07_phone_dict2: Entwicklung eines Telefonverzeichnisses – 2

Dipl.-Ing. Dr. Günter Kolousek

Creative Commons Namensnennung - Nicht kommerziell - Keine Bearbeitungen 4.0 International Lizenz

1 Allgemeines

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

2 Aufgabenstellung

Kopiere das gesamte Verzeichnis `06_phone_dict` auf `07_phone_dict` (allerdings mit den Mitteln der Versionsverwaltung!!!), da wir zwar ein neues Beispiel haben, allerdings nur den Code des alten erweitern.

Es geht darum, die `unordered_map` durch eine eigene Implementierung zu ersetzen. Warum? Weil es geht?! Weil es hier einiges zu lernen gibt.

Das Programm `phone` wird genauso funktionieren wie vorher.

3 Anleitung

1. Es geht eigentlich nur darum eine eigene Klasse zu entwickeln, die genau das tut, das auch `unordered_map` erledigt, nur *ohne* Container der Standardbibliothek zu verwenden! Die Idee ist eine Dictionary zu implementieren, das auf *direkte* Verkettung der Überläufer basiert.

   Dazu werden wir eine einfach verkettete Liste implementieren. Wiederhole daher den Inhalt des Foliensatzes `data_structures_simple`!

   Lege daher einmal ein Modul `sll` an, das eine einfach verkettete Liste basierend auf rohen Zeigern mit folgendem Interface implementiert (`.h` und `.cpp`):

   ```
   class List final {
     public:
       ~List();
       bool set(string key, int value);
       int search(string key);
       bool remove(string key);
       void clear();
   };
   ```

Es handelt sich um eine bewusst einfach gehaltene Version, da das eigentliche Implementieren einer Liste eh schon ausreichend im 2.JG geübt wurde. Der Schwerpunkt hier liegt auf der Speicherveraltung. Die Liste soll für den Speicher verantwortlich sein, d.h. sie soll `Node`-Instanzen anlegen und auch wieder löschen. Die boolschen Werte, die zurückgegeben werden, geben den Erfolg der jeweiligen Operation an. Die ganze Zahl, die `search` zurückgibt ist ein Problem! Was soll zurückgegeben werden, wenn der angegebene `key` nicht in der Liste zu finden ist?

Gut, dann ändern wir den Prototypen von search entsprechend um:

```
bool search(std::string key, int& value);
```

So, jetzt sieht es besser aus: Wir wissen, ob die Operation erfolgreich war oder nicht (und dies ist konsistent mit den anderen Operationen) und wir können auf den gefundenen Wert zurückgreifen.

Welche Alternativen hätte es noch gegeben?

- Wir hätten Pointer auf die gefundenen oder hinzugefügten `Node`-Instanzen zurückliefern können. In diesem Fall wäre auch die Rückgabe eines `nullptr` zur Anzeige einer fehlgeschlagenen Operation möglich. Aber das wäre auch nicht gut, da wir in diesem Fall einen Zugriff auf unsere internen Datenstrukturen ermöglichen und damit Tür und Tor für jede beliebige Schandtat offen wäre.
- Anstatt des Referenzparameters hätten wir einen Pointer verwenden können. In der Programmiersprache C wäre das die einzige Lösung gewesen.
- Wir hätten den Rückgabetyp auf `std::pair` ändern können.
- Wir hätten auch `std::optional` verwenden können.

Aber im Moment sind wir mit unserer Lösung zufrieden und können diese bequem implementieren!

Bedenke, dass es noch andere Probleme mit rohen Zeigern gibt:

- memory leaks
- dangling pointer
- double delete

Mit Arrays am Heap wird es noch einmal komplizierter, da dann `delete[]` verwendet werden muss...

Weiters hast du bemerkt, dass die Klasse als `final` markiert worden ist. Damit kann von dieser Klasse nicht mehr abgeleitet werden. Wieso das wichtig ist? Wir haben einen Destruktor implementiert und dieser ist nicht `virtual` und das wäre wirklich ein Problem! Doch davon später...

Ach ja, eine Klasse `Node` wirst du auch noch benötigen, die du gerne als `struct` realisieren darfst. Das ist ja auch kein Problem, da es sich um ein Implementierungsdetail handelt und es keiner sieht (d.h. darauf Zugriff hat).

2. Ok, das ist ja leicht zu implementieren, aber wie weißt du, dass dies auch funktionsfähig ist? Hier sind Unit-Tests eindeutig angebracht und daher von meiner Seite gefordert! Go!
3. Und was machst du, wenn dein Programm ”abstürzt”? Ok, vielleicht ist es dir das bis jetzt noch nicht passiert, aber dann baue so etwas doch einfach in dein Programm ein:

   ```
   int* fail_ptr{nullptr};
   cout << *fail_ptr << endl;
   ```

   Und dann starten, du erhältst so etwas wie

```
fish: "phone -l" terminated by signal SIGSEGV (Address boundary error)
```

Nicht so schön, nicht wahr? Ok, man könnte jetzt noch den Debugger bemühen, aber das heben wir uns für später einmal auf.

Was ist zu tun?

a) Zuerst einmal ist das Projekt natürlich im Debug-Modus zu übersetzen. Klar, ohne dem gibt es keine Symbole und auch der Debugger würde nicht besonders viel anzeigen können... Falls du es nicht weißt wie dies geht, dann → `meson_tutorial.pdf`!

b) "Installiere" `backward.cpp` und `backward.h` in deinem Projekt. Das reduziert sich darauf, dass du

- diese beiden Dateien in dein Projekt kopierst. Wenn du jetzt dein Projekt neu übersetzt und startest erhältst du so etwas in der folgenden Art:

  ```
  Stack trace (most recent call last):
  #3    Object "[0xffffffffffffffff]", at 0xffffffffffffffff, in
  #2    Object "phone", at 0x562fbb4325fd, in
  #1    Object "/usr/lib/libc.so.6", at 0x7f6493dc5ee2, in __libc_start_main
  #0    Object "phone", at 0x562fbb4321bc, in
  Segmentation fault (Address not mapped to object [(nil)])
  fish: "phone -l" terminated by signal SIGSEGV (Address boundary error)
  ```

  Hmm, so richtig toll ist das noch immer nicht, deshalb werden wir noch ein paar kleine Änderungen vornehmen.

- das folgende Präprozessormakro `-DBACKWARD_HAS_BFD=1` entsprechend dem Compiler übergibst (siehe `meson_tutorial.pdf`)

- und die Bibliothek `bfd` zu deinen Executables linkst. Das geht mittels `cc.find_library()`. Siehe dazu wieder `meson_tutorial.pdf`. Klarerweise muss diese auch installiert sein. Üblicherweise ist diese im Paket `binutils` und dieses Paket kannst du ganz einfach mittels deinem Paket-Manager deiner Wahl installieren.

c) Übersetzen, ausführen, staunen!

4. Jetzt kannst du deinen Testcode wieder entfernen!

5. Das ist ja schön und gut, aber haben wir jetzt wirklich alles richtig gemacht? Wie sieht es mit Memory-Leaks aus? Könnte sein, dass wir vergessen haben Speicher wieder freizugen, nicht wahr?

   Hier kommt uns netterweise das Tool `valgrind` zur Hilfe. Zuerst installieren mit dem Paketmanager deiner Wahl:

   ```
   $ sudo pacman -S valgrind
   ```

   Dann am Besten die Unit-Tests verwenden:

   ```
   $ valgrind test1
   ```

   Die relevante Ausgabe sollte dann in etwa folgendermaßen aussehen:

   ```
   ==11344==
   ==11344== HEAP SUMMARY:
   ==11344==     in use at exit: 0 bytes in 0 blocks
   ==11344==   total heap usage: 21 allocs, 21 frees, 75,368 bytes allocated
   ==11344==
   ==11344== All heap blocks were freed -- no leaks are possible
   ```

```
==11344==
==11344== For counts of detected and suppressed errors, rerun with: -v
==11344== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Natürlich kannst du gerne ein Memory-Leak nur zum Testen einbauen... Aber nicht vergessen, dass du den Code wieder entfernen musst.

Bei einem Aufruf der folgenden Art

```
$ valgrind -v test1
```

wird mehr Information angezeigt, sodass jeder Leak zu "sehen" ist. Aber natürlich gibt es dann halt auch mehr Ausgabe.

Prinzipiell sind die folgenden Optionen bzw. der folgende Aufruf sinnvoll:

```
$ valgrind -v --leak-check=full --show-leak-kinds=all test1
```

6. Der letzte große Schritt ist es, jetzt noch das eigentliche Hashingarray zu implementieren, das in unserem Fall `Dictionary` heißen soll. Was benötigen wir dazu?

   - Eine geeignet Hashfunktion. Dazu nehmen wir die Hashfunktion djb2 von Dan Bernstein (siehe Foliensatz `data_structures_hashing`) her. Den Parametertyp `unsigned char*` werden wir zu `const unsigned char*` umändern und als Rückgabewert werden wir anstatt `unsigned long` den Typalias `size_t` verwenden. Lese dir die entsprechende Information auf cppreference bzgl. `size_t` durch!

     Diese Hashfunktion ist eine Funktion, die wir nur in unserer Klasse `Dictionary` benötigen und auch nur zwecks der Implementierung der Funktion. Also wie und in welchem Teil der Klasse wird diese Funktion in unserer Klasse implementiert werden? Abgesehen von der vorherigen Frage, sollte es klar sein, dass diese Funktion in der `.cpp` Datei definiert werden soll.

   - Eine geeignete Abstraktion für unsere Klasse ist natürlich eine, die genauso aussieht wie die Klasse `PhoneDict` im vorherigen Beispiel, weil wir ja den Typ ersetzen wollen.

   - Bezüglich der Implementierung der Klasse wird es schon etwas komplizierter. Beginnen wir mit dem einfachen Teil. In privaten Teil der Klasse implementieren wir jetzt die folgenden Methoden:

     ```
     bool set_(string, in);
     bool search_(string, int&);
     bool remove_(string);
     void clear_();
     ```

     Es ist zwar prinzipiell nicht zwingend notwendig, aber manchmal werden private Methoden gekennzeichnet, in diesem Fall mit einem abschließenden underscore.

     Diese sollen ein Hashingarray auf Basis der direkten Verkettung der Überläufer realisieren.

     Weiters benötigen wir ein rohes Array einer bestimmten Größe von `List` Instanzen. Diese Größe wollen wir fix kodieren: 13 ist nicht so schlecht. Warum?

     Wie können wir diese *fix* kodieren sodass dieser Wert für alle Instanzen immer gleich ist? Als `const`? Dann würde es zur Laufzeit initialisiert und dann nicht mehr verändert werden. Hier gibt es zwei Möglichkeiten: als Klassenvariable oder als Instanzvariable. Als Instanzvariable macht das keinen Sinn: Es soll ja immer für alle Instanzen gleich bleiben. Dann doch besser als Klassenvariable. Hier wäre allerdings `constexpr` besser. Warum?

Voila, dann ist es auf einmal auch kein Problem mehr, diese ”Variable” als Größenangabe für das rohe Array zu verwenden!

Bei der Implementierung von `set_`, `get_` und `remove_` musst du den Index berechnen und damit auf den Pointer des unterliegenden C-Arrays der übergebenen `string`-Instanz zugreifen. Diesen kannst du mit welcher Methode bekommen? Die cppreference hilft wieder weiter.

Hier gibt es allerdings das ”Problem”, dass die besagte Methode einen Pointer vom Typ `const char*` Ponter zurückliefert, die statische Methode `hash` sich allerdings einen `const unsigned char*` erwartet. Hier hilft kein `static_cast`, da dieser ”cast” nur kompatible Typen wandelt, hier muss etwas ”stärkeres” her: `reinterpret_cast` ist das benötigte Werkzeug.

So, damit sind die Methoden `set_`, `search_`, `remove_` und `clear_` leicht zu implementieren.

- Jetzt zu den öffentlich sichtbaren Methoden, die wir benötigen, um den Typ `PhoneDict` durch unseren neuen Typ `Dictionary` zu ersetzen: `begin()`, `cbegin()`, `end()`, `cend()`, `find)`, `erase()`, `operator[]` und `clear()`. Diese sind insoferne ein Problem, dass diese jetzt zum Teil einen Iterator zurückliefern. Diesen müssten wir selber implementieren und das ist für das Erste gar nicht so einfach. Was ist also zu tun? Wir werden zu einem kleinen Trick greifen: Alle Einträge in unser `Dictionary` werden wir in *zusätzlich* noch in einem `std::vector<std::pair>` speichern, der wunderbar in der Lage ist die benötigten Iteratoren zurückzuliefern. Damit wird es wieder einfach.

  Beachte allerdings z.B. bei `find`, dass du zuerst in unserem eigenen Hashing-Array suchst und nur nur dann den benötigten Iterator mittels der Funktion `find_if` aus der Algorithmusbibliothek suchst, um den benötigten Iterator zurückzuliefern, denn sonst macht unserer Hasharray absolut keinen Sinn (abgesehen von dem Lernerfolg natürlich ;-)). Analoges gilt für `erase` und `operator[]`.

  `find_if` gibt es in der Algorithmusbibliothek der Standardbibliothek von C++. D.h. es ist mittels `#include <algorithm>` einzubinden (cppreference hilft wie immer weiter). `find_if` ist eine Funktion, die in einem Container der Standardbibliothek nach einem Wert sucht, der mittels einer Prädikatsfunktion spezifiziert wird. Das ist deshalb nötig, da unser Vektor `pair<string,int>` Werte enthält, wir aber nur nach dem `string`-Teil suchen wollen.

  Beim Hinzufügen ist ein `pair<string,int>`-Eintrag dem Vektor hinzuzufügen. Für das Anlegen verwendest du am besten die Funktion `std::make_pair()`!

  So, jetzt bist du in der Lage, `PhoneDict` durch `Dictionary` auszutauschen!

  Weitere Unit-Tests werden wir keine implementieren. Uns reicht es, dass wir die Funktionalität auf der Kommandozeile testen können.

  Fertig!!!

4 Übungszweck dieses Beispiels

- Implementierung eines Destruktors
- einfache Speicherverwaltung mit rohen Zeigern, Erkennen der Probleme!
- Wiederholung dynamische Datenstrukturen, `data_structures_simple`!
- Referenzparameter und Rückgabe mehrere Werte
- `size_t`

- Hashfunktionen implementieren
- `const`-Methoden
- `const` vs. `constexpr`
- `constexpr` für statische Variablen einer Klasse
- rohes Array
- `reinterpret_cast`
- `std::pair` und `std::make_pair()`
- `find_if` und Prädikatsfunktionen
- Einbinden einer Systembibliothek üben, den Debugmodus beim Übersetzen setzen und `backward.cpp` verwenden
- valgrind einsetzen