1.2. Datentypen

Referenztypen:

Eine Neuerung gegenüber C, Aliasnamen für Objekte mit Referenzsemantik ähnlich zur primären Objektsemantik von Java, aber

– für alle Typen (incl. build-in Typen)

– es gibt KEINE 'Nullreferenz'

in Anlehnung an die Syntax von Zeigervereinbarungen

```
int i=42;
// int& ri;
// ERROR: Referenzen MÜSSEN initialisiert werden
int& ri = i; // i alias ri
```

1.2. Datentypen

Konstantentypen:

Ein Typ `T` wird durch den Präfix `const` zu einem Konstantentyp, Objekte solcher Typen sind unveränderlich (per statischer Kontrolle durch den Compiler)

für Argumente von Funktionen bedeutet dies, dass die Funktion

1. die (nachprüfbare) Zusicherung gibt, dieses Argument NICHT zu verändern
2. beim Aufruf für das Argument auch konstante Objekte benutzt werden dürfen (was für non-const nicht erlaubt ist, weil ja die Funktion keine Zusicherung gegeben hat und daher ...)

```
const double pi=3.1415926; double someMathFkt(double);
const double x = someMathFkt(pi);  // call by value !
```

1.2. Datentypen (Konstantentypen)

Konstante Objekte müssen initialisiert werden (weil eine spätere Zuweisung nicht erlaubt ist)

konstante Objekte können auch über Zeiger nicht verändert werden, weil die Adresse einer `const T` Variablen vom Typ `const T*` ist

```
double* dp = &pi; // ERROR
*dp = 33.3;

const double* cdp = &pi;
*cdp = 33.3; // ERROR
```

1.2. Datentypen (Konstantentypen)

bei Zeigern ist wohl zu unterscheiden zwischen der constness des Zeigers selbst

```
int * const constant_pointer = &someint;
```

und der constness des referenzierten Objektes (Feldes)

```
const int * pointer_to_constant;
```

```
const int * const constant_pointer_to_constant = …;
```

1.2. Datentypen (Konstantentypen)

Referenzen (selbst) sind implizit const, es gibt jedoch Referenzen auf Konstantentypen

Wichtigste Anwendung: call by reference in-parameter

```
T t;
void foo(T& pt)
{
        pt.change();
}
foo(t); // call by reference: t itself changes
const T ct;
foo(ct); // ERROR
```

1.2. Datentypen (Konstantentypen)

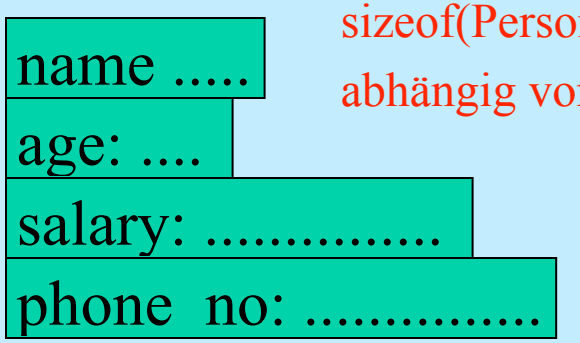
```
void foo(const T& pct)
{
        // pct.change(); ERROR
        pct.read_only();
}

const T ct;
foo(ct); // OK

class X{ public: void foo() const; void bar(); };
X x; const X cx; x.foo(); x.bar();
cx.foo(); /* OK */  cx.bar() // ERROR !
```

1.2. Datentypen

Strukturtypen (a la C):

heterogene Wertekombinationen unter einem Typnamen

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



sizeof(Person)
abhängig vom *alignment* !

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