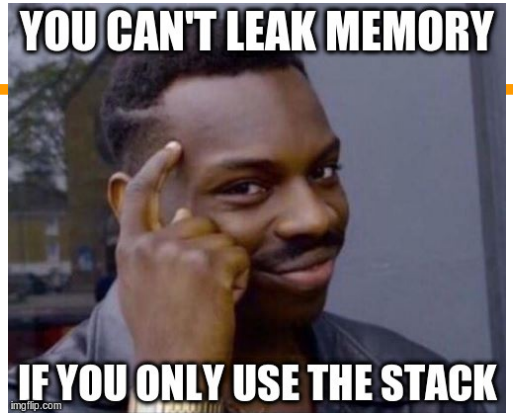
Container und Speicherverwaltung

- „Nackte“ Pointer
- RAII
- Smart Pointer
- Sequenzielle Container
- Assoziative Container
- Iteratoren
- Eigene Speicherverwaltung

„Nackte“ Pointer

- Probleme bei nackten Pointer
  - Unklar, was der Pointer darstellt (Iterator, Pointer auf Objekt/ Array/ Referenz)
  - Keine Unterscheidung zwischen owning/ non-owning
- Compiler kann nicht viel optimieren
- Ungesicherte Zugriffe, kein Bound-Checking
- Zeigerarithmetik unabhängig von Verwendungszweck
- http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rr-ptr
- In C++ ist deshalb die Regel:
  - **Pointer sind non-owning Verweise !!**

RAII



- Resource acquisition is initialization
- Man nutzt den impliziten Konstruktor/Destruktor Aufruf, vor allem bei Scopes, um Ressourcen zu managen.
- http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#r1-manage-resources-automatically-using-resource-handles-and-raii-resource-acquisition-is-initialization
- Vorteile:
  - Keine Speicherlecks bei temporären Variablen
  - Keine Speicherlecks bei unübersichtlichen returns oder exceptions
- **Funktioniert nur, wenn Destruktor auch am Ende des Scopes aufgerufen wird, also nicht bei nackten Zeigern!!**
- https://godbolt.org/g/Gx7k1a

Smart Pointer

- Unique Pointer (C++11)
  - Zeiger besitzt als einziger die Referenz auf das Objekt
  - Kann nicht kopiert, nur verschoben (move) werden
  - Ruft Destruktor auf, sobald das Objekt aus dem Scope verschwindet
  - Vereinfachter Konstruktoraufruf mit make_unique(Parameter)
    - Allerdings erst ab C++14
  - https://godbolt.org/g/62Y3js
  - http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rr-owner

Smart Pointer

- Shared Pointer
  - Pointer zählt Anzahl der Referenzen mit.
  - Eine Kopie erhöht den Zähler
  - Move vermeiden, da der ursprüngliche Pointer Null wird, die Anzahl der Referenzen aber unverändert bleiben
  - Ruft Destruktor auf, sobald die Anzahl der Referenzen 0 wird
- Weak Pointer
  - Non-Owning Referenz für Shared Pointer
- https://godbolt.org/g/b3CpeH

Sequenzielle Container

- Vector
  - Array ähnlicher Container, mit Bound-Checking, variabler Größe und automatischen Destruktor Aufruf
  - Name aus der Vektorisierung (SIMD-Befehle)
  - (Fast) Immer verwenden!!!
  - http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#slcon2-prefer-using-stl-vector-by-default-unless-you-have-a-reason-to-use-a-different-container
- Array (seit C++11)
  - Größe wird zur Kompilierzeit festgelegt
  - Keine dynamische Größenänderung
- https://godbolt.org/g/QjF8VG

Sequenzielle Container

- Pair
  - Container, der exakt 2 verschiedene Objekte aufnehmen kann
  - Möglichkeit, mehr als einen Wert bei einer Funktion zurückzugeben
  - Datentypen müssen zur Kompilierzeit feststehen.
  - Datenypen müssen einen Standartkonstruktor besitzen
- Tuple (C++11)
  - Allgemeinere Version von Pair, der beliebig viele verschiedene Objekte aufnehmen kann
  - Zusammenfassen mehrerer Tupels mittels Tuple_cat
- Tie (C++11)
  - Einfache Möglichkeit, Pair und Tuple-elemente an einzelne Variablen zu binden
- https://godbolt.org/g/k6TxH6

Sequenzielle Container

- List
  - Implementiert als doppelt-verkettete Liste
  - Schnelles zufälliges Einfügen und Löschen
  - Aber nicht sequenziell im Speicher → lange Zugriffszeiten

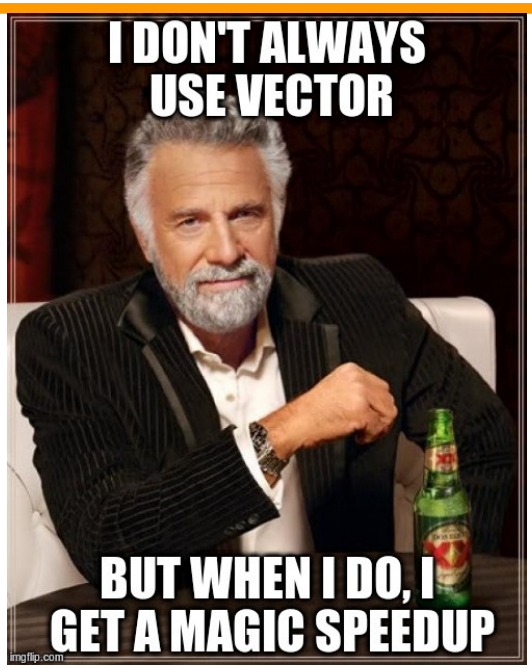
Assoziative Container

- Set
  - Nach Key Sortierter Kontainer
  - Keine Doppelten Keys
    - Ansonsten Multi-Set verwenden
  - Key muss „<“ Operator unterstüzten
- Map
  - Key-Value Container
  - Keine Doppelten Keys
    - Ansonsten Multi-Map verwenden
  - Key muss < Operator unterstüzten

Assoziative Container

- HashMap / HashSet (C++11)
  - Wird als unorderd_map /unorderd_set bezeichnet
  - Unsortierte Elemente
  - Langsames iterieren aber schnelle direktzugriffe
  - Ebenfalls in der Mulit-Variante vorhanden
- https://godbolt.org/g/SUGofB

Container



Empfehlung

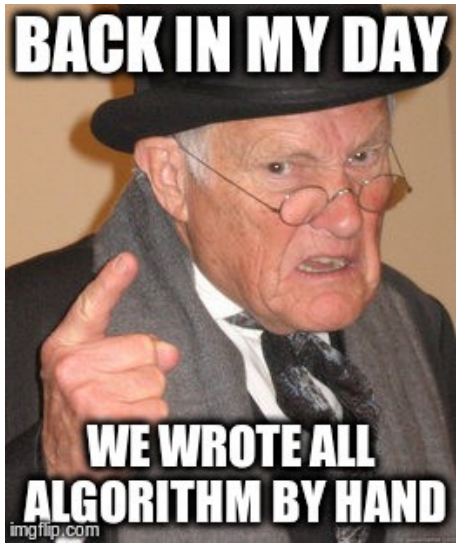
- Möglichst immer Vektor oder Array verwenden
- Bei Key-Value Paaren
  - Hash-map für Lesbarkeit
  - Falls zu langsam vector<pair<>>
- http://quick-bench.com/em6mejtn4Mem3wkRn1yxxw0R1J0

Iteratoren

- Iteratoren sind Pointer um auf Container zu Operieren
- Korrekter Datentyp ist *std::container<typ>::iterator*
- Können vom Container mit *.begin()* und *.end()* erhalten
  - *.end()* ist nicht das letzte Element, sondern eine Position danach
- Dereferenzierung mittels *
- http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#slcon3-avoid-bounds-errors
- https://godbolt.org/g/Vc2fh6
- http://quick-bench.com/pR0ee1uQlaXDlh3RC2njLm_VUrY

Eigene Speicherverwaltung

- Eigene Speicherverwaltung kann entweder durch überladen der new/delete Operationen erzeugt werden oder durch eigene Container-Klassen
- http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#r15-always-overload-matched-allocationdeallocation-pairs
- Man sollte wissen was man tut!!!
- Vorgefertigte Allocator Typen in C++
  - http://en.cppreference.com/w/cpp/memory/allocator
- Ansonsten Github / EASTL / BOOST
  - https://github.com/electronicarts/EASTL
  - http://www.boost.org/doc/libs/1_65_1/doc/html/container/extended_functionality.html#container.extended_functionality.extended_allocators



Nächstes Mal:

Lambda und Algorithmen