

Einführung in C++

David Reitter,
Universität Potsdam, 2001

C

- Dennis Richie: Weiterentwicklung von B, Implementierung auf PDP-11
- Kerningham / Richie: *The C programming Language* (1978)
- ANSI-C (1988)

C++

- Ab 1980, Bjarne Stroustrup: frühe Versionen von „C with Classes“
- 1983 erste Verwendung von C++ außerhalb einer Forschungseinrichtung
- Stroustrup: *The C++ Programming Language*. 1986
- ANSI C++ 1991
- ANSI C++ aktualisiert: 1998 (inklusive *Standard Template Library*)

Kurs: Einführung in C++

Termin:

- Montag, 16-18 Uhr, Hs. 24, Computerpool II

Scheinerwerb:

- Aufgaben nach der Sitzung: Alle bis auf ein Aufgabenblatt müssen bearbeitet werden – Abgabetermin (E-Mail) bis Montag 12.00
- Klausur

Fragen und Aufgabenlösungen an

- reitter@ling.uni-potsdam.de

Kurs: Einführung in C++

- Programmieren in C
- Objektorientiertes Entwickeln
- C++ Syntax
- Standard Template Library (Übersicht und Beispiele)
- Programming Guidelines

Kurs: Einführung in C++

1. Syntax von Standard C (Datentypen und Deklaration, Ablaufkontrolle, Operatoren). Compiler&Linker
2. C-Funktionen und deren Deklaration, Typwandlung. Arbeiten mit Zeigern
3. Komplexe Datentypen (struct, enum, union). Arrays
4. Objektorientierung und Modellierung in UML. Vererbung. (Eckel, Kap. 1)

Kurs: Einführung in C++

5. C++-Klassen und ihre Elemente (Deklaration/Definition). Zugriffsschutz (Eckel, Kap. 6)
6. Grundlegende STL-Klassen und objektorientiertes Programmieren
7. Container-STL-Klassen und Komplexitätsüberlegungen. Iteratoren
8. Sichtbarkeit. Vererbung. Freunde (Eckel, Kap.2a, 5)
9. Überladen von Operatoren und Funktionen

Kurs: Einführung in C++

10. Mehr über Konstruktoren (Aufrufreihenfolge, Copy Constructor)
11. Dokumentations-Standards. Tools zur JavaDoc-Dokumentation (doxygen). „Coding Style“ (Eckel, App.A)
12. C++ Streams, Datei-E/A. Pointer und Referenzen
13. Multiple Vererbung, Laufzeit-Typ-Informationen (virtuelle Methoden)

Literatur

- Bruce Eckel, *Thinking in C++*, 2nd Edition, Vols 1+2,
- Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1997 (3rd edition)

C++ Compiler und Entwicklungsumgebungen

- Borland C++ 5.0 (Compiler, frei)
- gcc (Compiler, frei GNU)
- Visual C++ 6.0 (IDE, GUI-Lib)
- Borland C++Builder 5 (IDE, GUI-Lib)
- Borland Kylix (IDE, Linux, GUI-Lib)
- Metroworks Codewarrior (IDE, MacOS, GUI-Lib)

Einstieg in C

- Hello World
- Elementare Datentypen
- Operatoren
- Ablaufkontrolle
- Deklaration von Funktionen, Header-Files
- Weitere Datentypen, Typumwandlung
- Funktionen und Funktionsargumente

Software-Entwicklung (1)

1. Problemstellung
2. Algorithmus („Idee“)
3. Implementierung
4. Kompilieren und Linken
5. Test. Fehlerhaft? Dann zurück zu 3.

Software-Entwicklung (2)

„Programming is about managing complexity“

- Aufgabenstellung? („Business Case“)
- Anwendungsfälle („Use Cases“)
- System-Anforderungen („Requirements“)
- Architektur (Module, Klassen) („Design“)
- Implementierung
- Test, Debugging, Dokumentation

Funktionsdeklarationen II

Call by Value

```
int generateEFree(int numStates);  
// (...)  
generateEFree(5);  
int size = generateEFree(anzahl);
```

Wert *anzahl* wird **kopiert** in *numStates*

void – "leer"

```
void generateFSA(void);  
(...)  
generateFSA();  
  
int i = generateFSA(); // illegal
```

Call by Reference

```
void erhoehe(int &i);  
  
erhoehe(5); // illegal  
erhoehe(anzahl);
```

Wert von *anzahl* wird nicht kopiert.
Vielmehr sind *i* und *anzahl* **die selbe**
Variable innerhalb von *erhoehe*!

```
void verdoppele(int &i)  
{  
    i += i;  
}  
  
main() {  
    int a = 3, b=4;  
    verdopple(a); verdopple(b);  
    cout << a;  
}
```

```

void erhoehe(const int &i) {
    i += i; // ILLEGAL
}

main() {
    int a = 3;
    verdopple(a); verdopple(a);
    cout << a;
}

```

Funktionsdeklaration

- Funktionen müssen deklariert werden
- Mit dem #include-Statement kann eine beliebige Datei im Sourcecode eingefügt werden
- Bibliotheken (mehrere Funktionen) werden mit #include eingebunden

Funktionsdeklaration III

datenbank.h:

```

const char *findItem(int index);
bool storeItem(int index, string str);

```

datenbank.cpp:

```

const char *findItem(int index) { ... }
bool storeItem(int index, string str) { ... }

```

gui.cpp:

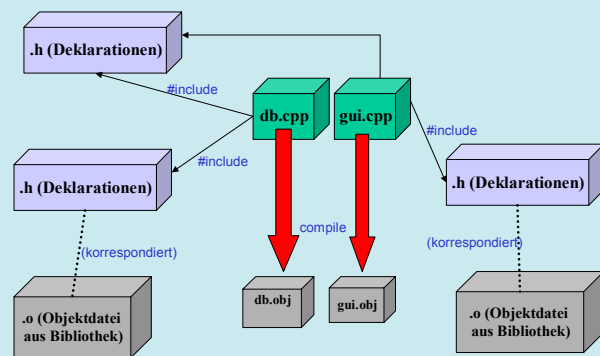
```

#include "datenbank.h"
//... Zugriff auf findItem() und storeItem()

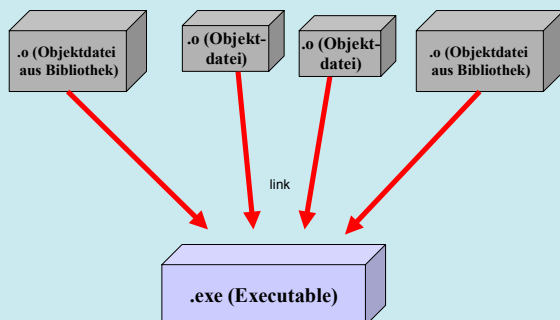
```

Compile: `datenbank.cpp, gui.cpp` - Link: `datenbank.obj, gui.obj`

Compiling



Linking



bcc32: alle cpp-Projektdateien werden kompiliert und gelinkt

Bibliotheken

"Header-Files" (Deklaration)

```

#include <iostream.h> // I/O (cin, cout etc.)
#include <stdlib.h>   // Allgemeines
#include <math.h>     // Mathematik

```

typedef struct

```
typedef struct
{
    int cas;
    int num;
    int gen;
} Merkmale;

// Nutzung:

Merkmale meineFS;
meineFS.num = 4;
cout << meineFS.gen;
```

Arrays

Arrays sind Datenlisten mit Elementen vom selben Typ

Array - Deklaration

```
int umsaetze[50];    // 50 ints
```

- Speicherplatz wird beim Kompilieren reserviert
- umsaetze wird als Variable initialisiert

Array - Zugriff

- `umsaetze[2] = 4900;`
- `if (umsaetze[2] > 5000) {...}`
- `umsaetze[5] ++;`
- `for(i=0; i<50; i++) {`
 `gesamt += umsaetze[i];`
 `umsaetze[i] = 0;`
}

Array – Zugriff II

- `umsaetze[0]` -> erstes Element
- `int umsaetze[50]` reserviert Elemente von `[0]` bis `[49]`
- keine Fehlermeldung / Warnung bei illegalen Zugriffen (`umsaetze[50]`, `umsaetze[400]`)!

```
float i[50];
i[0] = 1.2;
i[1] = 2.8;
i[50] = 0;    // fuehrt zu verstecktem Fehler!!

i[1]++;

cout << i[0];
```

Zeiger

Zeiger enthalten Speicheradressen

| Ptr | Wert | Bezeichner |
|-----|---------|--------------|
| 0 | 4576457 | konto |
| 4 | ??? | tagesumsatz |
| 8 | 789789 | gesamtumsatz |
| 12 | 0 | aktiva |
| 16 | -345 | passiva |
| 20 | 8 | pGUMs |

| Ptr | Wert | Bezeichner |
|-----|---------|--------------|
| 0 | 4576457 | konto |
| 4 | ??? | tagesumsatz |
| 8 | 789789 | gesamtumsatz |
| 12 | 0 | aktiva |
| 16 | -345 | passiva |
| 20 | 8 | pGUMs |

```
int konto = 4576457;
int tagesumsatz;
unsigned int
gesamtumsatz = 789789;
int aktiva = 0;
int passiva = -345;
int *pGUMs =
&gesamtumsatz;
```

```
konto == 4576457
&aktiva == 12      (Referenzierung)
*16 == -345       (Dereferenzierung)
*pGUMs == ???     (Dereferenzierung)
```

Zeiger und Arrays

- Eine Array-Variable ist nur ein Zeiger auf den Speicherbereich:

```
int primzahlen[100];
```

primzahlen ist ein Zeiger auf das erste 'int' der Liste:

```
*primzahlen == primzahlen[0]
```

| Ptr | Wert | Zugriff... |
|-----|------|------------|
| 168 | 11 | z[0] |
| 172 | ? | z[1] |
| 176 | ? | z[2] |
| 180 | ? | z[3] |
| 184 | 12 | z[4] |
| 188 | 16 | z[5] |

| Ptr | Wert | Zugriff... |
|-----|------|------------|
| 168 | 11 | z[0] |
| 172 | ? | z[1] |
| 176 | ? | z[2] |
| 180 | ? | z[3] |
| 184 | 12 | z[4] |
| 188 | 16 | z[5] |

```
int z[6];
z[0] = 11;
z[4] = 12;
z[5] = 16;
```

```
z == 168
z[3] == (nicht definiert!)
&(z[4]) == ???
*z == 11
```

Zeiger und Arrays

```
int i[50];      // dekl., reserv., defin.
int *j;        // dekl.
```

```
j = i;        // i ist vom Typ int*
```

```
j[10] = 2;    // Zugriff möglich!
```

```
j = &(i[9]);  // i[10] ist vom Typ int
j[13] = 555;
```

// i[w] ist nun 555 für welches w?

```
(int *)jahresListe[100]; int jahrIdxMax=0;
int geburtsJahr = 1956;
int heuteJahr = 2001;
```

```
jahresListe[jahrIdxMax++] = &geburtsJahr;
jahresListe[jahrIdxMax++] = &heuteJahr;
```

```
convertJahre(jahresListe, jahrIdxMax);
cout << geburtsJahr << endl;
cout << heuteJahr << endl;
```

```

void convertJahr(int **jListe, int jMax)
{
    for (int i=0; i<=jMax; i++) {
        *(jListe[i]) -= 1900;
    }
}

```

Char-Pointer : C-Strings

```

char *satz;           // Zeichenkette
const char *satz;    // Zeichenkette
satz = "Hello World."; // Zuweisung
const char *wort = "Telefon";

```

```

char satz[500]; // Res. / Dekl. / Zuw.

```

```

const char *satz = "guten Tag.";
const char *zweiterSatz = satz;

satz[0] = 'G';

cout << zweiterSatz;

```

Zeiger sind gefährlich...

```

const char *toString(int num) {

    char buf[50];
    buf = itoa(buf , num, 10);

    return buf;
}

```

```

const char *toString(int num) {

    char buf[50];
    buf = itoa(num, buf , 10);

    return buf;           // LOG. FEHLER
}

```

*Warnung W8075 cptr.cpp 10: Verdächtige
Zeigerumwandlung in Funktion
toString(int)*

```

const char *toString(int i);

```

```

const char *saySentence(int num);

main() {

    char *buffer[100];
    buffer = saySentence(55); ILLEGAL
    buffer[10] = '.';
    cout << buffer;
}

```

```

char *saySentence(int num);

main() {

    const char *buffer[100];
    buffer = saySentence(55); LEGAL, aber sinnlos

    buffer[10] = '.';
    cout << buffer;
}

```

```

char *saySentence(char * buf, int num);

main() {

    const char *buffer[50];
    saySentence(buffer, 55);
    buffer[10] = '\0';
    cout << buffer;
}

```

```

char *semanticsGuy = „Yvan Sag“;
const char *grammarGuy = „Carl Pollard“;

semanticsGuy[0] = `I`;
grammarGuy[0] = `K`; ILLEGAL

```

HA 9.

- Welchen Wert hat nach der Anweisung `float **meineVariable;` die Variable `meineVariable`?
 - 0. [6] Einen Zufallswert, denn `meineVariable` ist ja noch gar nicht initialisiert.
 - 1. [0] Einen Zufallswert, `meineVariable` ist nicht deklariert und darf noch nicht verwendet werden.
 - 2. [0] `meineVariable` ist 0 (NULL).
 - 3. [10] `meineVariable` ist initialisiert mit einem Zeiger auf ein `float`.

Pointer

- Nach der Deklaration `int *pMeineVar,` was liefert der Ausdruck `*pMeineVar`?
- Die deklarierte Variable heißt: ...?
- Sie enthält: ...?

Warnung

- Comparing signed and unsigned values—Compiler warning
- Conversion may lose significant digits—Compiler warning

Compiler-Fehler (1)

- If statement missing (—Compiler error
- Lvalue required—Compiler error
- Could not find a match for argument(s)—Compiler error
- Implicit conversion of 'type1' to 'type2' not allowed—Compiler error

Compiler-Fehler (2)

- Too many types in declaration—Compiler error
- Declaration was expected—Compiler error

Objektorientiertes Programmieren

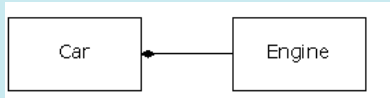
Einstieg in die OOP

- Typizitätsproblem: Was ist ein Tisch, was eine Katze?
- Objekte werden charakterisiert durch ihre Eigenschaften und Fähigkeiten
- Beim Programmieren beschreiben wir über Abstraktionsstufen natürliche und künstliche Objekte

OOP - 3

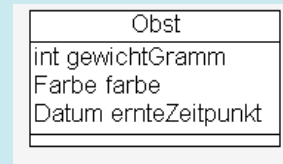
- Aristoteles: „Klasse von Fischen, Klassen von Vögeln“
- Klasse versus Instanz: „ein Fisch“ (Klasse) Wanda (Instanz)
- Jedes Objekt (Instanz) hat einen Typ (Klasse)

Objekte enthalten andere Objekte



- Ein Auto („Car“) hat genau einen Motor („Engine“).
- Ein Auto hat auch einen Preis, eine Farbe, eine Fahrleistung....

Klassendiagramm



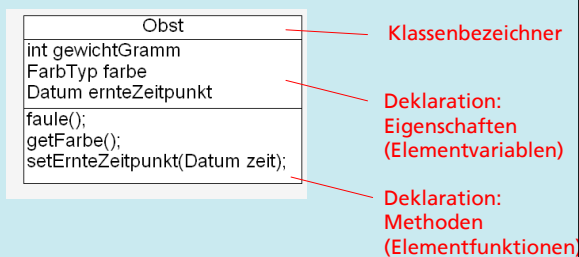
Jedes Objekt hat eine Schnittstelle nach außen

- Was kann ich tun? Welche Parameter (Argumente) müssen mir mitgeteilt werden?
- „Methoden“
- Andere Objekte senden mir „Mitteilungen“, die mir sagen, was ich tun soll.

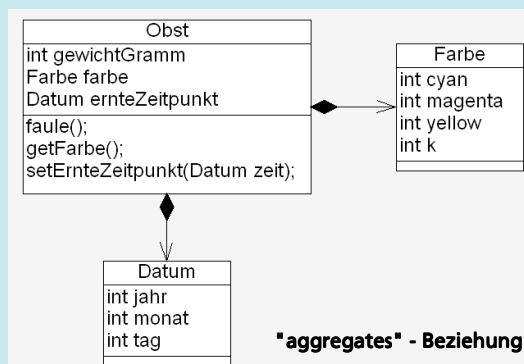
Elemente

- Schnittstellenelemente sind **öffentlich** ("public"). Sie sind für andere Klassen, die ein Objekt der Klasse nutzen, **sichtbar** (benutzbar).
- Interne Elemente: geschützte ("protected") und private ("private") Elemente.

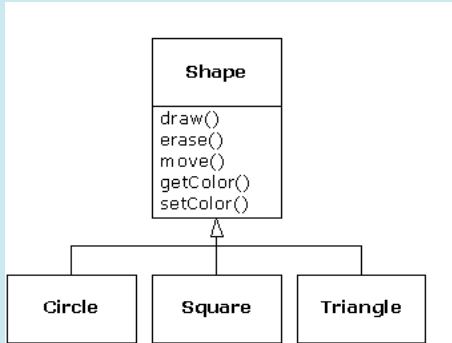
Klassendiagramm 2



Elemente



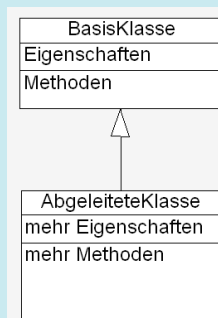
Ableitung von Klassen



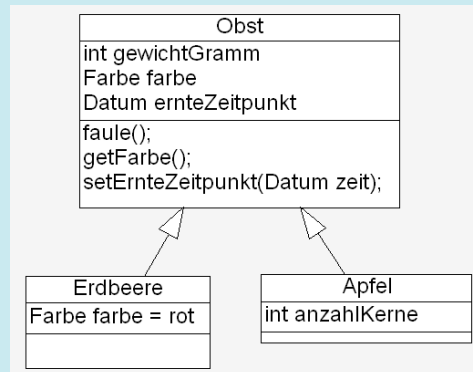
Vererbung

- Abgeleitete Klassen erben Elemente der Basisklasse:
 - Methoden (C++: Elementfunktionen)
 - Eigenschaften (C++: Elementvariablen)
 - alle weiteren Elemente
- (Virtuelle) Elemente können bei der Ableitung ("überladen") überschrieben werden

Ableitung im Klassendiagramm



Ableitung - Beispiel



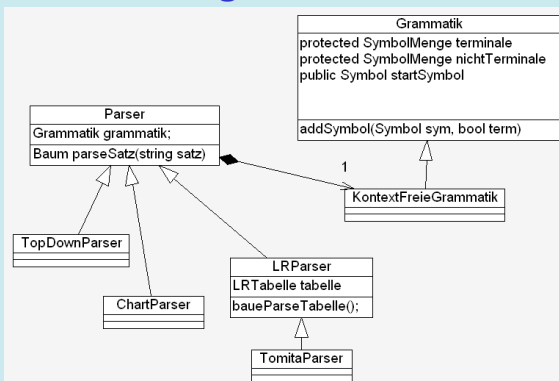
Versteckte Implementierung

- Andere Klassen wissen nur, dass ich etwas tun kann. Sie wissen nicht, wie ich es tue.
- Andere Klassen wissen nicht einmal alles, was ich tun kann. Manche meiner Aktionen behalte ich für mich („private“).
- Bei der Ableitung von Klassen werden nur öffentliche ("public") und geschützte ("protected") Elemente vererbt

Interface-Garantie

- Wird eine Klasse abgeleitet, ist garantiert, dass alle Interface-Elemente (public) der Basisklasse auch in der abgeleiteten Klasse verfügbar sind.

Klassendiagramm in der Praxis



Unified Modelling Language

- Visuelle Sprache, um komplexe Systeme zu modellieren
- Verschiedene Diagrammtypen mit standardisierten Schreibweisen
- UML 1.1 – Standard: Nov 1997

UML Diagramme

- Use Case Diagram
- Class Diagramm
- Behavior Diagrams:
 - statechart diagram
 - activity diagram
 - interaction: sequence and collaboration
- Implementation Diagrams:
 - component diagram
 - deployment diagram

Modellierungssoftware

- Rational Rose (prof.)
- WithClass 2000
- Metamill 1.0 (klein)

Die Programme bieten Code Reverse Engineering und Code Generation, div. UML-Diagramme, teilw. Dokumentation uvm.

OOP / UML Literatur

- Martin Fowler: **UML Distilled**, Second Edition: A Brief Guide to the Standard Object Modeling Language. Addison-Wesley
- <http://www.codeproject.com/cpp/oopuml.asp>

Klassen in C++

- Deklarationssyntax:

```

class Name {

    // Elemente

};
    
```

OOP (Eckel)

1. **Everything is an object.** Think of an object as a fancy variable; it stores data, but you can “make requests” to that object, asking it to perform operations on itself. In theory, you can take any conceptual component in the problem you’re trying to solve (dogs, buildings, services, etc.) and represent it as an object in your program.

OOP (Eckel)

2. **A program is a bunch of objects telling each other what to do by sending messages.** To make a request of an object, you “send a message” to that object. More concretely, you can think of a message as a request to call a function that belongs to a particular object.

OOP (Eckel)

3. **Each object has its own memory made up of other objects.** Put another way, you create a new kind of object by making a package containing existing objects. Thus, you can build complexity in a program while hiding it behind the simplicity of objects.

OOP (Eckel)

4. **Every object has a type.** Using the parlance, each object is an *instance* of a *class*, in which “class” is synonymous with “type.” The most important distinguishing characteristic of a class is “What messages can you send to it?”

OOP (Eckel)

5. **All objects of a particular type can receive the same messages.** This is actually a loaded statement, as you will see later. Because an object of type “circle” is also an object of type “shape,” a circle is guaranteed to accept shape messages. This means you can write code that talks to shapes and automatically handles anything that fits the description of a shape. This *substitutability* is one of the most powerful concepts in OOP.

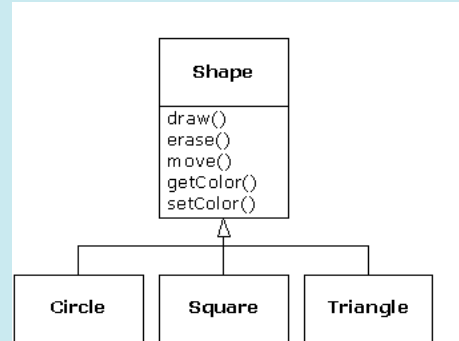
OOP – kurz gefasst

- Klassen beschreiben Mengen von Objekten, die selben Mengen von Eigenschaften und Fähigkeiten (Methoden) haben
- Wenn eine Klasse instantiiert wird, erhält man ein Objekt (eine Instanz)
- Die Instanz hat genau die Eigenschaften und Fähigkeiten der Klasse

Ableitung - Wdh

- Abgeleitete Klassen haben alle Elemente (Eigenschaften/Methoden) der Basisklasse(n)
- Zusätzlich können abgeleitete Klassen weitere Elemente enthalten
- Ableitung: "is-a" - Relation

Ableitung von Klassen



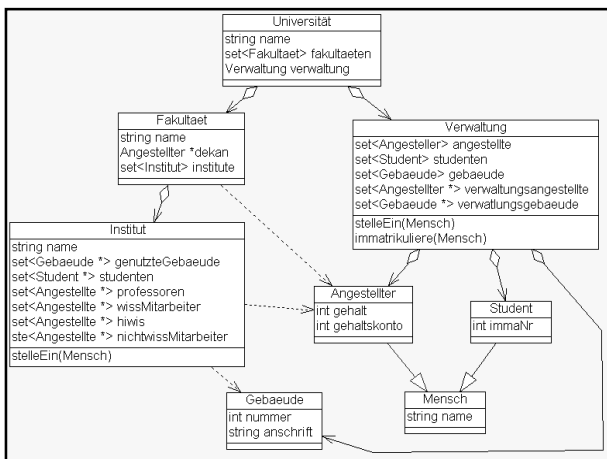
Zugriffsschutz

- **public:** Elemente sind für alle anderen Klassen sichtbar und werden vererbt
- **protected:** Elemente sind nach außen unsichtbar und werden vererbt
- **private:** Elemente sind nach außen unsichtbar und werden nicht vererbt

Klassendeklaration

```

class Name {
    public:
        // Elemente
    protected:
        // geschützte Elemente
    private:
        // private Elemente
};
    
```



Klassendeklaration

- Elemente einer Klasse werden wie C++ Variablen und Funktionen deklariert
- **Variablen:**

```

int koerper;
Datum geburtstag;
        
```
- **Elementfunktionen (Methoden):**

```

bool hatGeburtstag(void);
int calcNettoGehalt(float steuersatz);
char *toString(void);
        
```

Klassendeklaration (2)

```
class Mensch {
public:
    string name;
    int alter;
    int bruttoGehalt;
    int calcNettoGehalt(float steuersatz);
protected:
    // geschützte Elemente
private:
    // private Elemente
};
```

Klassendefinition (Implementierung)

- Methoden müssen definiert werden.
Syntax:

```
int Angestellter::calcNettoGehalt(float steuersatz) {
    return (bruttoGehalt * (1-steuersatz));
}
// "bruttoGehalt" muss eine Elementvariable der
// Klasse "Angestellter" sein (oder global).
```

Klassen-Implementierung

```
// falsch
int calcNettoGehalt(float steuersatz)
{
    return (bruttoGehalt * (1-steuersatz));
}

// (normale, datei-global bekannte Funktion!)
```

Klassen

- Deklaration → Header-File (.h)
- Definition ("Implementierung") → Code-File (.cpp)
- Der Dateiname sollte dem Klassennamen entsprechen (Java-Standard)

Deklaration/Definition

- Definition von Elementfunktionen direkt bei Deklaration möglich:

```
class Mensch {
public:
    string name;
    int alter;
    int bruttoGehalt;
    int calcNettoGehalt(float steuersatz)
        {return (bruttoGehalt * (1-steuersatz));}
};
```

Nutzung von Klassen

- Deklarationen von Variablen:

```
int jahresetat;
Universitaet uniPotsdam;
Angestellter neuerAngestellter;
string eingabe;
const char *begrueessung;
```

Nutzung von Klassen (2)

- Zugriff auf Elemente durch den .-Operator
- Elementvariablen (Eigenschaften):
`cout << neuerAngestellter.name << endl;`
- Elementfunktionen (Methoden):
`cout <<
 neuerAngestellter.calcNettoGehalt(0.35);`

C++ Typen

- einfache Datentypen (int, float, bool, char, long, double, ...)
- Strukturen (mit `typedef struct` definiert)
- Klassen
- Pointer auf Typen: `Typ*`
- Template-Klassen, z.B. `set<Typ>`

Deklarationen

Typ Bezeichner;

z.B.:

```
void Verwaltung::stelleEin(string _name) {  
  Mensch neuerAngestellter;  
  neuerAngestellter.name = _name;  
  angestellte.insert(neuerAngestellter);  
}
```

Ableitung in C++

```
// Klasse Mensch muss deklariert sein  
class Angestellter : public Mensch  
{  
  
  // neue Elemente.  
  // Elemente aus Mensch werden  
  // geerbt  
  
}
```

Aufgabe V

- Implementierung der erstellten Klassenhierarchie in C++-Dateien

Standard Template Library

- C++ Standard
- Klassenbibliothek
- Parametrisierte Klassen ("Templates")

```
set<int> t; // Menge von Integers  
vector<Datum> v; // Vektor von Daten
```

string-Klasse

- Klasse aus der Standard Template Library
- Einbinden über

```
#include <string>
using namespace std;
```
- Zuweisung von (const char*) Typen möglich: `string name = „Toni“;`

string -> const char*

```
string name = „Toni“;
const char *test = name.c_str();

// Achtung, begrenzte
// Gültigkeit des Pointers
```

string: Zugriff auf einzelne Elemente

```
string name = „Toni“;

char buchstabe = name[0];
char buchstabe2 = name.at(1);
```

string: Finden von Elementen

Aus Header-Datei (siehe Hilfe-Funktion zum string-Typ)

- `size_type find(E c, size_type pos = 0) const;`
- `size_type find(const E *s, size_type pos = 0) const;`
- `size_type find(const E *s, size_type pos, size_type n) const;`
- `size_type find(const basic_string& str, size_type pos = 0) const;`

Zeichen finden

```
string name = "Bill Clinton";
int lz_pos = name.find(" ");

// wenn die Zeichenkette nicht
// gefunden wird:
lz_pos==string::npos
```

Teilstrings: string::substr()

```
string satz = „Toni ist jetzt
verheiratet.“;

string zweitesWort;

zweitesWort = satz.substr(6, 3);
```

Teilstrings - Beispiel

```
string vorname, nachname;
string name = "Bill Clinton";
int lz_pos = name.find(" ");
if (lz_pos != string::npos) {
    vorname = name.substr(0, lz_pos-1);
    nachname = name.substr(lz_pos+1,500);
} else {
    nachname = name;
}
```

Ausgabe von Strings

```
cout << initText << endl;

const char *ausgabe =
    initText.c_str();

// Achtung: Zeiger begrenzt gültig
```

Konkatenation

```
string eins = "Hallo";
string zwei = "wie geht's";
string satz = eins + " " + zwei;

// " " ist (const char*) und wird
// automatisch umgewandelt.
// nicht erlaubt aber:
// string satz2 =
//     "Hallo" + " Leute";
```

string: Typ oder Klasse?

string ist eine Abkürzung:

```
typedef basic_string<char> string;
typedef basic_string<wchar_t> wstring;
```

C++ Syntax: Konstruktoren

- besondere Elementfunktionen
- Deklaration: Name wie Klassenname; kein Return-Wert (auch nicht ,void'):

```
class Student : public Mensch {
    Student()
    { matrnr = 0; }
    Student(string name);
    • }
```

Konstruktoren (2)

- Werden aufgerufen, sobald Objekt erzeugt worden ist. Z.B. bei der Deklaration:

```
Student tina;
// ruft Konstruktor auf
```

- Lokale Variablen werden bei Funktionsaufruf angelegt; Elementvariablen bei Instanziierung der Klasse

Konstruktoren (3)

- Konstruktoren können Argumente erhalten.

```
Student tina („Tina Tüte“);
```

- `Student tine = „Tina Tüte“;`

Konstruktoren - Sonderfälle

- Default-Konstruktor: ohne Argument

```
Student tina;  
// ruft Default-Konstruktor auf
```

- Kopier-Konstruktor:

```
Student tina(thorsten);  
// Kopiert thorsten  
Student tine = thorsten;  
// kopiert thorsten
```

Destruktoren

- Werden bei Zerstörung eines Objekts aufgerufen.
- Keine Argumente, kein Return-Wert
- Deklaration:

```
class Student : public Mensch {  
    Student()  
    { /* Implementierung */ }  
    Student(string name);  
    ~Student();  
}
```

Aufruf Copy-Constructor

- Bei der Zuweisung mit `operator=` und bei der Initialisierung durch Kopieren
- Beim Aufruf von (Element-)Funktionen, wenn Argumente per „call-by-value“ übergeben werden
- insbesondere von vielen STL-Funktionen, wie `vector<T>::push_back`, `set<T>::insert`

Casting

```
// string err = "eins " + "zwei";  
// explizit:  
string okay = string("eins") +  
    string("zwei");  
  
// automatisch:  
string okay2 = string("eins") +  
    "zwei";
```

char[] versus string

- `string` ist dynamisch (kann wachsen). Ein C-Array ist konstant groß.
- `string` kann bequem verglichen (`==`), kopiert (`=`), gesetzt (`= „...“`) werden. Für C-Arrays benötigt man die Funktionen `strcmp()`, `strdup()` und `strcpy()`.
- `char[]` ist fehleranfällig (Gültigkeit, Größe, Memory Leaks)

const - Modifizierer

- „const“ garantiert, dass eine Funktion am Objekt nichts verändert

```
string toString(void) const;
```

- Die Elementfunktion darf keine Elementvariablen des Objekts verändern und keine nicht-konstanten Elementfunktionen der selben Klasse aufrufen.
- Stil: Alle Funktionen, die nichts verändern, sollten unbedingt auch als ‚const‘ deklariert werden!

const - Beispiel

```
class Complex;           // zus. Dekl.
class Complex {
public:
    Complex add(Complex second) const;
}
```

```
/* Objekt, für das add() aufgerufen
wird, wird niemals dadurch
verändert. */
```

Elementfunktionen: call by reference

```
class Complex;
class Complex {
public:
    Complex& add(const Complex& second) const;
}
```

```
/* Argument second wird NICHT KOPIERT vor
Ausführung der Elementfunktion. Die Funktion
darf second nicht verändern. → Zeitvorteil!
Return-Wert wird NICHT KOPIERT nach
Ausführung der Elementfunktion.
→ Zeitvorteil */
```

Elementfunktionen: call by pointer

```
class Complex;
class Complex {
public:
    Complex* add(const Complex* second) const;
}
```

```
/* Argument second ist ein Pointer. Das Objekt,
auf das er zeigt, wird nicht kopiert vor
Ausführung. Die Funktion darf das Objekt
nicht verändern. → Zeitvorteil!
Die Funktion gibt einen Pointer auf ein
Objekt zurück. Dieses darf aber nicht lokal
angelegt sein (sonst würde es zerstört!). */
```

Der -> Operator

- . ermöglicht Zugriff auf Element:
davidsAuto.motor.leistung = 78;
- -> interpretiert zusätzlich den linksstehenden Ausdruck als Pointer und dereferenziert diesen

```
pSatz = &satz;
// äquivalent:
satz.assignTags();
(*pSatz).assignTags();
pSatz->assignTags();
```

this

- spezielle Variable, nur bekannt in Elementfunktionen. Zeiger auf das Objekt, für welches die Funktion aufgerufen wurde:

```
Aufruf: autoFahrer.blinkt();
```

```
in blinkt(): this zeigt auf autoFahrer
```

this

```
(1) this->farbe = 24;  
(2) (*this).farbe = 24;  
(3) farbe = 24;
```

(farbe ist eine Elementvariable der Klasse, zu der obiger Code gehört)

Wo liegt der Unterschied zw. (1), (2), (3)?

this - Beispiel

```
Complex& Complex::increment(void) {  
  
    real++;  
    return *this;  
  
}  
  
// Rückgabe des aktuellen Objekts
```

static - Modifizierer

- Eine statische Elementvariable hat für alle Objekte den selben Wert, genauer: sie existiert nur einmal

```
static profileUsageCount;
```

- Eine statische Elementfunktion wird ohne Objekt genutzt

```
static float calcSqrt(float arg);
```

- Eine statische Funktion darf nur statische Elementvariablen verwenden

Statische Elemente

```
class Parser {  
  
    static int profileUsageCount;  
    static float calcSqrt(float arg);  
  
}
```

Statische Variablen

Statische Elementvariablen müssen deklariert und „implementiert“ werden:

... in cpp-Datei (datei-global):

```
int Parser::profileUsageCount;
```

Gültigkeit: nur in dieser cpp-Datei!

Nutzung ohne Angabe eines Objektes:

```
Parser::profileUsageCount++;
```

Nutzung aus der Klasse heraus auch möglich, dann:

```
profileUsageCount++;
```

Statische Funktionen

- Aufruf ohne Objekt-Angabe

```
Parser::initialize („myGrammar.gra“);
```

- Funktion darf keine Referenzen auf nichtstatische Elemente (Funktionen, Variablen) enthalten. Es existiert kein ‚this‘ Zeiger.

Statische Elemente

- Statische und nicht-statische Elemente dürfen beide in einer Klasse enthalten sein
- Statische Elemente dürfen nur statische Elemente referenzieren (Variablen nutzen, Funktionen aufrufen), weil kein ‚this‘ Zeiger vorhanden ist.
- Nicht-statische Elemente dürfen alle Arten von Elementen referenzieren

Klassendefinition - zirkulär

```
class Sign {
public:
    Sign();
    ~Sign();

    string attributeName;
    Sign value; // zirkuläre Def. ->
               // illegal!
}
```

1. An dieser Stelle ist „Sign“ als Typ noch nicht bekannt

Klassendefinition - zirkulär

```
// Klassendeklaration vorab, damit Sign bekannt
class Sign; // Dekl. „Sign“ ist eine Klasse

// Klassendefinition
class Sign {
public:
    Sign();
    ~Sign();
    string attributeName;
    Sign value;
}
```

2. Sign ist zwar als Typ bekannt, aber nicht definiert. Deshalb kann die Größe nicht festgestellt werden.

Klassendefinition - zirkulär

```
// Klassendeklaration vorab, damit Sign bekannt
class Sign; // Dekl. „Sign“ ist eine Klasse

// Klassendefinition
class Sign {
public:
    Sign();
    ~Sign();
    string attributeName;
    Sign *value; // als Pointer erlaubt
}
```

3. Die Größe eines Zeigers ist bekannt. Sign als Typ ist ebenfalls bekannt. → Erlaubt

Deklar. / Defini. / Impl.

```
class Sign; // Deklaration (meist unnötig)

class Sign {
    string attributeName;
    Sign *unify(const Sign *secSign);
} // Definition

Sign* Sign::unify(const Sign *secSign) {
    ...
} // Implementierung einer
// Elementfunktion
```

Aufrufreihenfolge

- Bei abgeleiteten Klassen wird der Standard-Konstruktor der Basisklasse vor dem eigentlichen aufgerufen
- C++ garantiert die vollständige Initialisierung eines Objektes

Aufrufreihenfolge festlegen

- Alternativ können andere Konstruktoren spezifiziert werden:

```
class Apfel : public Obst {
    Apfel() : Obst(rot), Ding(natur) { ... }
}
// Obst muss einen Konstruktor der Form
// Obst(FarbEnumTyp), Ding einen der Form
// Ding(ArtTyp) haben
```

Konstruktoraufruf für Elemente

```
class Apfel : public Obst {
    Apfel : Obst(rot), Ding(natur) { ...
}
}
```

STL: Container-Klassen

- enthalten mehrere Elemente desselben Typs
- unterscheiden sich in Speicheralgorithmus, also auch in der Komplexität der Zugriffszeit
- Methoden für's Hinzufügen und Auslesen von Elementen
- siehe *Eckel*, Vol 2, Kap 4

Container-Klassen (2)

- Parametrisierung durch Angabe eines Typs (und weiterer Parameter) in eckigen Klammern
- `set<int>` Eine Menge Integer
- `vector<string>` Ein Vektor Strings
- `map<int, string>` Eine int-string Map

Container-Klassen (3)

- automatisch geordnet:
`set<T>` (Menge)
`map<T1, T2>` ('Dictionary')
u.a.
- Freie Reihenfolge:
`vector<T>` (Vektor - dynamisch)
`array<T>` (Array - konstant)
u.a.

vector<T>

- Geordnete Liste
- Beliebiger Zugriff anhand des Indexes
- Zugriffszeit: $O(1)$
- Nicht festgelegte Größe (dynamisch)

vector<T>: Methoden

- Elemente hinzufügen:
vector<T>::push_back(const T &elem)

vector<T> - Beispiel

```
#include <vector>
vector<string> namen;
namen.push_back("Toni");
namen.push_back("Alfons");
cout<<namen[0]<<endl;
```

Iteratoren

- `class iterator;`
- vergleiche: Zeiger
- Iteratoren ermöglichen eine Referenz auf ein einzelnes Container-Element
- Die ursprüngliche Form eines Iterators: Index (in einem Array)
- Iteratoren können Zugriffsreihenfolge einschränken (z. B. nur vorwärts, nicht beliebig)

Iteratoren (2)

- Jede Container-Klasse definiert ihre eigenen Iterator-Typen
- Zugriff nur über Locator mit `::Operator`:

```
vector<string>::iterator vecIt;
vector<string>::const_iterator vecItc;
```

Iteratoren (3)

- Mit `++` kann man Iteratoren, die Vorwärtsverkettung erlauben, incrementieren.
- nach `it++` zeigt `it` auf das nächste Element im Container
- der `--` Operator ist nicht immer verfügbar

begin() / end()

- `begin()` liefert einen Iterator, der auf das erste Element eines Container-Objekts zeigt
- `end()` liefert einen Iterator, der hinter das letzte Element eines Container-Objekts zeigt

begin() / end() - Beispiel

```
void show(vector<string> teilnehmer)
{
    vector<string>::iterator tIt;
    for (tIt = teilnehmer.begin();
         tIt != teilnehmer.end();
         tIt++) {
        cout << *tIt << endl;
    }
}
```

vector<T>::iterator

- Elemente in Vektoren finden:
iterator find(const T &elem)
(liefert end() wenn nicht gefunden)

Dereferenzierung

- Iteratoren können wie Zeiger mit * dereferenziert werden:

```
map<int, Student> studenten;
map<int, Student>::iterator sIt =
    studenten.find(135480);
if (sIt != studenten.end())
{
    cout << (*sIt9).second.name << endl;
}
```

Dereferenzierung mit ->

- Der -> Operator wirkt auch bei Iteratoren wie von Zeigern gewohnt: Dereferenzierung des linksstehenden Ausdrucks:

```
map<int, Student> studenten;
map<int, Student>::iterator sIt =
    studenten.find(135480);
if (sIt != studenten.end())
{
    cout << sIt9->second.name << endl;
}
```

pair<T1,T2>

- Ein geordnetes Paar mit genau zwei Objekten von Typ T1 und T2
- Konstruktor nimmt zwei Argumente (Typ T1 und T2)
- Lesen und Schreiben der enthaltenen Objekte direkt über die Elementvariablen 'first' und 'second', also:

```
pair<string, string>
telefonbucheintrag("Robert", "030-490047810");
cout << telefonbucheintrag.second;
```

map<T1, T2>

- Geordnete Liste von Paaren; geordnet nach erstem Element
- Zugriff mit bekanntem ersten Element
- Interne Darstellung: Binärbaum
- Zugriffszeit: $O(\log n)$
- Speicherkomplexität: $O(n)$
- Ordnungsfunktion für T1 muss bekannt sein! (Default: less<T1>)

map<T1,T2>

- Hinzufügen von Elementen: Immer mit einem Objekt vom Typ pair<T1,T2>
- Lesen von Elementen: Normalerweise anhand eines Elements vom Typ T1 ("Schlüssel"). Die Funktion "find" gibt ein Objekt vom Typ pair<T1,T2> zurück.

map<T1,T2> (Beispiel)

```
#include <map>
map<string, AVM> lexikon;
lexikon.insert(pair<string, AVM>
    ("Radio",
    AVM("syn:head:cat:n")));
cout << lexikon.find("Radio")->second;
```

map (Iterator)

```
map<string, string> telefonliste;
map<string, string>::iterator telIt;
telIt = telefonliste.find("Peter");
if (telIt != telefonliste.end())
{
    cout << telIt->first << ": " <<
    telIt->second << endl;
}
```

set<T>

- geordnete Menge
- keine doppelten Elemente
- Zugriffszeit: $O(\log n)$
- Speicherkomplexität: $O(n)$
- schneller 'Enthalten'-Test
- Eine Ordnungsfunktion muss bekannt sein. Default: less<T>
- Eine Vergleichsfunktion muss bekannt sein. Default: operator==<T>

set<T> - Beispiel

```
#include <set>
set<string> stopWords;
stopWord.insert(string("der"));
stopWord.insert(string("kein"));
stopWord.insert(string("nicht"));
string givenWord = "kein";
if (stopWord.find(givenWord) !=
    stopWord.end())
    cout << givenWord << " is a stop word.";
```

set<T>::insert

- insert() und entsprechende Methoden aus anderen STL-Klassen fügen Kopien der Argumente ein.
- Wie werden Kopien erzeugt?
- Das C++-Programm erzeugt automatisch Kopien von Funktionsargumenten, zerstört diese aber auch wieder.

Hausaufgabe

- Aufgabenblatt 7 (Netz)
- Aufgabe VIII (Programmierprojekt) bis in 14 Tagen

Dynamische Objekte

- Nicht immer ist bekannt, welche und wieviele Objekte zur Laufzeit erzeugt werden sollen
- Mit dem Operator `new` kann man Objekte zur Laufzeit anlegen. Mit `delete` zerstört man sie wieder.
- `new` liefert einen Zeiger auf das erzeugte Objekt zurück

new / delete

```
Person *pAutoFahrer = new Person();
string *pGruss = new string("Hallo!");
(...)
delete pAutoFahrer;
delete pGruss;
```

new / delete 2

- `new` alloziert Speicher für das Objekt, `delete` gibt ihn wieder frei.

Objekte – dynamisch?

Lokale Variablen sind wie dynamische

- werden aber automatisch alloziert (bei Eintritt in Anweisungsblock)
- werden automatisch gelöscht (bei Austritt aus Anweisungsblock)

C++ / dynamisch

| | |
|---|--|
| <pre>int test (int a, int b) { string t1; t1 = "Hallo"; return a+b; }</pre> | <pre>int test (int a, int b) { string* pT1 = new string(); *pT1 = "Hallo"; delete pT1; return a+b; }</pre> |
|---|--|

Wann benötigt man dynamisch allozierte Objekte?

- Wenn Methoden Objekte erzeugen und deren Adresse zurückgeben sollen
- Wenn wg. Effizienz Objekte nicht bei 'return' kopiert werden sollen, sondern nur Zeiger

Beispiel: Tier-Klasse

- Animal-Klasse mit ,giveBirth'-Methode

```
Animal* giveBirth(void);
```

- Tiere sollten Geburtstag und Namen sowie Zeiger auf beide Elternteile besitzen und eine Liste mit Zeigern auf Nachkommen haben

C++ Syntax: Operatoren

- sind Elemente einer Klasse
- oder global definiert
- sind Methoden
- "Syntactic Sugar" (Eckel)

```
a+b == add(a,b)
```

Binäre Operatoren

```
class Complex; // zusätzliche Dekl.  
class Complex {  
public:  
    Complex operator+ (const Complex&  
        second) const;  
protected:  
    float real, imag;  
}
```

Unäre Operatoren

- stehen vor einem Term
- werden global, d.h. außerhalb der Klasse deklariert

```
Complex operator- (const Complex& value)  
const {  
    Complex ret;  
    ret.real = - value.real;  
    ret.imag = - value.imag;  
    return ret;  
} // globaler, unärer Operator
```

Fast alle Operatoren können überladen werden

- operator+ - Grundrechenarten
- operator++ - Post/Prefix
- operator= - Zuweisung
- operator& - Referenzierung
- operator* - Dereferenzierung
- operator[] - Zugriff auf Element

Operatoren-Präzedenz

- die Präzedenz von Operatoren kann nicht verändert werden!
- $4+3*7$ wird immer implizit geklammert als $+(4, *(3,7))$

Tier-Klasse

- Definition des Operators [] für die Tier-Klasse zum Zugriff auf die Kinder eines Tiers

Datei Ein/Ausgabe

- Verschiedene Ebenen für E/A:
 - System-Funktionen `fopen()`, `fclose()`
 - C-Funktionen: `fopen()`, `fclose()`, `FILE`
 - C++-Streams: `ifstream`, `ofstream`
- ‚Streams‘: Datenströme, die idR Daten zwischenspeichern können
- Stream-Typen sind auf C++-Klassen zurückzuführen

E/A Modi

- Ein- oder Ausgabe in die Datei?
- Datei neu anlegen („create“) oder Inhalte anhängen („append“)?
- Binär- oder Textmodus?
(Im Textmodus wird zeilenweise gelesen, Zeilenendmarkierungen werden automatisch verarbeitet.)

ifstream / ofstream

- Für Lese/Schreiboperationen ist ein ‚Stream‘-Objekt zu erzeugen:

```
ifstream  
    lexiconFile („lexicon.txt“ );  
ofstream  
    parseTreeFile („parse.txt“ );
```

E/A Modi

- Stream-Konstruktor akzeptiert Modus-Spezifikation:

```
ios::app („append“ - Anhängen)  
ios::binary (Binärmodus)
```

z.B.:

```
ofstream logFile („current.log“,  
                ios::app)
```

E/A Operatoren

<< Ausgabe, z.B.

```
logFile << „[Warnung] Satz nicht  
geparst:“ << sentence << endl;
```

E/A Operatoren

>> Eingabe eines einzelnen
Elementes:

```
int in;  
inputFile >> in;
```

E/A Funktionen

- getline() – Zeile einlesen:

```
char buffer[100];  
inputFile.getline(buffer, 99);  
// Zeile einlesen und als C-String in  
// ‚buffer‘ schreiben
```

Coding Style

- 80% der Arbeit an einem Quelltext wird für Wartung verwendet
- Kaum ein Programm wird von der Entwicklung bis ans Ende seiner Lebenszeit von einem Programmierer betreut
- Code-Konventionen erhöhen die Lesbarkeit, so dass neue Entwickler schneller damit zurechtkommen
- Coding-Style ist ein Schlüssel zu Teamwork und Effizienz

Dateinamen

| Suffix | File Type |
|--------|-----------------|
| .h | Header Files |
| .cpp | C++ source code |
| .c | C source code |

Der Dateiname sollte dem
Klassennamen entsprechen.

File Naming

- "Name the file after what it is. If you can't think of what it is that is a clue you have not thought through the design well enough. "
- "Compound names of over three words are a clue your design may be confusing various entities in your system. Revisit your design. "

C++ Header-Dateien

1. Initiale Kommentare
2. Klassenbeschreibung
3. #include-Statements
4. Klassendeklarationen

Datei-Organisation

- Maschinenabhängiger Code gehört in separate Dateien (Austauschbarkeit!)
- Betriebssystemspezifischer Code gehört in separate Dateien!
- Datei über ca. 70kB zeigen Ihnen, dass wahrscheinlich mit Ihrer Klassenarchitektur etwas nicht in Ordnung ist...

Kommentare

- Jede Klasse, jedes Element muss dokumentiert werden
- Dokumentation sollte während der Entwicklung geschrieben werden, nicht zwei Monate danach
- Dokumentation muss so verständlich sein, dass auch ein anderer Ihren Code versteht!

ccdoc / javadoc - Standard

```
/**
 * <brief description>
 *
 * <full description>
 * <directive>
 * <directive>
 * .
 * .
 * <directive>
 */
```

ccdoc / javadoc - Klassen

```
/**
 * A one line description of the class.
 *
 *
 * A longer description. alsdkjl qwoeijqo lkfjsdf
 * sdlfjkslf qwifjfo lksdf sfoijf lwkejfw
 *
 * @version 1.0
 * @author Theo Testmann
 * @see 200106_PROJ_ImplRequiremen.doc
 */
```

ccdoc / javadoc: Methoden

- @param <Param>** – Spezifikation eines Arguments (jedes Argument muss dokumentiert werden!)
- @return** – Dokumentation des Rückgabewerts
- @author** – explizite Autorenangabe

ccdoc / javadoc - Methoden

```
/**
 * retrieve an expansion list
 *
 * getProductionExpansion retrieves a precalculated
 * expansion list for a given Production.
 *
 * @param pProd A reference to the production which...
 *
 * @return A list of indexes referring to the exp...
 */
vector<long> getProductionExpansion(Production *pProd);
```

schlechte Kommentare

```
/**
 * getProductionExpansion
 *
 * returns a LongVec from a Production
 * @param val The value to assign to this object.
 *
 * @return A reference to this object.
 */
LongVec getProductionExpansion(Production *pProd);
```

besser:

```
/**
 * retrieve an expansion list
 *
 * getProductionExpansion retrieves a precalculated
 * expansion list for a given Production.
 *
 * @param pProd A reference to the production which
 *              is to be expanded
 *
 * @return A list of indexes referring to the
 *         expansion productions
 */
vector<long> getProductionExpansion(Production *pProd);
```

Benennung

- Namen müssen Sinn haben. Wenn sich der Inhalt einer Variable im Laufe der Entwicklung verändert, muss der Name angepasst werden.
Beispiel: configFile enthält einen Dateinamen. In der überarbeiteten Version enthält es einen Verzeichnispfad. Umbenennung in configPath nötig!

Benennung

- Niemals Namen gleich existierenden (STL-) Typen wählen, auch nicht für lokal gültige Bezeichner!

Klassen-Namen

- Großer Anfangsbuchstabe
- Großbuchstaben an jedem Wortanfang
- Keine _Unterstriche

```
class ChomskyNormalFormRule;
class TomitaParser;
class OntologyDBInterface;
```

Methodennamen

- Orientieren Sie sich am Rückgabewert
- Bezeichner wie bei Klassennamen

```
int getSize() const;
void setSize(int size);
string toString() const;
bool isAdmissible() const;
```

Argumentnamen

- beginnen mit Kleinbuchstaben
- neue Worte innerhalb des Arguments beginnen mit Großbuchstaben:

```
int showDialog(const Dialog&
               dialog, int userRef);
```

Lokale Variablen

- Nur Kleinbuchstaben benutzen
- Wort-Trennung mit Unterstrichen:

```
int MyClass::handleError(int errorNumber) {
    int error= osErr();
    Time time_of_error;
    ErrorProcessor error_processor;
}
```

Klassendeklaration

- Anordnung:
 1. public (Schnittstelle!)
 2. protected
 3. private
- Kon-/Destruktoren zuerst

Ableitungen

- leiten Sie lieber von STL-Klassen ab, als Elemente einzubinden. So erbt Ihre Klasse die passenden Methoden!
-

Typen

- definieren Sie mit typedef Klassenspezifische Typen. Vergeben Sie am Sinn orientierte Namen!

```
• typedef vector<long> LongVector;
• typedef map<long, vector<Sign*> > L_pSignVec_Map
besser:
• typedef long ProductionIdx;
• typedef map<long, vector<Sign*> > FirstList;
```

Optimierung

- Optimieren Sie nicht.
- Ihr System-Design und Ihre Algorithmen sollte für Geschwindigkeit sorgen.
- Optimieren Sie Code höchstens dann, wenn dort ein Flaschenhals ("bottleneck") entstanden ist!

Empfehlungen

- Eliminieren Sie möglichst alle Warnungen des Compilers
- Benutzen Sie // für Kommentare (Ausnahme: Kommentar-Blöcke)
- Formatieren Sie Ihren Code sauber (Einrückungen!)

Empfehlungen 2

- Vermeiden Sie globale Variablen wann immer möglich

Coding Style - Referenzen

- <http://www.possibility.com/Cpp/CppCodingStandard.html>
- <http://www.cs.umd.edu/users/cml/cstyle/Elementel-rules.html>
- <http://www.cs.umd.edu/users/cml/cstyle/pikestyle.html>
- <http://www.joelinoff.com/ccdoc/index.html>
- <http://www.doc-o-matic.com>
- <http://www.stack.nl/~dimitri/doxygen/>

Dokumentationstools

Generieren übersichtliche Referenzbücher (HTML, CHM, LaTeX, RTF, ...) aus Kommentaren im Quelltext

- ccdoc
- doxyGen (unterstützt JavaDoc-Standard)
- doc-o-matic
- uvm.

Referenz

- www.by타민-c.com
Alles rund um C++Builder und C++ Programmierung. Tipps, Tools, Code-Ressourcen und Tutorials
- <http://community.borland.com/cpp/>
Updates für C++Builder
- <http://vclcomponents.com/>
Komponenten für VCL (C++B, Delphi)

Betriebssystemspezifische Programmierung

| | |
|---------------------------------|----------------|
| Anwendung | |
| OS-Spezifische Anwendungsmodule | |
| Bibliothek (VCL, GLIBC, MFC...) | |
| Betriebssystem - Schnittstelle | Window-Manager |
| | Grafikserver |
| Hardware Abstraction Layer | Treiber |
| BIOS | BIOS |
| Mainboard-Hardware | Peripherie |

GUI-Bibliotheken für C++

- Visual Components Library (Borland, Windows+Linux)
- Microsoft Foundation Classes (MS, Windows)
- QT Toolkit (Trolltech, Win+Linux)
- div. andere

VCL-GUI-Programmierung

- Forms (Klassen, werden zur Laufzeit instanziiert)
- Komponenten (Klassen)
- Für 'Forms' und die enthaltenen Objekte stehen Zeiger-Variablen zur Verfügung (vgl. 'name' Eigenschaft im Objektinspektor)

VCL-GUI (OO)

- Forms verarbeiten Ereignisse ('Events')
- Mitteilungen über Ereignisse via Methodenaufruf
- Ereignisse zB onClose, onClick, onCreate
- Mapping Event->Methode im 'Objektinspektor'

Projekte?

- bitte fertig bis Semesterbeginn...
- viel Erfolg!