Fabian Hupfeld (35523329) und Konstantin Stitz (35532242)

Praktikumsaufgaben 2

Aufgabe 1.)

a)

dynamic_cast: *dynamic_cast<new_type>(expression)*

Ein dynamic_cast konvertiert Objekt-Pointer/Referenzen zur Laufzeit, hierbei wird auch zur Laufzeit überprüft, ob diese Konvertierung valide ist. Eine Konvertierung auf eine Oberklasse gibt den konvertierten Typ-Pointer/Referenz wieder, falls es sich bei new_type um eine Oberklasse des zu konvertieren Types handelt.

Eine Konvertierung auf eine Unterklasse ist nur möglich, wenn die Oberklasse eine polymorphe Klasse ist, also mindestens eine virtuelle Methode enthält. Falls die Konvertierung nicht erfolgreich ist, wird ein Null-Pointer zurückgegeben (wenn new_type ein Pointer ist). Wenn new_type eine Referenz ist wird stattdessen eine bad_cast-exception geworfen.

b)

static_cast: *static_cast<new_type>(ptr)*

Der Operator static_cast wird zur expliziten Typkonvertierung zur Compile-Zeit 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. Falls die Typen nicht verwandt sind, wird ein Compiler Error ausgegeben.

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) oder sichere Konvertierungen stattfinden. Man kann damit also 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.)

Mit dem friend-Operator kann eine andere Klasse Zugriff auf protected/private Variablen/Methoden einer anderen Klasse bekommen. Dieser Zugriff wird allerdings nicht vererbt.

Mit friend class KlasseX kann KlasseX auf alle Variablen/Methoden der Klasse zugreifen.

Mit friend type KlasseX:name(params) kann diese Funktion name von KlasseX auf alle Variablen/Methoden der Klasse zugreifen.

Das ist sinnvoll, falls manche Klassen auf Werte einer anderen Klasse Zugriff brauchen, ohne dass diese Werte allgemein zugänglich seien müssen. (Kapselprinzip)

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

b.)

In PyramidBlock.h muss die Klasse Field als friend gesetzt werden, da in dieser in der Methode addPyramidBlock mit new PyramidBlock ein neues Objekt von PyramidBlock erstellt wird.

In CubeBlock.h muss die Klasse Field als friend gesetzt werden, da in dieser in der Methode addCubeBlock mit new CubeBlock ein neues Objekt von CubeBlock erstellt wird.

In Block.h muss die Klasse Field als friend gesetzt werden, da in dieser in der Methode addBlock die Klasse Block verwendet wird.

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 speichert wie weit der Eingabestring bereits druchlaufen wurde und übergibt dies der Methode. 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.