

1. Elementares C++

P. S. offene Fragen

1. mehrfache Deklarationen in Klassen?

```
class X {  
    void foo();  
    void foo() { .... }  
};
```

Nein!

Memberfunktionen können in der Klasse (genau einmal) deklariert oder definiert werden. Nur wenn nur deklariert wurde, darf außerhalb der Klasse (nur) definiert werden.

2. Warum keine Kontrollflussanalyse in C++?

- exit, Exceptions <http://www.gotw.ca/gotw/020.htm>, asm-Einschlüsse
- dennoch: § 6.6.3: Flowing off the end of a function is equivalent to a return with no value; this results in undefined behavior in a value-returning function. -- in C legal wenn Wert nicht benutzt.
<http://stackoverflow.com/questions/1610030/why-can-you-return-from-a-non-void-function-without-returning-a-value-without-pr>



1. Elementares C++

1.4. Funktionen

- können **inline** sein: kein Aufruf, sondern (Seiteneffekt-freie und typgerechte) Substitution auf Quelltextebene:

```
inline int square(int i){return i*i;}  
int main() { std::cout << square(4); }  
            inline substitution  
  
int main() { std::cout << 4*4; } // u.U. sogar 16
```

- Ziel: Effizienz, auch wenn call overhead > 'Nutzeffekt' der Funktion
- Memberfunktionen, die im Klassenkörper definiert werden, sind implizit **inline**! (gute Kandidaten, weil meist kurz)



D. Knuth: "Premature optimization is the root of all evil ! "
siehe auch

www.ddj.com (search for: inline redux) und www.gotw.ca/gotw/033.htm

1.4. Funktionen

- können default arguments haben: ein Endstück der Argumentliste einer Deklaration mit Wertevorgaben

```
int atoi (const char* string, int base = 10);  
// ascii to int on radix base  
atoi ("110"); // --> atoi("110", 10) --> 110  
atoi ("110", 2); // --> atoi("110", 2) --> 6
```

Vorsicht Falle 1: void foo(char*=0);

void foo(char* =0);

1. Elementares C++

1.4. Funktionen

Vorsicht Falle 2:

```
int f(int);  
int f(int, int=0);  
f (1); // mehrdeutig: f(1) oder f(1,0)
```

- variable Argumentlisten a la `printf` in C++: ... ellipsis

```
extern "C" int printf (const char* fmt, ...);
```

`extern "C"` ist eine sog. linkage Direktive: hier kein name mangling

1. Elementares C++

1.4. Funktionen

- können überladen werden: gleicher Name, unterscheidbare Signatur (Rückgabetyp spielt KEINE Rolle!)

name mangling

```
class X{ public:  
    X();  
    X(int);  
    int foo();  
    int foo() const;  
    int foo(const X&);  
};  
  
int foo(int);  
double foo(double);  
void foo(char*, int);  
int printf(const char*, ...);
```

__1X
__1Xi
foo__1X
foo__C1X
foo__1XRC1X

foo__Fi
foo__Fd
foo__Fpc
printf__FPCce

```
$ g++ -c foo.cc  
$ nm foo.o  
00000000 w __1X  
00000000 w __1Xi  
....  
$ nm foo.o | c++filt  
00000000 w X::X(void)  
00000000 w X::X(int)  
....
```

1. Elementares C++

1.5. Strukturierte Anweisungen

(fast) wie in Java:

`while, do, for, if, switch, break, continue, return`

Deklaration in Blöcken sind Anweisungen: Deklaration von Objekten am Ort des Geschehens (wie in Java)

```
void foo()
{
    int i=0;
    bar(i); ....
    int j=3;
    bar(j); ....
}
```

Vorsicht Falle:
`if (x=1)`

1. Elementares C++

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;
```

Ersetzung durch:

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

1. Elementares C++

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
```

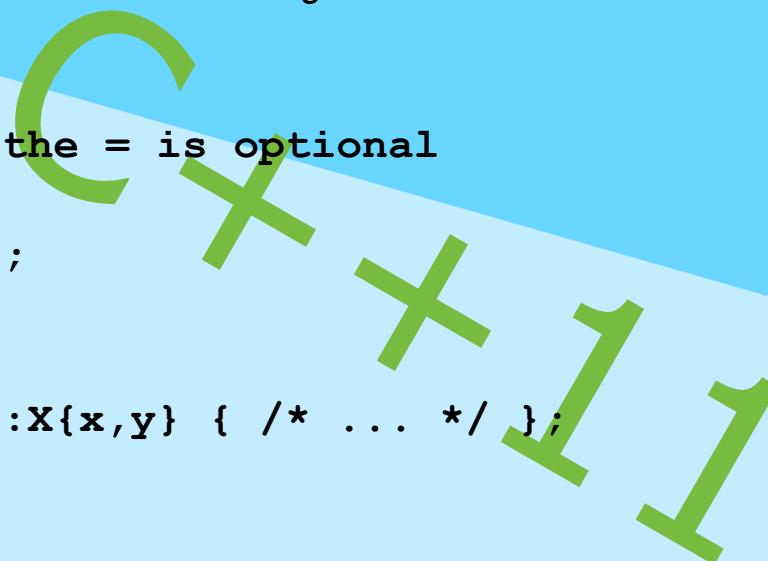
und

```
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
```

1. Elementares C++

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

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



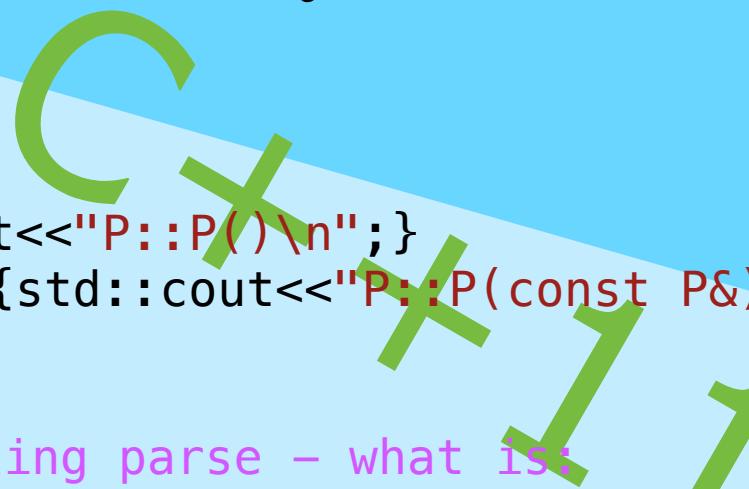
A decorative graphic in the background consists of several green arrows and crosses pointing towards the text. One large green arrow points from the top left towards the first code example. Two green crosses are positioned near the second and third code examples. Two green arrows point downwards from the fourth and fifth code examples. A single green cross is located above the sixth code example.

```
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  
};
```

1. Elementares C++

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:

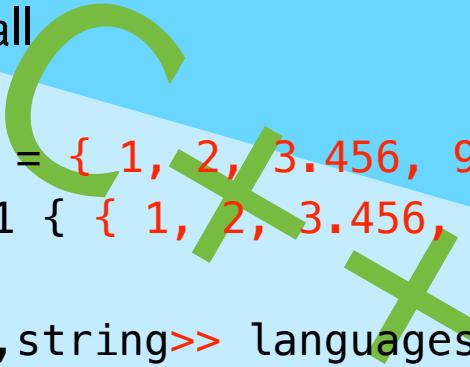
```
P p(P()); // ???
// p: P -> P :-(
```

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

1. Elementares C++

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

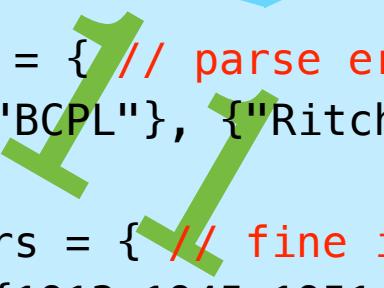
Handliche Listen überall



```
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"} };
```

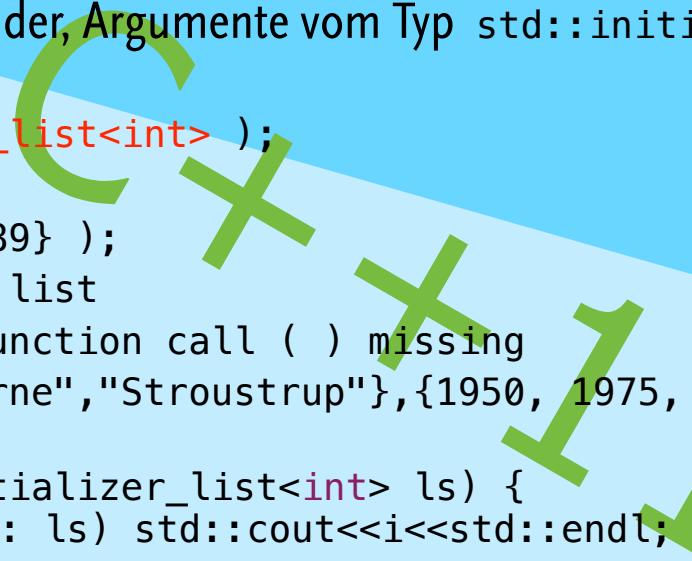


```
map<vector<string>,vector<int>> years = { // fine in C++11  
    { {"Maurice","Vincent","Wilkes"},{1913,1945,1951,1967,2000} },  
    { {"Martin","Ritchards"},{1982,2003,2007} },  
    { {"David","John","Wheeler"},{1927,1947,1951,2004} }  
};
```

1. Elementares C++

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.

1. Elementares C++

neu in C++11: no more narrowing

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

C

+

+

1

1

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. Elementares C++

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. Elementares C++

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 !