1.4. Funktionen

- mehrfache Deklarationen sind erlaubt
- für jede Funktion muss es (GENAU) eine Definition geben, ansonsten linker error [the one definition rule ODR]
- Deklarationen in `*.h` - Files, Definitionen in `*.cpp` - Files:

```
// foo.h:
int foo(int,int);
```

```
// foo.cpp
#include "foo.h"
int foo(int a, int b)
{return a+b;}
```

```
// prog.cpp:
#include "foo.h"
int main() { foo(123,456); }
```

1.4. Funktionen

Es gibt (anders als in Java) keine vollständige Analyse des Kontrollflusses:

Java

```
class flow {
  int foo(int i){
          if (i<0) return 42;
          if (i>=0) return 24;
  } // ERROR: Missing return statement
}
```

C++

```
int foo(int i){
          if (i<0) return 42;
          if (i>=0) return 24;
} // OK !
```

ABER: Verlassen einer (non-void) Funktion ohne Rückgabewert: undefined behaviour

1.4. Funktionen

- können `static` sein:
  1. Memberfunktionen: Klassenmethoden wie in Java (kein `this`)
  2. globale Funktionen: file scope

- können `static` (lokale) Daten enthalten:

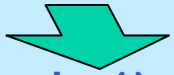
```
int foo() { static int i=2; return i*=i; }
int main(){
for (int i=0; i<3; ++i) std::cout<<foo(); // 416256
}
```

ACHTUNG: `std::cout<<foo()<<foo()<<foo();` ???

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)

Tony Hoare: "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);`

1.4. 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) ....
```

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);
    }
}
```