Pablo Lahmann, Informatik HS2019

Informatik Zusammenfassung

Bibliotheken

#include <iostream> Standardbibliothek für Input- und Output Streams (std::cout << usw.)

Datentypen

Bei Berechnungen mit verschiedenen Datentypen wird der „kleinere" Datentyp in den „grösseren" umgerechnet:
Boolean < int < unsigned int < float < double
Bsp: 3 / 6.0 = 0.5
char Zeichen (hat nach ASCII auch einen Zahlenwert)
string Folge von chars
int ganze Zahlen
unsigned int ganze positive Zahlen (grösserer Wertebereich); 5173**u** (kennzeichnet unsigned int)
(Umrechnungen versch. Datentypen später)
bool für Wahrheitswerte (true, false)
Float, Double Kommazahlen (double hat grösseren Wertebereich und mehr Genauigkeit)

Variablen besitzen Namen, Typ, Wert und Adresse (repräsentieren wechselnde Werte)
Ausdrücke stehen für Berechnungen (mit Operatoren verknüpft)

L-Wert ist immer links vom Gleichheitszeichen; veränderbarer Wert;
identifiziert Adresse (Speicherplatz); kann als R-Wert verwendet werden
z.B. **x** = 5; **a** = (x = 5)
R-Wert Ausdruck, der kein L-Wert ist; kann seinen Wert nicht ändern; Berechnungen
z.B. x = **5**; a = (**x = 5**) ; **55 * 6**; **a * a**

Eingabeoperator >> beide Operanden sind L-Werte: std::cin >> a
Ausgabeoperator << linker Operand ist L-Wert (Ausgabestrom), rechter Operand ist R-Wert
std::cout << a;
Zuweisungsoperator = weist einem L-Wert einen R-Wert zu: int a = 5;

Bool'sche Operatoren

&& logisches UND
| | logisches ODER
Es gilt **Kurzschlussauswertung**: wenn der linke Operand eine Eindeutige Aussage zulässt, wird der Rechte garnicht erst überprüft.
! logische Negation
== Gleichheit
Weitere: **<, <=, > , >=**

Präzedenzen: multiplikative Operatoren (*, /, %) haben höhere Präzedenz als Additive (+, -)
Assoziativität: arithmetische Operatoren sind linksassoziativ (von links nach rechts ausgewertet)
Rechtsassoziativ: = ,
Stelligkeit: unäre Operatoren (+, -)werden von Binären (+, -) ausgewertet: (-3) + 4

Präzedenzen aller Operatoren:
(oben höchste Priorität)
(je weiter oben, desto früher wird dieser Operator Ausgeführt)

Operator	Beschreibung	Assoziativität
`::`	Bereichsauflösung	von links nach rechts
`++ --`	Suffix-/Postfix-Inkrement und -Dekrement	
`()`	Funktionsaufruf	
`[]`	Arrayindizierung	
`.`	Elementselektion einer Referenz	
`->`	Elementselektion eines Zeigers	
`++ --`	Präfix-Inkrement und -Dekrement	von rechts nach links
`+ -`	unäres Plus und unäres Minus	
`! ~`	logisches NOT und bitweises NOT	
`(type)`	Typkonvertierung	
`*`	Dereferenzierung	
`&`	Adresse von	
`sizeof`	Typ-/Objektgröße	
`new, new[]`	Reservierung Dynamischen Speichers	
`delete, delete[]`	Freigabe Dynamischen Speichers	
`.* ->*`	Zeiger-auf-Element	von links nach rechts
`* / %`	Multiplikation, Division und Rest	
`+ -`	Addition und Subtraktion	
`<< >>`	bitweise Rechts- und Linksverschiebung	
`< <=`	kleiner-als und kleiner-gleich	
`> >=`	größer-als und größer-gleich	
`== !=`	gleich und ungleich	
`&`	bitweises AND	
`^`	bitweises XOR (entweder-oder)	
`|`	bitweises OR (ein oder beide)	
`&&`	logisches AND	
`||`	logisches OR	
`?:`	bedingte Zuweisung	von rechts nach links
`=`	einfache Zuweisung (automatische Unterstützung ist in C++-Klassen Vorgabe)	
`+= -=`	Zuweisung nach Addition/Subtraktion	
`*= /= %=`	Zuweisung nach Multiplikation, Division, und Rest	
`<<= >>=`	Zuweisung nach Links- bzw. Rechtsverschiebung	
`&= ^= |=`	Zuweisung nach bitweisem AND, XOR, und OR	
`throw`	Ausnahme werfen	
`,`	Komma (Sequenzoperator)	von links nach rechts

Modulo Rechnung für negative Zahlen: a = (a/b) * b + a % b (Div-Mod-Identität)

Post-Increment: expr++ der alte Wert von expr wird ausgegeben (**R-Wert**), danach wird expr um 1 erhöht
Prä-Increment: ++expr der Wert von expr wird um 1 erhöht, und expr wird als **L-Wert** zurückgegeben (kann also noch verändert werden)
(analog gibt es das Dekrement: - - expr und expr - -)

Arithmetische Zuweisung: a += b entspricht a = a + b (übertragbar auf andere Operatoren -, *, /, %)

Hexadezimalzahlen: auf Basis 16; geschrieben mit Präfix 0x

$h_n\, h_{n-1}\, \ldots\, h_1\, h_0$ entspricht $h_n \cdot 16^n + \ldots + h_1 \cdot 16^1 + h_0 \cdot 16^0$

Bsp.: 0x4b7 entspricht $4 * 16^2 + 11 * 16^1 + 7 * 16^0 = 1207$

Hex Nibbles

hex	bin	dec
0	0000	0
1	**0001**	**1**
2	**0010**	**2**
3	0011	3
4	**0100**	**4**
5	0101	5
6	0110	6
7	0111	7
8	**1000**	**8**
9	1001	9
a	1010	10
b	1011	11
c	1100	12
d	1101	13
e	1110	14
f	1111	15

Binärzahlen: analog auf Basis 2; mit Präfix 0b
Bsp.: 0b0110 entspricht $0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0 = 5$

Umrechnung von Datentypen:
Unsigned int → int: wenn a < 0, wird a von dem grössten unsigned int abgezogen
Bool → int: „**false**“ entspricht **0** und „**true**“ entspricht **1**
Int → bool: 0 entspricht **false** und jede Zahl **ungleich 0** wird zu **true**
Int → binär: wiederholt durch 2 teilen (bis 0); die Reste ergeben den Binärcode
Bsp.: 5 /2 = 2 R **1** ; 2 /2 = 1 R **0** ; 1 /2 R **1 → 5 = 0b101**
Kommazahl → binär: (Algorithmus) 1.Stelle vor dem Komma als Binärwert wählen
2. Nachkommastellen als Zahl verdoppeln → Vorgang wiederholen
Die Binärwerte ergeben nacheinander die Dezimaldarstellung (siehe Fliesskommazahlen)

Schleifen
Break: unterbricht gesamte Schleife
Continue: springt zum nächsten Schleifendurchlauf über

For-Schleife: **for (initial statement; condition; expression){body statement;}**

Bsp: for(int i = 0; i < 6; i++){std::cout << "hello";}

-falls condition true, wird body statement ausgeführt (danach die expression)

Entspricht: { int i = 0; while(condition) { BODY; ++i; }}

While-Schleife: while (condition) { Body };

Entspricht: for (; condition;) {Body};

Do-While: do {Body} while (condition);

Entspricht: Body; for (; condition;) {Body};

If-Bedingungen: if (condition) {statement1} else {statement2};

Bsp.: if (a < b) { a = b; } else { a++; };

-eine leere condition gilt als true

Switch: Fallunterscheidung; falls ein case wahr, werden alle darunter ausgeführt (bis break;)

Bsp.: switch (variable){

Case 1 : statement 1; break;

Case 2: [statement darunter wird ausgeführt]

Case 3: statement 2; break;

Default: statementdef; } [wenn keiner der Fälle eintritt]

assert: erfordert Bibliothek: #include <cassert>; Stoppt Programm bei Verletzung einer Bedingung (zu Testzwecken)

Bsp.: assert (a <= b);

De Morgan'sche Regeln: !(a && b) == (!a || !b) und !(a || b) == (!a && !b)

Const Variablen mit unveränderbarem Wert (Konstanten)

const int a = 5;

Block : { … } eine in einem Block deklarierte (lokale) Variable existiert nur bis zum Ende des Blocks

Veränderungen einer Variable gelten nur innerhalb eines Blocks

Fliesskommazahlen: Ein Fliesskommazahlensystem ist durch vier natürliche Zahlen definiert:

- $\beta \geq 2$, die Basis,
- $p \geq 1$, die Präzision (Stellenzahl),
- $e_{\min}$, der kleinste Exponent,
- $e_{\max}$, der grösste Exponent.

Bezeichnung:

$$F(\beta, p, e_{\min}, e_{\max})$$

enthält die Zahlen: $\pm\, d_0 \bullet d_1 \ldots d_{p-1} \times \beta^e,$

Normalisierte Form (mit $d_0 \neq 0$): ist eindeutig und wird bevorzugt (erste Ziffer ungleich 0 kommt vor das Komma)

Umwandlung von Dezimalzahl in Binärzahl:

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2
0.2	$b_1 = 0$	0.2	0.4
0.4	$b_2 = 0$	0.4	0.8
0.8	$b_3 = 0$	0.8	1.6
1.6	$b_4 = 1$	0.6	1.2
1.2	$b_5 = 1$	0.2	0.4

(1.1 hat nur periodische Binärdarstellung)

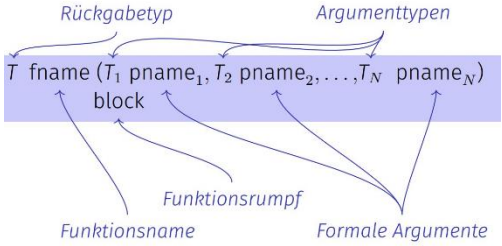
→deshalb **aufpassen bei Rechnungen mit Kommazahlen**

Regeln zur Fehlervermeidung:

1.keine gerundeten Fliesskommazahlen auf Gleichheit prüfen

2.keine Zahlen sehr unterschiedlicher Grösse aufaddieren (Bsp.: (10+0.5)+0.5 = 10)

3.keine zwei Zahlen sehr ähnlicher Grösse voneinander subtrahieren



Funktionen
Strukturierung des Programms (Unterteilung in kleine Teilaufgaben); selbstständiger Codeabschnitt

Funktionsaufruf: fname(expr1, expr2, …) (jede expr ist ein R-Wert)
Rückgabewert: return expr; Variable „expr" wird in den Rückgabetyp umgewandelt
Eine Funktion mit Rückgabetyp „**void**" hat **keinen Rückgabewert** (benötigt kein return;)

Jede Funktion (ausser main) braucht **PRE- / POST- Conditions** (vor die Funktionsdefinition)
PRE: Bedingungen, die für die Eingabewerte (Argumente) gelten müssen (so schwach wie möglich)
POST: Wie die Funktion in verschiedenen Fällen handelt (z.B. gibt true aus, falls a == 0)(so stark, genau wie möglich)
Mit **Assertions** lassen sich die Conditions überprüfen

Eine Funktion ist erst **nach** ihrer Deklaration im Code gültig (eine Definition entspricht einer Deklaration)
Eine **Deklaration** entspricht einer Funktionsdefinition, ohne den Teil { … } (pow (in a, int b);)

Funktionen aus der Standardbibliothek können aufgerufen werden mit dem Präfix **std::**
Bsp.: std::sqrt(a) (pow, abs, min(a,b)) → mit #include <cmath>

Referenztypen referenzieren eine bestehende Variable (vom gleichen Typ, mit dem sie initialisiert wurden)
-müssen immer mit einem L-Wert initialisiert werden (also Werte mit einer Adresse im Speicher) → Alias
-Funktionen mit Referenzen als Argumente, können diese Aufrufargumente ändern (Bsp.: swap-Funktion)
-Deklaration erfolgt mit **TYP& name** (Bsp.: bool& w)
Bsp.: int a = 3; **int& b = a**;
-die Referenz hat den gleichen Typ und die gleiche Funktionalität, wie eine normale Variable des gleich Typs

Pass-by-Reference (oder call-by-reference) sind Funktionsargumente, die einen Referenztypen haben (die Variable wird mit der Adresse des Aufrufarguments initialisiert, als L-Wert)
Pass-by-Value sind Funktionsargumente ohne Referenzen (die Variable wird mit dem Wert des Aufrufarguments als R-Wert initialisiert → eine Kopie wird erstellt)

Return-by-Reference sind referenzierte Rückgabetypen von Funktionen;
-der Funktionsaufruf wird zum L-Wert (Ausgabevariable kann noch verändert werden; Eingabewert muss L-Wert sein)
-daher möglich: inc(inc(a)); oder ++(inc(a));

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

Const Referenzen verbieten das Verändern des Ziels der Referenz; const Referenzen können auch mit R-Werten initialisiert werden (dies erlaubt Funktionsaufrufe trotz Referenzen, mit R-Werten zu machen)
-keine normale Referenz darf mit einer const-Referenz initialisiert werden
-Funktionsargumenttypen mit **pass-by-read-only-reference** sind **effizient** und **sicher** (Funktionsargumente mit const Referenz)
Form: **const Typ& name = Wert;**
Bsp.: const int& a = 3; oder const int& a = b;

Referenz-Richtlinie: ein referenziertes Objekt muss mindestens so lange leben, wie die Referenz (Bsp.: Objekt existiert nur innerhalb eines Blocks, oder Return-by-Reference, ohne referenzierte Aufrufargumente)

Vektoren erlauben eine beliebige Anzahl an Variablen des gleichen Typs zu initialisieren
-erforden #include <vector>
-Definition: **std::vector<datatype > vec_name (length, init_value);** (oder **"…" vec_name { 1, 2, 3, 4})**
-vec_name.at(i) [oder vec_name[i]] entspricht dem i-ten Element (Index) im Vektor (Zählung beginnt ab 0)

-vec_name.at(i) prüft vor Abruf die Indexgrenzen, vec_name[i] tut dies nicht (**.at** ist empfohlen)
Befehle: vec_name.push_back(a); vec_name.size()

Mehrdimensionale Vektoren (u.a. Matrizen) lassen sich initialisieren mit **std::vector<std::vector<int …>> name**

- axb-Matrix Bsp.: std::vector<std::vector<int>> matrix_name (a, std::vector<int>(b, 1));

Zugriff mit vec_name.at(i).at(j) oder vec_name[i][j]
→jeder Eintrag des Vektors wird selbst zum Vektor (usw.)

Using dient zur Neubenennung (Abkürzung) von langen Datentypen (z.B. Matrizen); wird zu beginn des Codes definiert (nach #includes)

Char ist ein Datentyp für einzelne Zeichen (formal für ganze Zahlen) ;
Literale werden mit einfachen Anführungszeichen geschrieben (’a‘)
(für strings werden doppelte benutzt: “vector“)
-jeder char hat eine, ihm zugeordnete, Zahl (gemäss ASCII-Code) Bsp.: ’a‘ == 97

Input / Output Streams bezeichnen abstrakte Ströme von chars
-benötigen **#include <iostream>**
-Funktionsaufruf hier mit caesar(std::cin, std::cout, s);
-**std::istream** ist ein allgemeiner Datentyp für Input-Streams
-**std::ostream** ist ein allgemeiner Datentyp für Output-Streams
-Streams können **nicht direkt kopiert** werden; in Funktionen sollten sie immer via Call-by-Reference weitergegeben werden

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

  in >> std::noskipws;

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

Datenströme von Datei zu Datei erfordern **#include <fstream>**
Std::ifstream ist ein Datentyp zum Auslesen einer Datei und kann nicht direkt kopiert werden (in Funktionen nur call-by-reference)

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

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

String ist ein komfortabler Datentyp für Zeichen; erfordert **#include <string>**
-kann initialisiert werden mit **std::string str_name (n, ’a‘);** (n: Länge, a: Initialwert)
(Klammer ist optional) Bsp.: std::string text = “Essen fertig!“
-Befehle: **str_name.size(); string1 == string2 ; string1 += string2; std::cout << string1;**
-analog zu Vektoren gibt es die Schreibweisen **string[i]** bzw. **string.at(i)** für das i-te Element

Datenströme von einer **String**variable aus erfordern zusätzlich **#include <sstream>**

```
std::string plaintext = "My password is 1234";
std::istringstream from(plaintext);

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

Falls von einem **Eingabestrom** auch die **Leerzeichen** (zu Beginn des Inputstreams) **gelesen** werden sollen, lässt sich der Befehl **std::noskipws** benutzen; Bsp.: std::cin >> std::noskipws;
-rückgängig mit std::cin >> ws;

Ein **leerer Eingabestrom** wird in bool zu **false** konvertiert; Dies erlaubt while-Schleifen, bis der Eingabestom zuende ist. Bsp.: **while (std::cin >> a)** { … } läuft bis keine Eingabe mehr.

Mit **my_stream.peek()** wird das nächste Zeichen in Stream aufgerufen, ohne es zu entfernen (im Datentyp int)
-Whitespaces werden nie ignoriert
Bsp.: int c = std::cin.peek();

Rekursive Funktionen brauchen immer einen garantierten Fortschritt in Richtung einer Abbruchbedingung (nach Selbstaufrufen)
-sonst wird ein **stack overflow** (Endlosschleife) riskiert
-daher ist neben dem rekursiven Teil immer ein Basisfall (Base-Case) notwendig

```
// POST: return value is n!
unsigned int fac(unsigned int n) {
 if (n <= 1)
    return 1;
  else
    return n * fac(n-1);
}
```

-Rekursion kann immer durch Schleifen und einen Aufrufstapel (nit gespeicherten Werten) simuliert werden
-Rekursion kann einfacher, aber auch weniger effizient sein

Formale Grammatiken geben Regeln für Zeichenfolgen vor (definiert eine "Sprache").
Eine solche Grammatik ist die **EBNF** (Extended Backus Naur Form):

-bei einem nicht-terminalen Symbol, muss ein Sprung zur entsprechenden Regel des Symbols gemacht werden
-hier entspricht eine Zeile einer Regel (Definition)

```
unsigned_integer = digits .
digit  = '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'.
digits = digit | digit digits.
```

Alternative
Terminales Symbol
Nicht-terminales Symbol

```
digits = digit { digit }.
```

Optionale Repetition

Bsp.: EBNF für Ausdrücke

```
factor     = unsigned_number
           | "(" expression ")"
           | "-" factor.

term       = factor { "*" factor | "/" factor }.

expression = term { "+" term |"-" term }.
```

Parsen bezeichnet das Feststellen, ob ein Satz nach einem EBNF gültig ist.
Ein **Parser** ist ein Programm zum Parsen; Aus jedem EBNF kann (fast automatisch) ein Parser gemacht werden
-**Regeln** (Definitionen) werden zu **Funktionen**
-**Alternativen** werden zu **If-Bedingungen**
-**Optionale Repetitionen** werden zu **While-Schleifen**
-**Nichtterminale Symbole** werden zu **Funktionsaufrufen**

Struct ist ein Container für Datentypen (definiert neuen Typ); Die Definition eines Structs hat immer ein **;**
-Allein der Zuweisungsoperator (=) wird automatisch erstellt. Der Rest muss selbst überladen werden
-Ein Struct hat verschiedene **Member (-variablen)** (mem1,..)
-Membervariablen können eine Invariante (als Kommentar) haben, die gültige Werte spezifiziert

Definition:	`struct str_name { int mem1; bool mem2; int mem3; };`
Objekt erstellen:	`str_name obj1;`
mit Startwerten:	`str_name obj2 = {3, true, 4};`
aus anderem Objekt:	`str_name obj3 = obj2;`
Zugriff auf Member:	`obj1.mem1`

In einem **Struct** sind Membervariablen und -funktionen standardmässig **nicht versteckt**.
In einer **class** ist standardmässig **alles versteckt**.

Class ist ein struct mit einem **private** und **public** Bereich (Kapselung); Eine Klasse besteht aus Daten und Funktionen (Member); Kapselung regelt deren **Zugriffskontrolle**; Auf den **private**-Teil können nur **Memberfunktionen** zugreifen
-**Memberfunktionen** erlauben kontrollierten Zugang zu privaten Membern; Die Deklaration erfolgt immer in der Klassendefinition; Die **Definition** der Funktion kann auch **extern** vorgenommen werden (dann ist für jede Funktion das Präfix **function_name::** (an den Namen der Memberfunktion) notwendig
  -in der Regel werden die **Klassendefinition** und **Funktionsdeklarationen** (class_name.h) von den **Definitionen** der Memberfunktionen (class_name.cpp) getrennt
-eine Memberfunktion wird aufgerufen mit **obj1.function_name(arg1, arg2);**
-**const Memberfunktion** verspricht, das implizite Argument (die **this**-Instanz) nicht zu verändern
  Geschrieben als: double get_value() **const** { … }
  -**const Objekte** dürfen nur const Memberfunktionen aufrufen
  -jede Memberfunktion hat einen Pointer **this** auf das implizite Argument (per Default)
    -auf Membervariablen des Objekts einer Klasse kann mit **this->mem1** (oder obj1.mem1 für structs) zugegriffen werden
    -auch *(this) ist gültig (L-Wert)

Konstruktoren sind Memberfunktionen einer Klasse, die den **Namen der Klasse** tragen (bei verschiedenen Argumenten kann sie auch mehrfach vorkommen); Konstruktoren müssen public sein; Membervariablen können auch im Funktionsrumpf definiert werden (nicht wie unten initialisiert)

-mit Deklaration **rational r (2, 3);**

```
class rational
{
public:
    rational (int num, int den)
        : n (num), d (den)
    {
        assert (den != 0);
    }
```

Initialisierung der Membervariablen

Funktionsrumpf.

alternative Konstruktoren:

rational r ;
rational r (2);

```
lic:
...
rational ()
   : n (0), d (1)
{}

rational (int num)
   : n (num), d (1)
{}
```

Leere Argumentliste

Leerer Funktionsrumpf

Für jeden Struct (ohne definierten Konstruktoren) gibt es einen **Default-Konstruktor** mit uninitialisierten Werten. Dieser lässt sich löschen mit **struct_name () = delete;** (zu den Memberfunktionen) → es können keine undefinierten Variablen deklariert werden

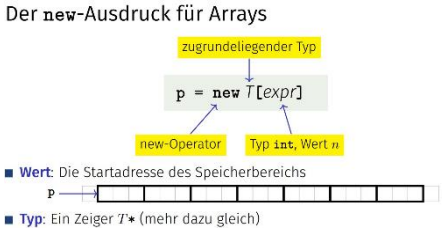
Operatorüberladung dient zur Erweiterung von Operatoren (z.B. +, *, /) auf mehr Datentypen (z.B. eigene Structs) zu erweitern; Dafür sind Funktionen mit Namen **operator*op*** notwendig (mit *op* als Operator)

-Bsp.: operator+ , operator<<

-anschliessend lässt sich der Operator (z.B. +) auf Variablen dieses Datentyps anwenden

Eine **Dynamische Datenstruktur** kann ihre Grösse zu Laufzeit ändern (Bsp.: Vektoren)
Dafür ist es notwendig einen **Speicherblock** von beliebiger Grösse zu **allozieren** (reservieren)
Bsp.: für Arrays der Länge n

Ein neuer zusammenhängender Speicherbereich
Für für n-Elemente vom Typ T wird alloziert
Mit dem **new-Ausdruck** für Arrays



Ein mit new allozierter Speicher muss immer mit **delete dealloziert** (freigegeben) werden

Der **new-Ausdruck** dient dem Erstellen eines Objekts dynamischer Lebensdauer. Dafür wird mit **new TYP(n)** (ohne Arrays) der nötige Speicher (von n-Plätzen) reserviert und der jeweilige Konstruktor aufgerufen; Der Rückgabewert von **new** ist ein **Pointer** auf das erste Element des neu erstellten Objekts; **new T** alloziert Speicher für ein einzelnes Objekt vom Typ T; Auch möglich ist die Schreibweise **new double{1, 2, 3}** (alloziert drei Speicherplätze mit Werten 1, 2, 3 vom Typ double)

Pointer sind Datentypen, die auf die Speicheradresse einer Variable mit dem gleichen Typ zeigen; werden deklariert mit type* name; Bsp.: int* p; (Zeiger auf einen Int)

-der Pointer muss immer auf seinen Datentypen zeigen

-der konkrete Wert eines Pointers ist die Adresse (Zahl im Hexadezimal) vom Typ T

Der **Adress-Operator &expr** gibt die Adresse eines Objekts aus; **&variable** gibt die Speicheradresse von variable; variable muss ein **L-Wert** sein

Bsp.: int* p = &number;

Der **Dereferenz-Operator *expr** gibt den Wert auf den ein Pointer zeigt zurück; ***p** gibt den Wert von number zurück; **expr** muss ein **R-Wert** (vom Typ type*) sein

Bsp.: int number = *p;

Die Adress- und Dereferenzoperatoren gleichen sich gegeseitig aus: *p == *(&*&*p)

Der **nullptr** ist ein spezieller Zeigerwert, der angibt, dass noch auf kein Objekt gezeigt wird (int* p = nullptr;); zeigt explizit ins Nichts

Zeiger-Arithmetik erlaubt das Iterieren eines Pointers entlang alloziertem Speicher; Bei alloziertem Speicher T* p = new T[n]; gibt ***(p + i)** den **Wert des i-ten Elements** an (alternativ gibt auch **p[i]** diesen Wert an)

-Zeigerverschiebung mit *(p + i) ist effizienter
-es gilt jedoch zu beachten, dass p + 1 nicht genau einen Speicherplatz weiter springt, sondern s-viele, wobei s der Speicherbedarf eines Objekts des jeweiligen Typs ist

Die **Übergabe eines Arrays** (oder Ausschnitts) erfolgt mit zwei Zeigern (begin und end); **begin** zeigt auf das erste Element und **end** zeigt hinter das letzte Element; gilt als leer, wenn begin == end

Const und Zeiger

Zeiger können selbst konstant sein oder auf konstante Objekte zeigen

Lies Deklaration von rechts nach links

`int const p;`	p ist eine konstante Ganzzahl
`int const* p;`	p ist ein Zeiger auf eine konstante Ganzzahl
`int* const p;`	p ist ein konstanter Zeiger auf eine Ganzzahl
`int const* const p;`	p ist ein konstanter Zeiger auf eine konstante Ganzzahl

***(this)** ist ein Zugriff auf das implizite Arugment einer Memberfunktion (das aufrufende Objekt); **this** ist ein Zeiger auf diese Instanz der Klasse

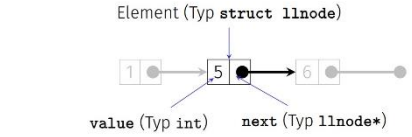
-bei Zugriffen **innerhalb** einer **Klasse**, wird das implizite Argument **automatisch** verwendet (this-> nicht notwendig)
-*this wird für Referenzen auf das implizite Argument zurückgegeben (L-Wert)

Beispiele für Dynamische Datenstrukturen sind der dynamische Vektor **avec** und die verkettete Liste **llvec**

-avec alloziert einen Speicherblock und definiert darauf einen Vektor
-llvec alloziert einzelne Blöcke, die aus einem Wert und Pointer auf das nächste Element der Liste bestehen

Verkettete Listen kein zusammenhängender Speicherbereich; Jedes Element zeigt auf seinen Nachfolger

Ein Vektor llvec besteht aus beliebig vielen llnodes;
Wobei **llvec** lediglich ein **Pointer** auf das **erste Element** der Liste ist;
Eine llnode kann beliebig hinzugefügt, oder entfernt werden;
Per Definition ist der Zeiger der letzten Node der **nullptr**




```
struct llnode {
  int value;
  llnode* next;

  llnode(int v, llnode* n): value(v), next(n) {} // Constructor
};
```

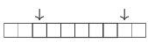
```
class llvec {
  llnode* head;
public: // Public interface identical to avec's
  llvec(unsigned int size);
  unsigned int size() const;
  ...
};
```

Entsprechend ist eine **Iteration** mittels einer for-Schleife für einen **llvec** möglich:
(n->value ist äquivalent zu n.value)

```
void llvec::print(std::ostream& sink) const {
  for (llnode* n = this->head;      // Zeiger auf erstes Element
       n != nullptr;                // Abbruch falls Ende erreicht
       n = n->next)                 // Zeiger elementweise voranschieben
  {
    sink << n->value << ' ';        // Aktuelles Element ausgeben
  }
}
```

Iteration in avec und llvec

- Iteration über ein *Array*:
  - Auf Startelement zeigen: `p = this->arr`
  - Auf aktuelles Element zugreifen: `*p`
  - Überprüfen, ob Ende erreicht: `p == this->arr + size`
  - Zeiger vorrücken: `p = p + 1`



- Iteration über eine *verkettete Liste*:
  - Auf Startelement zeigen: `p = this->head`
  - Auf aktuelles Element zugreifen: `p->value`
  - Überprüfen, ob Ende erreicht: `p == nullptr`
  - Zeiger vorrücken: `p = p->next`

Container sind Datenstrukturen für eine Ansammlung von Elementen; Sie organisieren Mengen auf verschiedene Weisen; jeder Container besitzt charakteristische Eigenschaften (Bsp.: avec und llvec haben verschiedene Vorteile) Jeder Container wird für verschiedene Anforderungen entwickelt

Ein **Iterator** geht geordnet über alle Elemente eines Containers. Für jeden Container sind folgende Iteratoren schon implementiert:

- `it = c.begin()`: Iterator aufs erste Element
- `it = c.end()`: Iterator *hinters* letzte Element
- `*it`: Zugriff aufs aktuelle Element
- `++it`: Iterator um ein Element verschieben

Bsp. Für Container sind std::vector, llvec, avec

Ein **Iterator** für **std::vector** kann aufgerufen (und abgekürzt) werden mit:

```
using ivit = std::vector<int>::iterator; // int-vector iterator

for (ivit it = v.begin();
    ...
```

Dies ermöglich **sequentielle Iteration** mittels eines Iterators über einen **llvec**:

```
llvec v(3); // v == {0, 0, 0}
for (llvec::iterator it = v.begin(); it != v.end(); ++it)
  std::cout << *it; // 000
```

Const-Iteratoren dienen dem Auslesen von Elementen eines Containers, sie verhindern jedoch das Verändern der Elemente (nur Lesezugriff, kein Schreibzugriff)

-ein solcher Iterator wird initialisiert mit **std::vector<int>::const_iterator name_it = vec_name.begin();**

-für einen **const-Vektor** muss ein const-Iterator verwendet werden

Der **delete-Ausdruck** dient dem Freigeben von Speicher, welcher mit new alloziert wurde; **jedes new braucht ein delete**
Denn, mit new erzeugte, Objekte haben **dynamische Speicherdauer** d.h. sie leben, bis sie explizit gelöscht werden

-wird ein Objekt mit delete gelöscht, sollten alle Pointer darauf auf den nullptr gesetzt werden, um dangling pointer (auf freigegebene Objekte) zu verhindern

-auch mehrfaches Deallozieren eines Objekts soll umgangen werden (undefined behavior)

-der delete-Ausdruck funktioniert analog mit **Arrays: delete[] expr** (mit expr als Pointer auf ein Array)

-bei Allokation mit **new type[n]** (für n-viele Speicherplätze)

Der **Destruktor** ist eine eindeutige Memberfunktion einer Klasse class mit der Deklaration: **~class();**
Er wird automatisch aufgerufen, sobald die **Speicherdauer** eines Objekts der Klasse **endet** (d.h. ein delete-Aufruf vom Typ class* oder wenn der Gültigkeitsbereich von dem Objekt endet)

-falls **kein Destruktor** deklariert ist, werden die **Destruktoren der Membervariablen** aufgerufen

-Problem bei z.B. llvec: nur der erste Zeiger der Liste wird dealloziert, der Rest der Liste aber nicht

-ein Destruktor für llvec müsste durch alle llnodes Iterieren und dabei jede llnode deallozieren

Mögliches **Problem**: bei Initialisieren eines Objekts aus llvec mit den Werten eines anderen llvec, darf nicht die Membervariable kopiert werden. In dem Fall entstehen zwei Pointer auf die gleiche Liste. Beim **Deallozieren** beider, versucht der zweite Vektor schon freigegebenen Speicher freizugeben (**Fehler**)

Lösung: eine **vollständige Kopie** der Link-Liste wird erstellt mithilfe des **Copy-Konstruktors**:

Im Copy-Konstruktor wird neuer Speicher alloziert, um ein neues Objekt der Klasse zu erzeugen

- Der Copy-Konstruktor einer Klasse *T* ist der eindeutige Konstruktor mit Deklaration

  T (`const` *T*`&` x);
- wird automatisch aufgerufen, wenn Werte vom Typ *T* mit Werten vom Typ `T` *initialisiert* werden

  `T x = t;` (t vom Typ T)
  `T x (t);`
- Falls kein Copy-Konstruktor deklariert ist, so wird er automatisch erzeugt (und initialisiert memberweise – Grund für obiges Problem)

Der Copy-Konstruktor wird jedoch nur bei Initialisierungen aufgerufen (int a = b;) aber nicht bei **Zuweisungen** (a=b;)
Der **Zuweisungsoperator** ist eine Operatorüberladung von **operator=** als **Memberfunktion**; löst das Problem für Zuweisungen (wenn passend implementiert)

-falls **kein Zuweisungsoperator** deklariert wurde, werden die Werte **memberweise** zugewiesen

Aufbau der **Memberfunktion**:
Es muss überprüft werden, dass keine Selbstzuweisung gemacht wird. Dann wird eine neue Variable (copy) initialisiert, sodass der Copy-Konstruktor aufgerufen wird. Anschliessend werden die Werte des impliziten Objekts vertauscht mit denen des neu initialisierten Objekts. Zurückgegeben wird der Wert des impliziten Objekts.

```
// POST: *this (left operand) becomes a
//           copy of s (right operand)
stack& stack::operator= (const stack& s){
  if (topn != s.topn){ // keine Selbstzuweisung
    stack copy = s; // Kopierkonstruktor
    std::swap(topn, copy.topn); // copy hat nun den Müll!
  } // copy wird aufgeräumt -> Dekonstruktion
  return *this; // Rueckgabe als L-Wert (Konvention)
}
```

Jeder **dynamische Datentyp** muss Konstruktoren besitzen;
Es gilt die **Dreierregel:** jede Klasse definiert entweder **Destruktor, Copy-Konstruktor und Zuweisungsoperator oder keines dieser Drei**