	Return const_reverse_iterator to reverse beginning (public member function)
crend (c++0)	Return const_reverse_iterator to reverse end (public member function)

Capacity:

size	Return size (public member function)
max_size	Return maximum size (public member function)
resize	Change size (public member function)
capacity	Return size of allocated storage capacity (public member function)
empty	Test whether vector is empty (public member function)
reserve	Request a change in capacity (public member function)
shrink_to_fit (c++0)	Shrink to fit (public member function)

Element access:

operator[]	Access element (public member function)
at	Access element (public member function)
front	Access first element (public member function)
back	Access last element (public member function)
data (c++0)	Access data (public member function)

Modifiers:

assign	Assign vector content (public member function)
push_back	Add element at the end (public member function)
pop_back	Delete last element (public member function)
insert	Insert elements (public member function)
erase	Erase elements (public member function)
swap	Swap content (public member function)
clear	Clear content (public member function)
emplace (c++0)	Construct and insert element (public member function)
emplace_back (c++0)	Construct and insert element at the end (public member function)

header

<cmath> (math.h)

C numerics library

Header <cmath> declares a set of functions to compute common mathematical operations and transformations:

Functions

Trigonometric functions

cos	Compute cosine (function)
sin	Compute sine (function)
tan	Compute tangent (function)
acos	Compute arc cosine (function)
asin	Compute arc sine (function)
atan	Compute arc tangent (function)

Exponential and logarithmic functions

exp	Compute exponential function (function)
log	Compute natural logarithm (function)
log10	Compute common logarithm (function)

Power functions

pow	Raise to power (function)
sqrt	Compute square root (function)

Rounding and remainder functions

ceil	Round up value (function)
floor	Round down value (function)
abs	Compute absolute value (function)

Usual arithmetic conversions

The arguments of the following arithmetic operators undergo implicit conversions for the purpose of obtaining the *common real type*, which is the type in which the calculation is performed:

- binary arithmetic *, /, %, +, -
- relational operators <, >, <=, >=, ==, !=
- binary bitwise arithmetic &, ^, |,
- the conditional operator ?:

1) If one operand is `long double`, `long double complex`, or `long double imaginary`, the other operand is implicitly converted as follows:
   - integer or real floating type to `long double`
   - complex type to `long double complex`
   - imaginary type to `long double imaginary`

2) Otherwise, if one operand is `double`, `double complex`, or `double imaginary`, the other operand is implicitly converted as follows:
   - integer or real floating type to `double`
   - complex type to `double complex`
   - imaginary type to `double imaginary`

3) Otherwise, if one operand is `float`, `float complex`, or `float imaginary`, the other operand is implicitly converted as follows:
   - integer type to `float` (the only real type possible is float, which remains as-is)
   - complex type remains `float complex`
   - imaginary type remains `float imaginary`

4) Otherwise, both operands are integers. Both operands undergo *integer promotions* (see below); then, after integer promotion, one of the following cases applies:
   - If the types are the same, that type is the common type.
   - Else, the types are different:
     - If the types have the same signedness (both signed or both unsigned), the operand whose type has the lesser *conversion rank*[1] is implicitly converted[2] to the other type.
     - Else, the operands have different signedness:
       - If the unsigned type has *conversion rank* greater than or equal to the rank of the signed type, then the operand with the signed type is implicitly converted to the unsigned type.
     - Else, the unsigned type has *conversion rank* less than the signed type:
       - If the signed type can represent all values of the unsigned type, then the operand with the unsigned type is implicitly converted to the signed type.
       - Else, both operands undergo implicit conversion to the unsigned type counterpart of the signed operand's type.

Bsp. switch case

```
int x = 2;
switch (x)
{
    case 1: printf("Choice is 1");
            break;
    case 2: printf("Choice is 2");
             break;
    case 3: printf("Choice is 3");
            break;
    default: printf("Choice other than 1, 2 and 3");
             break;
}
```

Fliesskommazahlensysteme

Ein Fliesskommazahlensystem ist durch vier natürliche Zahlen definiert:

- $\beta \geq 2$, die Basis,
- $p \geq 1$, die Präzision (Stellenzahl),
- $e_{\min}$, der kleinste Exponent,
- $e_{\max}$, der grösste Exponent.

Bezeichnung:

$F(\beta, p, e_{\min}, e_{\max})$ enthält die Zahlen

$$\pm \sum_{i=0}^{p-1} d_i \beta^{-i} \cdot \beta^e,$$

$$d_i \in \{0, \ldots, \beta - 1\}, \quad e \in \{e_{\min}, \ldots, e_{\max}\}.$$

In Basis-β-Darstellung:

$$\pm\, d_0{\bullet}d_1 \ldots d_{p-1} \times \beta^e,$$

Normalisierte Zahl:

$$\pm\, d_0{\bullet}d_1 \ldots d_{p-1} \times \beta^e, \qquad d_0 \neq 0$$

Der IEEE Standard 754

legt Fliesskommazahlensysteme und deren Rundungsverhalten fest und wird fast überall benutzt

- Single precision (`float`) Zahlen:

  $F^*(2, 24, -126, 127)$ (32 bit) plus $0, \infty, \ldots$

- Double precision (`double`) Zahlen:

  $F^*(2, 53, -1022, 1023)$ (64 bit) plus $0, \infty, \ldots$

- Alle arithmetischen Operationen runden das *exakte* Ergebnis auf die nächste darstellbare Zahl

Umrechnung dezimal → binär

Angenommen, $0 < x < 2$.

- Also: $x' = b_{-1}{\bullet}b_{-2}b_{-3}b_{-4} \ldots = 2 \cdot (x - b_0)$
- Schritt 1 (für x): Berechnen von b_0:

$$b_0 = \begin{cases} 1, & \text{falls } x \geq 1 \\ 0, & \text{sonst} \end{cases}$$

- Schritt 2 (für x): Berechnen von $b_{-1}, b_{-2}, \ldots$:

  Gehe zu Schritt 1 (für $x' = 2 \cdot (x - b_0)$)

Bsp. Binärdarstellung von 1.1_{10}

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2
0.2	$b_1 = 0$	0.2	0.4
0.4	$b_2 = 0$	0.4	0.8
0.8	$b_3 = 0$	0.8	1.6
1.6	$b_4 = 1$	0.6	1.2
1.2	$b_5 = 1$	0.2	0.4

Operator Rundung

- Division '`/`' von `int` und `unsigned int` wird zu 0 gerundet.
- Modulo '`%`', rest der Ganzzahldivision, bei negative Zahlen übernimmt '`%`' das Vorzeichen des linken Operanden.

Literale (Bsp. 42)

- `int`: `42`
- `unsigned int`: `42u`
- `float`: `42.0f`
- `double`: `42.0`

Basen (Bsp. 42)

- Potenz: `4.2e+1`
- Dezimal: `42`
- Binär: `0b101010`
- Oktal: `052`
- `Hexadezimal: 0x2A`

Hexadezimal:	256	16	1

10	11	12	13	14	15
A	B	C	D	E	F

Oktal:	512	64	8	1

Binär:	128	64	32	16	8	4	2	1	0.5	0.25	0.125

Arithmetic operators

Returns the result of specific arithmetic operation.

Operator name	Syntax	Overloadable	Prototype examples (for `class T`)	
			Inside class definition	Outside class definition
unary plus	`+a`	Yes	`T T::operator+() const;`	`T operator+(const T &a);`
unary minus	`-a`	Yes	`T T::operator-() const;`	`T operator-(const T &a);`
addition	`a + b`	Yes	`T T::operator+(const T2 &b) const;`	`T operator+(const T &a, const T2 &b);`
subtraction	`a - b`	Yes	`T T::operator-(const T2 &b) const;`	`T operator-(const T &a, const T2 &b);`
multiplication	`a * b`	Yes	`T T::operator*(const T2 &b) const;`	`T operator*(const T &a, const T2 &b);`
division	`a / b`	Yes	`T T::operator/(const T2 &b) const;`	`T operator/(const T &a, const T2 &b);`
modulo	`a % b`	Yes	`T T::operator%(const T2 &b) const;`	`T operator%(const T &a, const T2 &b);`

Assignment operators

Assignment operators modify the value of the object.

Operator name	Syntax	Overloadable	Prototype examples (for `class T`)	
			Inside class definition	Outside class definition
simple assignment	`a = b`	Yes	`T& T::operator =(const T2& b);`	N/A
addition assignment	`a += b`	Yes	`T& T::operator +=(const T2& b);`	`T& operator +=(T& a, const T2& b);`
subtraction assignment	`a -= b`	Yes	`T& T::operator -=(const T2& b);`	`T& operator -=(T& a, const T2& b);`
multiplication assignment	`a *= b`	Yes	`T& T::operator *=(const T2& b);`	`T& operator *=(T& a, const T2& b);`
division assignment	`a /= b`	Yes	`T& T::operator /=(const T2& b);`	`T& operator /=(T& a, const T2& b);`
modulo assignment	`a %= b`	Yes	`T& T::operator %=(const T2& b);`	`T& operator %=(T& a, const T2& b);`

Increment/decrement operators

Increment/decrement operators increment or decrement the value of the object.

Operator name	Syntax	Overloadable	Prototype examples (for `class T`)	
			Inside class definition	Outside class definition
pre-increment	`++a`	Yes	`T& T::operator++();`	`T& operator++(T& a);`
pre-decrement	`--a`	Yes	`T& T::operator--();`	`T& operator--(T& a);`
post-increment	`a++`	Yes	`T T::operator++(int);`	`T operator++(T& a, int);`
post-decrement	`a--`	Yes	`T T::operator--(int);`	`T operator--(T& a, int);`

Comparison operators

Compares the arguments.

Operator name	Syntax	Overloadable	Prototype examples (for `class T`)	
			As member function	As free (namespace) function
equal to	`a == b`	Yes	`bool T::operator ==(const T2 &b) const;`	`bool operator ==(const T &a, const T2 &b);`
not equal to	`a != b`	Yes	`bool T::operator !=(const T2 &b) const;`	`bool operator !=(const T &a, const T2 &b);`
less than	`a < b`	Yes	`bool T::operator <(const T2 &b) const;`	`bool operator <(const T &a, const T2 &b);`
greater than	`a > b`	Yes	`bool T::operator >(const T2 &b) const;`	`bool operator >(const T &a, const T2 &b);`
less than or equal to	`a <= b`	Yes	`bool T::operator <=(const T2 &b) const;`	`bool operator <=(const T &a, const T2 &b);`
greater than or equal to	`a >= b`	Yes	`bool T::operator >=(const T2 &b) const;`	`bool operator >=(const T &a, const T2 &b);`

Member access operators

Accesses a member of its operand.

Operator name	Syntax	Overloadable	Prototype examples (for `class T`)	
			Inside class definition	Outside class definition
subscript	`a[b]`	Yes	`R& T::operator[](S b);`	N/A
indirection	`*a`	Yes	`R& T::operator*();`	`R& operator*(T a);`
address-of	`&a`	Yes	`R* T::operator&();`	`R* operator&(T a);`
member of object	`a.b`	No	N/A	N/A
member of pointer	`a->b`	Yes	`R* T::operator->();`	N/A
pointer to member of object	`a.*b`	No	N/A	N/A
pointer to member of pointer	`a->*b`	Yes	`R& T::operator->*(S b);`	`R& operator->*(T a, S b);`