10. Referenztypen

Referenztypen: Definition und Initialisierung, Call By Value , Call by Reference, Temporäre Objekte, Const-Referenzen

Swap ?

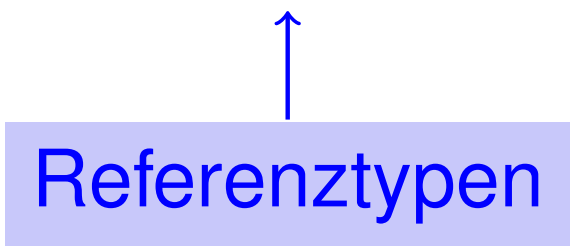
```
void swap (int a, int b) {
 int t = a;
 a = b;
 b = t;
}

int main(){
     int a = 2;
     int b = 1;
     swap (a, b);
     assert (a==1 && b == 2); // fail!
}
```

Swap !

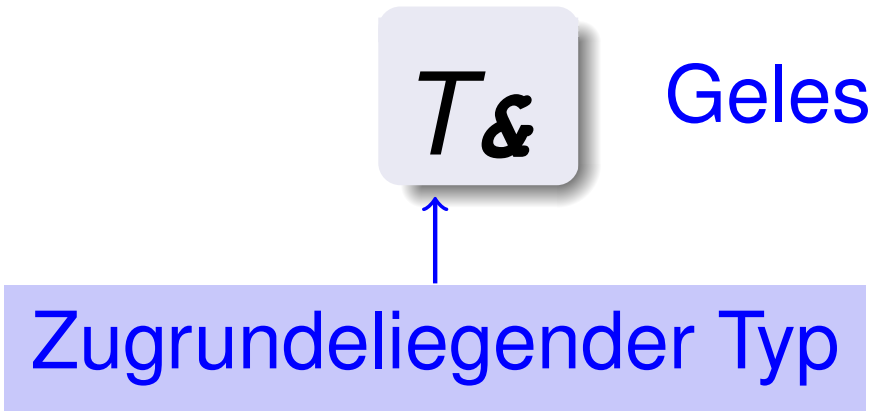
```
// POST: values of a and b are exchanged
void swap (int& a, int& b) {
 int t = a;
 a = b;
 b = t;
}

int main(){
     int a = 2;
     int b = 1;
     swap (a, b);
     assert (a==1 && b == 2); // ok!
}
```

Referenztypen

- Wir können Funktionen in die Lage versetzen, die Werte ihrer Aufrufargumente zu ändern!
- Kein neues Konzept auf der Funktionenseite, sondern eine neue Klasse von Typen



Referenztypen: Definition



- *T&* hat den gleichen Wertebereich und gleiche Funktionalität wie *T*, ...
- nur Initialisierung und Zuweisung funktionieren anders.

Referenztypen: Initialisierung

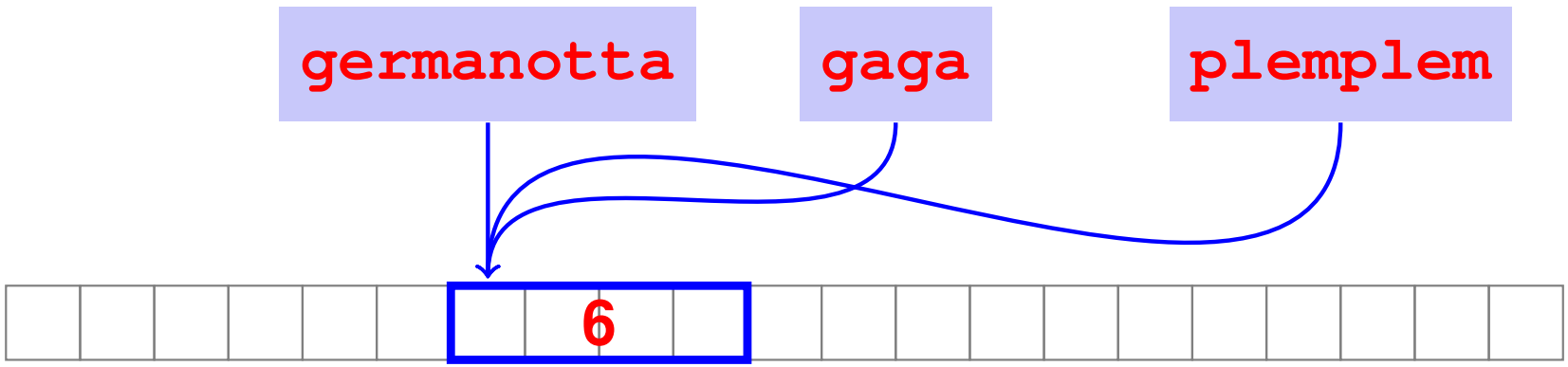
- Eine nicht konstante Variable mit Referenztyp (eine *Referenz*) kann nur mit einem L-Wert initialisiert werden.
- Die Variable wird dabei ein *Alias* des L-Werts (ein anderer Name für das referenzierte Objekt).

Stefani Germanotta alias Lady Gaga

```
int germanotta = 5;
int& gaga = germanotta; // alias
int& plemplem = gaga;//new alias
gaga = 6;
```

Zuweisung an den L-Wert hinter dem Alias

```
std::cout << germanotta; // 6
```



Referenztypen: Realisierung

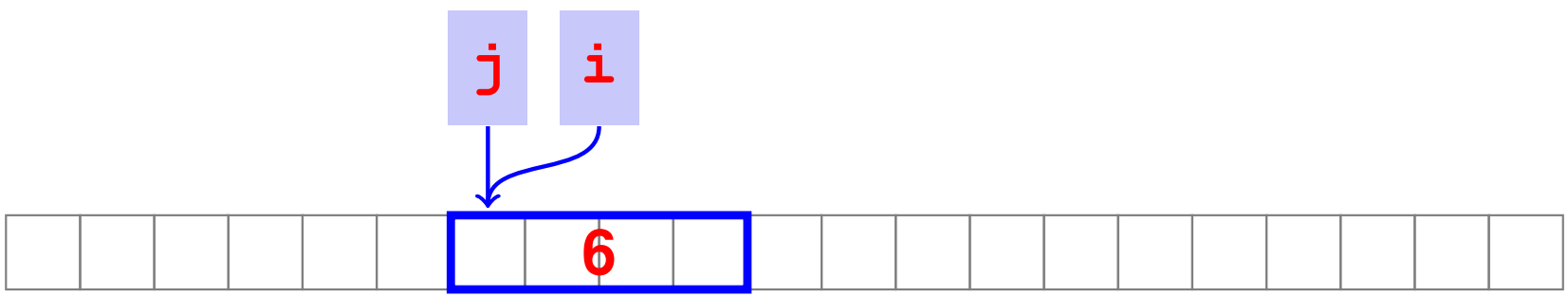
Intern wird ein Wert vom Typ *T&* durch die Adresse eines Objekts vom Typ *T* repräsentiert.

```
// Fehler: j muss Alias von irgendetwas sein
int& j;

// Fehler: Das Literal 5 hat keine Adresse
int& k= 5;
```

Call by Reference

Referenztypen erlauben Funktionen, die Werte ihrer Aufrufargumente zu ändern:

```
void increment (int& i) ← Initialisierung der formalen Argumente
{ // i becomes alias of call argument
    ++i;
}
...
int j = 5;
increment (j);
std::cout << j << "\n"; // outputs 6
```



Call by Reference

- formales Argument hat Referenztyp:
  $\Rightarrow$ **Call by Reference**

  formales Argument wird (intern) mit der *Adresse* des Aufrufarguments (L-Wert) initialisiert und wird damit zu einem *Alias*

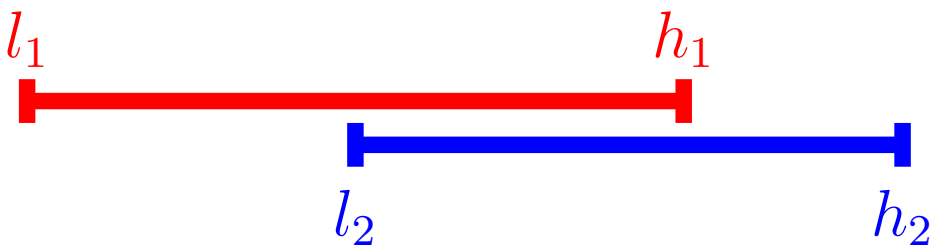
Call by Value

- formales Argument hat keinen Referenztyp:
  $\Rightarrow$ **Call by Value**

  formales Argument wird mit dem *Wert* des Aufrufarguments (R-Wert) initialisiert und wird damit zu einer *Kopie*

Rückgabe des Intervallschnitts

```
// PRE:  [a1, b1], [a2, b2] are (generalized) intervals,
// POST: returns true if [a1, b1], [a2, b2] intersect, in which case
//       [l, h] contains the intersection of [a1, b1], [a2, b2]
bool intervals_intersect (int& l, int& h,
                          int a1, int b1, int a2, int b2) {
  int l1 = std::min (a1, b1);
  int h1 = std::max (a1, b1);
  int l2 = std::min (a2, b2);
  int h2 = std::max (a2, b2);
  l = std::max (l1, l2);
  h = std::min (h1, h2);
  return l <= h;
}
...

int l = 0; int r = 0;
if (intervals_intersect (l, r, 0, 2, 1, 3))
    std::cout << "[" << l << "," << r << "]" << "\n"; // Ausgabe [1,2]
```



Referenzen weitergeben

```
// POST: a <= b
void sort (int& a, int& b) {
   if (a > b)
     std::swap (a, b);
}

// PRE:  [a1, b1], [a2, b2] are (generalized) intervals,
// POST: returns if [a1, b1], [a2, b2] intersect, in which case
//       [l, h] contains intersection of [a1, b1], [a2, b2]
bool intervals_intersect (int& l, int& h,
                          int a1, int b1, int a2, int b2) {
  sort (a1, b1);    // call by value ausgenutzt!
  sort (a2, b2);    // (Aufrufargumente unveraendert)
  l = std::max (a1, a2);
  h = std::min (b1, b2);
  return l <= h;
}
```

Return by Value / Reference

- Auch der Rückgabetyp einer Funktion kann ein Referenztyp sein (return by reference)
- In diesem Fall ist der Funktionsausfruf selbst einen L-Wert

```
int& increment (int& i)
{
    return ++i;
}
```

Exakt die Semantik des Prä-Inkrements

Temporäre Objekte

Was ist hier falsch?

```
int& foo (int i)
{
    return i;
}
```

Rückgabewert vom Typ `int&` wird Alias des formalen Arguments, dessen Speicherdauer aber nach Auswertung des Funktionsaufrufes endet.

```
int k = 3;
int& j = foo(k); // j ist Alias einer "Leiche"

std::cout << j << "\n";  // undefined behavior
```

Die Referenz-Richtlinie

Referenz-Richtlinie

Wenn man eine Referenz erzeugt, muss das Objekt, auf das sie verweist, mindestens so lange „leben” wie die Referenz selbst.

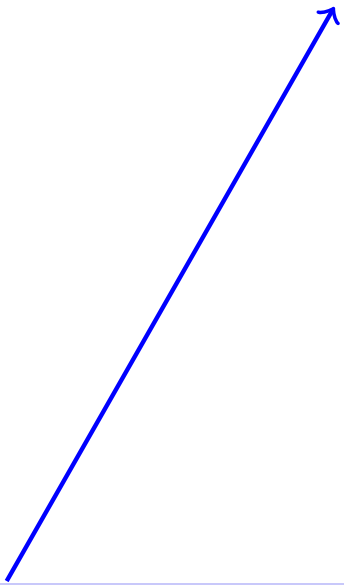
Const-Referenzen

```
void foo (const int& i)
{
    ...
}
```

Absurd?

Const-Referenzen

```
void foo (const fetter_typ& t)
{
    ...
}
```



Beispiel: `ifmp::integer`

Beim Aufruf muss nur eine Addresse kopiert werden Scheinbarer Nachteil: Operator kann nur mit L-Werten aufgerufen werden. **Die Wahrheit:** *für const-Referenzen gehen auch R-Werte*

Was genau ist konstant?

Betrachte L-Wert vom Typ **`const`** *T* (R-Werte sind ohnehin konstant)

- Fall 1: *T* ist kein Referenztyp

  Dann ist der L-Wert eine Konstante.

```
const int n = 5;
int& i = n; // error: const-qualification is discarded
i = 6;
```

Der Schummelversuch wird vom Compiler erkannt

Was genau ist konstant?

Betrachte L-Wert vom Typ **`const`** *T* (R-Werte sind ohnehin konstant)

- Fall 2: *T* ist Referenztyp

  Dann ist der L-Wert ein Lese-Alias, durch den der Wert dahinter nicht verändert werden darf.

```
int n = 5;
const int& i = n;// i: Lese-Alias von n
int& j = n;      // j: Lese-Schreib-Alias
i = 6;           // Fehler: i ist Lese-Alias
j = 6;           // ok: n bekommt Wert 6
```

Const-Referenzen

- haben Typ **`const`** *T* **`&`** (= **`const`** (*T* **`&`**))
- können auch mit R-Werten initialisiert werden (Compiler erzeugt temporäres Objekt ausreichender Lebensdauer)

`const` *T***`&`** **`r`** = *lvalue*;

r wird mit der Adresse von *lvalue* initialisiert (effizient)

`const` *T***`&`** **`r`** = *rvalue*;

r wird mit der Adresse eines temporären Objektes vom Wert des *rvalue* initialisiert (flexibel)

`const` *T* vs. `const` *T*`&`

Regel

`const` *T* kann als Argumenttyp immer durch `const` *T* `&` ersetzt werden; dies lohnt sich aber nicht für fundamentale Typen.

Beispiele folgen später in der Vorlesung

T vs. *T&*

Achtung!

T kann als Argumenttyp nicht immer durch *T&* ersetzt werden

11. Felder (Arrays) I

Feldtypen, Sieb des Eratosthenes,
Speicherlayout, Iteration, Vektoren, Zeichen, Texte

Felder: Motivation

- Wir können jetzt über Zahlen iterieren

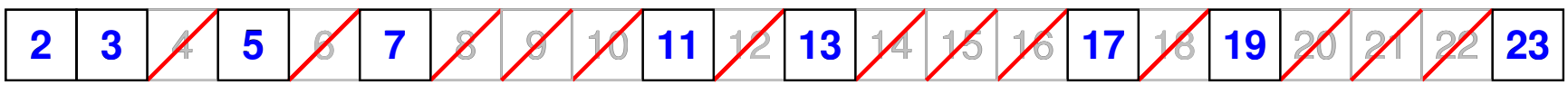
```
for (int i=0; i<n ; ++i) ...
```

- Oft muss man aber über *Daten* iterieren (Beispiel: Finde ein Kino in Zürich, das heute „C++ Runners” zeigt)
- Felder dienen zum Speichern *gleichartiger* Daten (Beispiel: Spielpläne aller Zürcher Kinos)

Felder: erste Anwendung

Das Sieb des Eratosthenes

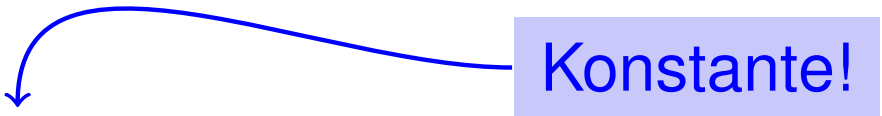
- berechnet alle Primzahlen $< n$
- Methode: Ausstreichen der Nicht-Primzahlen



Am Ende des Streichungsprozesses bleiben nur die Primzahlen übrig.

- Frage: wie streichen wir Zahlen aus ??
- Antwort: mit einem *Feld.*

Sieb des Eratosthenes: Initialisierung




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

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

Sieb des Eratosthenes: Berechnung

```
// 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";
```

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]

Zugrundeliegender Typ

Variable des Feld-Typs

Konstanter ganzzahliger Ausdruck
Wert gibt die Länge des Feldes an

Typ von **`a`**: „T[k]"
Wertebereich von **`a`**: T^k

Wert ist bei Kompilierung bekannt.
Z.B. Literal, Konstante

Beispiel: **`bool crossed_out[n]`**

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.

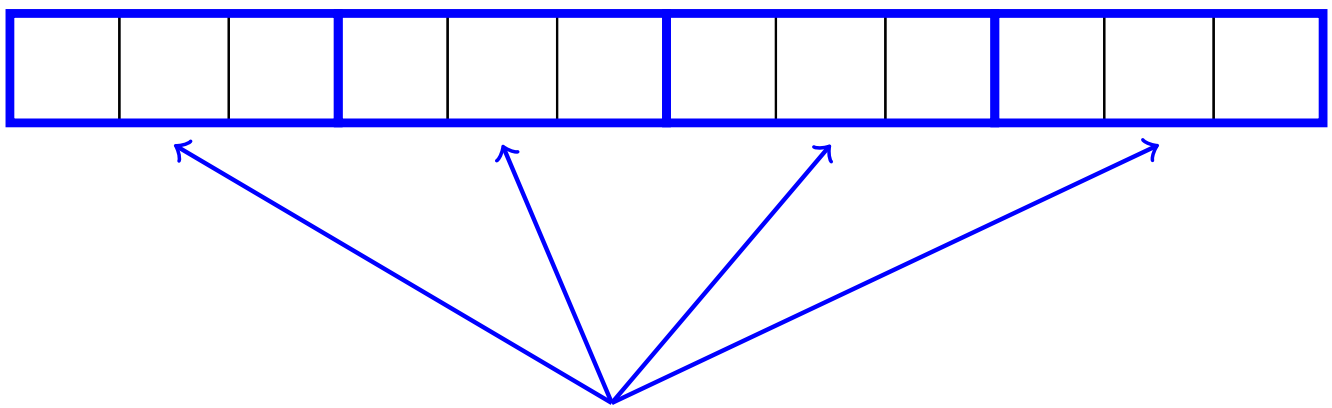
- `int a[] = {4, 3, 5, 2, 1};`

Auch ok: Länge wird vom Compiler deduziert

Speicherlayout eines Feldes

- Ein Feld belegt einen *zusammenhängenden* Speicherbereich

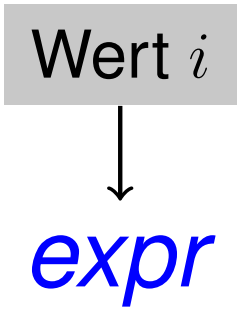
Beispiel: ein Feld mit 4 Elementen.



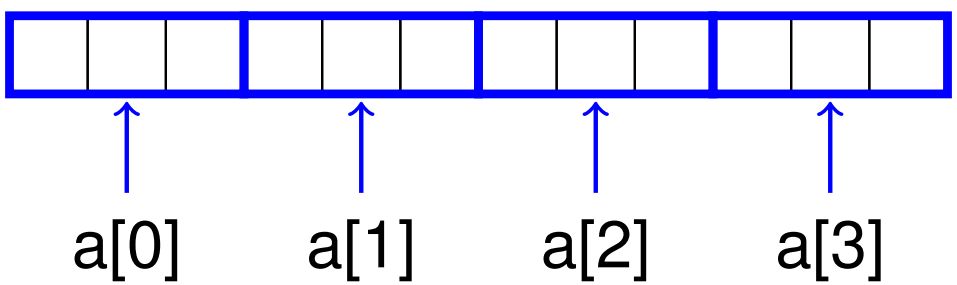
Speicherzellen für jeweils einen Wert vom Typ `T`

Wahlfreier Zugriff (Random Access)

Der L-Wert



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



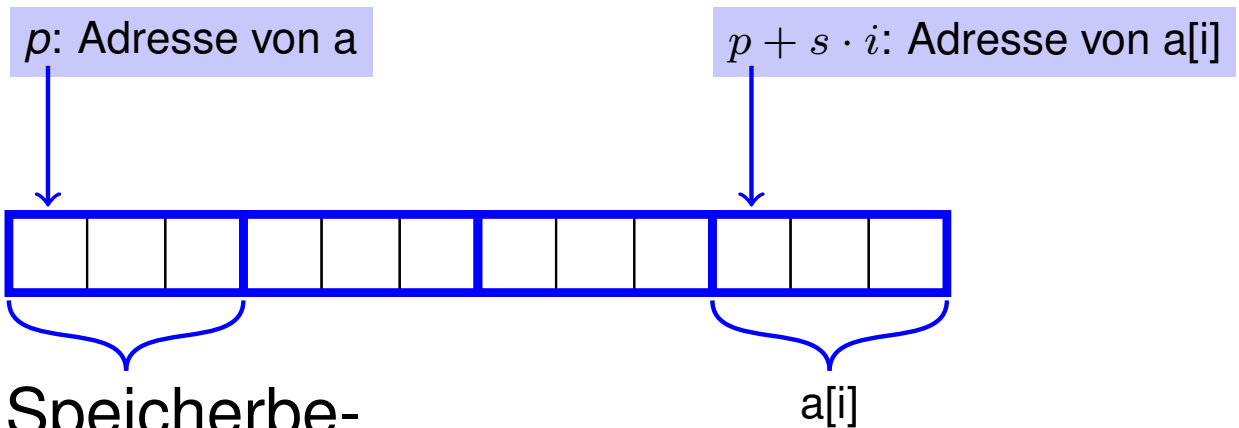
Wahlfreier Zugriff (Random Access)

a [*expr*]

Der Wert i von *expr* heisst *Feldindex*.
`[]`: Subskript-Operator

Wahlfreier Zugriff (Random Access)

- Wahlfreier Zugriff ist sehr effizient:



s: Speicherbedarf von T (in Zellen)

Felder sind primitiv

- Der Zugriff auf Elemente ausserhalb der gültigen Grenzen eines Feldes führt zu undefiniertem Verhalten.

```
int arr[10];
for (int i=0; i<=10; ++i)
  arr[i] = 30;// Fehler: Zugriff auf arr[10]!
```

Felder sind primitiv

Prüfung der Feldgrenzen

In Abwesenheit spezieller Compiler- oder Laufzeitunterstützung ist es die alleinige *Verantwortung des Programmierers*, die Gültigkeit aller Elementzugriffe zu prüfen.

Felder sind primitiv (II)

- 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 sind Felder sehr maschinennah und effizient, bieten aber keinen „Luxus” wie eingebautes Initialisieren und Kopieren.
- Fehlendes Prüfen der Feldgrenzen hat weitreichende Konsequenzen. Code mit unerlaubten aber möglichen Index-Zugriffen wurde von Schadsoftware schon (viel zu) oft ausgenutzt.
- Die Standard-Bibliothek bietet komfortable Alternativen (mehr dazu später)

Vektoren

- Offensichtlicher Nachteil statischer Felder: *konstante Feldlänge*

```
const unsigned int n = 1000;
bool crossed_out[n];
...
```

- Abhilfe: Verwendung des Typs **`Vector`** aus der Standardbibliothek

```
#include <vector>
...
std::vector<bool> crossed_out (n, false);
```

Initialisierung mit n Elementen Initialwert **`false`**.

Elementtyp, in spitzen Klammern

Sieb des Eratosthenes mit Vektoren

```
#include <iostream>
#include <vector>  // standard containers with array functionality
int main() {
  // input
  std::cout << "Compute prime numbers in {2,...,n−1} for n =? ";
  unsigned int n;
  std::cin >> n;

  // definition and initialization: provides us with Booleans
  // crossed_out[0],..., crossed_out[n−1], initialized to false
  std::vector<bool> crossed_out (n, 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;
}
```

Zeichen und Texte

- Texte haben wir schon gesehen:

```
std::cout << "Prime numbers in {2,...,999}:\n";
```

String-Literal

- Können wir auch „richtig” mit Texten arbeiten?
  Ja:

Zeichen:	Wert des fundamentalen Typs **`char`**
Text:	Feld mit zugrundeliegendem Typ **`char`**

Der Typ `char` (“character”)

- repräsentiert druckbare Zeichen (z.B. `'a'`) und *Steuerzeichen* (z.B. `'\n'`)

```
char c = 'a'
```

definiert Variable c vom Typ **`char`** mit Wert **`'a'`**

Literal vom Typ **`char`**

Der Typ `char` (“character”)

- ist formal ein ganzzahliger Typ
  - Werte konvertierbar nach **`int`** / **`unsigned int`**
  - Alle arithmetischen Operatoren verfügbar (Nutzen zweifelhaft: was ist **`'a'/'b'`** ?)
  - Werte belegen meistens 8 Bit

Wertebereich:
$\{-128, \ldots, 127\}$ oder $\{0, \ldots, 255\}$

Der ASCII-Code

- definiert konkrete Konversionsregeln
  `char` $\longrightarrow$ **`int / unsigned int`**
- wird von fast allen Plattformen benutzt

Zeichen $\longrightarrow \{0, \ldots, 127\}$

`'A', 'B', ... , 'Z'` $\longrightarrow 65, 66, ..., 90$

`'a', 'b', ... , 'z'` $\longrightarrow 97, 98, ..., 122$

`'0', '1', ... , '9'` $\longrightarrow 48, 49, ..., 57$

```
for (char c = 'a'; c <= 'z'; ++c)
    std::cout << c;
```

```
abcdefghijklmnopqrstuvwxyz
```

*Erweiterung von ASCII: UTF8

- Internationalisierung von Software $\Rightarrow$ grosse Zeichensätze nötig. Heute Üblich: Unicode, 100 Schriftsysteme, 110000 Zeichen.
- ASCII kann mit 7 Bits codiert werden. Ein achtes Bit ist verwendbar, um das Vorkommen weiterer Bits festzulegen $\rightarrow$ UTF8 Encoding

Bits	Encoding
7	`0xxxxxxx`
11	`110xxxxx 10xxxxxx`
16	`1110xxxx 10xxxxxx 10xxxxxx`
21	`11110xxx 10xxxxxx 10xxxxxx 10xxxxxx`
26	`111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx`
31	`1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx`

Interessante Eigenschaft: bei jedem Byte kann entschieden werden, ob UTF8 Zeichen beginnt.