Software-Technik: Vom Programmierer zur erfolgreichen …

1. Von der Idee zur Software
2. Funktionen und Datenstrukturen
3. Organisation des Quellcodes
4. Werte- und Referenzsemantik
5. Entwurf von Klassen
6. Fehlersuche
7. Software-Entwicklung im Team
8. Abstrakte Datentypen: Einheit von Daten und Funktionalität
9. Vielgestaltigkeit (Polymorphie)
10. Entwurfsprinzipien für Software

Lehrbuch: 4.1
Kompendium, 3. Auflage: 8.10, 9.1
Kompendium, 4. Auflage: 8.1 ... 8.17

Anhang A: Die Familie der C-Sprachen

Anhang B: Grundlagen der C++ und der Java-Programmierung

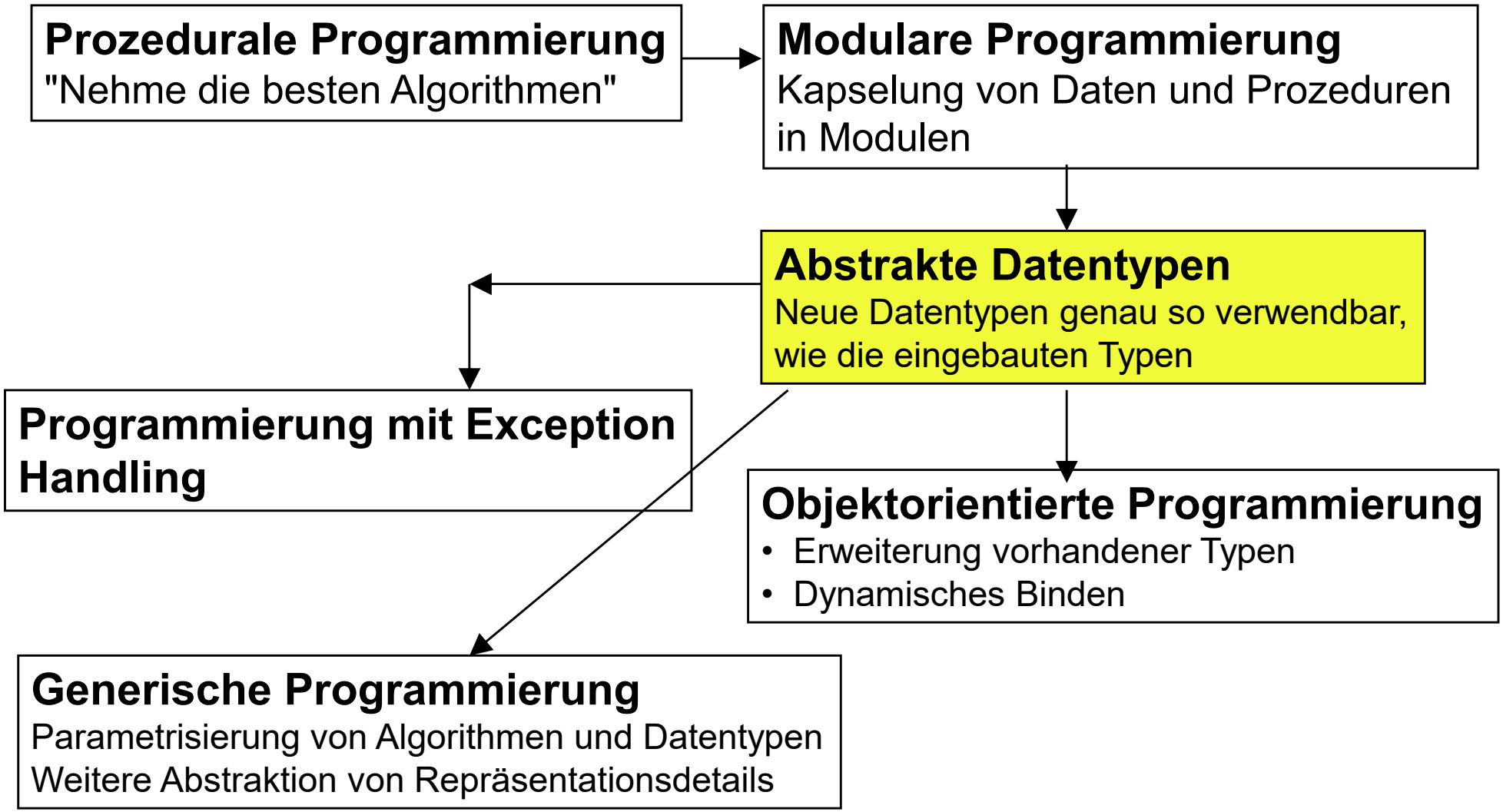
Software-Technik: Vom Programmierer zur erfolgreichen …

8 Abstrakte Datentypen: Einheit von Daten und Funktionalität

8.1 Die Bedeutung von Schnittstellen
8.2 Klassen als abstrakte Datentypen
8.3 Generische Programmierung, 2. Teil
8.4 Ausnahmebehandlung, 2. Teil
8.5 Zusammenfassung

Folien mit gelben Punkten ● am oberen rechten Rand sind weniger wichtig für das Verständnis der nachfolgenden Kapitel.

Programmierparadigmen



Wie arbeite ich mit einfachen Typen?

```
bool test (){
   int i1;
   i1 = 44 ;     // oder gleich int i1 = 44;

   if (j1  == i1) ….

   int k = j1 + i1;
```

So soll es auch mit den selbstdefinierten Typen möglich sein.

```
bool test (){
   int i1;
   i1 = 44 ;     // oder gleich int i1 = 44;

   if (j1  == i1) ….

   int k = j1 + i1;
```

```
bool test (){
   Vektor v1, v2;
   v1 = 44 ;     // oder Vektor v1(44);

   if (v1  == v2) ….

   Vektor sum = v1 + v2;
```

Konstruktoren und Destruktoren

Zugriffsrechte

```
class XYZ {
    . . .          // per Voreinst. private, d.h. kein Zugriff von außen
  public:
    . . .          // Zugriff für alle (Benutzer und Erben)
  protected:
    . . .          // Zugriff nur für Erben
  private:
    . . .          // kein Zugriff von außen
};
```

Im Gegensatz dazu ist die Voreinstellung bei der Struktur (struct) public, das ist übrigens der einzige Unterschied in C++ zwischen einer Klasse und einer Struktur!

Erzeugung einer Instanz (mit Klassen)

```
class Vektor{
public:
Vektor(int dim);
~Vektor();
void Init(int wert);
…
private:
   int* daten;
   int dimension;
};
```

```
Vektor v1(17);
Vektor v2(11);
```

```
Vektor::Vektor(int dim) {
   this->dimension = dim;
   this->daten = new int[dim];
}

Vektor::~Vektor(){
   delete[]   daten;
   daten = nullptr;
}
```

this-> bzw. this darf auch jeweils weggelassen werden.

Zerstörung einer Instanz (Destruktor)

Jede Variable, die in C++ über den new-Operator im Heap angelegt wurde, muss wieder freigegeben werden. Diesen Vorgang übernimmt standardmäßig der **Destruktor** einer Klasse.

Der Destruktor wird automatisch vom Compiler aufgerufen, wenn eine Klasseninstanz auf dem Stack angelegt wurde (was nur in C++ möglich ist).

Ein im Heap angelegtes Objekt obj muss in C++ durch den expliziten Destruktor-Aufruf delete obj; entfernt werden (Arrays: delete[] obj).

Eine wesentliche Aufgabe von Destruktoren ist es, nicht nur den Speicher, den das Objekt selbst belegt, freizugeben, sondern möglicherweise auch den Speicher von referenzierten Objekten.

```
Vektor::~Vektor(){
   delete[]   daten;
   daten = nullptr;
}
```

Aufgabe: Vektor als Klasse

Ausgehend von Ihrer Implementierung von Resize von dem Modul Vektor, implementieren Sie das Modul nun als Klasse / ADT.

```
/** Es werden zwei Vektoren erzeugt und mit
den Werten 81 und -10 belegt. Nach Aufruf
von Add sollte der ergebnis-Vektor den
Wert 71 enthalten.
*/
bool testPlus() {
  Vektor v1(4);
  const int wert1=81;
  v1.Init(wert1);

  Vektor v2(4);
  const int wert2= -10;
  v2.Init(wert2);

  Vektor result(4);
  v1.Plus(v2, result);

  for (int i=0; i<result.GetDim(); ++i)   {
    if (result.GetDatum(i) != (wert1+wert2)) {
      return false;
    }
  }
  return true;
}
```

Aufgabe: Matrix als Klasse

Ausgehend von Ihrer Implementierung von Resize von dem Modul Matrix, implementieren Sie das Modul nun als Klasse / ADT.

```
bool test1()
   {
   const int ze=3;
   const int sp=4;
   Matrix m1(ze,sp);

   for (int i = 0; i < ze; ++i)
      for (int j=0; j < sp; ++j
         m1.Setze(i,j, i*sp + j
         }
      } // for i
```

```
for (int i = 0; i < ze; ++i) {
   for (int j=0; j < sp; ++j) {
      if (m1.Lese(i,j) !=  (i*sp + j*2)) {
         return false;
         }
      }
   } // for i

return true;
}// Matrix wird auf dem Stack automatisch zerstoert
```

Weitere Details zu Konstruktoren und Destruktoren

Konstruktoren und Destruktoren

- Der **Standardkonstruktor** für eine Klasse X hat die Form **X();** und erzeugt ein Standardobjekt. Er verfügt über keine Parameter, er kann also für die Initialisierung nur voreingestellte Standardwerte benutzen oder z.B. auch Werte einlesen.

- Der **Kopierkonstruktor** (copy-constructor) für eine Klasse X hat die Form **X(const X& x);**, er kopiert die Werte eines anderen Objektes, das ihm als Parameter übergeben wird (siehe später).

- Der **Umwandlungskonstruktor** erhält die Anfangswerte in anderer Form, z.B. als String, und wandelt sie in die interne Darstellung um.

- Sonstige Konstruktoren.

- Der **Destruktor** hat die Form **~X();**, er löst ein bestehendes Objekt auf.

Konstruktoren Anwendung

```
class date {
      int t; int m; int j;
   public:
      // 4 Konstrukoren :
      date();                              // Standardkonstruktor
      date(int tt, int mm, int jj);
      date(const char *s);                 // Umwandlungskonstruktor
      date(const date& d);                 // Kopierkonstruktor

      // 1 Destruktor
      ~date();

      // 2 "normale" Elementfunktionen:
      void Set(int tt, int mm, int jj);
      void Print();
};
```

Konstruktoren, Anwendung (2)

```
date::date() { t=1; m=1; j=99; };

date::date(int tt, int mm, int jj) { t=tt; m=mm; j=jj; }

date::date(const char *s) {
    t = (s[0]-'0')*10 + s[1]-'0';
    m = (s[3]-'0')*10 + s[4]-'0';
    j = (s[6]-'0')*1000 + (s[7]-'0')*100 + (s[8]-'0')*10 + s[9]-'0';
}

date::date(const date& d) { t=d.t; m=d.m; j=d.j; }

date::~date() {};
```

Konstruktoren, Benutzung

```
# include "date1.h"
# include <iostream>
using namespace std;

int main() {
   date d1, d2(23, 8, 98), d3("16.12.2002"), d4;

   d4 = date("25.07.1977");
   date d5(d4);                    // Copy-Konstruktor-Aufruf
   . . .
}
```

Gefahr von Umwandlungs-Konstruktoren

```
void func(date d);

int main() {
date d1(17,4,1988);
func("17.04.1944");  // entspricht func(date("17.04.1944"));

char name[]="Hartmut Helmke";
func(name); // entspricht func(date(name));   ??????????????????
```

Verhinderung der automatischen Umwandlung durch Schlüsselwort explicit

```
class date {
   ...
   date(int tt, int mm, int jj);
    explicit date(char *s);                    // Umwandlungskonstruktor
   ...
   };
```

Die Bedeutung von Umwandlungskonstruktoren

Der Umwandlungskonstruktor der zuvor definierten Klasse date wird z.B. in folgenden Zusammenhängen verwendet:

```
date d1 ("16.12.2002");    // In der Deklaration: Beide
date d2 = "16.12.2002";    // Formen sind bedeutungsgleich
. . .
d1 = date("25.07.2003");  // In der Zuweisung: Expliziter und
d2 = "25.07.2003";         // impliziter Aufruf
```

Der implizite Aufruf kann wie folgt unterbunden werden:

```
class date {
    int t, m, d;
  public:
    . . .
    explicit date(char* s);
    . . .
};
```

Implizite Erzeugung von Konstruktoren

Wenn in einer Klasse kein Konstruktor explizit definiert ist, dann generiert das System automatisch einen Default-Standardkonstruktor und einen Default-Kopierkonstruktor. Der Default-Standardkonstruktor generiert keinen Speicherplatz und führt keine Initialisierungen durch. Der Default-Kopierkonstruktor erzeugt komponentenweise eine Kopie der Klasseninstanz, d.h. ein Objekt, das Zeiger beinhaltet, wird flach kopiert: es werden die Zeiger, aber nicht die damit verbundenen Speicherbereicheim Heap kopiert! Also:

- Die implizite Erzeugung eines Default-Standardkonstruktors und eines Default-Kopierkonstruktors kann in der Regel nur dann korrekt erfolgen, wenn die Klasse keine dynamisch verwalteten Daten (d.h. grob betrachtet: keine Zeiger) enthält.

- Ein Default-Standardkonstruktor wird nur dann automatisch vom System erzeugt,wenn kein anderer Konstruktor explizit definiert worden ist.

Default-Konstruktor

Ein Default-Standardkonstruktor wird nur dann automatisch vom System erzeugt, wenn kein anderer Konstruktor explizit definiert worden ist.

```
class X {
    public:

    X(int i);
/* ....*/
};

X x(17);
X x2;                // Syntaxfehler, da Standardkonstruktor nicht mehr
X feld[17];          // automatisch erzeugt wird
```

Aufruf des Standardkonstruktors

```
class X {
   public:
      int i;
      X() { i=4;}   // Deklaration und Definition des Standardkonstruktors
};


. . .

X xxx;                  // Der Standardkonstruktor wird ohne Klammern
                        // aufgerufen.

X xxxx();                // ist syntaktisch richtig. Es wird aber jeweils eine
X xxxx(void);            // Funktion deklariert, die keine Argumente
                         // hat und eine Instanz von X zurückliefert!!!
```

Übung: Was wird ausgegeben?

```
class Zahl
{
public:
   Zahl(int w=9);
   ~Zahl();
private:
   int wert;
};

Zahl::Zahl(int w)  {wert=w;
   cout << "+Z" << wert << " ";
}
Zahl::~Zahl() {
   cout << "-Z" << wert << " ";
}
```

```
void funk1() {
   Zahl z1(1), z2(2);
   cout << "fertig";
}
```

```
1. +Z1 +Z2 -Z1 -Z2
2. +Z1 +Z2 fertig -Z2 -Z1
3. +Z1 +Z2 fertig
4. +Z1 +Z2
5. +Z1 +Z2 fertig -Z1 -Z2
```

Übung: Was wird ausgegeben?

```
class Zahl
{
public:
   Zahl(int w=9);
   ~Zahl();
private:
   int wert;
};

Zahl::Zahl(int w)  {wert=w;
   cout << "+Z" << wert << " ";
}
Zahl::~Zahl() {
   cout << "-Z" << wert << " ";
}
```

1. +Z9 +Z9 +Z9 fertig –Z9 –Z9 –Z9
2. +Z9 +Z9 +Z9 fertig
3. +Z9 +Z9 +Z9 –Z9 –Z9 –Z9 fertig
4. +Z9 +Z9 +Z9 fertig –Z9

```
void funk2() {
   Zahl z1[3]; cout << "fertig";
}
```

Übung: Was wird ausgegeben?

```
class Zahl
{
public:
   Zahl(int w=9);
   ~Zahl();
private:
   int wert;
};

Zahl::Zahl(int w)  {wert=w;
   cout << "+Z" << wert << " ";
}
Zahl::~Zahl() {
   cout << "-Z" << wert << " ";
}
```

```
1. +Z4 +Z5
2. +Z4 +Z5 -Z4
3. +Z4 +Z5 -Z4 -Z5
4. +Z4 +Z5 -Z5 -Z4
```

```
void funk3() {
   Zahl* pz1 = new Zahl(4);
   Zahl* pz2 = new Zahl(5);
   delete pz1;
}
```

Übung: Was wird ausgegeben?

```
class Zahl
{
public:
   Zahl(int w=9);
   ~Zahl();
private:
   int wert;
};

Zahl::Zahl(int w)  {wert=w;
   cout << "+Z" << wert << " ";
}
Zahl::~Zahl() {
   cout << "-Z" << wert << " ";
}
```

```
void funk1() {
   Zahl z1(1), z2(2);
   cout << "fertig";
}
```

+Z1 +Z2 fertig –Z2 –Z1

```
void funk2() {
   Zahl z1[3]; cout << "fertig";
```

+Z9 +Z9 +Z9 fertig –Z9 –Z9 –Z9

```
void funk3() {
   Zahl* pz1 = new Zahl(4);
   Zahl* pz2 = new Zahl(5);
   delete pz1;
}
```

+Z4 +Z5 –Z4

Übung: Was wird ausgegeben?

```
class Zahl
{
public:
   Zahl(int w=9);
   ~Zahl();
private:
   int wert;
};

Zahl::Zahl(int w)  {wert=w;
   cout << "+Z" << wert << " ";
}
Zahl::~Zahl() {
   cout << "-Z" << wert << " ";
}
```

```
void funk1() {
   Zahl z[2];
   cout << "fertig";
}
```

```
1.  +Z9 +Z9 fertig -Z9 -Z9
2.  +Z9 +Z9 -Z9 -Z9 fertig
```

Übung: Was wird ausgegeben?

```
class Zahl
{
public:
   Zahl(int w=9);
   ~Zahl();
private:
   int wert;
};

Zahl::Zahl(int w)  {wert=w;
   cout << "+Z" << wert << " ";
}
Zahl::~Zahl() {
   cout << "-Z" << wert << " ";
}
```

```
void funk1() {
   Zahl z[2];
   cout << "fertig";
}
```

1. +Z9 +Z9 fertig –Z9 –Z9
2. +Z9 +Z9 –Z9 –Z9 fertig
3. fertig

Übung: Was wird ausgegeben?

```
class Zahl
{
public:
   Zahl(int w=9);
   ~Zahl();
private:
   int wert;
};

Zahl::Zahl(int w)  {wert=w;
   cout << "+Z" << wert << " ";
}
Zahl::~Zahl() {
   cout << "-Z" << wert << " ";
}
```

```
void funk2() {
   Zahl** z1[2]; cout << "fertig";

}
```

1. +Z9 +Z9 fertig –Z9 –Z9
2. +Z9 +Z9 –Z9 –Z9 fertig
3. fertig

Übung: Was wird ausgegeben?

```
class Zahl
{
public:
   Zahl(int w=9);
   ~Zahl();
private:
   int wert;
};

Zahl::Zahl(int w)  {wert=w;
   cout << "+Z" << wert << " ";
}
Zahl::~Zahl() {
   cout << "-Z" << wert << " ";
}
```

```
void funk1() {
   Zahl z[2];
   cout << "fertig";
}
```

```
+Z9 +Z9 fertig -Z9 -Z9
```

```
void funk2() {
   Zahl** z1[3]; cout << "fertig";
}
```

```
fertig
```

Übung: Was wird ausgegeben?

```
class Ziege
{
public:
   Ziege(int w);
   ~Ziege();
private:
   int wert;
};

Ziege::Ziege(int w)  {wert=w;
   cout << "+Z" << wert << " ";
}
Ziege::~Ziege() {
   cout << "-Z" << wert << " ";
}
```

```
void funk1() {
   Ziege z[2];
   cout << "fertig";
}
```

1. +Z9 +Z9 fertig –Z9 –Z9
2. +Z9 +Z9 –Z9 –Z9 fertig
3. fertig
4. Syntaxfehler

Übung: Was wird ausgegeben?

```
class Ziege
{
public:
   Ziege(int w);
   ~Ziege();
private:
   int wert;
};

Ziege::Ziege(int w)  {wert=w;
   cout << "+Z" << wert << " ";
}
Ziege::~Ziege() {
   cout << "-Z" << wert << " ";
}
```

```
1. +Z9 +Z9 fertig –Z9 –Z9
2. +Z9 +Z9 –Z9 –Z9 fertig
3. fertig
4. Syntaxfehler
```

```
void funk2() {
   Ziege** z1[3]; cout << "fertig";

}
```

Übung: Was wird ausgegeben?

```
class Ziege
{
public:
   Ziege(int w);
   ~Ziege();
private:
   int wert;
};

Ziege::Ziege(int w)  {wert=w;
   cout << "+Z" << wert << " ";
}
Ziege::~Ziege() {
   cout << "-Z" << wert << " ";
}
```

```
void funk1() {
   Ziege z[2];
   cout << "fertig";
}
```

```
Syntaxfehler, kein
Standard-Konstruktor vorhanden
```

```
void funk2() {
   Ziege** z1[3]; cout << "fertig";
}
```

```
fertig
```

Software-Technik: Vom Programmierer zur erfolgreichen …

8 Abstrakte Datentypen: Einheit von Daten und Funktionalität

8.1 Die Bedeutung von Schnittstellen

8.2 Klassen als abstrakte Datentypen

…

8.2.6 Anwendung: Implementierung von LogTrace

8.3 Generische Programmierung, 2. Teil

8.4 Ausnahmebehandlung, 2. Teil

8.5 Zusammenfassung

Folien mit gelben Punkten  am oberen rechten Rand sind weniger wichtiger für das Verständnis der nachfolgenden Kapitel.

Protokollierung von Funktionsaufrufen

Die LogTrace-Funktionalität

Weiteres Beispiel Motivation

```
void Func() {
   FunktionLog  var("Func");
   . . .
   Func2();
   . . .
}

void Func2() {
   FunktionLog  var("Func2");
   . . .
}
```

Es wird eine Funktionalität gewünscht, die am Anfang einer Funktion schreibt Funktion XXX betreten und beim Verlassen Funktion XXX verlassen.

Ausgabe sollte hier sein:

```
|>Func
    |> Func2
    <| Func2
<|Func
```

Weiteres Beispiel FunktionLog

```
class FunktionLog  {
      enum { MAX_LEN=50, BLANCS=3 };

      string FuncName;
      ...    // weiteres wie mi_aufrufEbene ...

public:
      // Gibt den Namen FuncName aus
      FunktionLog(string Name);

      // Gibt beim Verlassen der Funktion FuncName aus.
      ~FunktionLog();
};
```

Klasse FunktionLog, Teil 2

```
FunktionLog::FunktionLog(string Name) {
   FuncName=Name;

   for (int i=0; i < mi_aufrufEbene; ++i) {
      cout << setw(BLANCS) << " ";
   }
    ++mi_aufrufEbene;
   cout << "|> “  << FuncName;
   cout << endl;
}
```

```
FunktionLog::~FunktionLog() {
   --mi_aufrufEbene;
   for (int i=0; i < mi_aufrufEbene; ++i) {
      cout << setw(BLANCS) << " ";
   }
   cout << "<| “ << FuncName << endl;
}
```

FunktionLog, Anwendungsbeispiel

```
int main() {
  FunktionLog xxx("main");
  {
    FunktionLog xxx("Schleife 1");
    for (int i=0; i < 3; ++i) {
      FunktionLog xxx("i-Schleife ");
      for (int j=0; j < 2; ++j) {
        FunktionLog xxx("j-Schleife");
      }
    }
  }
}
```

Ausgabe:

```
|> main
  |> Schleife 1
    |> i-Schleife
      |> j-Schleife
      <| j-Schleife
      |> j-Schleife
      <| j-Schleife
    <| i-Schleife
    |> i-Schleife
      |> j-Schleife
      <| j-Schleife
....
<| i-Schleife
  <| Schleife 1
<| main
```