Objektorientierung

Sebastian Stabinger

SS2020

Strukturen

Strukturen - Nachteile

Sichtbarkeit

Es gibt keine Kontrolle wer auf Daten Zugriff hat: Jede Funktion kann auf alle Elemente der Struktur zugreifen. Das ist oft nicht erwünscht.

Initialisierung

Wir haben keine Möglichkeit eine Struktur automatisch zu initialisieren.

Hierarchien

Es gibt keine Möglichkeiten Abhängigkeiten zwischen Strukturen abzubilden. z.B. wird eine Struktur welche Komplexe Zahlen abbildet mehr mit einer Struktur zu tun haben welche Rationale Zahlen abbildet wie mit einer Struktur welche Personendaten abbildet. Solche Abhängigkeiten (welche Programmieraufwand sparen können) kann man mit Strukturen nicht abbilden.

Strukturen - Beispiel

Wir wollen einen Datentypen definieren welcher verwendet werden kann um den Durchschnitt mehrerer Werte auszurechnen.

Code

```
struct Average {
  double sum;
  int count;
  double avg;
};

void add(Average *avg, double val) {
    avg->sum += val;
    avg->count++;
    avg->avg = avg->sum / avg->count;
}
```

Strukturen - Komplettes Beispiel

```cpp
#include <iostream>
using namespace std;

struct Average {
  double sum;
  int count;
  double avg;
};

void add(Average *avg, double val) {
    avg->sum += val;
    avg->count++;
    avg->avg = avg->sum / avg->count;
}

int main() {
    Average avg = {0.0, 0, 0.0};
    add(&avg, 2.3);
    cout << avg.avg << endl;
    add(&avg, 323.2);
    add(&avg, 503.43);
    cout << avg.avg << endl;
}
// Ausgabe:
// 2.3
// 276.31
```

Strukturen - Nachteile an Hand des Beispiels

Sichtbarkeit

`sum` und `count` sind für den Benutzer nicht wichtig, aber man kann trotzdem drauf zugreifen.

Operationen

Bei Funktionen müssen wir immer einen Zeiger auf die `struct` übergeben um Werte davon ändern zu können. Zudem haben wir eine Funktion `add` bei der nicht sofort klar ist auf was sie sich bezieht.

Initialisierung

Wir möchten, dass `sum`, `count` und `avg` einer neuen Variable auf `0.0` gesetzt werden. Falls wir `sum` und `count` bereits haben hätten wir gerne, dass `avg` bei der Initialisierung berechnet wird.

Klassen

Von der Struktur zur Klasse

In C++ werden die vorher genannten Probleme von Strukturen mit Hilfe von Klassen gelöst. Eine objektorientierte Sprache muss keine Klassen haben, es ist aber sehr üblich.

- Definition von Sichtbarkeiten (z.B. `public`, `private`)
- Definition von Funktionen einer Klasse (Member Functions)
- Initialisierung mittels Konstruktoren
- Abhängigkeiten werden mittels Vererbung abgebildet

Syntax

```
class X { // X ist Name der Klasse
private:
  // Nur innerhalb der Klasse sichtbar

  // Enthält Sachen die nur für die
  // Implementierung intern benötigt werden
public:
  // Sichtbar für alle

  // Implementiert das sogenannte "Interface"
  // welches Benutzer der Klasse verwenden
};
```

`struct` von Vorher als Klasse

```
class Average {
public:
  double sum;
  int count;
  double avg;
};
```

Einschränkung der Sichtbarkeit

```
class Average {
private:
  double sum;
  int count;

public:
  double avg;
};
```

`Average avg; avg.sum = 100;` funktioniert nicht mehr.

Klassen - Member Functions

add-Funktion von Vorher

```
void add(Average *avg, double val) {
  avg->sum += val;
  avg->count++;
  avg->avg = avg->sum / avg->count;
}
```

- Die vorher definierte Funktion `add` wird nicht mehr funktionieren weil wir von außerhalb der Klasse keinen Zugriff auf `sum` und `count` haben.
- Dies löst man mit Hilfe von Funktionen welche innerhalb der Klasse definiert werden. Solche Funktionen bezeichnet man als Member Functions.

Klassen - Member Functions

```
class Average {
private:
  // Häufig lässt man private Variablen mit einem Unterstrich beginnen!
  double _avg;
  double _sum;
  int _count;

public:
  void add(double val) {
    _sum += val;
    _count++;
    _avg = _sum / _count;
  }

  double get_avg() { return _avg; }
};
```

Innerhalb einer Member Function haben wir Zugriff auf alle Elemente der Klasse (auch die, welche als `private` deklariert sind)

Klassen - Member Functions - Aufruf

Member Functions einer Klasse werden aufgerufen indem an den Variablennamen einer Klasse ein Punkt angehängt wird, gefolgt von dem Funktionsaufruf

z.B. für das vorherige Beispiel:

```
Average a;
a.add(12.4);
a.add(23.7);
cout << a.get_avg();
```

Klassen - Getter und Setter

- Um lesend und/oder schreibend auf private Variablen einer Klasse zugreifen zu können, müssen wir Member Functions verwenden.
- Man bezeichnet solche Funktionen üblicherweise als Getter– und Setter– Funktionen weil sie üblicherweise mit dem Zusatz `get_` bzw. `set_` anfangen.
- Über diese Funktionen lässt sich der Zugriff auf Variablen genauer steuern:
  - Nur eine `get_` Funktion: Wir können die Variable nur lesen
  - Nur eine `set_` Funktion: Wir können die Variable nur schreiben
- Welchen Vorteil haben wir mit einer privaten Variable und einer `get_` und `set_` Funktion (Warum nicht einfach `public`)?
  - Wir können prüfen ob geschriebene Werte gültig sind
  - Wir können Werte während des Lesens oder Schreibens konvertieren
  - ...

Klassen - Getter und Setter - Vorteil beim Avg-Beispiel

Wir können uns z.B. im Nachhinein entscheiden, dass der Durchschnitt erst ausgerechnet wird wenn er angefordert wird.
Code der die Klasse verwendet muss nicht geändert werden!

Beispiel

```
class Average {
private:
  // Häufig lässt man private Variablen mit einem Unterstrich beginnen!
  double _sum;
  int _count;

public:
  void add(double val) {
    _sum += val;
    _count++;
  }

  double get_avg() { return _sum / _count; }
};
```

Klassen - Initialisierung

- Die Initialisierung einer Klasse geschieht mittels eines sogenannten Konstruktors
- Konstruktoren sind Member Functions welche den gleichen Namen wie die Klasse und keinen Rückgabetyp haben
- Der Konstruktor wird ausgeführt, wenn man eine neue Instanz einer Klasse erzeugt. Also z.B. eine neue Variable dieser Klasse erzeugt.
- Der Konstruktor mit leerer Parameterliste wird als Standardkonstruktor (default constructor) bezeichnet und wird ausgeführt wenn man eine Variable der Klasse ohne Parameter anlegt. z.B. `Average a;`
- Man kann beliebig viele Konstruktoren definieren solange die Parameter unterschiedliche Typen haben (siehe Funktionsüberladung)

Klassen - Konstruktor Beispiel

```
class Average {
  // ...

public:
  // ...

  Average() { // Standardkonstruktor
    _sum = 0;
    _count = 0;
    _avg = 0;
  }

  Average(double sum, int count) { // Weiterer Konstruktor
    _sum = sum;
    _count = count;
    _avg = _sum / _count;
  }
  // ... get_avg() ...
};
// In main:
Average a;         // Standardkonstruktor
Average c(12, 6); // Zweiter Konstruktor (sum=12, count=6, avg=2)
cout << a.get_avg() << " " << c.get_avg() << endl;
a.add(5); a.add(12); c.add(12);
cout << a.get_avg() << " " << c.get_avg() << endl;
```

Übungen

Gemeinsame Übung

Wir implementieren gemeinsam eine Klasse für komplexe Zahlen

Complex Numbers

$$i = \sqrt{-1}$$

$$i^2 = -1$$

$$z = a + bi$$

$$(a+bi)+(c+di) = (a+c)+(b+d)i$$

$$(a+bi)-(c+di) = (a-c)+(b-d)i$$

$$(a+bi)(c+di) = (ac-bd)+(bc+ad)i$$

$$(a+bi)(a-bi) = a^2+b^2$$

$$|z| = |a+bi| = \sqrt{x^2+y^2}$$

$$\bar{z} = \overline{(a+bi)} = (a-bi)$$

DeMoivre's theorem

$$[r(\cos\theta + i\sin\theta)]^n = r^n(\cos n\theta + i\sin n\theta)$$

Aufgabe: Umschreiben des Spiels

- Schreiben Sie die Version des Spiels von der letzten Einheit (Speichern von Monstern in einem `vector`) so um, dass `Figure` keine Struktur sondern eine Klasse ist und wandeln sie alle nötigen Funktionen in Member Functions um.
- Überlegen Sie: Welche Variablen sollten `private`, welche `public` sein?
- Können Sie mehrere Konstruktoren sinnvoll einsetzen?