(Template-)Klassen

IPI Tutorium WS 19/20

Ziel: Klasse definieren, die ein dynamisches Array verwaltet, sodass der/die Entwickler/in sich nicht um die dynamische Speicherverwaltung (aka new und delete) kümmern muss (ähnliches Beispiel auch in der Vorlesung).

Objekte der Klasse sollen verwendet werden können, wie "normale" Arrays, d.h. folgendes Programm soll kompilieren:

```
int main() {
  int size;
  std::cout << "Arraygroesse eingeben";
  std::cin >> size;

  Array a(size);

  a[0] = 10;
  a[1] = 20;

  std::cout << a[0]; // 10
  std::cout << a[1]; // 20
}
```

(Frage: Wieso kann man in diesem Programm kein statisch erzeugtes Array verwenden?) ???

Eine erste Version der Klasse könnte so aussehen (die Zugriffe über [] funktionieren natürlich noch nicht):

```
class Array {
  private:
  int size;
  int *data;

  public:
  Array(int t_size) {
    size = t_size;
    data = new int[size];
  }
};
```

Der Speicher, der im Konstruktor mit new dynamisch allokiert wird, muss auch irgendwo wieder freigegeben werden. Wir definieren also eine Methode void free(), die diese Aufgabe übernimmt:

```
class Array {
  public:
  ...
  void free() {
    delete[] data;
  }
};
```

Man beachte die eckigen Klammern nach dem delete (warum sind die wichtig?). Jetzt hat man zwar eine Methode zur Verfügung, die den Speicher freigibt, allerdings muss man natürlich auch daran denken, die Methode auch aufzurufen, wenn das jeweilige Objekt nicht mehr benötigt wird (und man muss aufpassen, die Methode nicht aufzurufen und das Objekt danach weiter zu benutzen). Zum Glück gibt es in C++ eine spezielle Methode, die sich eignet solche "Aufräumarbeiten" zu erledigen: den Destruktor. Anstelle der Methode free() schreiben wir also:

```
class Array {
  public:
  ...
  ~Array() {
    delete[] data;
  }
};
```

Frage: Wann wird der Destruktor aufgerufen? ???

Bevor wir Arrayzugriffe mit eckigen Klammern implementieren, schreiben wir zunächst zwei Methoden int get(int pos) und void set(int pos, int val), die ein Arrayelement zurückgeben bzw. setzen:

```
class Array {
  public:
  ...
  void set(int pos, int val) {
    data[pos] val;
  }

  int get(int pos) {
    return data[pos];
  }
};
```

Frage: Kann man die Methoden so verändern, dass ungültige Arrayzugriffe verhindert werden (also Zugriffe der Form a.set(10,2), wenn a bspw. nur die Größe 5 hat)? ???

Nun ist folgendes möglich:

```
Array a(10);
a.set(0, 20);
a.get(0); // 20
```

???

Um nun Zugriffe wie im Beispiel am Anfang zu ermöglichen, *überladen* (was bedeutet das?) wir den eckige-Klammern-Operator:

```
class Array {
  public:
  ...
  int operator[](int pos) {
    return data[pos];
  }
};
```

Wollen wir mit diesem Operator nun sowohl die get, als auch die set-Methode ersetzen, gibt es noch ein Problem. Zuweisungen der Form

```
a[0] = 10;
```

führt zu dem etwas kryptischen Compilefehler `lvalue required as left operand of assignment`, was nichts anderes bedeutet, dass wir hier versuchen einen Wert zuzuweisen, wo kein Wert zugewisen werden kann. Das ist auch völlig klar: der Rückgabewert des operator[] ist int und wir versuchen dann einem int-Wert (nicht einer int-Variablen) den Wert 10 zuzuweisen, was natürlich keinen Sinn ergibt. Dieses Problem lässt sich leicht lösen: anstatt den *Wert* des jeweiligen Arrayelements zurückzugeben, geben wir einfach eine *Referenz* auf diesen Wert zurück. Dafür müssen wir lediglich den Rückgabetyp des Operators ändern, den Rest erledigt der Compiler:

```
class Array {
  public:
  ...
  int& operator[](int pos) {
    return data[pos];
  }
};
```

Das Beispielprogramm vom Anfang kompiliert nun und funktioniert wie erwartet. Vollständig ist unsere Klasse jedoch noch nicht. Betracte dazu folgendes Beispiel:

```
1  int main() {
2    Array a(10);
3    a[0] = 100;
4
5    Array b(a);
6    b[0] = 20;
7
8    std::cout << a[0] << ", ";
9    std::cout << b[0] << std::endl;
```

```
10  }
```

Übersetzt man dieses Programm und führt es aus, passieren zwei (evtl. unerwartete Dinge):

- Die Ausgabe ist `20, 20`
- Das Programm stürzt mit einem Fehler ab

Was passiert hier? Zunächst wird das Objekt `a` der Größe 10 erstellt und dessen erstes Element auf 100 gesetzt. In Zeile 5 wird nun das Objekt `b` erstellt, allerdings nicht mit dem oben definierten Konstruktor, sondern aus dem Objekt `a`, mithilfe des implizit definierten (d.h. automatisch vom Compiler generierten) Copy-Konstruktors. Dieser Konstruktor macht folgendes: er durchläuft alle Eigenschaften des Objekts `a` (also hier die Variablen `size` und `data`) und kopiert deren Werte in das Objekt `b`. Da das Array `data` in der Klasse jedoch als Zeiger gespeichert wird, werden nicht die einzelnen Arrayelemente kopiert, sondern nur der Zeiger. Dabei entsteht folgendes Problem: Es gibt nun zwei Objekte (`a` und `b`) die **auf das selbe Array zeigen**. Nun erklärt sich auch die Ausgabe. Da die beiden Objekte nun auf das selbe Array zeigen, führt eine Änderung des Arrays in `b` auch zu einer Änderung des Arrays in `b` (denn es handelt sich ja um das selbe Array!). Dieses Verhalten ist natürlich (in der Regel) unerwünscht, d.h. wir müssen den Copy-Konstruktor nun selbst definieren und das Array ordentlich kopieren:

```
class Array {
  public:
  ...
  Array(Array &other) {
    size = other.size;
    data = new int[size];
    for (int i=0; i<size; i++)
    data[i] = other.data[i];
  }
};
```

Nun funktioniert das Programm wie erwartet. Frage: Aus welchem Grund ist das Programm oben abgestürtzt? ???

Das selbe Problem tritt auch auf, wenn wir anstatt ein Objekt zu kopieren, ein Objekt einem anderen zuweisen (was genau ist der Unterschied?): ???

```
Array a(10);
a[0] = 10;
Array b(20);
b[0] = 20;

b = a;
std::cout << b[0] << std::endl; // 10
```

Das Problem löst man analog zu oben, jetzt muss man allerdings den Zuweisungsoperator definieren: Array& operator=(Array &other). Warum ist es sinnvoll, dass der Zuweisungsoperator den Rückgabetyp Array& hat?

???

Das oben beobachtete Problem tritt übrigens immer auf, wenn in einer Klasse dynamisch (also mit new speicher allokiert wird). Die Regel, dass man in diesem Fall die drei Methoden Destruktor, Copy-Konstruktor und Zuweisungsoperator selbst definieren muss, ist als *Rule of Three* bekannt. Nun haben wir eine vollständige Array-Klasse, die natürlich noch beliebig erweitert werden kann.

```
class Array {
  private:
  int *data;
  int size;
  public:
  Array(int t_size) {
    size = t_size;
    data = new int[size];
  }

  Array(Array &other) {
    size = other.size;
    data = new int[size];
    for (int i=0; i<size; i++)
    data[i] = other.data[i];
  }

  Array& operator=(Array &other) {
    // Uebung :)
  }

  int& operator[](int pos) {
    return data[pos];
  }

  ~Array() {
    delete[] data;
  }
};
```

Schön wäre es natürlich, wenn man nicht nur int-Arrays, sondern Arrays beliebiger Typen verwalten könnte. Dazu templatisieren wir die Klasse, d.h. überall wo wir den Datentyp des Arrays (also int verwenden, schreiben wir einfach T und deklarieren die Klasse als Templateklasse):

```
template <class T>
class Array {
  private:
```

```
  T *data;
  int size;
  public:
  Array(int t_size) {
    size = t_size;
    data = new T[size];
  }

  Array(Array &other) {
    size = other.size;
    data = new T[size];
    for (int i=0; i<size; i++)
    data[i] = other.data[i];
  }

  Array& operator=(Array &other) {
    // Uebung :)
  }

  T& operator[](int pos) {
    return data[pos];
  }

  ~Array() {
    delete[] data;
  }
};
```

Frage: Anstatt template <class T> kann man auch template <typename T> schreiben. Was ist der Unterschied?

???

Um diese Klasse verwenden zu können, muss nun beim erstellen des Objekts der Templateparameter gesetzt wernde, d.h.

```
Array<int> a(10); // int-Array mit 10 Elementen

Array<double> b(20); // double-Array mit 20 Elementen
```

Natürlich können auch komplexere Datentypen verwendet werden, wie bspw. die Klasse Complex aus der Vorlesung:

```
Array<Complex> c(3); // Array mit 3 komplexen Zahlen
```

Hier ist nun folgendes wichtig: Verwendet man alle drei Beispiele im selben Programm (also Array<int> a(10), Array<double> a(10) und Array<Complex> a(10)) dann werden **zur Compilezeit** drei unterschiedliche Klassen erzeugt, für jeden Templatetype eine. Das heißt, zur Laufzeit verhält sich das Programm so, als hätte man selbst für jeden einzelnen Typ eine separate Klasse geschrieben. Das bedeutet, trotz der Abstrahierung der Klasse (anstatt int-Array ein Array beliebigen Typs) wird das Programm nicht langsamer. Allerdings benötigt das

Programm länger zum Compilieren (da die ganzen Klassen ja vom Compiler erstellt werden müssen), was bei verschachtelten Templateklassen oft zu unerträglich lange Compilezeiten führt.

Ein weiterer Nachteil ist, dass Fehlermeldungen, die durch Templateklassen erzeugt werden in der Regel völlig unbrauchbar sind. Außerdem können verschachtelte Templateklasse (also bspw. Templateklassen die als Templateparameter selbst wieder Templateklassen verwenden) schnell sehr unübersichtlich werden.