II ELEMENTARES PROGRAMMIEREN IN C++

S. BARTELS, 26.3.2018

1. Aufbau eines Programms

Wir diskutieren typische Aspekte eines Computerprogramms am Beispiel der Programmiersprache C++, jedoch finden sich die nachfolgenden Grundkonzepte in jeder Programmiersprache wieder. Der Aufbau eines Programms in C++ hat stets die in Abbildung 1 gezeigte Struktur.

```
1 // Kommentar
2 #include <...>
3 type funktions_name(...){...}
4 int main(){
5   type variable;
6   anweisung;
7 }
```

Abbildung 1. Struktureller Aufbau eines C++-Programms.

Zeile 1 Es ist meist hilfreich ein Programm mit einem Kommentar zu beginnen, der zum Beispiel die Funktionsweise des Programms erläutert.

Zeile 2 Mit der Anweisung `#include<...>` werden Bibliotheken, die beispielsweise Befehle für Ein- und Ausgabe enthalten, eingebunden.

Zeile 3 Eine Unterfunktion wird durch Festlegung des Rückgabetyps, des Namens, der Argumente in runden Klammern sowie die in geschweiften Klammern aufgeführten konkreten Berechnungen definiert.

Zeile 4 Der Hauptteil des Programms ist eine Funktion mit dem Namen `main`. Die öffnende geschweifte Klammer markiert den Beginn der Anweisungen des Hauptprogramms. Der Rückgabewert ist hier als ganzzahlig festgelegt, kann aber auch weggelassen werden.

Zeile 5 In der Regel werden zunächst Variablen deklariert und häufig auch initialisiert, das heißt ihnen wird direkt ein Wert zugeordnet.

Zeile 6 Es folgen weitere Anweisungen wie Wertzuweisungen für Variablen, Funktionsaufrufe mit Variablen, Fallunterscheidungen, Kommentare, Schleifen oder weitere Variablendefinitionen.

Zeile 7 Die schließende geschweifte Klammer markiert das Ende der Berechnungen des Hauptprogramms.

2. Beispielprogramme

Wir betrachten einige Beispielprogramme und diskutieren anschließend die Funktionsweisen der verwendeten Befehle. Die Programme werden unter UNIX mit dem Kommando

```
g++ progname.cc -o progname.out
```

kompiliert beziehungsweise in Maschinensprache übersetzt und mittels

```
./progname.out
```

ausgeführt. In Browser-Compilern wie *C++ Shell*, der unter `http://cpp.sh/` verfügbar ist, können Programme direkt durch Wahl der Schaltfläche *Run* kompiliert und gestartet werden.

2.1. **Textausgabe.** Das in Abbildung 2 gezeigte Programm produziert nach Übersetzung und Ausführung die Ausgabe `etwas Text ...` mit anschließendem Zeilenumbruch auf dem Bildschirm. Nach einem Kommentar zum Kompilieren in der ersten Zeile folgt in der zweiten die Einbindung einer Bibliothek zur Ein- und Ausgabe von Informationen. Das Hauptprogramm besteht nur aus dem in Zeile 4 aufgeführten Druckbefehl.

```
1 // kompilieren: g++ ausgabe.cc -o ausgabe.out
2 #include <iostream>
3 int main(){
4   std::cout << "etwas Text ... " << std::endl;
5 }
```

Abbildung 2. Ausgabe von Text auf dem Bildschirm.

2.2. **Variablen.** Eine simple Addition von drei Variablen, die ganzzahlige Werte enthalten dürfen, ist in Abbildung 3 gezeigt. Bevor den Variablen Werte zugeordnet werden können, müssen sie deklariert werden. Dies kann direkt in einer Anweisung geschehen.

```
1 // kompilieren: g++ addition.cc -o addition.out
2 #include <iostream>
3 int main(){
4   int j, k, sum;
5   int m = 5;
6   j = 3; k = -4;
7   sum = j+k+m;
8   std::cout << "Summe von j, k und m ist " << sum << std::endl;
9 }
```

Abbildung 3. Addition ganzzahliger Variablen.

2.3. **Fallunterscheidungen.** Ein zentraler Bestandteil von Programmen sind Fallunterscheidungen, die abhängig von der Wahrheit einer oder mehrerer Aussagen unterschiedliche, in geschweiften Klammern eingefasste Programmteile ausführen. Das in Abbildung 4 gezeigte Programm prüft, ob eine eingegebene Zahl negativ ist.

```
1  // kompilieren: g++ faelle.cc -o faelle.out
2  #include <iostream>
3  int main(){
4    double x;
5    std::cout << "(Dezimal-) Zahl eingeben: ";
6    std::cin >> x;
7    if (x<0){
8      std::cout << "Die Zahl ist negativ" << std::endl;
9    }
10   else if (x>0){
11     std::cout << "Die Zahl ist positiv" << std::endl;
12   }
13   else{
14     std::cout << "Die Zahl ist Null" << std::endl;
15   }
16 }
```

ABBILDUNG 4. Fallunterscheidung zum Test auf Negativität.

2.4. **Aufzählungen.** Computer eignen sich sehr gut für simple Zählaufgaben, wie zum Beispiel das simple Zählen der Tage zwischen zwei Daten oder die Bildung einer großen Summe $1+2+3+\cdots+J$ für eine Zahl $J \geq 1$. Für solche Zwecke sind sogenannte for-Schleifen geeignet, deren Verwendung in Abbildung 5 gezeigt ist.

```
1  // kompilieren: g++ aufzaehlung.cc -o aufzaehlung.out
2  #include <iostream>
3  int main(){
4    int summe = 0, J;
5    std::cout << "Zahl J eingeben: "; std::cin >> J;
6    for (int j=1; j<=J; j=j+1){
7      summe = summe+j;
8    }
9    std::cout << "Die Summe der Zahlen von 1 bis J ist ";
10   std::cout << summe << std::endl;
11 }
```

ABBILDUNG 5. Verwendung einer for-Schleife zur Berechnung einer Summe.

2.5. **Wiederholungen.** Ein weiteres wichtiges Konzept ist das der Wiederholung gewisser Befehle, bis eine vorgegebene Bedingung erfüllt ist. Das in Abbildung 6 gezeigte Programm summiert so lange die natürlichen Zahlen $1, 2, 3, \ldots$, bis die Summer größer als eine vorgegebene Schranke ist.

```
// compile: g++ wiederholung.cc -o wiederholung.out
#include <iostream>
int main(){
  int summe = 0, j = 0, L, J;
  std::cout << "Summenschranke L: "; std::cin >> L;
  while (summe <= L){
    j = j+1;
    summe = summe+j;
  }
  J = j;
  std::cout << "Ueberschreiten der Schranke L durch";
  std::cout << "Summe 1+2+...+J fuer J = " << J << std::endl;
}
```

ABBILDUNG 6. Wiederholte Addition von Zahlen bis zur Überschreitung einer Schranke.

2.6. **Unterfunktionen.** Wenn bestimmte Rechenschritte oder Operationen in gleicher Weise an unterschiedlichen Stellen eines Programms benötigt werden, ist es sinnvoll eine (Unter-) Funktion zu definieren. Eine Funktion, die den Mittelwert zweier Dezimalzahlen zurück gibt, ist in Abbildung 7 gezeigt. Man beachte, dass der Wert von `x` im Hauptprogramm unverändert bleibt, selbst wenn die Unterfunktion die Variable modifizieren würde beispielsweise mit einer Zuweisung `x = 1.0;` oder einer anderen Anweisung.

```
// compile: g++ unterfunktion.cc -o unterfunktion.out
#include <iostream>
double mittelwert(double x, double y){
  double mittel;
  mittel = (x+y)/2.0;
  return mittel;
}
int main(){
  double x = 10.1, y = 22.3, z;
  z = mittelwert(x,y);
  std::cout << "Mittelwert ist " << z << std::endl;
}
```

ABBILDUNG 7. Unterfunktion zur Berechnung des Mittelwerts zweier Dezimalzahlen.

2.7. **Grundfunktionen.** Durch Einbinden der Bibliothek `cmath` lassen sich mathematische Grundfunktionen wie trigonometrische Funktionen und Potenzfunktionen verwenden. Abbildung 8 zeigt beispielhaft die praktische Realisierung

```
1 // kompilieren: g++ grundfunktionen.cc -o grundfunktionen.out
2 #include <iostream>
3 #include <cmath>
4 int main(){
5   double x = 3.5, y;
6   y = sin(1.0*M_PI)+exp(x)+pow(x,2.0)+fabs(-5.0)+sqrt(2.0);
7   std::cout << "y ist " << y << std::endl;
8 }
```

Abbildung 8. Verwendung mathematischer Grundfunktionen.

3. Bestandteile einer Programmiersprache

Programmiersprachen stellen Kommandos, auch als Befehle oder Anweisungen bezeichnet, zur Bearbeitung von Daten zur Verfügung, die sich in drei Kategorien einteilen lassen: Variablen, Kontrollstrukturen und Funktionen.

3.1. **Variablen.** Variablen dienen der Speicherung von Daten und ihre Verwendung ähnelt der in der Mathematik üblichen. Um effizient mit Speicher umzugehen und eine sinnvolle Verwendung der Variablen in Funktionen zu garantieren, muss der *Typ* einer Variable deklariert werden. Eine boolesche Variable benötigt zum Beispiel weniger Speicherplatz als eine Variable zur Speicherung einer Dezimalzahl. Variablen können im Laufe eines Programms verschiedene Werte zugeordnet werden. Programmiersprachen stellen Operationen wie arithmetische Ausdrücke und Vergleiche für unterschiedliche Typen zur Verfügung.

3.2. **Kontrollstrukturen.** Algorithmen enthalten häufig Fallunterscheidungen und Wiederholungen, die auch als *Selektoren* und *Iteratoren* bezeichnet werden. Beispielsweise soll ein Teil eines Algorithmus beziehungsweise eines Programms nur dann ausgeführt werden, wenn eine gewisse Bedingung erfüllt ist, oder eine Anweisung soll so lange wiederholt werden, bis eine vorgegebene Bedingung erfüllt ist. Kontrollstrukturen können geschachtelt verwendet werden.

3.3. **Funktionen.** Programmiersprachen stellen viele wichtige Operationen wie das Testen auf Gleichheit zweier Variablen oder die Berechnung der Quadratwurzel nichtnegativer Dezimalzahlen zur Verfügung. Weitere Funktionen können meist selbst definiert werden und ihre Verwendung entspricht wiederum sehr der in der Mathematik üblichen. Eine Funktion hat gewisse

Eingabeparameter, die auch als Argumente bezeichnet werden, und Ausgabeparameter, die dem oder den Werten der Funktion entsprechen. Beim Test auf Gleichheit sind die Eingabeparameter zwei Variablen gleichen Typs und der Ausgabeparameter eine boolesche Variable. Unterfunktionen können eigene Variablen enthalten, deren Wertzuweisung nur lokal, das heißt nicht für das Hauptprogramm, Auswirkungen haben.

4. Ausdrücke, Zuweisungen und Blöcke

Um mit Variablen arbeiten und sie in Kontrollstrukturen und Funktionen sinnvoll verwenden zu können, werden Ausdrücke, Zuweisungen und Blöcke benötigt.

4.1. **Ausdrücke.** Ein *Ausdruck* ist eine zulässige Formel, die Operationen mit Variablen und Konstanten enthält und deren Auswertung bei gültiger Belegung der Variablen zu einem Wert eines bestimmten Typs führt. Abstrakt können Ausdrücke induktiv definiert werden: konstante Werte und Variablen sind Ausdrücke und durch zulässige Verknüpfungen mittels Operationen eines oder mehrerer Ausdrücke entsteht ein neuer Ausdruck. Die Operationen können arithmetische Ausdrücke oder Unterfunktionen und die Variablen Dezimalzahlen sein wie beispielsweise in dem folgenden Ausdruck:

```
y+2.0*(x-sin(z))
```

Hier sollten `x,y,z` Dezimalzahlen sein, damit alle Operationen und der Ausdruck als Dezimalzahl wohldefiniert sind. Häufig verwendet werden auch Ausdrücke, deren Wert bei gültiger Auswertung ein boolescher Wert ist:

```
((j>5) && (j<10)) || (!b)
```

Hier ist `j` eine ganzzahlige und `b` eine boolesche Variable und der Ausdruck hat den Wert `true`, sofern der Wert `j` echt zwischen 5 und 10 liegt oder die boolesche Variable den Wert `false` oder `0` besitzt.

4.2. **Zuweisungen.** Variablen können Werte *zugewiesen* werden, die ihrem Typ entsprechen. Die Werte sind konstante oder variable Ausdrücke wie beispielsweise:

```
j = 5+6;
x = sin(1.0);
b = (j<10) || !(j>20);
```

Wird einer Variablen eines bestimmten Typs ein Wert eines anderen Typs zugeordnet, so kann eine Fehlermeldung entstehen, beispielsweise in folgendem Fall:

```
unsigned int j;
j = -3;
```

In C++ wird bei eigentlich unpassender Zuweisung versucht, eine Typenkonvertierung durchzuführen, das heißt beispielsweise, dass eine Dezimalzahl bei der Zuweisung automatisch auf eine ganze Zahl ab- oder aufgerundert wird.

4.3. **Blöcke.** Unter einem *Block* versteht man eine Folge von Anweisungen, die in geschweiften Klammern eingefasst sind. Die Anweisungen können dabei Variablendefinitionen, Wertzuweisungen oder Kontrollstrukturen sein. Blöcke definieren Unterfunktionen oder auch die Hauptfunktion eines Programms in C++. Ein Beispiel eines Blocks ist:

```
{int j = 0, k; bool b;
 k = 2*j; b = (j<k);}
```

Blöcke treten auch in Fallunterscheidungen und Wiederholungen auf.

5. Syntaktische Konventionen in C++

Es gibt einige generelle Regeln, die in C++ zu beachten sind:

- Variablennamen bestehen aus Buchstaben, Ziffern und Unterstrichen, wobei das erste Zeichen keine Ziffer sein darf und der Variablenname nicht nur aus einem Unterstrich bestehen darf.
- Es wird zwischen Groß- und Kleinbuchstaben unterschieden, das heißt beispielsweise, dass `Hausnr` und `HausNr` zwei unterschiedliche Variablen sind.
- Schlüsselwörter wie `if`, `else`, `while`, `true` oder `false` dürfen nicht als Variablen- oder Funktionsnamen verwendet werden.
- Zuweisungen werden in C++ stets mit einem Semikolon beendet.
- Leerzeilen, Leerzeichen und Zeilenumbrüche haben nur in wenigen speziellen Situation eine Bedeutung und werden andernfalls vom Compiler ignoriert. Ein Zeilenumbruch muss beispielsweise auf eine Compiler-Anweisung wie `#include<...>` folgen und ein Leerzeichen ist bei der Deklaration einer Variablen erforderlich.
- Bei Blöcken, die nur aus einer Anweisung bestehen, können die geschweiften Klammern weggelassen werden.

6. Wichtige Anweisungen in C++

Wir diskutieren nachfolgend die syntaktischen und semantischen Eigenschaften der wichtigsten Anweisungen des elementaren Programmierens in C++.

6.1. **Variablendeklarationen.** Eine Variablendeklaration deklariert eine Variable und besteht aus der Angabe des Typs der Variablen wie `int`, `double` oder `bool` und dem Variablennamen. Mehrere Variablen gleichen Typs können gemeinsam deklariert werden. Einer Variablen kann bei der Deklaration ein Wert zugeordnet werden:

```
type var;
type var = value;
```

6.2. **Wertzuweisungen.** Eine Wertzuweisung für eine Variable erfolgt durch Angabe der Variablennamen gefolgt von einem Gleichheitszeichen und einem gültigen Ausdruck, der bei seiner Auswertung einen Wert des Typs der Variable liefert:

```
var = expression;
```

6.3. **Fallunterscheidungen.** Eine Fallunterscheidung besteht mindestens aus dem Schlüsselwort `if`, einer booleschen Variablen `b` und einem Block, der ausgeführt wird, wenn `b` als wahr ausgewertet wird. Weitere Alternativfälle können mit oder ohne Bedingung folgen:

```
if (b){...} [else if (c){...} ... else{...}]
```

6.4. **Aufzählungen.** Eine Aufzählung oder for-Schleife besteht aus dem Schlüsselwort `for` einer Initialisierung, einer Abbruchbedingung und einer Iterationsanweisung sowie einem Befehlsblock. Nach der Initialisierung werden der Befehlsblock und die Iterationszuweisung so lange abwechselnd ausgeführt, wie die Abbruchbedingung als falsch ausgewertet wird:

```
for (ini; b_stop; iter){...}
```

6.5. **Wiederholungen.** Eine Wiederholung besteht aus einer Abbruchbedingung und einem Befehlsblock. Die Abbruchbedingung wird wiederholt ausgewertet und nach jeder wahren Auswertung wird der Befehlsblock ausgeführt.

```
while (b){...}
```

6.6. **Unterfunktionen.** Eine Unterfunktion besteht aus einem einfachen Rückgabetyp, dem Funktionsnamen, einer Deklaration von Variablen, die als Funktionsargumente verwendet werden, und einem Befehlsblock, der diese und weitere Variablen verwendet und mit dem Rückgabebefehl `return` den Wert der Berechnungen zurückgibt:

```
type fun(type arg1, ..., type argn){... return expr; ...}
```

7. Namensumgebungen und Inkrementoperatoren

Abschließend diskutieren wir übliche Abkürzungen einiger Befehle, die ein effizientes Programmieren ermöglichen, aber mit gewisser Vorsicht verwendet werden sollten.

7.1. **Namensumgebung.** Die Verwendung der Namensumgebung `std` erlaubt es, die Befehle zur Ein- und Ausgabe aus der Bibliothek `iostream` ohne den Zusatz `std::` zu verwenden. Unabhängig davon kann statt des Zeilenumbruchs mittels `endl` das Steuerzeichen `\n` benutzt werden, siehe Abbildung 9. Etwas veraltet ist das Kommando `fprint` aus der Bibliothek `stdio.h` für die formatierte Ausgabe von Text und Variablen.

7.2. **Inkrementoperatoren.** Sehr häufig wird bei einer Zuweisung der Wert einer Variablen durch Addition zu der Variablen definiert. In diesem Fall kann die Zuweisung folgendermaßen vereinfacht werden:

```
a += expr;   ⟺   a = a+expr;
```

Ist der Ausdruck durch den konstanten Wert 1 gegeben, so können die Zuweisungen weiter vereinfacht werden.

```cpp
// kompilieren: g++ namensumgebung.cc -o namensumgebung.out
#include <iostream>
using namespace std;
int main(){
  cout << "etwas Text ... " << endl;
  cout << "alternativer Zeilenumbruch \n";
}
```

ABBILDUNG 9. Verwendung einer Namensumgebung und eines alternativen Zeilenumbruchkommandos.

$$\texttt{j++} \iff \texttt{j = j+1;} \iff \texttt{j += 1;}$$

Wird der Befehl `j++` in einem Ausdruck wie zum Beispiel `5-j++` verwendet, so wird erst der umfangreichere Ausdruck mit dem Wert von `j` ausgewertet und anschließend der Wert von `j` erhöht. Soll die Veränderung von `j` vorher geschehen, so muss die Operation `++j` benutzt werden. Eine typische Verwendung von Inkrementoperatoren ist in Abbildung 10 gezeigt.

```cpp
// kompilieren: g++ inkrement.cc -o inkrement.out
#include <iostream>
using namespace std;
int main(){
  int summe = 0, J;
  cout << "Zahl J eingeben: "; cin >> J;
  for (int j=1; j<=J; j++) summe += j;
  cout << "Summe von 1 bis J ist " << summe << endl;
}
```

ABBILDUNG 10. Verwendung von Inkrementoperatoren in einer Aufzählung zur Berechnung der Summe $S = 1 + 2 + \cdots + J$.

7.3. **Typen in C++.** Die wichtigsten Typen zur Deklaration von Variablen sind `bool`, `int` und `double` für boolesche, ganzzahlige und Dezimalzahlwertige Variablen. Der leere Typ `void` wird verwendet, wenn eine Funktion keinen Rückgabewert hat. Der jeweilige Speicherbedarf und die jeweiligen Operationen sind in Tabelle 1 dargestellt. Der Speicherbedarf beziehungsweise die maximale Größe einer Zahl eines bestimmten Typs ist Compilerabhängig, kann aber mit dem Kommando `sizeof` festgestellt werden, wie in Abbildung 11 gezeigt ist. Für die Operationen gelten gewisse Prioritäten bei der Auswertung wie beispielsweise Punkt- vor Strichrechnung sowie NEGATION vor UND vor ODER. Gerade bei booleschen Ausdrücken sollte man sinnvollen Gebrauch von Klammern machen, die die höchste Priorität

haben. Mit dem Typ `char` können Variablen deklariert werden, deren Werte Buchstaben sind.

Typ	Werte	Speicher	Operationen	Vergleiche
`void`	–	0 Bit	–	–
`bool`	$\{0, 1\}$	1 Bit	`&&,||,!`	`==,!==`
`int`	$[\pm 2147483647]$	4 Bytes	`+,-,*,/,%`	`<,>,==,<=,>=,!==`
`double`	$[\pm 1,7 \cdot 10^{308}]$	8 Bytes	`+,-,*,/`	`<,>,==,<=,>=,!==`

TABELLE 1. Wichtigste Zahlenyypen in C++ und darauf definierte Operationen und logische Vergleiche. Die Symbole ! und % realisieren die logische Negation und die Modulo-Operation (Rest bei Division).

```
1  // kompilieren: g++ typen.cc -o typen.out
2  #include <iostream>
3  #include <limits.h>
4  using namespace std;
5  int main(){
6    cout << sizeof(int) << endl;
7    cout << sizeof(double) << endl;
8    cout << INT_MIN << ", " << INT_MAX << endl;
9    cout << sizeof(char) << endl;
10  }
```

ABBILDUNG 11. Bestimmung des Speicherbedarfs von Typen sowie minimaler und maximaler darstellbarer Zahlen ganzzahligen Typs. Ein Byte entspricht 8 Bits.

7.4. **Typumwandlung.** Semantisch fehlerhafte Wertzuweisungen führen in C++ selten zu Fehlermeldungen, da in den meisten Fällen eine automatische Anpassung des Typs vollzogen wird, das heißt zum Beispiel, dass für Variablen `j` und `x` vom Typ `int` und `double` die Zuweisungen

```
j = 3.74;
x = 1;
```

auf die Belegungen 3 und 1 führen, die Nachkommstellen der Dezimalzahl `3.74` werden also abgeschnitten. In Ausdrücken kann eine ungenaue Typverwendung jedoch zu fehlerhaften Berechnungen führen. Die Zuweisung

```
j = 1;
x = j/2;
```

führen auf die Belegung 0 von `x`, da die Division ganzer Zahlen als Division ohne Rest durchgeführt wird. Verwendet man statt des Werts `2` jedoch `2.0` so ist ist der Divisor als Dezimalzahl deklariert und das Ergebnis der Division ist ebenfalls eine Dezimalzahl, das heißt durch

```
j = 1;
x = j/2.0;
```

erhält man die Belegung 0.5. In Dezimalzahlrechnungen sollte man daher für konstante Werte die Form `1.0` statt `1` verwenden; möglich und gebräuchlich ist auch das Weglassen der Null, das heißt die Verwendung von `1.` statt `1.0`. Allgemein orientieren sich die arithmetischen Operationen immer am höherwertigen Typ der beteiligten Variablen oder Werte. Eine explizite Typumwandlung kann mit den Befehlen

```
j = (int)1.0;
x = (double)1;
```

erfolgen. Ein Beispielprogramm ist in Abbildung 12 gezeigt.

```
1  // kompilieren: g++ typumwandlung.cc -o typumwandlung.out
2  #include <iostream>
3  using namespace std;
4  int main(){
5    int j = 1;
6    double x;
7    x = j/2;
8    cout << x << endl;
9    x = j/2.0;
10   cout << x << endl;
11   j = (int)x;
12   cout << j << endl;
13  }
```

ABBILDUNG 12. Implizite und explizite Umwandlung von Typen in C++.