Übung zu
Algorithmen und Datenstrukturen (für ET/IT)

Wintersemester 2012/13

Jakob Vogel

Computer-Aided Medical Procedures
Technische Universität München

Administratives

- Zwischenergebnis der Umfrage
  - Die Meinungsabgabe ist tatsächlich anonym!
  - Bitte Hinweise auf Lösungsmöglichkeiten bei Kritik!
- Ab sofort regelmäßig Möglichkeit zu anonymem Feedback
- Ab sofort Aufgabenblätter mittwochs, um ggf. auf Probleme eingehen zu können
- Sie können uns natürlich auch direkt ansprechen!

Administratives

- Klausur
  - Probeklausur
- Übung
  - Aufgaben als Anreiz zur Beschäftigung mit den Problemen
  - Besuch der ZÜ und **Nachbereitung** der Folien, Aufgaben- und Lösungsblätter
  - Tutorfragestunden, Moodle-Forum, Sprechstunden
- Programmieren
  - Vermittlung *en passant*, Hauptziel sind Algorithmen und Datenstrukturen
  - Bereitgestellten Code modifizieren und experimentieren!

Wiederholung

Datentypen

Wir haben bisher die folgenden Datentypen behandelt:

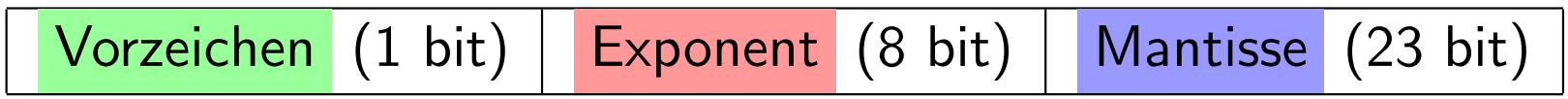
- Ganzzahlen
- Fließkommazahlen
- Wahrheitswerte (als Interpretation von Ganzzahlen)
- Schriftzeichen (ebenfalls als Interpretation von Ganzzahlen)
- Arrays (als Reihe von einem der oben genannten Typen)
- Strings (als Spezialfall eines Schriftzeichen-Arrays)

Fließkommazahlen

- Wissenschaftliche Darstellung (hier wieder absoluter Nullpunkt)

$$- \; 2.7315 \cdot 10^{2} \; °\mathrm{C}$$

- Speicherung (hier als 32-bit-`float`) als Tripel



- Alle drei Teile sind binär kodiert, der Wert berechnet sich zu

$$(-1)^{V} \cdot (1 + M) \cdot 2^{E - bias}$$

- Exponent ist `unsigned`, mit Vorzeichen via *bias*
- Mantisse speichert nur „Nachkomma-Teil“
- Sondermuster für $\pm\infty$, ± 0, `NaN`

Probleme

- Die Zahl π ist irrational:

  $\pi = 3.14159265358979323846264338327950288419716939 9...$

- „Verpacken" wir π in eine (vereinfachte, dezimale) Fließkommazahl:
  - $V = 0$ $(pi > 0)$
  - $M = 0.3141592654$ (10 Stellen Genauigkeit annehmend)
  - $E = 1$

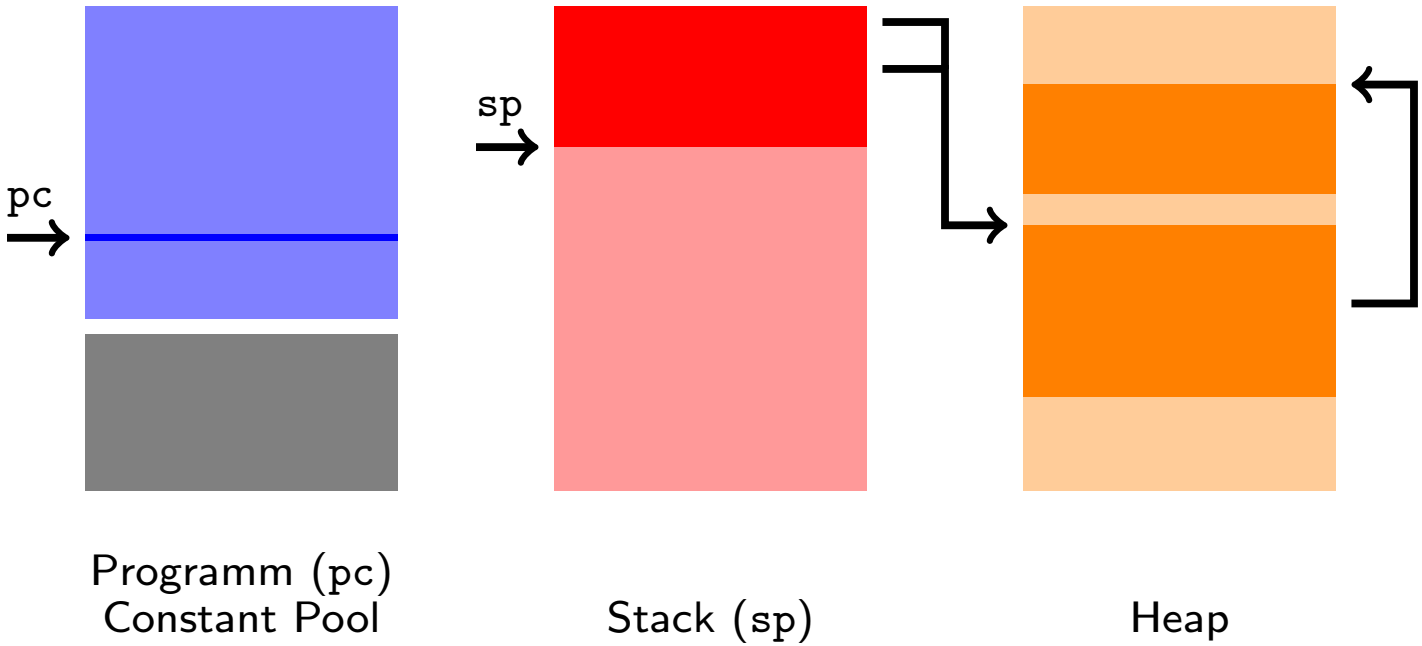
- Bei der Mantisse wurde gerundet, wir machen also Rechenfehler!
  - Zinsberechnung, ...

Fragen?

- Hello, World!
- Euklidischer Algorithmus
- Zahldarstellung und Bitweise Operationen
- Arrays (Speicher-Anforderung auf Heap)
- Stack (Kommt noch in Vorlesung!)

- Essenz des Call Stacks: Lokale Variablen...
  - ...haben fixe Größe.
  - ...sind nur innerhalb des Bezugsrahmens (*Scope*) gültig.
  - ...werden automatisch aufgeräumt.
- Objektorientierung kommt erst heute!

Erinnerungen an den Heap...



- Notwendig bei variablen Platz-Anforderungen
- Manuelle Anforderung und (einmalige!) Freigabe
  - C : `malloc(...)` und `free(...)`
  - C++: `new`/`new[]` und `delete`/`delete[]`


Neues

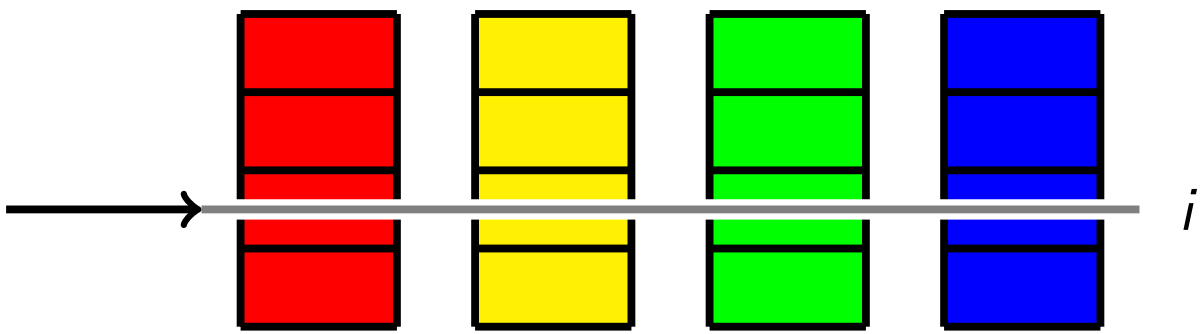
Gedankenexperiment: Ein Rennspiel (I)

Setting: Physikengine für n Autos

Jedes Auto hat Eigenschaften:

- Orts-, Geschwindigkeits- und Beschleunigungsvektoren
- Stellung von Lenkrad, Gas- und Bremspedal, Schaltung
- Treibstoffmenge, Turbo, Reifenhaftung, Schaden

Annahme: Für jede Eigenschaft erzeugen wir ein Array, und betrachten für Auto i die entsprechenden Zellen.



Gedankenexperiment: Ein Rennspiel (II)

Jedes Mal, wenn wir ein Auto kopieren wollen, müssen wir alle Arrays kennen und richtig auslesen/befüllen

- Laden von externen Fahrzeugmodellen („*Assets*“)
- Speichern und Laden von Spielständen
- Initialisierung für ein Rennen aus einer „Garage“
- Netzwerkkommunikation für Multiplayer

Problem: Wenn eines der Arrays irgendwo in Unordnung gerät, hat das überall Folgen!

Gedankenexperiment: Ein Rennspiel (III)

Gehen wir davon aus, dass wir es trotzdem richtig programmiert haben.

Aber wir haben die Daten für die Grafikengine vergessen:

- Lackierung
- Unterbodenbeleuchtung
- Scheibentönung

Wir brauchen ein besseres Konzept als Arrays!

Objektorientierung

Idee: Zusammengesetzte Datentypen!

- Alle Eigenschaften werden in einer *Klasse* zusammengefasst:
  „*Ein Auto hat einen Orts-, einen Beschleunigungsvektor...*“
- Ein konkretes Objekt entspricht dann einer *Instanz*, wobei den Eigenschaften Werte zugewiesen werden:
  „*Dieses spezielle Auto ist schwarz und fährt mit 220 km/h.*“
- Alle Eigenschaften definieren den *Zustand*, der über Funktionen manipuliert werden kann.

Gedankenexperiment: Ein Rennspiel (IV)

```
struct Auto
{
    float geschwindigkeit;
    // usw.

    // Konstruktor
    Auto()
    { /* Initialisiert Variablen, etwa Heap-Allokationen */ }

    // Destruktor
    virtual ~Auto()
    { /* Raeumt auf, etwa Heap-Freigaben */ }

    // Funktionen
    virtual void berechnePhysik()
    { /* Normale Funktion + Zugriff auf Zustand */ }

    // usw.
};
```

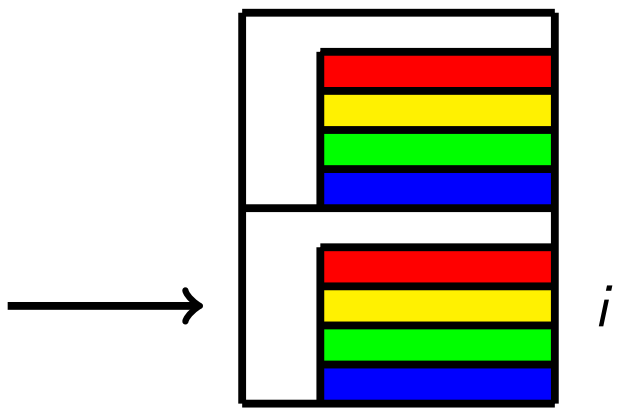
Gedankenexperiment: Ein Rennspiel (V)

```
Auto raser( /* Hier stehen Konstruktor-Parameter! */ );

// Zustand setzen
raser.geschwindigkeit = 220;
// usw.

// Berechne den Beschleunigungsvektor fuer die Physik-Engine
raser.berechnePhysik();
```

- Bei einem Array von Autos sind unsere Probleme viel geringer im Vergleich zu den Einzelarrays!



Objektorientierung

- Alle relevanten Eigenschaften sind zusammen in einem Objekt.
- Man kann Kopie und Serialisierung (für Speicherung) als Objekt-Funktion zentral erstellen.
- Im Konstruktor werden Initialisierungen erledigt, wie Heap-Allokationen. Dessen Aufruf erfolgt automatisch bei Erzeugen der Instanz.
- Der Destruktor sorgt für's Aufräumen, etwa durch Heap-Freigaben. Der Aufruf erfolgt automatisch bei `delete`, oder wenn die lokale Variable den Scope verlässt.

- Kapselung ermöglicht Modularisierung!

Objektorientierung

- Alle relevanten Eigenschaften sind zusammen in einem Objekt.
- Man kann Kopie und Serialisierung (für Speicherung) als Objekt-Funktion zentral erstellen.
- Im Konstruktor werden Initialisierungen erledigt, wie Heap-Allokationen. Dessen Aufruf erfolgt automatisch bei Erzeugen der Instanz.
- Der Destruktor sorgt für's Aufräumen, etwa durch Heap-Freigaben. Der Aufruf erfolgt automatisch bei `delete`, oder wenn die lokale Variable den Scope verlässt.

- Kapselung ermöglicht Modularisierung!


Code

Erinnerungen an das Array...

Drei „Schreibweisen“:

- ```
  // Array auf dem Stack
  int array[5];
  ```
- ```
  // Array im Heap mit Speichermanagement
  int* array = new int[n]; /* ... */ delete[] array;
  ```
- ```
  #include <vector>
  std::vector<int> array(n);
  ```
  ```

Und `std::vector<...>`?

- Kapselt explizites `new[]`/`delete[]` in Konstruktor und Destruktor.
- Bietet Hilfsfunktionen (etwa dynamisches Wachsen und Größenänderung).
- Für Interessierte: In der Klammerung `<...>` steht der Typ als *Template-Parameter*.

Programmieren in Zeiten von `std::vector`

```
void funktion()
{
    int* array = new int[n];


    if (abbruchbedingung)
    {

        // SPEICHERLECK!
        return;
    }

    delete[] array;
}
```

```
void funktion()
{
    std::vector<int> array;
    array.resize(n);

    if (abbruchbedingung)
    {
        // Ende Scope
        // --> Destruktor
        return;
    }

    // Ende Scope
}
```

Bequem, einfach, sicher, elegant!

Daher C++ statt C!