Aufgabenblatt 5

- Christian Rebischke (432108)
- Sajedeh Majdi (493981)

Aufgabe 1

Teilaufgabe 1

Die Größen der Objekt Instanzen von `Empty`, `EmptyDerived` und `NonEmpty` sind wie folgt:

```
Size of Empty Object: 1
Size of EmptyDerived Object: 1
Size of NonEmpty Object: 1
```

Alle Objekte sind demnach 1 Byte groß. Dies ist zurückzuführen auf den C++-Standard, welcher vorassetzt, dass ein Objekt einer leeren Klasse immer 1 Byte Groß sein muss, da sonst bei mehreren Objekten einer leeren Klasse deren Adresse gleich wäre, da ja kein Speicher allokiert werden würde. Auch nachzulesen hier: https://en.cppreference.com/w/cpp/language/ebo

Teilaufgabe 2

Wenn `Empty` keinen Speicherplatz beanspruchen würde wäre `int b` das erste Element im `struct`. Der `struct`-Pointer würde also auf `int b` zeigen, anstatt auf das Objekt der Klasse `Empty`. Dies hätte zur Folge, dass kein Speicher mehr nachträglich für das Objekt `Empty` allokiert werden könnte. In diesem Fall hätte also `c.b` die Speicheradresse des `struct` und eine Speicheradresse zu `c.b` wäre nicht existent da das leere Objekt der Klasse `Empty` keinen Speicher allokiert.

Teilaufgabe 3

Gegeben seien drei Instanzen von folgenden `structs`:

```
struct Composite {
    Empty a;
    int b;
};

struct CompositeChar {
    Empty a;
    char b;
};

struct CompositeLongLong {
    Empty a;
    long long b;
};
```

Daraus ergeben sich folgende Größen (in Bytes):

```
Size of composite: 8
Size of compositeChar: 2
Size of compositeLongLong: 16
```

Die Größen lassen sich durch das `Address Alignment` erklären. Dadurch, dass diese `structs` nicht `packed` sind, müssen die `structs` an die Wortbreiten des Prozessors angepasst werden. Da ein `int` auf meinem System (Arch Linux, 64bit, x86 Architektur) 4 Byte groß ist, wird das struct auf 8 Bytes alligned. Es wird immer das größte Element für das Alignment benutzt. Bei einem `char` im struct ist die Größe des `struct` 2 Bytes, da das Objekt der leeren Klasse on-default 1 Byte groß ist und ein `char` ebenfalls 1 Byte beansprucht. `long long` beansprucht 8 Bytes auf meinem System, dementsprechend wird der Speicher auf 16 Bytes für das `struct` aligned. Bei den *überflüssigen* Bytes handelt es sich um ein sogenanntes `Padding` um den Speicher zu alignen. `Address Alignment` wird von Compilern benutzt um einen schnellen Zugang auf den Speicher zu ermöglichen. Siehe auch: http://www.catb.org/esr/structure-packing/#_padding

Aufgabe 3

Teilaufgabe 1

Das Programm lässt sich nicht übersetzen, da via der Funktion `test()` nicht auf die Funktion `foo()` innerhalb der Funktion `bar()` in der Klasse `C` zugegriffen werden darf, da es sich um eine private Vererbung von `B` handelt. Dadurch sind alle öffentlichen Mitglieder der Klasse `B`, private Mitglieder der Klasse `C` und somit kann `test()` nicht mehr auf die Funktion zugreifen.

Teilaufgabe 2

Die Lösung sieht wie folgt aus:

```
//....
class C : public B { // Fixed Class C via setting B public
public:
    void bar() { foo(); }
};
//....
```

Durch das Setzen von `private B` zu `public B` lässt sich die Funktion wieder aufrufen. Die Ausgabe sieht danach wie folgt aus:

```
A::foo
B::foo
A::foo
B::foo
A::foo
```

Aufgabe 4

Beim Ausführen des Programms kann folgender Fehler festgestellt werden:

```
In destructor ‘Foo::~Foo()’:
error: throw will always call terminate() [-Werror=terminate] throw
```

Was da passiert nennt sich “Stack unwinding”. Dadurch, dass die Exception im Destruktor ausgelöst wird, wird der Stack erneut abgewickelt. Dieses Fehlerverhalten ist allerdings nicht im Standard vorgesehen, da der Destruktor normalerweise dazu dient Objekte zu entfernen und Ähnliches. Dadurch würde der Stack weiter abgewickelt werden bis zu Objekten die nicht mehr existieren, deshalb sollten Exceptions laut Standard immer im Destruktor gecatched werden. Destruktoren sollten keine Exceptions schmeißen.