1 Pointer-Initialisierung

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

```
1 int i = 'a';
2 const int ic = i;
3 const int *pic = &ic;
4 int *const cpi = &ic;
5 const int *const cpic = &ic;
```

1.1 Lösung

Alle bis auf:

```
1 int *const cpi = &ic;
2   Error: a value of type "const int *" cannot be used to initialize an entity
        of type "int *const".
```

2 Referenz-Initialisierung

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

```
1 int& i = 'a';
2 const int ic = i;
3 const int& ric = &ic;
4 int& const rpi = &ic;
5 const int& const cpic = &ic;
```

2.1 Lösung

Nur die zweite. Alle anderen führen zu Compile-Fehlern.

```
1 int& i = 'a';
2   Error: initial value of reference to non-const must be an lvalue
3 const int& ric = &ic;
4   Error: a value of type "const int *" cannot be used to initialize an entity
        of type "const int &"
5 int& const rpi = &ic;
```

```
6     Error: initial value of reference to non-const must be an lvalue
7  const int& const cpic = &ic;
8     Error: a value of type "const int *" cannot be used to initialize an entity
          of type "const int &"
```

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.

3.1 Lösung

```
1  void SwapPointer(int* num1, int* num2) {
2    int tmp = *num1;
3    *num1 = *num2;
4    *num2 = tmp;
5  }
6
7  void SwapReference(int& num1, int& num2) {    // NOLINT[runtime/references]
8    int tmp = num1;
9    num1 = num2;
10   num2 = tmp;
11 }
12
13 void TestParameterPassing() {
14   int a;
15   int b;
16   SwapPointer(&a, &b);
17   SwapReference(a, b);
18 }
```

Übrigens: Diese Funktionalität ist bereits in der Standard-Library implementiert: `std::swap`.

4 Variablen-Lebensdauer

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

```
1  int* MySmartFunc() {
2    int ghost_in_the_machine = 42;
3    return &ghost_in_the_machine;
```

```
4 }
```

4.1 Lösung

Segmentation Fault, weil der Scope der lokalen Variable nur innerhalb der geschweiften Klammer reicht. Die Adresse der lokalen Variable `ghost_in_the_machine` zeigt nach dem Verlassen der Funktion auf irgendeine Speicheradresse, die sehr wahrscheinlich nicht mehr für den Prozess verfügbar ist.

In einer entsprechend Konfigurierten Toolchain wird dies aber gar nicht erst kompilieren sondern mit einem Fehler, wie dem folgenden, abbrechen:

```
1 error: address of local variable 'ghost_in_the_machine' returned
```

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:

```
1  #pragma once
2
3  #include <iostream>
4
5  #include "string_inverter.h"
6
7  void TestMemoryManagement() {
8    std::cout << "-- TestMemoryManagement --" << std::endl;
9    StringInverter h("hello");
10   std::cout << "original: " << h.GetString() << std::endl;
11   h.Invert();
12   std::cout << "inverted: " << h.GetString() << std::endl;
13   std::cout << "--------------------------" << std::endl;
14 }
```

5.1 Lösung

string_inverter.h

```cpp
#include <string>

class StringInverter {
 public:
  explicit StringInverter(const std::string& init);
  ~StringInverter();

  std::string GetString() const;
  void Invert();

 private:
  size_t character_count_;
  char* data_;
};
```

string_inverter.cpp

```cpp
#include <utility>

StringInverter::StringInverter(const std::string& init)
    : character_count_(init.length()) {
  data_ = new char[character_count_ + 1];

  for (size_t i = 0; i < character_count_; ++i) {
    data_[i] = init[i];
  }

  data_[character_count_] = 0;
}

StringInverter::~StringInverter() {
  delete[] data_;
}

std::string StringInverter::GetString() const {
  return data_;
}

void StringInverter::Invert() {
  for (size_t i = 0; i < character_count_ / 2; ++i) {
    std::swap(data_[i], data_[character_count_ - 1 - i]);
  }
}
```

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:

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

wird der Compiler mit der Fehlermeldung meckern:

```
1 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?

6.1 Lösung

`42` ist ein Integer-Literal. Als solches kann es nie ein *lvalue* sein, sondern ist immer ein *rvalue*. `dRef` ist ein Alias für ein variables double Objekt. Es darf nicht auf einen *rvalue* verweisen, sondern muss auf einen *lvalue* verweisen. Wäre `dRef` ein Alias für einen unveränderbaren **`double`**, dann wäre die 42 in Ordnung.

```
1 double d = 42;    // first create a variable d (can be initialized)
2 double& dRef = d; // then use the variable d as the initializer for dRef
```