Übungen OOSWE/Progr. 2 (C++) – KE 5

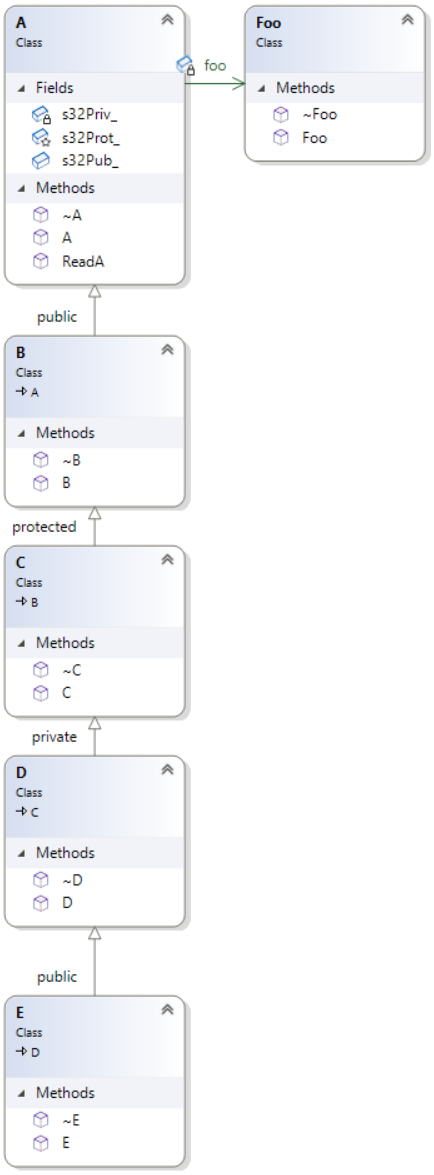
Der C/C++-Coding Styleguide ist einzuhalten.
Folgende Einstellungen sind für Debug und Release (All Configurations) vorzunehmen:

Einstellung	**Wert**
Solution Platform	x86
Properties->Conf. Properties->C/C++->General->Warning Level	Level4 (/W4)
Properties->Conf. Properties->C/C++->General->Treat Warnings As Errors	Yes (/WX)
Properties->Conf. Properties->C/C++->General->SDL checks	Yes (/sdl)
Properties->Conf. Properties->C/C++->Code Generation->Basic Runtime Checks	Default
Properties->Conf. Properties->C/C++->Code Generation->Security Checks	Enable Security Checks (/GS)
Properties->Conf. Properties->C/C++->Language->C++ Language Standard	ISO C++ 20 Standard

Legen Sie eine Solution an, die alle Aufgaben als Projekte enthält.
Die geforderten Kommentare sind in Englisch zu hinterlegen.

Aufgabe 1:

1.1 Realisieren Sie das folgende Klassendiagramm in C++ vollständig in der Datei „A.h“ (Namespace Hoo). Alle Funktionen sind somit implizit inline. Sobald der Kon- oder Destruktor einer Klasse in Debug aufgerufen wird, soll dies auf der Konsole ausgegeben werden. Foo foo sei private Objektvariable von A. Die Klasse Foo ist im Namespace vor class A zu deklarieren.



Das Klassendiagramm wurde automatisch vom *Class Designer* in Visual Studio erstellt (ggf. Nachinstallation). Der Class Designer ist kein vollständiges CASE-Tool wie der Enterprise Architect usw. Gewisse Einschränkungen sind zu akzeptieren. So wird bei den Beziehungstypen nicht weiter unterschieden (Assoziation, Aggregation, Komposition).

1.2 Versuchen Sie in den Konstruktoren von B, C, D und E die Attribute der Elterklasse A zu beschreiben.
Auf welche Attribute können Sie zugreifen? Erstellen Sie oberhalb von main als Kommentar eine Tabelle, die die Sichtbarkeit der Attribute s32Priv_, s32Prot_, s32Pub_ in den Klassen A-E festlegt.

1.3 Geben sie in main mit sizeof die Größe jeder Klasse aus. Allokieren Sie in main statisch ein Objekt der Klasse A und E.

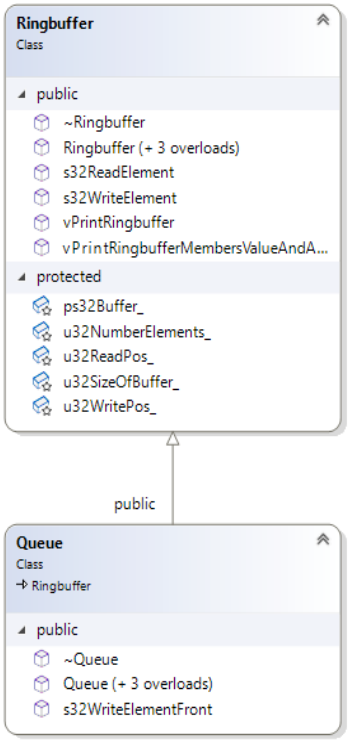
1.4 Allokieren Sie danach dynamisch zwei Objekte der Klassen A und B. Speichern Sie die Adressen beider Objekte in einem Zeiger der Klasse A (A*). Warum geht dies nicht für die Klassen C-E? Geben Sie den Speicher für die dynamischen Objekte wieder frei. Was fällt Ihnen dabei auf? Schreiben Sie die Antwort als Kommentar über main.

1.5 Erweitern Sie die Klasse A um eine Methode `void ReadA(A a)`. Kann man darin auf s32Priv_ von a zugreifen? Wo haben Sie dies schon realisiert? Schreiben Sie die Antwort als Kommentar über main.

Übungen OOSWE/Progr. 2 (C++) – KE 5

Aufgabe 2:

Es soll die Klasse **Ringbuffer** erweitert und daraus eine **Queue** generiert werden. Diese soll es erlauben, dass an der aktuellen u32ReadPos_ (vorderstes Element der Queue) ein Element eingefügt werden kann (u32WriteElementFront). **Erweiterung von Ringbuffer.cpp / .h!**



2.1 Kopieren Sie die Ringbufferklasse aus KE04_AG2. Es soll das FIFO-Prinzip „aufgeweicht“ werden, indem von Ringbuffer abgeleitet wird. Die zusätzliche Funktionalität von Ringbuffer sollte **eigentlich** nicht verändert werden. Dennoch ist eine Änderung notwendig:

- Die Sichtbarkeit der Attribute ist zu ändern. Warum?

Führen Sie danach eine public-Vererbung durch. Implementieren Sie s32WriteElementFront. Fügen Sie den folgenden Konstruktor hinzu:

```
Queue::Queue(uint32_t u32Size) noexcept(true) :
Ringbuffer(u32Size)
{ }
```

Dieser ruft den passenden Konstruktor der Basisklasse auf. Ohne den Zusatz `: Ringbuffer(u32Size)` würde der leere Konstruktor aufgerufen werden. Fügen Sie in allen Konstruktoren und in dem Destruktor Konsolenausgaben in Debug hinzu.

2.2 Allokieren Sie in main dynamisch ein Objekt der Klasse Queue (Queue* verwenden) und verifizieren Sie die Korrektheit Ihre Implementierung, indem Sie alle Methoden (auch geerbte) von Queue aufrufen. Löschen Sie anschließend das Objekt wieder. Verwenden Sie die Heapüberwachung, um Memory Leaks zu finden.

2.3 Implementieren Sie den Kopier- und den Move-Konstruktor (https://stackoverflow.com/questions/4086800/move-constructor-on-derived-object) für Queue. Stellen Sie sicher, dass der richtige Konstruktor der Basisklasse aufgerufen wird. Allokieren Sie dynamisch erneut ein Queue-Objekt, wobei 15 Elemente gespeichert werden können. Allokieren Sie sich weitere zwei dynamische Objekte von Queue in main, welche bei der Instanziierung den Kopier- und den Move-Konstruktor verwenden.

```
//Ringbuffer.h
Queue(const Queue& otherQueue) noexcept(true);
Queue(Queue&& otherQueue) noexcept(true);

//Ringbuffer.cpp
Queue::Queue(const Queue& rq) noexcept(true) : Ringbuffer(rq)
{}
Queue::Queue(Queue&& rq) noexcept(true): Ringbuffer(std::move(rq))
{}

// in main
// Copy Constructor is called
MyDataStructures::Queue* pQueue2 = new MyDataStructures::Queue(*pQueue1);
// Move Constructor is called
MyDataStructures::Queue* pQueue3 = new MyDataStructures::Queue(std::move(*pQueue1));
```

Geben anschließend die drei dynamischen Objekte aus (pQueue1, pQueue2, pQueue3). Verifizieren Sie die Korrektheit des Copy- und des Move-Konstruktors. Nach dem Löschen dieser drei Objekte dürften auch keine weiteren Memory Leaks vorhanden sein.

Übungen OOSWE/Progr. 2 (C++) – KE 5

Aufgabe 3:
Kopieren Sie die Ringbufferklasse aus KE05_AG2. Die Ringbufferklasse verletzt das sogenannte *Single-Responsibility-Prinzip*. Ringbuffer hatte bisher zwei Verantwortlichkeiten:

- Realisierung der Datenstruktur-Funktionalität
- Ausgabe der Datenstruktur.

Erstellen Sie eine neue Klasse RingbufferPrint im namespace MyDataStructures, welche die Klassenmethoden

```
static void vPrint(const Ringbuffer& crRingbuffer);
static void vPrintMembersValueAndAddress (const Ringbuffer& crRingbuffer);
```

enthält. Bei Verwendung dieser Klassenmethoden muss somit kein Objekt der Klasse RingbufferPrint instanziiert werden. Kopieren Sie den entsprechenden Code aus Ringbuffer in diese Methoden und löschen Sie anschließend die korrespondierenden Methoden aus Ringbuffer (h und cpp).

Dadurch hat die Klasse Ringbuffer nur noch eine Verantwortlichkeit.

Damit die Klasse RingbufferPrint auf die protected Attribute von Ringbuffer und Queue zugreifen kann, müssen beide Klassen dies erlauben. Überlegen Sie, wie Sie dies in Ringbuffer realisieren! Vererbt sich diese Änderung auch auf Queue?

Testen Sie die Erweiterung, indem Sie ein Objekt der Klasse Ringbuffer und ein Objekt der Klasse Queue **statisch** in main instanziieren. Rufen Sie vPrint und vPrintMembersValueAndAddress von RingbufferPrint danach auf. Im Anschluss daran sind die Methoden s32WriteElement, s32ReadElement (Ringbuffer- und Queue-Objekt) sowie s32WriteElementFront (nur für Queue-Objekt) aufzurufen.

Rufen Sie danach vPrint und vPrintMembersValueAndAddress von RingbufferPrint erneut auf.

Verifizieren Sie Korrektheit der beiden Klassenmethoden.

Es werden nur **zwei** Objekte statisch instanziiert. Wie oft werden ein Konstruktor und ein Destruktor aufgerufen?