Disclaimer

Diese Zusammenfassung wurde zur Vorlesung „Informatik I“ von Dr. Malte Schwerhoff und Dr. Hermann Lehner (FS20) erstellt. Die Zusammenfassung soll und darf gerne modifiziert werden (Latex-Files liegen bei), und soll dann auch weiterhin anderen Studenten zur Verfügung stehen.

Für Korrektheit und Vollständigkeit ist keine Gewähr.

Josephine Loehle und Leo Landolt, 21. Mai 2021
Angepasst von Loa Marx, 21. Dezember 2023

1 Grundlagen

1.1 Typen

1.1.1 Overview

Typ	Was	Werte	Bits
double	präzisere Reelle Zahlen	$1.7 * 10^{\pm 308}$	64
float	Reelle Zahlen	$3.4 * 10^{\pm 38}$	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		8

Bei einem Konflikt wird in den allgemeineren ↑ Zahlentyp konvertiert.
Bsp.: `int a + float b = float c`

1.2 Fliesskommazahlen

1.2.1 Fliesskommazahlensysteme

Durch 4 natürliche Zahlen definiert: F(β, p, e_{min}, e_{max})

$\beta \geq 2$	Die Basis
$\mathbf{p} \geq 1$	Die Präzision (= Stellenanzahl, Mantisse)
$\mathbf{e}_{min}$	Der kleinste Exponent
$\mathbf{e}_{max}$	Der grösste Exponent

1.2.2 Normalisierte Darstellung

$$\pm d_0.d_1 \ldots d_{p-1} * \beta^e = F*(\beta, p, e_{min}, e_{max})$$

- Eine Ziffer (!= 0) vor dem Komma, der Rest dahinter
- Normalisierte Darstellung ist eindeutig
- Die Zahl 0 und alle Zahlen kleiner als $\beta^{e_{min}}$ haben keine normalisierte Darstellung
- Zum Runden eine Stelle nach erwünschter Präzision anschauen: falls 0 einfach ignorieren, falls 1 die Stelle davor +1 rechnen

1.2.3 IEEE Standard 754

float	1 Bit für das Vorzeichen 23 Bit für den Signifikanden (= Bits nach dem Komma) 8 Bit für den Exponenten (254 Exponenten, 2 Spezialwerte)
	insgesamt 32 Bit F*(2, 24, -126, 127)
double	1 Bit für das Vorzeichen 52 Bit für den Signifikanden (= Mantisse -Bits) 11 Bit für den Exponenten (2046 Exponenten, 2 Spezialwerte)
	insgesamt 64 Bit F*(2, 53, -1022, 1023)
±0	Mantisse-Bits sind 0 Exponenten-Bits sind 0
±∞	Mantisse-Bits sind 0 Exponenten-Bits sind alles 1
nan	Mantisse-Bits > 0 Exponenten-Bits sind alles 1

1.2.4 Float und Double

float	7 Stellen Exponent bis ±38
double	15 Stellen Exponent bis ±308

1.2.5 Fliesskomma-Richtlinien

- Teste keine Fliesskommazahlen auf Gleichheit
- Addiere keine Fliesskommazahlen sehr unterschiedlicher Grösse
- Subtrahiere keine Fliesskommazahlen ähnlicher Grösse
- Der Modulo-Operator % existiert nicht
- Division ist mit Nachkommastellen

1.3 Konstanten

- const type name
- Wert der Variable name darf nicht mehr verändert werden.

1.4 Werte

1.4.1 L-Wert

Links vom Zuweisungsoperator.
Speicher, hat eine Adresse.
Kann seinen Wert ändern.

1.4.2 R-Wert

Rechts vom Zuweisungsoperator.
Kann seinen Wert nicht ändern.
L-Wert kann als R-Wert verwendet werden aber nicht andersherum.

1.5 Operatoren

1.5.1 Overview

Operator	Zeichen	Eingabe	Ausgabe
Arithmetische Operatoren			
Multiplikation	*	R-Werte	R-Wert
Division	/	R-Werte	R-Wert
Modulo	%	R-Werte	R-Wert
Addition	+	R-Werte	R-Wert
Substraktion	-	R-Werte	R-Wert
Post-Inkrement/Dekrement	expr++	L-Wert	R-Wert
Prä-Inkrement/Dekrement	++expr	L-Wert	L-Wert
Relationale Operatoren			
Kleiner als	<	R-Werte	R-Wert
Grösser gleich	>=	R-Werte	R-Wert
Gleich	==	R-Werte	R-Wert
Ungleich	!=	R-Werte	R-Wert
Logische Operatoren			
And	&&	R-Werte	R-Wert
Or	||	R-Werte	R-Wert
Not	!	R-Wert	R-Wert
Zuweisung	=	R-&L-Wert	L-Wert
Eingabe	>>	L-Werte	L-Wert
Ausgabe	<<	L-& R-Werte	L-Wert

1.5.2 Präzedenzen

1. Klammern
2. Not
3. Arithmetische Operatoren
   - Punkt vor Strich
   - Unäre Operatoren vor binären
4. Relationale Operatoren
5. Binäre logische Operatoren
   - && vor ||

1.5.3 Zusätzliches

- (a/b)*b + (a%b) == a
- XOR(a,b): (a || b) && !(a &&b)
- De Morgan: !(a && b) == (!a || !b) bzw. !(a || b) == (!a && !b)
- false && (...) → false
- true || (...) → true

1.6 Zahlensysteme

Binär	Basis ist 2
Dezimal	Basis ist 10
Hexadezimal	Basis ist 16 mit [0-9] ≙ [0-9] und [10-15] ≙ [A-F]

1.6.1 Umrechnung

Dezimal zu Basis	Teile die Dezimalzahl wiederholt durch die Basis und notiere die Reste. Die Zahl in der neuen Basis besteht aus den Resten in umgekehrter Reihenfolge: 18/2 = 9 R0 ⇒ 9/2 = 4 R1 ⇒ 4/2 = 2 R0 ⇒ 2/2 = 1 R0 ⇒ 1/2 = 0 R1 ⇒ 10010
Basis zu Dezimal	Basis-Potenzen bilden und aufaddieren:10010 ⇒ 1*16 + 0*8 + 0*4 + 1*2 + 0*1 ⇒ 18
Binär zu Hexa.	Jede HD-Ziffer kann einzeln in eine 4-Stellige Binärzahl umgewandelt werden. In derselben Reihenfolge aneinandergliedern: 1A3 ⇒ 1103 ⇒ 000110100011 ⇒ 110100011

1.6.2 Binär

2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1024	512	256	128	64	32	16	8	4	2	1

1.6.3 Hexadezimal

16^3	16^2	16^1	16^0
4096	256	16	1

2 Schleifen

2.1 if, else if, else

```
if (condition) {
   statement;
} \\ statement is executed if condition is true

else if (condition){
        statement;
} \\ like if-loop but only looked at if condition before false

else {statement;
} \\ executed if other conditions are false
```

2.2 for

```
for (initialisation; condition; expression) {
    statement;
} \\ (i) initialisation statement is executed.
   \\(ii) while condition == true, statement is executed, expression is executed.
```

2.3 while

```
while (condition) {
      statement;
} \\ statement is executed while condition == true
```

2.4 do while

```
do {
  statement;}
while (condition); \\ statement is executed once, afterwards while-loop
```

2.5 switch

```
int grade;
switch(grade) {
  case 6: statement1; \\matching case is executed
  case 5: statement2;
  case ...;
   break; \\ all statements until break are executed (Durchfallen)
  case 3: statement3;
  case ...;
   break;
default: statement else; \\ if no case matches
```

2.6 Schleifen umwandeln

Anweisung A	Anweisung B
`while (condition) statement;`	`for ( ; condition; ) statement;`
`do statement;` `while (condition);`	`statement` `while (condition) statement;`
`for (int i = 0; i < n; i++){` `  statement;`	`{int i = 0;` `while(i < n) { statement; i++; }}`

2.7 Sprunganweisungen

break;	Schleife wird sofort beendet.
continue;	Rest des Statements wird übersprungen. Man gelangt direkt zur Expression.

3 Funktionen

3.1 Grundlagen

3.1.1 Wieso Funktionen?

Kapseln häufig gebrauchte Funktionalitäten und vermeiden somit Code-Duplizierung.

3.1.2 Funktionsdefinition

```
Type fname (type1 pname1, type2 pname2, ..., typeN pnameN){
    block;
    return Type;
```

3.1.3 Funktionsaufruf

`fname(expression1, expression2, ..., expressionN)`
`\\ expression1−N müssen in type1−N konvertierbar sein.`

3.2 Der Typ void

- Fundamentaler Type mit leerem Wertebereich
- Gibt keinen Wert zurück, sondern hat nur einen Effekt
- Kein return nötig, aber möglich

3.3 Vor- und Nachbedingungen

\\ PRE: Was muss bei Funktionsaufruf gelten?
Was ist der Definitionsbereich der Funktion?
So schwach wie möglich → möglichst grosser Definitionsbereich
\\ POST: Was gilt nach dem Funktionsaufruf?
Welchen Wert gibt die Funktion zurück?
So stark wie möglich → möglichst detaillierte Aussage

3.4 Gültigkeitsbereich einer Funktion

- Funktion darf nach Deklaration verwendet werden, auch wenn sie erst später wirklich definiert wird.
- Deklaration einer Funktion ist wie Definition nur ohne {...}.

3.5 Wiederverwenden von Funktionen

3.5.1 Funktion Auslagern und Inkludieren

- Funktionen in eigene Datei schreiben (functions.cpp)
- Datei ins Arbeitsverzeichnis der Hauptdatei stecken und durch `#include "functions.cpp"` Funktionen inkludieren.
- Nachteil: compiler muss die Funktionsdefinition für jedes Programm neu übersetzen → Kann bei vielen und grossen Funktionen sehr lange dauern

3.5.2 Getrennte Übersetzung

- Funktionen ohne Main-Funktion kompilieren: functions.cpp → functions.o
- Deklarationen aller benötigten Symbole in einer Header-Datei functions.h
- Header-Datei ins Arbeitsverzeichnis stecken und durch `#include "functions.h"` inkludieren
- Vorteil: Quellcode (.cpp) wird nach dem Erzeugen vom Objectcode (.o) nicht mehr gebraucht → Code ist nicht öffentlich

3.5.3 Namensräume

```
namespace std{
} \\Innerhalb der Klammern kann std:: weggelassen werden
```

4 Referenztypen

4.1 Definition, Initialisierung und Zuweisung

`Type& alias = expr \\alias ist ein neuer Name für expr`

4.2 Call by

4.2.1 Reference

- Funktionsargumente sind (teilweise) Referenztypen.
- Durch den Alias ist es möglich, Funktionsargumente dauerhaft, auch ausserhalb der Funktion zu ändern.

4.2.2 Value

- Argumente haben keine Referenztypen.
- Beim Funktionsaufruf werden alle Argumente kopiert, am Ende der Funktion werden alle Kopien wieder gelöscht.
- Es ist nicht möglich, Funktionsargumente dauerhaft zu ändern.

4.3 Const-Referenzen

`const Type& == (const Type)&`
Es wird ein Lese-Alias gebildet, durch den der Wert dahinter nicht verändert werden darf.
Funktionsargumente müssen so nicht kopiert werden müssen → effizient

5 Vektoren & Strings

5.1 Vektoren

Dienen zum Speichern gleichartiger Daten in einem zusammenhängenden Speicherlayout.
Benötigt `#include <vector>`

5.1.1 Vektorinitialisierung

std::vector<int> vec(n);	Die n Elemente von vec werden mit Nullen initialisiert.
std::vector<int> vec(n, x);	Die n Elemente von vec werden mit x initialisiert.
std::vector<int> vec{a, b, c, d};	Der Vektor wird mit einer Initialisierungslist initialiseirt.
std::vector<int> vec;	Ein leerer Vektor wird initialisiert.

5.1.2 Matrixinitialisierung

std::vector<std::vector<int>> mat;	Eine leere Matrix wird initialisiert.
std::vector<std::vector<int>> mat = { {a, b, ...}, {...}, ...};	Matrix wird mittels Initialisierungsliste initialisiert.
std::vector<std::vector<int>> mat(n, std::vector<int>(m))	Eine n Mal m Matrix wird initialisiert.

5.1.3 Auf Elemente zugreifen

Vektor
vec[n] gibt einem das n-te Element eines Vektors.
vec.at(n) prüft zusätzlich noch die Vektorgrenzen
Matrix
m[n_1][n_2] mit 1. Zeile, 2. Spalte
m.at(n_1).at(n_2) prüft zusätzlich noch Matrixgrenzen
Achtung: Der Index von Vektoren beginnt bei 0!

5.1.4 Vektorfunktionen

.size()	Gibt Länge des Vektors zurück
.push_back(x)	Fügt Element x hinten an.
.empty()	Prüft, ob der Vektor leer ist.
.clear()	Löscht den Inhalt.

5.2 Zeichen und Texte

5.2.1 Der Typ char

`char c = 'a'; \\ Repräsentiert druckbare Zeichen und Steuerzeichen.`

5.2.2 Strings

```
std::string s ="I like Pie" \\ Entspricht Vektor von char-Elementen.
                            \\ Benötigt #include <string>

std::string text(n, 'a') \\ Initialisiert String der Länge n voller a.
```

- Strings können verglichen werden (`text1 == text2`)
- Um auf einzelne Characters zuzugreifen benutzt man die gleiche Schreibweise wie bei Vektoren.
- Man kann Texte zusammensetzen (`text1 += text2`)
- Die Grösse eines Textes kann ausgegeben werden mit text.length()

5.2.3 Ströme

Konsole	benötigt `#include <iostream>`
	`std::cin >> in;`
	`std::cout << out;`
	Nur als Referenz für Funktionsargument

Dateien	benötigt `#include <fstream>`
	`std::ifstream in(filename);`
	`std::ofstream out(filename);`
	Eingabe prüfen: `while (in >> input)`
	Analog dazu sstream für Strings

Abstrakt	`std::istream/std::ostream`

5.2.4 Der ASCII-Code

Definiert konkrete Konversionsregeln char ↔ (unsigned) int

6 Rekursion

6.1 Basics

6.1.1 Base Case

- Kleinst mögliches Problem
- Abbruchbedingung, die sicher erreicht wird
- Sonst unendliche Rekursion → verbrennt Zeit und Speicher (Stack-Overflow)

6.1.2 Der Aufrufstapel

- Bei jedem Funktionsaufruf kommt ein „Auftrag“ auf den Aufrufstapel
- Sobald der Base-Case erreicht wird, wird der Stapel von oben nach unten abgearbeitet und die Aufträge gelöscht

7 Structs & Classes

7.1 Structs

Um sich einen eigenen Typ zu basteln. Default ist public.

7.1.1 Definition

```
struct T { \\T is der Name des neuen Typs
        type1 name1;
        type2 name2; \\type1−n sind die Typen der Membervariablen.
        ....
        typen namen; \\name1−n sind die Namen der Membervariablen.
}
```

7.1.2 Member-Zugriff

`expr.name`$_k$

- expr ist vom Struct-Typ T
- name ist der Name einer Member-Variable des Typs T
- . ist der Member-Zugriff-Operator

7.1.3 Initialisierung

T t;	Membervariablen von t vom Typ T werden default-initialisiert (Wert undefiniert)
T t = {x, y, ...};	Membervariablen von t werden mit den Werten der Liste initialisiert.

7.2 Classes

7.2.1 Überladen von Funktionen

Man kann mehrere Funktionen mit dem gleichen Namen haben, solange die Signatur anders ist.
Signatur = Namen, Typen, Anzahl und Reihenfolge der Argumente.

Operator Overloading: `operator`op
Erlaubt es, die normalen Operatoren (`+,-,*,/`) für Structs und Classes zu definieren.
Operatoren sind dann normal anwendbar (Bsp: `x + y`)

7.2.2 Datenkapselung

- Wir verstecken die Repräsentation und bieten Funktionalität.
- Struct: nichts wird versteckt (default: public)
- Class: alles wird versteckt (default: private)

7.2.3 Deklaration

```
class T {
public: \\ Memberfunktionen der Klasse, auf die man Zugriff haben soll.
private: \\ Membervariablen, auf die man kein Zugriff haben soll
};
```

Out-of-class-Memberdefinition benötigt ein `T::`

7.2.4 Konstruktoren

- Sind spezielle Memberfunktionen einer Klasse, die den Namen der Klasse tragen
- Können überladen werden
- Werden bei der Variablendeklaration aufgerufen
- Gibt es keine Variablen bei Deklaration wird der Default-Konstruktor aufgerufen
  → wird automatisch erzeugt wenn keiner definiert

```
T::T(type1 x, type2 y) : name1 (x), name2 (y) {
    \\ Zusätzliche Bedingungen
}
```

Aufruf des Konstruktors:
- `T t = T(x, y);`
- `T t(x, y);`

7.2.5 this und ->

- this stellt einen Zeiger auf die momentane Instanz dar
- -> dereferenziert einen Zeiger und führt den Punktoperator aus

8 Dynamische Datenstrukturen

8.1 Arrays

8.1.1 Der new-Ausdruck

`p = new Type[n]`
- Type ist der zugrundeliegende Typ vom Array
- n it die Grösse des zusammenhängenden Speicherplatzes
- Der Speicher bleibt reserviert bis man ihn explizit freigibt.
- p ist ein Zeiger auf die Startadresse des Speicherbereichs

`p = new Type(Konstruktorargumente)`
- Speicher für ein neues Objekt vom Typ T wird alloziiert
- p ist ein Zeiger auf die Adresse des Objekts

8.1.2 Zeiger-Typen

Type* p	Zeiger auf den Typ Type
p = new...	Adresse eines neuen Objektes.
p = &expr	Adresse eines bereits bestehenden Objektes des Typen Type.

8.1.3 Dereferenz-Operator

`expr = *p`
- p ist ein Zeiger
- Gibt den Wert hinter der Adresse p zurück
- Adress- und Dereferenzoperator sind inverse Funktionen

8.1.4 Null-Zeiger

- Zeigerwert, der angibt, dass auf kein Objekt gezeigt wird
- Repräsentiert durch nullptr

8.1.5 Zeigerarithmetik

Zeiger plus int	p + i zeigt auf das i-te Element des Arrays.
Zeigersubstraktion	Differenz beschreibt, wie weit die Elemente voneinander entfernt liegen.

8.1.6 Auf Zeiger zugreifen

Sequentielle Iteration:

```
for (char* it = p; it != p + el; ++it){ \\ el = Anzahl Elemente im
Array
     statement;
}
```

Wahlfreier Zugriff: `p[i] == *(p + i)`

8.1.7 Statische Arrays

`p = int a[n]`
- Wie Array nur dass die Grösse nicht verändert werden kann.
- Vektoren > Dynamische Arrays > Statische Arrays

8.1.8 Arrays in Funktionen

Konvention	Array wird durch 2 Zeiger beschrieben.
begin	Zeiger auf das erste Element
end	Zeiger hinter das letzte Element
Array leer	wenn begin == end

8.1.9 Const und Zeiger

int const p1	p1 ist ein const int
int const* p2	p2 ist ein Zeiger auf einen const int
int* const p3	p3 ist ein const Zeiger auf einen int
int const* const p4	p4 ist ein const Zeiger auf einen const int

8.1.10 Shared Pointers

Pointertyp, der sich die Anzahl an Shared Pointern, die auf unser Objekt zeigen merkt und es automatisch löscht, sobald kein Pointer mehr darauf zeigt.
- `std::shared_ptr<T>` -> Shared Pointer Klasse auf Objekt von Typ T
- `sharedPtr.use_count()` -> Anzahl an Pointern, die auf das Objekt zeigt.

8.1.11 Unique Pointers

Pointertyp, von dem immer nur genau ein Pointer auf dasselbe Objekt zeigen darf und das objekt automatisch löscht, sobald der Unique Pointer out-of-scope geht.
- `std::unique_ptr<T>` -> Unique Pointer Klasse auf Objekt von Typ T
- `oldUniquePtr.move(newUniquePtr)` -> Anzahl an Pointern, die auf das Objekt zeigt.

8.2 Verkettete Listen

Kein zusammenhängender Speicherbereich sondern jedes element zeigt auf seinen Nachfolger.

8.2.1 Realisierung

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

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

8.2.2 Vektor durch Linked List

Entspricht einem Zeiger auf das erste Element.

```
class llvec {
      llnode* head;
public:
      llvec(unsigned int size);
      unsigned int size() const;
};
```

8.2.3 Brauchbare Funktionen

print
```
void llvec::print(std::ostream& sink) const {
     for (llnode* n = this->head; n != nullptr; n = n->next) {
         sink << n->value << ' '; }}
```

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

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

8.3 Speicherverwaltung

8.3.1 Der Delete-Ausdruck

`delete p;`
- p ist ein Zeiger, der auf ein vorher mit new erzeugtes Objekt zeigt

`delete[] p;`
- p ist ein Zeiger, der auf ein vorher mit new erzeugtes Array zeigt.

8.3.2 Der Destruktor

- Deklaration: `~Type()`
- Wird automatisch aufgerufen bei Aufruf von delete oder wenn Gültigkeitsbereich endet.
- Wird automatisch erzeugt wenn keiner definiert wird

8.3.3 Die Dreierregel

Destruktor	Löscht Elemente
Copy-Konstruktor	Deklaration: `Type(const Type& x)` Für korrektes Kopieren und Initialisieren
Zuweisungsoperator	`operator=` Wie Copy-Konstruktor, aber früherer „Müll“ wird aufgeräumt

8.4 Bäume

Sind verallgemeinerte Listen: Knoten können mehrere Nachfolger haben.

```
struct tnode {
       char op; double val;
       tnode* left; tnode* right;

tnode(char o, double v, tnode* l, tnode* r) : op(o), val(v), left(l),
right(r) {};
```

8.4.1 Brauchbare Funktionen

```
\\ POST: returns the size (number of nodes) of the subtree with root n
int size (const tnode* n) {
    if (n){
      return size(n->left) + size(n->right) + 1;
    }
   return 0;
}

\\ POST: evaluates the subtree with root n
double eval(const tnode* n) {
       assert(n);
       if (n->op == '=') return n->val; \\ Blatt
       double l = 0;
       if (n->left)) l = eval(n->left); \\ linker und rechter Ast
       double r = eval(n->right);
       switch(n->op){
             case '+': return l+r;
             case '-': return l-r;
             case '*': return l*r;
             case '/': return l/r;
             default: return 0;
       }
}
```

copy
```
tnode* copy (const tnode* n) {
       if (n == nullptr)
         return nullptr;
       return new tnode (n->op, n->val,
       copy(n->left), copy(n->right));
}
```

clear
```
void clear(tnode* n) {
     if(n){
       clear(n->left);
       clear(n->right);
       delete n;
     }
}
```

9 Containers

Ansammlungs-Datenstrukturen.

9.1 Funktionen

contains(c, e)	true, wenn Container c Element e enthält
min/max(c)	Gibt das grösste/kleinste Element zurück
sort(c)	Sortiert die Elemente
replace(c, e1, e1)	Ersetzt alle e1 in c durch e2
sample(c, n)	Wählt zufällig n elemente aus c aus

9.2 Iteratoren

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

9.2.1 Const-Iteratoren

`cname::const_iterator it = c.cbegin();`
Gestatten Lesezugriff bei konstanten Containern.

9.3 Beispiele

std::unordered_set<Type>	Ungeordnete, duplikatfreie Zusammenfassung von Elementen.
std::set<Type>	Geordnete, duplikatfreie Zusammenfassung von Elementen.

10 Extras

10.1 Assertions

`assert(expr);`

- benötigt `#include <cassert>`
- Programm wird beendet falls expr nicht true
- abschalten durch `#define NDEBUG`

10.2 Typ-Aliasse

```
using Name = Typ \\Typ kann neu mit Name angesprochen werden
```

10.3 Die Standardbibliothek

benötigt `#include`

`<iostream>`	`std::cin>>`	
	`std::cout<<`	
`<cmath>`	`std::pow()`	Potenzieren
	`std::sqrt()`	Quadratwurzel
	`std::abs()`	Absolutbetrag
`<algorithm>`	`std::max`	Maximum zweier Argumente

10.4 Fehlerquellen

- Deklarationen sind nur im Block gültig.
- Funktionsargumente nicht vergessen.
- Wenn man eine Referenz erzeugt, muss das Objekt, auf das sie verweist, mindestens so lange leben wie die Referenz selbst.
- Der Zugriff auf Elemente ausserhalb der gültigen Grenzen eines Vektors führt zu undefiniertem Verhalten.
- Verhalten von Funktionen ungleich void ist undefiniert, wenn das Ende des Rumpfes ohne return-Anweisung erreicht wird.
- Vergleichsoperatoren existieren für structs und classes per default nicht.