1) Mathematik Allgemein

Allg.: $a,b \in R^+$
Potenzen: $n,m \in \mathbb{R}$
Wurzeln: $n,m \in \mathbb{N}$

$$e^{i \cdot x} = \cos(x) + i \cdot \sin(x)$$
$$e^{\ln(x)} = x \qquad \ln(e) = 1$$
$$e^{-\ln(x)} = \tfrac{1}{x}$$

$$a^n a^m = a^{n+m}$$
$$\frac{a^n}{a^m} = a^{n-m}$$
$$(a^n)^m = a^{nm}$$
$$a^n b^n = (ab)^n$$
$$a^{-n} = \frac{1}{a^n}$$

$$\sqrt[n]{a^m} = a^{\frac{m}{n}}$$
$$\sqrt[n]{\sqrt[m]{a}} = \sqrt[nm]{a} = \sqrt[m]{\sqrt[n]{a}}$$
$$\sqrt[n]{a}\sqrt[n]{b} = \sqrt[n]{ab}$$
$$\frac{\sqrt[n]{a}}{\sqrt[n]{b}} = \sqrt[n]{\frac{a}{b}}$$
$$a^0 = 1 \qquad a^1 = a$$

$$y = \log_a x \quad \Leftrightarrow \quad a^y = x$$
$$\log_b(u \cdot v) = \log_b(u) + \log_b(v)$$
$$\log_b(u / v) = \log_b(u) - \log_b(v)$$
$$\log_b(u^r) = r \cdot \log_b(u)$$
$$\log_a(x) = \frac{\ln(x)}{\ln(a)} = \frac{\log(x)}{\log(a)}$$

Folgen / Reihen

Rekursiv: $a_{k+2} = a_{k+1} + a_k$ (Fibonacci)

$$fib(0) = 0$$
$$fib(1) = 1$$
$$fib(n+1) = fib(n) + fib(n-1) \quad \text{für} \quad n \geq 1$$

Bsp.: $s_n = \sum_{k=1}^{n} k = \frac{n(n+1)}{2}$

Bsp.: $s_n = \sum_{k=0}^{n} q^k = \frac{1-q^{n+1}}{1-q}$, $\sum_{k=0}^{n} k \cdot q^k = \frac{q}{(1-q)^2}$

Fakultät

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n = \prod_{k=1}^{n} k$$

Anwendungs-Bsp.: $e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!}$

2) Datentypen, Variablen, Datenstrukturen

int

Integer werden abgerundet!

```
double number = 5.765;
cout << (int)number;          // Ausgabe: 5
```

string / char

cstring ist wesentlich schneller als das string-Objekt, dafür hat es keine dynamische Speicherverwaltung. cstring = char array.

Beispiele:

```
#include <string>
string str("bla");
char *cstring;
cstr = str.c_str();
```

Einige String-Funktionen:

- **str.c_str()** lässt sich ein string-Objekt in einen cstring konvertieren.
- **str.length()** gibt die Länge eines strings zurück.
- **str.insert(n,t)** fügt im string str den string t an Position n ein.
- **str.erase(p,n)** löscht n Zeichen ab Position p in string str.
- **str.find(t)** gibt die Position zurück, wo t sich in str befindet.
- **str.swap(t)** vertauscht string s mit string t.

Konvertierung

Konvertierung von int zu string:

```
#include <string>
#include <sstream>

int someint=1;
ostringstream buffer;
buffer << someint;
string somestr = buffer.str();
```

Konvertierung von string zu double/float:

```
#include <iostream.h>
#include <string>

string somestr = "4.4982";
double number = atof(somestr.c_str());
cout << number+1;
// output: 5.4982
```

char / chararray

In Grosbuchstaben konvertieren: **char ch = toupper(char ch);**
In Kleinbuchstaben konvertieren: **char ch = tolower(char ch);**

Den char auf Position 123 aus der ASCII-Tabelle ausgeben:

```
int number = 123;
cout << (char)number;
```

Terminierung: Der letzte Buchstabe eines chararrays sollte ein '\0' sein (Terminierung).

Die Länge eines C-Strings gibt die Funktion **strlen** in der Bibliothek <cstring> an.

Stream

```
istringstream instr("bla");                    (?)
ostringstream outstr;
instr >> f;          // float f(String infloat);
outstr << i;         // int i(int instring);
s = outstr.str();
```

Iterator

```
string::iterator i;
for(i=str.begin(); i != str.end(); i++) {
      count << *i;
}
```

Mit einem iterator kann ein strig durchlaufen werden.
Reverse iterator: **string::reverse_iterator i;** (Rückwärts durchlaufen, Verwendung von **rbegin()** und **rend()**.)
Normalen Iterator von end() bis begin() verwenden mit "i--" ist das gleiche wie ein Reverse-iterator!

Array

Dynamisches Array:
Wenn Anzahl Plätze anfangs unbekannt:
double *list = new double[length];

// Dynamisches Array fuer Zahl/Zeichen
// 1 Element groesser fuer Terminierung '\0'
char *current = new char[(i-temp)+1];

Arraysgrösse: Die Grösse/Länge eines Arrays herausfinden: **array.length()**

new-Operator

Mit dem "new"-Operator wird dynamisch Speicher alloziiert (wenn man noch nicht weiss wie viele Arrayplätze genutzt werden.

new type; // Neue Var anlegen (Gibt Adresse des neuen Onjekts zurück)
delete ip; // Zelle, auf welche ip zeigt, wird gelöscht.
delete[] a; // Array "a" löschen

Pointer

Syntax:

int* zahl;

int *ptr = &varname;

zeiger = &var;

***zeiger = 1234;**

- Pointer haben als Inhalt eine Adresse; zeigen also auf einen anderen Platz im Speicher.
- Richtig definieren: **int b2, *b1, b3;**
  Entspricht dem gleichen: int* b1, b2, b3; (nur b1 wird zum Ptr.)
- **int *ptr = &varname;** erzeugt Pointer auf varname zeigend.
- Adresse setzen: **zeiger = &var;**
  Zugriff auf Wert: ***zeiger = 12344;**
  (letzteres Speichert "1234" in var ab.

Referenzen

int n;
int &nr = n;

n und nr sind Synonyme!

Für die Funktion **f(int &z)** sind nur beschränkte Aufrufe erlaubt:

- **f(k);** Erlaubt, wenn k ein int ist, da "k=…" erlaubt ist.
- **f(0);** Nicht erlaubt, da 0 eine Konstante ist ("0=…" nicht erlaubt).
- **f(a[2]);** Erlaubt, wenn a ein Array von int ist, da "a[2]=…" erlaubt.
- **f(i+1);** Nicht erlaubt (+-Operator gibt keine Referenz zurück).
- **f(1*k);** Nicht erlaubt (*-Operator gibt keine Referenz zurück).

Struct

Syntax:

```
struct sname {
      type1 name1;
      type2 name2;
} objectnames;
```

Beispiel:

```
struct point {
      int x;
      int y;
      double intensity;
};
```

- Speichern von mehreren Datentypen die zusammen gehören
- "objectnames" kann angegeben werden, falls direkt neue Structs initialisiert werden sollen.
- Instanz des structs erstellen: **sname newsobj;**
- Zugriff auf die Werte eines Structs:
  newsobj.name1 = bla…;
  cout << newsobj.name1;
- Initialisierung auch wie Array möglich: **point p = {5,27,0.75};**
  In diesem fall Daten wie in Struct von oben nach unten eingeben (siehe Beispiel)

Pfeiloperator

Bei Struct-Pointe *heute (Pointer eines Structs) wird mit Pfeiloperator auf die Elemente zugrgtiffen: **heute->tag**
oder: **(*heute).tag**

3) Abfragen / Wiederholungen

if

Syntax:

if(a != b) { }
else { }

- Abfrage per **if(expression) { /* code */ }**
- Oder-Abfrage: **if(a==1 || b==1) { }**
- Und-Abfrage: **if(a==1 && b==1) { }**
- Ungleich Abfrage: **if(a != 1) { }** oder **if(!(a==1)) { }**
- XOR-Abfrage: **if((a && !b) || (!a && b)) { }**

while / for

whle(expression) { }
for(int i=0;i<10;i++) { }

- While schleife wird ausgeführt, bis expression **false** zurück gibt.
- Austritt aus einer Schleife mit **break;**
- Abbruch des momentanen Schleifendurchlaufs und Einstieg in den nächsten mit **continue;**

Switch

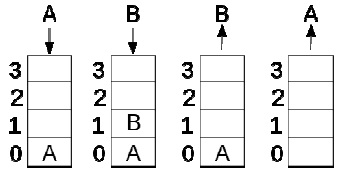
```
switch(zahl) {
    case 11:
    cout << "elf\n";
    break;
    case 12:
    cout << "zwoelf\n";
    break;
    default:
    cout << "Normalzahl\n";
    break;
}
```

- Abfrage einer Variable (im Bsp. int)
- Geht nicht mit string! Höchstens mit char.
- Wenn ein **break;** nach jedem case hinzugefügt wird, werden die folgenden Abfragen nicht mehr ausgeführt (Performancegewinn).
- Der Code unter dem Eintrag **default** wird ausgeführt, wenn keines der cases ausgeführt wurde.

Stack

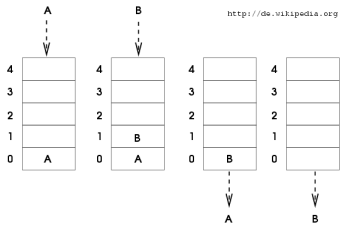
LIFO

Last In – First Out: Das LIFO-Prinzip ähnelt einem Papierstapel bzw eines Umzugkistenstapel. Elemente werden in genau der entgegengesetzten Reihenfolge abgerufen, in der sie zuvor abgelegt wurden, d.h. das erste ("unterste") Element wird als letztes abgerufen.



FIFO

First In – First Ou, häufig abgekürzt mit FIFO, gleichbedeutend mit First-Come First-Served bzw. FCFS, bezeichnet jegliche Verfahren der Speicherung, bei denen diejenigen Elemente, die zuerst gespeichert wurden, auch zuerst wieder aus dem Speicher entnommen werden. Eine solche Datenstruktur wird auch als Schlange bezeichnet.



FIFO Gesamtstruktur:

```
#include <iostream>
using namespace std;

struct stack{
    double value;
};

stack* stackarray;
int size=0;

void init() {
    // Stackarray initialisieren
    // …
}

int main (int argc, char * const argv[]) {
    init();

    cout << "////// Der Stack (" << size << ") wurde nitialisiert:\n";
    showStack();

    cout << "////// Oberstes Element entfernen\n";
    pop();
    showStack();

    cout << "////// Neues Element (324.213) zuoberst drauflegen\n";
    push(324.213);
    showStack();

    cout << "////// Groesse des Stacks: " << stacksize() << "\n";
    cout << "////// Stack loeschen.\n";
    clear();

    return 0;
}

void showStack() {
    for(int i=0;i<stacksize();i++) {
        cout << "Stack-Element " << i << ": ";
        cout << stackarray[i].value << "\n";
    }
}
```

FIFO pop und push Funktionen:

```
double pop() {
    double value=1;
    // Oberstes Element entfernen
    // Zuerst pruefen ob oberstes Element existiert
    if((size-1) >= 0) {
        value = stackarray[(size-1)].value;
        size--;
    }
    else {
        cout << "error\n";
    }
    return value;
}


void push(double value) {
    // Temporärarray anlegen
    stack temparr[size];
    for(int i=0;i<size;i++) {
        temparr[i].value = stackarray[i].value;
    }
    // Stacksize vergroessern:
    size++;
    stackarray = new stack[size];
    // Alte Elemente einkopieren:
    for(int i=0;i<(size-1);i++) {
        stackarray[i].value = temparr[i].value;
    }
    // Neues Element pushen
    stackarray[(size-1)].value = value;
}


int stacksize() {
    return size;
}


void clear() {
    delete stackarray;
}
```

4) OOP – Objektorientiertes Programmieren

Allgemein / Begriffe

Ein Objekt ist Variable mit Datentyp als Klasse. Objekt = Instanz.
Methoden sind Funktionen einer Klasse.
klassenname objektname; // Instanz / Objekt initialisieren

Zugriff

Auf Attribute (Elemente) der Klasse werden mit dem Punktoperator zugegriffen:
objektname.variable

Vererbung / protected

Abgeleitete Klassen übernehmen alles von ihrer Basisklasse.

protected-Variablen sind nur für Klassen und ihre Ableitungen verfügbar.

Klasse

Public

Syntax (Bsp.):

```
class tdatum {
        public:
        int tag;
        int monat, jahr;
}
```

```
class bla {
        void xy();
}
void bla::xy() {
}
```

Public:
Variablen sind öffentlich zugänglich.
Sonst äquivalent zu struct (dort ist standardmässig alles public).

Funktionen:
Wenn ausserhalb der Klasse definiert (nur Prototyp in der Klasse), zweimal Doppelpunkt für Zuweisung verwenden.

Funktionen können auch innerhalb der Klasse definiert werden. Dies ist vom Compiler her ein bisschen schneller.

this

Pointer auf momentanes Objekt:

```
void tRectangle::setValues(int x, int y) {
        this->x = x;
}
```

So kann der gleiche Variablenname verwendet werden, in der Klasse sowie Funktion.

Konstruktor

Destruktor

Syntax (Bsp.):

```
class someclass {
        public:
        someclass();
}
someclass:someclass() {
}
```

Konstruktor/Destruktor haben keinen Rückgabetyp, auch nicht void!

Konstruktor:
Funktion, welche bei Objektdefinition aufgerufen wird.
Eingabevariablen können übergeben und auch überladen werden..
Name des Konstruktors = "Klassenname".
Darin werden z.B. Variablen initialisiert.

Destruktor:
Der Destruktor hat keine Eingabevariablen.
Name des Destruktors = "~Klassenname" (~: Tilde-Zeichen)
Im Destruktor sollte mit "delete" Speicher freigegeben werden!

Kopier-Konstruktor

Klasseninstanzen werden Bitweise kopiert. Wenn Klassen also Pointer enthalten, zeigen diese ans gleiche Ziel. Ein Kopierkonstruktor hilft bei diesem Problem nach: Er erstellt z.B. neue Pointerziele.

friend

Funktionen in einer Klasse als friend definieren, welche auf private Daten der Klasse zugreifen dürfen (Funktionen mit erweitertem Zugriff).

Beispiel: **friend tRectangle scale(bla, …);**
Die Funktion tRectangle kann nun private Attribute / Methoden der Klasse verwenden.

Templates (STL: Standard Template Library)

Allgemein

Funktionen, bei denen Eingabetypen egal sind, komen für Templates in Frage.

Beispiel (Minimum berechnen):

```
template <class T>
T min(T a, T b) {
    return (a<b> ? <:b;
}
```

Beispiel (SWAP-Funktion):

```
template <typename T>
void swapT(T& a, T& b) {
    T tmp;
    tmp=a; a=b; b=tmp;
}
```

Es können ganze Klassen als Templates definiert werden.

vector

Dynamisches Array, Zugriff auf Daten via Nummer. **(?)**
Initialisieren: **vector<int> numbers(5);**
Bibliothek: **#include <vector>**
Mit **reserve()** kann Kapazität erweitert werden.
Mit **resize()** kann die Grösse (reell, besetzte Variablen) verändert werden
Modifikation der Elemente:

push_back()	Am Ende ein Element hinzufügen
pop_back()	Am Ende ein Element löschen
insert	
erase	
swap	

algorithm

Bietet Algorithmen wie z.B. **find**, … um z.B. Vektoren zu analysieren,
Bibliothek: **#include <algorithm>**

deque

Elemente können vorne und hinten hinzugefügt werden. **push_front()** / **pop_front()** und **push_back()** / **pop_back()**. Z.B. eine Warteschlange realisierbar (LIFO-Puffer).

list

Elemente sind irgendwo im Speicher abgelegt und mit Pointer verbunden.
Für list ist ebenfalls push/pop – front/back verwendbar.

5) Funktionen

Call by Value **void f(int x);**

"x" wird in der Funktion nicht veräündert, sondern nur verwendet. C++ legt eine temporäre Variable für "x" innerhalb der Funktion an.

Call by Reference (1) **void f(int &y);**

"y" kann von der Funktion verändert werden, da die Adresse von "y" übergeben wird. (Nicht für Arrays.)
Aufruf würde so aussehen (ohne "&"!): **f(number);**

Call by Reference (2) **void f(int *y);**

Arrays

Durch Aufruf **f(&x);** kann Pointer nach Dereferenzierung verändert werden. Einzige Möglichkeit Arrays mit Funktion zu verändern!
Funktionsdefinitionen für Array-Übergabe:

```
void f(int *a);
void f(int a[]);
void f(int a[10]);
```

Aufruf **f(a)** und **f(&a[0])** bewirken das Selbe!
Bei der Funktionsdefinitionen mit mehrdimensionalen Arrays muss die 2. Dimension definiert sein: Z.B.: **f(int a[10][5])** oder **f(int a[][5])**

Beispiele für (1) und (2), die äquivalent sind:

```
void changeVar(int &var) {
        var=5;
}
```

Aufruf: **changeVar(number);**

```
void changeVar(int *var) {
        (*var)=5;
}
```

Aufruf: **changeVar(&number);**

Funktion überladen	z.B.: **anzeigen(int i);** **anzeigen(double d);**	Funktionen können mit unterschiedlichen Aufrufparametern definiert werden.
Inline-Funktion	Funktionsdefinition: **inline int min(int a, int b);**	Um Programmausführung zu verschnellern: "inline"-Definition hinzufügen, somit wird die Funktion an jeden Ort im Programm des Aufrufs kopiert. → viel schneller
Rekursiver Funktionsaufruf	Beispiel Fakultät: **f(n) = n * f(n-1)**	Funktion ruft sich selbst auf. Kann u.U. auch indirekt geschehen.
Übergabevariablen "main"-Funktion	**int main(** **int argc, char * const argv[]** **)**	Variablen abgreifen, welche beim Programmaufruf auf der Kommandozeile übergeben werden. Erste Variable ist immer der Programmname selbst!

Default-Werte

Beispiel für Konstruktor: **classname::classname(int a=0, int b=0);**

So kann eine Funktion auch ohne Parameter aufgerufen werden.
(Dann werden die Default-Werte verwendet.)

Operatorfunktionen

Standardoperatoren können überladen werden.

Syntax: **rückgabetyp operator zeichen(parameter) { }**

Beispiel: **tRect tRecz::operator *(float s) {}**

Damit wird die Multiplukation eines floats mit einem tRect ermöglicht.
Der erste Parameter bezieht sich auf das Objekt. Am Beispiel ist also nur tRect * 2.3 und nicht 2.3 * tRect möglich!

Mathematik

#include <math.h>
sqrt(2) // Wurzel ziehen

Nützliche Funktionen / Operationen

File I/O

ofstream Klasse für File schreiben
ifstream Klasse für File lesen
} fstream Klasse für File lesen + schreiben

In File schreiben:

```
#include <fstream>

ofstream myfile;
myfile.open("file.txt", ios::out);
myfile << "Text für ins File";
myfile.close();
```

Modus beim öffnen:
ios::out zum schreiben
ios::in zum lesen
ios:app am Ende anhängen (append)
Kombination dieser Modi mit I-Operator. Wenn kein Modus ausgewählt wird, nimmt es den Standard.
z.B. bei ifstream: ios::in

File lesen:

```
#include <fstream>
#include <string>

myfile.open("file.txt", ios::in);
while(!myfile.eof()) {
        string zeile;
        getline(myfile,zeile);
        cout << zeile << endl;
}
```

Das Öffnen prüfen:

```
…
myfile.open("file.txt");
if(myfile.is_open()) {
        // File konnte geöffnet werden
}
else {
        cout << "Error beim öffnen!\n";
}
```

Datenfiles: Für Datenfiles verwendet man keine << und >> Operatoren, sondern **read()** und **write()**. Ausserdem den Modus **ios::binary** verwenden!

Zufallszahl

```
#include <cstdlib>
// Rand-Generator initialisieren
srand ( time(NULL) );

randzahl = rand() % 127 + 0;
```

Generieren einer Zufallszahl:
rand() % range + startnumber;

(value % 100) is in the range 0 to 99
(value % 100 + 1) is in the range 1 to 100
(value % 30 + 1985) is in the range 1985 to 2014

Verschlüsseln (bitweise XOR)

Um den Klartext zu verschlüsseln, wird der Reihe nach die xor Operation "^" auf die einzelnen Buchstaben des Klartext und mit den einzelnen Buchstaben des Passwort angewendet. Also wie folgt: **klartext[0]^passwort[0]**, **klartext[1]^passwort[1]**, ... Da das Passwort kürzer ist als der Text, muss beim Passwort wieder vorne begonnen werden (Tipp: modulo Rechnung). Zum Entschlüsseln wird genau die selbe Operation nochmals durch geführt. Also: **text[0]^passwort[0]**, **text[1]^passwort[1]**, ...

Sortieren

Das Sortieren durch Aufsteigen (englisch Bubble sort, "Blasensortierung") bezeichnet einen einfachen, stabilen Sortieralgorithmus, der eine Reihe zufällig angeordneter Elemente (etwa Zahlen) der Größe nach ordnet.

Swap (Vertauschen)

Eine Funktion zum Vertauschen von 2 Datentypen kann auf 2 Arten geschrieben werden:

```
void swap(int &a, int &b) {
        int tmp=b;
        b=a;
        a=tmp;
}
```

Aufruf: **swap(x,y);**

```
void swap2(int *a, int *b) {
        int tmp=*a;
        *a=*b;
        *b=tmp;
}
```

Aufruf: **swap2(&x,&y);**

6) Programme (Beispiele)

Standard input/output:

```
#include <iostream>

using namespace std;

double a1=0;

int main (int argc, char * const argv[]) {

        cout << "Bitte einen Double-Wert eingeben:\n";
        cout << "Wert eingeben: "; cin >> a1;
        cout << "Sie haben eingegeben: " << a1 << endl;

        return 0;
}
```

Alternative: **char input[256] = ""; cin.getline(input,256);**

Zusammengesetzte Datentypen und Zeiger:

```
#include <iostream>
using namespace std;

int n_global = 19;
float i_global = 0.23;

struct point {
        int x;
        int y;
        float color_rgb[3];
} ;
int main (int argc, char * const argv[]) {
        int n_lokal;
        int* r = &n_lokal;
        int** p = &r;
        point green = {23, 52, {0,0.99}};
        point red = {23, 52, {0.99}};
        point blue = {23, 52, {0,0,0.99}};
        point* tmp;
        tmp = &red; // a)
        *r = 42;
        r = &n_global;
        *r = 1+4;
        (*tmp).color_rgb[1] = green.color_rgb[1] - i_global; // b)
        (*tmp).color_rgb[2] = 1.0/n_global + blue.color_rgb[1]/2.5;
        green = *tmp;
        *&n_global = *&n_lokal +1;
        **p = 10 + *r; // c)
        cout << n_global << "\n";
        return 0;
}
```

Sortierung:

```
#include <iostream>
using namespace std;

double* zahlenArray = 0;
int n=0;

int main (int argc, char * const argv[]) {
    cout << "Wieviel Double-Werte sollen eingelesen werden?:";
    cin >> n;

    // Speicher für Array allozieren:
    zahlenArray = new double[n];

    // Werte einlesen:
    for (int i=0;i<n;i++) {
        cout << "Bitte Wert mit Index " << i << " eingeben: ";
        cin >> zahlenArray[i];
    }

    // Array zum ermitteln des groessten Index
    int index=0;
    int temp=0;
    for(int i=0;i<n;i++) {
        if(zahlenArray[i]>temp) {
            temp=zahlenArray[i];
            index=i;
        }
    }
    cout << "\nDer groesste Wert hat den Index: " << index << "\n";

    // Nach Groesse aufsteigend sortieren
    double tempvalue=0;
    int lastElement= 0;
    for(int i=0;i<n;i++) {
        lastElement = ((n-1)-i);
        for(int a=0;a<lastElement;a++) {
            // Wenn Element > Letztes, mit letztem vertauschen
            if(zahlenArray[a]>zahlenArray[lastElement]) {
                        tempvalue = zahlenArray[lastElement];
                        zahlenArray[lastElement]=zahlenArray[a];
                        zahlenArray[a]=tempvalue;
            }
        }
    }

    // Sortierte Liste
    cout << "\nDie Liste mit " << n << " Elementen ist nun sortiert:\n";
    for (int i=0;i<n;i++) {
        cout << zahlenArray[i] << "\n";
    }

    return 0;
}
```

Operatorüberladungen:

```
#include <iostream>
using namespace std;
class cPolynom {
private:
	double *coef;
	int deg;
public:
	int test;
	cPolynom();			// Leerer Konstruktor
	cPolynom(int n);		// Konstruktor, (n-ten Grades)
	cPolynom(const cPolynom  &uebergabeobj);
	cPolynom(double *koeffizienten, int n);	// Kopierkonstruktor
	~cPolynom();			// Destruktor
	void print() const;		// Polynom ausgeben
	int arraysize() const;	Groesse des Doublearrays (deg+1)
	double eval(int x_zero) const;
	void setCoef(double value, int position) const;
	void addPolynom(const cPolynom &obj);
	void substractPolynom(const cPolynom &obj);
	// Operatorueberladungen
	cPolynom & operator= (const cPolynom &obj);
	cPolynom operator+ (cPolynom &toadd);
	cPolynom operator- (cPolynom &tosubstract);
};
int main () {
	double v[] = {3,7,2};
	double w[] = {5,4,2,1};
	cPolynom a(v,3);
	cPolynom b(w,4);
	// Erstellt ein neues Polynom und initialisiert es mit der Addition
	// von a und b (dies braucht den Kopierkonstruktor).
	cPolynom c = a+b;
	// Zuweisungsoperator
	b = c + a;
	// Substraktionsoperators
	cPolynom d = a-b;
}
// KONSTRUKTOREN / DESTRUKTOREN
cPolynom::cPolynom() {
	deg=0;
	//test=0;
	coef = NULL;
}
cPolynom::cPolynom(int n) {
	coef = new double[n];
	deg = n-1;
	for(int i=0;i<n;i++) {
		coef[i]=0;
	}
}
cPolynom::cPolynom(double *koeffizienten, int n) {
	deg = n-1;
	coef = new double[n];
	for(int i=0;i<n;i++) {
		coef[i]=koeffizienten[i];
	}
}
// Kopierkonstruktor
cPolynom::cPolynom(const cPolynom &uebergabeobj) {
	deg = uebergabeobj.arraysize()-1;
	coef = new double[uebergabeobj.arraysize()];
	for(int i=0;i < uebergabeobj.arraysize();i++) {
		coef[i] = uebergabeobj.eval(i);
	}
}
// Destruktor
cPolynom::~cPolynom() {
	if(arraysize()>0) {
		delete[] coef;
	}
	else {
		// Das dynamische Array wurde nie erstellt...
		// cout << "(Nothing to delete.)\n";
	}
}
```

```
// OPERATORUEBERLADUNGEN
cPolynom & cPolynom::operator= (const cPolynom &obj) {
	// Only do assignment if "obj" is a different object from this.
    if (this != &obj) {
		deg = obj.arraysize()-1;
		coef = new double[obj.arraysize()];
		for(int i=0;i < obj.arraysize();i++) {
			coef[i] = obj.eval(i);
		}
	}
	return *this;
}
cPolynom cPolynom::operator+ ( cPolynom &toadd)  {
	// Aufruf: e = a.operator+(b);
	cPolynom result(*this); //copy von selber
	result.addPolynom(toadd);
	return result;
}
cPolynom cPolynom::operator- (cPolynom &tosubstract) {
	cPolynom result(*this); //copy von selber
	result.substractPolynom(tosubstract);
	return result;
}
// DIV FUNKTIONEN
// Polynomaddierung, wird von +Operator benötigt
void cPolynom::addPolynom(const cPolynom &obj) {
	cPolynom temppolynom(arraysize());
	for(int i=0;i <= deg; i++) {
		temppolynom.setCoef(eval(i),i);
	}	// Temporäres objekt vom alten --> Kopieren
	int degree = deg;		// Groesse fuer neues Objekt
	if(obj.arraysize()-1 > degree) { degree = obj.arraysize()-1; }
	coef = new double[(degree+1)];	// Objekt Dyn. neu anlegen
	deg = degree;
	// Neu angelegtes Objekt zuerst leer befuellen
	for(int i=0; i <= degree; i++) {
		// Werte von Temp-Obj und Obj zusammenkopieren
		coef[i]=0;
		if(temppolynom.arraysize()>i) {
			// Temppolynom zuaddieren
			coef[i]+=temppolynom.eval(i);
		}
		if(obj.arraysize()>i) {
			coef[i]+=obj.eval(i);	// Obj zuaddieren
		}
	}
}
// Polynomsubstrahierung, wird von -Operator benötigt
void cPolynom::substractPolynom(const cPolynom &obj) {
	// Von diesem Objekt ein anderes "obj" abziehen
	// Temporäres Objekt vom alten:
	cPolynom temppolynom(arraysize());
	for(int i=0;i <= deg; i++) {
		temppolynom.setCoef(eval(i),i);
	}
	// Groesse fuer neues Objekt
	int degree = deg;
	if(obj.arraysize()-1 > degree) { degree = obj.arraysize()-1; }
	// Objekt Dynamisch neu anlegen
	coef = new double[(degree+1)];
	deg = degree;
	// Neu angelegtes Objekt zuerst leer befuellen
	for(int i=0; i <= degree; i++) {
		// Werte von Obj dem Temp-Obj abziehen
		coef[i]=0;
		if(temppolynom.arraysize()>i) {
			// Temppolynom zuaddieren
			coef[i]+=temppolynom.eval(i);
		}
		if(obj.arraysize()>i) {
			// Obj abziehen
			coef[i]-=obj.eval(i);
		}
	}
}
```

7) Index