Informatik I – Übung 9

Pascal Schärli

pascscha@student.ethz.ch

03.05.2019

ETH zürich

Was gibts heute?

- Best-Of Vorlesung
  - Structs
  - Function overloading
  - Operator overloading
- Vorbesprechung

Best of Vorlesung

Structs

- Structs erlauben uns eigene "Datentypen" zu erstellen.
- Sie sind eine Kollektion von Variablen.
- Nach der Deklaration braucht es ein Semicolon;
- Es ist Konvention, dass Namen von Structs klein geschrieben sind und mit "_t" enden.

```
struct strange_t {
    int n;
    bool b;
    std::vector<int> a;
};
```

Structs

- Einen Struct kann man initialisieren in dem man alle Initialisierungswerte in geschwungenen {} klammern schreibt.
- Diese Werte werden dann in der entsprechenden Reihenfolge in die Variablen geschrieben
- Elemente können ausgelesen Werden mit "."

```
struct strange_t {
    int n;
    bool b;
    std::vector<int> a;
};


int main () {
    strange_t x = {1, false, {0,1,2}};

    std::cout << x.n    << std::endl; // Output: 1
    std::cout << x.b    << std::endl; // Output: 0
    std::cout << x.a[2] << std::endl; // Output: 2

    return 0;
}
```

Structs

Was ist der Output von diesem Programm?

```
2 1 0 1 1
```

```
#include <iostream>
#include <vector>

struct strange_t {
    int n;
    bool b;
    std::vector<int> a;
};

strange_t increase(strange_t x){
    strange_t out = x;
    out.n += 1;
    out.b = true;
    for(unsigned int i = 0; i < out.a.size(); i++){
        out.a[i] *= 1;
    }
    return out;
}

int main () {
    strange_t x = {1, false, {0,1,2}};

    x = increase(x);

    std::cout << x.n << " " << x.b << " "
              << x.a[0] << " " << x.a[1] << " " << x.a[1];

    return 0;
}
```

Geometry Exercise

In dieser Aufgabe Schauen wir uns dieser 3-D Vektor an. Skizziert folgende Funktionen:

1. Eine Funktion, welche zwei Vektoren addieren kann
2. Einen Struct "line_t", welcher eine Linie definiert
3. Eine Funktion welche eine Linie um einen Bestimmten Vektor verschieben kann.

```
struct vec_t {
    double x;
    double y;
    double z;
};

struct line_t {
    // TODO
};

// POST: returns the sum of two vectors
vec_t add(const vec_t& lhs, const vec_t& rhs) {

}

// POST: shifts a line by a vector
line_t shift_line(const line_t& line, const vec_t& amount
    // TODO
}
```

Geometry Exercise

```cpp
#include <iostream>

struct vec_t {
  double x;
  double y;
  double z;
};

struct line_t {
  vec_t start;
  vec_t end; // INV: start != end
};


// POST: returns the sum of two vectors
vec_t add(const vec_t& lhs, const vec_t& rhs) {
  vec_t out;
  out.x = lhs.x + rhs.x;
  out.y = lhs.y + rhs.y;
  out.z = lhs.z + rhs.z;
  return out;
}


// POST: returns a new line obtained by shifting l by v.
line_t shift_line (const line_t& line, const vec_t& amount) {
  line_t out;
  out.start = add(line.start, amount);
  out.end = add(line.end, amount);
  return out;
}
```

```cpp
int main () {
  vec a = {0.0, 0.0, 0.0};
  vec b = {1.0, 1.0, 1.0};
  line l = {a, b};
  vec c = {2.0, -1.0, 3.0};
  line new_l = shift_line(l, c);
  std::cout << "shifted line: ";
  std::cout << "(x:" << new_l.start.x;
  std::cout << ", y:" << new_l.start.y;
  std::cout << ", z:" << new_l.start.z << ") ";
  std::cout << "(x:" << new_l.end.x;
  std::cout << ", y:" << new_l.end.y;
  std::cout << ", z:" << new_l.end.z << ")" << std::endl;
}
```

Function Overloading

In C++ ist es möglich, dass mehrere Funktionen den gleichen Namen haben, solange sie unterschiedliche Übergabewerte haben.

```
int foo1(int a){ ... }
int foo1(int a, int b){ ... }
```

```
int foo2(int a){ ... }
int foo2(float a){ ... }
```

```
int foo3(int a){ ... }
int foo3(int b){ ... }
```

```
int foo4(int a){ ... }
float foo4(int a){ ... }
```

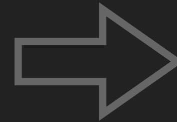
Function Overloading

```cpp
#include <iostream>

void out (const int i) {
    std::cout << i << " (int)\n";
}

void out (const double i) {
    std::cout << i << " (double)\n";
}

int main () {
    out(3.5);
    out(2);
    out(2.0);
    out(0);
    out(0.0);
    return 0;
}
```



```
3.5 (double)
2 (int)
2 (double)
0 (int)
0 (double)
```

Operator Overloading

Genau wie Funktionen können auch
Operatoren wie **+-*/** überschrieben werden.

```
struct vec_t {
    double x;
    double y;
    double z;
};

vec_t add(const vec_t& lhs, const vec_t& rhs) {
  vec_t out;
  out.x = lhs.x + rhs.x;
  out.y = lhs.y + rhs.y;
  out.z = lhs.z + rhs.z;
  return out;
}
```

```
struct vec_t {
    double x;
    double y;
    double z;
};

vec_t operator+(const vec_t& lhs, const vec_t& rhs){
  vec_t out;
  out.x = lhs.x + rhs.x;
  out.y = lhs.y + rhs.y;
  out.z = lhs.z + rhs.z;
  return out;
}
```

Operator Overloading

```
#include <iostream>

struct vec_t {
  double x;
  double y;
  double z;
};

// POST: returns the sum of two vectors
vec_t operator+(const vec_t& lhs, const vec_t& rhs) {
  vec_t out;
  out.x = lhs.x + rhs.x;
  out.y = lhs.y + rhs.y;
  out.z = lhs.z + rhs.z;
  return out;
}

std::ostream& operator<< (std::ostream &out, vec_t vec) {
    out << "(" << vec.x << ", " << vec.y << ", " << vec.z << ")";
    return out;
}

int main () {
  vec_t a = {1, 2, 1};
  vec_t b = {3, 3, 2};

  std::cout << a + b << std::endl;
}
```

Tribool

AND	false	unknown	true
false	*false*	*false*	*false*
unknown	*false*	*unknown*	*unknown*
true	*false*	*unknown*	*true*

OR	false	unknown	true
false	*false*	*unknown*	*true*
unknown	*unknown*	*unknown*	*true*
true	*true*	*true*	*true*

Tribool

```
int main(){
    tribool_t t1, t2;

    std::cout << "Please enter your first tribool: ";
    std::cin >> t1;

    std::cout << "Please enter your second tribool: ";
    std::cin >> t2;

    std::cout << t1 << " or " << t2 << " = " << (t1 || t2) << std::endl;
    std::cout << t1 << " and " << t2 << " = " << (t1 && t2) << std::endl;
}
```

Wir möchten einen "tribool" Struct erstellen.

Operator overloading soll uns helfen, dass dieser Wie normale Datentypen benutzt werden kann.

Tribool

```
1  struct tribool_t{
2      int value;
3  };
4
5  tribool_t tribool_false() {
6      tribool_t out = {0};
7      return out;
8  }
9
10 tribool_t tribool_unknown() {
11     tribool_t out = {1};
12     return out;
13 }
14
15 tribool_t tribool_true() {
16     tribool_t out = {2};
17     return out;
18 }
```

Tribool

```
1  #include <iostream>
2
3  struct tribool_t{
4      int value;
5  };
6
7  tribool_t tribool_false() {
8      tribool_t out = {0};
9      return out;
10 }
11
12 tribool_t tribool_unknown() {
13     tribool_t out = {1};
14     return out;
15 }
16
17 tribool_t tribool_true() {
18     tribool_t out = {2};
19     return out;
20 }
21
22 bool operator== (const tribool_t &lhs, const tribool_t &rhs) {
23   // TODO
24 }
25
26 tribool_t operator&& (const tribool_t &lhs, const tribool_t &rhs) {
27   // TODO
28 }
29
```

Tribool

```
bool operator== (const tribool_t &lhs, const tribool_t &rhs) {
  return lhs.value == rhs.value;
}
```

Tribool

```
tribool_t operator&& (const tribool_t &lhs, const tribool_t &rhs)
  if(lhs == tribool_false() || rhs == tribool_false()){
    return tribool_false();
  }
  else if(lhs == tribool_unknown() || rhs == tribool_unknown()){
    return tribool_unknown();
  }
  else{
    return tribool_true();
  }
}
```

Tribool

```
tribool_t operator|| (const tribool_t &lhs, const tribool_t &rhs)
  if(lhs == tribool_true() || rhs == tribool_true()){
    return tribool_true();
  }
  else if(lhs == tribool_unknown() || rhs == tribool_unknown()){
    return tribool_unknown();
  }
  else{
    return tribool_false();
  }
}
```

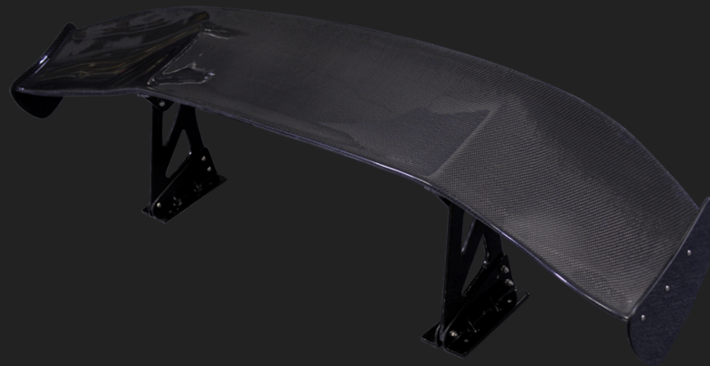
Tribool

```
std::istream& operator>> (std::istream& in, tribool_t& t){
  std::string value;
  in >> value;
  if(value == "false"){
    t = tribool_false();
  }
  else if(value == "true"){
    t = tribool_true();
  }
  else if(value == "unknown"){
    t = tribool_unknown();
  }
  return in;
}
```

Tribool

```cpp
std::ostream& operator<< (std::ostream &out, const tribool_t& t) {
  if (t == tribool_false()) {
    out << "false";
  } else if (t == tribool_unknown()) {
    out << "unknown";
  } else if(t == tribool_true()){
    out << "true";
  }
  else{
    out << "INVALID";
  }
  return out;
}
```

Vorbesprechung



Task 1: Finite Rings

Erstellt einen Struct, für eine Zahl, welche nur Werte von 0-6 annehmen kann.

1. *finite_ring.h* Invariante bestimmen (Was muss geten, damit der Struct korrekt ist)
2. *finite_ring.cpp* Implementiert die Addition, Subtraktion und Multiplikation

+	0	1	2	3	4	5	6
0	0	1	2	3	4	5	6
1	1	2	3	4	5	6	0
2	2	3	4	5	6	0	1
3	3	4	5	6	0	1	2
4	4	5	6	0	1	2	3
5	5	6	0	1	2	3	4
6	6	0	1	2	3	4	5

*	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6
2	0	2	4	6	1	3	5
3	0	3	6	2	5	1	4
4	0	4	1	5	2	6	3
5	0	5	3	1	6	4	2
6	0	6	5	4	3	2	1

Tipp: Modulo 7

Task 2a: Complex Numbers

Erstellt einen Struct, welcher Komplexe Zahlen darstellen kann.

1. *complex.h* Deklariert euren Struct und die Funktionen. (Wie in der Finite Rings Aufgabe schon gemacht wurde)
2. *complex.cpp* Implementiert Input-/Output Funktionen
3. *complex.cpp* Implementiert alle anderen Operatoren

Tipp:

```
std::istream& operator>>(std::istream& in, Complex& a)
```

```
std::ostream& operator<<(std::ostream& out, const Complex a)
```

Task 2b: Complex Calculator

Erstellt ein Programm, welche Komplexe Rechnungen lösen kann:

1. Kopiert eure Lösung von 2a
2. Kopiert den Inhalt von *calculator_double.txt* in *main.cpp*
3. Importiert euer *"complex.h"* File
4. Ersetzt double mit eurem Struct

Program of the Week

Viel Spass!