Informatik I

von P.Bölsterli, Fehler bitte an pbolsterli@student.ethz.ch

Basics

```
int main() {
  Programm
}
```

Ein- und Ausgabe:

Bibliothek:	`#include <iostream>`
Eingabe:	`std::cin >> var;`
Ausgabe:	`std::cout << "Yeet \n";`

Variablen

Name:	ree
Typ:	int
Wert:	i) undefiniert ii) 10
Adresse:	bestimmt Compiler

Deklaration:	`int ree;`
Definition:	`int ree = 10;`

Konstanten

Konstanten sind Variablen mit unveränderbarem Wert.

Präfix: `const`

```
const int c = 3e+8;
```

const-Richtlinie: Wenn eine Variable im Verlauf des Programmes seinen Wert nie ändert, dann sollte sie als Konstante definiert sein.

Fundamentale Datentypen

Typ	Was	Werte	Bits
`double`	Approximiert $\mathbb{R}$	$F^*(2, 53, -1022, 1023)$	64
`float`	Approximiert $\mathbb{R}$	$F^*(2, 24, -126, 127)$	32
`unsigned int`	Natürliche Zahlen	0 bis $2^{32} - 1$	32
`int`	Ganze Zahlen	-2^{31} bis $2^{31} - 1$	32
`bool`	Wahrheitswerte	true (1) oder false (0)	8
`char`	Zeichen (ASCII)	0 bis 255	8

Arithmetische Operationen: Gemischte Ausdrücke sind vom "allgemeineren" Typ (in der Tabelle der höhere Typ).

Konverter von Zahlen in ASCII (typ char) zum Typ int: x - '0'

Über- und Unterlauf

i) Über-/Unterlauf mit Typ *int* sind undefiniertes Verhalten, d.h. keine Fehlermeldungen, keine Garantien für das Ergebnis.

ii) Mit Typ *unsigned int* sind Über-/Unterlauf klar definiert, durch Rechnung modulo 2^B - 1. Beispiel: $0 - 1 = 2^{32} - 1$, bzw. $(2^{32} - 1) + 1 = 0$

std::string

Der Typ für Zeichenketten ist std::string. Ein paar Basics:

Bibliothek:	`#include <string>`
Initialisierung:	`std::string text = ''To'';`
Zugriff auf i-tes Zeichen:	`text.at(i);`
Konkatenieren:	`text += ''gether'';`
Länge abfragen:	`text.length();`

Fliesskommazahlen

Fliesskommazahlen besitzen 'Löcher' in ihrem Wertebereich, d.h. es wird immer auf die nächste darstellbare Zahl gerundet.

Normalisierte Darstellung

β	Basis ($\beta \geq 2$)
p	Präzision (Stellenanzahl)
$\mathbf{e}_{min}$	Kleinster mögliche Exponent
$\mathbf{e}_{max}$	Grösster mögliche Exponent

$$F^*(\beta, p, e_{min}, e_{max})$$
$$\pm d_0.d_1 \dots d_{p-1} \cdot \beta^e$$
$e \in \{e_{min}, \dots e_{max}\}$, $d_0 \neq 0$

Beispiel $F^*(2, 3, -2, 2)$ (nur positive Zahlen)

$d_0 \bullet d_1 d_2$	$e=-2$	$e=-1$	$e=0$	$e=1$	$e=2$
1.00_2	0.25	0.5	1	2	4
1.01_2	0.3125	0.625	1.25	2.5	5
1.10_2	0.375	0.75	1.5	3	6
1.11_2	0.4375	0.875	1.75	3.5	7

Fliesskomma-Richtlinien

i) Teste keine Fliesskommazahlen auf Gleichheit.
ii) Addiere keine Fliesskommazahlen sehr unterschiedlicher Grösse.
iii) Subtrahiere keine Fliesskommazahlen ähnlicher Grösse.

Literale

Literale besitzen einen festen Wert und Typ. Verschiedene Literale:

int:	10	double:	20.0
unsigned int:	10u	float:	20.0f
char:	'a'	string:	"Help me"

Verschiedene Zahlendarstellungen

Präfixe:

Hexadezimal:	`0x`
Oktal:	`0`
Binär:	`0b`

Dezimal:	`255`
Hexadezimal:	`0xff`
Oktal:	`0377`
Binär:	`0b11111111`

L-Wert und R-Wert

L-Wert:

i) besitzt Adresse
ii) Wert veränderbar
iii) Bsp: Variablen

R-Wert:

i) besitzt *keine Adresse*
ii) Wert konstant
iii) Bsp: Literale

Links vom Zuweisungsoperator. **R**echts vom Zuweisungsoperator.

L-Werte können als R-Werte benutzt werden, aber nicht umgekehrt!

Assertions

Assertions sind ein simples Tool um Laufzeitfehler aufzufangen.

1: Assertions abschalten
2: Bibliothek
3: Programm wird angehalten, wenn `expr` nicht wahr ist.

```
1  #define NDEBUG
2  #include <cassert>
3  assert(expr);
```

Operatoren

Die Auswertung erfolgt nach Präzedenz, beginnend mit Präzedenz 17!

Operator	Symbol	Präz.	Assoz.	L/R-Werte
Arithmetische Operatoren				
Negation	`-`	16	rechts	R → R
Multiplikation	`*`	14	rechts	RxR → R
Division	`/`	14	links	RxR → R
Modulo	`%`	14	links	RxR → R
Addition	`+`	13	links	RxR → R
Substraktion	`-`	13	links	RxR → R
Zuweisung	`=`	4	rechts	R → L
In-/Dekrement Operatoren				
Post-Inkrement	`expr++`	17	links	L → R
Post-Dekrement	`expr--`	17	links	L → R
Prä-Inkrement	`++expr`	16	rechts	L → L
Prä-Dekrement	`--expr`	16	rechts	L → L
Relationale Operatoren				
Kleiner als	`<`	11	links	RxR → R
Grösser als	`>`	11	links	RxR → R
Kleiner gleich	`<=`	11	links	RxR → R
Grösser gleich	`>=`	11	links	RxR → R
Gleich	`==`	10	links	RxR → R
Ungleich	`!=`	10	links	RxR → R
Logische Operatoren				
Logisches AND	`&&`	6	links	RxR → R
Logisches OR	`||`	5	links	RxR → R
Logisches NOT	`!`	16	rechts	R → R
Ein- und Ausgabe				
Eingabeoperator	`>>`	12	links	L → L
Ausgabeoperator	`<<`	12	links	R → L
Zeiger				
Adressoperator	`&`	16	rechts	L → R
Dereferenz-Operator	`*`	16	rechts	L → R

Ergänzungen: Division (/) und Modulo (%)

i) Falls zwei Ganze Zahlen (`int` oder `unsigned int`) dividiert werden, so rundet der Operator ganzzählig!

ii) Modulo gibt es nur für `int` und `unsigned int`. Bei negativen Zahlen übernimmt `%` das Vorzeichen des linken Operanden.

Kurzschlussauswertung

Logische Operatoren werten den linken Operanden zuerst aus. Steht das Ergebnis schon fest, wird der rechte Operand *nicht* mehr ausgewertet.

Richtlinie

Vermeide das Verändern von Variablen, welche im selben Ausdruck noch einmal verwendet werden!

Sonstiges

Standardbibliothek: `#include<cmath>`

Was	Code	Beschreibung
Potenzieren	`std::pow(2.5,2);`	$2.5^2 = 6.25$
Quadratwurzel	`std::sqrt(14.0625);`	$\sqrt{14.0625} = 3.75$
Absolutbetrag	`std::abs(-3);`	$\lvert -3 \rvert = 3$
Minimum	`std::min(z, 1.0);`	Minimum zweier Argumente *gleichen Typs*
Maximum	`std::max(z, 1.0);`	Maximum zweier Argumente *gleichen Typs*

Funktionalität(en) auslagern

i) Funktionen und Typen in eigene Datei schreiben (test.cpp).
ii) Dateien ins Arbeitsverzeichnis der Hauptdatei speichern.
iii) In der Hauptdatei Header inkludieren: `#include ''test.cpp''`.

Nachteil: Compiler muss Funktionsdefinition immer neu übersetzen (Zeitaufwand).

Getrennte Übersetzung

i) Funktionen und Typen in eigene Datei schreiben (test.cpp).
ii) test.cpp seperat kompilieren ⇒ test.o.
iii) Deklaration aller benötigten Symbole in Headerdatei (test.h).
iv) Headerdatei im Arbeitsverzeichnis speichern.
v) In der Hauptdatei inkludieren: `#include ''test.h''`.

Vorteil: Quellcode muss nur einmal übersetzt werden und ist vor Einblick geschützt.

Kontrolflussanweisungen

Blöcke

Ein Block gruppiert mehrere Anweisungen zu formell einer Anweisung. Normalerweise wird überall ein Block verwendet, anstelle von einer einzigen Anweisung.

```
{
  statements
  ⋮
}
```

Gültigkeitsbereich von lokalen Variablen

i) Variablen, die in einem Block deklariert sind ("lokale Variablen"), sind ausserhalb des Blocks nicht gültig. Kontrollanweisungen verhalten sich in diesem Zusammenhang wie Blöcke.

ii) Haben mehrerer Variablen den gleichen Namen, dann wird immer die Variable mit der am stärksten verschachtelten Deklaration gewählt.

if-else Anweisung

1) Ist `condition` wahr, dann wird `statement1` ausgeführt.

2) Ist `condition` falsch, dann wird `statement2` ausgeführt.

```
1  if (condition)
2    statement1
3  else
4    statement2
```

for Schleife

Eine for-Schleife iteriert eine Anweisung mit einer Laufvariable, solange `condition` erfüllt ist.

```
1  for (init statement; condition; expression)
2    body statement
```

`init statement`:	Deklaration Laufvariable	(Bsp: `int i = 0`)
`condition`:	Iterationsbedingung	(Bsp: `i < 10`)
`expression`:	Manipulation Laufvariable	(Bsp: `++i`)

Die Manipulation der Laufvariable wird *immer am Ende* der jeweiligen Iteration durchgeführt.

while Schleife

Eine while-Schleife iteriert eine Anweisung, solange `condition` erfüllt ist.

`condition`: Iterationsbedingung

```
1  while (condition)
2    statement
```

do-while Schleife

Einziger Unterschied zur while Schleife ist, dass die Anweisungen mindestens einmal ausgeführt werden. Dies ist so, da `condition` erst am Ende der Iteration geprüft wird.

`condition`: Iterationsbedingung

```
1  do
2    statement
3  while (condition)
```

Sprunganweisungen

1: Die umschliessende Iterationsschleife wird sofort beendet.
2: Überspringt den Rest der Anweisungen der jetzigen Iteration. Die Iteration wird aber **nicht** abgebrochen!

```
1  break;
2  continue;
```

switch Anweisung

Die switch Anweisung eignet sich um eine Fallunterscheidung zu treffen. Der Ausdruck `expr` wird ausgewertet und springt, falls ein entsprechender Fall vorhanden ist, zu diesem, ansonsten zum `default`-Fall.

`expr`	Muss zu `int` konvertierbar sein!
`case`	Definiert einen Fall mit einem *konstanten* int Wert (hier 1).
`break;`	Springt aus dem switch-case
`default:`	Standardfall

```
switch (expr) {
  case 1:
    statement
    break;
  default:
    statement2
}
```

Wird ein Fall nicht mit `break;` beendet, werden darunterliegende Fälle auch ausgeführt (genannt "durchfallen"), bis ein `break;` erreicht wird.

Funktionen

Definition

Eine Funktionsdefinition gehört oberhalb von `main()` hin. Die Argumente einer Funktion verhalten sich wie lokale Variablen im Funktionsblock.

`T`	Rückgabetyp
`fname`	Funktionsname
T_i	Argumenttypen
$pname_i$	Argumente

```
T fname(T1 pname1,...,Tn pnamen){
  statements
  return expr;
}

int main() {
  /* Funktionsaufruf */
  fname(expr1, ..., exprn);
}
```

Der Rückgabetyp kann auch vom Typ `void` sein, welcher der Typ mit leerem Wertebereich ist, d.h. keine Rückgabe.

Rückgabewert

Jede Funktion, die nicht den Rückgabetyp `void`, hat, **muss** ein `return` besitzen. Der Rückgabewert wird immer zum Rückgabetyp konvertiert und ergibt den Wert des Funktionsaufruf.

Vor- und Nachbedingungen

Es ist sinnvoll Funktionen gut zu dokumentieren mit PRE-/POST-Conditions (Vor- und Nachbedingungen).

Vorbedingungen:	Spezifiziert den Definitionsbereich der Funktion. - Sollten immer mit asserts geprüft werden.
Nachbedingungen:	Spezifiziert Wert und Effekt der Funktion. - Kann oft einfach mit asserts überprüft werden.

Forward Declarations

Eine Funktion kann im Programm nur in Zeilen verwendet werden, welche nach der Deklaration der Funktion kommen. Eine Forward Declaration ist einfach der Funktionskopf ohne den Funktionskörper:

```
T fname(T1 pname1,...,Tn pnamen);
```

Namensräume (namespace)

Mit namespaces kann man Funktionen, Typen, etc. katalogisieren. Ausserdem kann man bei grösseren Projekten mit namespaces verschiedenste Namenskonflikte verhindern.

```
namespace boring{ // neuer Namensraum
  void func() // gehört zum Namensraum 'boring'
}
main() {
  boring::func(); // Zugriff auf Element im Namensraum
}
```

Vektoren

Vektoren dienen zum Speichern gleichartiger Daten (gleicher Typ). Ein Vektor kann aus beliebigen Typen bestehen.

Initialisierung

```
auto vec = std::vector<int>(5);
    - Vektor mit 5 Elementen auf Wert 0 initialisiert.
auto vec = std::vector<int>(5,2);
    - Vektor mit 5 Elementen auf Wert 2 initialisiert.
std::vector<int> vec = {3,2,1};
    - Vektor wird mit einer Liste von Werten initialisiert.
std::vector<int> empty;
    - Deklariert und initialisert einen leeren Vektor.
```

Funktionalität

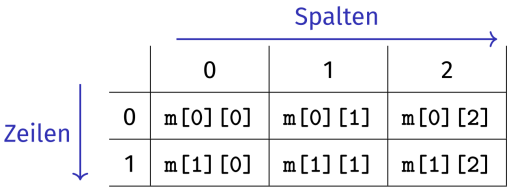
Nicht vergessen, dass der Index bei Vektoren mit 0 beginnt!

Bibliothek:	`#include<vector>`
Zugriff auf Element `i` ohne Index-Check:	`vec[i]`
Zugriff auf Element `i` mit Index-Check:	`vec.at(i);`
Fügt `x` hinten an:	`vec.push_back(x);`
Die Länge vom Vektor:	`vec.size();`
Ändert die Länge zu `v`:	`vec.resize(v);`
Prüft, ob Vektor leer ist:	`vec.empty();`
Löscht den Inhalt:	`vec.clear();`

Mehrdimensionale Vektoren

Funktioniert im Prinzip gleich wie ein 1D-Vektor.

Leere Matrix:	`std::vector<std::vector<int>> matrix;`
Initialisierung:	`auto matrix = std::vector<std::vector<int>>( rows, std::vector<int>(cols, value));`
Definition:	`std::vector<std::vector<int>> matrix( rows, std::vector<int>(cols, value));`
Zugriff:	`matrix.at(i).at(j);`



Einschub: Typ-Alias

Für lange Typen kann es sich lohnen eine Typ-Alias zu definieren:

Allgemein:	`using newName = Typ;`
Beispiel:	`using matrix = std::vector<std::vector<int>>;` `matrix m = ...;`

Referenzen

Eine Referenz funktioniert wie ein Alias für ein anderes Objekt. Intern ist eine Referenz die Adresse vom referenzierten Objekt, d.h. eine Zuweisung auf eine Referenz ändert das ursprüngliche Objekt.

Referenz vom Typ T:	`T& rname = lvalue;`
const-Referenz vom Typ T:	`const T& rname = rvalue;`

Const-Referenzen

Eine const-Referenz setzt **nicht** vorraus, dass das ursprüngliche Objekt auch const ist. Es sagt einfach aus, dass man über die Referenz nur *read-only* Zugriff auf die ursprüngliche Variable hat.

Bemerkung: Eine const-Referenz kann ausserdem mit einem R-Wert initialisiert werden, in diesem Fall erzeugt der Compiler ein temporäres Objekt mit ausreichender Lebensdauer.

Anwendung: Pass by Reference

Referenzen sind sehr nützlich um Funktionen nicht fundamental Datentypen wie Vektoren zu übergeben, da sie es erlauben die Werte des Aufrufargument zu ändern.

```
1  void func(std::vector<int>& i) {
2    statements
3  }
```

Pass by const Reference (`const T&`)

Überall wo es möglich ist, sollte man nicht fundamentale Typen als `const T&` übergeben. Dies spart Ressourcen und stellt klar, dass man das Argument nicht mit der Funktion verändert.

Pass by Value

Bei Pass by Value, d.h. das Argument hat kein Referenztyp, wird eine *Kopie des Objekts* übergeben.

Anwendung: Return by Reference

Return by Reference ermöglicht es, dass der Rückgabewert ein L-Wert ist anstelle von einem R-Wert.

```
1  int& increment(int& i) {
2    return ++i;
3  }

4  increment(increment(x));
```

Eine Funktion, welche eine Referenz als Argument nimmt, kann man nur mit sich selbst als Argument anwenden (Zeile 4), falls der Rückgabetyp auch eine Referenz ist. Sonst gibt die Funktion im Argument einen R-Wert zurück, obwohl ein L-Wert benötigt wird.

Referenz-Richtlinie

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

Ein- und Ausgabeströme

Generische Ströme (Abstrakter Typ)

Man muss für eine Funktion nicht unbedingt genau spezifizieren, was für eine Eingabeart (Konsole, File, etc.) notwendig ist. Man interagiert mit den generischen Strömen auch über den Eingabe- und Ausgabeoperator.

```
#include <iostream>
void func(std::istream& in, std::ostream& out)
```

Man muss die Argumente als normale Referenz deklarieren, da sie den Eingabestrom bzw. Ausgabestrom verändern durch lesen bzw. schreiben.

Filestream

Ein Beispiel mit Dateien zur Ein- und Ausgabe:

```
#include <fstream>
std::ifstream from = std::ifstream(''source.txt'');
std::ofstream to = std::ofstream(''destination.txt'');

int main ()
    func(from, to);
```

Stringstream

Ein weiteres Beispiel für einen Eingabestrom ist der Stringstream:

```
#include <sstream>
std::string text = ''Text!'';
std::istringstream from = std::istringstream(text);

int main ()
    func(from, std::cout);
```

Whitespaces

Per default ignorieren die Eingabeströme Whitespaces (Leerschläge, Zeilenumbrüche, Tabs, etc.). Dies kann man durch `std::noskipws` ändern (Zeile 1), nichtdestotrotz kann man Whitespaces entfernen durch `std::ws` (Zeile 2).

```
1  std::cin >> std::noskipws;
2  std::cin >> std::ws;
```

Eingabestrom als Bedingung für Kontrollanweisungen

Ist eine Eingabe im Eingabestrom vorhanden, dann kann man den nächsten Wert des Eingabestroms in eine Variable auslesen und die Bedingung evaluiert als 'true'.
Sind keine Eingaben mehr vorhanden im Eingabestrom, dann evaluiert die Bedingung als 'false'.

```
char input;
while(std::cin >> input)
  ...
```

Dasselbe geht natürlich auch für z.B. eine if-else-Bedingung!

Rekursion

Rekursion passiert, wenn eine Funktion sich selbst aufruft. Der Grundaufbau einer (funktionierenden) Rekursion ist, wie folgt:

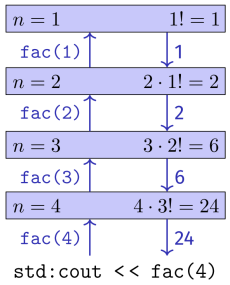
- **Basecase**: Abbruchsbedingung, welche die Rekursion stoppen soll. Grundsätzlich muss sich die Rekursion diesem Fall stetig annähern, ansonst gibt es eine unendliche Rekursion und einen *stack overflow*.
- **Rekursiver Teil**: Hier wird die Aufgabe der Funktion gelöst durch einen Aufruf der Funktion selbst, d.h. die Funktion führt etwas aus und ruft sich dann selbst auf (oder umgekehrt).

Der Aufrufstapel

Jeder rekursive Funktionsaufruf hat seine eigenen, unabhängigen Variablen und Argumente. Darstellung als Aufrufsstapel:

```
int fac(int n) {
  if (n <= 1)
    return 1;
  else
    return n * fac(n-1);
}

...
std::cout << fac(4);
```



Anwendungsbeispiele

Ein einfacheres Beispiel:

```
// PRE: -
// POST: print the digits of the input in reverse order to std::cout
void reverse(unsigned int n)
{
  /* Base Case: Nur noch eine Ziffer vorhanden */
  if(n < 10) {
    std::cout << n; // Gibt die allerletzte Ziffer aus
    return;         // keine weitere Rekursion
  }

  /* Rekursiver Teil der Funktion */
  std::cout << n % 10;  // Ausgabe der momentan letzten Ziffer
  reverse(n / 10);      // Rekursiver Aufruf ohne die letzte Ziffer
}
```

Ein etwas komplexeres Beispiel:

```
// PRE: -
// POST: Returns the number of valid paths through mat starting at (0, 0).
//       Valid paths can only move right or down.
int DFS(const std::vector<std::vector<bool>>& mat,
        unsigned int row, unsigned int col)
{
  const unsigned int n = mat.size();        // m x n Matrix
  const unsigned int m = mat.at(0).size();

  /* Both Basecases */
  if (row >= n || col >= m || mat.at(row).at(col))
    return 0; // Found a wall OR out of bounds
  if (row == n - 1 && col == m - 1)
    return 1; // Reached the bottom right corner

  /* Rekursiver Teil, überprüft beide möglichen Wege */
  return DFS(mat, row + 1, col) + DFS(mat, row, col + 1);
}

int valid_paths(const std::vector<std::vector<bool>>& mat)
{
    return DFS(mat, 0, 0); // Start of recursion
}
```

Extended Backus Naur Form (EBNF)

EBNF ist eine formale Grammatik. Man kann sie benutzen um Sätze zu produzieren oder um Sätze zu Parsen (überprüfen).

Sätze

Sätze bestehen aus folgenden Bestandteilen. Der Unterschied zwischen nicht-terminalen Symbolen und terminalen Symbolen ist, dass terminale Symbole keine weitere Funktionsaufrufe enthalten.

Nichtterminale Symbole:	Kann weiter zerlegt werden
Terminale Symbole:	Kann nicht weiter zerlegt werden
Alternative:	|
optionale Repetition:	{ . . . }
optional Einmal:	[. . .]
Satz beendet:	.

Beispielssatz: factor = number | ”(” expression ”)” | ”-” factor.

In Worten: *factor* besteht aus *number* ODER dem Zeichen ”(” gefolgt von *expression* und ”)” ODER dem Zeichen ”-” und *factor* selbst.

Parsing

Parsing ist das überprüfen, ob und wie ein Satz aus einer formellen Grammatik erzeugt werden kann. Grundsätzlich gilt:

	EBNF	Parser
Satz:	Funktion
Alternative:	if-Anweisung
Nichtterminales Symbol:	Funktionsaufrufe
optionale Repetition:	while-Anweisung

Einschub: peek() (`#include <iostream>`)

Die Funktion peek() erlaubt es das nächste Zeichen im Inputstream anzuschauen, *ohne es zu entfernen*. Dies ist notwendig für Parsing.

```
std::istringstream s1(''Hello, world.'');
char c1 = s1.peek(); // c1 = 'H'
```

Die Funktion ignoriert Whitespaces *nie* (unabhängig von std::noskipws).

Einschub: `lookahead(std::istream& is)`

Diese Funktion gibt den nächsten *non-whitespace* Charakter vom Eingabestrom, d.h. es funktioniert wie peek, aber es ignoriert Whitespaces.

Einschub: consume(std::istream& is, char expected)

Diese Funktion findet man in **`ebnf_helpers.h`**. Es ist wichtig zu überprüfen, welche Variante der Funktion drinnen ist, da es bisher zwei verschiedene gab. Die neuere Variante hat folgende Charakterisierung:

- Liest den nächsten Charakter vom Inputstream und gibt *True* zurück, falls es der gleiche Charakter ist, wie der Charakter im Argument der Funktion. Ausserdem wird der Charakter vom Inputstream **entfernt**.
- Ist es nicht der gleiche Charakter, dann gibt die Funktion *False* zurück und der Charakter wird in den Inputstream **zurückgelegt**!

Bem: Bei der älteren Variante ist der grosse Unterschied, dass der Charakter beim Fall *False* nicht zurückgelegt wird in den Inputstream.

Anwendung: Codebeispiele von Parsing

Für die *neuere* Version von `consume()` gibt es folgendes Beispiel:

```
/* Consumes next Var = "_" Var | Letter { Letter }. */
bool Var(std::istream& is) {
  if (consume(is, '_')) { // Prüft 1ste Alternative
    return Var(is);
  } else {                // Prüft 2te Alternative
    if (!Letter(is))
      return false;
    while (Letter(is)); // Prüft optionale Repetition
  }
  return true;  // Alles i.O.
}

/* Consumes next VarList = Var { "," Var } ":" Type. */
bool VarList(std::istream& is) {
  if (!Var(is))   // Prüft Var
    return false;
  while(consume(is, ',')) { // Prüft optionale ..
    if (!Var(is)) return false;   // ..Repetition
  }
  if (!consume(is, ':'))
    return false;
  if (!Type(is))  // Prüft Type
    return false;

  return true;  // Alles i.O.
}

/* Consumes next Decl = "var" VarList ";". */
bool Decl(std::istream& is){
  if (!VarDec(is))  // Prüft String "var"
    return false;
  if (!VarList(is)) // Prüft VarList
    return false;
  if (!consume(is, ';'))
    return false;

  return true;  // Alles i.O.
}
```

Für die *ältere* Version von `consume()` gibt es folgendes Beispiel:

```
/* Satz: MiddleCars = ("P" | "PP" | "PPP") "D". */
bool MiddleCars(std::istream& is) {
  unsigned int counter = 0;
  while(lookahead(is) == 'P' && counter < 3) {
    consume(is,'P');  // Prüft alle 3 Alternativen
    ++counter;
  }
  if(consume(is,'D') && counter > 0)
    return true;  // Prüft ob Satz mit "D" endet

  return false; // Nur inkorrekte Sätze enden hier!
}

/* Satz: Train = "E" {BoxCars} {MiddleCars} ServiceCars. */
bool Train(std::istream& is) {
  if (consume(is, "E")) { // Prüft Satzbeginn mit "E"
    while(lookahead(is) == 'B'){ // Prüft optionale Boxcar
      if(!BoxCars(is))  // Prüft Satz: BoxCars = {"BB"}.
        return false;
    }
    while(lookahead(is) == 'P'){ // Prüft optionale MiddleCars
      if(!MiddleCars(is))
        return false;
    }
    if(!ServiceCars(is))  // Prüft ServiceCars
      return false;

    return lookahead(is) == 0; // 'True' wenn der Satz zuende ist.
  }
  return false; // Nur inkorrekte Sätze enden hier!
}
```

Structs & Klassen

Structs

Structs sind Container für Datentypen. Die zugrundeliegende Typen können fundamentale aber auch benutzerdefinierte Typen sein.

Definition:	`struct strName { int mem1; bool mem2; };`
Default-Initialisierung:	`strName obj1;`
Initialisierung mit Startwerten:	`strName obj2 = {3,true};`
Initialisierung mit anderem Objekt:	`strName obj3 = obj2;`
Zugriff auf Member:	`obj1.mem1 = 10;`

Achtung: Bei Default-Init werden alle Member einzeln default-initialisiert, dies bedeutete für fundamentale Typen, dass sie uninitialisiert sind. Zugriff bevor einer Zuweisung ist undefined behavior!

Überladen von Funktionen

Es ist möglich mehrere Funktionen gleichen Namens zu definieren und deklarieren. Die 'richtige' Version wird aufgrund der Signatur der Funktion (Typ, Anzahl und Reihenfolge der Argumente) ausgewählt.

Funktion:	`int pow(int b, int e)` { ... }
Bsp. einer Überladung:	`int pow(int e)` { return pow(2,e); }

Achtung: Die Argumentnamen haben *keinen* Einfluss auf die Signatur!

Operator overloading

Operatoren (`+, -, *, /, +=, <<`, etc.) können auch überladen werden und so für eigene Structs und Classes definiert werden.

```
cname operator+(cname a, cname b)
cname operator-(cname a)
cname& operator+=(cname& a, cname b)
std::ostream& operator<<(std::ostream& o, const cname& r)
std::istream& operator>>(std::istream& in, cname& r)
```

Klassen

Eine Klasse erlaubt Kapselung via Zugriffkontrolle (private, public). Nur auf public Member kann von aussen zugegriffen werden. Der Gültigkeitsbereich von Membern in einer Klasse ist die ganze Klasse, unabhängig von der Deklarationsreihenfolge.

```
class className {
public:
  int getHidden();
private:
  int hidden;
};
```

Einziger Unterschied gegenüber structs ist, dass Members in structs per default public sind, während sie in classes per default private sind.

Memberfunktionen

Memberfunktionen stellen die Funktionalität einer Klasse bereit. Die Deklaration erfolgt immer in der Klassendefinition, die Definition der Memberfunktion ist auch extern möglich über den ::Operator.

```
int className::getHidden() {return hidden; }
```

Aufruf einer Memberfunktion:

```
className.gethidden();
```

const Memberfunktionen

Man kann durch `const` in der Funktionsdefinition versprechen, dass eine Memberfunktion nichts an der Objektinstanz verändert:

```
int className::getHidden const() {return hidden; }
```

Konstruktoren

Konstruktoren sind spezielle Memberfunktionen einer Klasse, die den Namen der Klasse tragen. Sie werden bei der Variablendeklaration aufgerufen (müssen public sein).

Konstruktor:	`className(int a): hidden(a)` { ... }
Default-Konstruktor:	`className(): hidden(0)` { ... }
Objektinitialisierung:	`auto myclass = className();`
Default-Konstruktor löschen:	`className() = delete;`

Ein Default-Konstruktor (kein Argument) wird automatisch erzeugt, falls eine Klasse keinen Konstuktor definiert hat.

`this` Zeiger

Member einer Klasse haben ein implizites Argument, nämlich das aufrufende Objekt. Mittels dem `this` Zeiger kann man explizit auf das Aufruferobjekt zugreifen.

Implizit:	`return hidden;`
Explizit umständlich:	`return (*this).hidden;`
Explizit:	`return this->hidden;`

Dies macht nur Sinn, falls der zu zugreifende Name lokal nicht eindeutig ist oder wenn man z.B. eine Referenz auf das implizite Aufruferobjekt zurückgeben will:

```
Complex& operator+= (const Complex& b) {
  real += b.real; // Manipulierung implizites Aufrufobjekt
  imag += b.imag; // Manipulierung implizites Aufrufobjekt
return *this; }
```

Allgemeine Ergänzungen

Pointy Vektor

Ist ein Funktionsargument: `std::vector<int>* v`, so funktioniert:

i) `v->size();`
ii) `v->at(i);`

Substrings von Strings

Mit der Funktion `substr(pos, count)` der Klasse `std::string` kann man einen beliebigen Substring vom String nehmen. Beispiel:

```
std::string test = "Hello";
std::cout << test.substr(2,3); // Ausgabe: llo
```

Gibt man der Funktion nur `pos` (Startposition) ohne `count` (Anzahl Charakter), dann gibt sie den ganzen Rest vom string aus.

Zeiger (Pointers)

Ein Zeiger enthält die Adresse eines Speicherplatzes.

Variable vom Typ `T`:	`T var;`
Zeiger vom Typ `T*`:	`T* ptr = & var;`
Uninitialiserter Zeiger:	`T* ptr = nullptr;`
Zugriff auf Speicherinhalt:	`std::cout << *ptr;`
Memberzugriff über Pointer:	`pointer->member;`

Das Dereferenzieren eines `nullptr` führt zu undefined behavoir!

Const und Zeiger

Am besten liest man die Deklaration von rechts nach links:

`int const* p2;`	Zeiger auf eine konstante Ganzzahl
`int* const p3;`	const-Zeiger auf eine Ganzzahl
`int const* const p4;`	const-Zeiger auf eine konstante Ganzzahl

Bem: Ein Zeiger auf `const T` kann auch auf `T` zeigen. Der Zeiger behandelt `T` dann einfach wie eine Konstante (Wie bei Referenzen).

Array

Dynamisch allozierte Liste, welche von der Grösse `size` ist und einen Zeiger zum ersten Element zurückgibt. Der Speicher dafür alloziert man selbst *und muss ihn auch selbst wieder deallozieren.*

Speicher allozieren:	`T* ptr = new T[size];`
Speicher deallozieren:	`delete[] ptr;`

Zeigerarithmetik

Im Prinzip gleich wie bei Variablen, d.h. `++ptr` führt zu einer Inkrementation vom Pointer und z.B. `ptr + 3` ändert den Pointer an sich nicht.

Array:	`int* ptr = new int[3]{1,2,3};`
Beispiel 3. Element:	`int* ptr2 = ptr + 2; // Zeigt auf '3'`
Wahlfreier Zugriff:	`*(ptr + 2) == ptr[2];`

Bem: Achten, dass kein 'Out of bounds' Zugriff stattfindet! Wahlfreier Zugriff ist 'teurer' (mehr Operationen), als einen Zeiger zu iterieren.

Anwendung: Sequentielle Iteration

Normaler Anwendungsfall für Zeigerarithmetik ist Iteration eines Array:

```
int* ptr = new int[3]{1,2,3};
for (int* itr = ptr; itr != ptr + 3; ++itr)
  std::cout << *itr << ' ';
```

Konvention: Übergabe eines Arrays an Funktionen

Es ist Konvention immer zwei Zeiger zu übergeben, einer zum ersten Elemente und einer zum Element **nach** dem letzten.

```
int* function(int* begin, int* end) { }
int* ptr = new int[3]{1,2,3};
function(ptr, ptr + 3); // ptr+3 ist Element nach dem letzten
```

Dynamische Allokation

Objekte welche mit dem `new` Operator erstellt werden, leben solange, bis sie mit `delete` gelöscht werden (dynamischer Lebensdauer).

Speicher allozieren:	`myclass* ptr = new myclass();`
Speicher deallozieren:	`delete ptr;`

Richtlinie: Wird ein Objekt gelöscht, dann sollte man alle Pointer, welche auf dieses Objekt zeigen, auf `nullptr` setzen.

Memory leaks

Wird ein Element nicht korrekt durch `delete` gelöscht, führt dies zu Speicherlecks, im schlimmsten Fall zu einem Heap Overflow.

Hängende Zeiger (dangling pointers)

Dereferenzieren oder deallozieren von einem Zeiger, welcher auf ein freigegebenes Objekt zeigt führt zu einem Programmabstürz, da man auf nicht allokierten Speicher zugreift.

Die Dreierregel

Klassen, welche dynamisch allokierte Membervariablen besitzen, sollten alle der folgenden drei Funktionen definiert haben.

Destruktor

Wird implizit aufgerufen, wenn die Speicherdauer eines Objekts endet.

```
~className();
```

Wenn kein Destruktor definiert ist, wird er automatisch erzeugt und ruft die Destruktoren für die Membervariablen auf.

copy-Konstruktor

Wird bei jeder Initialisierung, auch bei *pass-by-value*, aufgerufen.

```
className(const className& other) : var(...), ... {
  ... // Dynamische Membervariable(n) allokieren
  ... // Alle Variablen kopieren
}
```

Falls kein Copy-Konstruktor deklariert ist, wird er automatisch erzeugt mit memberweiser initialisierung.

Zuweisungsoperator

Wird bei allfälligen Zuweisungen *nach* der Initialisierung aufgerufen.

```
stack& stack::operator=(const stack& s) {
  if (topn != s.topn) { // Keine Selbstzuweisung
    stack copy = s; // Aufruf copy-Konstruktor
    std::swap(topn, copy.topn); // copy hat nun den Müll
    ...;      // Bei mehr Variablen, mehr swaps
  } // copy wird aufgeräumt
  return *this; // Rückgabe als L-Wert (Konvention)
}
```

Einschub: std::swap (Bibliothek `#include<algorithm>`)

Inhalte von zwei Variablen einfach tauschen: `std::swap(a,b);`.

Anwendung: Verkettete Listen (linked list)

Eine verkettete Liste ist ein Speicherlayout, bei welchem kein zusammenhängender Speicherbereich vorhanden ist.

Grundelement

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

Implementation

Eine sehr simple Implementation einer verketteten Liste ist:

```
class llvec {
  llnode* head; // Zeigt auf das erste Element
public:
  int& operator[](unsigned int i);
  void push_front(int e);
  llvec(unsigned int size); // Konstruktor
};
```

Der Zugriff ist bei einer verketteten Liste ineffizienter:

```
int& operator[](unsigned int i) {
  llnode* n = head;
  for(int j = 0; j < i; ++j)
    n = n->next;
  return n->value;
}
```

Ein grosser Vorteil ist das einfache Anfügen von Elementen:

```
void llvec::push_front(int e) {
  head = new llnode(e,head);
}
```

Der Konstruktor kann man so definieren:

```
llvec::llevc(unsigned int size) {
  head = nullptr;
  for(unsigned int i = 0; i < size; ++i)
    push_front(0);
}
```

Bäume

Bäume sind verallgemeinerte Listen, d.h. Knoten könen mehrere Nachfolger besitzen.

Container

Jeder Container hat charakteristische Eigenschaften, d.h. bestimmte Sachen macht er besonders effizient bzw. ineffizient. Es lohnt sich also einen geeigneten Container zu wählen.

Iteratoren

Iteration erfordert vom Prinzip her unabhängig vom Container die gleichen Schritte. In C++ gibt es deswegen zu jedem Container einen passenden Iterator. Implementierung vom Iterator ist für den Nutzer nicht relevant (Abstraktion).

Iterator für vector:	`std::vector<int>::iterator it;`
Iterator aufs erste Element:	`it = vec.begin();`
Iterator **hinters** letzte Element:	`it = vec.end();`
Zugriff auf aktuelles Element:	`*it;`
Iterator inkrementieren:	`++it;`

Anwendung: Sequentielle Iteration

```
using Iterator = std::vector<int>::iterator; // Alias
for (Iterator it = vec.begin(); it != vec.end(); ++it) {
  ... // Anweisungen
}
```

const Iterator

Neben einem Iterator sollte jeder Container auch einen Const-Iterator bereitstellen, welcher kein Schreibzugriff gewährleistet:

```
std::vector<int>::const_iterator itr = vec.begin();
```

Bereichsbasierte for-Schleife

Für die Iteration über einen ganzen Container gibt es auch folgende Möglichkeit:

```
for(int element: container_name)
  std::cout << element; // Ausgabe von allen Werten
```

Anwendung: Set (`#include<set>`)

Ein Set kann *kein* Element doppelt enthalten und besitzt keine bestimmte Reihenfolge (kein indexbasierter Zugriff). Es ist aber sehr effizient im überprüfen, ob ein Element enthalten ist und im einfügen bzw. löschen eines Elements.

Bibliothek:	`#include<set>`
Ungeordnetes Set:	`std::unordered_set<T> setname;`
Geordnetes Set:	`std::set<T> setname;`
Element hinzufügen:	`setname.insert(value);`
Intervall Initialisierung [b,e):	`std::set<int> setname(b,e);`

wobei `b = vec.begin()` und `e = vec.end()`

Element in ungeordnetem Feld suchen

Der Vergleich `set.find(e) == set.end()` gibt **wahr** zurück, wenn `e` im Set enthalten ist.

Eigener Iterator definieren

Der Iterator ist eine *innere Klasse*, d.h. sie wird innerhalb einer Klasse definiert (hier innerhalb der Klasse `llvec`). Um einen eigenen Iterator zu definieren benötigt man folgende Funktionalitäten:

```
class iterator {
  llnode* node; // Zeiger auf aktuelles Element
public:
  iterator(llnode* n);
  iterator& operator++();
  int& operator*() const;
  bool operator!=(const iterator& other) const;
};
```

Sowie folgende Memberfunktionen in der Klasse selbst:

```
class llvec {
public:
  class iterator {...};
  iterator begin(); // Gibt erstes Element
  iterator end(); // Zeigt hinter das letzte Element
};
```

Nützliche Funktionen für Container

Für alle Container kann man, solange es ihre charakteristischen Eigenschaften es erlauben, folgende Funktionen benutzen:

Bibliothek:	`#include<algorithm>`
Bereich [b,p) mit 5 füllen:	`std::fill(b,p,5);`
Den Bereich [b,e) sortieren:	`std::sort(b,e);`
Iterator auf Minimum im Bereich [b,p):	`std::min_element(b,p);`
Iterator auf den ersten Wert (2) in [b,p):	`std::find(b,p,2);`

Objekt orientierte Programmierung

Kapselung

Kapselung ist das Verbergen der Implementierungsdetails von Typen (privater Bereich) vor Benutzern. Anstelle vom direkten Zugriff definiert man eine Schnittstelle (öffentlicher Bereich) zum kontrollierten Zugriff auf Werte und Funktionalität.
Kapselung ermöglicht das Sicherstellen von Invarianten, sowie den Austausch von Implementierungen ohne Anpassungen von Benutzercode.

Subtyping (nicht prfgs relevant)

Typhierarchien mit Super- und Subtypen können Verwandtschaftbeziehungen sowie Abstraktionen und Spezialisierungen modellieren.

Ein Subtyp unterstützt mindestens die Funktionalität, die auch der Supertyp unterstützt - in der Regel aber mehr, d.h. Subtypen erweitern die Schnittstellen ihrer Supertypen (Spezialisierung).
Daher können Subtypen überall dort eingesetzt werden, wo Supertypen verlangt sind und Funktionen, die auf Supertypen operieren können, können auch auf Subtypen operieren.

```
struct Exp { ... }
struct BinExp : public Exp { .. }
```

Subtyprelationen sind transitiv, d.h. ein Subtyp von `BinExp` wäre auch ein Subtyp von `Exp`. Man sagt:
- `BinExp` ist eine von `Exp` *abgeleitete Klasse* (oder Subklasse, Kindklasse).
- `Exp` ist die *Basisklasse* (oder Superklasse, Elternklasse) von `BinExp`.

Vererbung

Abgeleitete Klassen erben die Funktionalität, d.h. die Implementierungen von Memberfunktionen, ihrer Elternklassen. Dies ermöglicht es, gemeinsam genutzten Code wiederverwenden zu können und vermeidet so Codeduplikation.

Geerbte Implementierungen können auch überschrieben werden, um zu erreichen, dass eine abgeleitete Klasse sich anders verhält als ihre Elternklasse.

Polymorphie und dynamische Bindung (nicht prfgs relevant)

Ein Zeiger vom *statischen Typ* $\texttt{T}_1$ kann zur Laufzeit auf Objekte vom *dynamischen Typ* $\texttt{T}_2$ zeigen, falls $\texttt{T}_2$ ein Subtyp von $\texttt{T}_1$ ist.
Wird eine virtuelle Memberfunktion von einem solchen Zeiger aus aufgerufen, so entscheidet der dynamische Typ darüber, welche Funktion ausgeführt wird (dynamische Bindung).

Zusammen mit Subtyping kann man so neue konkrete Typen zu einem bestehenden System hinzuzufügen, ohne dieses abändern zu müssen!
Bei Memberfunktionen ohne `virtual` wird immer die Memberfunktion des statischen Typ (hier `Exp`) ausgeführt.

```
struct Exp {
  virtual int size() const = 0;
}
```

Das `= 0` erzwingt eine Implementierung durch abgeleitete Klassen. Solange es in einer Klasse eine Memberfunktion mit `= 0` gibt, ist es eine *abstrakte Klasse*, welche nicht als Objekt instanziert werden kann.

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]