Eine kurze Einführung in die Programmierung mit C++

Dirk Zimmer

Eine Starthilfe für den Kurs Informatik I D-ITET
Teil 1: Grundlegende Sprachelemente

Vorwort

C++ ist eine komplexe und praxisorientierte Sprache mit vielen verschiedenen Aspekten. Diese Einführung beschränkt sich auf eine fundamentale Untermenge von Sprachelementen, die mit zahlreichen Beispielen unterlegt werden. Die wesentlichen Grundlagen des Programmierens lassen sich so besser und frei von unnötigem Ballast erlernen. Dieser Intention folgend, wird hier auf formale Erklärungswege verzichtet, dabei werden aber Ungenauigkeiten in Kauf genommen. Auf längere Sicht sollten Sie daher Ihre C++ Kenntnisse deutlich über diesen Text hinaus erweitern. Die Vorlesung und weiterführende Literatur können Ihnen dabei helfen.

Inhalt

1. Vom Quellcode zum Programm.

Programme werden typischerweise als reiner ASCII-Text in Texteditoren geschrieben. Die Programmiersprache (hier C++) hat hierbei eine Vermittlungsfunktion zwischen Mensch und Maschine. Die Dateien, die den Programmtext enthalten werden allgemein als Quellcode (engl.: Sourcecode) bezeichnet. Um diesen Quellcode in ein ausführbares Programm umzuwandeln benötigt man ein spezielles Übersetzungsprogramm. Dieses wird Compiler genannt. Die ausführbare Datei (engl.: Executable) repräsentiert Maschinencode und ist praktisch nur noch für den Computer lesbar.

2. Das Programm: Eine Abfolge von Anweisungen

Das folgende Programm fordert den Benutzer zur Eingabe zweier Zahlen auf und addiert diese anschliessend. Das Resultat wird auf der Konsole zurückgegeben. Der wesentliche Teil des Programms ist hier grau unterlegt. Dieser Teil besteht aus einer Abfolge von Anweisungen (engl.: Statement) die jeweils mit einem Semikolon beendet werden. Beim Ausführen eines Programms werden diese Statements der Reihe nach von oben nach unten abgearbeitet. Ein C++ Programm besteht also aus einer Sequenz von Anweisungen.

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

int main ()
{
  int a=0;                                          //Declare variables…
  int b=0;                                          //…and set initial values…
  int sum=0;                                        //…to zero.
  cout << "Please enter two numbers: ";             //prompt for input
  cin >> a;                                         //read in the 2 numbers
  cin >> b;
  sum = a+b;                                        //add the two numbers
  cout << "The sum is: " << sum << endl;
  return 0;                                         //terminate program
}
```

Normalerweise schreibt man eine Anweisung pro Linie. Dies ist jedoch nicht zwingend. Für den Compiler ist die Formatierung des Quellcodes irrelevant. Kommentare werden ebenfalls vom Compiler ignoriert. Diese werden durch einen vorangestellten Doppel-Slash // oder durch eine Klammerung mit /* */ gekennzeichnet. Nachfolgend wollen wir nun die einzelnen Komponenten, die ein Programm ausmachen, einzeln betrachten

3. Daten zwischenspeichern: Variablen

Eine wichtige Eigenschaft von Computern ist ihre Fähigkeit Daten zu speichern. Dem Programmierer dienen hierzu Variablen. In Variablen können Werte abgelegt, verändert und wieder herausgelesen werden.

Alle Daten werden in einem Computer als Sequenz von Bits, also von Nullen und Einsen, gespeichert. Damit das Programm weiss, wie es diese Bitsequenzen zu interpretieren hat, haben Variablen einen Typ.

Die wichtigsten Basistypen in C++ sind:

- **`int`**: für vorzeichenbehaftete Ganzzahlen. Bsp: 134
- **`double`**: für vorzeichenbehaftete Fliesskommazahlen. Bsp: -14.2e5
- **`char`**: für einzelne Zeichen (z. B.: Buchstaben). Bsp: 'a'
- **`bool`**: für Wahrheitswerte (mit den Werten **`true`** oder **`false`**).

Des Weiteren lassen sich mit den Typen `string` oder **`char`**`*` ganze Zeichenketten wie „Hallo“ speichern. Diese sind jedoch keine eigentlichen Basistypen.

Will man in C++ eine Variable deklarieren, so schreibt man zuerst ihren Typnamen und danach den Variablennamen. Für den Namen der Variablen gelten hierbei gewisse Regeln. Er darf nur aus Buchstaben und Ziffern sowie dem Unterstrich _ bestehen. Zudem ist es verboten mit einer Ziffer zu beginnen. Wichtig ist ausserdem, dass der Variablennamen eindeutig ist, also nicht gleich lautet wie ein reserviertes Wort der Sprache oder eine andere Variable, die vorher deklariert wurde. Das folgende Beispiel deklariert zuerst zwei Fliesskommazahlen und im Anschluss daran einen Character.

```
double laenge;
double breite;
char _zeichen;
```

Jede solche Deklaration repräsentiert ein Statement und wird mit einem Strichpunkt beendet. Deklarationen können an einer beliebigen Stelle im Code stehen. Erst nach der Deklaration ist es möglich die Variable zu verwenden. Als Folge einer Deklarationsanweisung wird Speicher für die Variable reserviert.

Weiterhin ist zu beachten, dass den Variablen kein standardisierter Initialwert zugewiesen wird. Wird bei der Deklaration kein Initialwert explizit angegeben, so ist der anfänglicher Wert als unbestimmt (zufällig) zu betrachten.

4. Grundlegende Operationen

Die meisten Operationen haben ein oder zwei Operanden eines gewissen Typs und liefern ein Resultat zurück mit einem bestimmten Rückgabetyp. Dieser ist dabei von der Operation abhängig.

Die wichtigste Operation ist die Zuweisungsoperation. Mit dieser lassen sich Werte von einer Konstanten oder einer Variablen in eine Variable übertragen. Für die Zuweisung dient der Operator = . Wichtig: Die Zuweisung ist keine Gleichung sondern eine Anweisung zum Kopieren von Werten.

Neben der Zuweisung gibt es die algebraischen Operatoren. Dies sind:

- + (Addition)
- − (Subtraktion)
- * (Multiplikation)
- / (Division)
- % (Modulo)

Wie in der Schule gelernt, bindet hierbei Punkt vor Strich. Ist eine andere Bindungsreihenfolge erwünscht so lassen sich die Ausdrücke klammern. Die beiden nachfolgenden Codesegmente

berechnen beide jeweils den Term b/(a+b). Das Resultat wird wiederum in a abgespeichert. Die erste Variante berechnet zuerst ein Zwischenresultat. Die zweite berechnet den Wert direkt.

```
double a = 12.1;
double b = 3.4;
double sum;
sum = a+b;
a = b/sum;
```

```
double a = 12.1;
double b = 3.4;
a = b/(a+b);
```

Neben den algebraischen Operatoren gibt es noch die Vergleichsoperatoren und die logischen Operatoren. Diese resultieren in einem Wert vom Typ `bool`. Also `true` (wahr) oder `false` (falsch).

Vergleichsoperatoren

- < (kleiner)
- <= (kleinergleich)
- == (gleich)
- != (ungleich)
- >= (grössergleich)
- > (grösser)

Logische Operatoren

- ! (Negation)
- && (logisches UND)
- || (logisches (nicht exklusives) ODER)

Das nachfolgende Beispiel wendet diese Operatoren an um zu überprüfen, ob das durch `laenge` und `breite` spezifizierte Rechteck mit 5% Toleranz einem Quadrat entspricht. Die Variable `res` nimmt je nach dem den Wert **`true`** oder **`false`** an.

```
double laenge = 42.7;
double breite = 43.1;
bool res;
res = laenge >= 0.95*breite && laenge <= 1.05*breite;
```

Wie Sie bemerken konnten, binden die einzelnen Operatoren mit unterschiedlichen Prioritäten. So binden die arithmetischen Operatoren stärker als die Vergleichsoperatoren, die wiederum stärker binden als die logischen Operatoren. Eine genaue Liste dieser Operator-Präzedenz können Sie unter [1] finden.

Typkonvertierung

Oft will man in einer Zuweisung einen Wert vom Typ A einer Variablen vom Typ B zuweisen. Hierzu muss eine Typkonvertierung vorgenommen werden. Dazu wird dem zu konvertierenden Ausdruck der Typname in runden Klammern vorangestellt.

```
double estimation=4.56;
int number;
number = (int)estimation;           //number is now 4;
```

Das obige Beispiel konvertiert eine Fliesskommazahl in eine Ganzzahl. Die Kommastellen werden dabei abgeschnitten. Es wird keine Rundung vorgenommen. Im nachfolgenden Beispiel wird der Nenner eines Bruchs in eine Fliesskommazahl umgewandelt. Ohne die Konvertierung würde die Division mit Ganzzahlen erfolgen und hätte das Ergebnis 0 anstatt 0.6.

```
int a=6;
int b=10;
double res;
res = a/(double)b;            //a is 0.6
```

Der C++ Compiler nimmt eine Vielzahl von Typkonvertierungen automatisch vor. Es ist jedoch kein schlechter Stil die Typkonvertierung dennoch explizit anzugeben. Ausserdem sind nicht alle Typkonvertierungen sinnvoll möglich.

Ein- und Ausgabe

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

Ohne weitere Erklärung sei hier angemerkt: Fügt man die oberen beiden Zeilen dem Beginn eines Programms hinzu, so stehen einem die Objekte `cin` und `cout` zu Verfügung. Mithilfe der Shift-Operatoren `<<` und `>>` lassen sich so Texte und Variablen auf der Konsole ausgeben. Ebenso können Text- oder Zahlenwerte in passende Variablen eingelesen werden. Das allererste Beispiel in diesem Text, demonstriert Ihnen die Anwendung von `cin` und `cout`.

5. Verzweigungen

Nicht immer soll ein Programm stur von oben nach unten ausgeführt werden. Abhängig vom aktuellen Zustand sollen gewisse Anweisungen übersprungen werden. Hierzu gibt es die **if**-Anweisung.

```
double zaehler = 5.0;
double nenner = 0.002;
double res;
if (nenner < 0.01)
    nenner = 0.01;
res = zaehler/nenner;
```

Das obige Beispiel setzt den Nenner eines Bruches auf 0.01. Diese Anweisung wird jedoch nur ausgeführt, wenn der Nenner vorher kleiner war. Von der **if**-Anweisung gibt es auch eine **if-else** Form. Diese hat zwei „Äste“, von denen jeweils nur einer ausgeführt wird. Damit liesse sich das obige Programm auch schreiben ohne den Wert von `nenner` zu verändern:

```
double zaehler = 5.0;
double nenner = 0.002;
double res;
if (nenner < 0.01)
    res = zaehler/0.01;
else
    res = zaehler/nenner;
```

In einem C++ Programm können Sie mit Hilfe der geschweiften Klammern { } eine Anweisungsfolge zu einer einzelnen Anweisung zusammenfassen. Ein solcher Block muss nicht per Semikolon abgeschlossen werden. Mit Hilfe dieser Blöcke können Sie das **if**-Statement auch für ganze Folgen von Statements verwenden:

```
cout << "Shall (a/5)*(a/5) be computed with integers? "
char c;
cin >> c;
cout << "Please enter the value for a: ";

if (c == 'y') {
    int a;
    int res;
    cin >> a;
    res = (a/5)*(a/5);
    cout << res;
} else {
    double a;
    double res;
    cin >> a;
    res = (a/5)*(a/5);
    cout << res;
}
```

Die Schreibweise mit Klammern ist in der Regel besser lesbar und weniger anfällig für Programmierfehler. Daher wird sie von nun an bevorzugt. Das obige Beispiel zeigt auch einen wichtigen Zusammenhang zwischen Anweisungsblöcken und Variablendeklarationen. Wie zu erkennen, werden die Variablen `a` und `res` scheinbar zweimal deklariert. Dies ist deshalb möglich, da Variablen nur innerhalb Ihres Anweisungsblocks sichtbar bzw. gültig sind. Obwohl also die letzten drei Zeilen der beiden Äste gleich sind, kann der Code nicht wie folgt vereinfacht werden:

```
char c;
cin >> c;
cout << "Please enter the value for a: ";

if (c == 'y') {
    int a;
    int res;
} else {
    double a;
    double res;
}
cin >> a;                    //Error: Unknown identifier: a
res = (a/5)*(a/5);           //Error: Unknown identifier: a, res
cout << res;                 //Error: Unknown identifier: res
```

Dieser Code ist ungültig, da die Variablen `a` und `res` ausserhalb ihrer Blöcke nicht mehr sichtbar sind. Mehr noch: Der Speicher für diese Variablen wird mit der Beendigung des Blocks wieder freigegeben. Die Variablen haben also nur eine begrenzte Lebensdauer (wie die meisten Variablen in einem Programm). Anmerkung: Der Sichtbarkeitsbereich einer Deklaration wird als Scope bezeichnet.

6. Schleifen

Das Überspringen bzw. bedingte Ausführen von Anweisungen ist nicht die einzige Manipulation des Programmablaufs, die von einem Programmierer erwünscht wird. Oft möchte man auch zurückspringen und bereits durchlaufene Programmteile wiederholen. Hierzu dienen Schleifen. Es gibt mehrere Arten von Schleifen. Die Wichtigste ist die **while**-Schleife.

Die **while**-Schleife wiederholt eine Anweisung, solange ihre Bedingung erfüllt ist. Die Bedingung wird am Anfang jedes Durchlaufs überprüft. Ist die Bedingung nie erfüllt, so wird die Anweisung folglich übersprungen. Beispielsweise lässt sich mit einer **while**-Schleife die Fakultät x = n! berechnen:

```
int n;
int x;
n=5;
x=1;
while (n>1) {
    x=x*n;
    n=n-1;
}
cout << x;
```

Zum besseren Verständnis betrachten wir die Werte für n und x während des Programmablaufs:

	x	**n**	**n>1**
Vor dem Schleifeneintritt	1	5	True
Nach dem 1. Durchlauf	5	4	True
Nach dem 2. Durchlauf	20	3	True
Nach dem 3. Durchlauf	60	2	True
Nach dem 4. Durchlauf	120	1	False
Nach Schleifenaustritt	120	1	False

Wie zu erkennen, wird die Schleife insgesamt viermal durchlaufen. Die Variable `n` wird dabei vor Beginn jedes Schleifendurchlaufs geprüft und ihr Wert in jedem Durchgang um eins verringert. Dieses Schema findet sich sehr häufig in Programmen. Die Variable `n` wird dann als Lauf- oder Iterationsvariable bezeichnet.

C++ bietet noch andere Schleifen an, wie zum Beispiel die **for-**Schleife. Die zusätzlichen Schleifentypen dienen vorwiegend der Bequemlichkeit, ermöglichen sie doch der Programmiersprache keine grundsätzlich neuen Möglichkeiten. Sie lassen sich durch eine **while**-Schleife ersetzen. Dies ist in der folgenden Tabelle für die **for**-Schleife erst anhand eines Beispiels, dann schematisch dargestellt. Das Beispiel berechnet die Summe der Zahlen 1 bis 10.

```
int sum=0;
int i;
for (i=1; i<=10; i=i+1)
{
  sum = sum+i;
}
```

```
int sum=0;
int i;
i=1;
while (i<=10) {
    sum = sum+i;
    i=i+1;
}
```

```
for (init-stmt; condition; incr_stmt)
{
    stmt;
}
```

```
init-stmt;
while (condition) {
    stmt;
    incr-stmt;
}
```

Auch wenn es keine technische Notwendigkeit gibt verschiedene Schleifentypen zu verwenden, so kann der Programmierer diese doch als sinnvolles Ausdrucksmittel einsetzen. Die Verwendung einer **for**-Schleife signalisiert typischerweise, dass die Anzahl Iterationen vor Schleifeneintritt bereits feststeht.

Schleifen und Bedingungen lassen sich beliebig ineinander verschachteln. Hiermit lassen sich mächtige Ablaufschemen erzeugen. Versuchen Sie das folgende kleine Programmsegment zu verstehen. Die dazugehörige Ausgabe auf der Konsole finden Sie rechts vom Quellcode.

```
int y;
int x;
for(y=1; y<=12; y=y+1) {
  for(x=1; x<=y; x=x+1) {
    if ((x+y)%2 != 0) {
      cout << '/';
    } else {
      cout << '\\';        //is character '\'
    }
  }
  cout << endl;
}
```

```
\
/\
\/\
/\/\
\/\/\
/\/\/\
\/\/\/\
/\/\/\/\
\/\/\/\/\
/\/\/\/\/\
\/\/\/\/\/\
/\/\/\/\/\/\
```

7. Funktionen

Bis jetzt haben wir nur Programme betrachtet, die aus einem einzigen Anweisungsblock bestehen. Für grössere Programmierprojekte ist dies jedoch nicht vorteilhaft. Betrachten wir hierzu die Berechnung des Binomialkoeffizienten mit Hilfe folgender Formel:

$$binom(n,k) = n! / (k! \cdot (n-k)!)$$

Das sture Ausrechnen dieser Formel erfordert das dreimalige Ausrechnen der Fakultät. Der nachfolgende Code erledigt diese Aufgabe:

```
int n = 10;
int k = 4;

int i;
int n_fak = 1;
for(i=2; i<=n; i=i+1) { n_fak = n_fak*i;}

int k_fak = 1;
for(i=2; i<=k; i=i+1) { k_fak = k_fak*i;}

int nk_fak = 1;
for(i=2; i<=n-k; i=i+1) { nk_fak = nk_fak*i;}

int res = n_fak / (k_fak*nk_fak);
```

Ganz offensichtlich wäre es von Vorteil, wenn man die Berechnung der Fakultät auslagern könnte. Hierzu dienen Funktionen. Das folgende Beispiel zeigt die Funktionsdefinition für die Berechnung der Fakultät.

```
int fak(int n)
{
    int res=1;
    int i;
    for(i=2; i<=n; i=i+1) res = res*i;
    return res;
}
```

Mithilfe dieser Funktion lässt sich nun der Binomialkoeffizient bequem berechnen. Hier ebenfalls als Funktion codiert:

```
int binom(int n, int k)
{
    return fak(n) / (fak(k)*fak(n-k));
}
```

Schauen wir uns nun den gesamten Mechanismus genauer an. Eine Funktion wird definiert durch folgende Sequenz der Elemente:

1. den Typ des Rückgabewertes,
2. den Namen der Funktion,
3. einer Liste mit Funktionsparametern mit deren Typ,
4. einem Anweisungsblock in geschweiften Klammern: dem Funktionskörper.

Innerhalb des Funktionskörpers können beliebige Anweisungen erfolgen und die Funktionsparameter können wie normale Variablen behandelt werden. Die Funktion wird beendet durch die **return-**Anweisung. Diese Anweisung legt auch gleichzeitig den Rückgabewert fest.

Die Verwendung einer Funktion bezeichnet man als Aufruf (engl.: Call). Um eine Funktion aufzurufen, schreibt man ihren Namen und in Klammern dahinter die passenden Parameterwerte. Der Wert dieses Ausdrucks entspricht dann dem Rückgabewert der Funktion. Beim Aufruf werden Werte übergeben, nicht die Variablen selbst. Eine Änderung der Parameter innerhalb der Funktion zieht damit keine Änderungen ausserhalb der Funktion nach sich.

Betrachten wir nun einige spezielle Funktionen. Im allerersten Programm wurde ohne weitere Erklärungen die Funktion `main` vorgestellt.

```
int main ()
{
…
}
```

Die Funktion `main` ist Bestandteil eines jeden eigenständigen C++ Programms und wird zum Programmstart aufgerufen. Der Befehl **`return`** innerhalb der `main`-Funktion beendet folglich das Programm. Wie wir sehen, hat diese Version von `main` keine Parameter. Es ist dennoch wichtig die leeren Klammern () zu schreiben, sowohl beim Aufruf als auch bei der Definition, damit der Bezeichner als Funktion zu erkennen ist. Soll eine Funktion keinen Wert zurückgeben, so verwendet man als Rückgabetyp den Bezeichner **`void`**. Hier kann dann auf ein **`return`**-Statement verzichtet werden.

Rekursion

Funktionen dienen nicht nur zur Strukturierung und zur Wiederverwendung von Code. Sie sind auch ein mächtiges Instrument zur Ablaufsteuerung. So kann sich eine Funktion auch selbst aufrufen. Einen solchen Aufruf bezeichnet man als rekursiv. Rekursion ist ein mächtiges Werkzeug in der Programmierung.

Folgendes Beispiel zeigt erneut eine mögliche Implementation der Fakultät. Hierzu dient uns die Rekursionsformel:

$$fak(n) = n \cdot fak(n\text{-}1)$$

Die Rekursion ist verankert für n gleich 0 bzw. 1:

$$fak(0) = fak(1) = 1$$

Diese Rekursionsformel lässt sich direkt in eine C++ Funktion übertragen:

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

Der Aufruf von `fak(3)` führt nun zum folgenden Programmablauf:

- `fak(3)` ruft `fak(2)` auf.
- `fak(2)` ruft `fak(1)` auf.
- `fak(1)` gibt 1 zurück
- `fak(2)` berechnet 2*1 und gibt 2 zurück
- `fak(3)` berechnet 3*2 und gibt 6 zurück.

8. Structs

Oft will man nicht nur den Code strukturieren, sondern auch die dazugehörigen Daten (Variablen). Structs bieten dem Programmierer die Möglichkeit Daten zusammenzufassen. Dies ist sinnvoll, wenn die Daten eine funktionale Einheit bilden.

```
struct Rechteck{
  double x;
  double y;
  double hoehe;
  double breite;
  int farbcode;
};
```

Das obige Beispiel deklariert einen Struct-Typ für Rechtecke. Achtung: Hier ist ein abschliessendes Semikolon notwendig. Die Koordinaten des Mittelpunkts werden durch `x` und `y` angegeben, die Ausdehnung durch `hoehe` und `breite`. Die folgende Funktion wandelt das Rechteck in ein möglichst ähnliches Quadrat um und zeigt damit auf, wie man mit Structs arbeiten kann.

```
Rechteck Symmetrize(Rechteck r)
{
  double groesse;
  groesse = (r.hoehe+r.breite)/2;
  Rechteck q;
  q = r;
  q.hoehe = groesse;
  q.breite = groesse;
  return q;
}
```

Instanzen von Structs werden erzeugt, indem man eine Variable vom entsprechenden Typ deklariert. Auf die einzelnen Elemente einer Struct kann dann per Punkt-Operator zugegriffen werden. Wie Sie sehen, können Structs auch als Ganzes überwiesen werden und ebenso als Parameter und Rückgabewert von Funktionen dienen.

9. Arrays

Das Deklarieren einzelner Variablen ist nicht immer ausreichend. Oft möchte man grosse Mengen, also ganze Felder von ihnen deklarieren. Hierzu dienen Arrays. Beispielsweise möchten wir von jedem Tag im Jahr die Durchschnittstemperatur speichern. Hierzu kann man folgendes Array deklarieren. Die nachfolgende Schleife initialisiert alle Werte mit Null.

```
double temperatur[365];
int i;
for (i=0; i<365; i=i+1) { temperatur[i] = 0.0; }
```

Bei der Deklaration eines Arrays wird die Anzahl Elemente durch eine konstante Zahl in eckigen Klammern angegeben. Der Zugriff auf ein einzelnes Element erfolgt dann ebenfalls durch eckige Klammern. In diesem Fall entspricht die Zahl einer Positionsangabe: Dem Index. Das erste Element des Feldes hat Index 0, das letzte Element hat den Index n-1. Hier also 364.

Für den Zugriff können selbstverständlich auch Variablen verwendet werden (Im obigen Beispiel: i). Stellen Sie dabei jedoch sicher, dass Sie nur auf gültige Indexwerte zugreifen. Der Zugriff auf ungültige Elemente kann zu Speicherzugriffsfehlern oder anderen Effekten führen. Structs und Arrays lassen sich untereinander zu komplexen Datenstrukturen kombinieren. Das nachfolgende Beispiel speichert die Wetterdaten (Temperatur, Windstärke) von 52 Wochen mit 7 Tagen. Die zwei geschachtelten for-Schleifen übernehmen die Initialisierung mit 0;

```
struct Wetter {
    double temperatur;
    int windstaerke;
};

Wetter data[52][7];
int w;
int d;
for (w=0; w<52; w=w+1) {
    for (d=0; d<7; d=d+1) {
        data[w][d].temperatur = 0.0;
        data[w][d].windstaerke = 0;
    }
}
```

Hier wurde ein zweidimensionales Array (ein Array von einem Array) erstellt. Der Typ eines Feldelements ist die Struct `Wetter`. Der Zugriff auf ein zweidimensionales Array erfolgt entsprechend mit zwei eckigen Klammern. Bitte beachten Sie: Arrays lassen sich nicht als Ganzes überweisen. Zum Kopieren von Arrays benötigen Sie eine Schleife:

```
int a[100];
int b[100];

a=b;                                          //error

for (int i=0; i<99; i=i+1) { a[i] = b[i]; }   //correct
```

Da sich Arrays nicht als Ganzes kopieren lassen, ist es heikel diese als Parameter in Funktionen zu verwenden. Hierbei wird das Array nicht kopiert sondern eine Referenz übergeben (eigentlich ein Pointer). Durch leere eckige Klammern können Arrays beliebiger Grösse übergeben werden.

Folgende Funktion dreht die Reihenfolge der Elemente in einem Array um. Diese Funktion gibt keinen Rückgabewert (daher **void**), sondern manipuliert den Inhalt im Array direkt. Anders als bei einfachen Parametern, wird hier das Originalarray abgeändert und nicht eine Kopie.

```
void reverse(int data[], int size)
{
  int h;
  int i = 0;
  size = size-1;
  while (i<size) {
      h = data[i];
      data[i] = data[size];
      data[size] = h;
      i = i+1;
      size = size-1;
  }
}
```

Auch von der main-Funktion gibt es eine Variante in der ihr ein Array übergeben wird:

```
int main(int argc, char* argv[])
{
   …
}
```

Wird ein Programm von der Kommandozeile aufgerufen, so lassen sich hierbei Kommandozeilenparameter angeben. Um diese zu verarbeiten wird die obige Variante der `main`-Funktion verwendet. Der Parameter `argv` ist hier ein Array von Zeichenketten die den Aufrufparametern entsprechen. Am Index 0 findet sich der Programmname selbst. `argc` enthält die Anzahl Argumente. Beispiel: Wird ein Programm mit der Kommandozeile

```
> ./Polynom 1.6 4 -5 3
```

aufgerufen so stellt sich der Inhalt von `argc` und `argv` wie folgt dar:

```
argc:      5

argv[0]:   "Polynom"
argv[1]:   "1.6"
argv[2]:   "4"
argv[3]:   "-5"
argv[4]:   "3"
```

10. Ein Programm in seinen Entwicklungsstufen

Abschliessend soll ein etwas grösseres Beispiel Schritt für Schritt diskutiert werden. Wir wollen ein Programm schreiben, mit dem Namen Polynom. Es soll einen Wert für x und eine (fast) beliebige Zahl von Koeffizienten als Aufrufparameter entgegennehmen. Dann soll das Polynom für x ausgewertet werden und das Resultat auf der Konsole erscheinen. Auf der Konsole wird das Programm also wie folgt aussehen:

```
> ./Polynom 2.1  2 -5 1.5 -3.5
  2*2.1^3 + -5*2.1^2 + 1.5*2.1 + -3.5 = -3.878
>
```

Als Programmieranfänger sollte man es meiden, ein solches Programm in einem Rutsch zu entwickeln und daher mehrere Zwischenschritte einlegen. In jedem der Schritte sollte das Programm kompilierbar und lauffähig sein.

Schritt 1: Das leere Gerüst

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

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

}
```

Da wir die Eingabeparameter einlesen wollen, brauchen wir die zweite Version der main-Funktion. Für die Ausgabe können wir schon einmal `iostream` einbinden.

Schritt 2: Deklaration der Variablen

Überlegen wir uns, welche Daten wir speichern müssen. Dies sind der Wert für x, die Koeffizienten sowie die Anzahl der Koeffizienten. Die Variable x ist ein Fliesskommatyp. Für die Anzahl Koeffizienten verwenden wir einen Integer. Die Koeffizienten selber gehören in ein Array vom Typ `double`. Da wir vorab nicht wissen, wie gross das Array sein muss, deklarieren wir es gross genug mit 100 Elementen.

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

int main(int argc, char* argv[])
{
  double x;
  int numCoeffs;
  double coeffs[100];

}
```

Schritt 3: Einlesen der Daten

Als nächstes sollen x und die Koeffizienten eingelesen werden. Hierzu müssen wir über das Array `argv` iterieren und die Zeichenketten **`char`**`*` in **`double`** verwandeln. Für letztgenannte Aufgabe durchsuchen wir die C/C++ Standardbibliotheken [1]. Dort ist die Funktion `atof` definiert: **`double`** `atof( const char *str );` Diese Funktion wandelt eine Zeichenkette in eine Fliesskommazahl um. Das ist genau, was wir brauchen.

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

int main(int argc, char* argv[])
{
  double x;
  int numCoeffs;
  double coeffs[100];

  x = atof(argv[1]);
  numCoeffs = argc-2;
  int i;
  for(i=0; i< numCoeffs; i=i+1) {
    coeffs[i]=atof(argv[i+2]);
  }
}
```

Schritt 4: Ausgeben der Daten

Um zu überprüfen ob wir die Daten korrekt eingegeben haben sollten wir diese auch wieder ausgeben. Dazu wird das Programm um eine Ausgabesektion erweitert. Es wird dabei berücksichtigt, dass die Ausgabe für die Potenzen 0 und 1 in gekürzter Form erfolgt:
Also `a*x + b` und nicht `a*x^1 + b*x^0`.

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

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

  double x;
  int numCoeffs;
  double coeffs[100];
  x = atof(argv[1]);
  numCoeffs = argc-2;
  int i;
  for(i=0; i< numCoeffs; i=i+1) {
      coeffs[i]=atof(argv[i+2]);
  }

  int degree;
  for(i=0; i< numCoeffs; i=i+1) {
      degree = numCoeffs -i-1;
      if (i>0) cout << " + ";
      cout << coeffs[i];
      if (degree >= 1) {
          cout << "*" << x;
          if (degree > 1) { cout << "^" << degree; }
      }
  }
  cout << " = " << "??" << endl;
}
```

Schritt 5: Evaluieren des Polynoms

Die Berechnung des Polynoms wird nun zwischen die Eingabe und die Ausgabe eingeschoben. Um die n-te Potenz von x zu berechnen brauchen wir die Funktion `pow(…)` aus der C Standardbibliothek. Diese ist ebenfalls unter [1] dokumentiert. Um die Funktion nützen zu können muss der zugehörige Header `math.h` per include-Befehl eingebunden werden. Die for-Schleife übernimmt das Aufsummieren der unterschiedlichen Potenzen mit ihren Koeffizienten.

```
#include <iostream>
#include <math.h>
using namespace std;

int main(int argc, char* argv[])
{
  double x;
  int n;
  double coeffs[100];

  x = atof(argv[1]);
  numCoeffs = argc-2;
  int i;
  for(i=0; i< numCoeffs; i=i+1) {
      coeffs[i]=atof(argv[i+2]);
  }

  int degree;
  double result=0;
  for(i=0; i< numCoeffs; i=i+1) {
      degree = numCoeffs -i-1;
      result = result + coeffs[i]*pow(x,degree);
  }

  for(i=0; i< numCoeffs; i=i+1) {
      degree = numCoeffs -i-1;
      if (i>0) cout << " + ";
      cout << coeffs[i];
      if (degree >= 1) {
          cout << "*" << x;
          if (degree > 1) { cout << "^" << degree;}
      }
  }
  cout << " = " << result << endl;
}
```

Schritt 6: Strukturieren des Programms

Das obige Programm ist schon komplett funktionstüchtig, allerdings nicht sehr strukturiert. Der komplette Code ist in einem einzigen Block untergebracht. Möchte man nachträglich Änderungen am Programm vornehmen oder Teile des Codes wiederverwenden, so wird sich dies als Nachteil herausstellen. Um das Programm strukturierter zu gestalten werden die einzelnen Teilaufgaben in Funktionen übertragen. Hinzugefügte Kommentare erleichtern zudem das Verständnis des Codes.

```
#include <iostream>
#include <math.h>
using namespace std;
```

```
/* Prints the polynomial with coefficients c on cout
   The parameter n defines the number of coefficients
*/
void printPolynom(double x, double c[], int n)
{
  int degree;          // current degree to process
  for(int i=0; i<n; i=i+1) {
        degree = n-i-1;
        if (i>0) cout << " + ";
        cout << c[i];
        if (degree >= 1) {
          cout << "*" << x;
          if (degree > 1) { cout << "^" << degree; }
        }
  }
}
/* Evaluates the polynomial and returns the result.
   The coefficients are specified in the parameter c
   The parameter n defines the number of coefficients
*/
double evalPolynom(double x, double c[], int n)
{
  int degree;          // current degree to process
  double result=0;     // return value
  for(int i=0; i<n; i=i+1) {
      degree = n-i-1;
      result = result + c[i]*pow(x,degree);
  }
  return result;
}

/*  Main routine:
  - processes the program arguments
  - prints the polynomial
  - evaluates the polynomial and prints the result
*/
int main(int argc, char* argv[])
{
  double x;            //Value of x in the polynomial
  int numCoeffs;       //number of Coefficients
  double coeffs[100];  //Coefficients of the polynomial

  x = atof(argv[1]);
  numCoeffs = argc-2;
  int i;
  for(i=0; i< numCoeffs; i=i+1) {
      coeffs[i]=atof(argv[i+2]);
  }

  printPolynom(x, coeffs, numCoeffs);

  cout << " = " << evalPolynom(x, coeffs, numCoeffs) << endl;
  return 0;
}
```

Schritt 7: Verbessern der Evaluierung
Die Evaluierung eines Polynoms kann cleverer gestaltet werden, wenn man bei den niedrigen Koeffizienten beginnt. Nur kann die entsprechende Potenz von x dadurch errechnet werden, dass man, beginnend bei 1, in jedem Schritt mit x multipliziert. Dadurch wird das aufwendige Ausrechnen der Potenzfunktion durch eine einfache Multiplikation ersetzt. Zudem ist `pow(…)` recht ungeeignet, da es nur für eine positive Basis definiert ist.

```
double evalPolynom(double x, double c[], int n)
{
  double result = 0;
  double p = 1;
  for(int i=n-1; i>=0; i=i-1) {
      result = result + c[i]*p;
      p = p*x;
  }
  return result;
}
```

Umgang mit Fehlermeldungen
Beim Entwickeln eines Programms unterlaufen dem Programmierer manchmal auch Fehler. Viele davon (aber längst nicht alle) werden vom Compiler abgefangen, der sich dann mit einer Liste von Fehlern meldet. Ein einzelner, kleiner Fehler wie das Vergessen eines Semikolons kann bei einem C++ Compiler zu einem Rattenschwanz an Fehlermeldungen führen. Meistens reicht es jedoch aus, sich auf den ersten Fehler der Liste zu stürzen. Für fast alle Typen von Fehlern gilt jedoch: Programmiert man in kleinen Schritten und testet regelmässig, so sind die Fehler leichter zu lokalisieren.

11. Guter Programmierstil, Kommentare

Die Formatierung des Programmcodes ist für den Compiler unerheblich nicht aber für den menschlichen Leser. Generell sollten Anweisung in Blöcken eingerückt werden. Wir empfehlen 2-4 Leerzeichen. Gut gewählte Variablennamen erleichtern das Verständnis. Allzu lange Namen erschweren jedoch den Lesevorgang. Es gilt abzuwägen und prägnante Ausdrücke zu finden.

Generell sollte ein Variablenname aussagekräftig sein, wenn sich sein Scope über einen grossen Bereich erstreckt. Auch Typ- und Funktionsnamen sollten immer aussagekräftig formuliert sein. Für Hilfsvariablen, die nur kurz eingesetzt werden, sind kurze Namen von 1-3 Zeichen in Ordnung. Es ist empfehlenswert Typ- und Variablenbezeichner durch eine einheitliche Schreibweise zu trennen. Beispielsweise sollten Namen von Typen stets mit einem grossen Buchstaben beginnen, die von Variablen jedoch mit einem kleinem.

Das Schreiben von Kommentaren ermöglicht das schnelle Nachvollziehen von fremden aber auch von eigenem Code. Jede Zeile zu kommentieren, ist im Allgemeinen jedoch Unsinn. Empfehlenswert ist es, Funktionen und wichtigen Abschnitten einen Kommentar voranzustellen. Ausserdem sollten die wichtigsten Variablen erläutert werden. Beim Kommentieren einer Funktion sollte festgehalten werden, für welchen Wertebereich die Funktion definiert ist.

12. Quellen im Internet

[1] www.cppreference.com (Gut zum Nachschlagen der Standard C/C++ Bibliotheken)
[2] www.cplusplus.com/doc/tutorial (Ein alternatives Tutorial in Englisch)
[3] www.parashift.com/c++-faq-lite (Ein FAQ zu manch interessanten Detailfragen)