Fachhochschule Nordwestschweiz
Hochschule für Technik

Programmieren in C++

Sander Staal, Prof. Dr. Christoph Stamm christoph.stamm@fhnw.ch

Übung 3: Templates

Lernziele

- Sie werden die Grundlagen von Stack-basierten Programmiersprachen verstehen und in der Lage sein, eigene solche Programme zu entwickeln.
- Sie werden die Prinzipien von *Templates*, *spezialisierten Template*s und *Variadic Templates* repetieren und vertiefen.
- Sie werden in der Lage sein, *Concepts* als erweitertes Tool für die Spezialisierung von Templates zu verwenden.

1 Einführung in das Thema

In dieser Testatübung vertiefen Sie Templates und Concepts in C++. Anhand zweier möglicher Implementationen einer Klasse, welche die Auswertung einer Stack-basierten Programmiersprache simuliert, werden Sie die verschiedenen Vorteile von Templates und Concepts kennenlernen.

1.1 Stack-basierte Programmiersprachen

Die **Stack-basierte Programmierung** ist ein Programmierparadigma, das hauptsächlich auf dem Stack-Maschinenmodell beruht. Dies bedeutet, dass Parameter nur mittels einer oder mehreren zentralen Stack-Datenstrukturen übergeben werden können. Herkömmliche Anweisungen (wie z.B. Schleifen) müssen daher häufig modifiziert werden, damit diese auf Stack-basierten Systemen funktionieren.

Die meisten der Stack-basierten Programmiersprachen arbeiten mit der sogenannten *Postfixnotation*, um die Argumente / Parameter für eine Anweisung anzugeben. Ein Beispiel: Die Addition `1 + 2` würde man in den herkömmlichen Programmiersprachen normalerweise mittels der *Infixnotation* wie folgt angeben: `1 plus 2`. Bei der Postfixnotation gibt man aber zuerst die Argumente an, bevor man anschliessend den Befehl schreibt. In unserem Beispiel wäre dies also `1 2 plus`. Als Randnotiz sei hier noch erwähnt, dass die umgekehrte Schreibweise der Postfixnotation (sprich, `plus 1 2`) auch als *Präfixnotation* bezeichnet wird. Für diese Übung hier ist diese dritte Notation jedoch nicht von weiterer Relevanz.

Der Vorteil der Postfixnotation besteht darin, dass immer genau ausgedrückt wird, welche Parameter auf dem Stack für eine Operation verwendet werden. Jeder Befehl, welcher bei einer Stack-basierten Programmiersprache nämlich ausgeführt wird, nimmt die erforderlichen Parameter immer vom oberen Ende des Stacks und platziert anschliessend das entsprechende Resultat der Operation wieder (oben) auf den Stack zurück. In unserem Beispiel `1 2 plus` würden wir also als Befehl eine Plus-Operation ausführen. Da dieser Operator zwei Operanden benötigt, werden anschliessend die beiden Parameter 1 und 2 vom Stack entfernt. Die beiden Parameter werden addiert und das Resultat `3` der Plus-Operation wird wieder zurück auf dem Stack platziert.

Beispiel PostScript

Einige der bekanntesten Programmiersprachen, welche dieses Paradigma verwenden, sind unter anderem Forth, STOIC, RPL und PostScript. Letzteres wird vor allem bei Druckern verwendet, um den Aufbau einer Druckdatei oder Grafik zu beschreiben, so dass diese auf allen Geräten möglichst verlustfrei in beliebiger Grösse und Auflösung gedruckt werden können. Hierbei liest der Interpreter auf dem Drucker das PostScript Programm und generiert die entsprechende Vektor- oder Rastergrafik. Hier ist ein kurzes Beispiel, wie in etwa ein PostScript Programm aussehen könnte:

```
/Courier 20 selectfont  % Schrift und Schriftgrösse auswählen
```

```
0 1 0 setrgbcolor       % Aktuelle Farbe auswählen
30 40 moveto            % Setze (30, 40) als Schreibposition...
(Hallo Welt!) show      % ... und gib dort den Text aus.
showpage
```

Dieses Programm schreibt «`Hallo Welt!`» in grüner Farbe an der Position (30, 40) in der Rastergrafik. Wie Sie sehen, werden immer zuerst die Parameter angegeben, damit diese auf den Stack platziert werden können, bevor anschliessend der dazugehörige Befehl ausgeführt wird. So wird z.B. durch die drei Parameter 0 1 0 angegeben, dass die Farbe Grün verwendet werden soll (im RGB Raum) oder dass die aktuelle Schreibposition auf den Punkt (30, 40) gesetzt wird.

1.2 Postfix-Interpreter in C++

Das Ziel dieser Übung wird nun sein, dass Sie Ihren eigenen Postfix-Interpreter entwickeln, so dass wir unsere *eigenen* Stack-basierten Programme in C++ implementieren können. Da aber natürlich die Unterstützung einer vollwertigen Programmiersprache (wie z.B. PostScript) den Rahmen dieser Übung deutlich sprengen würde, beschränken wir den Interpreter vorerst einmal nur auf die Auswertung von mathematischen Termen. Sprich, der Postfix-Interpreter soll am Anfang zuerst einmal nur die arithmetischen Grundoperationen wie Addition oder Multiplikation unterstützen.

1.3 Auswertung arithmetischer Operationen

Wie Sie bereits gesehen haben, folgt bei der Postfixnotation ein Operator immer seinen Operanden. Falls man also zum Beispiel die beiden Zahlen 3 und 5 addieren möchte, so schreibt man dies nicht als `3 + 5` (wie in der Infixnotation), sondern als `3 5 +`. Diese Notation funktioniert auch, falls wir mehrere Operationen in unserem Term besitzen. Hierbei muss nur immer sichergestellt werden, dass ein Operator immer unmittelbar nach dessen Operanden folgt. So wird z.B. der Term `3 – 4 + 5` (Infixnotation) mittels Postfixnotation als `3 4 – 5 +` dargestellt. Hierbei enthält der Term als erstes den Minusoperator, welcher den Wert 4 von 3 subtrahiert. Anschliessend wird dieses Resultat mit dem Wert 5 addiert.

Ein grosser Vorteil der Postfixnotation bei mathematischen Termen ist, dass *keine* Klammern benötigt werden, um die Reihenfolge der Auswertung der Operationen anzugeben (unter der Annahme, dass jeder verwendete Operator eine klar definierte Anzahl von Operanden besitzt). Bei der Infixnotation kann der Term `3 – 4 × 5` sowohl mittels den impliziten Klammern `3 – (4 × 5)`, als auch mit den expliziten Klammern `(3 - 4) × 5` geschrieben werden. Abhängig davon, welche Klammern verwendet werden, wird der Term auch zu einem anderen Wert ausgewertet. Bei der Postfixnotation ist dieses Szenario nicht möglich, da die Auswertungsreihenfolge immer klar gegeben ist. Der erste Term wird als `3 4 5 × -` geschrieben, während der zweite Term als `3 4 - 5 ×` geschrieben wird. Anbei finden Sie ein paar weitere Übungen, welche helfen sollen, dieses Prinzip etwas vertiefter und sattelfester zu verstehen:

Lernkontrolle

a) Werten Sie die folgende Postfix-Terme aus:

(i) `3 2 + -1 ×`

(ii) `3 1 + 5 - 2 4 - +`

(iii) `5 -2 15 13 - × +`

(iv) `-4 -3 × 1 5 + 2 / -`

b) Wandeln Sie folgende Terme von der Infixnotation in die Postfixnotation um:

(v) `(100 / 5) × 3`

(vi) `(-1) - (4 / 2)`

(vii) `5 × (10 + 6 + 3) - (4 / 2)`

(viii) `(8 × (1 + 3)) / 2`

Lösung

a) Die Lösungen sind:

(i) `(3 + 2) × (-1) = -5`

(ii) `((3 + 1) - 5) + (2 - 4) = -3`

(iii) `5 + ((-2) × (15 - 13)) = 1`

(iv) `(-3 × -4) - ((1 + 5) / 2) = 9`

b) Die Terme sind wie folgt:

(v) `100 5 / 3 ×`

(vi) `-1 4 2 / -`

(vii) `5 10 6 + 3 + × 4 2 / -`

(viii) `8 1 3 + × 2 /`

2 Aufgaben

Für die erfolgreiche Bearbeitung dieser Testatübung, lösen Sie bitte die nachstehenden Aufgaben. Beachten Sie, dass die vierte Aufgabe «Variadic Templates» als *optional* markiert ist und die Bearbeitung dieser Aufgabe daher freiwillig ist.

2.1 Aufgabe 1

Ihre erste Aufgabe besteht darin, eine Klasse zu implementieren, welche Integer-Ausdrücke in Postfixnotation auswerten kann. Verwenden Sie hierfür die bereits bestehende Dateien `IntegerInterpreter.h` und füllen Sie den fehlenden notwendigen Code ein. Sie können dabei annehmen, dass die Klasse Interpreter (vorerst) nur die vier arithmetischen Basisoperationen unterstützen muss: `+, -, *, /`.

Da es keinen einfachen Weg gibt, wie Operatoren und Operanden zusammengemixt werden können (da beide unterschiedliche Datentypen darstellen), verwenden wir in dieser ersten Implementation nur Strings, um unsere Integer-Ausdrücke mittels Postfixnotation auszudrücken:

```
interpreter.evaluate({“5”, “2”, “3”, “+”, “*”});
```

Nachdem Sie den notwendigen Code implementiert haben, stellen Sie bitte auch sicher, dass alle Unit-Tests in der Datei `UnitTest1.cpp` erfolgreich bestanden werden.

Tipp: Sie können einen String in C++ mittels der Funktion `std::stoi` in einen Integer umwandeln.

2.2 Aufgabe 2

Wie Sie vielleicht bereits bemerkt haben, funktioniert die momentane Implementation des Postfix-Interpreters nur für Integer-Ausdrücke, nicht jedoch für andere primitive Datentypen wie Gleitkommazahlen. Ihre nächste Aufgabe besteht nun darin, die Implementation des Interpreters mittels Templates zu erweitern, so dass diese für *beliebige* primitive Datentypen funktioniert (welche die vier Basisoperationen unterstützen). Für die Lösung dieser Aufgabe verwenden Sie bitte die bereits vorgegebene Datei `GenericInterpreter.hpp`.

String Konvertierung

Die erste Schwierigkeit, welche bei dieser Aufgabe auftaucht, ist ein gegebener String (mittels welchen wir die Zahlen in unseren Ausdrücken angeben) in den korrekten primitiven Datentypen umzuwandeln. Die Funktion `std::static_cast()` funktioniert in diesem Fall leider nicht, da es keinen trivialen cast von einem String zu einem primitiven Datentypen gibt. Wir können jedoch einen anderen Trick

verwenden, indem wir unsere eigene Konvertierungsfunktion für Strings (aka `convertString()`) mittels spezialisierten Templates implementieren! Hierfür definieren wir als erstes die folgende Basis-Template Funktion in unserer Header-Datei:

```
template<typename T>
T convertString(const std::string& s) {
    throw "Type not supported";
    return T();
}
```

Diese Basisimplementierung der Funktion gibt für jeden String den Standardwert des Datentyps `T` zurück. Fügen Sie nun spezialisierte Implementationen für die Template-Funktion `convertString()` in der Header-Datei `GenericInterpreter.hpp` ein, welche einen String in die gängigsten primitiven Datentypen umwandeln kann. Folgende Hilfsfunktionen könnten hierfür nützlich sein:

`std::stor`: Konvertiert einen String in einen `int`.

`std::stoll`: Konvertiert einen String in einen `long long`.

`std::stod`: Konvertiert einen String in einen `double`.

`std::stof`: Konvertiert einen String in einen `float`.

Ihre Implementation sollte alle diese Datentypen unterstützen. Sie können die Unit-Tests in der Datei `UnitTest2.cpp` verwenden, um Ihre Implementation zu testen.

Tipp: Damit es bei den Unit-Tests zu keinen Problemen kommt, muss man die spezialisierten Implementationen explizit als `inline` innerhalb der Header-Datei deklarieren:

```
template<>
inline int convertString<...>
...
```

Generischer Interpreter

Schreiben Sie nun eine generische Implementation der Klasse `PostfixInterpreter` in der Datei `GenericInterpreter.hpp`, so dass diese Ausdrücke nach der Postfixnotation für beliebige Datentypen auswerten kann. Verwenden Sie hierfür auch die gerade eben neu definierte `convertString()` Funktion, um einen String in den korrekten Datentypen umzuwandeln. Nachdem Sie den notwendigen Code implementiert haben, stellen Sie bitte auch sicher, dass alle Unit-Tests in der Datei `UnitTest2.cpp` erfolgreich bestanden werden.

Concepts

Damit der Postfix-Interpreter instanziiert werden kann, muss der generische Datentyp die vier Basisoperationen `+, -, *, /` unterstützen. Ansonsten erhalten wir einen Fehler bei der Instanziierung (z.B. können Sie versuchen, `PostfixInterpreter<std::string>` zu initialisieren). Dies ist natürlich suboptimal, weshalb wir unsere Template-Definition mittels Concepts besser spezifizieren wollen.

Wie Sie bereits in der Vorlesung gesehen haben, können Concepts dazu verwendet werden, Bedingungen für Template Parameter zu definieren. Möchten Sie z.B. sicherzustellen, dass ein Template Parameter den Plus-Operator ausführen kann, so können Sie Ihr eigenes Concept definieren, welches den Operator für zwei beliebige Instanzen des generischen Datentyps ausführt. Diese Bedingung stellt sicher, dass nur Template Parameter akzeptiert werden, welche diese Operation ausführen können. Der Syntax für dieses Beispiel sieht wie folgt aus:

```
template<typename T>
concept Addable = requires(T a, T b) {
    a + b;
};
```

Um dieses Concept für eine Template-basierte Methode oder Klasse zu verwenden, müssen Sie anschliessend nur das Concept innerhalb der Parameterliste des Templates verwenden:

```
template<Addable T>
class PostfixInterpreter {
    ...
};
```

Fügen Sie nun vier neue Concepts in der Datei `GenericInterpreter.hpp` hinzu, welche sicherstellen, dass ein generischer Parameter die vier Basisoperationen unterstützt: `+, -, *, /`. Fügen Sie anschliessend ein ***fünftes*** Concept `Computable` hinzu, welches die vier vorherigen Concepts mittels der &&-Relation verknüpft, und verwenden Sie dieses `Computable` Concept innerhalb der Parameterliste der Templatedefinition der Interpreter-Klasse. Hiermit soll sichergestellt werden, dass unsere Interpreter-Klasse nur mit Parametern instanziiert werden kann, welche auch die vier Basisoperationen unterstützen.

2.3 Aufgabe 3

In dieser Aufgabe werden Sie nun den Interpreter so erweitern, dass dieser beliebige Operationen (nicht nur die vier Basisoperationen) unterstützen kann, um einen Ausdruck in Postfixnotation auszuwerten. Für die Lösung dieser Aufgabe verwenden Sie bitte weiterhin die bereits bekannte Datei `GenericInterpreter.hpp` aus der vorherigen Aufgabe.

Neue Hilfsmethode

Die vier Basisoperationen, welche bisher in unserem Interpreter unterstützt werden, sind auch in der Standard Template Bibliothek als Funktoren (Klassen, welche den Funktionsoperator überladen) verfügbar. Sie finden die Klassen als `std::plus`, `std::minus`, `std::multiplies` und `std::divides`[1]. Ihre (vereinfachte) generische Implementation in der Standardbibliothek sieht wie folgt aus:

```
template<typename T>
struct plus {
    constexpr T operator()(const T& a, const T&b) const {
        return a + b;
    }
}
```

Implementieren Sie nun eine neue generische Hilfsmethode `apply()` in der Datei `GenericInterpreter.hpp`, welche für jeden beliebigen Funktor (die eine binäre Operation repräsentiert), die Operation auf dem Stack ausführt und das Resultat anschliessend wieder darauf ablegt. Zum Beispiel, anstatt den Codeblock

```
if (token == "+") {
    T b = stack.back();
    stack.pop_back();
    T a = stack.back();
    stack.pop_back();
    stack.push_back(a + b);
}
```

für die Addition auszuführen, möchten wir gerne einfach nur die generische Methode `apply()` aufrufen:

[1] https://en.cppreference.com/w/cpp/utility/functional

```
if (token == "+") {
    apply(std::plus<T>());
}
```

Beachten Sie, dass es nur eine einzige private Instanzmethode `apply(Func f)` braucht, welche die auszuführende Funktion `f` als Parameter bekommt. Stellen Sie am Ende sicher, dass alle Unit-Tests aus der Datei `UnitTest2.cpp` nach dieser Änderung weiterhin bestanden werden.

Concepts

Definieren Sie ein neues Concept namens `BinaryOperation<Func, T>` in der Datei `GenericInterpreter.hpp`, welches sicherstellt, dass eine Instanz des generischen Parameters eine binäre Operation ausführen kann. Orientieren Sie sich hierfür an der Implementation der bisherigen Concepts. Erweitern Sie anschliessend das Concept so, dass auch sichergestellt wird, dass das Resultat der binären Operation weiterhin vom gleichem Datentypen ist wie dessen Operanden. Um den Rückgabetypen eines Ausdrucks mittels einem Concepts zu überprüfen, kann folgende Syntax verwendet werden:

```
template<typename T>
concept ResultIsIntegral = requires(T a) {
    { a() } -> std::integral;
};
```

In diesem Beispiel wird überprüft, ob der Rückgabetyp eines Funktors mit der Operation `a()` ein Integral-Datentyp ist (z.B. `int`, `bool`, `char`). Dabei ist zu beachten, dass `std::integral` wiederum ein Konzept mit einem einzigen generischen Parameter ist, welcher hier nicht angegeben werden muss, da der Rückgabetyp des Ausdruckes automatisch den ersten generischen Parameter des Konzepts bestimmt.

Verwenden Sie anschliessend das neue Concept `BinaryOperation<Func, T>` in der Definition der generischen `apply()` Methode, um sicherzustellen, dass diese Methode nur dann instanziiert werden kann, falls der Funktor einen binären Operator unterstützt.

Tipp: Datentypen können mittels dem Concept `std::same_as<T1, T2>` überprüft werden.

Unäre Operationen

Nun möchten wir noch gerne unser Sortiment an unterstützten Operationen im Interpreter erweitern. Bisher haben wir uns immer nur mit binären Operationen befasst. Ein weiterer Operator, welcher wir nun in unserem Interpreter hinzufügen möchten, ist der Inkrement-Operator ++, welcher das oberste Element auf dem Stack um eins erhöht. Dieser Operator ist *unär,* da nur ein Operand benötigt wird.

Definieren Sie als erstes einen neuen Funktor `increment` in der Header-Datei `GenericInterpreter.hpp`, welches diesen neuen unären Operator implementiert. Sie können sich hierbei an der Implementation des `std::plus` Funktors orientieren. Überladen Sie anschliessend die generische Implementation der Methode `apply()` so, dass diese sowohl binäre als auch unäre Operationen ausführen kann. Führen Sie als letztes noch ein neues Concept `UnaryOperation` ein, welches (ähnlich wie das vorherige Concept) sicherstellt, dass ein gegebener Funktor eine unäre Operation ausführen kann und der Rückgabetyp vom gleichen Typ wie der Operand ist.

Nachdem Sie den notwendigen Code in der Header-Datei implementiert haben, stellen Sie bitte ausserdem sicher, dass alle Unit-Tests in der Datei `UnitTest3.cpp` erfolgreich bestanden werden.

2.4 Aufgabe 4 (optional)

Nachdem Sie nun einen vollumfänglichen und generischen Postfix-Interpreter in den vorherigen Aufgaben implementiert haben, gehen wir nochmals einen Schritt zurück an den Anfang. In dieser letzten Aufgabe werden wir nun versuchen, unsere Ausdrücke nicht mehr mittels Strings, sondern direkt als Variadic Parameters zu übergeben. Sprich, anstatt einen Ausdruck mittels einem Vektor aus Strings wie folgt auszuwerten:

```
interpreter.evaluate({“5”, “2”, “3”, “+”, “*”});
```

Sollte es genügen, einen Ausdruck direkt mit den korrekten Literalen und Funktoren anzugeben:

```
interpreter.evaluate(5, 2, 4, std::plus<int>(), std::multiplies<int>());
```

Für die Lösung dieser Aufgabe verwenden Sie bitte die bereits vorgegebene Datei `VariadicInterpreter.hpp`.

Generischer Variadic Interpreter

Öffnen Sie die Datei `VariadicInterpreter.hpp` und füllen Sie den fehlenden notwendigen Code ein, so dass der Interpreter einen gegebenen Ausdruck in Postfixnotation auswertet. Verwenden Sie hierfür überladene und rekursive generische Hilfsmethoden, um zu erkennen, ob der oberste Parameter ein Funktor oder ein numerischer Parameter ist.

Für die Implementation genügt es, wenn der Interpreter nur binäre Operatoren unterstützt. Stellen Sie am Ende sicher, dass alle Unit-Tests in der Datei `UnitTest4.cpp` erfolgreich bestanden werden.

Tipp: Um zu überprüfen, ob ein Parameter Pack leer ist oder nicht, können Sie folgenden Code verwenden:

```
template<typename... Args>
T evaluate(const Args&... args) {
  if constexpr (sizeof...(args) > 0) {
    ...
  }
}
```

Da ein Template immer während der Kompilierung instanziiert wird, muss der Ausdruck in der `if`-Anweisung zwingend als `constexpr` deklariert werden. Alternativ können Sie die Verankerung der Rekursion auch in einer eigenen Funktion implementieren, um zu erkennen, ob ein Parameter Pack leer ist.

Concepts

Damit ein Ausdruck im neuen Interpreter korrekt ausgewertet werden kann, sollte der Interpreter, soweit möglich, nicht zu viele Datentypen gleichzeitig als Variadic Parameters akzeptieren. Um die Anzahl möglicher Fehler einzuschränken, möchten wir daher ein neues Concept einführen, welches sicherstellt, dass alle numerischen Parameter den gleichen Datentyp haben oder zu diesem Datentyp implizit konvertierbar sind. Sprich, der Interpreter soll nur instanziiert werden, falls alle Operanden den gleichen Datentypen haben. Hierfür muss eine der folgenden Bedingungen für jeden einzelnen Variadic Parameter erfüllt sein:

- Der Parameter hat den gleichen Datentyp wie der generische Template Parameter des Interpreters.
- Der Parameter stellt ein binäres Funktionsobjekt dar.

Tipp: Um zu überprüfen, ob ein generischer Datentyp T in den Datentyp U konvertiert werden kann, können Sie folgendes Concept verwenden: `std::convertible_to<T, U>`[2].

Überlegen Sie sich, wie diese Bedingungen als einzelne Concepts formuliert werden können und fügen Sie diese in der Datei `VariadicInterpreter.hpp` hinzu. Um alle Variadic Parameter einzeln in einer Template Deklaration zu überprüfen, können Sie folgenden Syntax verwenden:

[2] https://en.cppreference.com/w/cpp/concepts/convertible_to

```
// Bedingung muss für alle Parameter erfüllt sein.
template<typename... Args>
   requires (Bedingung<Args> && ...)

// Bedingung muss für mindestens ein Parameter erfüllt sein.
template<typename... Args>
   requires (Bedingung<Args> || ...)
```

Die beiden Symbolen && ... und || ... sind Kurzformen um den Compiler zu sagen, dass eine Bedingung wie folgt erweitert werden soll:

```
// Bedingung muss für alle Parameter erfüllt sein.
template<typename... Args>
   requires (Bedingung<Args[0]> && Bedingung<Args[1]> && ...)
```

Falls das Parameter Pack `Args` leer ist, so evaluiert diese Bedingung automatisch zu `false`. Natürlich kann auch die `Bedingung` selbst wieder aus mehreren Unterbedingungen bestehen.

3 Abgabe

Stellen Sie sicher, dass alle Ihre Programme alle zur Verfügung gestellten Unit-Tests erfolgreich bestehen. Senden Sie anschliessend Ihre Implementationen

- `IntegerInterpreter.h`
- `GenericInterpreter.hpp`
- `VariadicInterpreter.hpp (nur falls Aufgabe gelöst)`

als **ungezippte** E-Mail-Anhänge an: christoph.stamm@fhnw.ch. Beachten Sie, dass die Bearbeitung der vierten Aufgabe “Variadic Templates” optional ist.