Programmierrichtlinien für Standard-C++

Prof. Dr. Oliver Bittel
Prof. Dr. Heiko von Drachenfels

Fachbereich Informatik
Fachhochschule Konstanz

Version 3.0
Datum: 1.3.2004

Hinweis zur Version 3.0:

Die inhaltlichen Änderungen gegenüber Version 2.1 sind der Tabelle zu entnehmen.
In erster Linie wurde der ISO-Standard eingearbeitet.

alte Nummer	**neue Nummer**	**Änderung**
2.1	2.1	Gültigkeitsbereich von Bezeichnern
2.2	2.2	Grundtyp bool (ISO) statt enum Bool.
	2.6	*Neu!* Namensräume
2.6	2.7	
4	4	Bemerkung zu continue gestrichen
4.7	4.7	for-Schleife nicht nur als Zählschleife
	4.9	*Neu!* Ausnahmebehandlung
5.1	5.1	keine Aufteilung in Deklarations - und Anweisungsteil
5.2	5.2	Ausgabeparameter auch als Zeiger
6	7.1	
7	7.2	
8.1	6.1	keine Attribute im Protected-Teil
8.3	6.3	Überschrift Konstruktoren und Destruktoren
9	7	
9.1	7.3	in Überschrift Typnamen statt Typen, enum Bool gestrichen (ISO)
9.2	7.4	
9.3	7.5	
9.4	7.6	Standardheader nach ISO ohne .h, Namen daraus mit std::
9.5	7.7	
10	8	Standardheader nach ISO ohne .h, Namen daraus mit std::, Implementierungsdateien immer .cpp geschachtelte Includes
	8.2	*Neu!* Schnittstellenmodul
10.2	8.3	Beispiel mit geschachteltem Include
10.6	8.5	Datenmodul (vormals Globale Daten)
10.4, 10.5	8.6	Konfigurationsmodul (vormals Typdefinitionen und Globale Konstanten) enum Bool gestrichen (ISO)
11	9	Bei Änderungen auch Name und Grund
	10	*Neu!* Hinweise zum ISO-Standard

1 EINLEITUNG 5

2 BEZEICHNER 6

2.1 ALLGEMEINE REGELN 6
2.2 VARIABLEN 6
2.3 KONSTANTEN 7
2.4 FUNKTIONEN 7
2.5 BENUTZERDEFINIERTE DATENTYPEN UND KLASSEN 7
2.6 NAMENSRÄUME 8
2.7 PRÄPROZESSORBEZEICHNER 8

3 FORMATIERUNG 9

3.1 LEERZEICHEN 9
3.2 LEERZEILEN 9
3.3 EINRÜCKUNG 10

4 KONTROLLSTRUKTUREN 11

4.1 ZUSAMMENGESETZTE ANWEISUNGEN UND BLÖCKE 11
4.2 IF- UND IF-ELSE-ANWEISUNGEN 11
4.3 ELSE-IF-ANWEISUNGEN 12
4.4 SWITCH-ANWEISUNGEN 12
4.5 WHILE-SCHLEIFEN 13
4.6 DO-WHILE-SCHLEIFEN 13
4.7 FOR-SCHLEIFEN 13
4.8 SCHLEIFEN MIT MEHRFACHEM AUSSTIEG (BREAK-ANWEISUNG) 14
4.9 AUSNAHMEBEHANDLUNG 14

5 FUNKTIONEN 15

5.1 FORMATIERUNG 15
5.2 PARAMETER 15
5.3 FUNKTIONEN MIT MEHRFACHEM AUSSTIEG 17
5.4 INLINE-FUNKTIONEN 18

6 KLASSEN 19

6.1 AUFBAU UND FORMATIERUNG VON KLASSEN 19
6.2 CONST-METHODEN 19
6.3 KONSTRUKTOREN UND DESTRUKTOR 20
6.4 VERERBUNG UND VIRTUELLE METHODEN 20

7 SONSTIGES 22

7.1 ZEIGER- UND REFERENZTYPEN 22
7.2 STRUKTURDATENTYPEN 22
7.3 BENUTZERDEFINIERTE TYPNAMEN 23
7.4 KONSTANTEN 23
7.5 ADRESSARITHMETIK 23
7.6 EIN/AUSGABE 24
7.7 DYNAMISCHER SPEICHER 25

8 MODULARISIERUNG 26

8.1 HAUPTMODUL 26
8.2 SCHNITTSTELLENMODUL 27
8.3 KLASSENMODUL 27
8.4 FUNKTIONSMODUL 28
8.5 DATENMODUL 28
8.6 DEFINITIONSMODUL 29

9 DOKUMENTATION 30

10 HINWEISE ZUM ISO-STANDARD 33

10.1 GNU-COMPILER 33
10.2 MICROSOFT VISUAL C++ .NET 34
10.3 MICROSOFT VISUAL C++ 6.0 35

11 LITERATURVERZEICHNIS 37

1 Einleitung

Programmierrichtlinien sollen helfen, einheitliche und daher besser verständliche Programme zu erstellen. Sie bestehen aus einer Sammlung von Regeln, die über die Sprachsyntax hinausgehen. Sie werden daher auch nicht vom Compiler erzwungen, sondern müßen durch Programmierdiziplin 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:

- leichter lesbare Pragramme (hiermit wird insbesondere die Einarbeitung in fremde Programme erleichtert),
- weniger fehlerhafte Programme,
- leichtere Anpass- und Erweiterbarkeit.

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 möglichweise 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 (sind für interne Zwecke von C++-Implementierungen reserviert).

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.

c) Der Gültigkeitsbereich (Scope) von Bezeichnern sollte so klein wie möglich gewählt werden.

2.2 Variablen

a) Variablen beginnen grundsätzlich mit einem Kleinbuchstaben.

b) Für Laufvariablen in for-Schleifen sollten Kleinbuchstaben wie i, j, k verwendet werden.

c) 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.

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

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

e) Bei Variablen vom Typ **`bool`** sollten Adjektive verwendet werden, z.B.

```
bool gefunden = false;

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 Funktionen[1]

a) Funktionen beginnen grundsätzlich mit einem Kleinbuchstaben.

b) Für Funktionen sollten möglichst Verben verwendet werden; z.B.

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

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);
```

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};
```

[1] Gilt auch für Methoden.

2.6 Namensräume

Für die globalen Bezeichner einer Funktions- oder Klassenbibliothek sollte eine Namensraum definiert werden, um Kollisionen mit den Bezeichnern anderer Bibliotheken zu vermeiden. Bezeichner für Namensräume sollten nur aus Kleinbuchstaben bestehen.

2.7 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

a) Umfangreiche (mehrzeilige) Definitionen (von Funktionen, Strukturdatentypen, Klassen, etc.) sind durch Leerzeilen zu trennen.

b) In einer Funktion sind Deklarationsteil und Anweisungsteil durch eine Leerzeile zu trennen.

c) 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.

4.1 Zusammengesetzte Anweisungen und Blöcke

Zusammengesetzte Anweisungen und Blöcke[2] werden mit geschweiften Klammern eingerahmt, 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)
{                  // Klammerung empfehlenswert
   if (x >= z)
      max = x;
   else
      max = z;
}
else
{                  // Klammerung empfehlenswert
   if (y >= z)
      max = y;
   else
      max = z;
}
```

[2]Ein 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 sind der While- und Do-While-Schleife immer dann vorzuziehen, wenn die Schleife eine Laufvariable benötigt, also z.B. beim Ablaufen von Aggregaten wie Feldern oder Listen. Die Laufvariable sollte im Schleifenkopf definiert werden, da sie nach der Schleife in der Regel nicht mehr gebraucht wird. Die Laufvariable darf im Schleifenrumpf nicht verändert werden. Es gilt die gleiche Einrückungsregel wie bei der while-Schleife.

Beispiele:

```
for (int i = 0; i < n; ++i)
   feld[i] = 0;

for (LinkedList *p = listPtr; p != 0; p = p->nextPtr)
   p->element = 0;
```

4.8 Schleifen mit mehrfachem Ausstieg (Break-Anweisung)

Nach der Methode der Strukturierten Programmierung sollten grundsätzlich Kontrollstrukturen mit genau einem Eingang und einem Ausgang vewendet 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
}
```

4.9 Ausnahmebehandlung

Fehlersituationen sollten durch das Werfen einer Ausnahme gemeldet werden, nicht durch Weitergabe einer Fehlernummer oder das Setzen von Statuscodes.

Ausnahmen sind als Klassen zu deklarieren. Ähnliche Ausnahmen sollten eine gemeinsame Oberklasse haben.

Es dürfen nur Objekte als Ausnahmen geworfen werden, keine Zeiger auf Objekte. Beim Fangen von Ausnahmen ist immer ein Referenztyp anzugeben, damit der tatsächliche Typ der Ausnahme nicht verloren geht.

Beispiel:

```
try
{
   // ...
   druckeKunde(int nr);
   // ...
}
catch (UnbekannteKundenNummer &x)
{
   std::cerr << x.text() << '\n';
}
catch (DruckerFehler &x)
{
   // ...
}
```

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;

   for (int i = 0; i < n; i++)
      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 Kassendefinition 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, Ausgabeparameter (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 zwingend):**

  Beispiele:

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


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 Referenz- oder 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)
{
   return x > y ? x : y;
}
```

6 Klassen

6.1 Aufbau und Formatierung von Klassen

Objekte sind Aggregierung von Attributen (Daten)[3] und Methoden[4]. 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-Konstruktor[5] (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 und auch dann nur in Ausnahmefällen):
   Hier werden alle Methoden vereinbart die sowohl privat als auch in abgeleiteten Klassen verwendet werden dürfen. Der Protected-Teil ist kaum stärker geschützt als der Public-Teil, denn jede abgeleitete Klasse hat Zugriff. Deshalb sollten auch hier keine Attribute vereinbart werden.

3. **Private-Teil:**
   Hier werden alle Methoden und Attribute vereinbart, die ausschließlich privat verwendet dürfen.

Die Formatierung geht aus folgendem Beispiel hervor:

```
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;
};
```

6.2 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

[3] In C++ data members genannt.
[4] In C++ member functions genannt.
[5] Das ist der parameterlose Konstruktor. Siehe Beispiel.

Objektzustand nicht und sind daher mit const vereinbart. Der Compiler gewährleistet, daß const-Methoden nicht schreibend auf Attribute zugreifen.

6.3 Konstruktoren und Destruktor

Enthalten Objekte einen dynamischen Anteil (Zeigerattributen wird dynamisch allokierter 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-Konstrukor (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;
};
```

6.4 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 virtuell[6] 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!).

[6] **`virtual`** in C++.

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

class Employee : public 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)
```

7 Sonstiges

7.1 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 2000]:
  * (bzw. &) wird zum Typ geschrieben, z.B.:

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

Da die Stroustrup-Schreibweise nicht der syntaktischen Konvention von C++ entspricht, darf höchstens eine Zeigervariable je Deklaration definiert wird. 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;
```

7.2 Strukturdatentypen

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

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

7.3 Benutzerdefinierte Typnamen

Neben der Modellierung von Daten als Aggregate (Felder, Strukturdatentypen und Klassen) spielt die Realisierung von Daten als Basisdatentypen (int, double, char, etc.) eine wichtige Rolle. Zum Beispiel sind Personendaten wie Alter, Personalnummer 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 Vergleich {lt, le, eq, ne, ge, gt};
Vergleich vergleiche(String s1, String 2);
```

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.

7.4 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;
```

7.5 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)
```

7.6 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>** Standard-Ein/Ausgabe
- **<fstream>** Datei-Ein/Ausgabe
- **<strstream>** 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>
#include <fstream>
#include <strstream>

int i;
double x;
Complex c;

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

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

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

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

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

```
struct Complex
{
   double real;
   double im;
};
           // Ueberladen des Ausgabeoperators:
std::ostream& operator<<(std::ostream &s, const Complex &c)
{
    return (s << c.real << "+" << c.im << "*i");
}
           // Ueberladen des Eingabeoperators:
std::istream& operator>>(std::istream &s, Complex &c)
{
    return (s >> c.real >> c.im);
}
```

7.7 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;
```

8 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.cpp* realisiert. Die Schnittstelle enthält in der Regel Typdefinitionen (insbesondere Kassendefinitionen), Funktionsprototypen, Konstanten etc.

Eine Header-Datei wird mittels **`#include`** zum einen in die zugehörige Implementierungs-Datei kopiert und zum anderen in alle Dateien, die Definitionen aus der Header-Datei verwenden. Insbesondere müssen Header-Dateien ihrerseits **`#include`**-Anweisungen enthalten, wenn sie in anderen Header-Dateien definierte Namen (z.B. Typnamen) enthalten. Um bei den daraus resultierenden geschachtelten Includes Mehrfachkopien und damit Mehrfachdefinitionen zu verhindern, sind alle Schnittstellendateien mit dem **`#ifndef/#define`**-Mechanismus zu versehen (siehe Beispiele in 8.2, 8.3, 8.4 und 8.5).

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

8.1 Hauptmodul

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

Beispiel:

```
// mainPersonal.cpp

#include <iostream>
#include <fstream>
#include "Person.h"
#include "statistik.h"

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

8.2 Schnittstellenmodul

Ein Schnittstellenmodul enthält die Realisierung genau einer Schnittstellenklasse (vergleiche interface in Java). Es besteht nur aus einer Header-Datei. Die Implementierungsdatei entfällt.

Beispiel:

```
// Printable.h

#ifndef PRINTABLE_H
#define PRINTABLE_H

class Printable
{
public:
   virtual ~Printable() {}
   virtual void print() = 0;
   // ...
};

#endif
```

8.3 Klassenmodul

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

Beispiel:

```
// Person.h

#ifndef PERSON_H
#define PERSON_H

#include "Printable"

class Person
: public virtual Printable
{
public:
   Person();
   void getName(char str[]) const;
   void setName(const char str[]);
   void print ();

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

#endif
```

```
// Person.cpp

#include "person.h"
#include <cstring>
#include <iostream>

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

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

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

void Person::print()
{
   std::cout << name;
}
```

8.4 Funktionsmodul

Ein Funktionsmodul besteht aus einer Menge von konzeptionell zusammengehörenden Funktionen ohne gemeinsame Daten. Dabei enthält die Header-Datei die Funktionsprotypen und die Implementierungdatei die Funktionsdefinitionen.[7]

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.cpp

#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(/*...*/)
{ /* ... */ }

// ...
```

8.5 Datenmodul

Globale Variablen verletzen das Lokalitäts- und 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:

```
// daten.h

#ifndef DATEN_H
#define DATEN_H

extern int daten[10];
// ...

#endif
```

```
// daten.cpp

int daten[10] = {0};
// ...
```

[7] Funktionen mit gemeinsamen Daten werden in C++ als Klassenmodul realisiert. In C, wo es kein Klassenkonzept gibt, wird ein sogenanntes Datenabstraktionsmodul realisiert: die Vereinbarung der Funktionsprototypen erfolgt in einer Schnittstellendatei und die Definition der Funktionen und gemeinsamen Daten in einer Implementierungsdatei; die Daten werden mit static nach außen verborgen.

8.6 Definitionsmodul

Modulübergreifende Typen und Konstanten können in einer oder mehreren Header-Dateien deklariert werden.

Beispiel:

```
// def.h

#ifndef DEF_H
#define DEF_H

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

const double pi = 3.14149
const int maxPerson = 1000;

#endif
```

9 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.

b) **Schnittstellen- und Implementierungsdateien**

Schnittstellen- und Implementierungsdateien sollten einen einheitlichen Vorspann besitzen, der folgendermaßen aufgebaut ist:[8]

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

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

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.cpp
//
// Enthält Implementierung der Methoden der Klasse Stack.
//
// Autor: O. Bittel
// Erstellt am: 6.3.1997
// Geändert am: 8.12.2003 Drachenfels Programmierrichtlinien 3.0

#include "stack.h"

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

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

// ...
```

10 Hinweise zum ISO-Standard

Im Jahr 1998 wurde der internationaler Standard für C++ verabschiedet (ISO/IEC 14882). Der Standard ist in den gängigen Compilern inzwischen weitgehend umgesetzt. Bei ISO-konformer Programmierung kann deshalb eine recht gute Portabilität erreicht werden. Zu den wichtigsten Neuerungen des Standards zählen:

- Es gibt einen Grundtyp **`bool`** mit Werten **`true`** und **`false`** (alles Schlüsselwörter).
- Es gibt Namensräume (**`namespace`**) zur Vermeidung von Namenskollisionen bei Bezeichnern mit globaler Sichtbarkeit.
- Die Header-Dateien der Standardbibliothek werden ohne Endung **`.h`** geschrieben und bei den aus der Sprache C übernommenen Header-Dateien wird dem Namen außerdem ein **`c`** vorangestellt, z.B.:

  `<iostream>` statt bisher **`<iostream.h>`**
  `<ctime>` statt bisher **`<time.h>`**

  Die alte Schreibweise gibt es weiter, aber nur damit alte Software übersetzbar bleibt. Bei Verwendung der neuen Schreibweise liegen alle Bezeichner mit globaler Sichtbarkeit aus der Standardbibliothek im Namensraum **`std`**, z.B.:

  `std::cout` statt bisher **`cout`**
  `std::time_t` statt bisher **`time_t`**

- Der Operator **`new`** wirft eine Ausnahme **`std::bad_alloc`**, wenn kein Speicher mehr verfügbar ist. Die Ausnahme ist in der Header-Datei **`<new>`** definiert.
- Es gibt neue Operatoren für die Typumwandlung:

  `dynamic_cast<T>(x)`
  `static_cast<T>(x)` statt bisher **`(T) x`**
  `const_cast<T>(x)`
  `reinterpret_cast<T>(x)`

- Im Kopf einer **`for`**-Schleife definierte Laufvariablen haben den Schleifenrumpf als Gültigkeitsbereich, sind also hinter der Schleife nicht mehr zugreifbar.

10.1 Gnu-Compiler

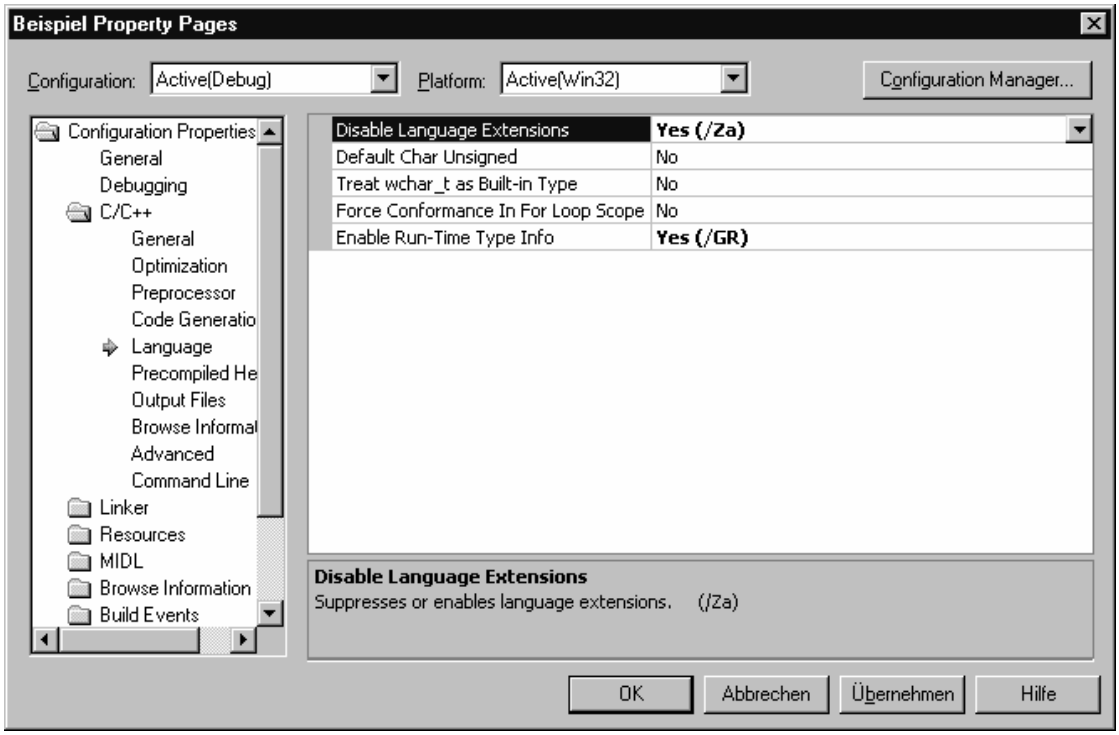
Die Gnu-Entwickler hatten erhebliche Mühe, den Standard umzusetzen. Mit Version 3.3 ist endlich ein recht guter Stand erreicht. Wer portablen Code schreiben will, muss aber unbedingt folgende Optionen angeben:

```
g++ -ansi -pedantic
```

Ohne die beiden Optionen werden Gnu-spezifische Spracherweiterungen zugelassen. Auch mit diesen Optionen wird leider für manche Syntaxfehler entgegen dem Standard nur eine Warnung ausgegeben.

10.2 Microsoft Visual C++ .NET

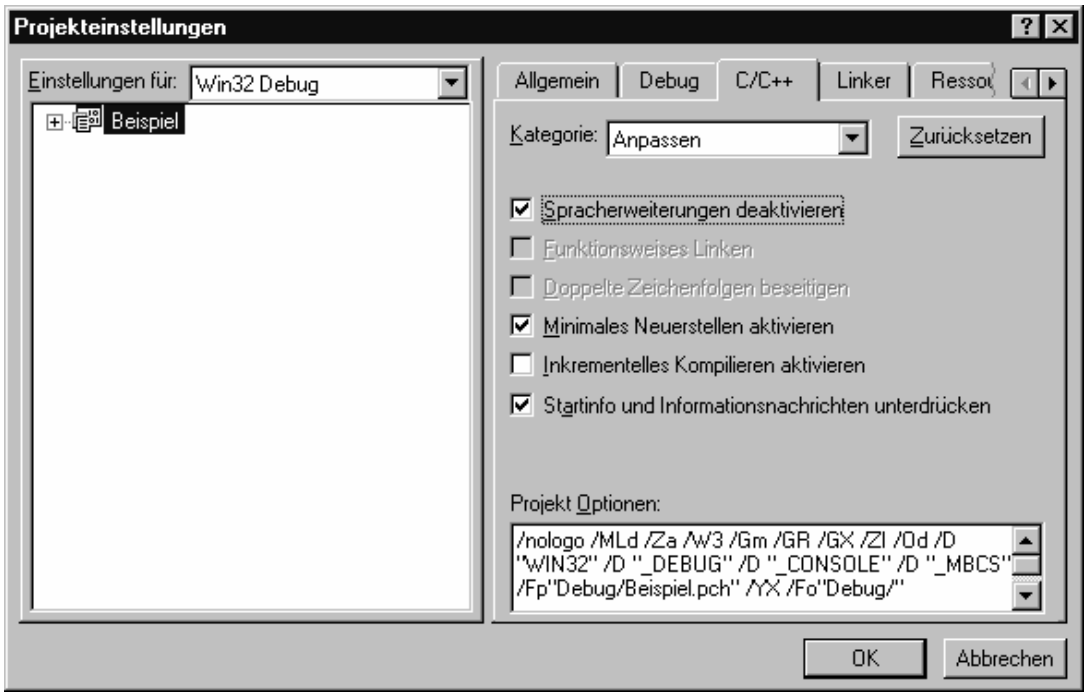
- Auch bei Microsofts aktuellem Compiler müssen spezielle Optionen eingestellt werden, wenn der ISO-Sprachumfang genutzt werden soll (/Za und /GR):

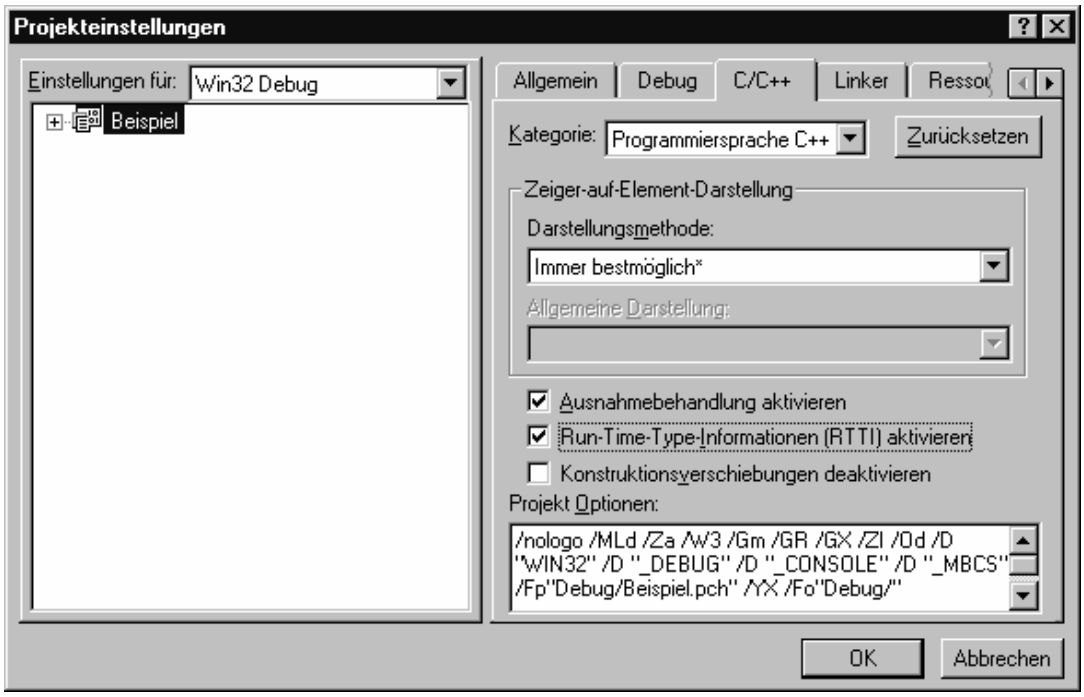


- Mit den obigen Einstellungen bekommt man oft Fehlermeldungen der Art
  `unexpected end of file`
  Sie lassen sich beseitigen, wenn man in den Header-Dateien nach dem **`#endif`** noch ein Return eingibt.
- Mit Vorsicht ist außerdem der Code-Wizard zum Einfügen von Klassen zu genießen. Er erzeugt keinen ISO-konformen Code. Der Code wird aber trotz obiger Einstellungen übersetzt (**`#pragma once`** statt **`ifndef/#define`**, leere Parameterlisten mit **`void`**, Nichtbeachtung von Gross- und Kleinschreibung bei Dateinamen in **`#include`**-Anweisungen).
- Die Header **`<iostream>`** (ISO) und **`<iostream.h>`** (veraltet) dürfen innerhalb eines Programms nicht vermischt benutzt werden.

10.3 Microsoft Visual C++ 6.0

- Die Vorgängerversion des zuvor beschiebenen Compilers war noch deutlich weiter vom ISO-Standard weg. Auch hier müssen Einstellungen vorgenommen werden (/Za und /GR):





- Bei for-Schleifen sind im Kopf definierte Laufvariablen entgegen dem ISO-Standard hinter der Schleife noch gültig. Soll dieser Compiler noch verwendet werden, schließt man **`for`**-Schleifen am besten in geschweifte Klammern ein.

- Die Namen aus der C-Bibliothek liegen trotz neuer Header ohne **.h** nicht im Namensraum **std**. Will man solche Namen nutzen, sollte man deshalb immer eine **using**-Direktive angeben:

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

time_t t = time(0); // std::time_t geht bei MSVC++ 6.0 nicht
```

- Die Header **<iostream>** (ISO) und **<iostream.h>** (veraltet) dürfen innerhalb eines Programms nicht vermischt benutzt werden.

11 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. (2000). Die C++-Programmiersprache, 4. aktualisierte Auflage, Addison-Wesley, (Deutsche Übersetzung der Special Edition).