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

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++ Ultimatum*" 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

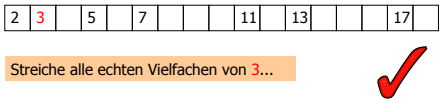
2	3		5		7		9		11		13		15		17	

...und gehe zur nächsten Zahl

Felder: erste Anwendung

Das Sieb des Eratosthenes

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



Streiche alle echten Vielfachen von 3...

Felder: erste Anwendung

Das Sieb des Eratosthenes

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

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()
{
  // definition and initialization: provides us with
  // Booleans crossed_out[0],..., crossed_out[999]
  bool crossed_out[1000];
  for (unsigned int i = 0; i < 1000; ++i)
    crossed_out[i] = false;

  // computation and output
  std::cout << "Prime numbers in {2,...,999}:\n";
  for (unsigned int i = 2; i < 1000; ++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 < 1000; m += i)
        crossed_out[m] = true;
    }
  std::cout << "\n";

  return 0;
}
```

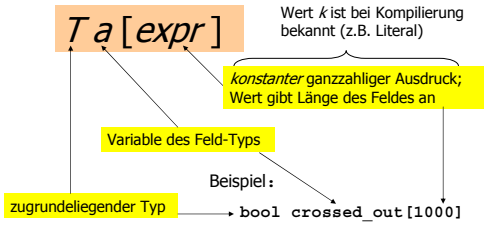
Berechnet alle Primzahlen < 1000

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

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):



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* 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

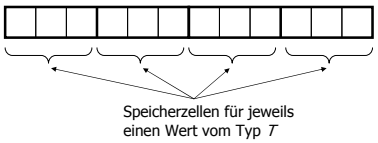
- `int a[] = {4, 3, 5, 2, 1};`

  auch ok; Länge wird deduziert

Speicherlayout eines Feldes

- Ein Feld belegt einen *zusammen-hängenden* Speicherbereich

Beispiel: Feld mit 4 Elementen



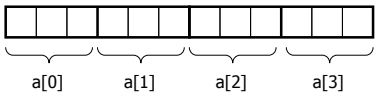
Wahlfreier Zugriff (*Random Access*)

Der L-Wert

Wert *i*

a [*expr*]

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



Wahlfreier Zugriff (*Random Access*)

a [*expr*]

Der Wert *i* von expr heisst *Feldindex*

[]: Subskript-Operator

Wahlfreier Zugriff (*Random Access*)

Die Deklaration

implizite Definition des Typs von *a*

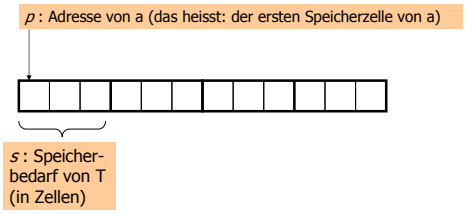
T a [*expr*]

kann man deshalb auch lesen als

"*a* [*expr*] ist vom Typ T"

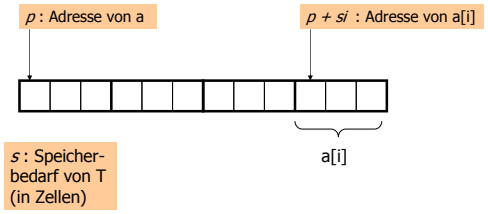
Wahlfreier Zugriff (*Random Access*)

- Wahlfreier Zugriff ist sehr effizient:



Wahlfreier Zugriff (*Random Access*)

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



Wahlfreier Zugriff (*Random Access*)

Warnung:

Die Länge *n* eines Feldes kann nicht abgefragt werden, man muss sie wissen. Für jeden Feldindex *i* muss

$0 \le i < n$

gelten, andernfalls ist das Programm-verhalten undefiniert!

Felder sind nicht *selbstbeschreibend*

- 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 nicht *selbstbeschreibend*

- Felder sind "Erblast" der Sprache C und aus heutiger Sicht primitiv
- Ein Feld "merkt sich" nur die Adresse des ersten Elements

bei legalen Indizes!

  - kein Problem bei *Random Access,* aber
  - beim Initialisieren / Zuweisen ist dann nicht klar, wieviele Elemente kopiert werden müssen

```
int a[5] = {4,3,5,2,1};
int b[5];
b = a; // wieviele Elemente haben a und b nochmal?
```

Felder als Daten-Container

Container:

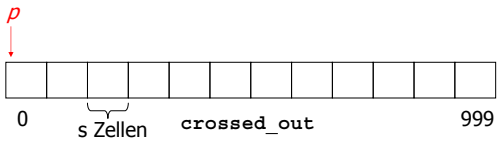
- Objekt, das andere Objekte speichern kann...
- ...und die Möglichkeit anbietet, über die gespeicherten Objekte zu *iterieren* (z.B. Kinospielpläne...)

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;
```

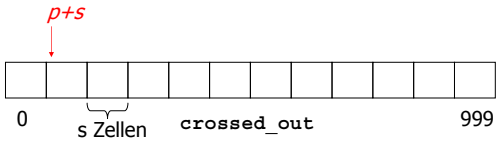
Berechnungsaufwand:



Iteration durch wahlfreien Zugriff

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

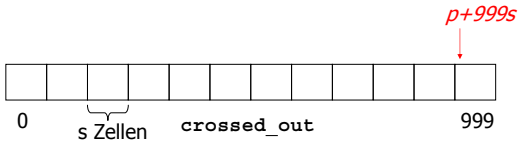
Berechnungsaufwand:



Iteration durch wahlfreien Zugriff

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

Berechnungsaufwand:

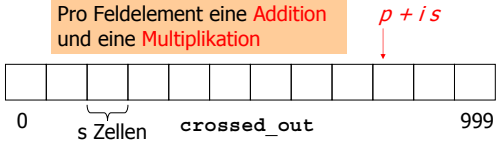


Iteration durch wahlfreien Zugriff

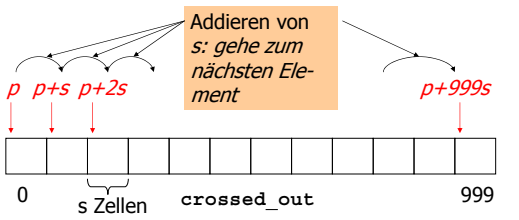
```
for (unsigned int i = 0; i < 1000; ++i)
    crossed_out[i] = false;
```

Berechnungsaufwand:

Pro Feldelement eine Addition und eine Multiplikation



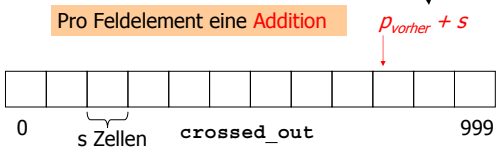
Effizientere und natürlichere Iteration



Effizientere und natürlichere Iteration durch *Zeiger*

```
bool* begin = crossed_out;      // pointer to first element
bool* end = crossed_out + 1000; // pointer 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



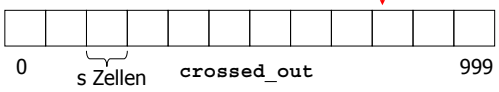
Effizientere und natürlichere Iteration durch *Zeiger*

```
bool* begin = crossed_out;      // po... st element
bool* end = crossed_out + ...   ... last element
// in the loop ...              ...ly points to all elements
for ... = end; ++p
  ... // *p is the element pointed to by p
```

wird noch genau erklärt!

Pro Feldelement eine Addition

$p_{vorher} + s$



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)

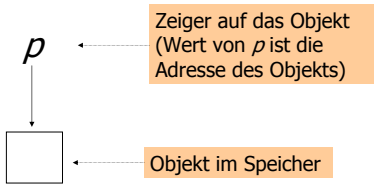
Zeiger-Typen

$T*$ 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

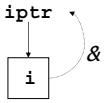


Adressoperator

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

& L-Wert

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



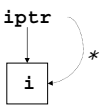
Kann man lesen als `int *iptr = &i` (d.h. `*iptr` ist vom Typ `int`).
Wieder ein Fall von impliziter Typdefinition!

Dereferenzierungsoperator

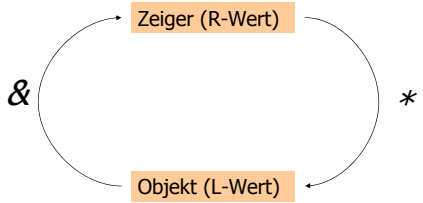
- liefert einen L-Wert für ein Objekt, gegeben durch einen Zeiger auf das Objekt

** R-Wert*

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



Dereferenzierungsoperator = Adressoperator $^{-1}$



Nichtinitialisierte Zeiger

- sind eine typische Fehlerquelle

```
int* iptr;     // nicht initialisiert
...
int j = *iptr; // Objekt an "zufaelliger" Adresse
               // --> undefiniertes Verhalten
```

Nullzeiger

- Zeiger, die (noch) nicht auf ein Objekt zeigen, sollten mit 0 initialisiert werden

```
int* iptr = 0;  // Nullzeiger
...
int j = *iptr;  // kein Objekt an dieser Adresse
                // --> "sicherer" Laufzeitfehler
```

Nullzeiger

- Zeiger, die (noch) nicht auf ein Objekt zeigen, sollten mit 0 initialisiert werden

```
int* iptr = 0;    // Nullzeiger
...
if (iptr != 0) {   // so ist's am besten
  int j = *iptr;
  ...
}
```

Felder "sind" Zeiger

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

Feld-nach-Zeiger-Konversion

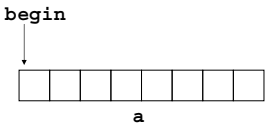
- Ergebnis der Konversion ist ein Zeiger auf das erste Element (Feldindex 0)
- Tritt ein Feld in einem Ausdruck auf, so wird es automatisch konvertiert

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;
```



Ist das äquivalent zu

```
int a[5];
int* begin = &a[0];
```

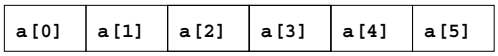
?

Nicht ganz, denn hier wird zusätzlich noch a[0] ausgewertet!

Zeiger-Arithmetik

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

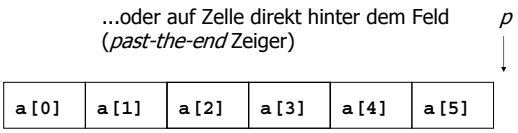
p : Zeiger auf ein Feldelement



Zeiger-Arithmetik

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

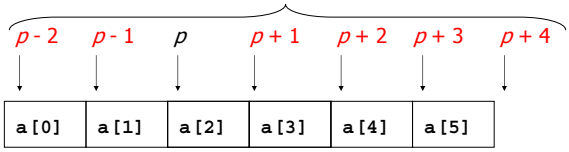
...oder auf Zelle direkt hinter dem Feld (*past-the-end* Zeiger)



Zeiger-Arithmetik

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

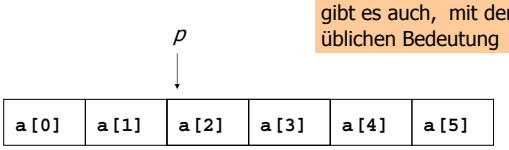
Addenden mit Ergebniszeiger ausserhalb dieses Bereichs sind illegal!



Zeiger-Arithmetik

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

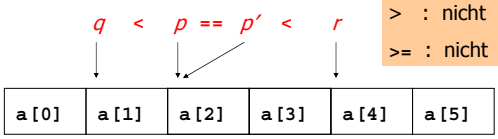
`+= , -= , ++ , --`
gibt es auch, mit der üblichen Bedeutung



Zeiger-Arithmetik

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

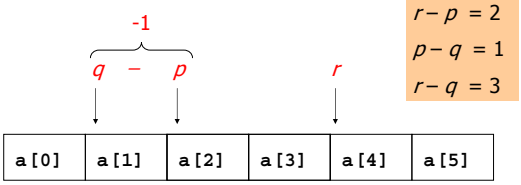
`!=` : nicht `==`
`<=` : `<` oder `==`
`>` : nicht `<=`
`>=` : nicht `<`



Zeiger-Arithmetik

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

$r - p = 2$
$p - q = 1$
$r - q = 3$



Die Wahrheit über den Subskript-Operator

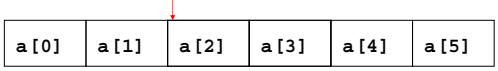
- arbeitet eigentlich auf Zeigern:

$a\,[\,expr\,]$

ist eine Abkürzung für

$*\,(a + expr\,)$

Bespiel: *expr* hat Wert 2



Die Wahrheit über den Subskript-Operator

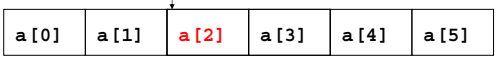
- arbeitet eigentlich auf Zeigern:

a [*expr*]

ist eine Abkürzung für

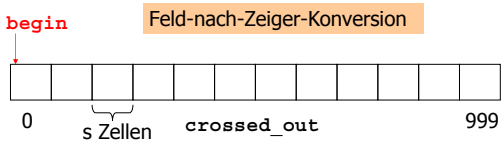
** (a + expr)*

Bespiel: *expr* hat Wert 2



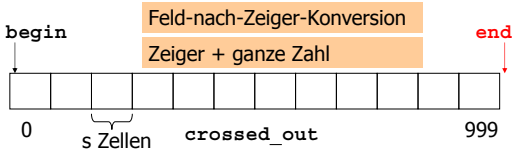
Iteration durch Zeiger

```
bool* begin = crossed_out;      // pointer to first element
bool* end = crossed_out + 1000; // pointer 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
```



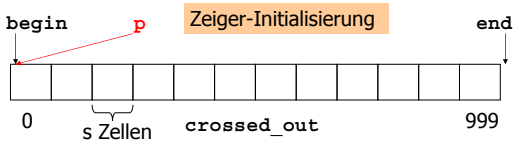
Iteration durch Zeiger

```
bool* begin = crossed_out;      // pointer to first element
bool* end = crossed_out + 1000; // pointer 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
```



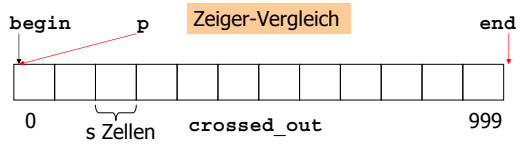
Iteration durch Zeiger

```
bool* begin = crossed_out;      // pointer to first element
bool* end = crossed_out + 1000; // pointer 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
```



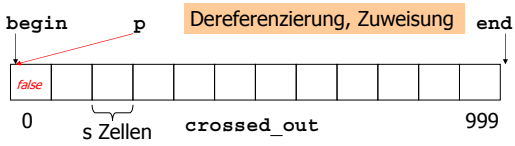
Iteration durch Zeiger

```
bool* begin = crossed_out;      // pointer to first element
bool* end = crossed_out + 1000; // pointer 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
```



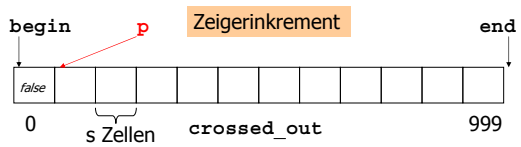
Iteration durch Zeiger

```
bool* begin = crossed_out;      // pointer to first element
bool* end = crossed_out + 1000; // pointer 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
```



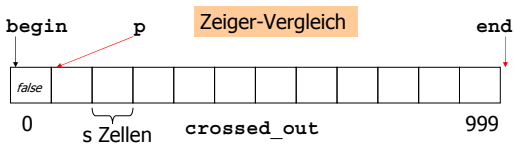
Iteration durch Zeiger

```
bool* begin = crossed_out;      // pointer to first element
bool* end = crossed_out + 1000; // pointer 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
```



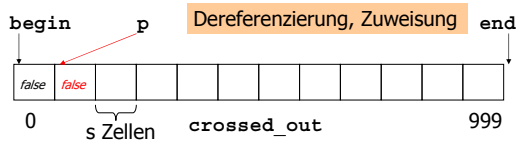
Iteration durch Zeiger

```
bool* begin = crossed_out;      // pointer to first element
bool* end = crossed_out + 1000; // pointer 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
```



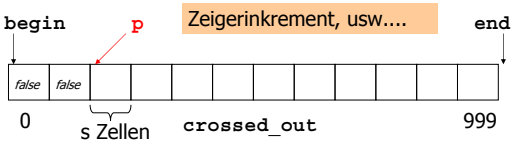
Iteration durch Zeiger

```
bool* begin = crossed_out;      // pointer to first element
bool* end = crossed_out + 1000; // pointer 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
```



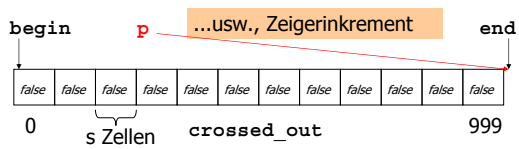
Iteration durch Zeiger

```
bool* begin = crossed_out;      // pointer to first element
bool* end = crossed_out + 1000; // pointer 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
```



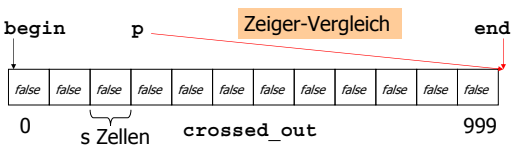
Iteration durch Zeiger

```
bool* begin = crossed_out;      // pointer to first element
bool* end = crossed_out + 1000; // pointer 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
```



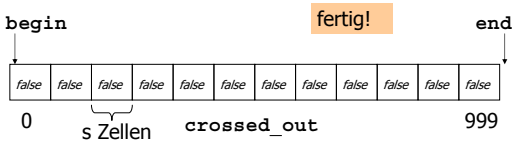
Iteration durch Zeiger

```
bool* begin = crossed_out;      // pointer to first element
bool* end = crossed_out + 1000; // pointer 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* begin = crossed_out;      // pointer to first element
bool* end = crossed_out + 1000; // pointer 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* [*expr*]

Ausdruck vom Typ *T** (Zeiger)

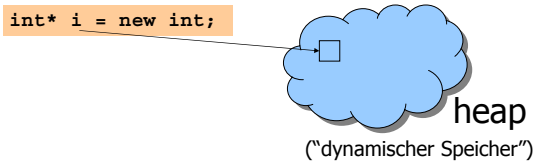
Typ **`int`**, Wert *n* ; *expr* nicht notwendigerweise konstant

`new`-Operator

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

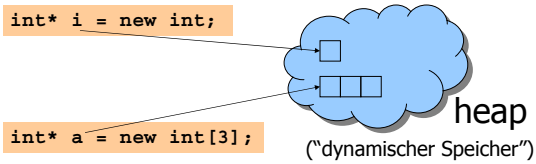
Der *Heap*

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



Der *Heap*

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



Sieb des Eratosthenes neu: *dynamischer* Speicher

```
int main()
{

  bool crossed_out[1000];
  for (unsigned int i = 0; i < 1000; ++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()
{

  bool crossed_out[1000];
  for (unsigned int i = 0; i < 1000; ++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*

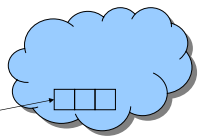
```
int* a = new int[3];
...
delete[] a;
```

delete-Operator

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

Dynamische-Speicher-Richtlinie

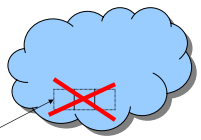
Zu jedem **new** gibt es ein passendes **delete**



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

Dynamische-Speicher-Richtlinie

Zu jedem **new** gibt es ein passendes **delete**



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

Dynamische-Speicher-Richtlinie

Zu jedem **`new`** gibt es ein passendes **`delete`**

Nichtbefolgung führt zu *Speicherlecks* (ungenutzter, aber nicht mehr verfügbarer Speicher auf dem Heap)