# 1 Analyse von Klassen-Padding

Um Memory-Zugriffe in `struct`/**`class`** und auch in Arrays zu optimieren, führt der Compiler Padding zwischen den einzelnen Attributen ein. In dieser Aufgabe sollen Sie eine Template-Funktion schreiben, welche es erlaubt, das Padding innerhalb der folgenden Struktur zu analysieren.

```
1 struct Default {
2   uint8_t  a;
3   uint32_t b;
4   uint16_t c;
5 };
```

Die Struktur wird mit dieser Definition das automatische Padding des Compilers erhalten. Um das Padding manuell zu beinflussen, kann `#pragma pack` folgendermassen verwendet werden:

```
1 #pragma pack(2)
2 struct Pack2 {
3   uint8_t  a;
4   uint32_t b;
5   uint16_t c;
6 };
7 #pragma pack()
```

Beachten Sie, dass Sie die Namen und die Reihenfolge der Attribute nicht verändern dürfen. Verändern Sie nur das Packing und die Typen der Attribute, damit ihre Analyse immer funktioniert.

Schreiben Sie die Template-Funktion, welche die zwei Instanzen von `T` anlegt, die Adressen der Attribute als `uintptr_t` speichert und anhand dieser Adressen die Attribut-Grösse und das Padding errechnet. Das Resultat soll folgendermassen auf die Konsole ausgegeben werden.

```
1 Type: 7Default
2 a: size=1 padding=3
3 b: size=4 padding=0
4 c: size=2 padding=2
```

Die Signature der Funktion sollte etwa so aussehen:

```
1 template<class T>
2 void AnalyzePadding() {
```

Um den («mangled») Namen des Typs `T` ausgeben zu können, können Sie `typeid(T).name()` verwenden. Beachte Sie, dass dies einen Platform-spezifischen String ausgibt.

### 1.1 Aufgabe

a) Implementieren Sie die Funktion und analysieren Sie damit unterschiedliches Padding und Attribut-Grössen.
b) Was passiert, wenn die Struktur die Attribute nach aufsteigender Grösse sortiert hat? Wie erklären Sie sich dieses Verhalten.

## 2 Memory-Pool

Gewisse Optimierungs-Probleme können stark von der Memory-Allozierung abhängen. Wenn ihr Algorithmus also sehr viel allozieren und deallozieren muss, wird sehr viel Zeit dabei verschwendet, da der Default-Allocator sehr komplex und auf alle möglichen Use-Cases ausgelegt ist. In solchen Fälle möchten Sie den Allocator eines Containers ersetzten oder ihren eigenen Memory-Pool direkt ansprechen.

In dieser Aufgabe sollen Sie einen solchen Memory-Pool schreiben und mittels Placement **new** den allozierten Speicher verwenden.

### 2.1 Aufgabe

#### 2.1.1 Teil 1

Um einen Memory-Pool zu implementieren gibt es viele unterschiedliche Strategien. Eine davon ist die *Free-List*. In dieser Übung soll die Free-List einmalig bei Instanziierung alles zu verwaltende Memory allozieren. Die interne Verwaltung soll mittels einem Stack von Pointern auf die zu verwaltende Memory-Blöcke organisiert sein. Das heisst, der zuletzt deallozierter Block wird bei der nächsten Allozierung direkt wieder ausgeliefert. Diese Struktur erlaubt somit *hot memory* (Speicher welcher noch im Cache ist, weil er erst kürzlich verwendet wurde), möglichst gut auszunutzen.

Schreiben Sie nun diese Klasse `FreeList` welche das folgende Interface erfüllen soll:

```
1  class FreeList {
2   public:
3    explicit FreeList(size_t block_count);
4
5    ExampleData* Allocate();
6    void Deallocate(ExampleData* const block);
7
8    size_t size() const;
```

Als Beispiel-Klasse für die Memory-Blöcke können Sie folgende Implementation verwenden:

example_data.h

```
#pragma once

#include <cstddef>
#include <cstdint>

struct ExampleData {
  static constexpr size_t kPayloadSize = 128;

  ExampleData(size_t index, uint8_t first_payload_byte);
  ~ExampleData();

  size_t index;
  volatile uint8_t payload[kPayloadSize];
};
```

example_data.cpp

```
#include "example_data.h"

ExampleData::ExampleData(size_t index, uint8_t first_payload_byte) : index(
    index) {
  payload[0] = first_payload_byte;
}

ExampleData::~ExampleData() {
  payload[0] = 0;
}
```

Um die Funktionalität zu überprüfen schreiben Sie am besten entsprechende Unit-Tests.

### 2.1.2 Teil 2

Als nächstes sollen Sie neben der normalen Deallozierungs-Methode noch eine weitere anbieten, welche die Instanz zuerst zerstört, bevor dann aber das Memory ebenfalls zurück in die Free-List eingefügt wird.

### 2.1.3 Teil 3

Schreiben Sie zudem einen Test, der nicht nur den Memory-Pool verwendet, sondern auf dem allozierte Memory mittels Placement **new** eine Klasse instanziiert.