1.2. Datentypen (Strukturtypen)

```
p.name = "Willibald Wusel";
```

Kombination mit Zeigern (dynamische Strukturobjekte)

```
Person* aNewPerson = new Person;
aNewPerson->age = 32;
// short hand for:
(* aNewPerson).age = 32;
```

Kombination mit Referenzen

```
void raise_salary (Person &p, int percentage) {
      p.salary *= 1 + percentage/100.0; // ? why .0 ?
}
raise_salary (p, 3);
```

1.2. Datentypen (Strukturtypen)

Strukturen sind in C++ de facto Klassen ohne Memberfunktionen und öffentlichem Zugriff auf alle Memberdaten!

```
struct Person {
        std::string name;
        int age;
        double salary;
        long phone_no;
};
```

```
class Person {
public:
        std::string name;
        int age;
        double salary;
        long phone_no;
};
```

1.2. Datentypen (Strukturtypen - Unions)

Es gibt noch die C-Variante, bei der alle Bestandteile eines solchen zusammengesetzten Typs an der gleichen Adresse (am Objektanfang) beginnen -> sog. Unions (spielen in C++ eine untergeordnete Rolle !)

```
union HACK {
        double d; // double precision ieee
        struct {
                unsigned :1,
                e:11;
        } s;
};
int NaN(double x) {
        HACK h; h.d = x; return h.s.e == 0x7ff;
}
```

1.3. Ausdrücke

ähnlich zu Java:

- Literale und Variablen `1.234` `"Huhh..."` `i`
- Anwendung von Operatoren auf Operanden
  `x+1` `std::cout<<4` `x=y` `foo(3,bar(7),&a)` `p->name[0]`

ABER:

- Reihenfolge der Berechnung **undefiniert** (bis auf &&, || und ,) !
- jeder Ausdruck liefert einen Wert (ggf. den leeren Wert bei Funktionen mit Rückgabetyp `void`), ein nicht-leerer Wert kann, muss aber nicht weiterverwendet werden (wie in Java)
- ein Ausdruck wird durch nachfolgendes Semikolon zu einer Anweisung !

```
f(3); // Ergebnis wird ignoriert
int k=f(4); // Ergebnis wird weiterverwendet

int main() { 42; } // KORREKTES C++ ???
```

1.4. Funktionen

- Memberfunktionen von Klassen oder außerhalb von Klassen (global, bzw. namespace-lokal)
- Unterscheidung in Deklaration (Angabe der Signatur) und Definition !

```
void foo(int); // optional: Parameternamen
void foo(int x) { .... }
class X { public: int foo(); // int foo(void) !
                  X& bar() {return *this; }
                  X(int); };
X::X(int i){ .... }
```

- jede Definition ist auch eine Deklaration
- Jede Funktion muss deklariert sein, bevor sie verwendet wird; Deklaration einer Memberfunktionen wirkt ab Klassenbeginn

1.4. Funktionen

- mehrfache Deklarationen sind erlaubt
- für jede Funktion muss es (GENAU) eine Definition geben, ansonsten linker error [the one definition rule ODR]
- Ausnahme: `inline` Funktionen: (identische) Definition in jeder verwendenden Übersetzungseinheit
- 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();` ???

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 Kontollflussanalyse in C++?

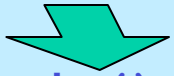
- `exit`, Exceptions http://www.gotw.ca/gotw/020.htm, `asm`-Einschlüsse
- dennoch: § 6.6.3: Flowing off the end of a function is equivalent to a return with no value; this results in undefined behavior in a value-returning function. -- in C legal wenn Wert nicht benutzt.

  http://stackoverflow.com/questions/1610030/why-can-you-return-from-a-non-void-function-without-returning-a-value-without-pr

1.4. 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