Programmierrichtlinien für C++

Grundlagen

- Bevorzugen Sie *const* und *inline* gegenüber *#define*
  - ✗ Bessere Unterstützung durch den Compiler

- Bevorzugen Sie <*iostream*> gegenüber <*stdio.h*>
  - ✗ Wegen Sicherheit durch Eingrenzung beim Namespace

- Bevorzugen Sie *new* und *delete* gegenüber *malloc* und *free*
- Bevorzugen Sie C++-Casts vor dem C-Cast
  - ✗ Unterschiede der Casts hervorheben
  - ✗ C-Casts im Quelltext schwer zu finden
  - ✗ Harter C-Cast in C++: `static_cast<Typ> (Ausdruck);`

- Mache Basisklassen abstrakt, die nicht am Ende der Hierarchie stehen
  - ✗ Führt zu komplizierten Problemen, Details *ME S.271*

- Vermeide grundlose Standardkonstruktion
  - ✗ Unnötiger Speicherverbrauch
  - ✗ Aber leider dadurch leicht schwierigere Handhabung

- Verhindere Exceptions bei Destruktoren
  - ✗ Würden zu *terminate* führen

```
Bubu::~Bubu()
{
   try{
         logDestruction(this);
   }catch(..) {}
}
```

- Fange Exceptions per Referenz
  - ✗ Vorteile gegenüber dem Fangen per Wert und Fangen per Zeiger

- Keine Verwendung von Smart Pointern
  - ✗ Wäre zwar an sich hilfreich und sinnvoll, aber in hinsicht auf die Prüfung dieses Semester eher kontraproduktiv

- Verwendung der *Standard Template Library*, wo immer es möglich ist
  - ✗ Verwenden von ausgereiften und geprüften Funktionen, ähnlich wie in Java

Speicherverwaltung

- Benutzen Sie die gleiche Form für korrespondierende *new*- und *delete*-Aufrufe
  - ✗ Falsch -> Undefiniertes Verhalten

```
string *stringPtr = new string[100];
delete stringPtr
```

  - ✗ Richtig:

```
string *stringPtr = new string[100];
delete [] stringPtr
```

- Rufen Sie für alle Zeiger-Datenelemente im Destruktor *delete* auf
  - ✗ Sicherheitsnetz falls zuvor ein Löschen vergessen wurde
  - ✗ Da das Löschen des NULL-Zeigers erlaubt ist, tut auch ein überflüssiger Aufruf nicht weh
- [Seien Sie auf *out-of-memory* Situationen vorbereitet]
  - ✗ Aufwendig, daher eventuell in nicht-sicherheitskritischem Umfeld zu vernachlässigen

Klassenverwaltung

- Bevorzugen Sie Initialisierung gegenüber Zuweisung im Konstruktor
  - ✗ Wurde in Vorlesung behandelt: Effizienz
- Führen Sie die Datenelemente in der Initialisierungsliste in der Reihenfolge ihrer Deklaration auf
  - ✗ Da Elemente anhand ihrer Deklaration initialisiert werden, erhöht dies die Verständlichkeit
- Stellen Sie sicher, dass Basisklassen einen virtuellen Destruktor haben
  - ✗ Wird zur korrekten Destruktion in Klassenhierarchien benötigt
- Lassen Sie *operator*= eine Referenz auf **this* zurückliefern
  - ✗ Ermöglicht die aus Java bekannten Zuweisungsketten
- Weisen Sie allen Datenelementen im *operator*= etwas zu, wenn dieser nötig ist
  - ✗ Automatisch generierter Zuweisungsoperator führt schnell zu Speicherlecks
- Prüfen Sie in Operator= auf Zuweisung an sich selbst
  - ✗ Effizienz durch Zeitersparnis
  - ✗ Mögliche Aliasing-Effekte

```
C& C::operator=(const C& rhs)
{
      if(this == &rhs) return *this;
...
}
```

Entwurf und Design

- Vermeiden Sie Datenelemente in der public-Schnittstelle
- Benutzen Sie const, wann immer möglich

- Benutzen Sie die Parameterübergabe via Referenz gegenüber der Parameterübergabe per Wert
- Unterteilen Sie den globalen Namensraum
  - ✗ Namensraumkonzept gemeinsam entwickeln

Implementation

- Liefern Sie keine Handles auf interne Daten zurück
  - ✗ Interne Handles können verdeckt zerstört werden

- Schieben Sie die Variablendefinition solange wie möglich auf
  - ✗ Moderner Ansatz von C++ für höhere Performanz
  - ✗ Zweck einer Variablen wird so deutlicher

- Verwenden Sie inline wohlüberlegt

- Minimieren Sie Kompilationsabhängigkeiten zwischen Dateien
  - ✗ Möglich durch forward-Deklarationen

```
class Person

class Bubu
{
      Person b;
}
```

  - ✗ Auch extreme Version durch <C>Impl möglich, siehe E S.182

Objektorientiertes Design

- Sorgen Sie dafür, dass öffentliche Vererbung **ist ein** bedeutet

- Unterscheiden Sie zwischen der Vererbung von Schnittstellen und der Vererbung von Implementationen

- Überschreiben Sie niemals eine geerbte, nicht virtuelle Funktion

- Redefinieren Sie niemals einen geerbten Default-Parameter

- Vermeiden Sie Downcasting in Klassenhierarchien

- Unterscheiden Sie zwischen Vererbung und Templates

- Keine Verwendung von Mehrfachvererbung
  - ✗ Mehr Probleme als Nutzen
  - ✗ Kann durch gutes Interface-/Klassendesign umgangen werden