C++ Crashkurs Tag 2 und 3

Mentoring SoSe 2019

Felix Droop, Julia Mellert & Anton Kriese
Freie Universität Berlin

13. Juni 2019

Inhalt

Referenzen vs Pointer

- Was ist der Unterschied zwischen Referenzen und Pointern?

Referenzen vs Pointer

- Was ist der Unterschied zwischen Referenzen und Pointern?
- Referenz: neuer Name für ein bestehendes Objekt

Referenzen vs Pointer

- Was ist der Unterschied zwischen Referenzen und Pointern?
- Referenz: neuer Name für ein bestehendes Objekt
- Pointer: vollständig neues Objekt, Wert ist Speicheradresse

Referenzen vs Pointer

- Was ist der Unterschied zwischen Referenzen und Pointern?
- Referenz: neuer Name für ein bestehendes Objekt
- Pointer: vollständig neues Objekt, Wert ist Speicheradresse

```
1  int main() {
2      int zahl{3};
3
4      int & drei{zahl};
5
6      int * dort_ist_zahl{&zahl}; // <- & gibt Adresse vom Folgenden
7  }
```

Was ist passiert?

```
1  int main() {
2     int zahl{3};
3
4     int & drei{zahl};
5
6     int * dort_ist_zahl{&zahl};
7  }
```

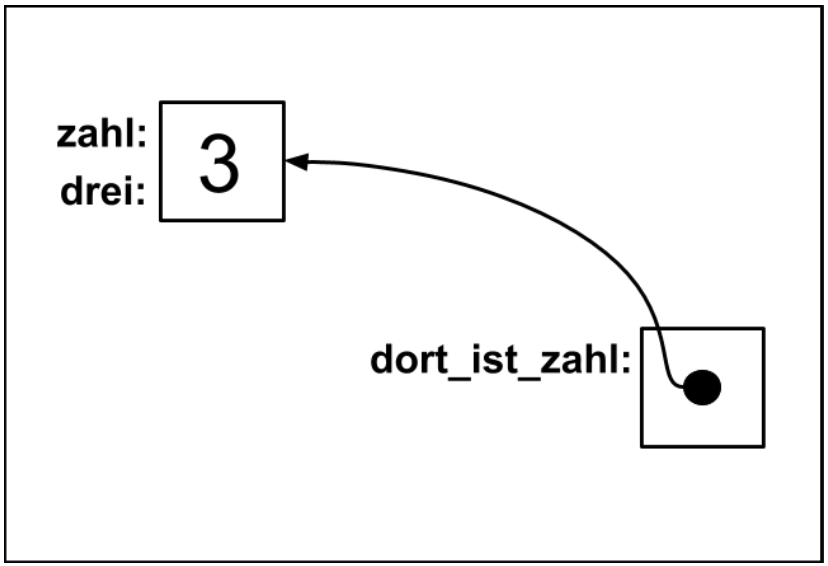

Abbildung: Speicherbild

Referenzen

- Selbst kein Objekt

Referenzen

- Selbst kein Objekt
- Wie ein weiterer Name für ein bestehendes Objekt

Referenzen

- Selbst kein Objekt
- Wie ein weiterer Name für ein bestehendes Objekt
- Verhält sich exakt genau so wie dieses

Referenzen

- Selbst kein Objekt
- Wie ein weiterer Name für ein bestehendes Objekt
- Verhält sich exakt genau so wie dieses
- Muss initialisiert werden

Referenzen

- Selbst kein Objekt
- Wie ein weiterer Name für ein bestehendes Objekt
- Verhält sich exakt genau so wie dieses
- Muss initialisiert werden
- Wichtigste Anwendung: Funktionsparameter

Referenzen

- Selbst kein Objekt
- Wie ein weiterer Name für ein bestehendes Objekt
- Verhält sich exakt genau so wie dieses
- Muss initialisiert werden
- Wichtigste Anwendung: Funktionsparameter

```
1  int main() {
2      int zahl{3};
3
4      int & drei{zahl};
5
6      drei = zahl + drei + zahl*drei;
7  }
```

Referenzen

- Selbst kein Objekt
- Wie ein weiterer Name für ein bestehendes Objekt
- Verhält sich exakt genau so wie dieses
- Muss initialisiert werden
- Wichtigste Anwendung: Funktionsparameter

```
1  int main() {
2      int zahl{3};
3
4      int & drei{zahl};
5
6      drei = zahl + drei + zahl*drei;
7  }
```

- Popquiz: Was ist zahl?

Referenzen

- Selbst kein Objekt
- Wie ein weiterer Name für ein bestehendes Objekt
- Verhält sich exakt genau so wie dieses
- Muss initialisiert werden
- Wichtigste Anwendung: Funktionsparameter

```
1  int main() {
2      int zahl{3};
3
4      int & drei{zahl};
5
6      drei = zahl + drei + zahl*drei;
7  }
```

- Popquiz: Was ist zahl?
- 15!

Pointer

- int * ist ein Datentyp wie int oder char

Pointer

- int * ist ein Datentyp wie int oder char
- Pointer ist selbst ein Objekt

Pointer

- int * ist ein Datentyp wie int oder char
- Pointer ist selbst ein Objekt
- Wert wird als Adresse interpretiert

Pointer

- int * ist ein Datentyp wie int oder char
- Pointer ist selbst ein Objekt
- Wert wird als Adresse interpretiert
- *-Operator zur Dereferenzierung

Pointer

- int * ist ein Datentyp wie int oder char
- Pointer ist selbst ein Objekt
- Wert wird als Adresse interpretiert
- *-Operator zur Dereferenzierung
- ==, <=, ++ z.B. auch möglich auf dem Pointer selbst

Pointer

- int * ist ein Datentyp wie int oder char
- Pointer ist selbst ein Objekt
- Wert wird als Adresse interpretiert
- *-Operator zur Dereferenzierung
- ==, <=, ++ z.B. auch möglich auf dem Pointer selbst

```
1  int main() {
2      int zahl{3};
3      int * dort_ist_zahl{&zahl};
4
5      *dort_ist_zahl = 15;
6      dort_ist_zahl += zahl;
7  }
```

- int * ist ein Datentyp wie int oder char
- Pointer ist selbst ein Objekt
- Wert wird als Adresse interpretiert
- *-Operator zur Dereferenzierung
- ==, <=, ++ z.B. auch möglich auf dem Pointer selbst

```
1  int main() {
2      int zahl{3};
3      int * dort_ist_zahl{&zahl};
4
5      *dort_ist_zahl = 15;
6      dort_ist_zahl += zahl;
7  }
```

- Popquiz Extreme: Welcher Wert steht in dort_ist_zahl (nicht der genaue Zahlenwert)?

Pointer

- int * ist ein Datentyp wie int oder char
- Pointer ist selbst ein Objekt
- Wert wird als Adresse interpretiert
- *-Operator zur Dereferenzierung
- ==, <=, ++ z.B. auch möglich auf dem Pointer selbst

```
1  int main() {
2      int zahl{3};
3      int * dort_ist_zahl{&zahl};
4
5      *dort_ist_zahl = 15;
6      dort_ist_zahl += zahl;
7  }
```

- Popquiz Extreme: Welcher Wert steht in dort_ist_zahl (nicht der genaue Zahlenwert)?
- Adresse von zahl + 15*4 in Byte (Nicht zuhause nachmachen!)

const

Das Schlüsselwort const kann verschiedene Bedeutungen haben:

- vor der Deklaration einer Variable
- bei der Übergabe einer Variable an eine Funktion
- nach einer Methodendeklaration

```
1  //der Wert von a kann sich nach der Deklaration nicht ändern
2  const int a = 3;

4  //der Wert von a kann innerhalb der Funktion nicht geändert werden
5  int square(const int a) { return a*a; }

7  //die Methode verändert nichts am Objekt
8  //falls non-const Methode auf const Objekt ausgeführt wird -> Error
9  std::string Student::getName() const { return name; }
```

Inhalt

Objektorientierte Programmierung

Objektorientierung beschreibt ein Programmierparadigma, welches die Abstrahierung von Daten und deren Verknüpfungen mittels Objekten ermöglicht.

- Vererbung
- Polymorphismus
- Abkapselung
- Persistenz

Begriffe:

- **Klasse**: Familie von Objekten, die gleiche Eigenschaften besitzen
- **Instanz**: Objekt einer Klasse mit indivuellen Werten
- **Membervariable**: Variable, die Teil einer Klasse ist
- **Methode**: Klasseneigene Funktion, die mit den Membern arbeitet

struct und class

- Konstrukte, die die Eigenschaften einer Familie von Objekten beschreiben
- funktional kaum Unterschiede zwischen struct und class
- Anwendung unterscheidet sich

```
1  struct Date {
2    //"Membervariablen"
3    int year{};
4    int month{};
5    int day{};
6    //"Member functions" (auch "methods")
7    void nextDay();
8    void print();
9  };
```

```
1  class Student {
2    public: //jeder kann hierauf zugreifen
3      std::string getName() const;
4      void changeName(const std::string &new_name);
5    private: //nur Methoden der Klasse selbst können hierauf zugreifen
6      std::string name{};
7  };
```

class

```
1   class Student {
2     public: //jeder kann hierrauf zugreifen
3       std::string getName() const;
4       void changeName(const std::string &new_name);
5       Date getBirthdate() const;
6       int getAge() const;
7       void exmatrikuliere();
8     private: //nur Methoden der Klasse selbst können hierrauf zugreifen
9       std::string name{};
10      const Date birthdate{};
11      const int matrikelnr{};
12      bool active{true};
13  };
```

- Bisher alles in eine .cpp-Datei geschrieben
- "good practice": Deklaration der Klasse -> .hpp, Definition -> .cpp

```
1  //Name der Datei: "Student.hpp"
2  #pragma once //sorgt dafür, dass das File nur einmal inkludiert wird
3
4  class Student {
5     public:
6        std::string getName() const;
7        void changeName(const std::string &new_name);
8        Student(const std::string &name);
9     private:
10       std::string name;
11 };
```

Dokumentstruktur

```cpp
//Name der Datei: "Student.cpp"
#include "Student.hpp"

std::string Student::getName() const {
   return name;
}
void Student::changeName(const std::string &newName){
   name = newName;
}
Student::Student(const std::string &nm){
   name = nm;
}
```

```cpp
//Name der Datei: "main.cpp" (Beispiel)
#include "Student.hpp"

int main(){
   Student knecht_ruprecht{};
}
```

Kompilieren: *g++ <compile flags> main.cpp Student.cpp -o ofile*

Inhalt

default constructor

Der *Standardkonstruktor* ist eine spezielle Memberfunktion ohne Rückgabewert, die dazu dient ein Objekt einer existierenden Klasse zu erstellen.

default constructor

Der *Standardkonstruktor* ist eine spezielle Memberfunktion ohne Rückgabewert, die dazu dient ein Objekt einer existierenden Klasse zu erstellen.

- erzeugt Objekt im Speicher
- falls kein Konstruktor vorhanden ist, wird automatisch einer vom System erzeugt

default constructor

Der *Standardkonstruktor* ist eine spezielle Memberfunktion ohne Rückgabewert, die dazu dient ein Objekt einer existierenden Klasse zu erstellen.

- erzeugt Objekt im Speicher
- falls kein Konstruktor vorhanden ist, wird automatisch einer vom System erzeugt

```
1   //in hpp
2   class Student{
3      public:
4         Student();
5         std::string name{};
6   }
7   //in cpp
8   Student::Student(){}
9   //oder mit Eigenschaften
10  Student::Student(){
11     name = "julia_mellert";
12  }
```

custom constructor

Der *custom constructor* ist auch ein Konstruktor. Im Gegensatz zu dem default constructor, kann er Parameter haben und überladen werden.

custom constructor

Der *custom constructor* ist auch ein Konstruktor. Im Gegensatz zu dem default constructor, kann er Parameter haben und überladen werden.

- sobald ein custom constructor existiert, wird nicht mehr automatisch ein Standardkonstruktor erzeugt

custom constructor

Der *custom constructor* ist auch ein Konstruktor. Im Gegensatz zu dem default constructor, kann er Parameter haben und überladen werden.

- sobald ein custom constructor existiert, wird nicht mehr automatisch ein Standardkonstruktor erzeugt

```
1   //hpp
2   class Student{
3      public:
4         Student(std::string n);
5      private:
6         std::string name;
7   }
8      //cpp
9   Student::Student(std::string n){
10     name = n;
11  }
12  //Bsp. in cpp in main
13  Student studi("julia_mellert");
```

Initialisierungsliste

Die *Initialisierungsliste* ist eine spezielle Form von Konstruktur, die dazu dient Membervariablen zu initialisieren. Insbesondere diejenigen, die const und damit unveränderbar sind.

Initialisierungsliste

Die *Initialisierungsliste* ist eine spezielle Form von Konstruktur, die dazu dient Membervariablen zu initialisieren. Insbesondere diejenigen, die const und damit unveränderbar sind.

```
1  //in hpp
2  class Student {
3     public:
4     //erstmal unwichtig
5     private:
6        std::string name{};
7        const Date birthdate{}; //hier mit const
8        const int matrikelnr{}; //hier mit const
9        bool active{true};
10 };
11 //in cpp
12 Student::Student() :
13    name{"julia_mellert"},
14    birthdate{1900,1,1}, /*aus struct*/
15    matrikelnr{5678912},
16    active{false}
17 {/*weiterer Code Nach Initialiserung der member*/}
```

Der copy constructor

Der *copy constructor* dient zur Iniatilisierung eines Objekts mit einem anderen. Das heißt, der gegebene Parameter ist eine Referenz auf ein Objekt mit derselben Klasse.

Der copy constructor

Der *copy constructor* dient zur Iniatilisierung eines Objekts mit einem anderen. Das heißt, der gegebene Parameter ist eine Referenz auf ein Objekt mit derselben Klasse.

- falls keiner vorhanden ist, wird er automatisch vom System erzeugt
- das Argumentobjekt soll nicht verändert werden, deswegen wird es mit const übergeben

Der copy constructor

Der *copy constructor* dient zur Iniatilisierung eines Objekts mit einem anderen. Das heißt, der gegebene Parameter ist eine Referenz auf ein Objekt mit derselben Klasse.

- falls keiner vorhanden ist, wird er automatisch vom System erzeugt
- das Argumentobjekt soll nicht verändert werden, deswegen wird es mit const übergeben

```
1   //copy constructor
2   Student(Student const & std){
3      name = std.name;
4   }
```

Inhalt

Wozu dient der *Destruktor*

Er dient dazu, nicht mehr benötigte Objekte zu löschen, und damit Speicherplatz wieder frei zugeben. Wird immer für jedes nicht globale Objekt am Ende seines Blockes aufgerufen.

Wozu dient der *Destruktor*

Er dient dazu, nicht mehr benötigte Objekte zu löschen, und damit Speicherplatz wieder frei zugeben. Wird immer für jedes nicht globale Objekt am Ende seines Blockes aufgerufen.

- keine Prameter oder Rückgabetypen
- falls kein Destruktor vorhanden ist, wird automatisch einer vom System erzeugt

Wozu dient der *Destruktor*

Er dient dazu, nicht mehr benötigte Objekte zu löschen, und damit Speicherplatz wieder frei zugeben. Wird immer für jedes nicht globale Objekt am Ende seines Blockes aufgerufen.

- keine Prameter oder Rückgabetypen
- falls kein Destruktor vorhanden ist, wird automatisch einer vom System erzeugt

```
1  class Student{
2      public:
3          ~Student();
4  }
```

Inhalt

Überladung

- Funktionen können den selben Namen haben, wenn sich die Parameter in Anzahl und/oder Typ unterschieden

- Funktionen können den selben Namen haben, wenn sich die Parameter in Anzahl und/oder Typ unterschieden

```
1  int add(int a, int b) { return a+b; }
2  double add(double a, double b) { return a+b; }
3  long add(long a, long b) { return a+b; }

5  int main() { return 0; } // nur genau eine main funktion!
```

Überladung

- Funktionen können den selben Namen haben, wenn sich die Parameter in Anzahl und/oder Typ unterschieden

```
1  int add(int a, int b) { return a+b; }
2  double add(double a, double b) { return a+b; }
3  long add(long a, long b) { return a+b; }

5  int main() { return 0; } // nur genau eine main funktion!
```

- Meinungs-Popquiz: Ist das schön?

Überladung

- Funktionen können den selben Namen haben, wenn sich die Parameter in Anzahl und/oder Typ unterschieden

```
1  int add(int a, int b) { return a+b; }
2  double add(double a, double b) { return a+b; }
3  long add(long a, long b) { return a+b; }

5  int main() { return 0; } // nur genau eine main funktion!
```

- Meinungs-Popquiz: Ist das schön?
- Nein.

Überladung

- Funktionen können den selben Namen haben, wenn sich die Parameter in Anzahl und/oder Typ unterschieden

```
1  int add(int a, int b) { return a+b; }
2  double add(double a, double b) { return a+b; }
3  long add(long a, long b) { return a+b; }

5  int main() { return 0; } // nur genau eine main funktion!
```

- Meinungs-Popquiz: Ist das schön?
- Nein.
- Lösung: Templates!

Template Funktionen

- Idee: Eine Funktionalität nur einmal für alle Datentypen schreiben

Template Funktionen

- Idee: Eine Funktionalität nur einmal für alle Datentypen schreiben
- die verschiedenen Funktionen generiert der Compiler

- Idee: Eine Funktionalität nur einmal für alle Datentypen schreiben
- die verschiedenen Funktionen generiert der Compiler

```
1  template <typename T>
2  T add(T a, T b) { return a+b; }

4  int main() {
5     int n{3}, m{15};
6     double d{3.3}, e{15.15};
7     add(n,m); // add für ints
8     add(d,e); // add für doubles
9  }
```

Template Klassen

- Selbe Idee: Die gleiche Klasse nur einmal für alle Datentypen schreiben

Template Klassen

- Selbe Idee: Die gleiche Klasse nur einmal für alle Datentypen schreiben
- die verschiedenen Klassen generiert wieder der Compiler

Template Klassen

- Selbe Idee: Die gleiche Klasse nur einmal für alle Datentypen schreiben
- die verschiedenen Klassen generiert wieder der Compiler
- Popquiz: Kennt jemand eine solche Klasse?

Template Klassen

- Selbe Idee: Die gleiche Klasse nur einmal für alle Datentypen schreiben
- die verschiedenen Klassen generiert wieder der Compiler
- Popquiz: Kennt jemand eine solche Klasse?
- std::vector!!

Template Klassen

- Selbe Idee: Die gleiche Klasse nur einmal für alle Datentypen schreiben
- die verschiedenen Klassen generiert wieder der Compiler
- Popquiz: Kennt jemand eine solche Klasse?
- std::vector!!

```
1   template <typename T>
2   class Gammelvector {
3      private:
4         T var;
5      public:
6         Gammelvector(T init) : var(init)
7         {}
8         T getVar() const {
9            return var;
10        }
11  };
```

Auslagern von Funktionsdefinitionen

```hpp
//.hpp:
template <typename T>
class Gammelvector {
   private:
      T var;
   public:
      Gammelvector(T init);
      T getVar();
};
```

```cpp
//.cpp:
template <typename T>
Gammelvector<T>::Gammelvector(T init) : var(init)
{}

template <typename T>
T Gammelvector<T>::getVar() {
   return var;
}

int main() {
   Gammelvector<char> g{'x'};
}
```