UNIVERSITÄT HEIDELBERG
Zukunft. Seit 1386.
IWR Interdisciplinary Center for Scientific Computing

IPI: Einführung in die Praktische Informatik
Wintersemester 2018
Prof. Björn Ommer
ommer@uni-heidelberg.de

Übungsblatt 7: Datentypen und Umgebungsvariablen

Abgabe am 12.12.2018, 13:00.

Aufgabe 1: Umgebungsvariablen **(7P)**

Gegeben sei das folgende Programm:

```
1  int g = 0;
2
3  int a_mod_b( int a, int b )
4  {
5      int m = a % b;
6      return m; [1]
7  }
8
9  int ggT( int a, int b )
10 {
11     g = g + 1;
12     int Null = 0;
13     if ( b == Null )
14         return a; [2]
15     else
16         return ggT( b, a_mod_b( a, b ) );
17 }
18
19 int main( void )
20 {
21     int a = 2;
22     int b = 14;
23     {
24         int a = 7;
25         int g = ggT( b, a );
26         b = g;
27     }
28     a = g;
29     return 0; [3]
30 }
```

a) Verwenden Sie das Umgebungsmodell um zu bestimmen, welche Umgebungen existieren, wenn das Programm die mit [1], [2] und [3] markierten Zeilen erreicht. Die markierten Zeilen sollen erreicht sein, aber noch nicht ausgeführt – im Falle [3] ist Zeile 28 ausgeführt worden, Zeile 29 aber noch nicht.
Stellen Sie die Umgebungen graphisch dar (wie in der Vorlesung) um zu zeigen welche Namen existieren und welche Werte Sie haben. Namen ohne definierten Wert markieren Sie mit `?`. [3 Punkte]

b) Compiler nutzen einen sogenannten Aufrufstapel (*call stack*) um Funktionsaufrufe zu implementieren. Die für einen Funktionsaufruf benötigten Daten (u. a. Funktionsargumente und lokale Variablen) werden in sogenannten *stack frames* gespeichert. Bei jedem Funktionsaufruf wird solch ein *stack frame* auf den *call stack* gelegt und am Ende der Funktion wieder heruntergenommen. Wir werden dies mit Hilfe des GNU Debuggers `gdb` genauer untersuchen.
Um Rückschlüsse von der ausführbaren Datei zum Quelltext zu ermöglichen, compilieren sie obigen Quelltext (`umgebung.cc`) mit der zusätzlichen Compileroption `-g`. Laden Sie das erzeugte Programm `<executable>` mit

```
gdb <executable>
```

in den Debugger. Eine ausführliche Dokumentation finden Sie unter `https://www.gnu.org/software/gdb/documentation/`. Alternativ können Sie auch innerhalb der Eingabeauffordung des Debuggers den Befehl `help` nutzen um weitere Informationen zu erhalten. Die für die Aufgabe benötigten Befehle sind im Folgenden kurz zusammengefasst:

- `run` Führt das Programm bis zum Erreichen eines sogenannten *breakpoints* (oder eines Fehlers) aus.
- `continue` Setzt die Ausführung nach Erreichen eines *breakpoints* bis zum nächsten *breakpoint* fort.
- `break <source>:<lineno>` Setzt einen *breakpoint* in Zeile `<lineno>` in der Quelltextdatei `<source>`.
- `backtrace` Zeigt eine Liste der aktuellen *stack frames* des *call stack*. Um Informationen über lokale Variablen der *stack frames* anzuzeigen kann `backtrace full` genutzt werden. Zeilen die mit # starten beschreiben einen *stack frame*.

Nutzen Sie *breakpoints* um an den mit 1, 2 und 3 markierten Zeilen die *stack frames* sowie die dazugehörigen lokalen Variablen ausgeben zu lassen. Geben Sie sowohl die geeigneten Befehle als auch die dazugehörigen Ausgaben an. Welche *stack frames* korrespondieren zu welchen Umgebungen aus Aufgabenteil a)? Worin unterscheidet sich das Umgebungsmodell aus der Vorlesung vom *call stack*? [4 Punkte]

Aufgabe 2: Ringpuffer **(5P)**

Das Ziel dieser Aufgabe ist es, einen *Ringpuffer* zu erstellen. Dies ist eine einfache Datenstruktur mit fester Kapazität, in die Elemente zur späteren Verwendung eingefügt werden können.

Informieren Sie sich zunächst selbstständig über die grundsätzliche Idee eines Ringpuffers (engl. *circular buffer*). Sie sollen eine einfache Variante des Ringpuffers basierend auf einem Array der Größe 10 realisieren. Sie benötigen neben dem eigentlichen Array noch zwei Variablen `in` und `out`, die den Index des ersten freien bzw. des ersten belegten Feldes markieren.
Das Programm soll nun folgendermaßen ablaufen:

- Solange eine positive Zahl eingegeben wird, wird sie in den Puffer (an die Position `in`) geschrieben. Überlegen Sie sich, wie Sie die Ringstruktur realisieren: Ist das Ende des Arrays erreicht, soll am Anfang weitergeschrieben werden.
- Ist der Puffer voll belegt, wird einfach der älteste Wert überschrieben. In diesem Fall soll jedoch zumindest eine Warnung ausgegeben werden.
- Wird eine 0 eingegeben, wird der älteste noch vorhandene Wert (an der Position `out`) ausgegeben. Beachten Sie, dass der Lesevorgang den Wert aus dem Puffer löscht.
- Nach jeder erfolgten Lese- oder Schreiboperation soll zudem der gesamte Puffer mit der Position beider Zeiger ausgegeben werden. Die Ausgabe könnte in etwa so aussehen:

```
   23
   123
   12
   300
   32
>  312
   3
<  123
   12
   31
```

- Die Eingabe einer negativen Zahl soll das Programm beenden.

Als Startkonfiguration verwenden Sie einen Puffer, in dem bereits eine einzige "1" steht. An welcher Stelle diese steht, spielt aufgrund der Ringstruktur keine Rolle. Die richtige Startposition der beiden Zeiger `in` und `out` ist jedoch essentiell.

Aufgabe 3: RPN Rechner **(8P)**
Ziel dieser Aufgabe ist es, ein Programm zu schreiben, das eine Zeichenfolge als

Eingabe erhält, versucht, diese als mathematische Formel in Umgekehrter Polnischer Notation (auch Postfixnotation oder Reverse Polish Notation, kurz RPN) zu interpretieren, und entweder das Ergebnis der Berechnung (gültiger Ausdruck) oder eine Fehlermeldung (ungültiger Ausdruck) ausgibt. So soll zum Beispiel die Zeichenfolge

```
67 55 - 54 6 / + 2 *
```

zur Ausgabe "42" führen, denn es ist die Postfixnotation für die Formel

$$(((67 - 55) + (54/6)) \cdot 2).$$

Für die Eingabe sollen die folgenden Regeln gelten:

- Ziffernfolgen stellen Integer-Zahlen dar
- Die Symbole '+','-','*' und '/' stehen für die entsprechenden Operationen, wie Sie sie schon in C++ benutzen
- Alle weitere Zeichen werden ignoriert und haben nur den Effekt, zwei Ziffernfolgen und damit Zahlen voneinander zu trennen, falls sie zwischen diesen auftauchen

Benutzen Sie einen Stack, auf dem Sie der Reihe nach die gelesenen Zahlen ablegen. Dieser soll ein `struct` sein, das ein `int`-Array fester, ausreichender Größe und ein `int` zum Zählen der hinterlegten Elemente enthält.
Sie dürfen keine Funktionen verwenden, die Zeichenketten in Zahlen umwandeln, und müssen also "zu Fuß" die Zahlen Zeichen für Zeichen konstruieren. Benutzen Sie dafür ein `bool`, um sich zu merken, ob das vorherige Zeichen eine Ziffer[1] war, und ein `int`, in dem Sie die Zahl zusammensetzen, bis die letzte Ziffer gelesen ist. Beachten Sie, dass das `push()` zum Speichern der Zahl auf dem Stack erst bei der nächsten Nichtziffer erfolgt, denn erst dann können Sie sicher sein, dass die Zahl zu Ende gelesen wurde.
Sobald einer der vier Operatoren gelesen wird, müssen Sie sich zwei Zahlen vom Stack holen, auf diese den Operator anwenden und das Ergebnis zurück auf den Stack legen.
Benutzen Sie die Datei `taschenrechner.cc` als Vorlage für Ihr Programm. Testen Sie Ihr Programm mit den folgenden Eingaben:

- `"67 55 - 54 6 / + 2 *"`
- `"2 7 * - 4 +"`
- `"16 2 2 4 2 /*Dies ist (k?)ein Kommentar*/ 4 2 / *"`

[1]Hinweis: Der numerische Wert eines Zeichens für eine Ziffer stimmt nicht mit der Ziffer überein, sondern ist durch die ASCII Tabelle (Kap. 4.3 im Skript) festgelegt. So z.B. `'0' == 48`