**Praktikumsaufgaben 2**

Aufgabe 1.)

a.)

Der Operator dynamic\_cast wird zu expliziten Typkonvertierung verwendet. Mit ihm ist es möglich, zur Laufzeit Pointer oder Referenzen auf Objekte in der Klassenhierarchie nach oben oder unten zu casten. Die Syntax ist: dynamic\_cast<new\_type>(expression). Wobei new\_type ein Pointer oder eine Referenz auf die Klasse darstellt, zu welcher konvertiert werden soll, in expression wird das zu konvertierende Objekt angegeben.

Wenn in der Klassenhierarchie nach unten gecastet werden soll, muss es sich um eine polymorphe Klasse handeln, die Klasse von der nach unten gecastet werden soll muss also eine virtuelle Methode enthalten, andernfalls gibt dynamic\_cast einen Nullpointer zurück wenn es sich um einen Pointer handelt oder wirft eine std::bad\_cast-exception wenn es sich um eine Referenz handelt. Dies sind auch die Rückgaben, wenn der cast allgemein nicht erfolgreich war. Handelt es sich also um Pointer sollte man nach der Konvertierung immer auf den Nullpointer überprüfen.

War der cast jedoch erfolgreich, so gibt dynamic\_cast einen Wert des Typs new\_type zurück.

b.)

Der Operator static\_cast wird zur expliziten Typkonvertierung verwendet. Mit ihm ist es bspw. möglich, zur Compile-Zeit numerische Werte (zum Beispiel float zu int), oder einen Pointer einer Basisklasse zu einem Pointer einer abgeleiteten Klasse, zu konvertieren. Die Syntax ist:

static\_cast<new\_type>(expression)

Dabei wird das in expression angegebene Objekt in den new\_type umgewandelt.

c.)

Bei static\_cast wird zur Laufzeit keine Typüberprüfung durchgeführt, sondern nur zur compile-Zeit, während bei dynamic\_cast eine Typüberprüfung zur Laufzeit stattfindet. Dies macht den Befehl zwar „langsamer“, aber dafür auch sicherer, da zur Laufzeit überprüft wird, dass keine inkompatiblen Datentypen gecastet werden. Bei einem static\_cast ist zudem die Konvertierung von einem Pointer einer polymorphen Basisklasse zu einer abgeleiteten Klasse nicht definiert.

Static\_cast sollte also eher dann benutzt werden, wenn zum Beispiel numerische Werte (double zu int etc.) konvertiert werden sollen (also die veraltete c-type-Konvertierung ersetzen). Man kann damit zwar auch innerhalb einer Vererbungshierarchie von Klassen casten (solange es keine polymorphe Basisklasse ist), jedoch ist hier dynamic\_cast sicherer, da zur Laufzeit auf den korrekten Datentyp geachtet wird und im Gegensatz zu static\_cast kein undefiniertes Verhalten aufweist, falls in einer Hierarchie nach unten zu einem Typ gecastet wird, der gar nicht dem des Objektes entspricht.

Aufgabe 2.)

a.)

Man kann mit dem friend-Operator innerhalb einer Klasse diese als „friend“ einer anderen Klasse oder einer Methode definieren. Dafür schreibt man einfach in die Klasse, die der „friend“ der anderen Klasse oder Methode sein soll:

friend *funktionsdeklaration* bzw. friend class *klasse*

Dies bewirkt, dass die als Freund deklarierte Methode oder Klasse vollständigen Zugriff auf alle Felder und Methoden der Klasse hat (auch private und protected). Freundschaft wird jedoch nicht vererbt oder transitiv weitergegeben.

Diese Methode ist dann sinnvoll, wenn eine andere Klasse zugriff auf alle Felder bekommen soll, ohne das gleich alle Klassen Zugriff haben (also alles auf public, womit das Kapselprinzip komplett hinfällig wäre) und ohne das die Klasse in einer Vererbungshierarchie mit der anderen Klasse steht, also weder Subklasse, noch Oberklasse ist.

Mögliche Verwendungszwecke wären zum Beispiel, wenn man den Code testet und die Testmethoden umfangreichen Einblick in die Werte der Klasse bekommen sollen, ohne das dabei die Kapselung zerstört wird, indem man diese Testmethoden als friend deklariert.

b.)

(In PyramidBlock.h muss die Methode addPyramidBlock der Klasse Field als friend hinzugefügt werden, damit in Field.cpp in der Methode addPyramidBlock mit new ein neuer PyramidBlock angelegt und hinzugefügt werden kann.) 🡪 doch ganze Klasse als Friend?, weil fehlen …. Friend-Klassen

Aufgabe 3.)

a.)

Nachdem die Methode World::split mit dem Eingabestring als Parameter aufgerufen wurde, werden zuerst 3 Variablen definiert und deklariert. Zum einen den Char separator, indem gespeichert bei welchem Zeichen der String gesplittet werden soll, hier ein Leerzeichen. Dann noch die Variable begin, die mit 0 initialisiert wird und in dem folgenden Schleifendurchlauf dann immer auf die Position des letzten Leerzeichens gesetzt wird, sowie den vector result, indem die aufgesplitteten Bestandteile des Eingabestrings als Strings gespeichert werden.

Der eigentliche Algorithmus der split-Methode befindet sich in einer while-Schleife, die solange über den Eingabestring iteriert, bis dieser komplett durchlaufen ist. Dies geschieht dadurch, dass auf den Eingabestring „in“ die Methode find\_first\_not\_of() angewendet wird. Diese Methode erhält 2 Parameter, zum einen den separator und den aktuellen wert von begin. Begin „sagt“ der Methode, wie weit der Eingabestring bereits durchlaufen wurde und an welcher Stelle sie beginnen soll. Der Separator ist der char, der nicht gefunden werden soll. find\_first\_not\_of() gibt also, beginnend ab begin, die Stelle des Eingabestrings wieder, an dem zum Ersten mal nicht das Leerzeichen steht. Diese Stelle wird als der neue begin-wert gespeichert. Std::string::npos gibt die maximale Anzahl der Stellen des Eingabestrings, also die Länge, an. Solange der gerade neu festgelegte begin-Wert noch nicht das Ende des Strings erreicht hat, ist die Schleifenbedingung erfüllt und es wird in die while-Schleife gesprungen.

Innerhalb der Schleife wird mittels der Methode find\_first\_of() auf dem Eingabestring die Position des Strings in der variablen end gespeichert, an welcher nach dem gerade neu festgelegten begin das nächste Leerzeichen folgt. Nun hat man also durch die Positionsvariablen begin und end auf dem Eingabestring einen Substring ohne ein Leerzeichen, welchen man durch die Methode substr(), der man den Beginn des Teilstrings(begin) und die Länge des Substrings (end-begin) übergibt, erhält. Dieser Teilstring wird mittels der Methode push\_back() an das Ende des vectors result hinzugefügt.

Nun wird die Variable begin auf end gesetzt, damit von dort aus in der nächsten Iteration der while-Schleife der nächste Teilstring ausfindig gemacht werden kann.

Die while-Schleife wird solange wiederholt, bis das Ende des Eingabestrings erreicht wurde, dann wird der vector result, der nun die Substrings des Eingabestrings enthält, zurückgegeben.

Unsicher: extra Hilfsfunktionen anlegen?