Die Aufgabe konkret

```
enum {MAX_VEKTOR_SIZE=10};
struct Vektor{
   int daten[MAX_VEKTOR_SIZE];
   int dimension;
};
void Init(Vektor& v1, int wert, int size);
void Init(Vektor& v1, int wert);
void Plus(const Vektor& v1,
          const Vektor& v2, Vektor& erg);
```

Wir ändern gegenüber der bisheriger Implementierung:
Die Vektorgröße wird in Init übergeben.
Bis zur Größe 10 ist erlaubt.

Die Aufgabe konkret (2)

```
void Init(Vektor& v1, int wert) {
    for (int i=0; i<v1.dimension; ++i){
        v1.daten[i] = wert;
    }
}
void Init(Vektor& v1, int wert, int size) {
    assert(size < MAX_VEKTOR_SIZE);
    v1.dimension = size;
    Init(v1, wert);
}
```

Test

```
/** Es wird ein Vektor erzeugt und mit dem Wert 81
gefüllt. Anschließend wird geprüft, ob auch jedes
Element den Wert 81 hat.
*/
bool testInit() {
    Vektor v;
    const int wert=81;
    const int vekSize = 4;
    Init(v, wert, vekSize);
    for (int i=0; i < vekSize; ++i)   {
        if (v.daten[i] != wert) {
            return false;
        }
    }
    return true;
}
```

Test zum Vektor-Vergrößern

```
/** Es wird ein Vektor mit 4 Elementen erzeugt
und mit dem Wert 80 gefuellt.
Anschliessend wird die Dimension des Vektors
auf 100 erhoeht und jedes Element mit 81 belegt.
Dann wird geprueft, ob auch jedes
der 100 Elemente den Wert 81 hat. */
bool testResize() {
   Vektor v;
   const int wert = 81;
   const int vekSize = 4;
   Init(v, wert - 1, vekSize);
   const int newVekSize = 100;
   Resize(v, newVekSize);
   Init(v, wert);
   for (int i = 0; i< newVekSize; ++i) {
      if (v.daten[i] != wert) {
         return false;
      }
   }
   return true;
```

Bauen Sie mal Resize!!!

Einschub: Zeiger

Zeiger - Motivation

```
struct Vektor{
   int anz;
   int data[3];
};
```

```
Vektor v;
v.anz=3;
const int wert=81;
Init(v, wert);
int k=42;
```

```
Init(Vektor& v, int w) {
for (int i=0; i < v.anz; ++i) {
   v.data[i] = w;
  }
}
```

Speicherbelegung

v 1000	anz	3
1004	data[0]	81
1008	data[1]	81
1012	data[2]	81
wert 1016		81
k 1020		42

Zeiger - Motivation

```
struct Vektor{
   int anz;
   int data[4];
};
```

```
Vektor v;
v.anz=4;
const int wert=81;
Init(v, wert);
int k=42;
```

```
Init(Vektor& v, int w) {
for (int i=0; i < v.anz; ++i) {
   v.data[i] = w;
  }
}
```

Speicherbelegung

v 1000	anz	4
1004	data[0]	81
1008	data[1]	81
1012	data[2]	81
1016	data[3]	81
wert 1020		81
k 1024		42

Wie können wir die Größe des Vektors zur Laufzeit vergrößern, d.h. ein dynamisches Array data mit z.B. 600 Elementen erhalten?

Die Variablen wert und k werden aber weiterhin die Speicherstellen ab 1020 belegen?

Zeiger – Motivation / Funktionen

```
int k=42;
int arr[4];
arr[2]=15;
int j=11;
```

Speicherbelegung

```
v 1000   k      42
  1004   arr[0]   ?
  1008   arr[1]   ?
  1012   arr[2]   15
  1016   arr[3]   ?
j 1020             11
```

```
void funk(int a) {
   int k= 11;
   a=64;
};

int main() {
   int j=4;
   funk(j);
   int y = 19;
}
```

```
1000  j   4
1004  a   4
1008  k   11
```

Speicherbelegung

```
1000  j   4
1004  a   64
1008  k   11
```

Speicherbelegung

```
1000  j   4
1004  y   19
```

Clicker

```
void funk(int a) {
   int k= 11;
   a=64;      /* jetzt */
};

int main() {
   int j=4;
   funk(j);
   int y = 19;
}
```

Speicherbelegung 1

1000	j	4
1004	a	64
1008	k	11

Speicherbelegung 2

1000	j	64
1004	a	64
1008	k	11

Welche Speicherbelegung ist richtig?

1. 4 / 64
2. 64 / 64

Zeiger – Motivation / Funktionen mit Referenzen

```
void funk(int& a) {
   int k= 11;
   a=64;
};

int main() {
   int j=4;
   funk(j);
   int y = 19;
}
```

1000	j	4
1004	a	1000
1008	k	11

Speicherbelegung

1000	j	64
1004	a	1000
1008	k	11

Speicherbelegung

1000	j	64
1004	y	19

Clicker

```
void funk(int& a) {
   int k= 11;
   a=64;      /* jetzt */
};

int main() {
   int j=4;
   funk(j);
   int y = 19;
}
```

Speicherbelegung 1

1000	j	64
1004	a	1000
1008	k	11

Speicherbelegung 2

1000	j	64
1004	a	64
1008	k	11

Welche Speicherbelegung ist richtig?

1. 64 / 1000
2. 64 / 64

Zeiger – Motivation / Arrays

```
void funk(int a[]) {
   int k=11;
   a[2] = 9;
};

int main() {
   int j=4;
   int arr[3];
   arr[2] = 5;
   funk(arr);
   int y = 19;
}
```

```
1000  j    4
1004  arr[0] ?
1008  arr[1] ?
1012  arr[2]  5
1016  a    1004
1020  k    11
```

Speicherbelegung

```
1000  j    4
1004  arr[0] ?
1008  arr[1] ?
1012  arr[2]  9
1016  a    1004
1020  k    11
```

Speicherbelegung

```
1000  j    4
1004  arr[0] ?
1008  arr[1] ?
1012  arr[2]  9
1016  y    19
```

Zeiger – Motivation: Zurück zum Ausgangsproblem

```
struct Vektor{
   int anz;
   int data[4];
};
```

```
Vektor v;
v.anz=4;
const int wert=81;
Init(v, wert);
int k=42:
```

Gewünschte Speicherbelegung

```
v 1000   anz      4
  1004   data   4000
k 1008           42
```

```
4000 data[0]   81
4004 data[1]   81
4008 data[2]   81
4012 data[3]   81
```

Zeiger – Motivation: Zurück zum Ausgangsproblem

```
struct Vektor{
   int anz;
   int* data;
};
```

data speichert nun eine Adresse.
Es ist ein Zeiger.

```
Vektor v;
v.anz=4;
const int wert=81;
Init(v, wert);
int k=42:
```

Gewünschte Speicherbelegung

```
v 1000   anz      4
  1004   data   4000
k 1008           42
```

```
4000 data[0]   81
4004 data[1]   81
4008 data[2]   81
4012 data[3]   81
```

Zeiger – Motivation: Zurück zum Ausgangsproblem (2)

Gewünschte Speicherbelegung

```
v 1000   anz      4
  1004   data   4000
k 1008           42
```

```
4000 data[0]   81
4004 data[1]   81
4008 data[2]   81
4012 data[3]   81
```

Vergrößerung wäre einfach

```
v 1000   anz      100
  1004   data   7000
k 1008           42
```

```
7000 data[0]   81
7004 data[1]   81
…
7392 data[98]   ?
7396 data[99]   ?
```

Zeiger - Definition von Zeigervariablen

Bei der Definition eines Zeigers wird der Dereferenzierungsoperator ("*") vor den Bezeichner geschrieben. Werden die Bezeichner mit Kommas getrennt in einer Definitionsliste angegeben, so muss der Dereferenzierungsoperator vor jeden einzelnen Bezeichner geschrieben werden, der als Zeiger definiert werden soll.

Aus Gründen der Übersichtlichkeit empfiehlt es sich jedoch, **jede** Variable in einer eigenen Zeile zu definieren. Der Stern sollte direkt (ohne Blank) hinter dem Typen stehen:

```
float summe;
float* psumme;
```

Besser nicht so:

```
float summe,  * psumme;
long *LZ, LZ2:
```

Zeiger - Adressoperator

Um die Adresse einer Variablen (einen sogenannten L-Wert) zu erhalten, muss ein spezieller Operator verwendet werden.

(Lvalue = location for a value)

Dieser Operator wird als Adressoperator bezeichnet. Er wird mit dem Zeichen "&" (Ampersand) symbolisiert.

Zum Beispiel:

```
int Io = 1024;
int* IZ = &Io;          // IZ erhält die Adresse von Io
                        // und nicht den Wert 1024
```

Zeiger – Adressoperator (2)

Ein Zeiger kann auch mit einem anderen Zeiger desselben Typs initialisiert werden.

```
int Io = 1024;
int* IZ = &Io;
int* IZ2 = IZ: // ok jetzt enthält auch IZ2 die Adresse von Io

   int* IZ3 = &IZ: // Fehler

         int ** IZ4 = &IZ:   // Zeiger auf Zeiger
```

Der *- Operator

„*****“ ist der Inhaltsoperator. Damit wird ein Zeiger dereferenziert. Die Vereinbarung ***„int* pi;“*** oder auch ***„int *pi;“*** kann gelesen werden als „*der Inhalt von pi ist vom Typ int, also ist pi vom Typ Zeiger auf int*“:

```
int i = 1234;
int* pi;          // pi ist also vom Typ „Zeiger auf int“
pi = &i;          // pi zeigt jetzt auf i
*pi = 5678;       // i hat jetzt den Wert 5678,
                  // der Zugriff auf i erfolgt hier durch Dereferenzieren
                  // des Zeigers pi.
```

Also:

```
Typ* ptr;                  // hier wird eine Zeigervariable definiert
*ptr = ...; ... = *ptr;    // hier wird ptr dereferenziert und damit auf
                              den Inhalt zugegriffen
```

Referenz / Zeiger oder Wert als Übergabeparameter

```
void funkPtr(int* i) {
 *i = 64;}
void funkWert(int i) {
 i = 127;}
void funkRef(int& i) {
 i = 888;
}
```

```
void main(){
   int j=11;
   funkWert(j);
   cout << j;  // 11
   funkPtr(&j)
   cout << j; // 64
   funkRef(j)
   cout << j; // 888
}
```

Clicker

```
int i = 1234;
int* pi;
pi = &i;
*pi = 14;
            .
cout << i <<  " "  << (*pi);
```

Bildschirmausgabe ?

1. 1234 14
2. 1234 1234
3. 14 14
4. 14 1234

Zeiger und Typen

Jeder Zeiger ist mit einem Objekt eines bestimmten Typs verbunden
Der Datentyp gibt den Typ des Datenobjekts an, das mit Hilfe dieses Zeigers adressiert wird.
Ein Zeiger des Typs *int* zeigt zum Beispiel auf ein Objekt des Typs *int*.

```
int   intvar;
int*  intZeiger = &intvar;
```

Um auf ein Objekt des Typs *double* zu zeigen, muss der Zeiger mit dem Datentyp *double* definiert werden.

```
double dvar;
double*  dZeiger = &dvar;
```

Test zum Vektor-Vergrößern

```
/** Es wird ein Vektor mit 4 Elementen erzeugt
und mit dem Wert 80 gefuellt.
Anschliessend wird die Dimension des Vektors
auf 100 erhoeht und jedes Element mit 81 belegt.
Dann wird geprueft, ob auch jedes
der 100 Elemente den Wert 81 hat. */
bool testResize() {
   Vektor v;
   const int wert = 81;
   const int vekSize = 4;
   Init(v, wert - 1, vekSize);
   const int newVekSize = 100;
   Resize(v, newVekSize);
   Init(v, wert);
   for (int i = 0; i< newVekSize; ++i) {
      if (v.daten[i] != wert) {
         return false;
      }
   }
   return true;
```

Bauen Sie jetzt Resize!!!

Clicker

Wie machen wir weiter?

1. Alles klar, wie man das realisiert, ist mir klar. Würde ich gerne jetzt alleine machen (ggf. mit Hilfen vom Dozenten).

2. Alles klar, wie man das realisiert, ist mir klar. Dozent soll einfach die Lösung „vorturnen“. Ich weiß, wie ich Speicher vom Betriebssystem anfordern und auch wieder freigeben kann.
3. Noch sehr nebelig, wie die eigene Lösung aussehen könnte. Ich brauche noch mehr Inputs und in 15 Minuten entscheiden wir dann ggf. nochmals.

4. Das interessiert mich jetzt gar nicht, hätte gerne ein anderes Thema.

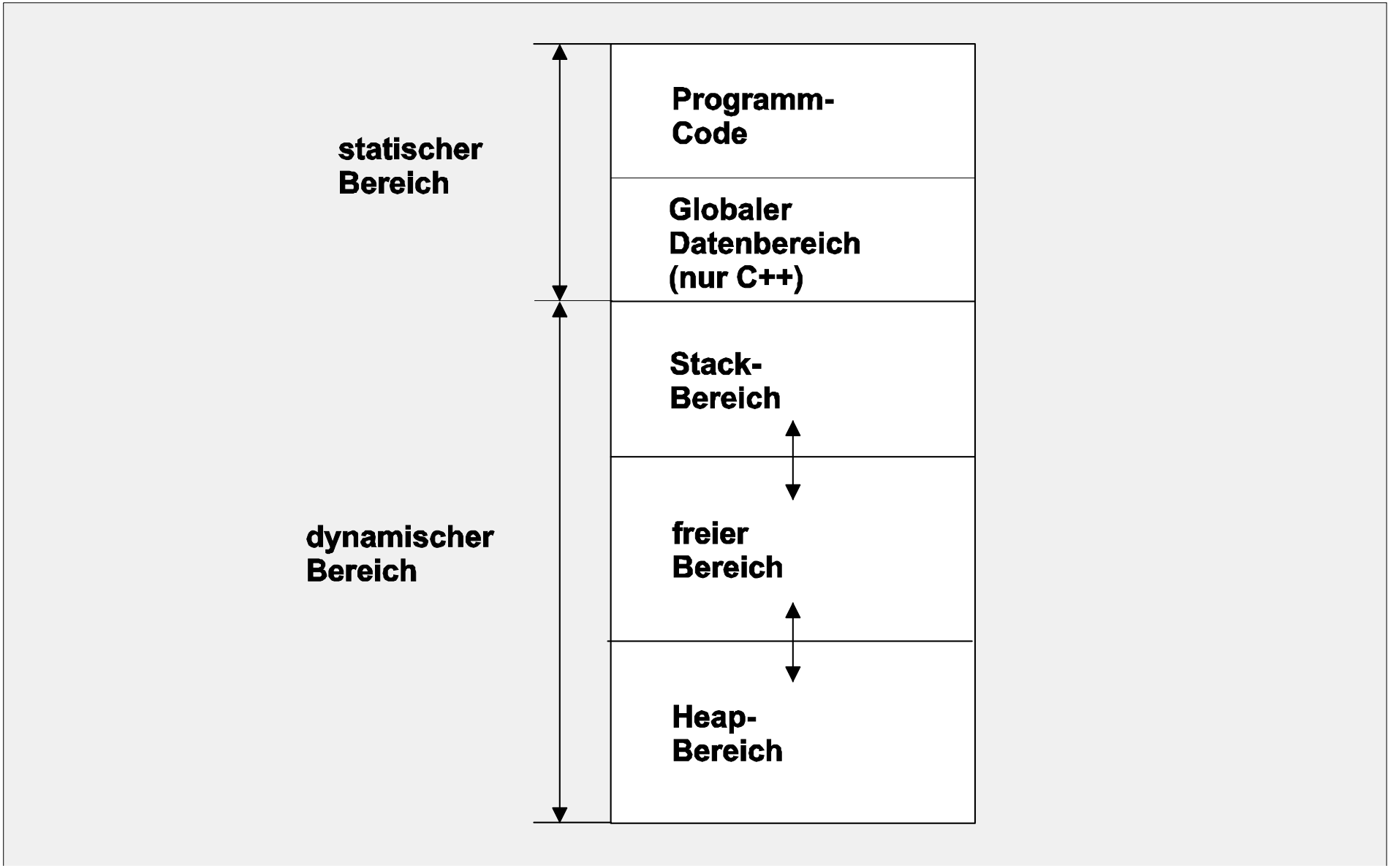
Einschub: Dynamische Speicherverwaltung auf dem Heap bzw. auf dem Stack

Einführung

In C/C++ hat der Benutzer generell zwei verschiedene Möglichkeiten, auf Daten zuzugreifen:

- Er kann direkt mit dem Wert arbeiten (**Wertesemantik**), oder
- er kann indirekt über einen Zeiger auf den Wert zugreifen (**Zeigersemantik**).
- Bei der Wertesemantik werden die Daten vom System auf dem *Programmstack* verwaltet, und
- Bei der Zeigersemantik muss der Programmierer explizit auf dem *Programmheap* Speicherplatz reservieren und später auch wieder freigeben.

Speicherverwaltung für C++- und Java-Programme



Werte- und Zeigersemantik

Wertesemantik:

```
int a, b;
a = 123;
b = a;
. . .
```

Zeigersemantik in C++:

```
int *a, *b;
. . .
a = new int;
b = new int;

*a = 123;
*b = *a;
. . .
delete a;
delete b;
```

Operator *new*

Mit dem Operator *new* wird Speicher im Heap-Speicher angefordert und einer Zeigervariablen zugewiesen:

```
int *iptr =   new int;

int *kptr = new int(64);  // Initialisierung mit 64

int *jptr = new int [30];  // Speicherbereich für 30 int-Typen
//  Es ist nicht garantiert, dass sie (für Standardtypen) initialisiert werden.
```

Wenn in C++ eine Speicheranforderung nicht erfüllt werden kann, wird ein Ausnahmeobjekt vom Typ *bad_alloc* erzeugt, auf das dann im Rahmen einer Ausnahmebehandlung reagiert werden kann, Details dazu siehe den Teil *Fehler- und Ausnahmebehandlung*.

Operator delete

Wird der über new angeforderte Speicherbereich nicht mehr benötigt, muss er mittels delete explizit wieder freigegeben werden:

```
int *pint = new int;
int *pintfeld = new int [64];
...
delete pint;          // Speicher für einen int wird freigegeben.
delete [] pintfeld;   // Speicher für ein int-Feld wird freigegeben.
```

delete kann nur in Zusammenhang mit new eingesetzt werden:

```
int i, *pint = new int;
int *pint2 = &i;
pint = pint2;
delete  pint;  // Fehler, da Speicher nicht durch new angefordert wurde
```

Die Anwendung von delete auf einen NULL-Zeiger hat keine Auswirkung.

Anwendung von new / delete: Dynamische Felder

```
char name[20];

strcpy(name, “Helmke“);




strcpy(name,
   “Leutheuser-Schnarrenberger“);
// Das Verhalten ist hier undefiniert.
```

```
char* pname;
...
pname =  new char[20];

strcpy(pname, “Helmke“);

delete [] pname;
pname = new char[50];

strcpy(pname,
   “Leutheuser-Schnarrenberger“);
```

Werte- und Zeigersemantik mit Vektoren

Wertesementik

```
int vector[10];
. . .
. . .
. . .
. . .
vector[i]   = x;        /* so      */
*(vector+i) = x;        /* oder so */
. . .
y = vector[j];          /* so      */
y = *(vector+j);        /* oder so */
. . .
```

Zeigersemantik

```
int* vector;

vector = new int[10];
//vector = (int*)malloc(10*sizeof(int));

. . .
vector[i]   = x;        /* so      */
*(vector+i) = x;        /* oder so */
. . .
y = vector[j];          /* so      */
y = *(vector+j);        /* oder so */
. . .
delete[] vector;
/* bzw. free(vector); */
```

Entweder *malloc* und *free* oder *new* und *delete*, nicht mischen!

nullptr; (NULL-Zeiger, 0-Zeiger)

Es gibt einen besonderen Zeiger, der als Konstante mit dem Namen nullptr definiert ist. Der nullptr-Zeiger kann jeder Zeigervariablen zugewiesen werden, und jede Zeigervariable kann auf den Wert nullptr getestet werden;

```
p = nullptr;
if (p == nullptr)
```

NULL-Zeiger, 0-Zeiger

Vor C++-11 wurde der die Konstante NULL verwendet. Sie ist nicht mehr zu empfehlen.

Es gibt einen besonderen Zeigerwert, der als Konstante mit dem Namen NULL definiert ist. Die interne Darstellung der NULL ist implementierungsabhängig, aber kompatibel zur ganzzahligen Null (0). Der NULL-Zeiger kann jeder Zeigervariablen zugewiesen werden, und jede Zeigervariable kann auf den Wert NULL getestet werden;

```
p = NULL;              /* Beide Anweisungen sind        */
p = 0;                 /* korrekt und bedeutungsgleich! */
. . .
if (p==NULL) . . .     /* Alle drei                     */
if (!p) . . .          /* Abfragen sind korrekt         */
if (p==0) . . .        /* und bedeutungsgleich!         */
```

nullptr (2)

Diese Operationen können auch dazu verwendet werden, um zu ermitteln, ob ein Zeiger schon initialisiert wurde:

```
int* ptr = nullptr;
int* ptr2;
. . .

if (ptr != nullptr)
{
   if (ptr == ptr2)
      . . .
```

nullptr

Ein delete/delete[] auf einen Zeiger mit nullptr-Wert ist immer erlaubt. Der Aufruf macht einfach gar nichts.

Ein Zeiger ist am Anfang vom Compiler nicht initialisiert, man muss dieses explizit durchführen (später werden wir diese Aufgabe in den Konstruktor verlegen.

```
if (ptr != nullptr)
{ // Abfrage nicht erforderlich
   delete ptr;
      . . .
```

```
delete ptr;
// guter Programmiersitl
// verhindert doppelt Freigeben
ptr = nullptr;
      . . .
```

Empfehlungen

„delete“ auf nullptr immer erlaubt. Es hat keine Wirkung.

```
if (p!= nullptr)  {delete p;}
```

kann vereinfacht werden zu:

```
delete p;  // Intern (auf Assemblerebene) wird auf nullptr abgefragt
```

Nach delete-Aufruf den Zeiger immer mit nullptr initialisieren

```
delete p;
p = nullptr;  // dann kann ein weiteres delete p; keinen Schaden
              // anrichten
```

Test zum Vektor-Vergrößern

```
/** Es wird ein Vektor mit 4 Elementen erzeugt
und mit dem Wert 80 gefuellt.
Anschliessend wird die Dimension des Vektors
auf 100 erhoeht und jedes Element mit 81 belegt.
Dann wird geprueft, ob auch jedes
der 100 Elemente den Wert 81 hat. */
bool testResize() {
   Vektor v;
   const int wert = 81;
   const int vekSize = 4;
   Init(v, wert - 1, vekSize);
   const int newVekSize = 100;
   Resize(v, newVekSize);
   Init(v, wert);
   for (int i = 0; i< newVekSize; ++i) {
      if (v.daten[i] != wert) {
         return false;
      }
   }
   return true;
```

Bauen Sie mal Resize!!!

Hausaufgabe

Machen Sie das gleiche nochmals für eine Matrix.
Ich werde es nicht kontrollieren, aber wenn Sie wollen dürfen Sie mir gerne Ihren Code schicken (vor allem wenn er nicht läuft).

```
/
bool test1()   {
   const int ze=3;
   const int sp=4;
   Matrix m1;
   Init(m1, ze, sp);

   for (int i = 0; i < ze; ++i) {
      for (int j=0; j < sp; ++j) {
         Setze(m1,i,j, i*sp + j*2);
         }
      } // for i
```

```
   for (int i = 0; i < ze; ++i) {
      for (int j=0; j < sp; ++j) {
         if (Lese(m1,i,j) !=  (i*sp + j*2)) {
            Free(m1);
            return false;
            }
         }
      } // for i
Free(m1);
return true;
}
```

Die Aufgabe konkret

```
/**
Es wird eine 3*4 Matrix angelegt und einige Werte
werden gesetzt, die dann ausgelesen werden.
Liefert das Auslesen die erwarteten Werte, wird true
geliefert
*/
bool test1()
```

```
const int ze=3;
const int sp=4;
Matrix m1(ze,sp);

for (int i = 0; i < ze; ++i) {
   for (int j=0; j < sp; ++j) {
      m1.Setze(i,j, i*sp + j*2);
      }
   } // for i
```

```
for (int i = 0; i < ze; ++i) {
   for (int j=0; j < sp; ++j) {
      if (m1.Lese(i,j) !=  (i*sp + j*2)) {
         return false;
         }
      }
   } // for i
return true;
}
```

Clicker

Wie machen wir weiter?

1. Wollen Sie nun den Stand der Klasse Vektor haben und alleine (mit Dozenten Unterstützung, wenn erforderlich) die Lösung zu Ende entwickeln?
2. Alles klar, ich würde nun gerne mit Klasse Matrix/Sprachausgabe beginnen.
3. Dozent soll Klasse „Vektor“ vor-turnen und Code hochladen. Dann versuche ich mich hier an der Klasse „Matrix“ und versuche den Rest zu Hause.
4. Dozent soll Klasse „Vektor“ vorturnen und Code hochladen. Dann versuche ich mich hier in der Vorlesung an der Klasse „Matrix“. Zu Hause werde ich aber nichts machen können.

Sie finden den Code hier

https://public.ostfalia.de/~helmke/

3. Vorlesung; Fr. 17.10.2025

Folienkopien	Inhalt
Wiederholung / Ankündigung (11.10.2025)	Wiederholung aus der le
Aufteilung von Code auf Dateien (01.09.2025)	Die Aufteilung von Cod
Motivation von Zeigern und Heap-Speicher (01.09.2025)	Wie kann man Vektoren
Programm-Code	**Inhalt**
Ausgangscode für Aufteilung von Vektor auf Dateien (11.10.2025)	
Lösung fürVisual Studio 2022, (11.10.2025) CMake (11.10.2025)	In der Vorlesung wird de erklärt.
Ausgangscode für dynamischen Vektoren (11.10.2025)	
Lösung für Visual Studio 2022, (11.10.2025) CMake (11.10.2025)	In der Vorlesung wird de

Clicker

Feedback

1. Das war alles viel zu schnell. Ich hätte gerne mehr Übungen heute gehabt.
2. Das war ganz viel Neues. Habe ich grob verstanden, aber beim Selbermachen wird es nebelig. Ich werde es mir zu Hause nochmals anschauen und glaube, dass sich der Nebel dann lichtet.
3. Viel Neues, aber ich denke ich kriege die Matrixübung/Sprechfunk nun auch zu Hause hin.
4. Das meiste kannte ich.