Projektarbeit C++ DHBW Stuttgart SoSe 2021

Verwaltung geplanter Auslieferungen für einen Quadrocopter

Für einen Quadrocopter zur Materialauslieferung soll die Wegestrecke berechnet werden. Grundlage für diese Berechnung ist eine Liste von Wegepunkten, die anzufliegen sind.

Vereinfachend wird angenommen, dass sich der Quadrocopter in einer Flughöhe bewegt, in der sich keine Kollisionsprobleme ergeben, sodass die direkte Strecke zwischen zwei Wegepunkten als Entfernung herangezogen werden kann. Ebenso werden die Aufwände zum Erreichen der Flughöhe und zum Landen an den Zielorten nicht in unsere Kalkulation eingerechnet.

In einem ersten Ansatz soll vorausgesetzt werden, dass die Wegepunkte in der Reihenfolge abgeflogen werden, in der sie in der intern zu verwaltenden Liste vorgegeben sind, sodass keine Wegeoptimierung durch Umsortieren der anzufliegenden Ziele gefordert ist.

Repräsentation eines Wegepunkts

Zur Realisierung eines Wegepunkts ist eine Klasse *WayPoint* zu deklarieren. Diese Klasse hält intern drei Informationen:

- Einen Namen für den Zielort
- Die geografische Breite des Zielorts
- Die geografische Länge des Zielorts

Den Namen für den Zielort typisieren wir aus der Standardbibliothek mit dem Typ *std::string*.

Geografische Breite und Länge typisieren wir mit *double*, die angenommene Einheit ist *Grad*. Wir verwenden aber zur Datenabstraktion die Klasse *std::pair* aus der Standardbibliothek und vereinfachen uns die Schreibweise der Typisierung durch eine *using* Direktive:

```
using LOCATION = std::pair<double, double>;
```

Für unsere Berechnungen werden wir die Zahl π benötigen. Erst mit dem Standard *C++20* hat diese Konstante den Weg in die Bibliothek gefunden! Wir ‚basteln' uns aus diesem Grund in einer akzeptablen Genauigkeit die Zahl π selbst als globale Konstante selbst (im Header *WayPoint.h*):

```
static const double M_PI = atan(1.0) * 4;
```

Somit enthält der Attributbereich der Klasse *WayPoint* weder Zeiger noch Referenzen. Wir haben also keine Probleme beim Kopieren oder Verschieben von Objekten der Klasse *WayPoint* und verwenden den gesamten Satz der vom Compiler generierten Versionen. Zur besseren Lesbarkeit geben wir das aber im Code mit dem Schlüsselwort *default* explizit an.

Durch einen speziellen Konstruktor, den wir zusätzlich kodieren, sind diese Attribute zu belegen! Dabei sind die Eingangsparameter für Länge und Breite zu überprüfen und ggfs. Exceptions (*std::out_of_*range) zu werfen:

- Der Betrag des ganzzahligen Anteils der Breite muss kleiner als 90 Grad sein
- Der Betrag des ganzzahligen Anteils der Länge muss kleiner 180 Grad sein

Dipl.-Ing. Thomas Marczinkowsky, Firma insystas
Kreuznacher Str. 47, 70372 Stuttgart, Tel. 0711-530 696 55, Fax 0711-530 696 56, mobil 0163-65 116 11
Thomas.Marczinkowsky@insystas.de

Zudem schreiben wir einen Satz von Getter-/Setter-Methoden für unsere Attribute:

- `latitude()/latitude(double)`
- `longitude()/longitude(double)`
- `name()/name(const std::string&)`

Zur Ausgabe eines Wegepunkts auf der Konsole überladen wir *operator*<<, der in C++ nur als Funktion überladbar ist.

Die Ausgabe der Koordinaten des Wegepunkts soll in der standardisierten Form *GMS* (Grad-Minuten-Sekunden) erfolgen.

Beispiel:

48°46‘33“N 9°10‘37“E

Zur Umrechnung und Formatierung implementieren wir dazu in der Klasse *WayPoint* eine statische Methode:

```
static std::string Degree2GMS(double latitude, double longitude);
```

Durch die Verwendung des Default Konstruktors der Klasse kann der Name eines Wegepunkts leer sein. In diesem Fall ist dieser als „<Empty>“ auf der Konsole auszugeben.

Die Klasse *WayPoint* soll uns zusätzlich die Entfernung zwischen zwei Wegepunkten berechnen können. Dazu brauchen wir zwei weitere Hilfsfunktionen:

```
static double radians(double arg);
static double distance(const WayPoint& from, const WayPoint& to);
```

Die Methode *distance* sei vorgegeben:

```
double WayPoint::distance(const WayPoint& from, const WayPoint& to)
{
   double lat = radians((from.latitude() + to.latitude()) / 2.0);
   double dx = 111.3 * std::cos(lat) * (from.longitude() - to.longitude());
   double dy = 111.3 * (from.latitude() - to.latitude());

   return std::sqrt(dx * dx + dy * dy);
}
```

Dipl.-Ing. Thomas Marczinkowsky, Firma insystas
Kreuznacher Str. 47, 70372 Stuttgart, Tel. 0711-530 696 55, Fax 0711-530 696 56, mobil 0163-65 116 11
Thomas.Marczinkowsky@insystas.de

Repräsentation der Wegeliste

Für die Wegeliste ist eine Klasse *WayPointContainer* zu deklarieren. Dazu soll der Container *std::array* der Standardbibliothek eingesetzt werden, der Zeiger auf Elemente der Klasse *WayPoint* halten soll.

Auch hier vereinfachen wir uns die Schreibweise des Typs durch eine *using* Direktive:

```
template <typename T, size_t n>
using Container = std::array<T, n>;
```

Die Größe des internen Arrays begrenzen wir auf 100 Elemente. Dazu deklarieren wir im privaten Bereich der Klasse *WayPointContainer* eine statische Konstante.

Das Array zur Aufnahme der Wegepunkte instanziieren wir direkt bei der Deklaration auf dem Heap und initialisieren die Elemente des Arrays mit *nullptr*.

Zudem brauchen wir ein Attribut, das die Anzahl der ins Array eingetragenen Wegepunkte mitzählt und beim Einbringen achten wir darauf, dass das Array dicht belegt wird (keine Lücken!), um uns die Garbage Collection im Destruktor einfach zu halten.

In unserem internen Array werden sich Heap Objekte vom Typ *WayPoint* befinden, die in einer noch im Folgenden zu definierenden Methode *add(.)* instanziiert werden. Aus diesem Grund müssen wir für das Kopieren von Objekten der Klasse *WayPointContainer* selbst Sorge tragen, indem wir den Copy Konstruktor und den Assignment Operator selbst schreiben!

Ebenso schreiben wir für die Klasse *WayPointContainer* eine Überladung des Index Operators, um mit den eckigen Klammern auf Elemente unseres internen Arrays zugreifen zu können. Bei der Implementierung müssen wir den Index prüfen und ggfs. eine entsprechende Ausnahme werfen.

Eine kleine zusätzliche Methode *count* gibt uns die Zahl der im Array befindlichen Wegepunkt zurück.

Die schon oben erwähnte Methode *add* hat die Signatur:

```
void add(const WayPoint& arg);
```

Um unsere Instanzen von *WayPointContainer* von der Lebenszeit der an der Schnittstelle übergebenen Instanz vom Typ *WayPoint* unabhängig zu halten, kreieren wir in der Implementierung ein neues Heap Objekt unter Verwendung des Copy Konstruktors von *WayPoint*. Zum Einschreiben in das interne Array verwenden wir die Methode *at* von *std::array*. Wir sparen uns so die Überprüfung des Array Index, da diese Methode die Überschreitung der Array Grenze selbst überwacht und ggfs. eine Ausnahme wirft.

Für unsere Klasse schreiben wir ebenso noch eine Methode *print*, die alle Informationen zur Instanz formatiert auf der Konsole ausgibt. Für die Wegepunkte können wir innerhalb der Methode unseren schon überladenen Operator << verwenden.

Folgende Informationen sind auf der Konsole auszugeben:

- Anzahl der Wegepunkte
- Index und Information jedes Wegepunkts des Arrays
- Die Gesamt-Distanz vom ersten Wegepunkt bis zum letzten Wegepunkt

Die Methode *print* erhält einen optionalen Parameter:

```
void print(bool contentFlag = true) const;
```

Die Methode *print* soll dabei, wenn das optionale Flag auf ‚false' gesetzt ist, anstatt dem Inhalt der Wegepunkte lediglich die Startadresse der Instanz des Wegepunkts ausgeben. Das kann uns eine Hilfe beim Debuggen unseres Codes sein.

Um die Gesamt-Distanz vom ersten Wegepunkt bis zum letzten Wegepunkt ausgeben zu können, schreiben wir schließlich noch eine Methode *distance*:

```
double distance() const;
```

Zur Berechnung bedienen wir uns dazu der schon in der Klasse *WayPoint* implementierten statischen Berechnungsmethode zur Distanz zweier Wegepunkte.

Abstrakte Klasse *QuadroCopterModel* zur Modellierung unterschiedlicher Typen

Um unterschiedliche Typen eines Quadrocopters realisieren zu können, designen wir uns nun eine abstrakte Klasse *QuadroCopterModel*.

Im Attributbereich, den wir ‚protected' deklarieren, weil wir von dieser Klasse ableiten und in der Child-Klasse auf diese Attribute zugreifen wollen, deklarieren wir die folgenden Attribute:

```
protected:
   std::string m_model;
   double m_range{ 0 };
   WayPointContainer m_Container;
```

Ebenso im *protected* Bereich deklarieren wir eine Methode *configure*. Diese Methode wollen wir polymorph in den Child Klassen überladen. Wir verzichten auf eine Standardimplementierung in der Basisklasse und deklarieren deshalb *rein virtuell*:

```
virtual void configure(const std::string& model, double range) = 0;
```

Unsere Modell-Klasse enthält weder Zeiger noch Referenzen und somit können wir alle vom Compiler zu generierenden Konstruktoren und Methoden als *default* übernehmen.

Um unsere gewünschte Funktionalität zu garantieren, fordern wir zudem von den Child Klassen die Implementierung der folgenden Methoden:

```
virtual void add(const WayPoint& arg) = 0;
virtual void print() const = 0;
virtual double distance() const = 0;
```

Unsere gesamte Klassen-Schnittstelle ist damit rein virtuell, wir benötigen also keinerlei Implementierung, weshalb es auch kein Modul *QuadroCopterModel.cpp* gibt.

Klasse für ‚reale' Quadrocopter

Wir haben nun alle Ingredienzien zusammengetragen und kommen zur letzten Klasse *QuadroCopter*.

Um die einzelnen Instanzen unserer Klasse *QuadroCopter* auch in der Ausgabe an der Konsole unterscheiden zu können, spendieren wir der Klasse ein Attribut für den Namen des Quadrocopters vom Typ *std::string*.

Jedem Quadrocopter ist ein Wegepunkt zugeordnet, von dem er die Auslieferung startet und zu dem er zur Aufladung zurückkehren muss. Das ergibt ein weiteres Attribut in unserer Klasse.

Wir erlauben keine Instanziierung über einen Default Konstruktor!

Stattdessen schreiben wir einen speziellen Konstruktor, dem übergeben wird:

- Der Name des Quadrokopters
- Den ‚Heimatort' des Quadrokopters (Head Quarter)
- Einen Namen für das Modell des Quadrocopters
- Die Reichweite des Quadrocopters

```
QuadroCopter(const std::string& name,
             const WayPoint& home,
             const std::string& model,
             double range);
```

Wir verhindern im Rahmen dieser Aufgabe das Ableiten von unserer Klasse *QuadroCopter*, werden die zu überschreibende Methode *configure* für späteren Ausbau aber *protected* qualifizieren.

```
class QuadroCopter final : public QuadroCopterModel
{…
```

Die restlichen Methoden qualifizieren wir *public*:

```
public:
   void add(const WayPoint& arg) override;
   double distance() const override;
   void print() const override;
```

Die Implementierung der Methode *add* ist einfach. Wir delegieren an den Container.

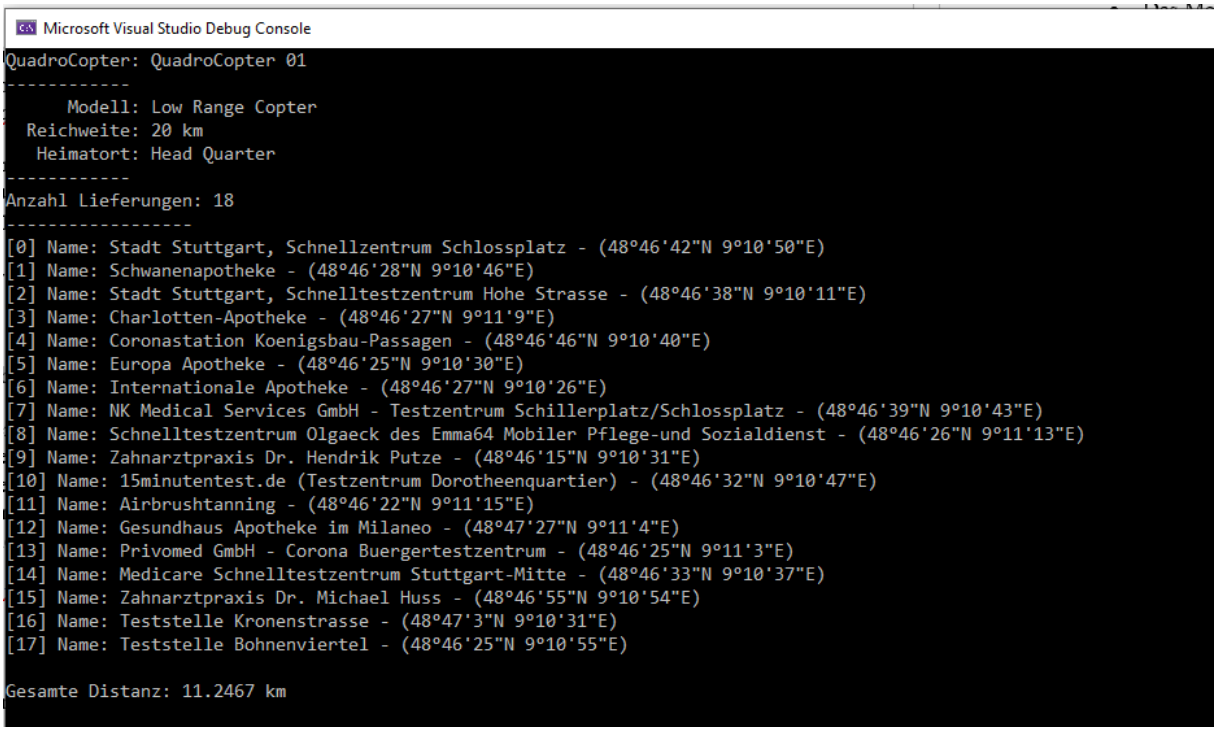
Die Methode *distance* ist auch überschaubar. Der Container liefert uns die gesamte Strecke der Wegepunkte. Wir müssen also nur noch den Weg vom Heimatort des Quadrocopters zum ersten Wegepunkt des Containers und die Strecke vom letzten Wegepunkt des Containers wieder zurück zum Heimatort dazurechnen.

Die Methode *print* gibt aus:

- Den Namen des Quadrocopters
- Das Modell des Quadrocopters
- Die Reichweite des Quadrocopters
- Den Heimatort des Quadrocopters

Den Rest der Ausgabe delegieren wir in der Methode an den Container.

Hier eine Ausgabe meiner Musterlösung als Beispiel:



Der Aufgabe beigestellt ist ein Modul *main.cpp*. Aber seien Sie bitte hier vorsichtig! Das Modul schafft Ihnen lediglich eine gewisse Grundbasis an Datenmaterial und eine Beispielausgabe. Die Testabdeckung ist minimal. Schreiben Sie also zur Sicherheit für jeden Teilschritt zusätzlich eigene Tests zum Prüfen der Funktionalität aller Klassen und sorgen Sie selbst für eine größtmögliche Testabdeckung.

Sollte Ihre Berechnung der Distanz von der obigen Beispielausgabe abweichen, so kann das mehrere Gründe haben. Die beiden wahrscheinlichsten sind wohl

- Ihre Lösung enthält noch Fehler
- Meine Musterlösung ist ‚bescheiden'! 😊

Beim Lösen der Aufgabe wünsche ich viel Spaß und Erfolg! Sollten Fragen auftreten, können Sie mich jederzeit über E-Mail kontaktieren!

Dipl.-Ing. Thomas Marczinkowsky, Firma insystas
Kreuznacher Str. 47, 70372 Stuttgart, Tel. 0711-530 696 55, Fax 0711-530 696 56, mobil 0163-65 116 11
Thomas.Marczinkowsky@insystas.de