

1. Elementares C++

1.2. Datentypen

Zeiger: Indirektion per Speicheradresse!

```
int* pi; // ein u.U. NICHT initialisierter Zeiger
```

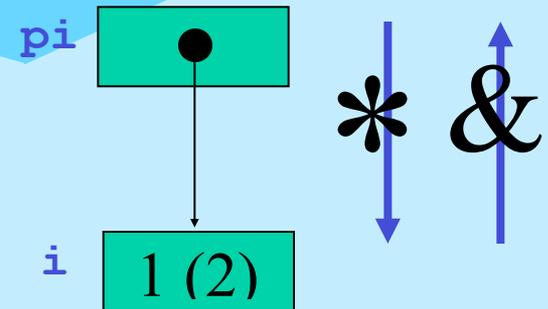
```
int i=1;  
pi = & i; // Adressoperator !
```

Zeiger sind vollständig typisiert:

```
double x;  
// ERROR: pi = & x;
```

Umkehroperation zu & ist die sog. Dereferenzierung:

```
*pi = 2;
```



1. Elementares C++

1.2. Datentypen (Zeiger)

```
int* pi = new int; // ein neues anonymes int auf dem Heap
                // pi ist (bislang) der einzig Zugang
                // no more C: int* pi = malloc(sizeof(int));
```

```
int* ap = pi; // 2 Verweise, 1 Objekt !
```

```
pi = 0; // ausgezeichneter Zeigerwert <==> KEIN Objekt
```

```
ap = 0; // letzte Referenz weg: KEINE garbage collection
        // sondern ein memory leak
```

daher zuvor:

```
delete ap; // kein leak !
           // no more C: free(pi);
```

1. Elementares C++

1.2. Datentypen (Zeiger)

ACHTUNG: nach `delete zeiger;` ist u.U. in `zeiger` immer noch die gleiche Adresse enthalten, jeder Zugriff darüber ist jedoch undefiniert !

Empfehlung: `delete pi; pi=0;`

auch Felder können dynamisch erzeugt werden:

```
int* pf = new int [100]; // pf zeigt auf erstes von 100 int's
```

Zeiger auf Felder sind mit `delete[]` zu deallokieren:

```
delete[] pf; pf=0;
```

1. Elementares C++

1.2. Datentypen (Zeiger)

Zeiger auf Klassentypen (~ Java-Objektsemantik)

```
class X {  
public:  
    X() {std::cout<<"hi\n";}   
    ~X() {std::cout<<"bye\n";}   
};
```

auch: `new X()` ; möglich aber nicht zu empfehlen !

```
void foo() { X* px = new X; } // hi & leak  
void bar() { X* px = new X; delete px; } // hi & bye
```

In jeder (nicht static) Memberfunktion ist `this` ein Zeiger auf das Objekt, an dem der Aufruf der Funktion erfolgte (anders als in Java !)

1. Elementares C++

1.2. Datentypen (Zeiger)

Java - `new` vs. C++ - `new`

```
class X {}

class Main {
    public static void main(String s[]) {
        // X x = new X; nicht ohne leere Parameterliste
        X x = new X();
        // int i = new int; new nur für Klassen erlaubt
        // int i = new int(); auch so nicht
        int i[] = new int[10]; // Felder sind Objekte
    }
}
```

1. Elementares C++

1.2. Datentypen (Zeiger)

Java - **new** vs. C++ - **new**

```
class X {};  
  
int main() {  
    X *x1 = new X;           // besser so  
    X *x2 = new X();        // als so  
    int *i1 = new int;      // di  
    int *i2 = new int();    // to  
    // int i[] = new int[10]; so nicht:  
    // Feldvariablen sind Konstanten & die Größe  
    // von i ist unbestimmt  
    int *i = new int[10]; // ein Zeiger kann  
    // eins oder viele referenzieren !  
}
```

1. Elementares C++

nullptr

```
void foo(int) {std::cout<<"foo(int)\n";}
void foo(int*) {std::cout<<"foo(int*)\n";}
...
foo(0);
foo(NULL);
foo(nullptr);

int* pi = nullptr;

if (pi == nullptr) pi = new int; // same as:
if (!pi) pi = new int;
```

1. Elementares C++

Warum besser keine Klammern bei parameterlosen Konstruktorrufen ?

```
T* pt = new T(); // ok
```

```
T t = T(); // ok, aber redundant
```

```
T t (); // auch ok, aber kein Objekt vom Typ T !
```

?

```
void foo(); // Funktionsdeklaration !!!
```

```
T t (); // dito
```

```
T t;
```

1. Elementares C++

1.2. Datentypen (Zeiger)

Felder sind konstante Zeiger:

```
T someTs [30];  
T* pt = someTs;  
T* qt = &someTs[0];
```

```
using std::cout;...  
cout<<1["]<<2["]<<endl;  
Korrektes C++ ?  
wenn ja, was wird ausgegeben ?
```

`pt[i]` ist nur eine abkürzende Notation von `*(pt+i)`

Die Zeigerarithmetik erfolgt modulo `sizeof(T)`

`pt` ist ein Zeiger auf's erste `T` im Feld

`pt+1` ist ein Zeiger auf's zweite `T` im Feld ...