Übung 2: Detaildesign mit Zeiger/Referenzen und Klassen

Zweck:

- Detaildesign mit Zeiger und Referenzen üben.
- Klasse implementieren aufgrund applikatorischer Vorgaben.
- Mit Strings als `char`-Array arbeiten.
- Bearbeitung von Arrays aus Objekten.
- Allozierung auf *Stack* und *Heap* vergleichen.
- *Operator-Overloading* implementieren.
- Umgang mit *const*-Objekten.

Ausgangslage:

Gegeben ist der Rumpf der Klasse *Person,* ein Testprogramm (`PersonTest.cpp`) incl. *Makefile* auf Moodle.

Es soll mit obiger Ausgangslage ein entsprechendes *Makefile*-Projekt in Eclipse/CDT aufgesetzt werden (*File > New > Makefile Project with Existing Code*).

Aufgabe 1: Zeiger und Referenzen

Es sollen zwei Funktionen erstellt werden, in welchen jeweils ein String mit ***new*** erzeugt wird (z.B. `new string("Hallo")`) und dieser dann der aufrufenden Funktion zur Verfügung gestellt wird.
Die Rückgabe dieses Strings soll ***nicht*** mittels Return-Wert, sondern mittels *formalem Parameter* erfolgen (⇨ *Out-Parameter* in UML).
Anforderungen an die beiden zu erstellenden Funktionen:

- die erste Funktion soll nur mittels Pointer gelöst werden.
- die zweite Funktion soll auch den Referenz-Datentyp verwenden.

Wichtig: Es soll pro Funktion nur *ein einziges* String-Objekt geben (obiges mit `new` erzeugte String-Objekt), keine zusätzlichen Kopien davon!
Hinweise:

- Die Stellen welche angepasst werden müssen sind mit einem /* ToDo: */ markiert ⇨ Menü: *Window>Show View>Tasks*
- Wenn es nicht so funktioniert wie angenommen: eine Zeichnung machen ;-)

Aufgabe 2: Klasse, Strings, char-Array, Arrays von Objekten

Die Klasse *Person* soll derart realisiert werden, dass die Funktion *main()* übersetzt werden kann.

Dazu soll jeweils in den Funktionen die *'Kommentar-Beginne'* sukzessiv nach hinten *verschoben* werden und die entsprechend nötigen Methoden oder Funktion implementiert werden.

Hinweise:

- Für den *Namen* in der Klasse Person soll ein `char`-Array verwendet werden (z.B. `char mName[NAME_LEN+1];`)
- Um einen *String* (`const char*`) in einen `char`-Array zu kopieren
  ⇨ C-Funktion `strcpy()`
  `strcpy(char* destination, const char* source)`
  `destination`: char-Pointer der auf den char-Array zeigt wohin der String kopiert werden soll.
  `source`: char-Pointer der auf String zeigt, von welchem kopiert werden soll.

Aufgabe 3: *Stack vs. Heap*

In der Funktion `aufgabe3()` soll nach dem Test auf dem Stack ein weiterer Test hinzugefügt werden, bei dem die Objekte nun auf dem Heap alloziert werden.

Hinweis:
Je nach Entwicklungsumgebung und Kompiler muss man mit Optionen explizit festlegen, dass die neuen Features des C++11-Standards (`#include <chrono>`) benutzt werden können.

- Bei *Eclipse/CDT*:
  Projekt-Properties>C/C++ General>Paths and Symbols># Symbols>GNU C++:
  Mit “Add...” ein neues Symbol hinzufügen:
  - Name: ***__GXX_EXPERIMENTAL_CXX0X__***
  - Value: ***1***
- Bei *Eclipse/CDT* mit *Cygwin* und einem *Executable*-Projekt:
  Projekt-Properties>C/C++ Build>Settings>Cygwin C++ Compiler>Miscellaneous:
  bei “Other flags” hinten hinzufügen: ***-std=c++11***
- Bei einem *Makefile*-Projekt:
  Obige Option im Makefile bei den Kompiler-Flags hinzufügen.
  Somit: `CFLAGS = -g` ***`-std=c++11`***

Aufgabe 4: *Operator-Overloading* und *const*-Objekte

- Für die Klasse *Person* soll der *Links-Shift-Operator* (**<<**) erstellt werden, sodass Personen-Objekte damit auf *Standard-Output* (`cout`) ausgegeben werden können.

```
Person bond(4711, "James Bond");
cout << bond;

/* Session-Log:
Name: James Bond       Nr: 4711
*/
```

- Im Weiteren soll sichergestellt werden, dass folgende Anweisungen möglich sind:

```
Person john = "John Smith";

const Person bond(4711, "James Bond");
cout << bond.getNr();
```

