1 Pointer-Initialisierung

Welche der folgenden Pointer-Initialisierungen sind legal? Probieren Sie in ihrer Entwicklungsumgebung.

```
int i = 'a';
const int ic = i;
const int *pic = &ic;
int *const cpi = &ic;
const int *const cpic = &ic;
```

2 Referenz-Initialisierung

Welche der folgenden Referenz-Initialisierungen sind legal? Probieren Sie in ihrer Entwicklungsumgebung.

```
int& i = 'a';
const int ic = i;
const int& ric = &ic;
int& const rpi = &ic;
const int& const cpic = &ic;
```

3 Parameter-Übergabe

Schreiben Sie eine Prozedur `void Swap(...)`, welche die Werte zweier ganzen Zahlen a, b austauscht, zuerst mit `int*` und dann mit `int&` als Parametertyp. Zeigen Sie auch wie sich der Aufruf der entsprechenden Prozeduren syntaktisch unterscheidet.

4 Variablen-Lebensdauer

Welches Ergebnis erwarten Sie, wenn die folgende Funktion aufgerufen wird:

```
int* MySmartFunc() {
  int ghost_in_the_machine = 42;
  return &ghost_in_the_machine;
}
```

5 Memory-Management

Sie sollen eine Klasse schreiben, welche intern Memory auf dem Heap alloziert und verwendet. Stellen Sie mittels Destruktor sicher, dass ihr Memory immer korrekt abgeräumt wird.

Die Klasse soll einen rohen C-String verwalten und eine Methode anbieten, welche den String invertiert. Die Übergabe des Strings und der Getter des Strings soll mittels `std::string` implementiert werden.

Die Verwendung der Klasse soll so funktionieren:

```
#pragma once

#include <iostream>

#include "string_inverter.h"

void TestMemoryManagement() {
  std::cout << "-- TestMemoryManagement --" << std::endl;
  StringInverter h("hello");
  std::cout << "original: " << h.GetString() << std::endl;
  h.Invert();
  std::cout << "inverted: " << h.GetString() << std::endl;
  std::cout << "--------------------------" << std::endl;
}
```

6 *lvalue*-Referenzen

Ein *lvalue* wird von B. Stroustrup wie folgt definiert:

> We can allocate and use «variables» that do not have names, and it is possible to assign to strange looking expressions (e.g., *p[a + 10] = 7). Consequently, there is a need for a name for «something in memory.» This is the simplest and most fundamental notion of an object. That is, an object is a contiguous region of storage; an lvalue is an expression that refers to an object. The word lvalue was originally coined to mean «something that can be on the left-hand side of an assignment.»

Wenn Sie jetzt in einem C++-Programm eine Referenz wie folgt definieren:

```
double& dRef = 42;
```

wird der Compiler mit der Fehlermeldung meckern:

```
Error: initial value of reference to non-const must be a lvalue
```

Dagegen bleibt der Compiler ruhig, wenn Sie folgendes schreiben:

```
1  const double& dRef = 42;
```

Können Sie dieses Verhalten des Compilers anhand der Definition von *lvalue* erklären?