neu in C++11: range-based for

```
int array[] = { 1, 2, 3, 4, 5 };

for (int x : array) // value
    x *= 2;

for (int& x : array) // reference
    x *= 2;
```

C++11

Ersetzung durch:

```
{
    auto && __range = range-init;
    for ( auto __begin = begin-expr,  __end = end-expr;  __begin != __end;  ++__begin ) {
        for-range-declaration = *__begin;
        statement
    }
}
```

neu in C++11: uniform initialization syntax/semantic

C++(98) hat verschiedene Wege zur Initialisierung, je nach Objekttyp und Kontext. -> fehleranfällig und nicht konsistent


```
string a[] = { "foo", " bar" }; // ok: initialize array variable
vector<string> v = { "foo", " bar" }; // error: initializer list for non-aggregate vector
void f(string a[]); f( { "foo", " bar" } ); // syntax error: block as argument
```

und

```
int a = 2;           // assignment style
int[] aa = { 2, 3 }; // assignment style with list
complex z(1,2);      // functional style initialization
x = Ptr(y);          // functional style for conversion/cast/construction
```

neu in C++11: uniform initialization syntax/semantic

C++11 {}-initializer lists für alle Initialisierungen:

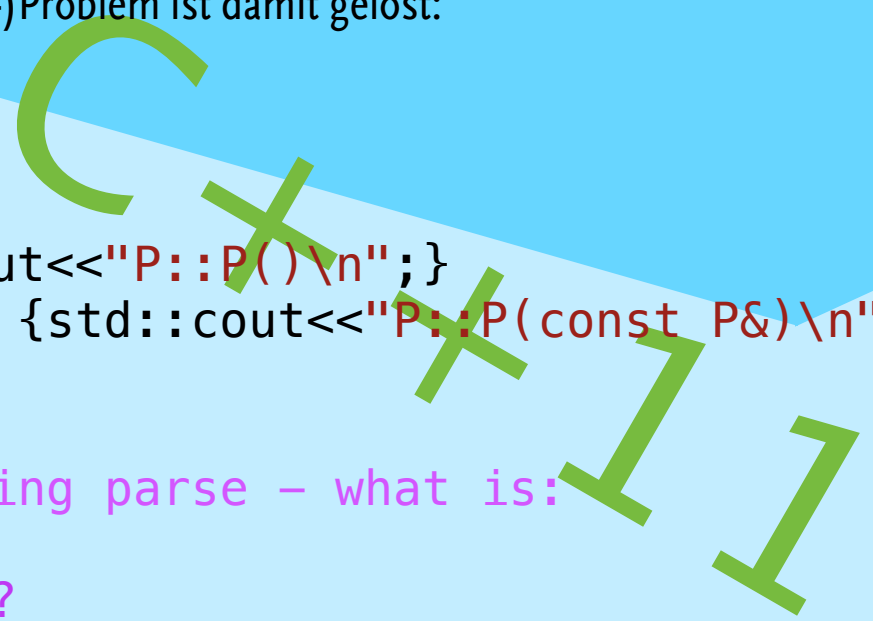
```
X x1 = X{1,2};
X x2 = {1,2}; // the = is optional
X x3{1,2};
X* p = new X{1,2};

struct D : X {
 D(int x, int y) :X{x,y} { /* ... */ };
};

struct S {
 int a[3];
 S(int x, int y, int z) :a{x,y,z} { /* ... */ };
 // solution to an old problem
};
```

neu in C++11: uniform initialization syntax/semantic

Auch ein altes (Parse-)Problem ist damit gelöst:




```
struct P
{
    P(){std::cout<<"P::P()\n";}
    P(const P&) {std::cout<<"P::P(const P&)\n";}
};

// C++ most vexing parse – what is:

P p(P()); // ???
// p: P -> P :-(

P p{P()}; // default constructed P
```

neu in C++11: uniform initialization syntax/semantic

Handliche Listen überall

C++11

```
vector<double> v = { 1, 2, 3.456, 99.99 };
vector<double> v1 { { 1, 2, 3.456, 99.99 } };

list<pair<string,string>> languages = { // parse error in C++98
 {"Nygaard","Simula"}, {"Richards","BCPL"}, {"Ritchie","C"} };

map<vector<string>,vector<int>> years = { // fine in C++11
  { {"Maurice","Vincent","Wilkes"},{1913,1945,1951,1967,2000} },
  { {"Martin","Ritchards"},{1982,2003,2007} },
  { {"David","John","Wheeler"},{1927,1947,1951,2004} }
};
```

neu in C++11: uniform initialization syntax/semantic

Nicht mehr nur für Felder, Argumente vom Typ `std::initializer_list<T>` möglich.

```
void f( initializer_list<int> );
f( {1,2} );
f( {23,345,4567,56789} );
f({}); // the empty list
f{1,2}; // error: function call ( ) missing
years.insert({{"Bjarne","Stroustrup"},{1950, 1975, 1985}});

void print(std::initializer_list<int> ls) {
     for(const auto i: ls) std::cout<<i<<std::endl;
}
... print ({1,2,3,4,5,6,7,8,9});
```

Konstruktoren mit einem einzigen Argument vom Typ **std::initializer_list** heißen initializer-list Konstruktoren. Die Standardcontainer, string, regex etc. haben solche.

neu in C++11: no more narrowing

```
int x = 7.3; // Ouch!
void f(int);
f(7.3); // Ouch!
```

C++11

mit {} Initialisierung nicht:

```
int x1 = {7.3}; // error: narrowing
double d = 7;
int x2{d}; // error: narrowing (double to int)
char x3{7};
     // ok: even though 7 is an int, this is not narrowing
vector<int> vi = { 1, 2.3, 4, 5.6 };
     // error: double to int narrowing
```

1.5. Strukturierte Anweisungen

echtes Lokalitätsprinzip lokaler Blöcke in C++ (nicht in Java):

```
class varscope {
  static void bar(int i){}
  void foo() {
        for (int i=0; i<10; ++i)
                bar(i);
        for (int i=0; i<10; ++i)
                bar(i);
        int i=123;
        bar(i);
        {
                int i=234;
// varscope.java:12: Variable 'i' is already defined in this method.
                bar(i);
        }
  }
}
```

1.5. Strukturierte Anweisungen

echtes Lokalitätsprinzip lokaler Blöcke in C++ (nicht in Java):

```
int i = 666;
static void bar(int i){}
void foo(){
        for (int i=0; i<10; ++i)
                bar(i);
        for (int i=0; i<10; ++i)
                bar(i);
        int i=123;
        bar(i);
        {
                int i=234;  // hides all outer i's
                bar(i);
                bar(::i);   // global i
        } // i==123 !
}
```

Objekte definieren wenn sie gebraucht werden;
sie vernichten (lassen), sobald sie nicht mehr gebraucht werden !

Wo und wie lange leben Objekte ?

	globale Objekte	lokale Objekte	dynamische Objekte
entstehen durch ...	globale Objektvereinbarung: `T o;`	blocklokale Objektvereinbarung: `{ .. T o; ..}`	durch expliziten Aufruf von new: `T*op=new T[N];`
Objekte sind initialisiert	*builtin*-Typen: ja, auf 0 Klassentypen: durch Aufruf eines Konstruktors (*	*builtin*-Typen: nein ! Klassentypen: durch Aufruf eines Konstruktors (*	*builtin*-Typen: nein ! Klassentypen: durch Aufruf eines Konstruktors (*
werden vernichtet ...	automatisch nach (!) Programmende	automatisch beim Verlassen des Blockes Sonderfall: ***temporaries*** (**	durch expliziten Aufruf von delete: `delete[] pi;`
residieren im ...	globalen Datenbereich (bereits vom Compiler geplant und vor Programmstart	***Stack*** (dehnt sich dynamisch und sequentiell aus)	***Heap*** (dehnt sich dynamisch und nicht sequentiell aus)

(* u.U. ohne nutzerspezifische Initialisierung (s. default ctor)
(am nächsten sequence point (typischerweise ;)**

1.5. Strukturierte Anweisungen

Switch-Anweisung (C++): **`switch`** (expression) **`statement`**

statement i.allg. strukturiert mittels case: / default: aber mit mehr Freiheiten als in Java

Beispiel: Duff's Device
(Tom Duff 1983)

```
void send
(register short *to,
register short *from,
register count)
{ do *to = *from++; while(--count>0); }

// to: some device register
```

```
void send (register short *to, register short *from,
           register count) {
  register n = (count+7)/8;
  switch (count%8){
          case 0:  do{      *to = *from++;
          case 7:           *to = *from++;
          case 6:           *to = *from++;
          case 5:           *to = *from++;
          case 4:           *to = *from++;
          case 3:           *to = *from++;
          case 2:           *to = *from++;
          case 1:           *to = *from++;
                  } while(--n > 0);
  }
}
```

1.5. Strukturierte Anweisungen

Switch-Anweisung (C++):

Initialisierungen dürfen nicht 'übersprungen' werden:

```
switch (i) {
        int v1 = 2;  // ERROR: jump past initialized variable
case 1:
        int v2 = 3;
        // ....
case 2:
        if (v2 == 7) // ERROR: jump past initialized variable
        // ....
}
```

1.5. Un : -) Strukturierte Anweisung

goto - the don't use statement

```
loop:
// ....
goto loop;
```

```
goto skip:
// ....
skip: //....
```

Initialisierungen dürfen auch nicht 'übersprungen' werden:

1.5. Strukturierte Anweisungen

Exception Handling : syntaktisch wie in Java (kein `finally`)

```
try {
        // things that may throw or not
}
catch ( Exception1 e ) {
        // handle e
}
catch ( Exception2 e ) {
        // handle e
} ......
```

bei Auftreten einer Ausnahme wird der `try`-Block verlassen und zu einem passenden (ggf. übergeordneten) `catch`-Block verzweigt, zuvor werden alle Destruktoren lokaler Objekte gerufen, die erfolgreich konstruiert wurden !

1.5. Strukturierte Anweisungen: Exception Handling

```
#include <iostream>

using std::cout; using std::endl;

class X { public:
                X(int i=0){cout<<"X("<<i<<")\n";}
                ~X(){cout<<"~X()\n";}
};

void foo(int i) {
        try {   X local;
                if (i==1) throw "ooops";
                else if (i==2) throw 42;
        }
        catch (const char* why) {
                cout<<why<<endl;
        }
}
```

1.5. Strukturierte Anweisungen: Exception Handling

```
//... cont.
int main() {
        try {
                X x1(1);
                foo(1);
                X x2(2);
                foo(2);
        }
        catch (int r) {
                cout<<"exception: code "<<r<<endl;
        }
        catch (...) {
                cout<<"something thrown: don't know what\n";
        }
}
```

```
X(1)
X(0)
~X()
ooops
X(2)
X(0)
~X()
~X()
~X()
exception: code 42
```