Inhaltsverzeichnis

[1. Autobau („wmb“) 2](#_Toc26146667)

[1.1. Lösungsidee 2](#_Toc26146668)

[1.2. Code 3](#_Toc26146669)

[1.3. Testfälle 10](#_Toc26146670)

[2. ADT Graph 12](#_Toc26146671)

[2.1. Lösungsidee 12](#_Toc26146672)

[2.2. Code 13](#_Toc26146673)

[2.3. Testfälle 18](#_Toc26146674)

# Autobau („wmb“)

## Lösungsidee

Es wurden 3 Klassen (car, tire, engine) implementiert. Diese Klassen haben den Zweck einen Autozusammenbau darzustellen. Die einzelnen Klassen beinhalten einige Informationen über das entsprechende Autoteil bzw. über das gesamte Auto. Die Klasse car stellt das gesamte zusammengebaute Auto dar. Car beinhaltet zwei zusätzlich zu einigen String und Integer Komponenten noch eine Datenkomponente vom Typ tire und eine Komponente vom typ engine.

Annahme: Grundsätzlich werden pro Auto nicht alle 4 Reifen extra gespeichert (in Form einer Datenstruktur) sondern lediglich ein gesamter Reifensatz.

Für jede Klasse wurde ein Konstruktor implementiert, der die jeweiligen Datenkomponenten mit den Übergabeparametern initialisiert. Wenn ein Objekt des der Klasse cars erstellt wird, so müssen auch zwei Objekte der Klassen tire und engine übergeben werden.

Für jede Klasse wurde der << Operator überschrieben, sodass eine Ausgabe aller Klassen mit *cout << Classname;* möglich ist.

Annahme: Da die Angabe meiner Ansicht nach etwas missverständlich formuliert wurde, war ich mir nicht zu 100% sicher, ob der >> Operator auch überschrieben werden musste.

Grundsätzlich wäre eine Lösung mit enums um einiges schöner, jedoch ist dies für die Ausgabe problematisch, da nur der „index“ der einzelnen Enums ausgegeben wird. Für eine korrekte Ausgabe müsste eine „Mapping“ Funktion geschrieben werde, die die enums in Strings umwandelt. Jedoch erschien mir dies für einen zu großen Overhead.

## Code

|  |
| --- |
| Car.h |
| *// // Created by Michael Neuhold on 28.11.19. //* #ifndef **ADT\_CAR\_H** #define **ADT\_CAR\_H** #include **<iostream>** #include **"./main.h"** #include **"./tire.h"** #include **"./engine.h"  using namespace** std;  **class** car {  **friend** std::ostream &**operator**<<(std::ostream &os,**const** car &finished\_car); **public**:  car(**const** string type, string color,**const int** serial\_number,**const** date\_t production\_date,**const** string production\_place,**const** string gearbox,**const** string type\_of\_drive,**int** top\_speed, **int** weight, tire &, engine &);  ~car() = **default**; **private**:  string type;  string color;  **int** serial\_number;  date\_t production\_date;  string production\_place;  string gearbox;  string type\_of\_drive;  **int** top\_speed;  **int** weight;  tire car\_tire;  engine car\_engine; };  #endif *//ADT\_CAR\_H* |

|  |
| --- |
| Car.cpp |
| *// // Created by Michael Neuhold on 28.11.19. //* #include **<iostream>** #include **"car.h"** #include **"./main.h"** #include **"./tire.h"  using namespace** std;  car::car(**const** string type, string color,**const int** serial\_number,  **const** date\_t production\_date,**const** string production\_place,  **const** string gearbox,**const** string type\_of\_drive, **int** top\_speed,  **int** weight, tire &car\_tire, engine &car\_engine) : car\_tire(car\_tire), car\_engine(car\_engine) {  **this** -> type = type;  **this** -> color = color;  **this** -> serial\_number = serial\_number;  **this** -> production\_date = production\_date;  **this** -> production\_place = production\_place;  **this** -> gearbox = gearbox;  **this** -> type\_of\_drive = type\_of\_drive;  **this** -> top\_speed = top\_speed;  **this** -> weight = weight;  **this** -> car\_tire = car\_tire;  **this** -> car\_engine = car\_engine; }  std::ostream &**operator**<<(std::ostream &os,**const** car &finished\_car) {  **return** os << **"car: {\n"  "\ttype: "** << finished\_car.type <<  **"\n\tcolor: "** << finished\_car.color <<  **"\n\tserial number: "** << finished\_car.serial\_number <<  **"\n\tproduction\_date: "** << finished\_car.production\_date.day <<  **"."** << finished\_car.production\_date.month <<  **"."** << finished\_car.production\_date.year <<  **"\n\tproduction\_place: "** << finished\_car.production\_place <<  **"\n\tgearbox: "** << finished\_car.gearbox <<  **"\n\ttype\_of\_drive: "** << finished\_car.type\_of\_drive <<  **"\n\ttop\_speed: "** << finished\_car.top\_speed <<  **"\n\tweight: "** << finished\_car.weight <<  **"\n\t"** << finished\_car.car\_tire <<  **"\t"** << finished\_car.car\_engine <<  **" }"** << std::endl; } |

|  |
| --- |
| Engine.h |
| *// // Created by Michael Neuhold on 28.11.19. //* #ifndef **ADT\_ENGINE\_H** #define **ADT\_ENGINE\_H** #include **<iostream>** #include **"./main.h"  class** engine {  **friend** std::ostream &**operator**<<(std::ostream &,**const** engine &); **public**:  engine(**const int** engine\_number,**const** string fuel\_type, **int** power, **double** n\_consumption,**const** date\_t production\_date);  ~engine() = **default**; **private**:  **int** engine\_number;  **int** power;  **double** n\_consumption;  string fuel\_type;  date\_t production\_date; };  #endif *//ADT\_ENGINE\_H* |

|  |
| --- |
| Engine.cpp |
| *// // Created by Michael Neuhold on 28.11.19. //* #include **"./main.h"** #include **"./engine.h"** engine::engine(**const int** engine\_number,**const** string fuel\_type, **int** power, **double** n\_consumption,**const** date\_t production\_date) {  **this** -> engine\_number = engine\_number;  **this** -> fuel\_type = fuel\_type;  **this** -> power = power;  **this** -> n\_consumption = n\_consumption;  **this** -> production\_date = production\_date; }  std::ostream &**operator**<<(std::ostream &os,**const** engine &car\_engine) {  **return** os << **"engine: { \n"  "\tengine\_number: "** << car\_engine.engine\_number <<  **"\n\tfuel\_type: "** << car\_engine.fuel\_type <<  **"\n\tpower: "** << car\_engine.power <<  **"\n\tn\_consuption: "** << car\_engine.n\_consumption <<  **"\n\tproduction\_date: "**<< car\_engine.production\_date.day <<  **"."** << car\_engine.production\_date.month <<  **"."** << car\_engine.production\_date.year <<  **"\n\t}"** << std::endl; } |

|  |
| --- |
| Tire.h |
| *// // Created by Michael Neuhold on 28.11.19. //* #ifndef **ADT\_TIRE\_H** #define **ADT\_TIRE\_H** #include **<iostream>** #include **"main.h"  using namespace** std;  **class** tire {  **friend** std::ostream &**operator**<<(std::ostream &os,**const** tire &car\_tire); **public**:  tire(**const double** rim\_diameter,**const int** production\_year,**const** string speed\_index,**const** string manufacturer);  ~tire() = **default**; **private**:  **double** rim\_diameter;  **int** procduction\_year;  string speed\_index;  string manufacturer; };  #endif *//ADT\_TIRE\_H* |

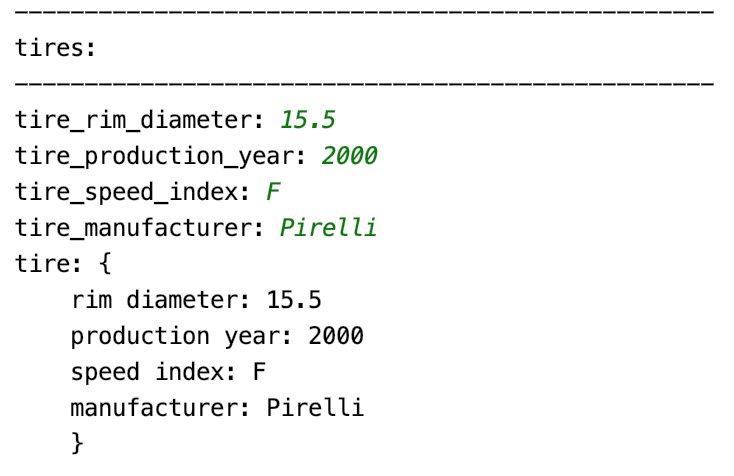
|  |
| --- |
| Tire.cpp |
| *// // Created by Michael Neuhold on 28.11.19. //* #include **"tire.h"** tire::tire(**const double** rim\_diameter,**const int** production\_year,**const** string speed\_index,**const** string manufacturer) {  **this** -> rim\_diameter = rim\_diameter;  **this** -> procduction\_year = production\_year;  **this** -> speed\_index = speed\_index;  **this** -> manufacturer = manufacturer; }  std::ostream &**operator**<<(std::ostream &os,**const** tire &car\_tire) {  **return** os << **"tire: {\n"  "\trim diameter: "** << car\_tire.rim\_diameter <<  **"\n\tproduction year: "** << car\_tire.procduction\_year <<  **"\n\tspeed index: "** << car\_tire.speed\_index <<  **"\n\tmanufacturer: "** << car\_tire.manufacturer <<  **"\n\t}"** << std::endl; } |

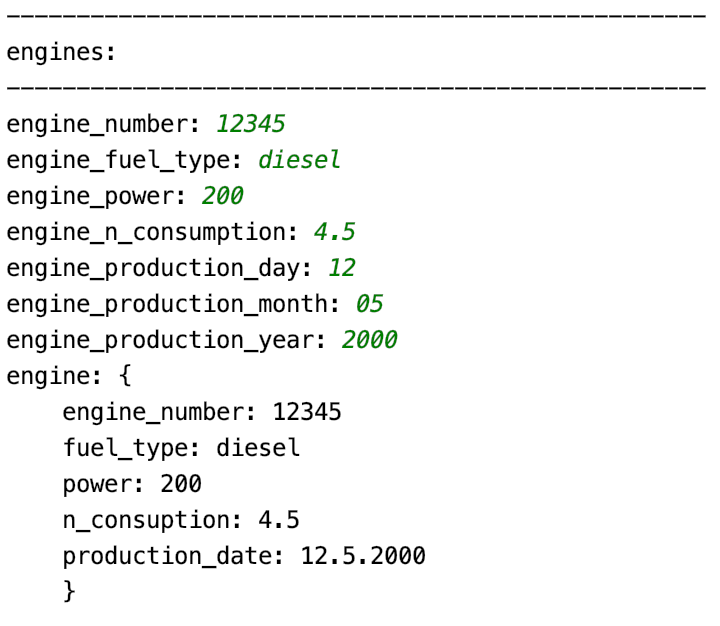
|  |
| --- |
| Main.h (eigentlich types.h) |
| *// // Created by Michael Neuhold on 28.11.19. //* #ifndef **ADT\_MAIN\_H** #define **ADT\_MAIN\_H** #include **<iostream>  using namespace** std;  *// date type* **typedef struct** {  **int** day;  **int** month;  **int** year; } date\_t;  *// color enum // enum color\_t { black , white , red , yellow , grey };  // speed index enum // enum speed\_index\_t { E , F , G , J , K , L , M , N , P , Q , R , S , T , U , H , V , W , Y };  // fuel type enum // enum fuel\_t { diesel , benzin , electric };* #endif *//ADT\_MAIN\_H* |

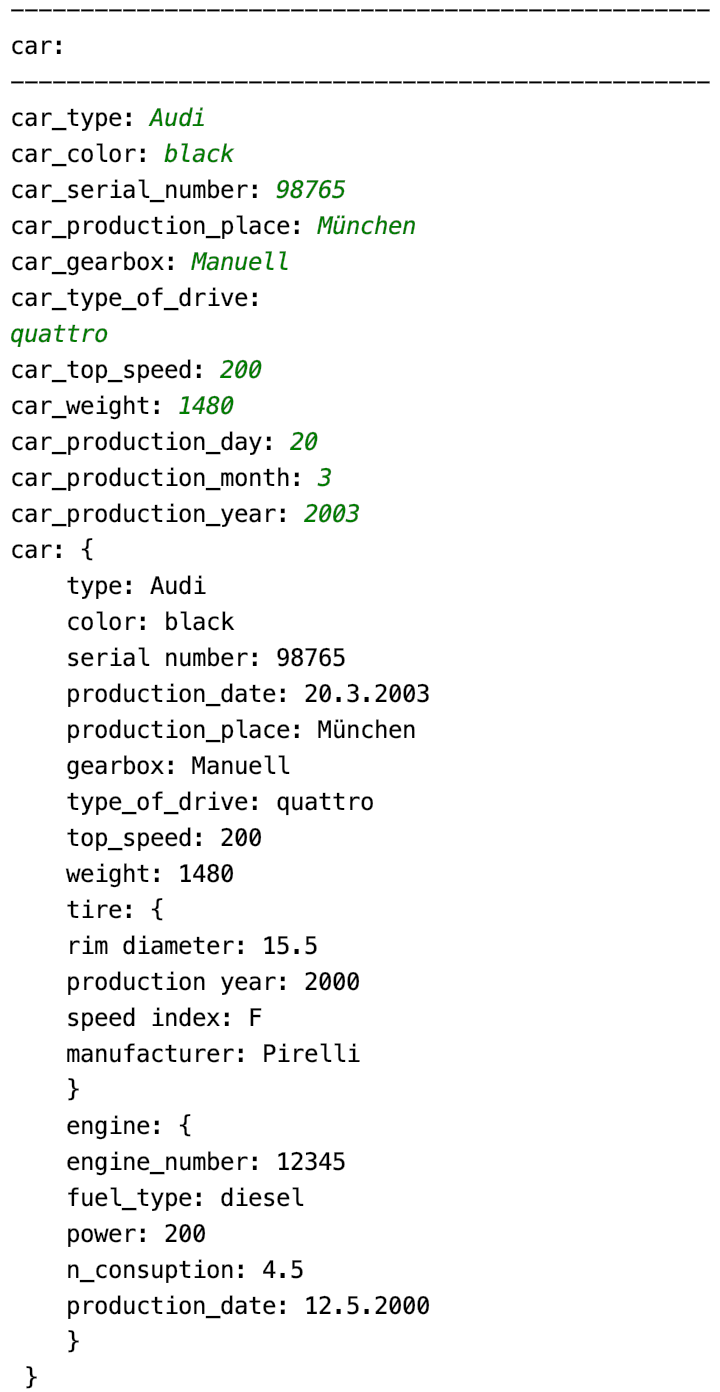
|  |
| --- |
| Main.cpp |
| #include **<iostream>** #include **"./main.h"** #include **"./tire.h"** #include **"./engine.h"** #include **"./car.h"  using namespace** std; s **void** Separator() {  **for**(**int** i = 0; i < 50; i++ ) {  cout << **"-"**;  }  cout << endl; } **void** Header(**const** string &header\_text) {  Separator();  cout << header\_text << endl;  Separator(); }  **int** main() {   Header(**"tires:"**);   */\*------------------------------------\*/  // input values* **double** tire\_rim\_diameter;  **int** tire\_production\_year;  string tire\_speed\_index;  string tire\_manufacturer;  */\*------------------------------------\*/* cout << **"tire\_rim\_diameter: "**;  cin >> tire\_rim\_diameter;  cout << **"tire\_production\_year: "**;  cin >> tire\_production\_year;  cout << **"tire\_speed\_index: "**;  cin >> tire\_speed\_index;  cout << **"tire\_manufacturer: "**;  cin >> tire\_manufacturer;  */\*------------------------------------\*/   // init new tries  // double rim\_diameter, int production\_year, string speed\_index, string manufacturer* tire t1(tire\_rim\_diameter, tire\_production\_year, tire\_speed\_index, tire\_manufacturer);  *// output of tires* cout << t1;   */\*--------------------------------------------------------------------------------------------\*/* Header(**"engines:"**);   */\*------------------------------------\*/  // input values* **int** engine\_number;  string engine\_fuel\_type;  **int** engine\_power;  **double** engine\_n\_consumption;  **int** engine\_production\_day;  **int** engine\_production\_month;  **int** engine\_production\_year;  */\*------------------------------------\*/* cout << **"engine\_number: "**;  cin >> engine\_number;  cout << **"engine\_fuel\_type: "**;  cin >> engine\_fuel\_type;  cout << **"engine\_power: "**;  cin >> engine\_power;  cout << **"engine\_n\_consumption: "**;  cin >> engine\_n\_consumption;  cout << **"engine\_production\_day: "**;  cin >> engine\_production\_day;  cout << **"engine\_production\_month: "**;  cin >> engine\_production\_month;  cout << **"engine\_production\_year: "**;  cin >> engine\_production\_year;  */\*------------------------------------\*/  // init new engines  // int engine\_number, string fuel\_type, int power, double n\_consumption, date\_t production\_date);* date\_t production\_date\_engine{ .day = engine\_production\_day , .month = engine\_production\_month , .year = engine\_production\_year };  engine e1(engine\_number,engine\_fuel\_type, engine\_power, engine\_n\_consumption, production\_date\_engine);  *// output of engines* cout << e1;   */\*--------------------------------------------------------------------------------------------\*/* Header(**"car:"**);   */\*------------------------------------\*/  // input values* string car\_type;  string car\_color;  **int** car\_serial\_number;  string car\_production\_place;  string car\_gearbox;  string car\_type\_of\_drive;  **int** car\_top\_speed;  **int** car\_weight;  **int** car\_production\_day;  **int** car\_production\_month;  **int** car\_production\_year;  */\*------------------------------------\*/* cout << **"car\_type: "**;  cin >> car\_type;  cout << **"car\_color: "**;  cin >> car\_color;  cout << **"car\_serial\_number: "**;  cin >> car\_serial\_number;  cout << **"car\_production\_place: "**;  cin >> car\_production\_place;  cout << **"car\_gearbox: "**;  cin >> car\_gearbox;  cout << **"car\_type\_of\_drive: "**;  cin >> car\_type\_of\_drive;  cout << **"car\_top\_speed: "**;  cin >> car\_top\_speed;  cout << **"car\_weight: "**;  cin >> car\_weight;  cout << **"car\_production\_day: "**;  cin >> car\_production\_day;  cout << **"car\_production\_month: "**;  cin >> car\_production\_month;  cout << **"car\_production\_year: "**;  cin >> car\_production\_year;  */\*------------------------------------\*/  // init new cars  // string type, string color, int serial\_number, date\_t production\_date, string production\_place, string gearbox, string type\_of\_drive, int top\_speed, int weight, tire &tire, engine &engine );* date\_t production\_date\_car{ .day = car\_production\_day , .month = car\_production\_month , .year = car\_production\_year };  car c1(car\_type,car\_color,car\_serial\_number,production\_date\_car,car\_production\_place,  car\_gearbox,car\_type\_of\_drive,car\_top\_speed,car\_weight,t1,e1);  *// output of cars* cout << c1;   */\*--------------------------------------------------------------------------------------------\*/* } |

## Testfälle

Bei diesem Testfall wird wie aus dem Main-File zu entnehmen alle Wert von der Konsole eingelesen und im Anschluss mit dem überladenen << Operator ausgegeben.







# ADT Graph

## Lösungsidee

Grundsätzlich wurde der Lösungsansatz von der Übung 5 übernommen. Graph und Vertex wurden als Klasse implementiert. Die Klasse Vertex beinhaltet lediglich eine Datenkomponente Payload und einen Getter für diese Datenkomponente. In der Klasse Graph stehen weiter Funktionen wie zum Beispiel das Hinzufügen von Vertex-Nodes und Edges bereit (wie in der Angabe beschreiben).

Um die eingefügten Vertex – Elemente eindeutig identifizieren zu können, wird nach dem Hinzufügen ein Objekt der Klasse handle\_t zurückgegeben. Handle\_t besitz eine Datenkomponente, welche beim Anlegen des Objektes mit dem Index des Feldes initialisiert wird, an dem sich der eingefügte Vertex befindet.

Es wurde eine Print Funktion implementiert, in der im Grunde „out“ wie „cout“ verwendet werden kann.

Der Dijkstra Shortest Path Algorithmus wurde rekursiv implementiert. Der Algorithmus kann für jeden angelegten Graphen angewandt werden. Voraussetzung für diese Implementierung ist, dass keine Vertex – Elemente gelöscht werden, da der Algorithmus mit den aufsteigenden Indexwerten arbeitet. Es wurde der rekursiven Funktion immer ein Hilfsfeld mit den bereits verwendeten Graph-Nodes mitgegeben (die nicht mehr besucht werden können). In der rekursiven Funktion wird dann immer das Feld „shortest path“ aktualisiert, wenn der Pfad zu einer Node kleiner ist als der bisher eingetragene. Diese Implementierung stellt einen Greedy Algorithmus dar.

## Code

|  |
| --- |
| Graph\_t.h |
| *// // Created by Michael Neuhold on 29.11.19. //* #ifndef **ADT\_GRAPH\_T\_H** #define **ADT\_GRAPH\_T\_H** #include **"./vertex\_t.h"** #include **"./handle\_t.h"  class** graph\_t {  **public**:   graph\_t() = **delete**;  graph\_t(std::string graph\_name);  ~graph\_t();  handle\_t add\_vertex(vertex\_t vertex);  **void** add\_edge(handle\_t **const** from, handle\_t **const** to, **int** weight);  **int** shortest\_path (handle\_t **const** from, handle\_t **const** to) **const**;  std::ostream & print (std::ostream & out) **const**;  **private**:  std::string graph\_name{**"test1"**};  **int** vertex\_count{0};  vertex\_t \*\*vertex\_list;  **int** \*\*matrix;  **void** dijkstra(**int** from, **int** \*vertex\_visited, **int** \*shortest\_path, **int** not\_visited\_count, **int** current\_path) **const**;  };  #endif *//ADT\_GRAPH\_T\_H* |

|  |
| --- |
| Graph\_t.cpp |
| *// // Created by Michael Neuhold on 29.11.19. //* #include **"graph\_t.h"** #include **"vertex\_t.h"** #include **"handle\_t.h"** */\*---------------------------------------------------------------------------------\*/* graph\_t::graph\_t(std::string graph\_name) {  *// graph name* **this** -> graph\_name = graph\_name;  *// allocate memory* **this** -> vertex\_list = **new** vertex\_t\*[**this** -> vertex\_count]; }  */\*---------------------------------------------------------------------------------\*/* handle\_t graph\_t::add\_vertex(vertex\_t vertex) {  vertex\_t \*v = **new** vertex\_t {vertex.get\_payload()};   **this** -> vertex\_count++;  **if**(**this** -> vertex\_count - 1 == 0) {  **this** -> vertex\_list = **new** vertex\_t\*[1];  **this** -> vertex\_list[**this** -> vertex\_count - 1] = v;   **this** -> matrix = **new int**\*[1];  **this** -> matrix[0] = **new int**[1];  **this** -> matrix[0][0] = 0;  } **else** {  **this** -> vertex\_list = (vertex\_t\*\*)realloc(**this** -> vertex\_list, **sizeof**(vertex\_t\*) \* **this** -> vertex\_count);  **this** -> vertex\_list[**this** -> vertex\_count - 1] = v;   **this** -> matrix = (**int**\*\*)realloc(**this** -> matrix, **sizeof**(**int**\*) \* **this** -> vertex\_count);  **for**(**int** i = 0; i < **this** -> vertex\_count - 1; i++) {  **this** -> matrix[i] = (**int**\*)realloc(**this** -> matrix[i], **sizeof**(**int**) \* **this** -> vertex\_count);  **this** -> matrix[i][vertex\_count - 1] = 0;  }  **this** -> matrix[vertex\_count - 1] = **new int**[vertex\_count];  **for**(**int** i = 0; i < vertex\_count; i++) {  **this** -> matrix[vertex\_count - 1][i] = 0;  }  }   handle\_t handler(vertex\_count - 1);  **return** handler; }  */\*---------------------------------------------------------------------------------\*/* std::ostream & graph\_t::print (std::ostream & out) **const** {  out << **this** -> graph\_name << **"\n"**;  out << **" "**;  **for**(**int** i = 0; i < **this** -> vertex\_count; i++){  out << **" "** << **this** -> vertex\_list[i] -> get\_payload();  }  out << std::endl;  **for**(**int** i = 0; i < **this** -> vertex\_count; i++) {  out << **this** -> vertex\_list[i] -> get\_payload() << **" "**;  **for**(**int** j = 0; j < **this** -> vertex\_count; j++) {  out << matrix[i][j] << **" "**;  }  out << **"\n"**;  }  **return** out; }  */\*---------------------------------------------------------------------------------\*/* **void** graph\_t::add\_edge(handle\_t **const** from, handle\_t **const** to, **int const** weight) {  **this** -> matrix[from.getIdentifier()][to.getIdentifier()] = weight; }  */\*---------------------------------------------------------------------------------\*/ // shortest path algorithm* **static bool** in\_visited(**int** i, **int**\* vertex\_visited, **int** n) {  **int** j = 0;  **while**(j < n) {  **if**(i == vertex\_visited[j]) {  **return true**;  }  j++;  }  **return false**; }  **void** graph\_t::dijkstra(**int** from, **int** \*vertex\_visited, **int** \*shortest\_path, **int** not\_visited\_count, **int** current\_path) **const** {  **if**(not\_visited\_count == 0) {  **return**;  }  **int** current\_shortest\_path = **INT\_MAX**;  **int** current\_shortest\_index = **INT\_MAX**;  **for**(**int** i = 0; i < **this** -> vertex\_count; i++) {  **if**(**this** -> matrix[from][i]) {  **if**(shortest\_path[i] > current\_path + **this** -> matrix[from][i]) {  shortest\_path[i] = current\_path + **this** -> matrix[from][i];  }  **if**(current\_shortest\_path > **this** -> matrix[from][i] && !in\_visited(i, vertex\_visited, **this** -> vertex\_count - not\_visited\_count)) {  current\_shortest\_path = **this** -> matrix[from][i];  current\_shortest\_index = i;  }  }  }  **if**(current\_shortest\_index != **INT\_MAX**) {  vertex\_visited[**this** -> vertex\_count - not\_visited\_count] = current\_shortest\_index;  dijkstra(current\_shortest\_index, vertex\_visited, shortest\_path, not\_visited\_count - 1, current\_path + current\_shortest\_path);  } **else** {  **return**;  } }  **int** graph\_t::shortest\_path (handle\_t **const** from, handle\_t **const** to) **const** {  **int** \*vertex\_vistited = **new int**[**this** -> vertex\_count];  **int** \*shortest\_path = **new int**[**this** -> vertex\_count];  **for**(**int** i = 0; i < **this** -> vertex\_count;i++) {  shortest\_path[i] = **INT\_MAX**;  }  shortest\_path[from.getIdentifier()] = 0;  **int** current\_path{0};  dijkstra(from.getIdentifier(),vertex\_vistited,shortest\_path, **this** -> vertex\_count, current\_path);   *// return to-entry from shortest path "table"* **return** shortest\_path[to.getIdentifier()]; }  */\*---------------------------------------------------------------------------------\*/* graph\_t::~graph\_t() {  *// free allocated memory* **for**(**int** i = 0; i < **this** -> vertex\_count; i++) {  **delete** [] **this** -> matrix[i];  }  **delete** [] **this** -> matrix;  **delete** [] **this** -> vertex\_list; }  */\*---------------------------------------------------------------------------------\*/* |

|  |
| --- |
| Handle\_t.h |
| *// // Created by Michael Neuhold on 01.12.19. //* #ifndef **ADT\_HANDLE\_T\_H** #define **ADT\_HANDLE\_T\_H   class** handle\_t {  **public**:  handle\_t(**int** index);  ~handle\_t() = **default**;  **int** getIdentifier() **const**;  **private**:  **int** identifier;  };  #endif *//ADT\_HANDLE\_T\_H* |

|  |
| --- |
| Handle\_t.cpp |
| *// // Created by Michael Neuhold on 01.12.19. //* #include **"handle\_t.h"** handle\_t::handle\_t(**int** index) {  **this** -> identifier = index; }  **int** handle\_t::getIdentifier() **const** {  **return** identifier; } |

|  |
| --- |
| Vertex\_t.h |
| *// // Created by Michael Neuhold on 29.11.19. //* #ifndef **ADT\_VERTEX\_T\_H** #define **ADT\_VERTEX\_T\_H** #include **<iostream>  class** vertex\_t { **public**:  vertex\_t() = **delete**;  vertex\_t(std::string payload);  ~vertex\_t() = **default**;  std::string get\_payload();  **private**:  std::string payload;  };  #endif *//ADT\_VERTEX\_T\_H* |

|  |
| --- |
| Vertex\_t.cpp |
| *// // Created by Michael Neuhold on 29.11.19. //* #include **"vertex\_t.h"** */\*---------------------------------------------------------------------------------\*/* vertex\_t::vertex\_t(std::string payload) {  **this** -> payload = payload; }  */\*---------------------------------------------------------------------------------\*/* std::string vertex\_t::get\_payload() {  **return this** -> payload; } |

|  |
| --- |
| Main.h |
| #include **<iostream>** #include **"./vertex\_t.h"** #include **"./graph\_t.h"  using namespace** std;  **void** Separator() {  **for** (**int** i = 0; i < 50; i++) {  cout << **"-"**;  }  cout << endl; }  **void** Header(**const** string &header\_text) {  Separator();  cout << header\_text << endl;  Separator(); }  **int** main() {  Header(**"graphs"**);   *// create new graph* graph\_t g1(**"Graph 1"**);  graph\_t g2(**"Graph 2"**);   *// create new vertex* vertex\_t v1(**"A"**);  vertex\_t v2(**"B"**);  vertex\_t v3(**"C"**);  vertex\_t v4(**"D"**);  vertex\_t v5(**"E"**);   vertex\_t v6(**"X"**);  vertex\_t v7(**"Y"**);  vertex\_t v8(**"Z"**);   *// add vertex to graph* handle\_t A = g1.add\_vertex(v1);  handle\_t B = g1.add\_vertex(v2);  handle\_t C = g1.add\_vertex(v3);  handle\_t D = g1.add\_vertex(v4);  handle\_t E = g1.add\_vertex(v5);   handle\_t X = g2.add\_vertex(v6);  handle\_t Y = g2.add\_vertex(v7);  handle\_t Z = g2.add\_vertex(v8);   *// print graph* g1.print(std::cout);  Separator();  g2.print(std::cout);  Separator();   *// add edges to graph* g1.add\_edge(A,B,6); *// A -> B* g1.add\_edge(A,D,1); *// A -> D* g1.add\_edge(D,B,2); *// D -> B* g1.add\_edge(D,E,1); *// D -> E* g1.add\_edge(E,C,5); *// E -> C* g1.add\_edge(E,B,2); *// B -> C* g1.add\_edge(B,C,5); *// E -> B* g2.add\_edge(X,Y,4);  g2.add\_edge(Y,Z,3);   *// print graph* g1.print(std::cout);  Separator();  g2.print(std::cout);  Separator();   *// shortest path* cout << **"g1 shortest path from A to C: "** << g1.shortest\_path(A,C) << endl;  cout << **"g1 shortest path from A to B: "** << g1.shortest\_path(A,B) << endl;   Separator();  } |

## Testfälle

