ACIC BIF5C1

Letzte Änderung am 22.10.2020

Josef Wermann
if18b182@technikum-wien.at

Inhalt

1. Übung1 – Simple String Class

Die Abgabe beinhaltet 3 files:

- **MyString.hpp:** Headerfile des MyString-Class
- **MyString.cpp:** Implementierung der MyString-Class
- **MyStringClassMain.cpp**: Main-Function, um die MyString-Class zu testen.

1.1. MyString.hpp

Das Headerfile definiert die Methoden, die die String-Class bis jetzt beinhaltet (Erweiterung folgt wohl in den nächsten Übungen)

Private werden hierbei die Variablen char* data und size_t length definiert: ein char-Pointer für das Array, in dem sich der String befindet, bzw. befinden soll, sowie die Länge des String. size_t habe ich statt unsigned int generell verwendet, zum Teil aus Gewohnheit, zum anderen, um abgesichert zu sein, auf unterschiedlichen Rechnerarchitekturen fehlerfrei kompilieren zu können.

MyString(const char* x, const char* y, const std::size_t length) ist ein spezieller privater Constructor, der als Hilfskonstruktor für die Concatenate-Funktion dient.

Weiters werden unterschiedliche public constructors definiert (simple und const char*), c_str(), um auf das char-array zugreifen zu können, das selbsterklärende Concatenate(), ein destructor, der auf „default“ gesetzt wird und eine friend-Referenz auf den cout(<<) overload, um mittels cout, MyString einfacher auf die Konsole ausgeben zu können.

Size_t length speichert die Länge ohne das abschließende `\0` des strings, weil ich der Meinung bin, dass hier nur die tatsächliche Länge des strings interessant ist. Geholt werden, kann diese Länge über die public function GetLength();

1.2. MyString.cpp

Der private Constructor MyString(const char* x, const char* y, const std::size_t length) nimmt die beiden strings, erhält ausserdem die Länge des neuen zu bildenden Strings, und überträgt die Strings nacheinander mithilfe einer Lamdafunktion in einen neuen String. Ein abschließendes ‘\0‘ wird selbstverständlich angehängt.

c_str() gibt den char* data zurück.

GetLength gibt size_t length zurück.

Die public constructors sollten im code selbsterklärend sein. Relevant ist eigentlich nur, dass der eine constructor die Aufgabenstellung, dass MyString durch ein const char* initialisiert werden kann, gewährleistet.

MyString MyString::Concatenate(const MyString str) const berechnet die Länge des neuen Strings und ruft dann als return den privaten constructor auf, der als Hilfskonstruktor für eben diese Funktion implementiert ist. Wie auch bei anderen in diesem file implementierten funktionen, soll das const am Ende der Funktionsdefinition gewährleisten, dass das Objekt selbst nicht verändert wird.

std::ostream& operator<<(std::ostream& os, const MyString& str) schließlich ist als reine Hilfe gedacht und überladet den cout operator <<, um eben MyStrings über cout ausgeben zu können und damit leichter testen zu können.

1.3. MyStringClassMain.cpp

Hier werden innerhalb der Main-Methode die bisher implementierten Funktionen getestet. Unterschiedliche MyStrings werden angelegt (dabei werden die unterschiedlichen Möglichkeiten MyString str1("Hallo"); und auto* str2 = new MyString("Welt!"); genutzt. Ein MyString wird mithilfe von c_str() Zeichen für Zeichen mit Abständen ausgegeben, und schließlich werden 2 Strings zu einem neuen verknüpft und die Gesamtlänge des neuen Strings ausgegeben.

2. Übung 2 - Rule of Three/Five

2.1. Überarbeitung

Zwecks einfacherer Bewertung und Übersicht wurde die Projektstruktur verändert:
Der gesamte Code (Main-Funktion und String-class) befindet sich nun in der Datei „wermann.cpp“.

Ein Destructor, der das data-array deleted wurde ergänzt.

Für Testzwecke wurde der cout(<<) operator override so modifiziert, dass er, wenn er einen nullptr bekommt „nullptr“ ausgiebt, statt eine exception zu werfen.

Namespace MyString wurde um die String-class gelegt.

2.2. Copy constructor & copy assignment operator

Der copy-constructor nimmt als einen const String& als Argument, da dieser ohnehin nicht verändert werden soll und darf. Die length wird dabei direkt in der Initialisierungsliste zugewiesen.

Im copy assignment operator findet die Zuweisung innerhalb der geschwungenen Klammern statt, weil zuvor ein Check ausgeführt wird, ob es sich nicht um ein self-assignment handelt, und in diesem Fall die Zuweisung unnötig wäre.

Returniert wird in beiden Fällen ein this-Pointer.

In der Main-Methode finden sich Tests zu Beiden, wo ich mittels Breakpoints gecheckt habe, ob die richtigen Funktionen aufgerufen werden und die richtigen Strings kopiert werden.

2.3. Move constructor & move assignment operator

Sowohl der move constructor als auch der move assignment operator benutzen „noexcept“, da sie keine Exceptions werfen, und weil für move constructor und assignment operator so vorgesehen, ansonsten garnicht aufgerufen werden würden.

Wieder wird im Fall des move constructors direkt per Initialisierungsliste zugewiesen, beim move assignment operator allerings nicht, weil davor ein self-assignment check durchgeführt wird.

Es wird kein const entgegengenommen, weil die „source“ verändert werden muss => data wird auf nullptr gesetzt und length auf 0. Hab mich zwar auch herumgespielt und versucht die „source“ z.B. Mittels destructur direkt zu vernichten, hat aber nicht geklappt :D

Wieder finden sich in der Main-Methode einige Tests, in denen ich mit Breakpoints prüfe, ob die richtige Funktion aufgerufen wird und per cout checke, ob der richtige String bewegt wurde.

Eine Zusätzliche Funktion MoveString(MyString& str1, MyString& str2) const wurde hinzugefügt um den move assignment operator zu testen. Wieder const-Methode, weil direkt in der Methode nichts verändert werden soll.

2.4. Compiler Test

Zum Schluss habe ich das Programm mit unterschiedlichen Compilern getestet:

Es gab nur ein Warning unter Clang betreffend den Aufruf in der Main, der das self-assignment prüfen soll. Das war zwar beabsichtigt, um aber keine Warnings zu produzieren, wurde die Zeile auskommentiert.

3. Übung 3 – Overloaded Operators

3.1. += Overloads

2 Overloads für += wurden implementiert:

1) Mit Argument „const MyString&“: const weil der „addierte“ String nicht verändert werden soll. Die Funktion selbst ist nicht const, da der originale String sehr wohl verändert wird. Die Länge des neuen Strings wird ausgerechnet, die beiden Strings werden in einen neuen String kopiert und schließlich wird der Original-String gelöscht bzw. überschrieben und als Referenz zurückgegeben.
2) Mit Argument „const char*“: selbige const-Begründungen, wie oben. Auch ansonsten gleiches Bild wie oben, mit dem Zusatz, dass die Länge des const char* erst ausgelesen wird, um einen neuen MyString erstellen zu können.

Beide Overloads werden in der Main-Methode gestestet und darauf ausgerichtet, so wenige Kopiervorgänge zu machen, wie möglich.

Visual Studio meckert zwar ein Warning, dass die Kopie des übergebenen Wertes jeweils zu einem Buffer overrun führen kann, das kann er aber nicht! Auch tritt das Warning mit unterschiedlichen compilern in der bash nicht auf.

3.2. + Overloads

3 Overloads für + wurden implementiert. Alle wurden als friend deklariert, da sie ja einen neuen String zurückgeben und ansonsten nur ein Argument besitzen könnten:

1) Nimmt zwei const MyString Referenzen entgegen: Erstellt einen neuen String in dem er str1 kopiert und str2 mit der concatenate-Funktion anhängt. Hier den += operator zu benutzen wäre falsch, weil eine zusätzliche Kopie notwendig wäre, da ja die originalen Strings nicht überschrieben werden sollen.
2) Nimmt eine const MyString und ein const char* -Referenz entgegen. Analog zu dem vorigen, nur dass hier tatsächlich der += operator angewendet werden kann, weil kein zusätzlicher Kopiervorgang notwendig wird.
3) Zum Spaß noch ein dritter overload, der zuerst ein const char* und danach ein const MyString Referenz entgegennimmt: Hier wird der char* in ein MyString-Objekt umgewandelt und der übergebene MyString per concatenate-Funktion angehängt.

Sämtliche Overloads wurden in der Main-Methode getestet und per Breakpoints überprüft, ob sie den kürzesten Weg nehmen. Ausserdem wird gechekct ob die Originalstrings erhalten geblieben sind.

3.3. Conversion to const char*

Einfache implizite coneversion operator MyString to const char*, indem this.c_str() zurückgegeben wird. Getestest mit puts().