

Programmierrichtlinien für C++

Prof. Dr. Oliver Bittel
Fachbereich Informatik
Fachhochschule Konstanz

Version 2.1
Datum: 19.2.1998

Hinweis zur Version 2.1:

Außer einigen Schreibfehlern, Formulierungen und erklärenden Ergänzungen wurde gegenüber Version 1.0 folgendes geändert:

alte Nummer	neue Nummer	Änderung
2.5	2.5	Klassen müssen nicht mit C beginnen; z.B. stack statt Cstack .
3.1.c	3.1.c	bei Funktionen darf vor jedem Parameter ein Leerzeichen stehen; also auch nach der öffnenden Klammer.
3.3	3.3	Einrücktiefe beträgt 3 (statt 4) Leerzeichen
	6	Neu! Stroustrup-Stil bei Zeiger- und Referenztypen.
7.3		Regel für Default-Konstruktor wurde gestrichen.
	8.4	Neu! Regeln für virtuelle Methoden.

1 EINLEITUNG	3
2 BEZEICHNER	3
2.1 ALLGEMEINE REGELN	3
2.2 VARIABLEN	4
2.3 KONSTANTEN	4
2.4 FUNKTIONEN	4
2.5 BENUTZERDEFINIERTER DATENTYPEN UND KLASSEN	5
2.6 PRÄPROZESSORBEZEICHNER	5
3 FORMATIERUNG.....	5
3.1 LEERZEICHEN	5
3.2 LEERZEILEN	6
3.3 EINRÜCKUNG.....	6
4 KONTROLLSTRUKTUREN	6
4.1 ZUSAMMENGESetzte ANWEISUNGEN UND BLÖCKE.....	7
4.2 IF- UND IF-ELSE-ANWEISUNGEN	7
4.3 ELSE-IF-ANWEISUNGEN	8
4.4 SWITCH-ANWEISUNGEN.....	8
4.5 WHILE-SCHLEIFEN.....	9
4.6 DO-WHILE-SCHLEIFEN	9
4.7 FOR-SCHLEIFEN.....	9
4.8 SCHLEIFEN MIT MEHRFACHEM AUSSTIEG (BREAK-ANWEISUNG).....	9
5 FUNKTIONEN.....	10
5.1 FORMATIERUNG.....	10
5.2 PARAMETER	10
5.3 FUNKTIONEN MIT MEHRFACHEM AUSSTIEG	12
5.4 INLINE-FUNKTIONEN	13
6 ZEIGER- UND REFERENZTYPEN.....	13
7 STRUKTURDATENTYPEN	14
8 KLASSEN.....	14
8.1 AUFBAU UND FORMATIERUNG VON KLASSEN	14
8.2 CONST-METHODEN	15
8.3 OBJEKTE MIT DYNAMISCHEM ANTEIL: KONSTRUKTOREN UND DESTRUKTOR	15
8.4 VERERBUNG UND VIRTUELLE METHODEN	16
9 SONSTIGES	16
9.1 BENUTZERDEFINIERTER TYPEN	16
9.2 KONSTANTEN	17
9.3 ADRESSARITHMETIK	17
9.4 EIN/AUSGABE.....	18
9.5 DYNAMISCHER SPEICHER	19
10 MODULARISIERUNG.....	19
10.1 HAUPTMODUL.....	19
10.2 KLASSENMODUL	20
10.3 FUNKTIONSMODUL.....	20
10.4 TYPDEFINITIONEN.....	21
10.5 GLOBALE KONSTANTEN	21
10.6 GLOBALE VARIABLEN	22
11 DOKUMENTATION	22
12 LITERATURVERZEICHNIS.....	24

1 Einleitung

Programmierrichtlinien sollen helfen, einheitliche und daher besser verständliche Programme zu erstellen. Wie beispielsweise bei einem CD-Spieler für die Grundfunktionalität typischerweise einheitliche Bedienfelder gegeben sind, so sollen in einem Programm Kontrollstrukturen und Bezeichner eine einheitliche Form besitzen.

Programmierrichtlinien bestehen aus einer Sammlung von Regeln, die über die Sprachsyntax hinausgehen. Sie werden daher auch nicht vom Compiler erzwungen, sondern müssen durch Programmierdisziplin umgesetzt werden. Üblicherweise wird im Rahmen von *Code-Reviews* (Teil des Softwareentwicklungsprozeß) das Einhalten von Programmierrichtlinien überprüft.

Das Einhalten von Programmierrichtlinien ist ein wichtiges Software-Qualitätsmerkmal. Die Regeln sollten daher immer befolgt werden. Ausnahmen sind zwar gestattet, sind aber immer gesondert zu begründen. Insbesondere dienen die Richtlinien folgenden Zielen:

- Einheitliche, einer bestimmten Form genügende Programme sind leichter lesbar und daher besser zu erweitern, anzupassen oder auf Fehler zu prüfen.
- Die Einarbeitung in fremde Programme wird erleichtert.
- Das Einschränken von Programmstrukturen auf bestimmte Standardformen ist weniger fehleranfällig.

In der Informatikausbildung an der FH Konstanz wird C++ in vielen Veranstaltungen eingesetzt: Programmiertechnik, Algorithmen und Datenstrukturen, Objektorientiertes Programmieren, Software-Engineering, etc. Die vorliegende Richtlinienbeschreibung soll auch dazu dienen, in all diesen Veranstaltungen einen einheitlichen Programmierstil zu erreichen. Es ist jedoch zu beachten, daß in den Veranstaltungen in den unteren Semestern nicht alle C++-Sprachkonzepte eingeführt werden und daher möglicherweise nicht alle Richtlinien zum Tragen kommen.

Einige Teile der Richtlinien sind [Balzert 96] entnommen. Es sei schließlich noch darauf hingewiesen, daß die Richtlinien keinen Ersatz für ein Programmiersprachenhandbuch darstellen sondern eine Ergänzung.

2 Bezeichner

Als Bezeichner sollen natürlichsprachige oder problemnahe Namen oder bei zu lang werdenden Namen verständliche Abkürzungen verwendet werden. Prinzipiell sollte die Aussagekraft bzw. Länge eines Namens mit der Größe seines Gültigkeitsbereichs zunehmen. Insbesondere gelten folgende Regeln.

2.1 Allgemeine Regeln

- a) Kein Bezeichner beginnt mit einem Unterstrich (wird oft von System-Präprozessornamen verwendet).
- b) Bei zusammengesetzten Namen wird jeder angehängte Namen groß geschrieben; z.B.

```
int anzahlWorte;  
int windGeschw;  
  
class BinaryTree;
```

Zusammengesetzte Namen mit Unterstrich (z.B **wind_Geschw**) sind zu vermeiden.

2.2 Variablen

- Variablen beginnen grundsätzlich mit einem Kleinbuchstaben.
- Für Laufvariablen in for-Schleifen sollten Kleinbuchstaben wie i, j, k verwendet werden.
- Das Präfix a-/ein- besitzt eine spezielle Bedeutung:
aTyp bzw. *einTyp* ist eine Variable vom Typ *Typ*, z.B.

```
Person aPerson;
Kunde einKunde;
```

Variablenbezeichner, die sich vom entsprechenden Typbezeichner nur durch einen klein geschriebenen Anfangsbuchstaben unterscheiden, sind nicht gestattet. Z.B. ist `Person person;` nicht erlaubt.

- Für Zeigervariablen kann zur besseren Kennzeichnung das Postfix Z bzw. Ptr verwendet werden; z.B.:

```
double* dbPtr;
Person* aPersonPtr;
Kunde* einKundeZ;
```

- Bei Variablen vom Typ `Bool`¹ sollten Adjektive verwendet werden, Z.B.

```
Bool gefunden;

while (!gefunden)
{
    // ...
}
```

2.3 Konstanten

Konstanten beginnen grundsätzlich mit einem Kleinbuchstaben. Das gilt auch für Konstanten, die durch einen Aufzählungstyp (`enum`) definiert werden. Beispiel:

```
const double pi = 3.14;
const int maxPers = 100;

enum AmpelFarbe {rot, gelb, gruen};
```

2.4 Funktionen²

- Funktionen beginnen grundsätzlich mit einem Kleinbuchstaben.
- Für Funktionen sollten möglichst Verben verwendet werden; z.B.

```
void sortiere(int zahlen[], int n);
void drucke(Complex c);
```

- Geht das relevante Objekt, mit dem die Aktion ausgeführt wird, nicht aus der Parameterliste hervor, dann ist es in dem Namen aufzunehmen; z.B.

```
void druckeKunde(int nr);
```

¹`Bool` wird üblicherweise durch `enum Bool {false,true};` definiert. Im neuen C++-ANSI-Standard gibt es hierfür den eingebauten Datentyp `bool` mit den beiden Werten `false` und `true`.

²Gilt auch für Methoden.

2.5 Benutzerdefinierte Datentypen und Klassen

Benutzerdefinierte Datentypen und Klassen beginnen grundsätzlich mit einem Großbuchstaben. Beispiele:

```
typedef int Alter;

struct Complex
{
    double real;
    double im;
};

class Person
{
public:
    void setAlter(int n);
    int getAlter() const;
    // ...
private:
    int alter;
    // ...
};

enum Monat {jan, feb, mar, apr, mai, jun,
            jul, aug, sep, okt, nov, dez};
```

2.6 Präprozessorbezeichner

Mit `#define` definierte Präprozessorbezeichner bestehen grundsätzlich nur aus Großbuchstaben. Zur Hervorhebung zusammengesetzter Namen darf der Unterstrich verwendet werden. Z.B.

```
#define PI 3.14;
#define MAX_LIST 100;
```

Auf Präprozessorbezeichner sollte zugunsten von Konstanten (2.3) und inline-Funktionen (5.4) verzichtet werden.

3 Formatierung

3.1 Leerzeichen

a) Bei binären Operatoren werden Operanden durch jeweils ein Leerzeichen getrennt, z.B.

```
x = y + z;
```

b) Zur besseren Erkennung von Teilaudrücken dürfen Leerzeichen weggelassen werden, z.B.

```
x = 2*y + 3*z;
if (1 <= x+y && x+y <= 10) // ...
x = a[i+j];
```

Bei Referenzierungen, Dereferenzierungen, Feldzugriffen und Komponentenzugriffen bei Strukturen und Objekten stehen keine Leerzeichen, z.B.

```
xPtr = &x;
y = *xPtr;
y = a[i][j];
cout << person.alter << endl;
cout << personPtr->alter << endl;
```

- c) Zwischen Funktionsname und Klammer steht kein Leerzeichen. Nach der öffnenden und vor der schließenden Klammer stehen keine Leerzeichen. Von dieser Regel abweichend darf nach einer öffnenden Funktionsklammer ein Leerzeichen stehen. Geschieht dies, dann sollte vor jedem Funktionsparameter ein Leerzeichen eingefügt werden.

Beispiel:

```
y = sin(x);
z = pow(x, y);

for (i = 0; i < 10; i++) //...
```

- d) Nach Schlüsselwörtern (if, while, for, else, etc.) steht grundsätzlich ein Leerzeichen (wenn nicht bereits eine Leerzeile folgt).

3.2 Leerzeilen

- Umfangreiche (mehrzeilige) Definitionen (von Funktionen, Strukturdatentypen, Klassen, etc.) sind durch Leerzeilen zu trennen.
- In einer Funktion sind Deklarationsteil und Anweisungsteil durch eine Leerzeile zu trennen.
- Umfangreiche Kontrollstrukturen sind durch Leerzeilen zu trennen.

3.3 Einrückung

Bei allen Kontrollstrukturen, Strukturdatentypen und Klassen gilt eine einheitliche Einrücktiefe. Eine Einrücktiefe von 3 Leerzeichen ist vermutlich die am meisten verwendete Einrücktiefe.

```
struct Point
{
    double x;
    double y;
};

if (Bedingung1)
{
    Anweisung1;
    if (Bedingung2)
        Anweisung2;
}
```

Bei manchen Kontrollstrukturen und bei Klassendefinitionen sind bei der Einrückung einige Besonderheiten zu beachten (siehe Kapitel 4 und 8).

4 Kontrollstrukturen

Es sind nur Kontrollstrukturen in der folgenden beschriebenen Form gestattet. Auf **goto** sollte aus bekannten Gründen (Strukturierte Programmierung) grundsätzlich verzichtet werden. **continue** sollte vermieden werden.

4.1 Zusammengesetzte Anweisungen und Blöcke

Zusammengesetzte Anweisungen und Blöcke³ werden mit geschweiften Klammern eingrahmt, die jeweils in einer Zeile für sich stehen; z.B.

```

{
    sum = 0;
    for (i = 0; i < n; i++)
        sum += a[i];
    cout << sum;
}

{
    int sum = 0;

    for (i = 0; i < n; i++)
        sum += a[i];
    cout << sum;
}
    
```

4.2 If- und if-else-Anweisungen

Ist der then- bzw. else-Teil eine zusammengesetzte Anweisung (oder ein Block), so wird diese nicht eingerückt. In allen anderen Fällen wird eingerückt. Ist die then- oder else-Anweisung mehrzeilig, dann ist eine zusätzliche Einklammerung mit geschweiften Klammern empfehlenswert.

Beispiel:

```

if (x > 0)
{
    sum = sum + x;
    n++;
}

if (x >= y)
{
    if (x >= z)
        max = x;
    else
        max = z;
}
else
{
    if (y >= z)
        max = y;
    else
        max = z;
}
    
```

³Ein Block ist eine zusammengesetzte Anweisung mit Deklarationsteil.

4.3 Else-if-Anweisungen

Um bei else-if-Kaskaden eine zu starke Einrückung des Programmtextes zu vermeiden, sollten else und if in eine Zeile geschrieben werden; z.B.

```

if (0 <= x && x <= 1)
    Anweisung1;
else if (1 < x && x <= 2)
    Anweisung2;
else if (2 < x && x <= 3)
    Anweisung3;
else // x < 0 || 3 < x
    Anweisung4;
    
```

4.4 Switch-Anweisungen

Es ist darauf zu achten, daß die einzelnen case-Teile mit break abgeschlossen werden. Der Default-Teil steht grundsätzlich als letzter Fall in der switch-Anweisung. Der default-Teil darf entfallen.

Beispiel:

```

switch (c)
{
    case 'a':
        anzA++;
        break;
    case 'b':
        anzB++;
        break;
    default:
        anzSonst++;
        break;
}
    
```

Werden mehrere Fälle in der gleichen Weise behandelt, dürfen die entsprechenden case-Ausdrücke in einer Folge geschrieben werden; z.B.

```

switch (c)
{
    case 'a':
    case 'A':
        anzA++;
        break;
    case 'b':
    case 'B':
        anzB++;
        break;
    default:
        anzSonst++;
        break;
}
    
```

4.5 While-Schleifen

Ist der Schleifenrumpf eine zusammengesetzte Anweisung, so wird diese nicht eingerückt. In allen anderen Fällen wird eingerückt. Ist der Schleifenrumpf eine mehrzeilige Anweisung, dann ist eine zusätzliche Einklammerung mit geschweiften Klammern empfehlenswert.

Beispiel:

```
i = 0;
sum = 0;

while (cin >> a[i])
{
    sum = sum + a[i];
    i++;
}
```

4.6 Do-While-Schleifen

Der Schleifenrumpf wird grundsätzlich mit geschweiften Klammern eingeklammert. Das Schlüsselwort **while** steht dabei direkt hinter der schließenden geschweiften Klammer. Dies ist wichtig, um eine Verwechslung mit einer while-Schleife mit leerer Anweisung auszuschließen.

Beispiel:

```
// GGT-Berechnung nach Euklid
do
{
    r = x % y;
    x = y;
    y = r;
} while (x > 0);
```

4.7 For-Schleifen

For-Schleifen sollten ausschließlich zur Realisierung von auf- oder absteigenden Zählschleifen verwendet werden. Die Zählvariable (Laufvariable) darf dabei im Schleifenrumpf nicht verändert werden. Es gilt die gleiche Einrückungsregel wie bei der while-Schleife.

Beispiel:

```
for (i = 1; i <= n; i++)    // aufsteigend
    h = h + 1/double(i);

for (i = n; i >= 1; i--)    // absteigend
    h = h + 1/double(i);
```

4.8 Schleifen mit mehrfachem Ausstieg (Break-Anweisung)

Nach der Methode der Strukturierten Programmierung sollten grundsätzlich Kontrollstrukturen mit genau einem Eingang und einem Ausgang verwendet werden. In manchen Fällen jedoch sind Schleifen mit Mehrfachausstieg leichter verständlich und daher vorzuziehen. Mehrfachausstiege werden mit **break** realisiert. Beispiel:

```

char str[10];

while (cin >> str)
{
    toLower(str);
    if (strcmp(str,"quit") == 0)
        break;
    if (strcmp(str,"stop") == 0)
        break;
    // bearbeite str
}
    
```

5 Funktionen

Diese Regeln gelten auch für Methoden und überladene Operatoren.

5.1 Formatierung

Funktionen sind wie folgt zu formatieren:

```

int sum(int a[], int n)
{
    int s = 0;                // Deklarationsteil
    int i;

    for (i = 0; i < n; i++) // Anweisungsteil
        s += a[i];
    return s;
}
    
```

Besteht ein Funktionsrumpf nur aus einer kurzen Anweisung, kann er vollständig in eine Zeile geschrieben werden; z.B.

```

int anzahl()
{ return maxAnzahl;}
    
```

Wird eine Methode innerhalb einer Klassendefinition definiert (inline-Methode) und besteht ihr Rumpf nur aus einer kurzen Anweisung, dann darf ihre Definition in einer Zeile erfolgen.

5.2 Parameter

Auf der konzeptionellen Ebene gibt es drei Parameterarten: Eingabeparameter, Ausgabe-parameter (Ergebnisparameter) und Ein/Ausgabeparameter (transiente Parameter). Zur Realisierung gibt es in C++ verschiedene Parameterübergabemechanismen. Es sollte wie folgt vorgegangen werden:

a) Eingabeparameter:

Es ist unter den folgenden 3 Realisierungsmöglichkeiten auszuwählen:

- **Wertübergabe (call-by-value):**

Parameter mit Basisdatentypen sollten grundsätzlich mit Wertübergabe übergeben werden.

Bei Übergabe von Strukturen und Objekten ist ihre Größe zu berücksichtigen: bei Wertübergabe wird der aktuelle Parameter komponentenweise in den formalen

Parameter kopiert, was zeitaufwendig sein kann. Die Wertübergabe ist dann nur zu wählen, falls eine Kopie des aktuellen Parameters in der Funktion tatsächlich gewünscht wird. Andernfalls ist die konstante Referenzübergabe (siehe nächsten Punkt) vorzuziehen. Beispiel:

```
struct Point
{
    double x;
    double y;
};

Point add(Point p1, Point p2)
{
    p1.x += p2.x;
    p1.y += p2.y;
    return p1;
}
```

- **Konstante Referenzübergabe (call-by-reference):**

Werden Strukturen oder Objekte übergeben, so kann durch Referenzübergabe aufwendiges Kopieren vermieden werden. Um unerwünschtes Verändern des aktuellen Parameters zu vermeiden (Eingabeparameter!), sollte der formale Parameter mit `const` als konstante Referenz vereinbart werden. Zu beachten ist jedoch, daß auf die formalen Parameter dann nur lesend zugegriffen werden darf. Beispiel:

```
Point add(const Point& p1, const Point& p2)
{
    // Beachte, dass nun p1 und p2 in der Funktion
    // nicht verändert werden dürfen.
    Point p;

    p.x = p1.x + p2.x;
    p.y = p1.y + p2.y;
    return p;
}
```

- **Konstante Zeigerübergabe bei Feldern:**

Felder können nur als Zeiger übergeben werden. Um unerwünschtes Verändern der Feldelemente des aktuellen Parameters zu vermeiden (Eingabeparameter!), sollten die Feldelemente des formalen Parameters mit `const` vereinbart werden. Beispiel:

```
void print(const int daten[], int n)
{
    for (int i = 0; i < n; i++)
        cout << daten[i] << endl;
}
```

Die Schreibweise `int daten[]` ist der gleichwertigen Schreibweise `int* daten` vorzuziehen, da hiermit direkt ersichtlich ist, daß ein Feld übergeben wird.

b) Ausgabeparameter:

Es ist unter den folgenden 3 Realisierungsmöglichkeiten auszuwählen:

- **Referenzübergabe (call-by-reference):**

Beispiel:

```
void add(Point p1, Point p2, Point& p)
// p ist Ausgabeparameter
{
    p.x = p1.x + p2.x;
    p.y = p1.y + p2.y;
}
```

- **Zeigerübergabe bei Feldern:**

Beispiel:

```
void genFib(int fib[], int n)
// fib ist Ausgabeparameter
{
    fib[0] = 0;
    fib[1] = 1;
    for (int i = 2; i < n; i++)
        fib[i] = fib[i-1] + fib[i-2];
}
```

- **Rückgabewert mit return:**

Beispiel:

```
Point add(Point p1, Point p2)
{
    Point p;

    p.x = p1.x + p2.x;
    p.y = p1.y + p2.y;
    return p;
}
```

c) Ein/Ausgabeparameter:

Ein/Ausgabeparameter sollten entweder durch Referenzübergabe oder bei Feldern durch Zeigerübergabe realisiert werden (siehe b) Ausgabeparameter).

5.3 Funktionen mit mehrfachem Ausstieg

Ähnlich wie bei Schleifen mit Mehrfachausstieg sind in manchen Fällen Funktionen mit mehrfachem Ausstieg (mehrere return-Anweisungen) leichter verständlich und daher vorzuziehen. Beispiel:

```
int search(char c, char* text[], int anzZeilen)
{
    for (int i = 0; i < anzZeilen; i++)
        for (int j = 0; text[i][j] != '\0'; j++)
            if (text[i][j] == c)
                return 1;    // gefunden
    return 0;    // nicht gefunden
}
```

5.4 Inline-Funktionen

Grundsätzlich sind inline-Funktionen statt Präprozessor-Makros zu verwenden, da im Gegensatz zum Makromechanismus eine Typüberprüfung der Parameter durch den Compiler stattfindet. Beispiel:

```
inline int max(int x, int y)
{
    if (x >= y)
        return x;
    else
        return y;
}
```

6 Zeiger- und Referenztypen

Für einen beliebigen Datentyp T ist T^* bzw $T\&$ der entsprechende abgeleitete Zeiger- bzw. Referenztyp. Werden Variablen (oder Parameter) von einem Zeiger- oder Referenztyp deklariert, ist einer der beiden folgenden Stile konsistent einzuhalten. Jedoch ist der Stroustrup-Stil vorzuziehen, da bei diesem Stil der Typ der Variablen besser ersichtlich ist.

- **Kernighan-Ritchie-Stil** [Kernighan und Ritchie 1978]:

* (bzw. &) wird zur Variablen geschrieben, z.B.:

```
char *s1, *s2;
Person *p;
```

- **Stroustrup-Stil** [Stroustrup 1991]:

* (bzw. &) wird zum Typ geschrieben, z.B.:

```
char* s1;
char* s2;
Person* p;
```

Da die Schreibweise nicht der syntaktischen Konvention von C++ entspricht, darf höchstens eine Zeigervariable je Deklaration definiert werden. Beispielsweise ist

```
char* s1, s2;
```

gleichwertig zu

```
char* s1;
char s2;
```

Die Entscheidung für einen der beiden Stile kann durch benutzerdefinierte Zeigertypen (siehe 9.1) umgangen werden. Beispiel:

```
typedef char* CharPtr;
typedef Person* PersonPtr;

CharPtr s1, s2;
PersonPtr p;
```

7 Strukturdatentypen

Strukturen sind Aggregation von Daten i.a. unterschiedlichen Typs. Jede Datenvereinbarung wird in eine separate Zeile geschrieben; z.B.

```
struct Person
{
    char name[20];
    int alter;
    Datum geb;
};
```

8 Klassen

8.1 Aufbau und Formatierung von Klassen

Objekte sind Aggregation von Attributen (Daten)⁴ und Methoden⁵. Daten und Methoden werden in der Klassendefinition vereinbart, wobei folgende Reihenfolge einzuhalten ist:

1. **Public-Teil:**

Hier werden alle öffentlichen Methoden vereinbart. Attribute sind aufgrund des Geheimnisprinzips (information hiding) nicht öffentlich zu machen. Bei den Methoden ist folgende Reihenfolge empfehlenswert:

- Default-Konstruktor⁶ (sofern benötigt)
- Sonstige Konstruktoren (sofern benötigt)
- Destruktor (sofern benötigt)
- Operatoren, z.B. Zuweisung, Vergleich, etc. (sofern benötigt)
- Sonstige Methoden

2. **Protected-Teil** (nur bei Vererbung):

Hier werden alle Methoden und Attribute vereinbart die sowohl privat als auch in abgeleiteten Klassen zur Verfügung gestellt werden sollen.

3. **Private-Teil:**

Hier werden alle Methoden und Attribute vereinbart, die ausschließlich privat verwendet dürfen.

Die Formatierung geht aus folgendem Beispiel hervor:

⁴In C++ data members genannt.

⁵In C++ member functions genannt.

⁶Das ist der parameterlose Konstruktor. Siehe Beispiel.

```

class Person
{
public:
    Person();                // Default-Konstruktor
    Person(int nr);         // Konstruktor
    void setName(const char str[]);
    void getName(char str[]) const;
    void setGehalt(int betrag);
    int getGehalt() const;

private:
    int perNr;
    char name[20];
    int gehalt;
};
    
```

8.2 const-Methoden

Methoden, die den Objekt-Zustand unverändert lassen, sollten als const definiert werden. Beispielsweise verändern die Methoden getName und getGehalt im oberen Beispiel den Objektzustand nicht und sind daher mit const vereinbart. Der Compiler gewährleistet, daß const-Methoden nicht schreibend auf Attribute zugreifen.

8.3 Objekte mit dynamischem Anteil: Konstruktoren und Destruktor

Enthalten Objekte einen dynamischen Anteil (Zeigerattributen wird dynamisch allokiertes Speicherplatz zugewiesen), dann muß immer ein Destruktor und wenigstens ein Konstruktor definiert werden. Der (die) Konstruktor(en) allokiert(en) dynamisch Speicher, während der Destruktor ihn freigibt.

Außerdem sollte für den Fall, daß eine Klasse Typ eines Werteparameters oder eines Rückgabewertes ist, ein Kopierkonstruktor definiert werden, der das Kopieren der Zeigerattributen mit ihren dereferenzierten Bestandteilen bei der Parameterübergabe bzw. Rückgabe übernimmt. Fehlt ein Kopierkonstruktor, werden bei Zeigerattributen nur die Zeiger kopiert, was zu späteren Laufzeitfehlern durch den Destruktor führt.

Bei Objekt-Zuweisungen (mit dem Zuweisungsoperator) entsteht die gleiche Problematik. Daher muß entweder auf Objekt-Zuweisungen verzichtet oder aber der Zuweisungsoperator geeignet überladen werden.

Beispiel:

```

class Person
{
public:
    Person();                // Default-Konstruktor
    Person(int nr);
    // Kopier-Konstruktor (sofern benötigt):
    Person(const Person& p);
    ~Person();              // Destruktor
    // Ueberladener Zuweisungsoperator (sofern benötigt):
    Person& operator=(const Person& p);
    void setName(const char str[]);
    void getName(char str[]) const;
};
    
```

```

    void setGehalt(int betrag);
    int getGehalt() const;

private:
    int perNr;
    char* name;
    int gehalt;
};

```

8.4 Vererbung und virtuelle Methoden

Bei Vererbung wird in der Regel eine dynamische Bindung der Methoden (außer Konstruktoren) gewünscht. Daher sollten Methoden von Klassen, die für Vererbung vorgesehen sind, grundsätzlich als virtuell⁷ deklariert werden.

Bei Destruktoren ist dies sogar zwingend notwendig, wie das folgende Beispiel zeigt. Nur durch die Deklaration des Destruktors `~Person()` als virtuell (1), wird bei der Ausführung von `delete p` (2) erreicht, daß der erforderliche Destruktor `~Employee()` aufgerufen wird (dynamische Bindung!). Würde `virtual` fehlen, dann würde der Destruktor `~Person()` ausgeführt werden (statische Bindung!).

```

class Person
{
    // ...
    Person(char* n)
    virtual ~Person();           (1)
    virtual print();
    // ...
    char* name;
}

class Employee : Person
{
    // ...
    Employee(char* n, char* cn);
    virtual ~Employee();
    virtual print();
    // ...
    char* companyName;
}

Person* p = new Employee("Maier", "FHK");

p->print(); // Employee::print() wird aufgerufen
delete p;  // Employee::~~Employee() wird aufgerufen (2)

```

9 Sonstiges

9.1 Benutzerdefinierte Typen

Neben der Modellierung von Daten als Aggregate (Felder, Strukturdatentypen und Klassen) spielt die Realisierung von Daten als Basisdatentypen (`int`, `double`, `char`, etc.) eine wichtige

⁷ `virtual` in C++.

Rolle. Zum Beispiel liefern Prüffunktionen einen booleschen Wert 0 oder 1 zurück; Personendaten wie Alter, Personalnummer sind ganzzahlige Werte; Meßwerte wie Stromstärke, Spannung etc. werden als Gleitpunktzahlen implementiert.

Hier sollten statt Basisdatentypen mit **enum** und **typedef** aussagefähigere Typbezeichnungen eingeführt werden. Bei endlichen Datentypen mit kleinem Wertebereich ist ein Aufzählungstyp (**enum**) vorzuziehen und ansonsten die **typedef**-Variante zu wählen.

Beispiele:

```
typedef short int Alter;
typedef int PersonalNr;
Alter get(PersonalNr nr);

typedef float Spannung;
Spannung get();

enum Bool {true,false};
Bool istVorhanden(int x, int a[], int n)

enum Vergleich {lt, le, eq, ne, ge, gt};
Vergleich vergleiche(String s1, String s2);
```

Vorteil dieser Vorgehensweise sind leichter anpassbare Programme. Wird beispielsweise eine höhere Genauigkeit bei den Spannungswerten erwartet, so genügt es

```
typedef float Spannung;
```

(an genau einer Stelle im Programm!) durch

```
typedef double Spannung;
```

zu ersetzen.

9.2 Konstanten

Um eine höhere Programmflexibilität zu erreichen, sollten für konstante Werte grundsätzlich symbolische Konstanten verwendet werden. Insbesondere sollten Feldgrößen symbolisch definiert werden. Symbolische Konstanten werden durch konstante Variablen-Deklarationen realisiert. Beispiele:

```
const double pi = 3.14159;
const int maxZeilen = 100;
char* text[maxZeilen];
```

Voneinander abhängige Konstanten lassen sich durch Formeln wiedergeben; z.B.

```
const int hoehe = 10;
const int breite = 20;
const int flaeche = hoehe*breite;
```

9.3 Adressarithmetik

Aufgrund der besseren Lesbarkeit sind grundsätzlich indizierte Zugriffe auf Feldelemente statt Zugriffe über Zeiger zu verwenden. Beispielsweise ist

```
a[i][j]
```

besser lesbar als

```
*(*(a+i)+j)
```

9.4 Ein/Ausgabe

Bei Ein/Ausgabefunktionen sind möglichst die Funktionen der *stream*-Klassen zu verwenden. Die entsprechenden C-Routinen sind weder typsicher noch erlauben sie die Integration benutzerdefinierter Klassen in das Ein/Ausgabesystem.

Die wichtigsten *stream*-Klassen sind verwendbar durch:

- `<iostream.h>` Standard-Ein/Ausgabe
- `<fstream.h>` Datei-Ein/Ausgabe
- `<strstream.h>` Ein/Ausgabe mit Zeichenketten (*Strings*) als Quelle bzw. Ziel

Folgendes Beispiel zeigt die Verwendung dieser Klassen. Insbesondere kann der Datentyp `Complex` bei der Ein/Ausgabe wie ein Basisdatentyp behandelt werden, vorausgesetzt der Ein- bzw. Ausgabeoperator ist geeignet überladen worden (siehe Kasten nächste Seite).

```
#include <iostream.h>
#include <fstream.h>
#include <strstream.h>

int i;
double x;
Complex c;

// Standard-Ein/Ausgabe
cin >> i >> x >> c;
cout << "i = " << i << endl;
cout << "x = " << x << endl;
cout << "c = " << c << endl;

// Datei-Ein/Ausgabe
ifstream fin("eingabe.txt");
ofstream fout("ausgabe.txt");

while (fin >> c)
{
    // bearbeite c: ...
    fout << c << endl;
}

// Ein/Ausgabe mit Strings als Ziel bzw. Quelle
char quelle[80];
char ziel[80];
istrstream strIn(quelle,80);
ostrstream strOut(ziel,80);

cin.getline(quelle,80);
strIn >> c;
strOut << "c = " << c << '\0';
cout << ziel;
// Von String lesen
// Auf String schreiben
```

```

struct Complex
{
    double real;
    double im;
};

// Ueberladen des Ausgabeoperators:
ostream& operator<<(ostream &s, const Complex &c)
{
    return (s << c.real << "+" << c.im << "*i");
}

// Ueberladen des Eingabeoperators:
istream& operator>>(istream &s, Complex &c)
{
    return (s >> c.real >> c.im);
}
    
```

9.5 Dynamischer Speicher

Auf jeder Allokierung dynamischen Speichers mit dem **new**-Operator muß irgendwann seine Freigabe mit dem **delete**-Operator erfolgen. Beispiel:

```

double* daten;
CPerson* aPersonPtr;
// ...
daten = new double[100];
aPersonPtr = new CPerson;
// ...
delete [] daten;
delete aPersonPtr;
    
```

10 Modularisierung

Programme sind in Module aufzuteilen (Modularisierung). In einem Softwareprojekt ist eine vernünftige Modularisierung ein entscheidender Erfolgsfaktor.

Ein Modul besteht aus einer Schnittstellenspezifikation und ihrer Implementierung. In C++ wird die Schnittstelle als Header-Datei *datei.h* (oder auch *datei.hpp*) und die Implementierung als *datei.c* (oder auch *datei.cpp*) realisiert. Die Schnittstelle enthält in der Regel Typdefinitionen (insbesondere Kassendefinitionen), Funktionsprototypen, Konstanten etc.

Um Mehrfachdefinitionen zu vermeiden, sind alle Schnittstellendateien mit dem **#ifndef**-Mechanismus zu versehen (siehe Beispiel in 10.2).

Ein Modul (Schnittstellen- und Implementierungsdatei) sollten von einer der folgenden Bauart sein.

10.1 Hauptmodul

Das Hauptprogramm `main` steht in der Datei `main.c` (oder auch `mainApplikationsname.c`). Die Schnittstellendatei entfällt.

Beispiel:

```
// mainPersonal.c
#include <iostream.h>
#include <fstream.h>
#include "person.h"
#include "statistik.h"

int main()
{
// ...
}
```

10.2 Klassenmodul

Ein Klassenmodul enthält die Realisierung genau einer Klasse. Dabei enthält die Header-Datei die Klassendefinition und die Implementierungsdatei die Methodendefinitionen.

Beispiel:

```
// person.h
#ifndef PERSON_H
#define PERSON_H

class Person
{
public:
    Person();
    void getName(char str[]) const;
    void setName(const char str[]);
    // ...

private:
    char name[20];
    // ...
};

#endif
```

```
// person.c
#include "person.h"
#include <string.h>

Person::Person()
{ // ...
}

void Person::getName(char str[]) const
{
    strcpy(str,name);
}

void Person::setName(const char str[])
{
    strcpy(name,str);
}

// ...
```

10.3 Funktionsmodul

Ein Funktionsmodul besteht aus einer Menge von konzeptionell zusammengehörenden Funktionen ohne gemeinsame Daten. Dabei enthält die Header-Datei die Funktionsprototypen und die Implementierungsdatei die Funktionsdefinitionen.⁸

⁸Funktionen mit gemeinsamen Daten werden in einem sogenannten Datenmodul zusammengefasst. In C++ jedoch macht das flexiblere Klassenkonzept das Datenmodul überflüssig. Bei Programmentwicklung unter C erfolgt die Vereinbarung der Funktionsprototypen in einer Schnittstellendatei und die Definition der Funktionen und gemeinsamen Daten in einer Implementierungsdatei; die Daten werden mit static nach außen verborgen.

Beispiel:

```
// statistik.h

#ifndef STATISTIK_H
#define STATISTIK_H

double mittelwert(double x[], int n);
double varianz(double x[], int n);
void regressionsGerade(/*...*/);

// ...

#endif
```

```
// statistik.c

#include statistik.h

double mittelwert(double x[], int n)
{
    double s = 0;

    for (int i = 0; i < n; i++)
        s += x[i];
    return s/n;
}

double varianz(double x[], int n)
{ /* ... */ }

void regressionGerade(/*...*/)
{ /* ... */ }

// ...
```

10.4 Typdefinitionen

Globale benutzerdefinierte Typen, die keine Klassen sind, werden in eine odere mehrere Header-Dateien definiert.

Beispiel:

```
// type.h

#ifndef TYPE_H
#define TYPE_H

typedef float Spannung;
enum Bool {true,false};
enum Vergleich {lt, le, eq, ne, ge, gt};

#endif
```

10.5 Globale Konstanten

Globale Konstanten werden in eine odere mehrere Header-Dateien definiert.

Beispiel:

```
// const.h

#ifndef CONST_H
#define CONST_H

const double pi = 3.14149
const int maxPesron = 1000;

// ...

#endif
```

10.6 Globale Variablen

Globale Variable verletzen das Geheimnisprinzip (information hiding) und sollten möglichst vermieden werden. Sollten globale Variablen dennoch notwendig sein, dann erfolgt die Spezifikation der Daten (extern-Deklaration) in einer Schnittstellendatei und die Definition der Daten in einer Implementierungsdatei. Jedes Modul, das Zugriff auf die globalen Daten benötigt, fügt die Schnittstellendatei ein.

Beispiel:

```
// global.h

#ifndef GLOBAL_H
#define GLOBAL_H

extern int daten[10];
extern int n;

// ...

#endif
```

```
// global.c

int daten[10] = {0};
int n = 0;

// ...
```

11 Dokumentation

Integraler Bestandteil jedes Programms ist eine geeignete Dokumentation. Die hinreichende Dokumentierung von Datenvereinbarungen und komplizierteren Kontrollstrukturen versteht sich von selbst. Für die Dokumentierung von Funktionen (und Methoden) und der Schnittstellen- und Implementierungsdateien ist ein einheitlicher Stil zu wählen.

a) Funktionen

Bei Funktionen ist das Ein/Ausgabeverhalten zu beschreiben. Dazu gehört auch die Klassifizierung der Parameter in Ein-, Ausgabe- und Ein/Ausgabeparameter. Bei Eingabeparameter sind erforderliche Vorbedingungen und bei Ausgabeparametern die von den Funktionen garantierten Nachbedingungen festzulegen. Eventuell können auch Angaben zur Laufzeit- und Speicherplatzkomplexität hinzugefügt werden.

Darüberhinaus sind Seiteneffekte (Änderung globaler Variablen) unbedingt anzugeben. Bei Nicht-const-Methoden ist die Änderung des Objektzustands zu beschreiben.

Beispiel: siehe unten.

a) Schnittstellen- und Implementierungsdateien

Schnittstellen- und Implementierungsdateien sollten einen einheitlichen Vorspann besitzen, der folgendermaßen aufgebaut ist:⁹

⁹Ergänzend kann noch eine Versionsnummer und ein Statusvermerk (in Bearbeitung, vorgelegt, akzeptiert) hinzukommen.

```

// Datei-Name
//
// Aufgabenbeschreibung
//
// Autor: Name
// Erstellt am: Datum
// Geändert am: Datum
// Weitere Änderungstermine
    
```

Beispiel:

```

// stack.h
//
// Enthält Klassendefinition Stack für Keller mit
// ganzzahligen Elementen. Ein Keller kann im Prinzip
// beliebig viele Elemente aufnehmen.
//
// Autor: O. Bittel
// Erstellt am: 6.3.1997

#ifndef STACK_H
#define STACK_H

class Stack
{
public:
    Stack();
    // Konstruktor; legt einen leeren Keller an.
    ~Stack()
    // Destruktor
    int push(int elem);
    // kellert Element in Keller ein
    // int elem:    E; einzukellerndes Element
    // Return:      1, falls einkellern OK
    //              0, sonst (Speicherplatzprobleme)
    int pop(int& elem);
    // kellert Element aus Keller aus
    // int& elem:   A; ausgekellertes Element
    // Return:      1, falls Auskellern OK
    //              0, sonst (Keller ist leer);
    // ...

private:
    struct Node
    {
        Node* next;
        int element;
    };
    Node* topPtr; // Keller als verkettete Liste; topPtr
                // zeigt auf oberstes Kellerelement
    int anz;      // Anzahl Kellerelemente
    // ...
};

#endif
    
```

```
// stack.c
//
// Enthält Implementierung der Methoden der Klasse Stack.
//
// Autor: O. Bittel
// Erstellt am: 6.3.1997
// Geändert am: 7.3.1997

#include "stack.h"

Stack::Stack()
{ /* ... */ }

Stack::~Stack()
{ /* ... */ }
// ...
```

Danksagung

Ich möchte mich bei meinen Kollegen Dr. Hager und Dr. Schmid ganz herzlich für die vielen nützlichen Hinweise und Verbesserungsvorschläge bedanken.

12 Literaturverzeichnis

- Balzert, H. (1996). Lehrbuch der Software-Technik, Spektrum Akademischer Verlag.
- Kernighan, B.W., Ritchie, D.M. (1978). The C Programming Language, Prentice Hall.
- Stroustrup, B. (1991). The C++ Programming Language, 2nd ed., Addison-Wesley.