Implementierung Einfachvererbung

- jedes Objekt enthält Zeiger auf dynamischen Typ
- Typ enthält Array von Zeigern auf Methoden
- Methode intern als Array-Index dargestellt,
  Variable als Offset von Objektadresse
- dynamisches Binden: Methode in Array-Eintrag des dynamischen Typs aufrufen, Variablenzugriff über Offset
- Einfachvererbung: Indexe und Offsets bleiben erhalten

  zusätzliche Einträge und Offsets hinten angehängt

  ererbte Methode = kopierter Array-Eintrag

  überschriebene Methode = überschriebener Array-Eintrag

Implementierung Interfaces

- einfache Impl. für Mehrfachvererbung ungeeignet
- Array-Eintrag für jedes implementierte Interface
  (zeigt auf Interface, sonst wie Methodenzeiger)
- Interface enthält Array von Methodenzeigern
  (und Variablenoffsets wenn unterstützt)
- eine Indirektion mehr, Variablenzugriffe teurer
- Einfachvererbung von Interfaces durch hinten anhängen
- Mehrfachvererbung: zusätzliche Interfaces in Array
- „Optimierung“: Interface-Array direkt in Objekt

Implementierung Mehrfachvererbung 1

- im Prinzip Techniken für Interfaces verwendbar
  wenn nötig mehrere Arrays (VFT = virtual function table)
- Objektlayout ererbter Klassen in C++ direkt übernommen
  (für up-cast auf Objekten notwendig)
- up-cast ändert Zeiger um Konstante „delta“ auf Anfang des Teilobjekts
- Anfang jedes Teilobjekts zeigt auf eigenen VFT
- bei Vergleichen auf Identität „delta“ berücksichtigen

Implementierung Mehrfachvererbung 2

- Methodenaufruf bei Mehrfachvererbung (allgemein):

```
1.   load [objectReg + #VFToffset], tableReg
2.   load [tableReg + #deltaOffset], deltaReg
3.   load [tableReg + #selectorOffest], methodReg
4.   add objectReg, deltaReg, objectReg
5.   call methodReg
```

- da „delta“ meist 0, Optimierung mit „thunks“:

  Befehle 2 und 4 weglassen

  statt dessen bei Mehrfachvererbung Sprung auf „thunk“:

```
thunk: add objectReg, #delta, objectReg
       jump #method
```

Optimierungsmöglichkeiten

- wenn Methode statisch bekannt:
  - direkter Sprung statt über VFT
  - „inlining“ und weitere Optimierungen möglich
- Methode eher statisch bekannt wenn Compiler komplette Typhierarchie kennt (Eiffel)
- teilweises „inlining“ (Spezialisierung) durch „thunks“
- durch Wachsen in beide Richtungen und bei statisch bekannter Typhierarchie „thunks“ fast immer vermeidbar
- dynamisches Caching: zuerst auf selbe Methode wie bei letztem Aufruf springen, dann schauen ob Typ passt und nötigenfalls weiterspringen

Ist Code-Mehrfachvererbung nötig?

- technische Antwort:
  Vererbung ist überhaupt nicht nötig, nur Subtyping
- Grund: Code-Vererbung lässt sich z.B. über Decorator-Pattern praktisch immer simulieren
- aber Code-Vererbung ist praktisch und sinnvoll
- Kombination einfache Vererbung von Code und mehrfache Interface-Vererbung fordert frühzeitige Entscheidung, ob Interface oder Klasse
- daher Code-Mehrfachvererbung auch sinnvoll, wenn nur selten benötigt

Code-Mehrfachvererbung: Beispiele

- eine Oberklasse implementiert eine Template Method, eine andere implementiert eine Operation der Template Method
- eine Oberklasse implementiert nur inhaltlichen Code, eine andere nur Synchronisationscode
- beide Beispiele sind „mixins“ im weiteren Sinne, wobei eine Unterklasse bestimmt, aus welchen vorgefertigten austauschbaren Teilen sie besteht
- alternative Lösungen sind jeweils möglich, aber teilweise schwer realisierbar wenn Oberklassen-Code nicht sichtbar ist oder nicht geändert werden soll

Implementierung Exception Handling

- Compiler erzeugt Tabelle, die Adressbereichen (Blöcken) Exception-Handler zuordnet (für jeden Exception-Typ)
- nach Auftritt wird Handler für aktuellen Instruction Pointer (IP) gesucht und (wenn vorhanden) darauf gesprungen
- existiert kein passender Handler, wird aus aktueller Routine zurückgekehrt und mit neuem IP (Aufrufstelle) gesucht
- für Code nach Rückkehr aus „main“ existiert Handler
- kein Overhead ohne Exception, Handler-Suche billig, Objekterzeugung manchmal teuer
- bei Retry-Semantik ruft Handler ursprünglichen Code auf

C++ Templates

- auch Konstanten als Typparameter:

```
template<class T, int n> class buffer { ... };
```

- partielle/volle Spezialisierung mit „pattern matching“

```
template<int n> class buffer<bool, n> { ... };

template<> class buffer<bool, 0> { ... };
```

- implizite Instanzierung wenn keine passende Spezialisierung
- Methoden nur erzeugt wenn im Code verwendet

Ausdrucksstärke von Templates

- statische Auswertung von Ausdrücken verwendbar:

```
template<int x, int n> struct power {
  static const int r = x * power<x, n-1>::r;
};
template<int x> struct power<x, 0> {
  static const int r = 1;
};
```

- auch algebraische Datentypen ausdrückbar:

```
template<class Head, class Tail> struct Cons { };
struct Nil { };
typedef Cons<int, Cons<float, Nil> > list_of_types;
```

Policy-Based Programming

```
#include <iostream>
template<typename lang> class HelloWorld : public lang {
   public: void Run() { cout << Message() << endl; }
};
#include <string>
class HelloWorld_Msg_German {
   protected: string Message() { return "Hallo Welt!"; }
};
typedef HelloWorld<HelloWorld_Msg_German> MyHelloWorld;
MyHelloWorld hello;
hello.Run();
```

Beispiel für Strategy (Design Pattern)

```
interface IMessage { String message(); }
class DMessage implements IMessage {
    public String message() {return "Hallo Welt!";}
}
class HelloWorld {
    private IMessage msg;
    public HelloWorld(IMessage msg) {this.msg = msg;}
    public void run() {System.out.println(msg.message());}
}
class Main {
    public static void main(String[] args) {
        HelloWorld hello = new HelloWorld(new DMessage());
        hello.run();
    }
}
```