Zeiger und Arrays in C++

IPI Tutorium WS 19/20

Anders als in früheren IPI Vorlesungen ~~habt ihr leider das Pech~~ / ~~müsst ihr unglücklicherweise~~ dürft ihr euch an der Herausforderung erfreuen, euch intensiv mit Zeigern auseinanderzusetzen. Diese Zusammenfassung soll euch das ganze Thema an ein paar einfachen Beispielen näher bringen.

Was sind Zeiger?

Überhaupt nicht zu verstehen, was es mit Zeigern (engl. Pointern) auf sich hat, ist zu Beginn leider ganz normal. Zunächst sind Zeiger aber eigentlich auch nur Variablen, wie ihr sie bereits kennt. Allerdings haben diese Variablen nun einen speziellen Zweck. Ein Zeiger enthält nämlich nicht einfach nur eine Zahl, sondern zusätzlich die Information, dass es sich bei dieser Zahl um eine Adresse handelt. Mit Adresse meine ich hier die Stelle, an der die Variable im Arbeitsspeicher eures Computer steht. Ein Zeiger alleine ergibt also wenig Sinn, er wird erst nützlich, wenn er die Adresse einer anderen Variablen bzw. eines anderen Objektes enthält. Das ganze sieht bspw. so aus:

```
int zahl = 5;
int* zeiger = &zahl;
```

Die Variable zahl ist eine übliche Integer-Variable. zeiger hingegen ist (wie der Name vermuten lässt) ein Zeiger, der nun die Adresse, der Variablen zahl enthält. Man sagt in diesem Fall "zeiger zeigt auf zahl". Dabei macht das & nichts anderes, als die Adresse der Variablen auszugeben. Deshalb nennt man & in diesem Zusammenhang den Adressoperator.

Wie geht man nun mit Zeigern um? Betrachten wir dazu das folgende Beispiel:

```
std::cout << zahl << std::endl;
std::cout << zeiger << std::endl;
```

Was ist die Ausgabe des Programms? Die erste Zeile ist klar: es wird einfach 5 ausgegeben. Die Ausgabe der zweiten Zeile ist jedoch sowas wie 0x7fff31f823d0. Auch wenn es erstmal nicht so aussieht, aber das ist einfach nur eine Zahl (in sog. Hexadezimalschreibweise). Bei dieser Zahl handelt es sich um die Arbeitsspeicheradresse, die in zeiger gespeichert ist, d. h. die Adresse, an der die Variable zahl steht. Mit dieser Adresse können wir natürlich nichts anfangen. Um nun den Wert der Variablen Zahl auszulesen, geht man folgendermaßen vor

```
std::cout << *zeiger << std::endl;
```

Die Ausgabe hier ist jetzt wie erwartet 5. Man beachte, dass der * Operator hier jetzt eine andere Bedeutung hat, also oben! Oben gehört der Operator quasi zum Typ int dazu und macht klar, dass es sich bei der Variablen um einen Zeiger handelt. Hier sorgt der Operator allerdings dafür, dass nicht die Adresse der Variablen auf die der Zeiger zeigt ausgegeben wird, sondern deren Inhalt. Aus diesem Grund heißt der Operator in diesem Fall Dereferenzierungsoperator.

Zeiger können natürlich auch für Variablen anderer Typen definiert werden

```
double dValue = 2.0;
double* dPointer = &dValue;
```

und auch für Objekte, d. h. Instanzen eines struct (und später auch einer class)

```
struct Complex {
  double real;
  double imag;
};

Complex c;
c.real = 2;

Complex* cPointer = &c;

std::cout << cPointer->real << std::endl;
// Ausgabe: 2
```

In diesem Beispiel sehen wir auch, wie sich der Zugriff auf die Member eines Struct ändert, wenn man zeiger benutzt (c.real = 2 vs. cPointer->real).

Wofür benutzt man Zeiger?

Nachdem wir geklärt haben, was Zeiger überhaupt sind, betrachten wir jetzt ein paar Beispiele, die zeigen (lol) wofür man Zeiger benutzt.

Oben haben wir bisher Pointer nur verwendet, um auf die referenzierte Variable zuzugreifen und sie auszulesen. Man kann die Variablen aber natürlich auch verändern. Dazu muss man den Pointer, wie auch schon beim Lesen, zunächst dereferenzieren.

```
int zahl;
int* zeiger = &zahl;

*zeiger = 10;

std::cout << zahl << std::endl;
// Ausgabe: 10
```

Das wird vor allem interessant, wenn Zeiger an Funktionen übergeben werden. Betrachten wir zunächst ein Beispiel ohne Zeiger.

```
struct GanzGanzGrosseMatrix {
  ...
};

bool ist_invertierbar(GanzGanzGrossematrix matrix) {
  if (...)
  return true;
  else
  return false;
}

int main() {
  GanzGanzGrossematrix m(10000); // 10000 . Zeilen x 10000 Spalten

  if (ist_invertierbar(m))
  std::cout << "Matrix ist invertierbar";
}
```

Die struct soll hier einen Datentyp für eine sehr große Matrix darstellen. Hat die Matrix Größe 10000x10000 und speichert floats, dann bräuchte die Matrix ~ 40GB Platz im Arbeitsspeicher. Wird nun die Funktion ist_invertierbar aufgerufen, so müssen (je nach Implementierung der struct) unter Umständen 40GB im Arbeitsspeicher hin- und herkopiert werden, was offensichtlich viel zu ineffizient ist. Implementiert man die Funktion allerdings so

```
bool ist_invertierbar(GanzGanzGrossematrix *matrix) {
  if (...)
  return true;
  else
  return false;
}
```

und ruft sie so auf

```
if (ist_invertierbar(&m))
```

dann muss lediglich die Adresse des Objekts matrix kopiert werden, also nur ein paar Bytes.

C++ bietet dazu allerdings eine schönere Alternative an, nämlich die sog. Referenzen. Referenzen sind syntaktischer Zucker (engl. syntactic sugar) für solche Beispiele wie oben, d. h. sie ermöglichen es dem/der Programmierer*in mithilfe einer einfacheren Schreibweise, das selbe Ziel zu erreichen. Das sieht dann so aus:

```
bool ist_invertierbar(GanzGanzGrossematrix &matrix) {
  if (...)
  return true;
  else
  return false;
```

```
}
...
if (ist_invertierbar(m))
```

Der große Vorteil von Referenzen gegenüber Zeigern ist, dass man am Aufruf der Funktion nichts ändern muss (im Vergleich zur Version ohne Zeiger). Aber Achtung! Der &-Operator hat hier eine andere Bedeutung als oben. Hier wird damit verdeutlicht, dass das Argument eine Referenz ist und nicht wie oben, die Adresse einer Variablen zurückgegeben. Ein weiterer Vorteil, bzw. eher ein Nachteil von Zeigern ist, dass man mit der Zeiger-Version auch versehentlich die Funktion mit einem Zeiger aufrufen kann, der auf gar kein Objekt zeigt:

```
GanzGanzGrossematrix* matrixZeiger;

if (ist_invertierbar(matrixZeiger))
```

Versucht die Funktion `ist_invertierbar` auf diesen Zeiger zuzugreifen (indem sie bspw. versucht den Zeiger zu dereferenzieren), dann bricht das Programm (sehr wahrscheinlich) mit einem sog. Segmentation-Error ab. Was in einem nicht initialisierter Zeiger für eine Adresse steht ist immer unterschiedlich, wenn man Glück hat enthält der Zeiger einfach nur den Wert 0 (der sog. Nullzeiger). Das kann bei Referenzen nicht passieren, da man keine Referenzen erstellen kann, die keiner Variablen zugordnet sind.

Einfach Verkettete Listen. Für die Übungsaufgaben sind besonderns sog. einfach und doppelt verkettete Listen von Bedeutung. Diese können mithilfe von Zeigern sehr effizient implementiert werden.

Zunächst betrachten wir einfach verkettete Listen. Dazu definieren wir als erstes eine `struct`, die ein Element der Liste repräsentiert. Hier wollen wir `int`-Variablen in der Liste speichern.

```
struct Element {
  int value;
  Element* next = 0;
};
```

Ein Element besteht also einerseits aus dem Wert, der im Element steht und andererseits aus einem Zeiger auf seinen Nachfolger. Wir setzen hier den Zeiger auf das nächste Element zunächst auf 0, um später prüfen zu können, ob ein Elemente einen Nachfolger hat oder nicht. Eine einfach verkettete Liste ist dann eine Liste, die einfach einen Zeiger auf das erste Element enthält. Da dieser Zeiger dann auf das nächste Element zeigt, und dieser wiederum auf das nächste etc. kann man so die Liste von vorne nach hinten durchlaufen.

```
struct List {
  Element* first = 0;
}
```

Um nun bspw. alle Elemente einer einfach verketteten Liste auszugeben, kann man wie folgt vorgehen:

```
void printList(List list) {
  Element *current = list.first;
  while(current != 0) {
    std::cout << current->value << " ";
    current = current->next;
  }
  std::cout << std::endl;
}
```

Um die Funktion auszuprobieren müssen wir ein paar Elemente erstellen und in die Liste schreiben (zum Beispiel ans Ende der Liste). Idealerweise hätten wir gerne eine Funktion die das für uns übernimmt. Diese Funktion bekommt einerseits die Liste (als Zeiger, da wir ja wirklich die Liste selbst verändern wollen und nicht nur eine übergebene Kopie) und andererseits den Wert, der eingefügt werden soll.

```
void insert(List *list, int value) {
  Element *newElement = new Element;
  newElement->value = value;

  Element *current = list->first;
  while(current->next != 0) {
    current = current->next;
  }

  current->next = newElement;
}
```

Wir erstellen also ein neues Element, weisen ihm den übergebenen Wert zu und durchlaufen die Liste von vorne bis zum Ende der Liste und Fügen das Element dort als Nachfolger des derzeit letzen Elements ein.

Diese Funktion hat zwei Probleme: zum einen funktioniert sie nicht, wenn die Liste noch leer ist und zum anderen dauert das Einfügen umso länger, je länger die Liste ist. Deshalb fügt man in einer einfach verketteten Liste Element üblicherweise am Anfang ein. Die Vorgehensweise ist dabei folgendermaßen (ich gebe euch hier kein Beispiel, das könnt ihr mal selbst probieren): Prüfe, ob Liste leer ist. Falls ja, dann setze einfach das erste Element der Liste auf das neu erstellte Element. Falls nein, dann mache das derzeit erste Element der Liste zum Nachfolger des neuen Elements und aktualisiere das erste Element der Liste. (Frage: Welche Komplexität hat das Einfügen am Ende der Liste und welche das Einfügen am Anfang?)

Außerdem sollte einem sofort ein weiteres Problem auffallen. In der obigen Funktion erstellt man neue Element mit new. Wie in der Vorlesung besprochen, muss es zu jedem new immer auch ein delete geben. Das heißt ihr müsst auf jeden Fall noch eine Funktion schreiben, die die ganze Liste löscht (d.h. die Liste durchläuft und jedes für jedes Element

Einfach Verkettete Listen. Für die Übungsaufgaben sind besonderns sog. einfach und doppelt verkettete Listen von Bedeutung. Diese können mithilfe von Zeigern sehr effizient implementiert werden.

Zunächst betrachten wir einfach verkettete Listen. Dazu definieren wir als erstes eine struct, die ein Element der Liste repräsentiert. Hier wollen wir int-Variablen in der Liste speichern.

```
struct Element {
  int value;
  Element* next = 0;
};
```

Ein Element besteht also einerseits aus dem Wert, der im Element steht und andererseits aus einem Zeiger auf seinen Nachfolger. Wir setzen hier den Zeiger auf das nächste Element zunächst auf 0, um später prüfen zu können, ob ein Elemente einen Nachfolger hat oder nicht. Eine einfach verkettete Liste ist dann eine Liste, die einfach einen Zeiger auf das erste Element enthält. Da dieser Zeiger dann auf das nächste Element zeigt, und dieser wiederum auf das nächste etc. kann man so die Liste von vorne nach hinten durchlaufen.

```
struct List {
  Element* first = 0;
}
```

Um nun bspw. alle Elemente einer einfach verketteten Liste auszugeben, kann man wie folgt vorgehen:

```
void printList(List list) {
  Element *current = list.first;
  while(current != 0) {
    std::cout << current->value << " ";
    current = current->next;
  }
  std::cout << std::endl;
}
```

Um die Funktion auszuprobieren müssen wir ein paar Elemente erstellen und in die Liste schreiben (zum Beispiel ans Ende der Liste). Idealerweise hätten wir gerne eine Funktion die das für uns übernimmt. Diese Funktion bekommt einerseits die Liste (als Zeiger, da wir ja wirklich die Liste selbst verändern wollen und nicht nur eine übergebene Kopie) und andererseits den Wert, der eingefügt werden soll.

```
void insert(List *list, int value) {
  Element *newElement = new Element;
  newElement->value = value;

  Element *current = list->first;
  while(current->next != 0) {
    current = current->next;
  }
```

```
    current->next = newElement;
}
```

Wir erstellen also ein neues Element, weisen ihm den übergebenen Wert zu und durchlaufen die Liste von vorne bis zum Ende der Liste und Fügen das Element dort als Nachfolger des derzeit letzen Elements ein.

Diese Funktion hat zwei Probleme: zum einen funktioniert sie nicht, wenn die Liste noch leer ist und zum anderen dauert das Einfügen umso länger, je länger die Liste ist. Deshalb fügt man in einer einfach verketteten Liste Element üblicherweise am Anfang ein. Die Vorgehensweise ist dabei folgendermaßen (ich gebe euch hier kein Beispiel, das könnt ihr mal selbst probieren): Prüfe, ob Liste leer ist. Falls ja, dann setze einfach das erste Element der Liste auf das neu erstellte Element. Falls nein, dann mache das derzeit erste Element der Liste zum Nachfolger des neuen Elements und aktualisiere das erste Element der Liste. (Frage: Welche Komplexität hat das Einfügen am Ende der Liste und welche das Einfügen am Anfang?)

Außerdem sollte einem sofort ein weiteres Problem auffallen. In der obigen Funktion erstellt man neue Element mit new. Wie in der Vorlesung besprochen, muss es zu jedem new immer auch ein delete geben. Das heißt ihr müsst auf jeden Fall noch eine Funktion schreiben, die die ganze Liste löscht (d.h. die Liste durchläuft und jedes für jedes Element delete element aufruft; dabei muss man nur aufpassen, dass man nicht versehentlich ein Element löscht, bevor man seinen Nachfolger zwischengespeichert hat, sonst kann man den nämlich nicht mehr löschen.)

Auch ist es sinnvoll, eine Funktion zur Verfügung zu haben, die nur ein Element am Anfang der Liste löscht (und das gelöschte Element zurückgibt). Eine Funktion dafür könnte ungefähr so aussehen:

```
Element* remove(List *list) {
  Element* rem = list->first;
  list->first = list->first->next;
  return rem;
}
...
Element *rem = remove(&list);
```

Auch hier muss man wieder ein paar Sonderfälle beachte (leere Liste). Hat man diese Funktion geschrieben, dann ist es natürlich einfach, eine Funktion zu schreiben, die die ganze Liste löscht (indem man einfach so lange das erste Element entfernt, bis es kein erstes Element mehr gibt). delete element aufruft; dabei muss man nur aufpassen, dass man nicht versehentlich ein Element löscht, bevor man seinen Nachfolger zwischengespeichert hat, sonst kann man den nämlich nicht mehr löschen.)

Auch ist es sinnvoll, eine Funktion zur Verfügung zu haben, die nur ein Element am

Anfang der Liste löscht (und das gelöschte Element zurückgibt). Eine Funktion dafür könnte ungefähr so aussehen:

```
Element* remove(List *list) {
  Element* rem = list->first;
  list->first = list->first->next;
  return rem;
}
...
Element *rem = remove(&list);
```

Auch hier muss man wieder ein paar Sonderfälle beachte (leere Liste). Hat man diese Funktion geschrieben, dann ist es natürlich einfach, eine Funktion zu schreiben, die die ganze Liste löscht (indem man einfach so lange das erste Element entfernt, bis es kein erstes Element mehr gibt).

Doppelt verkettete Listen. Doppelt verkettete Listen sind Listen, bei denen die Elemente folgendermaßen aussehen

```
struct Element {
  int value;
  Element *prev;
  Element *next;
};
```

Das heißt, Elemente zeigen jetzt nicht nur auf ihren Nachfolger, sondern auch auf ihren Vorgänger. Die Liste sieht dann so aus:

```
struct List {
  Element* first;
  Element* last;
};
```

Man speichert nun also neben dem ersten Element der Liste auch das Letzte. Alles, was im Abschnitt vorher diskutiert wurde gilt hier genau so, nur eben in die andere Richtung. Da man nun zu jedem Element immer Vorgänger und Nachfolger angeben muss, werden die Einfüge- und Löschfunktionen natürlich ein bisschen komplizierter und man muss mehrere Sonderfälle beachten. Fangt also am besten mit einer einfach verketteten Liste an und erweitert sie Stück für Stück zu einer doppelt verketteten Liste. Habt ihr die Liste erstellt, könnt ihr sie folgendermaßen (wie auf dem Übungszettel verlangt) mit Zufallszahlen befüllen:

```
#include <cstdlib> // fuer rand()
...
DList list;
for (int i = 0; i < 50000; i++) {
  Element* element = new Element;
```

```
    element->value = rand();
    insert(&dlist, element);
  }
```

Ich hoffe diese Erklärungen helfen euch ein bisschen weiter. Ich werden die Datei in den nächsten Tagen noch erweitern, und was zum Zusammenhang zu Arrays und Zeigern schreiben. Für den aktuellen Zettel ist aber erstmal der Teil oben wichtig.

Arrays == Zeiger?

todo