# Themen

- Zeiger
- dynamische Speicherverwaltung
- Referenzen

# Zeiger (Pointer)

Zeiger sind Variablen, die auf eine Adresse im Hauptspeicher verweisen.

# Speicheradressen

# Zeiger Deklaration 1/2

```
int* intZeiger;

double *doubleZeiger;

Bruch* bruchZeiger;

Bruch** zeigerAufBruchZeiger;
```

# Zeiger Deklaration 2/2

```
int* intZeiger1,intZeiger2;

int *intZeiger1,*intZeiger2;
```

# Adress-Operator (&)

```
int a = 34;

std::cout << "Adresse von a: ";
std::cout << &a << "\n";
```

# Zeiger Initialisierung

```
Datum* datumsZeiger = 2000000;

Datum datum(15,Januar,2007);

Datum* datumsZeiger2 = &datum;
```

# Zeiger Dereferenzierung

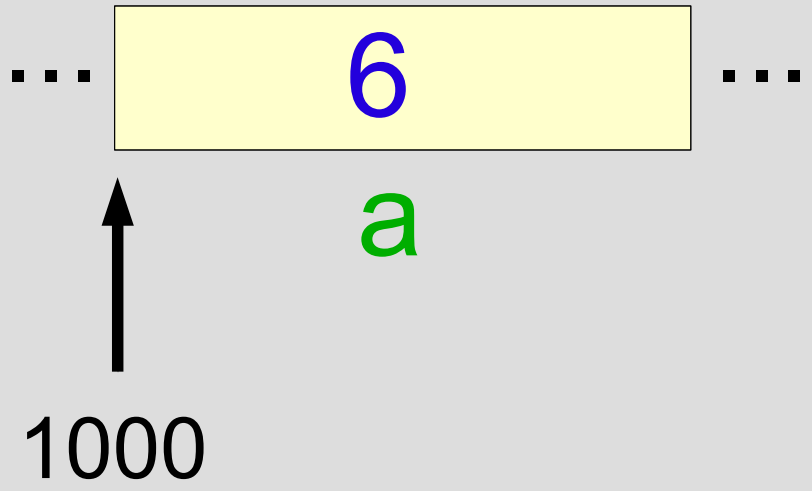
```
Datum datum(15,Januar,2007);

Datum* datumsZeiger = &datum;

(*datumsZeiger).getJahr();

datumsZeiger->getJahr();
```

# Zeiger Beispiel

```
int a=5;
int *pA=&a;
std::cout << "a: " << a << "\n";
std::cout << "Adresse: " << pA << "\n";
*pA=7;
std::cout << "a: " << a << "\n";
std::cout << "Adresse: " << pA << "\n";
pA=10;
std::cout << "a: " << a << "\n";
std::cout << "Adresse: " << pA << "\n";
*pA=25; // !!!
```

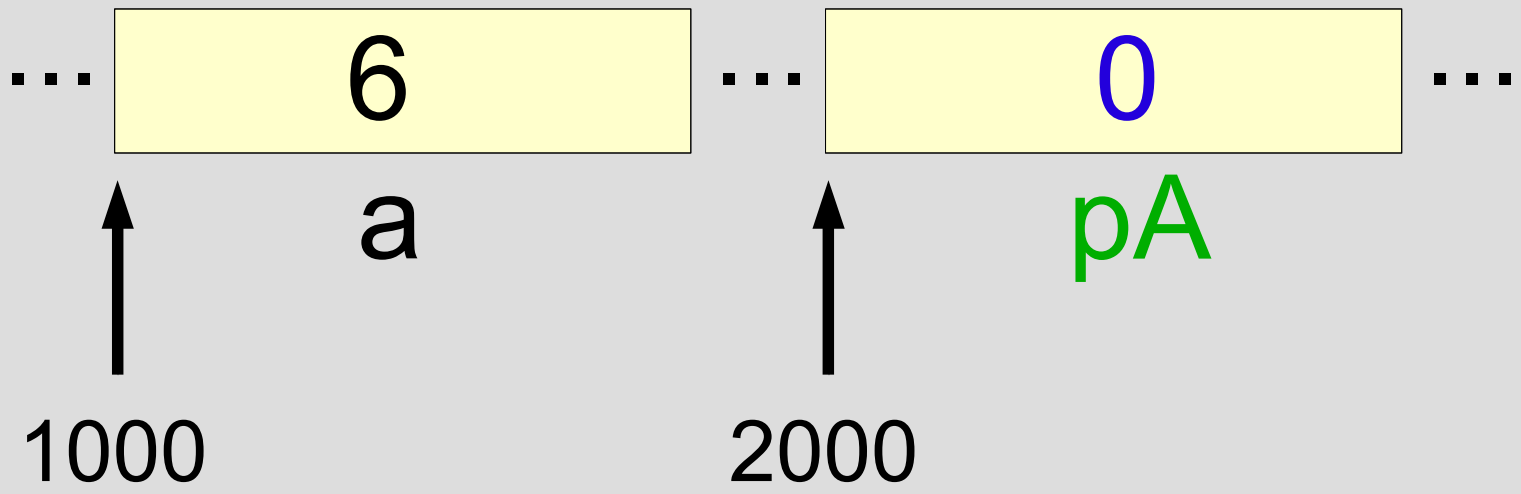
# Zeiger Beispiel 1/6

```
int a=6;
```



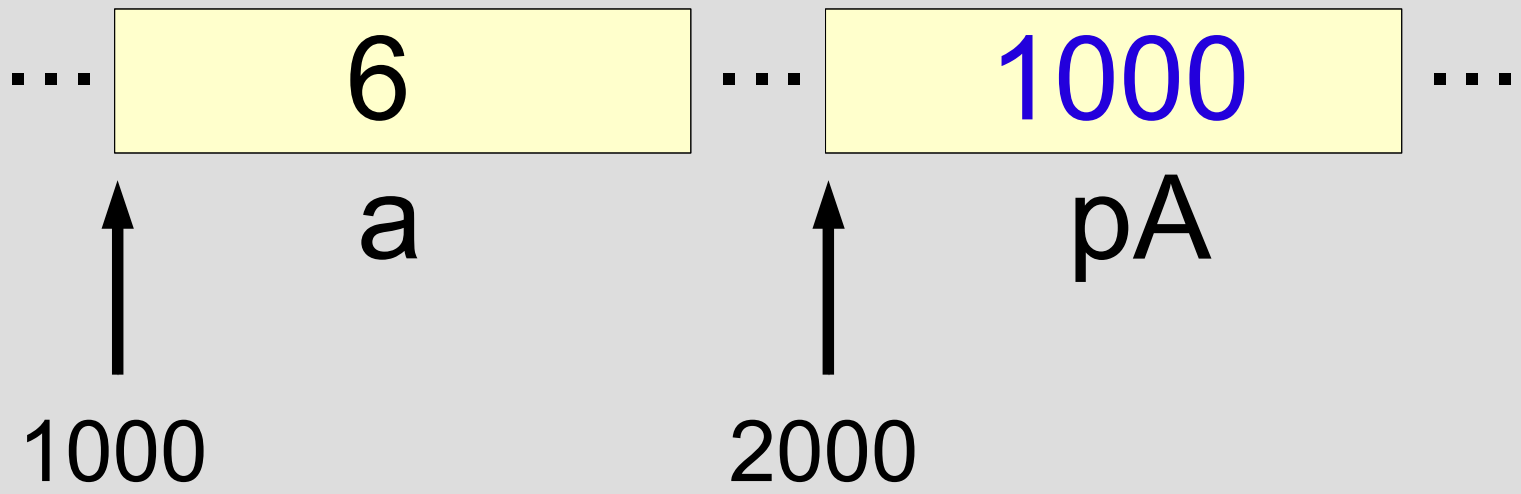
# Zeiger Beispiel 2/6

```
int* pA=0;
```



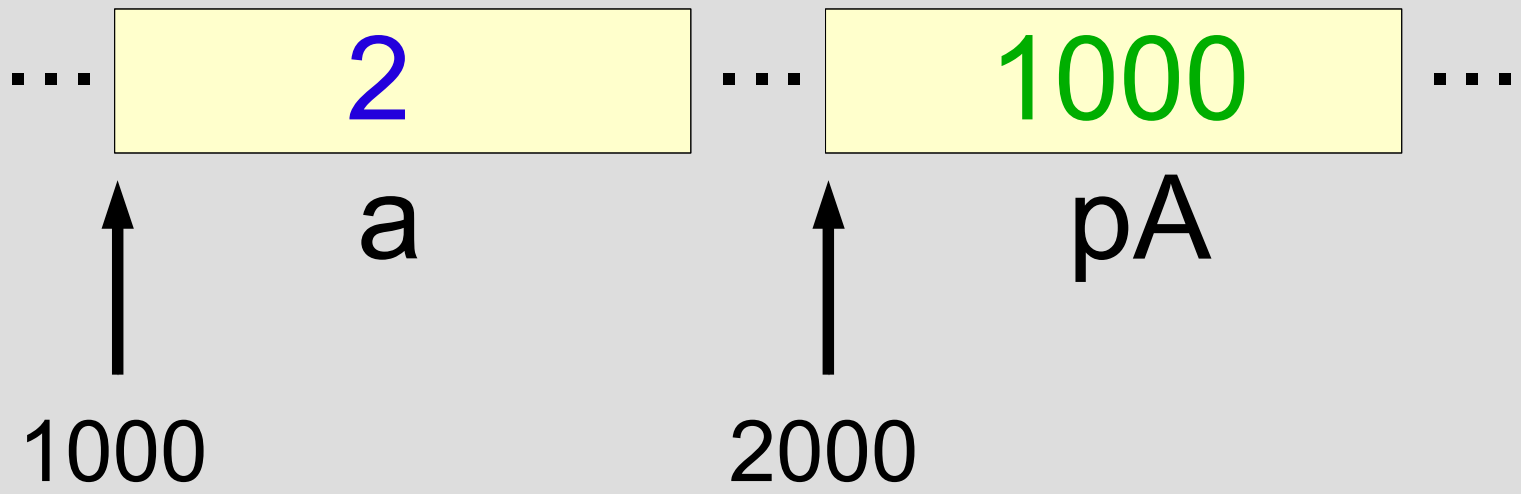
# Zeiger Beispiel 3/6

```
int* pA=&a;
```



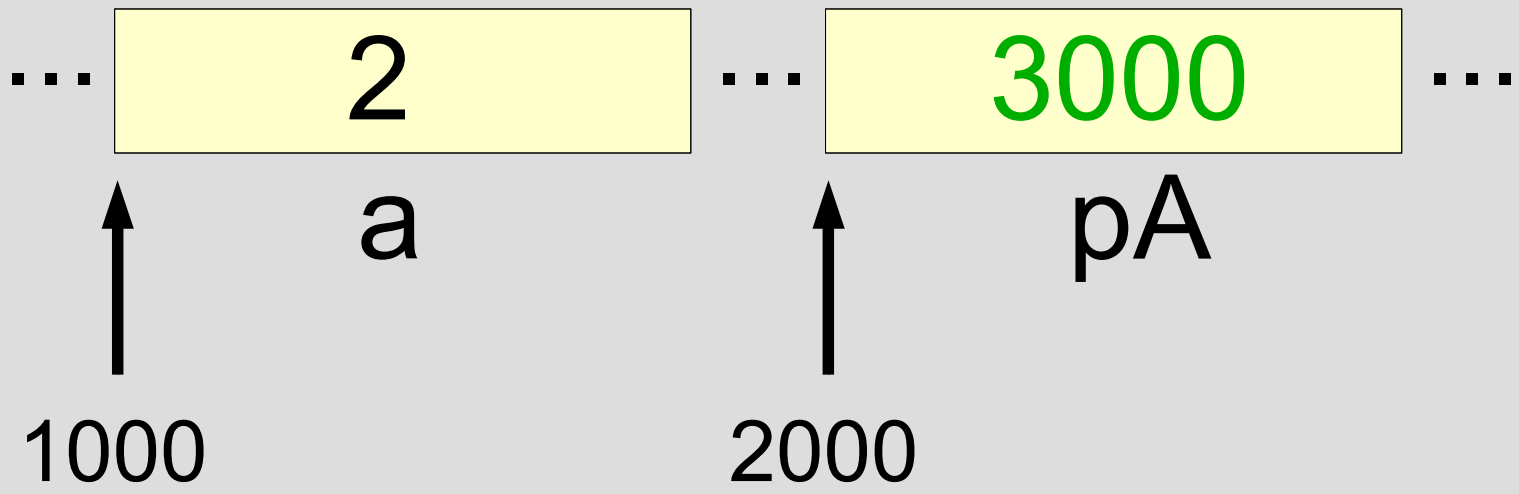
# Zeiger Beispiel 4/6

```
*pA=2;
```



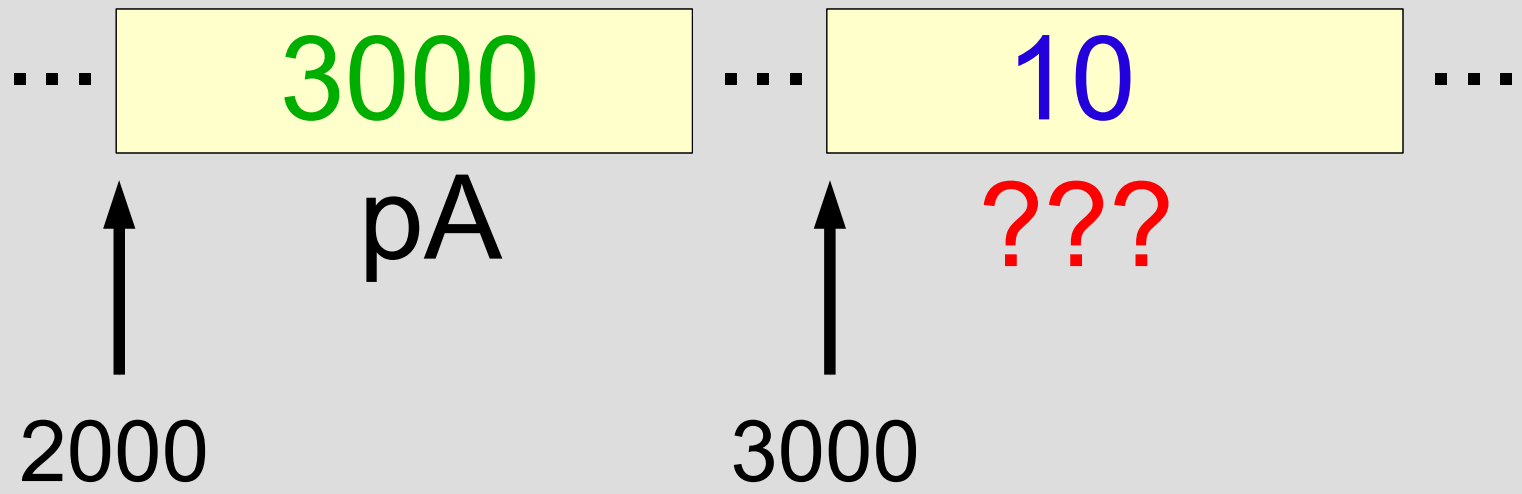
# Zeiger Beispiel 5/6

```
pA=3000;
```



# Zeiger Beispiel 6/6

```
*pA=10;
```



# const mit Zeigern 1/3

```
int a=6;

const int* pA=&a;

*pA=7; // Fehler. Wert auf den pA
       // zeigt ist konstant
```

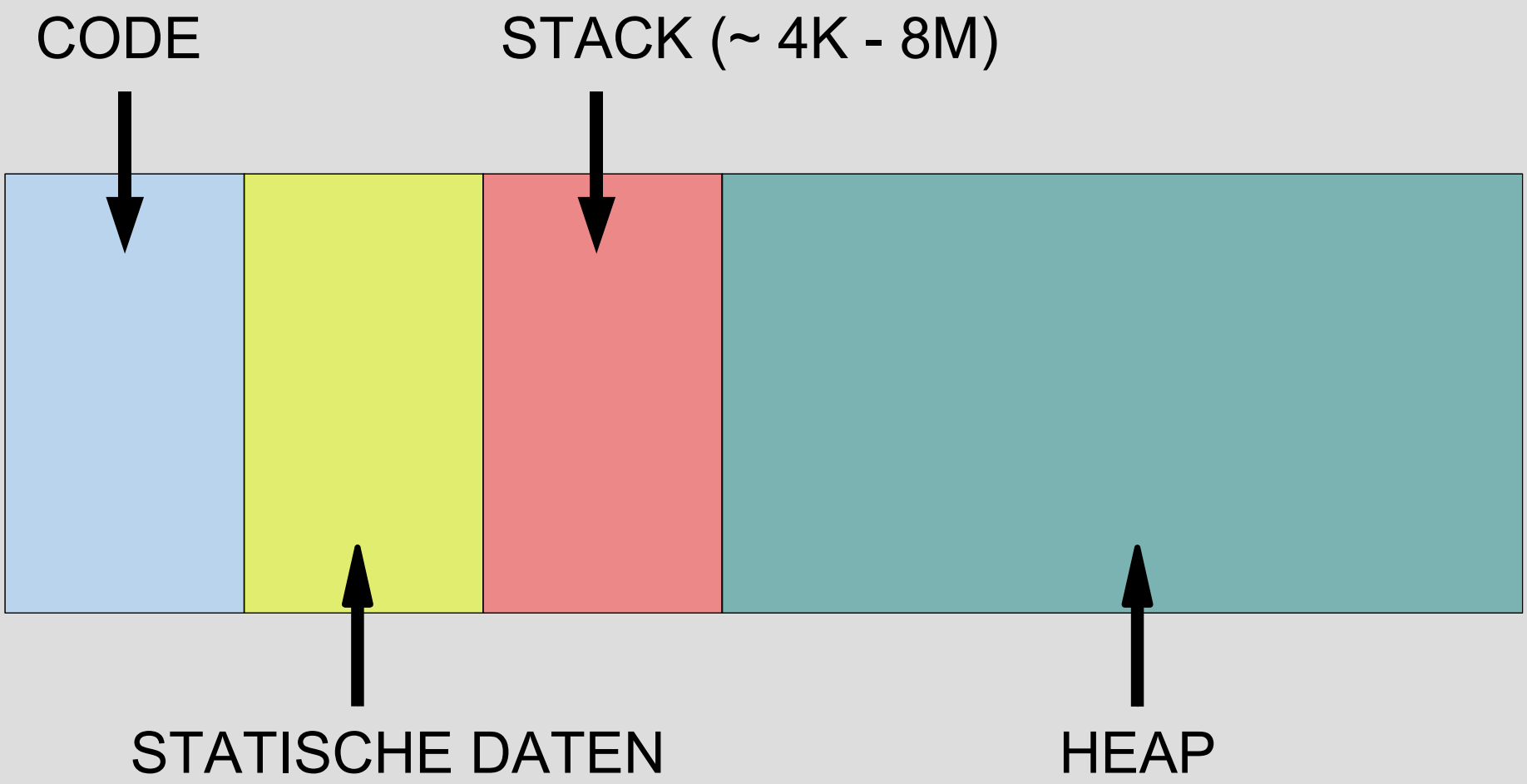
# const mit Zeigern 2/3

```
int a=6;

int b=12;

int* const pA=&a;

pA=&b;  // Fehler pA ist konstant
```

# const mit Zeigern 3/3

```
int a=6;

int b=12;

const int* const pA=&a;

pA=&b;  // Fehler pA ist konstant

*pA=12 // Fehler. Wert auf den pA
       // zeigt ist konstant
```

# Stack- & Heap-Speicher



# STACK <-> HEAP

- Stack ist schneller, aber Speicher ist begrenzt
- Heap ist etwas langsamer, aber es ist viel mehr Platz als im Stack

=> braucht man viele Objekte muss man sie auf dem Heap anlegen

# new / delete

```
Datum d1(15,Januar,2007); // STACK


Datum* d2;

d2=new Datum(1,April,2007); // HEAP

delete d2;
```

# Felder als Zeiger

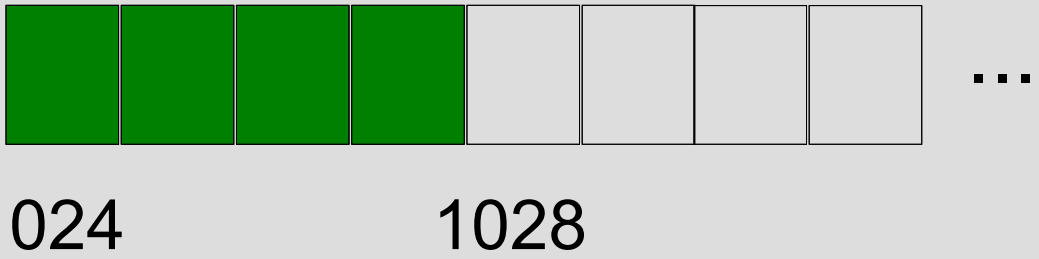
```
int feld[5] = {1, 2, 3, 4, 5};

int *pFeld=feld;

feld[2]=25; // -> *(feld+2)=25
```

# Zeigerarithmetik

```
int feld[5] = {1, 2, 3, 4, 5};

int *pFeld=feld; // = feld[0]

pFeld += 1; // = feld[1]
```

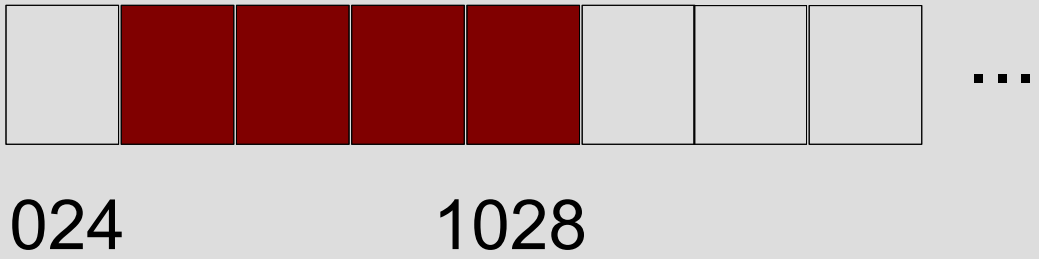
# Zeigerarithmetik

```
int* pInt=1024;
```



# Zeigerarithmetik

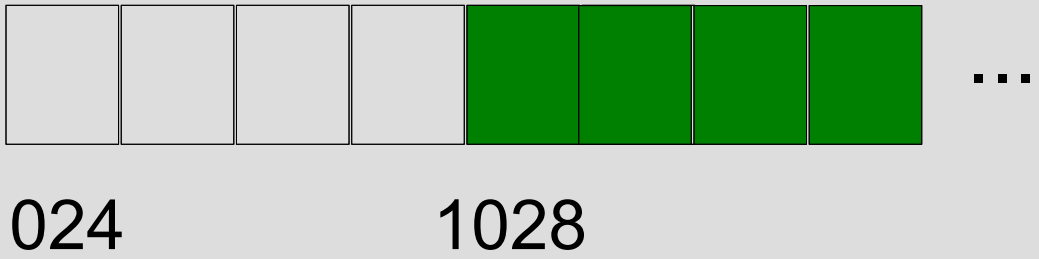
```
pInt += 1; // 1025
```




Falsch

# Zeigerarithmetik

```
pInt += 1; // 1028
```



# new[] / delete[]

```
Bruch *pBruch=new Bruch[20000];
...
delete[] pBruch;
```

# Zeiger als Felder

```
Bruch *pBruch=new Bruch[20000];

*(pBruch+10) = 100;

pBruch[10]=100;
```

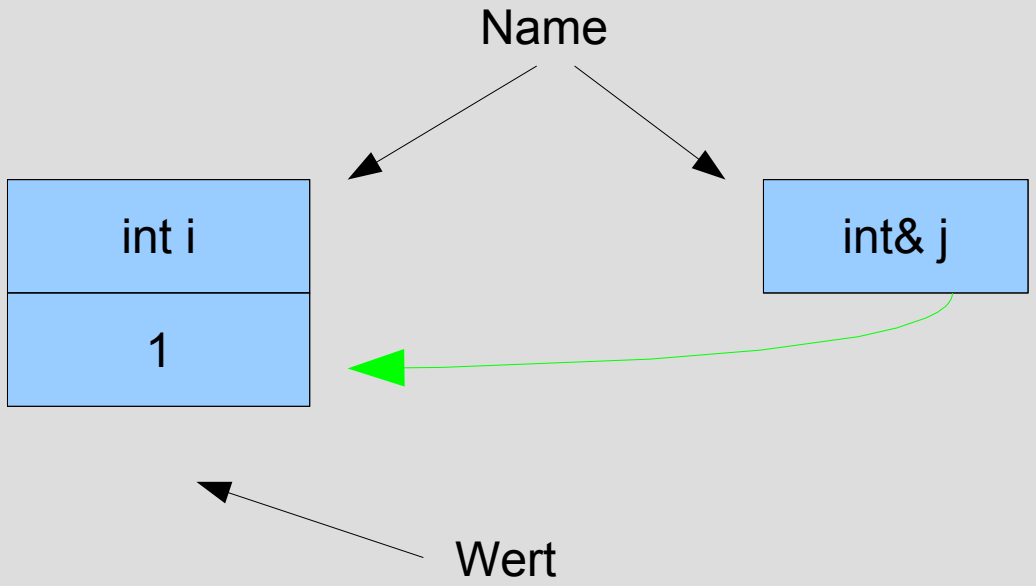
# wichtige Regeln für Zeiger

- wenn ein Zeiger keine gültige Adresse enthält, sollte man ihn mit dem Wert 0 initialisieren.
- mit **`new`** erzeugte Objekte immer mit **`delete`** wieder freigeben
- vor man einen Zeiger benutzt sollte man prüfen, ob er auf einen gültigen Wert zeigt (nicht 0)

# Referenzvariablen 1/2

- Weiterer Name für existierendes Objekt
- Kein selbständiges Objekt, nur „lebensfähig“ im Zusammenhang mit referenziertem Objekt

```
int i = 1;

int& j = i;
```



# Referenzvariablen 2/2

- Referenzvariable muß bei der Definition sofort initialisiert werden

```
// ERROR -- `i' deklariert aber nicht initialisiert
int& i;
```

- Referenzen sind unveränderlich --> können nicht wiederverwendet werden

# Referenzvariablen - Beispiel

```
int i = 5;
int& j = i;

i++;   // i, j haben welchen Wert?
j++;
```

# Referenzvariablen - Beispiel

```
int i = 5;
int& j = i;

i++;   // i, j haben welchen Wert?
j++;

int k = 1;
j = k;   // was passiert hier?
```

# Referenzvariablen - Beispiel

```
int i = 5;
int& j = i;

i++;   // i, j haben welchen Wert?
j++;

int k = 1;
j = k;   // !!! hier wird nicht die
         Objektreferenz verändert,
         sondern der Wert von k
         wird j und i zugewiesen!
```

# Referenzvariablen – Beispiel

```
int zahlen[5] = {5, 4, 3, 2, 1};

int& i = zahlen[3]; // i referenziert
                       einzelnes Element
                       im Feld zahlen


i++;  // Element 4 im Feld zahlen
         bzw. i wird um 1 erhöht
```

# Warum Referenzen?

- Man kann als Referenz übergebene Objekte in der aufgerufenen Funktion ändern
- sparen Stackspeicher und CPU Zeit wenn man sie als Funktionsparameter verwendet

# Call-By-Value Beispiel

```
void Bruch::mult(Bruch b1,Bruch b2)
{
  b1.zaehler *= b2.zaehler;
  b1.nenner  *= b2.nenner;
}

Bruch b1(1,2);
Bruch b2(1,2);
b1.mult(b1,b2);
```

# Call-By-Reference Beispiel

```
void Bruch::mult(Bruch& b1,Bruch& b2)
{
  b1.zaehler *= b2.zaehler;
  b1.nenner  *= b2.nenner;
}

Bruch b1(1,2);
Bruch b2(1,2);
b1.mult(b1,b2);
```

# Referenzparameter 1/2

- Formale Parameter in Funktionen, um Ergebnisse wieder zurückzugeben
- Nützlich für Funktionen mit **mehr als einem** Ergebnis, die zurückgegeben werden sollen

```
bool quot(int z, int n, int& quot, int& rest)
{
      if(n == 0)
         return(false);
      quot = z / n;
      rest = z % n;
      return(true);
}
```

# Referenzparameter 2/2

```
int main()
{
      int q, r;
      quot(2, 3, q, r);          // ok

      int v[5]= {5,4,3,2,1};
      quot(4, 5, v[1], v[3]); // ok

      int& ir = q;
      quot(4, 5, r, 1); // "1" ist kein Objekt
                        // -> Ergebnis ist weg
}
```

# `const` Referenzen

```
int i = 5;
const int& j = i;

i++;
j++;// Fehler, j ist const
```

# const Referenz Parameter

```
void Bruch::mult(Bruch& b1,
                 const Bruch& b2)
{
  b1.zaehler *= b2.zaehler;
  b1.nenner  *= b2.nenner;
}

Bruch b1(1,2);
Bruch b2(1,2);
b1.mult(b1,b2);     // b1=1/4
```

# `const` Referenz Parameter

```
void Bruch::mult(const Bruch& b2)
{
  zaehler *= b2.zaehler;
  nenner  *= b2.nenner;
}

Bruch b1(1,2);
Bruch b2(1,2);
b1.mult(b2);     // b1=1/4
```