11. Referenztypen

Referenztypen: Definition und Initialisierung, Pass by Value, Pass by Reference, temporäre Objekte, Const-Referenzen

Swap!

```
// POST: values of x and y have been exchanged
void swap(int& x, int& y) {
 int t = x;
 x = y;
 y = 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*

Referenztypen: Definition

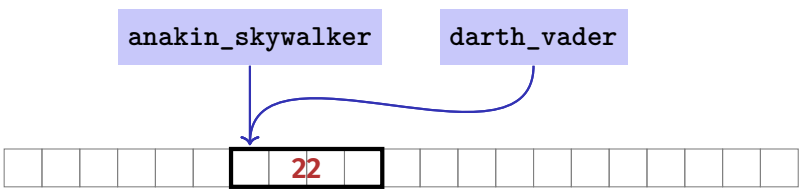
- *T&* hat den gleichen Wertebereich und gleiche Funktionalität wie *T* …
- …aber Initialisierung und Zuweisung funktionieren anders

Anakin Skywalker alias Darth Vader

Anakin Skywalker alias Darth Vader

```
int anakin_skywalker = 9;
int& darth_vader = anakin_skywalker; // Alias
darth_vader = 22;

std::cout << anakin_skywalker; // 22
```

Zuweisung an den L-Wert hinter dem Alias



Referenztypen: Initialisierung & Zuweisung

```
int& darth_vader = anakin_skywalker;
darth_vader = 22; // Effekt: anakin_skywalker = 22
```

- Eine Variable mit Referenztyp (eine *Referenz*) muss mit einem L-Wert initialisiert werden
- Die Variable wird dabei ein *Alias* des L-Werts (ein anderer Name für das referenzierte Objekt)
- Zuweisung an die Referenz erfolgt an das Objekt *hinter* dem Alias

Referenztypen: Realisierung

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

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

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

Pass by Reference

Referenztypen erlauben Funktionen, die Werte ihrer Aufrufargumente zu ändern

```
void increment (int& i) {
  ++i;
}
...
int j = 5;
increment (j);
std::cout << j; // 6
```

Initialisierung der formalen Argumente: `i` wird Alias des Aufrufarguments `j`

Pass by Reference

Formales Argument *hat* Referenztyp:

$\Rightarrow$ *Pass by Reference*

Formales Argument wird (intern) mit der *Adresse* des Aufruf-arguments (L-Wert) initialisiert und wird damit zu einem *Alias*.

Pass by Value

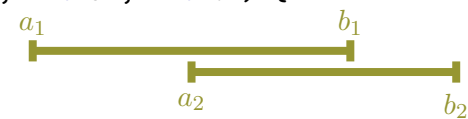
Formales Argument *hat keinen* Referenztyp:

$\Rightarrow$ *Pass by Value*

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

Referenzen im Kontext von intervals_intersect

```
// 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) {
  sort(a1, b1);
  sort(a2, b2);
  l = std::max(a1, a2); // Zuweisungen
  h = std::min(b1, b2); // via Referenzen
  return l <= h;
}
...
int lo = 0; int hi = 0;
if (intervals_intersect(lo, hi, 0, 2, 1, 3)) // Initialisierung
    std::cout << "[" << lo << "," << hi << "]" << "\n"; // [1,2]
```



Referenzen im Kontext von intervals_intersect

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

bool intervals_intersect(int& l, int& h,
                         int a1, int b1, int a2, int b2) {
  sort(a1, b1); // Initialisierung
  sort(a2, b2); // Initialisierung
  l = std::max(a1, a2);
  h = std::min(b1, b2);
  return l <= h;
}
```

Return by Reference

- Auch der Rückgabetyp einer Funktion kann ein Referenztyp sein: *Return by Reference*

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

- Aufruf **`inc(x)`**, für eine **`int`**-Variable **`x`**, hat exakt die Semantik des Prä-Inkrements **`++x`**
- Funktionsausfruf *selbst* ist nun ein L-Wert
- Daher möglich: **`inc(inc(x))`** oder **`++(inc(x))`**

Temporäre Objekte

Was ist hier falsch?

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

Rückgabewert vom Typ `int&` wird Alias des formalen Arguments (lokale Variable `i`), dessen Speicherdauer aber nach Auswertung des Funktionsaufrufes endet

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

- haben Typ **`const`** *T* **`&`**
- Typ kann verstanden werden als „(**`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 (pragmatisch)

Was genau ist konstant?

Betrachte L-Wert vom Typ `const` *T*. **Fall: 1** *T ist kein* Referenztyp.

⇒ Dann ist der *L-Wert eine Konstante*

```
const int n = 5;
int& a = n; // Compilerfehler: const-qualification discarded
a = 6;
```

Unser *Schummelversuch* wird vom Compiler erkannt

Was genau ist konstant?

Betrachte L-Wert vom Typ `const` *T*. **Fall 2:** *T ist* Referenztyp.

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

```
int n = 5;

const int& r = n; // r ist Lese-Alias von n
r = 6;            // Compilerfehler: read-only reference

int& rw = n;      // rw ist Lese-Schreib-Alias
rw = 6;           // OK
```

Wann `const` *T*`&` verwenden?

```
void f_1(T& arg);
```

```
void f_2(const T& arg);
```

- Argumenttypen sind Referenzen; Aufrufargumente werden daher nicht kopiert, was effizient ist
- Aber nur **`f_2`** „verspricht“ Argument nicht zu verändern

Regel

Funktionsargumenttypen, falls möglich, als **`const`** *T*`&` (*pass by read-only reference*) deklarieren: effizient *und* sicher.

Lohnt sich i.d.R. nicht für fundamentale Typen (**`int`**, **`double`**, …). Wir lernen später in der Vorlesung Typen kennen, die mehr Speicherplatz benötigen.

12. Vektoren I

Vektoren, Sieb des Eratosthenes, Speicherlayout, Iteration

Vektoren: 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++ Runner 2049“ zeigt)
- Vektoren dienen zum Speichern *gleichartiger* Daten (Beispiel: Spielpläne aller Zürcher Kinos)

Vektoren: erste Anwendung

Das Sieb des Erathostenes

- 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 *Vektor*.

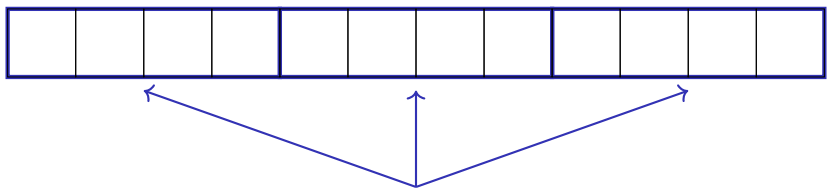
Sieb des Eratosthenes mit Vektoren

```
#include <iostream>
#include <vector> // standard containers with vector 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;
}
```

Speicherlayout eines Vektors

Ein Vektor belegt einen *zusammenhängenden* Speicherbereich

Beispiel: Ein Vektor mit 3 Elementen vom Type `T`



Speichersegmente für jeweils einen Wert vom Typ `T`

(`T` belegt z.B. 4 Bytes)

Wahlfreier Zugriff (Random Access)

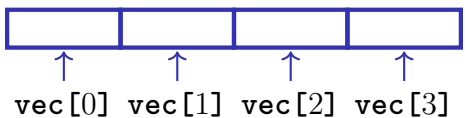
Gegeben

- Vektor **`vec`** mit **`T`**-Elementen
- **`int`**-Ausdruck **`exp`** mit Wert $i \geq 0$

Dann ist der Ausdruck

```
vec[exp]
```

- ein *L-Wert* vom Type **`T`**
- der sich auf das i-te Element von **`vec`** bezieht (Zählung ab 0!)



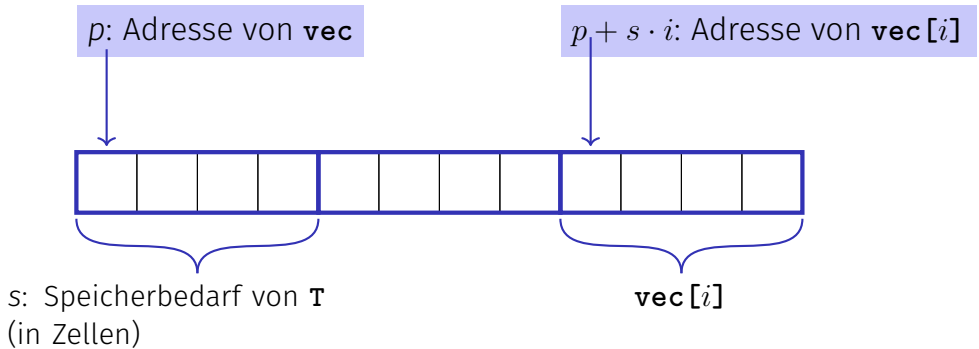
Wahlfreier Zugriff (Random Access)

```
vec[exp]
```

- Der Wert i von **`exp`** heisst *Index*
- **`[]`** ist der *Index-Operator* (auch *Subskript-Operator*)

Wahlfreier Zugriff (Random Access)

Wahlfreier Zugriff ist sehr effizient:



Vektorinitialisierung

- `std::vector<int> vec(5);`
  Die 5 Elemente von `vec` werden mit Nullen initialisiert
- `std::vector<int> vec(5, 2);`
  Die 5 Elemente von `vec` werden mit 2 initialisiert
- `std::vector<int> vec{4, 3, 5, 2, 1};`
  Der Vektor wird mit einer *Initialisierungsliste* initialisiert
- `std::vector<int> vec;`
  Ein leerer Vektor wird initialisiert

Achtung

Der Zugriff auf Elemente ausserhalb der gültigen Grenzen eines Vektors führt zu *undefiniertem Verhalten*

```
std::vector vec(10);
for (unsigned int i = 0; i <= 10; ++i)
  vec[i] = 30; // Laufzeit-Fehler: Zugriff auf vec[10]
```

Achtung

Prüfung der Indexgrenzen

Bei Verwendung des Indexoperators auf einem Vektor ist es die alleinige **Verantwortung des Programmierers**, die Gültigkeit aller Elementzugriffe zu prüfen.

Vektoren bieten viel Funktionalität

Hier ein paar Beispielfunktionen, weitere folgen später in der Vorlesung.

```
std::vector<int> v(10);
std::cout << v.at(10);
  // Zugriff mit Index-Check → Laufzeitfehler
  // Ideal für Hausaufgaben


v.push_back(-1); // -1 is appended (added at end)
std::cout << v.size(); // outputs 11
std::cout << v.at(10); // outputs -1
```

13. Zeichen und Texte I

Zeichen und Texte, ASCII, UTF-8, Caesar-Code

Zeichen und Texte

- Texte haben wir schon gesehen:

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

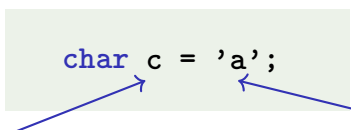
String-Literal

- Können wir auch „richtig“ mit Texten arbeiten? Ja!

Zeichen:	Wert des fundamentalen Typs **`char`**
Text:	**`std::string`** ≈ Vektor von **`char`** Elementen

Der Typ `char` („character“)

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



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`** $\longrightarrow$ **`(unsigned) int`**

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

  `'A', 'B', ... , 'Z'` $\longrightarrow 65, 66, ..., 90$
  `'a', 'b', ... , 'z'` $\longrightarrow 97, 98, ..., 122$
  `'0', '1', ... , '9'` $\longrightarrow 48, 49, ..., 57$

- Wird von allen gängigen Computersystemen unterstützt
- Erlaubt Arithmetik über Zeichen

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

Erweiterung von ASCII: Unicode, UTF-8

- Internationalisierung von Software $\Rightarrow$ grosse Zeichensätze nötig. Heute daher üblich:
  - Zeichensatz *Unicode*: 150 Schriftsysteme, ca. 137'000 Zeichen
  - Codierungsstandard *UTF-8*: Abbildung Zeichen $\leftrightarrow$ Zahlen
- UTF-8 ist eine *Multibyte-Codierung*: Häufig genutzte Zeichen (z.B. lat. Alphabet) belegen nur ein Byte, andere Zeichen bis zu vier
- Länge einer Zeichen-Bytefolge wird dabei durch Bitmuster codiert

Nutzbare Bits	Bitmuster
7	`0xxxxxxx`
11	`110xxxxx 10xxxxxx`
16	`1110xxxx 10xxxxxx 10xxxxxx`
21	`11110xxx 10xxxxxx 10xxxxxx 10xxxxxx`

Einige Unicode-Zeichen in UTF-8

Symbol	Codierung (jeweils 16 Bit)
ئی	11101111 10101111 10111001
	11100010 10011000 10100000
	11100010 10011000 10000011
	11100010 10011000 10011001
A	01000001

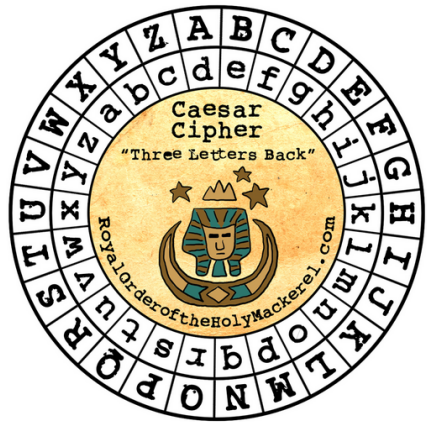
P.S.: Suchen Sie mal nach **`apple "unicode of death"`** P.S.: Unicode & UTF-8 sind nicht klausurrelevant

http://t-a-w.blogspot.ch/2008/12/funny-characters-in-unicode.html

Caesar-Code

Ersetze jedes druckbare Zeichen in einem Text durch seinen Vor-Vor-Vorgänger.

' '	(32)	→	'|'	(124)
'!'	(33)	→	'}'	(125)
		...		
'D'	(68)	→	'A'	(65)
'E'	(69)	→	'B'	(66)
		...		
~	(126)	→	'{'	(123)



Caesar-Code: `shift`-Funktion

```
// PRE:  divisor > 0
// POST: return the remainder of dividend / divisor
//       with 0 <= result < divisor
int mod(int dividend, int divisor);

// POST: if c is one of the 95 printable ASCII characters, c is
//       cyclically shifted s printable characters to the right
char shift(char c, int s) {
    if (c >= 32 && c <= 126) { // c is printable
      c = 32 + mod(c - 32 + s,95);
    }

    return c;
}
```

"**- 32**" transforms interval $[32, 126]$ to $[0, 94]$
"**mod(**x**, 95)**" computes $x \bmod 95$ in $[0, 94]$
"**32 +**" transforms $[0, 94]$ back to $[32, 126]$

Caesar-Code: `caesar`-Funktion

```
// POST: Each character read from std::cin was shifted cyclically
//       by s characters and afterwards written to std::cout
void caesar(int s) {
  std::cin >> std::noskipws; // #include <ios>

  char next;
  while (std::cin >> next) {
    std::cout << shift(next, s);
  }
}
```

Konversion nach **`bool`**: liefert *false* genau dann, wenn die Eingabe leer ist

Verschiebung druckbarer Zeichen um **`s`**

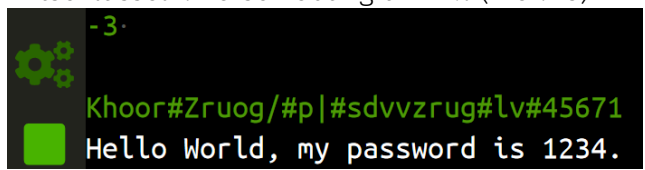
Caesar-Code:

Hauptprogramm

```
int main() {
  int s;
  std::cin >> s;

  // Shift input by s
  caesar(s);

  return 0;
}
```

Verschlüsseln: Verschiebung um n (hier: 3)

Entschlüsseln: Verschiebung um $-n$ (hier: -3)



Caesar-Code: Generalisierung

```
void caesar(int s) {
  std::cin >> std::noskipws;

  char next;
  while (std::cin >> next) {
    std::cout << shift(next, s);
  }
}
```

- Momentan nur von **`std::cin`** nach **`std::cout`**

- Besser: von beliebiger Zeichenquelle (Konsole, Datei, …) zu beliebiger Zeichensenke (Konsole, …)