CX1 Syntaxbeschreibung

CX1 Syntaxbeschreibung

Der schnelle Weg zur automatisierung ihrer Reports

Klaus Riedl

Das Handbuch beschreibt die Syntax der in X1 verwendeten Skriptsprache CX1.

Table of Contents

Index

1 CX1- Sprachreferenz

Die Sprache CX1

Um einerseits möglichst vielen Benutzern den Einstieg in das *X1-Skripting* so leicht wie möglich zu machen und andererseits auch den höchsten Anforderungen der Profis gerecht zu werden haben wir als Basis unserer Skript- Sprache die wohl am weitesten verbreitete Profisprache C/C++ verwendet. CX1 stellt eine syntaktische Untermenge von C/C++ dar. In diesem Handbuch zeigen wir ihnen die Sprachelemente, die CX1 bietet und illustrieren diese mit zahlreichen Beispielen.

1.1 Syntaxdiagramm

Die Syntax von CX1 ist eine Untermenge der C/C++ - Syntax. Die Syntax gliedert sich im wesentlichen in

- Ausdrücke
- Literale (Konstanten)
- Deklarationen
- Anweisungen

Ausdrück:

```
expression:
  assignment-expression
  expression , assignment-expression

assignment-expression
  conditional-expression
  unary-expression assignment-operator assignment-expression

assignment-operator:
  einer aus = *= /= %= += -= >>= <<= &= ^= |=

conditional-expression:
  logical-or-expression
  logical-or-expression ?  expression :  conditional-expression

logical-or-expression:
  logical-and-expression
  logical-or-expression ||  logical-and-expression

logical-and-expression:
  inclusive-or-expression
  logical-and-expression &&  inclusive-or-expression

inclusive-or-expression:
  exclusive-or-expression
  inclusive-or-expression |  exclusive-or-expression

exclusive-or-expression:
  and-expression
  exclusive-or-expression ^  and-expression

and-expression:
  equality-expression:
  and-expression &  equality-expression

equality-expression:
  relational-expression
  equality-expression ==  relational-expression
  equality-expression !=  relational-expression

relational-expression:
  shift-expression
  relational-expression <  shift-expression
  relational-expression >  shift-expression
  relational-expression <=  shift-expression
  relational-expression >=  shift-expression

shift-expression:
  additive-expression
  shift-expression <<  additive-expression
  shift-expression >>  additive-expression

additive-expression:
  multiplikative-expression
  additive-expression +  multiplikative-expression
  additive-expression -  multiplikative-expression

multiplikative-expression
  cast-expression
```

```
  multiplikative-expression *  cast-expression
  multiplikative-expression /  cast-expression
  multiplikative-expression %  cast-expression

cast-expression
  unary-expression
  (  type-name )  cast-expression

unary-expression:
  postfix-expression
  ++  unary-expression
  --  unary-expression
  unary-operator cast-expression
  sizeof  unary-expression
  sizeof  type-name
  allocation-expression
  deallocation-expression

unary-operator:
  einer aus * & + - ! ~

allocation-expression:
  new  new-typename new-initializer opt

new-typename:
  type-specifier-list new-declarator opt

new-declarator:

new-initializer
  ( initializer-list)

deallocation-expression:
  delete  cast-expression

postfix-expression:
  primary-expression
  postfix-expression [  expression ]
  postfix-expression (  expression-list opt)
  postfix-expression .  name
  postfix-expression ->  name
  postfix-expression ++
  postfix-expression --

expression-list:
  assignment-expression
  expression-list ,  assignment-expression

primary-expression:
  literal
  this ::  identifier
  name
```

Konstanten:

```
literal:
  integer-constant
  character-constant
  floating-constant
  string-literal

integer-constant::
  decimal-constant integer-suffix opt
  octal-constant integer-suffix opt
  hexadecimal-constant integer-suffix opt

decimal-constant:
  nonzero-digit
  decimal-constant digit

octal-constant:
  0 octal-constant octal-digit
```

hexadecimal-constant:
  0x hexadecimal-digit
  0X hexadecimal-digit
  hexadecimal-constant hexadecimal-digit

nonzero-digit: einer aus
  einer aus **1 2 3 4 5 6 7 8 9**

octal-digit: einer aus
  einer aus **0 1 2 3 4 5 6 7**

hexadecimal-digit: one of
  einer aus **0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F**

integer-suffix:
  unsigned-suffix long-suffix$_{opt}$
  long-suffix unsigned-suffix$_{opt}$

unsigned-suffix:
  einer aus **u U**

long-suffix:
  einer aus **l L**

character constant:
  'c-char-sequence'
  L 'c-char-sequence'

c-char-sequence:
  c-char
  c-char-sequence c-char

c-char:
  any member of the source character set except the single quote('), backslash(\), or newline character
  escape-sequence

escape-sequence:
  simple-escape-sequence
  octal-escape-sequence
  hexadecimal-escape-sequence

simple-escape-sequence: one of
  einer aus **\' \" \? \\ \a \b \f \n \r \t \v**

octal-escape-sequence:
  **** octal-digit
  **** octal-digit octal-digit
  **** octal-digit octal-digit octal-digit

hexadecimal-escape-sequence:
  \x hexadecimal-digit
  hexadecimal-escape-sequence hexadecimal-digit

floating-point-constant:
  fractional-constant exponent-part$_{opt}$ floating-suffix$_{opt}$
  digit-sequence exponent-part floating-suffix$_{opt}$

fractional-constant:
  digit-sequence$_{opt}$ **.** digit-sequence
  digit-sequence **.** exponent-part:**e** sign$_{opt}$ digit-sequence
  E sign$_{opt}$ digit-sequence

sign: one of
  + –

digit-sequence
  digit
  digit-sequence digit

floating-suffix: one of
  f l F L

string-literal :

```
  " s-char-sequence opt"
  L" s-char-sequence opt"

s-char-sequence :
  s-char
  s-char-sequence s-char

s-char :
  any member of the source character set except the double quotation mark ("), backslash
  (\), or newline character

escape-sequence
```

Deklarationen:

```
declaration:
decl-specifiers opt declarator-list opt;
function-definition
method-definiton X1-specific

decl-specifiers:
  decl-specifiers opt decl-specifier

decl-specifier:
  storage-class-specifier
  type-specifier
  fct-specifier

storage-class-specifier:
  auto
  register
  static
  extern

fct-specifier:
  inline
  virtual

type-specifier:
  simple-type-name
  class-specifier
  const
  volatile

simple-type-name:
  type-name
  char
  short
  int
  long
  signed
  unsigned
  float
  double
  void

type-name:
  identifier

declaration-list:
  declaration
  declaration-list declaration

declarator-list:
  init-declarator
  declarator-list ,  init-declarator

init-declarator:
  ms-modifier-list opt declarator initializer opt

declarator:
  dname
  ptr-operator declarator
```

```
  declarator (  argument-declaration-list )
  declarator [  constant-expression opt ] (  declarator )

ptr-operator:
  * &

dname:
  name

argument-declaration-list:
  arg-declaration-list opt ... opt
  arg-declaration-list , ...

arg-declaration-list:
  argument-declaration
  arg-declaration-list ,  argument-declaration

argument-declaration:
  decl-specifiers declarator

function-definition:
  decl-specifiers opt declarator fct-body

method-definition:
  decl-specifiers opt method  object-link opt ::  declarator fct-body

object-link
  ( name )

fct-body:
  compound-statement

initializer:
  =  expression
  (  expression-list )
```

Anweisungen:

```
statement:
  labeled-statement
  expression-statement
  compound-statement
  selection-statement
  iteration-statement
  jump-statement
  Deklarations-Anweisung

labeled-statement:
  identifier :  statement
  case  constant-expression :  statement
  default:  statement

expression-statement:
  expression opt ;

compound-statement:
  {  statement-list opt }

statement-list:
  statement
  statement-list statement

selection-statement:
  if (  expression )  statement
  if (  expression )  statement else  statement
  switch (  expression )  statement

iteration-statement:
  while (  expression )  statement
  do  statement while (  expression ) ;
  for (  for-init-statement expression opt ;  expression opt )  statement

for-init-statement:
```

```
  expression-statement
  declaration-statement

jump-statement:
  break;
  continue;
  return expression opt;
  goto identifier ;

declaration-statement:
  declaration

class-specifier:
  class-head {  member-list opt }

class-head:
  class-key identifier opt base-spec opt
  class-key class-name base-spec opt

class-key
  class
  struct
  union

member-list:
  member-declaration member-list opt
  access-specifier :  member-list opt

member-declaration:
  decl-specifiers opt member-declarator-list ;

member-declarator-list:
  member-declarator
  member-declarator-list , member-declarator

member-declarator:
  ms-modifier-list opt declarator

base-spec:
  :  base-specifier

base-specifier:
  complete-class-name

access-specifier:
  private
  protected
  public

ms-modifier-list:
  ms-modifier ms-modifier-list opt

ms-modifier:
  __cdecl
  __stdcall
```

1.2 Anweisung

1.2.1 Anweisung-Übersicht

Syntax:

```
statement
  labeled-statement
  expression-statement
  compound-statement
  selection-statement
  iteration-statement
  jump-statement
  declaration-statement
```

1.2.2 Auswahlanweisungen

selection-statement

Syntax:

```
selection-statement:
    if (  expression)  statement
    if (  expression )  statement else  statement
    switch (  expression )  statement
```

Beschreibung:
selection-statements bestimmen den Programmablauf durch die Auswertung bedingter Ausdrücke. Es gibt zwei Arten von selection-statements: Das statement **if...else** und das statement **switch** .

1.2.3 Ausdrucksanweisung

Syntax

```
expression-statement:
    expressionopt;
```

Jede expression, der ein Semikolon folgt, stellt ein expression-statement dar:

CX1 führt ein expression-statement durch Auswerten der expression aus. Alle Nebeneffekte dieser Auswertung werden vor der Auswertung der nächsten Anweisung erzeugt. Die meisten expression-statement sind Zuweisungsanweisungen oder Funktionsaufrufe.

Die Null Anweisung ist ein Sonderfall. Sie besteht nur aus einem Semikolon (;). Die Null-Anweisung hat keine Auswirkung, ist aber in manchen Fällen ganz sinnvoll, zum Beispiel wenn die Syntax von CX1 eine Anweisung erwartet, das Programm aber keine benötigt.

1.2.4 Iterations-Anweisung

iteration-statement

Syntax:

```
iteration-statement:
  while ( expression ) statement
  do statement while ( expression ) ;
  for ( for-init-statement expressionopt; expressionopt ) statement

for-init-statement
  expression-statement
  declaration-statement
```

Beschreibung:
iterations-statement veranlassen statements zu mehrmaligen Ausführung abhängig von einem logischen Abbruchkriterium.

CX1bietet drei iteration-statement: **while** , **do** und **for** .Jede dieser Anweisung läuft bis die Abbruchbedingung Null ergibt.

---

do- statement

Syntax:

```
  do statement  while   ( expression);
```

Beschreibung:
Mit einem statement **do** wird eine **do ... while** - Schleife aufgebaut.

Das *statement* wird solange wiederholt, wie der Wert von *expression* ungleich Null bleibt. Da die Bedingung nach jeder Ausführung von *statement* geprüft wird, wird die Schleife mindestens einmal durchlaufen.

Beispiel:

```
void method::OnRun()
{
  int i;
  i=0;
  do
  {
    i++;
  }
  while ( i<10 )
}
```

---

while- statement

Syntax:

```
  while ( Ausdruck )  statement
```

Beschreibung:
Das Schlüsselwort *while* dient zum Programmieren einer while-Schleife. Das statement wird so lange wiederholt, bis der Wert von Ausdruck Null ist. Die Prüfung findet statt, bevor statement ausgeführt wird. Deshalb wird die Schleife keinmal durchlaufen, falls Anweisung zu Anfang des ersten Durchlaufs den Wert Null ergibt.

Beispiel:

```
void method::OnRun()
{
  int i;
  i=0;
  while (i<10)
  {
    i++;
  }
}
```

---

for- statement

```
Syntax:
  for ( for-init-statement expression-1opt statement )

for-init-statement
```

```
  expression- statement
  declaration- statement
```

Beschreibung:
Die Anwendung *for* dient zum Programmieren einer Schleife. Die Anweisung wird so oft ausgefürt,bis der Wert von *expression-1* Falsch ist. Vor dem ersten Durchlauf der Schleife werden die Variablen für die Schleife durch das for-init-statement initialisiert. Nach jedem Durchlauf der Schleife wird der *expression-2* ausgewertet. Alle expressions sind optional. Wenn *expression-1* weggelassen wird, wird dafür der Wert wahr angenommen.

Beispiel:
```
int method::OnRun()
{
// klassische Schleife mit Initialisierung von  i
  for ( int i=0; i<10; i++)
  {
  }
}
```

1.2.5 Label-Anweisung

labeled-statement

Syntax:
```
labeled-statement
  identifier : statement
  case  constant-expression :  statement
  default:  statement
```

Beschreibung:
Wenn sie das Programm zu einer bestimmten Anweisung springen lassen wollen, müssen sie diese Anweisung durch ein *Label* kennzeichnen.

Beispiel:
```
int method::OnRun()
{
  int i=0;
//Jetzt kommt ein Label
ANFANG:
  i++;
  if (i<10)
    goto ANFANG;
  MessageBox(0, "jetzt ist i>=10");
  return 0;
}
```

1.2.6 Blockanweisung

compound-statement

Syntax:
```
compound-statement:
  { statement-listopt }

statement-list:
  statement
  statement-list  statement
```

Beschreibung:
Ein compound-statement besteht aus einer (möglicherweise leeren) Folge von statements, die mit geschweiften Klammern geklammert sind.

Der Rumpf einer Funktion ist beispielweise ein compound-statement.

1.2.7 Sprung-Anweisung

jump-statement

Syntax:

```
jump-statement:
  break;
  continue;
  return expression opt;
  goto identifier ;
```

Beschreibung:
Das statement **break** bewirkt innerhalb einer Schleife, daß die Ablaufkontrolle zum ersten statement hinter der innersten Schleife, in der **break** steht, springt.Das statement **continue** wird innerhalb von Schleifen benutzt, um die Steuerung an das Ende der innersten Schleife, die zu dem Schleifenkonstrukt (z.B. *for* oder *while*) gehört, springen zu lassen. An diesem Punkt wird dann die Fortsetzungsbedingung der Schleife erneut geprüft.Das statement **return** dient dazu, die aktuelle Funktion zu verlassen und zur aufrufenden Funktion zurückzukehren. Ein Rückgabewert ist derzeit ohne Bedeutung.Das statement **goto** dient dazu, die Ablaufkontrolle zu dem durch **idientifier** angegebenen lokalen Label springen zu lassen.

Beispiel:

```
int method::OnRun()
{
  int i;
START:
  i.Edit("positive Zahl eingeben");
  if (i<0)
    goto START;

  Messagebox(0, "Jetzt ist die Zahl positiv");
  return 0;
}
```

1.3 Ausdrücke

1.3.1 Additiv

Unär

Syntax:
Von den folgenden unären Plusausdrücken
```
  + cast-Ausdruck
  - cast-Ausdruck
```

muß der Operand `cast-Ausdruck` von einem arithmetischen Typ sein.

Wirkung:
```
  + cast-Ausdruck
```
Wert des Operanden nach etwaigen erforderlichen Ganzzahltyp-Erweiterungen.

```
  - cast-Ausdruck
```
Negativer Wert des Operanden nach etwaigen erforderlichen Ganzzahltyp-Erweiterungen.

Binär

Syntax:
```
multiplikative-expression
  additive-expression + multiplikative-expression
  additive-expression - multiplikative-expression
```

Zulässige Operandentypen für *op1 + op2* :
op1 und *op2* sind beide von einem arithmetischen Typ. *op1* ist ein Ganzzahltyp, und *op2* ist ein Zeiger auf ein Objekt. *op2* ist ein Ganzzahltyp, und *op1* ist ein Zeiger auf ein Objekt.
Im ersten Fall werden die Operanden den arithmetischen Standardkonvertierungen unterworfen, und das Ergebnis ist die arithmetische Summe der Operanden.

In den beiden anderen Fällen gelten die Regeln für Zeigerarithmetik.

Zulässige Operandentypen für op1 - op2:
op1 und *op2* sind beide von einem arithmetischen Typ.*op1* und *op2* sind Zeiger auf kompatible Objekttypen.*op1* ist ein Zeiger auf ein Objekt, und *op2* ist ein Ganzzahltyp.
Im ersten Fall werden die Operanden den arithmetischen Standardkonvertierungen unterworfen, und das Ergebnis ist die arithmetische Summe der Operanden.

In den beiden anderen Fällen gelten die Regeln für Zeigerarithmetik.

1.3.2 Array-Indizierung-Operator

Array-Indizierung-Operator

Eckige Klammern [ ] enthalten die Indizes bei ein- oder mehrdimensionalen Arrays. Der Ausdruck

`<exp1>[exp2]`

ist nach Definition gleichwertig mit

`*((exp1) + (exp2))`

wobei *exp1* ist ein Zeiger, und *exp2* ist ein Ganzzahlwert ist.

1.3.3 bedingter Ausdruck

bedingter Ausdruck

Syntax:

```
logical-or-expression
logical-or-expression ? expression : condition-expression
```

Bemerkungen:

Der Bedingungsoperator **?:** ist ein ternärer Operator.

Im Ausdruck E1 ? E2 : E3 wird E1 zuerst ausgewertet. Wenn sein Wert ungleich Null (Wahr) ist, so wird danach E2 ausgewertet und E3 ignoriert. Wenn E1 Null (Falsch) ergibt, so wird E3 ausgewertet und E2 ignoriert.

Das Ergebnis von E1 ? E2 : E3 ist entweder der Wert von E2 oder von E3, abhängig davon, welcher dieser Werte ausgewertet wurde.

E1 muß ein Ausdruck von skalarem Typ sein. E2 und E3 müssen einer der folgenden Regeln entsprechen:
Beide sind arithmetische Typen. In diesem Fall unterliegen E2 und E3 den arithmetischen Standardkonvertierungen und das Ergebnis entspricht dem üblichen Ergebnistyp dieser Konvertierungen.Beide Operanden sind kompatible Struktur- oder Varianten-Typen. Das Ergebnis ist vom Struktur- oder Varianten-Typ von E2 und E3.Beide Operanden sind vom Typ void. Das Ergebnis ist vom Typ void.Beide Operanden sind Zeiger auf qualifizierte oder unqualifizierte kompatible Typen. Der Typ des Ergebnisses ist ein Zeiger auf einen Typ, der alle Typqualifizierer der Typen, auf die beide Operanden zeigen, besitzt.Einer der Operanden ist ein Zeiger, der andere eine Nullzeiger-Konstante. Der Typ des Ergebnisses ist ein Zeiger auf einen Typ, der alle Typqualifizierer der Typen, auf die beide Operanden zeigen, besitzt.Einer der Operanden ist ein Zeiger auf ein Objekt oder auf einen unvollständigen Typ, der andere ist ein Zeiger auf einen qualifizierten oder unqualifizierten void-Typ. Das Ergebnis ist vom Typ des Operanden, der nicht auf void zeigt.

1.3.4 Ausdrucksliste

expression-list

Syntax:

```
postfix-expression ( <argument-expression-list> )
```

Bemerkungen:

Runde Klammern () dienen zur Gliederung von Ausdrücken, Einfassung von Bedingungsausdrücken, Kennzeichnung von Funktionsaufrufen und -parametern. Der Wert eines Funktionsaufrufausdrucks wird, falls er einen hat, von der **return** -Anweisung in der Funktionsdefinition bestimmt. Argument-Ausdruck-Liste ist eine durch Kommata getrennte Liste von Ausdrücken beliebigen Typs, die die Funktionsargumente darstellen.

1.3.5 Gleichheit

Gleichheits-Ausdruck

Syntax:

```
equality-expression:
  relational-expression
  equality-expression ==  relational-expression
  equality-expression !=  relational-expression
```

Es gibt zwei equality-expressions: **==** und **!=** . Sie überprüfen die Gleichheit bzw. die Ungleichheit von arithmetischen Werten oder Zeigern. Die Anwendungsregeln sind denen für die relationalen Operatoren sehr ähnlich.

Hinweis:

Beachten Sie, daß **==** und **!=** eine niedrigere Abarbeitungspriorität haben als die relationalen Operatoren **<**, **>**, **<=** und **>=**. Mit **==** und **!=** können Sie auch bestimmte Zeigertypen auf Gleichheit und Ungleichheit miteinander vergleichen, bei denen die relationalen Operatoren nicht erlaubt sind.

1.3.6 Komma-Operator

Syntax:
```
assignment-expression
  expression , assignment-expression
```

Bemerkungen:
Das Komma trennt die Elemente einer Liste von Funktionsargumenten.

Außerdem wird das Komma auch als Operator in Kommaausdrücken verwendet. Es ist zulässig, diese beiden Verwendungsarten zu mischen, aber es müssen dann Klammern zur Klarstellung gesetzt werden.

Der linke Operand E1 wird als **void** -Ausdruck ausgewertet. Danach wird E2 ausgewertet und liefert den Wert und den Typ des Kommaausdrucks. Dies kann auch rekursiv geschehen; bei dem Ausdruck

E1, E2, ..., En

wird von links nach rechts jeder Einzelausdruck E_i ausgewertet, und der Wert und der Typ von E_n liefern das Ergebnis des gesamten Ausdrucks.

Zur Vermeidung von Mehrdeutigkeiten bei der Verwendung von Kommas in Funktionsargument- oder initialiser-list müssen Klammern eingesetzt werden. Im folgenden Beispiel
func(i, (j = 1, j + 4), k);
wird die Funktion *func* mit drei Argumenten *(i, 5, k),* nicht mit vier, aufgerufen.

1.3.7 Logische Operatoren

Logische Operatoren

Syntax:
```
logical-and-espression && inclusive-or-expression
logical-or-expression || logical-and-expression
! cast-Ausdruck
```

Bemerkungen:
Operanden in logischen Ausdrücken müssen von einem skalaren Typ sein.

&& *Logical and*

liefert das Ergebnis Wahr (1), wenn beide Ausdrücke bei der Auswertung einen Wert ungleich Null ergeben, andernfalls ist das Ergebnis Falsch (0). Wenn der erste Ausdruck Falsch ergibt, wird der zweite nicht ausgewertet.

|| *Logical or*

liefert das Ergebnis Wahr (1), wenn mindestens einer der beiden Ausdrücke bei der Auswertung einen Wert ungleich Null ergibt, andernfalls ist das Ergebnis Falsch (0). Wenn der erste Ausdruck Wahr ergibt, wird der zweite nicht ausgewertet.

! *Logical not*

liefert das Ergebnis Wahr (1), wenn der gesamte Ausdruck einen Wert von Null hat, andernfalls ist das Ergebnis Falsch (0). Der Ausdruck !E ist gleichwertig mit (0 = = E).

1.3.8 Multiplikativ

multiplikative-expression

Syntax:

```
cast-expression
  multiplikative-expression * cast-expression
  multiplikative-expression / cast-expression
  multiplikative-expression % cast-expression
```

Bemerkungen
Es gibt drei multiplikative Operatoren:

- ***** (Multiplikation)
- **/** (Division)
- **%** (Modulus, Rest)

(op1 ***** op2) Produkt der beiden Operanden
(op1 **/** op2) Quotient (op1 dividiert durch op2)
(op1 **%** op2) Rest bei der Division (op1 dividiert durch op2)

Für die Operatoren **/** und **%** muß op2 ungleich Null sein, op2 = 0 ergibt einen Fehler (man kann nicht durch Null teilen).

Falls op1 und op2 Ganzzahlwerte sind und der Quotient nicht ganzzahlig ist, gilt folgendes: Wenn op1 und op2 gleiche Vorzeichen haben, so ergibt op1 / op2 die größte ganze Zahl, die kleiner als der wahre Quotient ist, und op1 % op2 erhält das Vorzeichen von op1.Wenn op1 und op2 entgegengesetzte Vorzeichen haben, so ergibt op1 / op2 die kleinste ganze Zahl, die größer als der exakte Quotient ist, und op1 % op2 erhält das Vorzeichen von op1.

Hinweis:
Es wird immer in Richtung auf Null gerundet. Der Operator ***** ist kontextabhängig und kann auch als der Referenzierungs-/Dereferenzierungsoperator interpretiert werden.

1.3.9 oder (exclusive)

Syntax:

```
exclusive-or-expression:
  and-expression
  exclusive-or-expression ^ and-expression
```

Bemerkungen:
Die bitweisen Operatoren dienen zur Änderung einzelner Bits anstelle einer kompletten Zahl.

Bitweises exclusive-or; vergleicht paarweise je zwei korrespondierende Bits und setzt das entsprechende Bit im Ergebnis auf 1, wenn beide Bits unterschiedlich sind, andernfalls auf 0.

1.3.10 oder (inclusive)

Syntax:

```
inclusive-or-expression
  exclusive-or-expression
  inclusive-or-expression | exclusive-or-expression
```

Bemerkungen:
Die bitweisen Operatoren dienen zur Änderung einzelner Bits anstelle einer kompletten Zahl.

Bitweises inclusive-or; vergleicht paarweise je zwei korrespondierende Bits und setzt das entsprechende Bit im Ergebnis auf 1, wenn eins oder beide Bits 1 sind, andernfalls auf 0.

1.3.11 Postfix-Operatoren

Postfix-Operator

Syntax:

```
postfix-expression(expression-listopt)
postfix-expression[expression]
postfix-expression . name
postfix-expression -> name
postfix-expression ++
postfix-expression --
```

Beschreibung:
( ) Dient zur Gliederung von Ausdrücken, zur Einfassung von Bedingungsausdrücken und zur Kennzeichnung von Funktionsaufrufen und -parametern.

[ ] Dient zur Markierung von Array-Indizes. Dient zum Zugriff auf Struktur- und Variantenelemente.

. und **->** Dient zum Zugriff auf Struktur- und Variantenelemente.

Operatoren . und ->

Syntax:

```
postfix-expression . name
postfix-expression -> name
```

Eine Element-Auswahl bezieht sich auf Strukturen und Klassen. Ein derartiger Ausdruck hat den Wert und Typ des gewählten Elements.

Die Liste beschreibt die beiden Möglichkeiten der Element-Auswahl:

In der ersten Form ist *postfix-expression* ein Wert vom Klassen- oder Strukturtyp und name benennt ein Element einer Klasse oder Struktur. Der Wert der Opertation ist derjenige des bezeichneten Elements und ist ein L-Wert, wenn *postfix-expression* ein L-Wert ist.

In der zweiten Form ist *postfix-expression* ein Zeiger auf eine Klasse oder Struktur und *name* benennt ein Element der Klasse oder Struktur. Wert der Opertation ist derjenige des bezeichneten Elements und ist ein L-Wert.

Beispiel:
Für einen Zeiger einer Klasse sind die expressions

expression -> name

und

(**expression*).*name*

gleichwertig.

Postfix-Inkrement-Operator (++)

Syntax:

```
postfix-expression ++ (postinkrement)
```

Beschreibung:
Der als Operand dienende Ausdruck muß von einem skalaren Typ (arithmetischer Typ oder Zeiger) sein; außerdem muß er ein modifizierbarer L-Wert sein. Für den Postinkrement-Operator gilt, daß der Wert des gesamten Ausdrucks gleich dem Wert der postfix-expression vor der Inkrementierung ist. Nachdem der postfix-expression ausgewertet ist, wird der Operand um 1 erhöht.

Für Zeigertypen gelten die Regeln der Zeigerarithmetik.

Postfix-Dekrement-Operator (--)

Syntax:

```
postfix-expression -- (postdekrement)
```

Beschreibung:
Für den Dekrement-Operator gelten dieselben Regeln wie für den Inkrementoperator, mit dem einzigen Unterschied, daß der Operand um 1 vermindert wird, bevor bzw. nachdem der gesamte Ausdruck ausgewertet wird.

1.3.12 Postfix-Ausdruck

postfix-expression

Syntax:

```
postfix-expression:
  primary-expression
  postfix-expression [  expression ]
  postfix-expression (  expression-listopt)
  postfix-expression .  name
  postfix-expression ->  name
  postfix-expression ++
  postfix-expression --
```

Beschreibung:
Postfix-Ausdrücke dienen zum Aufruf von Postfix-Operatoren.

1.3.13 Primär-Ausdruck

primary-expression

Syntax:

```
  literal
  this
  ::  (expression)
  name

Literal:
  integer-constant
  character-constant
  floating-constant
  string-literal

name:
    identifier
```

Beschreibung:
Das Schlüsselwort **this** kann nur innerhalb des Rumpfes einer (Klassen-) Elementfunktion benutzt werden.

Der Zugriffsoperator **::** erlaubt die Bezugnahme auf Typen, Objekte, Funktionen oder Aufzählungen, deren Bezeichner nicht sichtbar sind.

Die runden Klammern um einen *Ausdruck* verändern den einfachen Ausdruck nicht.

primary-expression *name* bezeichnet Ausdrücke, die auf Operatoren zur Elementauswahl folgen (Punkt . und ->). Deshalb muß *name* entweder ein L-Wert oder eine Funktion sein.

Ein *identifier* ist eine primary-expression, sofern er in geeigneter Weise deklariert wurde.

1.3.14 Relation

Vergleichsoperatoren

Syntax:

```
relational-expression:
  shift-expression
  relational-expression <  shift-expression
  relational-expression >  shift-expression
  relational-expression <=  shift-expression
  relational-expression >=  shift-expression
```

Bemerkungen:
Relational-expressions werden für die Prüfung von Ausdrücken auf Gleichheit oder Ungleichheit benutzt. Wenn die Aussage wahr ist, ergibt der relationale Ausdruck den Wert Wahr (1), andernfalls Falsch (0).

> größer als
< kleiner als
>= größer als oder gleich
<= kleiner als oder gleich

In dem Ausdruck

E1 <*operato* r> E2

müssen die Operanden eine der folgenden Bedingungen erfüllen: E1 und E2 sind beides arithmetische Typen. E1und E2 sind beides Zeiger auf qualifizierte oder unqualifizierte Versionen von kompatiblen Typen. E1 oder E2 ist ein Zeiger auf ein Objekt oder auf einen unvollständigen Typ, und der jeweils andere Ausdruck ist ein Zeiger auf eine qualifizierte oder unqualifizierte Version des Typs void. E1 oder E2 ist ein Zeiger, und der jeweils andere Ausdruck ist eine Nullzeiger-Konstante.

1.3.15 Schiebe-Ausdruck

Schiebe-Ausdruck

Syntax:

```
shift-expression:
    additive-expression
    shift-expression <<  additive-expression
    shift-expression >>  additive-expression
```

Beschreibung:

<< *shift bitweise nach links*

verschiebt alle Bits nach links, wobei das jeweils ganz linke Bit verworfen und das ganz rechte auf 0 gesetzt wird.

>> *shift bitweise nach rechts*

verschiebt alle Bits nach rechts, wobei das jeweils ganz rechte Bit verworfen und wenn kein Vorzeichen vorhanden ist, das ganz linke auf 0 gesetzt wird, ansonsten wird das Vorzeichen übernommen.

Beide Operanden eines bitweisen Operators müssen einen Ganzzahltyp haben.

Hinweis:
Die Operatoren **<<** und **>>** sind kontextabhängig.

>> kann auch der Eingabe-Operator in einem Ein-/Ausgabeausdruck sein
<< kann auch der Ausgabe-Operator in einem Ein-/Ausgabeausdruck sein

1.3.16 Unäre Operatoren

Unary- Operatoren

& Referenzierung
* Dereferenzierung
+ Unary plus
- Unary minus
! logical not
~ Bitweises not

Allgemein
"Unary" = *ein* Operand (im Gegensatz zu *binary* Operatoren mit zwei Operanden)

Priorität ist höher als binäre Operatoren

Syntax

```
& cast-expression
* cast-expression
```

Bemerkungen:
Die Operatoren **&** und ***** arbeiten zusammen, um Zeiger zu referenzieren bzw. zu dereferenzieren, die an Funktionen übergeben werden.

Referenzierungsoperator (&)

Beschreibung:
Der Referenzierungsoperator wird verwendet, um die Adresse eines Zeigers an eine Funktion außerhalb von *main()* zu übergeben.

Der Operand (**cast** -expression) muß eins der folgenden sein:
ein Funktionsdesignatorein L-Wert, der ein Objekt bezeichnet, das kein Bitfeld ist und nicht die Speicherklasse **register** hat. Hat der Operand den Typ *typ* , so ist das Ergebnis ein Zeiger auf *typ* .

Einige Bezeichner, die keine L-Werte sind, wie Funktionsnamen und Array-Namen, werden automatisch konvertiert zu *pointer-to-X* -Typen, wenn sie in bestimmten Zusammenhängen auftreten. Der Operator & kann mit solchen Objekten gebraucht werden, aber dessen Gebrauch ist redundant und daher entmutigend.

Beispiel:

```
T t1 = 1, t2 = 2;
T *ptr = &t1;  // Initialisierter Zeiger
*ptr = t2;     // Gleicher Effekt wie t1 = t2
T *ptr = &t1   // wird behandelt als T *ptr;
ptr = &t1;
```

Also wird *ptr* oder **ptr* zugewiesen. Sobald ptr initialisiert worden ist mir der Adresse *&t1* , kann er sicher dereferenziert werden, damit sich der L-Wert **ptr* ergibt.

---

*Dereferenzierungsoperator (*)*

Beschreibung:
Der Dereferenzierungsoperator kann in einem Variablenausdruck verwendet werden, um einen Zeiger zu erzeugen. Außerdem kann dieser Operator in **external** -Funktionen benutzt werden, um den Wert eines Zeigers, der per Referenz übergeben wurde, zu erhalten.

Ist der Operand vom Typ "Zeiger auf Funktion", so ist das Ergebnis ein Funktionsdesignator.

Ist der Operand ein Zeiger auf ein Objekt, so ist das Ergebnis ein L-Wert, der das Objekt bezeichnet.

Unter jeder der folgenden Bedingungen ist das Ergebnis der Dereferenzierung undefiniert:
Die Cast-expression ist ein Nullzeiger.Die Cast-expression liefert die Adresse einer auto-Variablen und die Ausführung des betreffenden Blocks ist bereits beendet.

Hinweis:
Mit & wird auch der Operator für das bitweise UND bezeichnet.

Das Zeichen * kann auch der multiplikative-operator sein.

Bitweises not (~)

Beschreibung:
invertiert jedes Bit. Dieser Operator wird auch zum Erzeugen von Destruktoren benutzt.

1.3.17 Unärer Ausdruck

unary expression

Syntax

```
unary-expression:
  postfix-expression
  ++  unary-expression
  --  unary-expression
  unary-operator cast-expression
  sizeof  unary-expression
  sizeof  type-name
  allocation-expression  new
  deallocation-expression  delete
```

Inkrement-Operator (++)

Syntax:

```
  postfix-expression ++  (postinkrement)
  ++ unary-expression    (preinkrement)
```

Beschreibung:
Der als Operand dienende Ausdruck muß von einem skalaren Typ (arithmetischer Typ oder Zeiger) sein; außerdem muß er ein modifizierbarer L-Wert sein.

Postinkrement-Operator
Für den Postinkrement-Operator gilt, daß der Wert des gesamten Ausdrucks gleich dem Wert der postfix-expression vor der Inkrementierung ist. Nachdem der postfix-expression ausgewertet ist, wird der Operand um 1 erhöht.

Präinkrement-Operator
Der Operand wird um 1 erhöht, bevor sein Wert ausgewertet wird. Der Wert des gesamten Ausdrucks ist gleich dem erhöhten Wert des Operanden. Der Erhöhungswert wird dem Typ des Operanden angepaßt.

Für Zeigertypen gelten die Regeln der Zeigerarithmetik.

Dekrement-Operator (--)

Syntax:

```
  postfix-expression --  (postdekrement)
  -- unary-expression    (predekrement)
```

Beschreibung:
Für den Dekrement-Operator gelten dieselben Regeln wie für den Inkrementoperator, mit dem einzigen Unterschied, daß der Operand um 1 vermindert wird, bevor bzw. nachdem der gesamte Ausdruck ausgewertet wird.

Der Operator sizeof

Beschreibung:
Der Operator **sizeof** ist auf zwei verschiedene Arten verwendbar:

```
sizeof unary-expression
sizeof type-name
```

Das Ergebnins ist in beiden Fällen einen Konstante vom Typ **int** , die die Größe des vom Operanden (bis auf einige Ausnahmen abhängig vom Typ) verwendeten Speicherplatzes in Bytes angibt. Die Anzahl des für jeden Typ reservierten Speicherplatz hängt von der Maschine ab.

Bei seiner ersten Verwendung wird der Typ vom Ausdruck des Operanden ohne eine Auswertung des Ausdrucks (und damit ohne Seiteneffekte) bestimmt. Wenn der Operand vom Typ **char** (**signed** or **unsigned**) ist, liefert **sizeof** den Wert 1. Wenn der Operand zwar vomTyp Array, aber kein Parameter ist, ist das Ergebnis die gesamte Anzehl von Bytes im Array (mit anderen Worten wird ein Array-Name nicht in einen Zeigertyp konvertiert). Die Anzahl von Elementen in einem Array ist *sizeof array/ sizeof array[0] .*

Wenn der Operand ein Parameter ist, der als Arraytyp oder als Funktionstyp deklariert wurde, liefert **sizeof** die Größe des Zeigers. Bei Strukturen oder Varianten liefert **sizeof** die Gesamtanzahl an Bytes einschließlich allen Inhaltes.

Sie können **sizeof** nicht mit folgenden Ausdrücken verwenden: Funktionstypen, unvollständigen Typen, oder einen *lvalue* , der ein Objekt eines Bitfeldes bezeichnet.

In C++ liefert *sizeof(classtype)* die Größe des Objektes, wobei classtype von einigen Basisklassen abgeleitet ist (Denken Sie daran, daß das die Größe der Basisklasse mit einschließt).

---

allocation-expression

Syntax:
`::`$_{opt}$ **`new`** `typname` **`(`**`initializer`**`)`**$_{opt}$

Beschreibung:
Der Operator **new** ermöglicht eine dynamische Speicherreservierung, erfordert stets die Angabe eines Datentyps für *typname*. Der Operator **new** versucht, ein Objekt des Typs *Typ* zu erzeugen, indem er (falls möglich) **sizeof** (*Typ*) Bytes im freien Speicherbereich (auch Heap genannt) reserviert. Dabei berechnet **new** die Größe von Typ, ohne einen expliziten **sizeof** -Operator zu benötigen. Außerdem ist der zurückgegebene Zeiger vom passenden Typ "Zeiger auf *Typ* ", so daß keine explizite Typumwandlung erforderlich ist. Die Lebensdauer eines Objekts **new** erstreckt sich von seiner Erzeugung bis zu seiner Zerstörung durch den Operator **delete**, der die Speicherreservierung löscht, oder bis zum Programmende.

Bei erfolgreicher Reservierung gibt **new** einen Zeiger auf das neue Objekt zurück.

Auf eine Anforderung nach Reservierung von null Bytes wird ein Zeiger ungleich NULL zurückgegeben. Wiederholte Anforderungen nach null Bytes ergeben jeweils unterschiedliche Zeiger ungleich NULL.

---

deallocation-expression

Syntax:
`::`$_{opt}$ **`delete`** `cast_expression`

Beschreibung:
Der Operator **delete** dient zum Löschen dynamischer Speicherreservierungen; er gibt einen Speicherblock frei, der durch einen vorhergehenden Aufruf von **new** reserviert wurde.

1.3.18 und

Syntax:
```
and-expression:
  equality-expression:
  and-expression & equality-expression
```

Beschreibung:
Bitweises and; vergleicht paarweise je zwei korrespondierende Bits und setzt das entsprechende Bit im Ergebnis auf 1, wenn beide Bits 1 sind, andernfalls auf 0.

1.3.19 cast

cast-Ausdruck

Syntax:

```
cast-expression
  unary-expression
  (  type-name )  cast-expression
```

Beschreibung:
Cast- Ausdrücke dienen zur Typ- Umwandlung. Schreiben sie vor den umzuwandelnden Wert den gewünschten Typ in Klammern. Beachten sie, daß nicht zwischen allen Typen eine Typ-Umwandlung möglich ist.
Beispiel:

```
double d=1.23;
int i=(int)d;  //Typ- cast von double auf int
```

1.3.20 Zuweisung

assignment-expression

Syntax:

```
  unary-expression assignment-operator assignment-expression
```

Bemerkungen:
Es gibt folgende assignment-Operatoren:
= *= /= %= += -= >>= <<= &= ^= |=

Der Operator = ist dabei der einzige einfache assignment-Operator, die anderen sind kombinierte assignment-assignment-Operatorn.

Im Ausdruck E1 = E2 muß E1 ein modifizierbarer L-Wert. sein. Der assignment-expression selbst ist kein L-Wert.

Der Ausdruck E1 *op* = E2 hat den gleichen Effekt wie E1 = E1 *op* E2 außer, daß der L-Wert E1 im ersten Fall nur einmal ausgewertet wird.

Beispiel:
E1 += E2 ist gleichwertig mit E1 = E1 + E2. Der Wert des Zuweisungsausdrucks nach Ausführung der Zuweisung ist E1.

Bei einfachen sowie bei kombinierten Zuweisungen müssen die Operanden E1 und E2 einer der folgenden Bedingungen entsprechen:
E1 ist eine qualifizierte oder unqualifizierte Version eines arithmetischen Typs, und E2 hat einen arithmetischen Typ.E1 ist eine qualifizierte oder unqualifizierte Version eines Struktur- oder Varianten-Typs, der mit dem Typ von E2 kompatibel ist.E1 und E2 sind Zeiger auf qualifizierte oder unqualifizierte Versionen von kompatiblen Typen, und der Typ, auf den der linke Zeiger zeigt, hat alle Qualifizierer des Typs, auf den der rechte Zeiger zeigt.Einer der Operanden E1 und E2 ist ein Zeiger auf ein Objekt oder auf einen unvollständigen Typ, und der andere Operand ist ein Zeiger auf einen qualifizierten oder unqualifizierten void-Typ. Der Typ, auf den der linke Zeiger zeigt, hat alle Qualifizierer des Typs, auf den der rechte Zeiger zeigt.E1 ist ein Zeiger, und E2 ist die Nullzeiger-Konstante (NULL).

Hinweis:
Leerzeichen innerhalb der Operatoren für kombinierte Zuweisung (zum Beispiel +<Leerzeichen>=) führen zu Fehlern.

1.4 Deklaration

Eine **Deklaration** ist eine Liste mit Namen. Diese Namen werden manchmal *Deklaratoren* oder *Bezeichner* genannt. Die Deklaration beginnt mit den optionalen Speicherklassen-Bezeichnern, den Typ-Bezeichnern und anderen Modifizierern. Die Bezeichner werden durch Kommata voneinander getrennt. Die Liste wird durch ein Semikolon abgeschlossen.

Einfache Deklarationen von Variablenbezeichnern haben folgendes Muster:

Datentyp var1 =init1$_{opt}$, var2 =init2$_{opt}$, ...;

wobei *var1, var2* ,... eine beliebige Folge verschiedener Bezeichner darstellt, die zusätzlich Initialisierer enthalten kann. Alle Variablen sind vom Typ *Datentyp* .

Beispielsweise erzeugt

```
int x = 1, y = 2;
```

zwei Integer-Variablen x und y (und initialisiert sie mit den Werten 1 bzw. 2).

Es handelt sich hierbei um definierende Deklarationen, das heißt, es erfolgt eine Speicherzuweisung und die optionalen Initialisierer werden angewendet.

1.4.1 Deklaratorliste

Folgende Objekte können deklariert werden:

- Variablen
- Funktionen
- Klassen und Elemente von Klassen (C++)
- Typen
- Strukturen-, Varianten- und Aufzählungs-Tags
- Elemente von Strukturen
- Elemente von Varianten
- Arrays anderer Typen
- Aufzählungskonstanten
- Anweisungs-Sprungmarken (Labels)
- Präprozessor-Makros

Die rekursive Natur der Deklarationssyntax erlaubt sehr komplexe Deklaratoren. Sie sollten deshalb versuchen, durch Verwendung von typedef die Lesbarkeit zu verbessern.

Die *init-declarator-list* ist eine durch Kommata getrennte Sequenz von Deklaratoren, die jeweils mehrere Typ-Informationen, Initialisierer oder beides haben kann. Die Deklaratoren beinhalten die *identifiers* . Die *declaration-specifiers* beinhaltet eine Reihe von Typen und *storage-class specifiers* , welche das Verhältnis und Speicherdauer,zeigen.

1.4.2 Deklarator

declarator

Syntax:

```
declarator-list:
    init-declarator
    declarator-list ,  Init-declarator

Init-declarator
    declarator initializeropt

declarator:
    dname
    ptr-Operator declarator
    declarator (  argument-declarations-list )
    declarator [  constant- expressionopt] (  declarator )
```

1.4.3 Deklaration

Declaration

Syntax:

```
declaration:
  decl-specifiersopt declarator-listeopt;
  function-definition
  method-definiton

decl-specifiers:
  decl-specifiersopt decl-specifier

decl-specifier:
  storage-class-specifier
  type-specifier
  fct-specifier

storage-class-specifier:
  auto
  register
  static
  extern

fct-specifier:
  inline
  virtual

type-specifier:
  simple-type-name
  const
  volatile
```

Die Schlüsselworte der storage-class-specifier, fct-specifier und type-specifier sind derzeit noch nicht verfügbar.

Beschreibung:
Die Deklaration beginnt mit den optionalen storage-class-Bezeichnern, den Typ-Bezeichnern und anderen Modifizierern. Die Bezeichner werden durch Kommata voneinander getrennt. Die Liste wird durch ein Semikolon abgeschlossen.

Einfache declarations von Variablenbezeichnern haben folgendes Muster:

Datentyp var1 = initopt , var2 , ...;

wobei *var1, var2* ,... eine beliebige Folge verschiedener Bezeichner darstellt, die zusätzlich Initialisierer enthalten kann. Alle Variablen sind vom Typ *Datentyp* .

Beispielsweise erzeugt

int x = 1, y = 2;

zwei Integer-Variablen x und y (und initialisiert sie mit den Werten 1 bzw. 2).

Es handelt sich hierbei um definierende declarations, das heißt, es erfolgt eine Speicherzuweisung und die optionalen Initialisierer werden angewendet.

Die Syntax der declarations-list für formale Parameter ist derjenigen der declarator in normalen Bezeichnerdeklarationen ähnlich.
Beispiele:
int func(void) {// keine Argumente
int func(T1 t1, T2 t2, T3 t3=1) { // drei einfache Parameter, der letzte mit

```
                                    // Vorgabewert
int func(T1* ptr1, T2& tref) {// ein Zeiger und eine Referenz
int func(register int i) {// fordert ein Register für das Argument an
int func(char *str,...) {/* ein String und eine variable Anzahl weiterer
Argumente oder eine feste Anzahl von Argumenten mit
variablen Typen */
```

Parameter mit voreingestellten Werten müssen die letzten Argumente in der Parameterliste sein. Die Argumente können vom Typ Skalar, Struktur, Variante oder Aufzählung sein. Es kann sich auch um Zeiger oder Referenzen auf Strukturen und Varianten oder um Zeiger auf Funktionen, Klassen oder Arrays handeln.

Das Auslassungszeichen (...) zeigt an, daß die Funktion bei verschiedenen Gelegenheiten mit unterschiedlich vielen Argumenten aufgerufen werden kann. Vor dem Auslassungszeichen kann eine Liste mit bekannten Argumentdeklarationen stehen. Diese Form des Prototyps verringert den Umfang der Überprüfung, die der Compiler durchführen kann.

storage-class-specifier:
storage-class-specifier bestimmen den Speicherort (Datensegment, Register, Heap oder Stack) eines Objektes sowie seine Lebensdauer (die sich entweder über die gesamte Laufzeit des Programms oder nur auf die Ausf¨hrung eines bestimmten Programmblocks erstrecken kann). Die storage-class kann durch Syntaxelemente innerhalb einer declaration, durch Plazierung der declaration im Quelltext oder durch beides festgelegt werden.

Mit dem speziellen Operator sizeof(der während der Kompilierung ausgewertet wird) können Sie für jeden benutzerdefinierten und standardmäßigen Typ die Größe in Byte ermitteln.

1.4.4 identifier

Deklaration

identifier
Dies ist die formale Definition für einen identifier:

```
identifier
  not-digit
  identifier not-digit
  identifier digit

not-digit: eines der folgenden Zeichen:
  a b c d e f g h i j k l m n o p q r s t u v w x y z _
  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

digit: eines der folgenden Zeichen:
  0 1 2 3 4 5 6 7 8 9
```

Namen- und Längenbeschränkungen:

Bezeichner sind beliebige Namen von beliebiger Länge für Klassen, Objekte, Funktionen, Variablen, benutzerdefinierte Datentypen, etc. Bezeichner können die Buchstaben a bis z, A bis Z, den Unterstrich (_) und die Ziffern 0 bis 9 enthalten. Es gibt eine Einschränkung:
Das erste Zeichen muß ein Buchstabe oder der Unterstrich sein.

1.4.5 Variablendeklaration

Typ-Bezeichner

Syntax:

```
simple-type-name:
  type-name
  char
  short
  int
  long
  signed
  unsigned
  float
  double
  void

type-name:
  identifier
```

---

char

Syntax:

```
  [signed|unsigned] char <identifier> ;
```

Beschreibung:
Dieser Typ-Bezeichner dient zur Deklaration einer Zeichenvariablen; Variablen vom Typ char belegen 1 Byte an Speicherplatz.

Eine char-Variable kann signed, unsigned oder unspezifiziert sein.

Objekte, die als Typ char deklariert sind, können jedes Zeichen aus dem nicht erweiterten ASCII-Zeichensatz aufnehmen.

---

short

Syntax:

```
  [signed|unsigned] short <identifier> ;
```

Beschreibung:
Der Typ-Modifizierer short wird verwendet, wenn die Größe einer Variablen kleiner als beim Typ int sein soll. Dieser Modifizierer kann auf den Basistyp int angewendet werden.

Fehlt in einer Deklaration die Angabe des Basistyps, so wird dafür *int* angenommen.

---

int

Syntax:

```
  [signed|unsigned] int <identifier> ;
```

Beschreibung:
Der Typ-Bezeichner int dient zum Definieren eines int-Datentyps.
int-Variablen können signed (Voreinstellung) oder unsigned sein.

---

long

Syntax:

```
  long [int] <identifier> ;
  [long] double <identifier> ;
```

Beschreibung:
Bei Verwendung zur Modifizierung des Typs int wird bewirkt, daß zur Speicherung des Typ int vier Bytes an Speicherplatz verfügbar sind. Bei der Modifizierung des Typs double stehen 16 Byte zur Verfügung.

---

signed

Syntax:
```
signed <typ> <variable> ;
```
Beschreibung:
Der Typ-Modifizierer signed wird verwendet, wenn der Wert einer Variablen sowohl positiv als auch negativ sein kann. Dieser Modifizierer kann auf die Basistypen int, char und long short angewendet werden.

Fehlt in einer Deklaration die Angabe des Basistyps, so wird dafür int angenommen.

---

unsigned

Syntax:
```
unsigned <typ> <variable> ;
```
Beschreibung:
Der Typmodifizierer unsigned wird verwendet, wenn der Wert einer Variablen stets positiv ist. Dieser Modifizierer kann auf die Basistypen int, char, long und short angewendet werden.

Fehlt in einer Deklaration die Angabe des Basistyps, so wird dafür int angenommen.

---

float

Syntax:
```
float <Bezeichner> ;
```
Beschreibung:
Der Typspezifizierer **float** legt fest, daß ein Name für einen Gleitkommadatentyp steht. **float** hat eine grösse von 32 Bit und einen Wertebereich von $3.4 * (10^{-38})$ bis $3.4 * (10^{+38})$.

---

double

Syntax:
```
[long] double   <Bezeichner> ;
```
Beschreibung:
Der Typspezifizierer double wird bei der Deklaration von Namen verwendet, die für Gleitkommatypen stehen. Der optionale Modifizierer long bewirkt, daß die Gleitkommawerte eine höhere Stellenzahl (also Genauigkeit) haben.

---

void

Syntax:
```
void <Bezeichner> ;
```
Beschreibung:
Das Schlüsselwort void bezeichnet den Typ des Rückgabewerts von Funktionen, die keinen Wert zurückliefern.

Beispiel:
```
void hallo(char *name)
{
  printf("Hallo, %s.",name);
}
```
Die Angabe void in der Parameterliste einer Funktion bedeutet, daß die Funktion keine Parameter hat.

Beispiel:
```
int init(void)
{
  return 1;
}
```

1.5 Konstanten

1.5.1 string-literal

string literal

Syntax:

```
string-literal :
    " s-char-sequenceopt" L" s-char-sequenceopt"

s-char-sequence :
    s-char
    s-char-sequence s-char

s-char :
    any member of the source character set except the double
    quotation mark ("), backslash (\), or newline character

escape-sequence
```

Beschreibung:
String-Konstanten (oder String-Literale) sind eine spezielle Konstantenkategorie für feste Zeichenfolgen. Derartige Konstanten setzen sich aus einer Reihe beliebig vieler Zeichen in Anführungszeichen zusammen:

```
"This is literally a string!"
```

Der Null-String (leerer String) ist:

```
""
```

Die Zeichen, die innerhalb der Anführungszeichen stehen, können auch Escape-Sequenzen enthalten. Der Quelltext

```
"\t\t\"Name\"\\\tAddresse\n\n"
```

gibt folgendes aus:

```
"Name"\      Addresse
```

Vor "Name" stehen zwei Tabulatoren, vor Adresse ein Tabulator. Danach folgen zwei neue Zeilen. \" sorgt für die Ausgabe der Anführungszeichen im String.

1.5.2 floating-point-constant

floating-point-constant

Syntax

```
floating-point-constant:
    fractional-constant exponent-partopt floating-suffixopt
    digit-sequence exponent-part floating-suffixopt

fractional-constant:
    digit-sequenceopt.  digit-sequence
    digit-sequence .

exponent-part:
    e  signopt digit-sequence
    E  signopt digit-sequence

sign: one of
    + –

digit-sequence
    digit
    digit-sequence digit

floating-suffix: one of
    f l F L
```

Beschreibung:
Eine Gleitkommakonstante setzt sich aus fünf Bestandteilen zusammen:

- Vorkommastellen (digit-sequence)
- Dezimalpunkt
- Nachkommastellen (digit-sequence)
- e oder E und ein vorzeichenbehafteter Integerexponent
- Suffix: f, F oder l, L

Sie können entweder die Vorkomma- oder Nachkommastellen wegfallen lassen (aber nicht beide). Sie können entweder den Dezimalpunkt oder den Buchstaben e (E) und den Exponenten weglassen (aber nicht beide). Diese Regeln ermöglichen sowohl die konventionelle als auch die wissenschaftliche Notation (mit Exponenten).

Negative Gleitkommakonstanten werden als positive Konstanten mit vorangestelltem unären Minusoperator behandelt.

Gleitkommakonstanten ohne Suffix sind vom Datentyp double. Durch Anfügen eines f- oder F-Suffixes können Sie jedoch einer Gleitkommakonstanten den Typ float, durch Anfügen eines l- oder L-Suffixes den Typ long double zuweisen. Long double benötigt 8 Byte Speicher.

1.5.3 character-constant

character-constant

Syntax:

```
character-constant
  'c-character'

c-character
  alle Zeichen außer Abostroph( ' ), Backslash( \ ) und Newline.
  escape-Sequenz

Escape-Sequenz
  \' \" \\ \a \b \f \n \r \t \v
```

Beispiel:

```
  char c='X';//normaler character
  char cr='\r';//Escape-Sequenz für carriage return
```

Escape-Sequenz	ASCII- Wert	Zeichen
\\	92	backslash
\?	63	Fragezeichen
\'	39	Abostroph
\"	34	Anführungszeichen
\0	0	Null
\n	10	newline
\t	9	horizontaler Tab
\v	11	vertikaler Tab
\b	8	backspace
\r	13	carriage return
\f	12	formfeed
\a	7	alert

Syntax:

```
octal-escape-sequence:
  \ octal-digit
  \ octal-digit octal-digit
  \ octal-digit octal-digit octal-digit

hexadecimal-escape-sequence:
  \x hexadecimal-digit
  hexadecimal-escape-sequence hexadecimal-digit
```

Ein Backslash in Kombination mit oktalen oder hexadezimalen Zahlen repräsentiert den ASCII- oder Steuercode, der diesem Wert entspricht, z.B. '\03' für *Ctrl-C* oder '\x3F' für das Fragezeichen. Eine Escape-Sequenz kann einen String mit bis zu drei oktalen oder beliebig vielen hexadezimalen digits umfassen, vorausgesetzt, der Wert liegt im zulässigen Bereich des Datentyps char. Größere Zahlen liefern Compilerfehler. Die Oktalzahl \777 ist zum Beispiel größer als der zulässige Höchstwert \377 und ergibt somit einen Fehler. Das erste nicht-oktale oder nicht-hexadezimale Zeichen in einer oktalen oder hexadezimalen Escape-Sequenz markiert das Ende der Sequenz.

1.5.4 integer-constant

integer-constant

Syntax:

```
integer-constant
  decimal-constant integer-suffixopt
  octal-constant integer-suffixopt
  hexadecimal-constant integer-suffixopt

decimal-constant
  digit1-9
  decimal-constant digit

octal-constant:
  0
  octal-constante octal-digit

hexadecimal-constant:
  0x   hexadecimal-digit
  0X   hexadecimal-digit
  hexadecimal-constant hexadecimal-digit

digit 1-9
  1 2 3 4 5 6 7 8 9

octal-digit
  0 1 2 3 4 5 6 7

hexadecimal-digit
  0 1 2 3 4 5 6 7 8 9
  a b c d e f
  A B C D E F

integer-Suffix
  l L
```

Beispiel:

```
  int i=157;//dezimal-Konstante
  int j=0365;//oktal-Konstante
  int k=0192;//fehlerhafte oktal-Konstante wegen '9'
  int l=0x3fff;//hexadezimal-Konstane
  long l=1234567L;//dezimal-Konstante mit Suffix
```

1.5.5 oct_hex_escape

Escape-Sequenzen für Oktal- und Hexdezimalzahlen

Ein Backslash in Kombination mit oktalen oder hexadezimalen Zahlen repräsentiert den ASCII- oder Steuercode, der diesem Wert entspricht, z.B. '\03' für *Ctrl-C* oder '\x3F' für das Fragezeichen. Eine Escape-Sequenz kann einen String mit bis zu drei oktalen oder beliebig vielen hexadezimalen Ziffern umfassen, vorausgesetzt, der Wert liegt im zulässigen Bereich des Datentyps char. Größere Zahlen liefern Compilerfehler. Die Oktalzahl \777 ist zum Beispiel größer als der zulässige Höchstwert \377 und ergibt somit einen Fehler. Das erste nicht-oktale oder nicht-hexadezimale Zeichen in einer oktalen oder hexadezimalen Escape-Sequenz markiert das Ende der Sequenz.

1.5.6 l_r_value

L- und R-Werte

L-Werte
Ein L-Wert (*lvalue*) ist ein Objektzeiger: ein Ausdruck, der ein Objekt kennzeichnet. Ein Beispiel für einen L-Wert-Ausdruck ist *P, wobei P ein Ausdruck für einen Zeiger ungleich Null ist. Ein modifizierbarer L-Wert ist ein Bezeichner oder ein Ausdruck, der zu einem Objekt gehört, auf das zugegriffen werden kann und das im Speicher geändert werden darf. Ein const-Zeiger auf eine Konstante ist zum Beispiel kein modifizierbarer L-Wert. Ein Zeiger auf eine Konstante kann geändert werden, nicht aber deren dereferenzierter Wert.

Historisch steht L für "links", das heißt, daß ein L-Wert auf der linken Seite einer Zuweisungsanweisung (auf der empfangenden Seite) stehen darf. Jetzt dürfen nur noch modifizierbare L-Werte auf der linken Seite einer Zuweisungsanweisung stehen. Sind a und b beispielsweise nichtkonstante Integer-Bezeichner mit einem entsprechend zugewiesenen Speicherplatz, dann handelt es sich bei beiden um modifizierbare L-Werte. Somit sind Zuweisungen wie etwa a=1 und b=a+b zulässig.

R-Werte
Der Ausdruck a + b ist kein L-Wert. a + b = a ist nicht zulässig, weil der Ausdruck auf der linken Seite kein Objekt bezeichnet. Derartige Ausdrücke werden oft als R-Werte bezeichnet (als Abkürzung für "rechte" Werte).

Index

- A -

- B -

- C -

- D -

- E -

- F -

- G -

- H -

- I -

- J -

- K -

- L -

- M -

- N -

- O -

- P -

- R -

- S -

- T -

- U -

- V -

- W -

- Z -