Eine Einführung in modernes C++

Teil 2 – Fortgeschrittenes C++

Paul Nykiel

2. Mai 2020

Design Pattern

Const-Correctness

- Alles per Referenz: Super Effizient aber Fehlerquelle

Const-Correctness

- Alles per Referenz: Super Effizient aber Fehlerquelle
- Const-Referenzen

Const-Correctness

- Alles per Referenz: Super Effizient aber Fehlerquelle
- Const-Referenzen
- `const int &a = b;`

Const-Correctness

- Alles per Referenz: Super Effizient aber Fehlerquelle
- Const-Referenzen
- `const int &a = b;`
- Const-Memberfunktionen

Const-Correctness

- Alles per Referenz: Super Effizient aber Fehlerquelle
- Const-Referenzen
- `const int &a = b;`
- Const-Memberfunktionen
- `int getX() const {...`

Const-Correctness

- Alles per Referenz: Super Effizient aber Fehlerquelle
- Const-Referenzen
- `const int &a = b;`
- Const-Memberfunktionen
- `int getX() const {...`
- `mutable`

Beispiel: Const-Correctness

RAII

- Resource acquisition is initialization

RAII

- Resource acquisition is initialization
- Objekt akquiriert Resourcen im Konstruktor und gibt sie im Destruktor frei

RAII

- Resource acquisition is initialization
- Objekt akquiriert Resourcen im Konstruktor und gibt sie im Destruktor frei
- ```
  void doStuff() {
      std::lock_guard<std::mutex> lockGuard{mutex};
      shared_resource = 17;
      shared_resource += functionThatCanThrow();
  }
  ```
  ```

RAII

- Resource acquisition is initialization
- Objekt akquiriert Resourcen im Konstruktor und gibt sie im Destruktor frei
- ```
  void doStuff() {
      std::lock_guard<std::mutex> lockGuard{mutex};
      shared_resource = 17;
      shared_resource += functionThatCanThrow();
  }
  ```
- Auch bei eigenen Klassen anwenden, Klassen sollten nach Konstruktor in korrektem Zustand sein
  ```

OOP in C++

Klassendeklaration

```
class A : public B {
    public:
        A(int c, int d);
        int getD() const;
    private:
        int d;
};
```

Klassendefinition

```
A::A(int c, int d) : B{c}, d{d} {
    // More code
}

int A::getD() const {
    return this->d;
}
```

Namespaces

```
namespace mynamespace {
    int calculate() { return 17; }
}

int main() {
    return mynamespace::calculate();
}
```

Namespaces

```
namespace mynamespace {
    int calculate() { return 17; }
}

int main() {
    return mynamespace::calculate();
}
```

- Struktur

Namespaces

```
namespace mynamespace {
    int calculate() { return 17; }
}

int main() {
    return mynamespace::calculate();
}
```

- Struktur
- Keinen Einfluss auf Sichtbarkeit

Namespaces

```
namespace mynamespace {
    int calculate() { return 17; }
}

int main() {
    return mynamespace::calculate();
}
```

- Struktur
- Keinen Einfluss auf Sichtbarkeit
- Namespaces nicht mit `using` einbinden

Beispiel: HelloWorld OOP

Noch mehr C++

Casts und Null-Pointer

- `static_cast<T>(a)`

Casts und Null-Pointer

- `static_cast<T>(a)`
- `dynamic_cast<T>(a)`

Casts und Null-Pointer

- `static_cast<T>(a)`
- `dynamic_cast<T>(a)`
- `0`, `NULL` und `nullptr`

Trailing return-type

```
// Normale Syntax
std::vector<std::set<double>> a(double b) {

// Trailing return-type
auto a(double b) -> std::vector<std::set<double>> {
```

Type-Deduction

```
float f = 0;
auto i = 0;
auto i2 = i;
auto i3 = static_cast<int>(f);
decltype(i3) i4 = 12;
```

Kurzeinführung Templates als Generics

```
template<typename T>
auto max(T i, T j) -> T {
    if (i > j) {
        return i;
    } else {
        return j;
    }
}

max<int>(1,2);
max(1,2);
```

STL

STL

- Standard Template Library

STL

- Standard Template Library
- Utility

STL

- Standard Template Library
- Utility
- Container

STL

- Standard Template Library
- Utility
- Container
- Algorithmen

STL

- Standard Template Library
- Utility
- Container
- Algorithmen
- IO

STL

- Standard Template Library
- Utility
- Container
- Algorithmen
- IO
- Concurrency

Container

	Auf Element zugreifen	Element einfügen
`std::array<T, N>`	$\mathcal{O}(1)$	X
`std::vector<T>`	$\mathcal{O}(1)$	$\mathcal{O}(n)$
`std::deque<T>`	$\mathcal{O}(1)$	$\mathcal{O}(n)$ bzw. $\mathcal{O}(1)$
`std::list<T>`	$\mathcal{O}(n)$	$\mathcal{O}(1)$

Iteratoren

```
std::vector<int> a = {1,2,17,42,1337};
int b = 0;

for (std::vector<int>::iterator it = a.begin();
        it != a.end(); ++it) {
    b += *it;
}
```

for-each

```
std::vector<int> a = {1,2,17,42,1337};
int b = 0;

for (const auto &i : a) {
    b += i;
}
```

Weitere Container und Aggregationstypen

- Assoziative-Container: `std::set<T>` und `std::map<K, V>`

Weitere Container und Aggregationstypen

- Assoziative-Container: `std::set<T>` und `std::map<K, V>`
- Sammlung verschiedener Objekte: `std::tuple<T...>` und `std::pair<T1, T2>`

Weitere Container und Aggregationstypen

- Assoziative-Container: `std::set<T>` und `std::map<K, V>`
- Sammlung verschiedener Objekte: `std::tuple<T...>` und `std::pair<T1, T2>`
- Objekt das nicht vorhanden sein muss: `std::optional<T>`

Praxis

Praxis:

Praxis: Huffman-Codierer

Vorgehen

- Datei einlesen

Vorgehen

- Datei einlesen
- Relative Häufigkeiten ($\approx$ Wahrscheinlichkeiten) berechnen (Byteweise)

Vorgehen

- Datei einlesen
- Relative Häufigkeiten ($\approx$ Wahrscheinlichkeiten) berechnen (Byteweise)
- Huffman-Baum konstruieren

Vorgehen

- Datei einlesen
- Relative Häufigkeiten ($\approx$ Wahrscheinlichkeiten) berechnen (Byteweise)
- Huffman-Baum konstruieren
  - Menge aller Symbole mit zugehörigen Wahrscheinlichkeiten

Vorgehen

- Datei einlesen
- Relative Häufigkeiten ($\approx$ Wahrscheinlichkeiten) berechnen (Byteweise)
- Huffman-Baum konstruieren
  - Menge aller Symbole mit zugehörigen Wahrscheinlichkeiten
  - Zwei Symbole geringster Wahrscheinlichkeit finden

Vorgehen

- Datei einlesen
- Relative Häufigkeiten ($\approx$ Wahrscheinlichkeiten) berechnen (Byteweise)
- Huffman-Baum konstruieren
  - Menge aller Symbole mit zugehörigen Wahrscheinlichkeiten
  - Zwei Symbole geringster Wahrscheinlichkeit finden
  - Symbole aus Menge Entfernen

Vorgehen

- Datei einlesen
- Relative Häufigkeiten ($\approx$ Wahrscheinlichkeiten) berechnen (Byteweise)
- Huffman-Baum konstruieren
  - Menge aller Symbole mit zugehörigen Wahrscheinlichkeiten
  - Zwei Symbole geringster Wahrscheinlichkeit finden
  - Symbole aus Menge Entfernen
  - Zu neuem Knoten kombinieren

Vorgehen

- Datei einlesen
- Relative Häufigkeiten ($\approx$ Wahrscheinlichkeiten) berechnen (Byteweise)
- Huffman-Baum konstruieren
  - Menge aller Symbole mit zugehörigen Wahrscheinlichkeiten
  - Zwei Symbole geringster Wahrscheinlichkeit finden
  - Symbole aus Menge Entfernen
  - Zu neuem Knoten kombinieren
  - Knoten zu Menge hinzufügen

Vorgehen

- Datei einlesen
- Relative Häufigkeiten ($\approx$ Wahrscheinlichkeiten) berechnen (Byteweise)
- Huffman-Baum konstruieren
  - Menge aller Symbole mit zugehörigen Wahrscheinlichkeiten
  - Zwei Symbole geringster Wahrscheinlichkeit finden
  - Symbole aus Menge Entfernen
  - Zu neuem Knoten kombinieren
  - Knoten zu Menge hinzufügen
- Abbildung ausgeben

Anforderungen

- Eigene `template` Klasse für Binärbäume

Anforderungen

- Eigene `template` Klasse für Binärbäume
- Vorgestellten Konzepte nutzen

Anforderungen

- Eigene `template` Klasse für Binärbäume
- Vorgestellten Konzepte nutzen
- Überlegt inwiefern der Code gut getestet werden kann (wir werden in Teil 3 Unittests ergänzen)