9 Programmieren von Klassen

9.1 Einführung

Zentrales Konzept in der Objektorientierten Programmierung ist die Kapselung von Daten und Elementfunktionen, die sogenannte „Encapsulation“. In Objektorientierten Programmiersprachen können Daten und Elementfunktionen an die Objekte gebunden werden, für die sie zuständig sind. Der Programmierer legt genau fest, wie die Daten modifiziert werden können.

- Will man erreichen, dass auf die Daten oder Elementfunktionen eines Objekts zugegriffen werden kann, so müssen diese als `public` definiert werden.
- Will man erreichen, dass die Daten oder Elementfunktionen eines Objekts nach außen verborgen bleiben, so müssen diese als `private` definiert werden.

9.2 Deklaration von Klassen

Im Deklarationsteil von Klassen wird der Name der Klasse, die Objekte und die Prototypen der Elementfunktionen des Objekts festgelegt. Außerdem wird definiert, wie die Daten und Elementfunktionen nach außen freigegeben werden.

Beispiel 9.1: Klassendeklaration

```
// Deklarationsteil der myclass.hpp

class MyClass {
public:
    public functions and data MyClass

private:
    private functions and data MyClass
};
```

Diejenigen Elementfunktionen und Daten, die innerhalb einer derartigen Klassendeklaration deklariert werden, heißen Mitglieder (members) dieser Klasse: „Memberfunctions“ und „Member“.

Der Spezifizierung `private` kann auch weggelassen werden, da die Voreinstellung des Zugriffsrecht `private` ist. `private`-Members sind ausschließlich von den anderen Mitgliedern

dieser Klasse erreichbar.

Um Funktionen und Daten einer Klasse auch von anderen Programmteilen, die Objekte diese Klasse verwenden, erreichbar zu machen, müssen sie als `public` definiert werden.

Innerhalb des Deklarationsteils sollen nur die Deklaration der Member und der Memberfunktionen stehen.
Der Implementationsteil der Memberfunktionen, also der Programmcode, wird unterhalb des Deklarationsteils in der Deklarationsdatei hinzugefügt.

Beispiel 9.2: Implementationsteil

```
// Implementationsteil der myclass.hpp

ergebnistyp ClassName::function - name(parameter - list) {
    ... // body of function
}
```

Die Deklarationsdatei mit Implementationsteil soll in der Regel die Extension `hpp` tragen.

Hinweis: Es wäre auch möglich den Deklarationsteil und Implementationsteil in zwei Dateien auszulagern. Hierbei würde die Deklarationsdatei die Dateiendung `.h` bekommen und nur den Deklarationsteil der Klasse umfassen. Die Implementation der dort enthaltenen Memberfunktionen wird dann außerhalb der Deklarationsdatei in einer Implementationsdatei vorgenommen. Diese würde dann die Dateiendung `.cpp` bekommen. In der Implementationsdatei müsste dann noch die Deklarationsdatei am Anfang mittels `#include` Statement eingebunden werden. Der Übersicht halber wird auf diese Trennung hier allerdings verzichtet; deswegen enthält die Deklarationsdatei in unserem Beispiel schon den Implementationsteil.

Damit im Implementationsteil eindeutig definiert ist, zu welcher Klasse eine Memberfunktion gehört, muss dem Namen der Memberfunktion der Klassenname mit dem Resolutionsoperator `::` vorgestellt werden.

9.3 Deklaration von Klassenobjekten

Mit der Definition von Klassen wird lediglich die Datenstruktur im Speicher festgelegt, es wird aber noch kein Speicher für ein Objekt dieser Klasse zur Verfügung gestellt. Dies wird mit der Objektdeklaration erledigt, dem Objektnamen wird der Klassenname vorgestellt:

```
ClassName object_name;
```

Bei der Deklaration wird Speicher für dieses neue Objekt alloziiert und die `public` definierten Daten und Funktionen sind über den Objektnamen zugänglich.

9.4 Beispiel mit Klassen

Beispiel 9.3: Eine Klasse substantiv

```
// file: Klassenprogrammierung/substantiv.hpp
// description:

#include <iostream>

#ifndef SUBSTANTIV_HPP
#define SUBSTANTIV_HPP

class Substantiv {
public:
    void intoGrundform(std::string &vollform);
    void druckeLexikoneintrag();
private:
    bool genitiv(std::string &wort);
    std::string vollform;
    std::string grundform;
    std::string morphem;
};

void Substantiv::intoGrundform(std::string &wort) {
    vollform = wort;
    if (genitiv(wort))
        std::cout << " Genitiv entdeckt " << std::endl;
    else
        std::cout << " Endung nicht erkannt " << std::endl;
}

bool Substantiv::genitiv(std::string &wort) {
    int lastChar = wort.length() - 1;
    if (wort[lastChar - 1] == 'e' && wort[lastChar] == 's') {
        morphem = "es";
        grundform.assign(wort, 0, lastChar - 1);
        return true;
    }
    return false;
}

void Substantiv::druckeLexikoneintrag() {
    if (grundform != "")
        std::cout << " Grundform = " << grundform << std::endl;
    if (morphem != "")
        std::cout << " Morphem = " << morphem << std::endl;
    if (vollform != "")
        std::cout << " Vollform = " << vollform << std::endl;
}
#endif
```

```
// file: Klassenprogrammierung/mainSubstantiv.cpp
// description: Anwender der Substantiv Klasse

#include "substantiv.hpp"
#include <iostream>

using namespace std;

int main() {
    string wort;
    Substantiv subst;

    cout << " Hello, Programm mainSubstantiv.cpp " << endl;
    cout << " Genitiv eines Substantivs eingeben: ";
    cin >> wort;
    subst.intoGrundform(wort);
    subst.druckeLexikoneintrag();
    return 0;
}
```

Übersetzen des Beispiels:

```
g++ -o mainSubstantiv  mainSubstantiv.cpp
```

Aufgaben und Übungen zum Kapitel – Programmieren von Klassen

Übung 9.1

Schreiben Sie die Implementation der Klasse substantiv aus dem Skript ab und erweitern Sie die Klasse substantiv.

Schreiben Sie eine Deklarationsdatei mit Implementationsteil : `substantiv.hpp`

Schreiben Sie ein Hauptprogramm : `mainSubstantiv.cpp`

Übung 9.2

Wie lautet der Kompilierungsbefehl?

Übung 9.3

Es sollen folgende private Methoden zur Klasse substantiv hinzugefügt werden:

```
plural(string &wort)
```

Es soll die Pluralendung `'en'` erkannt werden.

Übung 9.4

`tolower()` Es soll der private Eintrag der Vollform in ein klein geschriebenes Wort konvertiert werden.

Übung 9.5

Die Methode `into_grundform` soll jeden Neueintrag automatisch mit der `private`-Methode `tolower()` in ein kleingeschriebenes Wort konvertieren und so unter `vollform` speichern.

Übung 9.6

Testen Sie das Programm mit den Wörtern „Hauses“ und „Sonnen“.

9.5 Initialisierung der Daten eines Objekts

Bei der Deklaration eines Objekts wird in C++ automatisch Speicher für die Daten des Objekts besorgt. Die Speicheralloziierung wird vom „Default-Konstruktor“ realisiert. Dieser Konstruktor besorgt nur Speicher, weist den Objektdaten aber keine Defaultwerte zu. Möchte der Programmierer bei der Deklaration den Objektdaten spezielle Werte zuweisen, muss er innerhalb der Klassendeklaration eine eigene Konstruktorfunktion definieren, die den Default-Konstruktor überschreibt.

In C++ können über Konstruktoren auch spezielle Initialisierungswerte für die Daten der Objekte definiert werden. Die Werte können bei der Deklaration der Objekte als Argumente übergeben werden. Anzahl und Art der Argumente entscheidet welche Konstruktorfunktion aufgerufen wird.

Beachte:

Da selbstdefinierte Konstruktoren die Initialisierungswerte der Objekte nicht verändern dürfen, müssen die Konstruktoren mit `const` Argumenten definiert werden.

Damit beim Aufruf der Konstruktorfunktion die Argumente nicht unnötig auf den STACK kopiert werden, empfiehlt es sich, die Argumente als Referenz zu definieren.

Beispiel 9.4: Verwendung von Konstruktorfunktionen

```
// file: Klassenprogrammierung/constructor.cpp
// description:

#include <iostream>
#include <string>

using namespace std;

int main() {
    string wort1; // Defaultkonstruktor
    string wort2("wie gehts"); // weiterer Konstruktor

    cout << " Hello, Programm Konstruktor " << endl;
    cout << " Default Konstruktor: " << wort1 << endl;
    cout << " Default Konstruktor: " << wort2 << endl;
    return 0;
}
```

Beispiel 9.5: Eigene Konstruktorfunktionen

```
// file: Klassenprogrammierung/constructorSelf.hpp
// description: Deklarationsdatei für die Klasse lexikon
```

```
#ifndef CONSTRUCTORSELF_HPP
#define CONSTRUCTORSELF_HPP

#include <iostream>

class Lexikon {
public:
    void druckeEinstellungen();
    Lexikon();
    Lexikon(const std::string &MyLand, const std::string &MyUmlaute);
    Lexikon(const std::string &MyLand);

private:
    std::string land;
    std::string umlaute;
};

Lexikon::Lexikon() {
    land = "garkeins";
    umlaute = "";
};

Lexikon::Lexikon(const std::string &MyLand, const std::string &MyUmlaute) {
    land = MyLand;
    umlaute = MyUmlaute;
};


Lexikon::Lexikon(const std::string &MyLand) {
    land = MyLand;
};

void Lexikon::druckeEinstellungen() {
    std::cout << " Einstellungen = " << std::endl;
    std::cout << " Land = " << land << std::endl;
    std::cout << " Umlaute = " << umlaute << std::endl;
};

#endif

// file: Klassenprogrammierung/mainConstructorSelf.cpp
// description: Anwender der Klasse lexikon

#include "constructorSelf.hpp"
#include <iostream>

using namespace std;

int main() {
    Lexikon meins;
    Lexikon englisch("englisch");
```

```
    Lexikon deutsch("deutsch", "äöü");

    cout << " Hello, mainConstructorSelf " << endl;
    cout << " Konstruktor " << endl;
    meins.druckeEinstellungen();

    cout << " Konstruktor " << endl;
    englisch.druckeEinstellungen();
    cout << " Konstruktor " << endl;
    deutsch.druckeEinstellungen();
    return 0;
}
```

9.6 Löschen der Daten eines Objekts

Wenn die Lebensdauer einer temporären Variable erlischt, wird sie automatisch gelöscht. Arbeitet man mit einer Variablen einer selbstdefinierten Klasse, dann wird beim Löschen der Variable in der Klassendefinition eine Destruktorfunktion gesucht und aufgerufen. Falls die Klasse keine eigene Destruktorfunktion definiert hat, wird ein sogenannter Defaultkonstruktor aufgerufen. Der Aufruf eines Defaultdestruktors kann zu Problemen bei komplexen Daten führen, da nur die Adressen der Variablen (=flache Memberdaten) gelöscht werden, aber nicht die Daten, auf die die Variable zeigt (=Tiefe Memberdaten). Diese tiefen Memberdaten bleiben als „Speicherleaks“ erhalten.

```
class Lexikon {
public:
    void druckeEinstellungen();
    Lexikon();
    Lexikon(const std::string &MyLand, const std::string &MyUmlaute);
    Lexikon(const std::string &MyLand);
    ~Lexikon();   // Destruktor der Klasse Lexikon

private:
    std::string land;
    std::string umlaute;
};
```

9.7 Kopieren der Daten eines Objekts

Wenn ein Objekt der Klasse zusammengesetzten privaten Speicher verwaltet, muss man bei der Kopie eines Objekts beachten, dass nur die Adressen der privaten Daten kopiert werden, nicht deren dahinterliegenden Daten. Es wird eine *flache Kopie* und keine *tiefe Kopie* des Objekt realisiert. Damit ein Objekt beim Kopieren auch ihre privaten Daten kopiert auf die sie zeigt, muss im `public` Teil der Klassendefinition eine sogenannte **Copy-Constructor** Funktion definiert werden. Die Copy-Constructor Funktion hat eine genau vorgeschriebene Syntax: Er hat keinen Return-Wert und hat als Argument eine `const` Referenz eines Klassenobjekts.

```
class Lexikon {
public:
    void druckeEinstellungen();
    Lexikon();
    Lexikon(const std::string &MyLand, const std::string &MyUmlaute);
    Lexikon(const std::string &MyLand);
    ~Lexikon();   // Destruktor der Klasse Lexikon
    Lexikon(const Lexikon &l) // Copy Constructor

private:
    std::string land;
    std::string umlaute;
};
```

9.8 Das Überladen von Operatoren

Mit dieser Eigenschaft können in C++ Operatoren im Bezug auf eine Klasse neu definiert werden, wobei bisherige Definitionen des Operators nicht geändert werden. Wenn ein Operator überladen werden soll, kann eine `member operator` Funktion oder `friend operator Funktion` relativ zu einer Klasse definiert werden. Werden Objekte solcher Klassen mit dem neu definierten Operator aufgerufen, wird die jeweilige Operator Funktion aufgerufen.

Die member-Operatorfunktion

unter `public` :

```
ClassName operator#(arg-list)
```

die Funktionsdefinition :

```
return-type ClassName::operator#(arg-list)
{
// Auszuführende Operation
}
```

Der `return-type` kann beliebig sein, die `arg-list` hängt von der Stelligkeit des Operators ab.

Einschränkungen:

- Die Präzedenz eines Operators kann nicht geändert werden
- Die Stelligkeit eines bereits definierten Operators kann nicht geändert werden
- Operatorfunktionen dürfen kein Default-Argument haben
- Folgende Operatoren können nicht überladen werden : . :: .* ?
- Präprozessoroperatoren können nicht überladen werden

9.9 Überladen von relationalen und logischen Operatoren

Die Operatoren werden wie bei binären Operatoren übergeben. Hier muss das Ergebnis der Operation ein ganzzahliger `true`- oder `false`-Wert sein.

Beispiel 9.6

```
// Vergleichsoperator

int StrType::operator==(const StrType &Str) {
    return Str.text == text; // Vergleich bei Strings ist definiert !
}
```

9.10 Überladen von unären Operatoren

Bei neuen Compilern wird bei der Operatordefinition unterschieden, ob der Operator als Prä- oder Postoperator vorliegt, und es kann jeweils eine andere Operation aufgerufen werden.

1. *Fall Präoperator:* `++s1`
   Hier wird die Operatorfunktion ohne Parameter definiert.

2. *Fall Postoperator:* `s1++`
   Zur Unterscheidung wird bei der Operatordefinition ein fiktives Argument angegeben.

Beispiel 9.7

```
////////////////////////////////////////////
//aus der  Deklarationsdatei:
StrType &StrType::operator++();
StrType &StrType::operator++(int notused);
////////////////////////////////////////////
```

```
//aus dem  Implementationsteil:

// Prae-Operator Left Operator

StrType &StrType::operator++() {
    int i;
    for (i = 0; i < anz_char; i++) {
        if (islower(text[i])) text[i] = toupper(text[i]);
        else
            text[i] = tolower(text[i]);
    }
}

// Post - Operator Right Operator

StrType &StrType::operator++(int notused) {
    int i;
    for (i = 0; i < anz_char; i++) {
        if (isupper(text[i])) text[i] = tolower(text[i]);
        else
            text[i] = toupper(text[i]);
    }
}
////////////////////////////////////////////
// aus dem Hauptprogramm:
StrType s1;

++s1;
s1++;
```

9.11 Überladen des Output-Operators

Da selbstdefinierte Klassen beliebige Daten in beliebigen Datenstrukturen zusammenfassen können, ist es selbstverständlich, dass Objekte einer Klasse nicht mit dem Standard-Output-Operator ausgegeben werden können. Um dies zu ermöglichen, muss man in der Klasse den Output-Operator |«| überladen.

Beispiel 9.8

```
ostream& operator<<(ostream& os, const StrType &str)
{
    os << "`Land="' << str.land << ' Umlaute sind:' << str.umlaute;
    return os;
}
```

Formatierungsvorschläge:

Klassennamen schreiben wir im UpperCamelCase, d.h. das gesamte Wort beginnt mit einen Großbuchstaben und auch jedes Teilwort. Dasselbe wollen wir für Konstanten, Structures,

Enumerations (Aufzählungen) und Typedefs verwenden. z.B.:

```
enum BackgroundColour  {
   Cyan,
   Magenta,
   Yellow
};

const int FixedWidth = 1;
```

Compound Types, also Klassen und structs oder Typedefs, die Objekte definieren, sollten als Namen ein Nomen bekommen. z.B.:

```
class Costumer{
 //..
};
```

(Collections bekommen einen Plural als Namen)

10 Vererbung

10.1 Einleitung

Vererbung (inheritance) ist ein Mechanismus, mit dem Klassen hierarchisch aufgebaut werden können. Aus einer Oberklasse werden Unterklassen abgeleitet. Die Unterklasse kann die Attribute und Methoden der Oberklasse übernehmen (also „erben“) und gleichzeitig eigene Eigenschaften definieren.
Eine Hierarchie von Klassen führt dadurch von der allgemeinsten Klasse hin zur spezifischsten. Diejenige Klasse, deren Eigenschaften vererbt werden, heißt Basisklasse `(base class)` oder Oberklasse. Diejenige Klasse, die diese Eigenschaften erbt, heißt abgeleitete Klasse `(derived class)` oder Unterklasse. Bei der Vererbung spricht man von einer „is-a“ Beziehung: Die Unterklasse „is-a“ Oberklasse.

Beispiel 10.1: Beziehung zwischen Ober- und Unterklasse

Oberklasse : Tier
Unterklasse: Esel
Jeder Esel ist ein Tier: ein Esel „is-a“ Tier. Jede Unterklasse erbt die Eigenschaften und Methoden der Oberklasse. In der Unterklasse werden nur die neu hinzukommenden Eigenschaften und Methoden definiert. Beispiel:
Klasse Tier: Ein Tier hat einen bestimmten Namen, Eigenheit, Geschlecht und kann einen Laut machen:

```
class Tier {
public:
    Tier(std::wstring tierName):name(tierName) {}

    void eigenheiten() {
        std::wcout << L"Mein Name ist: " << name << std::endl;
    }
    void macheLaut();
    void istEin();

private:
    std::wstring name;
    std::wstring geschlecht;

protected:
    std::wstring laut;
};
```