

Einführung in „fligh_plan.C“

Aufgabe 6.4 (Flugplan) [6 Punkte]

In der Datei U6A4 direct flights.txt steht in jeder Zeile eine direkte Flugverbindung zwischen zwei Flughäfen sowie die Start- und Landezeiten. Alle diese Flüge finden innerhalb eines Tages statt. Ihre Aufgabe ist es nun auch alle indirekten Verbindungen mit maximal *drei Zwischenlandungen* innerhalb eines Tages zu berechnen. Sie können davon ausgehen, dass es immer nur eine direkte Verbindung an diesem Tag zwischen zwei Flughäfen gibt. In dieser Aufgabe dürfen Sie nur assoziative Container (set,multiset,map,multimap) verwenden.

Allerdings kann es bei dieser Iteration passieren, dass Flughäfen wiederholt innerhalb einer Flugverbindung vorkommen. Das müssen Sie natürlich verhindern. Ausserdem müssen alle indirekten Verbindungen die zeitlichen Zwänge erfüllen. D.h. jede Zwischenlandung muss zeitlich vor dem anschlussflug stattfinden. Nehmen Sie an, dass Umsteigen keine Zeit kostet!

Die Ausgabe ihres Programms U6A4 flight_plan.C soll für jede direkte/indirekte Verbindung (in alphabetischer Reihenfolge) den Namen des Start- und Zielflughafens, die Start- und Landezeit und in Klammern die Reihenfolge der benötigten Flughäfen (im Drei-Buchstabencode) enthalten. Die Zuordnung der Airportcodes und Namen erhalten sie durch Einlesen der Datei U6A4 airports.txt. Exemplarisch für Amsterdam sollte ihre Ausgabe dann etwa so aussehen:

```
>./U6A4_flight_plan U6A4_direct_flights.txt U6A4_airports.txt
Amsterdam Rome 9:00 11:15 (AMS ROM)
Amsterdam Frankfurt 9:00 13:45 (AMS ROM FRA)
```

Zunächst einmal betrachten wir uns die Dateien mit den Informationen genauer:

U6A4_airports.txt:

```
PAR Paris
FRA Frankfurt
NYC New York
LON London
ROM Rome
AMS Amsterdam
VIE Vienna
CHI Chicago
CAI Cairo
TUN Tunis
SIN Singapore
YUL Montreal
BJS Peking
MOW Moscow
```

U6A4_direct_flights.txt

```
PAR FRA 11:00 12:20
PAR LON 14:20 15:00
FRA NYC 13:00 16:10
FRA AMS 7:20 8:30
FRA ROM 9:30 11:15
FRA SIN 6:15 18:25
FRA VIE 13:30 14:30
VIE MOW 15:55 20:00
NYC CHI 16:45 18:15
CHI YUL 18:40 21:45
SIN BJS 19:00 23:30
AMS ROM 9:00 11:15
ROM FRA 12:00 13:45
ROM CAI 11:00 14:00
CAI TUN 14:10 15:20
```

Das wichtigste zu begin ist eine korrekte Datenstruktur aufzubauen mit der es sich leicht arbeiten lässt, aufmerksame Leser der Dateien erkennen zunächst zwei Probleme die es hierbei geben könnte:

- i. New York sind zwei Worte
- ii. Die Uhrzeiten lassen sich nicht als Zahl lesen wegen dem :
- iii. Manche Uhrzeiten sind < 10:00 sind somit kürzer als fünf Zeichen (dazu später mehr)

Das macht aber an sich gar nicht so viel aus. Folgend der Code unseres Programm Grundgerüsts:

```
//flight_plan.C

using std::string;

//Multi Map für landings: flugnummer | ( flughafen, zeit) :
typedef std::multimap<int, std::pair<string, string> > Flightplan_L;
//Multi Map für take_offs: flughafen | ( flugnummer, zeit) :
typedef std::multimap<string, std::pair<int, string> > Flightplan_TO;
// Map für Flughäfenkürzel: kürzel | name
typedef std::map<string, string> Airport;
//Flight; Trippel für start oder landung: flughafen | (flugnummer, zeit)
typedef std::pair<string, std::pair<int, string> > Flight;

Airport airports;
Flightplan_TO take_off;
Flightplan_L landing;

int main()
{
}
}
```

Was wurde hier gemacht?

Es wurden vier **typedef** gesetzt um vier doch recht lange Codezeilen abzukürzen. Diese **typedef** stehen für wichtige Elemente unserer Datenstruktur.

Flightplan_L soll später unsere Landezeiten und Zielflughäfen beinhalten und zwar in folgender Form:
(schlüssel :: wert1 , wert2)

```
1 FRA 12:20
2 LON 15:00
3 NYC 16:10
...
```

Flightplan_TO soll später unsere Take-Off-Zeiten und Startflughäfen beinhalten und zwar in folgender Form: (schlüssel :: wert1 , wert2)

```
PAR 1 11:00
PAR 2 14:20
FRA 3 13:00
...
```

Was soll das ganze?

Das erste Ziel ist es, den Schlüssel unserer Daten auf eine wert zu beschränken um später die Suche nach den Anschlussflügen zu erleichtern. Desweiteren wurde darauf geachtet nur einen Wert und nicht etwa ein `pair` als Schlüssel zu verwenden.

Airport beinhaltet alle Kürzel als Schlüssel und den Ort des Flughafens als Wert.
Zu **Flight** später mehr. Anschließend werden noch Objekte dieser Typen erstellt.

Das Einlesen von airports.txt:

```
//auslesen der flughafenkürzel
int getAirports( Airport& a )
{
    std::ifstream af( "U6A4_airports.txt" );

    if( !af ) //datei vorhanden?
    {
        string err;
        throw err = "File not found";
        return 2;
    }

    while( af.good() ) //wenn stream ok...
    {
        string s;
        string shrt;
        string lng = "";

        getline( af, s ); //hole ganz zeile
        if( s == "" ) // wenn leere zeile
        {
            continue; //mache bei nächster weiter
        }
        std::stringstream ss( s ); //umwandeln in stringstream

        std::istream_iterator<string> s_it( ss );
        std::istream_iterator<string> s_it_end;

        //auslesen der zeile
        if( ss.good() )
        {
            shrt = *s_it++; //erstes wort ist kürzel
        }
        while( s_it != s_it_end ) //rest ist Ort
        {
            lng = lng + *s_it++ + " ";
        }
        //schreibe in flughafenliste
        a.insert( std::make_pair( shrt, lng ) );
    }

    return 0;
}
```

Um die main-Methode zu entlasten wird das Einlesen der Datei(en) in einer Methode geschrieben, was auch ideal ist um eventuellen Fehler abzufangen oder um das Einlesen zu erweitern, falls man mehrerer Dateilisten gleicher Art einzulesen.

Grundsätzlich sollten alle Eventualitäten Beachtet werden, die beim Lesen einer Datei / eines Datenstroms generell auftreten können! Auch wenn es hier unsinnig erscheinen mag, doch was würde passieren wenn die Datei, warum auch immer, nicht vorhanden ist?

Es fällt auf das die Methode ein `int` zurückliefert, das hat an sich keine Bedeutung, genauso ok wäre ein `bool` oder `void` (keine Rückgabe).

Zunächst setzten wir einen `ifstream` (input file stream) auf unsere Datei. Der Name `af` steht für airportfile, Bezeichnungen für Variablen sollten niemals willkürlich festgelegt werden, mit der Zeit gewöhnt sich jeder seine Stil an.

Wichtig ist es stets zu überprüfen ob es diese Datei überhaupt da gibt wo sie sein sollte!

```
if( !af ) //datei vorhanden?
{
    string err;
    throw err = "File not found";
    return 2;
}
```

Wenn die Datei nicht vorhanden ist, wird ein String `err` (error) mit dem Inhalt „*File not found*“ geworfen, diesen sollte man später unbedingt abfangen, sonst werdet ihr die Fehlermeldung nicht sehen, falls sie je auftritt.

Als nächstes lesen wir Zeilenweise solange wie es was gibt aus der Datei.

Getline kenne wahrscheinlich die wenigsten, hier die Erklärung von cpreference.com

```
#include <string>
istream& getline( istream& is, string& s, char delimiter = '\n' );
```

The C++ string class defines the global function `getline()` to read strings from an I/O stream. The `getline()` function, which is not part of the string class, reads a line from *is* and stores it into *s*. If a character *delimiter* is specified, then `getline()` will use *delimiter* to decide when to stop reading data.

Auf Deutsch: Der erste Parameter in der () muss ein stream, genauer ein inputstream sein, das ist die Quelle der Daten, der zweite Parameter ist ein String hier landen die ausgelesenen Zeichen der Zeile. Der dritte Parameter ist der sog. delimiter, dieser ist ein `char`-wert welcher als Abbruch dienen soll, das heißt das Zeilenende ist nicht zwangsweise am Ende einer Zeile! Wenn der dritte Parameter nicht übergeben wird, so ist er automatisch `\n` was für einen Zeilenumbruch steht.

Wir prüfen nun ob es sich um eine leere Zeile handelt, wenn ja gehen wir zurück zur Schleifenbedingung und lesen die nächste Zeile (`getline` setzt den iterator automatisch auf die folgende Zeile)

Wenn es sich nicht um eine leere Zeile handelt, dann wandeln wir unseren String in einen stringstream um, setzen einen Start-Iterator und einen End-Iterator (darum ein stringstream)

Und lesen den stream nun wie gewohnt wortweise aus, wobei das erste Wort das Kürzel darstellt alles andere den Namen. Das Ganze wird anschließend mit `insert` in unseren pseudo Datentyp `Airports` geschoben.

Einlesen von direct_flights.txt:

```
int getFlights( Flightplan_TO& to, Flightplan_L& l)
{
    std::ifstream af( "U6A4_direct_flights.txt" );

    if( !af ) //datei ok?
    {
        string err;
        throw err = "File not found";
        return 0;
    }

    std::istream_iterator<string> afi( af );
    std::istream_iterator<string> eof;

    int line = 0;
    while( afi != eof )
    {
        line++; // zeile ergibt flugnummer
        string shrt1 = *afi++; //startflughafen
        string shrt2 = *afi++; //ankunftsflughafen
        string time1 = *afi++; //startzeit
        string time2 = *afi++; //endzeit

        if( time1.size() < 5 )
        {
            time1 = "0" + time1; // wenn vor "10:00"
        }
        if( time2.size() < 5 )
        {
            //damit zB aus 7:00 ein 07:00 wird zwecks zeitenvergleich
            time2 = "0" + time2;
        }

        //speichere takeoff
        to.insert( std::make_pair( shrt1 , std::make_pair( line, time1 ) ) );
        //getrennt von landung
        l.insert( std::make_pair( line, std::make_pair( shrt2, time2 ) ) );
    }
    return 0;
}
```

Dies geht größtenteils analog wir müssen lediglich nicht darauf achten dass wir eine ganze Zeile einlesen, da Kürzel und Zeit je ein zusammenhängender String sind, das wir unsere Zeiten minimal modifizieren und dass wir in zwei unterschiedlichen `multimap`'s speichern müssen.

Wir müssen nun unsere main wie folgt erweitern:

```
int main()
{
    try
    {
        getAirports( airports ); // lädt flughäfen
        getFlights( take_off, landing ); // lädt start und landezeiten
    }
    catch( string err )
    {
        std::cerr << err << std::endl;
    }
}
...
```

Nun kommen wir zur eigentlichen Aufgabe:

```
...
// setzt nen interator auf den ersten flughafen
Airport::iterator air_it = airports.begin();
std::multimap<string, string> all_flights;

while( air_it != airports.end() ) //...alle flughäfen durchlaufen
{

    std::set< Flight > next = getNextFlights( air_it->first , "00:00" );
    // setzt iterator auf ersten flug in der liste
    std::set< Flight >::iterator next_it = next.begin();
    while( next_it != next.end() ) //...alle flüge
    {
        //pseudo liste aller flughäfen für indirekten flug
        string list_of_airports = next_it->first;
        int i = 1; // #der (zwischen)landungen
        std::multimap<string, string> flightlist;
        // liste aller flüge ab hier
        flightlist =find( *next_it , i, list_of_airports, flightlist, *next_it );

        // aufllesen der liste aller flüge ab aktuellem flughafen
        std::multimap<string, string>::iterator fp_it = flightlist.begin();
        while( fp_it != flightlist.end() )
        {
            all_flights.insert( *fp_it );
            fp_it++;
        }
        next_it++;
    }
    air_it++;
}
...
```

In `std::multimap<string, string> all_flights;` wird die spätere Ausgabe gespeichert und soll uns weiter nicht interessieren (vorerst!), interessanter ist die `while`-Schleife. Mit der `While`-schleife durchlaufen wir alle unsere Flughäfen welche wir eben in `Airports` eingelsen haben. Da `maps` / `multimaps` aufgrund ihrer Datenstruktur sortiert vorliegen fangen wir automatisch beim lexikogrpsiche erstem Flughafen (AMS Amsterdam) an.

Nun gehen wir erstmal näher auf `Flights` ein, zur Erinnerung nochmal das `typedef`:

```
//Flight; Trippel für start oder landung: flughafen | (flugnummer, zeit)
typedef std::pair<string, std::pair<int, string> > Flight;
```

Ziel ist es hier einen Flughafen mit Flugnummer und Uhrzeit zu speichern, wobei es keinerlei Rolle spielt, ob Abflug oder Ankunft gespeichert werden soll. Die Flugnummer bestand aus der Zeile in der der Flug in `direct_flights.txt` stand. Das erzeugte Set unserer Flights wird mit `getNextFlights` gefüllt:

```
// ermittle weiterflugmöglichkeiten
std::set< Flight > getNextFlights( string search_airport , string start_time )
{
    // setzt iteratorbereich für alle abflüge des aktuellen flughafens
    std::pair<Flightplan_TO::iterator, Flightplan_TO::iterator> air_to_p_it =
        take_off.equal_range( search_airport );
    // setzt iterator auf ersten flug in der liste
    Flightplan_TO::iterator air_to_it = air_to_p_it.first;

    std::set< Flight > f;
    while( air_to_it != air_to_p_it.second ) //solange im range
    {
        if( air_to_it->second.second.compare( start_time ) >= 0 )
        {
            f.insert( *air_to_it );
        }
        air_to_it++;
    }

    return f;
}
```

Hier werden alle Flüge eines bestimmten Flughafens (`search_airport`) ab einer bestimmten Uhrzeit (`start_time`) gesucht und als ein Set ausgegeben, dabei sucht er lediglich nach Take-off also nur startende Flieger.

Kurz nähere Infos zu folgende Zeile: `air_to_it->second.second.compare(start_time)`

Der `->` Operator verweist bei Zeigern / Iteratoren von mehrwertigen Objekten (`pair`, `map`) auf eines der Elemente, welches mit `first` und `second` gekennzeichnet wird. `air_to_it->second` verweist somit auf den Wert unseres `Flightplan_TO` bei dem es sich um ein `std::pair<int, string>` handelt.

Das `.second` verweist wiederum auf den zweiten Wert eben dieses Pairs bei dem es sich um ein String handelt und das `.compare` ruft eine Methode des Strings auf um diesen mit `start_time` zu vergleichen.

cppreference.com:

```
#include <string>
int compare( const string& str );
int compare( const char* str );
```

[...]

The `compare()` function either compares `str` to the current string in a variety of ways, returning

Return Value	Case	Return Value
less than zero	<code>this < str</code>	less than zero
zero	<code>this == str</code>	zero
greater than zero	<code>this > str</code>	greater than zero

Zurück in der main:

In unserem next stehen also nun alle Flüge die vom Aktuellen Flughafen (AMS) aus ab 00:00 gehen. Bei uns müsste nun folgendes in next stehen:

```
AMS 12 11:15
```

Für FRA sähe unser set wie folgt aus:

```
FRA 3 16:10
FRA 4 08:30
FRA 5 11:15
FRA 6 18:25
FRA 7 14:30
```

Wir gehen nun dieses set für jeden Flug durch, schreiben jeweils das Kürzel des Flughafens in eine list_of_airports (string) welche wir später für die Ausgabe brauchen. Und starten mit find eine suche aller weitergehenden Flüge. Lesen das Ergebnis schließlich aus und schieben es in unsere Ausgabe map (später mehr dazu)

Nun zum kern des Programms find:

```
std::multimap<string, string> find( Flight act_airport, int& number_of_airports ,
    string& list_of_airports, std::multimap<string, string> flightlist,
    Flight start_airport )
{
    // zeige auf landeflughafen der aktuellen flugnummer
    Flightplan_L::iterator l_it = landing.find( act_airport.second.first );
    string landing_airport = l_it->second.first; //hole namen des landeflughafens

    //war das flugzeug auf dieser route schon?
    if( list_of_airports.find( landing_airport ) != string::npos)
    {
        return flightlist; // gebe aus
    }
    else //...wenn nicht
    {
        //...füge namen zur flughafenliste hinzu die angefliegen werden
        //zu liste hinzufügen
        list_of_airports = list_of_airports + " " + landing_airport;
        string list_of_airports_cache = list_of_airports; //sichern
        int number_of_airports_cache = number_of_airports; //sichern

        //füge aktuelle flugroute zu zur summe aller möglichen routen hinzu
        std::stringstream ss("");
        ss << getAirportname( start_airport.first )
            << getAirportname( l_it->second.first )
            << start_airport.second.second << " "
            << l_it->second.second;

        flightlist.insert(std::make_pair(list_of_airports, ss.str()));

        // finde nächste verbindungsflüge;
        std::set< Flight > next = getNextFlights( landing_airport,
            l_it->second.second );

        //durchlaufe alle gefundene verbindungsflüge
        std::set< Flight >::iterator next_it = next.begin();
        while( next_it != next.end() )
        {
```

```

        number_of_airports++; //landungen + 1
        //zahl der zwischenlandungen soll 5 nicht überschreiten
        if( number_of_airports > 5 )
        { //wenn zuviele landungen, beende suche
            break;
        } //...sonst suche weiter
        flightlist = find( *next_it++, number_of_airports,
            list_of_airports, flightlist, start_airport );
        list_of_airports = list_of_airports_cache; //hole zurück
        number_of_airports = number_of_airports_cache; // hole zurück
    }

    return flightlist; //gibt flüge aus
}
}
}

```

Wir holen uns zunächst das Ziel des aktuellen Fluges und prüfen ob wir schon da waren (steht dann in `list_of_airports`, s.u.), wenn wir diesen Flughafen bereits angefliegen haben, dann sind wir bereits fertig. Wenn nicht, dann könne wir von hier aus weiterfliegen. Hierzu fügen wir zunächst den LandeFlughafen (nächster Startflughafen) in unser string `list_of_airports` hinten an. Unser string könnte nun wie folgt aussehen:

```
AMS ROM
```

Kopieren wir unser `list_of_airports` sowie unser `number_of_airports` (welche die Landungen zählt die wir auf unserer Reise haben) was nötig ist, da wir rekursiv weitersuchen werden und wir so unsere „alten Zustände“ behalten. Wir schreiben uns dann ein string (über ein stringstream) das wie folgt aussieht:

```
Amsterdam Rome 09:00 11:00
```

Wozu wir die Methode `getAirportname()` verwenden welche lediglich das Kürzel in den Namen umwandelt. Dieses erzeugte string schieben wir in die `flightlist` wo alle möglichen Flüge drin „gespeichert“ werden. Anschließend holen wir vom Aktuellen Zielflughafen (nun neuer Startflughafen) wie in der `main` mit `getNextFlights()` die folge Flüge und rufen analog zur `main`-Methode `find()` auf, mit der Bedingung, das `number_of_airports` nicht größer sein darf als 5. Nach dem `find`-Aufruf werden wieder die „alten“ gespeicherten `list_of_airport` und `number_of_airports` wiederher gestellt (da sie in `find` verändert wurden). Wenn alle Flüge durchlaufen sind springen wir zur `main` zurück und gehen das ganze für den nächsten Flughafen von vorne durch.

In der `main` haben wir alle erhalten werte in einer `multimap all_flights` gespeichert, diese wird einfach mit iteratoren ausgelesen und ausgeben. Aber warum haben wir es nicht vorher schon ausgeben sondern erst in die `map` geschoben?

Ganz einfach, wie oben erwähnt werden `maps / multimaps` lexikographisch sortiert, das heißt unsere Ausgabe ist nun wie gefordert Alphabetisch und zwar sortiert nach den Kürzeln der angeflogenen Flughäfen, da wir diese als Schlüssel hatten.

Die Ausgabe sollte wie folgt aussehen:

```

Amsterdam Rome 09:00 11:15 (AMS ROM)
Amsterdam Frankfurt 09:00 13:45 (AMS ROM FRA)
Cairo Tunis 14:10 15:20 (CAI TUN)
Chicago Montreal 18:40 21:45 (CHI YUL)
Frankfurt Amsterdam 07:20 08:30 (FRA AMS)
Frankfurt Rome 07:20 11:15 (FRA AMS ROM)
Frankfurt New York 13:00 16:10 (FRA NYC)
Frankfurt Chicago 13:00 18:15 (FRA NYC CHI)
Frankfurt Montreal 13:00 21:45 (FRA NYC CHI YUL)
Frankfurt Rome 09:30 11:15 (FRA ROM)
Frankfurt Singapore 06:15 18:25 (FRA SIN)

```

```
Frankfurt Peking 06:15 23:30 (FRA SIN BJS)
Frankfurt Vienna 13:30 14:30 (FRA VIE)
Frankfurt Moscow 13:30 20:00 (FRA VIE MOW)
New York Chicago 16:45 18:15 (NYC CHI)
New York Montreal 16:45 21:45 (NYC CHI YUL)
Paris Frankfurt 11:00 12:20 (PAR FRA)
Paris New York 11:00 16:10 (PAR FRA NYC)
Paris Chicago 11:00 18:15 (PAR FRA NYC CHI)
Paris Montreal 11:00 21:45 (PAR FRA NYC CHI YUL)
Paris Vienna 11:00 14:30 (PAR FRA VIE)
Paris Moscow 11:00 20:00 (PAR FRA VIE MOW)
Paris London 14:20 15:00 (PAR LON)
Rome Cairo 11:00 14:00 (ROM CAI)
Rome Tunis 11:00 15:20 (ROM CAI TUN)
Rome Frankfurt 12:00 13:45 (ROM FRA)
Singapore Peking 19:00 23:30 (SIN BJS)
Vienna Moscow 15:55 20:00 (VIE MOW)
```

Noch kurz ein paar worte zum sog. pipen.

Wenn wir ein Programm starten welches eine Ausgabe mit `cout` beinhaltet, so könne wir diese Ausgabe unter Linux (cygwin) in andere Programme oder Dateien umleiten. Der „Befehl“ sieht in unserem Fall wie folgt aus:

```
./U6A4_flight_plan > datei.txt
```

Worauf wir eine Datei.txt erhalten mit obigem Inhalt.