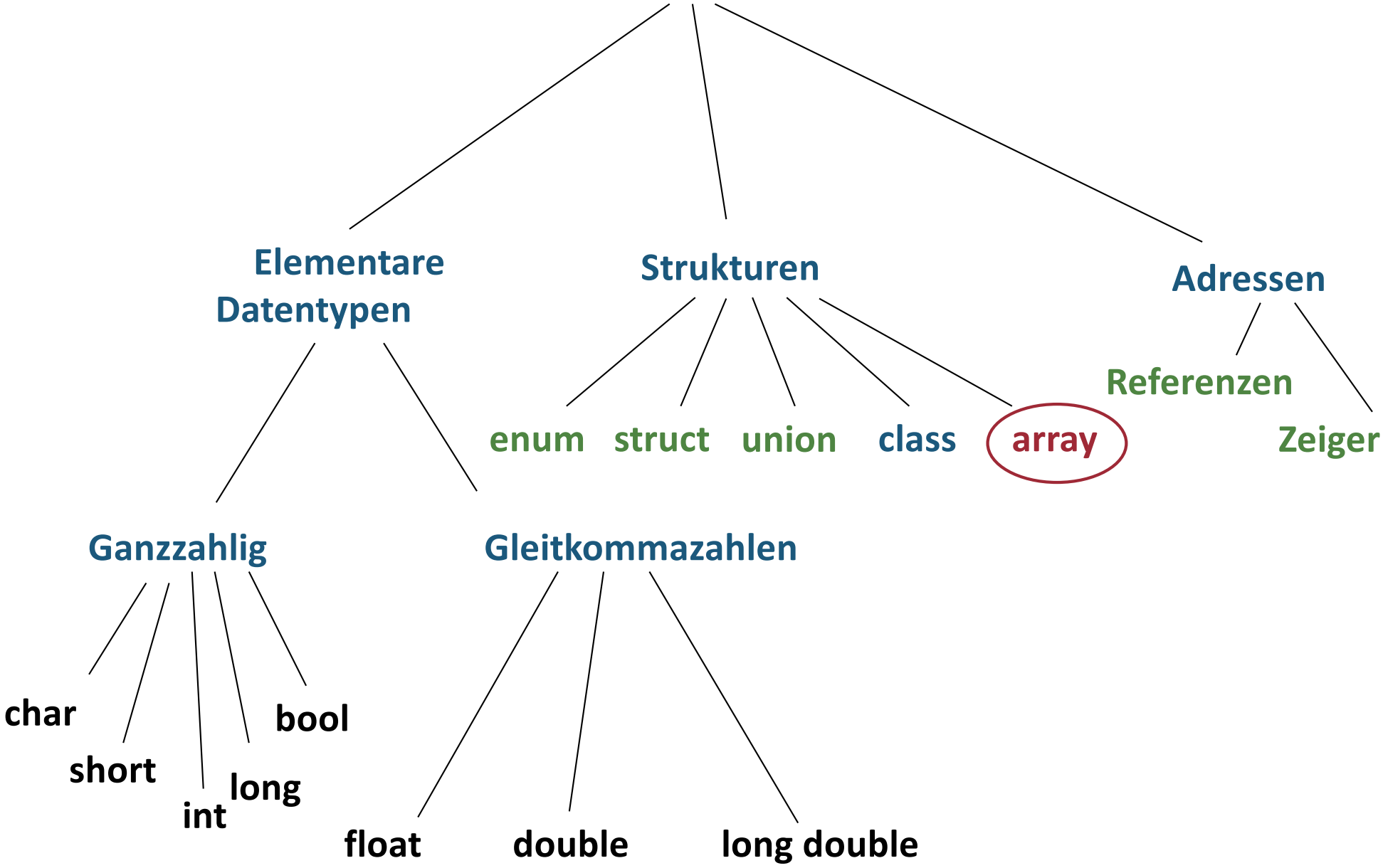
K04

Zeiger & Co

1. Zeiger
   - Verwendung und Zeigerarithmetik
2. Referenzen
3. **Arrays**
4. Zeigertabellen
5. Funktionszeiger

Wiederholung: Das Typsystem in C++

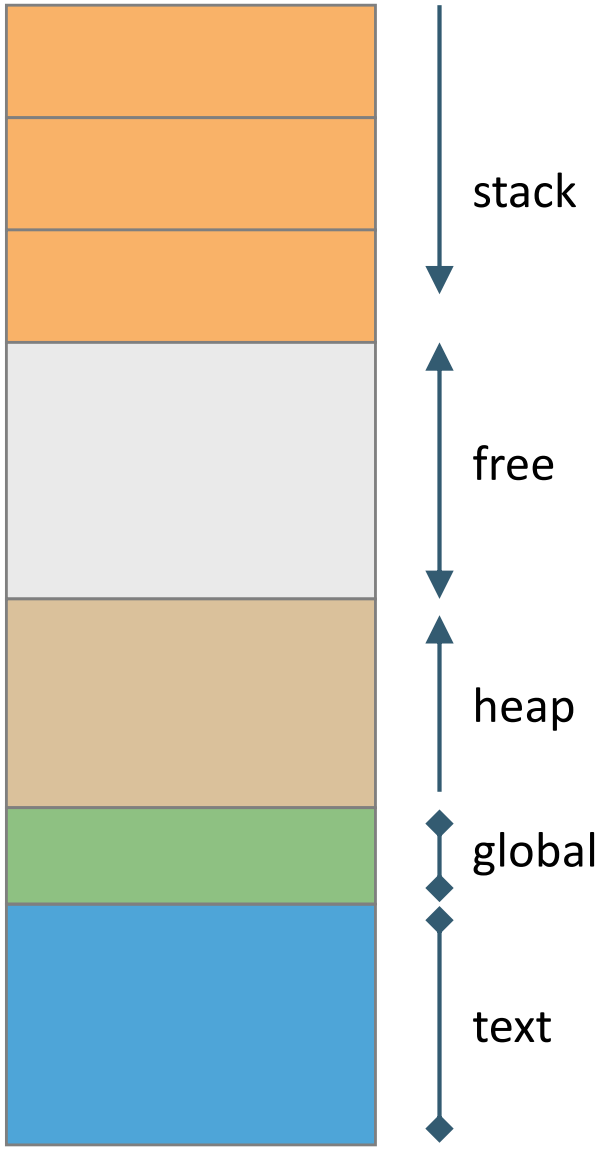


Wiederholung: Speichermodell

- C++ Programme enthalten drei Arten von Variablen:
  - Lokale Variablen
  - Globale, statische Variablen
  - Dynamisch allokierte Variablen

Beispiel:

```
int          n = 1;
static int m = 2;
int main ()
{
   float x, *y;
   static int y = 2;
}
```



Freispeicher: Nutzung & Verwaltung

- Problem: Wie können dynamische Daten im Speicher gehalten werden, deren Lebenszeit nur zur Laufzeit bestimmt werden kann (also beim Kompilieren noch nicht bekannt ist)?
  - Globale oder lokale Variablen funktionieren nicht.

- Idee:
  - Zeiger verwenden, um auf diese dynamischen Daten zu zeigen
  - Heap mit **`new`** und **`delete`** (reservierte Schlüsselwörter in C++)

- **`new`**: Speicher**anforderung** für Zeigervariable
- **`delete`**: Speicher**freigabe** von Zeigervariable

Der **new**-Operator

```
// Anlegen einer einzigen Zeigervariable von „Type“
Type* x = new Type;
// Anlegen eines Arrays von „Type“
Type* x = new Type[intexpr];
Type* x = new Type(arglist); // Initial. über Konstruktor
int* i = new int, *j = new int(4);
```

- Mit **new** werden neue Instanzen von Elementartypen, Strukturen, oder Klassen angelegt bzw. der benötigte Speicher reserviert
  - Bei Elementartypen erfolgt per Default keine Initialisierung; wenn gewünscht dann durch z.B. **int(*n*)** wobei ***n*** der Initialwert ist
  - Bei Klassen kann Konstruktor angeben werden; ansonsten erfolgt Initialisierung via Standardkonstruktor
  - ***intexpr*** kann jeder Ausdruck sein, der einen positiven **int** zurückliefert
- **new** reserviert benötigten Speicher konsekutiv und liefert die Adresse des ersten Elementes zurück: **(*intexpr* × sizeof(Type))** Bytes

Der **`delete`**-Operator

- Syntax, um Zeigervariablen freizugeben

```
// Freigabe einer einzigen Zeigervariable
delete zeiger;
// Freigabe eines Arrays
delete[] zeiger;
```

- Bei Freigabe einer Zeigervariable vom Typ einer Klasse wird deren Destruktor aufgerufen, sofern vorhanden
- Achtung: der Inhalt der Zeigers wird durch **`delete`** nicht verändert, aber der Speicher, auf den er verweist, ist danach „ungültig“, d.h. er kann wieder neu allokiert werden

Interne Abläufe bei **`new`** bzw. **`delete`**

- **`new`**
  - Speicher der Grösse **`sizeof(T)`** auf Heap suchen
  - Eventuell Heap nach oben grösser machen
  - Zeiger auf Objekt (= Speicherblock) zurückgeben

- **`delete`**
  - Nimmt Zeiger auf Objekt entgegen
  - Speicher freigeben
  - Eventuell Speicherblöcke zusammenlegen

Speicherknappheit

```
#include <new>
struct Foo { long a,b,c,d,e,f,g,h,i,j,k,l; };
void f()
{
   try {
      for (;;) new Foo;
   }
   catch (bad_alloc&) {
      cerr << "Speicher erschöpft!" << endl;
   }
}
```

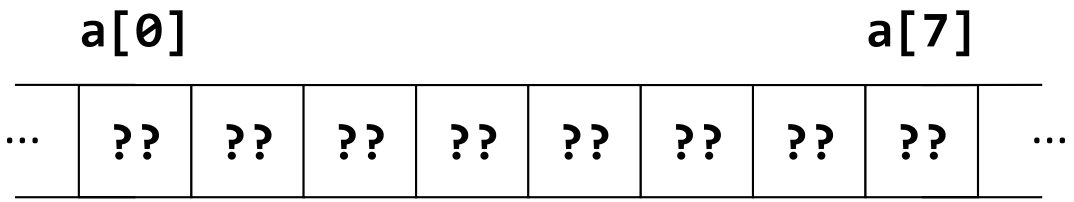
- Egal wieviel Speicher zur Verfügung steht, dieses Beispiel wird irgendwann den **`bad_alloc`** Handler aufrufen.

Definition eines Arrays (Feldes)

Beispiel:

```
int  a[8];          // 8 Integer
int  a[2*4];        // dito
```

Danach sieht **a** im Speicher so aus:



- Der Speicher ist also allokiert, aber nicht initialisiert.
- Im Allgemeinen erfolgt der Zugriff über nichtkonstante Integer-Werte, deren Werte der Compiler nicht kennt; z.B. `a[i]` wobei `i` eine beliebige Integer-Variable ist.

Statische Initialisierung von Arrays

Syntax:

```
int smallPrimes[7] = {2, 3, 5, 7, 11, 13, 17};

float rotmatrix[2][2] =  // zweidimensionales Array
{
    { cos(a), sin(a)},   // [0,0], [0,1]
    {-sin(a), cos(a)}    // [1,0], [1,1]
};


int vectr[100] = {1, 2}; // alle anderen Werte a[2-99]
                         // sind dann 0 initialisiert
```

Beziehung zw. Arrays und Zeigern

- Arrays gibt es eigentlich gar nicht in C/C++!
- Arrays werden mit konstanten Zeigern und Zeiger-Arithmetik implementiert.

Deklaration/Definition:

Was macht `int a[8]`?

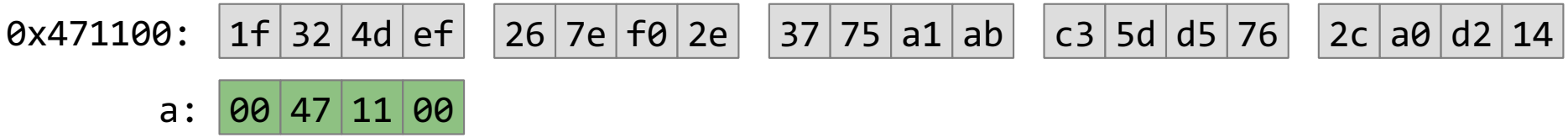
- Reserviert zusammenhängenden Speicherblock für 8 **`int`**-Zahlen.
- Deklariert `int* const a;`
  und initialisiert `a` mit Adresse des ersten Elements.
- Elementzugriff ist Zeiger-Arithmetik:
  Ausdruck `a[i]` ist äquivalent zu `*(a+i)`

Beispiel – Zeiger und Felder (i)

```
int* a = new int[5]; // Reserviere einen Speicherbereich, der
                     // 5 Integer aufnehmen kann.
```

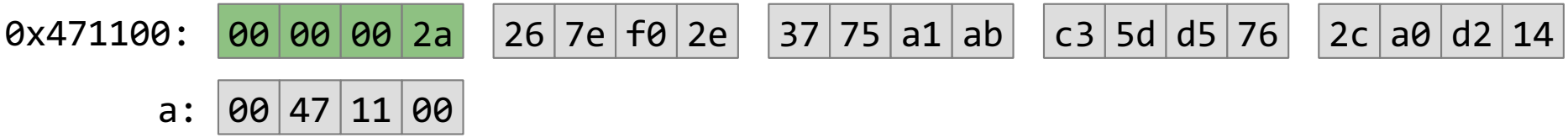
Beispiel – Zeiger und Felder (ii)

```
int* a = new int[5]; // Reserviere einen Speicherbereich, der
                     // 5 Integer aufnehmen kann. In a wird die
                     // Anfangsadresse dieses Bereichs gespeichert.
```



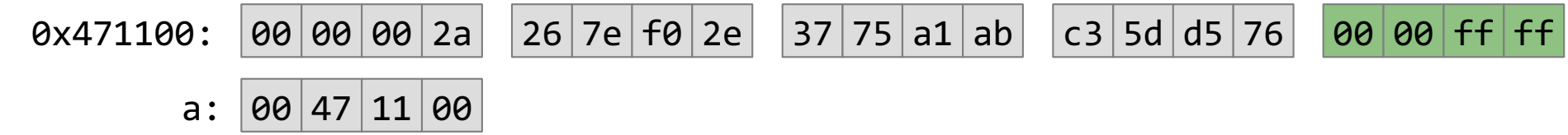
Beispiel – Zeiger und Felder (iii)

```
int* a = new int[5]; // Reserviere einen Speicherbereich, der
                     // 5 Integer aufnehmen kann. In a wird die
                     // Anfangsadresse dieses Bereichs gespeichert.

a[0] = 42;       // schreibe 42 in das erste Element
```



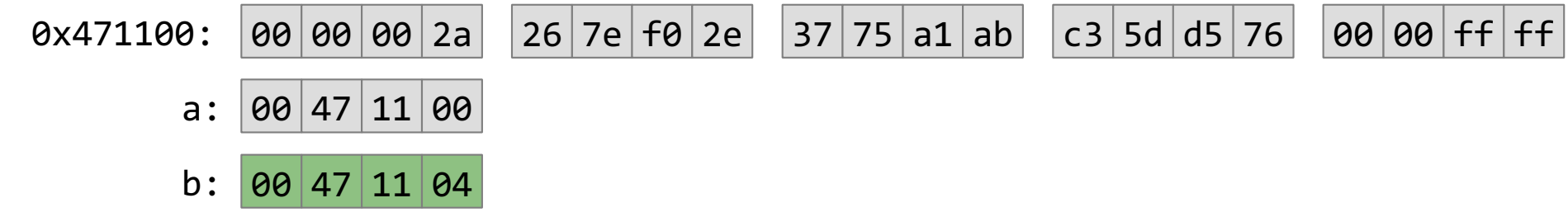
Beispiel – Zeiger und Felder (iv)

```
int* a = new int[5]; // Reserviere einen Speicherbereich, der
                     // 5 Integer aufnehmen kann. In a wird die
                     // Anfangsadresse dieses Bereichs gespeichert.

a[0] = 42;       // schreibe 42 in das erste Element

a[4] = 65535;    // schreibe 65535 in das letzte Element
```



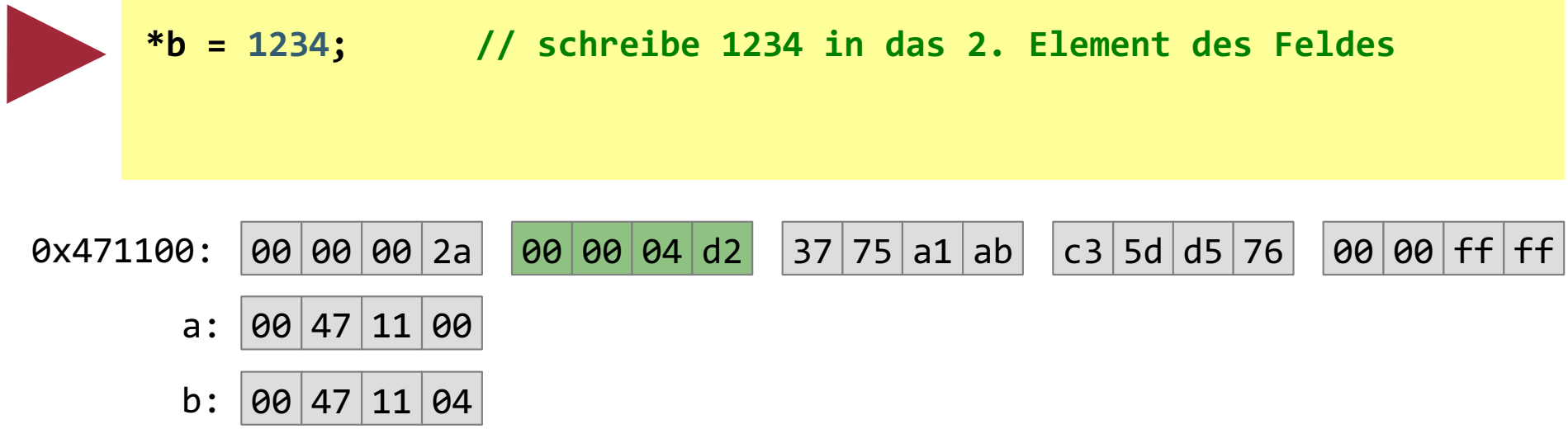
Beispiel – Zeiger und Felder (v)

```
int* a = new int[5]; // Reserviere einen Speicherbereich, der
                     // 5 Integer aufnehmen kann. In a wird die
                     // Anfangsadresse dieses Bereichs gespeichert.

a[0] = 42;       // schreibe 42 in das erste Element

a[4] = 65535;    // schreibe 65535 in das letzte Element

int* b = a + 1; // Rechenoperationen bei Zeigern berücksichtigen
                // deren Datentyp! b zeigt 4 Bytes hinter a
```



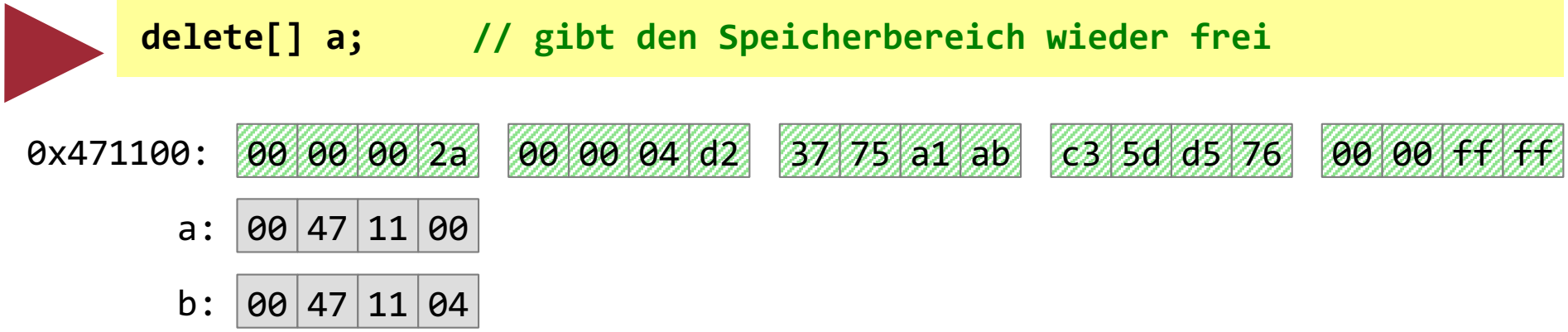
Beispiel – Zeiger und Felder (vi)

```
int* a = new int[5]; // Reserviere einen Speicherbereich, der
                     // 5 Integer aufnehmen kann. In a wird die
                     // Anfangsadresse dieses Bereichs gespeichert.

a[0] = 42;       // schreibe 42 in das erste Element

a[4] = 65535;    // schreibe 65535 in das letzte Element

int* b = a + 1; // Rechenoperationen bei Zeigern berücksichtigen
                // deren Datentyp! b zeigt 4 Bytes hinter a

*b = 1234;       // schreibe 1234 in das 2. Element des Feldes
```



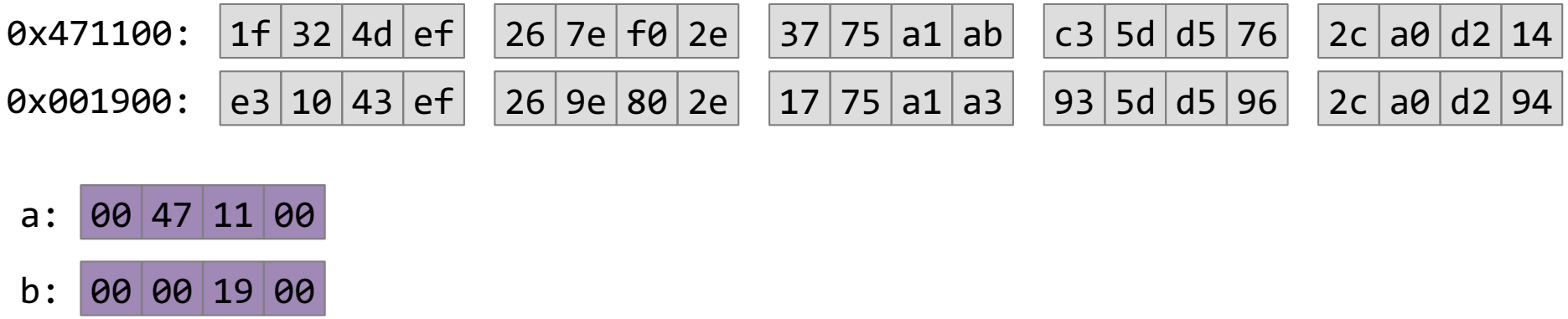
Beispiel – Zeiger und Felder (vii)

```
int* a = new int[5]; // Reserviere einen Speicherbereich, der
                     // 5 Integer aufnehmen kann. In a wird die
                     // Anfangsadresse dieses Bereichs gespeichert.

a[0] = 42;       // schreibe 42 in das erste Element

a[4] = 65535;    // schreibe 65535 in das letzte Element

int* b = a + 1; // Rechenoperationen bei Zeigern berücksichtigen
                // deren Datentyp! b zeigt 4 Bytes hinter a

*b = 1234;       // schreibe 1234 in das 2. Element des Feldes

delete[] a;      // gibt den Speicherbereich wieder frei
```



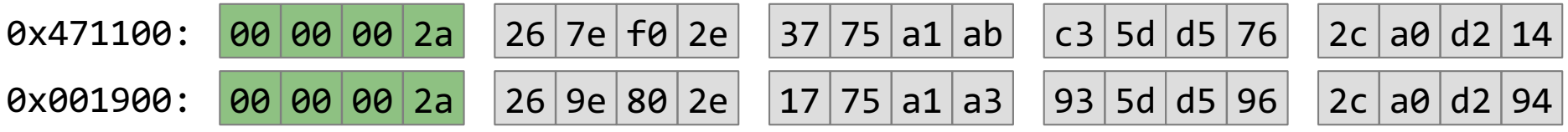
Beispiel – Zeiger und Felder (viii)

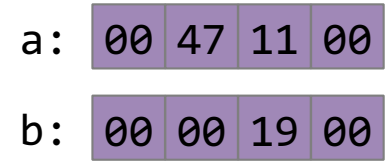
```
int* a = new int[5];    // Reserviere einen Speicherbereich, der
                        // 5 integers aufnehmen kann. In a wird die
                        // Anfangsadresse dieses Bereichs gespeichert
int b[5];          // Reserv. Speicherbereich für 5 int auf Stack.
                   // In b wird Anfangsadresse von diesem Bereich
                   // gespeichert.
```



Beispiel – Zeiger und Felder (ix)

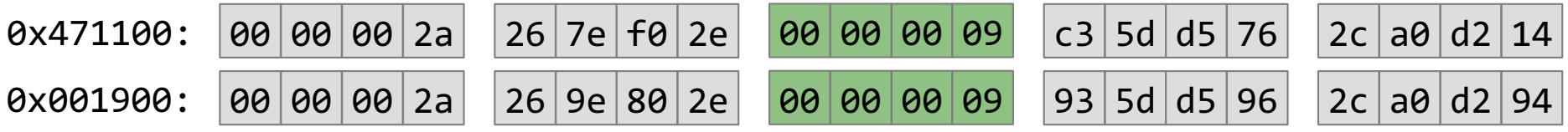
```
int* a = new int[5];    // Reserviere einen Speicherbereich, der
                        // 5 integers aufnehmen kann. In a wird die
                        // Anfangsadresse dieses Bereichs gespeichert
int b[5];          // Reserv. Speicherbereich für 5 int auf Stack.
                   // In b wird Anfangsadresse von diesem Bereich
                   // gespeichert.

a[0] = 42;         // schreibe 42 in das erste Element
b[0] = 42;         // schreibe 42 in das erste Element
```

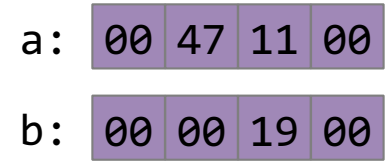




Beispiel – Zeiger und Felder (x)

```
int* a = new int[5];    // Reserviere einen Speicherbereich, der
                        // 5 integers aufnehmen kann. In a wird die
                        // Anfangsadresse dieses Bereichs gespeichert
int b[5];          // Reserv. Speicherbereich für 5 int auf Stack.
                   // In b wird Anfangsadresse von diesem Bereich
                   // gespeichert.

a[0] = 42;         // schreibe 42 in das erste Element
b[0] = 42;         // schreibe 42 in das erste Element

*(a+2) = 9;        // schreibe 9 in das dritte Element von a
*(b+2) = 9;        // schreibe 9 in das dritte Element von b
```





Zeiger & Felder

Noch ein Beispiel (syntax at its „best“ ...)

```
int a[4] = {0, 1, 2, 3};
int* q = a;       // Zuweisung von Pointern
int* p = & a[0]; // Anfangsadresse über Adress-Operator
                  // p,q zeigen jetzt auf dasselbe Element
*p = 100;         // a = {100, 1, 2, 3}
q[1] = *(a+2);    // a = {100, 2, 2, 3}
q += 2;           // q zeigt jetzt auf a[2]
q[0] = 300;       // a = {100, 2, 300, 3}
a = q;            // Kompilerfehler! int[4] versus int*
```

Zeiger & Felder: Ungesicherter Zugriff

- Feld wird in C/C++ nur durch einen Zeiger auf den Anfang des Speicherbereichs dargestellt.
- Der Zugriff auf Feldelemente wird **nicht** überprüft.

```
int  a[5] = { 1, 2, 3, 5, 7 };
int  b = 100;

a[0] = 42;  // Zugriff auf Feld Nr.1: legal
a[5] = 137; // Zugriff auf Feld Nr.6: illegal aber möglich
            // → Seiteneffekt: verändert eventuell b !!!

int* x = new int[9];
x[5] = 42;  // Zugriff auf Feld Nr.6: legal
x[9]= 137;  // Zugriff auf Feld Nr.10:
                        illegal → meist Absturz
```

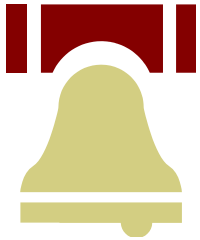
Warum erlaubt man unges. Zugriffe?

- Zugriff auf illegales Feld: es wird einfach in den entsprechenden Speicher geschrieben (bzw. von dort gelesen).
- Glücklicher Fall: Speicher gehört anderem Programm und das Betriebssystem meldet eine Zugriffsverletzung **`(Segmentation Fault)`**.
- Fataler Fall: Speicher gehört dem Programm selbst und die Änderung bewirkt unerklärliche Seiteneffekte.
- Ergo: der Programmierer/die Programmiererin muss sich selbst die Grösse des jeweiligen Speichers merken.

\+ Optimale Performance und Speicherplatznutzung.

\+ Auf dieser Basis können beliebig komfortablere, aber langsamere Felder (z.B. mit überprüften Zugriffen) entwickelt werden.

- Umgekehrt wäre dies nicht möglich!



Lebenszeit dyn. allokierter Objekte

- Allokation (und ggf. Initialisierung) von Objekten auf dem Heap mit **`new`**.
- Lebensende von dynamisch erzeugten Objekten: **`delete`**.
- Muss man **`delete`** aufrufen?
  - Allgemein: der gesamte Speicher wird vom Betriebssystem am Ende des Programm freigegeben.
  - **Ja**: da sonst **Memory-Leaks** entstehen können.

Faustregel: Zu jedem `new` ein `delete`.

Was ist/wie entsteht ein Memory Leak?

- Wenn z.B. Programme dynamischen Speicher anlegen und später vergessen, ihn wieder freizugeben, dann entstehen Memory Leaks.

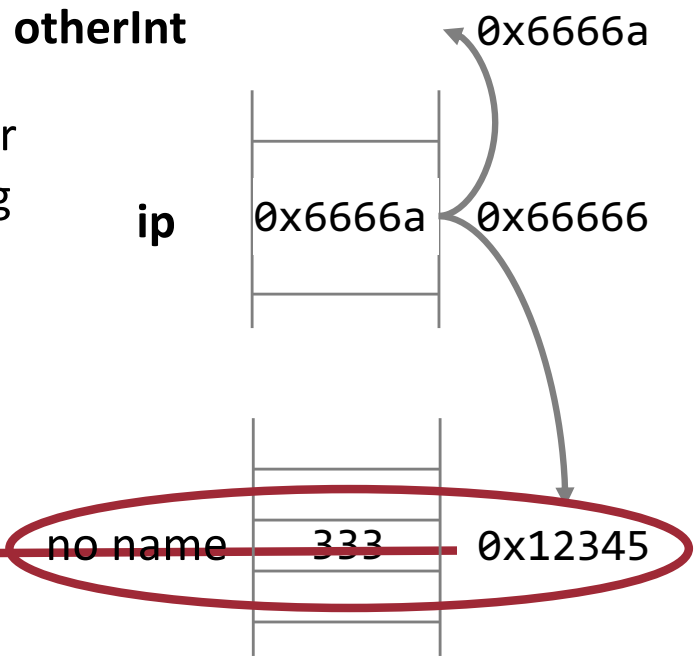
```
void doSomething (float value, int ntimes) {

  float* array = new float[ntimes];
  for (int i=0; i<ntimes; ++i) {
    *(array + i) = value * float(i);  // entweder so ..
    *(array++)   = value * float(i);  // .. oder so ...
  }
  return;  // Achtung: array wird nicht wieder freigegeben,
           // der angelegte Speicher ist „verloren“
}
```

- C++ besitzt keinen automatischen Garbage-Collector, der die Freigabe des nicht mehr benötigten Speichers übernimmt.

Speicherbug – Overwrite Pointer

```
int* ip      = new int(333);
. . .
delete ip;

ip = &otherInt;
```

Allokiere Speicher
& Wertzuweisung



Zeiger **ip** zeigt auf neue Adresse
(zeigt auf neuen integer Wert)

Speicherleck!
Speicher kann nicht mehr erreicht werden

Speicherbug – Buffer Overflow

- Buffer Overflow (Puffer-Überlauf)
  - Ursache: man schreibt über Grenzen eines allokierten Blocks hinaus.
  - Mögliche Folgen (oft schlecht reproduzierbar):
    - Sporadische Core-Dumps (segmentation fault)
    - Falsche Werte in anderen Variablen (z.B. `NaN`, `1.xxE38`, ...)
    - Return aus Funktion „killt“ das Programm

Beispiel:

```
float*  a = new float[10];
int*    b = new int[10];
for (int i=0; i < 20; i++)
   a[i]  = 1.0;
```

Speicherbug – Dangling Pointer

- Dangling Pointer („Hängende Zeiger“)
  - Ursache: Zeiger wird verwendet, nachdem er, und damit der zugehörige Speicher, freigegeben wurde.
  - Mögliche Folgen:
    - Falsche Werte in anderen dynamischen Variablen.

```
struct MyStruct {
    int m;
};
MyStruct* s1 = new MyStruct;
s1->m = 17;
delete s1;
// tue etwas, z.B. Allokation neuer Variable(n) mit new
cout << s1->m;
```

```
Ausgabe: 42
// oder auch etwas anderes
```

Speicherbug – Double Delete

- Double Delete
  - Spezieller Fall von Dangling Pointer
  - Ursache: Speicher wird zweimal freigegeben
  - Folge:
    - Löscht evtl. Speicher, der einem anderen Programmteil gehört
    - Führt evtl. zum Absturz

Beispiel

```
struct MyStruct {
   int m;
};
MyStruct* s1 = new MyStruct;
s1->m = 17;
delete s1;
// tue etwas, z.B. Allokation neuer Variable(n) mit new
delete s1;  // erneutes Freigeben von s1
```

Noch mehr Fallstricke ...

Wo bzw. warum stürzt diese Funktion ab?

```
int someFunction (int a, int b) {
  int* i;

  if ((a % b) == 0) {
    /* ... */
    i = new int;
  } else if ((b % a) == 0) {
    /* ... */
    i = new int;
  }
  // i möglicherweise uninitialisiert! (z.B. a=3, b=2)
  *i = (a > b) ? a : b;
  return a*b*(*i);
}
```

Lösungen für diese Fallstricke

- Tools: Verwendung von speziellen „Memory Checkern“ (z.B. valgrind, purify, …)
- Bibliotheken: Smart Pointer oder Garbage Collection

- Eine etwas armselige Lösung für Double Delete:

```
MyStruct* s1 = new MyStruct;
s1->m = 17;
delete s1;
s1 = NULL;  // s1 ist nun ein NULL pointer
. . .
delete s1;  // double delete, tut nichts
```

Radikale Lösung für „Zeigerproblematik“

- Programmiersprachen ohne Zeiger (wie z.B. Java, Python)
  - Variablen mit nichtelementaren Typen sind implementiert als Zeiger
  - Programmierer sieht die Zeiger nicht (kein ***- und **&**-Operator vorhanden)
  - Laufzeitumgebung (Interpreter/Virtual Machine) erkennt, wenn ein Objekt vom Programm nicht mehr benutzt wird (kein Zeiger zeigt mehr auf Objekt)
  - Garbage-Collector (GC) läuft im Hintergrund ständig mit
  - Für nichtzeitkritische Applikationen eine sehr gute und in der Praxis bewährte Lösung
    - Es werden keine Garantien zum GC gegeben, d.h. er kann für beliebig lange Unterbrechungen sorgen (wenn er aufräumt); zumindest verursacht er zusätzliche Last im System.
  - Diese Sprachen werden auch verwaltete Sprachen genannt, weil der gesamte Speicher automatisch verwaltet wird.

Borrow Checking in *Rust*

- Sichere Speicherverwaltung ohne Garbage Collection, durch statische Analyse zur Compilezeit.
- Grundlegende Idee: Einführung der Konzepte
  - Besitz (*ownership*) eines Objektes im Speicher.
  - Ausleihen (*borrowing*) des Besitzes eines Objektes im Speicher entweder an genau eine veränderliche (*mutable*) Referenzvariable oder beliebig viele unveränderliche (*immutable*) Referenzvariablen.
    → Compiler kann dies nachverfolgen.
- Darüber hinaus:
  - **`null`** als Wert für Referenzen nicht erlaubt.
  - Referenzen können die Lebensdauer des referenzierten Objektes im Speicher nicht „überleben“ (schliesst Dangling Pointer aus).
  - Unterscheidung zwischen sicheren und unsicheren Codeabschnitten. Letztere erlauben praktisch dieselben Speicherprobleme wie C/C++.

Mehrdimensionale Felder (i)

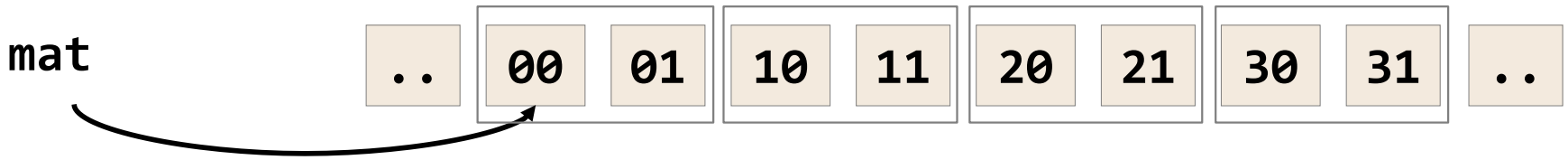
- Grundsätzlich ist Speicher immer linear
  - n-dimensionale Arrays werden durch geeignete Indizierung „simuliert“
- C++ bietet eingebaute n-dimensionale Felder fester Länge:

```
int mat[4][2];
for (int j=0; j<2; ++j) {
  for (int i=0; i<4; ++i) {
    mat[i][j] = 10*i + j;
  }
}
```

bzw.

```
int mat[4][2];
for (int i=0; i<4; ++i) {
  for (int j=0; j<2; ++j) {
    mat[i][j] = 10*i + j;
  }
}
```

- **mat** ist ein Zeiger auf **int**:



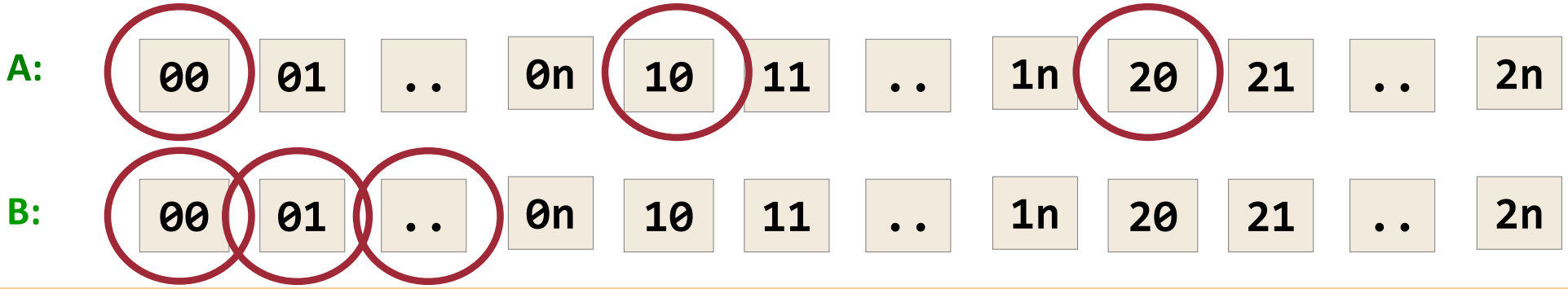
Mehrdimensionale Felder (ii)

Welche Initialisierung ist schneller?

```
// Version A
int mat[4][n];
for (int j=0; j<n; ++j) {
  for (int i=0; i<4; ++i) {
    mat[i][j] = 10*i + j;
  }
}
```

```
// Version B
int mat[4][n];
for (int i=0; i<4; ++i) {
  for (int j=0; j<n; ++j) {
    mat[i][j] = 10*i + j;
  }
}
```

- Zeilenweise Ordnung: Daten liegen dicht bezüglich der Zeilen **j**:



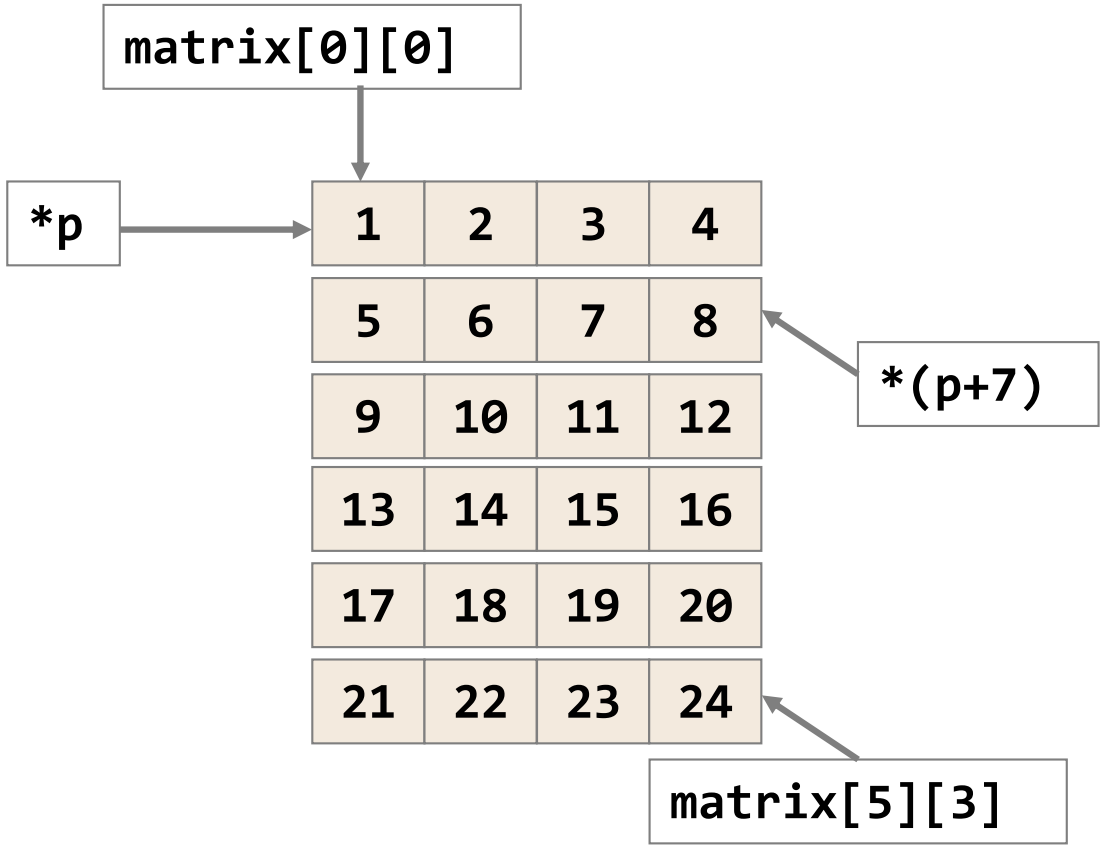
Mehrdimensionale Felder (iii)

- Version B nutzt aufeinander folgende Daten im Speicher
- Version B ist daher deutlich schneller* als Version A
- C++ Compiler macht die Optimierung automatisch (nur möglich für einfache Beispiele)

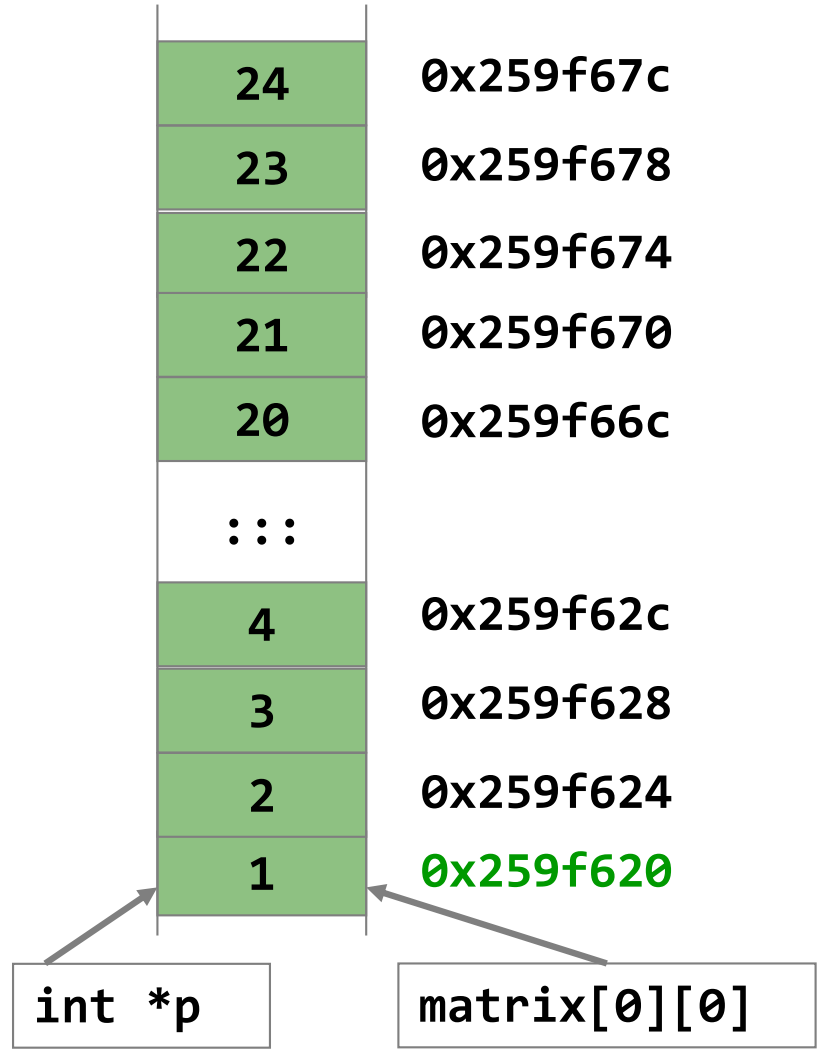
* Je nach Hardware. Auf einem Testsystem wurde z.B. ca. Faktor 20 gemessen.

2D-Arrays (Matrizen)

```
int matrix[6][4];
// ... Fill with 1 .. 24
int* p = &matrix[0][0];
```



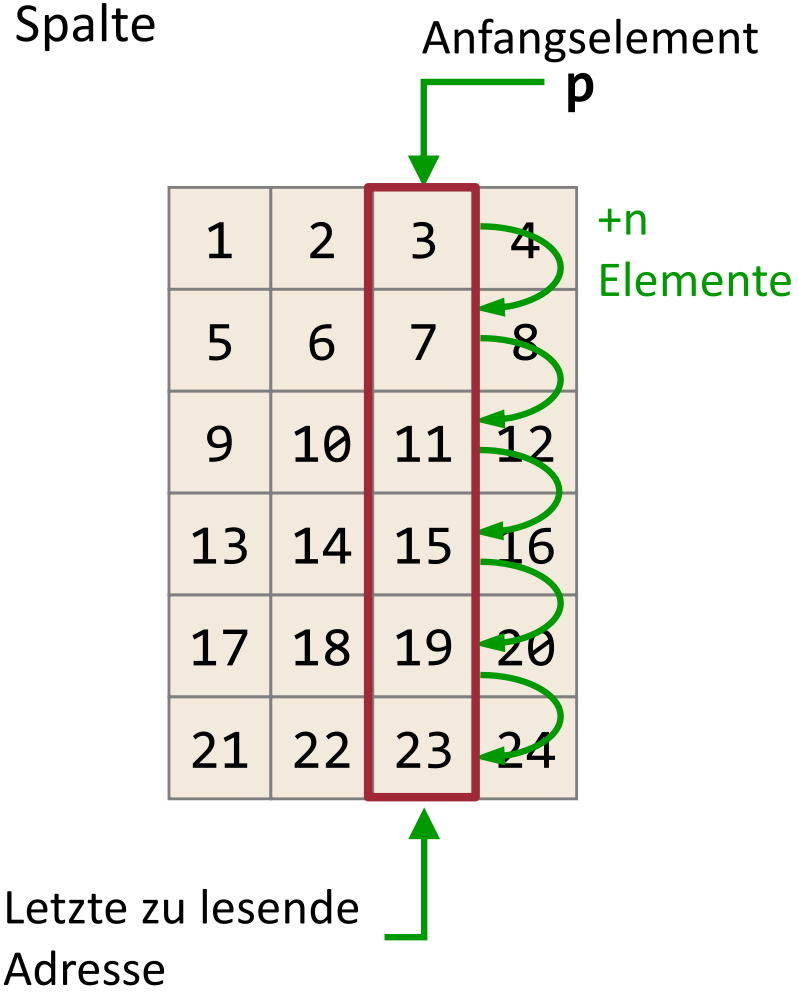
Repräsentation im Speicher



2D-Arrays: Zeiger-Arithmetik

- Gegeben: m×n Matrix wobei im Bild m=6, n=4
- Aufgabe: Drucke die Werte der vorletzten Spalte

```
for ( int* p =  &matrix[0][n-2];
          p <= &matrix[m-1][n-2];
          p += n
    )
   cout << *p << " ";
cout<<endl;
```



Alternative: Arrays von Arrays (i)

- Man kann mehrdimensionale Arrays wie eben beschrieben erzeugen (als eindimensionaler Vektor), oder als **n** einzelne Vektoren!
- **n** einzelne Vektoren:

```
int* array_of_array[6]; // Array von Zeigern auf int*
for (int i = 0; i < 6; i ++)
   array_of_array[i] = new int[4];
```

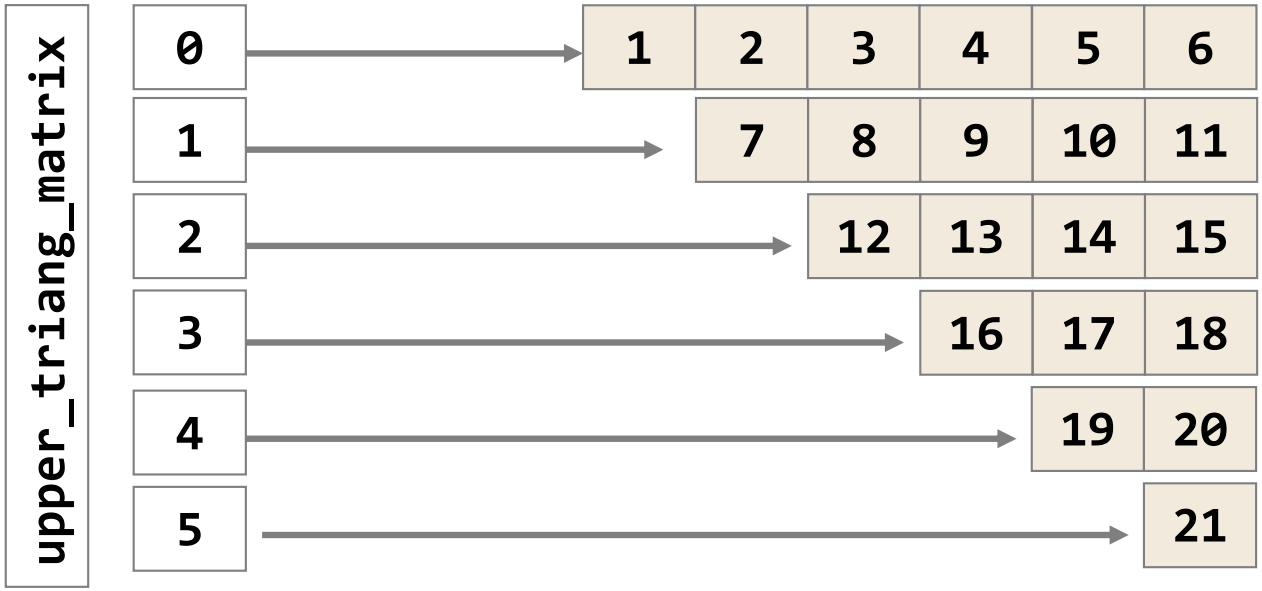
- Der Zugriff erfolgt wie bereits bekannt:

```
int k = array_of_array[i][j];
int k = *(array_of_array[i] + j);
int k = *(  *(array_of_array + i)  + j);
```

Alternative: Arrays von Arrays (ii)

- Vorteil: Man kann auch nichtrechteckige, mehrdimensionale Arrays erzeugen:

```
int* upper_triang_matrix[6]; // Array von Zeigern auf int
for (int i = 0; i < 6; i ++)
   upper_triang_matrix[i] = new int[6-i];
```



Nachteil – durch interne Organisation

- Rechteckige zweidimensionale Matrix **m[6][4]** :

```
float* m = new float[6*4];
for (int i=0; i<6; i++)
  for (int j=0; j<4; j++)
    m[ i*4 + j] = 10*i + j;
    // *(m + 4*i + j) Zeiger-Arithmetik
```

- Arrays von Arrays

```
float* m[6];
for (int i=0; i<6; i++) m[i] = new float[4];
x = m[i][j];           //  ist äquivalent zu
x = *( *(m+i) + j);    //  Zeiger-Arithmetik
```

- Fazit: Verwendung von Array von Arrays ist im Allgemeinen langsamer
  - Zugriff auf Speicher ist langsamer als Rechenoperationen
  - Cache Misses

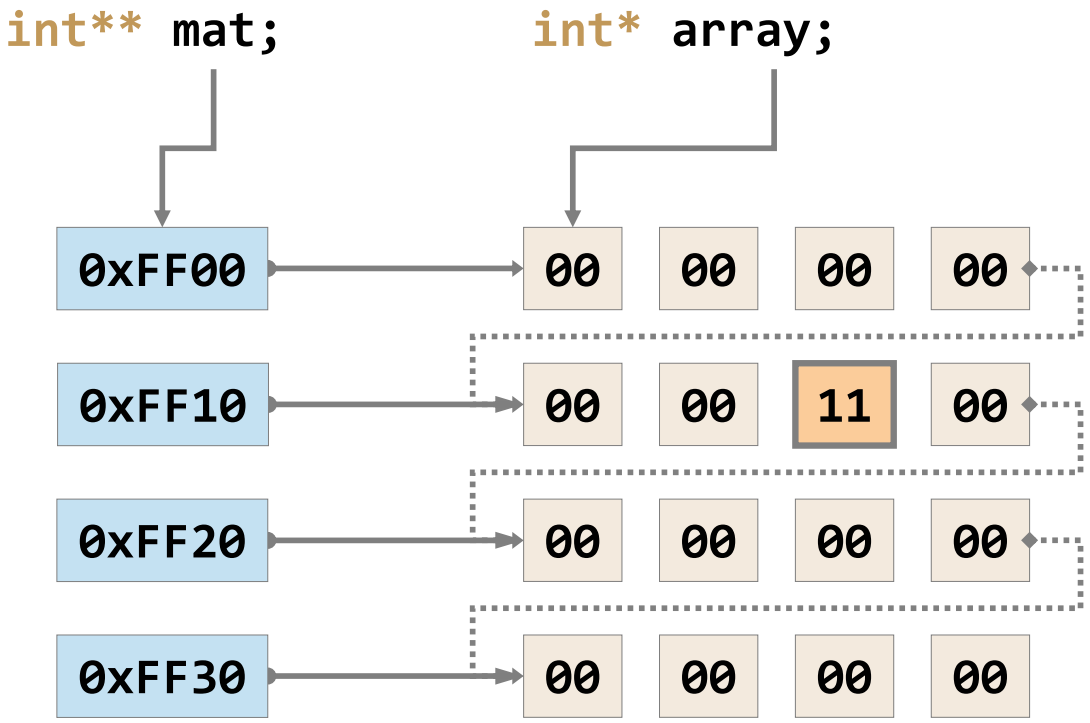
K04

Zeiger & Co

1. Zeiger
   - Verwendung und Zeigerarithmetik
2. Referenzen
3. Arrays
4. **Zeigertabellen**
5. Funktionszeiger

N-dimensionale Arrays: Alternative

- Verwendung von Zeigertabellen (Zeiger auf Zeiger) anstelle von unschönen Indexberechnungen




```
int*  array = new int[16];
int** mat = new (int*)[4];

mat[0] = &array[0];
mat[1] = &array[4];
/* ... */

// Zugriff, vorher
array[1*4+2] = 11;

// Zugriff, jetzt
mat[1][2] = 11;
```

K04

Zeiger & Co

1. Zeiger
   - Verwendung und Zeigerarithmetik
2. Referenzen
3. Arrays
4. Zeigertabellen
5. **Funktionszeiger**

Funktionszeiger (i)

Was kann man alles mit Funktionen tun?

- Man kann sie aufrufen.
- Man kann ihre Adresse – der Einstiegspunkt im Code-Segment – ermitteln und den Zeiger später benutzen, um die Funktion aufzurufen
  - Braucht man um **Funktionen höherer Ordnung** realisieren zu können → Funktionale Programmierung
  - Callback-Funktionen: z.B. prozessspezifische Ereignisbehandlung in Betriebssystemen.

- Es gilt: **`sizeof(fp) == sizeof(void*)`** wobei **`fp`** ein Funktionszeiger ist. Ist **`fp`** jedoch ein Methodenzeiger (d.h. auf Funktion einer Klasse), dann gilt dies im Allgemeinen nicht mehr, da virtuelle Methoden korrekt behandelt werden müssen (zu virtuellen Methoden später mehr).

Funktionszeiger (ii)

Beispiel:

```
// zwei Funktionen mit bis auf den Namen identischer Signatur
int plus(int x, int y) { return x+y; }
int mult(int x, int y) { return x*y; }

int (*fp) (int, int);  /* Zeiger auf Funktionen die
Rückgabetyp int haben und zwei Parameter vom Typ int */

typedef int (*FP)(int, int);  // dasselbe mit Alias

FP FPT_PLUS = &plus; // Dekl. & Zuweisung der Addr. von plus
cout << plus(2,3) << "," << FPT_PLUS(4,5) << endl;
```

```
5,9
```