

# 11. Referenztypen

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

## Swap!

```
// POST: values of x and y are 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! 😊
}
```

378

379

## 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& Gelesen als „T-Referenz“

Zugrundeliegender Typ

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

380

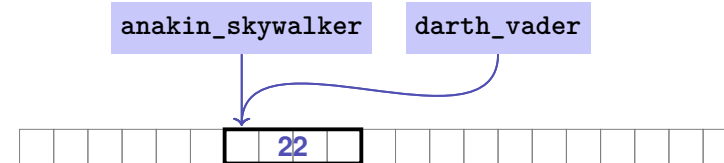
381

## 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
```



382

383

## Referenztypen: Initialisierung & Zuweisung

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

- Eine 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).
- Zuweisung an die Referenz erfolgt an das **Objekt** hinter dem Alias.

384

## 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: Das Literal 5 hat keine Adresse
```

385

## Pass by Reference

Referenztypen erlauben Funktionen, die Werte ihrer Aufrufargumente zu ändern:

```
void increment (int& i) ← Initialisierung der formalen Argumente
{ // i wird Alias des Aufrufarguments
  ++i;
}
...
int j = 5;
increment (j);
std::cout << j << "\n"; // 6
```



386

## Pass by Reference

Formales Argument hat Referenztyp:

⇒ **Pass by Reference**

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

387

## Pass by Value

Formales Argument hat keinen Referenztyp:

⇒ **Pass by Value**

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

388

## 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]
```



389

## 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;
}
```

390

## Return by Value / Reference

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

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

Exakt die Semantik des Prä-Inkrement

391

## 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
```

392

## 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.

393

## 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)

394

## Wann const T& ?

### Regel

Argumenttyp `const T &` (pass by *read-only* reference) wird aus Effizienzgründen anstatt `T` (pass by value) benutzt, wenn der Typ `T` grossen Speicherbedarf hat. Für fundamentale Typen (`int`, `double`,...) lohnt es sich aber nicht.

Beispiele folgen später in der Vorlesung

395

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

Der Schummelversuch wird vom Compiler erkannt

396

## 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 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
```

397

## 12. Vektoren und Strings I

Vektoren, Sieb des Eratosthenes, Speicherlayout, Iteration, Zeichen und Texte, ASCII, UTF-8, Caesar-Code

### 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)

398

399

### Vektoren: erste Anwendung

Das Sieb des Eratosthenes

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

|   |   |              |   |              |   |              |              |               |    |               |    |               |               |               |    |               |    |               |               |               |    |
|---|---|--------------|---|--------------|---|--------------|--------------|---------------|----|---------------|----|---------------|---------------|---------------|----|---------------|----|---------------|---------------|---------------|----|
| 2 | 3 | <del>4</del> | 5 | <del>6</del> | 7 | <del>8</del> | <del>9</del> | <del>10</del> | 11 | <del>12</del> | 13 | <del>14</del> | <del>15</del> | <del>16</del> | 17 | <del>18</del> | 19 | <del>20</del> | <del>21</del> | <del>22</del> | 23 |
|---|---|--------------|---|--------------|---|--------------|--------------|---------------|----|---------------|----|---------------|---------------|---------------|----|---------------|----|---------------|---------------|---------------|----|

Am Ende des Streichungsprozesses bleiben nur die Primzahlen übrig.

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

400

### 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);

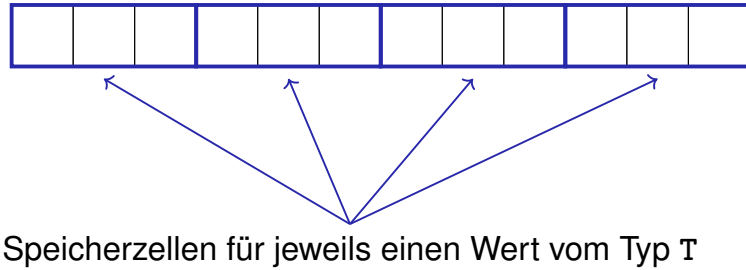
    // computation and output
    std::cout << "Prime numbers in {2,...," << n-1 << "}: \n";
    for (unsigned int i = 2; i < n; ++i)
        if (!crossed_out[i]) { // i is prime
            std::cout << i << " ";
            // cross out all proper multiples of i
            for (unsigned int m = 2*i; m < n; m += i)
                crossed_out[m] = true;
        }
    std::cout << "\n";
    return 0;
}
```

403

## Speicherlayout eines Vektors

- Ein Vector belegt einen *zusammenhängenden* Speicherbereich

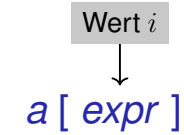
Beispiel: ein Vektor mit 4 Elementen



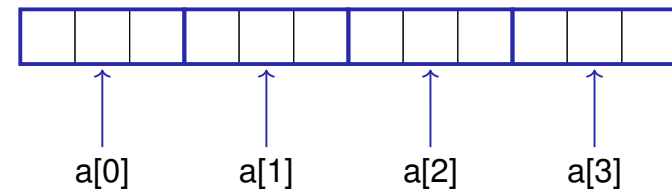
404

## Wahlfreier Zugriff (Random Access)

Der L-Wert



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



405

## Wahlfreier Zugriff (Random Access)

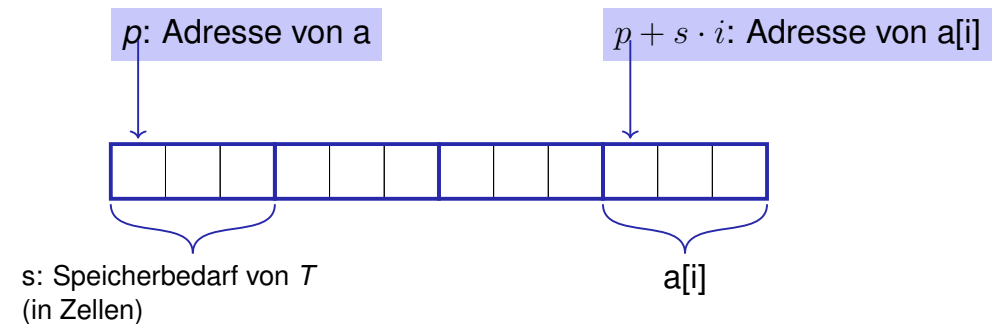
$a [ \text{expr} ]$

Der Wert  $i$  von  $\text{expr}$  heisst *Index*.  
[]: Subskript-Operator

406

## Wahlfreier Zugriff (Random Access)

- Wahlfreier Zugriff ist sehr effizient:



407

## Vektor Initialisierung

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

408

## Achtung

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

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

409

## Achtung

### Prüfung der Indexgrenzen

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

410

## Vektoren bieten Komfort

```
std::vector<int> v (10);  
v.at(5) = 3; // with bound check  
v.push_back(8); // 8 is appended  
std::vector<int> w = v; // w is initialized with v  
int sz = v.size(); // sz = 11
```

413



## 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`  $\approx$  Vektor von `char` Elementen

## 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

414

415

## 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, ..., 127} oder {0, ..., 255}

## Der ASCII-Code

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

Zeichen  $\rightarrow$  {0, ..., 127}  
'A', 'B', ... , 'Z'  $\rightarrow$  65, 66, ..., 90  
'a', 'b', ... , 'z'  $\rightarrow$  97, 98, ..., 122  
'0', '1', ... , '9'  $\rightarrow$  48, 49, ..., 57

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

416

417

## Erweiterung von ASCII: UTF-8

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

| 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 ein UTF8 Zeichen beginnt.

## Einige Zeichen in UTF-8

| Symbol | Codierung (jeweils 16 Bit)  |
|--------|-----------------------------|
| ن      | 11101111 10101111 10111001  |
| ☠      | 11100010 10011000 10100000  |
| ☃      | 11100010 10011000 10000011  |
| ☪      | 11100010 10011000 100111001 |
| A      | 01000001                    |

P.S.: Suchen Sie mal nach apple "unicode of death"

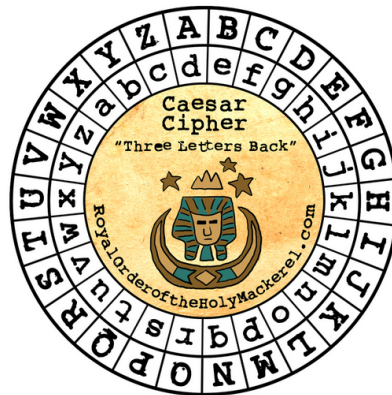
418

http://t-a-w.blogspot.ch/2008/12/funny-characters-in-unicode.html  
419

## Caesar-Code

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

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



420

## 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 printable
        c = 32 + mod(c - 32 + s, 95);
    }
    return c;
}
```

"- 32" transforms interval [32, 126] to [0, 94]  
 "32 +" transforms interval [0, 94] back to [32, 126]  
 mod(x,95) is the representative of  $x \pmod{95}$  in interval [0, 94]

421

## 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.

Verschiebt nur druckbare Zeichen.

422

## Caesar-Code:

## Hauptprogramm

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

    // Shift input by s
    caesar(s);

    return 0;
}
```

Verschlüsseln: Verschiebung um  $n$  (hier: 3)

```
3
Hello World, my password is 1234.
Khoor#Zruog/#p|#sdvvzrug#lv#45671
```

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

```
-3
Khoor#Zruog/#p|#sdvvzrug#lv#45671
Hello World, my password is 1234.
```

423

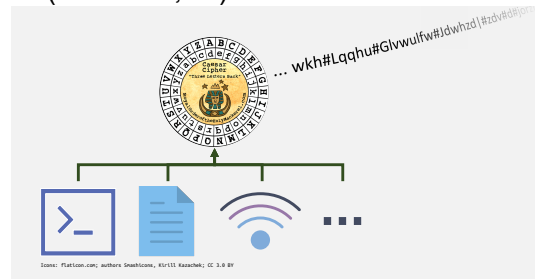
## Caesar-Code: Generalisierung

```
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, ...)



424

## Caesar-Code: Generalisierung

```
void caesar(std::istream& in,
            std::ostream& out,
            int s) {

    in >> std::noskipws;

    char next;
    while (in >> next) {
        out << shift(next, s);
    }
}
```

- `std::istream/std::ostream` ist ein *generischer Eingabe-/Ausgabestrom* an chars
- Aufruf der Funktion erfolgt mit *spezifischen Strömen*, z.B.: Konsole (`std::cin/cout`), Dateien (`std::i/ofstream`), Strings (`std::i/ostringstream`)

425

## Caesar-Code: Generalisierung, Beispiel 1

```
#include <iostream>
...

// in void main():
caesar(std::cin, std::cout, s);
```

Aufruf der generischen caesar-Funktion: Von `std::cin` nach `std::cout`

426

## Caesar-Code: Generalisierung, Beispiel 2

```
#include <iostream>
#include <fstream>
...

// in void main():
std::string from_file_name = ...; // Name of file to read from
std::string to_file_name = ...; // Name of file to write to
std::ifstream from(from_file_name); // Input file stream
std::ofstream to(to_file_name); // Output file stream

caesar(from, to, s);
```

Aufruf der generischen caesar-Funktion: Von Datei zu Datei

427

## Caesar-Code: Generalisierung, Beispiel 3

```
#include <iostream>
#include <sstream>
...

// in void main():
std::string plaintext = "My password is 1234";
std::istringstream from(plaintext);

caesar(from, std::cout, s);
```

Aufruf der generischen caesar-Funktion: Von einem String nach `std::cout`

428