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);`

1.4. 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

1.4. Funktionen

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

	name mangling
`class X{ public:`	
`  X();`	`__1X`
`  X(int);`	`__1Xi`
`  int foo();`	`foo__1X`
`  int foo() const;`	`foo__C1X`
`  int foo(const X&);`	`foo__1XRC1X`
`};`	
`int foo(int);`	`foo__Fi`
`double foo(double);`	`foo__Fd`
`void foo(char*, int);`	`foo__Fpci`
`int printf(const char*, ...);`	`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)
....
```

1.5. 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) ....
```

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

C++11

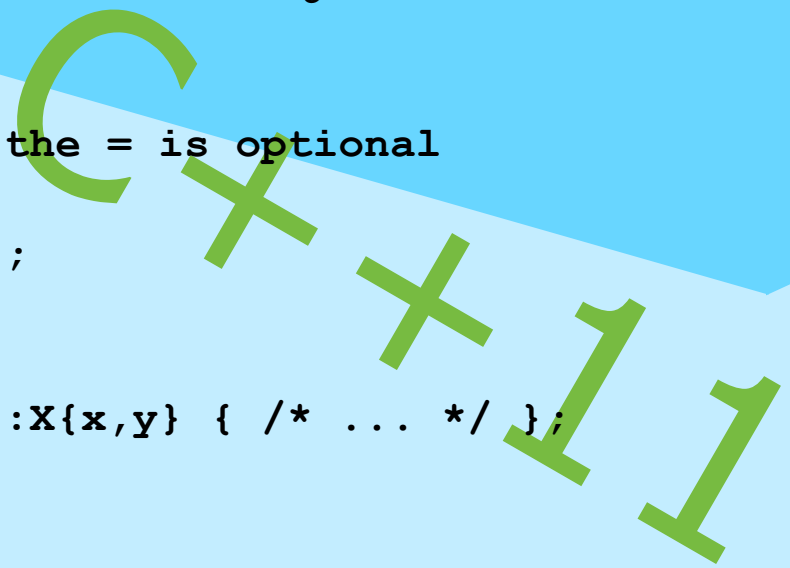
```
string a[] = { "foo", " bar" }; // ok: initialize array variable
vector<string> v = { "foo", " bar" }; // error: initializer list for non-aggregate vector
void f(string a[]); f( { "foo", " bar" } ); // syntax error: block as argument
```

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:

C++11

```
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 !