Objektorientierte Programmierung mit C++ (WS 2010)

Dr. Andreas F. Borchert, Tobias Brosch
Institut für Angewandte Informationsverarbeitung
Universität Ulm

Blatt 5: Abgabetermin 24. November 2010

1 Dynamic Objects

Nicht immer wissen wir im voraus, wie groß ein Array sein soll. Damit scheiden non-dynamic (static) arrays aus. Wir können jedoch Objekte und Arrays auf dem Heap (dynamic memory) mit `new` bzw. `new []` anlegen. Objekte, die so angelegt werden, leben zunächst einmal ewig (bis zum Programmende). Wer also memory-leaks bauen möchte ist hier genau richtig!

Für alle, die so angelegte Objekte auch wieder loswerden möchten, gibt es `delete` bzw. `delete []`, welche auf dem von `new` bzw. `new []` zurückgegebenen Zeiger aufgerufen werden. Da das Verhalten jedoch nur für einen `delete` Aufruf pro allokiertem Objekt definiert ist, sollte man tunlichst darauf achten, dies auch nur einmal zu tun!

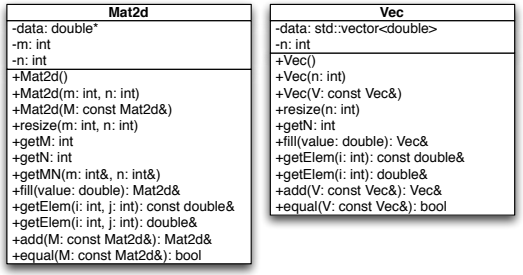
Eine Möglichkeit, ist den Zeiger nach einem `delete` Aufruf auf 0 zu setzen, da `delete` auf einem 0-Zeiger nichts macht.

```
#include <iostream>
#include <stdexcept>
#include <stdlib.h>
const int maxSiz = 1000000;
int n;
std::cout << "Enter size n in [0, " << maxSiz << "]" << std::endl;
// read n from shell:
std::cin >> n;
if(!(n > 0 && n < maxSiz))
   throw std::runtime_error("n not in valid range!");

double *a = new double[n];
for(int i = 0; i < n; ++i)
   a[i] = rand();
for(int i = 0; i < n; ++i)
   std::cout << a[i] << ",";
std::cout << std::endl;

delete [] a;
a = 0;
```


2 Eine einfache Matrix und Vektor Klasse



Implementiert die im UML-Diagramm gegebenen Klassen `Mat2d` und `Vec`. Beachtet dabei folgende Punkte:

- Achtet auf `const`-Correctness (Alles, das `const` deklariert werden kann, auch `const` deklarieren)!
- Achtet im copy-Konstruktor `Mat2d(const Mat2d& M)` bzw. `Vec(const Vec& V)` darauf, dass auch die Elemente kopiert werden!
- Die Elemente von `Mat2d` sollen in einem dynamisch allokiertem Array `data` untergebracht werden. Vergesst also nicht, einen Destruktor zu schreiben, in dem der Speicher wieder freigegeben wird!
- Überprüft bei einem resize in `Mat2d`, ob Ihr neuen Speicher braucht (es reicht, wenn Ihr überprüft, ob die Anzahl der Elemente im aktuellen resize größer wird). Wenn nicht, verwendet den bisher belegten Speicher weiter. Falls ja, achtet darauf den zuvor allokierten Speicher freizugeben!
- Verwendet in `Vec` die jeweils passenden Operatoren von `std::vector`.
- Verwendet in den Methoden `fill` und `add` den Iterator-Style (sowohl in `Mat2d`, als auch in `Vec`).
- Macht euch bewusst, dass die Methode `void add(const Mat2d& M)` die Funktionalität des Operators `void operator+=(const Mat2d& M)` implementiert (dazu später mehr).
- `fill` und `add` liefern jeweils eine Referenz auf sich selbst zurück (damit können später auf dem Ergebnis direkt weitere Methoden aufgerufen werden!).

Die Klassen sollen in den Dateien `Mat2d.h`, `Mat2d.C`, `Vec.h` und `Vec.C` deklariert bzw. implementiert werden.

Schreibt eine Datei `main.C`, die eine Testfunktion beinhaltet (die auch aufgerufen wird), um folgende Dinge zu prüfen:

- Sei `M1` gefüllt mit 1, `M2` gefüllt mit 2. Überprüft, ob `M1.add(M1).equal(M2)`. Prüft gleiches für zwei Vektoren.
- Sei `M1` gefüllt mit Zufallszahlen. Initialisiert `M2` über den copy-Konstruktor mit `M1`. Ruft `M1.add(M2)` auf. Überprüft anschließend, ob jedes Element von `M1` gleich dem Doppelten von `M2` ist.
- Sei $m \neq n$, sei `M1` $m \times n$ 1–gefüllte Matrix und `M2` $n \times m$ Matrix. Generiert `M2` aus `M1` mittels copy-Konstruktor und `resize`. Überprüft, ob `!M1.equal(M2)`.

Schreibt euch ein eigenes Makefile, dass sämtliche Abhängigkeiten berücksichtigt. Submitted die Dateien `Mat2d.h`, `Mat2d.C`, `Vec.h`, `Vec.C`, `main.C` und `Makefile`:

```
submit cpp 5 main.C Mat2d.h Mat2d.C Vec.h Vec.C Makefile
```

Viel Spaß!