Felder (Arrays) und Zeiger (Pointers) - Teil I

Feldtypen, Sieb des Eratosthenes, Iteration, Zeigertypen, Zeigerarithmetik, dynamische Speicherverwaltung

Felder: Motivation

- Wir können jetzt über Zahlen iterieren:
  `for (int i=0; i<n; ++i) {...}`
- Oft muss man aber über *Daten* iterieren (Beispiel: finde ein Kino in Zürich, das heute "*The C++ Legacy*" zeigt)
- Felder dienen zum Speichern von Folgen *gleichartiger* Daten (Beispiel: Spielpläne aller Zürcher Kinos)

Felder: erste Anwendung

Das Sieb des Eratosthenes

- berechnet alle Primzahlen < *n*
- Methode: Ausstreichen der Nicht-Primzahlen

2	3		5		7		9		11		13		15		17	

Streiche alle echten Vielfachen von 2...

Felder: erste Anwendung

Das Sieb des Eratosthenes

- berechnet alle Primzahlen < *n*
- Methode: Ausstreichen der Nicht-Primzahlen

2	3		5		7		9		11		13		15		17	

...und gehe zur nächsten Zahl

Felder: erste Anwendung

Das Sieb des Eratosthenes

- berechnet alle Primzahlen < *n*
- Methode: Ausstreichen der Nicht-Primzahlen

2	3		5		7				11		13				17	

Streiche alle echten Vielfachen von 3...

Felder: erste Anwendung

Das Sieb des Eratosthenes

- berechnet alle Primzahlen < *n*
- Methode: Ausstreichen der Nicht-Primzahlen

2	3		5		7				11		13				17	

...und gehe zur nächsten Zahl

Felder: erste Anwendung

Das Sieb des Eratosthenes

- berechnet alle Primzahlen < *n*
- Methode: Ausstreichen der Nicht-Primzahlen

2	3		5		7				11		13				17	

Am Ende des Streichungsprozesses bleiben genau die Primzahlen übrig!

Felder: erste Anwendung

Das Sieb des Eratosthenes

- berechnet alle Primzahlen < *n*
- Methode: Ausstreichen der Nicht-Primzahlen
- Frage: wie streichen wir Zahlen aus???

mit einem Feld!

Felder: Implementierung "Sieb des Eratosthenes"

```
int main()
{
  const unsigned int n = 1000;

  // definition and initialization: provides us with
  // Booleans crossed_out[0],..., crossed_out[n-1]
  bool crossed_out[n];
  for (unsigned int i = 0; i < n; ++i)
    crossed_out[i] = false;

  // computation and output
  std::cout << "Prime numbers in {2,..., " << n-1 << "}:\n";
  for (unsigned int i = 2; i < n; ++i)
    if (!crossed_out[i]) {
      // i is prime
      std::cout << i << " ";
      // cross out all proper multiples of i
      for (unsigned int m = 2*i; m < n; m += i)
        crossed_out[m] = true;
    }
  std::cout << "\n";

  return 0;
}
```

Berechnet alle Primzahlen < 1000

n ist eine Konstante, sollte also auch so deklariert werden!

Felder: Implementierung "Sieb des Eratosthenes"

```
int main()
{
  const unsigned int n = 1000;

  // definition and initialization: provides us with
  // Booleans crossed_out[0],..., crossed_out[n-1]
  bool crossed_out[n];
  for (unsigned int i = 0; i < n; ++i)
    crossed_out[i] = false;

  // computation and output
  std::cout << "Prime numbers in {2,..., " << n-1 << "}:\n";
  for (unsigned int i = 2; i < n; ++i)
    if (!crossed_out[i]) {
      // i is prime
      std::cout << i << " ";
      // cross out all proper multiples of i
      for (unsigned int m = 2*i; m < n; m += i)
        crossed_out[m] = true;
    }
  std::cout << "\n";

  return 0;
}
```

Feld: `crossed_out[i]` gibt an, ob i schon ausgestrichen wurde

Felder: Implementierung "Sieb des Eratosthenes"

```
int main()
{
  const unsigned int n = 1000;

  // definition and initialization: provides us with
  // Booleans crossed_out[0],..., crossed_out[n-1]
  bool crossed_out[n];
  for (unsigned int i = 0; i < n; ++i)
    crossed_out[i] = false;

  // computation and output
  std::cout << "Prime numbers in {2,..., " << n-1 << "}:\n";
  for (unsigned int i = 2; i < n; ++i)
    if (!crossed_out[i]) {
      // i is prime
      std::cout << i << " ";
      // cross out all proper multiples of i
      for (unsigned int m = 2*i; m < n; m += i)
        crossed_out[m] = true;
    }
  std::cout << "\n";

  return 0;
}
```

Das Sieb: gehe zur jeweils nächsten nichtgestrichenen Zahl `i` (diese ist Primzahl), gib sie aus und streiche alle echten Vielfachen von `i` aus

Felder: Definition

Deklaration einer *Feldvariablen* (array):

T a [*expr*]

Wert *k* ist bei Kompilierung bekannt (z.B. Literal, Konstante)

konstanter ganzzahliger Ausdruck; Wert gibt Länge des Feldes an

Variable des Feld-Typs

zugrundeliegender Typ

Typ von *a* : "*T* [*k*]"

Wertebereich von *T* [*k*]: T^k

Felder: Definition

Deklaration einer *Feldvariablen* (array):

T a [expr]

Wert *k* ist bei Kompilierung bekannt (z.B. Literal, Konstante)

konstanter ganzzahliger Ausdruck; Wert gibt Länge des Feldes an

Variable des Feld-Typs

zugrundeliegender Typ

Beispiel:

```
bool crossed_out[n]
```

Felder variabler Länge?

Praktischer (aber nicht erlaubt) wäre:

```
int main()
{
  // input of n
  std::cout << "Compute prime numbers in [2, n) for n = ?";
  unsigned int n;
  std::cin >> n;

  // definition and initialization: provides us with
  // Booleans crossed_out[0],..., crossed_out[n-1]
  bool crossed_out[n]; // Fehler!
  ...
}
```

kein konstanter Ausdruck!

Felder variabler Länge?

Praktischer (und erlaubt) wäre:

```
int main()
{
  // input of n
  std::cout << "Compute prime numbers in [2, n) for n = ?";
  unsigned int n;
  std::cin >> n;

  // definition and initialization: provides us with
  // Booleans crossed_out[0],..., crossed_out[n-1]
  bool* const crossed_out = new bool[n]; // ok!
  ...
}
```

so geht es (Erklärung folgt)!

Feld-Initialisierung

- `int a[5];`

Die 5 Elemente von `a` bleiben uninitialisiert (können später Werte zugewiesen bekommen)

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

Die 5 Elemente von `a` werden mit einer *Initialisierungsliste* initialisiert

auch ok; Länge wird deduziert

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

Speicherlayout eines Feldes

- Ein Feld belegt einen *zusammen-hängenden* Speicherbereich

Beispiel: Feld mit 4 Elementen

Speicherzellen für jeweils einen Wert vom Typ *T*

Wahlfreier Zugriff (*Random Access)*

Der L-Wert

a [*expr*]

Wert *i*

hat Typ T und bezieht sich auf das *i* –te Element des Feldes *a* (Zählung ab 0)

a[0] a[1] a[2] a[3]

Wahlfreier Zugriff (*Random Access*)

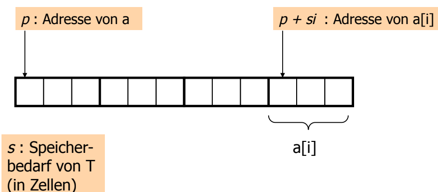
a [*expr*]

Der Wert *i* von expr heisst *Feldindex*

[]: Subskript-Operator

Wahlfreier Zugriff (*Random Access*)

- Wahlfreier Zugriff ist sehr effizient (Reduktion auf Adressarithmetik):



Felder sind primitiv

- Man kann Felder nicht wie bei anderen Typen initialisieren und zuweisen:

```
int a[5] = {4,3,5,2,1};
int b[5];
b = a;         // Fehler!
int c[5] = a; // Fehler!
```

Warum?

Felder sind primitiv

- Felder sind "Erblast" der Sprache C und aus heutiger Sicht primitiv
- In C, Felder sind sehr maschinennah und effizient, bieten aber keinen Luxus wie eingebautes Initialisieren und Kopieren
- Die Standard-Bibliothek bietet komfor-table Alternativen (mehr dazu später)

Felder als Daten-Container

Container:

- Objekt, das andere Objekte speichern kann...
- ...und die Möglichkeit anbietet, über die gespeicherten Objekte zu *iterieren* (Kinoprogramme...)

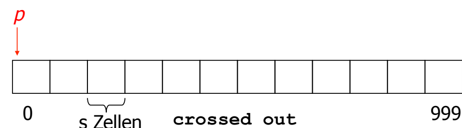
Iteration in Feldern geht über wahlfreien Zugriff: a[0], a[1],...,a[n-1]

Iteration durch wahlfreien Zugriff

```
for (unsigned int i = 0; i < 1000; ++i)
    crossed_out[i] = false;
```

Berechnungsaufwand:

p



Iteration durch wahlfreien Zugriff

```
for (unsigned int i = 0; i < 1000; ++i)
    crossed_out[i] = false;
```

Berechnungsaufwand (Adressberechnung):

p+s

0 s Zellen `crossed_out` 999

Iteration durch wahlfreien Zugriff

```
for (unsigned int i = 0; i < 1000; ++i)
    crossed_out[i] = false;
```

Berechnungsaufwand (Adressberechnung):

p+2s

0 s Zellen `crossed_out` 999

Iteration durch wahlfreien Zugriff

```
for (unsigned int i = 0; i < 1000; ++i)
    crossed_out[i] = false;
```

Berechnungsaufwand (Adressberechnung):

p+3s

0 s Zellen `crossed_out` 999

Iteration durch wahlfreien Zugriff

```
for (unsigned int i = 0; i < 1000; ++i)
    crossed_out[i] = false;
```

Berechnungsaufwand (Adressberechnung):

p+999s

0 s Zellen `crossed_out` 999

Iteration durch wahlfreien Zugriff

```
for (unsigned int i = 0; i < 1000; ++i)
    crossed_out[i] = false;
```

Berechnungsaufwand (Adressberechnung):

Pro Feldelement eine Addition und eine Multiplikation

p + i s

0 s Zellen `crossed_out` 999

Effizientere und natürlichere Iteration

Addieren von *s: gehe zum nächsten Element*

p *p+s* *p+2s* *p+999s*

0 s Zellen `crossed_out` 999

Effizientere und natürlichere Iteration

Berechnungsaufwand (Adressberechnung):

Pro Feldelement eine Addition $p_{vorher} + s$

0 s Zellen `crossed_out` 999

Effizientere und natürlichere Iteration durch *Zeiger*

```
bool* const begin = crossed_out;     // ptr to first element
bool* const end = crossed_out + n;   // ptr after last element
// in the loop, pointer p successively points to all elements
for (bool* p = begin; p != end; ++p)
  *p = false; // *p is the element pointed to by p
```

Pro Feldelement eine Addition $p_{vorher} + s$

0 s Zellen `crossed_out` 999

Effizientere und natürlichere Iteration durch *Zeiger*

```
bool* const begin = crossed_out;     // ptr to first element
bool* const end = crossed_out + n;   // ptr after last element
// in the loop, pointer p successively points to all elements
for (bool* p = begin; p != end; ++p)
  *p = false; // *p is the element pointed to by p
```

wird noch genau erklärt!

Pro Feldelement eine Addition $p_{vorher} + s$

0 s Zellen `crossed_out` 999

Buchlesen: Wahlfreier Zugriff vs. natürliche Iteration

Wahlfreier Zugriff:
- öffne Buch auf S.1
- klappe Buch zu
- öffne Buch auf S.2-3
- klappe Buch zu
- öffne Buch auf S.4-5
- ...

Natürliche Iteration:
- öffne Buch auf S.1
- blättere um
- blättere um
- blättere um
- blättere um
- ...

Zeiger

- erlauben das Repräsentieren von und das Rechnen mit *Adressen*
- unterstützen insbesondere die Operation "*gehe zum nächsten Element eines Feldes*"
- sind mit Vorsicht zu verwenden (beim *Ver*rechnen mit Adressen stürzt meist das Programm ab)

Zeiger-Typen

T *

sprich: "Zeiger auf *T* "

zugrundeliegender Typ

- *T* * hat als mögliche Werte Adressen von Objekten des Typs *T*
- Ausdrücke vom Typ *T* * heissen *Zeiger*

Zeiger: Visualisierung

p

Zeiger auf das Objekt (Wert von *p* ist die Adresse des Objekts)

Objekt im Speicher

Adressoperator

- liefert einen Zeiger (R-Wert) auf ein beliebiges Objekt, gegeben durch einen L-Wert

& L-Wert

```
int i = 5;
int* iptr = &i;
```

iptr

&

i

Dereferenzierungsoperator

- liefert einen L-Wert für ein Objekt, gegeben durch einen Zeiger auf das Objekt

** R-Wert*

```
int i = 5;
int* iptr = &i;
int j = *iptr; // j = 5
```

iptr

*

i

Dereferenzierungsoperator = Adressoperator $^{-1}$

Zeiger (R-Wert)

&

*

Objekt (L-Wert)

Felder "sind" Zeiger

- Jedes Feld vom Typ *T* [*k*] ist in den Typ *T** konvertierbar

Feld-nach-Zeiger-Konversion

- Ergebnis der Konversion ist ein Zeiger auf das erste Element (Feldindex 0)
- Tritt ein Feld in einem Ausdruck auf, so wird es automatisch konvertiert

Im Rechner passiert dabei nichts: ein Feld ist ohnehin nur durch die Adresse des ersten Elements repräsentiert.

Felder "sind" Zeiger

Beispiel:

```
int a[5];
int* begin = a;
```

begin

a

Zeiger-Arithmetik

- Zeiger `{+, -}` ganze Zahl
- Zeiger `{==, !=, <, >, <=, >=}` Zeiger
- Zeiger - Zeiger

p : Zeiger auf ein Feldelement

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]

Zeiger-Arithmetik

- Zeiger `{+, -}` ganze Zahl
- Zeiger `{==, !=, <, >, <=, >=}` Zeiger
- Zeiger - Zeiger

...oder auf Zelle direkt hinter dem Feld (*past-the-end* Zeiger) *p*

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]

Zeiger-Arithmetik

- Zeiger `{+, -}` ganze Zahl
- Zeiger `{==, !=, <, >, <=, >=}` Zeiger
- Zeiger - Zeiger

$p-2$ $p-1$ p $p+1$ $p+2$ $p+3$ $p+4$

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]

Zeiger-Arithmetik

- Zeiger `{+, -}` ganze Zahl
- Zeiger `{==, !=, <, >, <=, >=}` Zeiger
- Zeiger - Zeiger

Addenden mit Ergebniszeiger außerhalb dieses Bereichs sind illegal!

$p-2$ $p-1$ p $p+1$ $p+2$ $p+3$ $p+4$

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]

Zeiger-Arithmetik

- Zeiger `{+, -}` ganze Zahl
- Zeiger `{==, !=, <, >, <=, >=}` Zeiger
- Zeiger - Zeiger

`+= , -= , ++ , --` gibt es auch, mit der üblichen Bedeutung

p

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]

Zeiger-Arithmetik

- Zeiger `{+, -}` ganze Zahl
- Zeiger `{==, !=, <, >, <=, >=}` Zeiger
- Zeiger - Zeiger

`!=` : nicht `==`
`<=` : `<` oder `==`
`>` : nicht `<=`
`>=` : nicht `<`

q < p == p' < r

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]

Zeiger-Arithmetik

- Zeiger `{+, -}` ganze Zahl
- Zeiger `{==, !=, <, >, <=, >=}` Zeiger
- Zeiger – Zeiger

Zeiger-Arithmetik

- Zeiger `{+, -}` ganze Zahl
- Zeiger `{==, !=, <, >, <=, >=}` Zeiger
- Zeiger – Zeiger

Zeiger-Arithmetik

- Zeiger `{+, -}` ganze Zahl
- Zeiger `{==, !=, <, >, <=, >=}` Zeiger
- Zeiger – Zeiger

Die Wahrheit über den Subskript-Operator

- arbeitet eigentlich auf Zeigern:

a [*expr*]

ist eine Abkürzung für

* (*a* + *expr*)

Iteration durch Zeiger

```
bool* const begin = crossed_out;         // ptr to first element
bool* const end = crossed_out + n;       // ptr after last element
// in the loop, pointer p successively points to all elements
for (bool* p = begin; p != end; ++p)
  *p = false; // *p is the element pointed to by p
```


Iteration durch Zeiger

```
bool* const begin = crossed_out;         // ptr to first element
bool* const end = crossed_out + n;       // ptr after last element
// in the loop, pointer p successively points to all elements
for (bool* p = begin; p != end; ++p)
  *p = false; // *p is the element pointed to by p
```

Iteration durch Zeiger

begin ist eine Konstante!

```
bool* const begin = crossed_out;      // ptr to first element
bool* const end = crossed_out + n;    // ptr after last element
// in the loop, pointer p successively points to all elements
for (bool* p = begin; p != end; ++p)
  *p = false; // *p is the element pointed to by p
```


Iteration durch Zeiger

`const bool* begin` wäre ein variabler Zeiger auf `const bool`!

```
bool* const begin = crossed_out;      // ptr to first element
bool* const end = crossed_out + n;    // ptr after last element
// in the loop, pointer p successively points to all elements
for (bool* p = begin; p != end; ++p)
  *p = false; // *p is the element pointed to by p
```


Iteration durch Zeiger

```
bool* const begin = crossed_out;      // ptr to first element
bool* const end = crossed_out + n;    // ptr after last element
// in the loop, pointer p successively points to all elements
for (bool* p = begin; p != end; ++p)
  *p = false; // *p is the element pointed to by p
```


Iteration durch Zeiger

```
bool* const begin = crossed_out;      // ptr to first element
bool* const end = crossed_out + n;    // ptr after last element
// in the loop, pointer p successively points to all elements
for (bool* p = begin; p != end; ++p)
  *p = false; // *p is the element pointed to by p
```


Iteration durch Zeiger

```
bool* const begin = crossed_out;      // ptr to first element
bool* const end = crossed_out + n;    // ptr after last element
// in the loop, pointer p successively points to all elements
for (bool* p = begin; p != end; ++p)
  *p = false; // *p is the element pointed to by p
```


Iteration durch Zeiger

```
bool* const begin = crossed_out;      // ptr to first element
bool* const end = crossed_out + n;    // ptr after last element
// in the loop, pointer p successively points to all elements
for (bool* p = begin; p != end; ++p)
  *p = false; // *p is the element pointed to by p
```

Iteration durch Zeiger

```
bool* const begin = crossed_out;     // ptr to first element
bool* const end = crossed_out + n;   // ptr after last element
// in the loop, pointer p successively points to all elements
for (bool* p = begin; p != end; ++p)
  *p = false; // *p is the element pointed to by p
```

begin p Zeigerinkrement end

false

0 s Zellen crossed_out n-1

Iteration durch Zeiger

```
bool* const begin = crossed_out;     // ptr to first element
bool* const end = crossed_out + n;   // ptr after last element
// in the loop, pointer p successively points to all elements
for (bool* p = begin; p != end; ++p)
  *p = false; // *p is the element pointed to by p
```

begin p Zeiger-Vergleich end

false

0 s Zellen crossed_out n-1

Iteration durch Zeiger

```
bool* const begin = crossed_out;     // ptr to first element
bool* const end = crossed_out + n;   // ptr after last element
// in the loop, pointer p successively points to all elements
for (bool* p = begin; p != end; ++p)
  *p = false; // *p is the element pointed to by p
```

begin p Dereferenzierung, Zuweisung end

false false

0 s Zellen crossed_out n-1

Iteration durch Zeiger

```
bool* const begin = crossed_out;     // ptr to first element
bool* const end = crossed_out + n;   // ptr after last element
// in the loop, pointer p successively points to all elements
for (bool* p = begin; p != end; ++p)
  *p = false; // *p is the element pointed to by p
```

begin p Zeigerinkrement, usw.... end

false false

0 s Zellen crossed_out n-1

Iteration durch Zeiger

```
bool* const begin = crossed_out;     // ptr to first element
bool* const end = crossed_out + n;   // ptr after last element
// in the loop, pointer p successively points to all elements
for (bool* p = begin; p != end; ++p)
  *p = false; // *p is the element pointed to by p
```

begin p ...usw., Zeigerinkrement end

false false false false false false false false false false false false

0 s Zellen crossed_out n-1

Iteration durch Zeiger

```
bool* const begin = crossed_out;     // ptr to first element
bool* const end = crossed_out + n;   // ptr after last element
// in the loop, pointer p successively points to all elements
for (bool* p = begin; p != end; ++p)
  *p = false; // *p is the element pointed to by p
```

begin p Zeiger-Vergleich end

false false false false false false false false false false false false

0 s Zellen crossed_out n-1

Iteration durch Zeiger

```
bool* const begin = crossed_out;     // ptr to first element
bool* const end = crossed_out + n;   // ptr after last element
// in the loop, pointer p successively points to all elements
for (bool* p = begin; p != end; ++p)
  *p = false; // *p is the element pointed to by p
```


Warum Zeiger?

- Die (geringfügig) schnellere Iteration ist nicht der Punkt (Lesbarkeit spricht oft eher für wahlfreien Zugriff)
- **Grund 1**: wir brauchen sie für Felder mit variabler Länge (gleich...)
- **Grund 2**: **std::** Container-Algorithmen (Sortieren,...) brauchen *Iteratoren*

Zeiger sind die Iteratoren der Felder!

Dynamischer Speicher

- wie "besorgen" wir Speicher, der bei Kompilierung nicht vorhersehbar ist?
  - Sieb des Eratosthenes mit Eingabe von n
  - Allgemein: Feld variabler Länge

New-Ausdrücke

new *T*

- Effekt: neuer Speicher für ein Objekt vom Typ *T* wird bereitgestellt; der Wert des Ausdrucks ist dessen Adresse

New-Ausdrücke

new *T* [*expr*]

- Effekt: neuer Speicher für ein Feld der Länge *n* mit zugrundeliegendem Typ *T* wird bereitgestellt; Wert des Ausdrucks ist Adresse des ersten Elements

Der *Heap*

- Hauptspeicherbereich, aus dem das Programm neuen Speicher "holen" kann.

```
int* i = new int;
int* a = new int[3];
```

Sieb des Eratosthenes neu: *dynamischer* Speicher

```
int main()
{



  const unsigned int n = 1000;


  bool crossed_out[n];
  for (unsigned int i = 0; i < n; ++i)
    crossed_out[i] = false;

  // computation and output
  ....


  return 0;
}
```

```
int main()
{
  // input
  std::cout << "Compute prime numbers
    in {2,...,n-1} for n =? ";
  unsigned int n;
  std::cin >> n;

  bool* crossed_out = new bool[n];
  for (unsigned int i = 0; i < n; ++i)
    crossed_out[i] = false;

  // computation and output
  ....
  delete[] crossed_out;

  return 0;
}
```

Subskript-Operator auf *Zeiger* `crossed_out`

Sieb des Eratosthenes neu: *dynamischer* Speicher

```
int main()
{



  const unsigned int n = 1000;


  bool crossed_out[n];
  for (unsigned int i = 0; i < n; ++i)
    crossed_out[i] = false;

  // computation and output
  ....


  return 0;
}
```

```
int main()
{
  // input
  std::cout << "Compute prime numbers
    in {2,...,n-1} for n =? ";
  unsigned int n;
  std::cin >> n;

  bool* crossed_out = new bool[n];
  for (unsigned int i = 0; i < n; ++i)
    crossed_out[i] = false;

  // computation and output
  ....
  delete[] crossed_out;

  return 0;
}
```

Freigabe des nicht mehr benötigten dynamischen Speichers

Delete-Ausdrücke

Mit **`new`** erzeugte Objekte haben dynamische Speicherdauer: sie leben, bis sie explizit *gelöscht* werden:

`delete` *expr*

```
int* i = new int;
...
delete i;
```

`delete`-Operator

Zeiger vom Typ *T* *, der auf ein vorher mit `new` bereitgestelltes Objekt zeigt; Effekt: Speicher auf dem Heap wird wieder freigegeben.

Delete-Ausdrücke

Mit **`new`** erzeugte Objekte haben dynamische Speicherdauer: sie leben, bis sie explizit *gelöscht* werden.

`delete[]` *expr*

```
int* a = new int[3];
...
delete[] a;
```

`delete`-Operator

Zeiger vom Typ *T* *, der auf ein vorher mit `new` bereitgestelltes Feld zeigt; Effekt: Speicher auf dem Heap wird wieder freigegeben.