06_phone_dict: Entwicklung eines Telefonverzeichnisses

Dipl.-Ing. Dr. Günter Kolousek

1 Allgemeines

- **Es gelten die gleichen Richtlinien wie beim ersten Beispiel!!!**

2 Aufgabenstellung

Schreibe ein C++ Programm `phone`, das ein einfaches Telefonbuch implementiert.

Die folgende Datei `phone_numbers.json` stelle ich zur Verfügung:

```
{
  "maxi" : 4711,
  "mini" : 4712,
  "otto" : 1111,
  "erna" : 2222
}
```

Damit hat die Benutzerschnittstelle folgendermaßen zu funktionieren:

```
$ phone -h
Stores, retrieves, and deletes phone numbers
Usage: phone [Options] [NAME] [NUMBER]

Positionals:
  NAME TEXT Excludes: -l      name to query or set
  NUMBER INT Needs: NAME Excludes: -e
                              phone number to set

Options:
  -h,--help                   Print this help message and exit
  -f,--file TEXT (Env:PHONE_NUMBERS_FILE)
                              The file to be processed (default: phone_numbers.json)
  -l Excludes: -e NAME        List the content of the phone dictionary
  -e Needs: NAME Excludes: -l NUMBER
                              Erase the entry for the given NAME
$ phone -l
erna: 2222
maxi: 4711
mini: 4712
```

```
otto: 1111
$ phone maxi
4711
$ phone maxi2
no user maxi2
$ phone maxi2 1234
number 1234 set for maxi2
$ phone maxi2
1234
$ phone -e maxi2
name maxi2 erased
$ phone -e maxi2
no user maxi2
$ phone -l maxi2
-l excludes NAME
Run with --help for more information.
$ phone -e maxi2 1234
-e excludes NUMBER
Run with --help for more information.
$ phone -l -e
-l excludes -e
Run with --help for more information.
$ phone -e
-e requires NAME
Run with --help for more information.
```

Was ist hier zu sehen bzw. wichtig?

- Der Dateiname kann über eine Option angegeben werden. Erfolgt dies nicht, dann wird der Wert in einer Umgebungsvariable `PHONE_NUMBERS_FILE` gesucht. Gibt es auch diese nicht, dann wird der Defaultwert `phone_numbers.json` verwendet.
- Es gibt ein Flag `-l`, das jedoch nicht mit `NAME` gemeinsam vorkommen darf.
- Es gibt ein Flag `-e`, das jedoch nicht mit `NUMBER` gemeinsam vorkommen darf.
- Flag `-l` darf nicht gemeinsam mit `-e` vorkommen.
- Flag `-e` benötigt `NAME`.
- Alle Fehler- oder Statusmeldungen gehen nach `stderr`. D.h. nur die Ausgabe der gesuchten Nummer geht nach `stdout`. Damit kann die Ausgabe wiederum mit einer Pipes verbunden werden.
- Jetzt werden die Exitcodes festgelegt:
  - 0 … Operation erfolgreich
  - 1 … Operation fehlgeschlagen
  - weitere sind von `CLI11` selber (≥100)
- Zur Verarbeitung der JSON-Dateien und Daten verwenden wir die header-only Bibliothek `JSON for Modern C++`.

3 Anleitung

Schreibe ein Programm entsprechend der Aufgabenstellung.

1. Beginne zuerst damit die Benutzerschnittstelle zu implementieren:
   - `add_option` bzw. `add_flag` liefert jeweils einen Pointer auf `CLI::Option` zurück. Dieses Objekt kann verwendet werden.
   - Methode `CLI::Option::excludes` zum Ausschließen einer anderen Option
   - Methode `CLI::Option::needs` wenn eine Option eine andere benötigt
   - Methode `CLI::Option::envname` legt den Namen der Umgebungsvariable fest, die herangezogen werden soll, wenn keine Eingabe getätigt wird. Gibt es auch diese Umgebungsvariable nicht, dann wird automatisch der Wert der Initialisierung der zugrundeliegenden Variable genommen.
2. Als nächstes speichere `json.hpp` an eine geeignete Stelle in deinem Dateisystem und konfiguriere die Meson-Optionen und deine `meson.build`.
3. Implementiere die Funktionalität mittels `json`:
   - Für das Lesen aus einer Datei bzw. für das Schreiben in eine Datei wurde von `json` der Operator >> bzw. << überladen!
   - Für den Zugriff auf das `json`-Objekt wurde auch der Operator `[]` überladen!
   - Auch das Iterieren funktioniert wie in C++ üblich. Speziell für C++ 17 kann sogar das "structured binding" verwendet werden:

     ```
     for (auto& [key, value] : phone_number.items()) {
     ```

     Nur `items()` ist "neu", aber das macht nichts.
   - Der Zugriff auf ein `json`-Objekt kann also wie in Container-Objekten der Standardbibliothek erfolgen. Das betrifft zum Beispiel das Suchen nach nach einem Key:

     ```
     if (phone_numbers.find(name) != end(phone_numbers))
     ```

     `end()` liefert einen Iterator zurück, der auf ein imaginäres Element zeigt, das *nach* dem letzten Element der Collection steht und `find` liefert genau diesen Iterator zurück, wenn das gesuchte Objekt nicht in der Collection enthalten ist.
   - Aber auch das Löschen funktioniert auf diese Weise:

     ```
     phone_numbers.erase(name);
     ```
4. Denke an die Exit-Codes!
5. In weiterer Folge geht es darum einen eigenen Typ `PhoneDict` anzulegen, der wie ein Dictionary funktioniert und eine ähnliche Schnittstelle wie der Type `json` aufweist:
   a) Nimm dir die Folien `data_structures_hashing` zur Hand und wiederhole! Du wirst dieses Wissen benötigen. Früher oder später.
   b) Lege ein Modul `phone_dict` an. Es soll sich um ein header-only Modul handeln, damit wird die Implementierung der Methoden ein Kinderspiel, da das Schlüsselwort `auto` seine volle Wirkung entfalten kann. Beginne mit einer einfachen Klassendefinition `PhoneDict`, die vorerst als Attribut nur eine `std::unordered_map` besitzt, die als Key einen String und als Value eine ganze Zahl aufweist. Bedenke, dass der Operator `[]` in einer map beim Zugriff auf einen nicht existierenden Key einen neuen Eintrag anlegt!!! Je nach Implementierung ist dies in diesem Beispiel aber gar nicht von Relevanz!

      Der Einfachheit halber implementieren wir vorerst die folgenden Methoden:
      - `begin()` liefert das Ergebnis der Methode `begin()` der `unordered_map` zurück.

- `cbegin()` analog zu `begin()` nur wird letztendlich ein konstanter Iterator zurückgeliefert. Für was wird dieser benötigt? Wieso markiert man eine Methode mit `const`?
- `end()` liefert das Ergebnis der Methode `end()` der `unordered_map` zurück.
- `cend()` analog zu `end()` und `cbegin()`.
- `find(const string&)` liefert das Ergebnis der Methode `find()` der `unordered_map` zurück.
- `erase(const string&)` liefert das Ergebnis der Methode `erase()` der `unorderd_map` zurück.
- `operator[](const string&)` liefert das Ergebnis der Methode `operator[]()` der `unorderd_map` zurück.

D.h., dass diese Methoden lediglich an die `unordered_map` delegieren. Das ist einfach!

Weiters benötigen wir noch überladene Methoden für >> und <<. Dabei kann es sich klarerweise um keine Methoden unserer Klasse handeln.

Implementiere deshalb diese als freien Funktionen:

```
inline ostream& operator<<(ostream&, const PhoneDict&);
inline istream& operator>>(istream&, PhoneDict&);
```

Damit diese freien Funktionen allerdings auf das Innere/das Geheime der Klasse `PhoneDict` zugreifen dürfen müssen diese Funktionen Freunde der Klasse sein und das geht nur, wenn in der Klasse diese Funktionsprototypen mit dem Schlüsselwort `friend` als Freunde angegeben werden.

Für diese Implementierung verwenden wir wieder die Funktionalität von `json`. Beachte bitte, dass aus technischen Gründen im `operator<<` das `json`-Objekt folgendermaßen angelegt werden muss:

```
nlohmann::json dict;
dict = pd.dict;
```

Die geschwungenen Klammern der einheitlichen Initialisierung funktionieren hier leider nicht so: es würde seitens `json` zuerst ein JSON-Array angelegt werden, das unser JSON-Objekt beinhaltet und das wollen wir nicht.

Die genaue Regel, die für die Bibliothek `nlohmann::json` ist: Verwendest du die geschwungenen Klammern, dann wird ein JSON Array angelegt, *außer* innerhalb der geschwungenen Klammern befinden sich Initializer-Listen, die aus genau 2 Elementen (in der Struktur Key und Value)! Ist ein bisschen kompliziert, also hier ein paar Beispiele, die diesen Sachverhalt illustrieren:

```
json v1=1.0;  // 1.0
json v2{1.0};  // [1.0]
json v3{{"a", 1}, {"b", 2}};  // {"a" : 1, "b" : 2}
json v4{{"a", 1, 2}};  // [["a", 1, 2]]
json v5{{1, 2}};  // [[1, 2]]
```

Die Quintessenz daraus ist, dass für diese Bibliothek die "vereinheitlichte" Initialisierung nicht verwendet werden kann, da `json v1{v2};` ein JSON Array anlegt, dessen einziges Element `v2` ist...

c) Stelle jetzt auf dein neues Modul um. Eigentlich ist nur `json` durch `PhoneDict` auszutauschen, mit einer Ausnahme: Wir haben keine `items()` Methode implementiert und das ist auch nicht notwendig, da wir die Methoden `begin()` und `end()` implementiert haben und diese ausreichen,

um eine range-basierte Schleife einzusetzen. D.h. weg mit dem Aufruf der `items()` Methode und alles sollte wie vorher funktionieren.

6. Unit-Tests wollen wir ausnahmsweise weglassen!

4 Übungszweck dieses Beispiels

- Vertiefung von `CLI11` und Exitcodes
- Verwendung der header-only Bibliothek `JSON for Modern C++`
- Einführung in Container der Standardbibliothek, im speziellen `unordered_map`
- Grundprinzip des Iterierens
- Überladen von Operatoren
- `friend`
- Überladen von `operator<<`
- Wiederholung der Hashing-Datenstrukturen, `data_structures_hashing`