Programmieren / Algorithmen & Datenstrukturen

Grundlagen (ii)

Prof. Dr. Skroch

Universitatea
BABEŞ-BOLYAI

Grundlagen (ii)

Inhalt.

- Bitoperationen
- Zeiger, Datenfelder, Listen
- Beziehungen zwischen Klassen
- Ein- und Ausgabe
- Überblick: C++ Standardbibliothek

Binäres Rechnen

Zur Erinnerung: Dezimal-, Binär- und Hexadezimalsystem.

- Dezimalsystem, zehn Symbole ("Basis 10")
  { 0,1,2,3,4,5,6,7,8,9 }

  10^0 10^1 10^2 10^3 ... $3 \cdot 10^0 + 1 \cdot 10^1 + 2 \cdot 10^2 + 0 \cdot 10^3 =$

  3 1 2 0 ... = **213**

- Binärsystem, zwei Symbole ("Basis 2")
  { 0,1 }

  2^0 2^1 2^2 2^3 2^4 2^5 2^6 2^7 ...

  1 0 1 0 1 0 1 1 ...

  $2^0 + 2^2 + 2^4 + 2^6 + 2^7 =$ **11010101_{bin}**

- Hexadezimalsystem, sechzehn Symbole ("Basis 16"):
  { 0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f }

  16^0 16^1 16^2 ...

  5 d 0 ...

  $5 \cdot 16^0 + 13 \cdot 16^1 =$ **$d5_{hex}$**

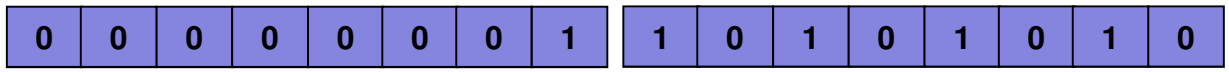
Binäres Rechnen

Zur Erinnerung: Dezimal-, Binär- und Hexadezimalsystem.

- Umwandlung einer Dezimalzahl in eine Binärzahl
  - Möglich durch wiederholte $\text{modulo } 2$ Bildung.
- Beispiel

426	$: 2 = 213$	Rest	0_{bin}
213	$: 2 = 106$	Rest	1_{bin}
106	$: 2 = 53$	Rest	0_{bin}
53	$: 2 = 26$	Rest	1_{bin}
26	$: 2 = 13$	Rest	0_{bin}
13	$: 2 = 6$	Rest	1_{bin}
6	$: 2 = 3$	Rest	0_{bin}
3	$: 2 = 1$	Rest	1_{bin}
1	$: 2 = 0$	Rest	1_{bin}

Ergebnis:



- Soll in eine andere Basis als 2 umgerechnet werden, wählt man statt 2 als Modul die Zielbasis, also z.B. für Hexadezimalzahlen die 16.

Binäres Rechnen

Zur Erinnerung: Dezimal-, Binär- und Hexadezimalsystem.

- Umwandlung einer Binärzahl in eine Dezimalzahl
  - Möglich durch einfaches Multiplizieren.
- Beispiel

$$110101010_{bin} =$$
$$= 0\cdot2^0 + 1\cdot2^1 + 0\cdot2^2 + 1\cdot2^3 + 0\cdot2^4 + 1\cdot2^5 + 0\cdot2^6 + 1\cdot2^7 + 1\cdot2^8 =$$
$$= 1 + 8 + 32 + 128 + 256 = 426$$

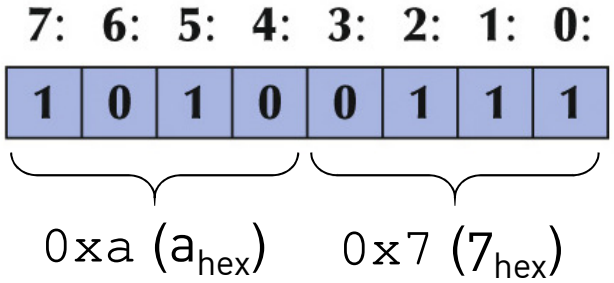
- Soll aus einer anderen Basis als 2 umgerechnet werden, geht man einfach entsprechend der anderen Basis vor, also z.B. für Hexadezimalzahlen:

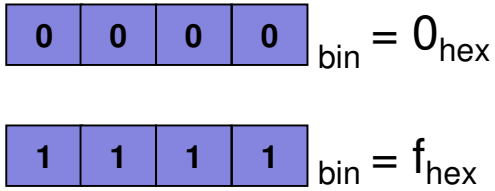
$$1aa_{hex} =$$
$$= a_{hex}\cdot16^0 + a_{hex}\cdot16^1 + 1_{hex}\cdot16^2 =$$
$$= 10\cdot16^0 + 10\cdot16^1 + 1\cdot16^2 =$$
$$= 10 + 160 + 256 = 426$$

Binäres Rechnen

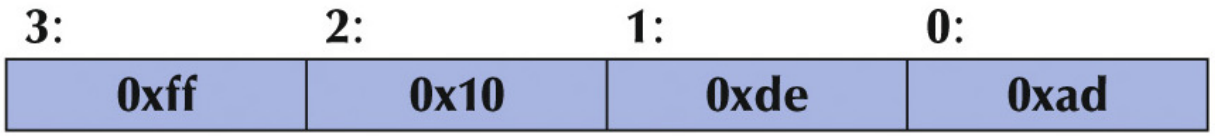
Bits und Bytes.

- Ein *Byte* kann man sich als die Folge von 8 Bit vorstellen.





- Ein *Word* kann man sich als die Folge von 4 Byte vorstellen.



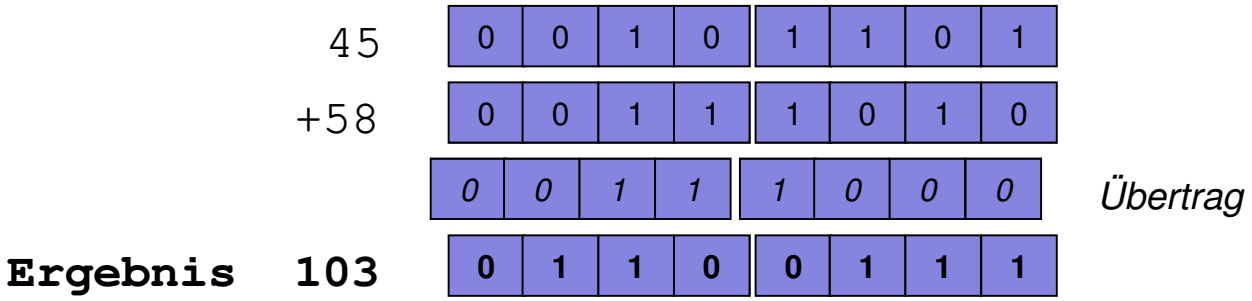
- Per Konvention nummeriert man in C++ die Bits eines Byte und die Bytes eines Word von rechts (niederwertigstes Ende) nach links (höchstwertiges Ende).

Bildquelle: Stroustrup.

Binäres Rechnen und Bitoperationen

Positive ganze Zahlen und ganzzahlige Addition.

- Integer-Addition (durch einfache Hardware-Schaltlogik sehr schnell).
  - Die Bits werden stellenweise beginnend mit dem kleinstwertigen Bit addiert.
  - Entsteht kein Übertrag wird mit dem nächsthöheren Bit weiter gemacht.
  - Entsteht der Übertrag `1`, geht er auf das nächsthöhere Bit über.
  - `0+0 = 0`, kein Übertrag.
  - `1+0 = 1`, kein Übertrag.
  - `0+1 = 1`, kein Übertrag.
  - `1+1 = 0`, Übertrag.



Binäres Rechnen und Bitoperationen

Negative ganze Zahlen.

- Positive ganze Zahlen werden einfach so, wie sie im Binärsystem dargestellt sind, mit einer Binärstelle pro Bit gespeichert.
- Bei negativen Zahlen ist diese Darstellungsweise aufgrund des Vorzeichens ungeeignet.
- Die Ansätze der Informatik zur Darstellung von negativen Zahlen im Computer laufen darauf hinaus, dass ein bestimmtes Bit das Vorzeichen angibt.
  - Der "naive" Ansatz, einfach ein Bit, z.B. das höchstwertige in dem für den Typ verfügbaren Speicherbereich, dafür herzunehmen, hat gravierende Nachteile (z.B. ist die Null nicht mehr eindeutig).
- Heutige Rechnerarchitekturen stellen negative ganze Zahlen daher meist im sog. 2-Komplement (auch Zweierkomplement genannt) dar.
  - Das b-Komplement einer Zahl x wird als $w_b(x)$ bezeichnet.

Binäres Rechnen und Bitoperationen

10-Komplement und negative ganze Dezimalzahlen.

- 10-Komplement und Dezimalzahlen.
  - $w_{10}(z)$ bezeichnet das 10-Komplement einer Dezimalzahl z.
  - $w_{10}(z)$ kann man sich als Ergänzung von z zu 10^n vorstellen, wobei n die Anzahl der Stellen von z ist.
    - Das 10-Komplement von 42?
    - $w_{10}(42) = 10^2 - 42 = 58$
- Dezimalsubtraktion kann als Addition des 10-Komplements dargestellt werden (Prinzip vieler mechanischer Rechenmaschinen), Beispiel:

+ 4'328 – 2'009	+ 4'328 + **7'991**	**$w_{10}(2009) = 10^4 - 2009 = 7991$**
+ 2'319	+ 12'319	*Ein Übertrag an der höchsten Stelle bedeutet, dass das Ergebnis positiv ist (der Übertrag wird verworfen).*
+ 1'234 – 3'579	+ 1'234 + **6'421**	**$w_{10}(3579) = 10^4 - 3579 = 6421$**
– 2'345	+ _7'655	*Kein Übertrag an der höchsten Stelle bedeutet, dass das Ergebnis negativ ist (d.h. die Zahl ist ein 10-Komplement).* **$w_{10}(7655) = 10^4 - 7655 = 2345$**

Das **10-Komplement** einer Dezimalzahl x kann auch berechnet werden als $w_{10}(x) = w_9(x) + 1$.

Das **9-Komplement** der Dezimalzahl x kann durch Ergänzung jeder einzelnen Ziffer aus x zur 9 gebildet werden.

Dezimalzahl	2 0 0 9	
Rest zur 9	7 9 9 0	(9-Komplement)
Plus 1	7 9 9 1	(10-Komplement)

Damit ist keinerlei Subtraktion i.e.S. mehr erforderlich.

Binäres Rechnen und Bitoperationen

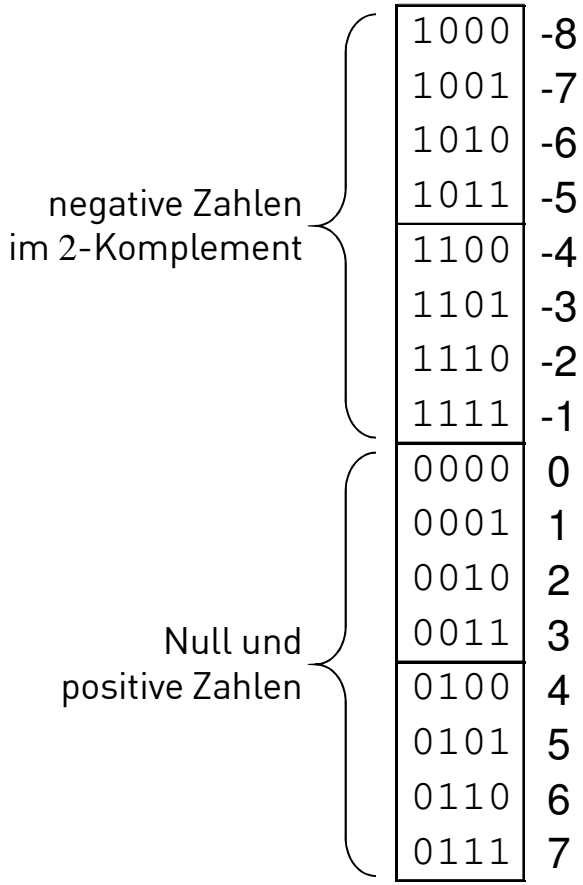
2-Komplement und negative ganze Binärzahlen.

- Zahlen im Computer sind Binärzahlen und $b = 2$.
- Binäre negative Ganzzahlen werden im Computer normalerweise in der 2-Komplement Darstellung gespeichert.
- Vorteile
  - Keine zwei Nullen (pos. und neg.) mehr im Wertebereich.
  - Trotzdem Vorzeichenbit vorhanden.
  - 2-Komplement sehr schnell bestimmbar (einfache Hardware-Schaltlogik).
  - Mit negativen Ganzzahlen im 2-Komplement kann der Computer *Addition* und *Subtraktion* effizient durchführen.
    - Subtraktion kann auf 2-Komplementbildung und Addition zurückgeführt werden (ähnlich wie bei Dezimalzahlen).
    - Keine Fallunterscheidungen (Vorzeichen der Summanden) sind mehr erforderlich.

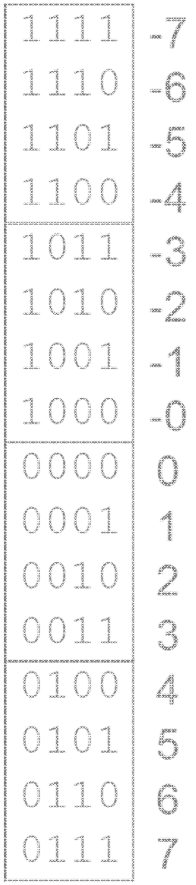
Binäres Rechnen und Bitoperationen

2-Komplement vs. Darstellung mit Betrag und Vorzeichenbit.

- Darstellung der negativen ganzen Binärzahlen im 2-Komplement der positiven ganzen Binärzahlen.



Zum Vergleich rechts: Darstellung mit Vorzeichenbit und Betrag.



Binäres Rechnen und Bitoperationen

2-Komplement und Subtraktion mit ganzen Zahlen.

- Berechnung des 2-Komplements $w_2(x)$ mit Binärzahlen x.
  1) Die Binärdarstellung der Zahl x bitweise invertieren (1-Komplement).
  2) Die 1 addieren.

  Sehen Sie sich nochmals das 10- / 9-Komplement bei Dezimalzahlen an.
- Beispiel $x = 5$
  - Ermittlung des 2-Komplements von 0101_{bin} (d.h. **+5**)
    - bitweise invertieren ergibt das 1-Komplement 1010
    - 0001 addieren ergibt: **1011** (d.h. **-5**)
- Beispiel $x = -6$
  - das 2-Komplement von 1010_{bin} (d.h. **-6**)
    - bitweise invertieren ergibt das 1-Komplement 0101
    - 0001 addieren ergibt: **0110** (d.h. **+6**)

Binäres Rechnen und Bitoperationen

2-Komplement und Subtraktion mit ganzen Zahlen.

- Darstellung der Subtraktion $a–b$ durch 2-Komplement und Addition.
  1) 2-Komplement von b bilden.
  2) Addition von a und dem 2-Komplement von b durchführen.
     a) Tritt hier ein Übertrag an der höchstwertigen Stelle auf, liegt das Ergebnis direkt vor (d.h. es ist positiv), der Übertrag wird ignoriert.
     b) Tritt hier kein solcher Übertrag auf, liegt das Ergebnis als 2-Komplement vor (d.h. es ist negativ).
- Beispiel (8 Bit Wertebereich): $a–b = 58–45$, d.h. $a = 58$ und $b = 45$
  - Die Binärdarstellung von b ist `0010 1101`
    bitweise invertiert: `1101 0010`, die `0000 0001` addiert: `1101 0011`
  - `0011 1010 + 1101 0011 = 1 0000 1101`
    Fall 2a) : 13
- Beispiel (8 Bit Wertebereich): $a–b = 27–42$, d.h. $a = 27$ und $b = 42$
  - Die Binärdarstellung von b ist `0010 1010`
    bitweise invertiert `1101 0101`, die `0000 0001` addiert: `1101 0110`
  - `0001 1011 + 1101 0110 = _ 1111 0001`
    Fall 2b): -15

Binäres Rechnen und Bitoperationen

Beispiele 2-Komplement-Subtraktion.

4-17		7	6	5	4	3	2	1	0				
		128	64	32	16	8	4	2	1				
1	**17**				**1**	**0**	**0**	**0**	**1**	(plus) 17			
2	**~17**				0	1	1	1	0	bitweise Inversion von Zeile 1			
3	**1**				0	0	0	0	1	Addition von 1 zu Zeile 2			
4	**(~17)+1**				0	1	1	1	1	ergibt 2-Komplement von 17, also "minus 17"			
5	**4**				0	0	1	0	0	"plus 4"			
6	**4+(~17)+1**				**1**	**0**	**0**	**1**	**1**	Ergebnis: minus 13 (in 2-Komplement Darstellung)			

58-45		7	6	5	4	3	2	1	0				
		128	64	32	16	8	4	2	1				
1	**45**			**1**	**0**	**1**	**1**	**0**	**1**	(plus) 45			
2	**~45**			0	1	0	0	1	0	bitweise Inversion von Zeile 1			
3	**1**			0	0	0	0	0	1	Addition von 1 zu Zeile 2			
4	**(~45)+1**			0	1	0	0	1	1	ergibt 2-Komplement von 45, also "minus 45"			
5	**58**			1	1	1	0	1	0	"plus 58"			
6	**58+(~45)+1**		1	**0**	**0**	**1**	**1**	**0**	**1**	Ergebnis: plus 13			

Binäres Rechnen und Bitoperationen

Vorzeichen, `signed` und `unsigned` in C++.

- Was stimmt nicht mit dem folgenden Quellcode:

```
void loop() {
  unsigned char max {256};
  for( signed char c{0}; c<max; ++c ) cout << int{c} << ' ';
}
```

  - Führen Sie das Beispiel aus und erklären Sie es.

- Was passiert hier:

```
unsigned int ui{-1};
int si{ui};
cout << "ui ist " << ui << ", si ist " << si;
```

  - Führen Sie auch dieses Beispiel aus und erklären Sie es.

- Folgerung: vermeiden Sie wenn möglich, in C++ Typen mit und ohne Vorzeichen zu vermischen.

Binäres Rechnen und Bitoperationen

Bitoperationen in C++.

- Ganzzahlige Multiplikation wird in Rechnerarchitekturen meist auf Bitverschiebungen und Additionen zurückgeführt.

`operator<<` `a<<b` verschiebt `a` bitweise um `b` Positionen nach links (zum höherwertigen Ende), wobei rechts Nullen nachrücken.
`5 << 3` ist **40**

`operator>>` `a>>b` verschiebt `a` bitweise um `b` Positionen nach rechts (zum niederwertigen Ende), wobei links Nullen nachrücken.
`5 >> 3` ist **0**

`operator&` Ergibt an einer Bitposition 1, wenn beide Operanden an dieser Bitposition 1 haben. "AND"
`5 & 3` ist **1**

`operator|` Ergibt an einer Bitposition 1, wenn mindestens einer der beiden Operanden an dieser Bitposition 1 hat. "OR"
`5 | 3` ist **7**

`operator^` Ergibt an einer Bitposition 1, wenn genau einer der beiden Operanden an dieser Bitposition 1 hat. "XOR"
`5 ^ 3` ist **6**

`operator~` Negiert jede Bitposition "NOT"
`~5` ist **–6**

Binäres Rechnen und Bitoperationen

Bitoperationen in C++.

- Mit dem nachfolgenden Quellcode kann man ganze Zahlen in ihre Bitdarstellungen verwandeln:

```
int x{0};
while( cin >> x )
   cout << dec << x << "=="
        << hex << "0x" << x << "=="
        << bitset<8*sizeof(int)>(x) << '\n'; // mehr zu sizeof() folgt…
```

- Hier kommt der `bitset`-Typ aus der StdLib zum Einsatz.
  - `bitset` Container unterstützen auch den indizierten Zugriff auf die Bits.
  - Der Operator << (für Bitshift *und* Stromausgabe überladen) funktioniert mit `bitset` Objekten direkt wie oben gezeigt.
- Wie kann man das nur mit elementaren Sprachmitteln ausgeben?

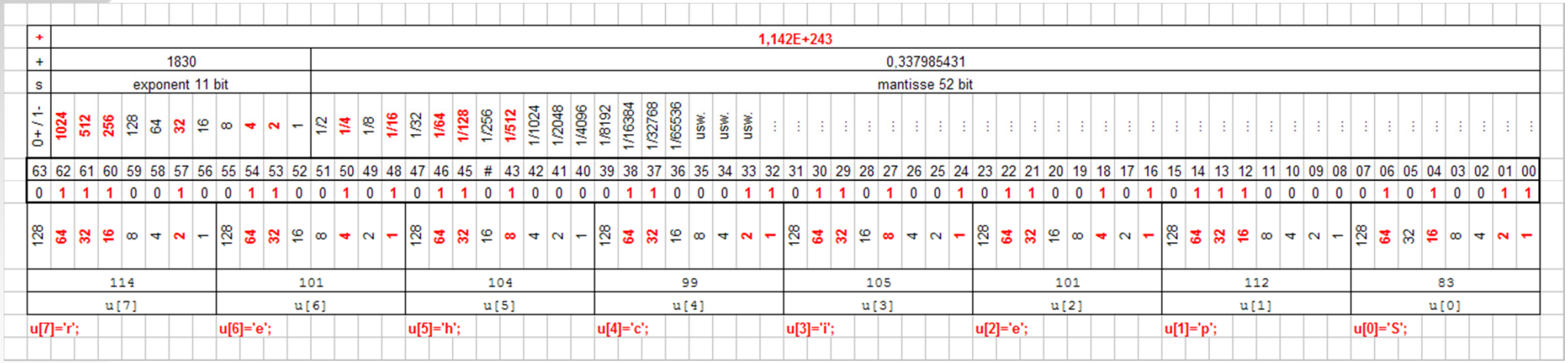
```
/* z.B. so, nach K+R: */
unsigned long int x{42};
for( int p{31}; p >= 0; --p ) {
   cout << ( (x>>p) & ~(~0<<1) );
   if( (p%8==0)&&(p!=0) ) cout << '|'; // sieht besser aus
}
```

Binäres Rechnen und Bitoperationen

Darstellung von Gleitkommazahlen.

- Binäre Gleitkomma-Operationen sind nicht trivial.
- Moderne Computer haben für Gleitkomma-Operationen eigene Hardware (Floating Point Unit).
- Darstellung von Gleitkommazahlen (IEEE 754) am Beispiel:

```
double d {1.142e243};  unsigned char u[] {"Speicher"};
```



$(1+man) * 2^{(exp-1023)}$

Einige Beispielfragen

Bitoperationen.

- Was ist fade_{hex}?
- Wie lautet die Dezimaldarstellung von `0010 1101 1101 1110`?
- Wie lautet die Binärdarstellung von $11'742$?
- Wie viele Ziffern hat das Hexadezimalsystem? Wie lauten diese?
- Wie lautet die Hexadezimaldarstellung von $47'871$?
- Wie erklären Sie sich, dass die Gleichung 1+1=10 stimmt?
- Was ist das 10-Komplement von 100?
- Berechnen Sie drei 2-Komplemente:
  `0010 1100`, `0011 0100` und `1111 1101`
- Führen Sie die Subtraktion als 2-Komplement Addition mit binären Darstellungen für die folgenden vier Beispiele aus:
  $4\text{-}17$, $2\text{-}512$, $4321\text{-}1234$, $64\text{-}31$
- Welchen Wert hat der Ausdruck `16<<3`? Welchen Wert hat `5>>2`? Erklären Sie Schritt für Schritt, warum das so ist.

Einige Beispielfragen

Bitoperationen.

- Was macht dieser Quellcode:

```
using uint = unsigned int;
using uchar = unsigned char;

int main () {
  const uint m {883U};
  uchar c {'\0'};
  while( cin >> c ) {
    cout << uchar{ c^m } << endl;
    cout << uchar{ (c^m)^m } << endl << endl;
  }
  return 0;
}
```

Grundlagen (ii)

Inhalt.

- Bitoperationen
- Zeiger, Datenfelder, Listen
- Beziehungen zwischen Klassen
- Ein- und Ausgabe
- Überblick: C++ Standardbibliothek

Speicher, Adressen und Zeiger

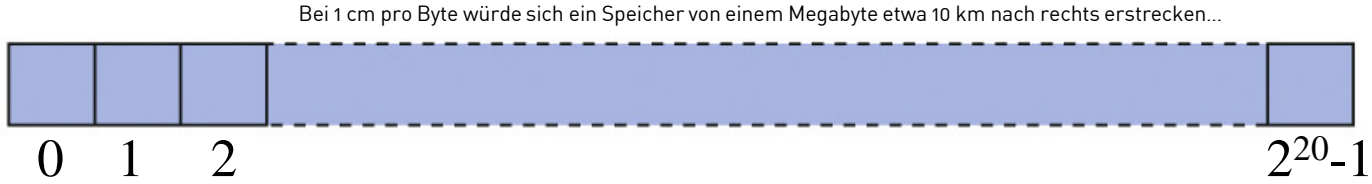
Zeiger arbeiten direkt mit der Speicherhardware.

- Wir haben den Komfort von Typen wie `vector` oder `string` aus der StdLib kennen gelernt (und uns vielleicht schon daran gewöhnt).
  - Wir haben uns noch nicht näher damit auseinander gesetzt, wie beispielsweise ein Objekt vom `vector` Typ zur Laufzeit mehr und weniger Elemente beinhalten kann, oder wie ein `string` Objekt zusätzliche Zeichen aufnimmt.
  - Die von uns schon eingesetzten Mittel wie `vector::push_back()`, die in der StdLib implementiert sind, haben wir noch nicht wirklich verstanden.
- Wir wollen so etwas mit elementaren Sprachmitteln programmieren.
  - Von der Hardware *direkt* unterstützt werden nur Bytefolgen im Speicher.
- Warum sollte man das tun und nicht einfach weiter `vector` & Co. nehmen?
  - Um Quellcode zu verstehen, der direkt auf Speicher zugreift.
  - Um in der Lage zu sein, bei Bedarf auch selbst neue, derartige Typen nach eigenen Ideen zu bauen.
  - Um Programmiersprachen effizient in der vollen Einsatzbreite zu verwenden, was manchmal nicht ohne Zeiger und Datenfelder geht.
  - *Um zu verstehen was passiert, wenn Software auf Hardware trifft.*

Speicher, Adressen und Zeiger

Adressen kann man sich als fortlaufende Nummern vorstellen, die den Arbeitsspeicher des Computers durchnummerieren.

- Der Arbeitsspeicher des Computers besteht aus einer Folge von Bytes.
  - Wir können die Bytes durchnummerieren, beginnend bei 0 und endend mit dem letzten Byte.
  - Eine Nummer, die die Position eines Bytes im Speicher angibt, nennen wir *Adresse*.
- So können Sie sich einen Speicherbereich von einem MB vorstellen:



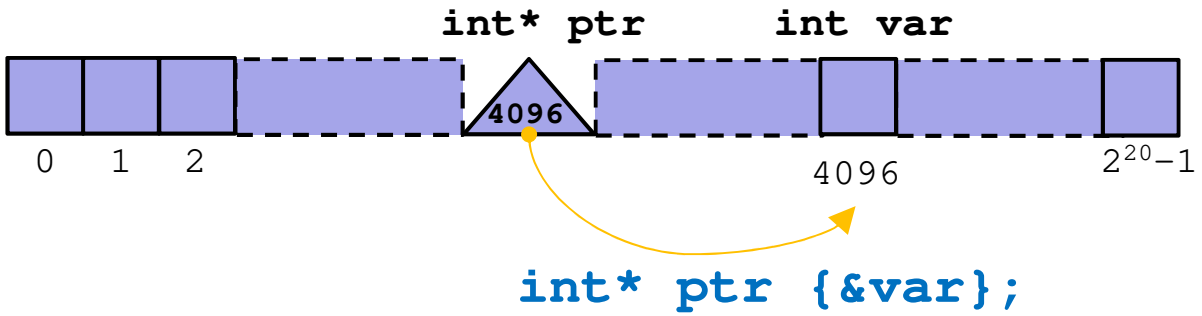
- Alles, was wir im Speicher ablegen, hat eine Adresse.
  - `int var {17};`
  - Diese Anweisung reserviert irgendwo im Arbeitsspeicher, vielleicht an der Adresse 4096, ein "`int`-großes" Speicherstück für den Namen `var` und schreibt dort ein Bitmuster, welches als `int` den Wert 17 darstellt.
- Speicheradressen, wie etwa die 4096 im Beispiel, können in C++ auch als Werte behandelt, d.h. als Objekte gespeichert und bearbeitet, werden.

Bildquelle: Stroustrup.

Speicher, Adressen und Zeiger

Für jeden Typ `T` kann man ein Objekt vom Zeigertyp "Zeiger auf `T`" erzeugen.

- Der Typ eines Objekts, das die Adresse eines `int` Werts enthält, heißt "Zeiger auf `int`" oder "`int`-Zeiger", Notation des Typnamens: **`int*`**.



  - Der Operator `&` liefert die Adresse eines Objekts.

- Die Zeigertypen sind integrierte Typen von C++.
  (D.h. auch: Zeigerobjekte sind wie bei allen integrierten Typen üblicherweise *nicht* von selbst initialisiert.)
- Für *jeden* Typ gibt es auch einen Zeigertyp, der bei einer Deklaration mit dem Typ-Suffix `*` gekennzeichnet wird.
  - z.B. ist `char* pc` ein "Zeiger auf `char`" namens `pc`,
    oder `char** argv` ein "Zeiger auf `char`-Zeiger" namens `argv`.
  - Englisch pointer, bei Zeigernamen in "ungarischer Notation" wird daher oft die Abkürzung `ptr` oder `p` verwendet.

Bildquelle: Stroustrup.

Speicher, Adressen und Zeiger

Für jeden Typ `T` kann man ein Objekt vom Zeigertyp "Zeiger auf `T`" erzeugen.

- Das Zeigerkonstrukt ist erfahrungsgemäß eines der herausforderndsten Sprachmittel beim Programmieren.
- Im Zusammenhang mit Zeigerobjekten werden hauptsächlich zwei Operationen verwendet.
  - *Adressoperation* (Operatorsymbol **&**), die auf Objekte (von beliebigem Typ) angewandt wird und deren Speicheradresse liefert.
  - *Inhaltsoperation* (Operatorsymbol *****), die auf Zeigerobjekte angewandt wird und den Wert liefert, der unter der im Zeiger gespeicherten Adresse abgelegt ist.
- Beispiele:
  - `double d {0.5}; double* pd {&d}; cout << pd << ' ' << *pd;`
  - `int i {42};  int* pi {&i};  cout << pi << ' ' << *pi;`
- Zeiger können mit dem Operator * auch L-Werte sein.
  - D.h. der Inhaltsoperator ist auch auf der linken Seite einer Zuweisung erlaubt, es wird "durch den Zeiger hindurch" auf die Speicherstelle zugewiesen, auf die er zeigt.
  - Auch Umwandlungen wie `int` nach `double` sind dabei erlaubt.

```
*pi = 42;    *pd = 2.718;    *pd = *pi;
```

Speicher, Adressen und Zeiger

Für jeden Typ `T` kann man ein Objekt vom Zeigertyp "Zeiger auf `T`" erzeugen.

- Zeigerwerte sind zwar ganze Zahlen, aber nicht vom `int` Typ.
  - "Worauf zeigt eine `int` Variable?" ist keine korrekte Frage, denn `int` Werte haben nicht die spezielle Eigenschaft von Zeigerwerten, Speicheradressen zu sein.
  - Zeigertypen stellen Operationen für Speicheradressen zur Verfügung (sog. *Zeigerarithmetik*), während `int` Typen u.a. arithmetische Operationen für ganze Zahlen aus der Mathematik bereit stellen.

```
int i {pi}; /* Fehler */      pi = 64; /* Fehler */
```

- Zeiger auf den einen Typ sind keine Zeiger auf einen anderen Typ, Zeigertypen umzuwandeln ist *grundsätzlich* nicht typsicher.

```
pd = pi; /* Fehler */         pi = pd; /* Fehler */
```

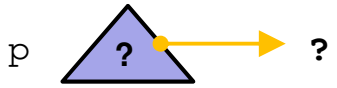
- Beachten Sie auch, dass der Inhaltsoperator das selbe Symbol verwendet wie der Namenssuffix für Zeiger-Deklarationen.

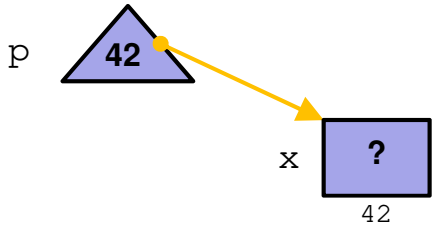
```
double d {};
double* pd {&d}; // Namenssuffix * mit Typnamen
*pd = 3.21;      // Präfix-Inhaltsoperator * mit Namen von Zeigerobjekten
```

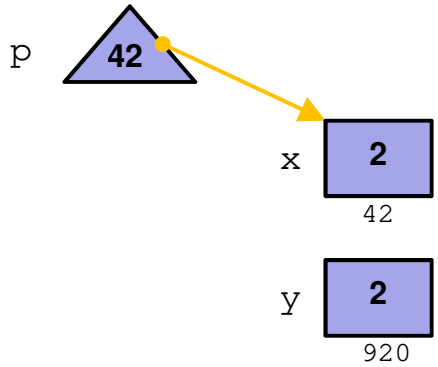
Zeiger

Einige Beispiele zur Veranschaulichung.

C++	Erklärung
`int* p;`	Die definierte Variable namens `p` ist ein Zeiger auf `int`. Da nicht initialisiert wurde, ist ihr Wert (d.h. die Adresse) unbestimmt.
`int x;` `p = &x;`	Die Adresse der definierten `int`-Variablen namens `x` wird in `p` gespeichert ("`p` zeigt auf `x`").
`x = 2;` `int y {*p};`	`p` zeigt noch auf `x`. Die `int`-Variable `y` wird definiert und mit dem Wert `2` (dem Inhalt der Adresse, auf die `p` zeigt) initialisiert.



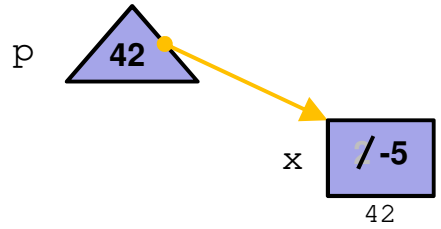


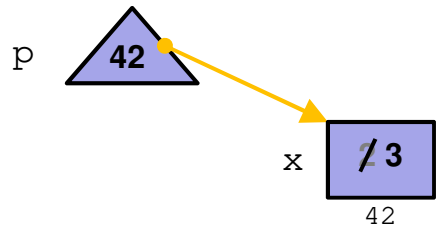


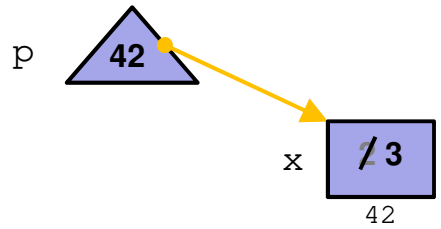
Zeiger

Einige Beispiele zur Veranschaulichung.

C++	Erklärung	

`int x {2};`
`int* p {&x};`
`*p = -5;`
	Zuerst wird die Speicherstelle, auf die `p` zeigt, ermittelt (Adresse von `x`), dann der Wert an dieser Speicherstelle auf `-5` gesetzt.	

`int x {2};`
`int* p {&x};`
`*p += 1;`
	Inkrementiert den Wert der Variablen `x`.	

`int x {2};`
`int* p {&x};`
`(*p)++;`
	Inkrementiert den Wert der Variablen `x`. Die Klammern sind wegen des Vorrangs der Operatoren erforderlich.	

Zeiger

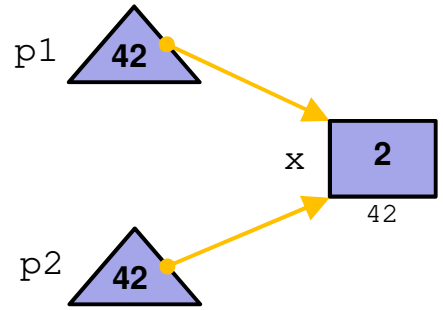
Einige Beispiele zur Veranschaulichung.

C++

```
int x {2};
int* p1 {&x};
int* p2 {p1};
```

Erklärung

Der Variablen namens `p2` wird der Wert der Variablen namens `p1` zugewiesen; dadurch zeigt `p2` auf dasselbe Ziel wie `p1`.

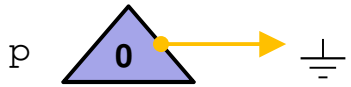



```
int* ip1, x1;
int * ip2, x2;
int *ip3, x3;
```

Äquivalente Ausdrücke, da * sich nur auf den direkt folgenden Namen bezieht (d.h. `x1`, `x2`, `x3` sind hier keine Zeiger).

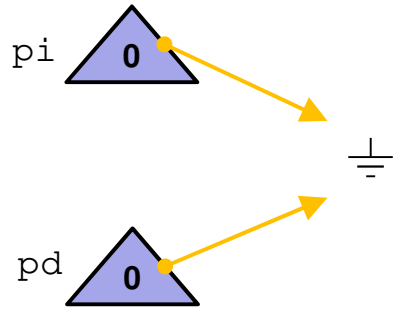
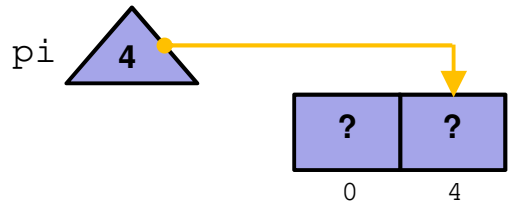
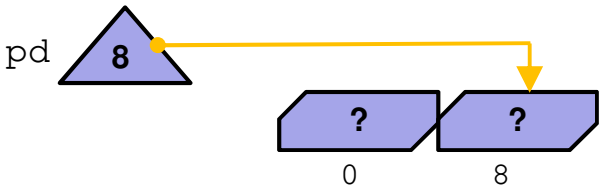
```
int* p {nullptr};
```

`p` ist der *Nullzeiger* auf `int`. Der Wert von `p` ist keine gültige Adresse, `p` ist damit aber klar initialisiert.



Zeiger

Einige Beispiele zur Veranschaulichung.

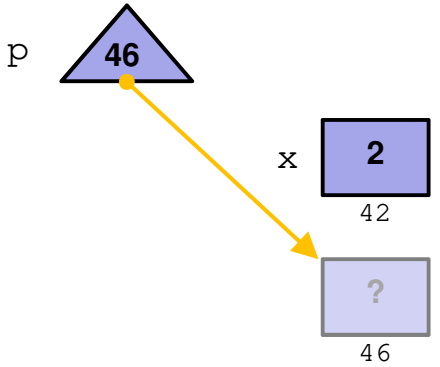
C++	Erklärung	
`int* pi {};` `double* pd {};`	`pi` ist der Nullzeiger auf `int` und `pd` ist der Nullzeiger auf `double`.	
`pi++;`	`pi` zeigt "ein `int` weiter" in den Speicher (Zeigerarithmetik).	
`pd++;`	`pd` zeigt "ein `double` weiter" in den Speicher (Zeigerarithmetik).	

Zeiger

Einige Beispiele zur Veranschaulichung.

C++	Erklärung

`int x {2};`
`int* p {&x};`
`(*p++)++;`
Vermeiden Sie solche Konstruktionen.
	Fehler: wendet den (mehrfach überladenen) Operator ++ zuerst auf die in `p` gespeicherte Adresse an (womit p "ein `int` weiter" in den Speicher zeigt), betrachtet dann das Bitmuster an dieser Speicherstelle als `int` und manipuliert die Bits entsprechend der `int`-Operation "inkrementieren".



- Inhaltsoperator `*` und Adressoperator `&` haben höheren Rang als arithmetische Operatoren (aber geringeren Rang als ++ und −−).
  - `*(p+1)` bzw. `*p++` meint den `int`-Wert, der "ein `int`-großes Stück weiter" im Speicher liegt, und C++ prüft *nicht*, ob das Sinn macht.
  - `(*p)+1` bzw. `*p+1` bzw. `(*p)++` und bedeutet, dass der `int`-Wert an der Stelle, auf die der Zeiger zeigt, gelesen und zu `1` addiert wird.

Zeiger und `const`

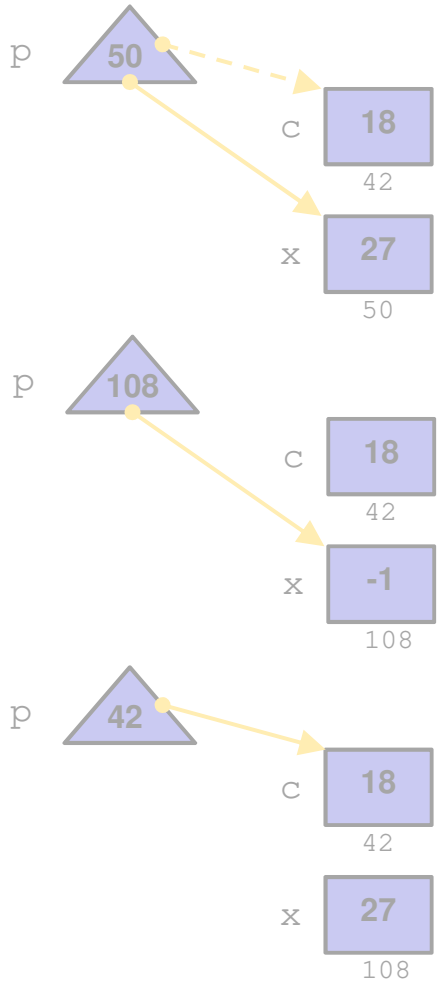
Einige Beispiele zur Veranschaulichung.

C++	Erklärung

`const int c{18};`
`const int x{27};`
`const int* p{&c};`
`p = &x;`
	`p` ist Zeiger auf eine `int`-Konstante. Der Zeiger kann danach woanders hin zeigen.

`const int c{18};`
`int x{27};`
`int* const p{&x};`
`*p = -1;`
	`p` ist konstanter Zeiger auf eine `int`-Variable. Der Zeiger kann danach nicht verändert werden, aber das Objekt, auf das er zeigt.

`const int c{18};`
`int x{27};`
`const int* const p{&c};`
	`p` ist konstanter Zeiger auf eine `int`-Konstante. Der Zeiger kann danach nicht verändert werden, das Objekt, auf das er zeigt auch nicht.



Zeiger und `sizeof()`

Wie viel Speicher belegt beispielsweise ein `int`-Wert oder ein `int*`-Wert?

- Der `sizeof()` Operator kann auf Typnamen und Ausdrücke angewandt werden.
  - Für Typen liefert er die Größe eines Objekts des angegebenen Typs.
  - Für Ausdrücke liefert er die Größe des Typs, von dem das Ergebnis des Ausdrucks ist.
- Der `sizeof()` Operator gibt die Anzahl Bytes zurück.
  - Genauer: eine positive ganze Zahl in Vielfachen von `sizeof( char )`.
  - `sizeof( char )` ist in C++ per Definition gleich 1.
  - Zweckmäßigerweise wird ein C++ Compiler für `char` genau ein Byte reservieren.
- Es gibt *keine* Garantie dafür, dass ein Typ unter jeder Implementierung von C++ dieselbe Größe hat.
  - `sizeof( int )` liefert auf vielen PCs den Wert 4 und d.h. normalerweise ist damit ein `int` 32 Bit groß.
- Beispiele

```
cout << sizeof( char ) << ' ' << sizeof( 'a' );
cout << sizeof( int )  << ' ' << sizeof( 3*3 );
cout << sizeof( int* ) << ' ' << sizeof( pi );

vector<double> vd{ 163.0, 2.718, (1.0/3.0), 4.2 };
cout << sizeof( vd ); // zaehlt die Elemente nicht mit
```

Sehen Sie sich nochmals die Grundlagen zum `vector` Container an (L1, G3).

Zeiger

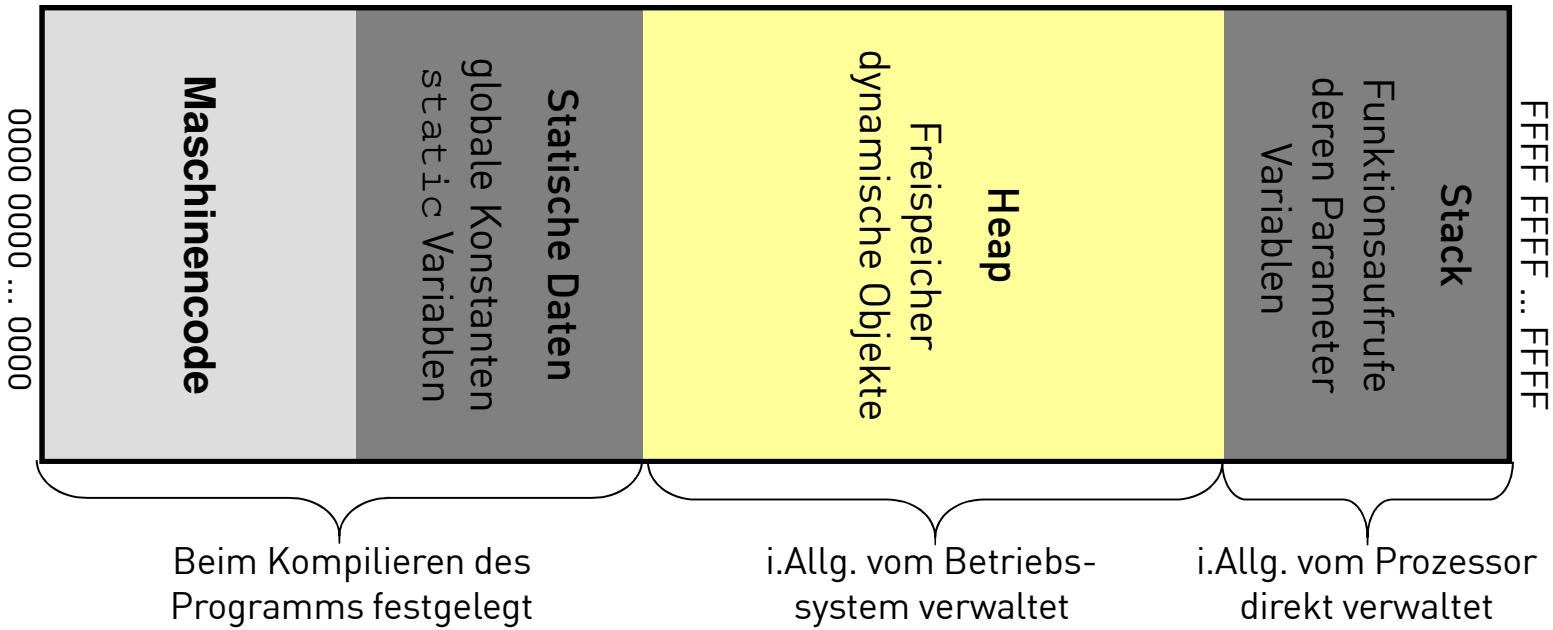
Sie haben gesehen: Zeiger operieren nah an der Hardware, oft kein freundliches Umfeld für Programmierer.

- Zeigeroperationen können aus unterschiedlichen Gründen notwendig werden.
  - Beispielsweise weil Namen von Objekten unbekannt sind oder nicht existieren, das Programm aber auf die Objekte zugreifen will.
- Wie genau der Compiler Speicher reserviert ist abhängig von der Implementierung des Compilers, daher ist es ein Fehler beispielsweise:
  - einen `char`-Zeiger an einen `int`-Zeiger zuzuweisen,
    - `char` ist meist kleiner als `int`, `int` Werte könnten dann über den Zeiger in ein nur `char`-großes Stück des Speichers geschrieben werden,
  - die `int`-Operation "inkrementieren" mit dem Speicherbereich auszuführen, der auf den Speicherbereich folgt, auf den der `int`-Zeiger zeigt,
    - der Speicherbereich, der auf den für eine Variable reservierten Speicherbereich folgt, kann alle möglichen Inhalte haben (die Typsicherheit würde drastisch verletzt).
- Derartige Anweisungen hätten sehr wahrscheinlich zur Folge, dass Speicherbereiche überschrieben werden, die nicht überschrieben werden sollten.
  - Solche und ähnliche Fehler kommen bei der Programmierung mit Zeigern ungewollt und regelmäßig vor, und sind meist nicht einfach zu entdecken.

Freispeicher

Was tun, wenn sich zur Laufzeit des Programms zusätzlicher Speicherbedarf ergibt?

- Wir haben bis jetzt darauf verzichtet, uns genauer anzusehen, wie `vector` und andere dynamische Datenstrukturen implementiert sind.
  - Aus didaktischen Gründen, um so schnell wie möglich sinnhafte Programme schreiben zu können.
- Wir wollen jetzt verstehen, wie z.B. `vector` weiteren Speicher für seine zusätzlichen Elemente bekommt.
- Ein C++ Programm benutzt den Speicher in etwa so:



Speicherarten

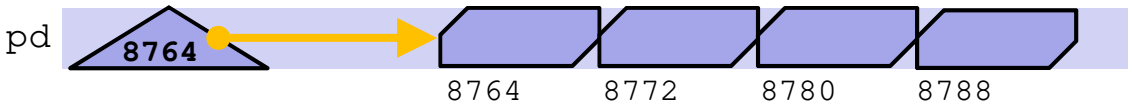
Statischer Speicher, Stackspeicher und Heapspeicher

- *Statischer Speicher* wird vom Linker reserviert und besteht so lange, wie das Programm ausgeführt wird.
- *Stackspeicher* wird beim Aufruf einer Funktion reserviert und wieder freigegeben, wenn aus der Funktion zurückgekehrt wird (*automatisch*).
  - Der Stackspeicher wird oft durch den Prozessor direkt verwaltet.
- *Heapspeicher ("Freispeicher")* kann im Programm durch spezielle Syntax ausdrücklich reserviert und wieder freigegeben werden (*dynamisch*).
  - Der Heap- bzw. Freispeicher wird i. Allg. vom Betriebssystem verwaltet.
- Stack und Heap sind oft so organisiert, dass sie an beiden Enden des adressierbaren Speichers liegen und aufeinander zuwachsen.
  - Wenn sie sich treffen, ist der Speicher erschöpft.

Zeiger, Datenfelder und `new`

Speicher im Heap reservieren (Allokation).

- In C++ gibt es den Operator `new` zur Anforderung von Speicher im Heap
  `double* pd { new double[4] };`
  - Mit dieser Anweisung fordert C++ das Betriebssystem zur Laufzeit auf
    - dem Programm im Heap lückenlos fortlaufenden Speicherplatz für vier `double` Werte bereit zu stellen (man spricht auch von Allokation oder Speicherzuteilung)
    - und die Adresse des ersten `double` Objekts zurückzuliefern.
  - Damit wir auf diesen Speicher zugreifen können müssen wir wissen, wo er ist.
  - Wir initialisieren also einen `double`-Zeiger namens `pd` mit der zurückgegebenen Adresse.



- Ein funktionaler Hauptgrund für die Reservierung von Speicher im Heap ist das Erzeugen von Objekten innerhalb einer Funktion, die sich an den Aufrufer zurückgeben lassen, denn

  dieser Speicher folgt *nicht* den C++ Gültigkeitsregeln.

Bildquelle: Stroustrup.

Zeiger, Datenfelder und `new`

Speicher im Heap reservieren (Allokation).

- Der Operator `new` gibt einen Zeiger auf das bereitgestellte Objekt zurück.
  - Ist das zur Verfügung gestellte Objekt vom Typ `T`, so ist der von `new` zurückgegebene Zeiger vom Typ `T*`.
- Der Wert des Zeigers ist die Adresse des (ersten) Speicherbytes.
- `new` reserviert mit `[]` mehrere Objekte eines Typs auf einmal.
  - Man spricht von einem *Datenfeld (Array)* von Objekten.
  - Es wird ein Zeiger auf das erste dieser Objekte zurückgegeben.
- Die Anzahl der Objekte, die angelegt werden sollen, kann durch eine Variable angegeben werden.
  - Dadurch wird erst zur Laufzeit festgelegt, für wie viele Objekte Speicher reserviert werden soll.
- Der Zeiger weiß *nicht*, auf wie viele Objekte er zeigt.

```
int* pi { new int[4] };
double* pd { new double[n] };
```

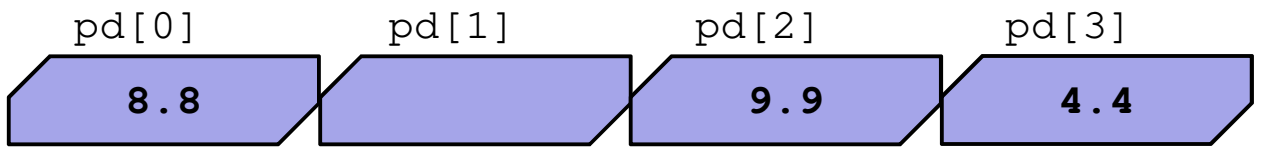
- Der Zeiger weiß aber immer, auf welchen Typ er zeigt.

Zugriff über Zeiger

Neben dem Inhaltsoperator `*` kann man auch den Indexoperator `[]` auf Zeiger anwenden, `*(a+i)` ist äquivalent zu `a[i]` (Adressarithmetik).

- In Datenfeldern ist `*p` identisch mit `p[0]` (der Indexoperator beginnt bei `0`).

```
double* pd { new double[4] };
double x { *pd };   // oder: double x { pd[0] };
double y { pd[2] }; // oder: double y { *(pd+2) };
*pd = 8.8; // schreibt in das (erste) Objekt, auf das pd zeigt
pd[2] = 9.9; // schreibt in das dritte Objekt, auf das pd zeigt
pd[3] = 4.4;
double z { pd[3] };
```



- Wird der Operator `[]` auf einen Zeiger angewandt, dann betrachtet er den Speicher als eine fortlaufende Folge (Datenfeld) von Objekten (des Typs, der in der Zeigerdeklaration angegeben wurde), und den Zeiger als Verweis auf das erste Objekt im Datenfeld.
- Es gibt ***keinerlei* Überprüfung** der Zugriffe.

Bildquelle: Stroustrup.

Zeiger und Typen

Zeiger auf Objekte von unterschiedlichem Typ sind selbst auch unterschiedliche Typen.

- Zuweisungen zwischen unterschiedlichen Zeigertypen sind *nicht* erlaubt.
  1) Verletzung der Typsicherheit (vgl. auch weiter vorne).
  2) Zeigerarithmetik: der Inhaltsoperator `*` und der Indexoperator `[]` benötigen die Größe des Elementtyps, um berechnen zu können, wo im Speicher sich ein bestimmtes Element befindet (Zeigerarithmetik).
     - Der Wert `pi[2]` liegt beispielsweise im Speicher genau zwei `int`-Blöcke, d.h. zweimal `sizeof(int)`, hinter dem Wert `pi[0]`.
     - Der Wert `*(pd+4)` liegt genau vier `double`-Blöcke, d.h. viermal `sizeof(double)` hinter dem Wert `*pd`.
- Obwohl (beispielsweise) `int` Werte in `double` Werte umgewandelt werden können, sind die entsprechenden Zeiger inkompatibel.

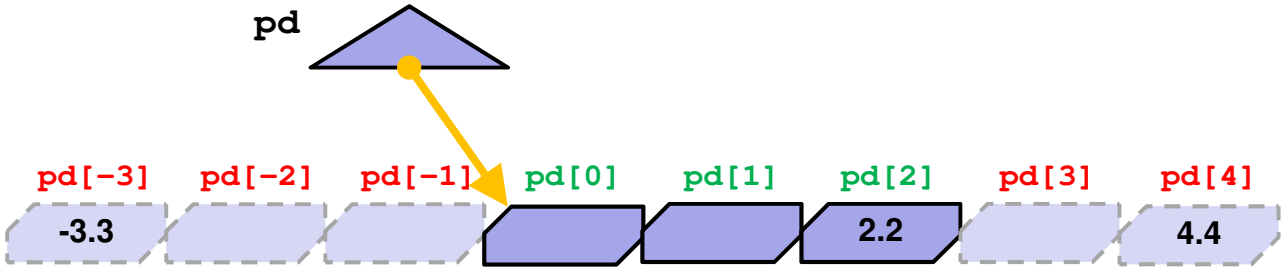
```
double* pd { new int[4] };    // Fehler, falscher Typ
int* pi { new double[4] };    // Fehler, falscher Typ
```

Zeiger, Datenfelder und Bereiche

Der Zeiger weiß (nur/immerhin), auf welchen Typ er zeigt, er weiß aber *nicht*, auf wie viele Elemente er zeigt.

```
double* pd { new double[3] };
pd[2] = 2.2; *(pd+4) = 4.4; pd[-3] = -3.3;
```

- Gibt es für `pd` ein drittes Element `pd[2]`? Ein fünftes Element `pd[4]`?
  - Der Compiler prüft es nicht, sogar der Zugriff auf z.B. `pd[-3]` ist möglich.



- Man weiss nicht, wofür die Speicherbereiche verwendet werden, auf die z.B. `pd[-3]` und `pd[4]` verweisen.
  - Sie sind sicher nicht mehr Teil des Datenfelds.
  - Höchstwahrscheinlich gehören sie zu anderen Objekten, in die mit dem Quellcode wüst hineingeschrieben wird.

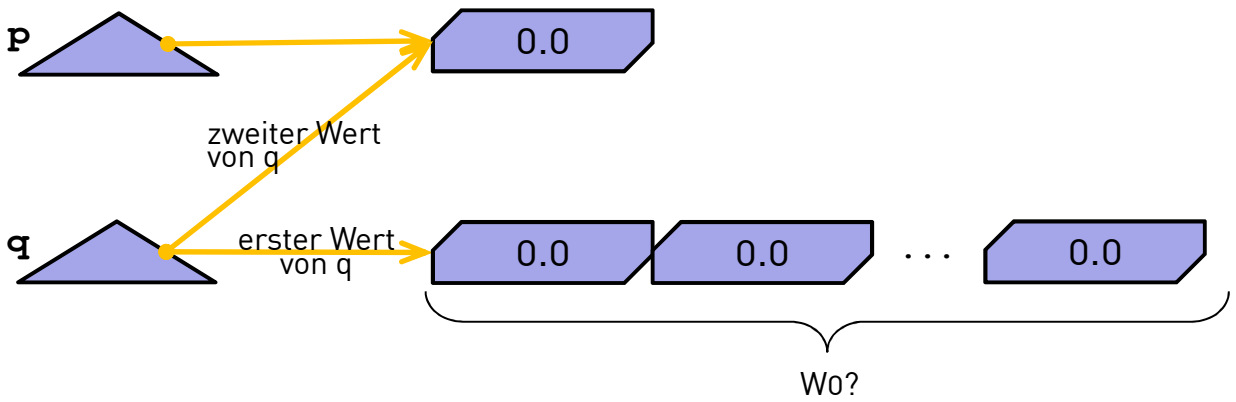
Bildquelle: Stroustrup.

Zeiger, Datenfelder und Bereiche

Der Zeiger weiß (nur/immerhin), auf welchen Typ er zeigt, er weiß aber *nicht*, auf wie viele Elemente er zeigt.

- Zuweisungsfehler zwischen Zeigern gleichen Typs werden vom Compiler auch *nicht* abgefangen.

```
double* p { new double{} };
double* q { new double[1000]{} };
q[699] = 7.7;              // OK
q = p;                     // beide sind Zeiger auf double
double d { q[699] };       // Fehler, Bereichsverletzung
```



Bildquelle: Stroustrup.

Initialisierung von Datenfeldern

Auch für Datenfelder (Arrays) gibt es die komfortable Syntax zur Initialisierung.

```
int ai[] {2,4,6,8};
// int-Datenfeld aus vier int
// Datenfelder koennen, so wie hier, auch im Stack liegen (kein new verwendet)

int ai2[100] {0,1,2,3};
// die letzten 96 int im Datenfeld (im Stack) sind mit 0 initialisiert

double ad[100] {};
// alle double im Datenfeld sind mit 0.0 initialisiert

ad - 1 + ( sizeof( ad ) / sizeof( *ad ) )
// Adresse des letzten double im Datenfeld
```

- Sonderfall `char`-Datenfelder

```
char ac[] { "Hallo C++!" };
// char-Datenfeld aus 11 char:
// der Compiler zaehlt die 10 char und haengt ein '\0' an
// identisch:
// char ac[] { 'H', 'a', 'l', 'l', 'o', ' ', 'C', '+', '+', '!', '\0' };
// char-Datenfelder sind sog. "C-Strings"
// sie sind nicht vom C++ Typ std::string
```

Initialisierung von Zeigern

Zeiger sollen initialisiert sein, ebenso die Objekte, auf die Sie verweisen.

```
double* p; // oh je, das geht gleich schief (nichts ist initialisiert)
double* p1 { new double }; // Zeiger OK, Zielwert nicht
double* p2 { new double{5.5} }; // Zeiger und Zielwert OK
double* p3 { new double[5] }; // Zeiger OK, Zielwerte nicht
double* p4 { new double[10]{} } // Zeiger und Zielwerte OK
*p = 7.5; // grober Fehler
```

- Die letzte Quellcodezeile wird das Bitmuster für den `double` Wert `7.5` *irgendwo* wüst in den Speicher schreiben, ein grober Fehler.
- Der Nullzeiger `double* p { nullptr }; // Nullzeiger`
  `// alternativ: double* p{};`
  - Zeigt "ins Leere", d.h. der Wert der Zeigervariablen ist zwar klar definiert, ist aber keine gültige Speicheradresse.
    - Zeiger können so rudimentär auf ihre Gültigkeit geprüft werden:
      `if( p == nullptr ) //…`
  - Wird u.a. verwendet, wenn ein neu definierter Zeiger nicht initialisiert werden kann… das kommt ab und zu vor, treffen Sie auf diese Situation, stellen Sie sich aber immer die Frage: warum einen Zeiger definieren, wenn es noch kein Objekt gibt, auf das er zeigen könnte?

Initialisierung von Zeigern

Zeiger sollen initialisiert sein, ebenso die Objekte, auf die Sie verweisen.

- Mit `new` reservierter Speicher ist für die *integrierten Typen* nicht initialisiert.
  - Man kann/muss also z.B. die `{}` Syntax zur Initialisierung verwenden.
- Für Objekte und Datenfelder mit *benutzerdefiniertem* `T` kann man den Wert des Standardkonstruktors vom Typ `T` hernehmen (falls dieser existiert).

```
T* pT1 { new T };       // Std-Konstruktor T() (falls ex.) wird verwendet
T* pT2 { new T[10] }; // Std-Konstruktor T() (falls ex.) fuer alle 10 Elemente
```

- Einzelne Objekte können mit der `()` Syntax oder der `{}` Syntax initialisiert werden.

```
T* pT3 { new T{7} };   // Initialisierung von pT3, ohne Std-Konstruktor
```

- Ggf. kann/muss man das Datenfeld explizit initialisieren:

```
double* pd { new double[5] { 12.4, 5.3, 15.2, 8.1, 24.0 } };
for( int i{}; i<5; ++i ) pd[i]=i;
```

  Jetzt zeigt `pd` auf das erste von fünf Objekten vom Typ `double`,
  die die Werte `0.0`, `1.0`, `2.0`, `3.0` und `4.0` haben.

- Zeiger *und* die Objekte, auf die sie zeigen, sind grundsätzlich immer zu initialisieren.

Bemerkung: Hat ihr Compiler einen "Debug"-Modus? Dieser Modus sorgt oft dafür, dass alles initialisiert wird, aber das letztlich ausgelieferte Programm wird gewöhnlich nicht im Debug-Modus übersetzt...

Zeiger und L-Referenzen

Zeiger und L-Referenzen werden beide mittels Speicheradressen implementiert, handhaben diese Adressen aber etwas unterschiedlich.

- Stellen Sie sich eine L-Referenz
  - entweder als alternativen Namen für ein Objekt vor,
  - oder als unveränderlichen und automatisch immer mit Inhaltsoperator versehenen Zeiger auf ein Objekt vor.
- Zum Unterschied von Zeigern und L-Referenzen
  - Eine Zuweisung an einen Zeiger ändert den Wert des Zeigers (nicht den Wert, auf den der Zeiger verweist).
  - Eine Zuweisung an eine L-Referenz ändert das Objekt, auf das sich die L-Referenz bezieht (nicht die L-Referenz selbst, die per Definition unveränderlich ist).
  - Um einen Zeiger zu erhalten wird normalerweise `new` oder `&` (der Adressoperator) benötigt.
  - Um auf ein Objekt zuzugreifen, auf das ein Zeiger verweist, wird der Inhaltsoperator `*` oder der Indexoperator `[]` verwendet.
  - Einmal initialisiert kann man eine L-Referenz nicht mehr auf ein anderes Objekt umlenken (d.h. wenn auf unterschiedliche Objekte zugegriffen werden soll sind L-Referenzen ungeeignet).

Zeiger und L-Referenzen

Zeiger und L-Referenzen werden beide mittels Speicheradressen implementiert, handhaben diese Adressen aber etwas unterschiedlich.

- Zeigertypen

```
int x {10};
int* p {&x};
*p = 7;
int x2 {*p};
int* p2 {&x2};
p2 = p;
p = &x2;
```

- L-Referenztypen

```
int y {10}; int yy {22};
int& r {y};
r = 7;
int y2 {r};
int& r2 {y2};
r2 = r;  // der Wert von y2 wird 7
//r = &y2;  // Fehler
r = yy;   // der Wert von y wird 22
```

- Die vorletzte Zeile der L-Referenzen
  - ist syntaktisch unzulässig,
  - und auch die Absicht des Programmierers ist schon falsch: L-Referenzen kann man nach der Initialisierung nicht mehr auf ein anderes Objekt verweisen lassen.
    - Sehen Sie sich zur weiteren Verdeutlichung die letzte Zeile der L-Referenzen an: der Name `r` bleibt *immer* ein Alias für das Objekt namens `y`.

Zeiger und L-Referenzen als Parameter

Zeiger und L-Referenzen können an Funktionen übergeben werden.

- Möchte man den Wert einer Variablen ändern, indem eine Funktion den neuen Wert auf der Grundlage eines alten Werts berechnet, so gibt es drei Alternativen:
  - Berechnen des neuen Werts und Rückgabe, **`int inc_val( int x ) { return x+1; }`**,
  - Übergeben einer L-Referenz, **`void inc_ref( int& r ) { ++r; }`**,
  - Übergeben eines Zeigers, Einsatz des Inhaltsoperators auf den Zeiger um den Wert an der erhaltenen Adresse zu erhalten, **`void inc_ptr( int* p ) { ++(*p); }`**.
  - Aufrufe:

```
int i {7};
inc_ptr(&i);
inc_ref(i);
i = inc_val(i);
```

```
int* pi {&i};
inc_ptr(pi);
inc_ref(*pi);
*pi = inc_val(*pi);
```

- Die zu bevorzugende Möglichkeit hängt von der Natur der Funktion ab, vorrangig sollte über Rückgabe des Ergebnisses oder über L-Referenzargumente nachgedacht werden.
- **Vorsicht:** im Gegensatz zur ansonsten üblichen call-by-value Übergabe-Mechanik wird C++ bei *Datenfeldern* in Funktionsargumenten keine Kopien sondern stets *Adressen* übergeben.

Speicher im Heap freigeben mit `delete`

Um die *Freigabe* des im Heap angeforderten Speichers muss sich der Programmierer selbst kümmern.

- **Mit `new` erzeugte Objekte unterliegen *nicht* den Gültigkeitsregeln, sie existieren so lange, bis sie wieder gelöscht werden (Operation `delete`) .**
- `delete` wird auf die Zeiger angewendet, die von `new` zurückgeliefert wurden und gibt den reservierten Speicher für zukünftige Allokationen wieder frei.
- Es gibt zwei Varianten, es muss die jeweils richtige verwendet werden.
  - **`delete`** gibt den mit `new` reservierten Speicher für ein einzelnes Objekt frei.
  - **`delete[]`** gibt den mit `new` reservierten Speicher für ein Datenfeld von Objekten frei.
  - Der Compiler prüft *nicht*, ob die richtige Variante verwendet wurde.
  - C++ prüft *nicht*, ob der Speicher noch benutzt wird.
  - Ein Objekt zweimal freizugeben ist ein grober Fehler.
  - Das `delete` auf Nullzeiger ist (zum Glück) wirkungslos.
- Speicherplatz für mit `new` erzeugte Objekte, auf die kein Zeiger mehr zeigt, ist verloren (d.h. aus dem Programm heraus unerreichbar), man spricht von einem sog. Speicherleck.
- Bei einem Zeiger, der noch auf Speicherplatz zeigt, der schon mit `delete` freigegeben wurde, spricht man von sog. hängenden Zeiger.

Speicher im Heap freigeben mit `delete`

Speicherlecks.

```
01 double* calc( int r_size, int max ) {
02    double* pd { new double[max]{} };
03    double* pd_res { new double[r_size]{} };
04    // verwende pd um bestimmte Ergebnisse zu berechnen
05    // lege die Ergebnisse in pd_res ab
06    return pd_res;
07 }

double* r { calc( 100, 1000 ) };
```

- Nach dem Aufruf von `calc()` ist ein für 1000 `double` Objekte reservierter Speicher unerreichbar – ein sog. *Speicherleck*.
  - Verbesserte Zeile 6: `delete[] pd; return pd_res;`
- (Erst) Wenn das Programm endet, wird der gesamte Speicher automatisch ans Betriebssystem zurück gegeben.
  - Keine "Garbage Collection" in C++.
    - Betriebssysteme? Eingebettete Systeme? Bibliotheken wie die StdLib?
    - Allgemein Programme, die längere Zeit ununterbrochen laufen?

Exkurs: Speicher im Heap automatisch freigeben

Sog. Freispeicherverwaltung.

- Freispeicherverwaltung ("Garbage Collection") ist in C++ 11 nicht normiert (im Gegensatz zu manchen anderen Programmiersprachen).
  - Man kann z.B. in Java mit "new" dynamisch Speicher anfordern, eine "delete" Anweisung gibt aber es nicht, dafür eine automatische Garbage Collection in einem eigenen Thread.
  - Es gibt für C++ annehmbare Bibliotheken zur Einbindung von Garbage Collection.

- Zwei bekanntete Techniken der Garbage Collection:
  - *Reference Counting*: Jede neue / entfernte Referenz auf einen dynamisch allokierten Speicher wird mitgezählt; ist der Zähler für ein Objekt auf null, so ist der Speicher nicht mehr erreichbar und wird freigegeben. Dieses Verfahren ist *ungeeignet*, denn es kann durch zirkuläre Referenzen passieren, dass manche Zähler nicht mehr null werden können.
  - *Mark-and-Sweep*: Ausgehend von den immer existierenden Objekten (globale wie `main`, statische) werden rekursiv alle von ihnen referenzierten Objekte markiert usw., bis keine weiteren Objekte markiert werden können (Mark-Phase). Alle nicht markierten Objekte werden dann gelöscht (Sweep-Phase).

Konstruktoren, Destruktoren, `new` und `delete`

In Klassen existiert komplementär zu den Konstruktoren ein Destruktor, der ggf. mit `new` reservierten Speicher freigeben muss.

- Der Destruktor ist eine typenlose Methode mit leerer Parameterliste, die nicht überladen werden kann und deren Name aus dem Klassenname mit einer davor gesetzen Tilde besteht, z.B. `~myVector()`.
- Die grundlegende Idee ist:
  - In C++ wird beim Anlegen eines Objekts ein Konstruktor aufgerufen, und beim Auflösen der Destruktor.
  - Ein Objekt fordert die Ressourcen, die es benötigt, im Konstruktor an.
  - Am Ende der Lebensdauer des Objekts gibt der Destruktor alle Ressourcen frei, die sich noch im Besitz des Objekts befinden.
- Der gebräuchlichste Anwendungsfall für die Definition von Destruktoren ist das Freigeben von Speicher, der in einem Konstruktor mit `new` reserviert wurde.
  - "*Ein 'nacktes' `new` außerhalb eines Konstruktors ist wie eine Einladung, das korrespondierende `delete` zu vergessen*" (Stroustrup)

Konstruktoren, Destruktoren, `new` und `delete`

In Klassen existiert komplementär zu den Konstruktoren ein Destruktor, der ggf. mit `new` reservierten Speicher freigeben muss.

- C++ Mechanismus:
  - Ein Konstruktor wird implizit aufgerufen, wenn ein Objekt erzeugt wird.
  - Ein Destruktor, falls vorhanden, wird implizit aufgerufen, wenn der Gültigkeitsbereich des Objekts verlassen wird.
- Daraus ergibt sich der Grundsatz: eine Klasse, die dynamisch Ressourcen (Speicher, ...) anfordert, macht dies (nur) in Konstruktoren und braucht dann auch einen entsprechenden Destruktor zur Freigabe der Ressourcen.
  - Wir erinnern uns an den `vector<T>` Typ, welcher zur Laufzeit zusätzliche Elemente aufnehmen kann, etwa mittels `push_back()`
- Zur Vermeidung von Komplikationen werden Destruktoren *nicht* direkt aufgerufen, sondern nur implizit, d.h. üblicherweise automatisch beim Verlassen des Gültigkeitsbereichs eines Objekts.

Konstruktoren, Destruktoren, `new` und `delete`

Möglicher Quellcode eines eigenen `myVector` Containers für `double` Elemente, der sich hierbei so wie `std::vector<double>` verhält.

```
class myVector {
    int sz;
    double* elem;
  public:
    myVector( ) : sz{ 0 }, elem{ nullptr } { }
    explicit myVector( int s )
      : sz{ s }, elem{ new double[s]{} } { }
    ~myVector( ) { delete[] elem; }
    int size( ) const { return sz; }
    double get( int n ) const { return elem[n]; } // sog. "Getter"
    void set( int n, double v ) { elem[n]=v; } // sog. "Setter"
};

myVector* f( int s ) {
   myVector mv( s );
   myVector* p { new myVector(s) };
   // fuelle *p
   return p;
}
```

Konstruktoren, Destruktoren, `new` und `delete`

Die Implementierung des `std::vector` Containertyps ist ein schönes Beispiel für das eben erklärte Prinzip.

- Wird ein `vector` Objekt mit dem Operator `new` im Heap angelegt ("nackt"),
  - reserviert der Operator `new` zuerst den Speicher für das `vector` Objekt
  - und ruft dann zur Initialisierung einen `vector` Konstruktor auf.
    - `vector` Konstruktoren reservieren den Speicher für die Elemente im Container, und initialisieren diese mittels Aufruf eines Elementtyp-Konstruktors (falls dieser existiert).
- Wird das `vector` Objekt mit dem Operator `delete` wieder aufgelöst,
  - ruft der Operator `delete` zuerst den `vector` Destruktor auf,
    - der `vector` Destruktor ruft zunächst für jedes Element im Container den Elementtyp-Destruktor auf (falls dieser existiert) und gibt anschließend den Speicher frei, den die Elemente im Container belegen,
  - danach gibt der Operator `delete` den Speicher des `vector` Objekts frei.
- Dieses Konstruktoren / Destruktoren Schema lässt sich rekursiv anwenden, als Beispiel dieser Quellcode für `std::vector<double>`

```
vector< vector<double> >* p { new vector< vector<double> >(10) };
//…
delete p;
```

LEERE SEITE

Zeiger auf Klassenobjekte

Zeiger auf Klassenobjekte können in der gleichen Weise verwendet werden wie Zeiger auf Variablen der integrierten Typen.

- Syntax für den Zugriff auf die Member eines Objekts über Zeiger: Pfeiloperator **->**

```
myVector v(4);
int x {v.size()};
double d {v.get(3)};
```

```
myVector* p {&v};
int x {p->size()}; //(*p).size()
double d {p->get(3)}; //(*p).get(3)
```

- Zugriff aus Memberfunktionen auf das Objekt selbst: **this** Zeiger.

```
const myVector* myVector::adresse() const { return this; }
myVector x{};
cout << x.adresse(); // identisch: cout << &x;
```

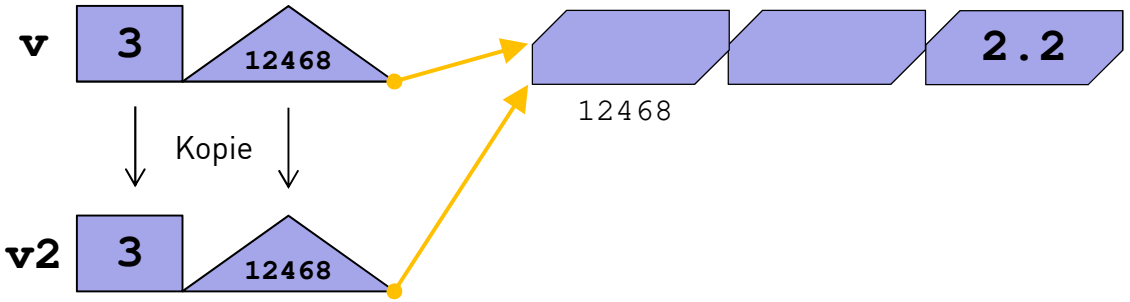
  - Jede nicht-statische Memberfunktion erhält als "versteckten" Parameter einen Zeiger auf das Objekt, in dessen Kontext sie aufgerufen wurde:
    den konstanten **this** Zeiger, der stets die Adresse des Objekts enthält, mit dem die Memberfunktion (über den Punktoperator) aufgerufen wurde.

Initialisierung durch Kopie

Warum funktioniert der folgende Quellcode nicht wie vielleicht intuitiv vermutet?

```
void f1( ) {
    myVector v( 3 );
    v.set( 2, 2.2 );
    myVector v2{ v }; // Was passiert hier?
}
```

- Beim Kopieren eines Objekts werden die Attribute kopiert, also `sz` und `elem`, und d.h. `v2` enthält *nicht* Kopien der Elemente von `v`, sondern:



- Grobe Fehler beim Verlassen der Funktion `f1()`:
  - Der Destruktor wird für `v2` und für `v` aufgerufen (zweimal).
  - Da die beiden `elem` Zeiger von `v2` und `v` auf die selben Elemente zeigen, werden die Elemente zweimal freigegeben.

Bildquelle: Stroustrup.

Kopierkonstruktoren

Der Typ `myVector` benötigt hierfür seine eigene, passend definierte Operation, den sog. Kopierkonstruktor.

```
class myVector {
    int sz; double* elem;
  public:
    myVector( const myVector& ); // Kopierkonstruktor
    // …
};
myVector::myVector( const myVector& r )
    : sz{ r.sz }, elem{ new double[r.sz] } {
    for( int i{}; i<r.sz; ++i ) elem[i] = r.elem[i];
 }
```

- Jetzt funktioniert der Quellcode

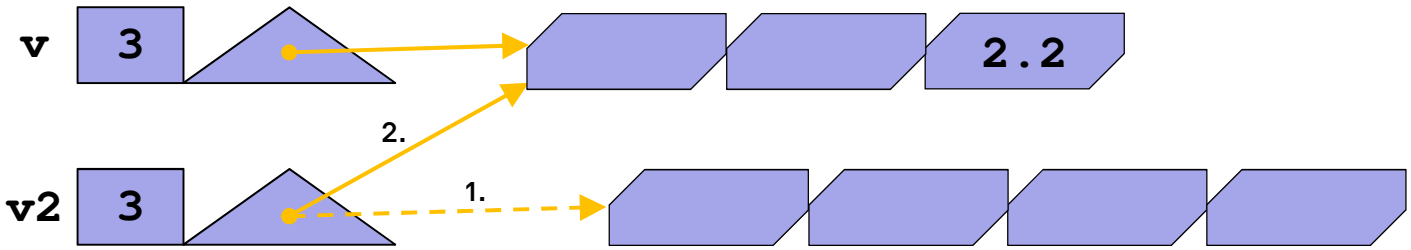
```
void f1( ) {
    myVector v( 3 );
    v.set( 2, 2.2 );
    myVector v2{ v }; // funktioniert jetzt auch: myVector v2 = v;
}
```

Zuweisung

Was richtet der folgende Quellcode an?

```
void f2( ) {
    myVector v( 3 );
    v.set( 2, 2.2 );
    myVector v2( 4 );
    v2 = v; // Was passiert hier?
}
```

- Es werden wieder die Attribute des `myVector` Objekts kopiert.
- Am Ende der Funktion `f2()` ist die Situation so:



- Grobe Fehler beim Verlassen der Funktion `f2()`:
  - Die Elemente von `v` werden zweimal freigegeben.
  - Die Elemente von `v2` werden ein Speicherleck.

Bildquelle: Stroustrup.

Zuweisungsoperatoren

Der Typ `myVector` benötigt seinen eigenen, passend definierten Zuweisungsoperator.

```
class myVector {
    int sz;
    double* elem;
  public:
    myVector& operator=( const myVector& ); // Zuweisung
    // …
};
myVector& myVector::operator=( const myVector& r ) {
    if( this != &r ) {
      // sog. "copy and swap":
      double* p { new double[r.sz] };
      for( int i{}; i<r.sz; ++i ) p[i] = r.elem[i];
      delete[] elem; // das alte Datenfeld freigeben
      elem = p;      // den Zeiger umhaengen
      sz = r.sz;     // die Groesse richtig setzen
    }
    return *this;  // das (eigene) Objekt zurueckgeben
}
```

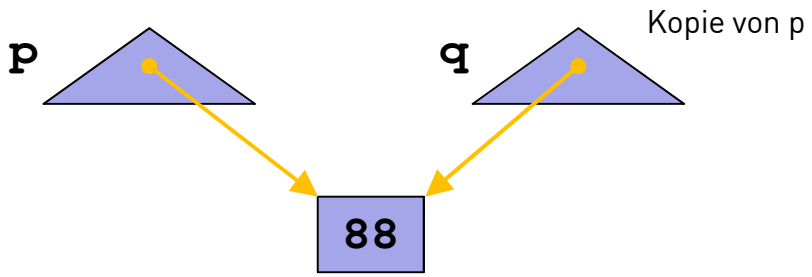
Flachkopie vs Tiefkopie

Flaches Kopieren (Zeiger- / Referenzsemantik) wird von tiefem Kopieren (Wertesemantik) unterschieden.

- Shallow copy (Flachkopie)
  - Nur der Zeiger (d.h. die Speicheradresse) wird kopiert, die beiden Zeiger verweisen dann auf das selbe Objekt (nach dem Kopieren gibt es zwei Zeiger aber immer noch nur ein Objekt).

```
int* p { new int{77} };
int* q { p };
*p = 88;
```

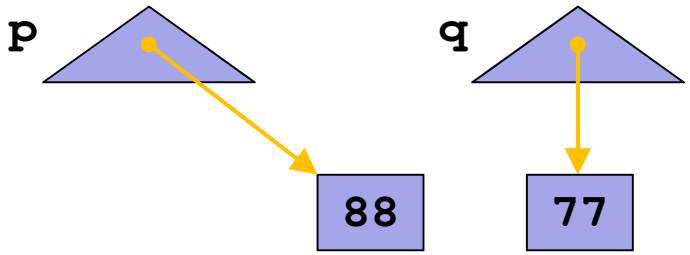
  - Sog. *Zeiger-* bzw. *Referenzsemantik.*



- Deep copy (Tiefkopie)
  - Der Zeiger *und* der Speicherbereich, auf den er zeigt, werden kopiert, so dass es dann zwei Zeiger und zwei unterschiedliche Objekte gibt.

```
int* p { new int{77} };
int* q { new int{*p} };
*p = 88;
```

  - Sog. *Wertesemantik.*



Essentielle Operationen

Bei der Definition eines Typs (z.B. namens `T`) sind mindestens diese fünf Operationen für Objekte der Klasse zu bedenken.

- Standardkonstruktor.
  - `T::T( );`
- Weitere Konstruktoren mit einem oder mehreren Parametern.
  - *Hinweis:* jeder Konstruktor, der genau einen Parameter übernimmt, definiert eine implizite Umwandlung vom Typ des Parameters in den Typ seiner Klasse.
    - Normalerweise ist das vermutlich nicht gewünscht... unterbinden kann man es für den Zuweisungsoperator durch Deklaration des Konstruktors als `explicit` (vgl. z.B. die `myVector` Klasse weiter vorne).
- Kopierkonstruktor zur Initialisierung (engl. "copy constructor").
  - `T::T( const T& );`
- Zuweisungsoperation (engl. "copy assignment").
  - `T& T::operator=( const T& );`
- Destruktor, falls in einem Konstruktor Ressourcen (Speicher, ...) angefordert wurden.
  - `T::~T();`

Essentielle Operationen

Der Compiler generiert folgende Operationen automatisch:

- Standardkonstruktor **`T::T( )`**
  - Wird so generiert, dass diejenigen nicht-`static` Datenmember initialisiert werden, die ohne Parameter initialisierbar sind (d.h. deren Typen einen Standardkonstruktor besitzen).
  - Dabei werden Datenmember von eingebautem Typ (z.B. `int`) *nicht* (!) initialisiert.
- Kopierkonstruktor **`T::T( const T& )`** und Zuweisungsoperation **`T& T::operator=( const T& )`**
  - Alle nicht-`static` Datenmember werden kopiert.
- Verschiebekonstruktor **`T::T( T&& )`** und Verschiebeoperation **`T& T::operator=( T&& )`**
  - Alle nicht-`static` Datenmember werden verschoben.
  - Sog. R-Referenzen als Parameter.
  - Wird im Rahmen dieser Lehrveranstaltung nicht weiter behandelt.
- Destruktor **`T::~T( )`**

Essentielle Operationen

Hinweise zu den vom Compiler generierten Operationen.

- Falls die Klasse Member hat, die nicht ohne Parameter initialisiert werden können: *es wird kein Standardkonstruktor generiert.*
- Falls die Klasse irgend einen benutzerdefinierten Konstruktor besitzt: *es wird weder Standardkonstruktor noch Destruktor generiert.*
- Falls die Klasse einen benutzerdefinierten Destruktor, oder benutzerdefinierten Kopier- oder Verschiebekonstruktor, oder benutzerdefinierte Zuweisungs- oder Verschiebeoperation besitzt: *es wird weder Destruktor, noch Kopier- oder Verschiebekonstruktor, noch Zuweisungs- oder Verschiebeoperation generiert.*

- Man kann die automatische Erzeugung aber mit `=default` erzwingen, z.B. **`~T()=default;`** für den Destruktor.
- Man kann die automatische Erzeugung auch mit `=delete` unterbinden, z.B. **`T( const T& )=delete;`** für den Kopierkonstruktor.
  - Mit der `=delete` Syntax kann *jede* Methode verboten werden, nicht nur die automatisch generierbaren.

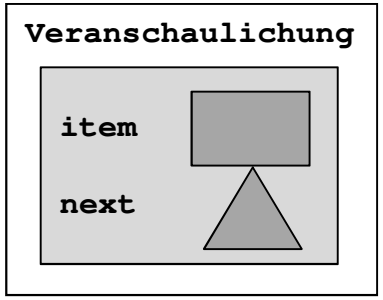
LEERE SEITE

Anwendungsbeispiel für Zeiger

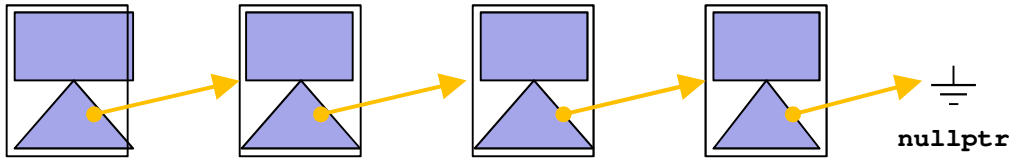
Eine einfach verkettete Liste implementiert einen Stapel.

- Die Implementierung eines Stapels als dynamische, einfach verkettete Liste ist ein Beispiel für dynamische Speicherstrukturen, Rekursion und Zeiger.
- Bei einem Stapel wird immer von oben gearbeitet (LIFO, last in first out), d.h.
  - neue Elemente kommen oben auf den Stapel,
  - das zuletzt abgelegte Element entnimmt man zuerst.
- Ein Stapel hat die Form:

```
// C++
class SiLL {
  // ...
  private:
    int item;
    SiLL* next;
};
```



- Damit lassen sich die Elemente verketten:



Anwendungsbeispiel für Zeiger

Ein Stapel für `int` Werte, Implementierung als einfach verkettete Liste.

- Der Typ und seine Konstruktoren

```
class SiLL { // Singly-Linked List, einfach verkettete Liste
  public:
    SiLL( int i, SiLL* p ) : item{ i }, next{ p }
      { }
    SiLL( ) : SiLL{ 0, nullptr }
      { }
    SiLL* push( int i );
    SiLL* pop( );
  private:
    int item;
    SiLL* next;
};
```

Anwendungsbeispiel für Zeiger

Ein Stapel für `int` Werte, Implementierung als einfach verkettete Liste.

- Ablegen eines neuen `int`-Elements auf den Stapel

```
SiLL* SiLL::push( int i ) {
    return new SiLL{ i, this };
}
```

- Entfernen des obersten Elements vom Stapel

```
SiLL* SiLL::pop( ) {
    if( next == nullptr ) return this;
    SiLL tmp { *this };
    delete this;
    return tmp.next;
}
```

Übung

Ein Stapel für `int` Werte, Implementierung als einfach verkettete Liste.

- Bringen Sie die `SiLL`-Klasse in einem `main()` Treiber zur Ausführung.
- Erweitern Sie die Schnittstelle der Klasse um eine Methode zur möglichst benutzerfreundlichen und informativen Ausgabe eines kompletten `SiLL`-Objekts.
- Legen Sie in einer Schleife einige `int` Werte auf den Stapel und entfernen Sie diese wieder.
- Wenn Sie genug Zeit haben implementieren Sie ein einfaches Menü zur interaktiven Auswahl von "push" (ablegen), "pop" (entfernen) und "print" (alles ausgeben).

Übung, Teil 2

Erkenntnisse und Beobachtungen aus der Implementierung.

- Speicheroperationen mit `new` und `delete` finden nur im `push()` bzw. im korrespondierenden `pop()` statt.
  - Gemäß der Semantik eines Stapels.
- Was passiert aber, wenn ein `SiLL` Objekt seinen Gültigkeitsbereich verlässt, bevor alle `int` Werte auf dem Stapel mittels `pop()` wieder entfernt wurden?
- Welche Lösungsmöglichkeit sehen Sie?
- Eine verbesserte Lösung soll sicherstellen, dass
  - Speicher dynamisch mit `new` in Konstruktoren angefordert wird und
  - jeder dynamisch angeforderter Speicher im Destruktor mit `delete` wieder freigegeben wird.

Anwendungsbeispiel für Zeiger

Ein Stapel für `int` Werte, verbesserte Implementierung mit einem eigenen Membertyp für die Elemente.

- Verbesserte Version mit einem Membertyp für die Listenelemente.

```
class SiLL {
   public:
      SiLL( ) : top{ new SiLLNode } {}
      ~SiLL( );
      void push( int );
      int pop( );
   private:
      struct SiLLNode { // nested type
          int item;
          SiLLNode* next{};
          SiLLNode() : SiLLNode{ 0, nullptr } {}
          SiLLNode( int i, SiLLNode* p ) : item{ i }, next{ p } {}
      };
      SiLLNode* top; // Listenanfang (Sentinel)
};
```

- Nun lässt sich ein einfacher Destruktor für `SiLL` Objekte definieren.
  - `SiLLNode` Objekte benötigen keinen eigenen Destruktor, da die `SiLLNode` Konstruktoren keine dynamischen Ressourcen (`new`) anfordern.

Anwendungsbeispiel für Zeiger

Ein Stapel für `int` Werte, verbesserte Implementierung mit einem eigenen Membertyp für die Elemente.

- Ablegen und entfernen für `SiLL` Objekte mit eigenem Membertyp für die Listenelemente.

```
void SiLL::push( int v ) {
    top = new SiLLNode{ v, top };
}

int SiLL::pop( ) {
    if( top->next != nullptr ) { // d.h. falls die Liste nicht leer ist
        SiLLNode tmp{ *top }; // Bem.: Kopierkonstruktor…
        delete top;
        top = tmp.next; // Bem.: Zuweisungsoperator…
        return tmp.item;
    }
    return top->item;
}
```

Anwendungsbeispiel für Zeiger

Ein Stapel für `int` Werte, verbesserte Implementierung mit einem eigenen Membertyp für die Elemente.

- Ein Destruktor lässt sich für `SiLL` Objekte mit eigenem Membertyp für die Listenelemente definieren.

```
SiLL::~SiLL( ) {
    SiLLNode* tmp1{ top };
    SiLLNode* tmp2{ nullptr };
    while( tmp1 != nullptr ) { // Speicher komplett zurueckgeben
        tmp2 = tmp1->next;
        delete tmp1;
        tmp1 = tmp2;
    }
}
```

LEERE SEITE

Einige Beispielfragen

Zeiger, Datenfelder, Listen.

- Warum reicht es uns nicht, pragmatisch vorzugehen und komfortable Typen wie `vector<T>` einfach anzuwenden? Warum müssen wir die Implementierungsdetails verstehen?
- Was ist ein Zeiger?
- Was ist eine Adresse? Wie werden Adressen in C++ manipuliert?
- Wie nennt man die beiden wichtigsten Zeigeroperationen? Was tun sie und wie ist ihre Notation?
- Kann ein `char`-Zeiger auch auf einen `int`-Wert zeigen? Warum bzw. warum nicht?
- Erklären Sie, was in diesem Quellcode passiert:

  ```
  int* p{}; int x {2}; p = &x; p += 1;
  ```

  Wie würden Sie diesen Quellcode verbessern?
- Wie finden Sie heraus, wieviel Speicherplatz eine `int`-Adresse benötigt?

Einige Beispielfragen

Zeiger, Datenfelder, Listen.

- Was ist der Unterschied zwischen Stack und Heap?
- Was ist ein Datenfeld? Wie lautet der englische Begriff für Datenfeld?
- Erklären Sie an einem *eigenen* Beispiel mindestens einen Grund, aus dem ein Programm zur Laufzeit zusätzlichen Speicher benötigen könnte.
- Wie fordert ein Programm zur Laufzeit weiteren Speicherplatz an? Erklären Sie es mit eigenen Worten.
- Wie könnte eine Quellcodezeile aussehen, die zusätzlichen Speicher für vier `double`-Werte bereit stellt?
- Erklären Sie die Unterschiede und die Gemeinsamkeiten zwischen

```
cin >> n; double* pd {new double[n]{}}; /*fuelle pd*/ cout << pd[1];
cin >> n; double* pd {new double[n]{}}; /*fuelle pd*/ cout << *(pd++);
```

- Was halten Sie von diesem Quellcode: `ad[-3] = 4.2;`
- Welche Informationen hat ein Zeiger über das Objekt, auf das er zeigt?

Einige Beispielfragen

Zeiger, Datenfelder, Listen.

- Sie haben einen Zeiger auf ein Objekt. Mit welchem Operator kann dieser Zeiger auf die `public` Member des Objekttyps zugreifen?
- Was ist ein Nullzeiger? Wofür braucht man so etwas?
- Was ist der *wesentliche* Unterschied bei der Speicherverwaltung zwischen statischen und dynamischen Objekten?
- Was ist ein Speicherleck?
- Welchen Gültigkeitsbereich haben `i`, `j` und `pi` in diesem Quellcode:

```
void f( int i ) {
    int* pi{};
    for( int j{}; j<i; ++j ) pi = new int{};
}
int main() { f(10); return 0; }
```

  Was passiert hier mit dem dynamisch reservierten Speicher?
- Wer kümmert sich um den dynamisch angeforderten Speicher, wenn er nicht mehr im Programm benötigt wird?

Einige Beispielfragen

Zeiger, Datenfelder, Listen.

- Wie könnte eine Quellcodezeile aussehen, die den dynamisch reservierten Speicher für ein `double`-Datenfeld namens `pd` freigibt?
- Was ist `int*k[10]` ?
- Was ist `int(*k)[10]` ?
- Was ist der Unterschied zwischen `vector<int>(1000)` und `vector<int>[1000]` und `vector<int>{1000}` ?
- Was ist ein Destruktor? Geben Sie ein eigenes Beispiel.
- Wofür definiert man Kopierkonstruktoren und Zuweisungsoperatoren bei benutzerdefinierten Typen?
- Was ist `this`? Wofür braucht man `this`? Geben Sie ein Beispiel.
- Was ist `*this`, und was ist `&this`?
- Erklären Sie, was Sie unter einer einfach verketteten Liste verstehen.
- Warum haben wir in der zweiten Version der einfach verketteten Liste zusätzlich zum Listentyp einen eigenen Typ für die Knoten definiert?

Grundlagen (ii)

Inhalt.

- Bitoperationen
- Zeiger, Datenfelder, Listen
- Beziehungen zwischen Klassen
- Ein- und Ausgabe
- Überblick: C++ Standardbibliothek

Klassendiagramme

Darstellung von Klassen und deren Beziehungen.

- Notation einer einzelnen Klasse.

Klassenname
Attribute (Datenmember)
Methoden (Memberfunktionen)

Klassenname
+Attribut-1-Name : Typ [=Defaultwert] #Attribut-2-Name : Typ [=Defaultwert] [-]Attribut-3-Name : Typ [=Defaultwert] *...alle weiteren Attribute der Klasse*
+Methode-1-Name(Parametertypen) : Ergebnistyp #Methode-2-Name(Parametertypen) : Ergebnistyp [-]Methode-3-Name(Parametertypen) : Ergebnistyp *...alle weiteren Methoden der Klasse*

public + protected #

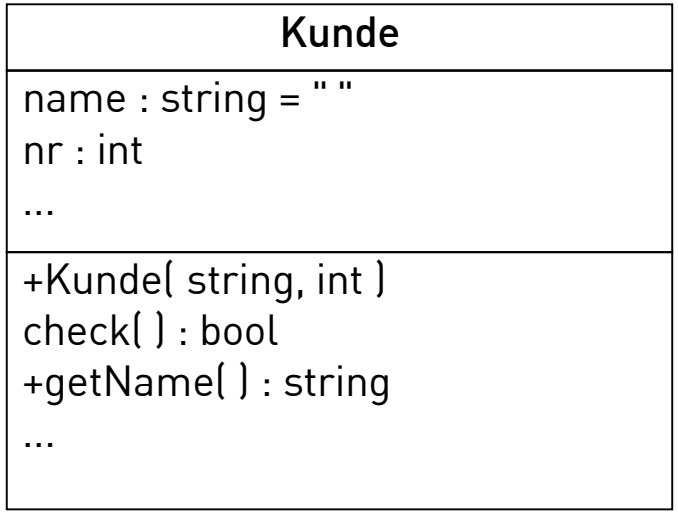
private – (oder leer)

- Wir verwenden die OMT (object modelling technique) nach Rumbaugh et al.

Klassendiagramme

Darstellung von Klassen und deren Beziehungen.

- Notation einer einzelnen Klasse: Beispiel.




```
class Kunde {
  private:
    string name;
    int nr;
    bool check( );
    //...
  public:
    Kunde( string s, int knr  )
      : name{s}, nr{knr} { }
    string getName( ) const;
    //...
};
```

- Eine Klasse umfasst *nur* die für den vorgesehenen Programmzweck *relevanten* Attribute und Methoden.

Übung

Vertiefung: Attribute, Methoden und Klassen.

- *Teil 1 von 2.* Nehmen Sie an, Ihr Autoschlüssel sei in den Gulli vor Ihrem Haus gefallen und Sie möchten ihn mit einem *Draht* wieder herausholen. Sie finden in einer Kiste in Ihrem Keller unterschiedliche Drähte, die sich mit der Zeit dort angesammelt haben. Welche der folgenden Drahteigenschaften wäre für Sie bei der Auswahl des geeigneten Drahts für die Schlüsselrettungsaktion relevant:

Farbe der Isolierung	Material des Drahts
Elektrischer Widerstand (Ohm)	Salzwasserfestigkeit der Isolierung
Gewicht	Feuerbeständigkeit der Isolierung
Länge	Einfaches Abisolieren
Durchmesser	Preis
Steifheit	Reissfestigkeit
Temperaturbeständigkeit	<welche noch?>

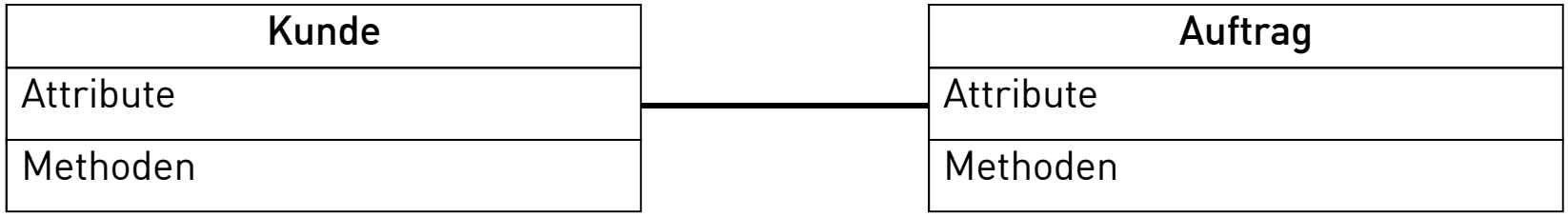
- *Teil 2 von 2.* Erstellen Sie für jede der folgenden "Draht-Anwendungen" eine Liste mit relevanten Drahteigenschaften, erklären Sie für jede Eigenschaft, warum sie für die Anwendung wichtig ist:

Mobile basteln	Überland-Hochspannungsleitung
Flugzeugelektronik	Vogelhäuschen am Baum aufhängen
Gitarrensaite	Fliegengitter

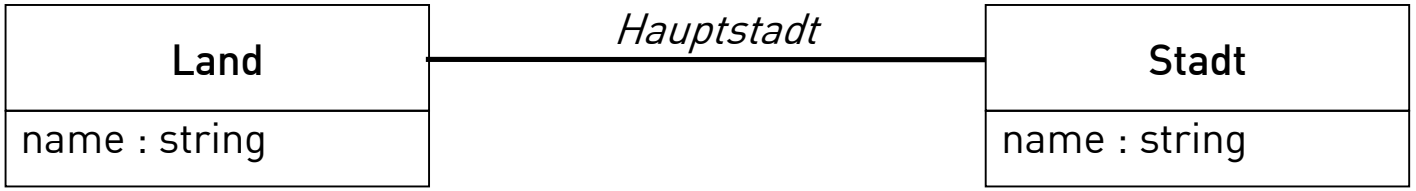
Assoziationen zwischen Klassen

Assoziationen sind Relationen zwischen Typen (Klassen) und sind immer wesenhaft bidirektional.

- Assoziationen sind zunächst nicht genauer ausgeführte Beziehungen zwischen zwei oder mehreren verschiedenen, durch Klassen beschriebenen Typen.
  - Bemerkung: Assoziationen sind u.a. auch die Grundidee hinter Relationen zwischen Daten (relationales Datenmodell).
- Eine Assoziation wird durch eine durchgezogene Linie zwischen den Klassen dargestellt.



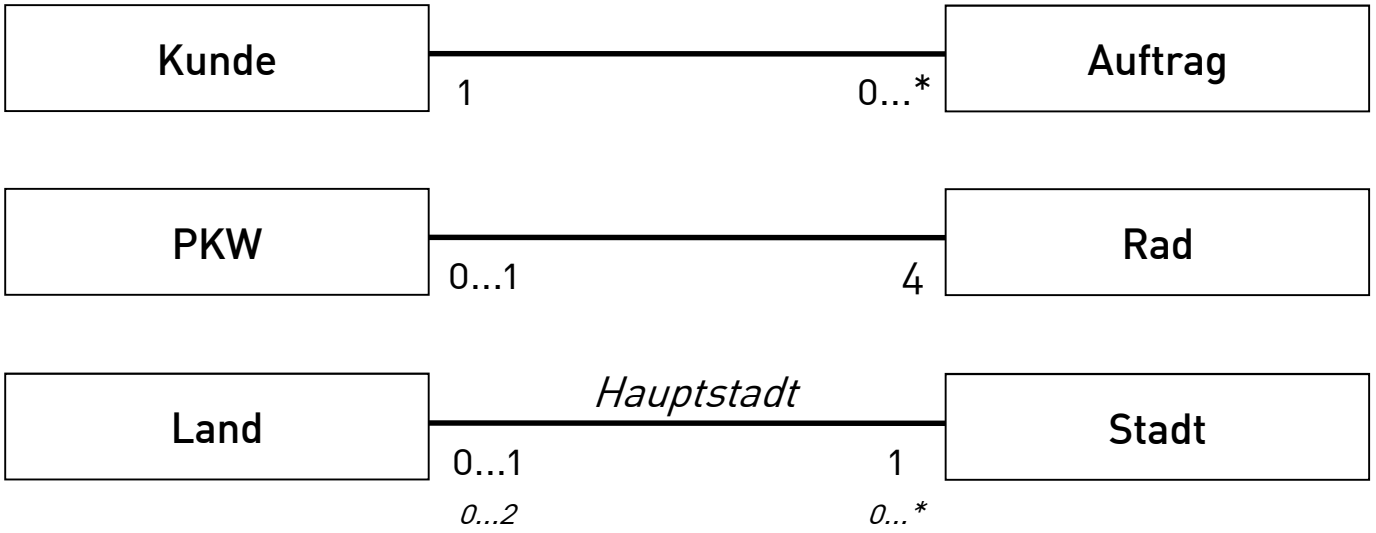
- Eine Assoziation kann optional einen Namen haben, der dann kursiv zur Linie dazugeschrieben wird.



Assoziationen zwischen Klassen

Multiplizität einer Assoziation.

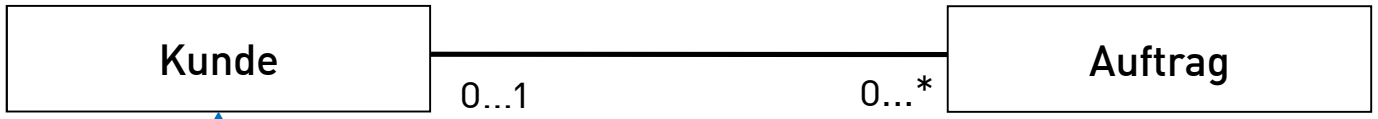
- Die Multiplizität einer Assoziation spezifiziert, wie viele Objekte eines Typs mit einem Objekt eines assoziierten Typs verbunden sein können.
  - Begrenzt die Zahl der Objekte nach oben und unten.
  - Wird am *Ende* der Assoziationsline angegeben.
    - Assoziationen sind wesenhaft bidirektional, haben also zwei Enden.
  - Ohne angegebene Multiplizität ist 1 gemeint.



Beispiel

Rudimentärer Quellcode: `Kunde` und `Auftrag`.

- Assoziation:

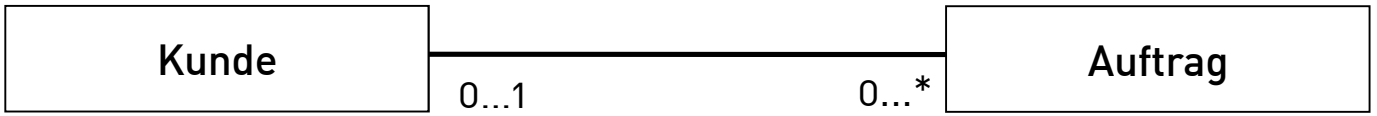



```
class Auftrag {
  public:
     Auftrag( string s ) : id{s}, pk{nullptr} {/*…*/}
     void print( ) const;
     string getID( ) const { return id; }
      //…
      // Assoziations-Operationen:
     void setKunde( Kunde* );
     Kunde* getKunde( ) const;
      //…
   private:
     string id;
     // Assoziationszeiger:
     Kunde* pk;
};
```

Beispiel

Rudimentärer Quellcode: `Kunde` und `Auftrag`.

- Assoziation:

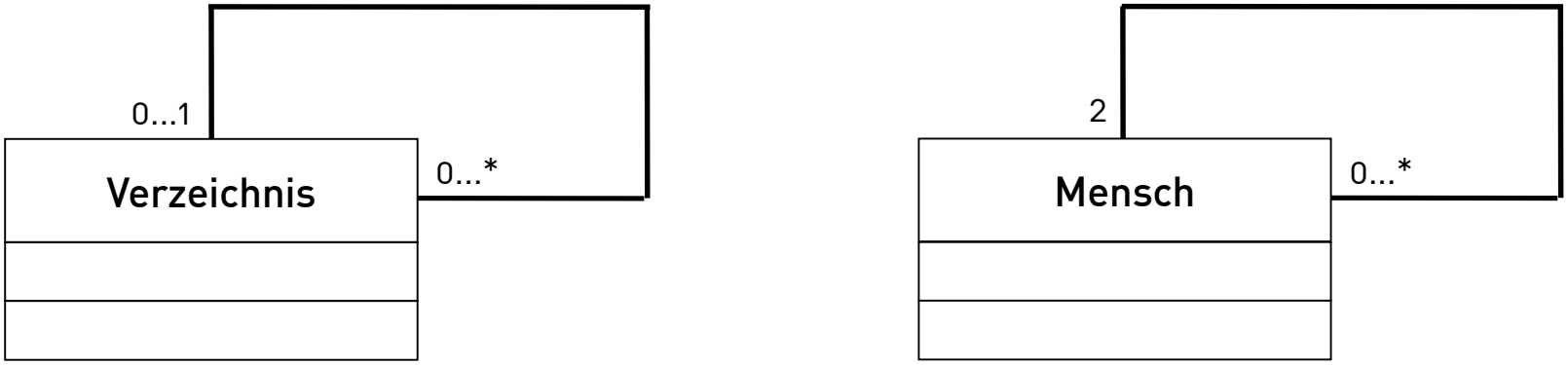



```
class Kunde {
  public:
    Kunde( string s ) : name{s}, vpa{} { /*…*/ }
    void print( ) const;
    string getName( ) const { return name; }
    //…
    // Assoziations-Operationen:
    void addAuftrag( Auftrag* );
    vector<Auftrag*> getAuftraege( ) const;

    //…
  private:
    string name;
    // Assoziationszeiger:
    vector<Auftrag*> vpa;
};
```

Reflexive Assoziationen

Beziehungen zwischen verschiedenen Objekten desselben Typs.

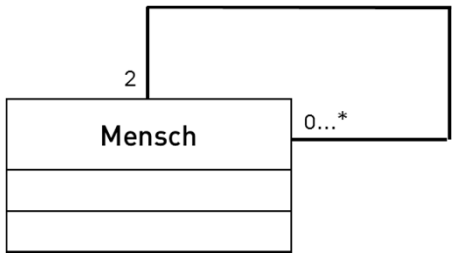
- Beispiel



- Reflexive Assoziationen sind, genau wie nicht-reflexive Assoziationen, wesenhaft bidirektional.
- Reflexive Assoziationen haben daher, genau wie nicht-reflexive Assoziationen, *zwei* Multiplizitäten.

Reflexive Assoziationen

Beziehungen zwischen verschiedenen Objekten desselben Typs.




```
class Hb {
    std::string name;
    Hb* const fath;
    Hb* const moth;
    std::vector<Hb*> kids;
  public:
    Hb() = delete;
    Hb( const std::string& n,
        Hb* const f,
        Hb* const m )
     : name{ n },
       fath{ f },
       moth{ m },
       kids{ }
    {
      if( fath ) fath->add_k( this );
      if( moth ) moth->add_k( this );
    }
    void add_k( Hb* kk ) {
      kids.push_back( kk );
    }
    void print( bool = true ) const;
    void print_kids() const;
};
```

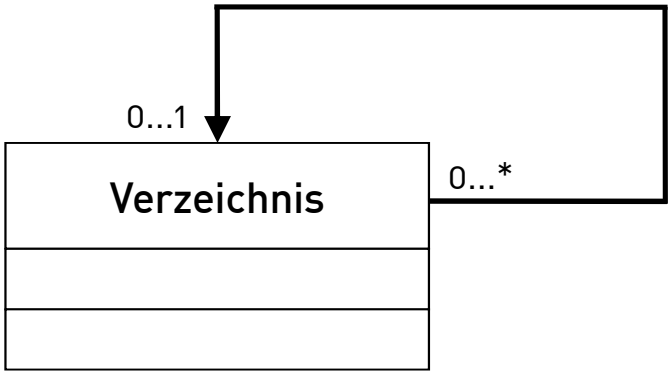
```
void Hb::print( bool b ) const {
  std::cout << name << ' ';
  if( b ) {
    if( fath ) std::cout << fath->name << ' ';
    else std::cout << "unknown ";
    if( moth ) std::cout << moth->name << ' ';
    else std::cout << "unknown ";
  }
  std::cout << '\n';
}

// Iterator fuer die Schleife einsetzen
// (mehr dazu folgt noch):
using citer = std::vector<Hb*>::const_iterator;
void Hb::print_kids() const {
  for( citer ci{ kids.begin() };
       ci != kids.end(); ++ci ) {
    (*ci)->print( false );
  }
}
```

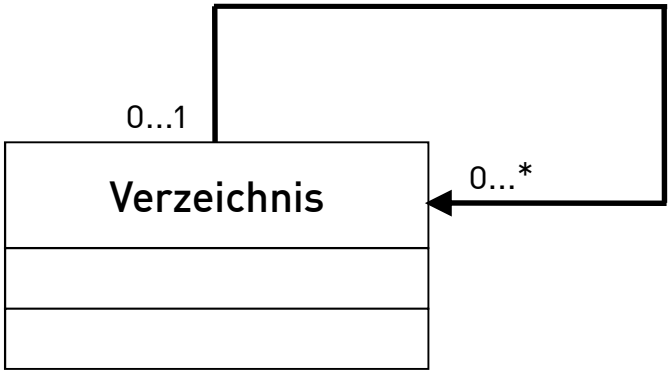
```
Hb ad{ "Adam", static_cast<Hb* const>(nullptr),
               static_cast<Hb* const>(nullptr) };
Hb ev{ "Eva", static_cast<Hb* const>(nullptr),
              static_cast<Hb* const>(nullptr) };
Hb ka{ "Kain", &ad, &ev };
ad.print();
ev.print_kids();
```

Einschränkung der Assoziationsrichtung

Die sog. "Navigierbarkeit".

- Eine Assoziation kann in der sog. "Navigierbarkeit" eingeschränkt werden.
  - Drückt aus, dass man im Programm die bidirektionale Assoziation nur in einer Richtung *verwendet.*
  - Darstellung durch eine Pfeilspitze an der Assoziationslinie.
  - Ohne Pfeilspitze ist die uneingeschränkte Navigierbarkeit gemeint.
- Beispiel:





- Verwendung der Assoziation ist im Programm nur in Richtung zum übergeordneten Verzeichnis im Baum vorgesehen.
- Verwendung der Assoziation ist im Programm nur in Richtung zu den Unterverzeichnissen im Baum vorgesehen.

Aggregation und Komposition

Aggregation und Komposition bezeichnen "Teil-Ganzes"-Beziehungen zwischen Objekten eines Gruppentyps und *eines* anderen Typs.

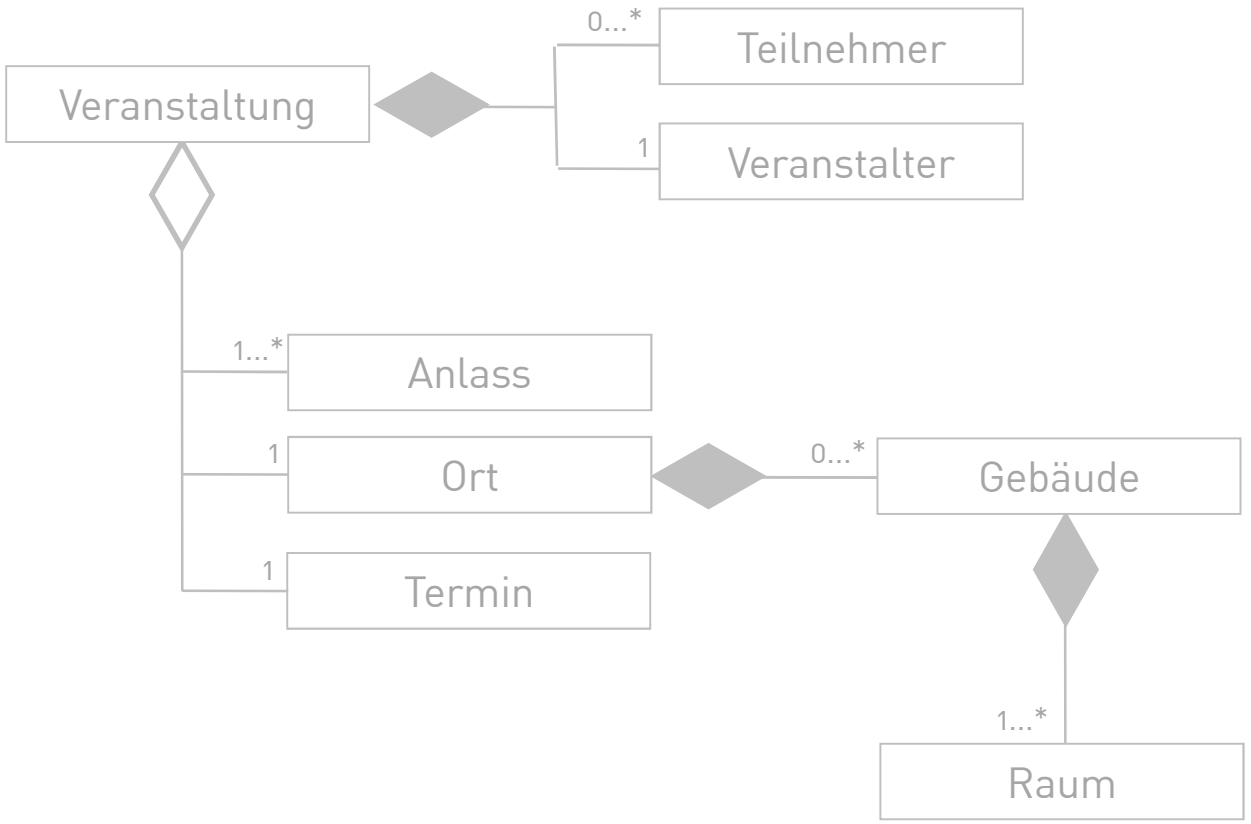
- Aggregation und Komposition
  - Wichtige Sonderformen der Assoziation.
- Beide Konstrukte stellen eine "ein B hat ein A" Beziehung her,
  - zwischen den Objekten eines übergeordneten Gruppentyps B
  - und den Objekten *eines* anderen Typs A.
  - Symbol: eine Raute am *Gruppentyp* der Assoziationslinie.
- Aggregation: kann mit Zeigermembern implementiert werden.
  - Das Teilobjekt vom Typ A kann unabhängig existieren, auch ohne dass es ein Gruppenobjekt vom Typ B gibt.
  - Symbol: nicht ausgefüllte Raute.

- Komposition: kann mit Attributen implementiert werden.
  - Das Teilobjekt vom Typ A kann nur existieren, wenn es ein Gruppenobjekt vom Typ B gibt.
  - Symbol: ausgefüllte Raute.

Aggregation und Komposition

Aggregation und Komposition bezeichnen "Teil-Ganzes"-Beziehungen zwischen Objekten eines Gruppentyps und *eines* anderen Typs.

- Beispiel



Einige Beispielfragen

Beziehungen zwischen Klassen.

- Wie lautet der C++ Quellcode zur Definition der folgenden Klasse:

Ratio
zaehl : long nenn : long = 1
+Ratio( ) +zaehler( ) : long +nenner( ) : long +print( ) : void kuerze( ) : void +malGleich(const Ratio&) : Ratio& +plusGleich(const Ratio&) : Ratio&

- Wie sieht das Klassendiagramm zu folgendem C++ Quellcode aus:

```
class Kunde {
  public:
    Kunde( );
    void setA( Auftrag* pa );
    Auftrag* getA( ) const;
  private:
    string name;
    Auftrag* pa;
};
class Auftrag {
  public:
    Auftrag( );
    int getID( ) const;
    Kunde* getK( ) const;
  private:
    int id;
    Kunde* pk;
 };
```

Einige Beispielfragen

Beziehungen zwischen Klassen.

- Erklären Sie, was eine Assoziation zwischen zwei Klassen ist, geben Sie zur Verdeutlichung ein *eigenes* Beispiel.
- Wie wird die Assoziation zwischen Klassen in Klassendiagrammen dargestellt?
- Was verstehen Sie unter der Multiplizität einer Assoziation? Wie wird diese in Klassendiagrammen dargestellt?
- Welche unterschiedlichen Multiplizitäten sind denkbar? Geben Sie einige eigene Beispiele.
- Wie werden Assoziationen zwischen Klassen programmiert?
- Erklären Sie, was eine reflexive Assoziation ist.
- Was versteht man unter der "Navigierbarkeit" zwischen Klassen?
- Warum wird die Navigierbarkeit manchmal eingeschränkt, obwohl Assoziationen immer wesenhaft bidirektional sind?

Einige Beispielfragen

Beziehungen zwischen Klassen.

- Was ist Aggregation? Verdeutlichen Sie das Konstrukt an einem eigenen Beispiel.
- Wie wird Aggregation in Klassendiagrammen dargestellt?
- Was ist Komposition? Verdeutlichen Sie das Konstrukt an einem eigenen Beispiel.
- Wie wird Komposition in Klassendiagrammen dargestellt?
- Welche wesentliche Gemeinsamkeit besteht zwischen Aggregation und Komposition?
- Was ist der wesentliche Unterschied zwischen Aggregation und Komposition?

Grundlagen (ii)

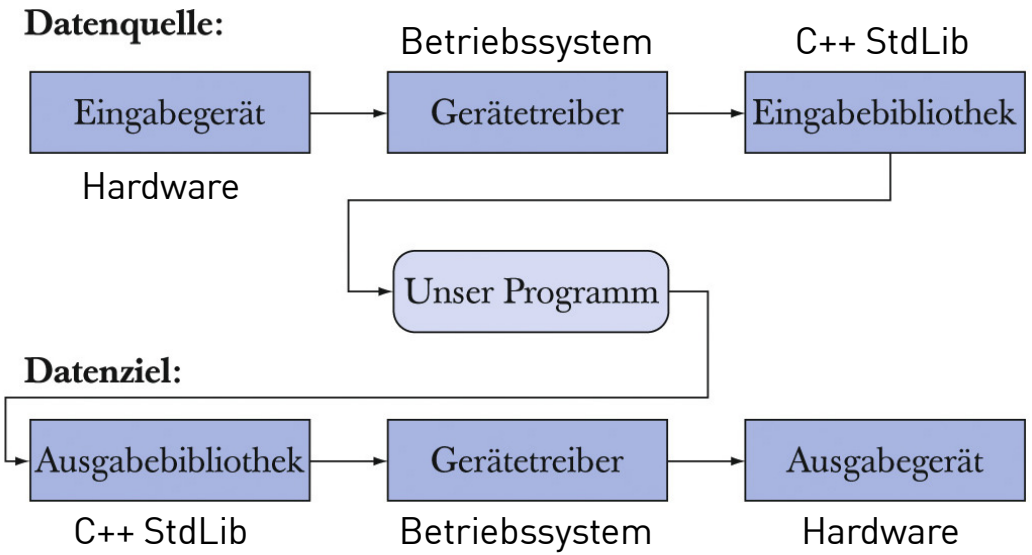
Inhalt.

- Bitoperationen
- Zeiger, Datenfelder, Listen
- Beziehungen zwischen Klassen
- Ein- und Ausgabe
- Überblick: C++ Standardbibliothek

Ein- und Ausgabe

E/A Schichtenmodell.

- Moderne Betriebsysteme lagern die Bedienung von Ein- und Ausgabegeräten in spezielle Software (Gerätetreiber) aus und greifen dann über eine E/A-Bibliothek, die die Ein- und Ausgabe von den physikalischen Geräten abstrahiert, auf diese Gerätetreiber zu.
- Wir betrachten zunächst sämtliche Ein- und Ausgaben als *Ströme von Bytes* (`unsigned char`), die von der *E/A-Bibliothek* gehandhabt werden.
- Wir konzentrieren uns darauf, Daten aus diesen Streams zu lesen und in sie zu schreiben.

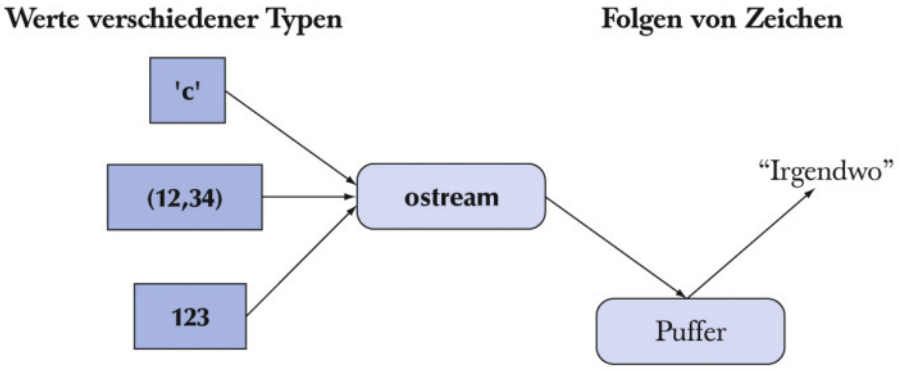


Bildquelle: Stroustrup.

Ein- und Ausgabe

C++ E/A-Strom-Modell: `ostream` Typ für Ausgabeströme.

- `ostream`
  - Typ aus der StdLib für die Arbeit mit Ausgabeströmen.
    - `cout` (der Standard-Ausgabestrom) ist ein Objekt vom Typ `ostream`.
  - `ostream` Objekte wandeln die Werte verschiedener Typen aus dem Programm in Folgen von Bytes um...
  - ...und senden diese Bytes dann "irgendwohin" (ans Betriebssystem) weiter.
  - Verfügen über einen *Ausgabe-Puffer* für die Kommunikation mit dem Betriebssystem.
    - Alle Daten, die in einen `ostream` Strom geschrieben werden, werden in diesem Puffer gespeichert, während das `ostream` Objekt mit dem Betriebssystem kommuniziert (dieser Effekt bleibt vom Benutzer weitgehend unbemerkt).

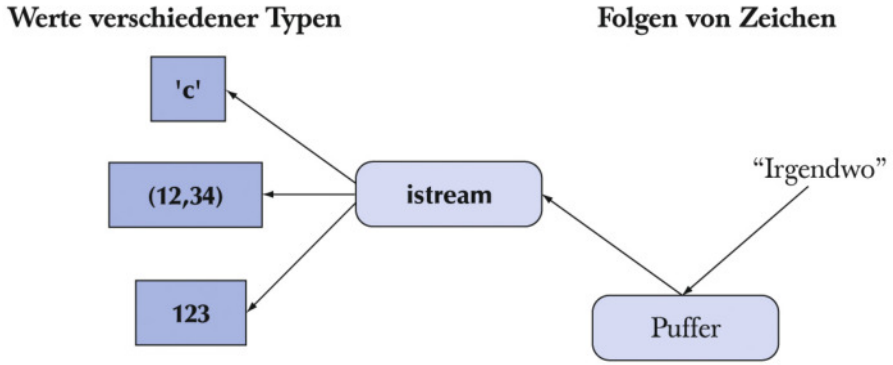


Bildquelle: Stroustrup.

Ein- und Ausgabe

C++ E/A-Strom-Modell: `istream` Typ für Eingabeströme.

- `istream`
  - Typ aus der StdLib für die Arbeit mit Eingabeströmen.
    - `cin` (der Standard-Eingabestrom) ist ein Objekt vom Typ `istream`.
  - `istream` Objekte holen "irgendwoher" (vom Betriebssystem) Folgen von Bytes...
  - ... und wandeln diese Folgen von Bytes in die Werte verschiedener Typen des Programms um.
  - Verfügen über einen *Eingabe-Puffer* für die Kommunikation mit dem Betriebssystem.
    - Die `istream` Pufferung kann deutlich spürbar sein: wenn z.B. ein `istream` Objekt verwendet wird, das mit der Tastatur verbunden ist, bleibt der eingetippte Input solange im Puffer stehen, bis die Enter-Taste betätigt wird.



Bildquelle: Stroustrup.

Ein- und Ausgabe

Die vier Standard-Stromobjekte.

- Beim Programmstart werden automatisch die folgenden vier Stromobjekte erzeugt und geöffnet:

```
istream cin;     // Standardeingabe (Tastatur)
ostream cout;    // Standardausgabe (Bildschirm)
ostream cerr;    // Standard-Fehlerausgabe (Bildschirm)
ostream clog;    // Standard-Logausgabe (Bildschirm)
```

- Die Objekte sind für Formatierung und Verwaltung ihres Stromstatus selbst zuständig.
  - Sie wandeln binäre Werte nach bestimmten Regeln in Zeichenströme um, und umgekehrt.
  - Ihre Typen sind von einer sog. Basisklasse namens `ios` "abgeleitet".
- Der *Transport* des Zeichenstroms und seine *Pufferung* erfolgt durch Objekte, deren Typen vom Typ `streambuf` "abgeleitet" sind.
  - Jedem E/A-Strom ist ein solches "Pufferobjekt" zugeordnet.
  - Die Kopplung erfolgt durch einen Zeiger im `istream` Objekt bzw. im `ostream` Objekt, der auf ein jeweiliges `streambuf` Objekt verweist.

Ein- und Ausgabe

Verbindung von `cin` und `cout`.

- Ein `istream` Objekt kann mit maximal einem `ostream` Objekt durch Aufruf der folgenden `istream` Methode verbunden werden:

  ```
  ostream* istream::tie( ostream* );
  ```

  - Durch *`istr`*`.tie( )` (ohne Argument) kann man abfragen, mit welchem `ostream` Objekt das `istream` Objekt namens *`istr`* verknüpft ist (liefert die Adresse bzw. `nullptr` falls keine Verknüpfung vorliegt).
  - Durch *`istr`*`.tie( nullptr )` wird die Verknüpfung zum `ostream` Objekt des `istream` Objekts namens *`istr`* aufgehoben.
- Die Verknüpfung bewirkt, dass vor jeder Leseoperation aus dem `istream` Objekt der Puffer des `ostream` Objekts geleert (d.h. ausgegeben) wird.
- Standardmäßig ist `cin` mit `cout` verknüpft.

Formatierte Ausgabe

Für die formatierte Ausgabe gibt es den für Objekte der Klasse `ostream` mehrfach überladenen Operator `<<`.

- In der `ostream` Klasse ist der Operator << (sog. Stream Insertion) für die üblichen Typen überladen.

```
double d {2.718}; cout << "d ist "; cout << d; cout << '\n';
```

- Je nach Typ wird ein bestimmtes, sinnvolles Standard-Ausgabeformat verwendet.
- **Das Ergebnis der Operation `<<` ist eine Referenz auf das `ostream` Objekt selbst, so dass man schreiben kann.**

```
double d {2.718}; cout << "d ist " << d << '\n';
```

  - **Vorsicht: die Verknüpfung (hier Ausgabe) erfolgt zwar von links nach rechts, die *Auswertungsreihenfolge* ist aber *undefiniert*.**
- Das Ausgabeformat kann weitgehend gesteuert werden.
  - Vorzugsweise wird das Format durch dafür definierte *Manipulatoren* gesteuert.
  - Das Format kann auch durch Parameter (Memberbits) des `ostream` Objekts und deren Änderung durch Schnittstellen-Methoden gesteuert werden.
    - Ein umständlicheres Verfahren als der Einsatz von Manipulatoren.
    - Kann aber notwendig sein, falls kein geeigneter Manipulator existiert.

Formatierte Ausgabe

Manipulatoren steuern die Ausgabeformatierung.

- Ausgabemanipulatoren sind Objekte aus der StdLib,
  - die (genau wie die auszugebenden Werte) in einen Operator << Ausdruck eingefügt werden können,
  - die normalerweise selbst keine Ausgabe erzeugen,
  - die aber Parameter (Memberbits) des Ausgabestroms ändern, wofür der Ausdruck nicht unterbrochen werden muss.

```
#include <iostream>
#include <iomanip>
using std::setw; using std::left; using std::right;
using std::dec;  using std::hex;  using std::oct;

int main( ) {
    int x {42};
    std::cout << setw(15) << left  << "dezimal:"
              << setw(10) << right << dec << x << '\n'
              << setw(15) << left  << "hexadezimal:"
              << setw(10) << right << hex << x << '\n'
              << setw(15) << left  << "oktal:"
              << setw(10) << right << oct << x << std::endl;
    return 0;
}
```

Formatierte Ausgabe

Manipulatoren steuern die Ausgabeformatierung.

- Die parameterlosen Manipulatoren wie `left`, `endl`, `ws`
  - sind durch Einbindung von `<iostream>` verfügbar,
  - sind Namen von (also Zeiger auf) Funktionen und im Prinzip so deklariert:

    ```
    ostream& (*manipulator)(ostream& stream).
    ```

  - Die Operatorfunktion `<<` der Klasse `ostream` ist für Manipulatoren speziell überladen:

    ```
    ostream& operator<<( ostream& (*manip)(ostream& stream) ) {
        return (*manip)(stream);
    }
    ```

    Die Manipulatorfunktion wird also mit dem aktuellen `ostream` Objekt als Parameter aufgerufen, damit hat z.B. `cout<<hex` die gleiche Funktion wie `hex( cout )`.
- Die Manipulatoren mit Parametern wie `setw( int n )`
  - sind durch Einbindung von `<iomanip>` verfügbar,
  - sind Aufrufe einer Funktion, die den Konstruktor einer bestimmten Klasse aufruft und das erzeugte Objekt zurück gibt (ein komplizierterer Vorgang als bei den parameterlosen Manipulatoren).

Anpassung der formatierten Ausgabe

Einfaches Überladen der Stromausgabe <<.

- Sie erinnern sich an die `Date` Klasse aus dem Grundlagenteil dieser Lehrveranstaltung.
- Stream Insertion wurde dort bereits für Objekte vom `Date` Typ überladen.

```
ostream& operator<<( ostream& os, const Date& d ) {
   return os << d.day() <<'-'<< d.month() <<'-'<< d.year();
}
```

- Diese Operatorfunktion bedeutet für Objekte `d1` und `d2` vom Typ `Date`
  - `cout << d1;`
    steht für den Aufruf `operator<<( cout, d1 );`
  - `cout << d1 << d2;`
    steht für den Aufruf `operator<<( operator<<(cout,d1), d2 );`
  - Usw.
- `operator<<()` übernimmt als erstes Argument eine `ostream` Referenz, und gibt sie als Rückgabewert zurück.

Unformatierte Ausgabe

Für die unformatierte Ausgabe gibt es Methoden an der Schnittstelle zur `ostream` Klasse.

- Die Ausgabe über folgende Methoden der `ostream` Klasse
  - wird nicht wie << von Formateinstellungen beeinflusst
  - und ist etwas schneller.

```
ostream& ostream::put( char c );
```

- Schreibt das Zeichen `c` in den Ausgabestrom.

```
ostream& ostream::write( char* a, int n );
```

- Schreibt n Zeichen ab Adresse a in den Ausgabestrom.
- Kann zur binären Ausgabe eingesetzt werden.

  z.B.
  ```
  double x{42.1};
  ostr.write( static_cast<char*>(&x), sizeof(x) );
  ```
  (Einsatz meist wenn *`ostr`* eine Ausgabe*datei* ist.)

Formatierte Eingabe

Für die formatierte Eingabe gibt es den für Objekte der Klasse `istream` mehrfach überladenen Operator `>>`.

- Die Eingabe erfolgt mit dem für die üblichen Typen überladenen Operator >> (Stream Extraction).
  - Der linke Operand ist ein `istream` Objekt.
  - Der rechte Operand ist eine Variable eines unterstützten Typs.
- Führender Whitespace wird standardmäßig bei allen Typen überlesen.
- Das Einlesen vom Strom endet beim ersten Zeichen, das nicht mehr zum Typ der Zielvariable passt.
  - Dieses Zeichen verbleibt im Eingabestrom und ist das erste, das von der nächsten Eingabe gelesen wird.
- Ist das erste nichtleere Zeichen für die jeweilige Variable unzulässig, geht der Eingabestrom in einen Fehlerzustand (`fail`).
  - Alle folgenden Eingaben aus diesem Strom bleiben wirkungslos (d.h. die Werte der weiter einzulesenden Variablen bleiben unverändert).
- Bei Zeichenketten (`string`, `char*`) beendet Whitespace per Konvention das Einlesen.

Formatierte Eingabe

Auch Eingabeformate können über Manipulatoren gesteuert werden.

- Beispiele:
  - `dec`, `hex`, `oct`
    geben die Zahlenbasis bei ganzzahligen Eingaben vor.
  - `setw(n)` begrenzt die Länge einer auf `char*` eingelesenen Zeichenfolge.
    - Wenn `n!=0` ist werden maximal `n-1` Zeichen gelesen.
    - `char zeile[60]{}; cin >> setw(sizeof(zeile)) >> zeile;`
    - Umfasst die eingelesene Zeichenkette mehr als 59 Zeichen, werden die restlichen Zeichen von der nächsten Eingabe gelesen (was dann zu Fehlern führen dürfte).
  - `skipws`, `noskipws`
    schalten das Überlesen von führendem Whitespace ein und aus.

Unformatierte Eingabe

Für die unformatierte Eingabe gibt es Methoden in der Schnittstelle zur `istream` Klasse.

- Die Eingabe über folgende Methoden der `istream` Klasse
  - transportiert Bytes unverändert vom Medium,
  - arbeitet unabhängig von Formateinstellungen,
    - insbes. werden führende Leerzeichen nicht überlesen,

```
istream& istream::get( char& c );
```

  - Liest das nächste Zeichen und speichert es bei `c`,
  - beim Versuch, über das Ende der Eingabe hinaus weiterzulesen, wird ein Fehlerstatusbit (`fail`) des `istream` Objekts gesetzt und `c` bleibt unverändert.

```
istream&
istream::get( char* s, int n, char t='\n');
```

  - Liest maximal n-1 Zeichen und speichert sie als Zeichenfolge ab `s`.
  - Das Einlesen endet, sobald das Zeichen `t` in der Eingabe auftaucht.
  - Das Zeichen `t` verbleibt im Eingabestrom und wird *nicht* in die Zeichenfolge übernommen.
    - Es kann mit einem weiteren `get()` ausgelesen werden.
    - Es kann mit der `ignore()` Methode (s.u.) verworfen werden.
  - Die Zeichenfolge wird automatisch mit einem `'\0'` als n-tes Zeichen abgeschlossen.

Unformatierte Eingabe

Für die unformatierte Eingabe gibt es Methoden in der Schnittstelle zur `istream` Klasse.

```
istream& istream::read( char* a, int n );
```

- Liest n Bytes (oder weniger, wenn das Eingabeende vorher erreicht ist) und legt sie ab Adresse `a` ab.
- Kann zur binären Eingabe verwendet werden.

  z.B.
  ```
  double x;
  istr.read( static_cast<char*>(&x), sizeof(x) );
  ```
  (Kann eingesetzt werden, wenn *`istr`* eine Eingabe*datei* ist.)

```
istream& istream::ignore( int n, char t=EOF );
```

- Überliest aus dem Eingabstrom maximal entweder n Zeichen oder bis zum Zeichen `t`, je nachdem was vorher der Fall ist.
- Damit lassen sich überflüssige Eingaben (etwa in Benutzerdialogen) überlesen.

Unformatierte Eingabe

Für die unformatierte Eingabe gibt es Methoden in der Schnittstelle zur `istream` Klasse.

`int istream::peek( );`

- Liefert das nächste Zeichen, ohne es aus dem Datenstrom zu entfernen.
- Liefert `EOF` (end of file) am Dateiende.
- Ermöglicht damit die Vorschau auf das kommende Zeichen im Strom.
  - Das ge-`peek`-te Zeichen ist das nächste Zeichen, das vom Strom gelesen wird.
  - Nach `cin.peek()` kann man z.B. entscheiden, auf welche Variable man einlesen will.

`istream& istream::putback( char c );`

- Stellt das Zeichen `c` "oben" in den Datenstrom.
- `c` ist dann das nächste Zeichen, das vom Strom gelesen wird.

Zur Fehlerbehandlung für Eingabeströme

C++ E/A-Strom-Modell: `istream` Typ für Eingabeströme.

- Bei der `ostream` Ausgabe geht man davon aus, dass normalerweise nur sehr selten Fehler auftreten.
  - Für Bildschirmausgaben ist das meist zutreffend, für Dateiausgaben nicht.
- Bei der `istream` Eingabe treten dagegen normalerweise eine ganze Menge von Fehlern auf.
- Fehler werden durch vier Bits des sog. *Stromstatus* dargestellt.
  - `ios_base::goodbit  // Schnittstelle bool good()`
    Alles OK, die Operation war erfolgreich.
  - `ios_base::eofbit  // Schnittstelle bool eof()`
    end-of-file, das Ende der Eingabe wurde erreicht (Strg-Z / Strg-D), normalerweise kein Fehler, das nächste Lesen der Eingabe wird aber zu einem Fehler führen.
  - **`ios_base::failbit  // Schnittstelle bool fail()`**
    Ein "noch korrigierbarer" Fehler ist aufgetreten (z.B. unpassende Variable).
  - `ios_base::badbit  // Schnittstelle bool bad()`
    Ein "nicht mehr korrigierbarer" Fehler ist aufgetreten, d.h. der Puffer ist unbrauchbar.

Zur Fehlerbehandlung für Eingabeströme

C++ E/A-Strom-Modell: `istream` Typ für Eingabeströme.

- Das Statuswort insgesamt erhält man mit
  `ios_base::iostate ios::rdstate() const;`
- Den Status kann man setzen mit
  `void ios::clear( ios_base::iostate = ios_base::goodbit );`
- Daraus ergibt sich folgende Prüflogik:

```
int i{}; cin >> i;
if( !cin ) {  // D.h.: falls cin.good() false ergibt
  if( cin.bad() ) error( "Schwerer Fehler in cin!" );
  if( cin.eof() ) {  /* Keine weiteren Eingaben erwartet */ }
  if( cin.fail() ) {    // Etwas Unerwartetes ist passiert:
                        // kontrolliertes Programmende
    cin.clear();        // oder den aufgetretenen Feher
                        // irgendwie beheben und dann z.B.
                        // eine neue Eingaben vorbereiten
  }
  //...
}
```

Anpassung der formatierten Eingabe

Überladener Stream Extraction Operator >>.

- Stream Extraction für einen gegebenen Typ zu überladen ist im Kern die Anwendung korrekter Fehlerbehandlung und kann daher schnell recht knifflig werden.
- Ein rudimentäres Quellcode-Beispiel für unsere `Date` Klasse:

```
istream& operator>>( istream& is, Date& dd ) {
   int y{}, m{}, d{};
   char c1{}, c2{}, c3{}, c4{};
   is >> c1 >> y >> c2 >> m >> c3 >> d >> c4;
   if( !is ) return is;
   if( c1!='(' || c2!=',' || c3!=',' || c4!=')' ) {
     is.clear( ios_base::failbit );
     return is;
   }
   dd = Date { y, Date::Month( m ), d };
   return is;
}
```

Ein- und Ausgabe und Zeichenketten

Zeichenketten als Objekte vom Typ `std::string` und als `'\0'`-terminierte `char` Datenfelder ("Zeichenketten im C-Stil").

- Wir haben für Zeichenketten den Typ `string` im `namespace std` aus der StdLib kennen gelernt.
- Zeichenketten können aber, wie wir auch schon gesehen haben, ebenso als `'\0'`-terminierte *Datenfelder* von `char` dargestellt werden.
  - Vor der Normierung von C++ war das der übliche Weg, daher wird diese Darstellung einer Zeichenkette auch als "C-Stil" bezeichnet.
  - Sie wissen über Datenfelder, dass sie entweder fixe Länge haben oder sich der Programmierer selbst um die dynamische Speicherverwaltung kümmern muss (das gilt auch für Zeichenketten im C-Stil).
- Viele Systemschnittstellen arbeiten mit dieser Low-Level Darstellung und können mit Objekten vom Typ `std::string` nicht umgehen.
  - Über den Header `<cstring>` gibt es einige Funktionen für `'\0'`-terminierte `char` Datenfelder.

Ein- und Ausgabe und Zeichenketten

Zeichenketten als Objekte vom Typ `std::string` und als `'\0'`-terminierte `char` Datenfelder ("Zeichenketten im C-Stil").

- Die `c_str()` Methode der `string` Klasse erzeugt aus einem Objekt vom C++ Typ `std::string` eine Low-Level-Zeichenkette im C-Stil.
  - Im Prinzip durch Einkopieren der Zeichen und Anhängen des `'\0'` Zeichens als Terminator.
  - Quellcode-Beispiel

```
#include <cstring>
#include <string>
using namespace std;

int main ( ) {
    string str { "Hallo C++" };
    char* cstr { new char[ str.size()+1 ]{} };

    strcpy( cstr, str.c_str() );
    // cstr enthält jetzt eine Kopie von str im C-Stil
    //...
    delete[] cstr; cstr=nullptr;
    return 0;
}
```

Ein- und Ausgabe und Zeichenketten

Zum Vergleich von `string` Objekten aus der C++ StdLib und `'\0'`-terminierten `char` Datenfeldern im C-Stil.

```
// C++ Zeichenkette
// string::operator= ueberladen für char*
#include <iostream>
#include <string>

int main( )
{
  std::string a {"zusammen"};
  std::string b {"gesetzt"};

  std::string c {a+b};




  std::cout << c << std::endl;
  //...

  return 0;
}
```

```
// Zeichenkette im C-Stil
// dyn. Speicher aber mit new/delete (C++)
#include <stdlib.h>
#include <string.h>

int main( )
{
  char* a = "zusammen";
  char* b = "gesetzt";

  char* c =
    new char[ strlen(a)+strlen(b)+1 ];
  strcpy( c,a );
  strcat( c,b );

  printf( "%s\n", c );
  //...
  delete[] c; c = nullptr;
  return 0;
}
```

Einige Beispielfragen

Ein- und Ausgabe.

- Nennen Sie fünf Arten von E/A-Geräten, über die Daten von und zu einem Programm fließen.
- Wie löst Software das Problem, dass Computer mit einer großen Vielfalt von potenziell möglichen Ein- und Ausgabegeräten kommunizieren müssen?
- Was sind `istream` und `ostream`, und was ist ihre wesentliche Aufgabe?
- Welche vier Stromobjekte stehen beim Programmstart automatisch zur Verfügung?
- Warum sind die Ein- und Ausgabeströme gepuffert?
- Was ist Stream Insertion? Was ist Stream Extraction?
- Warum ist das Ergebnis einer Stream Insertion Operation bzw. einer Stream Extraction Operation eine Referenz auf den Ein- bzw. Ausgabestrom selbst? Welchen praktischen Vorteil hat das?
- Was sind E/A-Manipulatoren? Wofür werden sie eingesetzt? Welche kennen Sie?

Einige Beispielfragen

Ein- und Ausgabe.

- Erklären Sie die vier Stream-Zustände der StdLib.
- In welcher Hinsicht bereitet die Eingabe mehr und größere Schwierigkeiten als die Ausgabe?
- In welcher Hinsicht bereitet die Ausgabe mehr und größere Schwierigkeiten als die Eingabe?
- Wie unterscheiden sich Objekte vom Typ `std::string` von `'\0'`-terminierten `char`-Datenfeldern?
- Warum werden für die Darstellung von Zeichenketten Objekte vom Typ `std::string` bevorzugt, und nicht `'\0'`-terminierte `char`-Datenfelder?
- Warum werden `'\0'`-terminierte `char`-Datenfelder trotzdem häufig gebraucht?

Grundlagen (ii)

Inhalt.

- Bitoperationen
- Zeiger, Datenfelder, Listen
- Beziehungen zwischen Klassen
- Ein- und Ausgabe
- Überblick: C++ Standardbibliothek

Die C++ Standardbibliothek (StdLib)

Bedeutung der StdLib.

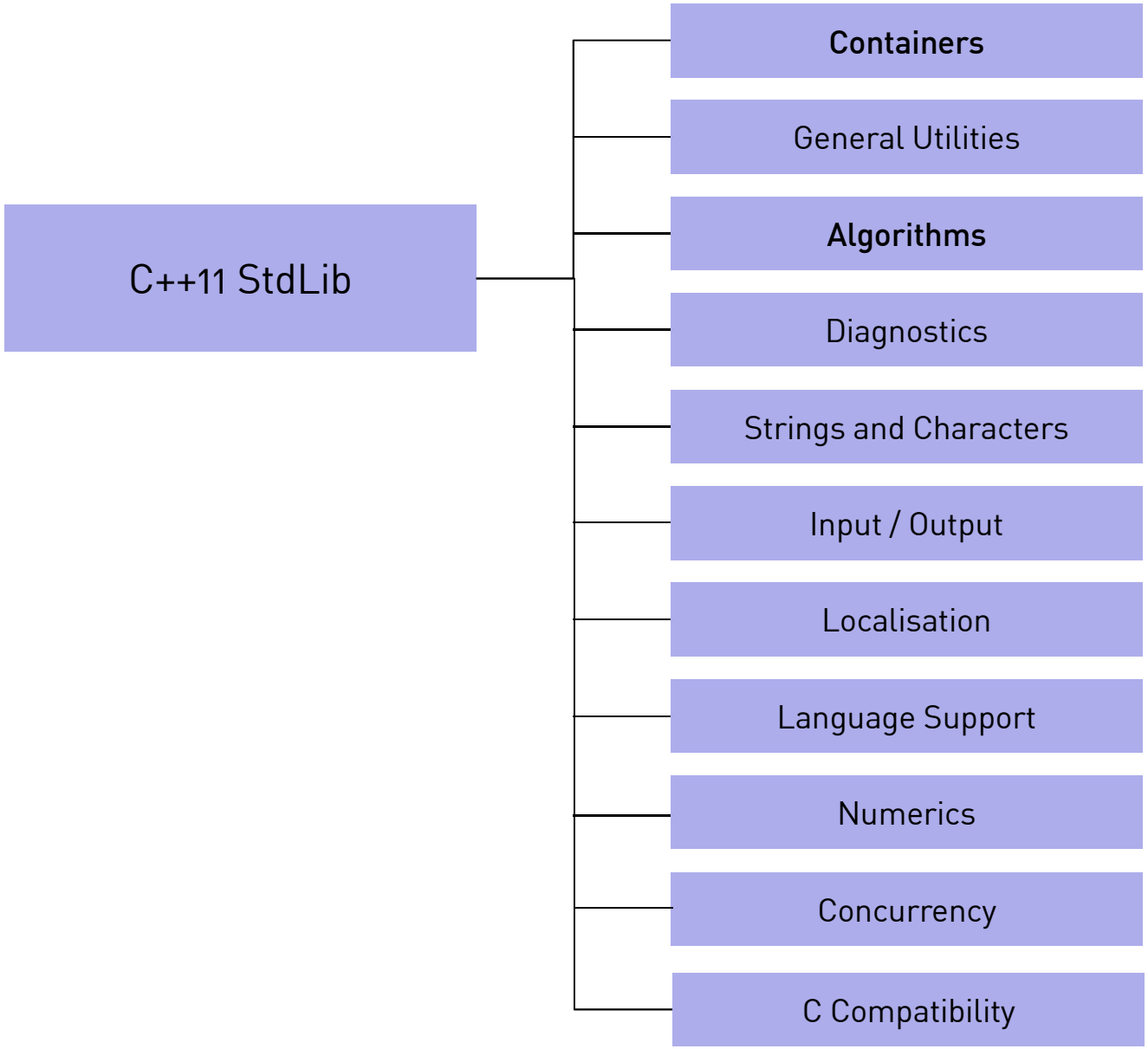
- Die StdLib ist ein fester Bestandteil von C++.
  - Parallel zur Sprache wurde auch die C++ Bibliothek normiert,
  - der bei weitem umfangreichste Teil des Verfahrens.
- Die StdLib befindet sich komplett im Namensraum `std`.
  - Sie erinnern sich, wo Sie die Deklarationen der StdLib finden und wie Sie diese für Ihre Programme zugänglich machen:

```
#include <iostream>     // Headerdatei einkopieren
using std::cout;        // Namensraum angeben
```

- Angesichts der Menge der in der StdLib verfügbaren Mittel ist es zunächst wichtig, einen groben Überblick zu bekommen (und idealerweise nicht mehr zu verlieren).
- Wichtige Teile der StdLib, sog. *Container*, *Iteratoren* und *Algorithmen*, werden noch näher behandelt.

Die Standardbibliothek (StdLib)

Bestandteile der C++11 StdLib.



Die Standardbibliothek (StdLib)

Die C-Headerdateien gibt es in der StdLib auch noch.

- Die Header in der ersten Spalte sind für strenge C-Kompatibilität gedacht und verwenden den globalen Namensraum.
- Die Header in der zweiten Spalte nutzen den C++ Namensraum `std` und sind ansonsten gleich.

C-Headerdatei	*Neue Form*	*Erläuterung*
<assert.h>	<cassert>	Ist der Ausdruck nicht wahr, bricht das Programm mit einer Fehlermeldung ab.
<ctype.h>	<cctype>	Klassifizierung von Zeichen.
<errno.h>	<cerrno>	Error-Fehlernummern für Bibliotheksfunktionen.
<float.h>	<cfloat>	Minimaler und maximaler Wert von Fließkommazahl-Typen.
<iso646.h>	<ciso646>	Operatormakros.
<limits.h>	<climits>	Minimaler und maximaler Wert von Ganzzahl-Typen.
<locale.h>	<clocale>	Kulturelle Besonderheiten abbilden.
<math.h>	<cmath>	Mathematische Funktionen.
<setjmp.h>	<csetjmp>	Ausführung von nicht-lokalen goto-Anweisungen.
<signal.h>	<csignal>	Legt eine Aktion fest, die bei einem bestimmten Signal ausgelöst wird. Fehlerroutinen.
<stdarg.h>	<cstdarg>	Variable Argumentlisten für Funktionen mit einer variablen Anzahl von Parametern.
<stddef.h>	<cstddef>	Definition von einfachen Typen und Makros wie zum Beispiel den NULL-Zeiger.
<stdio.h>	<cstdio>	Ein- und Ausgabe auf die Konsole oder in Dateien.
<stdlib.h>	<cstdlib>	Allgemeines wie Konstanten für den Programmabbruch.
<string.h>	<cstring>	C-Funktionen für nullterminierte Zeichenfelder.
<time.h>	<ctime>	Zeit- und Datumsfunktionen.
<wchar.h>	<cwchar>	Umwandlung von Strings zu Zahlen für den Unicode-Zeichensatz.
<wctype.h>	<cwctype>	Zeichenuntersuchung für den Unicode-Zeichensatz.

Quelle: RRZN Hannover.

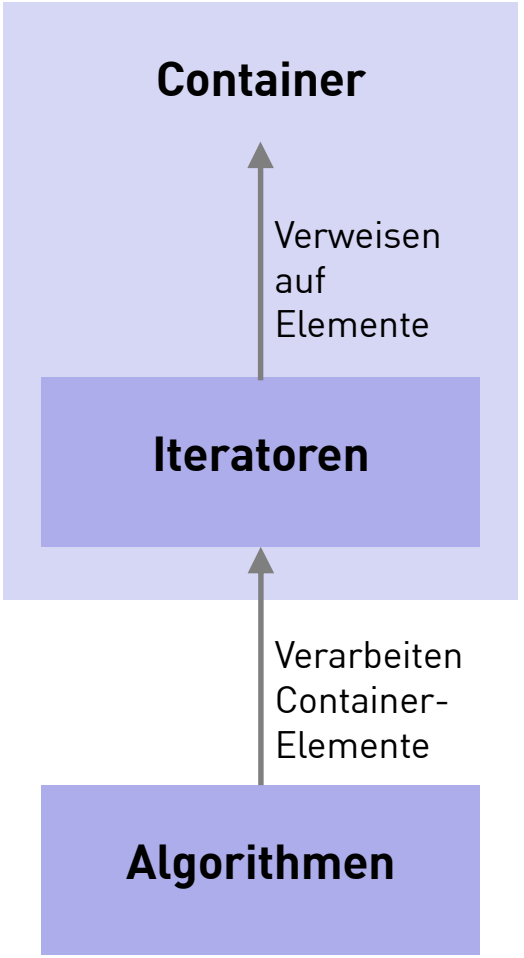
Container der Standard Library

Containertypen, ihre Iteratoren und Algorithmen sind ein wichtiger Teil der StdLib.

- *Container* nehmen Elemente von beliebigem Typ `T` auf.
  Beispiel:
  `vector<T>`

- *Iteratoren* ermöglichen den Zugriff auf die Containerelemente.
  `input, output, forward, bidirectional, random access`

- *Algorithmen* verarbeiten die Elemente im Container.
  Beispiele:
  `count(), sort()`



Einige Beispielfragen

C++ StdLib.

- Welchen `namespace` verwendet die StdLib?
- Wo sind die Deklarationen der von der StdLib bereit gestellten Mittel?
- Wie können zwei Quellcodezeilen aussehen, die in der Quellcodedatei auftauchen müssen, bevor man den Typ `string` aus der StdLib verwenden kann?
- Was sind die Containertypen der StdLib?
- Welchen Unterschied gibt es zwischen *time.h* und *ctime*?

Nächste Einheit:
Templates