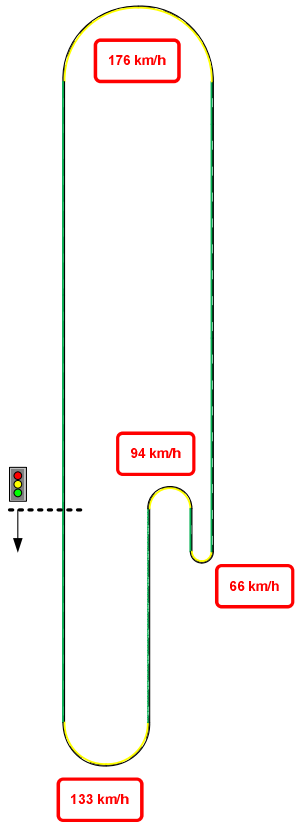
4. Praktische Übung in "Programmieren": Fahrdatenauswertung

Zur **AUFGABENSTELLUNG**

Ein Kfz wird mit einem intelligenten Tempomaten (Cruise-Control) ausgestattet. Zum Überprüfen der Funktionalität wird das Fahrverhalten auf einer Teststrecke protokolliert. Die Teststrecke wird durch eine geeignete Datei beschrieben (Streckenbeschreibungsdatei).
Es ist ein ANSI-C-Programm zu erstellen, das die protokollierten Daten auswertet. In den Kurven der Teststrecke darf eine definierte Maximalgeschwindigkeit nicht überschritten werden. Sollte dies der Fall sein, dann ist eine Alarmmeldung auszugeben.



VORGABEN

- Bei der **Streckenbeschreibungsdatei** handelt es sich um eine Textdatei, die in jeder gültigen Zeile jeweils einen **kritischen Streckenabschnitt (Kurve)** durch einen Datensatz beschreibt, der die folgenden drei Werte umfasst :
  - Entfernung des Kuvenbeginns von der Startlinie in **[m]**
  - Entfernung des Kurvenendes von der Startlinie in **[m]**
  - maximal erlaubte Geschwindigkeit in der Kurve in **[m/s]**

- Mit der Datei **`race_track.txt`** wird für Testzwecke eine geeignete Streckenbeschreibungsdatei bereitgestellt (Strecke s. Abb.)

- Programmintern wird ein Datensatz in jeweils einem Listenelement des nachfolgend definierten Structure-Typs **`TrackDescription`** abgelegt.

```
typedef
  struct track_descr_t
  { float begin_pos, end_pos;    /* Beginn- und Ende-Pos. des kritischen Streckenabschhnitts (Kurve) */
    float max_speed;             /* maximal erlaubte Geschwindigkeit in der Kurve */
    struct track_descr_t* next;  /* Verkettungspointer zum nächsten Listenelement */
  } TrackDescription;
```

- Die Definition dieses Structure-Typs ist in der vorgegebenen Headerdatei **`track_util.h`** zusammen mit den Deklarationen der nachfolgend beschriebenen Funktionen enthalten.

- Diese zur Bearbeitung der Streckenbeschreibungsdatei benötigten Funktionen sind bereits implementiert. Sie stehen in der vorgegebenen Quelldatei **`track_util.c`** zur Verfügung :

  ◇ **`FILE *openTrackFile(const char *dateiname);`**
  Öffnet die durch `dateiname` referierte Datei als Textdatei zum Lesen.
  Rückgabe : Filepointer der geöffneten Datei bei Erfolg
  bzw `NULL` im Fehlerfall

  ◇ **`int readTrackData(FILE *fp, TrackDescription *tdp);`**
  Liest die nächste gültige Zeile aus der durch `fp` referierten Streckenbeschreibungsdatei und legt den darin enthaltenen Datensatz in die durch `tdp` referierte Struktur ab. Überliest Kommentarzeilen (`'#'`).
  Rückgabe : `1` bei erfolgreichen Einlesen einer gültigen Zeile
  bzw `EOF` im Fehlerfall oder am Dateiende

  ◇ **`int closeTrackFile(FILE *fp);`**
  Schließt die durch `fp` referierte Datei.
  Rückgabe : 0 wenn Schließen erfolgreich war
  bzw `EOF` wenn `fp == NULL` ist oder Schließen nicht erfolgreich war

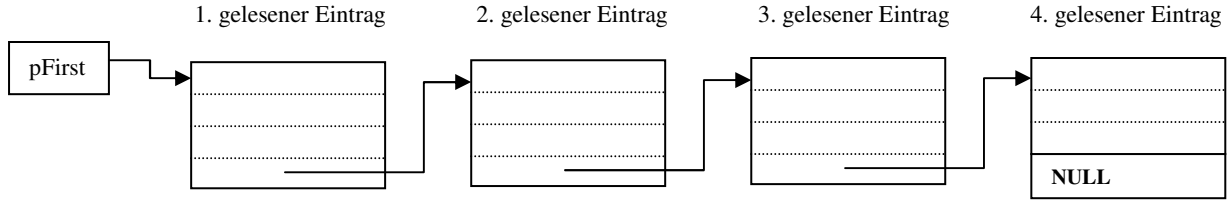
- Die ebenfalls zur Verfügung gestellte Datei **`fahrdata_m.c`** enthält die `main()`–Funktion für ein Testprogramm zur Aufgabe Teil A.

weiter Seite 2

Aufgabe Teil A:

- Im ersten Teil der Übung soll die **Streckenbeschreibungsdatei** eingelesen, die darin enthaltenen Information im Speicher in einer **einfach verketteten linearen Liste** abgelegt und anschließend in die Standardausgabe in geeigneter Form ausgegeben werden.

- Die einzelnen aus der Streckenbeschreibungsdatei eingelesenden Datensätze sollen dabei so in der Liste angeordnet werden, dass das Element mit den jeweils **neu eingelesenen** Daten an das **Listenende** angefügt wird (Beispiel für 4 Einträge s. nachstehende Abb.).



- Für die Lösung der Aufgabe werden die nachfolgend beschriebenen Funktionen benötigt.
  Implementieren Sie diese Funktionen in der Quelldatei **`track_control.c`** und **vereinbaren** Sie sie in der Headerdatei **`track_control.h`** :

  ◇ **`TrackDescription *copyTrackData(const TrackDescription *orig);`**
  Die Funktion alloziert einen Speicherbereich für ein `TrackDescription`-Element auf dem Heap. Sollte Speicher zur Verfügung stehen, kopiert sie die Informationen aus dem durch **`orig`** referierten Objekt in diesen Speicherbereich und gibt dessen Anfangs-Adresse als Funktionswert zurück.
  Sollte kein Speicher mehr zur Verfügung stehen, gibt die Funktion `NULL` zurück.

  ◇ **`void freeTrackData(TrackDescription *pFirst);`**
  Die Funktion gibt die allozierten Speicherbereiche aller `TrackDescription`-Elemente frei, die sich in der durch **`pFirst`** referierten verketteten Liste befinden.

  ◇ **`TrackDescription *handleTrack(const char *fname);`**
  Die Funktion öffnet die durch **`fname`** referierte Streckenbeschreibungsdatei, liest nacheinander alle darin enthaltenen Datensätze ein, legt diese im Speicher in der oben beschriebenen einfach verketteten Liste ab und schließt anschließend die Datei.
  Schlägt das Öffnen der Datei oder die Speicherallokation für ein neues Listenelement fehl, gibt die Funktion eine jeweils geeignete Fehlermeldung auf `stderr` aus.
  Im Falle eines Speicherallokations-Fehlschlags gibt sie den Speicher aller bis dahin bereits allozierten Listenelemente wieder frei.
  Rückgabe : Pointer auf den Listenanfang (Listenelement mit zuerst gelesenem Eintrag) bei Erfolg
  bzw `NULL` bei Misserfolg (Datei-Öffnungs- oder Speicherallokationsfehler)

  Sie **müssen** für die Implementierung die in `track_util.c` bereitgestellten Funktionen sowie die von Ihnen erstellten Funktionen `copyTrackData()` und `freeTrackData()` (s. oben) einsetzen.

  ◇ **`void outputTrackData(TrackDescription *pFirst);`**
  Die Funktion gibt Informationen aller kritischen Streckenabschnitte, die in der durch **`pFirst`** referierten verketteten Liste enthalten sind, in die Standardausgabe zeilenweise aus.
  Die Ausgabe für einen Streckenabschnitt entspricht folgendem Format:

```
Von Position  2749.80 m  bis  3299.60 m  ->  max. Geschw.  176.49 km/h
```

  (Beispiel der Ausgabe eines Eintrags aus der zur Verfügung gestellten Datei **`race_track.txt`**)

- Erstellen Sie unter Verwendung der Dateien **`track_util.c`** und **`fahrdata_m.c`** (enthält `main()`) ein **ablauffähiges Programm** und **testen** Sie damit Ihre erstellten Funktionen.

weiter Seite 3

Aufgabe Teil B:

- Im zweiten Teil der Übung soll das Programm um die Auswertung der Protokolldaten einer Testfahrt ergänzt werden.
  Die Auswertung erfolgt hinsichtlich der Überprüfung auf Einhaltung der in kritischen Streckenabschnitten (Kurven) festgelegten maximal zulässigen Geschwindigkeit.
  Wenn das Fahrzeug in einer Kurve die maximal erlaubte Geschwindigkeit überschritten hat, ist eine Alarmmeldung in die Standardausgabe auszugeben.

- Die Protokolldaten werden über die Standardeingabe als `float`-Wertepaare eingelesen.
  Der erste Wert stellt die aktuelle Position des Fahrzeugs auf der Teststrecke in **[m]** dar,
  der zweite Wert ist die aktuelle Geschwindigkeit des Fahrzeugs an der Stelle in **[m/s]**.
  Zur Erleichterung der Eingabe stehen die Protokolldaten von zwei Testfahrten in den Textdateien **`race_data_orig.txt`** und **`race_data_err_orig.txt`** zur Verfügung. Diese können mittels Umleitung der Standardeingabe (Programmaufruf aus der Kommandozeile !) eingelesen werden.

- Nach der Überprüfung einer Testfahrt sind zur weiteren Auswertung die Anzahl der Alarmmeldungen, die Gesamtfahrzeit (in **[s]**) und die Durchschnittsgeschwindigkeit der gesamten Fahrt (in **[m/s]**) von Interesse.
  Zur zusammenfassenden Darstellung dieser Testfahrt-Ergebnisse dient der nachfolgend definierte Structure-Typ **`RaceResults`** :

```
typedef
  struct
  { int alerts;             /* Anzahl der Alarmmeldungen */
    float time;             /* Gesamtfahrzeit in Vorwärtsfahrt */
    float average_speed;    /* errechnete Durchschnittsgeschwindigkeit */
  } RaceResults;
```

- Zur oben beschriebenen Auswertung einer Testfahrt dient die Funktion

  `RaceResults doRace(TrackDescription *pFirst)`;

- **Ergänzen** Sie die **Headerdatei** **`track_control.h`** um die **Deklaration** dieser **Funktion** sowie die **Definition** des Typs **`RaceResults`**.

- **Implementieren** Sie die Funktion **`doRace()`** in der Quelldatei **`track_control.c`** gemäß den nachfolgenden Vorgaben:
  - ▻ Zwei `float`-Werte sind ständig von der Standardeingabe zu lesen. Wenn keine zwei `float`-Werte mehr gelesen werden können oder wenn die neue Position des Fahrzeug kleiner als die vorhergehende Position ist (Rückwärtsfahrt !), ist die Einleseschleife zu beenden; andernfalls ist die verkettete Liste nach einem passenden Eintrag, in den die Position (1. Wert) fällt, zu durchsuchen.
  - ▻ Wird ein Eintrag gefunden, dann ist die max. zulässige Geschwindigkeit für den Streckenabschnitt mit der über die Standardeingabe eingelesenen Geschwindigkeit (2. Wert) zu vergleichen.
  - ▻ Sollte die maximale Geschwindigkeit überschritten worden sein, dann ist eine Alarmmeldung in die Standardausgabe auszugeben, in der die aktuelle Position in **[m]**, die aktuelle Geschwindigkeit in [**km/h**] und die max. zulässige Geschwindigkeit in [**km/h]** enthalten ist.
    Zusätzlich muss die entsprechende Komponente des Rückgabeobjekts erhöht werden.
  - ▻ Sollte das Fahrzeug rückwärts gefahren sein, dann ist eine Fehlermeldung in die Standardausgabe auszugeben, die die Position enthält, an der die Rückwärtsfahrt entdeckt wurde, und die Anzahl der bisher aufgetretenen Alarmmeldungen im Rückgabeobjekt zu invertieren (auf einen negativen Wert zu setzen).
  - ▻ Die Durchschnittsgeschwindigkeit der gesamten Testfahrt ist zu ermitteln, unter der Voraussetzung, dass die Geschwindigkeit von der alten Position des Fahrzeugs zur neuen Position immer durch **konstante Beschleunigung/Verzögerung** erreicht wird.

- **Testen** Sie Ihre Funktionsimplementierung, in dem Sie **`main()`** in **`fahrdata_m.c`** um den Aufruf von `doRace()` und die Ausgabe der im Rückgabe-Objekt enthaltenen Werte in die Standardausgabe **ergänzen**.

- **Diskutieren** Sie mit Ihren Kommilitonen und Ihrem Betreuer, ob die folgende Funktionssignatur für die Funktion `doRace()` nicht geeigneter wäre:

  `void doRace(RaceResults *, const TrackDescription *);`