

Language C++

Explanations

xyz	terminal “xyz” (use these characters or words verbatim)
<i>abc</i>	meta symbol: replace “ <i>abc</i> ” by definition of <i>abc</i>
\doteq	definition symbol (if applicable also within an alternative) —in contrast terminal equal sign: =
	meta symbol for alternative—in contrast terminal vertical bar:
[]	meta bracket symbols—in contrast terminal square brackets: []
<i>abc</i> _{0..n}	zero or more times “ <i>abc</i> ”, sequence of an arbitrary number of “ <i>abc</i> ” (also zero)
<i>abc</i> _{1..n}	one or more times “ <i>abc</i> ”, sequence with at least one “ <i>abc</i> ”
<i>abc</i> _{opt}	optional “ <i>abc</i> ” (same meaning as: <i>abc</i> _{0..1})
$\left[\begin{array}{l} abc \\ def \end{array} \right]$	equivalent to: [<i>abc</i> <i>def</i>]
Op2 ^{Op2}	reference to operator (hierarchy level)
¹⁰	reference to syntax element
[NC]	definition not complete
[Forb]	forbidden, not allowed (in this course)
<text>	special term in German
[...]	explanation (not belonging to syntax)

Language notes:

(*nothing*) in C and C++

C++ only in C++

C only in C

C/C++ in C; also possible and usual in C++ (mostly used as contrast to **C++**)

C (C++) in C; in C++ possible, too, but not recommended: in C++ better constructs available

C++(new) C++, new, perhaps not yet implemented in current compilers

C++(old) C++, old, perhaps still implemented in current compilers

1 Syntax (selection)

1.1 Complete selection

xxx⁻¹*LIST* \doteq *xxx* [, *xxx*]_{0..n}

TranslationUnit (*source file*) \doteq ¹⁰*Declaration*_{0..n}

¹⁰*Declaration* [NC] \doteq
¹¹*SimpleDeclaration* | ³³*FunctionDefinition* | **C++(new)** ⁴⁰*NamespaceDefinition*

¹¹*SimpleDeclaration* \doteq
¹²*DeclSpecifier*_{0..n} [³⁰*Declarator* [Decl.-Point] *Initializer*_{opt}]⁻¹*LIST*_{opt} ;

¹²*DeclSpecifier* \doteq
¹³*StorageClassSpecifier* | ¹⁴*TypeSpecifier* | typedef |
C++ ¹⁵*FunctionSpecifier* | **C++** friend

¹³*StorageClassSpecifier* \doteq auto | register | static | extern | **C++(new)** mutable

¹⁴*TypeSpecifier* [NC] \doteq
char | short | int | long | signed | unsigned | float | double | void |
C++(new) bool | ²¹*ClassSpecifier* | ²³*EnumSpecifier* | ^{32a}⊗_{opt} ²⁰*TypeName* |
²⁵*CvQualifier*

¹⁵*FunctionSpecifier* **C++** \doteq inline | virtual | **C++(new)** explicit

²⁰*TypeName* \doteq *ClassName* | *EnumName* | *TypedefName*

²¹*ClassSpecifier* [NC] \doteq
C/C++ [**struct** | **union**] *Identifier*_{opt} { *MemberDeclaration*_{0..n} } |
C++ [**class** | **struct** | **union**] *ClassName*_{opt} [: *BaseSpecifier*⁻¹*LIST*]_{opt}
{ [²²*AccessSpecifier* : | *MemberDeclaration*]_{0..n} }

²²*AccessSpecifier* **C++** \doteq private | protected | public

²³*EnumSpecifier* \doteq enum *EnumName*_{opt} { ²⁴*EnumeratorDefinition*⁻¹*LIST*_{opt} }

²⁴*EnumeratorDefinition* \doteq *Enumerator* [= *ConstantExpressionIntegralType*]_{opt}

²⁵*CvQualifier* \doteq const | volatile

³⁰*Declarator* \doteq ³²*DirectDeclarator* | ³¹*Pointeroperator* ³⁰*Declarator*

³¹*Pointeroperator* [NC] \doteq * ²⁵*CvQualifier*_{0..n} | **C++** &

³²*DirectDeclarator* [NC] \doteq
^{32a}⊗_{opt} *Identifier* | (³⁰*Declarator*) | ³²*DirectDeclarator* [*ConstantExpression*_{opt}] |

$\boxed{\text{C/C++}}$ ³² *DirectDeclarator* (*ParameterDeclaration*-¹*LIST*_{opt}) |
 $\boxed{\text{C++}}$ ³² *DirectDeclarator* (*ParameterDeclaration*-¹*LIST*_{opt}) ²⁵ *CvQualifier*_{0..n} |
 $\boxed{\text{C++}}$ ^{32a} \otimes_{opt} *OperatorFunctionName* \doteq **operator** *Operator* |
 $\boxed{\text{C++}}$ ^{32a} \otimes_{opt} *ConversionFunctionName* \doteq **operator** ¹⁴ *TypSpecifier*_{1..n} ³¹ *Pointeroperator*_{0..n} |
 $\boxed{\text{C++}}$ ^{32a} \otimes_{opt} \sim *ClassName*
^{32a} \otimes (*NestedNameSpecifier*) _[NC] \doteq [*ClassName* ::]_{1..n}
³³ *FunctionDefinition* \doteq
 $\boxed{\text{C/C++}}$ ¹² *DeclSpecifier*_{0..n} ³⁰ *Declarator* ⁵⁴ *CompoundStatement* |
 $\boxed{\text{C++}}$ ¹² *DeclSpecifier*_{0..n} ³⁰ *Declarator* [: *MemberInitializer*-¹*LIST*]_{opt} ⁵⁴ *CompoundStatement*
⁴⁰ *NamespaceDefinition* $\boxed{\text{C++(new)}}$ \doteq **namespace** *NamespaceName*_{opt} { ¹⁰ *Declaration*_{0..n} }
⁵⁰ *Statement* \doteq
⁵¹ *ExpressionStatement* | ⁵² *LabelledStatement* | ⁵³ *JumpStatement* |
⁵⁴ *CompoundStatement* | ⁵⁵ *SelectionStatement* | ⁵⁶ *IterationStatement* |
 $\boxed{\text{C++}}$ ⁵⁷ *DeclarationStatement*
⁵¹ *ExpressionStatement* \doteq *Expression*_{opt} ;
⁵² *LabelledStatement* \doteq
case *ConstantExpression* : ⁵⁰ *Statement* | **default** : ⁵⁰ *Statement* |
_[Forb] *Identifier* : ⁵⁰ *Statement*
⁵³ *JumpStatement* \doteq
break ; | **continue** ; | **return** *Expression*_{opt} ; | _[Forb] **goto** *Identifier* ;
⁵⁴ *CompoundStatement (Block)* \doteq
 $\boxed{\text{C++}}$ { ⁵⁰ *Statement*_{0..n} } | $\boxed{\text{C/C++}}$ { ¹¹ *SimpleDeclaration*_{0..n} ⁵⁰ *Statement*_{0..n} }
⁵⁵ *SelectionStatement* \doteq ⁶⁰ *if-statement* | ⁶¹ *switch-statement*
⁵⁶ *IterationStatement* \doteq ⁷⁰ *while-statement* | ⁷¹ *do-while-statement* | ⁷² *for-statement*
⁵⁷ *DeclarationStatement* $\boxed{\text{C++}}$ _[NC] \doteq ¹¹ *SimpleDeclaration*
⁶⁰ *if-statement* \doteq **if** (⁹⁹ *Condition*) ⁵⁰ *Statement* [**else** ⁵⁰ *Statement*]_{opt}
⁶¹ *switch-statement* \doteq **switch** (⁹⁹ *Condition*) *Statement*
Note: Statement normally as ⁵⁴ *CompoundStatement*, **within it several times** ⁵² *LabelledStatement*
⁷⁰ *while-statement* \doteq **while** (⁹⁹ *Condition*) ⁵⁰ *Statement*
⁷¹ *do-while-statement* \doteq **do** ⁵⁰ *Statement* **while** (*Expression*) ;
⁷² *for-statement* \doteq
 $\boxed{\text{C/C++}}$ **for** (*Expression1*_{opt} ; *Expression2*_{opt} ; *Expression3*_{opt}) ⁵⁰ *Statement* |
 $\boxed{\text{C++}}$ **for** (¹¹ *SimpleDeclaration1* ⁹⁹ *Condition2*_{opt} ; *Expression3*_{opt}) ⁵⁰ *Statement*
Summarized $\boxed{\text{C++}}$ (**Note: if** $\boxed{\text{C++(new)}}$, **note to number** ⁹⁹ **is also valid for SimpleDeclaration**):
for ([⁵¹ *ExpressionStatement1*] ⁹⁹ *Condition2*_{opt} ; *Expression3*_{opt}) ⁵⁰ *Statement*
Note: 1 Initialization, 2 Loop entry condition (missing: true), 3 Reinitialization
⁹⁹ *Condition* \doteq *Expression* | $\boxed{\text{C++(new)}}$ ¹⁴ *TypeSpecifier*_{1..n} ³⁰ *Declarator* = *Expression*
Note: 2nd alternative $\boxed{\text{C++(new)}}$: **Variable scope only within the control structure (statement** ^{60,61,70,72} **).**

1.2 Syntax selection for 1st semester

xxx-¹*LIST* \doteq *xxx* [, *xxx*]_{0..n}

Translation unit (source file) \doteq ¹⁰ *Declaration*_{0..n}

¹⁰ *Declaration* \doteq ¹¹ *SimpleDeclaration* | ³³ *FunctionDefinition* | $\boxed{\text{C++(new)}}$ ⁴⁰ *NamespaceDefinition*

¹¹ *SimpleDeclaration* \doteq
¹² *DeclSpecifier*_{0..n} [³² *DirectDeclarator* _[Decl.-Point] *Initializer*_{opt}]-¹*LIST*_{opt} ;

¹² *DeclSpecifier* \doteq
char | **short** | **int** | **long** | **signed** | **unsigned** | **float** | **double** | **void** |
bool | *TypedefName* | **typedef**

³²*DirectDeclarator* \doteq
Identifier | (³²*DirectDeclarator*) | ³²*DirectDeclarator* [*ConstantExpression_{opt}*] |
³²*DirectDeclarator* (*ParameterDeclaration*-¹*LIST_{opt}*)

³³*FunctionDefinition* \doteq
¹²*DeclSpezifizierer_{0..n}* ³²*DirectDeclarator* ⁵⁴*CompoundStatement*

⁴⁰*NamespaceDefinition* C++(new) \doteq namespace *NamespaceName_{opt}* { ¹⁰*Declaration_{0..n}* }

⁵⁰*Statement* \doteq
⁵¹*ExpressionStatement* | ⁵²*LabelledStatement* | ⁵³*JumpStatement* |
⁵⁴*CompoundStatement* | ⁵⁵*SelectionStatement* | ⁵⁶*IterationStatement* |
⁵⁷*DeclarationStatement*

⁵¹*ExpressionStatement* \doteq *Expression_{opt}* ;

⁵²*LabelledStatement* \doteq case *ConstantExpression* : ⁵⁰*Statement* | default : ⁵⁰*Statement*

⁵³*JumpStatement* \doteq break ; | continue ; | return *Expression_{opt}* ;

⁵⁴*CompoundStatement (Block)* \doteq { ⁵⁰*Statement_{0..n}* }

⁵⁵*SelectionStatement* \doteq ⁶⁰*if-statement* | ⁶¹*switch-statement*

⁵⁶*IterationStatement* \doteq ⁷⁰*while-statement* | ⁷¹*do-while-statement* | ⁷²*for-statement*

⁵⁷*DeclarationStatement* \doteq ¹¹*SimpleDeclaration*

⁶⁰*if-statement* \doteq if (*Expression*) ⁵⁰*Statement* [else ⁵⁰*Statement*]_{opt}

⁶¹*switch-statement* \doteq switch (*Expression*) ⁵⁴*CompoundStatement*

⁷⁰*while-statement* \doteq while (*Expression*) ⁵⁰*Statement*

⁷¹*do-while-statement* \doteq do ⁵⁰*Statement* while (*Expression*) ;

⁷²*for-statement* \doteq for (*Expression1_{opt}* ; *Expression2_{opt}* ; *Expression3_{opt}*) ⁵⁰*Statement*
Note: 1 Initialization, 2 Loop entry condition (missing: true), 3 Reinitialization

2 Operators

Levels 1 to 17: decreasing precedence Associativity: \longrightarrow ; exceptions (“@”): \longleftarrow

1	a	::	C++	global	6	ab	+ -	addition, subtraction
1	b	::	C++	scope resolution	7	ab	<< >>	bit shift
2	a	[]		index	8	abcd	< <= > >=	comparison
	b	()		function call	9	ab	== !=	(un-)equality
	c	()	C++	conv. SimpleTypeName	10		&	bitwise And
	de	. ->		member access	11		^	bitwise exclusive Or
	fg	++ --		postfix increment/decr.	12			bitwise inclusive Or
	h-k	<i>kind_{cast}<>()</i>	C++(new)	typ conv., s.below	13		&&	logical And
i	<i>typeid</i>	C++(new)	type identif.	14			logical inclusive Or	
3@	ab	++ --		prefix increment/decr.	15@		? :	conditional operator
	unary c	~		bit inversion	ternary			
	prefix d	!		logical negation	16@	a	=	(simple) assignment
	ef	+ -		sign		bcde	*= /= %= +=	compound assignment
	g	&		address		fgh	-- >>= <<=	
	h	*		dereference		ijk	&= ^= =	
	i	()		type conversion	17		,	comma operator
	j	sizeof		memory space				
kl	new delete	C++	free store					
4	ab	->* .*	C++	member access				
5	a	*		multiplication				
	b	/		division				
	c	%		remainder				

Not overloadable: :: . sizeof .* ?:
Op2h-k C++(new): *kind_{cast}<>()* type conv.:
kind \doteq static | dynamic | const | reinterpret

3 Standard streams C++

Note for C (C++): Library function families from `<cstdio>`
output: `printf/puts/putchar`, input: `scanf/gets/getchar`

3.1 Special expression statements (<iostream>)

—value of the whole expressions (main effect): stream object reference (see note)—

Output into standard output unit \doteq `cout [<< Expression]0..n ;`

Output into error output unit \doteq `[cerr | clog] [<< Expression]0..n ;`

(in UNIX redirectable, in DOS not, in Windows possibly)

Note: `C++` `cerr`, `C++(new)` `clog`

Input from standard input unit \doteq `cin [>> Variable]0..n ;`

Read with operator >>: discard leading WhSpaces (white spaces: Space, Tab, CR, LF, NewPage, vertTab), read until (excluding) next inappropriate character (string: WhSpace) – in contrast reading all characters see (3.3)

Number of characters currently available in the buffer of `cin`: `cin.rdbuf()->in_avail()`

Note: stream objects (e. g. `cin`, `cout`) or their references in the case of Boolean evaluation:

`C++(new)` `false` if error/endoffile, otherwise `true` (or `C++(old)` pointer 0, otherwise pointer!=0);
in the case of `false` (0): before continuing, absolutely necessary to call member function `clear()`!

3.2 Formatting (mostly for output)

All settings mentioned remain fixed until next alteration (exception: `setw/width`)

<u>manipulator</u>	<u>member function</u>	
<code>endl</code>	—	NewLine+FlushBuffer
<code>ends</code>	—	Nullbyte+FlushBuffer
<code>flush</code>	<code>flush</code> [⊗]	FlushBuffer
<code>dec</code> , <code>oct</code> , <code>hex</code>	—	set base for integer numbers (default: <code>dec</code>)
<code>setfill(int)</code> [⊙]	<code>fill(char)</code> ^{△▽}	fill character (default: <code>'␣'</code>)
<code>setw(int)</code> [⊙]	<code>width(int)</code> ^{△▽}	minimum output width; automatic reset to default 0
<code>setprecision(int)</code> [⊙]	<code>precision(int)</code> ^{△▽}	default: 6
if <code>ios::scientific/fixed</code> set:		number of decimals after point
both not set		number of leading digits, in total
– Further manipulators see (3.4): same name as flags, additionally partly also negation –		

Manipulation of the flags, see (3.4)

<code>setiosflags(long int)</code> [⊙]	<code>setf(long int)</code> [▽]	set only the 1 bit locations
—	<code>setf(val,mask)</code> [▽]	change the 1 bit locations of <code>mask</code> into values of <code>val</code> (both <code>long int</code> ; s. also (3.4))
<code>resetiosflags(long int)</code> [⊙]	<code>unsetf(long int)</code> [▽]	clear only the 1 bit locations
—	<code>flags(long int)</code> ^{△▽}	complete adoption of the bits

Notes: [⊙] <iomanip>, all others: <iostream>; [△] call without parameter: return current value; [▽] call with parameter: set new value, return old value; [⊗] return stream object reference

3.3 Special functions (as member functions) – for reading all characters, also WhSpaces –

Return value of the three following functions: stream object reference, see (3.1 note)

`get(char &character)` read character into `character` – also umlauts/“ß”

`getline(char *target,int num,char end='\n')`

`get(char *target,int num,char end='\n')` [⚠] no successive use of this kind of `get`!

read string into `target`, guaranteed null byte termination;

read maximum `num-1` characters (less, if `end` or EOF is encountered

before); `end` character is discarded (`getline`) or not discarded (`get`);

in neither case, the `end` character is copied to `target`.

`get()` [⚠] returns character as type `int`, EOF if error/endoffile; pay attention if umlauts/“ß”!!

3.4 Important flags (enumeration constants of the class `ios`)

All settings remain fixed until next alteration; bitwise inclusive Or^{Op12} combination of the flags possible

`right*`, `left*`, `internal*` Positioning right adjusted (default), left adjusted,
set fill character between sign and number; mask `adjustfield`[▽]

`dec*`, `oct*`, `hex*` integer number base, default: `dec`; mask `basefield`[▽]

`scientific*`, `fixed*` floating point, fixed point notation; default: `noth.`; mask `floatfield`[▽]

`showbase`[⊗] output integer number base (prefix in C/C++ notation)

`showpoint`[⊗] closing zeros (in any case with `scientific/fixed`)

`showpos`[⊗] output also plus sign if positive value

`uppercase`[⊗] capital letters in hex./exp. notation (otherwise lowercase)

`boolalpha`[⊗] `C++(new)` symbolic Boolean values (default: 0, 1) – also for input

`skipws`[⊗] discard WhSpaces (default)

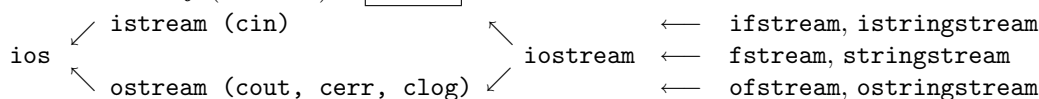
Notes: * max. 1 bit set; also as manipulator (3.2), then implicit clearing of conflicting flags; [⊗] also as

manipulator (3.2), additional manipulator form `no...` (clear), both perhaps `C++(new)`; ∇ mask name (class ios) for overload of `setf` with par. `mask` (3.2)

4 General properties of streams `C++`

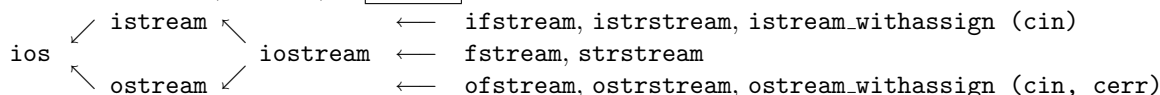
Class hierarchy (selection)

`C++(new)`



Class hierarchy (selection)

`C++(old)`



Selection of *member functions*, *enumeration constants* (all constants from ios, i. e. `ios::constant`)

Declarations in `<iostream>`, File handling in `<fstream>`, param. not always shown

Class ios:

stream state: *goodbit*, *eofbit*¹⁾, *failbit*²⁾, *badbit*³⁾
 logical: *good*, *eof*, *fail* (*failbit*∨*badbit*), *bad*; reset: *clear*⁴⁾, bits: *rdstate*
 logical test (*StreamObj*) or (!*StreamObj*) equiv. to !*fail* or *fail*

formatting and other flags see (3.2), (3.4)

Class istream: *getline*⁵⁾, *putback*(char)⁶⁾, *ignore*(cnt=1, end=EOF)⁷⁾, *peek*⁸⁾
*operator>>*⁹⁾, *get*⁵⁾, *read*(char*, size_t)¹⁰⁾, *tellg*¹¹⁾, *seekg*¹¹⁾

Class ostream: *operator<<*⁹⁾, *put*, *write*(char*, size_t)¹⁰⁾, *tellp*¹¹⁾, *seekp*¹¹⁾,
flush

Classes ifstream, ofstream, fstream:

File mode(*ios*):*in* (default for *istream*), *out* (default for *ostream*),
*ate*¹²⁾, *app*¹³⁾, *trunc*¹⁴⁾, *binary* (binary mode; default: text mode)
open(*FileName*)¹⁵⁾, *open*(*FileName*, *Mode*)¹⁵⁾, *close*¹⁶⁾

Notes: ¹⁾ Returns stream object reference (except *get* if without par.), s. (3.3); ¹⁾ additionally *failbit* is set;
²⁾ most recent operation not successful, continuation after *clear* presumably possible; ³⁾ stream is corrupted, continuation hardly sensible; ⁴⁾ without par. clearing all error bits (NECESSARY after error), otherwise with par. (*newBitState*); ⁵⁾ see (3.3); ⁶⁾ additionally also `C++(new)` *unget* without par.: most recently read character; ⁷⁾ *end* character (par. type *int*) is discarded, too, if appl.; ⁸⁾ type *int* (as *get* without par., see (3.3)), character remains in stream; ⁹⁾ with many overloads; ¹⁰⁾ also well applicable for binary files (set flag *binary*); ¹¹⁾ *...p* (*put*), *...g* (*get*), *streampos tell...* returns current position, *seek...* (*tell...-value*) absolute positioning, *seek...* (*Offset*, *Ref*) relative positioning with *Ref beg*, *cur*, *end* (*beg*: as abs. positioning), pointer positions *...g* and *...p* are identical when same buffer; ¹²⁾ after opening, position to endoffile (“at end”); ¹³⁾ write only at endoffile; ¹⁴⁾ deletes file if *out*∧¬(*ate*∨*app*); ¹⁵⁾ parameter of both *open* overloads also possible already in constructor, possibly not with *fstream*; ¹⁶⁾ also implicitly by destructor

5 Scope, storage class, linkage

5.1 Scope (Gültigkeitsbereich) for names:

- local (name called local/internal): from decl. point (within block) until equiv. block end, ⁵⁵*Selection*/⁵⁶*IterationStatement* with ⁹⁹*Condition*: `C++(new)` compl. statement implicitly blocked
- global (name called global/external): from decl. point (outside block) until program file end, `C++` access despite hiding possible with unary prefix operator `::`^{Op1}
- function: only for labels⁵² _[Forb]
- class: class members

Important: a name is hidden by a new declaration of the same name in subordinate blocks.

Note: Name of enum/class type and name of *fcn./var.* in same scope level possible; in this case, enum./cl. type name is hidden, however accessible with `enum/class/struct/union TypeName` (`[C]`): only in this way because “tag” (Etikett) with own name space).

5.2 Storage class (Speicherklasse): static, automatic; addit. also: dyn. controlled (*new*, *delete*)

– NOT to be mixed up with ¹³*StorageClassSpecifier* –

	global name	local name - with <code>static</code> ¹⁾ -	local name - without <code>static</code> ²⁾ -
storage class		static	automatic
lifetime		program	block
implicit initializ.		yes (0 or ³⁾)	no or ³⁾
explicit initializ.		with const. expr. ⁴⁾	with arbitr. expr.
- how often		once	each time
scope	global	local	local

Notes:
 1) or also `extern`, but then in gen. only declaration
 2) missing or `auto` or `register`
 3) if objects: default constr.
 4) `C++(new)` also non constant

5.3 Linkage (Bindung): external, internal, none

decl. specifier	global name	local name
none	external l. ¹⁾	no l.
<code>extern const_{opt}</code>	external l. ²⁾	external l. ³⁾
<code>static</code>	internal l.	no l.
<code>C++ inline</code>	internal l.	—
<code>const</code> (without <code>extern</code>)	internal l.	no l.

Notes: 1) function def./decl./variable def.
 2) declaration (if var. with initial.: def.)
 3) only declaration;
 if name known, adopt linkage
 Rem.: • `typedef` name always no linkage
 • `C++` class name always ext. linkage

6 Overload of operators `C++`

kind of operator	expression	global function equivalent expression function declaration	member function equivalent expression function declaration
binary	$a \otimes b$	$\text{operator} \otimes (a, b)$ $Te \text{ operator} \otimes (Ta \text{ para}, Tb \text{ parb});$	$a.\text{operator} \otimes (b)$ $Te \text{ operator} \otimes (Tb \text{ parb});$
unary prefix (Op3)	$\otimes a$	$\text{operator} \otimes (a)$ $Te \text{ operator} \otimes (Ta \text{ para});$	$a.\text{operator} \otimes ()$ $Te \text{ operator} \otimes ();$
incr./decr. postfix	$a \otimes$	$\text{operator} \otimes (a, 1)$ $Te \text{ operator} \otimes (Ta \text{ para}, \text{int});$	$a.\text{operator} \otimes (1)$ $Te \text{ operator} \otimes (\text{int});$

a, b expressions, Ta, Tb their types, $para, parb$ parameters, Te result type, \otimes operator symbol

Note: overload of unary postfix operators (Op2) see C++ books

7 Type conversions

7.1 Arithmetic conversions (arithmetische Umwandlungen)

Conversion with many binary operators, if operands of different types. Abbreviation (here):

$Conv(T) \doteq$ IF one operand of type T , THEN conversion other operand \rightarrow type T , result type is T , END
 ELSE continue

$R_{type} \doteq$ range of values of $type$

Conversion rules:

- $Conv(\text{long double})$ (7.4)
- $Conv(\text{double})$ (7.4)
- $Conv(\text{float})$ (7.4)
- integral promotion (7.2) on both operands
- $Conv(\text{unsigned long int})$ (7.3)
- IF long int and unsigned int present:
 IF $R_{\text{unsigned int}} \subseteq R_{\text{long int}}$ THEN
 unsigned int \rightarrow long int (7.3)
 ELSE both \rightarrow unsigned long (7.3)
- $Conv(\text{long int})$ (7.3)
- $Conv(\text{unsigned int})$ (7.3)
- - now only int present -

7.2 Integral promotion (ganzzahlige Typangleichung, Ganzzahl-Erweiterung)

Conversion of the source type (`char/short/integer-bit-field`, in `C` `C++(old)` enumeration type):

IF $R_{\text{source type}} \subseteq R_{\text{int}}$ THEN \rightarrow int ELSE \rightarrow unsigned int (7.3)

`C++(new)` with `bool`: `false` \rightarrow 0, `true` \rightarrow 1

`C++(new)` with enumeration type: \rightarrow type with suitable range of values: int, unsigned int, long int, unsigned long int

7.3 Integral conversion (ganzzahlige Typumwandlung)

IntegerValue \rightarrow unsigned-IntegerValue: conversion into value mod 2^n (n number of bits of the new type)

IntegerValue \rightarrow signed-IntegerValue: value remains if possible, otherwise impl. defined behavior

7.4 Floating point conversions

FloatPointType → IntegerType: cut fraction part; if value not representable, behavior undef.

IntegerType → FloatPointType: if value representable, rounding up/down*), otherwise undefined

FloatPointType → more precise FloatPointType: value remains

FloatPointType → less precise FloatPointType: if value representable, round. up/down*), otherwise undef.

*) only if necessary; in this case no general specification for rounding up or down

7.5 Type of integer constants

Constant has the first type in which the value can be represented:

- constant without suffix: `int`, (only octal/hex. :) `unsigned int`, `long int`, `unsigned long int`
- constant with suffix `U` or `u`: `unsigned int`, `unsigned long int`
- constant with suffix `L` or `l`: `long int`, `unsigned long int`