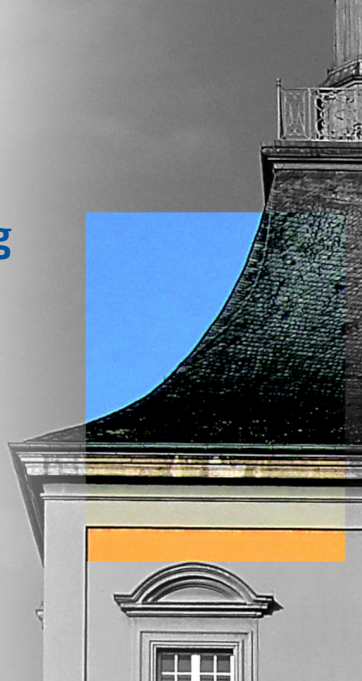
UNIVERSITÄT BONN

Algorithmen und Programmierung

Algorithmen II

Dr. Felix Jonathan Boes
boes@cs.uni-bonn.de
Institut für Informatik



KURZE WIEDERHOLUNG

Abstrakte Datentypen und Datenstrukturen

Erinnerung

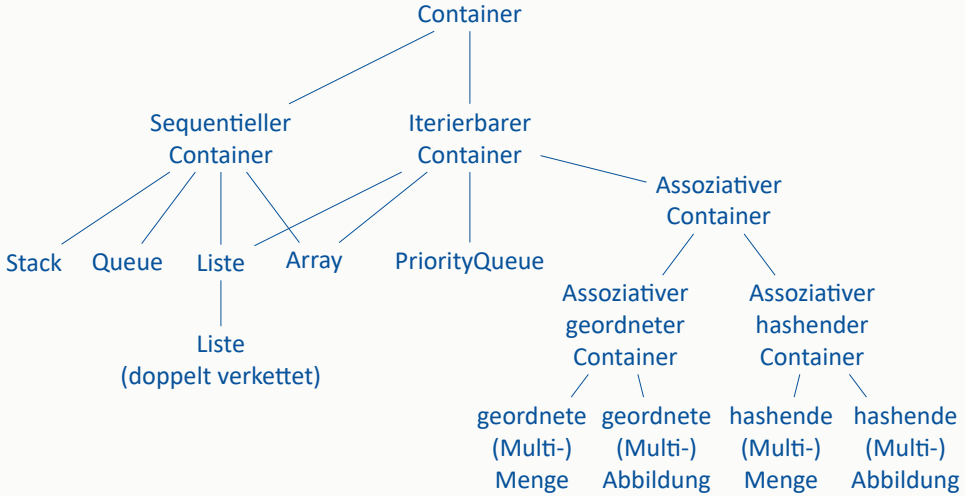
Abstrakte Datentypen modellieren Kollektionen von Objekten, definiert welche Operationen dabei zur Verfügung stehen und wie effizient diese Operationen sind

Datenstrukturen spezifizieren wie Kollektionen von Objekten und zugehörige Operationen verwirklicht werden.

Abstrakte Datentypen werden von Datenstrukturen realisiert

UNIVERSITÄT BONN

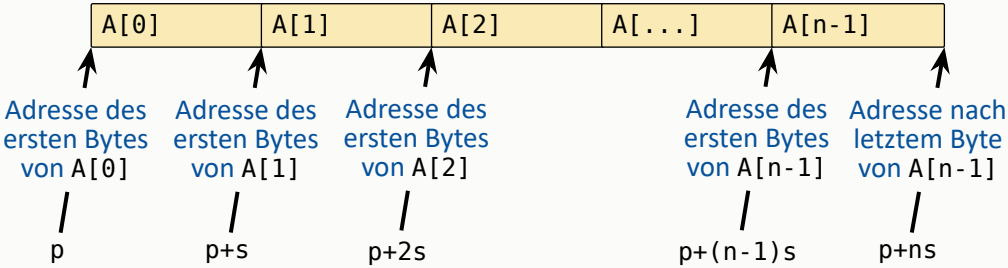
Die (wichtigsten) abstrakten Datentypen im Überblick



UNIVERSITÄT BONN

Realisierung dynamische Arrays

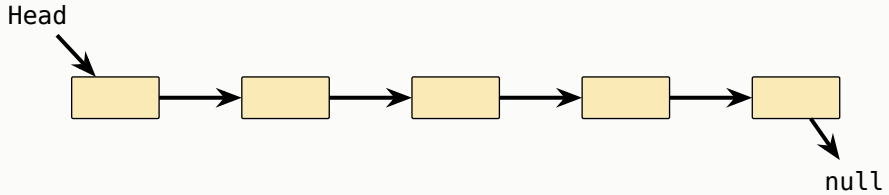
Für Objekte vom Typ T der Größe s wird ein dynamisches Array der Länge n wie folgt realisiert. Das Objekt zum Index i liegt an der Adresse $p + i \cdot s$.



UNIVERSITÄT BONN

Realisierung Liste

Die Objekte einer Liste werden in Nodes (Knoten) gespeichert. Jeder Knoten enthält zusätzlich die Information wer der direkte Nachfolger ist (falls ein Nachfolger existiert). Die Liste merkt sich den Head (Kopf).

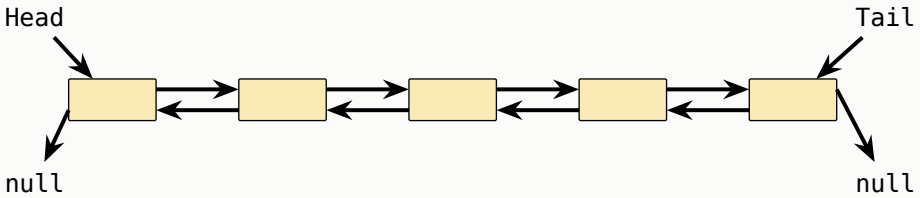


Man kann neue Objekte an beliebigen Stellen mit Aufwand $\mathcal{O}(1)$ einfügen oder entfernen. Dabei muss der jeweilige Vorgänger angegeben werden.

UNIVERSITÄT BONN

Realisierung doppelt verkettete Liste

Die Objekte einer Liste werden in Nodes (Knoten) gespeichert. Jeder Knoten enthält zusätzlich die Information wer der direkte Vorgänger und Nachfolger ist. Die Liste merkt sich den Head (Kopf) und den Tail (Fuß).

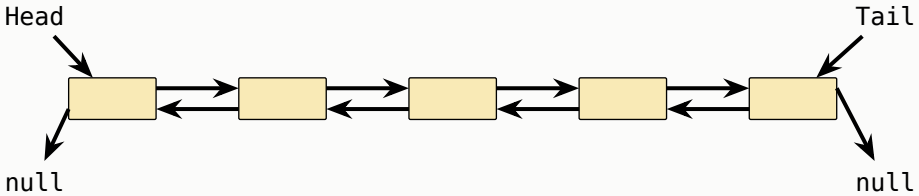


Man kann neue Objekte an beliebigen Stellen mit Aufwand $\mathcal{O}(1)$ einfügen oder entfernen. Dabei muss der jeweilige Vorgänger angegeben werden.

UNIVERSITÄT BONN

Realisierung Queue

Die Objekte einer Liste werden in Nodes (Knoten) gespeichert. Jeder Knoten enthält zusätzlich die Information wer der direkte Vorgänger und Nachfolger ist. Die Liste merkt sich den Head (Kopf) und den Tail (Fuß).



Man kann neue Objekte an beliebigen Stellen mit Aufwand $\mathcal{O}(1)$ hinten einfügen oder vorne entfernen.

UNIVERSITÄT BONN

Realisierung Stack

Ein sequentieller Container ist ein **Stack** falls die folgenden Eigenschaften erfüllt sind.

- Objekte können am Anfang des Stacks eingefügt werden.
- Objekte können am Anfang des Stacks entfernt werden.
- Das Einfügen oder Entfernen am Anfang hat eine durchschnittliche Laufzeit von $\mathcal{O}(1)$.

Stacks werden durch die Datenstruktur realisiert die auch ein Array oder eine Liste realisieren.

Dynamische Speicherverwaltung

Dynamische Speicherverwaltung im Allgemeinen

UNIVERSITÄT BONN

Adressvariablen und Heapspeicher im Allgemeinen

Wir haben verstanden dass es konzeptionell nötig ist, die Lebensdauer einer Variablen von der Funktionsrückkehr zu entkoppeln. Um dies in imperativen und objektorientierten Programmiersprachen zu erreichen, werden **Adressvariablen** und der **Heapspeicher** verwendet.

Eine **Adressvariable** speichert **virtuelle Adressen** eines **festgelegten Typs**. Da sich Adressvariablen den Ort eines Objekts merken, werden Adressvariablen auch **Zeiger** genannt. Über die Adresse kann man auf den dort liegenden Objekten operieren.

Der **Heapspeicher** wird verwendet, um Objekte **dynamisch** zu organisieren, also unabhängig vom aktuellen Stackframe. Die jeweilige Programmiersprache definiert wie Objekte auf dem Heap angelegt werden und wie diese wieder entfernt werden.

UNIVERSITÄT BONN

Typische Strategien um Objekte vom Heap zu entfernen

Die folgenden drei Strategien zur Objektentfernung finden oft Anwendung.

Explizites Aufräumen (C, C++) Die Nutzer:innen müssen einen expliziten Befehl abgeben. Hier ist maximale Effizienz und minimale Nachvollziehbarkeit gegeben. Wird das Aufräumen an einer Heapadresse vergessen, lebt der verwendete Heapspeicher bis zum Programmende.

Implizites Aufräumen (C++) Der Speicher wird aufgeräumt falls gar kein Zeiger mehr auf den zugehörigen Heapspeicher zeigt. Hier ist sehr hohe Effizienz und viel Nachvollziehbarkeit gegeben. Aber, falls auf dem Heap ein Kreis von aufeinander zeigenden Zeigern existiert, lebt der verwendete Heapspeicher bis zum Programmende.

Vollautomatisch (Java, Python) Alle Stackvariablen sind Adressvariablen und alle Objekte liegen auf dem Heap. Regelmäßig werden alle Objekte entfernt, die nicht mehr vom Stack aus erreichbar sind. Hier ist weniger Effizienz und sehr hohe Nachvollziehbarkeit gegeben. Der Heapspeicher wird garantiert aufgeräumt.

Haben Sie Fragen?

Dynamische Speicherverwaltung

Dynamische Speicherverwaltung in C++ mit Smart Pointern

Offene Frage

Welche C++-Konzepte und Methoden sollen verwendet werden, um dynamisch reservierten Speicher zu verwalten?

UNIVERSITÄT BONN

Einleitung

In C++ gibt es zwei unterschiedliche Herangehensweisen um Objekte auf dem Heapspeicher zu verwalten.

Heapspeicher, der implizit freigegeben wird, wird mit **Smart Pointern** organisiert. Wir diskutieren Smart Pointer im Detail.

Heapspeicher, der explizit wieder freigegeben werden muss, wird mit **Raw Pointern** sowie den zugehörigen **new**-Expressions und **delete**-Expressions organisiert. Wir diskutieren Raw Pointer nur kurz.

UNIVERSITÄT BONN

Smart Pointer im Überblick

Beim Schreiben moderner C++-Anwendungen sollten stets **Smart Pointer** verwendet werden. Smart Pointer sorgen für wartbareren, fehlerunanfälligeren Code.

- Moderne C++-Projekte die keine betriebssystem-/hardwarenahen Aufgaben lösen verwenden die Smart Pointer: **Shared Pointer**, **Weak Pointer** und **Unique Pointer**.
- Ehemals moderne C++-Projekte die keine betriebssystem-/hardwarenahen Aufgaben lösen verwenden den Smart Pointer: **Auto Pointer**. Dieser wird seit C++17 nicht mehr unterstützt.

Um mit Smart Pointern zu arbeiten, müssen die Header `<memory>` eingebunden werden.

Dynamische Speicherverwaltung

Dynamische Speicherverwaltung in C++ mit Shared Pointer

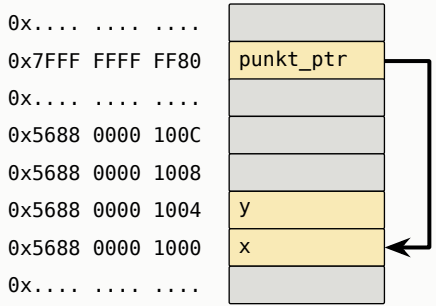
UNIVERSITÄT BONN

Shared Pointer im Speicher

Um Objekte dynamisch auf dem Heap zu verwalten verwenden wir **Shared Pointer**.

```
class Punkt2D {
  ...
private:
  double x;
  double y;
};


void f() {
  shared_ptr<Punkt2D> punkt_ptr;
  punkt_prt = make_shared<Punkt2D>(4.2, 0.9); ←
  ...
}
```



Das erzeugte Objekt wird über einen oder mehrere **Zeiger** (in Form von Shared Pointern) verwaltet. Wenn das Objekt durch keinen Shared Pointer mehr verwaltet wird, wird es „automatisch“ aufgeräumt.

UNIVERSITÄT BONN

Speicherfreigabe via Reference Counting

Shared Pointer erlauben es, dass ein Objekt von mehreren Smart Pointern verwaltet wird. Dabei weiß jedes so verwaltete Objekt, wieviele Shared Pointer es verwalten. Sobald das Objekt von keinem Shared Pointer mehr verwaltet wird wird es „automatisch vom Heap entfernt“[1] Die Technik dahinter heißt **Reference Counting**.

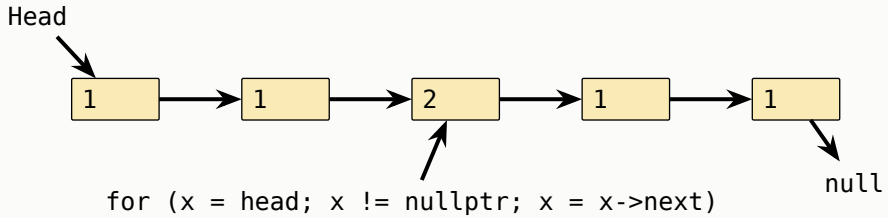
[1] Genauer wird der Destruktur des Objekts aufgerufen und anschließend wird der Speicher freigegeben.

Livedemo

UNIVERSITÄT BONN

Zusammenfassung der Livedemo

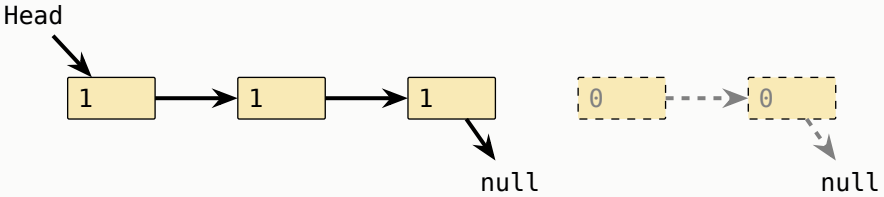
Jeder Knoten wird von mindestens einem Shared Pointer verwaltet. Die Anzahl verwaltenden Shared Pointer wird hier im Konten notiert.



UNIVERSITÄT BONN

Zusammenfassung der Livedemo

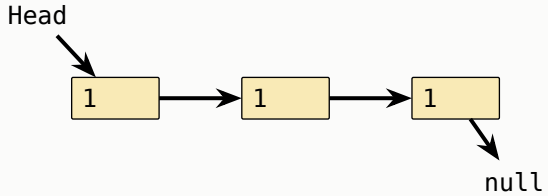
Jeder Knoten wird von mindestens einem Shared Pointer verwaltet. Die Anzahl verwaltenden Shared Pointer wird hier im Konten notiert.



UNIVERSITÄT BONN

Zusammenfassung der Livedemo

Jeder Knoten wird von mindestens einem Shared Pointer verwaltet. Die Anzahl verwaltenden Shared Pointer wird hier im Konten notiert.



UNIVERSITÄT BONN

Shared Pointer erzeugen

Ein neu angelegter Shared Pointer verwaltet im Standardfall kein Objekt.

```
std::shared_ptr<MeineKlasse> ptr;  // Erzeugt ein Shared Pointer der kein Objekt verwaltet.
```

Um ein Shared Pointer zu erzeugen der ein neues Objekt verwaltet, wird die die Funktion `std::make_shared` verwendet. Die Parameter dieser Funktion entsprechen den Konstruktorparametern des zu erzeugenden Objekts.

```
std::shared_ptr<MeineKlasse> ptr;  // Erzeugt ein Shared Pointer der kein Objekt verwaltet.
ptr = std::make_shared<MeineKlasse>( /* Konstruktorparameter von MeineKlasse */ );
```

UNIVERSITÄT BONN

Zugriff auf das verwaltete Objekt I

Wenn ein Shared Pointer `ptr` ein Objekt verwaltet, dann erhält man mit `*ptr` eine Referenz auf das verwaltete Objekt.

```
std::shared_ptr<int> ptr = std::make_shared<int>(42);
std::cout << *ptr << std::endl; // Druckt 42
*ptr = 55;                      // Setzt das verwaltete Objekt auf 55
std::cout << *ptr << std::endl; // Druckt 55
```

UNIVERSITÄT BONN

Zugriff auf das verwaltete Objekt II

Wenn ein Shared Pointer `ptr` ein Objekt verwaltet, dann greift man auf die Member mit `(*ptr).member` zu. Eine leserliche Abkürzung dafür ist `ptr->member`.

```
class MeineKlasse {
public:
  // ...
  int eine_member_fkt();
  // ...
};

std::shared_ptr<MeineKlasse> ptr = std::make_shared<MeineKlasse>( /* Konstruktorparameter */ );
std::cout << (*ptr).eine_member_fkt() << std::endl; // Druckt eine Zahl
std::cout << ptr->eine_member_fkt()   << std::endl; // Druckt eine Zahl
```

UNIVERSITÄT BONN

Mit Shared Pointer arbeiten

Um ein Shared Pointer bei der Verwaltung eines bereits verwalteten Objekts zu beteiligen, verwendet man den Zuweisungsoperator =

```
std::shared_ptr<int> ptr = std::make_shared<int>(42);
std::shared_ptr<int> anderer_ptr = ptr; // Beide verwalten dasselbe Objekt.
*anderer_ptr = 55;                      // Setzt das verwaltete Objekt auf 55
std::cout << *ptr << std::endl;         // Druckt 55
```

Um die Anzahl der verwaltenden Shared Pointer zu erfahren, stellen Shared Pointer die Memberfunktion `use_count` zur Verfügung.

```
std::shared_ptr<int> ptr = std::make_shared<int>(42);
std::shared_ptr<int> anderer_ptr = ptr;    // Beide verwalten dasselbe Objekt.
std::cout << ptr.use_count() << std::endl; // Druckt 2
```

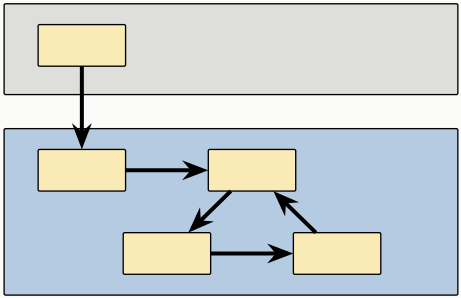
Um die Verwaltung eines Objekts aufzugeben, stellen Shared Pointer die Memberfunktion `reset` zur Verfügung.

UNIVERSITÄT BONN

Zur impliziten Objektvernichtung I

Bei Shared Pointern wird ein verwaltetes Objekt vernichtet, sobald gar kein Shared Pointer mehr auf das Objekt zeigt.

Durch den unachtsamen Entwurf eines Programms kann es dazu kommen, dass im Heap eine Folge von Shared Pointern existieren, die zyklisch aufeinander zeigen ohne vom Stack aus erreichbar zu sein.

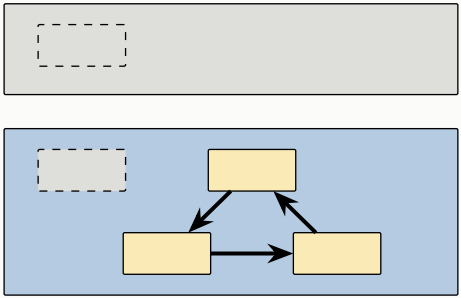


UNIVERSITÄT BONN

Zur impliziten Objektvernichtung I

Bei Shared Pointern wird ein verwaltetes Objekt vernichtet, sobald gar kein Shared Pointer mehr auf das Objekt zeigt.

Durch den unachtsamen Entwurf eines Programms kann es dazu kommen, dass im Heap eine Folge von Shared Pointern existieren, die zyklisch aufeinander zeigen ohne vom Stack aus erreichbar zu sein.

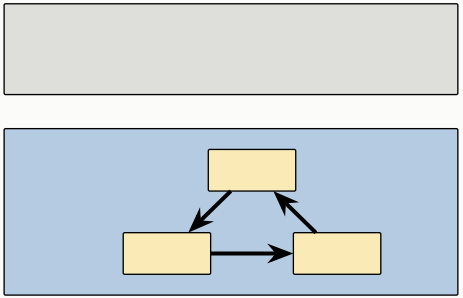


UNIVERSITÄT BONN

Zur impliziten Objektvernichtung I

Bei Shared Pointern wird ein verwaltetes Objekt vernichtet, sobald gar kein Shared Pointer mehr auf das Objekt zeigt.

Durch den unachtsamen Entwurf eines Programms kann es dazu kommen, dass im Heap eine Folge von Shared Pointern existieren, die zyklisch aufeinander zeigen ohne vom Stack aus erreichbar zu sein.



UNIVERSITÄT BONN

Zur impliziten Objektvernichtung II

Falls im Heap eine Folge von Shared Pointern existiert, welche

- zyklisch aufeinander zeigen und
- vom Stack aus (implizit) nicht erreichbar sind

dann spricht man von einer **zirkulären Zeigerabhängigkeit**.

Objekte die von Zeigern verwaltet werden, die in einer zirkulären Abhängigkeit zueinander stehen, werden **nicht vom Heap entfernt**.

Haben Sie Fragen?

UNIVERSITÄT BONN

Weak Pointer

Um sich nur die Adresse eines Objekts zu merken, ohne es zu verwalten, verwendet man **Weak Pointer**.

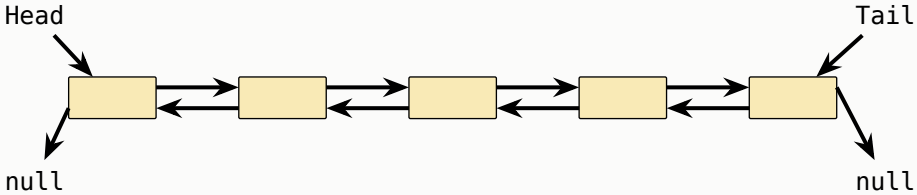
Durch die geschickte Verwendung von Shared Pointern und Weak Pointern können zyklische Abhängigkeiten praktisch immer verhindert werden.

Livedemo

UNIVERSITÄT BONN

Zusammenfassung der Livedemo I

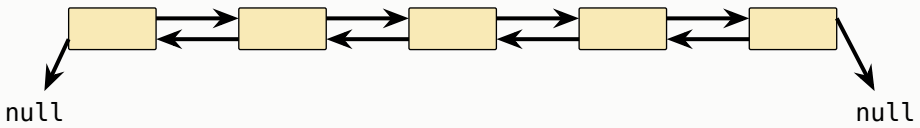
Falls wir eine doppelt verkettete Liste ausschließlich mit Shared Pointer realisieren, dann führt das Vergessen von Kopf und Fuß zu einer zyklischen Abhängigkeit.



UNIVERSITÄT BONN

Zusammenfassung der Livedemo I

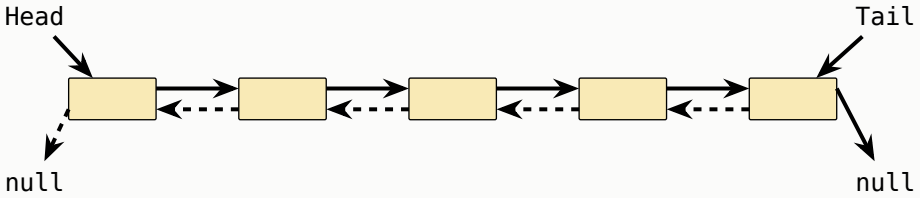
Falls wir eine doppelt verkettete Liste ausschließlich mit Shared Pointer realisieren, dann führt das Vergessen von Kopf und Fuß zu einer zyklischen Abhängigkeit.



UNIVERSITÄT BONN

Zusammenfassung der Livedemo II

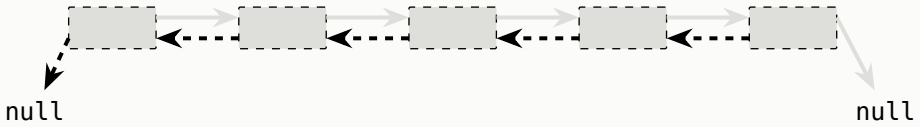
Falls wir eine doppelt verkettete Liste geschickt mit Shared Pointer und Weak Pointern realisieren, dann führt das Löschen von Kopf und Fuß nicht zu einer zyklischen Abhängigkeit.



UNIVERSITÄT BONN

Zusammenfassung der Livedemo II

Falls wir eine doppelt verkettete Liste geschickt mit Shared Pointer und Weak Pointern realisieren, dann führt das Löschen von Kopf und Fuß nicht zu einer zyklischen Abhängigkeit.



UNIVERSITÄT BONN

Zusammenfassung der Livedemo II

Falls wir eine doppelt verkettete Liste geschickt mit Shared Pointer und Weak Pointern realisieren, dann führt das Löschen von Kopf und Fuß nicht zu einer zyklischen Abhängigkeit.

UNIVERSITÄT BONN

Zugriff auf Objekte mit Weak Pointern

Weak Pointer können nicht direkt auf verwaltete Objekte zugreifen. Um auf ein verwaltetes Objekt zuzugreifen, muss aus dem Weak Pointer erst ein zugehöriger Shared Pointer erzeugt werden. Dazu wird die Memberfunktion `lock()` verwendet.

```
// Legt neuen Shared Pointer an
std::shared_ptr<int> ptr = std::make_shared<int>(42);
// Legt Weak Pointer an, der auf dasselbe Objekt zeigt
std::weak_ptr<int>   anderer_ptr = ptr;
// Objekt wird von EINEM Pointer verwaltet
std::cout << ptr.use_count() << std::endl;
// Illegal! Weak Pointer erlaubt keinen direkten Zugriff!
std::cout << *anderer_ptr << std::endl;
// Zugriff ist nach Erzeugung des temporären Shared Pointer zulässig.
// Das Objekt wird in diesem Moment von ZWEI Pointern verwaltet.
std::cout << (anderer_ptr.lock()).use_count() << std::endl;
// Der temporäre Shared Pointer verwaltet das Objekt nicht mehr.
// Das Objekt wird also von EINEM Pointer verwaltet
std::cout << ptr.use_count() << std::endl;
```

Haben Sie Fragen?

UNIVERSITÄT BONN

Unique Pointer

Objekte, die durch genau einen Zeiger verwaltet werden sollen, werden von **Unique Pointern** verwaltet.

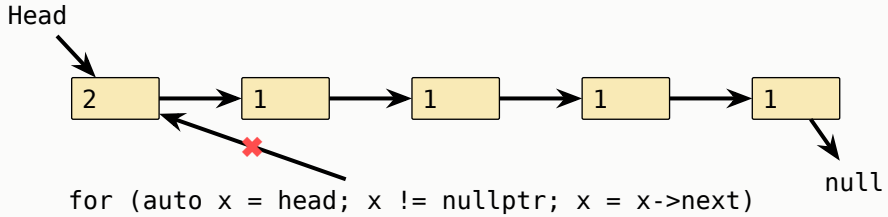
Die korrekte und nachvollziehbare Verwendung von Unique Pointern muss einstudiert werden.

Livedemo

UNIVERSITÄT BONN

Zusammenfassung der Livedemo

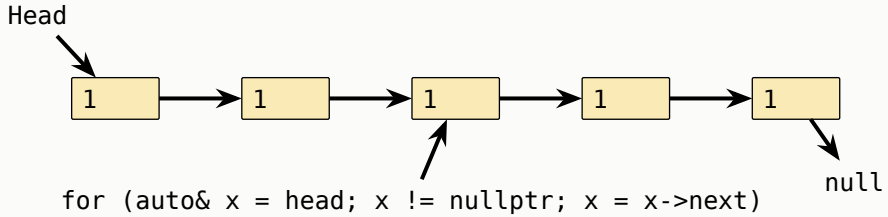
Jeder Knoten wird von genau einem Unique Pointer verwaltet. Deshalb ist der Zugriff über eine Kopie unzulässig, aber der Zugriff über eine Referenz erlaubt.



UNIVERSITÄT BONN

Zusammenfassung der Livedemo

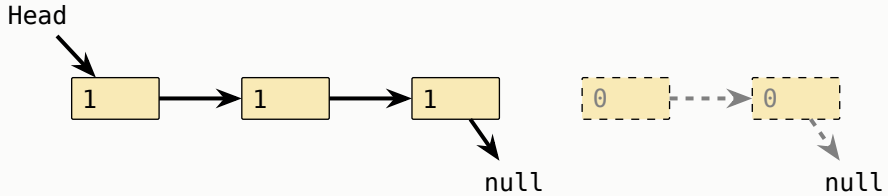
Jeder Knoten wird von genau einem Unique Pointer verwaltet. Deshalb ist der Zugriff über eine Kopie unzulässig, aber der Zugriff über eine Referenz erlaubt.



UNIVERSITÄT BONN

Zusammenfassung der Livedemo

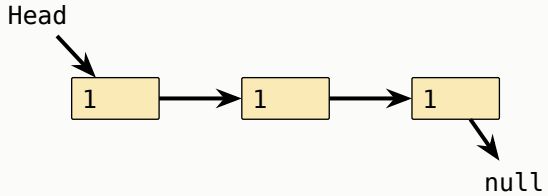
Jeder Knoten wird von genau einem Unique Pointer verwaltet. Deshalb ist der Zugriff über eine Kopie unzulässig, aber der Zugriff über eine Referenz erlaubt.



UNIVERSITÄT BONN

Zusammenfassung der Livedemo

Jeder Knoten wird von genau einem Unique Pointer verwaltet. Deshalb ist der Zugriff über eine Kopie unzulässig, aber der Zugriff über eine Referenz erlaubt.



UNIVERSITÄT BONN

Unique Pointer erzeugen

Um ein Unique Pointer zu erzeugen der ein neues Objekt verwaltet, wird die die Funktion `std::make_unique` verwendet. Die Parameter dieser Funktion entsprechen den Konstruktorparametern des zu erzeugenden Objekts.

```
std::unique_ptr<MeineKlasse> ptr;  // Erzeugt ein Shared Pointer der kein Objekt verwaltet.
ptr = std::make_unique<MeineKlasse>( /* Konstruktorparameter von MeineKlasse */ );
```

Wir erwarten, dass eine Zuweisung = dazu führt, dass beide Unique Pointer dasselbe Objekt verwalten. Das führt somit zu einem Fehler.

```
// Erzeuge Unique Pointer die kein Objekt verwalten.
std::unique_ptr<MeineKlasse> ptr;
std::unique_ptr<MeineKlasse> anderer_ptr;
// Erezuge Objekt auf dem Heap welches von ptr verwaltet wird
ptr = std::make_unique<MeineKlasse>( /* Konstruktorparameter von MeineKlasse */ );
// Illegal, da nur ein Unique Pointer das Objekt verwalten darf
anderer_ptr = ptr;
```

UNIVERSITÄT BONN

Mit Unique Pointern arbeiten

Auswahl wichtiger Memberfunktionen von Unique Pointern:

- `reset` entfernt das aktuell verwaltete Objekt von Heap
- `swap` tauscht die verwalteten Objekte zwischen zwei Unique Pointern aus (die Objekte behalten dabei Ihre Position im Heap)
- Der Unique Pointer kann als `bool`-Wert interpretiert werden. Dieser ist genau dann **`true`** wenn der Unique Pointer ein Objekt auf dem Heap verwaltet.

Haben Sie Fragen?

UNIVERSITÄT BONN

Zusammenfassung

Um in C++ Objekte auf dem Heap zu verwalten, verwendet man **Smart Pointer**.

- **Shared Pointer** verwalten gemeinsam ein Objekt. Das (zur Verwaltung angelegte) Objekt wird aufgeräumt, sobald kein Shared Pointer das Objekt mehr verwaltet.
- **Weak Pointer** merken sich die Adresse eines Objekts. Weak Pointer können nicht direkt auf das Objekt zugreifen. Durch die Erzeugung eines zugehörigen Shared Pointer, kann das Objekt auch verwaltet werden.
- **Unique Pointer** verwalten allein ein Objekt. Das (zur Verwaltung angelegte) Objekt wird aufgeräumt, sobald der Unique Pointer das Objekt nicht mehr verwaltet.

Achtung: Objekte die durch **zirkulären abhängige Zeiger** verwaltet werden, werden nicht vom Heap entfernt. Die Vermeidung von **zirkulären Zeigerabhängigkeiten** muss durch die Programmierer:innen sichergestellt werden.

Zusammenfassung

Mit der Hilfe von Smart Pointern werden Objekte auf dem Heap verwaltet

Diese Objekte werden explizit angelegt und implizit vernichtet

Haben Sie Fragen?

Dynamische Speicherverwaltung

Dynamische Speicherverwaltung in C++ mit Raw Pointern

Ausblick

Es gibt sehr gute, seltene Gründe um mit Raw Pointern zu arbeiten

Mit der Hilfe von Raw Pointern werden Objekte auf dem Heap verwaltet

Diese Objekte werden explizit angelegt und explizit vernichtet

Raw Pointer im Überblick

Beim Schreiben von betriebssystem-/hardwarenahen C++-Anwendungen, oder wenn es andere gute Gründe gibt, können **Raw Pointer**[2] verwendert werden.

Raw Pointer sind hardwarenäher und führen deshalb, bei ungeübter oder unsachgemäßer Nutzung, zu unerwartetem Verhalten. Das unerwartete Verhalten tritt meist erst nach dem Kompilieren in Erscheinung. Die so entstehenden Fehler sind nur mit viel Übung zu finden und zu vermeiden.

Hinweis: Nur weil in einem C++-Projekt Raw Pointer verwendet werden, sind Autoren nicht automatisch klug oder erfahren. Oft werden Raw Pointer verwendet, weil betriebssystem-/hardwarenahe Programmierung nötig istoder die Vorteile von Smart Pointern unbekannt sind oder diese nicht gewertschätz werden.

Wir diskutieren hier **Raw Pointern** nur sehr oberflächlich.

[2] Raw Pointer werden auch manchmal C-Pointer genannt

UNIVERSITÄT BONN

Heapobjekte erzeugen und nutzen

Um ein Objekt vom Typ T auf dem Heap anzulegen, wird die Expression **new** T; verwendet. Der Typ dieser Expression ist der Raw Pointer T*.

```
class MeineKlasse {
public:
  // ...
  int eine_member_fkt();
  // ...
};

MeineKlasse* ptr;  // Erzeugt einen Raw Pointer der auf Objekte vom Typ MeineKlasse zeigen kann.
ptr = new MeineKlasse ( /* Konstruktorparameter von MeineKlasse */ );
```

Der Zugriff auf das erzeugte Objekt ist wie bei Smart Pointern.

```
std::cout << (*ptr).eine_member_fkt() << std::endl; // Druckt eine Zahl
std::cout << ptr->eine_member_fkt()   << std::endl; // Druckt eine Zahl
```

Heapobjekte vernichten

Objekte die mit **new** auf dem Heap erzeugt wurden, müssen explizit mit **delete** vernichtet werden. Objekte die nicht vernichtet werden, bleiben im Speicher erhalten.

```
class MeineKlasse {
public:
  // ...
  int eine_member_fkt();
  // ...
};

MeineKlasse* ptr;  // Erzeugt einen Raw Pointer der auf Objekte vom Typ MeineKlasse zeigen kann.
ptr = new MeineKlasse ( /* Konstruktorparameter von MeineKlasse */ );
...
delete ptr;        // Vernichtet das vorher angelegte Heapobjekt
```

Haben Sie Fragen?

Zusammenfassung

Wir haben gelernt wie Heapobjekte erzeugt, verwaltet und vernichtet werden

Zur dynamischen Speicherverwaltung sollen üblicherweise Smart Pointer verwendet werden

Bei Smart Pointern muss auf zirkuläre Unabhängigkeit geachtet werden

Raw Pointern bieten die größte Flexibilität und Fehleranfälligkeit

Abstrakte Datentypen und Datenstrukturen

Sequentielle Container und deren Realisierung in C++ - Liste

Ziel

Realisierung des abstrakte Datentyp LISTE in C++

UNIVERSITÄT BONN

Liste realisiert durch Datenstruktur

```
// A Node stores its succesor and some data
class Node{
public:
  Node(int data);

// Für ein nachvollziehbares Projekt sollte der folgende, schlechte Stil (!)
// nicht verwendet werden. Der schlechte Stil (Membervariablen sind public) wird
// hier nur deshalb verwendet, um die Datenstruktur schnell und live zu entwickeln.
public:
  std::shared_ptr<Node> next;
  int data_;
};

// In the end, the Nodes are dynamically organized on the heap.
// Therefore, we use Shared Pointer for the organization of Nodes.
// It will turn out, that shared_ptr is the best way because we want
// to iterate through the nodes (i.e., we want multiple references at once).

// The type "std::shared_ptr<Node>" is abbriviated with "Nodeptr"
typedef std::shared_ptr<Node> Nodeptr;
```

UNIVERSITÄT BONN

Liste realisiert durch Datenstruktur

```
class Liste{
public:
  Liste();

  // Insert and delete Nodes
  Nodeptr insert_front(int x);
  Nodeptr insert_after(const Nodeptr&, int x);
  Nodeptr remove_front();
  Nodeptr remove_after(const Nodeptr&);

  // Get the next Node of the current List
  Nodeptr next(const Nodeptr&);

  // Print the current List
  void print() const;

private:
  Nodeptr head;
};
```

```
Nodeptr Liste::insert_front(int x) {
  // head
  //   |
  //   v
  // [ a ] -- next --> [ b ] -- next --> [ c ] -- next --> ...
  //
  //   n
  //   |
  //   v
  // [ x ] -- next --> null
  Nodeptr n = std::make_shared<Node>(x);

  // head
  //   |
  //   v
  //  ...
  //
  //   n
  //   |
  //   v
  // [ x ] -- next --> [ a ] -- next --> [ b ] -- next --> [ c ] -- next --> ...
  n->next = head;

  // head
  //   |
  //   v
  // [ x ] -- next --> [ a ] -- next --> [ b ] -- next --> [ c ] -- next --> ...
  //
  //   n
  //   |
  //   v
  //  ...
  head = n;

  return head;
```

UNIVERSITÄT BONN

Liste realisiert durch Datenstruktur

```
void Liste::print() const {
  Nodeptr current = head;
  while (current) {
    std::cout << current->data_ << " -> " ;
    current = current -> next;
  }
  std::cout << "null" << std::endl;
}
```

UNIVERSITÄT BONN

Liste realisiert durch Datenstruktur

```
Nodeptr Liste::remove_front() {
  // If the List is empty, there is nothing to do
  if (not head){
    return nullptr;
  }

  // The head is going to be removed
  Nodeptr removed = head;
  // The new head is the successor of the current head
  head = head->next;
  // Return the removed Node
  return removed;
}
```

Zusammenfassung

Der abstrakte Datentyp LISTE wurde in C++ implementiert

Haben Sie Fragen?

Abstrakte Datentypen und Datenstrukturen

Sequentielle Container und deren Realisierung in C++

Ziel

Realisierung der abstrakten Datentyp DOPPELT VERKETTETE LISTE und QUEUE in C++

UNIVERSITÄT BONN

Sequentielle Container und deren Realisierung in C++

Die doppelt verkettete Liste wird analog zur einfach verketteten Liste in C++ realisiert. Um zirkuläre Abhängigkeiten zu vermeiden, werden Shared Pointer verwendet, die in Richtung Fuß zeigen und Weak Pointer, die in Richtung Kopf zeigen.

Die Queue wird ähnlich zur (einfachen oder doppelt verketteten) Liste implementiert.

Zusammenfassung

Den abstrakten Datentyp DOPPELT VERKETTETE LISTE und QUEUE wird in C++ ähnlich wie die LISTE implementiert

Die korrekte Auswahl von Shared Pointern und Weak Pointern verhindern zirkuläre Abhängigkeiten

Haben Sie Fragen?